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

Benoît Ballenghien and Burkhart Wolff

March 8, 2024

Abstract

Recently, a modern version of Roscoe and Brookes [2] Failure-Divergence Semantics for CSP has been formalized in Isabelle [6] 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 [2] Failure-Divergence Semantics for CSP has been formalized in Isabelle [6] 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, several possibilities are discussed.

As a bonus, we have defined three new operators: Sliding, Throw and Interrupt which are of particular pragmatic interest in operational semantics. Moreover, we have proven new "'laws" for HOL-CSP improving the latter.

The present work addresses this issue by providing the (to our knowledge) first formal theory of operational behaviour derived from HOL-CSP via a bridge definition between the denotational and the operational semantics. In fact, several possibilities are discussed.

As a bonus, we have defined three new operators: Sliding, Throw and Interrupt which are of particular pragmatic interest in operational semantics. Moreover, we have proven new "'laws" for HOL-CSP improving the latter.

Contents

Chapter 1

Introduction

1.1 Motivations

HOL-CSP [6] 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" [3] and the semantic details in a joint paper of Roscoe and Brooks "An improved failures model for communicating processes" [2].

Basically, the session HOL-CSP introduces the type $'\alpha$ *process*, 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.

Additionally, we developed the theory of the interrupt operators *Sliding*, *Throw* and *Interrupt* [4] which have been traditionally introduced in the context of operational semantics. This part of the present theory reintroduces denotational semantics for these operators and constructs on this basis

the operational laws for them.

The overall objective of this work is to provide a formal, machine checked foundation for the laws provided by Roscoe in [3, 5]. In several places, our formalization efforts led to slight modifications of the original definitions in order to achieve the goal of a combined integrated theory. In some cases – in particular in connection with the *Interrupt* operator definition – some corrections have been necessary since the fundamental invariants were not respected.

1.2 The Global Architecture of HOL-CSP_OpSem

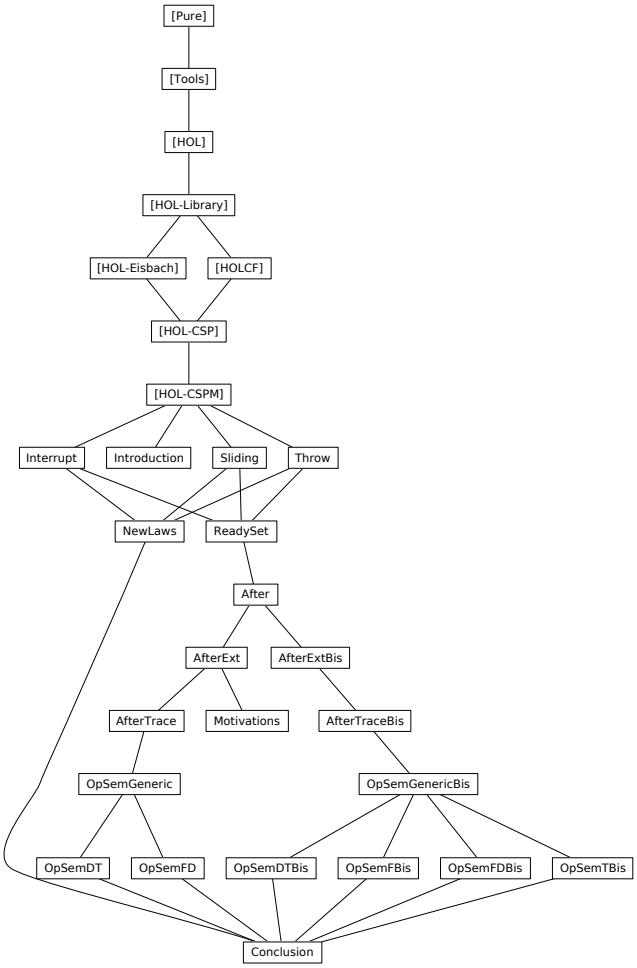

Figure 1.1: The overall architecture

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

New Operators

Three operators of CSP has not been defined yet in `HOL-CSP` (and not in `HOL-CSPM` either): Sliding, Interrupt and Throw. Since they are mentioned by Roscoe [4] (and since he provides operational laws for them too in [5]), it would be a shame not to include them in our work.

We will therefore define them now before moving on to the construction of our correspondence between semantics.

2.1 The Sliding Operator (also called Timeout)

theory *Sliding*
imports *HOL−CSPM.CSPM*
begin

2.1.1 Definition

definition *Sliding* :: ‹'α *process* ⇒ 'α *process* ⇒ 'α *process*› (**infixl** ‹▷› *78*)
where ‹*P* ▷ *Q* ≡ (*P* □ *Q*) ⊓ *Q*›

— See if we want to define a MultiSliding operator like MultiSeq.

2.1.2 Projections

lemma *F-Sliding*:
‹ℱ (*P* ▷ *Q*) = ℱ *Q* ∪ {(*s*, *X*). *s* ≠ [] ∧ (*s*, *X*) ∈ ℱ *P* ∨
s = [] ∧ (*s* ∈ 𝒟 *P* ∨ *tick* ∉ *X* ∧ [*tick*] ∈ 𝒯 *P*)}›
by (*auto simp add*: *Sliding-def F-Ndet F-Det NF-ND is-processT6-S2*)

corollary ‹ℱ (*Mprefix A P* ▷ *Q*) = ℱ *Q* ∪ {(*s*, *X*) ∈ ℱ (*Mprefix A P*). *s* ≠ []}›
by (*auto simp add*: *F-Sliding*)

lemma *D-Sliding*: ‹𝒟 (*P* ▷ *Q*) = 𝒟 *P* ∪ 𝒟 *Q*›
by (*simp add*: *Sliding-def D-Ndet D-Det*)

lemma *T-Sliding*: ‹$\mathcal{T}\ (P \rhd Q) = \mathcal{T}\ P \cup \mathcal{T}\ Q$›
by (*simp add*: *Sliding-def T-Ndet T-Det*)

2.1.3 Monotony

lemma *mono-right-Sliding-F*: ‹$Q \sqsubseteq_F Q' \Longrightarrow P \rhd Q \sqsubseteq_F P \rhd Q'$›
and *mono-Sliding-D*: ‹$P \sqsubseteq_D P' \Longrightarrow Q \sqsubseteq_D Q' \Longrightarrow P \rhd Q \sqsubseteq_D P' \rhd Q'$›
and *mono-Sliding-T*: ‹$P \sqsubseteq_T P' \Longrightarrow Q \sqsubseteq_T Q' \Longrightarrow P \rhd Q \sqsubseteq_T P' \rhd Q'$›
and *mono-Sliding-FD*: ‹$P \sqsubseteq_{FD} P' \Longrightarrow Q \sqsubseteq_{FD} Q' \Longrightarrow P \rhd Q \sqsubseteq_{FD} P' \rhd Q'$›
and *mono-Sliding-DT*: ‹$P \sqsubseteq_{DT} P' \Longrightarrow Q \sqsubseteq_{DT} Q' \Longrightarrow P \rhd Q \sqsubseteq_{DT} P' \rhd Q'$›
by (*simp add*: *failure-refine-def F-Sliding subset-iff*)
(*simp-all add*: *Sliding-def*)

2.1.4 Properties

lemma *Sliding-id*: ‹$P \rhd P = P$›
by (*simp add*: *Det-id Ndet-id Sliding-def*)

lemma *STOP-Sliding*: ‹$STOP \rhd P = P$›
unfolding *Sliding-def* **by** (*simp add*: *Det-commute Det-STOP Ndet-id*)

Of course, $P \rhd STOP \neq STOP$ and $P \rhd STOP \neq P$ in general.

lemma ‹$\exists P.\ P \rhd STOP \neq STOP \wedge P \rhd STOP \neq P$›
proof (*intro exI*)
show ‹$SKIP \rhd STOP \neq STOP \wedge SKIP \rhd STOP \neq SKIP$›
by (*metis Det-STOP Ndet-commute SKIP-F-iff SKIP-Neq-STOP*
STOP-F-iff Sliding-def mono-Ndet-F-left)
qed

But we still have this result.

lemma *Sliding-is-STOP-iff*: ‹$P \rhd Q = STOP \longleftrightarrow P = STOP \wedge Q = STOP$›
by (*auto simp add*: *STOP-iff-T T-Sliding intro*: *Nil-elem-T*)

lemma *Sliding-STOP-Det*: ‹$(P \rhd STOP) \Box Q = P \rhd Q$›
by (*simp add*: *Det-STOP Det-commute Det-distrib Sliding-def*)

lemma *BOT-Sliding*: ‹$\bot \rhd P = \bot$›
and *Sliding-BOT*: ‹$P \rhd \bot = \bot$›
unfolding *Sliding-def* **by** (*simp-all add*: *Det-commute Det-BOT Ndet-commute Ndet-BOT*)

lemma *Sliding-is-BOT-iff*: ‹$P \rhd Q = \bot \longleftrightarrow P = \bot \vee Q = \bot$›
by (*simp add*: *Det-is-BOT-iff Ndet-is-BOT-iff Sliding-def*)

lemma *Sliding-assoc*: ‹$P1 \rhd P2 \rhd P3 = P1 \rhd (P2 \rhd P3)$›
by (*metis Det-assoc Det-commute Det-distrib Ndet-assoc*

Ndet-commute Ndet-distrib Ndet-id Sliding-def)

lemma *SKIP-Sliding*: ‹*SKIP* ▷ *P* = *P* ⊓ *SKIP*›
by (*auto simp add*: *Sliding-def Process-eq-spec F-Ndet*
F-Det T-SKIP D-Ndet D-Det F-SKIP D-SKIP NF-ND)

lemma *Sliding-SKIP*: ‹*P* ▷ *SKIP* = *P* □ *SKIP*›
by (*auto simp add*: *Sliding-def Process-eq-spec F-Ndet*
F-Det T-SKIP D-Ndet D-Det F-SKIP D-SKIP)

lemma *Sliding-Det*: ‹(*P* ▷ *P'*) □ *Q* = *P* ▷ *P'* □ *Q*›
by (*metis Det-assoc Det-commute Det-distrib Sliding-def*)

lemma *Sliding-Ndet*: ‹(*P* ⊓ *P'*) ▷ *Q* = (*P* ▷ *Q*) ⊓ (*P'* ▷ *Q*)›
‹*P* ▷ (*Q* ⊓ *Q'*) = (*P* ▷ *Q*) ⊓ (*P* ▷ *Q'*)›
by (*auto simp add*: *Process-eq-spec F-Ndet D-Ndet T-Ndet F-Sliding D-Sliding*)

lemma *Renaming-Sliding*:
‹*Renaming* (*P* ▷ *Q*) *f* = *Renaming P f* ▷ *Renaming Q f*›
by (*simp add*: *Renaming-Det Renaming-Ndet Sliding-def*)

lemma *events-Sliding*: ‹*events-of* (*P* ▷ *Q*) = *events-of P* ∪ *events-of Q*›
unfolding *Sliding-def* **by** (*simp add*: *events-of-def T-Det T-Ndet*)

2.1.5 Continuity

From the definition, continuity is obvious.

lemma *Sliding-cont*[*simp*] : ‹*cont f* ⟹ *cont g* ⟹ *cont* (λ*x*. *f x* ▷ *g x*)›
by (*simp add*: *Sliding-def*)

end

2.2 The Throw Operator

theory *Throw*
imports *HOL−CSPM.CSPM*
begin

2.2.1 Definition

The Throw operator allows error handling. Whenever an error (or more generally any event $ev\ e \in ev\ `\ A$) occurs in P, P is shut down and $Q\ e$ is started.

This operator can somehow be seen as a generalization of sequential composition (**;**): P terminates on any event in $ev\ `\ A$ rather than *tick* (however it do not hide these events like (**;**) do for *tick*, but we can use an additional $\lambda P.\ P \setminus A$).

This is a relatively new addition to CSP (see [4, p.140]).

lift-definition *Throw* :: ‹['α *process*, 'α *set*, 'α ⇒ 'α *process*] ⇒ 'α *process*›
is ‹λ *P A Q*.
({(*t1*, *X*) ∈ ℱ *P*. *set t1* ∩ *ev* ` *A* = {}} ∪
{(*t1* @ *t2*, *X*)| *t1 t2 X*. *t1* ∈ 𝒟 *P* ∧ *tickFree t1* ∧
set t1 ∩ *ev* ` *A* = {} ∧ *front-tickFree t2*} ∪
{(*t1* @ *ev a* # *t2*, *X*)| *t1 a t2 X*. *t1* @ [*ev a*] ∈ 𝒯 *P* ∧ *set t1* ∩ *ev* ` *A* = {} ∧
a ∈ *A* ∧ (*t2*, *X*) ∈ ℱ (*Q a*)},
{*t1* @ *t2*| *t1 t2*. *t1* ∈ 𝒟 *P* ∧ *tickFree t1* ∧
set t1 ∩ *ev* ` *A* = {} ∧ *front-tickFree t2*} ∪
{*t1* @ *ev a* # *t2*| *t1 a t2*. *t1* @ [*ev a*] ∈ 𝒯 *P* ∧ *set t1* ∩ *ev* ` *A* = {} ∧
a ∈ *A* ∧ *t2* ∈ 𝒟 (*Q a*)})›
proof −
show ‹*?thesis P A Q*› (**is** ‹*is-process* (*?f*, *?d*)›) **for** *P A Q*
unfolding *is-process-def FAILURES-def DIVERGENCES-def fst-conv snd-conv*
proof (*intro conjI allI impI*; (*elim conjE*)*?*)
show ‹([], {}) ∈ *?f*› **by** (*simp add*: *is-processT1*)
next
show ‹(*s*, *X*) ∈ *?f* ⟹ *front-tickFree s*› **for** *s X*
apply (*simp*, *elim disjE exE*)
subgoal by (*metis is-processT*)
subgoal by (*solves* ‹*simp add*: *front-tickFree-append*›)
by (*metis F-T append-Cons append-Nil append-T-imp-tickFree butlast.simps*(*2*) *event.simps*(*3*)
front-tickFree-append is-processT2-TR last-ConsL not-Cons-self tickFree-butlast)
next
show ‹(*s* @ *t*, {}) ∈ *?f* ⟹ (*s*, {}) ∈ *?f*› **for** *s t*
proof (*induct t rule*: *rev-induct*)
case *Nil*
thus ‹(*s*, {}) ∈ *?f*› **by** *simp*
next
case (*snoc b t*)
consider ‹(*s* @ *t* @ [*b*], {}) ∈ ℱ *P*› ‹(*set s* ∪ *set t*) ∩ *ev* ` *A* = {}›
| ‹∃ *t1 t2*. *s* @ *t* @ [*b*] = *t1* @ *t2* ∧ *t1* ∈ 𝒟 *P* ∧ *tickFree t1* ∧
set t1 ∩ *ev* ` *A* = {} ∧ *front-tickFree t2* ∨
(∃ *a*. *s* @ *t* @ [*b*] = *t1* @ *ev a* # *t2* ∧ *t1* @ [*ev a*] ∈ 𝒯 *P* ∧
set t1 ∩ *ev* ` *A* = {} ∧ *a* ∈ *A* ∧ (*t2*, {}) ∈ ℱ (*Q a*))›
using *snoc.prems* **by** *simp blast*

```
    thus ‹(s, {}) ∈ ?f›
    proof cases
      show ‹(s @ t @ [b], {}) ∈ 𝓕 P ⟹ (set s ∪ set t) ∩ ev ‘ A = {} ⟹ (s,
{}) ∈ ?f›
        by (drule is-processT3[rule-format]) (simp add: Int-Un-distrib2)
    next
      assume ‹∃ t1 t2. s @ t @ [b] = t1 @ t2 ∧ t1 ∈ 𝓓 P ∧ tickFree t1 ∧
                set t1 ∩ ev ‘ A = {} ∧ front-tickFree t2 ∨
                (∃ a. s @ t @ [b] = t1 @ ev a # t2 ∧ t1 @ [ev a] ∈ 𝓣 P ∧
                    set t1 ∩ ev ‘ A = {} ∧ a ∈ A ∧ (t2, {}) ∈ 𝓕 (Q a))›
        (is ‹∃ t1 t2. ?disj t1 t2›)
      then obtain t1 t2 where ‹?disj t1 t2› by blast
      show ‹(s, {}) ∈ ?f›
      apply (rule snoc.hyps)
        using ‹?disj t1 t2› apply (elim disjE exE)
         apply (all ‹cases t2 rule: rev-cases›, simp-all)
           subgoal by (metis Int-Un-distrib2 T-F D-T Un-empty append.assoc
is-processT3-SR set-append)
        subgoal by (metis front-tickFree-dw-closed)
        subgoal by (meson T-F process-charn)
        by (metis process-charn)
    qed
  qed
next
  show ‹(s, Y) ∈ ?f ⟹ X ⊆ Y ⟹ (s, X) ∈ ?f› for s X Y
    by simp (metis is-processT4)
next
  fix s X Y
  assume assms : ‹(s, X) ∈ ?f› ‹∀ c. c ∈ Y ⟶ (s @ [c], {}) ∉ ?f›
  consider ‹(s, X) ∈ 𝓕 P› ‹set s ∩ ev ‘ A = {}›
    | ‹∃ t1 t2. s = t1 @ t2 ∧ t1 ∈ 𝓓 P ∧ tickFree t1 ∧
              set t1 ∩ ev ‘ A = {} ∧ front-tickFree t2›
    | ‹∃ t1 a t2. s = t1 @ ev a # t2 ∧ t1 @ [ev a] ∈ 𝓣 P ∧
                set t1 ∩ ev ‘ A = {} ∧ a ∈ A ∧ (t2, X) ∈ 𝓕 (Q a)›
    using assms(1) by blast
  thus ‹(s, X ∪ Y) ∈ ?f›
  proof cases
    assume ∗ : ‹(s, X) ∈ 𝓕 P› ‹set s ∩ ev ‘ A = {}›
    have ‹(s @ [c], {}) ∉ 𝓕 P› if ‹c ∈ Y› for c
    proof (cases ‹c ∈ ev ‘ A›)
      from ∗(2) assms(2)[rule-format, OF that]
      show ‹c ∈ ev ‘ A ⟹ (s @ [c], {}) ∉ 𝓕 P›
        by auto (metis F-T is-processT1)
    next
      from ∗(2) assms(2)[rule-format, OF that]
      show ‹c ∉ ev ‘ A ⟹ (s @ [c], {}) ∉ 𝓕 P› by simp
    qed
    with ∗(1) is-processT5 have ‹(s, X ∪ Y) ∈ 𝓕 P› by blast
    with ∗(2) show ‹(s, X ∪ Y) ∈ ?f› by blast
```

```
  next
    assume ‹∃t1 t2. s = t1 @ t2 ∧ t1 ∈ 𝒟 P ∧ tickFree t1 ∧
                  set t1 ∩ ev ‘ A = {} ∧ front-tickFree t2›
    hence ‹s ∈ ?d› by blast
    thus ‹(s, X ∪ Y) ∈ ?f› by simp (metis NF-ND)
  next
    assume ‹∃t1 a t2. s = t1 @ ev a # t2 ∧ t1 @ [ev a] ∈ 𝒯 P ∧
                    set t1 ∩ ev ‘ A = {} ∧ a ∈ A ∧ (t2, X) ∈ ℱ (Q a)›
    then obtain t1 a t2
      where ∗ : ‹s = t1 @ ev a # t2› ‹t1 @ [ev a] ∈ 𝒯 P›
                ‹set t1 ∩ ev ‘ A = {}› ‹a ∈ A› ‹(t2, X) ∈ ℱ (Q a)› by blast
    have ‹(t2 @ [c], {}) ∉ ℱ (Q a)› if ‹c ∈ Y› for c
      using assms(2)[rule-format, OF that, simplified, THEN conjunct2,
                     THEN conjunct2, rule-format, of a t1 ‹t2 @ [c]›]
      by (simp add: ∗(1, 2, 3, 4))
    with ∗(5) is-processT5 have ∗∗ : ‹(t2, X ∪ Y) ∈ ℱ (Q a)› by blast
    show ‹(s, X ∪ Y) ∈ ?f›
      using ∗(1, 2, 3, 4) ∗∗ by blast
  qed
next
  have ∗ : ‹⋀s t1 a t2. s @ [tick] = t1 @ ev a # t2 ⟹ ∃t2'. t2 = t2' @ [tick]›
    by (simp add: snoc-eq-iff-butlast split: if-split-asm)
       (metis append-butlast-last-id)
  show ‹(s @ [tick], {}) ∈ ?f ⟹ (s, X − {tick}) ∈ ?f› for s X
    apply (simp, elim disjE exE conjE)
    subgoal by (solves ‹simp add: is-processT6›)
    subgoal by (metis butlast-append butlast-snoc front-tickFree-butlast
        non-tickFree-tick tickFree-Nil tickFree-append
        tickFree-implies-front-tickFree)
    by (frule ∗, elim exE, simp, metis is-processT6)
next
  show ‹⟦s ∈ ?d; tickFree s; front-tickFree t⟧ ⟹ s @ t ∈ ?d› for s t
    by auto (metis front-tickFree-append, metis is-processT7)
next
  show ‹s ∈ ?d ⟹ (s, X) ∈ ?f› for s X
    by simp (metis NF-ND)

next
  show ‹s @ [tick] ∈ ?d ⟹ s ∈ ?d› for s
    apply (simp, elim disjE)
    by (metis butlast-append butlast-snoc front-tickFree-butlast non-tickFree-tick
          tickFree-Nil tickFree-append tickFree-implies-front-tickFree)
       (metis D-T T-nonTickFree-imp-decomp append-single-T-imp-tickFree
          butlast.simps(2) butlast-append butlast-snoc event.distinct(1)
          process-charn tickFree-Cons tickFree-append)
qed
qed
```

We add some syntactic sugar.

syntax -*Throw* :: ‹['α *process*, *pttrn*, 'α *set*, 'α ⇒ 'α *process*] ⇒ 'α *process*›
(‹((-) Θ (-∈-)./ (-))› [73, 0, 0, 73] 72)
translations *P* Θ *a* ∈ *A*. *Q* ⇌ *CONST Throw P A* (λ*a*. *Q*)

abbreviation *Throw-without-free-var* ::
‹['α *process*, 'α *set*, 'α *process*] ⇒ 'α *process*› (‹((-) Θ (-)/ (-))› [73, 0, 73] 72)
where ‹*P* Θ *A Q* ≡ *P* Θ *a* ∈ *A*. *Q*›

Now we can write $P\ \Theta\ a{\in}A.\ Q\ a$, and when we do not want Q to be parameterized we can just write $P\ \Theta\ A\ Q$.

lemma ‹*P* Θ *a* ∈ *A*. *Q* = *P* Θ *A Q*› **by** (*fact refl*)

2.2.2 Projections

lemma *F-Throw*:
‹*F* (*P* Θ *a* ∈ *A*. *Q a*) =
{(*t1*, *X*) ∈ *F P*. *set t1* ∩ *ev* ‘ *A* = {}} ∪
{(*t1* @ *t2*, *X*)| *t1 t2 X*.
t1 ∈ *D P* ∧ *tickFree t1* ∧ *set t1* ∩ *ev* ‘ *A* = {} ∧ *front-tickFree t2*} ∪
{(*t1* @ *ev a* # *t2*, *X*)| *t1 a t2 X*.
t1 @ [*ev a*] ∈ *T P* ∧ *set t1* ∩ *ev* ‘ *A* = {} ∧ *a* ∈ *A* ∧ (*t2*, *X*) ∈ *F* (*Q a*)}›
by (*simp add*: *Failures-def FAILURES-def Throw.rep-eq*)

lemma *D-Throw*:
‹*D* (*P* Θ *a* ∈ *A*. *Q a*) =
{*t1* @ *t2*| *t1 t2*.
t1 ∈ *D P* ∧ *tickFree t1* ∧ *set t1* ∩ *ev* ‘ *A* = {} ∧ *front-tickFree t2*} ∪
{*t1* @ *ev a* # *t2*| *t1 a t2*.
t1 @ [*ev a*] ∈ *T P* ∧ *set t1* ∩ *ev* ‘ *A* = {} ∧ *a* ∈ *A* ∧ *t2* ∈ *D* (*Q a*)}›
by (*simp add*: *Divergences-def DIVERGENCES-def Throw.rep-eq*)

lemma *T-Throw*:
‹*T* (*P* Θ *a* ∈ *A*. *Q a*) =
{*t1* ∈ *T P*. *set t1* ∩ *ev* ‘ *A* = {}} ∪
{*t1* @ *t2*| *t1 t2*.
t1 ∈ *D P* ∧ *tickFree t1* ∧ *set t1* ∩ *ev* ‘ *A* = {} ∧ *front-tickFree t2*} ∪
{*t1* @ *ev a* # *t2*| *t1 a t2*.
t1 @ [*ev a*] ∈ *T P* ∧ *set t1* ∩ *ev* ‘ *A* = {} ∧ *a* ∈ *A* ∧ *t2* ∈ *T* (*Q a*)}›
by (*auto simp add*: *Traces-def TRACES-def Failures-def*[*symmetric*] *F-Throw*)
blast+

2.2.3 Monotony

lemma *min-elems-Un-subset*:
‹*min-elems* (*A* ∪ *B*) ⊆ *min-elems A* ∪ (*min-elems B* − *A*)›

by (*auto simp add*: *min-elems-def subset-iff*)

lemma *mono-Throw*[*simp*] : ‹$P\ \Theta\ a \in A.\ Q\ a \sqsubseteq P'\ \Theta\ a \in A.\ Q'\ a$›
if ‹$P \sqsubseteq P'$› **and** ‹$\forall a \in A.\ Q\ a \sqsubseteq Q'\ a$›
proof (*unfold le-approx-def Ra-def*, *safe*)
from *le-approx1*[*OF that*(*1*)] *le-approx-lemma-T*[*OF that*(*1*)]
le-approx1[*OF that*(*2*)[*rule-format*]]
show ‹$s \in \mathcal{D}\ (P'\ \Theta\ a \in A.\ Q'\ a) \Longrightarrow s \in \mathcal{D}\ (P\ \Theta\ a \in A.\ Q\ a)$› **for** *s*
by (*simp add*: *D-Throw subset-iff*) *metis*
next
fix *s X*
assume *assms* : ‹$s \notin \mathcal{D}\ (P\ \Theta\ a \in A.\ Q\ a)$› ‹$(s,\ X) \in \mathcal{F}\ (P\ \Theta\ a \in A.\ Q\ a)$›
from *assms*(*2*) **consider** ‹$(s,\ X) \in \mathcal{F}\ P$› ‹*set s* ∩ *ev* ‘ *A* = {}›
| ‹$\exists$ *t1 t2*. *s* = *t1* @ *t2* ∧ *t1* ∈ $\mathcal{D}$ *P* ∧ *tickFree t1* ∧
set t1 ∩ *ev* ‘ *A* = {} ∧ *front-tickFree t2*›
| ‹$\exists$ *t1 a t2*. *s* = *t1* @ *ev a* # *t2* ∧ *t1* @ [*ev a*] ∈ $\mathcal{T}$ *P* ∧
set t1 ∩ *ev* ‘ *A* = {} ∧ *a* ∈ *A* ∧ (*t2*, *X*) ∈ $\mathcal{F}$ (*Q a*)›
by (*simp add*: *F-Throw*) *blast*
thus ‹$(s,\ X) \in \mathcal{F}\ (P'\ \Theta\ a \in A.\ Q'\ a)$›
proof *cases*
assume ∗ : ‹$(s,\ X) \in \mathcal{F}\ P$› ‹*set s* ∩ *ev* ‘ *A* = {}›
from *assms*(*1*)[*simplified D-Throw*, *simplified*, *THEN conjunct1*, *rule-format*, *of s*]
assms(*1*)[*simplified D-Throw*, *simplified*, *THEN conjunct1*, *rule-format*, *of* ‹*butlast s*›]
have ∗∗ : ‹$s \notin \mathcal{D}\ P$›
using ∗(*2*) **apply** (*cases* ‹*tickFree s*›, *auto*)
by (*metis append-butlast-last-id disjoint-iff front-tickFree-butlast*
front-tickFree-single in-set-butlastD process-charn tickFree-butlast)
show ‹$(s,\ X) \in \mathcal{F}\ P \Longrightarrow$ *set s* ∩ *ev* ‘ *A* = {} $\Longrightarrow$ $(s,\ X) \in \mathcal{F}\ (Throw\ P'\ A\ Q')$›
by (*simp add*: *F-Throw le-approx2*[*OF that*(*1*) ∗∗])
next
assume ‹$\exists$ *t1 t2*. *s* = *t1* @ *t2* ∧ *t1* ∈ $\mathcal{D}$ *P* ∧ *tickFree t1* ∧
set t1 ∩ *ev* ‘ *A* = {} ∧ *front-tickFree t2*›
with *assms*(*1*) **show** ‹$(s,\ X) \in \mathcal{F}\ (Throw\ P'\ A\ Q')$›
by (*simp add*: *F-Throw D-Throw*)
next
assume ‹$\exists$ *t1 a t2*. *s* = *t1* @ *ev a* # *t2* ∧ *t1* @ [*ev a*] ∈ $\mathcal{T}$ *P* ∧
set t1 ∩ *ev* ‘ *A* = {} ∧ *a* ∈ *A* ∧ (*t2*, *X*) ∈ $\mathcal{F}$ (*Q a*)›
then obtain *t1 a t2*
where ∗ : ‹*s* = *t1* @ *ev a* # *t2*› ‹*t1* @ [*ev a*] ∈ $\mathcal{T}$ *P*›
‹*set t1* ∩ *ev* ‘ *A* = {}› ‹*a* ∈ *A*› ‹(*t2*, *X*) ∈ $\mathcal{F}$ (*Q a*)› **by** *blast*
from ∗(*2*) *append-single-T-imp-tickFree* **have** ∗∗ : ‹*tickFree t1*› **by** *blast*
have ∗∗∗ : ‹(*t2*, *X*) ∈ $\mathcal{F}$ (*Q*′ *a*)›
by (*fact assms*(*1*)[*simplified D-Throw*, *simplified*, *THEN conjunct2*, *rule-format*,
OF ∗(*4*, *3*, *2*, *1*), *THEN le-approx2*[*OF that*(*2*)[*rule-format*, *OF* ∗(*4*)]],
of X, *simplified* ∗(*5*), *simplified*])
have ∗∗∗∗ : ‹*t1* ∉ $\mathcal{D}$ *P*›

```
    apply (rule notI)
     apply (drule assms(1)[simplified D-Throw, simplified, THEN conjunct1,
rule-format,
                           OF *(3) **, of ‹ev a # t2›, simplified *(1), simplified])
    by (metis *(1) assms(2) front-tickFree-mono process-charn)
  show ‹(s, X) ∈ ℱ (Throw P' A Q')›
    apply (simp add: F-Throw D-Throw *(1))
    by (metis *(2, 3, 4) *** **** T-F-spec le-approx2 min-elems6 that(1))
 qed
next
 from le-approx1[OF that(1)] le-approx2[OF that(1)] le-approx2T[OF that(1)]
     le-approx2[OF that(2)[rule-format]]
 show ‹s ∉ 𝒟 (P Θ a ∈ A. Q a) ⟹
      (s, X) ∈ ℱ (P' Θ a ∈ A. Q' a) ⟹ (s, X) ∈ ℱ (P Θ a ∈ A. Q a)› for s X
  apply (simp add: F-Throw D-Throw subset-eq, safe, simp-all)
  by (meson NF-ND) (metis D-T)+
next
 define S-left
  where ‹S-left ≡ {t1 @ t2 |t1 t2. t1 ∈ 𝒟 P ∧ tickFree t1 ∧
                set t1 ∩ ev ‘ A = {} ∧ front-tickFree t2}›
 define S-right
  where ‹S-right ≡ {t1 @ ev a # t2 |t1 a t2. t1 @ [ev a] ∈ 𝒯 P ∧
                 set t1 ∩ ev ‘ A = {} ∧ a ∈ A ∧ t2 ∈ 𝒟 (Q a)}›

 have * : ‹min-elems (𝒟 (P Θ a ∈ A. Q a)) ⊆ min-elems S-left ∪ (min-elems
S-right − S-left)›
  unfolding S-left-def S-right-def
  by (simp add: D-Throw min-elems-Un-subset)
 have ** : ‹min-elems S-left = {t1 ∈ min-elems (𝒟 P). set t1 ∩ ev ‘ A = {}}›
  unfolding S-left-def min-elems-def le-list-def less-list-def
  apply (simp, safe)
  subgoal by (solves ‹meson is-processT7-S›)
 subgoal by (metis Int-Un-distrib2 Un-empty append.right-neutral front-tickFree-Nil
      front-tickFree-append front-tickFree-mono set-append)
  subgoal by (metis IntI append-Nil2 front-tickFree-Nil imageI)
  subgoal by (metis list.distinct(1) nonTickFree-n-frontTickFree process-charn
self-append-conv)
  by (metis Nil-is-append-conv append-eq-appendI self-append-conv)

 { fix t1 a t2
  assume assms : ‹t1 @ [ev a] ∈ 𝒯 P› ‹set t1 ∩ ev ‘ A = {}› ‹a ∈ A›
            ‹t2 ∈ (𝒟 (Q a))› ‹t1 @ ev a # t2 ∈ min-elems S-right› ‹t1 @ ev a
# t2 ∉ S-left›
  have ‹t2 ∈ min-elems (𝒟 (Q a))›
     ‹t1 @ [ev a] ∈ 𝒟 P ⟹ t1 @ [ev a] ∈ min-elems (𝒟 P)›
  proof (all ‹rule ccontr›)
    assume ‹t2 ∉ min-elems (𝒟 (Q a))›
    with assms(4) obtain t2' where ‹t2' < t2 › ‹t2' ∈ 𝒟 (Q a)›
     unfolding min-elems-def by blast
```

```
    hence ‹t1 @ ev a # t2' ∈ S-right› ‹t1 @ ev a # t2' < t1 @ ev a # t2›
      unfolding S-right-def using assms(1, 2, 3)
      by (auto simp add: less-append less-cons)
    with assms(5) min-elems-no nless-le show False by blast
  next
    assume ‹t1 @ [ev a] ∈ 𝒟 P› ‹t1 @ [ev a] ∉ min-elems (𝒟 P)›
    hence ‹t1 ∈ 𝒟 P› using min-elems1 by blast
    with ‹t1 @ [ev a] ∈ 𝒟 P› have ‹t1 @ ev a # t2 ∈ S-left›
      by (simp add: S-left-def)
        (metis D-imp-front-tickFree append-Cons append-Nil assms(2, 4)
            event.simps(3) front-tickFree-append tickFree-Nil
            front-tickFree-implies-tickFree tickFree-Cons)
    with assms(6) show False by simp
  qed
 } note *** = this
 have **** : ‹min-elems S-right − S-left ⊆
          {t1 @ ev a # t2 |t1 a t2. t1 @ [ev a] ∈ 𝒯 P − 𝒟 P ∧
           set t1 ∩ ev ‘ A = {} ∧ a ∈ A ∧ t2 ∈ min-elems (𝒟 (Q a))} ∪
          {t1 @ ev a # t2 |t1 a t2. t1 @ [ev a] ∈ min-elems (𝒟 P) ∧
           set t1 ∩ ev ‘ A = {} ∧ a ∈ A ∧ t2 ∈ min-elems (𝒟 (Q a))}›
  apply (intro subsetI, simp, elim conjE)
  apply (frule set-mp[OF min-elems-le-self], subst (asm) (2) S-right-def)
  using *** by fastforce

 fix s
 assume assm: ‹s ∈ min-elems (𝒟 (P Θ a ∈ A. Q a))›
 from set-mp[OF *, OF this]
 consider ‹s ∈ min-elems (𝒟 P)› ‹set s ∩ ev ‘ A = {}›
   | ‹∃ t1 a t2.
      s = t1 @ ev a # t2 ∧ set t1 ∩ ev ‘ A = {} ∧ a ∈ A ∧ t2 ∈ min-elems (𝒟
(Q a)) ∧
     (t1 @ [ev a] ∈ min-elems (𝒟 P) ∨ t1 @ [ev a] ∈ 𝒯 P ∧ t1 @ [ev a] ∉ 𝒟 P)›
   using **** by (simp add: **) blast
 thus ‹s ∈ 𝒯 (P' Θ a ∈ A. Q' a)›
 proof cases
   show ‹s ∈ min-elems (𝒟 P) ⟹ set s ∩ ev ‘ A = {} ⟹ s ∈ 𝒯 (Throw P' A
Q')›
     by (drule set-mp[OF le-approx3[OF that(1)]], simp add: T-Throw)
 next
   assume ‹∃ t1 a t2.
         s = t1 @ ev a # t2 ∧ set t1 ∩ ev ‘ A = {} ∧ a ∈ A ∧ t2 ∈ min-elems
(𝒟 (Q a)) ∧
         (t1 @ [ev a] ∈ min-elems (𝒟 P) ∨ t1 @ [ev a] ∈ 𝒯 P ∧ t1 @ [ev a] ∉ 𝒟
P)›
   then obtain t1 a t2
     where ***** : ‹s = t1 @ ev a # t2› ‹set t1 ∩ ev ‘ A = {}›
                 ‹a ∈ A› ‹t2 ∈ min-elems (𝒟 (Q a))›
                 ‹t1 @ [ev a] ∈ min-elems (𝒟 P) ∨
                  t1 @ [ev a] ∈ 𝒯 P ∧ t1 @ [ev a] ∉ 𝒟 P› by blast
```

```
    have ‹t1 @ [ev a] ∈ 𝒯 P' ∧ t2 ∈ 𝒯 (Q' a)›
      by (meson *****(3, 4, 5) le-approx2T le-approx3 subsetD that)
    with ***** show ‹s ∈ 𝒯 (Throw P' A Q')›
      by (simp add: T-Throw) blast
  qed
qed


lemma mono-right-Throw-F :
  ‹∀ a ∈ A. Q a ⊑F Q' a ⟹ P Θ a ∈ A. Q a ⊑F P Θ a ∈ A. Q' a›
  unfolding failure-refine-def
  by (simp add: F-Throw subset-iff disjoint-iff) blast

lemma mono-right-Throw-T :
  ‹∀ a ∈ A. Q a ⊑T Q' a ⟹ P Θ a ∈ A. Q a ⊑T P Θ a ∈ A. Q' a›
  unfolding trace-refine-def
  by (simp add: T-Throw subset-iff disjoint-iff) blast

lemma mono-right-Throw-D:
  ‹∀ a ∈ A. Q a ⊑D Q' a ⟹ P Θ a ∈ A. Q a ⊑D P Θ a ∈ A. Q' a›
  unfolding divergence-refine-def
  by (simp add: D-Throw subset-iff disjoint-iff) blast

lemma mono-Throw-FD : ‹P ⊑FD P' ⟹ ∀ a ∈ A. Q a ⊑FD Q' a ⟹
                      P Θ a ∈ A. Q a ⊑FD P' Θ a ∈ A. Q' a›
  apply (rule trans-FD[of - ‹P' Θ a ∈ A. Q a›])
  subgoal by (simp add: failure-divergence-refine-def
                  le-ref-def F-Throw D-Throw subset-iff; safe;
        metis [[metis-verbose = false]] T-F no-Trace-implies-no-Failure)
  by (meson leFD-imp-leD leFD-imp-leF leF-leD-imp-leFD
          mono-right-Throw-D mono-right-Throw-F)

lemma mono-Throw-DT : ‹P ⊑DT P' ⟹ ∀ a ∈ A. Q a ⊑DT Q' a ⟹
                      P Θ a ∈ A. Q a ⊑DT P' Θ a ∈ A. Q' a›
  apply (rule trans-DT[of - ‹P' Θ a ∈ A. Q a›])
  subgoal by (simp add: trace-divergence-refine-def trace-refine-def
                divergence-refine-def D-Throw T-Throw subset-iff, blast)
 by (simp add: mono-right-Throw-D mono-right-Throw-T trace-divergence-refine-def)
```

2.2.4 Properties

```
lemma Throw-STOP: ‹STOP Θ a ∈ A. Q a = STOP›
  by (auto simp add: STOP-iff-T T-Throw T-STOP D-STOP)

lemma Throw-SKIP: ‹SKIP Θ a ∈ A. Q a = SKIP›
  by (auto simp add: Process-eq-spec F-Throw F-SKIP D-Throw D-SKIP T-SKIP)

lemma Throw-BOT: ‹⊥ Θ a ∈ A. Q a = ⊥›
  by (simp add: BOT-iff-D D-Throw D-UU)
```

lemma *Throw-is-BOT-iff*: ‹$P\ \Theta\ a \in A.\ Q\ a = \bot \longleftrightarrow P = \bot$›
by (*simp add*: *BOT-iff-D D-Throw*)

lemma *Throw-empty-set*: ‹$P\ \Theta\ a \in \{\}.\ Q\ a = P$›
by (*auto simp add*: *Process-eq-spec F-Throw D-Throw*
D-expand[*symmetric*] *is-processT7-S is-processT8-S*)

lemma *Throw-Ndet*:
‹$P \sqcap P' \Theta\ a \in A.\ Q\ a = (P\ \Theta\ a \in A.\ Q\ a) \sqcap (P'\ \Theta\ a \in A.\ Q\ a)$›
‹$P\ \Theta\ a \in A.\ Q\ a \sqcap Q'\ a = (P\ \Theta\ a \in A.\ Q\ a) \sqcap (P\ \Theta\ a \in A.\ Q'\ a)$›
by (*simp add*: *Process-eq-spec F-Throw F-Ndet D-Throw D-Ndet T-Ndet*,
safe, *simp-all*; *blast*)+

lemma *Throw-Det*:
‹$P \Box P' \Theta\ a \in A.\ Q\ a = (P\ \Theta\ a \in A.\ Q\ a) \Box (P'\ \Theta\ a \in A.\ Q\ a)$›
‹$P\ \Theta\ a \in A.\ Q\ a \sqcap Q'\ a = (P\ \Theta\ a \in A.\ Q\ a) \Box (P\ \Theta\ a \in A.\ Q'\ a)$›
proof −
show ‹*Throw* $(P \Box P')\ A\ Q = Throw\ P\ A\ Q \Box Throw\ P'\ A\ Q$› (**is** ‹*?lhs* = *?rhs*›)
proof (*subst Process-eq-spec-optimized*, *safe*)
show ‹$s \in \mathcal{D}$ *?lhs* $\Longrightarrow s \in \mathcal{D}$ *?rhs*› **for** *s*
by (*auto simp add*: *D-Det T-Det D-Throw*)
next
show ‹$s \in \mathcal{D}$ *?rhs* $\Longrightarrow s \in \mathcal{D}$ *?lhs*› **for** *s*
by (*simp add*: *D-Det T-Det D-Throw*) *blast*
next
fix *s X*
assume *same-div*: ‹$\mathcal{D}$ *?lhs* = $\mathcal{D}$ *?rhs*›
assume ‹$(s, X) \in \mathcal{F}$ *?lhs*›
hence ‹$(s, X) \in \mathcal{F}\ (P \Box P') \wedge$ *set s* $\cap$ *ev* ‘ *A* = $\{\} \vee s \in \mathcal{D}$ *?lhs* $\vee$
$(\exists$ *t1 a t2*. *s* = *t1* @ *ev a* # *t2* $\wedge$ *t1* @ $[ev\ a] \in \mathcal{T}\ (P \Box P') \wedge$
set t1 $\cap$ *ev* ‘ *A* = $\{\} \wedge a \in A \wedge (t2, X) \in \mathcal{F}\ (Q\ a)$)›
by (*simp add*: *F-Throw D-Throw*) *blast*
thus ‹$(s, X) \in \mathcal{F}$ *?rhs*›
apply (*elim disjE exE*)
subgoal by (*cases s*, *auto simp add*: *F-Det F-Throw T-Throw D-Throw*)[*1*]
subgoal by (*use D-F same-div* **in** *blast*)
by (*auto simp add*: *F-Det T-Det F-Throw T-Throw D-Throw*)
next
show ‹$(s, X) \in \mathcal{F}$ *?rhs* $\Longrightarrow (s, X) \in \mathcal{F}$ *?lhs*› **for** *s X*
apply (*simp add*: *F-Det*, *elim disjE conjE*)
by (*auto simp add*: *F-Det D-Det T-Det F-Throw D-Throw*
T-Throw D-T is-processT7-S subset-iff)
(*simp-all add*: *Cons-eq-append-conv*)
qed
next

```
  show ‹P Θ a ∈ A. Q a □ Q' a = Throw P A Q □ Throw P A Q'› (is ‹?lhs Q Q' = ?rhs›)
  proof (subst Process-eq-spec-optimized, safe)
    show ‹s ∈ 𝒟 (?lhs Q Q') ⟹ s ∈ 𝒟 ?rhs› for s
      by (auto simp add: D-Det D-Throw D-Ndet)
  next
    show ‹s ∈ 𝒟 ?rhs ⟹ s ∈ 𝒟 (?lhs Q Q')› for s
      by (simp add: D-Det D-Throw D-Ndet) blast
  next
    fix s X
    assume same-div: ‹𝒟 (?lhs Q Q') = 𝒟 ?rhs›
    assume ‹(s, X) ∈ ℱ (?lhs Q Q')›
    hence ‹(s, X) ∈ ℱ P ∧ set s ∩ ev ‘ A = {} ∨ s ∈ 𝒟 (?lhs Q Q') ∨
          (∃ t1 a t2. s = t1 @ ev a # t2 ∧ t1 @ [ev a] ∈ 𝒯 P ∧
                  set t1 ∩ ev ‘ A = {} ∧ a ∈ A ∧ (t2, X) ∈ ℱ (Q a □ Q' a))›
      by (simp add: F-Throw D-Throw) blast
    thus ‹(s, X) ∈ ℱ ?rhs›
      apply (elim disjE exE)
      subgoal by (solves ‹simp add: F-Det F-Throw›)
      subgoal by (use D-F same-div in blast)
      by (auto simp add: F-Det F-Ndet F-Throw)
  next
    show ‹(s, X) ∈ ℱ ?rhs ⟹ (s, X) ∈ ℱ (?lhs Q Q')› for s X
    proof (cases s)
      show ‹s = [] ⟹ (s, X) ∈ ℱ ?rhs ⟹ (s, X) ∈ ℱ (?lhs Q Q')›
        by (auto simp add: F-Det F-Throw D-Throw T-Throw
                           is-processT6-S2 Cons-eq-append-conv)
    next
      fix a s'
      have * : ‹(a # s', X) ∈ ℱ (Throw P A Q) ⟹
              (a # s', X) ∈ ℱ (?lhs Q Q')› for Q Q'
        by (auto simp add: F-Throw F-Det F-Ndet)
      show ‹s = a # s' ⟹ (s, X) ∈ ℱ ?rhs ⟹ (s, X) ∈ ℱ (?lhs Q Q')›
        apply (simp add: F-Det, elim disjE)
        subgoal by (erule *)
        by (subst Ndet-commute) (erule *)
    qed
  qed
qed

lemma Throw-GlobalNdet:
  ‹(□a ∈ A. P a) Θ b ∈ B. Q b = □a ∈ A. (P a Θ b ∈ B. Q b)›
  ‹P' Θ a ∈ A. (□b ∈ B. Q' a b) =
   (if B = {} then P' Θ A STOP else □b ∈ B. (P' Θ a ∈ A. Q' a b))›
  by (simp add: Process-eq-spec F-Throw D-Throw
                F-GlobalNdet D-GlobalNdet T-GlobalNdet, safe, simp-all; blast)
     (simp add: Process-eq-spec F-Throw D-Throw
                F-GlobalNdet D-GlobalNdet T-GlobalNdet F-STOP D-STOP; blast)
```

lemma *Throw-disjoint-events*: ‹$A \cap$ *events-of* $P = \{\} \Longrightarrow P \ \Theta\ a \in A.\ Q\ a = P$›
proof (*subst Process-eq-spec-optimized*, *safe*)
  show ‹$A \cap$ *events-of* $P = \{\} \Longrightarrow s \in \mathcal{D}\ (\mathit{Throw}\ P\ A\ Q) \Longrightarrow s \in \mathcal{D}\ P$› **for** s
    by (*simp add*: *D-Throw disjoint-iff events-of-def*)
      (*meson in-set-conv-decomp is-processT7-S*)
next
  show ‹$A \cap$ *events-of* $P = \{\} \Longrightarrow s \in \mathcal{D}\ P \Longrightarrow s \in \mathcal{D}\ (\mathit{Throw}\ P\ A\ Q)$› **for** s
    by (*simp add*: *D-Throw disjoint-iff events-of-def image-iff*)
      (*metis* (*no-types*, *lifting*) *D-T append-Nil2 butlast-snoc front-tickFree-mono*
          front-tickFree-butlast process-charn nonTickFree-n-frontTickFree)
next
  show ‹$A \cap$ *events-of* $P = \{\} \Longrightarrow (s,\ X) \in \mathcal{F}\ (\mathit{Throw}\ P\ A\ Q) \Longrightarrow (s,\ X) \in \mathcal{F}$
P› **for** $s\ X$
    by (*simp add*: *F-Throw disjoint-iff events-of-def*)
      (*meson in-set-conv-decomp process-charn*)
next
  show ‹$A \cap$ *events-of* $P = \{\} \Longrightarrow (s,\ X) \in \mathcal{F}\ P \Longrightarrow (s,\ X) \in \mathcal{F}\ (\mathit{Throw}\ P\ A\ Q)$›
for $s\ X$
    by (*simp add*: *F-Throw disjoint-iff events-of-def image-iff*) (*metis F-T*)
qed

lemma *events-Throw*:
  ‹*events-of* $(P\ \Theta\ a \in A.\ Q\ a) \subseteq$ *events-of* $P \cup (\bigcup a \in (A \cap$ *events-of* $P).$ *events-of*
$(Q\ a))$›
proof (*intro subsetI*)
  fix e
  assume ‹$e \in$ *events-of* $(P\ \Theta\ a \in A.\ Q\ a)$›
  then obtain s **where** $*$: ‹$ev\ e \in$ *set* s› ‹$s \in \mathcal{T}\ (P\ \Theta\ a \in A.\ Q\ a)$›
    by (*simp add*: *events-of-def*) *blast*
  from $*(2)$ **consider** ‹$s \in \mathcal{T}\ P$› ‹*set* $s \cap ev$ ‘ $A = \{\}$›
    | ‹$\exists$ *t1 t2*. $s =$ *t1* @ *t2* $\wedge$ *t1* $\in \mathcal{D}\ P \wedge$ *tickFree t1* $\wedge$
          set t1 $\cap ev$ ‘ $A = \{\} \wedge$ *front-tickFree t2*›
    | ‹$\exists$ *t1 a t2*. $s =$ *t1* @ $ev\ a \#$ *t2* $\wedge$ *t1* @ $[ev\ a] \in \mathcal{T}\ P \wedge$
          set t1 $\cap ev$ ‘ $A = \{\} \wedge a \in A \wedge$ *t2* $\in \mathcal{T}\ (Q\ a)$›
    by (*simp add*: *T-Throw*) *blast*
  thus ‹$e \in$ *events-of* $P \cup (\bigcup a \in (A \cap$ *events-of* $P).$ *events-of* $(Q\ a))$›
  proof *cases*
    from $*(1)$ **show** ‹$s \in \mathcal{T}\ P \Longrightarrow$ *set* $s \cap ev$ ‘ $A = \{\} \Longrightarrow$
          $e \in$ *events-of* $P \cup (\bigcup a \in A \cap$ *events-of* $P.$ *events-of* $(Q\ a))$›
      by (*simp add*: *events-of-def*) *blast*
  next
    show ‹$\exists$ *t1 t2*. $s =$ *t1* @ *t2* $\wedge$ *t1* $\in \mathcal{D}\ P \wedge$ *tickFree t1* $\wedge$
          set t1 $\cap ev$ ‘ $A = \{\} \wedge$ *front-tickFree t2* $\Longrightarrow$
        $e \in$ *events-of* $P \cup (\bigcup a \in (A \cap$ *events-of* $P).$ *events-of* $(Q\ a))$›
      by (*metis UNIV-I UnI1 empty-iff events-div*)
  next
    assume ‹$\exists$ *t1 a t2*. $s =$ *t1* @ $ev\ a \#$ *t2* $\wedge$ *t1* @ $[ev\ a] \in \mathcal{T}\ P \wedge$

set t1 ∩ ev ‘ A = {} ∧ a ∈ A ∧ t2 ∈ T (Q a)›
then obtain *t1 a t2*
where ∗∗ : ‹*s = t1 @ ev a # t2*› ‹*t1 @ [ev a] ∈ T P*›
‹*set t1 ∩ ev ‘ A = {}*› ‹*a ∈ A*› ‹*t2 ∈ T (Q a)*› **by** *blast*
from ∗(*1*) ∗∗(*1*) **have** ‹*ev e ∈ set (t1 @ [ev a]) ∨ ev e ∈ set t2*› **by** *simp*
thus ‹*e ∈ events-of P ∪ (⋃ a ∈ (A ∩ events-of P). events-of (Q a))*›
proof (*elim disjE*)
show ‹*ev e ∈ set (t1 @ [ev a]) ⟹*
e ∈ events-of P ∪ (⋃ a∈A ∩ events-of P. events-of (Q a))›
unfolding *events-of-def* **using** ∗∗(*2*) **by** *blast*
next
show ‹*ev e ∈ set t2 ⟹ e ∈ events-of P ∪ (⋃ a∈A ∩ events-of P. events-of (Q a))*›
unfolding *events-of-def* **using** ∗∗(*2*, *4*, *5*) *mem-Collect-eq* **by** *fastforce*
qed
qed
qed

2.2.5 Key Property

lemma *Throw-Mprefix*:
‹(□*a* ∈ *A* → *P a*) Θ *b* ∈ *B*. *Q b* =
□*a* ∈ *A* → (*if a* ∈ *B then Q a else P a* 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*. *s = t1 @ t2 ∧ t1 ∈ D (Mprefix A P) ∧ tickFree t1 ∧*
set t1 ∩ ev ‘ B = {} ∧ front-tickFree t2›
| ‹∃ *t1 b t2*. *s = t1 @ ev b # t2 ∧ t1 @ [ev b] ∈ T (Mprefix A P) ∧*
set t1 ∩ ev ‘ B = {} ∧ b ∈ B ∧ t2 ∈ D (Q b)›
by (*simp add*: *D-Throw*) *blast*
thus ‹*s ∈ D ?rhs*›
proof *cases*
assume ‹∃ *t1 t2*. *s = t1 @ t2 ∧ t1 ∈ D (Mprefix A P) ∧ tickFree t1 ∧*
set t1 ∩ ev ‘ B = {} ∧ front-tickFree t2›
then obtain *t1 t2*
where ∗ : ‹*s = t1 @ t2*› ‹*t1 ∈ D (Mprefix A P)*› ‹*tickFree t1*›
‹*set t1 ∩ ev ‘ B = {}*› ‹*front-tickFree t2*› **by** *blast*
from ∗(*2*) **obtain** *a t1′* **where** ∗∗ : ‹*t1 = ev a # t1′*› ‹*a ∈ A*› ‹*t1′ ∈ D (P a)*›
by (*simp add*: *D-Mprefix*) (*metis event.inject image-iff list.collapse*)
from ∗(*4*) ∗∗(*1*) **have** ∗∗∗ : ‹*a ∉ B*› **by** (*simp add*: *image-iff*)
have ‹*t1′ @ t2 ∈ D (Throw (P a) B Q)*›
using ∗(*3*, *4*, *5*) ∗∗(*1*, *3*) **by** (*auto simp add*: *D-Throw*)
with ∗∗∗ **show** ‹*s ∈ D ?rhs*›
by (*simp add*: *D-Mprefix* ∗(*1*) ∗∗(*1*, *2*))
next
assume ‹∃ *t1 b t2*. *s = t1 @ ev b # t2 ∧ t1 @ [ev b] ∈ T (Mprefix A P) ∧*
set t1 ∩ ev ‘ B = {} ∧ b ∈ B ∧ t2 ∈ D (Q b)›

```
    then obtain t1 b t2
      where ∗ : ‹s = t1 @ ev b # t2› ‹t1 @ [ev b] ∈ 𝒯 (Mprefix A P)›
              ‹set t1 ∩ ev ‘ B = {}› ‹b ∈ B› ‹t2 ∈ 𝒟 (Q b)› by blast
    show ‹s ∈ 𝒟 ?rhs›
    proof (cases ‹t1›)
      from ∗(2) show ‹t1 = [] ⟹ s ∈ 𝒟 ?rhs›
        by (simp add: D-Mprefix T-Mprefix ∗(1, 4, 5))
    next
      fix a t1'
      assume ‹t1 = a # t1'›
      then obtain a' where ‹t1 = ev a' # t1'›
        by (metis ∗(2) append-single-T-imp-tickFree event.exhaust tickFree-Cons)
      with ∗(2, 3, 4, 5) show ‹s ∈ 𝒟 ?rhs›
        by (auto simp add: ∗(1) D-Mprefix T-Mprefix D-Throw)
    qed
  qed
next
  fix s
  assume ‹s ∈ 𝒟 ?rhs›
  then obtain a s' where ∗ : ‹a ∈ A› ‹s = ev a # s'›
                         ‹s' ∈ 𝒟 (if a ∈ B then Q a else Throw (P a) B Q)›
    by (simp add: D-Mprefix) (metis event.inject image-iff list.collapse)
  show ‹s ∈ 𝒟 ?lhs›
  proof (cases ‹a ∈ B›)
    assume ‹a ∈ B›
    hence ∗∗ : ‹[] @ [ev a] ∈ 𝒯 (Mprefix A P) ∧ set [] ∩ ev ‘ B = {} ∧ s' ∈ 𝒟 (Q
a)›
      using ∗(3) by (simp add: T-Mprefix Nil-elem-T ∗(1))
    show ‹s ∈ 𝒟 ?lhs›
      by (simp add: D-Throw) (metis ∗(2) ∗∗ ‹a ∈ B› append-Nil)
  next
    assume ‹a ∉ B›
    with ∗(2, 3)
    consider ‹∃ t1 t2. s = ev a # t1 @ t2 ∧ t1 ∈ 𝒟 (P a) ∧ tickFree t1 ∧
                       set t1 ∩ ev ‘ B = {} ∧ front-tickFree t2›
      | ‹∃ t1 b t2. s = ev a # 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
      assume ‹∃ t1 t2. s = ev a # t1 @ t2 ∧ t1 ∈ 𝒟 (P a) ∧ tickFree t1 ∧
                       set t1 ∩ ev ‘ B = {} ∧ front-tickFree t2›
      then obtain t1 t2
        where ∗∗ : ‹s = ev a # t1 @ t2› ‹t1 ∈ 𝒟 (P a)› ‹tickFree t1›
                   ‹set t1 ∩ ev ‘ B = {}› ‹front-tickFree t2› by blast
      have ∗∗∗ : ‹ev a # t1 ∈ 𝒟 (Mprefix A P) ∧ tickFree (ev a # t1) ∧
                  set (ev a # t1) ∩ ev ‘ B = {}›
        by (simp add: D-Mprefix image-iff ∗(1) ∗∗(2, 3, 4) ‹a ∉ B›)
      show ‹s ∈ 𝒟 ?lhs›
```

```
      by (simp add: D-Throw) (metis **(1, 5) *** append-Cons)
    next
      assume ‹∃ t1 b t2. s = ev a # t1 @ ev b # t2 ∧ t1 @ [ev b] ∈ 𝒯 (P a) ∧
                    set t1 ∩ ev ‘ B = {} ∧ b ∈ B ∧ t2 ∈ 𝒟 (Q b)›
      then obtain t1 b t2
        where ** : ‹s = ev a # t1 @ ev b # t2› ‹t1 @ [ev b] ∈ 𝒯 (P a)›
                ‹set t1 ∩ ev ‘ B = {}› ‹b ∈ B› ‹t2 ∈ 𝒟 (Q b)› by blast
      have *** : ‹(ev a # t1) @ [ev b] ∈ 𝒯 (Mprefix A P) ∧ set (ev a # t1) ∩ ev
‘ B = {}›
        by (simp add: T-Mprefix image-iff *(1) **(2, 3) ‹a ∉ B›)
      show ‹s ∈ 𝒟 ?lhs›
        by (simp add: D-Throw) (metis **(1, 4, 5) *** append-Cons)
    qed
  qed
next
  fix s X
  assume same-div : ‹𝒟 ?lhs = 𝒟 ?rhs›
  assume ‹(s, X) ∈ ℱ ?lhs›
  then consider ‹(s, X) ∈ ℱ (Mprefix A P)› ‹set s ∩ ev ‘ B = {}›
    | ‹s ∈ 𝒟 ?lhs›
    | ‹∃ t1 b t2. s = t1 @ ev b # t2 ∧ t1 @ [ev b] ∈ 𝒯 (Mprefix A P) ∧
                set t1 ∩ ev ‘ B = {} ∧ b ∈ B ∧ (t2, X) ∈ ℱ (Q b)›
    by (simp add: F-Throw D-Throw) blast
  thus ‹(s, X) ∈ ℱ ?rhs›
  proof cases
    show ‹(s, X) ∈ ℱ (Mprefix A P) ⟹ set s ∩ ev ‘ B = {} ⟹ (s, X) ∈ ℱ ?rhs›
      by (simp add: F-Mprefix F-Throw)
        (metis disjoint-iff hd-in-set imageI list.set-sel(2))
  next
    show ‹s ∈ 𝒟 ?lhs ⟹ (s, X) ∈ ℱ ?rhs›
      using same-div D-F by blast
  next
    assume ‹∃ t1 b t2. s = t1 @ ev b # t2 ∧ t1 @ [ev b] ∈ 𝒯 (Mprefix A P) ∧
                  set t1 ∩ ev ‘ B = {} ∧ b ∈ B ∧ (t2, X) ∈ ℱ (Q b)›
    then obtain t1 b t2
      where * : ‹s = t1 @ ev b # t2› ‹t1 @ [ev b] ∈ 𝒯 (Mprefix A P)›
              ‹set t1 ∩ ev ‘ B = {}› ‹b ∈ B› ‹(t2, X) ∈ ℱ (Q b)› by blast
    show ‹(s, X) ∈ ℱ ?rhs›
    proof (cases t1)
      from *(2) show ‹t1 = [] ⟹ (s, X) ∈ ℱ ?rhs›
        by (auto simp add: F-Mprefix T-Mprefix F-Throw *(1, 4, 5))
    next
      fix a t1'
      assume ‹t1 = a # t1'›
      then obtain a' where ‹t1 = ev a' # t1'›
        by (metis *(2) append-single-T-imp-tickFree event.exhaust tickFree-Cons)
      with *(2, 3, 5) show ‹(s, X) ∈ ℱ ?rhs›
        by (auto simp add: F-Mprefix T-Mprefix F-Throw *(1, 4))
    qed
```

```
  qed
next
  show ‹(s, X) ∈ ℱ ?rhs ⟹ (s, X) ∈ ℱ ?lhs› for s X
  proof (cases s)
    show ‹s = [] ⟹ (s, X) ∈ ℱ ?rhs ⟹ (s, X) ∈ ℱ ?lhs›
      by (simp add: F-Mprefix F-Throw)
  next
    fix a s'
    assume assms : ‹s = a # s'› ‹(s, X) ∈ ℱ ?rhs›
    from assms(2) obtain a'
      where ∗ : ‹a' ∈ A› ‹s = ev a' # s'›
            ‹(s', X) ∈ ℱ (if a' ∈ B then Q a' else Throw (P a') B Q)›
      by (simp add: assms(1) F-Mprefix) blast
    show ‹(s, X) ∈ ℱ ?lhs›
    proof (cases ‹a' ∈ B›)
      assume ‹a' ∈ B›
      hence ∗∗ : ‹[] @ [ev a'] ∈ 𝒯 (Mprefix A P) ∧
                  set [] ∩ ev ‘ B = {} ∧ (s', X) ∈ ℱ (Q a')›
        using ∗(3) by (simp add: T-Mprefix Nil-elem-T ∗(1))
      show ‹(s, X) ∈ ℱ ?lhs›
        by (simp add: F-Throw) (metis ∗(2) ∗∗ ‹a' ∈ B› append-Nil)
    next
      assume ‹a' ∉ B›
      then consider ‹(s', X) ∈ ℱ (P a')› ‹set s' ∩ ev ‘ B = {}›
        | ‹∃ t1 t2. s' = t1 @ t2 ∧ t1 ∈ 𝒟 (P a') ∧ tickFree t1 ∧
                    set t1 ∩ ev ‘ B = {} ∧ front-tickFree t2›
        | ‹∃ t1 b t2. s' = t1 @ ev b # t2 ∧ t1 @ [ev b] ∈ 𝒯 (P a') ∧
                      set t1 ∩ ev ‘ B = {} ∧ b ∈ B ∧ (t2, X) ∈ ℱ (Q b)›
        using ∗(3) by (simp add: F-Throw D-Throw) blast
      thus ‹(s, X) ∈ ℱ ?lhs›
      proof cases
        show ‹(s', X) ∈ ℱ (P a') ⟹ set s' ∩ ev ‘ B = {} ⟹ (s, X) ∈ ℱ ?lhs›
          by (simp add: F-Mprefix F-Throw ∗(1, 2) ‹a' ∉ B› image-iff)
      next
        assume ‹∃ t1 t2. s' = t1 @ t2 ∧ t1 ∈ 𝒟 (P a') ∧ tickFree t1 ∧
                         set t1 ∩ ev ‘ B = {} ∧ front-tickFree t2›
        then obtain t1 t2
          where ∗∗ : ‹s' = t1 @ t2› ‹t1 ∈ 𝒟 (P a')› ‹tickFree t1›
                ‹set t1 ∩ ev ‘ B = {}› ‹front-tickFree t2› by blast
        have ∗∗∗ : ‹s = (ev a' # t1) @ t2 ∧ ev a' # t1 ∈ 𝒟 (Mprefix A P) ∧
                  tickFree (ev a' # t1) ∧ set (ev a' # t1) ∩ ev ‘ B = {}›
          by (simp add: D-Mprefix ‹a' ∉ B› image-iff ∗(1, 2) ∗∗(1, 2, 3, 4))
        show ‹(s, X) ∈ ℱ ?lhs›
          by (simp add: F-Throw F-Mprefix) (metis ∗∗(5) ∗∗∗)
      next
        assume ‹∃ t1 b t2. s' = t1 @ ev b # t2 ∧ t1 @ [ev b] ∈ 𝒯 (P a') ∧
                           set t1 ∩ ev ‘ B = {} ∧ b ∈ B ∧ (t2, X) ∈ ℱ (Q b)›
        then obtain 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 blast
      have ∗∗∗ : ‹s = (ev a′ # t1) @ ev b # t2 ∧ set (ev a′ # t1) ∩ ev ‘ B = {}
∧
                (ev a′ # t1) @ [ev b] ∈ 𝒯 (Mprefix A P)›
        by (simp add: T-Mprefix ‹a′ ∉ B› image-iff ∗(1, 2) ∗∗(1, 2, 3))
       show ‹(s, X) ∈ ℱ ?lhs›
        by (simp add: F-Throw F-Mprefix) (metis ∗∗(4, 5) ∗∗∗)
     qed
   qed
  qed
qed


corollary Throw-prefix: ‹(a → P) Θ b ∈ B. Q b =
                        (a → (if a ∈ B then Q a else (P Θ b ∈ B. Q b)))›
 unfolding write0-def by (auto simp add: Throw-Mprefix intro: mono-Mprefix-eq)

corollary Throw-Mndetprefix:
  ‹(⊓a ∈ A → P a) Θ b ∈ B. Q b =
   ⊓a ∈ A → (if a ∈ B then Q a else P a Θ b ∈ B. Q b)›
  apply (subst Mndetprefix-GlobalNdet)
  apply (simp add: Throw-GlobalNdet(1) Throw-prefix)
  apply (subst Mndetprefix-GlobalNdet[symmetric])
  by simp


— We may prove some results about deadlock freeness as corollaries
```

2.2.6 Continuity

```
lemma chain-left-Throw: ‹chain Y ⟹ chain (λi. Y i Θ a ∈ A. Q a)›
  by (simp add: chain-def)

lemma chain-right-Throw: ‹chain Y ⟹ chain (λi. P Θ a ∈ A. Y i a)›
  by (simp add: chain-def fun-belowD)


lemma cont-left-prem-Throw :
  ‹(⨆ i. Y i) Θ a ∈ A. Q a = (⨆ i. Y i Θ a ∈ A. Q a)›
  (is ‹?lhs = ?rhs›) if chain : ‹chain Y›
proof (subst Process-eq-spec, safe)
 show ‹s ∈ 𝒟 ?lhs ⟹ s ∈ 𝒟 ?rhs› for s
   by (auto simp add: limproc-is-thelub chain
                      chain-left-Throw D-Throw T-LUB D-LUB)
next
  fix s
  define S
    where ‹S i ≡ {t1. ∃ t2. s = t1 @ t2 ∧ t1 ∈ 𝒟 (Y i) ∧ tickFree t1 ∧
                          set t1 ∩ ev ‘ A = {} ∧ front-tickFree t2} ∪
```

$\{t1.\ \exists\, a\ t2.\ s = t1\ @\ ev\ a\ \#\ t2 \wedge t1\ @\ [ev\ a] \in \mathcal{T}\ (Y\ i) \wedge tickFree\ t1 \wedge$
$set\ t1 \cap ev\ `\ A = \{\} \wedge a \in A \wedge t2 \in \mathcal{D}\ (Q\ a)\}\rangle$ **for** i
assume $\langle s \in \mathcal{D}\ ?rhs\rangle$
hence *ftF*: $\langle$*front-tickFree s*$\rangle$ **using** *D-imp-front-tickFree* **by** *blast*
from $\langle s \in \mathcal{D}\ ?rhs\rangle$ **have** $\langle s \in \mathcal{D}\ (Y\ i\ \Theta\ a \in A.\ Q\ a)\rangle$ **for** i
by (*simp add*: *limproc-is-thelub D-LUB chain-left-Throw chain*)
hence $\langle\forall\, i.\ S\ i \neq \{\}\rangle$
by (*simp add*: *S-def D-Throw*)
(*metis ftF front-tickFree-mono list.distinct*(1))
moreover have $\langle$*finite* $(S\ 0)\rangle$
unfolding *S-def*
apply (*rule finite-subset*[*of* - $\langle\{t1.\ \exists\, t2.\ s = t1\ @\ t2\}\rangle$], *blast*)
by (*metis prefixes-fin*)
moreover have $\langle\forall\, i.\ S\ (Suc\ i) \subseteq S\ i\rangle$
unfolding *S-def* **apply** (*intro allI Un-mono subsetI*; *simp*)
by (*metis in-mono le-approx1 po-class.chainE chain*)
(*metis le-approx-lemma-T po-class.chain-def subset-eq chain*)
ultimately have $\langle(\bigcap i.\ S\ i) \neq \{\}\rangle$
by (*rule Inter-nonempty-finite-chained-sets*)
then obtain *t1* **where** $*$: $\langle\forall\, i.\ t1 \in S\ i\rangle$
by (*meson INT-iff ex-in-conv iso-tuple-UNIV-I*)
show $\langle s \in \mathcal{D}\ ?lhs\rangle$
proof (*cases* $\langle\exists\, j\ a\ t2.\ s = t1\ @\ ev\ a\ \#\ t2 \wedge t1\ @\ [ev\ a] \in \mathcal{T}\ (Y\ j) \wedge a \in A \wedge t2 \in \mathcal{D}\ (Q\ a)\rangle$)
case *True*
then obtain *j a t2* **where** $**$: $\langle s = t1\ @\ ev\ a\ \#\ t2\rangle$ $\langle t1\ @\ [ev\ a] \in \mathcal{T}\ (Y\ j)\rangle$ $\langle a \in A\rangle$ $\langle t2 \in \mathcal{D}\ (Q\ a)\rangle$ **by** *blast*
from $*$ $**(1)$ **have** $\langle\forall\, i.\ t1\ @\ [ev\ a] \in \mathcal{T}\ (Y\ i)\rangle$
by (*simp add*: *S-def*) (*meson D-T front-tickFree-single is-processT7-S*)
with $*$ $**(1, 3, 4)$ **show** $\langle s \in \mathcal{D}\ ?lhs\rangle$
by (*simp add*: *S-def D-Throw limproc-is-thelub chain T-LUB*) *blast*
next
case *False*
with $*$ **have** $\langle\forall\, i.\ \exists\, t2.\ s = t1\ @\ t2 \wedge t1 \in \mathcal{D}\ (Y\ i) \wedge$ *front-tickFree t2*$\rangle$
by (*simp add*: *S-def*) *blast*
hence $\langle\exists\, t2.\ s = t1\ @\ t2 \wedge (\forall\, i.\ t1 \in \mathcal{D}\ (Y\ i)) \wedge$ *front-tickFree t2*$\rangle$ **by** *blast*
with $*$ **show** $\langle s \in \mathcal{D}\ ?lhs\rangle$
by (*simp add*: *S-def D-Throw limproc-is-thelub chain D-LUB*) *blast*
qed
next
show $\langle(s, X) \in \mathcal{F}\ ?lhs \Longrightarrow (s, X) \in \mathcal{F}\ ?rhs\rangle$ **for** $s\ X$
by (*auto simp add*: *limproc-is-thelub chain chain-left-Throw F-Throw F-LUB T-LUB D-LUB*)
next
fix $s\ X$
define S
where $\langle S\ i \equiv \{t1.\ s = t1 \wedge (t1, X) \in \mathcal{F}\ (Y\ i) \wedge set\ t1 \cap ev\ `\ A = \{\}\} \cup$
$\{t1.\ \exists\, t2.\ s = t1\ @\ t2 \wedge t1 \in \mathcal{D}\ (Y\ i) \wedge tickFree\ t1 \wedge$

$$\textit{set } t1 \cap \textit{ev} \mathbin{`} A = \{\} \wedge \textit{front-tickFree } t2\} \cup$$
$$\{t1.\ \exists\, a\ t2.\ s = t1 \mathbin{@} \textit{ev}\ a \mathbin{\#} t2 \wedge t1 \mathbin{@} [\textit{ev}\ a] \in \mathcal{T}\ (Y\ i) \wedge$$
$$\textit{set } t1 \cap \textit{ev} \mathbin{`} A = \{\} \wedge a \in A \wedge (t2,\ X) \in \mathcal{F}\ (Q\ a)\}$$ **for** i

assume ‹$(s,\ X) \in \mathcal{F}$ *?rhs*›
hence *ftF*: ‹*front-tickFree s*› **using** *is-processT2* **by** *blast*
from ‹$(s,\ X) \in \mathcal{F}$ *?rhs*› **have** ‹$(s,\ X) \in \mathcal{F}\ (Y\ i \ominus a \in A.\ Q\ a)$› **for** i
  by (*simp add*: *limproc-is-thelub F-LUB chain-left-Throw chain*)

hence ‹$\forall\, i.\ S\ i \neq \{\}$› **by** (*simp add*: *S-def F-Throw*, *safe*; *simp*, *blast*)
moreover have ‹*finite* $(S\ 0)$›
  unfolding *S-def*
  apply (*intro finite-UnI*)
  apply (*all* ‹*rule finite-subset*[*of* - ‹$\{t1.\ \exists\, t2.\ s = t1 \mathbin{@} t2\}$›], *blast*›)
  by (*metis prefixes-fin*)+
moreover have ‹$\forall\, i.\ S\ (\textit{Suc}\ i) \subseteq S\ i$›
  unfolding *S-def* **apply** (*intro allI Un-mono subsetI*; *simp*)
  subgoal by (*meson is-processT8 po-class.chainE proc-ord2a chain*)
  subgoal by (*metis in-mono le-approx1 po-class.chainE chain*)
  by (*metis le-approx-lemma-T po-class.chain-def subset-eq chain*)
ultimately have ‹$(\bigcap i.\ S\ i) \neq \{\}$›
  by (*rule Inter-nonempty-finite-chained-sets*)
then obtain *t1* **where** $*$: ‹$\forall\, i.\ t1 \in S\ i$›
  by (*meson INT-iff ex-in-conv iso-tuple-UNIV-I*)
show ‹$(s,\ X) \in \mathcal{F}$ *?lhs*›
proof (*cases* ‹$\exists\, j\ a\ t2.\ s = t1 \mathbin{@} \textit{ev}\ a \mathbin{\#} t2 \wedge t1 \mathbin{@} [\textit{ev}\ a] \in \mathcal{T}\ (Y\ j) \wedge$
  $a \in A \wedge (t2,\ X) \in \mathcal{F}\ (Q\ a)$›)
  case *True1* : *True*
  then obtain *j a t2* **where** $**$: ‹$s = t1 \mathbin{@} \textit{ev}\ a \mathbin{\#} t2$› ‹$t1 \mathbin{@} [\textit{ev}\ a] \in \mathcal{T}\ (Y\ j)$›
    ‹$a \in A$› ‹$(t2,\ X) \in \mathcal{F}\ (Q\ a)$› **by** *blast*
  from $*$ $**(1)$ **have** ‹$\forall\, i.\ t1 \mathbin{@} [\textit{ev}\ a] \in \mathcal{T}\ (Y\ i)$›
    by (*simp add*: *S-def*) (*meson D-T front-tickFree-single is-processT7*)
  with $*$ $**(1,\ 3,\ 4)$ **show** ‹$(s,\ X) \in \mathcal{F}$ *?lhs*›
    by (*simp add*: *S-def F-Throw limproc-is-thelub chain T-LUB*) *blast*
next
  case *False1* : *False*
  show ‹$(s,\ X) \in \mathcal{F}$ *?lhs*›
  proof (*cases* ‹$\forall\, i.\ t1 \in \mathcal{D}\ (Y\ i)$›)
    case *True2* : *True*
    with $*$ **show** ‹$(s,\ X) \in \mathcal{F}$ *?lhs*›
      by (*simp add*: *S-def F-Throw limproc-is-thelub chain*)
        (*metis D-LUB-2 append-Nil2 front-tickFree-mono ftF process-charn chain*)
  next
    case *False2* : *False*
    then obtain *j* **where** ‹$t1 \notin \mathcal{D}\ (Y\ j)$› **by** *blast*
    with *False1* $*$ **have** $**$: ‹$s = t1 \wedge (t1,\ X) \in \mathcal{F}\ (Y\ j) \wedge \textit{set } t1 \cap \textit{ev} \mathbin{`} A = \{\}$›
      by (*simp add*: *S-def*) *blast*
    with $*$ *D-F* **have** ‹$\forall\, i.\ (t1,\ X) \in \mathcal{F}\ (Y\ i)$› **by** (*auto simp add*: *S-def*)
    with $**$ **show** ‹$(s,\ X) \in \mathcal{F}$ *?lhs*›
      by (*simp add*: *F-Throw limproc-is-thelub F-LUB chain*)

```
    qed
  qed
qed


lemma cont-right-prem-Throw :
  ‹P ⊖ a ∈ A. (⨆ i. Y i a) = (⨆ i. P ⊖ a ∈ A. Y i a)›
  (is ‹?lhs = ?rhs›) if chain : ‹chain Y›
proof (subst Process-eq-spec, safe)
  show ‹s ∈ 𝒟 ?lhs ⟹ s ∈ 𝒟 ?rhs› for s
    by (simp add: limproc-is-thelub chain chain-right-Throw
                  ch2ch-fun[OF chain] D-Throw D-LUB) blast
next
  fix s
  assume ‹s ∈ 𝒟 ?rhs›
  define S
    where ‹S i ≡ {t1. ∃ t2. s = t1 @ t2 ∧ t1 ∈ 𝒟 P ∧ tickFree t1 ∧
                        set t1 ∩ ev ‘ A = {} ∧ front-tickFree t2} ∪
               {t1. ∃ a t2. s = t1 @ ev a # t2 ∧ t1 @ [ev a] ∈ 𝒯 P ∧
                         set t1 ∩ ev ‘ A = {} ∧ a ∈ A ∧ t2 ∈ 𝒟 (Y i a)}› for i
  assume ‹s ∈ 𝒟 ?rhs›
  hence ‹s ∈ 𝒟 (P ⊖ a ∈ A. Y i a)› for i
    by (simp add: limproc-is-thelub D-LUB chain-right-Throw chain)
  hence ‹∀ i. S i ≠ {}› by (simp add: S-def D-Throw) metis
  moreover have ‹finite (S 0)›
    unfolding S-def
    apply (rule finite-subset[of - ‹{t1. ∃ t2. s = t1 @ t2}›], blast)
    by (metis prefixes-fin)
  moreover have ‹∀ i. S (Suc i) ⊆ S i›
    unfolding S-def apply (intro allI Un-mono subsetI; simp)
    by (metis fun-belowD le-approx1 po-class.chainE subset-iff chain)
  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)
  then consider ‹t1 ∈ 𝒟 P› ‹tickFree t1›
                ‹set t1 ∩ ev ‘ A = {}› ‹∃ t2. s = t1 @ t2 ∧ front-tickFree t2›
    | ‹set t1 ∩ ev ‘ A = {}›
      ‹∀ i. ∃ a t2. s = t1 @ ev a # t2 ∧ t1 @ [ev a] ∈ 𝒯 P ∧ a ∈ A ∧ t2 ∈ 𝒟 (Y i
a)›
    by (simp add: S-def) blast
  thus ‹s ∈ 𝒟 ?lhs›
  proof cases
    show ‹t1 ∈ 𝒟 P ⟹ tickFree t1 ⟹ set t1 ∩ ev ‘ A = {} ⟹
          ∃ t2. s = t1 @ t2 ∧ front-tickFree t2 ⟹ s ∈ 𝒟 ?lhs›
      by (simp add: D-Throw) blast
  next
    assume assms: ‹set t1 ∩ ev ‘ A = {}›
                  ‹∀ i. ∃ a t2. s = t1 @ ev a # t2 ∧ t1 @ [ev a] ∈ 𝒯 P ∧
```

```
                    a ∈ A ∧ t2 ∈ 𝒟 (Y i a)›
    from assms(2) obtain a t2
      where ∗ : ‹s = t1 @ ev a # t2› ‹t1 @ [ev a] ∈ 𝒯 P› ‹a ∈ A› by blast
    with assms(2) have ‹∀ i. t2 ∈ 𝒟 (Y i a)› by blast
    with assms(1) ∗(1, 2, 3) show ‹s ∈ 𝒟 ?lhs›
      by (simp add: D-Throw limproc-is-thelub chain ch2ch-fun D-LUB) blast
  qed
next
  show ‹(s, X) ∈ ℱ ?lhs ⟹ (s, X) ∈ ℱ ?rhs› for s X
    by (simp add: limproc-is-thelub chain chain-right-Throw
                  ch2ch-fun[OF chain] F-Throw F-LUB T-LUB D-LUB) blast
next
  fix s X
  define S
    where ‹S i ≡ {t1. s = t1 ∧ (t1, X) ∈ ℱ P ∧ set t1 ∩ ev ‘ A = {}} ∪
                 {t1. ∃ t2. s = t1 @ t2 ∧ t1 ∈ 𝒟 P ∧ tickFree t1 ∧
                       set t1 ∩ ev ‘ A = {} ∧ front-tickFree t2} ∪
                 {t1. ∃ a t2. s = t1 @ ev a # t2 ∧ t1 @ [ev a] ∈ 𝒯 P ∧
                       set t1 ∩ ev ‘ A = {} ∧ a ∈ A ∧ (t2, X) ∈ ℱ (Y i a)}› for i
  assume ‹(s, X) ∈ ℱ ?rhs›
  hence ‹(s, X) ∈ ℱ (P ⊖ a ∈ A. Y i a)› for i
    by (simp add: limproc-is-thelub F-LUB chain-right-Throw chain)
  hence ‹∀ i. S i ≠ {}› by (simp add: S-def F-Throw, safe; simp, blast)
  moreover have ‹finite (S 0)›
    unfolding S-def
    apply (intro finite-UnI)
    apply (all ‹rule finite-subset[of - ‹{t1. ∃ t2. s = t1 @ t2}›], blast›)
    by (metis prefixes-fin)+
  moreover have ‹∀ i. S (Suc i) ⊆ S i›
    unfolding S-def apply (intro allI Un-mono subsetI; simp)
    by (metis NF-ND fun-below-iff po-class.chain-def proc-ord2a chain)
  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)
  then consider ‹s = t1 ∧ (t1, X) ∈ ℱ P› ‹set t1 ∩ ev ‘ A = {}›
    | ‹∃ t2. s = t1 @ t2 ∧ t1 ∈ 𝒟 P ∧ tickFree t1 ∧
          set t1 ∩ ev ‘ A = {} ∧ front-tickFree t2›
    | ‹set t1 ∩ ev ‘ A = {}›
      ‹∀ i. ∃ a t2. s = t1 @ ev a # t2 ∧ t1 @ [ev a] ∈ 𝒯 P ∧ a ∈ A ∧ (t2, X) ∈
ℱ (Y i a)›
    by (simp add: S-def) blast
  thus ‹(s, X) ∈ ℱ ?lhs›
  proof cases
    show ‹s = t1 ∧ (t1, X) ∈ ℱ P ⟹ set t1 ∩ ev ‘ A = {} ⟹ (s, X) ∈ ℱ ?lhs›
      by (simp add: F-Throw)
  next
    show ‹∃ t2. s = t1 @ t2 ∧ t1 ∈ 𝒟 P ∧ tickFree t1 ∧
            set t1 ∩ ev ‘ A = {} ∧ front-tickFree t2 ⟹ (s, X) ∈ ℱ ?lhs›
```

```
    by (simp add: F-Throw) blast
  next
    assume assms: ‹set t1 ∩ ev ‘ A = {}›
                  ‹∀ i. ∃ a t2. s = t1 @ ev a # t2 ∧ t1 @ [ev a] ∈ 𝒯 P ∧
                                a ∈ A ∧ (t2, X) ∈ ℱ (Y i a)›
    from this(2) obtain a t2
      where ∗ : ‹s = t1 @ ev a # t2› ‹t1 @ [ev a] ∈ 𝒯 P› ‹a ∈ A› by blast
    with assms(2) have ‹∀ i. (t2, X) ∈ ℱ (Y i a)› by blast
    with ∗(1, 2, 3) assms(1) show ‹(s, X) ∈ ℱ ?lhs›
      by (simp add: F-Throw limproc-is-thelub ch2ch-fun chain F-LUB) blast
  qed
qed


lemma Throw-cont[simp] :
  assumes cont-f : ‹cont f› and cont-g : ‹∀ a. cont (g a)›
  shows ‹cont (λx. f x Θ a ∈ A. g a x)›
proof −
  have ∗ : ‹cont (λy. y Θ a ∈ A. g a x)› for x
    by (rule contI2, rule monofunI, solves simp, simp add: cont-left-prem-Throw)
  have ‹cont (Throw y A)› for y
    by (simp add: contI2 cont-right-prem-Throw fun-belowD lub-fun monofunI)
  hence ∗∗ : ‹cont (λx. y Θ a ∈ A. g a x)› for y
    by (rule cont-compose) (simp add: cont-g)
  show ?thesis by (fact cont-apply[OF cont-f ∗ ∗∗])
qed

end
```

2.3 The Interrupt Operator

```
theory Interrupt
  imports HOL−CSPM.CSPM
begin
```

2.3.1 Definition

We want to add the binary operator of interruption of P by Q: it behaves like P except that at any time Q can take over.

The definition provided by Roscoe [4, p.239] does not respect the invariant *is-process*: it seems like *tick* is not handled.

We propose here our corrected version.

```
lift-definition Interrupt :: ‹['α process, 'α process] ⇒ 'α process› (infixl ‹△› 75)
  is ‹λP Q.
```

```
  ({(t1 @ [tick], X)| t1 X. t1 @ [tick] ∈ T P} ∪
   {(t1, X − {tick})| t1 X. t1 @ [tick] ∈ T P} ∪
   {(t1, X) ∈ F P. tickFree t1 ∧ ([], X) ∈ F Q} ∪
   {(t1 @ t2, X)| t1 t2 X. t1 ∈ T P ∧ tickFree t1 ∧ (t2, X) ∈ F Q ∧ t2 ≠ []} ∪
   {(t1, X − {tick})| t1 X. t1 ∈ T P ∧ tickFree t1 ∧ [tick] ∈ T Q} ∪
   {(t1, X). t1 ∈ D P} ∪
   {(t1 @ t2, X)| t1 t2 X. t1 ∈ T P ∧ tickFree t1 ∧ t2 ∈ D Q},
   D P ∪ {t1 @ t2| t1 t2. t1 ∈ T P ∧ tickFree t1 ∧ t2 ∈ D Q})›
proof −
  show ‹?thesis P Q›
  (is ‹is-process (?f1 ∪ ?f2 ∪ ?f3 ∪ ?f4 ∪ ?f5 ∪ ?f6 ∪ ?f7, ?d1 ∪ ?d2)›) for P Q
    unfolding is-process-def FAILURES-def DIVERGENCES-def fst-conv snd-conv
  proof (intro conjI allI impI)
    have ‹([], {}) ∈ ?f3› by (simp add: is-processT1)
    thus ‹([], {}) ∈ ?f1 ∪ ?f2 ∪ ?f3 ∪ ?f4 ∪ ?f5 ∪ ?f6 ∪ ?f7› by fast
  next
    show ‹(s, X) ∈ ?f1 ∪ ?f2 ∪ ?f3 ∪ ?f4 ∪ ?f5 ∪ ?f6 ∪ ?f7 ⟹ front-tickFree
s› for s X
      by (simp add: is-processT2 D-imp-front-tickFree front-tickFree-append)
        (meson front-tickFree-append front-tickFree-dw-closed is-processT2-TR pro-
cess-charn)
  next
    fix s t
    show ‹(s @ t, {}) ∈ ?f1 ∪ ?f2 ∪ ?f3 ∪ ?f4 ∪ ?f5 ∪ ?f6 ∪ ?f7 ⟹
            (s, {}) ∈ ?f1 ∪ ?f2 ∪ ?f3 ∪ ?f4 ∪ ?f5 ∪ ?f6 ∪ ?f7›
    proof (induct t rule: rev-induct)
      show ‹(s @ [], {}) ∈ ?f1 ∪ ?f2 ∪ ?f3 ∪ ?f4 ∪ ?f5 ∪ ?f6 ∪ ?f7 ⟹
              (s, {}) ∈ ?f1 ∪ ?f2 ∪ ?f3 ∪ ?f4 ∪ ?f5 ∪ ?f6 ∪ ?f7› by simp
    next
      fix a t
      assume assm : ‹(s @ t @ [a], {}) ∈ ?f1 ∪ ?f2 ∪ ?f3 ∪ ?f4 ∪ ?f5 ∪ ?f6 ∪
?f7›
        and hyp : ‹(s @ t, {}) ∈ ?f1 ∪ ?f2 ∪ ?f3 ∪ ?f4 ∪ ?f5 ∪ ?f6 ∪ ?f7 ⟹
                    (s, {}) ∈ ?f1 ∪ ?f2 ∪ ?f3 ∪ ?f4 ∪ ?f5 ∪ ?f6 ∪ ?f7›
      from assm have ‹(s @ t @ [a], {}) ∈ ?f1 ∨ (s @ t @ [a], {}) ∈ ?f2 ∨
        (s @ t @ [a], {}) ∈ ?f3 ∨ (s @ t @ [a], {}) ∈ ?f4 ∨ (s @ t @ [a], {}) ∈ ?f5
∨
        (s @ t @ [a], {}) ∈ ?f6 ∨ (s @ t @ [a], {}) ∈ ?f7› by fast
      thus ‹(s, {}) ∈ ?f1 ∪ ?f2 ∪ ?f3 ∪ ?f4 ∪ ?f5 ∪ ?f6 ∪ ?f7›
      proof (elim disjE)
        assume ‹(s @ t @ [a], {}) ∈ ?f1›
        hence ‹(s, {}) ∈ ?f3›
       by simp (meson NF-NT append-T-imp-tickFree is-processT snoc-eq-iff-butlast)

        thus ‹(s, {}) ∈ ?f1 ∪ ?f2 ∪ ?f3 ∪ ?f4 ∪ ?f5 ∪ ?f6 ∪ ?f7› by blast
      next
        assume ‹(s @ t @ [a], {}) ∈ ?f2›
        hence ‹(s, {}) ∈ ?f3›
       by simp (metis NF-NT Nil-is-append-conv append-T-imp-tickFree is-processT
```

```
list.discI)
      thus ‹(s, {}) ∈ ?f1 ∪ ?f2 ∪ ?f3 ∪ ?f4 ∪ ?f5 ∪ ?f6 ∪ ?f7› by blast
    next
      assume ‹(s @ t @ [a], {}) ∈ ?f3›
      with is-processT3 have ‹(s, {}) ∈ ?f3› by simp blast
      thus ‹(s, {}) ∈ ?f1 ∪ ?f2 ∪ ?f3 ∪ ?f4 ∪ ?f5 ∪ ?f6 ∪ ?f7› by blast
    next
      assume ‹(s @ t @ [a], {}) ∈ ?f4›
      then obtain t1 t2 where ∗ : ‹s @ t = t1 @ t2› ‹t1 ∈ T P›
                               ‹tickFree t1› ‹(t2 @ [a], {}) ∈ F Q›
        by simp (metis butlast-append last-appendR snoc-eq-iff-butlast)
      show ‹(s, {}) ∈ ?f1 ∪ ?f2 ∪ ?f3 ∪ ?f4 ∪ ?f5 ∪ ?f6 ∪ ?f7›
      proof (cases ‹t2 = []›)
        assume ‹t2 = []›
        with ∗(1, 2, 3) have ‹(s, {}) ∈ ?f3›
          by simp (metis T-F process-charn tickFree-append)
        thus ‹(s, {}) ∈ ?f1 ∪ ?f2 ∪ ?f3 ∪ ?f4 ∪ ?f5 ∪ ?f6 ∪ ?f7› by blast
      next
        assume ‹t2 ≠ []›
        with ∗ is-processT3 have ‹(s @ t, {}) ∈ ?f4› by simp blast
        thus ‹(s, {}) ∈ ?f1 ∪ ?f2 ∪ ?f3 ∪ ?f4 ∪ ?f5 ∪ ?f6 ∪ ?f7› by (intro hyp)
blast
      qed
    next
      assume ‹(s @ t @ [a], {}) ∈ ?f5›
      hence ‹(s, {}) ∈ ?f3› by simp (metis T-F process-charn)
      thus ‹(s, {}) ∈ ?f1 ∪ ?f2 ∪ ?f3 ∪ ?f4 ∪ ?f5 ∪ ?f6 ∪ ?f7› by blast
    next
      assume ‹(s @ t @ [a], {}) ∈ ?f6›
      hence ‹(s, {}) ∈ ?f3›
        by simp (meson front-tickFree-mono is-processT snoc-eq-iff-butlast)
      thus ‹(s, {}) ∈ ?f1 ∪ ?f2 ∪ ?f3 ∪ ?f4 ∪ ?f5 ∪ ?f6 ∪ ?f7› by blast
    next
      assume ‹(s @ t @ [a], {}) ∈ ?f7›
      then obtain t1 t2 where ∗ : ‹s @ t @ [a] = t1 @ t2› ‹t1 ∈ T P›
                               ‹tickFree t1› ‹t2 ∈ D Q› by blast
      hence ‹(s @ t, {}) ∈ (if length t2 ≤ 1 then ?f3 else ?f4)›
        apply (cases t2 rule: rev-cases; simp)
        by (metis T-F append-assoc process-charn tickFree-append)
           (metis D-T T-F is-processT3-ST)
      thus ‹(s, {}) ∈ ?f1 ∪ ?f2 ∪ ?f3 ∪ ?f4 ∪ ?f5 ∪ ?f6 ∪ ?f7›
        by (intro hyp) (meson UnI1 UnI2)
    qed
  qed
next
  fix s X Y
  assume assm : ‹(s, Y) ∈ ?f1 ∪ ?f2 ∪ ?f3 ∪ ?f4 ∪ ?f5 ∪ ?f6 ∪ ?f7 ∧ X ⊆ Y›
  hence ‹(s, Y) ∈ ?f1 ∨ (s, Y) ∈ ?f2 ∨ (s, Y) ∈ ?f3 ∨ (s, Y) ∈ ?f4 ∨
        (s, Y) ∈ ?f5 ∨ (s, Y) ∈ ?f6 ∨ (s, Y) ∈ ?f7› by fast
```

thus ‹$(s,\ X) \in$ *?f1* $\cup$ *?f2* $\cup$ *?f3* $\cup$ *?f4* $\cup$ *?f5* $\cup$ *?f6* $\cup$ *?f7*›
proof (*elim disjE*)
assume ‹$(s,\ Y) \in$ *?f1*›
hence ‹$(s,\ X) \in$ *?f1*› **by** *simp*
thus ‹$(s,\ X) \in$ *?f1* $\cup$ *?f2* $\cup$ *?f3* $\cup$ *?f4* $\cup$ *?f5* $\cup$ *?f6* $\cup$ *?f7*› **by** *blast*
next
assume ‹$(s,\ Y) \in$ *?f2*›
with *assm*[*THEN conjunct2*] **have** ‹$(s,\ X) \in$ *?f2*› **by** *simp blast*
thus ‹$(s,\ X) \in$ *?f1* $\cup$ *?f2* $\cup$ *?f3* $\cup$ *?f4* $\cup$ *?f5* $\cup$ *?f6* $\cup$ *?f7*› **by** *blast*
next
assume ‹$(s,\ Y) \in$ *?f3*›
with *assm*[*THEN conjunct2*] *is-processT4* **have** ‹$(s,\ X) \in$ *?f3*› **by** *simp blast*
thus ‹$(s,\ X) \in$ *?f1* $\cup$ *?f2* $\cup$ *?f3* $\cup$ *?f4* $\cup$ *?f5* $\cup$ *?f6* $\cup$ *?f7*› **by** *blast*
next
assume ‹$(s,\ Y) \in$ *?f4*›
with *assm*[*THEN conjunct2*] *is-processT4* **have** ‹$(s,\ X) \in$ *?f4*› **by** *simp blast*
thus ‹$(s,\ X) \in$ *?f1* $\cup$ *?f2* $\cup$ *?f3* $\cup$ *?f4* $\cup$ *?f5* $\cup$ *?f6* $\cup$ *?f7*› **by** *blast*
next
assume ‹$(s,\ Y) \in$ *?f5*›
with *assm*[*THEN conjunct2*] **have** ‹$(s,\ X) \in$ *?f5*› **by** *simp blast*
thus ‹$(s,\ X) \in$ *?f1* $\cup$ *?f2* $\cup$ *?f3* $\cup$ *?f4* $\cup$ *?f5* $\cup$ *?f6* $\cup$ *?f7*› **by** *blast*
next
assume ‹$(s,\ Y) \in$ *?f6*›
hence ‹$(s,\ X) \in$ *?f6*› **by** *simp*
thus ‹$(s,\ X) \in$ *?f1* $\cup$ *?f2* $\cup$ *?f3* $\cup$ *?f4* $\cup$ *?f5* $\cup$ *?f6* $\cup$ *?f7*› **by** *blast*
next
assume ‹$(s,\ Y) \in$ *?f7*›
hence ‹$(s,\ X) \in$ *?f7*› **by** *simp*
thus ‹$(s,\ X) \in$ *?f1* $\cup$ *?f2* $\cup$ *?f3* $\cup$ *?f4* $\cup$ *?f5* $\cup$ *?f6* $\cup$ *?f7*› **by** *blast*
qed
next
fix *s X Y*
assume *assm* : ‹$(s,\ X) \in$ *?f1* $\cup$ *?f2* $\cup$ *?f3* $\cup$ *?f4* $\cup$ *?f5* $\cup$ *?f6* $\cup$ *?f7* $\wedge$
$(\forall c.\ c \in Y \longrightarrow (s\ @\ [c],\ \{\}) \notin$ *?f1* $\cup$ *?f2* $\cup$ *?f3* $\cup$ *?f4* $\cup$ *?f5* $\cup$ *?f6* $\cup$ *?f7*)›
have ‹$(s,\ X) \in$ *?f1* $\vee$ $(s,\ X) \in$ *?f2* $\vee$ $(s,\ X) \in$ *?f3* $\vee$ $(s,\ X) \in$ *?f4* $\vee$
$(s,\ X) \in$ *?f5* $\vee$ $(s,\ X) \in$ *?f6* $\vee$ $(s,\ X) \in$ *?f7*› **using** *assm*[*THEN conjunct1*]
by *fast*
thus ‹$(s,\ X \cup Y) \in$ *?f1* $\cup$ *?f2* $\cup$ *?f3* $\cup$ *?f4* $\cup$ *?f5* $\cup$ *?f6* $\cup$ *?f7*›
proof (*elim disjE*)
assume ‹$(s,\ X) \in$ *?f1*›
hence ‹$(s,\ X \cup Y) \in$ *?f1*› **by** *simp*
thus ‹$(s,\ X \cup Y) \in$ *?f1* $\cup$ *?f2* $\cup$ *?f3* $\cup$ *?f4* $\cup$ *?f5* $\cup$ *?f6* $\cup$ *?f7*› **by** *blast*
next
assume ‹$(s,\ X) \in$ *?f2*›
with *assm*[*THEN conjunct2*] **have** ‹$(s,\ X \cup Y) \in$ *?f2*› **by** *simp blast*
thus ‹$(s,\ X \cup Y) \in$ *?f1* $\cup$ *?f2* $\cup$ *?f3* $\cup$ *?f4* $\cup$ *?f5* $\cup$ *?f6* $\cup$ *?f7*› **by** *blast*
next
assume ‹$(s,\ X) \in$ *?f3*›

with *assm*[*THEN conjunct2*] **have** ‹(*s*, *X* ∪ *Y*) ∈ *?f3*›
by *simp* (*metis F-T T-F append-Nil is-processT5-S7′ list.distinct*(*1*))
thus ‹(*s*, *X* ∪ *Y*) ∈ *?f1* ∪ *?f2* ∪ *?f3* ∪ *?f4* ∪ *?f5* ∪ *?f6* ∪ *?f7*› **by** *blast*
next
assume ‹(*s*, *X*) ∈ *?f4*›
with *assm*[*THEN conjunct2*] **have** ‹(*s*, *X* ∪ *Y*) ∈ *?f4*›
by *simp* (*metis append.assoc append-is-Nil-conv is-processT5-S1*)
thus ‹(*s*, *X* ∪ *Y*) ∈ *?f1* ∪ *?f2* ∪ *?f3* ∪ *?f4* ∪ *?f5* ∪ *?f6* ∪ *?f7*› **by** *blast*
next
assume ‹(*s*, *X*) ∈ *?f5*›
with *assm*[*THEN conjunct2*] **have** ‹(*s*, *X* ∪ *Y*) ∈ *?f5*›
by *simp* (*metis Diff-empty Diff-insert0 T-F Un-Diff not-Cons-self*)
thus ‹(*s*, *X* ∪ *Y*) ∈ *?f1* ∪ *?f2* ∪ *?f3* ∪ *?f4* ∪ *?f5* ∪ *?f6* ∪ *?f7*› **by** *blast*
next
assume ‹(*s*, *X*) ∈ *?f6*›
hence ‹(*s*, *X* ∪ *Y*) ∈ *?f6*› **by** *simp*
thus ‹(*s*, *X* ∪ *Y*) ∈ *?f1* ∪ *?f2* ∪ *?f3* ∪ *?f4* ∪ *?f5* ∪ *?f6* ∪ *?f7*› **by** *blast*
next
assume ‹(*s*, *X*) ∈ *?f7*›
hence ‹(*s*, *X* ∪ *Y*) ∈ *?f7*› **by** *simp*
thus ‹(*s*, *X* ∪ *Y*) ∈ *?f1* ∪ *?f2* ∪ *?f3* ∪ *?f4* ∪ *?f5* ∪ *?f6* ∪ *?f7*› **by** *blast*
qed
next
fix *s X*
have ∗ : ‹(*s* @ [*tick*], {}) ∉ *?f2* ∪ *?f3* ∪ *?f5*›
by *simp* (*metis Cons-eq-appendI append-self-conv2 front-tickFree-mono*
is-processT2-TR list.distinct(*1*) *non-tickFree-tick*)
assume ‹(*s* @ [*tick*], {}) ∈ *?f1* ∪ *?f2* ∪ *?f3* ∪ *?f4* ∪ *?f5* ∪ *?f6* ∪ *?f7*›
with ∗ **have** ‹(*s* @ [*tick*], {}) ∈ *?f1* ∨ (*s* @ [*tick*], {}) ∈ *?f4* ∨
(*s* @ [*tick*], {}) ∈ *?f6* ∨ (*s* @ [*tick*], {}) ∈ *?f7*› **by** *fast*
thus ‹(*s*, *X* − {*tick*}) ∈ *?f1* ∪ *?f2* ∪ *?f3* ∪ *?f4* ∪ *?f5* ∪ *?f6* ∪ *?f7*›
proof (*elim disjE*)
assume ‹(*s* @ [*tick*], {}) ∈ *?f1*›
hence ‹(*s*, *X* − {*tick*}) ∈ *?f2*› **by** *blast*
thus ‹(*s*, *X* − {*tick*}) ∈ *?f1* ∪ *?f2* ∪ *?f3* ∪ *?f4* ∪ *?f5* ∪ *?f6* ∪ *?f7*› **by** *blast*
next
assume ‹(*s* @ [*tick*], {}) ∈ *?f4*›
then obtain *t1 t2*
where ∗∗ : ‹*s* = *t1* @ *t2*› ‹*t1* ∈ *T P*› ‹*tickFree t1*› ‹(*t2* @ [*tick*], {}) ∈ *F*
Q›
by *simp* (*metis butlast-append last-appendR snoc-eq-iff-butlast*)
hence ‹(*s*, *X* − {*tick*}) ∈ (*if t2* = [] *then ?f5 else ?f4*)›
by *simp* (*metis F-T process-charn self-append-conv2*)
thus ‹(*s*, *X* − {*tick*}) ∈ *?f1* ∪ *?f2* ∪ *?f3* ∪ *?f4* ∪ *?f5* ∪ *?f6* ∪ *?f7*› **by** (*meson*
UnCI)
next
assume ‹(*s* @ [*tick*], {}) ∈ *?f6*›
with *is-processT9* **have** ‹*s* ∈ *D P*› **by** *simp blast*
thus ‹(*s*, *X* − {*tick*}) ∈ *?f1* ∪ *?f2* ∪ *?f3* ∪ *?f4* ∪ *?f5* ∪ *?f6* ∪ *?f7*› **by** *blast*

```
    next
      assume ‹(s @ [tick], {}) ∈ ?f7›
      then obtain t1 t2 where ** : ‹s @ [tick] = t1 @ t2› ‹t1 ∈ 𝒯 P›
                                   ‹tickFree t1› ‹t2 ∈ 𝒟 Q› by blast
      from **(1, 3, 4) obtain t2' where ‹t2 = t2' @ [tick]› ‹t2' @ [tick] ∈ 𝒟 Q›
        by (cases t2 rule: rev-cases) auto
      with **(1) is-processT9 have ‹s = t1 @ t2' ∧ t2' ∈ 𝒟 Q› by simp blast
      with **(2, 3) have ‹(s, X − {tick}) ∈ ?f7› by simp blast
      thus ‹(s, X − {tick}) ∈ ?f1 ∪ ?f2 ∪ ?f3 ∪ ?f4 ∪ ?f5 ∪ ?f6 ∪ ?f7› by blast
    qed
  next
    show ‹s ∈ ?d1 ∪ ?d2 ∧ tickFree s ∧ front-tickFree t ⟹ s @ t ∈ ?d1 ∪ ?d2›
for s t
      apply (simp, elim conjE disjE exE)
      by (solves ‹simp add: is-processT7›)
         (meson append.assoc is-processT7 tickFree-append)
  next
    show ‹s ∈ ?d1 ∪ ?d2 ⟹ (s, X) ∈ ?f1 ∪ ?f2 ∪ ?f3 ∪ ?f4 ∪ ?f5 ∪ ?f6 ∪ ?f7›
for s X
      by blast
  next
    fix s
    assume ‹s @ [tick] ∈ ?d1 ∪ ?d2›
    then consider ‹s @ [tick] ∈ ?d1› | ‹s @ [tick] ∈ ?d2› by blast
    thus ‹s ∈ ?d1 ∪ ?d2›
    proof cases
      assume ‹s @ [tick] ∈ ?d1›
      with is-processT have ‹s ∈ ?d1› by blast
      thus ‹s ∈ ?d1 ∪ ?d2› by blast
    next
      assume ‹s @ [tick] ∈ ?d2›
      then obtain t1 t2 where ** : ‹s @ [tick] = t1 @ t2› ‹t1 ∈ 𝒯 P›
                                   ‹tickFree t1› ‹t2 ∈ 𝒟 Q› by blast
      from **(1, 3, 4) obtain t2' where ‹t2 = t2' @ [tick]› ‹t2' @ [tick] ∈ 𝒟 Q›
        by (cases t2 rule: rev-cases) auto
      with **(1) is-processT9 have ‹s = t1 @ t2' ∧ t2' ∈ 𝒟 Q› by simp blast
      with **(2, 3) have ‹s ∈ ?d2› by simp blast
      thus ‹s ∈ ?d1 ∪ ?d2› by blast
    qed
  qed
qed
```

2.3.2 Projections

```
lemma F-Interrupt :
  ‹ℱ (P △ Q) =
   {(t1 @ [tick], X)| t1 X. t1 @ [tick] ∈ 𝒯 P} ∪
   {(t1, X − {tick})| t1 X. t1 @ [tick] ∈ 𝒯 P} ∪
   {(t1, X) ∈ ℱ P. tickFree t1 ∧ ([], X) ∈ ℱ Q} ∪
```

$\{(t1 \mathbin{@} t2,\ X) |\ t1\ t2\ X.\ t1 \in \mathcal{T}\ P \wedge tickFree\ t1 \wedge (t2,\ X) \in \mathcal{F}\ Q \wedge t2 \neq []\} \cup$
$\{(t1,\ X - \{tick\}) |\ t1\ X.\ t1 \in \mathcal{T}\ P \wedge tickFree\ t1 \wedge [tick] \in \mathcal{T}\ Q\} \cup$
$\{(t1,\ X).\ t1 \in \mathcal{D}\ P\} \cup$
$\{(t1 \mathbin{@} t2,\ X) |\ t1\ t2\ X.\ t1 \in \mathcal{T}\ P \wedge tickFree\ t1 \wedge t2 \in \mathcal{D}\ Q\}\rangle$
by (*simp add*: *Failures-def FAILURES-def Interrupt.rep-eq*)

lemma *D-Interrupt* :
$\langle \mathcal{D}\ (P \mathbin{\triangle} Q) = \mathcal{D}\ P \cup \{t1 \mathbin{@} t2 |\ t1\ t2.\ t1 \in \mathcal{T}\ P \wedge tickFree\ t1 \wedge t2 \in \mathcal{D}\ Q\}\rangle$
by (*simp add*: *Divergences-def DIVERGENCES-def Interrupt.rep-eq*)

lemma *T-Interrupt* :
$\langle \mathcal{T}\ (P \mathbin{\triangle} Q) = \mathcal{T}\ P \cup \{t1 \mathbin{@} t2 |\ t1\ t2.\ t1 \in \mathcal{T}\ P \wedge tickFree\ t1 \wedge t2 \in \mathcal{T}\ Q\}\rangle$
apply (*simp add*: *Traces-def TRACES-def Failures-def*[*symmetric*] *F-Interrupt*)

apply (*safe*, *simp-all add*: *is-processT8*)
subgoal by (*metis is-processT3-SR*)
subgoal by *auto*
subgoal by *auto*
subgoal by (*metis is-processT8-S*)
subgoal by (*metis is-processT4-empty nonTickFree-n-frontTickFree process-charn*)
by (*metis append.right-neutral is-processT4-empty tickFree-Nil*)

2.3.3 Monotony

lemma *mono-Interrupt*[*simp*]: $\langle P \mathbin{\triangle} Q \sqsubseteq P' \mathbin{\triangle} Q'\rangle$ **if** $\langle P \sqsubseteq P'\rangle$ **and** $\langle Q \sqsubseteq Q'\rangle$
proof (*unfold le-approx-def*, *intro conjI allI impI subsetI*)
show $\langle s \in \mathcal{D}\ (P' \mathbin{\triangle} Q') \Longrightarrow s \in \mathcal{D}\ (P \mathbin{\triangle} Q)\rangle$ **for** *s*
using *that*[*THEN le-approx1*] *D-T that*(*1*)[*THEN le-approx2T*]
by (*simp add*: *D-Interrupt*) *blast*
next
show $\langle s \notin \mathcal{D}\ (P \mathbin{\triangle} Q) \Longrightarrow \mathcal{R}_a\ (P \mathbin{\triangle} Q)\ s = \mathcal{R}_a\ (P' \mathbin{\triangle} Q')\ s\rangle$ **for** *s*
apply (*simp add*: *D-Interrupt Ra-def F-Interrupt*,
intro subset-antisym subsetI; *simp*, *elim disjE*)
subgoal by (*metis le-approx2T that*(*1*))
subgoal by (*metis is-processT9 le-approx2T that*(*1*))
subgoal by (*metis F-T append.right-neutral le-approx2 that*)
subgoal by (*metis is-processT2 is-processT7 le-approx2T proc-ord2a that*)
subgoal by (*metis* (*no-types*, *lifting*) *append-Nil2 le-approx2T min-elems6*
no-Trace-implies-no-Failure self-append-conv2 that)
subgoal by *metis*
subgoal by (*metis le-approx2T that*(*1*))
subgoal by (*metis le-approx-lemma-T subset-eq that*(*1*))
subgoal by (*metis is-processT8-S le-approx2 that*)
subgoal by (*metis is-processT2 is-processT7-S le-approx2 le-approx2T that*)
subgoal by (*metis D-T le-approx2T that*)
subgoal by (*metis in-mono le-approx1 that*(*1*))
by (*metis le-approx1 le-approx2T process-charn subsetD that*)

next
from *that*[*THEN le-approx3*]
show ‹$s \in$ *min-elems* ($\mathcal{D}$ ($P \triangle Q$)) $\Longrightarrow$ $s \in \mathcal{T}$ ($P' \triangle Q'$)› **for** s
by (*auto simp add*: *min-elems-def D-Interrupt T-Interrupt subset-iff*)
(*metis le-approx2T le-list-def less-append order-le-imp-less-or-eq that*(*1*))
qed

lemma *mono-Interrupt-T*: ‹$P \sqsubseteq_T P' \Longrightarrow Q \sqsubseteq_T Q' \Longrightarrow P \triangle Q \sqsubseteq_T P' \triangle Q'$›
unfolding *trace-refine-def* **by** (*auto simp add*: *T-Interrupt*)

lemma *mono-right-Interrupt-D*: ‹$Q \sqsubseteq_D Q' \Longrightarrow P \triangle Q \sqsubseteq_D P \triangle Q'$›
unfolding *divergence-refine-def* **by** (*auto simp add*: *D-Interrupt*)

— We have no monotony, even partial, with ($\sqsubseteq_F$).

lemma *mono-Interrupt-FD*:
‹$P \sqsubseteq_{FD} P' \Longrightarrow Q \sqsubseteq_{FD} Q' \Longrightarrow P \triangle Q \sqsubseteq_{FD} P' \triangle Q'$›
unfolding *failure-divergence-refine-def le-ref-def*
by (*simp add*: *D-Interrupt F-Interrupt*, *safe*;
metis [[*metis-verbose* = *false*]] *F-subset-imp-T-subset subsetD*)

lemma *mono-Interrupt-DT*:
‹$P \sqsubseteq_{DT} P' \Longrightarrow Q \sqsubseteq_{DT} Q' \Longrightarrow P \triangle Q \sqsubseteq_{DT} P' \triangle Q'$›
unfolding *trace-divergence-refine-def trace-refine-def divergence-refine-def*
by (*auto simp add*: *T-Interrupt D-Interrupt subset-iff*)

2.3.4 Properties

lemma *Interrupt-STOP-neutral* : ‹$P \triangle STOP = P$› ‹$STOP \triangle P = P$›
apply (*all* ‹*subst Process-eq-spec*›, *safe*)
apply (*simp-all add*: *Process-eq-spec F-Interrupt D-Interrupt F-STOP T-STOP*
D-STOP T-F is-processT6 is-processT8-S tick-T-F)
subgoal by (*meson process-charn tick-T-F*)
subgoal by (*metis F-T T-nonTickFree-imp-decomp*)
subgoal by (*meson DiffE insertI1 is-processT6-S2 is-processT8-S*)
by *blast*

lemma *Interrupt-BOT-absorb* : ‹$P \triangle \bot = \bot$› ‹$\bot \triangle P = \bot$›
by (*simp-all add*: *BOT-iff-D D-Interrupt D-UU Nil-elem-T*)

lemma *Interrupt-is-BOT-iff* : ‹$P \triangle Q = \bot \longleftrightarrow P = \bot \vee Q = \bot$›
by (*simp add*: *BOT-iff-D D-Interrupt Nil-elem-T*)

lemma *events-Interrupt*: ‹*events-of* ($P \triangle Q$) = *events-of* $P \cup$ *events-of* Q›
apply (*intro subset-antisym subsetI*; *simp add*: *events-of-def T-Interrupt*)
by (*metis UnE set-append*) (*metis Nil-elem-T append-Nil tickFree-Nil*)

lemma *Interrupt-Ndet-distrib* : ‹$P \triangle Q1 \sqcap Q2 = (P \triangle Q1) \sqcap (P \triangle Q2)$›
(**is** ‹*?lhs* = *?rhs*›)
proof (*subst Process-eq-spec-optimized*, *safe*)
show ‹$s \in \mathcal{D}$ *?lhs* $\Longrightarrow s \in \mathcal{D}$ *?rhs*› **for** *s*
by (*auto simp add*: *D-Interrupt D-Ndet*)
next
show ‹$s \in \mathcal{D}$ *?rhs* $\Longrightarrow s \in \mathcal{D}$ *?lhs*› **for** *s*
by (*auto simp add*: *D-Interrupt D-Ndet*)
next
fix *s X*
assume *same-div* : ‹$\mathcal{D}$ *?lhs* = $\mathcal{D}$ *?rhs*›
assume ‹$(s, X) \in \mathcal{F}$ *?lhs*›
then consider ‹$s \in \mathcal{D}$ $(P \triangle Q1 \sqcap Q2)$›
| ‹$\exists t1.\ s = t1 \ @\ [tick] \wedge t1 \ @\ [tick] \in \mathcal{T}\ P$›
| ‹$s \ @\ [tick] \in \mathcal{T}\ P \wedge tick \notin X$›
| ‹$(s, X) \in \mathcal{F}\ P \wedge tickFree\ s \wedge ([], X) \in \mathcal{F}\ (Q1 \sqcap Q2)$›
| ‹$\exists t1\ t2.\ s = t1 \ @\ t2 \wedge t1 \in \mathcal{T}\ P \wedge tickFree\ t1 \wedge (t2, X) \in \mathcal{F}\ (Q1 \sqcap Q2) \wedge t2 \neq []$›
| ‹$s \in \mathcal{T}\ P \wedge tickFree\ s \wedge [tick] \in \mathcal{T}\ (Q1 \sqcap Q2) \wedge tick \notin X$›
by (*simp add*: *F-Interrupt D-Interrupt*) *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
show ‹$\exists t1.\ s = t1 \ @\ [tick] \wedge t1 \ @\ [tick] \in \mathcal{T}\ P \Longrightarrow (s, X) \in \mathcal{F}$ *?rhs*›
by (*simp add*: *F-Ndet F-Interrupt*)
next
show ‹$s \ @\ [tick] \in \mathcal{T}\ P \wedge tick \notin X \Longrightarrow (s, X) \in \mathcal{F}$ *?rhs*›
by (*simp add*: *F-Ndet F-Interrupt*) (*metis Diff-insert-absorb*)
next
show ‹$(s, X) \in \mathcal{F}\ P \wedge tickFree\ s \wedge ([], X) \in \mathcal{F}\ (Q1 \sqcap Q2) \Longrightarrow (s, X) \in \mathcal{F}$ *?rhs*›
by (*simp add*: *F-Ndet F-Interrupt*) *blast*
next
show ‹$\exists t1\ t2.\ s = t1 \ @\ t2 \wedge t1 \in \mathcal{T}\ P \wedge tickFree\ t1 \wedge (t2, X) \in \mathcal{F}\ (Q1 \sqcap Q2) \wedge t2 \neq [] \Longrightarrow (s, X) \in \mathcal{F}$ *?rhs*›
by (*simp add*: *F-Ndet F-Interrupt*) *metis*
next
show ‹$s \in \mathcal{T}\ P \wedge tickFree\ s \wedge [tick] \in \mathcal{T}\ (Q1 \sqcap Q2) \wedge tick \notin X \Longrightarrow (s, X) \in \mathcal{F}$ *?rhs*›
by (*simp add*: *F-Interrupt F-Ndet T-Ndet*) (*metis Diff-insert-absorb*)
qed
next
have ‹$(s, X) \in \mathcal{F}\ (P \triangle Q1) \Longrightarrow (s, X) \in \mathcal{F}\ (P \triangle Q1 \sqcap Q2)$› **for** *s X Q1 Q2*
by (*simp add*: *F-Interrupt F-Ndet D-Ndet T-Ndet*) *blast*
thus ‹$(s, X) \in \mathcal{F}$ *?rhs* $\Longrightarrow (s, X) \in \mathcal{F}$ *?lhs*› **for** *s X*
by (*simp add*: *F-Ndet*) (*metis Ndet-commute*)

qed

lemma *Ndet-Interrupt-distrib* : ‹*P1* ⊓ *P2* △ *Q* = (*P1* △ *Q*) ⊓ (*P2* △ *Q*)›
(**is** ‹*?lhs* = *?rhs*›)
proof (*subst Process-eq-spec-optimized*, *safe*)
show ‹*s* ∈ 𝒟 *?lhs* ⟹ *s* ∈ 𝒟 *?rhs*› **for** *s*
by (*auto simp add*: *D-Ndet T-Ndet D-Interrupt*)
next
show ‹*s* ∈ 𝒟 *?rhs* ⟹ *s* ∈ 𝒟 *?lhs*› **for** *s*
by (*auto simp add*: *D-Ndet T-Ndet D-Interrupt*)
next
fix *s X*
assume *same-div* : ‹𝒟 *?lhs* = 𝒟 *?rhs*›
assume ‹(*s*, *X*) ∈ ℱ *?lhs*›
then consider ‹*s* ∈ 𝒟 *?lhs*›
| ‹∃ *t1*. *s* = *t1* @ [*tick*] ∧ *t1* @ [*tick*] ∈ 𝒯 (*P1* ⊓ *P2*)›
| ‹*s* @ [*tick*] ∈ 𝒯 (*P1* ⊓ *P2*) ∧ *tick* ∉ *X*›
| ‹(*s*, *X*) ∈ ℱ (*P1* ⊓ *P2*) ∧ *tickFree s* ∧ ([], *X*) ∈ ℱ *Q*›
| ‹∃ *t1 t2*. *s* = *t1* @ *t2* ∧ *t1* ∈ 𝒯 (*P1* ⊓ *P2*) ∧ *tickFree t1* ∧ (*t2*, *X*) ∈ ℱ *Q* ∧ *t2* ≠ []›
| ‹*s* ∈ 𝒯 (*P1* ⊓ *P2*) ∧ *tickFree s* ∧ [*tick*] ∈ 𝒯 *Q* ∧ *tick* ∉ *X*›
by (*simp add*: *F-Interrupt D-Interrupt*) *blast*
thus ‹(*s*, *X*) ∈ ℱ *?rhs*›
proof *cases*
from *same-div D-F* **show** ‹*s* ∈ 𝒟 *?lhs* ⟹ (*s*, *X*) ∈ ℱ *?rhs*› **by** *blast*
next
show ‹∃ *t1*. *s* = *t1* @ [*tick*] ∧ *t1* @ [*tick*] ∈ 𝒯 (*P1* ⊓ *P2*) ⟹ (*s*, *X*) ∈ ℱ *?rhs*›
by (*simp add*: *T-Ndet F-Ndet F-Interrupt*) *metis*
next
show ‹*s* @ [*tick*] ∈ 𝒯 (*P1* ⊓ *P2*) ∧ *tick* ∉ *X* ⟹ (*s*, *X*) ∈ ℱ *?rhs*›
by (*simp add*: *T-Ndet F-Ndet F-Interrupt*) (*metis Diff-insert-absorb*)
next
show ‹(*s*, *X*) ∈ ℱ (*P1* ⊓ *P2*) ∧ *tickFree s* ∧ ([], *X*) ∈ ℱ *Q* ⟹ (*s*, *X*) ∈ ℱ *?rhs*›
by (*simp add*: *F-Ndet F-Interrupt*) *blast*
next
show ‹∃ *t1 t2*. *s* = *t1* @ *t2* ∧ *t1* ∈ 𝒯 (*P1* ⊓ *P2*) ∧ *tickFree t1* ∧
(*t2*, *X*) ∈ ℱ *Q* ∧ *t2* ≠ [] ⟹ (*s*, *X*) ∈ ℱ *?rhs*›
by (*simp add*: *T-Ndet F-Ndet F-Interrupt*) *metis*
next
show ‹*s* ∈ 𝒯 (*P1* ⊓ *P2*) ∧ *tickFree s* ∧ [*tick*] ∈ 𝒯 *Q* ∧ *tick* ∉ *X* ⟹ (*s*, *X*) ∈ ℱ *?rhs*›
by (*simp add*: *T-Ndet F-Ndet F-Interrupt*) (*metis Diff-insert-absorb*)
qed
next
have ‹(*s*, *X*) ∈ ℱ (*P1* △ *Q*) ⟹ (*s*, *X*) ∈ ℱ (*P1* ⊓ *P2* △ *Q*)› **for** *s X P1 P2*
by (*simp add*: *F-Interrupt F-Ndet D-Ndet T-Ndet*) *blast*
thus ‹(*s*, *X*) ∈ ℱ *?rhs* ⟹ (*s*, *X*) ∈ ℱ *?lhs*› **for** *s X*

by (*simp add*: *F-Ndet*) (*metis Ndet-commute*)
qed

lemma *Interrupt-assoc*: ‹$P \triangle (Q \triangle R) = P \triangle Q \triangle R$› (**is** ‹*?lhs* = *?rhs*›)
proof −
have ‹*?lhs* = *?rhs*› **if** *non-BOT* : ‹$P \neq \bot$› ‹$Q \neq \bot$› ‹$R \neq \bot$›
proof (*subst Process-eq-spec-optimized*, *safe*)
fix *s*
assume ‹$s \in \mathcal{D}$ *?lhs*›
then consider ‹$s \in \mathcal{D}$ *P*›
| ‹$\exists$ *t1 t2*. $s = t1 \mathbin{@} t2 \wedge t1 \in \mathcal{T}$ *P* $\wedge$ *tickFree t1* $\wedge$ $t2 \in \mathcal{D}$ $(Q \triangle R)$›
by (*simp add*: *D-Interrupt*) *blast*
thus ‹$s \in \mathcal{D}$ *?rhs*›
proof *cases*
show ‹$s \in \mathcal{D}$ $P \Longrightarrow s \in \mathcal{D}$ *?rhs*› **by** (*simp add*: *D-Interrupt*)
next
assume ‹$\exists$ *t1 t2*. $s = t1 \mathbin{@} t2 \wedge t1 \in \mathcal{T}$ *P* $\wedge$ *tickFree t1* $\wedge$ $t2 \in \mathcal{D}$ $(Q \triangle R)$›
then obtain *t1 t2* **where** ∗ : ‹$s = t1 \mathbin{@} t2$› ‹$t1 \in \mathcal{T}$ *P*›
‹*tickFree t1*› ‹$t2 \in \mathcal{D}$ $(Q \triangle R)$› **by** *blast*
from ∗(*4*) **consider** ‹$t2 \in \mathcal{D}$ *Q*›
| ‹$\exists$ *u1 u2*. $t2 = u1 \mathbin{@} u2 \wedge u1 \in \mathcal{T}$ *Q* $\wedge$ *tickFree u1* $\wedge$ $u2 \in \mathcal{D}$ *R*›
by (*simp add*: *D-Interrupt*) *blast*
thus ‹$s \in \mathcal{D}$ *?rhs*›
proof *cases*
from ∗(*1*, *2*, *3*) **show** ‹$t2 \in \mathcal{D}$ $Q \Longrightarrow s \in \mathcal{D}$ *?rhs*› **by** (*simp add*: *D-Interrupt*) *blast*
next
show ‹$\exists$ *u1 u2*. $t2 = u1 \mathbin{@} u2 \wedge u1 \in \mathcal{T}$ *Q* $\wedge$ *tickFree u1* $\wedge$ $u2 \in \mathcal{D}$ $R \Longrightarrow s \in \mathcal{D}$ *?rhs*›
by (*simp add*: ∗(*1*) *D-Interrupt T-Interrupt*)
(*metis* ∗(*2*, *3*) *append-assoc tickFree-append*)
qed
qed
next
fix *s*
assume ‹$s \in \mathcal{D}$ *?rhs*›
then consider ‹$s \in \mathcal{D}$ $(P \triangle Q)$›
| ‹$\exists$ *t1 t2*. $s = t1 \mathbin{@} t2 \wedge t1 \in \mathcal{T}$ $(P \triangle Q) \wedge$ *tickFree t1* $\wedge$ $t2 \in \mathcal{D}$ *R*›
by (*simp add*: *D-Interrupt*) *blast*
thus ‹$s \in \mathcal{D}$ *?lhs*›
proof *cases*
show ‹$s \in \mathcal{D}$ $(P \triangle Q) \Longrightarrow s \in \mathcal{D}$ *?lhs*› **by** (*simp add*: *D-Interrupt*) *blast*
next
assume ‹$\exists$ *t1 t2*. $s = t1 \mathbin{@} t2 \wedge t1 \in \mathcal{T}$ $(P \triangle Q) \wedge$ *tickFree t1* $\wedge$ $t2 \in \mathcal{D}$ *R*›
then obtain *t1 t2* **where** ∗ : ‹$s = t1 \mathbin{@} t2$› ‹$t1 \in \mathcal{T}$ $(P \triangle Q)$›
‹*tickFree t1*› ‹$t2 \in \mathcal{D}$ *R*› **by** *blast*
from ∗(*2*) **consider** ‹$t1 \in \mathcal{T}$ *P*›
| ‹$\exists$ *u1 u2*. $t1 = u1 \mathbin{@} u2 \wedge u1 \in \mathcal{T}$ *P* $\wedge$ *tickFree u1* $\wedge$ $u2 \in \mathcal{T}$ *Q*›

```
      by (simp add: T-Interrupt) blast
    thus ‹s ∈ 𝒟 ?lhs›
    proof cases
      show ‹t1 ∈ 𝒯 P ⟹ s ∈ 𝒟 ?lhs›
        by (simp add: D-Interrupt ∗(1))
          (metis ∗(3, 4) Nil-elem-T append-Nil tickFree-Nil)
    next
      show ‹∃ u1 u2. t1 = u1 @ u2 ∧ u1 ∈ 𝒯 P ∧ tickFree u1 ∧ u2 ∈ 𝒯 Q ⟹
s ∈ 𝒟 ?lhs›
        by (simp add: D-Interrupt ∗(1))
          (metis ∗(3, 4) append.assoc tickFree-append)
    qed
  qed
next
  fix s X
  assume same-div: ‹𝒟 ?lhs = 𝒟 ?rhs›
  assume ‹(s, X) ∈ ℱ ?lhs›
  then consider ‹s ∈ 𝒟 ?lhs›
    | ‹∃ t1. s = t1 @ [tick] ∧ t1 @ [tick] ∈ 𝒯 P›
    | ‹s @ [tick] ∈ 𝒯 P ∧ tick ∉ X›
    | ‹(s, X) ∈ ℱ P ∧ tickFree s ∧ ([], X) ∈ ℱ (Q △ R)›
    | ‹∃ t1 t2. s = t1 @ t2 ∧ t1 ∈ 𝒯 P ∧ tickFree t1 ∧ (t2, X) ∈ ℱ (Q △ R) ∧
t2 ≠ []›
    | ‹s ∈ 𝒯 P ∧ tickFree s ∧ [tick] ∈ 𝒯 (Q △ R) ∧ tick ∉ X›
    by (subst (asm) F-Interrupt, simp add: D-Interrupt) blast
  thus ‹(s, X) ∈ ℱ ?rhs›
  proof cases
    from same-div D-F show ‹s ∈ 𝒟 ?lhs ⟹ (s, X) ∈ ℱ ?rhs› by blast
  next
    show ‹∃ t1. s = t1 @ [tick] ∧ t1 @ [tick] ∈ 𝒯 P ⟹ (s, X) ∈ ℱ ?rhs›
      by (auto simp add: F-Interrupt T-Interrupt)
  next
    show ‹s @ [tick] ∈ 𝒯 P ∧ tick ∉ X ⟹ (s, X) ∈ ℱ ?rhs›
      by (simp add: F-Interrupt T-Interrupt) (metis Diff-insert-absorb)
  next
    assume assm : ‹(s, X) ∈ ℱ P ∧ tickFree s ∧ ([], X) ∈ ℱ (Q △ R)›
    with non-BOT(2, 3) consider ‹[tick] ∈ 𝒯 Q ∧ tick ∉ X›
      | ‹([], X) ∈ ℱ Q ∧ ([], X) ∈ ℱ R›
      | ‹[] ∈ 𝒯 Q ∧ [tick] ∈ 𝒯 R ∧ tick ∉ X›
      by (simp add: F-Interrupt Nil-elem-T BOT-iff-D) blast
    thus ‹(s, X) ∈ ℱ ?rhs›
    proof cases
      show ‹[tick] ∈ 𝒯 Q ∧ tick ∉ X ⟹ (s, X) ∈ ℱ ?rhs›
      by (simp add: F-Interrupt T-Interrupt) (metis Diff-insert-absorb F-T assm)
    next
      show ‹([], X) ∈ ℱ Q ∧ ([], X) ∈ ℱ R ⟹ (s, X) ∈ ℱ ?rhs›
        by (simp add: F-Interrupt assm)
    next
      show ‹[] ∈ 𝒯 Q ∧ [tick] ∈ 𝒯 R ∧ tick ∉ X ⟹ (s, X) ∈ ℱ ?rhs›
```

by (*simp add*: *F-Interrupt T-Interrupt*) (*metis Diff-insert-absorb F-T assm*)
qed
next
assume ‹∃ *t1 t2*. *s* = *t1* @ *t2* ∧ *t1* ∈ $\mathcal{T}$ *P* ∧ *tickFree t1* ∧
(*t2*, *X*) ∈ $\mathcal{F}$ (*Q* △ *R*) ∧ *t2* ≠ []›
then obtain *t1 t2* **where** ∗ : ‹*s* = *t1* @ *t2*› ‹*t1* ∈ $\mathcal{T}$ *P*› ‹*tickFree t1*›
‹(*t2*, *X*) ∈ $\mathcal{F}$ (*Q* △ *R*)› ‹*t2* ≠ []› **by** *blast*
from ∗(*4*) **consider** ‹*t2* ∈ $\mathcal{D}$ (*Q* △ *R*)›
| ‹∃ *u1*. *t2* = *u1* @ [*tick*] ∧ *u1* @ [*tick*] ∈ $\mathcal{T}$ *Q*›
| ‹*t2* @ [*tick*] ∈ $\mathcal{T}$ *Q* ∧ *tick* ∉ *X*›
| ‹(*t2*, *X*) ∈ $\mathcal{F}$ *Q* ∧ *tickFree t2* ∧ ([], *X*) ∈ $\mathcal{F}$ *R*›
| ‹∃ *u1 u2*. *t2* = *u1* @ *u2* ∧ *u1* ∈ $\mathcal{T}$ *Q* ∧ *tickFree u1* ∧ (*u2*, *X*) ∈ $\mathcal{F}$ *R* ∧ *u2* ≠ []›
| ‹*t2* ∈ $\mathcal{T}$ *Q* ∧ *tickFree t2* ∧ [*tick*] ∈ $\mathcal{T}$ *R* ∧ *tick* ∉ *X*›
by (*simp add*: *F-Interrupt D-Interrupt*) *blast*
thus ‹(*s*, *X*) ∈ $\mathcal{F}$ *?rhs*›
proof *cases*
assume ‹*t2* ∈ $\mathcal{D}$ (*Q* △ *R*)›
with ∗(*1*, *2*, *3*) **have** ‹*s* ∈ $\mathcal{D}$ *?lhs*› **by** (*simp add*: *D-Interrupt*) *blast*
with *same-div D-F* **show** ‹(*s*, *X*) ∈ $\mathcal{F}$ *?rhs*› **by** *blast*
next
from ∗(*1*, *2*, *3*) **show** ‹∃ *u1*. *t2* = *u1* @ [*tick*] ∧ *u1* @ [*tick*] ∈ $\mathcal{T}$ *Q* ⟹ (*s*, *X*) ∈ $\mathcal{F}$ *?rhs*›
by (*simp add*: *F-Interrupt T-Interrupt*) (*metis append-assoc*)
next
from ∗(*1*, *2*, *3*) **show** ‹*t2* @ [*tick*] ∈ $\mathcal{T}$ *Q* ∧ *tick* ∉ *X* ⟹ (*s*, *X*) ∈ $\mathcal{F}$ *?rhs*›
by (*simp add*: *F-Interrupt T-Interrupt*) (*metis Diff-insert-absorb*)
next
from ∗(*1*) **show** ‹(*t2*, *X*) ∈ $\mathcal{F}$ *Q* ∧ *tickFree t2* ∧ ([], *X*) ∈ $\mathcal{F}$ *R* ⟹ (*s*, *X*) ∈ $\mathcal{F}$ *?rhs*›
by (*simp add*: *F-Interrupt T-Interrupt*) (*metis* ∗(*2*, *3*, *5*))
next
from ∗(*1*, *2*, *3*) **show** ‹∃ *u1 u2*. *t2* = *u1* @ *u2* ∧ *u1* ∈ $\mathcal{T}$ *Q* ∧ *tickFree u1* ∧
(*u2*, *X*) ∈ $\mathcal{F}$ *R* ∧ *u2* ≠ [] ⟹ (*s*, *X*) ∈ $\mathcal{F}$ *?rhs*›
by (*simp add*: *F-Interrupt T-Interrupt*)
(*metis* (*mono-tags*, *lifting*) *append-assoc tickFree-append*)
next
from ∗(*1*, *2*, *3*) **show** ‹*t2* ∈ $\mathcal{T}$ *Q* ∧ *tickFree t2* ∧ [*tick*] ∈ $\mathcal{T}$ *R* ∧
tick ∉ *X* ⟹ (*s*, *X*) ∈ $\mathcal{F}$ *?rhs*›
by (*simp add*: *F-Interrupt T-Interrupt*) (*metis Diff-insert-absorb*)
qed
next
show ‹*s* ∈ $\mathcal{T}$ *P* ∧ *tickFree s* ∧ [*tick*] ∈ $\mathcal{T}$ (*Q* △ *R*) ∧ *tick* ∉ *X* ⟹ (*s*, *X*) ∈ $\mathcal{F}$ *?rhs*›
by (*simp add*: *F-Interrupt T-Interrupt*)
(*metis Diff-insert-absorb append-Nil hd-append2 hd-in-set list.sel*(*1*) *tickFree-def*)
qed

```
  next
    fix s X
    assume same-div : ‹𝒟 ?lhs = 𝒟 ?rhs›
    assume ‹(s, X) ∈ ℱ ?rhs›
    then consider ‹s ∈ 𝒟 ?rhs›
      | ‹∃ t1. s = t1 @ [tick] ∧ t1 @ [tick] ∈ 𝒯 (P △ Q)›
      | ‹s @ [tick] ∈ 𝒯 (P △ Q) ∧ tick ∉ X›
      | ‹(s, X) ∈ ℱ (P △ Q) ∧ tickFree s ∧ ([], X) ∈ ℱ R›
      | ‹∃ t1 t2. s = t1 @ t2 ∧ t1 ∈ 𝒯 (P △ Q) ∧ tickFree t1 ∧ (t2, X) ∈ ℱ R ∧
t2 ≠ []›
      | ‹s ∈ 𝒯 (P △ Q) ∧ tickFree s ∧ [tick] ∈ 𝒯 R ∧ tick ∉ X›
      by (subst (asm) F-Interrupt, simp add: D-Interrupt) 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 ‹∃ t1. s = t1 @ [tick] ∧ t1 @ [tick] ∈ 𝒯 (P △ Q) ⟹ (s, X) ∈ ℱ ?lhs›
       by (simp add: F-Interrupt T-Interrupt)
          (metis last-append self-append-conv snoc-eq-iff-butlast)
    next
      assume assm : ‹s @ [tick] ∈ 𝒯 (P △ Q) ∧ tick ∉ X›
      from assm[THEN conjunct1] consider ‹s @ [tick] ∈ 𝒯 P›
        | ‹∃ t1 t2. s @ [tick] = t1 @ t2 ∧ t1 ∈ 𝒯 P ∧ tickFree t1 ∧ t2 ∈ 𝒯 Q›
        by (simp add: T-Interrupt) blast
      thus ‹(s, X) ∈ ℱ ?lhs›
      proof cases
        show ‹s @ [tick] ∈ 𝒯 P ⟹ (s, X) ∈ ℱ ?lhs›
          by (simp add: F-Interrupt) (metis Diff-insert-absorb assm[THEN con-
junct2])
      next
        show ‹∃ t1 t2. s @ [tick] = t1 @ t2 ∧ t1 ∈ 𝒯 P ∧
                    tickFree t1 ∧ t2 ∈ 𝒯 Q ⟹ (s, X) ∈ ℱ ?lhs›
          by (simp add: F-Interrupt T-Interrupt)
            (metis Diff-insert-absorb T-nonTickFree-imp-decomp
                  append.right-neutral append-Nil assm[THEN conjunct2]
                  butlast.simps(2) butlast-append non-tickFree-tick tickFree-append)
      qed
    next
      assume assm : ‹(s, X) ∈ ℱ (P △ Q) ∧ tickFree s ∧ ([], X) ∈ ℱ R›
      from assm[THEN conjunct1] consider ‹s ∈ 𝒟 (P △ Q)›
        | ‹∃ t1. s = t1 @ [tick] ∧ t1 @ [tick] ∈ 𝒯 P›
        | ‹s @ [tick] ∈ 𝒯 P ∧ tick ∉ X›
        | ‹(s, X) ∈ ℱ P ∧ tickFree s ∧ ([], X) ∈ ℱ Q›
        | ‹∃ t1 t2. s = t1 @ t2 ∧ t1 ∈ 𝒯 P ∧ tickFree t1 ∧ (t2, X) ∈ ℱ Q ∧ t2 ≠
[]›
        | ‹s ∈ 𝒯 P ∧ tickFree s ∧ [tick] ∈ 𝒯 Q ∧ tick ∉ X›
        by (simp add: F-Interrupt D-Interrupt) blast
      thus ‹(s, X) ∈ ℱ ?lhs›
      proof cases
```

assume ‹$s \in \mathcal{D}\ (P \triangle Q)$›
hence ‹$s \in \mathcal{D}$ *?rhs*› **by** (*simp add*: *D-Interrupt*)
with *same-div D-F* **show** ‹$(s, X) \in \mathcal{F}$ *?lhs*› **by** *blast*
next
show ‹$\exists$ *t1*. $s = t1 \mathbin{@} [tick] \wedge t1 \mathbin{@} [tick] \in \mathcal{T}\ P \Longrightarrow (s, X) \in \mathcal{F}$ *?lhs*›
by (*simp add*: *F-Interrupt*)
next
show ‹$s \mathbin{@} [tick] \in \mathcal{T}\ P \wedge tick \notin X \Longrightarrow (s, X) \in \mathcal{F}$ *?lhs*›
by (*simp add*: *F-Interrupt*) (*metis Diff-insert-absorb*)
next
show ‹$(s, X) \in \mathcal{F}\ P \wedge$ *tickFree s* $\wedge ([], X) \in \mathcal{F}\ Q \Longrightarrow (s, X) \in \mathcal{F}$ *?lhs*›
by (*simp add*: *F-Interrupt assm*[*THEN conjunct2*])
next
show ‹$\exists$ *t1 t2*. $s = t1 \mathbin{@} t2 \wedge t1 \in \mathcal{T}\ P \wedge$ *tickFree t1* $\wedge$
$(t2, X) \in \mathcal{F}\ Q \wedge t2 \neq [] \Longrightarrow (s, X) \in \mathcal{F}$ *?lhs*›
by (*simp add*: *F-Interrupt*) (*metis assm*[*THEN conjunct2*] *tickFree-append*)
next
show ‹$s \in \mathcal{T}\ P \wedge$ *tickFree s* $\wedge [tick] \in \mathcal{T}\ Q \wedge tick \notin X \Longrightarrow (s, X) \in \mathcal{F}$ *?lhs*›
by (*simp add*: *F-Interrupt T-Interrupt*) (*metis Diff-insert-absorb*)
qed
next
assume ‹$\exists$ *t1 t2*. $s = t1 \mathbin{@} t2 \wedge t1 \in \mathcal{T}\ (P \triangle Q) \wedge$
tickFree t1 $\wedge\ (t2, X) \in \mathcal{F}\ R \wedge t2 \neq []$›
then obtain *t1 t2* **where** $*$: ‹$s = t1 \mathbin{@} t2$› ‹$t1 \in \mathcal{T}\ (P \triangle Q)$›
‹*tickFree t1*› ‹$(t2, X) \in \mathcal{F}\ R$› ‹$t2 \neq []$› **by** *blast*
from $*(2)$ **consider** ‹$t1 \in \mathcal{T}\ P$›
| ‹$\exists$ *u1 u2*. $t1 = u1 \mathbin{@} u2 \wedge u1 \in \mathcal{T}\ P \wedge$ *tickFree u1* $\wedge\ u2 \in \mathcal{T}\ Q$›
by (*simp add*: *T-Interrupt*) *blast*
thus ‹$(s, X) \in \mathcal{F}$ *?lhs*›
proof *cases*
from $*(1, 3, 4, 5)$ **show** ‹$t1 \in \mathcal{T}\ P \Longrightarrow (s, X) \in \mathcal{F}$ *?lhs*›
by (*simp add*: *F-Interrupt T-Interrupt*)
(*metis Nil-elem-T append-Nil tickFree-Nil*)
next
from $*(1, 3, 4, 5)$ **show** ‹$\exists$ *u1 u2*. $t1 = u1 \mathbin{@} u2 \wedge u1 \in \mathcal{T}\ P \wedge$
tickFree u1 $\wedge\ u2 \in \mathcal{T}\ Q \Longrightarrow (s, X) \in \mathcal{F}$ *?lhs*›
by (*elim exE*, *simp add*: *F-Interrupt*) (*metis append-is-Nil-conv*)
qed
next
show ‹$s \in \mathcal{T}\ (P \triangle Q) \wedge$ *tickFree s* $\wedge [tick] \in \mathcal{T}\ R \wedge tick \notin X \Longrightarrow (s, X) \in \mathcal{F}$ *?lhs*›
by (*simp add*: *F-Interrupt T-Interrupt*)
(*metis Diff-insert-absorb Nil-elem-T append.right-neutral append-Nil tickFree-append*)
qed
qed

thus ‹*?lhs* = *?rhs*›

apply (*cases* ‹$P = \bot$›, *solves* ‹*simp add*: *Interrupt-BOT-absorb*›)
apply (*cases* ‹$Q = \bot$›, *solves* ‹*simp add*: *Interrupt-BOT-absorb*›)
apply (*cases* ‹$R = \bot$›, *solves* ‹*simp add*: *Interrupt-BOT-absorb*›)
by *blast*
qed

2.3.5 Key Property

lemma *Interrupt-Mprefix*:
‹$(\Box a \in A \rightarrow P\ a) \mathbin{\triangle} Q = Q \mathbin{\Box} (\Box a \in A \rightarrow P\ a \mathbin{\triangle} Q)$› (**is** ‹*?lhs* = *?rhs*›)
proof (*subst Process-eq-spec-optimized*, *safe*)
fix *s*
assume ‹$s \in \mathcal{D}$ *?lhs*›
then consider ‹$s \in \mathcal{D}\ (\Box a \in A \rightarrow P\ a)$›
| ‹$\exists$ *t1 t2*. $s = t1 \mathbin{@} t2 \wedge t1 \in \mathcal{T}\ (\Box a \in A \rightarrow P\ a) \wedge$ *tickFree t1* $\wedge\ t2 \in \mathcal{D}\ Q$›
by (*simp add*: *D-Interrupt*) *blast*
thus ‹$s \in \mathcal{D}$ *?rhs*›
proof *cases*
show ‹$s \in \mathcal{D}\ (\Box a \in A \rightarrow P\ a) \Longrightarrow s \in \mathcal{D}$ *?rhs*›
by (*simp add*: *D-Det D-Mprefix D-Interrupt*) *blast*
next
assume ‹$\exists$ *t1 t2*. $s = t1 \mathbin{@} t2 \wedge t1 \in \mathcal{T}\ (\Box a \in A \rightarrow P\ a) \wedge$ *tickFree t1* $\wedge\ t2 \in \mathcal{D}\ Q$›
then obtain *t1 t2* **where** ‹$s = t1 \mathbin{@} t2$› ‹$t1 \in \mathcal{T}\ (\Box a \in A \rightarrow P\ a)$›
‹*tickFree t1*› ‹$t2 \in \mathcal{D}\ Q$› **by** *blast*
thus ‹$s \in \mathcal{D}$ *?rhs*›
by (*simp add*: *D-Det D-Mprefix T-Mprefix D-Interrupt*)
(*metis* (*no-types*, *opaque-lifting*) *hd-append2 imageI*
self-append-conv2 tickFree-tl tl-append2)
qed
next
fix *s*
assume ‹$s \in \mathcal{D}$ *?rhs*›
then consider ‹$s \in \mathcal{D}\ Q$› | ‹$\exists\ a\ s'.\ s = ev\ a \#\ s' \wedge a \in A \wedge s' \in \mathcal{D}\ (P\ a \mathbin{\triangle} Q)$›
by (*simp add*: *D-Det D-Mprefix image-iff*) (*metis event.inject list.exhaust-sel*)
thus ‹$s \in \mathcal{D}$ *?lhs*›
proof *cases*
show ‹$s \in \mathcal{D}\ Q \Longrightarrow s \in \mathcal{D}$ *?lhs*›
apply (*simp add*: *D-Interrupt*)
using *Nil-elem-T tickFree-Nil* **by** *blast*
next
assume ‹$\exists\ a\ s'.\ s = ev\ a \#\ s' \wedge a \in A \wedge s' \in \mathcal{D}\ (P\ a \mathbin{\triangle} Q)$›
then obtain *a s'*
where $*$: ‹$s = ev\ a \#\ s'$› ‹$a \in A$›
‹$s' \in \mathcal{D}\ (P\ a) \vee (\exists$ *t1 t2*. $s' = t1 \mathbin{@} t2 \wedge t1 \in \mathcal{T}\ (P\ a) \wedge$
tickFree t1 $\wedge\ t2 \in \mathcal{D}\ Q$)›
by (*simp add*: *D-Interrupt*) *blast*
from $*(3)$ **show** ‹$s \in \mathcal{D}$ *?lhs*›
apply (*elim disjE exE*)

subgoal by (*solves* ‹*simp add*: ∗(*1*, *2*) *D-Interrupt D-Mprefix image-iff*›)
by (*simp add*: ∗(*1*) *D-Interrupt T-Mprefix*)
(*metis* ∗(*2*) *Cons-eq-appendI event.distinct*(*1*) *list.sel*(*1*, *3*) *tickFree-Cons*)
qed
next
fix *s X*
assume *same-div* : ‹$\mathcal{D}$ *?lhs* = $\mathcal{D}$ *?rhs*›
assume ‹(*s*, *X*) ∈ $\mathcal{F}$ *?lhs*›
then consider ‹*s* ∈ $\mathcal{D}$ *?lhs*›
| ‹∃ *t1*. *s* = *t1* @ [*tick*] ∧ *t1* @ [*tick*] ∈ $\mathcal{T}$ (*Mprefix A P*)›
| ‹*s* @ [*tick*] ∈ $\mathcal{T}$ (*Mprefix A P*) ∧ *tick* ∉ *X*›
| ‹(*s*, *X*) ∈ $\mathcal{F}$ (*Mprefix A P*) ∧ *tickFree s* ∧ ([], *X*) ∈ $\mathcal{F}$ *Q*›
| ‹∃ *t1 t2*. *s* = *t1* @ *t2* ∧ *t1* ∈ $\mathcal{T}$ (*Mprefix A P*) ∧ *tickFree t1* ∧ (*t2*, *X*) ∈ $\mathcal{F}$ *Q* ∧ *t2* ≠ []›
| ‹*s* ∈ $\mathcal{T}$ (*Mprefix A P*) ∧ *tickFree s* ∧ [*tick*] ∈ $\mathcal{T}$ *Q* ∧ *tick* ∉ *X*›
by (*simp add*: *F-Interrupt D-Interrupt*) *blast*
thus ‹(*s*, *X*) ∈ $\mathcal{F}$ *?rhs*›
proof *cases*
from *D-F same-div* **show** ‹*s* ∈ $\mathcal{D}$ *?lhs* ⟹ (*s*, *X*) ∈ $\mathcal{F}$ *?rhs*› **by** *blast*
next
show ‹∃ *t1*. *s* = *t1* @ [*tick*] ∧ *t1* @ [*tick*] ∈ $\mathcal{T}$ (*Mprefix A P*) ⟹ (*s*, *X*) ∈ $\mathcal{F}$ *?rhs*›
by (*elim exE*, *simp add*: *F-Det T-Mprefix F-Mprefix F-Interrupt image-iff*)
(*metis event.distinct*(*1*) *hd-append list.sel*(*1*) *tl-append2*)
next
show ‹*s* @ [*tick*] ∈ $\mathcal{T}$ (*Mprefix A P*) ∧ *tick* ∉ *X* ⟹ (*s*, *X*) ∈ $\mathcal{F}$ *?rhs*›
by (*simp add*: *F-Det T-Mprefix F-Mprefix F-Interrupt image-iff*)
(*metis Diff-insert-absorb event.simps*(*3*) *hd-append list.sel*(*1*) *tl-append-if*)
next
show ‹(*s*, *X*) ∈ $\mathcal{F}$ (*Mprefix A P*) ∧ *tickFree s* ∧ ([], *X*) ∈ $\mathcal{F}$ *Q* ⟹ (*s*, *X*) ∈ $\mathcal{F}$ *?rhs*›
by (*simp add*: *F-Det F-Mprefix F-Interrupt image-iff*) (*metis tickFree-tl*)
next
show ‹∃ *t1 t2*. *s* = *t1* @ *t2* ∧ *t1* ∈ $\mathcal{T}$ (*Mprefix A P*) ∧
tickFree t1 ∧ (*t2*, *X*) ∈ $\mathcal{F}$ *Q* ∧ *t2* ≠ [] ⟹ (*s*, *X*) ∈ $\mathcal{F}$ *?rhs*›
by (*elim exE*, *simp add*: *F-Det T-Mprefix F-Mprefix F-Interrupt image-iff*)
(*metis hd-append2 self-append-conv2 tickFree-tl tl-append2*)
next
show ‹*s* ∈ $\mathcal{T}$ (*Mprefix A P*) ∧ *tickFree s* ∧ [*tick*] ∈ $\mathcal{T}$ *Q* ∧ *tick* ∉ *X* ⟹ (*s*, *X*) ∈ $\mathcal{F}$ *?rhs*›
by (*simp add*: *F-Det T-Mprefix F-Mprefix F-Interrupt image-iff*)
(*metis Diff-insert-absorb tickFree-tl*)
qed
next
fix *s X*
assume *same-div* : ‹$\mathcal{D}$ *?lhs* = $\mathcal{D}$ *?rhs*›
assume *assm* : ‹(*s*, *X*) ∈ $\mathcal{F}$ *?rhs*›
show ‹(*s*, *X*) ∈ $\mathcal{F}$ *?lhs*›
proof (*cases* ‹*s* = []›)

from *assm* **show** ‹$s = [] \Longrightarrow (s,\ X) \in \mathcal{F}\ ?lhs$›
by (*simp add*: *F-Det F-Mprefix F-Interrupt Nil-elem-T*) *blast*
next
assume ‹$s \neq []$›
with *assm* **consider** ‹$(s,\ X) \in \mathcal{F}\ Q$›
| ‹$\exists\, a\ s'.\ s = ev\ a\ \#\ s' \wedge a \in A \wedge (s',\ X) \in \mathcal{F}\ (P\ a \mathbin{\triangle} Q)$›
by (*simp add*: *F-Det F-Mprefix image-iff*) (*metis event.inject list.exhaust-sel*)
thus ‹$(s,\ X) \in \mathcal{F}\ ?lhs$›
proof *cases*
show ‹$(s,\ X) \in \mathcal{F}\ Q \Longrightarrow (s,\ X) \in \mathcal{F}\ ?lhs$›
by (*simp add*: *F-Interrupt*)
(*metis Nil-elem-T* ‹$s \neq []$› *append-Nil tickFree-Nil*)
next
assume ‹$\exists\, a\ s'.\ s = ev\ a\ \#\ s' \wedge a \in A \wedge (s',\ X) \in \mathcal{F}\ (P\ a \mathbin{\triangle} Q)$›
then obtain *a s'*
where $*$: ‹$s = ev\ a\ \#\ s'$› ‹$a \in A$› ‹$(s',\ X) \in \mathcal{F}\ (P\ a \mathbin{\triangle} Q)$› **by** *blast*
from $*(3)$ **consider** ‹$s' \in \mathcal{D}\ (P\ a \mathbin{\triangle} Q)$›
| ‹$\exists\, t1.\ s' = t1\ @\ [tick] \wedge t1\ @\ [tick] \in \mathcal{T}\ (P\ a)$›
| ‹$s'\ @\ [tick] \in \mathcal{T}\ (P\ a) \wedge tick \notin X$›
| ‹$(s',\ X) \in \mathcal{F}\ (P\ a) \wedge tickFree\ s' \wedge ([],\ X) \in \mathcal{F}\ Q$›
| ‹$\exists\, t1\ t2.\ s' = t1\ @\ t2 \wedge t1 \in \mathcal{T}\ (P\ a) \wedge tickFree\ t1 \wedge (t2,\ X) \in \mathcal{F}\ Q \wedge t2 \neq []$›
| ‹$s' \in \mathcal{T}\ (P\ a) \wedge tickFree\ s' \wedge [tick] \in \mathcal{T}\ Q \wedge tick \notin X$›
by (*simp add*: *F-Interrupt D-Interrupt*) *blast*
thus ‹$(s,\ X) \in \mathcal{F}\ ?lhs$›
proof *cases*
assume ‹$s' \in \mathcal{D}\ (P\ a \mathbin{\triangle} Q)$›
hence ‹$s \in \mathcal{D}\ ?lhs$›
by (*simp add*: *D-Interrupt D-Mprefix T-Mprefix* $*(1,\ 2)$ *image-iff*)
(*metis* $*(2)$ *append-Cons event.distinct(1) list.sel(1, 3) tickFree-Cons*)
with *D-F same-div* **show** ‹$(s,\ X) \in \mathcal{F}\ ?lhs$› **by** *blast*
next
show ‹$\exists\, t1.\ s' = t1\ @\ [tick] \wedge t1\ @\ [tick] \in \mathcal{T}\ (P\ a) \Longrightarrow (s,\ X) \in \mathcal{F}\ ?lhs$›
by (*elim exE*, *simp add*: $*(1,\ 2)$ *F-Interrupt T-Mprefix*)
next
show ‹$s'\ @\ [tick] \in \mathcal{T}\ (P\ a) \wedge tick \notin X \Longrightarrow (s,\ X) \in \mathcal{F}\ ?lhs$›
by (*simp add*: $*(1,\ 2)$ *F-Interrupt T-Mprefix*) *blast*
next
show ‹$(s',\ X) \in \mathcal{F}\ (P\ a) \wedge tickFree\ s' \wedge ([],\ X) \in \mathcal{F}\ Q \Longrightarrow (s,\ X) \in \mathcal{F}\ ?lhs$›
by (*simp add*: $*(1,\ 2)$ *F-Interrupt F-Mprefix*)
next
show ‹$\exists\, t1\ t2.\ s' = t1\ @\ t2 \wedge t1 \in \mathcal{T}\ (P\ a) \wedge tickFree\ t1 \wedge (t2,\ X) \in \mathcal{F}\ Q \wedge t2 \neq [] \Longrightarrow (s,\ X) \in \mathcal{F}\ ?lhs$›
by (*elim exE*, *simp add*: *F-Interrupt T-Mprefix* $*(1)$)
(*metis* $*(2)$ *Cons-eq-appendI event.distinct(1) list.sel(1, 3) tickFree-Cons*)
next
show ‹$s' \in \mathcal{T}\ (P\ a) \wedge tickFree\ s' \wedge [tick] \in \mathcal{T}\ Q \wedge tick \notin X \Longrightarrow (s,\ X) \in \mathcal{F}\ ?lhs$›
by (*simp add*: *F-Interrupt T-Mprefix* $*(1,\ 2)$ *image-iff*) *blast*

qed
qed
qed
qed

corollary ‹$(\Box a \in A \to P\ a) \mathbin{\triangle} (\Box b \in B \to Q\ b) =$
$\Box x \in A \cup B \to ($ *if* $x \in A \cap B$ *then* $(P\ x \mathbin{\triangle} (\Box b \in B \to Q\ b)) \sqcap Q\ x$
else if $x \in A$ *then* $P\ x \mathbin{\triangle} (\Box b \in B \to Q\ b)$
else $Q\ x)$›
apply (*subst Interrupt-Mprefix*, *subst Mprefix-Det-distr*)
by (*metis Det-commute Mprefix-Det-distr*)

2.3.6 Continuity

lemma *chain-left-Interrupt*: ‹*chain* $Y \Longrightarrow$ *chain* $(\lambda i.\ Y\ i \mathbin{\triangle} Q)$›
by (*simp add*: *chain-def*)

lemma *chain-right-Interrupt*: ‹*chain* $Y \Longrightarrow$ *chain* $(\lambda i.\ P \mathbin{\triangle} Y\ i)$›
by(*simp add*: *chain-def*)

lemma *cont-left-prem-Interrupt* : ‹$(\bigsqcup\ i.\ Y\ i) \mathbin{\triangle} Q = (\bigsqcup\ i.\ Y\ i \mathbin{\triangle} Q)$›
(**is** ‹*?lhs* = *?rhs*›) **if** *chain* : ‹*chain* Y›
proof (*subst Process-eq-spec*, *safe*)
show ‹$s \in \mathcal{D}$ *?lhs* $\Longrightarrow s \in \mathcal{D}$ *?rhs*› **for** s
by (*simp add*: *limproc-is-thelub chain chain-left-Interrupt*
D-Interrupt T-LUB D-LUB) *blast*
next
fix s
define S
where ‹$S\ i \equiv \{t1.\ s = t1 \wedge t1 \in \mathcal{D}\ (Y\ i)\} \cup$
$\{t1.\ \exists t2.\ s = t1\ @\ t2 \wedge t1 \in \mathcal{T}\ (Y\ i) \wedge$ *tickFree t1* $\wedge\ t2 \in \mathcal{D}\ Q\}$›
for i
assume ‹$s \in \mathcal{D}$ *?rhs*›
hence ‹$s \in \mathcal{D}\ (Y\ i \mathbin{\triangle} Q)$› **for** i
by (*simp add*: *limproc-is-thelub D-LUB chain-left-Interrupt chain*)
hence ‹$\forall i.\ S\ i \neq \{\}$› **by** (*simp add*: *S-def D-Interrupt*) *blast*
moreover have ‹*finite* $(S\ 0)$›
unfolding *S-def*
apply (*rule finite-subset*[*of* - ‹$\{t1.\ \exists t2.\ s = t1\ @\ t2\}$›], *blast*)
by (*metis prefixes-fin*)
moreover have ‹$\forall i.\ S\ (Suc\ i) \subseteq S\ i$›
unfolding *S-def* **apply** (*intro allI Un-mono subsetI*; *simp*)
by (*metis in-mono le-approx1 po-class.chainE chain*)
(*metis le-approx-lemma-T po-class.chain-def subset-eq chain*)
ultimately have ‹$(\bigcap i.\ S\ i) \neq \{\}$›
by (*rule Inter-nonempty-finite-chained-sets*)
then obtain *t1* **where** $*$: ‹$\forall i.\ t1 \in S\ i$›

by (*meson INT-iff ex-in-conv iso-tuple-UNIV-I*)
show ‹$s \in \mathcal{D}$ *?lhs*›
proof (*cases* ‹$\forall\, i.\ s \in \mathcal{D}\ (Y\ i)$›)
case *True*
thus ‹$s \in \mathcal{D}$ *?lhs*› **by** (*simp add*: *D-Interrupt limproc-is-thelub D-LUB chain*)
next
case *False*
with $*$ **obtain** *j t2* **where** $**$: ‹$s = t1\ @\ t2$› ‹$t1 \in \mathcal{T}\ (Y\ j)$› ‹ *tickFree t1*› ‹*t2* $\in \mathcal{D}\ Q$›
by (*simp add*: *S-def*) *blast*
from $*$ *D-T* **have** ‹$\forall\, i.\ t1 \in \mathcal{T}\ (Y\ i)$› **by** (*simp add*: *S-def*) *blast*
with $**(1,\ 3,\ 4)$ **show** ‹$s \in \mathcal{D}$ *?lhs*›
by (*simp add*: *D-Interrupt limproc-is-thelub T-LUB chain*) *blast*
qed
next
show ‹$(s,\ X) \in \mathcal{F}$ *?lhs* $\Longrightarrow$ $(s,\ X) \in \mathcal{F}$ *?rhs*› **for** *s X*
by (*simp add*: *limproc-is-thelub chain chain-left-Interrupt*
F-Interrupt F-LUB T-LUB D-LUB) *blast*
next
fix *s X*
define *S*
where ‹$S\ i \equiv \{t1.\ s = t1\ @\ [tick] \wedge t1\ @\ [tick] \in \mathcal{T}\ (Y\ i)\} \cup$
$\{t1.\ s = t1 \wedge t1\ @\ [tick] \in \mathcal{T}\ (Y\ i) \wedge tick \notin X\} \cup$
$\{t1.\ s = t1 \wedge (t1,\ X) \in \mathcal{F}\ (Y\ i) \wedge tickFree\ t1 \wedge ([],\ X) \in \mathcal{F}\ Q\} \cup$
$\{t1.\ \exists\, t2.\ s = t1\ @\ t2 \wedge t1 \in \mathcal{T}\ (Y\ i) \wedge tickFree\ t1 \wedge (t2,\ X) \in \mathcal{F}\ Q \wedge t2 \neq []\} \cup$
$\{t1.\ s = t1 \wedge t1 \in \mathcal{T}\ (Y\ i) \wedge tickFree\ t1 \wedge [tick] \in \mathcal{T}\ Q \wedge tick \notin X\} \cup$
$\{t1.\ s = t1 \wedge t1 \in \mathcal{D}\ (Y\ i)\} \cup$
$\{t1.\ \exists\, t2.\ s = t1\ @\ t2 \wedge t1 \in \mathcal{T}\ (Y\ i) \wedge tickFree\ t1 \wedge t2 \in \mathcal{D}\ Q\}$›
for *i*
assume ‹$(s,\ X) \in \mathcal{F}$ *?rhs*›
hence ‹$(s,\ X) \in \mathcal{F}\ (Y\ i \triangle Q)$› **for** *i*
by (*simp add*: *limproc-is-thelub F-LUB chain-left-Interrupt chain*)
hence ‹$\forall\, i.\ S\ i \neq \{\}$› **by** (*simp add*: *S-def F-Interrupt*, *safe*; *simp*, *blast*)
moreover have ‹*finite* $(S\ 0)$›
unfolding *S-def*
apply (*intro finite-UnI*)
apply (*all* ‹*rule finite-subset*[*of* - ‹$\{t1.\ \exists\, t2.\ s = t1\ @\ t2\}$›], *blast*›)
by (*metis prefixes-fin*)+
moreover have ‹$\forall\, i.\ S\ (Suc\ i) \subseteq S\ i$›
unfolding *S-def* **apply** (*intro allI Un-mono subsetI*; *simp*)
subgoal using *D-T le-approx2T po-class.chain-def chain* **by** *blast*
subgoal using *D-T le-approx2T po-class.chain-def chain* **by** *blast*
subgoal using *is-processT8-S le-approx2 po-class.chainE chain* **by** *blast*
subgoal by (*metis NT-ND le-approx2T po-class.chainE chain*)
subgoal using *D-T le-approx2T po-class.chain-def chain* **by** *blast*
subgoal by (*meson in-mono le-approx1 po-class.chainE chain*)
by (*metis NT-ND le-approx2T po-class.chainE chain*)

ultimately have ‹$(\bigcap i.\ S\ i) \neq \{\}$›
by (*rule Inter-nonempty-finite-chained-sets*)
then obtain *t1* **where** $*$: ‹$\forall\, i.\ t1 \in S\ i$›
by (*meson INT-iff ex-in-conv iso-tuple-UNIV-I*)
show ‹$(s,\ X) \in \mathcal{F}$ *?lhs*›
proof (*cases* ‹$\exists j\ t2.\ s = t1\ @\ t2 \wedge t1 \in \mathcal{T}\ (Y\ j) \wedge$ *tickFree t1* $\wedge\ t2 \in \mathcal{D}\ Q$›)
case *True1* : *True*
then obtain *j t2* **where** $**$: ‹$s = t1\ @\ t2$› ‹$t1 \in \mathcal{T}\ (Y\ j)$›
‹*tickFree t1*› ‹$t2 \in \mathcal{D}\ Q$› **by** *blast*
from $*$ *F-T NT-ND is-processT3-ST* **have** ‹$\forall\, i.\ t1 \in \mathcal{T}\ (Y\ i)$›
by (*simp add*: *S-def*) *blast*
with $**(1,\ 3,\ 4)$ **show** ‹$(s,\ X) \in \mathcal{F}$ *?lhs*›
by (*simp add*: *F-Interrupt limproc-is-thelub chain T-LUB*) *blast*
next
case *False1* : *False*
show ‹$(s,\ X) \in \mathcal{F}$ *?lhs*›
proof (*cases* ‹$\exists j\ t2.\ s = t1\ @\ t2 \wedge t1 \in \mathcal{T}\ (Y\ j) \wedge$ *tickFree t1* $\wedge$
$(t2,\ X) \in \mathcal{F}\ Q \wedge t2 \neq []$›)
case *True2* : *True*
then obtain *j t2* **where** $**$: ‹$s = t1\ @\ t2$› ‹$t1 \in \mathcal{T}\ (Y\ j)$› ‹*tickFree t1*›
‹$(t2,\ X) \in \mathcal{F}\ Q$› ‹$t2 \neq []$› **by** *blast*
from $*$ *F-T NT-ND is-processT3-ST* **have** ‹$\forall\, i.\ t1 \in \mathcal{T}\ (Y\ i)$›
by (*simp add*: *S-def*) *blast*
with $**(1,\ 3,\ 4,\ 5)$ *False1* **show** ‹$(s,\ X) \in \mathcal{F}$ *?lhs*›
by (*simp add*: *F-Interrupt limproc-is-thelub chain T-LUB*) *blast*
next
case *False2*: *False*
show ‹$(s,\ X) \in \mathcal{F}$ *?lhs*›
proof (*cases* ‹$\exists j.\ s = t1 \wedge t1 \in \mathcal{T}\ (Y\ j) \wedge$ *tickFree t1* $\wedge\ [tick] \in \mathcal{T}\ Q \wedge tick \notin X$›)
case *True3* : *True*
then obtain *j* **where** $**$: ‹$s = t1$› ‹$t1 \in \mathcal{T}\ (Y\ j)$› ‹*tickFree t1*›
‹$[tick] \in \mathcal{T}\ Q$› ‹$tick \notin X$› **by** *blast*
from $*$ *F-T NT-ND is-processT3-ST* **have** ‹$\forall\, i.\ t1 \in \mathcal{T}\ (Y\ i)$›
by (*simp add*: *S-def*) *blast*
with $**(1,\ 3,\ 4,\ 5)$ *False1* **show** ‹$(s,\ X) \in \mathcal{F}$ *?lhs*›
by (*simp add*: *F-Interrupt limproc-is-thelub chain T-LUB*) *blast*
next
case *False3*: *False*
show ‹$(s,\ X) \in \mathcal{F}$ *?lhs*›
proof (*cases* ‹$\exists j.\ s = t1 \wedge (t1,\ X) \in \mathcal{F}\ (Y\ j) \wedge$ *tickFree t1* $\wedge\ ([],\ X) \in \mathcal{F}\ Q$›)
case *True4* : *True*
then obtain *j* **where** $**$: ‹$s = t1$› ‹$(t1,\ X) \in \mathcal{F}\ (Y\ j)$› ‹*tickFree t1*› ‹$([],\ X) \in \mathcal{F}\ Q$› **by** *blast*
have ‹$(t1,\ X) \in \mathcal{F}\ (Y\ i)$› **for** *i*
using $*[$*rule-format*, *of i*$]$ **apply** (*simp add*: *S-def* $**(1)$, *elim disjE*)
subgoal by (*solves* ‹*simp add*: *T-F-spec is-processT6-S1*›)
subgoal by (*solves* ‹*simp add*: *T-F-spec is-processT6-S1*›)

```
      subgoal using **(1) False3 by blast
      subgoal by (solves ‹simp add: is-processT8›)
      using **(1) False1 by blast
    with **(1, 3, 4) show ‹(s, X) ∈ ℱ ?lhs›
      by (simp add: F-Interrupt limproc-is-thelub chain F-LUB)
  next
    case False4: False
    show ‹(s, X) ∈ ℱ ?lhs›
    proof (cases ‹∃j. s = t1 @ [tick] ∧ t1 @ [tick] ∈ 𝒯 (Y j)›)
      case True5 : True
      then obtain j where ** : ‹s = t1 @ [tick]› ‹t1 @ [tick] ∈ 𝒯 (Y j)› by
blast
      from * False1 False2 have ‹∀i. t1 @ [tick] ∈ 𝒯 (Y i)›
        by (auto simp add: S-def **(1))
      with **(1) show ‹(s, X) ∈ ℱ ?lhs›
        by (simp add: F-Interrupt limproc-is-thelub chain T-LUB)
    next
      case False5 : False
      show ‹(s, X) ∈ ℱ ?lhs›
      proof (cases ‹∃j. s = t1 ∧ t1 @ [tick] ∈ 𝒯 (Y j) ∧ tick ∉ X›)
        case True6 : True
       then obtain j where ** : ‹s = t1› ‹t1 @ [tick] ∈ 𝒯 (Y j)› ‹tick ∉ X›
by blast
        have ‹t1 @ [tick] ∈ 𝒯 (Y i)› for i
          using *[rule-format, of i] apply (simp add: S-def **(1), elim disjE)
          subgoal by (solves simp)
          subgoal using **(1) False4 by blast
          subgoal using **(1) False3 by blast
              subgoal using **(2) D-T front-tickFree-mono is-processT2-TR
is-processT7 by blast
          subgoal using **(1) False1 by blast
          done
        with **(1, 3) show ‹(s, X) ∈ ℱ ?lhs›
          by (simp add: F-Interrupt limproc-is-thelub chain T-LUB)
            (metis Diff-insert-absorb)
      next
        case False6: False
        have ‹s = t1 ∧ t1 ∈ 𝒟 (Y i)› for i
          using *[rule-format, of i] apply (simp add: S-def, elim disjE)
          subgoal using False5 by blast
          subgoal using False6 by blast
          subgoal using False4 by blast
          subgoal using False2 by blast
          subgoal using False3 by blast
          subgoal by (solves simp)
          subgoal using False1 by blast
          done
        thus ‹(s, X) ∈ ℱ ?lhs›
          by (simp add: F-Interrupt limproc-is-thelub chain D-LUB)
```

```
        qed
      qed
    qed
  qed
 qed
qed


lemma cont-right-prem-Interrupt : ‹P △ (⨆ i. Y i) = (⨆ i. P △ Y i)›
 (is ‹?lhs = ?rhs›) if chain : ‹chain Y›
proof (subst Process-eq-spec, safe)
 show ‹s ∈ 𝒟 ?lhs ⟹ s ∈ 𝒟 ?rhs› for s
  by (simp add: limproc-is-thelub chain chain-right-Interrupt
            D-Interrupt D-LUB) blast
next
 fix s
 define S where ‹S i ≡ {t1. ∃ t2. s = t1 @ t2 ∧ t1 ∈ 𝒯 P ∧
                                tickFree t1 ∧ t2 ∈ 𝒟 (Y i)}› for i
 assume assm : ‹s ∈ 𝒟 ?rhs›
 show ‹s ∈ 𝒟 ?lhs›
 proof (cases ‹s ∈ 𝒟 P›)
  show ‹s ∈ 𝒟 P ⟹ s ∈ 𝒟 ?lhs› by (simp add: D-Interrupt)
 next
  assume ‹s ∉ 𝒟 P›
  with assm have ‹∀ i. S i ≠ {}›
   by (simp add: limproc-is-thelub chain chain-right-Interrupt
             S-def D-Interrupt D-LUB) blast
  moreover have ‹finite (S 0)›
   unfolding S-def
   apply (rule finite-subset[of - ‹{t1. ∃ t2. s = t1 @ t2}›], blast)
   by (metis prefixes-fin)
  moreover have ‹∀ i. S (Suc i) ⊆ S i›
   unfolding S-def apply (intro allI Un-mono subsetI; simp)
   by (metis in-mono le-approx1 po-class.chainE chain)
  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 ‹s ∈ 𝒟 ?lhs›
   by (simp add: D-Interrupt limproc-is-thelub chain D-LUB S-def) blast
 qed
next
 show ‹(s, X) ∈ ℱ ?lhs ⟹ (s, X) ∈ ℱ ?rhs› for s X
  by (simp add: limproc-is-thelub chain chain-right-Interrupt
            F-Interrupt F-LUB T-LUB D-LUB) blast
next
 fix s X
 define S
```

```
    where ‹S i ≡ {t1. s = t1 ∧ (t1, X) ∈ ℱ P ∧ tickFree t1 ∧ ([], X) ∈ ℱ (Y i)}
∪
                {t1. ∃ t2. s = t1 @ t2 ∧ t1 ∈ 𝒯 P ∧ tickFree t1 ∧ (t2, X) ∈ ℱ (Y
i) ∧ t2 ≠ []} ∪
                {t1. s = t1 ∧ t1 ∈ 𝒯 P ∧ tickFree t1 ∧ [tick] ∈ 𝒯 (Y i) ∧ tick ∉
X} ∪
                {t1. ∃ t2. s = t1 @ t2 ∧ t1 ∈ 𝒯 P ∧ tickFree t1 ∧ t2 ∈ 𝒟 (Y i)}›
for i
  assume assm : ‹(s, X) ∈ ℱ ?rhs›
  show ‹(s, X) ∈ ℱ ?lhs›
  proof (cases ‹∃ t1. s = t1 @ [tick] ∧ t1 @ [tick] ∈ 𝒯 P ∨
                    s = t1 ∧ t1 @ [tick] ∈ 𝒯 P ∧ tick ∉ X ∨
                    s = t1 ∧ t1 ∈ 𝒟 P›)
    case True
    thus ‹(s, X) ∈ ℱ ?lhs›
      by (simp add: F-Interrupt) (metis Diff-insert-absorb)
  next
    case False
    with assm have ‹∀ i. S i ≠ {}›
      by (simp add: limproc-is-thelub chain chain-right-Interrupt
                    S-def F-Interrupt F-LUB) blast
    moreover have ‹finite (S 0)›
      unfolding S-def
      apply (rule finite-subset[of - ‹{t1. ∃ t2. s = t1 @ t2}›], blast)
      by (metis prefixes-fin)
    moreover have ‹∀ i. S (Suc i) ⊆ S i›
      unfolding S-def apply (intro allI Un-mono subsetI; simp)
      subgoal by (meson le-approx2 po-class.chainE process-charn chain)
      subgoal by (metis le-approx2 po-class.chainE process-charn chain)
    subgoal by (metis insert-absorb insert-subset le-approx-lemma-T po-class.chainE
chain)
      by (metis in-mono le-approx1 po-class.chainE chain)
    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)

    show ‹(s, X) ∈ ℱ ?lhs›
    proof (cases ‹∃ j t2. s = t1 @ t2 ∧ t1 ∈ 𝒯 P ∧ tickFree t1 ∧
                          (t2, X) ∈ ℱ (Y j) ∧ t2 ≠ []›)
      case True1 : True
      then obtain j t2
        where ∗∗ : ‹s = t1 @ t2› ‹t1 ∈ 𝒯 P› ‹tickFree t1›
                   ‹(t2, X) ∈ ℱ (Y j)› ‹t2 ≠ []› by blast
       from ∗ ∗∗(1, 5) D-F have ‹∀ i. (t2, X) ∈ ℱ (Y i)› by (simp add: S-def)
blast
      with ∗∗(1, 2, 3, 5) show ‹(s, X) ∈ ℱ ?lhs›
        by (simp add: F-Interrupt limproc-is-thelub chain F-LUB) blast
    next
```

```
      case False1: False
      show ‹(s, X) ∈ ℱ ?lhs›
      proof (cases ‹∃j. s = t1 ∧ (t1, X) ∈ ℱ P ∧ tickFree t1 ∧ ([], X) ∈ ℱ (Y
j)›)
        case True2 : True
        then obtain j where ** : ‹s = t1› ‹(t1, X) ∈ ℱ P›
                                 ‹tickFree t1› ‹([], X) ∈ ℱ (Y j)› by blast
        from * **(1) D-F is-processT6-S2 have ‹∀i. ([], X) ∈ ℱ (Y i)›
          by (simp add: S-def) blast
        with **(1, 2, 3) show ‹(s, X) ∈ ℱ ?lhs›
          by (simp add: F-Interrupt limproc-is-thelub chain F-LUB)
      next
        case False2: False
        show ‹(s, X) ∈ ℱ ?lhs›
        proof (cases ‹∃j. s = t1 ∧ t1 ∈ 𝒯 P ∧ tickFree t1 ∧
                        [tick] ∈ 𝒯 (Y j) ∧ tick ∉ X›)
          case True3 : True
          then obtain j where ** : ‹s = t1› ‹t1 ∈ 𝒯 P› ‹tickFree t1›
                                   ‹[tick] ∈ 𝒯 (Y j)› ‹tick ∉ X› by blast
          from * **(1) False2 have ‹∀i. [tick] ∈ 𝒯 (Y i)›
            by (simp add: S-def)
               (metis BOT-iff-D CollectI D-Bot NT-ND
                      front-tickFree-Nil front-tickFree-single)
          with **(1, 2, 3, 5) show ‹(s, X) ∈ ℱ ?lhs›
            by (simp add: F-Interrupt limproc-is-thelub chain T-LUB) blast
        next
          case False3 : False
          have ‹∃t2. s = t1 @ t2 ∧ t1 ∈ 𝒯 P ∧ tickFree t1 ∧ t2 ∈ 𝒟 (Y i)› for i
            using *[rule-format, of i] apply (simp add: S-def, elim disjE)
            using False1 False2 False3 by blast+

          thus ‹(s, X) ∈ ℱ ?lhs›
            by (simp add: F-Interrupt limproc-is-thelub chain D-LUB)
               (metis same-append-eq)
        qed
      qed
    qed
  qed
qed


lemma Interrupt-cont[simp] :
  assumes cont-f : ‹cont f› and cont-g : ‹cont g›
  shows ‹cont (λx. f x △ g x)›
proof −
  have * : ‹cont (λy. y △ g x)› for x
    by (rule contI2, rule monofunI, solves simp, simp add: cont-left-prem-Interrupt)
  have ‹⋀y. cont (Interrupt y)›
    by (simp add: contI2 cont-right-prem-Interrupt fun-belowD lub-fun monofunI)
```

hence $** : \langle cont\ (\lambda x.\ y \triangle g\ x)\rangle$ **for** y
by (*rule cont-compose*) (*simp add*: *cont-g*)
show *?thesis* **by** (*fact cont-apply*[*OF cont-f* $*$ $**$])
qed

end

Chapter 3

The Ready Set Notion

theory *ReadySet*
imports *Sliding Throw Interrupt*
begin

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 traces to begin with *ev e*.

Therefore we must define a notion that captures this idea.

We start by giving a notation to *tick* to be closer to Roscoe's book [3] (and also closer to classic CSP literature).

notation *tick* (‹✓›)

3.1 Definition

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

definition *ready-set* :: ‹'α *process* ⇒ 'α *event set*›
where ‹*ready-set P* ≡ {*a*. [*a*] ∈ $\mathcal{T}$ *P*}›

lemma *ready-set-def-bis*: ‹*ready-set P* = {*e*. ∃ *s*. *e* # *s* ∈ $\mathcal{T}$ *P*}›
and *Cons-in-T-imp-elem-ready-set*: ‹*e* # *s* ∈ $\mathcal{T}$ *P* ⟹ *e* ∈ *ready-set P*›
unfolding *ready-set-def* **using** *is-processT3-ST* **by** *force+*

We say here that the *ready-set* 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 *ready-set P* as the set of events that P can not refuse at first: {*e*. {*e*} ∉ $\mathcal{R}$ *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 (where it is called *initials*) [4, p.40].

notepad
begin
  fix e :: ‹$'\alpha$ *event*› — just fixing $'\alpha$ type

  define *bad-ready-set*
    where ‹*bad-ready-set* (P :: $'\alpha$ *process*) ≡ {e. {e} ∉ $\mathcal{R}$ P}› **for** P

  have *bad-ready-set-subset-ready-set*:
    ‹*bad-ready-set* P ⊆ *ready-set* P› **for** P
    unfolding *bad-ready-set-def ready-set-def Refusals-iff*
    using *F-T is-processT5-S6* **by** *blast*

  have *bad-behaviour-with-Ndet*:
    ‹∃ P Q. *bad-ready-set* (P ⊓ Q) ≠ *bad-ready-set* P ∪ *bad-ready-set* Q›
  proof (*intro exI*)
    show ‹*bad-ready-set* (*SKIP* ⊓ ⊥) ≠ *bad-ready-set SKIP* ∪ *bad-ready-set* ⊥›
    by (*simp add*: *Ndet-BOT*)
       (*simp add*: *bad-ready-set-def F-Ndet F-SKIP F-UU Refusals-iff*)
  qed
end

3.2 Anti-Mono Rules

lemma *anti-mono-ready-set-T*: ‹P $\sqsubseteq_T$ Q ⟹ *ready-set* Q ⊆ *ready-set* P›
  by (*simp add*: *Collect-mono-iff trace-refine-def ready-set-def subsetD*)

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

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

lemma *anti-mono-ready-set-FD*: ‹P $\sqsubseteq_{FD}$ Q ⟹ *ready-set* Q ⊆ *ready-set* P›
  by (*simp add*: *anti-mono-ready-set-F leFD-imp-leF*)

lemma *anti-mono-ready-set*: ‹P ⊑ Q ⟹ *ready-set* Q ⊆ *ready-set* P›
  by (*simp add*: *anti-mono-ready-set-T le-approx-lemma-T trace-refine-def*)

lemma *anti-mono-ready-set-DT*: ‹P $\sqsubseteq_{DT}$ Q ⟹ *ready-set* Q ⊆ *ready-set* P›
  by (*simp add*: *anti-mono-ready-set-T leDT-imp-leT*)

3.3 Behaviour of *ready-set* with *STOP*, *SKIP* and $\bot$

lemma *ready-set-STOP*: ‹*ready-set STOP* = {}›
by (*simp add*: *ready-set-def T-STOP*)

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

lemma *ready-set-empty-iff-STOP*: ‹*ready-set P* = {} ⟷ *P* = *STOP*›
proof (*intro iffI*)
{ assume ‹$\mathcal{T}$ *P* ≠ {[]}›
then obtain *a s* **where** ‹*a* # *s* ∈ $\mathcal{T}$ *P*›
by (*metis Nil-elem-T is-singleton-the-elem is-singletonI'*
list.exhaust-sel singletonI empty-iff)
hence ‹∃ *a*. [*a*] ∈ $\mathcal{T}$ *P*› **by** (*metis append-Cons append-Nil is-processT3-ST*)**}**
thus ‹*ready-set P* = {} ⟹ *P* = *STOP*›
by (*simp add*: *STOP-iff-T ready-set-def*) *presburger*
qed (*simp add*: *ready-set-STOP*)

lemma *ready-set-SKIP*: ‹*ready-set SKIP* = {✓}›
by (*simp add*: *ready-set-def T-SKIP*)

lemma *ready-set-BOT*: ‹*ready-set* ⊥ = *UNIV*›
by (*simp add*: *ready-set-def T-UU*)

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

lemma ‹∃ *P*. *ready-set P* = {✓} ∧ *P* ≠ *SKIP*›
proof (*intro exI*)
show ‹*ready-set* (*STOP* ⊓ *SKIP*) = {✓} ∧ *STOP* ⊓ *SKIP* ≠ *SKIP*›
by (*simp add*: *ready-set-def T-Ndet T-STOP T-SKIP*)
(*metis SKIP-F-iff SKIP-Neq-STOP idem-F mono-Ndet-F-left*)
qed

lemma ‹∃ *P*. *ready-set P* = *UNIV* ∧ *P* ≠ ⊥›
proof (*intro exI*)
show ‹*ready-set* ((□*a* ∈ *UNIV* → ⊥) ⊓ *SKIP*) = *UNIV* ∧ (□*a* ∈ *UNIV* → ⊥)
⊓ *SKIP* ≠ ⊥›
by (*auto simp add*: *ready-set-def T-Ndet T-Mprefix Nil-elem-T T-SKIP*
Ndet-is-BOT-iff SKIP-neq-BOT Mprefix-neq-BOT
intro: *event.exhaust*)
qed

3.4 Behaviour of *ready-set* with Operators of HOL-CSP

lemma *ready-set-Mprefix*: ‹*ready-set* (□*a* ∈ *A* → *P a*) = *ev* ‘ *A*›
and *ready-set-Mndetprefix*: ‹*ready-set* (⊓*a* ∈ *A* → *P a*) = *ev* ‘ *A*›
and *ready-set-prefix*: ‹*ready-set* (*a* → *Q*) = {*ev a*}›
by (*auto simp*: *ready-set-def T-Mndetprefix write0-def T-Mprefix Nil-elem-T*)

As discussed earlier, *ready-set* behaves well with ($\Box$) and ($\sqcap$).

lemma *ready-set-Det*: ‹*ready-set* (*P* □ *Q*) = *ready-set P* ∪ *ready-set Q*›
and *ready-set-Ndet*: ‹*ready-set* (*P* ⊓ *Q*) = *ready-set P* ∪ *ready-set Q*›
unfolding *ready-set-def* **by** (*auto simp add*: *T-Det T-Ndet*)

lemma *ready-set-Seq*:
‹*ready-set* (*P* ; *Q*) = (*if P* = ⊥ *then UNIV else ready-set P* − {✓} ∪
(*if* ✓ ∈ *ready-set P then ready-set Q else* {}))›
proof −
have ‹*ready-set* (*P* ; *Q*) = *ready-set P*› **if** ‹✓ ∉ *ready-set P*›
proof (*intro subset-antisym subsetI*)
fix *e*
assume ‹*e* ∈ *ready-set* (*P* ; *Q*)›
then obtain *s* **where** ‹*e* # *s* ∈ *T* (*P* ; *Q*)› **by** (*simp add*: *ready-set-def*)
then consider ‹*e* # *s* ∈ *T P*›
| ‹∃ *t1 t2*. *e* # *s* = *t1* @ *t2* ∧ *t1* @ [✓] ∈ *T P* ∧ *t2* ∈ *T Q*›
by (*simp add*: *T-Seq*) (*metis F-T NF-ND*)
thus ‹*e* ∈ *ready-set P*›
proof *cases*
show ‹*e* # *s* ∈ *T P* ⟹ *e* ∈ *ready-set P*›
by (*simp add*: *Cons-in-T-imp-elem-ready-set*)
next
assume ‹∃ *t1 t2*. *e* # *s* = *t1* @ *t2* ∧ *t1* @ [✓] ∈ *T P* ∧ *t2* ∈ *T Q*›
then obtain *t1 t2* **where** ∗ : ‹*e* # *s* = *t1* @ *t2*› ‹*t1* @ [✓] ∈ *T P*› **by** *blast*
with *that* **have** ‹*t1* ≠ [] ∧ *hd t1* = *e*›
by (*metis Cons-in-T-imp-elem-ready-set append-Nil hd-append2 list.sel*(*1*))
thus ‹*e* ∈ *ready-set P*›
by (*metis Cons-in-T-imp-elem-ready-set* ∗(*2*) *is-processT3-ST list.exhaust-sel*)
qed
next
show ‹*e* ∈ *ready-set P* ⟹ *e* ∈ *ready-set* (*P* ; *Q*)› **for** *e*
by (*simp add*: *ready-set-def T-Seq*)
(*metis Cons-in-T-imp-elem-ready-set Nil-elem-T that*
append.right-neutral is-processT5-S7 singletonD)
qed
also have ‹*ready-set* (*P* ; *Q*) = *ready-set P* − {✓} ∪ *ready-set Q*›
if ‹*P* ≠ ⊥› **and** ‹✓ ∈ *ready-set P*›
proof (*intro subset-antisym subsetI*)
fix *e*
assume ‹*e* ∈ *ready-set* (*P* ; *Q*)›
then obtain *s* **where** ‹*e* # *s* ∈ *T* (*P* ; *Q*)› **by** (*simp add*: *ready-set-def*)
with ‹*P* ≠ ⊥› **consider** ‹*e* # *s* ∈ *T P*› ‹*e* ≠ ✓›
| ‹∃ *t1 t2*. *e* # *s* = *t1* @ *t2* ∧ *t1* @ [✓] ∈ *T P* ∧ *t2* ∈ *T Q*›
by (*simp add*: *T-Seq BOT-iff-D*)
(*metis F-T append-T-imp-tickFree append-butlast-last-id butlast.simps*(*2*)
last-ConsL list.distinct(*1*) *process-charn tickFree-Cons*)
thus ‹*e* ∈ *ready-set P* − {✓} ∪ *ready-set Q*›
proof *cases*

show ‹$e \# s \in \mathcal{T}\ P \Longrightarrow e \neq \checkmark \Longrightarrow e \in$ *ready-set* $P - \{\checkmark\} \cup$ *ready-set* Q›
by (*simp add*: *Cons-in-T-imp-elem-ready-set*)
next
assume ‹$\exists$ *t1 t2*. $e \# s = t1 \mathbin{@} t2 \wedge t1 \mathbin{@} [\checkmark] \in \mathcal{T}\ P \wedge t2 \in \mathcal{T}\ Q$›
then obtain *t1 t2* **where** $*$: ‹$e \# s = t1 \mathbin{@} t2$› ‹$t1 \mathbin{@} [\checkmark] \in \mathcal{T}\ P$› ‹$t2 \in \mathcal{T}\ Q$› **by** *blast*
show ‹$e \in$ *ready-set* $P - \{\checkmark\} \cup$ *ready-set* Q›
proof (*cases t1*)
from $*(1, 3)$ **show** ‹$t1 = [] \Longrightarrow e \in$ *ready-set* $P - \{\checkmark\} \cup$ *ready-set* Q›
using *Cons-in-T-imp-elem-ready-set* **by** *auto*
next
fix e' $t1'$
assume ‹$t1 = e' \# t1'$›
with $*(1, 2)$ **have** ‹$t1 = e \# t1' \wedge e \neq \checkmark$›
by (*metis append-T-imp-tickFree hd-append list.sel*(1) *neq-Nil-conv tick-Free-Cons*)
with $*(2)$ **show** ‹$e \in$ *ready-set* $P - \{\checkmark\} \cup$ *ready-set* Q›
using *Cons-in-T-imp-elem-ready-set* **by** *auto*
qed
qed
next
fix *e*
assume ‹$e \in$ *ready-set* $P - \{\checkmark\} \cup$ *ready-set* Q›
then obtain *s* **where** ‹$e \neq \checkmark \wedge e \# s \in \mathcal{T}\ P \vee e \# s \in \mathcal{T}\ Q$›
unfolding *ready-set-def* **by** *blast*
thus ‹$e \in$ *ready-set* $(P\ ;\ Q)$›
proof (*elim disjE conjE*)
show ‹$e \neq \checkmark \Longrightarrow e \# s \in \mathcal{T}\ P \Longrightarrow e \in$ *ready-set* $(P\ ;\ Q)$›
by (*simp add*: *ready-set-def T-Seq*)
(*metis Nil-elem-T append-Cons append-Nil*
is-processT3-ST is-processT5-S7 singletonD)
next
have ‹$[\checkmark] \in \mathcal{T}\ P$› **using** *ready-set-def that*(2) **by** *auto*
thus ‹$e \# s \in \mathcal{T}\ Q \Longrightarrow e \in$ *ready-set* $(P\ ;\ Q)$›
by (*simp add*: *ready-set-def T-Seq*)
(*metis append-Cons append-Nil is-processT3-ST*)
qed
qed
ultimately show *?thesis* **by** (*simp add*: *BOT-Seq ready-set-BOT*)
qed

lemma *ready-set-Sync*:
‹*ready-set* $(P \llbracket S \rrbracket Q) =$
(*if* $P = \bot \vee Q = \bot$ *then UNIV*
else (*ready-set* $P -$ *insert* $\checkmark$ (*ev* ‘ S)) $\cup$
(*ready-set* $Q -$ *insert* $\checkmark$ (*ev* ‘ S)) $\cup$
ready-set $P \cap$ *ready-set* $Q \cap$ *insert* $\checkmark$ (*ev* ‘ S))›

```
  (is ‹ready-set (P ⟦S⟧ Q) = (if P = ⊥ ∨ Q = ⊥ then UNIV else ?rhs)›)
proof −
  have ‹ready-set (P ⟦S⟧ Q) = ?rhs› if non-BOT : ‹P ≠ ⊥› ‹Q ≠ ⊥›
  proof (intro subset-antisym subsetI)
    show ‹e ∈ ?rhs ⟹ e ∈ ready-set (P ⟦S⟧ Q)› for e
      by (use Nil-elem-T in ‹fastforce simp add: ready-set-def T-Sync›)
  next
    fix e
    assume ‹e ∈ ready-set (P ⟦S⟧ Q)›
    then consider ‹∃ t u. t ∈ 𝒯 P ∧ u ∈ 𝒯 Q ∧ [e] setinterleaves ((t, u), insert
✓ (ev ‘ S))›
      | ‹∃ t u r v. front-tickFree v ∧ (tickFree r ∨ v = []) ∧ [e] = r @ v ∧
                  r setinterleaves ((t, u), insert ✓ (ev ‘ S)) ∧
                  (t ∈ 𝒟 P ∧ u ∈ 𝒯 Q ∨ t ∈ 𝒟 Q ∧ u ∈ 𝒯 P)›
      by (simp add: ready-set-def T-Sync) blast
    thus ‹e ∈ ?rhs›
    proof cases
      assume ‹∃ t u. t ∈ 𝒯 P ∧ u ∈ 𝒯 Q ∧ [e] setinterleaves ((t, u), insert ✓ (ev
‘ S))›
      then obtain t u where assms : ‹t ∈ 𝒯 P› ‹u ∈ 𝒯 Q›
                                    ‹[e] setinterleaves ((t, u), insert ✓ (ev ‘ S))› by blast
      thus ‹e ∈ ?rhs›
        apply (cases t; cases u; simp add: ready-set-def split: if-split-asm)
        using empty-setinterleaving Sync.sym by blast+
    next
      assume ‹∃ t u r v. front-tickFree v ∧ (tickFree r ∨ v = []) ∧ [e] = r @ v ∧
                    r setinterleaves ((t, u), insert ✓ (ev ‘ S)) ∧
                    (t ∈ 𝒟 P ∧ u ∈ 𝒯 Q ∨ t ∈ 𝒟 Q ∧ u ∈ 𝒯 P)›
      then obtain t u r v
        where ∗ : ‹front-tickFree v› ‹tickFree r ∨ v = []› ‹[e] = r @ v›
                ‹r setinterleaves ((t, u), insert ✓ (ev ‘ S))›
                ‹t ∈ 𝒟 P ∧ u ∈ 𝒯 Q ∨ t ∈ 𝒟 Q ∧ u ∈ 𝒯 P› by blast
      have ‹r ≠ []› using ∗(4, 5) BOT-iff-D empty-setinterleaving that by blast
      hence ‹r = [e] ∧ v = []›
        by (metis (no-types, lifting) ∗(3) Nil-is-append-conv append-eq-Cons-conv)
      also obtain e′ t′ where ‹t = e′ # t′›
        using ∗(5) BOT-iff-D non-BOT by (cases t; blast)
      ultimately show ‹e ∈ ?rhs›
        using ∗(4, 5) apply (simp add: ready-set-def subset-iff T-Sync)
        apply (cases u; simp split: if-split-asm)
        by (metis (no-types, opaque-lifting) [[metis-verbose = false]]
                  D-T Sync.sym empty-setinterleaving)+
    qed
  qed
  thus ‹ready-set (P ⟦S⟧ Q) = (if P = ⊥ ∨ Q = ⊥ then UNIV else ?rhs)›
    by (simp add: Sync-BOT Sync-commute[of ⊥, simplified Sync-BOT] ready-set-BOT)
qed
```

lemma *ready-set-Renaming*:
‹*ready-set* (*Renaming P f*) = (*if P* = ⊥ *then UNIV else EvExt f* ‘ (*ready-set P*))›
proof −
{ fix *y t1 t2*
assume *assms*: ‹[] ∉ $\mathcal{D}$ *P*› ‹[*y*] = *map* (*EvExt f*) *t1* @ *t2*› ‹*t1* ∈ $\mathcal{D}$ *P*›
from *assms* **have** ‹*t2* = []›
by (*metis Nil-is-append-conv list.map-disc-iff list.sel*(*3*) *tl-append2*)
with *assms*(*2*) *D-T*[*OF assms*(*3*)] **have** ‹∃ *x*. [*x*] ∈ $\mathcal{T}$ *P* ∧ *y* = *EvExt f x*› **by** *auto*
}
thus *?thesis* **by** (*auto simp add*: *ready-set-def T-Renaming image-iff BOT-iff-D*)
qed

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

We start with two characterizations:

- one to understand $P \setminus S = \bot$
- the other 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-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* ∈ $\mathcal{D}$ *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** ‹[] ∈ $\mathcal{D}$ (*P* \ *S*)› **by** (*simp add*: *D-Hiding*) *metis*
qed

lemma *event-in-D-Hiding-iff* :
‹[*e*] ∈ $\mathcal{D}$ (*P* \ *S*) ⟷
P \ *S* = ⊥ ∨ (∃ *x t*. *e* = *ev x* ∧ *x* ∉ *S* ∧ [*ev x*] = *trace-hide t* (*ev* ‘ *S*) ∧
(*t* ∈ $\mathcal{D}$ *P* ∨ (∃*f*. *isInfHiddenRun f P S* ∧ *t* ∈ *range f*)))›
(**is** ‹[*e*] ∈ $\mathcal{D}$ (*P* \ *S*) ⟷ *P* \ *S* = ⊥ ∨ *?ugly-assertion*›)
proof (*intro iffI*)
assume *assm* : ‹[*e*] ∈ $\mathcal{D}$ (*P* \ *S*)›
show ‹*P* \ *S* = ⊥ ∨ *?ugly-assertion*›
proof (*cases e*)

```
    assume ‹e = ✓›
    with assm have ‹P \ S = ⊥›
      using BOT-iff-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-UU)
  next
    show ‹?ugly-assertion ⟹ [e] ∈ 𝒟 (P \ S)›
      by (elim exE, simp add: D-Hiding)
        (metis Hiding-tickFree append-Nil2 event.simps(3)
              front-tickFree-Nil tickFree-Cons tickFree-Nil)
  qed
qed
```

Now we can express *ready-set* $(P \setminus S)$. 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 ready-set-Hiding:
  ‹ready-set (P \ S) =
   (if P \ S = ⊥ then UNIV else
    {e. case e of ✓ ⇒ ∃ t. set t ⊆ ev ‘ S ∧ t @ [✓] ∈ 𝒯 P
        | ev x ⇒ x ∉ S ∧ ([ev x] ∈ 𝒟 (P \ S) ∨
                          (∃ t. [ev x] = trace-hide t (ev ‘ S) ∧ (t, ev ‘ S) ∈ ℱ P))})›
```

```
  (is ‹ready-set (P \ S) = (if P \ S = ⊥ then UNIV else ?set)›)
proof (split if-split, intro conjI impI)
 show ‹P \ S = ⊥ ⟹ ready-set (P \ S) = UNIV› by (simp add: ready-set-BOT)
next
  assume non-BOT : ‹P \ S ≠ ⊥›
  show ‹ready-set (P \ S) = ?set›
  proof (intro subset-antisym subsetI)
    fix e
    assume ready : ‹e ∈ ready-set (P \ S)›
    — This implies e ∉ ev ‘ S with our other assumptions.
    { fix x
      assume assms : ‹x ∈ S› ‹ev x ∈ ready-set (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: ready-set-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 ready have ∗ : ‹e ∉ ev ‘ S› using non-BOT by blast

    from ready consider ‹[e] ∈ 𝒟 (P \ S)›
      | ‹∃ t. [e] = trace-hide t (ev ‘ S) ∧ (t, ev ‘ S) ∈ ℱ P›
      unfolding ready-set-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›
        — because [✓] ∈ 𝒟 (P \ S) ⟹ P \ S = ⊥
        by (metis BOT-iff-D append-Nil event.exhaust non-BOT process-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 = ✓ ⟹ set (butlast t) ⊆ ev ‘ S ∧ butlast t @ [✓] ∈ 𝒯 P›
        using ** apply (cases t rule: rev-cases; simp add: split: if-split-asm)
        by (metis F-T filter-empty-conv subset-code(1))
           (metis Hiding-tickFree front-tickFree-implies-tickFree process-charn
tickFree-Cons)
      thus ‹e = ✓ ⟹ e ∈ ?set› 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 ‹e = ✓› ‹∃ t. set t ⊆ ev ‘ S ∧ t @ [✓] ∈ 𝒯 P›
    | ‹∃ x. e = ev x ∧ x ∉ S ∧
          ([ev x] ∈ 𝒟 (P \ S) ∨
           (∃ t. [ev x] = trace-hide t (ev ‘ S) ∧ (t, ev ‘ S) ∈ ℱ P))› by (cases e;
simp)
  thus ‹e ∈ ready-set (P \ S)›
  proof cases
    assume assms : ‹e = ✓› ‹∃ t. set t ⊆ ev ‘ S ∧ t @ [✓] ∈ 𝒯 P›
    from assms(2) obtain t
      where * : ‹set t ⊆ ev ‘ S› ‹t @ [✓] ∈ 𝒯 P› by blast
    have ** : ‹[e] = trace-hide (t @ [✓]) (ev ‘ S) ∧ (t @ [✓], ev ‘ S) ∈ ℱ P›
      using *(1) by (simp add: assms(1) image-iff tick-T-F[OF *(2)] subset-iff)
    show ‹e ∈ ready-set (P \ S)›
      unfolding ready-set-def by (simp add: T-Hiding) (use ** in blast)
  next
    show ‹∃ x. e = ev x ∧ x ∉ S ∧
          ([ev x] ∈ 𝒟 (P \ S) ∨
           (∃ t. [ev x] = trace-hide t (ev ‘ S) ∧ (t, ev ‘ S) ∈ ℱ P)) ⟹
         e ∈ ready-set (P \ S)›
      apply (elim exE conjE disjE)
      using Cons-in-T-imp-elem-ready-set D-T apply blast
      unfolding ready-set-def by (simp add: T-Hiding) blast
  qed
 qed
qed
```

In the end the result would look something like this:

ready-set $(P \setminus S) = (\mathit{if}\ \exists\, t.\ \mathit{set}\ t \subseteq \mathit{ev}\ `\ S \wedge (t \in \mathcal{D}\ P \vee (\exists f.\ \mathit{isInfHiddenRun}\ f\ P\ S \wedge t \in \mathit{range}\ f))\ \mathit{then}\ \mathit{UNIV}\ \mathit{else}\ \{e.\ \mathit{case}\ e\ \mathit{of}\ \mathit{ev}\ x \Rightarrow x \notin S \wedge (((\exists\, t.\ \mathit{set}\ t \subseteq \mathit{ev}\ `\ S \wedge (t \in \mathcal{D}\ P \vee (\exists f.\ \mathit{isInfHiddenRun}\ f\ P\ S \wedge t \in \mathit{range}\ f))) \vee$

$(\exists\, xa\ t.\ ev\ x = ev\ xa \wedge xa \notin S \wedge [ev\ xa] = \textit{trace-hide}\ t\ (ev\ `\ S) \wedge (t \in \mathcal{D}\ P \vee (\exists\, f.\ \textit{isInfHiddenRun}\ f\ P\ S \wedge t \in \textit{range}\ f)))) \vee (\exists\, t.\ [ev\ x] = \textit{trace-hide}\ t\ (ev\ `\ S) \wedge (t,\ ev\ `\ S) \in \mathcal{F}\ P)) \mid \checkmark \Rightarrow \exists\, t.\ \textit{set}\ t \subseteq ev\ `\ S \wedge t\ @\ [\checkmark] \in \mathcal{T}\ P\})$

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

corollary *ready-tick-Hiding-iff* :
‹$\checkmark \in$ *ready-set* $(P \setminus B) \longleftrightarrow P \setminus B = \bot \vee (\exists\, t.\ \textit{set}\ t \subseteq ev\ `\ B \wedge t\ @\ [\checkmark] \in \mathcal{T}\ P)$›
by (*simp add*: *ready-set-Hiding*)

corollary *ready-tick-imp-ready-tick-Hiding*:
‹$\checkmark \in$ *ready-set* $P \Longrightarrow \checkmark \in$ *ready-set* $(P \setminus B)$›
by (*subst ready-set-Hiding*, *simp add*: *ready-set-def*)
(*metis append-Nil empty-iff empty-set subset-iff*)

corollary *ready-inside-Hiding-iff* :
‹$e \in S \Longrightarrow ev\ e \in$ *ready-set* $(P \setminus S) \longleftrightarrow P \setminus S = \bot$›
by (*simp add*: *ready-set-Hiding*)

corollary *ready-notin-Hiding-iff* :
‹$e \notin S \Longrightarrow ev\ e \in$ *ready-set* $(P \setminus S) \longleftrightarrow$
$P \setminus S = \bot \vee$
$(\exists\, t.\ [ev\ e] = \textit{trace-hide}\ t\ (ev\ `\ S) \wedge$
$(t \in \mathcal{D}\ P \vee (\exists\, f.\ \textit{isInfHiddenRun}\ f\ P\ S \wedge t \in \textit{range}\ f) \vee (t,\ ev\ `\ S) \in \mathcal{F}\ P))$›
by (*auto simp add*: *ready-set-Hiding event-in-D-Hiding-iff split*: *if-split-asm*)

corollary *ready-notin-imp-ready-Hiding*:
‹$ev\ e \in$ *ready-set* $(P \setminus S)$› **if** *ready* : ‹$ev\ e \in$ *ready-set* P› **and** *notin* : ‹$e \notin S$›
proof −
from *inf-hidden*[*of* S ‹$[ev\ e]$› P]
consider ‹$\exists\, f.\ \textit{isInfHiddenRun}\ f\ P\ S \wedge [ev\ e] \in \textit{range}\ f$›
| ‹$\exists\, t.\ [ev\ e] = \textit{trace-hide}\ t\ (ev\ `\ S) \wedge (t,\ ev\ `\ S) \in \mathcal{F}\ P$›
by (*simp add*: *ready-set-Hiding image-iff*[*of* ‹$ev\ e$›] *notin*)
(*metis mem-Collect-eq ready ready-set-def*)
thus ‹$ev\ e \in$ *ready-set* $(P \setminus S)$›
proof *cases*
show ‹$\exists\, f.\ \textit{isInfHiddenRun}\ f\ P\ S \wedge [ev\ e] \in \textit{range}\ f \Longrightarrow ev\ e \in$ *ready-set* $(P \setminus S)$›
apply (*rule Cons-in-T-imp-elem-ready-set*[*of* ‹$ev\ e$› ‹$[]$›], *rule D-T*)
apply (*simp add*: *event-in-D-Hiding-iff image-iff*[*of* ‹$ev\ e$›] *notin*)
by (*metis* (*no-types*, *lifting*) *event.inject filter.simps image-iff notin*)
next
assume ‹$\exists\, t.\ [ev\ e] = \textit{trace-hide}\ t\ (ev\ `\ S) \wedge (t,\ ev\ `\ S) \in \mathcal{F}\ P$›
thus ‹$ev\ e \in$ *ready-set* $(P \setminus S)$› **by** (*simp add*: *ready-set-Hiding notin*)
qed

qed

3.5 Behaviour of *ready-set* with Operators of HOL-CSPM

lemma *ready-set-MultiDet*:
‹*finite A* ⟹ *ready-set* (*MultiDet A P*) = (⋃ *a* ∈ *A*. *ready-set* (*P a*))›
by (*induct A rule*: *finite-induct*)
(*simp-all add*: *ready-set-STOP ready-set-Det*)

lemma *ready-set-MultiNdet*:
‹*finite A* ⟹ *ready-set* (*MultiNdet A P*) = (⋃ *a* ∈ *A*. *ready-set* (*P a*))›
apply (*cases* ‹*A* = {}›, *simp add*: *ready-set-STOP*)
by (*rotate-tac*, *induct A rule*: *finite-set-induct-nonempty*)
(*simp-all add*: *ready-set-Ndet*)

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

lemma *ready-set-MultiSeq*:
‹*ready-set* (*MultiSeq L P*) =
(*if L* = [] *then* {✓}
else if P (*hd L*) = ⊥ *then UNIV*
else if ✓ ∈ *ready-set* (*P* (*hd L*))
then ready-set (*P* (*hd L*)) − {✓} ∪ *ready-set* (*MultiSeq* (*tl L*) *P*)
else ready-set (*P* (*hd L*)))›
by (*induct L*) (*simp-all add*: *ready-set-SKIP ready-set-Seq*)

lemma *ready-set-MultiSync*:
‹*ready-set* (⟦*S*⟧ *m* ∈# *M*. *P m*) =
(*if M* = {#} *then* {}
else if ∃ *m* ∈# *M*. *P m* = ⊥ *then UNIV*
else if ∃ *m*. *M* = {#*m*#} *then ready-set* (*P* (*THE m*. *M* = {#*m*#}))
else {*e*. ∃ *m* ∈# *M*. *e* ∈ *ready-set* (*P m*) − *insert* ✓ (*ev* ‘ *S*)} ∪
{*e* ∈ *insert* ✓ (*ev* ‘ *S*). ∀ *m* ∈# *M*. *e* ∈ *ready-set* (*P m*)})›
proof −
have ∗ : ‹*ready-set* (⟦*S*⟧ *m* ∈# *M* + {#*a*, *a*′#}. *P m*) =
{*e*. ∃ *m* ∈# *M* + {#*a*, *a*′#}. *e* ∈ *ready-set* (*P m*) − *insert* ✓ (*ev* ‘ *S*)} ∪
{*e* ∈ *insert* ✓ (*ev* ‘ *S*). ∀ *m* ∈# *M* + {#*a*, *a*′#}. *e* ∈ *ready-set* (*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 ready-set-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 $**$: ‹$\neg$ (*P a''* $= \bot \vee$ *MultiSync S* ($M' + \{\#a,\ a'\#\}$) *P* $= \bot$)›
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*: $*$ *ready-set-Sync* $**$ *2.hyps*(*3*), *auto*)
qed

show *?thesis*
proof (*cases* ‹$\exists\, m \in\# M.\ P\ m = \bot$›)
show ‹$\exists\, m \in\# M.\ P\ m = \bot \Longrightarrow$ *?thesis*›
by (*simp add*: *ready-set-STOP*) (*metis MultiSync-BOT-absorb ready-set-BOT*)
next
show ‹$\neg$ ($\exists\, m{\in}\#M.\ P\ m = \bot$) $\Longrightarrow$ *?thesis*›
proof (*cases* ‹$\exists\, a\ a'\ M'.\ M = M' + \{\#a,\ a'\#\}$›)
assume *assms* : ‹$\neg$ ($\exists\, m{\in}\#M.\ P\ m = \bot$)› ‹$\exists\, a\ a'\ M'.\ M = M' + \{\#a,\ a'\#\}$›
from *assms*(*2*) **obtain** *a a' M'* **where** ‹$M = M' + \{\#a,\ a'\#\}$› **by** *blast*
with $*$ *assms*(*1*) **show** *?thesis* **by** *simp*
next
assume ‹$\nexists\, a\ a'\ M'.\ M = M' + \{\#a,\ a'\#\}$›
hence ‹$M = \{\#\} \vee (\exists\, m.\ M = \{\#m\#\})$›
by (*metis add.right-neutral multiset-cases union-mset-add-mset-right*)
thus *?thesis* **by** (*auto simp add*: *ready-set-STOP ready-set-BOT*)
qed
qed
qed

3.6 Behaviour of *ready-set* with Operators of `HOL-CSP_OpSem`

lemma *ready-set-Sliding*:
‹*ready-set* (*P* $\rhd$ *Q*) = *ready-set P* $\cup$ *ready-set Q*›
unfolding *ready-set-def* **by** (*auto simp add*: *T-Sliding*)

lemma *ready-set-Throw*: ‹*ready-set* (*P* Θ *a* $\in$ *A*. *Q a*) = *ready-set P*›
apply (*intro subset-antisym subsetI*;
simp add: *ready-set-def T-Throw image-iff*)
apply (*elim disjE*)
apply (*solves* ‹*simp*›)
apply (*metis D-T process-charn*)
apply (*metis Nil-is-append-conv list.sel*(*3*) *neq-Nil-conv self-append-conv2 tl-append2*)
by (*metis Nil-elem-T append-Nil empty-set inf-bot-left*)

corollary *Throw-is-STOP-iff*: ‹*P* Θ *a* $\in$ *A*. *Q a* = *STOP* $\longleftrightarrow$ *P* = *STOP*›
by (*simp add*: *ready-set-empty-iff-STOP*[*symmetric*] *ready-set-Throw*)

lemma *ready-set-Interrupt*: ‹*ready-set* (*P* △ *Q*) = *ready-set P* ∪ *ready-set Q*›
apply (*intro subset-antisym subsetI*;
simp add: *ready-set-def T-Interrupt*)
by (*metis Nil-is-append-conv append.left-neutral*
append.right-neutral butlast.simps(*2*) *butlast-append*)
(*use Nil-elem-T tickFree-Nil* **in** *blast*)

corollary *Interrupt-is-STOP-iff*: ‹*P* △ *Q* = *STOP* ⟷ *P* = *STOP* ∧ *Q* = *STOP*›
by (*simp add*: *ready-set-empty-iff-STOP*[*symmetric*] *ready-set-Interrupt*)

3.7 Behaviour of *ready-set* with Reference Processes

lemma *ready-set-DF*: ‹*ready-set* (*DF A*) = *ev* ‘ *A*›
by (*subst DF-unfold*) (*simp add*: *ready-set-Mndetprefix*)

lemma *ready-set-DF*$_{SKIP}$: ‹*ready-set* (*DF*$_{SKIP}$ *A*) = *insert* ✓ (*ev* ‘ *A*)›
by (*subst DF*$_{SKIP}$*-unfold*)
(*simp add*: *ready-set-Mndetprefix ready-set-Ndet ready-set-SKIP*)

lemma *ready-set-RUN*: ‹*ready-set* (*RUN A*) = *ev* ‘ *A*›
by (*subst RUN-unfold*) (*simp add*: *ready-set-Mprefix*)

lemma *ready-set-CHAOS*: ‹*ready-set* (*CHAOS A*) = *ev* ‘ *A*›
by (*subst CHAOS-unfold*)
(*simp add*: *ready-set-Mprefix ready-set-Ndet ready-set-STOP*)

lemma *ready-set-CHAOS*$_{SKIP}$: ‹*ready-set* (*CHAOS*$_{SKIP}$ *A*) = *insert* ✓ (*ev* ‘ *A*)›
by (*subst CHAOS*$_{SKIP}$*-unfold*)
(*simp add*: *ready-set-Mprefix ready-set-Ndet*
ready-set-STOP ready-set-SKIP)

end

Chapter 4

Construction of the After Operator

theory *After*
imports *ReadySet*
begin

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

4.1 Definition

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

The intuitive way of doing so is to only keep the tails of the traces beginning by *ev e* (and similar for failures and divergences). However we have an issue if *ev e* $\notin$ *ready-set P* i.e. if no trace of P begins with *ev e*: the result would no longer verify the invariant *is-process* because its trace set would be empty. We must therefore distinguish this case and we agree to then obtain *STOP*. This convention is not really decisive since we will only use this operator when *ev e* $\in$ *ready-set P* to define operational semantics.

lift-definition *After* :: ‹['α *process*, 'α] ⇒ 'α *process*› (**infixl** ‹*after*› 77)
is ‹λ*P e*. *if ev e* ∈ *ready-set P*
then ({(*tl s*, *X*)| *s X*. (*s*, *X*) ∈ $\mathcal{F}$ *P* ∧ *s* ≠ [] ∧ *hd s* = *ev e*},
{ *tl s* | *s* . *s* ∈ $\mathcal{D}$ *P* ∧ *s* ≠ [] ∧ *hd s* = *ev e*})
else ({(*s*, *X*). *s* = []}, {})›
proof −
show ‹*?thesis P e*› (**is** ‹*is-process* (*if ev e* ∈ *ready-set P then* (*?f*, *?d*)
else ({(*s*, *X*). *s* = []}, {}))›) **for** *P e*
proof (*split if-split*, *intro conjI impI*)
show ‹*is-process* ({(*s*, *X*). *s* = []}, {})›
by (*simp add*: *is-process-REP-STOP*)

```
  next
    assume ready: ‹ev e ∈ ready-set P›
    show ‹is-process (?f, ?d)›
    unfolding is-process-def FAILURES-def DIVERGENCES-def fst-conv snd-conv
    proof (intro conjI impI allI)
      show ‹([], {}) ∈ ?f›
        using ready[unfolded ready-set-def T-F-spec[symmetric]] by force
    next
      show ‹(s, X) ∈ ?f ⟹ front-tickFree s› for s X
          by simp (metis butlast-rev butlast-tl front-tickFree-def is-processT2 tick-
Free-butlast)
    next
      show ‹(s @ t, {}) ∈ ?f ⟹ (s, {}) ∈ ?f› for s t
        by simp (metis (no-types, opaque-lifting) append-Cons is-processT3
                                                  list.sel(1, 3) neq-Nil-conv)
    next
      show ‹(s, Y) ∈ ?f ∧ X ⊆ Y ⟹ (s, X) ∈ ?f› for s X Y
        using is-processT4 by simp blast
    next
      show ‹(s, X) ∈ ?f ∧ (∀ c. c ∈ Y ⟶ (s @ [c], {}) ∉ ?f) ⟹ (s, X ∪ Y) ∈
?f› for s X Y
        using ready[unfolded ready-set-def T-F-spec[symmetric]]
        by auto (metis Nil-is-append-conv hd-append2 is-processT5 tl-append2)
    next
      show ‹(s @ [✓], {}) ∈ ?f ⟹ (s, X − {✓}) ∈ ?f› for s X
       by simp (metis (no-types, lifting) Cons-eq-appendI is-processT6 list.collapse
                                          list.distinct(1) list.sel(1, 3))
    next
      fix s t :: ‹'α trace›
      assume ‹s ∈ ?d ∧ tickFree s ∧ front-tickFree t›
      hence ‹s @ t = tl (ev e # s @ t) ∧ ev e # s @ t ∈ 𝒟 P ∧
            ev e # s @ t ≠ [] ∧ hd (ev e # s @ t) = ev e›
        by simp (metis append-Cons event.distinct(1) is-processT7-S
                  list.sel(1, 3) neq-Nil-conv tickFree-Cons)
      show ‹s @ t ∈ ?d›
        apply simp
        using ‹?this› by blast
    next
      show ‹s ∈ ?d ⟹ (s, X) ∈ ?f› for s X
        using NF-ND by blast
    next
      show ‹s @ [✓] ∈ ?d ⟹ s ∈ ?d› for s
      by auto (metis Cons-eq-appendI is-processT9-S-swap list.sel(1, 3) neq-Nil-conv)
    qed
  qed
qed
```

4.2 Projections

lemma *F-After*:
‹$\mathcal{F}$ (*P after e*) = (*if ev e* ∈ *ready-set P*
then {(*tl s*, *X*)| *s X*. (*s*, *X*) ∈ $\mathcal{F}$ *P* ∧ *s* ≠ [] ∧ *hd s* = *ev e*}
else {(*s*, *X*). *s* = []})›
by (*simp add*: *Failures-def After.rep-eq FAILURES-def*)

lemma *D-After*:
‹$\mathcal{D}$ (*P after e*) = (*if ev e* ∈ *ready-set P*
then {*tl s*| *s*. *s* ∈ $\mathcal{D}$ *P* ∧ *s* ≠ [] ∧ *hd s* = *ev e*}
else {})›
by (*simp add*: *Divergences-def After.rep-eq DIVERGENCES-def*)

lemma *T-After*:
‹$\mathcal{T}$ (*P after e*) = (*if ev e* ∈ *ready-set P*
then {*tl s*| *s*. *s* ∈ $\mathcal{T}$ *P* ∧ *s* ≠ [] ∧ *hd s* = *ev e*}
else {[]})›
by (*auto simp add*: *T-F-spec*[*symmetric*] *F-After*)

lemma *not-ready-After*: ‹*ev e* ∉ *ready-set P* ⟹ *P after e* = *STOP*›
by (*simp add*: *STOP-iff-T T-After*)

lemma *ready-set-After*: ‹*ready-set* (*P after e*) = {*a*. *ev e* # [*a*] ∈ $\mathcal{T}$ *P*}›
apply (*simp add*: *T-After ready-set-def*, *safe*)
apply (*metis list.exhaust-sel*)
apply (*metis list.discI list.sel*(*1*, *3*))
by (*simp add*: *is-processT3-ST-pref le-list-def*)

4.3 Monotony

lemma *mono-After* : ‹*P after e* ⊑ *Q after e*› **if** ‹*P* ⊑ *Q*›
proof (*subst le-approx-def*, *safe*)
from *that*[*THEN anti-mono-ready-set*] *that*[*THEN le-approx1*]
show ‹*s* ∈ $\mathcal{D}$ (*Q after e*) ⟹ *s* ∈ $\mathcal{D}$ (*P after e*)› **for** *s*
by (*simp add*: *D-After ready-set-def subset-iff split*: *if-split-asm*) *blast*
next
from *that*[*THEN anti-mono-ready-set*] *that*[*THEN le-approx2*]
show ‹*s* ∉ $\mathcal{D}$ (*P after e*) ⟹ *X* ∈ $\mathcal{R}_a$ (*P after e*) *s* ⟹ *X* ∈ $\mathcal{R}_a$ (*Q after e*) *s*›
for *s X*

apply (*simp add*: *Ra-def D-After F-After ready-set-def subset-iff split*: *if-split-asm*)
by (*metis F-T append-Cons append-Nil is-processT3-ST list.exhaust-sel*) *blast*
next
from *that*[*THEN anti-mono-ready-set*] *that*[*THEN le-approx2*]

```
  show ‹s ∉ 𝒟 (P after e) ⟹ X ∈ ℛa (Q after e) s ⟹ X ∈ ℛa (P after e) s›
for s X
   apply (simp add: Ra-def D-After F-After ready-set-def subset-iff split: if-split-asm)
    by blast (metis T-F-spec list.distinct(1) list.sel(1, 3))
next
  show ‹s ∈ min-elems (𝒟 (P after e)) ⟹ s ∈ 𝒯 (Q after e)› for s
  proof (cases ‹P = ⊥›)
    assume ‹s ∈ min-elems (𝒟 (P after e))› and ‹P = ⊥›
    hence ‹s = []›
      by (simp add: BOT-iff-D D-After ready-set-BOT D-UU min-elems-def)
         (metis front-tickFree-single less-list-def list.distinct(1) list.sel(1, 3) nil-le)
    thus ‹s ∈ 𝒯 (Q after e)› by (simp add: Nil-elem-T)
  next
    assume assms : ‹P ≠ ⊥› ‹s ∈ min-elems (𝒟 (P after e))›
    from assms(2)[THEN elem-min-elems] have * : ‹ev e # s ∈ 𝒟 P›
      by (simp add: D-After split: if-split-asm) (metis list.collapse)
    { assume ‹ev e # s ∉ min-elems (𝒟 P)›
      with assms(1) * obtain a t where ‹a # t ∈ 𝒟 P› ‹a # t < ev e # s›
        by (simp add: BOT-iff-D min-elems-def) (metis list.exhaust)
      hence ‹a = ev e ∧ t < s ∧ t ∈ 𝒟 (P after e)›
        by (simp add: le-list-def less-list-def D-After)
           (metis Cons-in-T-imp-elem-ready-set D-T list.discI list.sel(1, 3))
      hence False by (metis assms(2) less-list-def min-elems-no)
    }
    hence ‹ev e # s ∈ min-elems (𝒟 P)› by blast
    with le-approx3 that have ‹ev e # s ∈ 𝒯 Q› by blast
    thus ‹s ∈ 𝒯 (Q after e)›
      by (simp add: T-After)
         (metis Cons-in-T-imp-elem-ready-set list.discI list.sel(1, 3))
  qed
qed


lemma mono-After-T : ‹P ⊑T Q ⟹ P after e ⊑T Q after e›
  by (auto simp add: trace-refine-def T-After ready-set-def)
     (metis list.distinct(1) list.sel(1, 3))

lemma mono-After-F :
  ‹P ⊑F Q ⟹ ev e ∉ ready-set P ∨ ev e ∈ ready-set Q ⟹
   P after e ⊑F Q after e›
  using F-subset-imp-T-subset
  by (auto simp add: failure-refine-def F-After ready-set-def)

lemma mono-After-D : ‹P ⊑D Q ⟹ P after e ⊑D Q after e›
  by (auto simp add: divergence-refine-def D-After ready-set-def)
     (metis Cons-eq-appendI NT-ND append-self-conv2 is-processT3-ST list.collapse
subset-iff)

lemma mono-After-FD :
```

‹$P \sqsubseteq_{FD} Q \Longrightarrow ev\ e \notin ready\text{-}set\ P \vee ev\ e \in ready\text{-}set\ Q \Longrightarrow$
$P\ after\ e \sqsubseteq_{FD} Q\ after\ e$›
using *F-subset-imp-T-subset*
by (*simp add*: *failure-divergence-refine-def le-ref-def F-After D-After ready-set-def*)
blast

lemma *mono-After-DT* : ‹$P \sqsubseteq_{DT} Q \Longrightarrow P\ after\ e \sqsubseteq_{DT} Q\ after\ e$›
by (*simp add*: *mono-After-D mono-After-T trace-divergence-refine-def*)

4.4 Behaviour of *After* with *STOP*, *SKIP* and $\bot$

lemma *After-STOP*: ‹$STOP\ after\ e = STOP$›
by (*simp add*: *STOP-iff-T T-After ready-set-STOP*)

lemma *After-is-STOP-iff*:
‹$P\ after\ e = STOP \longleftrightarrow (\forall\, s.\ ev\ e\ \#\ s \in \mathcal{T}\ P \longrightarrow s = [])$›
apply (*simp add*: *STOP-iff-T T-After ready-set-def*, *safe*)
apply *fastforce*
apply (*metis list.collapse*)
using *is-processT3-ST* **by** *force*+

lemma *After-SKIP*: ‹$SKIP\ after\ e = STOP$›
by (*simp add*: *STOP-iff-T T-After ready-set-SKIP*)

lemma *After-BOT*: ‹$\bot\ after\ e = \bot$›
by (*force simp add*: *BOT-iff-D D-After ready-set-BOT D-UU*)

lemma *After-is-BOT-iff*: ‹$P\ after\ e = \bot \longleftrightarrow [ev\ e] \in \mathcal{D}\ P$›
using *hd-Cons-tl* **by** (*force simp add*: *BOT-iff-D D-After ready-set-def D-T*)

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

4.5.1 Loss of Determinism

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

lemma *After-Mprefix-is-After-Mndetprefix*:
‹$(\Box a \in A \rightarrow P\ a)\ after\ e = (\sqcap a \in A \rightarrow P\ a)\ after\ e$›
by (*subst Process-eq-spec*)
(*force simp add*: *ready-set-Mprefix ready-set-Mndetprefix F-After D-After*
F-Mprefix D-Mprefix F-Mndetprefix D-Mndetprefix write0-def)

lemma *After-Det-is-After-Ndet*: ‹$P \Box Q\ after\ e = P \sqcap Q\ after\ e$›
by (*subst Process-eq-spec*)
(*auto simp add*: *ready-set-Det ready-set-Ndet F-After D-After F-Det F-Ndet D-Det D-Ndet*)

lemma *After-Ndet*:
 ‹*P* ⊓ *Q after e* =
 (*if ev e* ∉ *ready-set P* ∧ *ev e* ∉ *ready-set Q then STOP*
  else if ev e ∈ *ready-set P* ∧ *ev e* ∈ *ready-set Q then* (*P after e*) ⊓ (*Q after e*)
  else if ev e ∈ *ready-set P then P after e else Q after e*)›
 (**is** ‹*P* ⊓ *Q after e* =
  (*if ?c1 then STOP else if ?c2 then* (*P after e*) ⊓ (*Q after e*) *else*
  if ?c3 then P after e else Q after e)›)
proof −
 have ‹*?c1* ⟹ *P* ⊓ *Q after e* = *STOP*›
  by (*simp add*: *Process-eq-spec F-After D-After ready-set-Ndet F-STOP D-STOP*)
 moreover have ‹*?c2* ⟹ *P* ⊓ *Q after e* = (*P after e*) ⊓ (*Q after e*)›
  by (*auto simp add*: *Process-eq-spec F-After D-After F-Ndet D-Ndet ready-set-Ndet*)
 moreover have ‹¬ *?c2* ⟹ *?c3* ⟹ *P* ⊓ *Q after e* = *P after e*›
  and ‹¬ *?c2* ⟹ ¬ *?c3* ⟹ *P* ⊓ *Q after e* = *Q after e*›
  by (*auto simp add*: *Process-eq-spec F-After D-After F-Ndet D-Ndet ready-set-Ndet*)
  (*metis Cons-in-T-imp-elem-ready-set F-T list.collapse*,
  metis Cons-in-T-imp-elem-ready-set D-T list.collapse)+
 ultimately show *?thesis* **by** *presburger*
qed

lemma *After-Mprefix*: ‹(□ *a* ∈ *A* → *P a*) *after e* = (*if e* ∈ *A then P e else STOP*)›
 by (*subst Process-eq-spec*, *auto simp add*: *F-After D-After ready-set-Mprefix*
  F-Mprefix D-Mprefix F-STOP D-STOP)
  (*metis image-eqI list.distinct*(*1*) *list.sel*(*1*, *3*))+

lemmas *After-Det* = *After-Ndet*[*folded After-Det-is-After-Ndet*]
 and *After-Mndetprefix* = *After-Mprefix*[*unfolded After-Mprefix-is-After-Mndetprefix*]
 and *After-prefix* = *After-Mprefix*[*of* ‹{*a*}› ‹λ*a*. *P*›, *folded write0-def*, *simplified*]
for *a P*

lemma ‹(*e* → *P*) *after e* = *P*› **by** (*simp add*: *After-prefix*)

This result justifies seeing *P after e* as the reciprocal operator of the prefix *e* → *P*.

However, we lose information with *After*: in general, *e* → (*P after e*) ≠ *P* (even when *ev e* ∈ *ready-set P* and *P* ≠ ⊥).

lemma ‹∃ *P*. (*e* → (*P after e*) ≠ *P*)›
proof (*intro exI*)
 show ‹*e* → (*SKIP after e*) ≠ *SKIP*›
  by (*metis Par-SKIP-SKIP SKIP-Neq-STOP prefix-Par-SKIP*)
qed

lemma ‹∃ *P*. *ev e* ∈ *ready-set P* ∧ (*e* → (*P after e*) ≠ *P*)›

proof (*intro exI*)
show ‹*ev e* ∈ *ready-set* ⊥ ∧ (*e* → (⊥ *after e*) ≠ ⊥)›
by (*simp add*: *ready-set-BOT Mprefix-neq-BOT write0-def*)
qed

lemma ‹∃ *P*. *ev e* ∈ *ready-set P* ∧ *P* ≠ ⊥ ∧ (*e* → (*P after e*) ≠ *P*)›
proof (*intro exI*)
define *P* **where** *P-def*: ‹*P* = (*e* → *STOP*) □ *SKIP*›
have ∗ : ‹*ev e* ∈ *ready-set P*› **by** (*simp add*: *P-def ready-set-Det ready-set-prefix*)
moreover have ‹*P* ≠ ⊥›
by (*simp add*: *Det-is-BOT-iff Mprefix-neq-BOT P-def SKIP-neq-BOT write0-def*)
moreover have ‹*e* → (*P after e*) = (*e* → *STOP*)›
by (*rule arg-cong*[**where** *f* = ‹λ*P*. (*e* → *P*)›])
(*simp add*: *P-def After-Det ready-set-SKIP ready-set-prefix After-prefix*)
moreover have ‹*e* → *STOP* ≠ *P*›
apply (*rule contrapos-nn*[*of* ‹*ready-set* (*e* → *STOP*) = *ready-set P*› ‹*e* → *STOP* = *P*›])
by (*simp add*: *P-def ready-set-Det ready-set-prefix ready-set-SKIP*)
(*erule arg-cong*)
ultimately show ‹*ev e* ∈ *ready-set P* ∧ *P* ≠ ⊥ ∧ (*e* → (*P after e*) ≠ *P*)› **by** *presburger*
qed

4.5.2 *After* Sequential Composition

The first goal is to obtain an equivalent of $e \rightarrow P \mathbin{;} Q = e \rightarrow (P \mathbin{;} Q)$. But in order to be exhaustive we also have to consider the possibility of Q taking the lead when $\checkmark \in$ *ready-set* P in the sequential composition $P \mathbin{;} Q$.

lemma *not-skippable-or-not-readyR-After-Seq*: ‹(*P* ; *Q*) *after e* = *P after e* ; *Q*›
if ‹✓ ∉ *ready-set P* ∨ *ev e* ∉ *ready-set Q*›
proof (*cases* ‹*P* = ⊥›)
show ‹*P* = ⊥ ⟹ (*P* ; *Q*) *after e* = *P after e* ; *Q*›
by (*simp add*: *BOT-Seq After-BOT*)
next
assume *non-BOT*: ‹*P* ≠ ⊥›
show ‹(*P* ; *Q*) *after e* = *P after e* ; *Q*›
proof (*subst Process-eq-spec-optimized*, *safe*)
fix *s*
assume ‹*s* ∈ 𝒟 ((*P* ; *Q*) *after e*)›
hence ∗ : ‹*ev e* # *s* ∈ 𝒟 (*P* ; *Q*)›
by (*simp add*: *D-After split*: *if-split-asm*) (*metis list.exhaust-sel*)
then consider ‹*ev e* # *s* ∈ 𝒟 *P*›
| ‹∃ *t1 t2*. *ev e* # *s* = *t1* @ *t2* ∧ *t1* @ [✓] ∈ 𝒯 *P* ∧ *t2* ∈ 𝒟 *Q*›
by (*simp add*: *D-Seq*) *blast*
thus ‹*s* ∈ 𝒟 (*P after e* ; *Q*)›
proof *cases*
show ‹*ev e* # *s* ∈ 𝒟 *P* ⟹ *s* ∈ 𝒟 (*P after e* ; *Q*)›
by (*simp add*: *D-Seq D-After*)
(*metis Cons-in-T-imp-elem-ready-set D-T list.discI list.sel*(*1*, *3*))

```
    next
      assume ‹∃ t1 t2. ev e # s = t1 @ t2 ∧ t1 @ [✓] ∈ 𝒯 P ∧ t2 ∈ 𝒟 Q›
      then obtain t1 t2 where ** : ‹ev e # s = t1 @ t2› ‹t1 @ [✓] ∈ 𝒯 P› ‹t2
∈ 𝒟 Q› by blast
    have ‹t1 ≠ []› by (metis ** Cons-in-T-imp-elem-ready-set D-T self-append-conv2
that)
      with **(1) obtain t1'
        where ‹t1 = ev e # t1'› ‹s = t1' @ t2› by (metis Cons-eq-append-conv)
      with **(2, 3) show ‹s ∈ 𝒟 (P after e ; Q)›
        by (simp add: D-Seq T-After)
          (metis Cons-in-T-imp-elem-ready-set list.distinct(1) list.sel(1, 3))
    qed
  next
    fix s
    assume ‹s ∈ 𝒟 (P after e ; Q)›
    hence ‹ev e # s ∈ 𝒟 P ∨ (∃ t1 t2. s = t1 @ t2 ∧ ev e # t1 @ [✓] ∈ 𝒯 P ∧
t2 ∈ 𝒟 Q)›
      by (simp add: D-Seq D-After T-After split: if-split-asm) (metis list.collapse)
    hence ‹ev e # s ∈ 𝒟 (P ; Q)›
      by (elim disjE; simp add: D-Seq) (metis append-Cons)
    thus ‹s ∈ 𝒟 ((P ; Q) after e)›
      by (simp add: D-After)
        (metis Cons-in-T-imp-elem-ready-set D-T list.discI list.sel(1, 3))
  next
    fix s X
    assume same-div : ‹𝒟 ((P ; Q) after e) = 𝒟 (P after e ; Q)›
    assume ‹(s, X) ∈ ℱ ((P ; Q) after e)›
    then consider ‹ev e ∈ ready-set (P ; Q)› ‹(ev e # s, X) ∈ ℱ (P ; Q)›
      | ‹ev e ∉ ready-set (P ; Q)› ‹s = []›
      by (simp add: F-After split: if-split-asm) (metis list.exhaust-sel)
    thus ‹(s, X) ∈ ℱ (P after e ; Q)›
    proof cases
      assume assms : ‹ev e ∈ ready-set (P ; Q)› ‹(ev e # s, X) ∈ ℱ (P ; Q)›
      from assms(2) consider ‹ev e # s ∈ 𝒟 (P ; Q)›
        | ‹(ev e # s, insert ✓ X) ∈ ℱ P› ‹tickFree s›
        | ‹∃ t1 t2. ev e # s = t1 @ t2 ∧ t1 @ [✓] ∈ 𝒯 P ∧ (t2, X) ∈ ℱ Q›
        by (simp add: F-Seq D-Seq) blast
      thus ‹(s, X) ∈ ℱ (P after e ; Q)›
      proof cases
        assume ‹ev e # s ∈ 𝒟 (P ; Q)›
        hence ‹s ∈ 𝒟 ((P ; Q) after e)›
          by (simp add: D-After assms(1)) (metis list.distinct(1) list.sel(1, 3))
        with same-div D-F show ‹(s, X) ∈ ℱ (P after e ; Q)› by blast
      next
       show ‹(ev e # s, insert ✓ X) ∈ ℱ P ⟹ tickFree s ⟹ (s, X) ∈ ℱ (P after
e ; Q)›
          by (simp add: F-Seq F-After)
            (metis Cons-in-T-imp-elem-ready-set F-T list.distinct(1) list.sel(1, 3))
      next
```

assume ‹∃ *t1 t2*. *ev e # s = t1 @ t2 ∧ t1 @ [✓] ∈ 𝒯 P ∧ (t2, X) ∈ ℱ Q*›
then obtain *t1 t2* **where** ∗ : ‹*ev e # s = t1 @ t2*› ‹*t1 @ [✓] ∈ 𝒯 P*› ‹(*t2, X*) ∈ ℱ *Q*› **by** *blast*
have ‹*t1 ≠ []*› **by** (*metis ∗ Cons-in-T-imp-elem-ready-set F-T self-append-conv2 that*)
with ∗(*1*) **obtain** *t1′*
where ‹*t1 = ev e # t1′*› ‹*s = t1′ @ t2*› **by** (*metis Cons-eq-append-conv*)
with ∗(*2, 3*) **show** ‹(*s, X*) ∈ ℱ (*P after e ; Q*)›
by (*simp add*: *F-Seq T-After*)
(*metis Cons-in-T-imp-elem-ready-set list.distinct*(*1*) *list.sel*(*1, 3*))
qed
next
show ‹*ev e ∉ ready-set* (*P ; Q*) ⟹ *s = []* ⟹ (*s, X*) ∈ ℱ (*P after e ; Q*)›
by (*simp add*: *F-Seq F-After non-BOT ready-set-Seq*)
qed
next
fix *s X*
assume *same-div* : ‹𝒟 ((*P ; Q*) *after e*) = 𝒟 (*P after e ; Q*)›
assume ‹(*s, X*) ∈ ℱ (*P after e ; Q*)›
then consider ‹*s* ∈ 𝒟 (*P after e ; Q*)›
| ‹(*s, insert ✓ X*) ∈ ℱ (*P after e*)› ‹*tickFree s*›
| ‹∃ *t1 t2*. *s = t1 @ t2 ∧ t1 @ [✓] ∈ 𝒯* (*P after e*) ∧ (*t2, X*) ∈ ℱ *Q*›
by (*simp add*: *F-Seq D-Seq*) *blast*
thus ‹(*s, X*) ∈ ℱ ((*P ; Q*) *after e*)›
proof *cases*
from *same-div D-F* **show** ‹*s* ∈ 𝒟 (*P after e ; Q*) ⟹ (*s, X*) ∈ ℱ ((*P ; Q*) *after e*)› **by** *blast*
next
from *that* **show** ‹(*s, insert ✓ X*) ∈ ℱ (*P after e*) ⟹ *tickFree s* ⟹
(*s, X*) ∈ ℱ ((*P ; Q*) *after e*)›
by (*simp add*: *F-After ready-set-Seq F-Seq non-BOT split*: *if-split-asm*)
(*metis event.distinct*(*1*) *list.collapse tickFree-Cons*)
next
assume ‹∃ *t1 t2*. *s = t1 @ t2 ∧ t1 @ [✓] ∈ 𝒯* (*P after e*) ∧ (*t2, X*) ∈ ℱ *Q*›
then obtain *t1 t2*
where ∗ : ‹*s = t1 @ t2*› ‹*t1 @ [✓] ∈ 𝒯* (*P after e*)› ‹(*t2, X*) ∈ ℱ *Q*› **by** *blast*
from ∗(*2*) **have** ‹*ev e # t1 @ [✓] ∈ 𝒯 P*›
by (*simp add*: *T-After split*: *if-split-asm*) (*metis list.exhaust-sel*)
with ∗(*1, 3*) **show** ‹(*s, X*) ∈ ℱ ((*P ; Q*) *after e*)›
by (*simp add*: *F-After F-Seq ready-set-Seq non-BOT*)
(*metis Cons-in-T-imp-elem-ready-set append-Cons list.distinct*(*1*) *list.sel*(*1, 3*))
qed
qed
qed

lemma ‹(*P ; STOP*) *after e = P after e ; STOP*›

by (*simp add*: *not-skippable-or-not-readyR-After-Seq ready-set-STOP*)

lemma *skippable-not-readyL-After-Seq*: ‹(*P* **;** *Q*) *after e* = *Q after e*›
if ‹✓ ∈ *ready-set P*› **and** ‹*ev e* ∉ *ready-set P*›
proof (*cases* ‹*P* = ⊥›)
from *that*(*2*) *ready-set-BOT* **show** ‹*P* = ⊥ ⟹ (*P* **;** *Q*) *after e* = *Q after e*› **by** *blast*
next
assume *non-BOT* : ‹*P* ≠ ⊥›
show ‹(*P* **;** *Q*) *after e* = *Q after e*›
proof (*subst Process-eq-spec-optimized*, *safe*)
fix *s*
assume ‹*s* ∈ *D* ((*P* **;** *Q*) *after e*)›
hence ‹*ev e* # *s* ∈ *D* (*P* **;** *Q*)›
by (*simp add*: *D-After split*: *if-split-asm*) (*metis list.exhaust-sel*)
with *that*(*2*) **obtain** *t1 t2* **where** ∗ : ‹*ev e* # *s* = *t1* @ *t2*› ‹*t1* @ [✓] ∈ *T P*› ‹*t2* ∈ *D Q*›
by (*simp add*: *D-Seq*) (*meson Cons-in-T-imp-elem-ready-set NT-ND*)
from ∗(*1*, *2*) *that*(*2*) *Cons-in-T-imp-elem-ready-set* **have** ‹*t1* = []›
by (*cases t1*; *simp*) *blast*
with ∗ **show** ‹*s* ∈ *D* (*Q after e*)›
by (*simp add*: *D-After*)
(*metis Cons-in-T-imp-elem-ready-set D-T list.discI list.sel*(*1*, *3*))
next
show ‹*s* ∈ *D* (*Q after e*) ⟹ *s* ∈ *D* ((*P* **;** *Q*) *after e*)› **for** *s*
by (*simp add*: *D-After D-Seq ready-set-Seq non-BOT that split*: *if-split-asm*)
(*metis append-Nil mem-Collect-eq ready-set-def that*(*1*))
next
fix *s X*
assume *same-div* : ‹*D* ((*P* **;** *Q*) *after e*) = *D* (*Q after e*)›
assume ‹(*s*, *X*) ∈ *F* ((*P* **;** *Q*) *after e*)›
then consider ‹*ev e* ∈ *ready-set* (*P* **;** *Q*)› ‹(*ev e* # *s*, *X*) ∈ *F* (*P* **;** *Q*)›
| ‹*ev e* ∉ *ready-set* (*P* **;** *Q*)› ‹*s* = []›
by (*simp add*: *F-After split*: *if-split-asm*) (*metis list.exhaust-sel*)
thus ‹(*s*, *X*) ∈ *F* (*Q after e*)›
proof *cases*
show ‹*ev e* ∈ *ready-set* (*P* **;** *Q*) ⟹ (*ev e* # *s*, *X*) ∈ *F* (*P* **;** *Q*) ⟹
(*s*, *X*) ∈ *F* (*Q after e*)›
by (*simp add*: *F-After F-Seq*)
(*metis Cons-in-T-imp-elem-ready-set D-T F-T append-Nil hd-append2*
is-processT3-ST list.exhaust-sel list.sel(*1*, *3*) *that*(*2*))
next
show ‹*ev e* ∉ *ready-set* (*P* **;** *Q*) ⟹ *s* = [] ⟹ (*s*, *X*) ∈ *F* (*Q after e*)›
by (*simp add*: *F-After ready-set-Seq non-BOT that*)
qed
next
show ‹(*s*, *X*) ∈ *F* (*Q after e*) ⟹ (*s*, *X*) ∈ *F* ((*P* **;** *Q*) *after e*)› **for** *s X*
by (*simp add*: *F-After F-Seq ready-set-Seq non-BOT that split*: *if-split-asm*)

```
      (metis append-Nil mem-Collect-eq ready-set-def that(1))
  qed
qed


lemma skippable-readyL-readyR-After-Seq: ‹(P ; Q) after e = (P after e ; Q) ⊓
(Q after e)›
  if ‹✓ ∈ ready-set P› ‹ev e ∈ ready-set P› ‹ev e ∈ ready-set Q›
proof (cases ‹P = ⊥›)
  show ‹P = ⊥ ⟹ (P ; Q) after e = (P after e ; Q) ⊓ (Q after e)›
   by (simp add: BOT-Seq After-BOT Ndet-commute[of ⊥, simplified Ndet-BOT])
next
  assume non-BOT : ‹P ≠ ⊥›
  show ‹(P ; Q) after e = (P after e ; Q) ⊓ (Q after e)›
  proof (subst Process-eq-spec-optimized, safe)
    fix s
    assume ‹s ∈ 𝒟 ((P ; Q) after e)›
    hence ‹ev e # s ∈ 𝒟 (P ; Q)›
     by (simp add: D-After ready-set-Seq non-BOT that(2)) (metis list.exhaust-sel)
    then consider ‹ev e # s ∈ 𝒟 P›
     | ‹∃ t1 t2. ev e # s = t1 @ t2 ∧ t1 @ [✓] ∈ 𝒯 P ∧ t2 ∈ 𝒟 Q›
     by (simp add: D-Seq) blast
    thus ‹s ∈ 𝒟 ((P after e ; Q) ⊓ (Q after e))›
    proof cases
      show ‹ev e # s ∈ 𝒟 P ⟹ s ∈ 𝒟 ((P after e ; Q) ⊓ (Q after e))›
        by (simp add: D-After D-Seq D-Ndet non-BOT that)
           (metis list.distinct(1) list.sel(1, 3))
    next
      assume ‹∃ t1 t2. ev e # s = t1 @ t2 ∧ t1 @ [✓] ∈ 𝒯 P ∧ t2 ∈ 𝒟 Q›
      then obtain t1 t2 where ∗ : ‹ev e # s = t1 @ t2› ‹t1 @ [✓] ∈ 𝒯 P› ‹t2 ∈
𝒟 Q› by blast
      show ‹s ∈ 𝒟 ((P after e ; Q) ⊓ (Q after e))›
      proof (cases ‹t1 = []›)
        from ∗(1, 3) show ‹t1 = [] ⟹ s ∈ 𝒟 ((P after e ; Q) ⊓ (Q after e))›
          by (simp add: D-After D-Ndet that(3))
             (metis list.distinct(1) list.sel(1, 3))
      next
        assume ‹t1 ≠ []›
        with ∗(1, 3) obtain t1' where ‹t1 = ev e # t1'› by (cases t1; simp)
        with ∗ show ‹s ∈ 𝒟 ((P after e ; Q) ⊓ (Q after e))›
          by (simp add: T-After D-Seq D-Ndet that(2))
             (metis list.distinct(1) list.sel(1, 3))
      qed
    qed
  next
    fix s
    assume ‹s ∈ 𝒟 ((P after e ; Q) ⊓ (Q after e))›
    then consider ‹s ∈ 𝒟 (P after e ; Q)› | ‹s ∈ 𝒟 (Q after e)›
      by (simp add: D-Ndet) blast
```

```
  thus ‹s ∈ 𝒟 ((P ; Q) after e)›
  proof cases
    show ‹s ∈ 𝒟 (P after e ; Q) ⟹ s ∈ 𝒟 ((P ; Q) after e)›
      apply (simp add: D-After T-After D-Seq ready-set-Seq non-BOT that(2),
elim disjE)
      by blast (metis append-Cons list.distinct(1) list.exhaust-sel list.sel(1, 3))
  next
    from that show ‹s ∈ 𝒟 (Q after e) ⟹ s ∈ 𝒟 ((P ; Q) after e)›
      by (simp add: D-After D-Seq ready-set-Seq non-BOT)
        (metis append-Nil mem-Collect-eq ready-set-def)
  qed
 next
  fix s X
  assume same-div : ‹𝒟 ((P ; Q) after e) = 𝒟 ((P after e ; Q) ⊓ (Q after e))›
  assume ‹(s, X) ∈ ℱ ((P ; Q) after e)›
  hence ‹(ev e # s, X) ∈ ℱ (P ; Q)›
   by (simp add: F-After ready-set-Seq non-BOT that(2)) (metis list.exhaust-sel)
  then consider ‹ev e # s ∈ 𝒟 P›
    | ‹(ev e # s, insert ✓ X) ∈ ℱ P› ‹tickFree s›
    | ‹∃t1 t2. ev e # s = t1 @ t2 ∧ t1 @ [✓] ∈ 𝒯 P ∧ (t2, X) ∈ ℱ Q›
    by (simp add: F-Seq D-Seq) blast
  thus ‹(s, X) ∈ ℱ ((P after e ; Q) ⊓ (Q after e))›
  proof cases
    assume ‹ev e # s ∈ 𝒟 P›
    hence ‹s ∈ 𝒟 (P after e ; Q)›
      by (simp add: D-After D-Seq that(2)) (metis list.distinct(1) list.sel(1, 3))
     with same-div D-Ndet D-F show ‹(s, X) ∈ ℱ ((P after e ; Q) ⊓ (Q after
e))› by blast
  next
    show ‹(ev e # s, insert ✓ X) ∈ ℱ P ⟹ tickFree s ⟹
          (s, X) ∈ ℱ ((P after e ; Q) ⊓ (Q after e))›
      by (simp add: F-Ndet F-Seq F-After that(2))
        (metis list.distinct(1) list.sel(1, 3))
  next
    assume ‹∃t1 t2. ev e # s = t1 @ t2 ∧ t1 @ [✓] ∈ 𝒯 P ∧ (t2, X) ∈ ℱ Q›
    then obtain t1 t2
     where ∗ : ‹ev e # s = t1 @ t2› ‹t1 @ [✓] ∈ 𝒯 P› ‹(t2, X) ∈ ℱ Q› by blast
    show ‹(s, X) ∈ ℱ ((P after e ; Q) ⊓ (Q after e))›
    proof (cases ‹t1 = []›)
     from ∗(1, 3) show ‹t1 = [] ⟹ (s, X) ∈ ℱ ((P after e ; Q) ⊓ (Q after e))›
       by (simp add: F-Ndet F-Seq F-After that(2, 3))
         (metis list.distinct(1) list.sel(1, 3))
    next
     assume ‹t1 ≠ []›
     with ∗(1, 3) obtain t1′ where ‹t1 = ev e # t1′› by (cases t1; simp)
     with ∗ show ‹(s, X) ∈ ℱ ((P after e ; Q) ⊓ (Q after e))›
       by (simp add: F-Ndet F-Seq F-After T-After that(2))
         (metis list.distinct(1) list.sel(1, 3))
    qed
```

```
    qed
  next
    fix s X
    assume same-div : ‹𝒟 ((P ; Q) after e) = 𝒟 ((P after e ; Q) ⊓ (Q after e))›
    assume ‹(s, X) ∈ ℱ ((P after e ; Q) ⊓ (Q after e))›
    then consider ‹(s, X) ∈ ℱ (P after e ; Q)› | ‹(s, X) ∈ ℱ (Q after e)›
      by (simp add: F-Ndet) blast
    thus ‹(s, X) ∈ ℱ ((P ; Q) after e)›
    proof cases
      assume ‹(s, X) ∈ ℱ (P after e ; Q)›
      then consider ‹s ∈ 𝒟 (P after e ; Q)›
        | ‹(s, insert ✓ X) ∈ ℱ (P after e)› ‹tickFree s›
        | ‹∃ t1 t2. s = t1 @ t2 ∧ t1 @ [✓] ∈ 𝒯 (P after e) ∧ (t2, X) ∈ ℱ Q›
        by (simp add: F-Seq D-Seq) blast
      thus ‹(s, X) ∈ ℱ ((P ; Q) after e)›
      proof cases
        show ‹s ∈ 𝒟 (P after e ; Q) ⟹ (s, X) ∈ ℱ ((P ; Q) after e)›
          using same-div D-Ndet D-F by blast
      next
        show ‹(s, insert ✓ X) ∈ ℱ (P after e) ⟹ tickFree s ⟹ (s, X) ∈ ℱ ((P ;
Q) after e)›
          by (simp add: F-After F-Seq ready-set-Seq that(2))
            (metis event.distinct(1) list.collapse tickFree-Cons)
      next
        show ‹∃ t1 t2. s = t1 @ t2 ∧ t1 @ [✓] ∈ 𝒯 (P after e) ∧ (t2, X) ∈ ℱ Q
⟹
                    (s, X) ∈ ℱ ((P ; Q) after e)›
          by (simp add: F-After T-After F-Seq ready-set-Seq non-BOT that(2))
            (metis append-Cons list.distinct(1) list.exhaust-sel list.sel(1, 3))
      qed
    next
      show ‹(s, X) ∈ ℱ (Q after e) ⟹ (s, X) ∈ ℱ ((P ; Q) after e)›
        by (simp add: F-After F-Seq ready-set-Seq that(3))
          (metis append-Nil mem-Collect-eq ready-set-def that(1))
    qed
  qed
qed


lemma not-readyL-not-skippable-or-not-readyR-After-Seq:
  ‹ev e ∉ ready-set P ⟹ ✓ ∉ ready-set P ∨ ev e ∉ ready-set Q ⟹
   (P ; Q) after e = STOP›
  by (simp add: not-skippable-or-not-readyR-After-Seq STOP-Seq not-ready-After)


lemma After-Seq:
  ‹(P ; Q) after e =
   (     if ev e ∉ ready-set P ∧ ev e ∉ ready-set Q then STOP
    else if ev e ∉ ready-set Q then P after e ; Q
```

else if ev e ∉ ready-set P then if ✓ ∈ ready-set P then Q after e else STOP
else if ✓ ∈ ready-set P then (P after e ; Q) ⊓ (Q after e) else P after e ; Q)›
by (*simp add*: *STOP-Seq not-ready-After not-skippable-or-not-readyR-After-Seq*
skippable-not-readyL-After-Seq skippable-readyL-readyR-After-Seq)

4.5.3 *After* Synchronization

Now let's focus on *Sync*. We want to obtain an equivalent of
⟦*?A* ∩ *?S* = {}; *?A′* ⊆ *?S*; *?B* ∩ *?S* = {}; *?B′* ⊆ *?S*⟧ ⟹ *Mprefix* (*?A* ∪ *?A′*) *?P* ⟦*?S*⟧ *Mprefix* (*?B* ∪ *?B′*) *?Q* = (□*x*∈*?A* → *?P x* ⟦*?S*⟧ *Mprefix* (*?B* ∪ *?B′*) *?Q*) □ (□*y*∈*?B* → *Mprefix* (*?A* ∪ *?A′*) *?P* ⟦*?S*⟧ *?Q y*) □ (□*x*∈*?A′* ∩ *?B′* → *?P x* ⟦*?S*⟧ *?Q x*)
We will also divide the task.

After version of
⟦*e* ∉ *?S*; *?B′* ⊆ *?S*⟧ ⟹ *e* → *P* ⟦*?S*⟧ *Mprefix ?B′ ?Q* = *e* → (*P* ⟦*?S*⟧ *Mprefix ?B′ ?Q*).

lemma *tickFree-tl*: ‹*tickFree s* ⟹ *tickFree*(*tl s*)›
— Remove this lemma, already in future versions of `HOL-CSP`.
by (*metis Nil-tl tickFree-tl*)

lemma *readyL-not-readyR-not-in-After-Sync*:
‹(*P* ⟦*S*⟧ *Q*) *after e* = *P after e* ⟦*S*⟧ *Q*›
if *ready-hyps*: ‹*ev e* ∈ *ready-set P*› ‹*ev e* ∉ *ready-set Q*› **and** *notin*: ‹*e* ∉ *S*›
proof (*subst Process-eq-spec-optimized*, *safe*)

{ fix *s X*
assume *assms* : ‹*P* ≠ ⊥› ‹*Q* ≠ ⊥› ‹(*ev e* # *s*, *X*) ∈ ℱ (*P* ⟦*S*⟧ *Q*)›
and *same-div* : ‹𝒟 ((*P* ⟦*S*⟧ *Q*) *after e*) = 𝒟 (*P after e* ⟦*S*⟧ *Q*)›
have ‹(*s*, *X*) ∈ ℱ (*P after e* ⟦*S*⟧ *Q*)›
proof (*cases* ‹*ev e* # *s* ∈ 𝒟 (*P* ⟦*S*⟧ *Q*)›)
case *True*
hence ‹*s* ∈ 𝒟 ((*P* ⟦*S*⟧ *Q*) *after e*)›
by (*force simp add*: *D-After ready-set-Sync ready-hyps*(*1*) *assms*(*1*, *2*) *notin*)
thus ‹(*s*, *X*) ∈ ℱ (*P after e* ⟦*S*⟧ *Q*)› **using** *NF-ND same-div* **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 e* # *s*) *setinterleaves* ((*s-P*, *s-Q*), *insert* ✓ (*ev* ‘ *S*))›
‹*X* = (*X-P* ∪ *X-Q*) ∩ *insert* ✓ (*ev* ‘ *S*) ∪ *X-P* ∩ *X-Q*›
by (*simp add*: *F-Sync D-Sync*) *blast*
have ∗∗ : ‹*s-P* ≠ [] ∧ *hd s-P* = *ev e* ∧ *s setinterleaves* ((*tl s-P*, *s-Q*), *insert* ✓ (*ev* ‘ *S*))›
using ∗(*3*) **by** (*cases s-P*; *cases s-Q*, *auto split*: *if-split-asm*)
(*metis* ∗(*2*) *After-is-STOP-iff CollectI F-T not-ready-After*
ready-set-def ready-hyps(*2*))+
hence ‹(*tl s-P*, *X-P*) ∈ ℱ (*P after e*) ∧ (*s-Q*, *X-Q*) ∈ ℱ *Q* ∧

```
        s setinterleaves ((tl s-P, s-Q), insert ✓ (ev ‘ S)) ∧
        X = (X-P ∪ X-Q) ∩ insert ✓ (ev ‘ S) ∪ X-P ∩ X-Q›
    apply (simp add: F-After ∗∗ ready-hyps(1))
    using ∗(1, 2, 4) ∗∗ by blast
  thus ‹(s, X) ∈ ℱ (P after e ⟦S⟧ Q)›
    by (simp add: F-Sync) blast
 qed
} note ∗ = this

show ‹(s, X) ∈ ℱ ((P ⟦S⟧ Q) after e) ⟹ (s, X) ∈ ℱ (P after e ⟦S⟧ Q)›
  if same-div : ‹𝒟 ((P ⟦S⟧ Q) after e) = 𝒟 (P after e ⟦S⟧ Q)› for s X
 apply (simp add: F-After ready-set-Sync ready-hyps notin image-iff
        split: if-split-asm)
  apply (erule disjE;
    simp add: After-BOT Sync-commute[of ⊥, simplified Sync-BOT] Sync-BOT
F-UU,
      metis butlast-tl front-tickFree-butlast tickFree-tl)
 by (metis ∗ list.exhaust-sel same-div)
next

fix s X
assume same-div : ‹𝒟 ((P ⟦S⟧ Q) after e) = 𝒟 (P after e ⟦S⟧ Q)›
{ assume assms : ‹P ≠ ⊥› ‹Q ≠ ⊥› ‹(s, X) ∈ ℱ (P after e ⟦S⟧ Q)›
 from assms(3) consider
   ‹∃ s-P s-Q X-P X-Q. (s-P, X-P) ∈ ℱ (P after e) ∧ (s-Q, X-Q) ∈ ℱ Q ∧
                  s setinterleaves ((s-P, s-Q), insert ✓ (ev ‘ S)) ∧
                  X = (X-P ∪ X-Q) ∩ insert ✓ (ev ‘ S) ∪ X-P ∩ X-Q› |
   ‹s ∈ 𝒟 (P after e ⟦S⟧ Q)›
   by (simp add: F-Sync D-Sync) blast
 hence ‹(ev e # s, X) ∈ ℱ (P ⟦S⟧ Q)›
 proof cases
   case 1
   then obtain s-P s-Q X-P X-Q
     where ∗ : ‹(ev e # s-P, X-P) ∈ ℱ P› ‹(s-Q, X-Q) ∈ ℱ Q›
           ‹s setinterleaves ((s-P, s-Q), insert ✓ (ev ‘ S))›
           ‹X = (X-P ∪ X-Q) ∩ insert ✓ (ev ‘ S) ∪ X-P ∩ X-Q›
     by (simp add: F-After ready-hyps(1)) (metis list.collapse)
   have ‹(s-Q, X-Q) ∈ ℱ Q ∧
       (ev e # s) setinterleaves ((ev e # s-P, s-Q), insert ✓ (ev ‘ S)) ∧
        X = (X-P ∪ X-Q) ∩ insert ✓ (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 e # s, X) ∈ ℱ (P ⟦S⟧ Q)›
     by (simp add: F-Sync) blast
 next
   case 2
   from 2[simplified same-div[symmetric]]
   have ‹ev e # s ∈ 𝒟 (P ⟦S⟧ Q)›
     by (simp add: D-After ready-hyps(1) ready-set-Sync
```

```
                assms(1, 2) notin image-iff, metis list.collapse)
    thus ‹(ev e # s, X) ∈ ℱ (P ⟦S⟧ Q)› using NF-ND by blast
   qed
  }
  thus ‹(s, X) ∈ ℱ (P after e ⟦S⟧ Q) ⟹ (s, X) ∈ ℱ ((P ⟦S⟧ Q) after e)›
   by (simp add: F-After ready-set-Sync ready-hyps After-BOT F-UU image-iff
                 Sync-commute[of ⊥, simplified Sync-BOT] Sync-BOT notin)
    (metis butlast.simps(2) event.distinct(1) front-tickFree-butlast front-tickFree-single
           list.distinct(1) list.sel(1, 3) process-charn tickFree-Cons)
next

  { fix s
   assume assms : ‹P ≠ ⊥› ‹Q ≠ ⊥› ‹ev e # s ∈ 𝒟 (P ⟦S⟧ Q)›
   from assms(3) obtain t u r v
    where ∗ : ‹front-tickFree v› ‹tickFree r ∨ v = []› ‹ev e # s = r @ v›
          ‹r setinterleaves ((t, u), insert ✓ (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 e›
    by (metis ∗(3, 4, 5) BOT-iff-D assms(1, 2) empty-setinterleaving hd-append
list.sel(1))
   hence ∗∗∗ : ‹(t ∈ 𝒟 P ∧ u ∈ 𝒯 Q ⟶ t ≠ [] ∧ hd t = ev e ∧
                          tl r setinterleaves ((tl t, u), insert ✓ (ev ‘ S))) ∧
                (t ∈ 𝒟 Q ∧ u ∈ 𝒯 P ⟶ u ≠ [] ∧ hd u = ev e ∧
                          tl r setinterleaves ((t, tl u), insert ✓ (ev ‘ S)))›
   using ∗(4) assms(1, 2)[simplified BOT-iff-D] ready-hyps[simplified ready-set-def]
    apply (cases t; cases u; simp split: if-split-asm)
    by (safe; simp; metis [[metis-verbose = false]] After-is-STOP-iff D-T
                          not-ready-After ready-hyps(2))+
   from ∗(5) have ‹s ∈ 𝒟 (P after e ⟦S⟧ Q)›
   proof (elim disjE)
    assume ‹t ∈ 𝒟 P ∧ u ∈ 𝒯 Q›
    hence ‹front-tickFree v ∧ (tickFree (tl r) ∨ v = []) ∧ s = tl r @ v ∧
          tl r setinterleaves ((tl t, u), insert ✓ (ev ‘ S)) ∧
          tl t ∈ 𝒟 (P after e) ∧ u ∈ 𝒯 Q›
     by (simp add: D-After ready-hyps,
         metis ∗(1, 2, 3) ∗∗ ∗∗∗ list.sel(3) tickFree-tl tl-append2)
    thus ‹s ∈ 𝒟 (P after e ⟦S⟧ Q)› by (simp add: D-Sync) blast
   next
    assume ‹t ∈ 𝒟 Q ∧ u ∈ 𝒯 P›
    hence ‹front-tickFree v ∧ (tickFree (tl r) ∨ v = []) ∧ s = tl r @ v ∧
          tl r setinterleaves ((t, tl u), insert ✓ (ev ‘ S)) ∧
          t ∈ 𝒟 Q ∧ tl u ∈ 𝒯 (P after e)›
     by (simp add: D-After T-After ready-hyps,
         metis ∗(1, 2, 3) ∗∗ ∗∗∗ list.sel(3) tickFree-tl tl-append2)
    thus ‹s ∈ 𝒟 (P after e ⟦S⟧ Q)›
     by (simp add: D-Sync) blast
   qed
  } note ∗ = this
```

```
  show ‹s ∈ 𝒟 ((P ⟦S⟧ Q) after e) ⟹ s ∈ 𝒟 (P after e ⟦S⟧ Q)› for s
    apply (simp add: D-After ready-set-Sync ready-hyps notin image-iff
            split: if-split-asm)
     apply (erule disjE;
        simp add: After-BOT Sync-commute[of ⊥, simplified Sync-BOT] Sync-BOT
D-UU,
           metis butlast-tl front-tickFree-butlast tickFree-tl)
    by (metis ∗ list.collapse)
next

  fix s
  { assume assms : ‹P ≠ ⊥› ‹Q ≠ ⊥› ‹s ∈ 𝒟 (P after e ⟦S⟧ Q)›
    from assms(3) obtain t u r v
      where ∗ : ‹front-tickFree v› ‹tickFree r ∨ v = []› ‹s = r @ v›
                ‹r setinterleaves ((t, u), insert ✓ (ev ' S))›
                ‹t ∈ 𝒟 (P after e) ∧ u ∈ 𝒯 Q ∨ t ∈ 𝒟 Q ∧ u ∈ 𝒯 (P after e)›
      by (simp add: D-Sync) blast
    from ∗(5) have ‹ev e # s ∈ 𝒟 (P ⟦S⟧ Q)›
    proof (elim disjE)
      assume ‹t ∈ 𝒟 (P after e) ∧ u ∈ 𝒯 Q›
      with ∗(1, 2, 3, 4) ready-hyps(1)
      have ∗∗ : ‹front-tickFree v ∧ (tickFree (ev e # r) ∨ v = []) ∧ s = r @ v ∧
                (ev e # r) setinterleaves ((ev e # t, u), insert ✓ (ev ' S)) ∧
                ev e # t ∈ 𝒟 P ∧ u ∈ 𝒯 Q›
        by (cases u; simp add: D-After notin image-iff, metis list.collapse)
      show ‹ev e # s ∈ 𝒟 (P ⟦S⟧ Q)›
        by (simp add: D-Sync) (metis ∗∗ Cons-eq-appendI)
    next
      assume ‹t ∈ 𝒟 Q ∧ u ∈ 𝒯 (P after e)›
      with ∗(1, 2, 3, 4) ready-hyps(1)
      have ∗∗ : ‹front-tickFree v ∧ (tickFree (ev e # r) ∨ v = []) ∧ s = r @ v ∧
                (ev e # r) setinterleaves ((t, ev e # u), insert ✓ (ev ' S)) ∧
                t ∈ 𝒟 Q ∧ ev e # u ∈ 𝒯 P›
        by (cases t; simp add: T-After notin image-iff, metis list.collapse)
      show ‹ev e # s ∈ 𝒟 (P ⟦S⟧ Q)›
        by (simp add: D-Sync) (metis ∗∗ Cons-eq-appendI)
    qed
  }
  thus ‹s ∈ 𝒟 (P after e ⟦S⟧ Q) ⟹ s ∈ 𝒟 ((P ⟦S⟧ Q) after e)›
    by (simp add: D-After ready-set-Sync ready-hyps After-BOT D-UU
                  Sync-commute[of ⊥, simplified Sync-BOT] Sync-BOT notin
image-iff)
     (metis D-imp-front-tickFree butlast.simps(2) event.distinct(1) front-tickFree-butlast

          list.discI list.sel(1, 3) tickFree-Cons tickFree-butlast)
qed
```

lemma *not-readyL-in-After-Sync*:
‹*ev e* ∉ *ready-set P* ⟹ *e* ∈ *S* ⟹
(*P* ⟦*S*⟧ *Q*) *after e* = (*if Q* = ⊥ *then* ⊥ *else STOP*)›
apply (*simp*, *intro conjI impI*)
by (*simp add*: *BOT-iff-D D-After Sync-BOT ready-set-BOT D-UU*,
metis front-tickFree-single list.sel(*1*) *list.sel*(*3*) *not-Cons-self*)
(*auto simp add*: *STOP-iff-T T-After ready-set-BOT ready-set-Sync*)

After version of ⟦*e* ∈ *?S*; *e* ∈ *?S*⟧ ⟹ *e* → *P* ⟦*?S*⟧ *e* → *Q* = *e* → (*P* ⟦*?S*⟧ *Q*).

lemma *readyL-readyR-in-After-Sync*:
‹(*P* ⟦*S*⟧ *Q*) *after e* = *P after e* ⟦*S*⟧ *Q after e*›
if *ready-hyps*: ‹*ev e* ∈ *ready-set P*› ‹*ev e* ∈ *ready-set Q*› **and** *inside*: ‹*e* ∈ *S*›
proof (*subst Process-eq-spec-optimized*, *safe*)

{ fix *s X*
assume *assms* : ‹*P* ≠ ⊥› ‹*Q* ≠ ⊥› ‹(*ev e* # *s*, *X*) ∈ ℱ (*P* ⟦*S*⟧ *Q*)›
and *same-div* : ‹𝒟 ((*P* ⟦*S*⟧ *Q*) *after e*) = 𝒟 (*P after e* ⟦*S*⟧ *Q after e*)›
from *assms*(*3*) **consider**
‹∃ *s-P s-Q X-P X-Q*. (*s-P*, *X-P*) ∈ ℱ *P* ∧ (*s-Q*, *X-Q*) ∈ ℱ *Q* ∧
(*ev e* # *s*) *setinterleaves* ((*s-P*, *s-Q*), *insert* ✓ (*ev* ‘ *S*)) ∧
X = (*X-P* ∪ *X-Q*) ∩ *insert* ✓ (*ev* ‘ *S*) ∪ *X-P* ∩ *X-Q*› |
‹*ev e* # *s* ∈ 𝒟 (*P* ⟦*S*⟧ *Q*)›
by (*simp add*: *F-Sync D-Sync*) *blast*
hence ‹(*s*, *X*) ∈ ℱ (*P after e* ⟦*S*⟧ *Q after e*)›
proof *cases*
case *1*
then obtain *s-P s-Q X-P X-Q*
where ∗ : ‹(*s-P*, *X-P*) ∈ ℱ *P*› ‹(*s-Q*, *X-Q*) ∈ ℱ *Q*›
‹(*ev e* # *s*) *setinterleaves* ((*s-P*, *s-Q*), *insert* ✓ (*ev* ‘ *S*))›
‹*X* = (*X-P* ∪ *X-Q*) ∩ *insert* ✓ (*ev* ‘ *S*) ∪ *X-P* ∩ *X-Q*› **by** *blast*
from ∗(*3*) **have** ‹*s-P* ≠ [] ∧ *hd s-P* = *ev e* ∧ *s-Q* ≠ [] ∧ *hd s-Q* = *ev e* ∧
s setinterleaves ((*tl s-P*, *tl s-Q*), *insert* ✓ (*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 e*) ∧ (*tl s-Q*, *X-Q*) ∈ ℱ (*Q after e*) ∧
s setinterleaves ((*tl s-P*, *tl s-Q*), *insert* ✓ (*ev* ‘ *S*)) ∧
X = (*X-P* ∪ *X-Q*) ∩ *insert* ✓ (*ev* ‘ *S*) ∪ *X-P* ∩ *X-Q*›
using ∗(*1*, *2*, *4*) *ready-hyps* **by** (*simp add*: *F-After*) *blast*
thus ‹(*s*, *X*) ∈ ℱ (*P after e* ⟦*S*⟧ *Q after e*)›
by (*simp add*: *F-Sync*) *blast*
next
assume ‹*ev e* # *s* ∈ 𝒟 (*P* ⟦*S*⟧ *Q*)›
hence ‹*s* ∈ 𝒟 ((*P* ⟦*S*⟧ *Q*) *after e*)›
by (*force simp add*: *D-After ready-set-Sync ready-hyps assms*(*1*, *2*))
from *this*[*simplified same-div*]
show ‹(*s*, *X*) ∈ ℱ (*P after e* ⟦*S*⟧ *Q after e*)› **using** *NF-ND* **by** *blast*
qed
} note ∗ = *this*

```
  show ‹(s, X) ∈ ℱ ((P ⟦S⟧ Q) after e) ⟹ (s, X) ∈ ℱ (P after e ⟦S⟧ Q after e)›
    if same-div : ‹𝒟 ((P ⟦S⟧ Q) after e) = 𝒟 (P after e ⟦S⟧ Q after e)› for s X
    apply (simp add: F-After ready-set-Sync ready-hyps split: if-split-asm)
      apply (erule disjE;
                simp add: After-BOT Sync-commute[of ⊥, simplified Sync-BOT]
Sync-BOT F-UU,
            metis butlast-tl front-tickFree-butlast tickFree-tl)
    by (metis ∗ list.exhaust-sel same-div)
next

  fix s X
  assume same-div : ‹𝒟 ((P ⟦S⟧ Q) after e) = 𝒟 (P after e ⟦S⟧ Q after e)›
  { assume assms : ‹P ≠ ⊥› ‹Q ≠ ⊥› ‹(s, X) ∈ ℱ (P after e ⟦S⟧ Q after e)›
    from assms(3) consider
      ‹∃ s-P s-Q X-P X-Q. (ev e # s-P, X-P) ∈ ℱ P ∧ (ev e # s-Q, X-Q) ∈ ℱ Q
∧
                        s setinterleaves ((s-P, s-Q), insert ✓ (ev ‘ S)) ∧
                        X = (X-P ∪ X-Q) ∩ insert ✓ (ev ‘ S) ∪ X-P ∩ X-Q› |
      ‹s ∈ 𝒟 (P after e ⟦S⟧ Q after e)›
      by (auto simp add: F-Sync D-Sync F-After ready-hyps) (metis list.collapse)
    hence ‹(ev e # s, X) ∈ ℱ (P ⟦S⟧ Q)›
    proof cases
      case 1
      then obtain s-P s-Q X-P X-Q
        where ∗ : ‹(ev e # s-P, X-P) ∈ ℱ P› ‹(ev e # s-Q, X-Q) ∈ ℱ Q›
              ‹s setinterleaves ((s-P, s-Q), insert ✓ (ev ‘ S))›
              ‹X = (X-P ∪ X-Q) ∩ insert ✓ (ev ‘ S) ∪ X-P ∩ X-Q› by blast
      have ∗∗ : ‹(ev e # s) setinterleaves ((ev e # s-P, ev e # s-Q), insert ✓ (ev ‘
S))›
        by (simp add: inside ∗(3))
      show ‹(ev e # s, X) ∈ ℱ (P ⟦S⟧ Q)›
        apply (simp add: F-Sync)
        using ∗(1, 2, 4) ∗∗ by blast
    next
      assume ‹s ∈ 𝒟 (P after e ⟦S⟧ Q after e)›
      with same-div[symmetric] have ‹ev e # s ∈ 𝒟 (P ⟦S⟧ Q)›
          by (simp add: D-After ready-hyps ready-set-Sync assms(1, 2)) (metis
list.collapse)
      with D-F show ‹(ev e # s, X) ∈ ℱ (P ⟦S⟧ Q)› by blast
    qed
  }
  thus ‹(s, X) ∈ ℱ (P after e ⟦S⟧ Q after e) ⟹ (s, X) ∈ ℱ ((P ⟦S⟧ Q) after e)›
    by (simp add: F-After ready-set-Sync ready-hyps After-BOT F-UU
                Sync-commute[of ⊥, simplified Sync-BOT] Sync-BOT)
       (metis butlast.simps(2) event.distinct(1) front-tickFree-butlast is-processT2
            list.distinct(1) list.sel(1, 3) tickFree-Cons tickFree-butlast)
next
```

```
  { fix s
    assume assms : ‹P ≠ ⊥› ‹Q ≠ ⊥› ‹ev e # s ∈ 𝒟 (P ⟦S⟧ Q)›
    from assms(3) obtain t u r v
      where * : ‹front-tickFree v› ‹tickFree r ∨ v = []› ‹ev e # s = r @ v›
                ‹r setinterleaves ((t, u), insert ✓ (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 e ∧ t ≠ [] ∧ hd t = ev e ∧ u ≠ [] ∧ hd u = ev e
∧
                tl r setinterleaves ((tl t, tl u), insert ✓ (ev ' S))›
      using *(3, 4, 5) inside assms(1, 2)[simplified BOT-iff-D]
      by (cases t; cases u; force split: if-split-asm)
    have ‹(tickFree (tl r) ∨ v = []) ∧ s = tl r @ v ∧
          (tl t ∈ 𝒟 (P after e) ∧ tl u ∈ 𝒯 (Q after e) ∨
           tl t ∈ 𝒟 (Q after e) ∧ tl u ∈ 𝒯 (P after e))›
      using *(2, 3, 5) ** apply (simp add: D-After ready-hyps T-After)
      by (metis tickFree-tl list.sel(3) tl-append2)
    with *(1) ** have ‹s ∈ 𝒟 (P after e ⟦S⟧ Q after e)› by (simp add: D-Sync)
blast
  } note * = this

  show ‹s ∈ 𝒟 ((P ⟦S⟧ Q) after e) ⟹ s ∈ 𝒟 (P after e ⟦S⟧ Q after e)› for s
    apply (simp add: D-After ready-set-Sync ready-hyps After-BOT D-UU
                split: if-split-asm)
     apply (erule disjE;
         simp add: After-BOT Sync-commute[of ⊥, simplified Sync-BOT] Sync-BOT
D-UU,
           metis butlast-tl front-tickFree-butlast tickFree-tl)
    by (metis * list.exhaust-sel)
next

  fix s
  { assume ‹s ∈ 𝒟 (P after e ⟦S⟧ Q after e)›
    from this obtain t u r v
      where * : ‹front-tickFree v› ‹tickFree r ∨ v = []› ‹s = r @ v›
                ‹r setinterleaves ((t, u), insert ✓ (ev ' S))›
                ‹ev e # t ∈ 𝒟 P ∧ ev e # u ∈ 𝒯 Q ∨ ev e # t ∈ 𝒟 Q ∧ ev e # u ∈
𝒯 P›
      by (simp add: D-Sync D-After T-After ready-hyps) (metis list.collapse)
    have ** : ‹(ev e # r) setinterleaves ((ev e # t, ev e # u), insert ✓ (ev ' S))›
      by (simp add: inside *(4))
    have ‹ev e # s ∈ 𝒟 (P ⟦S⟧ Q)›
      by (simp add: D-Sync inside)
        (metis *(1, 2, 3, 5) ** append-Cons event.distinct(1) tickFree-Cons)
  }
  thus ‹s ∈ 𝒟 (P after e ⟦S⟧ Q after e) ⟹ s ∈ 𝒟 ((P ⟦S⟧ Q) after e)›
    by (simp add: D-After ready-set-Sync ready-hyps After-BOT D-UU
                  Sync-commute[of ⊥, simplified Sync-BOT] Sync-BOT inside)
      (metis D-imp-front-tickFree list.distinct(1) list.sel(1, 3))
```

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 *readyL-readyR-not-in-After-Sync*:
‹(*P* ⟦*S*⟧ *Q*) *after e* = (*P after e* ⟦*S*⟧ *Q*) ⊓ (*P* ⟦*S*⟧ *Q after e*)›
if *ready-hyps*: ‹*ev e* ∈ *ready-set P*› ‹*ev e* ∈ *ready-set Q*› **and** *notin*: ‹*e* ∉ *S*›
proof (*subst Process-eq-spec-optimized*, *safe*)

{ fix *P Q s X s-P s-Q X-P X-Q*
assume *assms* : ‹(*s-P*, *X-P*) ∈ ℱ *P*› ‹(*s-Q*, *X-Q*) ∈ ℱ *Q*›
‹*X* = (*X-P* ∪ *X-Q*) ∩ *insert* ✓ (*ev* ‘ *S*) ∪ *X-P* ∩ *X-Q*›
‹*s-P* ≠ []› ‹*hd s-P* = *ev e*›
‹*s setinterleaves* ((*tl s-P*, *s-Q*), *insert* ✓ (*ev* ‘ *S*))›
‹*ev e* ∈ *ready-set P*›
from *assms*(*1*, *4*, *5*, *7*) **have** ‹(*tl s-P*, *X-P*) ∈ ℱ (*P after e*)›
by (*simp add*: *F-After*) *blast*
with *assms*(*2*, *3*, *6*) **have** ‹(*s*, *X*) ∈ ℱ (*P after e* ⟦*S*⟧ *Q*)›
by (*simp add*: *F-Sync*) *blast*
} note ∗ = *this*

{ fix *s X*
assume *assms* : ‹*P* ≠ ⊥› ‹*Q* ≠ ⊥› ‹(*ev e* # *s*, *X*) ∈ ℱ (*P* ⟦*S*⟧ *Q*)›
and *same-div* : ‹𝒟 ((*P* ⟦*S*⟧ *Q*) *after e*) = 𝒟 ((*P after e* ⟦*S*⟧ *Q*) ⊓ (*P* ⟦*S*⟧ *Q after e*))›
from *assms*(*3*) **consider**
‹∃ *s-P s-Q X-P X-Q*. (*s-P*, *X-P*) ∈ ℱ *P* ∧ (*s-Q*, *X-Q*) ∈ ℱ *Q* ∧
(*ev e* # *s*) *setinterleaves* ((*s-P*, *s-Q*), *insert* ✓ (*ev* ‘ *S*)) ∧
X = (*X-P* ∪ *X-Q*) ∩ *insert* ✓ (*ev* ‘ *S*) ∪ *X-P* ∩ *X-Q*› |
‹*s* ∈ 𝒟 ((*P* ⟦*S*⟧ *Q*) *after e*)›
by (*simp add*: *F-Sync D-After D-Sync ready-set-Sync assms*(*1*, *2*) *ready-hyps*)
(*metis* (*no-types*, *opaque-lifting*) *list.distinct*(*1*) *list.sel*(*1*, *3*))
hence ‹(*s*, *X*) ∈ ℱ ((*P after e* ⟦*S*⟧ *Q*) ⊓ (*P* ⟦*S*⟧ *Q after e*))›
proof *cases*
case *1*
then obtain *s-P s-Q X-P X-Q*
where ∗∗ : ‹(*s-P*, *X-P*) ∈ ℱ *P*› ‹(*s-Q*, *X-Q*) ∈ ℱ *Q*›
‹(*ev e* # *s*) *setinterleaves* ((*s-P*, *s-Q*), *insert* ✓ (*ev* ‘ *S*))›
‹*X* = (*X-P* ∪ *X-Q*) ∩ *insert* ✓ (*ev* ‘ *S*) ∪ *X-P* ∩ *X-Q*› **by** *blast*
have ‹*s-P* ≠ [] ∧ *hd s-P* = *ev e* ∧ *s setinterleaves* ((*tl s-P*, *s-Q*), *insert* ✓ (*ev* ‘ *S*)) ∨
s-Q ≠ [] ∧ *hd s-Q* = *ev e* ∧ *s setinterleaves* ((*s-P*, *tl s-Q*), *insert* ✓ (*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 e* ⟦*S*⟧ *Q*) ⊓ (*P* ⟦*S*⟧ *Q after e*))›
apply (*elim disjE*; *simp add*: *F-Ndet*)
subgoal using ∗ ∗∗(*1*, *2*, *4*) *ready-hyps*(*1*) **by** *blast*

apply (*rule disjI2*, *subst Sync-commute*, *rule* ∗[*OF* ∗∗(*2*, *1*)])
by (*simp-all add*: ∗∗(*4*) *Int-commute Un-commute Sync-commute Sync.sym ready-hyps*(*2*))
next
assume ‹$s \in \mathcal{D}\ ((P\ [\![S]\!]\ Q)\ after\ e)$›
from *this*[*simplified same-div*]
show ‹$(s,\ X) \in \mathcal{F}\ ((P\ after\ e\ [\![S]\!]\ Q) \sqcap (P\ [\![S]\!]\ Q\ after\ e))$› **using** *NF-ND* **by** *blast*
qed
} **note** ∗∗ = *this*

show ‹$(s,\ X) \in \mathcal{F}\ ((P\ [\![S]\!]\ Q)\ after\ e) \Longrightarrow (s,\ X) \in \mathcal{F}\ ((P\ after\ e\ [\![S]\!]\ Q) \sqcap (P\ [\![S]\!]\ Q\ after\ e))$›
if *same-div*: ‹$\mathcal{D}\ ((P\ [\![S]\!]\ Q)\ after\ e) = \mathcal{D}\ ((P\ after\ e\ [\![S]\!]\ Q) \sqcap (P\ [\![S]\!]\ Q\ after\ e))$› **for** *s X*
apply (*simp add*: *F-After ready-set-Sync ready-hyps split*: *if-split-asm*)
apply (*erule disjE*;
simp add: *After-BOT Sync-commute*[*of* ⊥, *simplified Sync-BOT*] *Sync-BOT F-UU Ndet-id*,
metis butlast-tl front-tickFree-butlast tickFree-tl)
by (*metis* ∗∗ *same-div list.exhaust-sel*)
next

{ **fix** *s X P Q*
assume *assms* : ‹$P \neq \bot$› ‹$Q \neq \bot$› ‹$(s,\ X) \in \mathcal{F}\ (P\ after\ e\ [\![S]\!]\ Q)$› ‹*ev e* ∈ *ready-set P*›
and *same-div* : ‹$\mathcal{D}\ ((P\ [\![S]\!]\ Q)\ after\ e) = \mathcal{D}\ ((P\ after\ e\ [\![S]\!]\ Q) \sqcap (P\ [\![S]\!]\ Q\ after\ e))$›
from *assms*(*3*)[*simplified F-Sync*, *simplified*] **consider**
‹$\exists$ *s-P s-Q X-P X-Q*. $(ev\ e \#\ s\text{-}P,\ X\text{-}P) \in \mathcal{F}\ P \wedge (s\text{-}Q,\ X\text{-}Q) \in \mathcal{F}\ Q \wedge$
s setinterleaves ((*s-P*, *s-Q*), *insert* ✓ (*ev* ‘ *S*)) ∧
$X = (X\text{-}P \cup X\text{-}Q) \cap$ *insert* ✓ (*ev* ‘ *S*) ∪ *X-P* ∩ *X-Q*› |
‹$s \in \mathcal{D}\ (P\ after\ e\ [\![S]\!]\ Q)$›
by (*simp add*: *F-Sync F-After assms*(*4*) *D-Sync*)
(*metis* (*no-types*, *opaque-lifting*) *list.exhaust-sel*)
hence ‹$(ev\ e \#\ s,\ X) \in \mathcal{F}\ (P\ [\![S]\!]\ Q)$›
proof *cases*
case *1*
then obtain *s-P s-Q X-P X-Q*
where ∗ : ‹$(ev\ e \#\ s\text{-}P,\ X\text{-}P) \in \mathcal{F}\ P$› ‹$(s\text{-}Q,\ X\text{-}Q) \in \mathcal{F}\ Q$›
‹*s setinterleaves* ((*s-P*, *s-Q*), *insert* ✓ (*ev* ‘ *S*))›
‹$X = (X\text{-}P \cup X\text{-}Q) \cap$ *insert* ✓ (*ev* ‘ *S*) ∪ *X-P* ∩ *X-Q*› **by** *blast*
have ‹(*ev e* # *s*) *setinterleaves* ((*ev e* # *s-P*, *s-Q*), *insert* ✓ (*ev* ‘ *S*))›
using ∗(*3*) **by** (*cases s-Q*; *simp add*: *notin image-iff*)
with ∗(*1*, *2*, *4*) **show** ‹$(ev\ e \#\ s,\ X) \in \mathcal{F}\ (P\ [\![S]\!]\ Q)$›
by (*simp add*: *F-Sync*) *blast*
next
assume ‹$s \in \mathcal{D}\ (P\ after\ e\ [\![S]\!]\ Q)$›
hence ‹$s \in \mathcal{D}\ ((P\ [\![S]\!]\ Q)\ after\ e)$› **using** *same-div*[*simplified D-Ndet*] **by** *fast*

hence ‹$(s, X) \in \mathcal{F}\ ((P\ [\![S]\!]\ Q)\ after\ e)$› **using** *process-charn* **by** *blast*
thus ‹$(ev\ e\ \#\ s, X) \in \mathcal{F}\ (P\ [\![S]\!]\ Q)$›
by (*simp add*: *F-After ready-set-Sync assms*(*1*, *2*, *4*) *notin image-iff*)
(*metis list.collapse*)
qed
} **note** $* = this$

show ‹$(s, X) \in \mathcal{F}\ ((P\ after\ e\ [\![S]\!]\ Q) \sqcap (P\ [\![S]\!]\ Q\ after\ e)) \Longrightarrow (s, X) \in \mathcal{F}\ ((P\ [\![S]\!]\ Q)\ after\ e)$›
if *same-div* : ‹$\mathcal{D}\ ((P\ [\![S]\!]\ Q)\ after\ e) = \mathcal{D}\ ((P\ after\ e\ [\![S]\!]\ Q) \sqcap (P\ [\![S]\!]\ Q\ after\ e))$› **for** $s\ X$
apply (*simp add*: *F-After ready-set-Sync ready-hyps After-BOT F-UU*
Sync-commute[*of* $\perp$, *simplified Sync-BOT*] *Sync-BOT Ndet-id*)
apply (*intro conjI impI*)
apply ((*metis butlast.simps*(*2*) *event.simps*(*3*) *front-tickFree-butlast tick-Free-Cons*
front-tickFree-single is-processT2 list.distinct(*1*) *list.sel*(*1*, *3*))+)[*2*]
by (*simp add*: *F-Ndet*)
(*metis* $*$ *Ndet-commute Sync-commute list.distinct*(*1*) *list.sel*(*1*, *3*) *ready-hyps same-div*)
next

{ **fix** s
assume *assms* : ‹$P \neq \perp$› ‹$Q \neq \perp$› ‹$ev\ e\ \#\ s \in \mathcal{D}\ (P\ [\![S]\!]\ Q)$›
from *assms*(*3*) **obtain** $t\ u\ r\ v$
where $**$: ‹*front-tickFree* v› ‹*tickFree* $r \vee v = []$› ‹$ev\ e\ \#\ s = r\ @\ v$›
‹r *setinterleaves* $((t, u)$, *insert* ✓ $(ev\ `\ S))$›
‹$t \in \mathcal{D}\ P \wedge u \in \mathcal{T}\ Q \vee t \in \mathcal{D}\ Q \wedge u \in \mathcal{T}\ P$› **by** (*simp add*: *D-Sync*) *blast*
have $***$: ‹$r \neq []$› **using** $**$(*4*, *5*) *BOT-iff-D assms*(*1*, *2*) *empty-setinterleaving* **by** *blast*
have ‹$t \neq [] \wedge hd\ t = ev\ e \wedge tl\ r$ *setinterleaves* $((tl\ t, u)$, *insert* ✓ $(ev\ `\ S)) \vee$
$u \neq [] \wedge hd\ u = ev\ e \wedge tl\ r$ *setinterleaves* $((t, tl\ u)$, *insert* ✓ $(ev\ `\ S))$›
using $**$(*3*, *4*) **by** (*cases t*; *cases u*, *auto simp add*: $***$ *notin split*: *if-split-asm*)
with $**$(*5*) **have** ‹$s \in \mathcal{D}\ ((P\ after\ e\ [\![S]\!]\ Q) \sqcap (P\ [\![S]\!]\ Q\ after\ e))$›
proof (*elim disjE*)
assume $*$: ‹$t \in \mathcal{D}\ P \wedge u \in \mathcal{T}\ Q$›
‹$t \neq [] \wedge hd\ t = ev\ e \wedge tl\ r$ *setinterleaves* $((tl\ t, u)$, *insert* ✓ $(ev\ `\ S))$›
hence ‹$s = tl\ r\ @\ v \wedge tl\ t \in \mathcal{D}\ (P\ after\ e) \wedge u \in \mathcal{T}\ Q$›
using $**$(*3*) $***$ *ready-hyps*(*1*) **apply** (*simp add*: *D-After*, *intro conjI*)
by (*metis list.sel*(*3*) *tl-append2*) *blast*
thus ‹$s \in \mathcal{D}\ ((P\ after\ e\ [\![S]\!]\ Q) \sqcap (P\ [\![S]\!]\ Q\ after\ e))$›
using $*$(*2*) $**$(*1*, *2*) *tickFree-tl* **by** (*simp add*: *D-Ndet D-Sync*) *blast*
next
assume $*$: ‹$t \in \mathcal{D}\ P \wedge u \in \mathcal{T}\ Q$›
‹$u \neq [] \wedge hd\ u = ev\ e \wedge tl\ r$ *setinterleaves* $((t, tl\ u)$, *insert* ✓ $(ev\ `\ S))$›
hence ‹$s = tl\ r\ @\ v \wedge t \in \mathcal{D}\ P \wedge tl\ u \in \mathcal{T}\ (Q\ after\ e)$›
using $**$(*3*) *ready-hyps*(*2*) $***$ **apply** (*simp add*: *T-After*, *intro conjI*)

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

show ‹$s \in \mathcal{D}\ ((P\ [\![S]\!]\ Q)\ after\ e) \Longrightarrow s \in \mathcal{D}\ ((P\ after\ e\ [\![S]\!]\ Q) \sqcap (P\ [\![S]\!]\ Q\ after\ e))$› **for** *s*
apply (*simp add*: *D-After ready-set-Sync ready-hyps After-BOT D-UU*
split: *if-split-asm*)
apply (*erule disjE*;
simp add: *After-BOT Sync-commute*[*of* ⊥, *simplified Sync-BOT*] *Sync-BOT Ndet-id D-UU*,
metis butlast-tl front-tickFree-butlast tickFree-tl)
by (*metis* ∗ *list.collapse*)
next

{ **fix** *s P Q*
assume *assms* : ‹$P \neq \bot$› ‹$Q \neq \bot$› ‹$s \in \mathcal{D}\ ((P\ after\ e\ [\![S]\!]\ Q))$› ‹$ev\ e \in ready\text{-}set\ P$›
from *assms*(*3*)[*simplified D-Sync*, *simplified*] **obtain** *t u r v*
where ∗ : ‹*front-tickFree v*› ‹*tickFree r* ∨ *v* = []› ‹*s* = *r* @ *v*›
‹*r setinterleaves* ((*t*, *u*), *insert* ✓ (*ev* ‘ *S*))›
‹*ev e* # *t* ∈ $\mathcal{D}$ *P* ∧ *u* ∈ $\mathcal{T}$ *Q* ∨ *t* ∈ $\mathcal{D}$ *Q* ∧ *ev e* # *u* ∈ $\mathcal{T}$ *P*›
by (*simp add*: *D-Sync D-After T-After assms*(*4*)) (*metis list.collapse*)
from ∗(*5*) **have** ‹*ev e* # *s* ∈ $\mathcal{D}$ (*P* [*S*] *Q*)›
proof (*elim disjE*)
assume ∗∗ : ‹*ev e* # *t* ∈ $\mathcal{D}$ *P* ∧ *u* ∈ $\mathcal{T}$ *Q*›
have ∗∗∗ : ‹(*ev e* # *r*) *setinterleaves* ((*ev e* # *t*, *u*), *insert* ✓ (*ev* ‘ *S*))›
using ∗(*4*) **by** (*cases u*; *simp add*: *notin image-iff* ∗(*1*, *2*, *3*))
show ‹*ev e* # *s* ∈ $\mathcal{D}$ (*P* [*S*] *Q*)›
by (*simp add*: *D-Sync*)

```
      (metis ∗(1, 2, 3) ∗∗ ∗∗∗ Cons-eq-appendI event.distinct(1) tickFree-Cons)
  next
    assume ∗∗ : ‹t ∈ 𝒟 Q ∧ ev e # u ∈ 𝒯 P›
    have ∗∗∗ : ‹(ev e # r) setinterleaves ((t, ev e # u), insert ✓ (ev ‘ S))›
      using ∗(4) by (cases t; simp add: notin image-iff ∗(1, 2, 3))
    show ‹ev e # s ∈ 𝒟 (P ⟦S⟧ Q)›
      by (simp add: D-Sync)
        (metis ∗(1, 2, 3) ∗∗ ∗∗∗ Cons-eq-appendI event.distinct(1) tickFree-Cons)
  qed
 } note ∗ = this

 show ‹s ∈ 𝒟 ((P after e ⟦S⟧ Q) ⊓ (P ⟦S⟧ Q after e)) ⟹ s ∈ 𝒟 ((P ⟦S⟧ Q)
after e)› for s
  apply (simp add: D-After ready-set-Sync ready-hyps After-BOT D-UU
                   Sync-commute[of ⊥, simplified Sync-BOT] Sync-BOT notin
Ndet-id)
  apply (intro conjI impI)
    apply ((metis D-imp-front-tickFree butlast.simps(2) event.distinct(1) tick-
Free-Cons
          front-tickFree-single list.discI list.sel(1, 3) front-tickFree-butlast)+)[2]
  by (simp add: D-Ndet)
     (metis ∗ list.distinct(1) list.sel(1, 3) mono-D-Sync ready-hyps)
qed


lemma not-readyL-not-readyR-After-Sync: ‹(P ⟦S⟧ Q) after e = STOP›
 if ready-hyps: ‹ev e ∉ ready-set P› ‹ev e ∉ ready-set Q›
 apply (subst not-ready-After, simp add: ready-set-Sync)
 using ready-set-BOT ready-hyps by auto
```

Finally, the monster theorem !

```
theorem After-Sync:
 ‹(P ⟦S⟧ Q) after e =
  (       if P = ⊥ ∨ Q = ⊥ then ⊥
   else   if ev e ∈ ready-set P ∧ ev e ∈ ready-set Q
        then   if e ∈ S then P after e ⟦S⟧ Q after e
               else (P after e ⟦S⟧ Q) ⊓ (P ⟦S⟧ Q after e)
   else   if e ∈ S then STOP
   else   if ev e ∈ ready-set P then P after e ⟦S⟧ Q
   else   if ev e ∈ ready-set Q then P ⟦S⟧ Q after e
   else STOP)›
 by (simp add: Sync-commute[of ⊥, simplified Sync-BOT] Sync-BOT After-BOT
               readyL-readyR-in-After-Sync readyL-readyR-not-in-After-Sync
               not-readyL-in-After-Sync readyL-not-readyR-not-in-After-Sync
               not-readyL-not-readyR-After-Sync)
  (metis Sync-commute not-readyL-in-After-Sync readyL-not-readyR-not-in-After-Sync)
```

4.5.4 *After* Hiding Operator

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

lemma *Hiding-FD-Hiding-After-if-ready-inside*:
  ‹$e \in B \Longrightarrow (P \setminus B) \sqsubseteq_{FD} (P \text{ after } e \setminus B)$›
 and *After-Hiding-FD-Hiding-After-if-ready-notin*:
  ‹$e \notin B \Longrightarrow (P \setminus B) \text{ after } e \sqsubseteq_{FD} (P \text{ after } e \setminus B)$›
 if *ready*: ‹*ev e* ∈ *ready-set P*›
 supply *ready′* = *ready-notin-imp-ready-Hiding*[*OF ready*]
proof −
 { fix *s*
  assume ‹$s \in \mathcal{D}\ (P \text{ after } e \setminus B)$›
  with *D-Hiding* **obtain** *t u*
   where ∗ : ‹*front-tickFree u*› ‹*tickFree t*› ‹s = *trace-hide t* (*ev* ‘ *B*) @ *u*›
        ‹$t \in \mathcal{D}\ (P \text{ after } e) \vee (\exists\ f.\ \text{isInfHiddenRun}\ f\ (P \text{ after } e)\ B \wedge t \in \text{range}$
f)›
   by *blast*
  from ∗(*4*) **have** ‹$s \in (\text{if } e \in B \text{ then } \mathcal{D}\ (P \setminus B) \text{ else } \mathcal{D}\ ((P \setminus B) \text{ after } e))$›
  proof (*elim disjE*)
   assume ‹$t \in \mathcal{D}\ (P \text{ after } e)$›
  hence ∗∗ : ‹*ev e* # *t* ∈ $\mathcal{D}$ *P*› **by** (*simp add*: *D-After ready*) (*metis list.exhaust-sel*)
   show ‹$s \in (\text{if } e \in B \text{ then } \mathcal{D}\ (P \setminus B) \text{ else } \mathcal{D}\ ((P \setminus B) \text{ after } e))$›
   proof (*split if-split*, *intro conjI impI*)
    assume ‹*e* ∈ *B*›
    with ∗(*3*) **have** ∗∗∗ : ‹s = *trace-hide* (*ev e* # *t*) (*ev* ‘ *B*) @ *u*› **by** *simp*
    show ‹$s \in \mathcal{D}\ (P \setminus B)$›
     by (*simp add*: *D-Hiding*)
      (*metis* ∗(*1*, *2*) ∗∗ ∗∗∗ *event.distinct*(*1*) *tickFree-Cons*)
   next
    assume ‹*e* ∉ *B*›
    with ∗(*3*) **have** ∗∗∗ : ‹*ev e* # s = *trace-hide* (*ev e* # *t*) (*ev* ‘ *B*) @ *u*›
     by (*simp add*: *image-iff*)
    have ‹*ev e* # $s \in \mathcal{D}\ (P \setminus B)$›
     by (*simp add*: *D-Hiding*)
      (*metis* ∗(*1*, *2*) ∗∗ ∗∗∗ *event.distinct*(*1*) *tickFree-Cons*)
    thus ‹$s \in \mathcal{D}\ ((P \setminus B) \text{ after } e)$›
     by (*simp add*: *D-After ready′* ‹*e* ∉ *B*›)
      (*metis list.distinct*(*1*) *list.sel*(*1*, *3*))
   qed
  next
   assume ‹$\exists f.\ \text{isInfHiddenRun}\ f\ (P \text{ after } e)\ B \wedge t \in \text{range}\ f$›
    then obtain *f* **where** ‹*isInfHiddenRun f* (*P after e*) *B*› ‹*t* ∈ *range f*› **by**
blast
   hence ∗∗ : ‹*isInfHiddenRun* (λ*i*. *ev e* # *f i*) *P B* ∧
        ev e # *t* ∈ *range* (λ*i*. *ev e* # *f i*)›
    by (*simp add*: *less-cons monotone-on-def T-After ready*)
     (*metis list.exhaust-sel rangeE rangeI*)
   show ‹$s \in (\text{if } e \in B \text{ then } \mathcal{D}\ (P \setminus B) \text{ else } \mathcal{D}\ ((P \setminus B) \text{ after } e))$›
   proof (*split if-split*, *intro conjI impI*)

```
      assume ‹e ∈ B›
      with *(3) have *** : ‹s = trace-hide (ev e # t) (ev ‘ B) @ u› by simp
      show ‹s ∈ 𝒟 (P \ B)›
        by (simp add: D-Hiding)
           (metis *(1, 2) ** *** event.distinct(1) tickFree-Cons)
    next
      assume ‹e ∉ B›
      with *(3) have *** : ‹ev e # s = trace-hide (ev e # t) (ev ‘ B) @ u›
        by (simp add: image-iff)
      have ‹ev e # s ∈ 𝒟 (P \ B)›
        by (simp add: D-Hiding)
           (metis *(1, 2) ** *** event.distinct(1) tickFree-Cons)
      thus ‹s ∈ 𝒟 ((P \ B) after e)›
        by (simp add: D-After ready' ‹e ∉ B›)
           (metis list.distinct(1) list.sel(1, 3))
    qed
  qed
 } note div-ref = this

 { fix s X
  assume ‹(s, X) ∈ ℱ (P after e \ B)›
  with F-Hiding D-Hiding consider
      ‹∃ t. s = trace-hide t (ev ‘ B) ∧ (t, X ∪ ev ‘ B) ∈ ℱ (P after e)›
    | ‹s ∈ 𝒟 (P after e \ B)› by blast
  hence ‹(s, X) ∈ (if e ∈ B then ℱ (P \ B) else ℱ ((P \ B) after e))›
  proof cases
    assume ‹∃ t. s = trace-hide t (ev ‘ B) ∧ (t, X ∪ ev ‘ B) ∈ ℱ (P after e)›
    then obtain t where * : ‹s = trace-hide t (ev ‘ B)› ‹(ev e # t, X ∪ ev ‘ B)
∈ ℱ P›
      by (simp add: F-After ready) (metis list.exhaust-sel)
    show ‹(s, X) ∈ (if e ∈ B then ℱ (P \ B) else ℱ ((P \ B) after e))›
    proof (split if-split, intro conjI impI)
      from * show ‹e ∈ B ⟹ (s, X) ∈ ℱ (P \ B)›
        by (simp add: F-Hiding) (metis filter.simps(2) image-eqI)
    next
      assume ‹e ∉ B›
      with *(1) have ** : ‹ev e # s = trace-hide (ev e # t) (ev ‘ B)›
        by (simp add: image-iff)
      show ‹(s, X) ∈ ℱ ((P \ B) after e)›
        by (simp add: F-After ready' ‹e ∉ B› F-Hiding)
           (metis *(2) ** list.discI list.sel(1, 3))
    qed
  next
    show ‹s ∈ 𝒟 (P after e \ B) ⟹ (s, X) ∈ (if e ∈ B then ℱ (P \ B) else ℱ
((P \ B) after e))›
      by (drule div-ref, simp split: if-split-asm; use NF-ND in blast)
  qed
 } note fail-ref = this
```

show ‹$e \in B \Longrightarrow (P \setminus B) \sqsubseteq_{FD} (P\ after\ e \setminus B)$›
and ‹$e \notin B \Longrightarrow (P \setminus B)\ after\ e \sqsubseteq_{FD} (P\ after\ e \setminus B)$›
unfolding *failure-divergence-refine-def le-ref-def* **using** *div-ref fail-ref* **by** *auto*
qed

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

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

corollary *Hiding-DT-Hiding-After-if-ready-inside*:
‹*ev e* ∈ *ready-set P* ⟹ *e* ∈ *B* ⟹ (*P* \ *B*) ⊑$_{DT}$ (*P after e* \ *B*)›
and *After-Hiding-DT-Hiding-After-if-ready-notin*:
‹*ev e* ∈ *ready-set P* ⟹ *e* ∉ *B* ⟹ (*P* \ *B*) *after e* ⊑$_{DT}$ (*P after e* \ *B*)›
by (*simp add*: *Hiding-D-Hiding-After-if-ready-inside*
Hiding-T-Hiding-After-if-ready-inside leD-leT-imp-leDT)
(*simp add*: *After-Hiding-D-Hiding-After-if-ready-notin*
After-Hiding-T-Hiding-After-if-ready-notin leD-leT-imp-leDT)

This is the best we can obtain: even by restricting ourselves to two events, we can already construct a counterexample.

lemma defines *P-def*: ‹*P* ≡ (*Suc 0* → (*0* → *STOP*)) ⊓ (*0* → *SKIP*)›
and *B-def*: ‹*B* ≡ {*Suc 0*}› **and** *e-def* : ‹*e* ≡ *0*› **and** *f-def*: ‹*f* ≡ *Suc 0*›
shows ‹*ev e* ∈ *ready-set P* ∧ *f* ∈ *B* ∧ *P* \ *B* ≠ *P after f* \ *B*›
and ‹*ev e* ∈ *ready-set P* ∧ *e* ∉ *B* ∧ (*P* \ *B*) *after e* ≠ *P after e* \ *B*›
unfolding *e-def f-def P-def B-def*
apply (*simp-all add*: *ready-set-Ndet ready-set-prefix*)
apply (*simp-all add*: *After-Ndet ready-set-prefix After-prefix Hiding-set-SKIP*)
apply (*simp-all add*: *Hiding-Ndet After-Ndet ready-set-prefix After-prefix Hiding-set-SKIP*
Hiding-set-STOP no-Hiding-write0 Hiding-write0)
by (*metis After-prefix Ndet-is-STOP-iff SKIP-Neq-STOP write0-Ndet*)
(*metis mono-Ndet-FD-right Ndet-commute SKIP-FD-iff SKIP-Neq-STOP STOP-FD-iff*)

4.5.5 *After* Renaming

lemma *After-Renaming*:
‹*Renaming P f after e* =

(*if* $P = \bot$ *then* $\bot$ *else*
$\sqcap a \in \{a.\ ev\ a \in ready\text{-}set\ P \wedge f\ a = e\}.\ Renaming\ (P\ after\ a)\ f$⟩
(**is** ‹*?lhs* = (*if* $P = \bot$ *then* $\bot$ *else ?rhs*)›)
— We treat the case $P = \bot$ separately because the set $\{a.\ f\ a = e\}$ may be empty, which implies $\sqcap\ a{\in}\{a.\ ev\ a \in ready\text{-}set\ P \wedge f\ a = e\}.\ Renaming\ (P\ after\ a)\ f = STOP$ while $Renaming\ P\ f\ after\ e = \bot$.

proof (*split if-split*, *intro conjI impI*)
show ‹$P = \bot \Longrightarrow$ *?lhs*= $\bot$›
by (*simp add*: *Renaming-BOT After-BOT GlobalNdet-id*)
next
assume *non-BOT*: ‹$P \neq \bot$›
show ‹*?lhs* = *?rhs*›
proof (*subst Process-eq-spec-optimized*, *safe*)
fix *s*
assume ‹$s \in \mathcal{D}$ *?lhs*›
hence $*$: ‹*ev e* ∈ *ready-set* (*Renaming P f*)› ‹*ev e* # *s* ∈ $\mathcal{D}$ (*Renaming P f*)›
by (*auto simp add*: *D-After split*: *if-split-asm*) (*metis list.exhaust-sel*)
from $*(2)$ **obtain** *t1 t2*
where $**$: ‹*tickFree t1*› ‹*front-tickFree t2*›
‹*ev e* # *s* = *map* (*EvExt f*) *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* = *e*›
by (*cases t1*; *simp add*: *BOT-iff-D EvExt-def*)
(*metis comp-apply event.exhaust event.inject event.simps*(4))
have ‹*ev a* ∈ *ready-set P*›
using $**(4)$ $***(1)$ *Cons-in-T-imp-elem-ready-set D-T* **by** *blast*
also have ‹*s* ∈ $\mathcal{D}$ (*Renaming* (*P after a*) *f*)›
using $**$ $***(1)$ **by** (*simp add*: *D-Renaming D-After calculation*)
(*metis list.discI list.sel*$(1, 3)$)
ultimately show ‹*s* ∈ $\mathcal{D}$ *?rhs*›
using $***(2)$ **by** (*simp add*: *D-GlobalNdet*) *blast*
next
show ‹*s* ∈ $\mathcal{D}$ *?rhs* $\Longrightarrow$ *s* ∈ $\mathcal{D}$ *?lhs*› **for** *s*
apply (*simp add*: *D-GlobalNdet D-Renaming D-After*
ready-set-Renaming non-BOT EvExt-def image-iff
split: *if-split-asm event.split*, *intro conjI*)
by (*metis* (*no-types*, *lifting*) *Nil-is-append-conv event.distinct*(1)
hd-append2 hd-map-EvExt list.collapse list.map-disc-iff
map-tl tickFree-Cons tl-append2) *blast*
next
fix *s X*
assume *same-div* : ‹$\mathcal{D}$ *?lhs* = $\mathcal{D}$ *?rhs*›
assume ‹(*s*, *X*) ∈ $\mathcal{F}$ *?lhs*›
then consider ‹*ev e* ∉ *ready-set* (*Renaming P f*)› ‹*s* = []›
| ‹*ev e* ∈ *ready-set* (*Renaming P f*)› ‹(*ev e* # *s*, *X*) ∈ $\mathcal{F}$ (*Renaming P f*)›
by (*simp add*: *F-After split*: *if-split-asm*) (*metis list.exhaust-sel*)
thus ‹(*s*, *X*) ∈ $\mathcal{F}$ *?rhs*›

```
  proof cases
    show ‹ev e ∉ ready-set (Renaming P f) ⟹ s = [] ⟹ (s, X) ∈ ℱ ?rhs›
      by (simp add: ready-set-Renaming non-BOT F-GlobalNdet F-Renaming
F-After)
         (metis ev-elem-anteced1 imageI vimage-eq)
  next
    assume assms : ‹ev e ∈ ready-set (Renaming P f)›
                   ‹(ev e # s, X) ∈ ℱ (Renaming P f)›
    from assms(2) consider ‹ev e # s ∈ 𝒟 (Renaming P f)›
      | ‹∃ s1. (s1, EvExt f −‘ X) ∈ ℱ P ∧ ev e # s = map (EvExt f) s1›
      by (simp add: F-Renaming D-Renaming) blast
    thus ‹(s, X) ∈ ℱ ?rhs›
    proof cases
      assume ‹ev e # s ∈ 𝒟 (Renaming P f)›
      hence ‹s ∈ 𝒟 ?lhs› by (force simp add: D-After assms(1))
      with D-F same-div show ‹(s, X) ∈ ℱ ?rhs› by blast
    next
     assume ‹∃ s1. (s1, EvExt f −‘ X) ∈ ℱ P ∧ ev e # s = map (EvExt f) s1›
      then obtain s1 where ∗ : ‹(s1, EvExt f −‘ X) ∈ ℱ P›
                             ‹ev e # s = map (EvExt f) s1› by meson
      from ∗(2) obtain a s1′
        where ∗∗ : ‹s1 = ev a # s1′› ‹f a = e›
        by (cases s1; simp; metis ∗(2) EvExt-ev1 event.inject
                                  hd-map-EvExt list.distinct(1) list.sel(1))
      have ‹ev a ∈ ready-set P›
        using ∗(1) ∗∗(1) Cons-in-T-imp-elem-ready-set F-T by blast
      also have ‹(s, X) ∈ ℱ (Renaming (P after a) f)›
        using ∗(1, 2) ∗∗(1) by (simp add: F-Renaming F-After calculation)
                               (metis list.distinct(1) list.sel(1, 3))
      ultimately show ‹(s, X) ∈ ℱ ?rhs›
        using ∗∗(2) by (simp add: F-GlobalNdet) blast
    qed
  qed
next
  fix s X
  assume same-div : ‹𝒟 ?lhs = 𝒟 ?rhs›
  assume ‹(s, X) ∈ ℱ ?rhs›
  then consider ‹∀ a. ev a ∈ ready-set P ⟶ f a ≠ e› ‹s = []›
    | ‹∃ a. f a = e ∧ ev a ∈ ready-set P ∧ (s, X) ∈ ℱ (Renaming (P after a) f)›
    by (auto simp add: F-GlobalNdet split: if-split-asm)
  thus ‹(s, X) ∈ ℱ ?lhs›
  proof cases
    show ‹∀ a. ev a ∈ ready-set P ⟶ f a ≠ e ⟹ s = [] ⟹ (s, X) ∈ ℱ ?lhs›
      by (auto simp add: F-After F-Renaming ready-set-Renaming non-BOT)
         (metis (no-types, opaque-lifting) EvExt-ev1 event.inject
                hd-map-EvExt list.distinct(1) list.sel(1) list.simps(9))
  next
    assume ‹∃ a. f a = e ∧ ev a ∈ ready-set P ∧
                (s, X) ∈ ℱ (Renaming (P after a) f)›
```

```
    then obtain a
      where ∗ : ‹f a = e› ‹ev a ∈ ready-set P›
              ‹(s, X) ∈ ℱ (Renaming (P after a) f)› by blast
    from ∗(3) consider ‹s ∈ 𝒟 (Renaming (P after a) f)›
      | ‹∃ s1. (s1, EvExt f −‘ X) ∈ ℱ (P after a) ∧ s = map (EvExt f) s1›
      by (simp add: F-Renaming D-Renaming) blast
    thus ‹(s, X) ∈ ℱ ?lhs›
    proof cases
      assume ‹s ∈ 𝒟 (Renaming (P after a) f)›
      with ∗(1, 2) have ‹s ∈ 𝒟 ?rhs› by (simp add: D-GlobalNdet) blast
      with D-F same-div show ‹(s, X) ∈ ℱ ?lhs› by blast
    next
      assume ‹∃ s1. (s1, EvExt f −‘ X) ∈ ℱ (P after a) ∧ s = map (EvExt f)
s1›
      then obtain s1 where ∗∗ : ‹(s1, EvExt f −‘ X) ∈ ℱ (P after a)›
                               ‹s = map (EvExt f) s1› by blast
      with ∗(1) have ∗∗∗ : ‹ev e ∉ ready-set (Renaming P f) ⟹ s = []›
        by (simp add: ready-set-Renaming non-BOT image-iff F-After
            split: if-split-asm) (metis hd-map hd-map-EvExt)
      { assume ‹ev e ∈ ready-set (Renaming P f)›
        hence ‹(ev a # s1, EvExt f −‘ X) ∈ ℱ P ∧ ev e # s = map (EvExt f)
(ev a # s1)›
          using ∗(1, 2) ∗∗ by (simp add: F-After ∗(2))
                              (metis hd-map hd-map-EvExt list.collapse)
        hence ‹(ev e # s, X) ∈ ℱ (Renaming P f)›
          by (auto simp add: F-Renaming)
      }
      thus ‹(s, X) ∈ ℱ ?lhs›
        by (simp add: F-After ∗∗∗) (metis list.discI list.sel(1, 3))
    qed
  qed
 qed
qed
```

4.6 Behaviour of *After* with Operators of HOL-CSPM

```
lemma After-MultiDet-is-After-MultiNdet:
 ‹finite A ⟹ (□ a ∈ A. P a) after e = (⨅ a ∈ A. P a) after e›
 apply (cases ‹A = {}›, simp)
 apply (rotate-tac, induct rule: finite-set-induct-nonempty, simp)
 by (simp add: After-Det-is-After-Ndet After-Ndet
               ready-set-MultiDet ready-set-MultiNdet)


lemma After-GlobalNdet: ‹(⊓ a ∈ A. P a) after e =
                          (  if ev e ∉ (⋃ a ∈ A. ready-set (P a)) then STOP
                           else (⊓ a ∈ {a ∈ A. ev e ∈ ready-set (P a)}. P a) after e)›
 apply (subst Process-eq-spec, auto simp add: not-ready-After ready-set-GlobalNdet)
  apply (auto simp add: F-After F-GlobalNdet D-After D-GlobalNdet ready-set-GlobalNdet
```

split: *if-split-asm*,
auto simp add: *ready-set-def*)
by (*metis F-T append-Cons append-Nil is-processT3-ST list.exhaust list.sel*(*1*))
(*metis D-T append-Cons append-Nil hd-Cons-tl is-processT3-ST*)

lemma *After-MultiNdet*: ‹*finite A* ⟹ (⨅ *a* ∈ *A*. *P a*) *after e* =
(*if ev e* ∉ (⋃ *a* ∈ *A*. *ready-set* (*P a*)) *then STOP*
else (⨅ *a* ∈ {*a* ∈ *A*. *ev e* ∈ *ready-set* (*P a*)}. *P a*) *after e*)›
by (*subst* (*1 2*) *finite-GlobalNdet-is-MultiNdet*[*symmetric*], *simp-all add*: *After-GlobalNdet*)

lemma *After-MultiDet*: ‹*finite A* ⟹
(□ *a* ∈ *A*. *P a*) *after e* =
(*if ev e* ∉ (⋃ *a* ∈ *A*. *ready-set* (*P a*)) *then STOP*
else (⨅ *a* ∈ {*a* ∈ *A*. *ev e* ∈ *ready-set* (*P a*)}. *P a*) *after e*)›
by (*simp add*: *After-MultiDet-is-After-MultiNdet After-MultiNdet*)

4.7 Behaviour of *After* with Operators of `HOL-CSP_OpSem`

4.7.1 *After* Sliding

lemma *After-Sliding*:
‹*P* ▷ *Q after e* =
(*if ev e* ∉ *ready-set P* ∧ *ev e* ∉ *ready-set Q then STOP*
else if ev e ∈ *ready-set P* ∧ *ev e* ∈ *ready-set Q then* (*P after e*) ⊓ (*Q after e*)
else if ev e ∈ *ready-set P then P after e else Q after e*)›
by (*simp add*: *Sliding-def After-Ndet After-Det ready-set-Det Ndet-id Ndet-assoc*)

An interesting corollary is that (▷) is also concerned by the loss of determinism (see *?P* □ *?Q after ?e* = *?P* ⊓ *?Q after ?e*).

lemma *After-Sliding-is-After-Ndet*: ‹*P* ▷ *Q after e* = *P* ⊓ *Q after e*›
by (*simp add*: *After-Ndet After-Sliding*)

4.7.2 *After* Throwing

lemma *After-Throw*:
‹(*P* Θ *a* ∈ *A*. *Q a*) *after e* =
(*if P* = ⊥ *then* ⊥
else if ev e ∈ *ready-set P then if e* ∈ *A then Q e*
else (*P after e*) Θ *a* ∈ *A*. *Q a*
else STOP)›
(**is** ‹*?lhs* = *?rhs*›)
proof −
have ‹*?lhs* = *Q e*› **if** ‹*P* ≠ ⊥› **and** ‹*ev e* ∈ *ready-set P*› **and** ‹*e* ∈ *A*›
proof (*subst Process-eq-spec*, *safe*)
fix *s*
assume ‹*s* ∈ 𝒟 *?lhs*›

with *that*(*2*) **have** ‹*ev e* # *s* ∈ $\mathcal{D}$ (*P* Θ *a* ∈ *A*. *Q a*)›
by (*simp add*: *D-After ready-set-Throw*) (*metis list.exhaust-sel*)
with *that*(*1*) **show** ‹*s* ∈ $\mathcal{D}$ (*Q e*)›
apply (*simp add*: *D-Throw disjoint-iff BOT-iff-D*, *elim disjE*)
by (*metis hd-append2 hd-in-set image-eqI list.sel*(*1*) *that*(*3*))
(*metis append-self-conv2 event.inject hd-append2*
hd-in-set image-eqI list.sel(*1*, *3*) *that*(*3*))
next
have ‹[*ev e*] ∈ $\mathcal{T}$ *P* ∧ *e* ∈ *A*› **using** *ready-set-def that*(*2*, *3*) **by** *blast*
thus ‹*s* ∈ $\mathcal{D}$ (*Q e*) ⟹ *s* ∈ $\mathcal{D}$ ((*P* Θ *a* ∈ *A*. *Q a*) *after e*)› **for** *s*
by (*simp add*: *D-After ready-set-Throw that*(*2*) *D-Throw*)
(*metis append-Nil empty-set inf-bot-left list.distinct*(*1*) *list.sel*(*1*, *3*))
next
fix *s X*
assume ‹(*s*, *X*) ∈ $\mathcal{F}$ *?lhs*›
with *that*(*2*) **have** ‹(*ev e* # *s*, *X*) ∈ $\mathcal{F}$ (*P* Θ *a* ∈ *A*. *Q a*)›
by (*simp add*: *F-After ready-set-Throw*) (*metis list.exhaust-sel*)
with *that*(*1*, *3*) **show** ‹(*s*, *X*) ∈ $\mathcal{F}$ (*Q e*)›
apply (*simp add*: *F-Throw image-iff BOT-iff-D*, *elim disjE*)
by (*metis disjoint-iff hd-append2 hd-in-set image-eqI list.sel*(*1*))
(*metis append-Nil event.inject hd-append2 imageI insert-disjoint*(*2*)
list.exhaust-sel list.sel(*1*, *3*) *list.simps*(*15*))
next
have ‹[*ev e*] ∈ $\mathcal{T}$ *P* ∧ *e* ∈ *A*› **using** *ready-set-def that*(*2*, *3*) **by** *blast*
thus ‹(*s*, *X*) ∈ $\mathcal{F}$ (*Q e*) ⟹ (*s*, *X*) ∈ $\mathcal{F}$ *?lhs*› **for** *s X*
by (*simp add*: *F-After ready-set-Throw that*(*2*) *F-Throw*)
(*metis append-Nil empty-set inf-bot-left list.distinct*(*1*) *list.sel*(*1*, *3*))
qed

also have ‹*?lhs* = (*P after e*) Θ *a* ∈ *A*. *Q a*›
if ‹*P* ≠ ⊥› **and** ‹*ev e* ∈ *ready-set P*› **and** ‹*e* ∉ *A*›
proof (*subst Process-eq-spec-optimized*, *safe*)
fix *s*
assume ‹*s* ∈ $\mathcal{D}$ *?lhs*›
with *that*(*2*) **have** ‹*ev e* # *s* ∈ $\mathcal{D}$ (*P* Θ *a* ∈ *A*. *Q a*)›
by (*simp add*: *D-After ready-set-Throw*) (*metis list.exhaust-sel*)
then consider ‹∃ *t1 t2*. *ev e* # *s* = *t1* @ *t2* ∧ *t1* ∈ $\mathcal{D}$ *P* ∧ *tickFree t1* ∧
set t1 ∩ *ev* ‘ *A* = {} ∧ *front-tickFree t2*›
| ‹∃ *t1 a t2*. *ev e* # *s* = *t1* @ *ev a* # *t2* ∧ *t1* @ [*ev a*] ∈ $\mathcal{T}$ *P* ∧
set t1 ∩ *ev* ‘ *A* = {} ∧ *a* ∈ *A* ∧ *t2* ∈ $\mathcal{D}$ (*Q a*)›
by (*simp add*: *D-Throw*) *blast*
thus ‹*s* ∈ $\mathcal{D}$ ((*P after e*) Θ *a* ∈ *A*. *Q a*)›
proof *cases*
assume ‹∃ *t1 t2*. *ev e* # *s* = *t1* @ *t2* ∧ *t1* ∈ $\mathcal{D}$ *P* ∧ *tickFree t1* ∧
set t1 ∩ *ev* ‘ *A* = {} ∧ *front-tickFree t2*›
then obtain *t1 t2*
where ∗ : ‹*ev e* # *s* = *t1* @ *t2*› ‹*t1* ∈ $\mathcal{D}$ *P*› ‹*tickFree t1*›
‹*set t1* ∩ *ev* ‘ *A* = {}› ‹*front-tickFree t2*› **by** *blast*
from *that*(*1*) ∗(*1*, *2*) *BOT-iff-D* **obtain** *t1′*

```
    where ‹t1 = ev e # t1'› by (cases t1) auto
  with ∗(1, 2, 3, 4) have ‹s = t1' @ t2 ∧ t1' ∈ 𝒟 (P after e) ∧
                           tickFree t1' ∧ set t1' ∩ ev ‘ A = {}›
    by (simp add: D-After that(2)) (metis list.discI list.sel(1, 3))
  with ∗(5) show ‹s ∈ 𝒟 ((P after e) Θ a ∈ A. Q a)›
    by (auto simp add: D-Throw)
 next
  assume ‹∃t1 a t2. ev e # s = t1 @ ev a # t2 ∧ t1 @ [ev a] ∈ 𝒯 P ∧
            set t1 ∩ ev ‘ A = {} ∧ a ∈ A ∧ t2 ∈ 𝒟 (Q a)›
  then obtain t1 a t2
    where ∗ : ‹ev e # s = t1 @ ev a # t2› ‹t1 @ [ev a] ∈ 𝒯 P›
          ‹set t1 ∩ ev ‘ A = {}› ‹a ∈ A› ‹t2 ∈ 𝒟 (Q a)› by blast
  have ‹e ≠ a› using ∗(4) that(3) by blast
  with ∗(1) obtain t1' where ∗∗ : ‹t1 = ev e # t1'› by (cases t1) auto
  with ∗(2) that(2) have ‹t1' @ [ev a] ∈ 𝒯 (P after e)›
    by (simp add: T-After) (metis list.distinct(1) list.sel(1, 3))
  thus ‹s ∈ 𝒟 ((P after e) Θ a ∈ A. Q a)›
    using ∗(1, 3, 4, 5) ∗∗ by (simp add: D-Throw) blast
 qed
next
 fix s
 assume ‹s ∈ 𝒟 ((P after e) Θ a ∈ A. Q a)›
 then consider ‹∃t1 t2. s = t1 @ t2 ∧ t1 ∈ 𝒟 (P after e) ∧ tickFree t1 ∧
                    set t1 ∩ ev ‘ A = {} ∧ front-tickFree t2›
  | ‹∃t1 a t2. s = t1 @ ev a # t2 ∧ t1 @ [ev a] ∈ 𝒯 (P after e) ∧
            set t1 ∩ ev ‘ A = {} ∧ a ∈ A ∧ t2 ∈ 𝒟 (Q a)›
    by (simp add: D-Throw) blast
 thus ‹s ∈ 𝒟 ?lhs›
 proof cases
  assume ‹∃t1 t2. s = t1 @ t2 ∧ t1 ∈ 𝒟 (P after e) ∧ tickFree t1 ∧
                set t1 ∩ ev ‘ A = {} ∧ front-tickFree t2›
  then obtain t1 t2
    where ∗ : ‹s = t1 @ t2› ‹t1 ∈ 𝒟 (P after e)› ‹tickFree t1›
          ‹set t1 ∩ ev ‘ A = {}› ‹front-tickFree t2› by blast
  from ∗(2) that(2) have ∗∗ : ‹ev e # t1 ∈ 𝒟 P›
    by (simp add: D-After) (metis list.exhaust-sel)
  have ∗∗∗ : ‹tickFree (ev e # t1) ∧ set (ev e # t1) ∩ ev ‘ A = {}›
    by (simp add: image-iff ∗(3, 4) that(3))
  show ‹s ∈ 𝒟 ?lhs›
    by (simp add: D-After D-Throw ready-set-Throw that(2))
       (metis ∗(1, 5) ∗∗ ∗∗∗ Cons-eq-appendI list.discI list.sel(1, 3))
 next
  assume ‹∃t1 a t2. s = t1 @ ev a # t2 ∧ t1 @ [ev a] ∈ 𝒯 (P after e) ∧
                set t1 ∩ ev ‘ A = {} ∧ a ∈ A ∧ t2 ∈ 𝒟 (Q a)›
  then obtain t1 a t2
    where ∗ : ‹s = t1 @ ev a # t2› ‹t1 @ [ev a] ∈ 𝒯 (P after e)›
          ‹set t1 ∩ ev ‘ A = {}› ‹a ∈ A› ‹t2 ∈ 𝒟 (Q a)› by blast
  from ∗(2) that(2) have ∗∗ : ‹ev e # t1 @ [ev a] ∈ 𝒯 P›
    by (simp add: T-After) (metis list.exhaust-sel)
```

```
    have *** : ‹set (ev e # t1) ∩ ev ‘ A = {}›
      by (simp add: image-iff *(3) that(3))
    show ‹s ∈ 𝒟 ?lhs›
      by (simp add: D-After D-Throw ready-set-Throw that(2))
        (metis *(1, 4, 5) ** *** Cons-eq-appendI list.discI list.sel(1, 3))
  qed
next
  fix s X
  assume same-div : ‹𝒟 ?lhs = 𝒟 (Throw (P after e) A Q)›
  assume ‹(s, X) ∈ ℱ ?lhs›
  with that(2) have ‹(ev e # s, X) ∈ ℱ (P Θ a ∈ A. Q a)›
    by (simp add: F-After ready-set-Throw) (metis list.exhaust-sel)
  then consider ‹ev e # s ∈ 𝒟 (P Θ a ∈ A. Q a)›
    | ‹(ev e # s, X) ∈ ℱ P› ‹set s ∩ ev ‘ A = {}›
    | ‹∃ t1 a t2. ev e # s = t1 @ ev a # t2 ∧ t1 @ [ev a] ∈ 𝒯 P ∧
              set t1 ∩ ev ‘ A = {} ∧ a ∈ A ∧ (t2, X) ∈ ℱ (Q a)›
    by (simp add: F-Throw D-Throw) blast
  thus ‹(s, X) ∈ ℱ ((P after e) Θ a ∈ A. Q a)›
  proof cases
    assume ‹ev e # s ∈ 𝒟 (P Θ a ∈ A. Q a)›
    hence ‹s ∈ 𝒟 ?lhs› by (force simp add: D-After ready-set-Throw that(2))
    with same-div D-F show ‹(s, X) ∈ ℱ ((P after e) Θ a ∈ A. Q a)› by blast
  next
    show ‹(ev e # s, X) ∈ ℱ P ⟹ set s ∩ ev ‘ A = {} ⟹
        (s, X) ∈ ℱ ((P after e) Θ a ∈ A. Q a)›
      by (simp add: F-Throw F-After that(2))
        (metis list.distinct(1) list.sel(1, 3))
  next
    assume ‹∃ t1 a t2. ev e # s = t1 @ ev a # t2 ∧ t1 @ [ev a] ∈ 𝒯 P ∧
                set t1 ∩ ev ‘ A = {} ∧ a ∈ A ∧ (t2, X) ∈ ℱ (Q a)›
    then obtain t1 a t2
      where * : ‹ev e # s = t1 @ ev a # t2› ‹t1 @ [ev a] ∈ 𝒯 P›
          ‹set t1 ∩ ev ‘ A = {}› ‹a ∈ A› ‹(t2, X) ∈ ℱ (Q a)› by blast
    have ‹e ≠ a› using *(4) that(3) by blast
    with *(1) obtain t1' where ‹t1 = ev e # t1'› by (cases t1) auto
    also have ‹t1' @ [ev a] ∈ 𝒯 (P after e) ∧ set t1' ∩ ev ‘ A = {}›
      using *(2, 3) by (simp add: image-iff T-After that(2) calculation)
                (metis list.distinct(1) list.sel(1, 3))
    ultimately show ‹(s, X) ∈ ℱ ((P after e) Θ a ∈ A. Q a)›
      using *(1, 4, 5) by (simp add: F-Throw) blast
  qed
next
  fix s X
  assume same-div : ‹𝒟 ?lhs = 𝒟 (Throw (P after e) A Q)›
  assume ‹(s, X) ∈ ℱ ((P after e) Θ a ∈ A. Q a)›
  then consider ‹s ∈ 𝒟 ((P after e) Θ a ∈ A. Q a)›
    | ‹(s, X) ∈ ℱ (P after e)› ‹set s ∩ ev ‘ A = {}›
    | ‹∃ t1 a t2. s = t1 @ ev a # t2 ∧ t1 @ [ev a] ∈ 𝒯 (P after e) ∧
              set t1 ∩ ev ‘ A = {} ∧ a ∈ A ∧ (t2, X) ∈ ℱ (Q a)›
```

```
    by (simp add: F-Throw D-Throw) blast
  thus ‹(s, X) ∈ ℱ ?lhs›
  proof cases
    show ‹s ∈ 𝒟 (Throw (P after e) A Q) ⟹ (s, X) ∈ ℱ ?lhs›
      using same-div D-F by blast
  next
    assume assms : ‹(s, X) ∈ ℱ (P after e)› ‹set s ∩ ev ‘ A = {}›
    from assms(2) have ∗ : ‹set (ev e # s) ∩ ev ‘ A = {}›
      by (simp add: image-iff that(3) assms(2))
    from assms(1) have ‹(ev e # s, X) ∈ ℱ P›
      by (simp add: F-After that(2)) (metis list.exhaust-sel)
    thus ‹(s, X) ∈ ℱ ?lhs›
      by (simp add: F-After F-Throw ready-set-Throw that(2))
        (metis ∗ list.discI list.sel(1, 3))
  next
    assume ‹∃ t1 a t2. s = t1 @ ev a # t2 ∧ t1 @ [ev a] ∈ 𝒯 (P after e) ∧
                    set t1 ∩ ev ‘ A = {} ∧ a ∈ A ∧ (t2, X) ∈ ℱ (Q a)›
    then obtain t1 a t2
      where ∗ : ‹s = t1 @ ev a # t2› ‹t1 @ [ev a] ∈ 𝒯 (P after e)›
            ‹set t1 ∩ ev ‘ A = {}› ‹a ∈ A› ‹(t2, X) ∈ ℱ (Q a)› by blast
    from ∗(2) that(2) have ∗∗ : ‹ev e # t1 @ [ev a] ∈ 𝒯 P›
      by (simp add: T-After) (metis list.exhaust-sel)
    have ∗∗∗ : ‹set (ev e # t1) ∩ ev ‘ A = {}›
      by (simp add: image-iff ∗(3) that(3))
    from ∗(1, 4, 5) show ‹(s, X) ∈ ℱ ?lhs›
      by (simp add: F-After F-Throw ready-set-Throw that(2))
        (metis ∗∗ ∗∗∗ append-Cons list.distinct(1) list.sel(1, 3))
  qed
qed

ultimately show ‹?lhs = ?rhs›
  by (simp add: Throw-BOT After-BOT STOP-iff-T T-After ready-set-Throw)
qed
```

4.7.3 *After* Interrupting

```
theorem After-Interrupt:
  ‹(P △ Q) after e =
   (   if ev e ∈ ready-set P ∧ ev e ∈ ready-set Q
    then (Q after e) ⊓ (P after e △ Q)
    else   if ev e ∈ ready-set P ∧ ev e ∉ ready-set Q
         then P after e △ Q
         else  if ev e ∉ ready-set P ∧ ev e ∈ ready-set Q
              then Q after e
              else STOP)›
proof −
  have ‹(P △ Q) after e ⊑FD Q after e› if ready: ‹ev e ∈ ready-set Q›
  proof (unfold failure-divergence-refine-def le-ref-def, safe)
    fix s
```

assume ‹$s \in \mathcal{D}\ (Q\ after\ e)$›
hence ‹$ev\ e\ \#\ s \in \mathcal{D}\ Q$›
by (*simp add*: *D-After ready*) (*metis list.exhaust-sel*)
thus ‹$s \in \mathcal{D}\ ((P \triangle Q)\ after\ e)$›
by (*simp add*: *D-After ready ready-set-Interrupt D-Interrupt*)
(*metis Nil-elem-T append-Nil list.distinct*(*1*) *list.sel*(*1*, *3*) *tickFree-Nil*)
next
show ‹$(s, X) \in \mathcal{F}\ (Q\ after\ e) \Longrightarrow (s, X) \in \mathcal{F}\ ((P \triangle Q)\ after\ e)$› **for** *s X*
by (*simp add*: *F-After ready-set-Interrupt ready F-Interrupt*)
(*metis Nil-elem-T append-Nil tickFree-Nil*)
qed

moreover have ‹$(P \triangle Q)\ after\ e \sqsubseteq_{FD} P\ after\ e \triangle Q$› **if** *ready*: ‹$ev\ e \in ready\text{-}set\ P$›
proof (*unfold failure-divergence-refine-def le-ref-def*, *safe*)
show ‹$s \in \mathcal{D}\ (P\ after\ e \triangle Q) \Longrightarrow s \in \mathcal{D}\ ((P \triangle Q)\ after\ e)$› **for** *s*
apply (*simp add*: *D-Interrupt D-After ready T-After ready-set-Interrupt*,
elim disjE)
by *blast* (*metis append-Cons event.simps*(*3*) *list.sel*(*1*, *3*) *neq-Nil-conv tick-Free-Cons*)
next
show ‹$(s, X) \in \mathcal{F}\ (P\ after\ e \triangle Q) \Longrightarrow (s, X) \in \mathcal{F}\ ((P \triangle Q)\ after\ e)$› **for** *s X*
apply (*simp add*: *F-Interrupt F-After ready ready-set-Interrupt T-After D-After*,
elim disjE)
by (*metis* (*no-types*, *opaque-lifting*) [[*metis-verbose* = *false*]]
append-Cons list.distinct(*1*) *event.distinct*(*1*)
list.exhaust-sel list.sel(*1*, *3*) *tickFree-Cons*)+
qed

moreover have ‹$Q\ after\ e \sqsubseteq_{FD} (P \triangle Q)\ after\ e$› **if** *not-ready*: ‹$ev\ e \notin ready\text{-}set\ P$›
proof (*unfold failure-divergence-refine-def le-ref-def*, *safe*)
show ‹$s \in \mathcal{D}\ ((P \triangle Q)\ after\ e) \Longrightarrow s \in \mathcal{D}\ (Q\ after\ e)$› **for** *s*
by (*simp add*: *D-After not-ready ready-set-Interrupt D-Interrupt*
split: *if-split-asm*)
(*metis Cons-in-T-imp-elem-ready-set D-T not-ready*
append-Nil hd-append2 list.exhaust-sel)
next
show ‹$(s, X) \in \mathcal{F}\ ((P \triangle Q)\ after\ e) \Longrightarrow (s, X) \in \mathcal{F}\ (Q\ after\ e)$› **for** *s X*
by (*simp add*: *F-After not-ready ready-set-Interrupt F-Interrupt*
split: *if-split-asm*, *elim exE disjE conjE*)
(*metis* (*no-types*, *opaque-lifting*) [[*metis-verbose* = *false*]]
Cons-in-T-imp-elem-ready-set F-T NF-ND hd-Cons-tl
snoc-eq-iff-butlast append-self-conv2 hd-append2 not-ready)+
qed

moreover have ‹$P\ after\ e \triangle Q \sqsubseteq_{FD} (P \triangle Q)\ after\ e$›
if *ready-hyps*: ‹$ev\ e \in ready\text{-}set\ P$› ‹$ev\ e \notin ready\text{-}set\ Q$›

proof (*unfold failure-divergence-refine-def le-ref-def*, *safe*)
show ‹$s \in \mathcal{D}\ ((P \triangle Q)\ after\ e) \Longrightarrow s \in \mathcal{D}\ (P\ after\ e \triangle Q)$› **for** s
by (*simp add*: *D-After T-After ready-hyps ready-set-Interrupt D-Interrupt*)
(*metis tickFree-tl Cons-in-T-imp-elem-ready-set D-T eq-Nil-appendI*
hd-append list.exhaust-sel ready-hyps(*2*) *tl-append-if*)
next
fix *s X*
assume ‹$(s, X) \in \mathcal{F}\ ((P \triangle Q)\ after\ e)$›
with *ready-hyps*(*1*) **have** ‹$(ev\ e\ \#\ s, X) \in \mathcal{F}\ (P \triangle Q)$›
by (*simp add*: *F-After ready-set-Interrupt*) (*metis list.exhaust-sel*)
thus ‹$(s, X) \in \mathcal{F}\ (P\ after\ e \triangle Q)$›
by (*simp add*: *F-Interrupt F-After D-After T-After ready-hyps*, *elim disjE*)
(*metis* (*no-types*, *opaque-lifting*) [[*metis-verbose* = *false*]] *tickFree-tl*
Cons-in-T-imp-elem-ready-set F-T NT-ND append-self-conv2 hd-append
event.distinct(*1*) *list.sel*(*1*, *3*) *list.distinct*(*1*) *ready-hyps*(*2*) *tl-append2*)+
qed

moreover have ‹$(Q\ after\ e) \sqcap (P\ after\ e \triangle Q) \sqsubseteq_{FD} (P \triangle Q)\ after\ e$›
if *both-ready*: ‹$ev\ e \in ready\text{-}set\ P$› ‹$ev\ e \in ready\text{-}set\ Q$›
proof (*unfold failure-divergence-refine-def le-ref-def*, *safe*)
fix *s*
assume ‹$s \in \mathcal{D}\ ((P \triangle Q)\ after\ e)$›
with *both-ready*(*1*) **have** ‹$ev\ e\ \#\ s \in \mathcal{D}\ (P \triangle Q)$›
by (*simp add*: *D-After ready-set-Interrupt*) (*metis list.exhaust-sel*)
thus ‹$s \in \mathcal{D}\ ((Q\ after\ e) \sqcap (P\ after\ e \triangle Q))$›
by (*simp add*: *D-Interrupt D-After T-After D-Ndet both-ready*)
(*metis tickFree-tl append-Cons append-Nil*
list.distinct(*1*) *list.exhaust-sel list.sel*(*1*, *3*))
next
fix *s X*
assume ‹$(s, X) \in \mathcal{F}\ ((P \triangle Q)\ after\ e)$›
with *both-ready*(*1*) **have** ‹$(ev\ e\ \#\ s, X) \in \mathcal{F}\ (P \triangle Q)$›
by (*simp add*: *F-After ready-set-Interrupt*) (*metis list.exhaust-sel*)
thus ‹$(s, X) \in \mathcal{F}\ ((Q\ after\ e) \sqcap (P\ after\ e \triangle Q))$›
by (*simp add*: *F-Interrupt F-After F-Ndet D-After T-After both-ready*, *elim disjE*)
(*metis* (*no-types*, *opaque-lifting*) [[*metis-verbose* = *false*]]
event.distinct(*1*) *hd-append2 list.distinct*(*1*) *list.sel*(*1*, *3*)
process-charn self-append-conv2 tickFree-tl tl-append2)+
qed

ultimately show *?thesis*
by (*auto simp add*: *STOP-FD-iff not-ready-After intro*: *FD-antisym*)
(*metis mono-Ndet-FD FD-antisym Ndet-id*)
qed

4.8 Behaviour of *After* with Reference Processes

lemma *After-DF*: ‹$DF\ A\ after\ e = (if\ e \in A\ then\ DF\ A\ else\ STOP)$›

by (*subst DF-unfold*, *subst After-Mndetprefix*, *simp*)

lemma *After-DF*$_{SKIP}$:
‹*DF*$_{SKIP}$ *A after e* = (*if e* ∈ *A then DF*$_{SKIP}$ *A else STOP*)›
by (*subst DF*$_{SKIP}$*-unfold*)
(*simp add*: *ready-set-SKIP ready-set-Mndetprefix After-Ndet After-Mndetprefix*)

lemma *After-RUN*: ‹*RUN A after e* = (*if e* ∈ *A then RUN A else STOP*)›
by (*subst RUN-unfold*, *subst After-Mprefix*, *simp*)

lemma *After-CHAOS*: ‹*CHAOS A after e* = (*if e* ∈ *A then CHAOS A else STOP*)›
by (*subst CHAOS-unfold*)
(*simp add*: *After-Ndet ready-set-STOP ready-set-Mprefix After-Mprefix*)

lemma *After-CHAOS*$_{SKIP}$:
‹*CHAOS*$_{SKIP}$ *A after e* = (*if e* ∈ *A then CHAOS*$_{SKIP}$ *A else STOP*)›
by (*subst CHAOS*$_{SKIP}$*-unfold*)
(*simp add*: *ready-set-Ndet ready-set-STOP ready-set-SKIP*
ready-set-Mprefix After-Ndet After-Mprefix)

lemma *DF-FD-After*: ‹*DF A* ⊑$_{FD}$ *P after e*›
if ‹*DF A* ⊑$_{FD}$ *P*› **and** ‹*ev e* ∈ *ready-set P*›
proof −
have ‹*DF A after e* ⊑$_{FD}$ *P after e*› **by** (*rule mono-After-FD*[*OF that*(*1*)])
(*use that*(*2*) **in** ‹*simp add*: *ready-set-DF image-iff*›)
also have ‹*e* ∈ *A*›
by (*metis After-DF DF-Univ-freeness DF-unfold STOP-FD-iff UNIV-I deadlock-free-def empty-iff*
mono-After-FD mt-Mndetprefix non-deadlock-free-STOP ready-set-STOP that)
ultimately show ‹*DF A* ⊑$_{FD}$ *P after e*› **by** (*subst* (*asm*) *After-DF*, *simp split*: *if-splits*)
qed

lemma *DF*$_{SKIP}$*-FD-After*: ‹*DF*$_{SKIP}$ *A* ⊑$_{FD}$ *P after e*›
if ‹*DF*$_{SKIP}$ *A* ⊑$_{FD}$ *P*› **and** ‹*ev e* ∈ *ready-set P*›
proof −
have ‹*DF*$_{SKIP}$ *A after e* ⊑$_{FD}$ *P after e*› **by** (*rule mono-After-FD*[*OF that*(*1*)])
(*use that*(*2*) **in** ‹*simp add*: *ready-set-DF image-iff*›)
also have ‹*e* ∈ *A*› **using** *anti-mono-ready-set-FD ready-set-DF*$_{SKIP}$ *that* **by** *fastforce*

ultimately show ‹*DF*$_{SKIP}$ *A* ⊑$_{FD}$ *P after e*› **by** (*subst* (*asm*) *After-DF*$_{SKIP}$, *simp split*: *if-splits*)
qed

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

corollary *deadlock-free-After*:
‹*deadlock-free P* ⟹
deadlock-free (*P after e*) ⟷ (*if ev e* ∈ *ready-set P then True else False*)›
apply (*simp add*: *non-deadlock-free-STOP not-ready-After*)
unfolding *deadlock-free-def* **by** (*intro impI DF-FD-After*)

corollary *deadlock-free*$_{SKIP}$*-After*:
‹*deadlock-free*$_{SKIP}$ *P* ⟹
deadlock-free$_{SKIP}$ (*P after e*) ⟷ (*if ev e* ∈ *ready-set P then True else False*)›
apply (*simp add*: *non-deadlock-free*$_{SKIP}$*-STOP not-ready-After*)
unfolding *deadlock-free*$_{SKIP}$*-FD* **by** (*intro impI DF*$_{SKIP}$*-FD-After*)

end

Chapter 5

Extension of the After Operator

5.1 The AfterExt Operator

theory *AfterExt*
imports *After*
begin

5.1.1 Definition

We just defined *P after e* for $P::'\alpha$ *process* and $e::'\alpha$; in other words we cannot handle ✓. We now introduce a generalisation for $e::'\alpha$ *event*.

definition *AfterExt* :: ‹['α process, 'α event] ⇒ 'α process› (**infixl** ‹*afterExt*› 77)
where ‹*P afterExt e* ≡ *case e of ev x* ⇒ *P after x*
| ✓ ⇒ *if P* = ⊥ *then* ⊥ *else STOP*›

lemma *not-ready-AfterExt*:
‹*e* ∉ *ready-set P* ⟹ *P afterExt e* = (*if P* = ⊥ *then* ⊥ *else STOP*)›
by (*auto simp add*: *AfterExt-def After-BOT intro*: *not-ready-After split*: *event.split*)

lemma *ready-set-AfterExt*:
‹*ready-set* (*P afterExt e*) =
(*if P* = ⊥ *then UNIV else if e* = ✓ *then* {} *else* {*a*. *e* # [*a*] ∈ 𝒯 *P*})›
by (*simp add*: *AfterExt-def ready-set-After ready-set-BOT*
ready-set-STOP T-UU front-tickFree-butlast
split: *event.split*)

5.1.2 Projections

lemma *F-AfterExt*:
‹ℱ (*P afterExt e*) =
(*if e* = ✓ ∧ *P* = ⊥ *then* {(*s*, *X*). *front-tickFree s*}

else if $e \in$ *ready-set* P *then* $\{(tl\ s,\ X) |\ s\ X.\ (s,\ X) \in \mathcal{F}\ P \wedge s \neq [] \wedge hd\ s = e\}$
else $\{(s,\ X).\ s = []\}$›
(**is** ‹- = *?rhs*›)
proof (*unfold AfterExt-def*, *split event.split*, *intro conjI allI impI*)
show ‹$e = ev\ x \Longrightarrow \mathcal{F}\ (P\ after\ x) =$ *?rhs*› **for** x
by (*simp add*: *F-After*)
next
show ‹$e = \checkmark \Longrightarrow \mathcal{F}$ (*if* $P = \bot$ *then* $\bot$ *else STOP*) = *?rhs*›
by (*simp add*: *F-UU F-STOP BOT-iff-D ready-set-def set-eq-iff*)
(*metis F-T T-nonTickFree-imp-decomp append-single-T-imp-tickFree hd-append2*
hd-in-set list.discI list.sel(*1*) *list.sel*(*3*) *tickFree-def tick-T-F*)
qed

lemma *D-AfterExt*:
‹$\mathcal{D}$ (P *afterExt* e) = (*if* $e = \checkmark \wedge P = \bot$ *then* $\{s.$ *front-tickFree* $s\}$
else $\{tl\ s |\ s\ .\ s \in \mathcal{D}\ P \wedge s \neq [] \wedge hd\ s = e\}$)›
(**is** ‹- = *?rhs*›)
proof (*unfold AfterExt-def*, *split event.split*, *intro conjI allI impI*)
show ‹$e = ev\ x \Longrightarrow \mathcal{D}\ (P\ after\ x) =$ *?rhs*› **for** x
by (*simp add*: *D-After*)
(*metis Cons-in-T-imp-elem-ready-set D-T list.exhaust-sel*)
next
show ‹$e = \checkmark \Longrightarrow \mathcal{D}$ (*if* $P = \bot$ *then* $\bot$ *else STOP*) = *?rhs*›
by (*simp add*: *D-UU D-STOP BOT-iff-D*)
(*metis D-imp-front-tickFree front-tickFree-implies-tickFree hd-append2*
hd-in-set is-processT9 nonTickFree-n-frontTickFree tickFree-def)
qed

lemma *T-AfterExt*:
‹$\mathcal{T}$ (P *afterExt* e) = (*if* $e = \checkmark \wedge P = \bot$ *then* $\{s.$ *front-tickFree* $s\}$
else insert $[]$ $\{tl\ s |\ s\ .\ s \in \mathcal{T}\ P \wedge s \neq [] \wedge hd\ s = e\}$)›
(**is** ‹- = *?rhs*›)
proof (*unfold AfterExt-def*, *split event.split*, *intro conjI allI impI*)
show ‹$e = ev\ x \Longrightarrow \mathcal{T}\ (P\ after\ x) =$ *?rhs*› **for** x
by (*simp add*: *T-After set-eq-iff subset-iff*)
(*metis Cons-in-T-imp-elem-ready-set list.collapse*
list.distinct(*1*) *list.sel*(*1*, *3*) *mem-Collect-eq ready-set-def*)
next
show ‹$e = \checkmark \Longrightarrow \mathcal{T}$ (*if* $P = \bot$ *then* $\bot$ *else STOP*) = *?rhs*›
by (*simp add*: *T-After T-UU T-STOP subset-iff*)
(*metis front-tickFree-charn hd-append2 hd-in-set*
is-processT2-TR list.sel(*3*) *tickFree-def tl-append-if*)
qed

5.1.3 Monotony

lemma *mono-AfterExt* : ‹$P \sqsubseteq Q \Longrightarrow P$ *afterExt* $e \sqsubseteq Q$ *afterExt* e›

by (*auto simp add*: *AfterExt-def mono-After split*: *event.split*)

lemma *mono-AfterExt-T* : ‹$P \sqsubseteq_T Q \Longrightarrow e \neq \checkmark \vee P = \bot \vee Q \neq \bot \Longrightarrow$
P *afterExt* $e \sqsubseteq_T Q$ *afterExt* e›
by (*auto simp add*: *AfterExt-def mono-After-T split*: *event.split*)

lemma *mono-AfterExt-F* :
‹$P \sqsubseteq_F Q \Longrightarrow ev\ e \notin$ *ready-set* $P \vee ev\ e \in$ *ready-set* $Q \Longrightarrow$
P *afterExt* $ev\ e \sqsubseteq_F Q$ *afterExt* $ev\ e$›
by (*simp add*: *AfterExt-def mono-After-F*)

lemma *mono-AfterExt-D* : ‹$P \sqsubseteq_D Q \Longrightarrow P$ *afterExt* $e \sqsubseteq_D Q$ *afterExt* e›
by (*auto simp add*: *AfterExt-def mono-After-D split*: *event.split*)
(*meson BOT-iff-D divergence-refine-def subset-iff*)

lemma *mono-AfterExt-FD* :
‹$P \sqsubseteq_{FD} Q \Longrightarrow e \notin$ *ready-set* $P \vee e \in$ *ready-set* $Q \Longrightarrow$
P *afterExt* $e \sqsubseteq_{FD} Q$ *afterExt* e›
by (*auto simp add*: *AfterExt-def mono-After-FD FD-antisym split*: *event.split*)

lemma *mono-AfterExt-DT* : ‹$P \sqsubseteq_{DT} Q \Longrightarrow P$ *afterExt* $e \sqsubseteq_{DT} Q$ *afterExt* e›
by (*auto simp add*: *AfterExt-def mono-After-DT split*: *event.split*)
(*meson BOT-iff-D divergence-refine-def leDT-imp-leD subsetD*)

5.1.4 Behaviour of *AfterExt* with *STOP*, *SKIP* and $\bot$

lemma *AfterExt-STOP*: ‹*STOP afterExt* e = *STOP*›
by (*simp add*: *STOP-neq-BOT STOP-iff-T T-AfterExt ready-set-STOP T-STOP*)

lemma *AfterExt-is-STOP-iff*:
‹P *afterExt* e = *STOP* $\longleftrightarrow P \neq \bot \wedge (\forall s.\ e\ \#\ s \in \mathcal{T}\ P \longrightarrow s = [])$›
apply (*cases e*; *simp add*: *AfterExt-def After-is-STOP-iff*)
by (*metis After-BOT After-is-STOP-iff STOP-neq-BOT*)
(*metis STOP-neq-BOT butlast.simps*(*2*) *front-tickFree-butlast is-processT2-TR tickFree-Cons*)

lemma *AfterExt-SKIP*: ‹*SKIP afterExt* e = *STOP*›
by (*auto simp add*: *SKIP-neq-BOT STOP-iff-T T-AfterExt ready-set-SKIP T-SKIP*)

lemma *AfterExt-BOT* : ‹$\bot$ *afterExt* $e = \bot$›
by (*force simp add*: *BOT-iff-D D-AfterExt D-UU*)

lemma *AfterExt-is-BOT-iff*: ‹P *afterExt* $e = \bot \longleftrightarrow [e] \in \mathcal{D}\ P$›
apply (*cases e*; *simp add*: *AfterExt-def After-is-BOT-iff*)
using *BOT-iff-D D-UU STOP-neq-BOT is-processT9-tick* **by** *fastforce*

5.1.5 Behaviour of *AfterExt* with Operators of HOL-CSP

Here again, we lose determinism.

lemma *AfterExt-Mprefix-is-AfterExt-Mndetprefix*:
‹(□a ∈ A → P a) afterExt e = (⊓a ∈ A → P a) afterExt e›
by (*simp add*: *AfterExt-def After-Mprefix-is-After-Mndetprefix*
Mprefix-neq-BOT Mndetprefix-neq-BOT split: *event.split*)

lemma *AfterExt-Det-is-AfterExt-Ndet*: ‹P □ Q afterExt e = P ⊓ Q afterExt e›
by (*simp add*: *AfterExt-def After-Det-is-After-Ndet Det-is-BOT-iff Ndet-is-BOT-iff*
split: *event.split*)

lemma *AfterExt-Ndet*:
‹P ⊓ Q afterExt e = (if e ∉ ready-set P ∪ ready-set Q then STOP
else case e of ev x ⇒ if ev x ∈ ready-set P ∩ ready-set Q
then (P afterExt ev x) ⊓ (Q afterExt ev x)
else if ev x ∈ ready-set P
then P afterExt ev x
else Q afterExt ev x
| ✓ ⇒ if P = ⊥ ∨ Q = ⊥ then ⊥ else STOP)›
by (*simp add*: *AfterExt-def Ndet-is-BOT-iff After-Ndet ready-set-BOT split*: *event.split*)

lemma *AfterExt-Mprefix*:
‹(□ a ∈ A → P a) afterExt e =
(case e of ev x ⇒ if x ∈ A then P x else STOP | ✓ ⇒ STOP)›
by (*simp add*: *AfterExt-def Mprefix-neq-BOT After-Mprefix split*: *event.split*)

corollary *AfterExt-prefix*:
‹(a → P) afterExt e =
(case e of ev x ⇒ if x = a then P else STOP | ✓ ⇒ STOP)›
unfolding *write0-def* **by** (*simp add*: *AfterExt-Mprefix split*: *event.split*)

lemmas *AfterExt-Det = AfterExt-Ndet*[*folded AfterExt-Det-is-AfterExt-Ndet*]
and *AfterExt-Mndetprefix = AfterExt-Mprefix*
[*unfolded AfterExt-Mprefix-is-AfterExt-Mndetprefix*]

lemma *Renaming-is-BOT-iff*: ‹*Renaming P f* = ⊥ ⟷ *P* = ⊥›
apply (*intro iffI*)
apply (*simp add*: *BOT-iff-D D-Renaming*)
by (*simp add*: *Renaming-BOT*)

lemma *Renaming-is-STOP-iff*: ‹*Renaming P f* = *STOP* ⟷ *P* = *STOP*›
apply (*intro iffI*, *simp-all add*: *Renaming-STOP*)

by (*auto simp add*: *STOP-iff-T T-Renaming intro*: *Nil-elem-T*)

lemma *AfterExt-Renaming*:
‹*Renaming P f afterExt e* =
(*if P* = ⊥ *then* ⊥ *else*
⊓*a* ∈ {*a*. *ev a* ∈ *ready-set P* ∧ *ev* (*f a*) = *e*}. *Renaming* (*P afterExt ev a*) *f*)›
by (*simp add*: *AfterExt-def After-Renaming Renaming-is-BOT-iff split*: *event.split*)

— Move this result in `HOL-CSP`
lemma *Seq-is-BOT-iff*: ‹*P* **;** *Q* = ⊥ ⟷ *P* = ⊥ ∨ ([✓] ∈ 𝒯 *P* ∧ *Q* = ⊥)›
by (*auto simp add*: *BOT-iff-D D-Seq D-UU*)

lemma *AfterExt-Seq*:
‹(*P* **;** *Q*) *afterExt e* =
(*if e* ∉ *ready-set P* ∧ *e* ∉ *ready-set Q then STOP*
else if e ∉ *ready-set Q then P afterExt e* **;** *Q*
else if e ∉ *ready-set P then if* ✓ ∈ *ready-set P then Q afterExt e else STOP*
else if ✓ ∈ *ready-set P then* (*P afterExt e* **;** *Q*) ⊓ (*Q afterExt e*)
else P afterExt e **;** *Q*)›
by (*simp add*: *AfterExt-def After-Seq Ndet-id Ndet-is-BOT-iff*
Seq-is-BOT-iff ready-set-def T-UU STOP-Seq split: *event.split*)

theorem *AfterExt-Sync*:
‹(*P* ⟦*S*⟧ *Q*) *afterExt e* =
(*if P* = ⊥ ∨ *Q* = ⊥ *then* ⊥
else case e of ✓ ⇒ *STOP*
| *ev x* ⇒ *if e* ∈ *ready-set P* ∧ *e* ∈ *ready-set Q*
then if x ∈ *S*
then P afterExt e ⟦*S*⟧ *Q afterExt e*
else (*P afterExt e* ⟦*S*⟧ *Q*) ⊓ (*P* ⟦*S*⟧ *Q afterExt e*)
else if e ∈ *ready-set P*
then if x ∈ *S then STOP else P afterExt e* ⟦*S*⟧ *Q*
else if e ∈ *ready-set Q*
then if x ∈ *S then STOP else P* ⟦*S*⟧ *Q afterExt e*
else STOP)›
by (*simp add*: *AfterExt-def After-Sync Sync-is-BOT-iff split*: *event.split*)

lemma *Hiding-FD-Hiding-AfterExt-if-ready-inside*:
‹*e* ∈ *B* ⟹ (*P* \ *B*) ⊑$_{FD}$ (*P afterExt ev e* \ *B*)›
and *AfterExt-Hiding-FD-Hiding-AfterExt-if-ready-notin*:
‹*e* ∉ *B* ⟹ (*P* \ *B*) *afterExt ev e* ⊑$_{FD}$ (*P afterExt ev e* \ *B*)›
if *ready*: ‹*ev e* ∈ *ready-set P*›

by (*simp add*: *AfterExt-def Hiding-FD-Hiding-After-if-ready-inside that*)
(*simp add*: *AfterExt-def After-Hiding-FD-Hiding-After-if-ready-notin that*)

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

and *AfterExt-Hiding-T-Hiding-AfterExt-if-ready-notin* =
AfterExt-Hiding-FD-Hiding-AfterExt-if-ready-notin[*THEN leFD-imp-leF*,
THEN leF-imp-leT]

corollary *Hiding-DT-Hiding-AfterExt-if-ready-inside*:
‹*ev e* ∈ *ready-set P* ⟹ *e* ∈ *B* ⟹ (*P* \ *B*) ⊑$_{DT}$ (*P afterExt ev e* \ *B*)›
and *AfterExt-Hiding-DT-Hiding-AfterExt-if-ready-notin*:
‹*ev e* ∈ *ready-set P* ⟹ *e* ∉ *B* ⟹
(*P* \ *B*) *afterExt ev e* ⊑$_{DT}$ (*P afterExt ev e* \ *B*)›
by (*simp add*: *Hiding-D-Hiding-AfterExt-if-ready-inside*
Hiding-T-Hiding-AfterExt-if-ready-inside leD-leT-imp-leDT)
(*simp add*: *AfterExt-Hiding-D-Hiding-AfterExt-if-ready-notin*
AfterExt-Hiding-T-Hiding-AfterExt-if-ready-notin leD-leT-imp-leDT)

5.1.6 Behaviour of *AfterExt* with Operators of HOL-CSPM

lemma *AfterExt-MultiDet-is-AfterExt-MultiNdet*:
‹*finite A* ⟹ (□ *a* ∈ *A*. *P a*) *afterExt e* = (⨅ *a* ∈ *A*. *P a*) *afterExt e*›
by (*auto simp add*: *AfterExt-def After-MultiDet-is-After-MultiNdet*
MultiDet-is-BOT-iff MultiNdet-is-BOT-iff split: *event.split*)

lemma *AfterExt-GlobalNdet*:
‹(⊓ *a* ∈ *A*. *P a*) *afterExt e* = (*if e* ∉ (⋃ *a* ∈ *A*. *ready-set* (*P a*)) *then STOP*
else (⊓ *a* ∈ {*a* ∈ *A*. *e* ∈ *ready-set* (*P a*)}. *P a*) *afterExt e*)›
and *AfterExt-MultiNdet*:
‹*finite A* ⟹ (⨅ *a* ∈ *A*. *P a*) *afterExt e* =
(*if e* ∉ (⋃ *a* ∈ *A*. *ready-set* (*P a*)) *then STOP*
else (⨅ *a* ∈ {*a* ∈ *A*. *e* ∈ *ready-set* (*P a*)}. *P a*) *afterExt e*)›
and *AfterExt-MultiDet*:
‹*finite A* ⟹ (□ *a* ∈ *A*. *P a*) *afterExt e* =
(*if e* ∉ (⋃ *a* ∈ *A*. *ready-set* (*P a*)) *then STOP*
else (⨅ *a* ∈ {*a* ∈ *A*. *e* ∈ *ready-set* (*P a*)}. *P a*) *afterExt e*)›

by ((*cases e*, *simp-all add*: *AfterExt-def*,
auto simp add: *AfterExt-def ready-set-BOT After-GlobalNdet GlobalNdet-is-BOT-iff*

After-MultiNdet After-MultiDet MultiDet-is-BOT-iff MultiN-
det-is-BOT-iff,
metis UNIV-I ready-set-BOT)[*1*])+

5.1.7 Behaviour of *AfterExt* with Operators of HOL-CSP_OpSem

lemma *AfterExt-Sliding-is-AfterExt-Ndet*:
‹$P \rhd Q$ *afterExt* $e = P \sqcap Q$ *afterExt* e›
by (*simp add*: *AfterExt-def After-Sliding-is-After-Ndet Sliding-is-BOT-iff Ndet-is-BOT-iff*
split: *event.split*)

lemmas *AfterExt-Sliding* = *AfterExt-Ndet*[*folded AfterExt-Sliding-is-AfterExt-Ndet*]

lemma *AfterExt-Throw*:
‹$(P \Theta\ a \in A.\ Q\ a)$ *afterExt* $e =$
(*if* $P = \bot$ *then* $\bot$
else case e *of* ✓ $\Rightarrow$ *STOP*
| $ev\ x \Rightarrow$ *if* $ev\ x \in$ *ready-set* P *then if* $x \in A$ *then* $Q\ x$
else $(P$ *after* $x) \Theta\ a \in A.\ Q\ a$ *else STOP*)›
by (*simp add*: *AfterExt-def After-Throw Throw-is-BOT-iff split*: *event.split*)

lemma *AfterExt-Interrupt*:
‹$(P \triangle Q)$ *afterExt* $e =$
(*if* $P = \bot \vee Q = \bot$ *then* $\bot$
else case e *of* ✓ $\Rightarrow$ *STOP*
| $ev\ x \Rightarrow$ *if* $ev\ x \in$ *ready-set* $P \wedge ev\ x \in$ *ready-set* Q
then $(Q$ *after* $x) \sqcap (P$ *after* $x \triangle Q)$
else if $ev\ x \in$ *ready-set* $P \wedge ev\ x \notin$ *ready-set* Q
then P *after* $x \triangle Q$
else if $ev\ x \notin$ *ready-set* $P \wedge ev\ x \in$ *ready-set* Q
then Q *after* x
else STOP)›
by (*simp add*: *AfterExt-def After-Interrupt Interrupt-is-BOT-iff*
Ndet-is-BOT-iff ready-set-BOT After-is-BOT-iff D-UU split: *event.split*)

5.1.8 Behaviour of *AfterExt* with Reference Processes

lemma *AfterExt-DF*:
‹*DF A afterExt e* =
(*case e of ev x* $\Rightarrow$ *if* $x \in A$ *then DF A else STOP* | ✓ $\Rightarrow$ *STOP*)›
by (*cases e*) (*simp-all add*: *AfterExt-def After-DF BOT-iff-D div-free-DF*)

lemma *AfterExt-DF*$_{SKIP}$:
‹*DF*$_{SKIP}$ *A afterExt e* =
(*case e of ev x* $\Rightarrow$ *if* $x \in A$ *then DF*$_{SKIP}$ *A else STOP* | ✓ $\Rightarrow$ *STOP*)›

by (*cases e*) (*simp-all add*: *AfterExt-def After-DF$_{SKIP}$ BOT-iff-D div-free-DF$_{SKIP}$*)

lemma *AfterExt-RUN*:
‹*RUN A afterExt e* =
(*case e of ev x* ⇒ *if x* ∈ *A then RUN A else STOP* | ✓ ⇒ *STOP*)›
by (*cases e*) (*simp-all add*: *AfterExt-def After-RUN BOT-iff-D div-free-RUN*)

lemma *AfterExt-CHAOS*:
‹*CHAOS A afterExt e* =
(*case e of ev x* ⇒ *if x* ∈ *A then CHAOS A else STOP* | ✓ ⇒ *STOP*)›
by (*cases e*) (*simp-all add*: *AfterExt-def After-CHAOS BOT-iff-D div-free-CHAOS*)

lemma *AfterExt-CHAOS$_{SKIP}$*:
‹*CHAOS$_{SKIP}$ A afterExt e* =
(*case e of ev x* ⇒ *if x* ∈ *A then CHAOS$_{SKIP}$ A else STOP* | ✓ ⇒ *STOP*)›
by (*cases e*) (*simp-all add*: *AfterExt-def After-CHAOS$_{SKIP}$ BOT-iff-D div-free-CHAOS$_{SKIP}$*)

lemma *DF-FD-AfterExt*:
‹*DF A* ⊑$_{FD}$ *P* ⟹ *e* ∈ *ready-set P* ⟹ *DF A* ⊑$_{FD}$ *P afterExt e*›
apply (*cases e*, *simp add*: *AfterExt-def DF-FD-After*)
by (*metis anti-mono-ready-set-FD event.distinct*(*1*) *image-iff ready-set-DF subsetD*)

lemma *DF$_{SKIP}$-FD-AfterExt*:
‹*DF$_{SKIP}$ A* ⊑$_{FD}$ *P* ⟹ *ev e* ∈ *ready-set P* ⟹ *DF$_{SKIP}$ A* ⊑$_{FD}$ *P afterExt ev e*›
by (*simp add*: *AfterExt-def DF$_{SKIP}$-FD-After*)

lemma *deadlock-free-AfterExt*:
‹*deadlock-free P* ⟹ *deadlock-free* (*P afterExt e*) ⟷
(*if e* ∈ *ready-set P* ∧ *e* ≠ ✓ *then True else False*)›
by (*cases e*)
(*simp add*: *AfterExt-def deadlock-free-After*,
simp add: *AfterExt-def BOT-iff-D deadlock-free-implies-div-free non-deadlock-free-STOP*)

lemma *deadlock-free$_{SKIP}$-AfterExt*:
‹*deadlock-free$_{SKIP}$ P* ⟹ *deadlock-free$_{SKIP}$* (*P afterExt e*) ⟷
(*if e* ∈ *ready-set P* ∧ *e* ≠ ✓ *then True else False*)›
by (*cases e*)
(*simp add*: *AfterExt-def deadlock-free$_{SKIP}$-After*,
simp add: *AfterExt-def BOT-iff-D deadlock-free$_{SKIP}$-implies-div-free non-deadlock-free$_{SKIP}$-STOP*)

end

5.2 The AfterTrace Operator

theory *AfterTrace*
imports *AfterExt HOL−CSPM.DeadlockResults*
begin

5.2.1 Definition

We just defined *P afterExt e* for *P*::*'α process* and *e*::*'α event*. Since a trace of a *P* is just an *'α event list*, the following inductive definition is natural.

fun *AfterTrace* :: ‹*'α process* ⇒ *'α trace* ⇒ *'α process*› (**infixl** ‹*afterTrace*› *77*)
where ‹*P afterTrace* [] = *P*›
| ‹*P afterTrace* (*e* # *s*) = *P afterExt e afterTrace s*›

We can also induct backward.

lemma *AfterTrace-append*: ‹*P afterTrace* (*s* @ *t*) = *P afterTrace s afterTrace t*›
apply (*induct t rule*: *rev-induct*, *simp*)
apply (*induct s rule*: *rev-induct*, *simp*)
by (*metis AfterTrace.simps append.assoc append.right-neutral append-Cons append-Nil*)

lemma *AfterTrace-snoc* : ‹*P afterTrace* (*s* @ [*e*]) = *P afterTrace s afterExt e*›
by (*simp add*: *AfterTrace-append*)

We have some immediate properties.

lemma *AfterTrace-BOT* : ‹⊥ *afterTrace s* = ⊥›
by (*induct s*) (*simp-all add*: *AfterExt-BOT*)

lemma *AfterTrace-STOP* : ‹*STOP afterTrace s* = *STOP*›
by (*induct s*) (*simp-all add*: *AfterExt-STOP*)

lemma *AfterTrace-SKIP* : ‹*SKIP afterTrace s* = (*if s* = [] *then SKIP else STOP*)›
by (*induct s*) (*simp-all add*: *AfterExt-SKIP AfterTrace-STOP*)

5.2.2 Projections

lemma *F-AfterTrace* : ‹(*s* @ *t*, *X*) ∈ *ℱ P* ⟹ (*t*, *X*) ∈ *ℱ* (*P afterTrace s*)›
apply (*induct s arbitrary*: *t rule*: *rev-induct*, *simp*)
apply (*simp add*: *AfterTrace-snoc F-AfterExt*)
by (*metis Cons-in-T-imp-elem-ready-set F-T butlast-tl front-tickFree-butlast is-processT2 list.distinct*(*1*) *list.sel*(*1*, *3*) *tickFree-tl*)

lemma *D-AfterTrace* : ‹*s* @ *t* ∈ *𝒟 P* ⟹ *t* ∈ *𝒟* (*P afterTrace s*)›
apply (*induct s arbitrary*: *t rule*: *rev-induct*, *simp*)
apply (*simp add*: *AfterTrace-snoc D-AfterExt*)
by (*metis D-imp-front-tickFree butlast-tl front-tickFree-butlast list.distinct*(*1*) *list.sel*(*1*, *3*) *tickFree-tl*)

lemma *T-AfterTrace* : ‹s @ $t \in \mathcal{T}\ P \Longrightarrow t \in \mathcal{T}\ (P\ afterTrace\ s)$›
using *F-AfterTrace T-F-spec* **by** *blast*

corollary *ready-set-AfterTrace* :
‹s @ e # $t \in \mathcal{T}\ P \Longrightarrow e \in ready\text{-}set\ (P\ afterTrace\ s)$›
by (*metis Cons-in-T-imp-elem-ready-set T-AfterTrace*)

corollary *F-imp-R-AfterTrace*: ‹$(s, X) \in \mathcal{F}\ P \Longrightarrow X \in \mathcal{R}\ (P\ afterTrace\ s)$›
by (*simp add*: *F-AfterTrace Refusals-iff*)

corollary *D-imp-AfterTrace-is-BOT*: ‹$s \in \mathcal{D}\ P \Longrightarrow P\ afterTrace\ s = \bot$›
by (*simp add*: *BOT-iff-D D-AfterTrace*)

5.2.3 Monotony

lemma *mono-AfterTrace* : ‹$P \sqsubseteq Q \Longrightarrow P\ afterTrace\ s \sqsubseteq Q\ afterTrace\ s$›
by (*induct s rule*: *rev-induct*) (*simp-all add*: *mono-AfterExt AfterTrace-snoc*)

lemma *mono-AfterTrace-T* :
‹$P \sqsubseteq_T Q \Longrightarrow s \in \mathcal{T}\ Q \Longrightarrow tickFree\ s \Longrightarrow P\ afterTrace\ s \sqsubseteq_T Q\ afterTrace\ s$›
by (*induct s rule*: *rev-induct*; *simp add*: *AfterTrace-snoc*)
(*rule mono-AfterExt-T*; *use is-processT3-ST* **in** *blast*)

lemma *mono-AfterTrace-F* :
‹$P \sqsubseteq_F Q \Longrightarrow s \in \mathcal{T}\ Q \Longrightarrow tickFree\ s \Longrightarrow P\ afterTrace\ s \sqsubseteq_F Q\ afterTrace\ s$›
by (*induct s rule*: *rev-induct*; *simp add*: *AfterTrace-snoc*)
(*metis event.exhaust is-processT3-ST mono-AfterExt-F ready-set-AfterTrace*)

lemma *mono-AfterTrace-D* : ‹$P \sqsubseteq_D Q \Longrightarrow P\ afterTrace\ s \sqsubseteq_D Q\ afterTrace\ s$›
by (*induct s rule*: *rev-induct*) (*simp-all add*: *mono-AfterExt-D AfterTrace-snoc*)

lemma *mono-AfterTrace-FD* :
‹$P \sqsubseteq_{FD} Q \Longrightarrow s \in \mathcal{T}\ Q \Longrightarrow P\ afterTrace\ s \sqsubseteq_{FD} Q\ afterTrace\ s$›
by (*induct s rule*: *rev-induct*; *simp add*: *AfterTrace-snoc*)
(*meson Cons-in-T-imp-elem-ready-set T-AfterTrace is-processT3-ST mono-AfterExt-FD*)

lemma *mono-AfterTrace-DT* :
‹$P \sqsubseteq_{DT} Q \Longrightarrow P\ afterTrace\ s \sqsubseteq_{DT} Q\ afterTrace\ s$›
by (*induct s rule*: *rev-induct*; *simp add*: *AfterTrace-snoc mono-AfterExt-DT*)

5.2.4 Another Definition of *events-of*

inductive *reachable-event* :: ‹$'\alpha\ process \Rightarrow\ '\alpha \Rightarrow bool$›
where ‹$ev\ e \in ready\text{-}set\ P \Longrightarrow reachable\text{-}event\ P\ e$›
| ‹$reachable\text{-}event\ (P\ after\ f)\ e \Longrightarrow reachable\text{-}event\ P\ e$›

lemma *events-of-iff-reachable-event*: ‹$e \in events\text{-}of\ P \longleftrightarrow reachable\text{-}event\ P\ e$›
proof (*intro iffI*)

```
  show ‹reachable-event P e ⟹ e ∈ events-of P›
    apply (induct rule: reachable-event.induct;
          simp add: T-After events-of-def ready-set-def split: if-split-asm)
    by (meson list.set-intros(1)) (meson list.set-sel(2))
next
  assume ‹e ∈ events-of P›
  then obtain s where ∗ : ‹s ∈ 𝒯 P› ‹ev e ∈ set s› unfolding events-of-def by
blast
  thus ‹reachable-event P e›
  proof (induct s arbitrary: P)
    show ‹⋀P e. ev e ∈ set [] ⟹ reachable-event P e› by simp
  next
    case (Cons f s)
    from Cons.prems(1) is-processT3-ST
    have ∗ : ‹f ∈ ready-set P› unfolding ready-set-def by force
    from Cons.prems(2) consider ‹f = ev e› | ‹ev e ∈ set s› by auto
    thus ‹reachable-event P e›
    proof cases
      assume ‹f = ev e›
      from ∗[simplified this]
      show ‹reachable-event P e› by (rule reachable-event.intros(1))
    next
      assume ‹ev e ∈ set s›
      show ‹reachable-event P e›
      proof (cases f)
        fix f'
        assume ‹f = ev f'›
          with ∗ Cons.prems(1) have ‹s ∈ 𝒯 (P after f')› by (force simp add:
T-After)
        show ‹reachable-event P e›
          apply (rule reachable-event.intros(2))
          by (rule Cons.hyps[OF ‹s ∈ 𝒯 (P after f')› ‹ev e ∈ set s›])
      next
        from Cons.prems(1) have ‹f = ✓ ⟹ s = []›
            by simp (metis butlast.simps(2) front-tickFree-butlast is-processT2-TR
tickFree-Cons)
        thus ‹f = ✓ ⟹ reachable-event P e›
          using ‹ev e ∈ set s› by force
      qed
    qed
  qed
qed


lemma reachable-event-BOT: ‹reachable-event ⊥ e›
  by (simp add: reachable-event.intros(1) ready-set-BOT)

lemma not-reachable-event-STOP: ‹¬ reachable-event STOP e›
  by (subst events-of-iff-reachable-event[symmetric])
```

```
    (unfold events-of-def, simp add: T-STOP)

lemma reachable-tick-SKIP: ‹¬ reachable-event SKIP ✓›
  by (subst events-of-iff-reachable-event[symmetric])
    (unfold events-of-def, simp add: T-SKIP)



lemma reachable-event-iff-in-ready-set-AfterTrace:
  ‹reachable-event P e ⟷ e ∈ {e| e s. ev e ∈ ready-set (P afterTrace s)}›
proof −
  have ‹reachable-event P e ⟹ ∃ s. ev e ∈ ready-set (P afterTrace s)›
  proof (induct rule: reachable-event.induct)
    case (1 e P)
    thus ?case by (metis AfterTrace.simps(1))
  next
    case (2 P f e)
    from 2.hyps(2) obtain s where ‹ev e ∈ ready-set (P after f afterTrace s)› by
blast
    hence ‹ev e ∈ ready-set (P afterTrace (ev f # s))› by (simp add: AfterExt-def)
    thus ?case by blast
  qed
  also have ‹ev e ∈ ready-set (P afterTrace s) ⟹ reachable-event P e› for s
  proof (induct s arbitrary: P)
    show ‹ev e ∈ ready-set (P afterTrace []) ⟹ reachable-event P e› for P
      by (simp add: reachable-event.intros(1))
  next
    fix f s P
    assume  hyp : ‹ev e ∈ ready-set (P afterTrace s) ⟹ reachable-event P e› for
P
    assume prem : ‹ev e ∈ ready-set (P afterTrace (f # s))›
    show ‹reachable-event P e›
    proof (cases f)
      fix f'
      assume ‹f = ev f'›
      show ‹reachable-event P e› by (rule reachable-event.intros(2)[OF hyp])
                                    (use prem in ‹simp add: AfterExt-def ‹f = ev f'››)
    next
      case tick
      with prem show ‹f = ✓ ⟹ reachable-event P e›
      by (simp add: AfterExt-def reachable-event-BOT AfterTrace-STOP ready-set-STOP

              split: if-split-asm)
    qed
  qed
  ultimately show ?thesis by blast
qed
```

5.2.5 Characterizations for Deadlock Freeness

lemma *deadlock-free-AfterExt-characterization*:
‹*deadlock-free* $P \longleftrightarrow$ *range ev* $\notin \mathcal{R}\ P \wedge$
$(\forall\, e \in$ *ready-set* $P.$ *deadlock-free* $(P$ *afterExt* $e))$›
(**is** ‹*deadlock-free* $P \longleftrightarrow$ *?rhs*›)
proof (*intro iffI*)
have ‹*range ev* $\notin \mathcal{R}\ P \longleftrightarrow$ *UNIV* $-$ $\{\checkmark\} \notin \mathcal{R}\ P$›
by (*metis Diff-insert-absorb UNIV-eq-I event.simps*(3) *event-set image-iff*)
thus ‹*deadlock-free* $P \Longrightarrow$ *?rhs*›
by (*metis DF-FD-AfterExt Diff-Diff-Int Diff-partition Diff-subset F-T*
$\textit{deadlock-free}_{SKIP}$*-is-right deadlock-free-def*
deadlock-free-implies-non-terminating $\textit{deadlock-free-is-deadlock-free}_{SKIP}$
inf-top-left is-processT5-S2a non-tickFree-tick Refusals-iff self-append-conv2)
next
assume *assm* : ‹*?rhs*›
with *BOT-iff-D process-charn* **have** *non-BOT* : ‹$P \neq \bot$› **by** (*metis Refusals-iff*)
show ‹*deadlock-free* P›
proof (*unfold deadlock-free-F failure-refine-def*, *safe*)
from *assm* **have** $*$: ‹*range ev* $\notin \mathcal{R}\ P$›
‹$e \in$ *ready-set* $P \Longrightarrow$
$\{(tl\ s,\ X)\ |s\ X.\ (s,\ X) \in \mathcal{F}\ P \wedge s \neq [] \wedge hd\ s = e\} \subseteq \mathcal{F}\ (DF\ UNIV)$› **for** e
by (*simp-all add*: *deadlock-free-F failure-refine-def F-AfterExt non-BOT*)

fix $s\ X$
assume $**$: ‹$(s,\ X) \in \mathcal{F}\ P$›
show ‹$(s,\ X) \in \mathcal{F}\ (DF\ UNIV)$›
proof (*cases s*)
show ‹$s = [] \Longrightarrow (s,\ X) \in \mathcal{F}\ (DF\ UNIV)$›
by (*subst F-DF*, *simp*)
(*metis* $*(1)$ $**$ *Refusals-iff image-subset-iff is-processT4*)
next
fix $e\ s'$
assume $***$: ‹$s = e\ \#\ s'$›
with $**$ *Cons-in-T-imp-elem-ready-set F-T* **have** ‹$e \in$ *ready-set* P› **by** *blast*
with $*(2)$[*OF this*] **show** ‹$(s,\ X) \in \mathcal{F}\ (DF\ UNIV)$›
by (*subst F-DF*, *simp add*: *subset-iff*)
(*metis* (*no-types*, *lifting*) $*(1)$ $**$ $***$ *CollectD Refusals-iff*
event-set hd-append2 hd-in-set in-set-conv-decomp-first insert-iff
is-processT6-S2 list.sel(1) *list.set-intros*(1) *rangeE ready-set-def*)
qed
qed
qed

lemma $\textit{deadlock-free}_{SKIP}$*-AfterExt-characterization*:
‹$\textit{deadlock-free}_{SKIP}\ P \longleftrightarrow$
UNIV $\notin \mathcal{R}\ P \wedge (\forall\, e \in$ *ready-set* $P - \{\checkmark\}.\ \textit{deadlock-free}_{SKIP}\ (P$ *afterExt* $e))$›
(**is** ‹$\textit{deadlock-free}_{SKIP}\ P \longleftrightarrow$ *?rhs*›)
proof (*intro iffI*)

```
  show ‹deadlock-free_SKIP P ⟹ ?rhs›
    by (metis Diff-iff Nil-elem-T deadlock-free_SKIP-AfterExt
          deadlock-free_SKIP-is-right insertI1 Refusals-iff tickFree-Nil)
next
  assume assm : ‹?rhs›
  with BOT-iff-D process-charn have non-BOT : ‹P ≠ ⊥› by (metis Refusals-iff)
  show ‹deadlock-free_SKIP P›
  proof (unfold deadlock-free_SKIP-def failure-refine-def, safe)
    from assm have ∗ : ‹UNIV ∉ ℛ P›
      ‹e ∈ ready-set P ∧ e ≠ ✓ ⟹
       {(tl s, X) |s X. (s, X) ∈ ℱ P ∧ s ≠ [] ∧ hd s = e} ⊆ ℱ (DF_SKIP UNIV)›
for e
    by (simp-all add: deadlock-free_SKIP-def failure-refine-def F-AfterExt non-BOT)

    fix s X
    assume ∗∗ : ‹(s, X) ∈ ℱ P›
    show ‹(s, X) ∈ ℱ (DF_SKIP UNIV)›
    proof (cases s)
      show ‹s = [] ⟹ (s, X) ∈ ℱ (DF_SKIP UNIV)›
        by (subst F-DF_SKIP, simp)
           (metis ∗(1) ∗∗ UNIV-eq-I event.exhaust Refusals-iff)
    next
      fix e s'
      assume ∗∗∗ : ‹s = e # s'›
      show ‹(s, X) ∈ ℱ (DF_SKIP UNIV)›
      proof (cases e)
        fix e'
        assume ‹e = ev e'›
        with ∗∗ ∗∗∗ Cons-in-T-imp-elem-ready-set F-T
        have ‹e ∈ ready-set P ∧ e ≠ ✓› by blast
        with ∗(2)[OF this] show ‹(s, X) ∈ ℱ (DF_SKIP UNIV)›
          by (subst F-DF_SKIP, simp add: subset-iff)
             (metis ∗∗ ∗∗∗ ‹e = ev e'› list.distinct(1) list.sel(1))
      next
        assume ‹e = ✓›
        hence ‹s = [✓]›
          by (metis ∗∗ ∗∗∗ F-T append-butlast-last-id append-single-T-imp-tickFree
                butlast.simps(2) list.distinct(1) tickFree-Cons)
        thus ‹(s, X) ∈ ℱ (DF_SKIP UNIV)›
          by (subst F-DF_SKIP, simp)
      qed
    qed
  qed
qed



end
```

Chapter 6

Motivations for our Definitions

theory *Motivations*
  imports *AfterExt*
begin

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 an external event e (which can possibly be ✓): $P \rightsquigarrow e\ P'$.

We will discuss in this theory some fundamental properties that we want $P \rightsquigarrow_\tau Q$ and $P \rightsquigarrow e\ Q$ to verify, and on 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* :: $\langle {'\alpha}\ process \Rightarrow {'\alpha}\ process \Rightarrow bool\rangle$ (**infixl** $\langle\rightsquigarrow_\tau\rangle$ *50*)
  where τ-*trans-eq* : $\langle P \rightsquigarrow_\tau P\rangle$
  | τ-*trans-NdetL* : $\langle P \sqcap Q \rightsquigarrow_\tau P\rangle$

| τ-trans-NdetR : ‹$P \sqcap Q \rightsquigarrow_\tau Q$›

— 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: ‹$(\rightsquigarrow_\tau) = (\sqsubseteq_{FD})$›
apply (*intro ext iffI*)
by (*metis mono-Ndet-FD-left Ndet-commute* τ*-trans.simps idem-FD*)
(*metis mono-Ndet-FD mono-Ndet-FD-right*
FD-antisym Ndet-id τ*-trans-NdetL idem-FD*)

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

First of all we want to prevent a process $P::'\alpha$ *process* to make a transition with $e::'\alpha$ *event* if P can not begin with e.

More formally, we want $P \rightsquigarrow e\ P' \Longrightarrow e \in$ *ready-set* P.

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

Finally, when $e \in$ *ready-set* P, we want to have $P \rightsquigarrow e\ P$ *afterExt* e.

This brings us to the following inductive definition:

inductive *event-trans-prem* :: ‹$'\alpha$ *process* $\Rightarrow$ $'\alpha$ *event* $\Rightarrow$ $'\alpha$ *process* $\Rightarrow$ *bool*›
where
τ*-left-absorb* : ‹$[\![e \in$ *ready-set* P'; $P \rightsquigarrow_\tau P'$; *event-trans-prem* P' e $P''[\!]$
$\Longrightarrow$ *event-trans-prem* P e P''›
| τ*-right-absorb* : ‹$[\![e \in$ *ready-set* P; *event-trans-prem* P e P'; $P' \rightsquigarrow_\tau P''[\!]$
$\Longrightarrow$ *event-trans-prem* P e P''›
| *ready-trans-to-AfterExt* : ‹$e \in$ *ready-set* P
$\Longrightarrow$ *event-trans-prem* P e (P *afterExt* e)›

abbreviation *event-trans* :: ‹$'\alpha$ *process* $\Rightarrow$ $'\alpha$ *event* $\Rightarrow$ $'\alpha$ *process* $\Rightarrow$ *bool*›
(‹- $\rightsquigarrow$-/ -› [*50*, *3*, *51*] *50*)
where ‹$P \rightsquigarrow e\ P' \equiv e \in$ *ready-set* $P \wedge$ *event-trans-prem* P e P'›

We immediately show that this event transition definition is equivalent to the following:

lemma *startable-imp-ev-trans-is-startable-and-FD-After*:
‹$(P \rightsquigarrow e\ P') \longleftrightarrow e \in$ *ready-set* $P \wedge P$ *afterExt* $e \rightsquigarrow_\tau P'$›
proof *safe*
show ‹$e \in$ *ready-set* $P \Longrightarrow$ *event-trans-prem* P e $P' \Longrightarrow P$ *afterExt* $e \rightsquigarrow_\tau P'$›
apply (*rotate-tac*, *induct rule*: *event-trans-prem.induct*)
apply (*metis* τ*-trans-is-FD mono-AfterExt-FD trans-FD*)
apply (*metis* τ*-trans-is-FD trans-FD*)
by (*simp add*: τ*-trans-is-FD*)
next
show ‹$e \in$ *ready-set* $P \Longrightarrow P$ *afterExt* $e \rightsquigarrow_\tau P' \Longrightarrow$ *event-trans-prem* P e P'›
by (*rule* τ*-right-absorb*[*of* e P ‹P *afterExt* e› P'])

(*simp-all add*: *ready-trans-to-AfterExt* τ*-trans-is-FD*)

qed

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

- $P \rightsquigarrow_\tau Q \equiv P \sqsubseteq_{FD} Q$
- $P \rightsquigarrow e\ Q \equiv e \in \textit{ready-set}\ P \wedge P\ \textit{afterExt}\ e \rightsquigarrow_\tau Q$

and possible variations with other refinements.

end

Chapter 7

Generic Operational Semantics as a Locale

theory *OpSemGeneric*
imports *AfterTrace*
begin

7.1 Definition

We define a correspondence pattern using a locale in order to factorize the redundant proofs.

locale *OpSemGeneric* =
fixes τ-*trans* :: ‹['α *process*, 'α *process*] ⇒ *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-ready-set*: ‹$P \rightsquigarrow_\tau Q \Longrightarrow$ *ready-set* $Q \subseteq$ *ready-set* P›
and τ-*trans-mono-AfterExt*:
‹$e \in$ *ready-set* $Q \Longrightarrow P \rightsquigarrow_\tau Q \Longrightarrow P$ *afterExt* $e \rightsquigarrow_\tau Q$ *afterExt* e›
begin

This locale needs to be instantiated with a binary relation ($\rightsquigarrow_\tau$) which:

- is compatible with ($\sqcap$)
- is transitive
- makes *ready-set* anti-monotonic
- makes *AfterExt* monotonic.

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

abbreviation *event-trans* :: ‹['α *process*, 'α *event*, 'α *process*] ⇒ *bool*›

$(\langle\text{-}/\ \leadsto\text{-}/\ \text{-}\rangle\ [50,\ 3,\ 51]\ 50)$
where $\langle P \leadsto e\ Q \equiv e \in \mathit{ready\text{-}set}\ P \wedge P\ \mathit{afterExt}\ e \leadsto_\tau Q\rangle$

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

lemma *τ-trans-eq*: $\langle P \leadsto_\tau P\rangle$
and *τ-trans-NdetR*: $\langle P \sqcap Q \leadsto_\tau Q\rangle$
and *BOT-τ-trans-anything*: $\langle \bot \leadsto_\tau P\rangle$
and *BOT-event-trans-anything*: $\langle \bot \leadsto e\ P\rangle$
by (*metis Ndet-id τ-trans-NdetL*,
metis Ndet-commute τ-trans-NdetL,
metis Ndet-BOT τ-trans-NdetL,
metis AfterExt-BOT Ndet-BOT UNIV-I τ-trans-NdetL ready-set-BOT)

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

lemma *event-trans-τ-trans*: $\langle P \leadsto e\ P' \Longrightarrow P' \leadsto_\tau P'' \Longrightarrow P \leadsto e\ P''\rangle$
by (*meson τ-trans-transitivity*)

lemma *τ-trans-event-trans*: $\langle P \leadsto_\tau P' \Longrightarrow P' \leadsto e\ P'' \Longrightarrow P \leadsto e\ P''\rangle$
using *τ-trans-mono-AfterExt τ-trans-transitivity τ-trans-anti-mono-ready-set* **by** *blast*

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

inductive *trace-trans* :: $\langle 'α\ \mathit{process} \Rightarrow 'α\ \mathit{trace} \Rightarrow 'α\ \mathit{process} \Rightarrow \mathit{bool}\rangle$
$(\langle\text{-}/\ \leadsto^*\text{-}/\ \text{-}\rangle\ [50,\ 3,\ 51]\ 50)$
where *trace-τ-trans* : $\langle P \leadsto_\tau P' \Longrightarrow P \leadsto^*\ []\ P'\rangle$
| *trace-tick-trans* : $\langle P \leadsto ✓\ \ P' \Longrightarrow P \leadsto^*\ [✓]\ P'\rangle$
| *trace-Cons-ev-trans* :
$\langle P \leadsto (ev\ e)\ P' \Longrightarrow P' \leadsto^*\ s\ P'' \Longrightarrow P \leadsto^*\ (ev\ e)\ \#\ s\ P''\rangle$

lemma *trace-trans-τ-trans*: $\langle P \leadsto^* s\ P' \Longrightarrow P' \leadsto_\tau P'' \Longrightarrow P \leadsto^* s\ P''\rangle$
apply (*induct rule*: *trace-trans.induct*)
subgoal using *τ-trans-transitivity trace-τ-trans* **by** *blast*
subgoal using *event-trans-τ-trans trace-tick-trans* **by** *blast*
using *trace-Cons-ev-trans* **by** *blast*

lemma *τ-trans-trace-trans*: $\langle P \leadsto_\tau P' \Longrightarrow P' \leadsto^* s\ P'' \Longrightarrow P \leadsto^* s\ P''\rangle$
apply (*rotate-tac*, *induct rule*: *trace-trans.induct*)
subgoal using *τ-trans-transitivity trace-τ-trans* **by** *blast*
subgoal using *τ-trans-event-trans trace-tick-trans* **by** *blast*
using *τ-trans-event-trans trace-Cons-ev-trans* **by** *blast*

lemma *BOT-trace-trans-anything* : $\langle \mathit{front\text{-}tickFree}\ s \Longrightarrow \bot \leadsto^* s\ P\rangle$
proof (*induct s arbitrary*: *P*)
show $\langle \bigwedge P.\ \mathit{front\text{-}tickFree}\ [] \Longrightarrow \bot \leadsto^* []\ P\rangle$

by (*simp add*: *BOT-τ-trans-anything trace-τ-trans*)
next
fix *e s P*
assume *prem*: ‹*front-tickFree* (*e # s*)› **and** *hyp*: ‹*front-tickFree s* ⟹ ⊥ ⇝* *s P*›
for *P*
have ∗ : ‹*front-tickFree s*›
by (*metis prem butlast.simps*(*2*) *front-tickFree-butlast front-tickFree-def tick-Free-Cons*)
show ‹⊥ ⇝* *e # s P*›
proof (*cases e*)
fix *e′*
assume ‹*e* = *ev e′*›
show ‹⊥ ⇝* *e # s P*›
by (*simp add*: ‹*e* = *ev e′*›)
(*rule trace-Cons-ev-trans*[*OF - hyp*];
simp add: ∗ *AfterExt-BOT BOT-τ-trans-anything ready-set-BOT*)
next
have ‹*e* = ✓ ⟹ *s* = []›
by (*metis prem butlast.simps*(*2*) *front-tickFree-butlast tickFree-Cons*)
thus ‹*e* = ✓ ⟹ ⊥ ⇝* *e # s P*›
by (*simp add*: *AfterExt-BOT BOT-τ-trans-anything ready-set-BOT trace-tick-trans*)
qed
qed

7.2 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* ⇝* *s Q* ⟹ *X* ∈ ℛ *Q* ⟹ (*s*, *X*) ∈ ℱ *P*›
if ‹∀ *P Q*. *P* ⇝τ *Q* ⟶ *P* ⊑*F* *Q*›
proof (*induct rule*: *trace-trans.induct*)
show ‹*P* ⇝τ *Q* ⟹ *X* ∈ ℛ *Q* ⟹ ([], *X*) ∈ ℱ *P*› **for** *P Q*
by (*meson failure-refine-def in-mono Refusals-iff that*)
next
show ‹*P* ⇝✓ *Q* ⟹ *X* ∈ ℛ *Q* ⟹ ([✓], *X*) ∈ ℱ *P*› **for** *P Q*
by (*metis append-Nil mem-Collect-eq ready-set-def tick-T-F*)
next
fix *P e Q s Q′*
assume ∗ : ‹*P* ⇝(*ev e*) *Q*› ‹*X* ∈ ℛ *Q′* ⟹ (*s*, *X*) ∈ ℱ *Q*› ‹*X* ∈ ℛ *Q′*›
have ‹*P afterExt ev e* ⊑*F* *Q*› **using** ∗(*1*) *that* **by** *blast*
hence ‹(*s*, *X*) ∈ ℱ (*P afterExt ev e*)› **by** (*simp add*: *failure-refine-def subsetD* ∗(*2*, *3*))
thus ‹(*ev e # s*, *X*) ∈ ℱ *P*› **by** (*simp add*: *F-AfterExt* ∗(*1*)) (*metis list.exhaust-sel*)
qed

lemma *trace-trans-imp-T-if-τ-trans-imp-leT*: ‹*P* ⇝* *s Q* ⟹ *s* ∈ 𝒯 *P*›
if ‹∀ *P Q*. *P* ⇝τ *Q* ⟶ *P* ⊑*T* *Q*›
proof (*induct rule*: *trace-trans.induct*)
show ‹*P* ⇝τ *Q* ⟹ [] ∈ 𝒯 *P*› **for** *P Q*

by (*simp add*: *Nil-elem-T*)
next
show ‹$P \rightsquigarrow\!\!\checkmark\ Q \Longrightarrow [\checkmark] \in \mathcal{T}\ P$› **for** $P\ Q$
by (*simp add*: *ready-set-def*)
next
fix $P\ e\ Q\ s\ Q'$
assume $*$: ‹$P \rightsquigarrow(ev\ e)\ Q$› ‹$s \in \mathcal{T}\ Q$›
have ‹$P\ afterExt\ ev\ e \sqsubseteq_T Q$› **using** $*(1)$ *that* **by** *blast*
hence ‹$s \in \mathcal{T}\ (P\ afterExt\ ev\ e)$› **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-AfterExt ready-set-def*)
qed

lemma *trace-trans-BOT-imp-D-if-τ-trans-imp-leD*: ‹$P \rightsquigarrow^* s\ \bot \Longrightarrow s \in \mathcal{D}\ P$›
if ‹$\forall\ P\ Q.\ P \rightsquigarrow_\tau Q \longrightarrow P \sqsubseteq_D Q$›
proof (*induct s arbitrary*: *P*)
show ‹$\bigwedge P.\ P \rightsquigarrow^* [] \bot \Longrightarrow [] \in \mathcal{D}\ P$›
by (*subst* (*asm*) *trace-trans.simps*, *auto*)
(*meson BOT-iff-D divergence-refine-def subsetD that*)
next
fix $e\ s\ P$
assume *prem* : ‹$P \rightsquigarrow^* e\ \#\ s\ \bot$›
assume *hyp*: ‹$P \rightsquigarrow^* s\ \bot \Longrightarrow s \in \mathcal{D}\ P$› **for** P
have ‹$P\ afterExt\ e \rightsquigarrow^* s\ \bot$›
using *prem* **by** (*cases rule*: *trace-trans.cases*)
(*auto simp add*: *trace-τ-trans intro*: *τ-trans-trace-trans*)
from *hyp*[*OF this*] **show** ‹$e\ \#\ s \in \mathcal{D}\ P$›
by (*fastforce intro*: *prem trace-trans.cases simp add*: *D-AfterExt D-UU split*: *if-split-asm*)
qed

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

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

show *f4* : ‹$tickFree\ s \Longrightarrow (P \rightsquigarrow^*\ s\ @\ [f]\ Q') \longleftrightarrow (\exists\ Q.\ P \rightsquigarrow^* s\ Q \wedge Q \rightsquigarrow f\ Q')$›
for $s\ f\ P\ Q'$
proof *safe*

```
    show ‹P ⇝* s @ [f] Q' ⟹ ∃ Q. P ⇝*s Q ∧ Q ⇝f Q'›
    proof (induct s arbitrary: P)
      case Nil
      thus ?case
        apply (subst (asm) trace-trans.simps, simp)
        using τ-trans-eq τ-trans-transitivity f1 by blast
    next
      case (Cons e s)
      from Cons.prems have ∗ : ‹e ∈ ready-set P ∧ P afterExt e ⇝*s @ [f] Q'›
        by (subst (asm) trace-trans.simps)
           (auto simp add:  intro: τ-trans-trace-trans)
      with Cons.hyps obtain Q where ∗∗ : ‹P afterExt e ⇝*s Q› ‹Q ⇝f 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 = ✓ ⟹ s = []› by (subst (asm) trace-trans.simps)
auto
        thus ‹e = ✓ ⟹ P ⇝*e # s Q› using ∗ ∗∗(1) f1 f2 by blast
      qed
      with ∗∗(2) show ‹∃ Q. P ⇝*e # s Q ∧ Q ⇝f Q'› by blast
    qed
  next
    show ‹tickFree s ⟹ P ⇝*s Q ⟹ f ∈ ready-set Q ⟹ Q afterExt f ⇝τ Q'
⟹
          P ⇝*s @ [f] Q'› for Q
    proof (induct s arbitrary: P Q)
      show ‹tickFree [] ⟹ P ⇝*[] Q ⟹ f ∈ ready-set Q ⟹ Q afterExt f ⇝τ
Q' ⟹
            P ⇝*[] @ [f] Q'› for P Q
        by (simp add: f1)
           (metis (full-types) τ-trans-eq τ-trans-event-trans event.exhaust
                               trace-Cons-ev-trans trace-τ-trans trace-tick-trans)
    next
      case (Cons e s)
      from Cons.prems(2) have ∗ : ‹e ∈ ready-set P ∧ P afterExt 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'›
          apply simp
```

```
        by (rule trace-Cons-ev-trans
              [OF - Cons.hyps[OF tickFree-tl[OF Cons.prems(1), simplified]
                                 *[THEN conjunct2] Cons.prems(3)]])
           (use * τ-trans-eq Cons.prems(4) in ‹blast+›)
    next
      show ‹e = ✓ ⟹ P ⇝*(e # s) @ [f] Q'› using Cons.prems(1) by auto
    qed
  qed
 qed

 show ‹front-tickFree (s @ t) ⟹ P ⇝*s @ t Q' ⟷ (∃ Q. P ⇝*s Q ∧ Q ⇝*t
Q')›
 proof (induct t arbitrary: Q' rule: rev-induct)
  show ‹P ⇝*s @ [] Q' ⟷ (∃ Q. P ⇝*s Q ∧ Q ⇝*[] Q')› for Q'
   by (metis τ-trans-eq append.right-neutral trace-trans-τ-trans f1)
 next
  case (snoc e t)
  show ‹P ⇝*s @ t @ [e] Q' ⟷ (∃ Q. P ⇝*s Q ∧ Q ⇝*t @ [e] Q')›
  proof (intro iffI)
   assume assm : ‹P ⇝*s @ t @ [e] Q'›
   from assm obtain Q where ‹P ⇝*s @ t Q› ‹Q ⇝e Q'›
    by (metis append.assoc f4 front-tickFree-implies-tickFree snoc.prems)
   also obtain R where ** : ‹P ⇝*s R› ‹R ⇝*t Q›
     by (metis calculation(1) append.assoc front-tickFree-dw-closed snoc.hyps
snoc.prems)
   ultimately show ‹∃ Q. P ⇝*s Q ∧ Q ⇝*t @ [e] Q'›
   by (metis append-is-Nil-conv f4 front-tickFree-mono list.distinct(1) snoc.prems)
  next
   assume ‹∃ Q. P ⇝*s Q ∧ Q ⇝*t @ [e] Q'›
   then obtain Q where ‹P ⇝*s Q› ‹Q ⇝*t @ [e] Q'› by blast
   also obtain R where ‹Q ⇝*t R› ‹R ⇝e Q'›
    by (metis append-is-Nil-conv calculation(2) f4
          front-tickFree-mono list.distinct(1) snoc.prems)
   ultimately show ‹P ⇝*s @ t @ [e] Q'›
    by (metis append-assoc f4 front-tickFree-implies-tickFree snoc.hyps
          snoc.prems tickFree-implies-front-tickFree)
  qed
 qed
qed
```

7.4 Finally: $P \rightsquigarrow^* s\ Q$ is P *afterTrace* $s \rightsquigarrow_\tau Q$

```
theorem T-imp-trace-trans-iff-AfterTrace-τ-trans :
 ‹s ∈ 𝒯 P ⟹ (P ⇝*s Q) ⟷ P afterTrace s ⇝τ Q›
proof (intro iffI)
 show ‹P ⇝*s Q ⟹ s ∈ 𝒯 P ⟹ P afterTrace s ⇝τ Q›
 proof (induct s arbitrary: P Q rule: rev-induct)
  show ‹⋀P Q. P ⇝*[] Q ⟹ [] ∈ 𝒯 P ⟹ P afterTrace [] ⇝τ Q›
   using trace-trans.cases by auto
```

```
  next
    fix s e P Q
    assume   hyp : ‹P ⇝*s Q ⟹ s ∈ 𝒯 P ⟹ P afterTrace s ↝τ Q› for P Q
    assume prems : ‹P ⇝*s @ [e] Q› ‹s @ [e] ∈ 𝒯 P›
    have ∗ : ‹e ∈ ready-set (P afterTrace s)›
      using prems(2) ready-set-AfterTrace by blast
    show ‹P afterTrace (s @ [e]) ↝τ Q›
      by (metis AfterTrace-snoc τ-trans-event-trans append-single-T-imp-tickFree
            hyp is-processT3-ST prems trace-trans-iff(4))
  qed
next
  show ‹P afterTrace s ↝τ Q ⟹ s ∈ 𝒯 P ⟹ P ⇝*s Q›
  proof (induct arbitrary: P Q rule: AfterTrace.induct)
    show ‹⋀P Q. P afterTrace [] ↝τ Q ⟹ [] ∈ 𝒯 P ⟹ P ⇝*[] Q›
      by (simp add: trace-τ-trans)
  next
    fix e and s :: ‹'α trace› and Q P
    assume   hyp : ‹P afterTrace s ↝τ Q ⟹ s ∈ 𝒯 P ⟹ P ⇝*s Q› for P Q
    assume prems : ‹P afterTrace (e # s) ↝τ Q› ‹e # s ∈ 𝒯 P›
    have ∗ : ‹e ∈ ready-set P ∧ s ∈ 𝒯 (P afterExt e)›
      by (metis AfterTrace.simps(1, 2) Cons-in-T-imp-elem-ready-set
            T-AfterTrace append-Cons append-Nil prems(2))
    show ‹P ⇝*e # s Q›
    proof (cases e)
      fix e'
      assume ∗∗ : ‹e = ev e'›
      from ∗ ∗∗ have ‹P ⇝(ev e') P afterExt (ev e')›
        by (simp add: τ-trans-eq)
      thus ‹P ⇝*e # s Q›
        by (subst ∗∗, rule trace-Cons-ev-trans[OF - hyp[OF prems(1)[simplified]
                                                  ∗[THEN conjunct2], simplified ∗∗]])
    next
      have ‹e = ✓ ⟹ s = []› by (metis butlast.simps(2) front-tickFree-butlast
                                   is-processT2-TR tickFree-Cons prems(2))
      thus ‹e = ✓ ⟹ P ⇝*e # s Q›
        using ∗ prems(1) trace-tick-trans by force
    qed
  qed
qed
```

As corollaries we obtain the reciprocal results of

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

$[\![\forall P\ Q.\ P \rightsquigarrow_\tau Q \longrightarrow P \sqsubseteq_T Q;\ ?P \rightsquigarrow^* ?s\ ?Q]\!] \Longrightarrow ?s \in \mathcal{T}\ ?P$

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

```
lemma F-imp-exists-trace-trans: ‹(s, X) ∈ 𝓕 P ⟹ ∃Q. (P ⇝*s Q) ∧ X ∈ ℛ Q›
  by (meson F-T F-imp-R-AfterTrace T-imp-trace-trans-iff-AfterTrace-τ-trans τ-trans-eq)
```

lemma *T-imp-exists-trace-trans*: ‹$s \in \mathcal{T}\ P \Longrightarrow \exists\, Q.\ P \rightsquigarrow^* s\ Q$›
using *F-imp-exists-trace-trans T-F* **by** *blast*

lemma *D-imp-trace-trans-BOT*: ‹$s \in \mathcal{D}\ P \Longrightarrow P \rightsquigarrow^* s\ \bot$›
by (*subst T-imp-trace-trans-iff-AfterTrace-τ-trans*, *simp add*: *D-T*)
(*metis BOT-iff-D D-AfterTrace τ-trans-eq self-append-conv*)

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

lemma *τ-trans-imp-leT-imp-STOP-trace-trans-iff*:
‹$\forall\, P\ Q.\ P \rightsquigarrow_\tau Q \longrightarrow P \sqsubseteq_T Q \Longrightarrow STOP \rightsquigarrow^* s\ P \longleftrightarrow s = [] \wedge P = STOP$›
using *STOP-T-iff* **by** (*subst trace-trans.simps*)
(*auto simp add*: *ready-set-STOP τ-trans-eq*)

lemma *τ-trans-imp-leF-imp-SKIP-trace-trans-iff*:
‹$\forall\, P\ Q.\ P \rightsquigarrow_\tau Q \longrightarrow P \sqsubseteq_F Q \Longrightarrow$
$SKIP \rightsquigarrow^* s\ P \longleftrightarrow s = [] \wedge P = SKIP \vee s = [\checkmark] \wedge P = STOP$›
using *SKIP-F-iff STOP-F-iff* **by** (*subst trace-trans.simps*)
(*auto simp add*: *AfterExt-SKIP ready-set-SKIP τ-trans-eq*)

lemma *τ-trans-imp-leT-imp-trace-trans-ready-set-subset-ready-set-AfterTrace*:
‹$\forall\, P\ Q.\ P \rightsquigarrow_\tau Q \longrightarrow P \sqsubseteq_T Q \Longrightarrow P \rightsquigarrow^* s\ Q \Longrightarrow$
$ready\text{-}set\ Q \subseteq ready\text{-}set\ (P\ afterTrace\ s)$›
using *T-imp-trace-trans-iff-AfterTrace-τ-trans*
anti-mono-ready-set-T trace-trans-imp-T-if-τ-trans-imp-leT **by** *blast*

lemma *τ-trans-imp-leT-imp-trace-trans-imp-ready-set*:
‹$\forall\, P\ Q.\ P \rightsquigarrow_\tau Q \longrightarrow P \sqsubseteq_T Q \Longrightarrow P \rightsquigarrow^* (s\ @\ e\ \#\ t)\ Q \Longrightarrow$
$e \in ready\text{-}set\ (P\ afterTrace\ s)$›
using *ready-set-AfterTrace trace-trans-imp-T-if-τ-trans-imp-leT* **by** *blast*

lemma *trace-trans-iff-T-and-AfterTrace-τ-trans-if-τ-trans-imp-leT*:
‹$\forall\, P\ Q.\ P \rightsquigarrow_\tau Q \longrightarrow P \sqsubseteq_T Q \Longrightarrow$
$(P \rightsquigarrow^* s\ Q) \longleftrightarrow s \in \mathcal{T}\ P \wedge P\ afterTrace\ s \rightsquigarrow_\tau Q$›
using *T-imp-trace-trans-iff-AfterTrace-τ-trans trace-trans-imp-T-if-τ-trans-imp-leT*
by *blast*

7.5 General Rules of Operational Semantics

Some rules of operational semantics are consequences of *OpSemGeneric*'s axioms without needing to specify more what $P \rightsquigarrow_\tau Q$ is.

lemma *SKIP-trans-tick*: ‹$SKIP \rightsquigarrow\checkmark\ STOP$›
by (*simp add*: *AfterExt-SKIP τ-trans-eq ready-set-SKIP*)

lemma *tick-trans-imp-BOT-L-or-STOP-R*: ‹$P \rightsquigarrow\checkmark\ Q \Longrightarrow P = \bot \vee Q = STOP$›

by (*metis* τ*-trans-anti-mono-ready-set ready-set-AfterExt ready-set-empty-iff-STOP subset-empty*)

lemma *STOP-trace-trans-iff* : ‹$STOP \rightsquigarrow^* s\ P \longleftrightarrow s = [] \wedge P = STOP$›
by (*metis AfterExt-SKIP SKIP-neq-BOT SKIP-trans-tick empty-iff trace-trans.cases ready-set-STOP tick-trans-imp-BOT-L-or-STOP-R trace-trans-iff*(*1*))

lemma *ready-tick-imp-*τ*-trans-SKIP*: ‹$P \rightsquigarrow_\tau SKIP$› **if** ‹$\checkmark \in$ *ready-set* P›
proof −
from *that* **have** ‹$P \sqsubseteq_{FD} SKIP$›
unfolding *failure-divergence-refine-def le-ref-def*
by (*auto simp add*: *F-SKIP D-SKIP subset-iff ready-set-def is-processT6-S2*)
(*metis append-Nil tick-T-F*)
then obtain Q **where** ‹$P = Q \sqcap SKIP$›
by (*metis mono-Ndet-FD mono-Ndet-FD-left FD-antisym Ndet-id idem-FD*)
thus ‹$P \rightsquigarrow_\tau SKIP$› **by** (*simp add*: τ*-trans-NdetR*)
qed

lemma *exists-tick-trans-is-ready-tick*: ‹$(\exists P'.\ P \rightsquigarrow\checkmark P') \longleftrightarrow \checkmark \in$ *ready-set* P›
using τ*-trans-eq* **by** *blast*

lemma *tick-trans-iff* : ‹$P \rightsquigarrow\checkmark P' \longleftrightarrow P = \bot \vee P \rightsquigarrow_\tau SKIP \wedge P' = STOP$›
by (*metis AfterExt-BOT BOT-*τ*-trans-anything SKIP-trans-tick* τ*-trans-event-trans*

ready-tick-imp-τ*-trans-SKIP tick-trans-imp-BOT-L-or-STOP-R*)

lemma *SKIP-cant-ev-trans*: ‹$\neg$ $SKIP \rightsquigarrow(ev\ e)\ STOP$›
by (*simp add*: *ready-set-SKIP*)

lemma *STOP-cant-event-trans*: ‹$\neg$ $STOP \rightsquigarrow e\ P$›
by (*simp add*: *ready-set-STOP*)

lemma *ev-trans-Mprefix*: ‹$e \in A \Longrightarrow \Box a \in A \rightarrow P\ a \rightsquigarrow(ev\ e)\ (P\ e)$›
by (*simp add*: *AfterExt-def After-Mprefix* τ*-trans-eq ready-set-Mprefix*)

lemma *ev-trans-Mndetprefix*: ‹$e \in A \Longrightarrow \sqcap a \in A \rightarrow P\ a \rightsquigarrow(ev\ e)\ (P\ e)$›
by (*simp add*: *AfterExt-def After-Mndetprefix* τ*-trans-eq ready-set-Mndetprefix*)

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

lemma τ-*trans-MultiNdet*: ‹*finite* $A \Longrightarrow x \in A \Longrightarrow (\bigsqcap a \in A.\ P\ a) \rightsquigarrow_\tau P\ x$›
by (*metis MultiNdet-insert'* τ-*trans-NdetL emptyE insert-absorb*)

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

lemma *fix-point-*τ-*trans*: ‹*cont* $f \Longrightarrow P = (\mu\ X.\ f\ X) \Longrightarrow P \rightsquigarrow_\tau f\ P$›
by (*metis* τ-*trans-eq cont-process-rec*)

lemma *event-trans-DetL*: ‹$P \rightsquigarrow e\ P' \Longrightarrow P \Box Q \rightsquigarrow e\ P'$›
by (*metis AfterExt-Det-is-AfterExt-Ndet Un-iff* τ-*trans-NdetL* τ-*trans-event-trans ready-set-Det*)

lemma *event-trans-DetR*: ‹$Q \rightsquigarrow e\ Q' \Longrightarrow P \Box Q \rightsquigarrow e\ Q'$›
by (*metis Det-commute event-trans-DetL*)

lemma *event-trans-MultiDet*:
‹*finite* $A \Longrightarrow a \in A \Longrightarrow P\ a \rightsquigarrow e\ Q \Longrightarrow (\Box a \in A.\ P\ a) \rightsquigarrow e\ Q$›
by (*metis MultiDet-insert' event-trans-DetL insert-absorb*)

lemma *Sliding-event-transL*: ‹$P \rightsquigarrow e\ P' \Longrightarrow P \rhd Q \rightsquigarrow e\ P'$›
unfolding *Sliding-def*
apply (*drule event-trans-DetL*[*of e P P' Q*])
using τ-*trans-NdetL* τ-*trans-event-trans* **by** *blast*

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

lemma ‹$\exists P\ P'\ Q.\ P \rightsquigarrow\checkmark P' \wedge \neg P\ ;\ Q \rightsquigarrow\checkmark P'\ ;\ Q$›
proof (*intro exI*)
show ‹$SKIP \rightsquigarrow\checkmark STOP \wedge \neg SKIP\ ;\ STOP \rightsquigarrow\checkmark STOP\ ;\ STOP$›
by (*simp add*: *SKIP-Seq SKIP-trans-tick STOP-cant-event-trans*)
qed

lemma *ev-trans-SeqR*:
‹✓ ∈ *ready-set P* ⟹ *Q* ⇝(*ev e*) *Q′* ⟹ *P* **;** *Q* ⇝(*ev e*) *Q′*›
apply (*simp add*: *AfterExt-Seq ready-set-Seq AfterExt-BOT BOT-Seq*)
using *τ-trans-NdetR τ-trans-transitivity* **by** *blast*

lemma ‹*SKIP* ⟦*S*⟧ *SKIP* ⇝✓ *STOP*›
by (*simp add*: *SKIP-trans-tick Sync-SKIP-SKIP*)

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

lemma *tick-trans-Hiding*: ‹*P* \ *B* ⇝✓ *STOP*› **if** ‹*P* ⇝✓ *P′*›
proof −
have ‹(*P* \ *B*) *afterExt* ✓ = *STOP* ⊓ (*P* \ *B*) *afterExt* ✓›
by (*simp add*: *AfterExt-def Ndet-is-BOT-iff*)
thus ‹*P* \ *B* ⇝✓ *STOP*›
by (*simp add*: *AfterExt-def BOT-τ-trans-anything τ-trans-eq*
ready-tick-imp-ready-tick-Hiding that)
qed

The following lemma is useless since the locale mechanism forces f to be of type $'\alpha \Rightarrow '\alpha$ while it could be $'\alpha \Rightarrow '\beta$. We will have to prove it again on each instantiation of the locale.

lemma ‹*Renaming P f* ⇝✓ *STOP*› **if** ‹*P* ⇝✓ *P′*›
proof −
have ‹*Renaming P f afterExt* ✓ = *STOP* ⊓ *Renaming P f afterExt* ✓›
by (*metis AfterExt-Sliding-is-AfterExt-Ndet STOP-Sliding*)
thus ‹*Renaming P f* ⇝✓ *STOP*›
by (*simp add*: *that tick-eq-EvExt AfterExt-def τ-trans-eq*
ready-set-Renaming BOT-τ-trans-anything)
qed

end

end

Chapter 8

Failure Divergence Operational Semantics

theory *OpSemFD*
imports *OpSemGeneric HOL−Library.LaTeXsugar*
begin

As announced in the motivations, the first definition we want to try is with $(\sqsubseteq_{FD})$.

abbreviation τ*-trans* :: ‹$'\alpha$ *process* ⇒ $'\alpha$ *process* ⇒ *bool*› (**infixl** ‹$_{FD}\rightsquigarrow_{\tau}$› *50*)
where ‹$P\ _{FD}\rightsquigarrow_{\tau}\ Q \equiv P \sqsubseteq_{FD} Q$›

We now instantiate the locale of *HOL−CSP-OpSem.OpSemGeneric*.

interpretation *OpSemGeneric* ‹$(_{FD}\rightsquigarrow_{\tau})$›
using *trans-FD* **by** *unfold-locales*
(*auto simp add*: *anti-mono-ready-set-FD mono-AfterExt-FD*)

notation *event-trans* (‹-/ $_{FD}\rightsquigarrow$-/ -› [*50*, *3*, *51*] *50*)
notation *trace-trans* (‹-/ $_{FD}\rightsquigarrow^{*}$-/ -› [*50*, *3*, *51*] *50*)

lemma ‹$P\ _{FD}\rightsquigarrow\ e\ P' \Longrightarrow P'\ _{FD}\rightsquigarrow_{\tau}\ P'' \Longrightarrow P\ _{FD}\rightsquigarrow\ e\ P''$›
‹$P\ _{FD}\rightsquigarrow_{\tau}\ P' \Longrightarrow P'\ _{FD}\rightsquigarrow\ e\ P'' \Longrightarrow P\ _{FD}\rightsquigarrow\ e\ P''$›
by (*fact event-trans-τ-trans τ-trans-event-trans*)+

8.1 Operational Semantics Laws

SKIP law

lemma ‹$SKIP\ _{FD}\rightsquigarrow✓\ STOP$›
by (*fact SKIP-trans-tick*)

$e \rightarrow P$ laws

lemma ‹$e \in A \Longrightarrow \Box a \in A \rightarrow P\ a\ _{FD}\rightsquigarrow(ev\ e)\ (P\ e)$›

by (*fact ev-trans-Mprefix*)

lemma ‹$e \in A \Longrightarrow \sqcap a \in A \rightarrow P\ a\ {}_{FD}\!\!\rightsquigarrow (ev\ e)\ (P\ e)$›
by (*fact ev-trans-Mndetprefix*)

lemma ‹$e \rightarrow P\ {}_{FD}\!\!\rightsquigarrow\ (ev\ e)\ P$›
by (*fact ev-trans-prefix*)

$P \sqcap Q$ laws

lemma ‹$P \sqcap Q\ {}_{FD}\!\!\rightsquigarrow_{\tau}\ P$›
and ‹$P \sqcap Q\ {}_{FD}\!\!\rightsquigarrow_{\tau}\ Q$›
by (*fact* τ*-trans-NdetL* τ*-trans-NdetR*)+

lemma ‹$a \in A \Longrightarrow (\sqcap a \in A.\ P\ a)\ {}_{FD}\!\!\rightsquigarrow_{\tau}\ P\ a$›
by (*fact* τ*-trans-GlobalNdet*)

lemma ‹*finite* $A \Longrightarrow a \in A \Longrightarrow (\bigsqcap a \in A.\ P\ a)\ {}_{FD}\!\!\rightsquigarrow_{\tau}\ P\ a$›
by (*fact* τ*-trans-MultiNdet*)

$\mu\ x.\ f\ x$ law

lemma ‹*cont* $f \Longrightarrow P = (\mu\ X.\ f\ X) \Longrightarrow P\ {}_{FD}\!\!\rightsquigarrow_{\tau}\ f\ P$›
by (*fact fix-point-*τ*-trans*)

$P \Box Q$ laws

lemma τ*-trans-DetL*: ‹$P\ {}_{FD}\!\!\rightsquigarrow_{\tau}\ P' \Longrightarrow P \Box Q\ {}_{FD}\!\!\rightsquigarrow_{\tau}\ P' \Box Q$ ›
and τ*-trans-DetR*: ‹$Q\ {}_{FD}\!\!\rightsquigarrow_{\tau}\ Q' \Longrightarrow P \Box Q\ {}_{FD}\!\!\rightsquigarrow_{\tau}\ P\ \Box\ Q'$›
by *simp-all*

lemma τ*-trans-MultiDet*:
‹*finite* $A \Longrightarrow \forall\, a \in A.\ P\ a\ {}_{FD}\!\!\rightsquigarrow_{\tau}\ P'\ a \Longrightarrow$
$(\Box a \in A.\ P\ a)\ {}_{FD}\!\!\rightsquigarrow_{\tau}\ (\Box a \in A.\ P'\ a)$›
by (*fact mono-MultiDet-FD*)

lemma ‹$P\ {}_{FD}\!\!\rightsquigarrow e\ P' \Longrightarrow P \Box Q\ {}_{FD}\!\!\rightsquigarrow e\ P'$›
and ‹$Q\ {}_{FD}\!\!\rightsquigarrow e\ Q' \Longrightarrow P \Box Q\ {}_{FD}\!\!\rightsquigarrow e\ Q'$›
by (*fact event-trans-DetL event-trans-DetR*)+

lemma ‹*finite* $A \Longrightarrow a \in A \Longrightarrow P\ a\ {}_{FD}\!\!\rightsquigarrow e\ Q \Longrightarrow (\Box a \in A.\ P\ a)\ {}_{FD}\!\!\rightsquigarrow e\ Q$›
by (*fact event-trans-MultiDet*)

P **;** Q laws

lemma τ*-trans-SeqL*: ‹$P\ {}_{FD}\!\!\rightsquigarrow_{\tau}\ P' \Longrightarrow P\ ;\ Q\ {}_{FD}\!\!\rightsquigarrow_{\tau}\ P'\ ;\ Q$›
by *simp*

lemma *ev-trans-SeqL*: ‹$P\ {}_{FD}\!\!\rightsquigarrow (ev\ e)\ P' \Longrightarrow P\ ;\ Q\ {}_{FD}\!\!\rightsquigarrow (ev\ e)\ P'\ ;\ Q$›

by (*auto simp add*: *ready-set-Seq AfterExt-Seq*)

lemma *τ-trans-SeqR*: ‹∃ *P′*. *P* $_{FD}\!\rightsquigarrow\!\checkmark$ *P′* ⟹ *P* ; *Q* $_{FD}\!\rightsquigarrow_{\tau}$ *Q*›
by (*metis mono-Seq-FD SKIP-Seq τ-trans-eq ready-tick-imp-τ-trans-SKIP*)

lemma ‹✓ ∈ *ready-set P* ⟹ *Q* $_{FD}\!\rightsquigarrow$(*ev e*) *Q′* ⟹ *P* ; *Q* $_{FD}\!\rightsquigarrow$(*ev e*) *Q′*›

by (*fact ev-trans-SeqR*)

P \ *B* laws

lemma *τ-trans-Hiding*: ‹*P* $_{FD}\!\rightsquigarrow_{\tau}$ *P′* ⟹ *P* \ *B* $_{FD}\!\rightsquigarrow_{\tau}$ *P′* \ *B*›
by (*fact mono-Hiding-FD*)

lemma *ev-trans-Hiding-notin*:
‹*P* $_{FD}\!\rightsquigarrow$(*ev e*) *P′* ⟹ *e* ∉ *B* ⟹ *P* \ *B* $_{FD}\!\rightsquigarrow$(*ev e*) *P′* \ *B*›
by (*metis AfterExt-def After-Hiding-FD-Hiding-After-if-ready-notin mono-Hiding-FD*

event-trans-τ-trans event.simps(*4*) *ready-notin-imp-ready-Hiding*)

lemma ‹*P* $_{FD}\!\rightsquigarrow\!\checkmark$ *P′* ⟹ *P* \ *B* $_{FD}\!\rightsquigarrow\!\checkmark$ *STOP*›
by (*fact tick-trans-Hiding*)

lemma *ev-trans-Hiding-inside*:
‹*P* $_{FD}\!\rightsquigarrow$(*ev e*) *P′* ⟹ *e* ∈ *B* ⟹ *P* \ *B* $_{FD}\!\rightsquigarrow_{\tau}$ *P′* \ *B*›
by (*metis AfterExt-def Hiding-FD-Hiding-After-if-ready-inside*
mono-Hiding-FD event.simps(*4*) *trans-FD*)

Renaming P f laws

lemma *τ-trans-Renaming*:
‹*P* $_{FD}\!\rightsquigarrow_{\tau}$ *P′* ⟹ *Renaming P f* $_{FD}\!\rightsquigarrow_{\tau}$ *Renaming P′ f*›
by (*fact mono-Renaming-FD*)

lemma *tick-trans-Renaming*: ‹*P* $_{FD}\!\rightsquigarrow\!\checkmark$ *P′* ⟹ *Renaming P f* $_{FD}\!\rightsquigarrow\!\checkmark$ *STOP*›
by (*simp add*: *AfterExt-def ready-set-Renaming tick-eq-EvExt*)

lemma *ev-trans-Renaming*:
‹*f a* = *b* ⟹ *P* $_{FD}\!\rightsquigarrow$(*ev a*) *P′* ⟹ *Renaming P f* $_{FD}\!\rightsquigarrow$(*ev b*) (*Renaming P′ f*)›
apply (*simp add*: *AfterExt-Renaming Renaming-BOT ready-set-BOT ready-set-Renaming*)
apply (*intro conjI impI*)
apply (*meson ev-elem-anteced1 imageI vimageI2*)
apply (*rule τ-trans-transitivity*[*of* - ‹*Renaming* (*P afterExt ev a*) *f*›])
apply (*solves* ‹*rule τ-trans-GlobalNdet*, *simp*›)
by (*simp add*: *τ-trans-Renaming*)

lemma ‹*P* $_{FD}\!\rightsquigarrow_{\tau}$ *P′* ⟹ *P* ⟦*a* := *b*⟧ $_{FD}\!\rightsquigarrow_{\tau}$ *P′* ⟦*a* := *b*⟧›
by (*fact τ-trans-Renaming*)

lemma $\langle P \ {}_{FD}\!\rightsquigarrow\!\checkmark \ P' \Longrightarrow P \ [\![a := b]\!] \ {}_{FD}\!\rightsquigarrow\!\checkmark \ STOP\rangle$
by (*fact tick-trans-Renaming*)

lemma *ev-trans-RenamingF*:
$\langle P \ {}_{FD}\!\rightsquigarrow\!(ev\ a) \ P' \Longrightarrow P \ [\![a := b]\!] \ {}_{FD}\!\rightsquigarrow\!(ev\ b) \ P' \ [\![a := b]\!]\rangle$
by (*metis ev-trans-Renaming fun-upd-same*)

$P \ [\![S]\!] \ Q$ laws

lemma τ-*trans-SyncL*: $\langle P \ {}_{FD}\!\rightsquigarrow_{\tau} \ P' \Longrightarrow P \ [\![S]\!] \ Q \ {}_{FD}\!\rightsquigarrow_{\tau} \ P' \ [\![S]\!] \ Q\rangle$
and τ-*trans-SyncR*: $\langle Q \ {}_{FD}\!\rightsquigarrow_{\tau} \ Q' \Longrightarrow P \ [\![S]\!] \ Q \ {}_{FD}\!\rightsquigarrow_{\tau} \ P \ [\![S]\!] \ Q'\rangle$
by *simp-all*

lemma *ev-trans-SyncL*:
$\langle e \notin S \Longrightarrow P \ {}_{FD}\!\rightsquigarrow\!(ev\ e) \ P' \Longrightarrow P \ [\![S]\!] \ Q \ {}_{FD}\!\rightsquigarrow\!(ev\ e) \ P' \ [\![S]\!] \ Q \ \rangle$
and *ev-trans-SyncR*:
$\langle e \notin S \Longrightarrow Q \ {}_{FD}\!\rightsquigarrow\!(ev\ e) \ Q' \Longrightarrow P \ [\![S]\!] \ Q \ {}_{FD}\!\rightsquigarrow\!(ev\ e) \ P \ \ [\![S]\!] \ Q'\rangle$
by (*simp-all add*: *AfterExt-Sync ready-set-Sync image-iff*)

lemma *ev-trans-SyncLR*:
$\langle [\![e \in S;\ P \ {}_{FD}\!\rightsquigarrow\!(ev\ e) \ P';\ Q \ {}_{FD}\!\rightsquigarrow\!(ev\ e) \ Q']\!] \Longrightarrow$
$P \ [\![S]\!] \ Q \ {}_{FD}\!\rightsquigarrow\!(ev\ e) \ P' \ [\![S]\!] \ Q'\rangle$
by (*simp add*: *AfterExt-Sync ready-set-Sync*)

From here we slightly defer from Roscoe's laws for *Sync*: we obtain the following rules for *SKIP* instead of *STOP*.

lemma *tick-trans-SyncL*: $\langle P \ {}_{FD}\!\rightsquigarrow\!\checkmark \ P' \Longrightarrow P \ [\![S]\!] \ Q \ {}_{FD}\!\rightsquigarrow_{\tau} \ SKIP \ [\![S]\!] \ Q\rangle$
and *tick-trans-SyncR*: $\langle Q \ {}_{FD}\!\rightsquigarrow\!\checkmark \ Q' \Longrightarrow P \ [\![S]\!] \ Q \ {}_{FD}\!\rightsquigarrow_{\tau} \ P \ [\![S]\!] \ SKIP\rangle$
by (*simp-all add*: *ready-tick-imp-*τ*-trans-SKIP*)

lemma *tick-trans-SKIP-Sync-SKIP*: $\langle SKIP \ [\![S]\!] \ SKIP \ {}_{FD}\!\rightsquigarrow\!\checkmark \ STOP\rangle$
by (*simp add*: *SKIP-trans-tick Sync-SKIP-SKIP*)

lemma τ-*trans-SKIP-Sync-SKIP*: $\langle SKIP \ [\![S]\!] \ SKIP \ {}_{FD}\!\rightsquigarrow_{\tau} \ SKIP\rangle$
by (*simp add*: *Sync-SKIP-SKIP*)

$P \rhd Q$ laws

lemma *Sliding-*τ*-trans-left*: $\langle P \ {}_{FD}\!\rightsquigarrow_{\tau} \ P' \Longrightarrow P \rhd Q \ {}_{FD}\!\rightsquigarrow_{\tau} \ P' \rhd Q\rangle$
by (*simp add*: *Sliding-def*)

lemma $\langle P \ {}_{FD}\!\rightsquigarrow e \ P' \Longrightarrow P \rhd Q \ {}_{FD}\!\rightsquigarrow e \ P'\rangle$
by (*fact Sliding-event-transL*)

lemma $\langle P \rhd Q \ {}_{FD}\!\rightsquigarrow_{\tau} \ Q\rangle$
by (*fact Sliding-*τ*-transR*)

$P \triangle Q$ laws

lemma *Interrupt-*τ*-trans-left*: $\langle P \ {}_{FD}\!\rightsquigarrow_{\tau} \ P' \Longrightarrow P \triangle Q \ {}_{FD}\!\rightsquigarrow_{\tau} \ P' \triangle Q\rangle$

by (*simp add*: *mono-Interrupt-FD*)

lemma *Interrupt-τ-trans-right*: ‹$Q\ {}_{FD}\!\!\rightsquigarrow_{\tau}\ Q' \Longrightarrow P \triangle Q\ {}_{FD}\!\!\rightsquigarrow_{\tau}\ P \triangle Q'$›
by (*simp add*: *mono-Interrupt-FD*)

lemma *Interrupt-ev-trans-left*:
‹$P\ {}_{FD}\!\!\rightsquigarrow(ev\ e)\ P' \Longrightarrow P \triangle Q\ {}_{FD}\!\!\rightsquigarrow(ev\ e)\ P' \triangle Q$›
by (*simp add*: *AfterExt-def After-Interrupt Interrupt-τ-trans-left ready-set-Interrupt*)

lemma *Interrupt-ev-trans-right*: ‹$Q\ {}_{FD}\!\!\rightsquigarrow(ev\ e)\ Q' \Longrightarrow P \triangle Q\ {}_{FD}\!\!\rightsquigarrow(ev\ e)\ Q'$›
by (*simp add*: *AfterExt-def After-Interrupt ready-set-Interrupt*)

Throw P A Q laws

lemma *Throw-τ-trans-left*:
‹$P\ {}_{FD}\!\!\rightsquigarrow_{\tau}\ P' \Longrightarrow P\ \Theta\ a \in A.\ Q\ a\ {}_{FD}\!\!\rightsquigarrow_{\tau}\ P'\ \Theta\ a \in A.\ Q\ a$›
by (*simp add*: *mono-Throw-FD*)

lemma *Throw-τ-trans-right*:
‹$\forall a \in A.\ Q\ a\ {}_{FD}\!\!\rightsquigarrow_{\tau}\ Q'\ a \Longrightarrow P\ \Theta\ a \in A.\ Q\ a\ {}_{FD}\!\!\rightsquigarrow_{\tau}\ P\ \Theta\ a \in A.\ Q'\ a$›
by (*simp add*: *mono-Throw-FD*)

lemma *Throw-event-trans-left*:
‹$P\ {}_{FD}\!\!\rightsquigarrow e\ P' \Longrightarrow e \notin ev\ `\ A \Longrightarrow P\ \Theta\ a \in A.\ Q\ a\ {}_{FD}\!\!\rightsquigarrow e\ (P'\ \Theta\ a \in A.\ Q\ a)$›
apply (*simp add*: *AfterExt-Throw ready-set-Throw image-iff split*: *event.split*)
apply (*intro conjI impI*)
by (*metis AfterExt-def Throw-τ-trans-left event.simps*(*4*))
(*solves* ‹*simp add*: *Throw-STOP tick-trans-iff*›)

lemma *Throw-ev-trans-right*:
‹$P\ {}_{FD}\!\!\rightsquigarrow(ev\ e)\ P' \Longrightarrow e \in A \Longrightarrow P\ \Theta\ a \in A.\ Q\ a\ {}_{FD}\!\!\rightsquigarrow(ev\ e)\ (Q\ e)$›
by (*simp add*: *AfterExt-Throw ready-set-Throw split*: *event.split*)

lemma ‹*front-tickFree s* $\Longrightarrow \bot\ {}_{FD}\!\!\rightsquigarrow^{*} s\ P$›
by (*fact BOT-trace-trans-anything*)

8.2 Reality Checks

lemma ‹$STOP\ {}_{FD}\!\!\rightsquigarrow^{*} s\ P \longleftrightarrow s = [] \wedge P = STOP$›
by (*fact STOP-trace-trans-iff*)

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

lemma *F-iff-exists-trans* :
‹$(s,\ X) \in \mathcal{F}\ P \longleftrightarrow (\exists\, P'.\ (P\ {}_{FD}\!\rightsquigarrow^{*} s\ P') \wedge X \in \mathcal{R}\ P')$›
using *F-imp-exists-trace-trans leFD-imp-leF trace-trans-imp-F-if-τ-trans-imp-leF*
by *blast*

lemma *T-iff-exists-trans* : ‹$s \in \mathcal{T}\ P \longleftrightarrow (\exists\, P'.\ P\ {}_{FD}\!\rightsquigarrow^{*} s\ P')$›
by (*meson T-imp-exists-trace-trans leFD-imp-leF leF-imp-leT trace-trans-imp-T-if-τ-trans-imp-leT*)

lemma *D-iff-trace-trans-BOT*: ‹$s \in \mathcal{D}\ P \longleftrightarrow P\ {}_{FD}\!\rightsquigarrow^{*} s\ \bot$›
using *D-imp-trace-trans-BOT leFD-imp-leD trace-trans-BOT-imp-D-if-τ-trans-imp-leD*
by *blast*

8.3 Other Results

lemma *trace-trans-ready-set-subset-ready-set-AfterTrace*:
‹$P\ {}_{FD}\!\rightsquigarrow^{*} s\ Q \Longrightarrow \textit{ready-set}\ Q \subseteq \textit{ready-set}\ (P\ \textit{afterTrace}\ s)$›
by (*metis T-iff-exists-trans T-imp-trace-trans-iff-AfterTrace-τ-trans τ-trans-anti-mono-ready-set*)

lemma *trace-trans-imp-ready-set*:
‹$P\ {}_{FD}\!\rightsquigarrow^{*}(s\ @\ e\ \#\ t)\ Q \Longrightarrow e \in \textit{ready-set}\ (P\ \textit{afterTrace}\ s)$›
using *T-iff-exists-trans ready-set-AfterTrace* **by** *blast*

lemma *AfterTrace-τ-trans-if-τ-trans-imp-leT* :
‹$(P\ {}_{FD}\!\rightsquigarrow^{*} s\ Q) \longleftrightarrow s \in \mathcal{T}\ P \wedge P\ \textit{afterTrace}\ s\ {}_{FD}\!\rightsquigarrow_{\tau}\ Q$›
using *T-iff-exists-trans T-imp-trace-trans-iff-AfterTrace-τ-trans* **by** *blast*

lemma ‹*deadlock-free* $P \longleftrightarrow \textit{DF}\ \textit{UNIV}\ {}_{FD}\!\rightsquigarrow_{\tau}\ P$›
by (*simp add*: *deadlock-free-def*)

lemma ‹$\textit{deadlock-free}_{SKIP}\ P \longleftrightarrow \textit{DF}_{SKIP}\ \textit{UNIV}\ {}_{FD}\!\rightsquigarrow_{\tau}\ P$›
by (*fact deadlock-free*$_{SKIP}$*-FD*)

8.4 Summary: Operational Rules

In this section, we will just write down the operational laws that we have proven in a fancy way.

Absorbance rules

$$\frac{?P\ {}_{FD}\!\rightsquigarrow ?e\ ?P' \qquad ?P'\ {}_{FD}\!\rightsquigarrow_{\tau}\ ?P''}{?P\ {}_{FD}\!\rightsquigarrow ?e\ ?P''}$$

$$\frac{?P\ {}_{FD}\!\rightsquigarrow_{\tau}\ ?P' \qquad ?P'\ {}_{FD}\!\rightsquigarrow ?e\ ?P''}{?P\ {}_{FD}\!\rightsquigarrow ?e\ ?P''}$$

SKIP **rule**

$$\frac{}{SKIP \ {}_{FD}\!\rightsquigarrow\!\checkmark \ STOP}$$

$e \rightarrow P$ **rules**

$$\frac{?e \in ?A}{\Box a{\in}?A \rightarrow \ ?P \ a \ {}_{FD}\!\rightsquigarrow\! ev \ ?e \ ?P \ ?e} \qquad \frac{?e \in ?A}{\sqcap a{\in}?A \rightarrow \ \ ?P \ a \ {}_{FD}\!\rightsquigarrow\! ev \ ?e \ ?P \ ?e}$$

$$\frac{}{?e \rightarrow \ ?P \ {}_{FD}\!\rightsquigarrow\! ev \ ?e \ ?P}$$

($\sqcap$) **rules**

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

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

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

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

($\Box$) **rules**

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

$$\frac{?P \ {}_{FD}\!\rightsquigarrow ?e \ ?P'}{?P \ \Box \ ?Q \ {}_{FD}\!\rightsquigarrow ?e \ ?P'} \qquad \frac{?Q \ {}_{FD}\!\rightsquigarrow ?e \ ?Q'}{?P \ \Box \ ?Q \ {}_{FD}\!\rightsquigarrow ?e \ ?Q'}$$

(;) **rules**

$$\frac{?P \ {}_{FD}\!\rightsquigarrow_{\tau} \ ?P'}{?P \ ; \ ?Q \ {}_{FD}\!\rightsquigarrow_{\tau} \ ?P' \ ; \ ?Q} \qquad \frac{?P \ {}_{FD}\!\rightsquigarrow ev \ ?e \ ?P'}{?P \ ; \ ?Q \ {}_{FD}\!\rightsquigarrow ev \ ?e \ ?P' \ ; \ ?Q}$$

$$\frac{\exists \, P'. \ ?P \ {}_{FD}\!\rightsquigarrow\!\checkmark \ P'}{?P \ ; \ ?Q \ {}_{FD}\!\rightsquigarrow_{\tau} \ ?Q}$$

Hiding **rules**

$$\frac{?P \ {}_{FD}\!\rightsquigarrow_{\tau} \ ?P'}{?P \setminus ?B \ {}_{FD}\!\rightsquigarrow_{\tau} \ ?P' \setminus ?B} \qquad \frac{?P \ {}_{FD}\!\rightsquigarrow\checkmark \ ?P'}{?P \setminus ?B \ {}_{FD}\!\rightsquigarrow\checkmark \ STOP}$$

$$\frac{?P \ {}_{FD}\!\rightsquigarrow ev \ ?e \ ?P' \qquad ?e \notin ?B}{?P \setminus ?B \ {}_{FD}\!\rightsquigarrow ev \ ?e \ ?P' \setminus ?B} \qquad \frac{?P \ {}_{FD}\!\rightsquigarrow ev \ ?e \ ?P' \qquad ?e \in ?B}{?P \setminus ?B \ {}_{FD}\!\rightsquigarrow_{\tau} \ ?P' \setminus ?B}$$

Renaming **rules**

$$\frac{?P \ {}_{FD}\!\rightsquigarrow_{\tau} \ ?P'}{Renaming \ ?P \ ?f \ {}_{FD}\!\rightsquigarrow_{\tau} \ Renaming \ ?P' \ ?f}$$

$$\frac{?P \ {}_{FD}\!\rightsquigarrow\checkmark \ ?P'}{Renaming \ ?P \ ?f \ {}_{FD}\!\rightsquigarrow\checkmark \ STOP}$$

$$\frac{?f \ ?a = ?b \qquad ?P \ {}_{FD}\!\rightsquigarrow ev \ ?a \ ?P'}{Renaming \ ?P \ ?f \ {}_{FD}\!\rightsquigarrow ev \ ?b \ Renaming \ ?P' \ ?f}$$

Sync **rules**

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

$$\frac{?e \notin ?S \qquad ?P \ {}_{FD}\!\rightsquigarrow ev \ ?e \ ?P'}{?P \ [\![?S]\!] \ ?Q \ {}_{FD}\!\rightsquigarrow ev \ ?e \ ?P' \ [\![?S]\!] \ ?Q}$$

$$\frac{?e \notin ?S \qquad ?Q \ {}_{FD}\!\rightsquigarrow ev \ ?e \ ?Q'}{?P \ [\![?S]\!] \ ?Q \ {}_{FD}\!\rightsquigarrow ev \ ?e \ ?P \ [\![?S]\!] \ ?Q'}$$

$$\frac{?e \in ?S \qquad ?P \ {}_{FD}\!\rightsquigarrow ev \ ?e \ ?P' \qquad ?Q \ {}_{FD}\!\rightsquigarrow ev \ ?e \ ?Q'}{?P \ [\![?S]\!] \ ?Q \ {}_{FD}\!\rightsquigarrow ev \ ?e \ ?P' \ [\![?S]\!] \ ?Q'}$$

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

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

$$\frac{}{SKIP \ [\![?S]\!] \ SKIP \ {}_{FD}\!\rightsquigarrow_{\tau} \ SKIP}$$

($\triangleright$) **rules**

$$\frac{?P \ {}_{FD}\!\rightsquigarrow_{\tau} \ ?P'}{?P \triangleright ?Q \ {}_{FD}\!\rightsquigarrow_{\tau} \ ?P' \triangleright ?Q} \qquad \frac{?P \ {}_{FD}\!\rightsquigarrow ?e \ ?P'}{?P \triangleright ?Q \ {}_{FD}\!\rightsquigarrow ?e \ ?P'}$$

$$\frac{}{?P \rhd ?Q \ {}_{FD}\!\rightsquigarrow_{\tau} \ ?Q}$$

($\triangle$) **rules**

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

$$\frac{?P \ {}_{FD}\!\rightsquigarrow ev \ ?e \ ?P'}{?P \triangle ?Q \ {}_{FD}\!\rightsquigarrow ev \ ?e \ ?P' \triangle ?Q} \qquad \frac{?Q \ {}_{FD}\!\rightsquigarrow ev \ ?e \ ?Q'}{?P \triangle ?Q \ {}_{FD}\!\rightsquigarrow ev \ ?e \ ?Q'}$$

Throw **rules**

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

$$\frac{\forall \, a{\in}?A. \ ?Q \ a \ {}_{FD}\!\rightsquigarrow_{\tau} \ ?Q' \ a}{?P \ \Theta \ a{\in}?A. \ ?Q \ a \ {}_{FD}\!\rightsquigarrow_{\tau} \ ?P \ \Theta \ a{\in}?A. \ ?Q' \ a}$$

$$\frac{?P \ {}_{FD}\!\rightsquigarrow ?e \ ?P' \qquad ?e \notin ev \ `\ ?A}{?P \ \Theta \ a{\in}?A. \ ?Q \ a \ {}_{FD}\!\rightsquigarrow ?e \ ?P' \ \Theta \ a{\in}?A. \ ?Q \ a}$$

$$\frac{?P \ {}_{FD}\!\rightsquigarrow ev \ ?e \ ?P' \qquad ?e \in ?A}{?P \ \Theta \ a{\in}?A. \ ?Q \ a \ {}_{FD}\!\rightsquigarrow ev \ ?e \ ?Q \ ?e}$$

end

Chapter 9

Trace Divergence Operational Semantics

theory *OpSemDT*
imports *OpSemGeneric HOL−Library.LaTeXsugar*
begin

The definition with ($\sqsubseteq_{FD}$) motivates us to try with other refinements.

abbreviation *τ-trans* :: ‹'α *process* ⇒ 'α *process* ⇒ *bool*› (**infixl** ‹$_{DT}\!\rightsquigarrow_{\tau}$› *50*)
where ‹*P* $_{DT}\!\rightsquigarrow_{\tau}$ *Q* ≡ *P* $\sqsubseteq_{DT}$ *Q*›

We now instantiate the locale of *HOL−CSP-OpSem.OpSemGeneric*.

interpretation *OpSemGeneric* ‹($_{DT}\!\rightsquigarrow_{\tau}$)›
using *trans-DT* **by** *unfold-locales*
(*auto simp add*: *anti-mono-ready-set-DT mono-AfterExt-DT*)

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

lemma ‹*P* $_{DT}\!\rightsquigarrow$ *e* *P'* ⟹ *P'* $_{DT}\!\rightsquigarrow_{\tau}$ *P''* ⟹ *P* $_{DT}\!\rightsquigarrow$ *e* *P''*›
‹*P* $_{DT}\!\rightsquigarrow_{\tau}$ *P'* ⟹ *P'* $_{DT}\!\rightsquigarrow$ *e* *P''* ⟹ *P* $_{DT}\!\rightsquigarrow$ *e* *P''*›
by (*fact event-trans-τ-trans τ-trans-event-trans*)+

9.1 Operational Semantics Laws

SKIP law

lemma ‹*SKIP* $_{DT}\!\rightsquigarrow$✓ *STOP*›
by (*fact SKIP-trans-tick*)

$e \rightarrow P$ laws

lemma ‹*e* ∈ *A* ⟹ □*a* ∈ *A* → *P a* $_{DT}\!\rightsquigarrow$(*ev e*) (*P e*)›
by (*fact ev-trans-Mprefix*)

lemma ‹$e \in A \Longrightarrow \sqcap a \in A \rightarrow P\ a\ {}_{DT}\!\!\rightsquigarrow (ev\ e)\ (P\ e)$›
by (*fact ev-trans-Mndetprefix*)

lemma ‹$e \rightarrow P\ {}_{DT}\!\!\rightsquigarrow\ (ev\ e)\ P$›
by (*fact ev-trans-prefix*)

$P \sqcap Q$ laws

lemma ‹$P \sqcap Q\ {}_{DT}\!\!\rightsquigarrow_{\tau}\ P$›
and ‹$P \sqcap Q\ {}_{DT}\!\!\rightsquigarrow_{\tau}\ Q$›
by (*fact τ-trans-NdetL τ-trans-NdetR*)+

lemma ‹$a \in A \Longrightarrow (\sqcap a \in A.\ P\ a)\ {}_{DT}\!\!\rightsquigarrow_{\tau}\ P\ a$›
by (*fact τ-trans-GlobalNdet*)

lemma ‹*finite* $A \Longrightarrow a \in A \Longrightarrow (\bigsqcap a \in A.\ P\ a)\ {}_{DT}\!\!\rightsquigarrow_{\tau}\ P\ a$›
by (*fact τ-trans-MultiNdet*)

$\mu\ x.\ f\ x$ law

lemma ‹*cont* $f \Longrightarrow P = (\mu\ X.\ f\ X) \Longrightarrow P\ {}_{DT}\!\!\rightsquigarrow_{\tau}\ f\ P$›
by (*fact fix-point-τ-trans*)

$P \Box Q$ laws, more powerful

lemma *τ-trans-DetL*: ‹$P \Box Q\ {}_{DT}\!\!\rightsquigarrow_{\tau}\ P$›
and *τ-trans-DetR*: ‹$P \Box Q\ {}_{DT}\!\!\rightsquigarrow_{\tau}\ Q$›
by (*metis Det-STOP τ-trans-eq leDT-STOP mono-Det-DT*)
(*metis Det-STOP Det-commute τ-trans-eq leDT-STOP mono-Det-DT*)

lemma *τ-trans-MultiDet*: ‹*finite* $A \Longrightarrow a \in A \Longrightarrow (\Box a \in A.\ P\ a)\ {}_{DT}\!\!\rightsquigarrow_{\tau}\ P\ a$›
by (*metis MultiDet-insert′ τ-trans-DetL insert-absorb*)

P **;** Q laws

lemma *τ-trans-SeqL*: ‹$P\ {}_{DT}\!\!\rightsquigarrow_{\tau}\ P' \Longrightarrow P$ **;** $Q\ {}_{DT}\!\!\rightsquigarrow_{\tau}\ P'$ **;** Q›
by *simp*

lemma *ev-trans-SeqL*: ‹$P\ {}_{DT}\!\!\rightsquigarrow (ev\ e)\ P' \Longrightarrow P$ **;** $Q\ {}_{DT}\!\!\rightsquigarrow (ev\ e)\ P'$ **;** Q›

by (*auto simp add*: *ready-set-Seq AfterExt-Seq*)

lemma *τ-trans-SeqR*: ‹$\exists P'.\ P\ {}_{DT}\!\!\rightsquigarrow\checkmark\ P' \Longrightarrow P$ **;** $Q\ {}_{DT}\!\!\rightsquigarrow_{\tau}\ Q$›
by (*metis SKIP-Seq mono-Seq-DT-left ready-tick-imp-τ-trans-SKIP*)

lemma ‹$\checkmark \in$ *ready-set* $P \Longrightarrow Q\ {}_{DT}\!\!\rightsquigarrow (ev\ e)\ Q' \Longrightarrow P$ **;** $Q\ {}_{DT}\!\!\rightsquigarrow (ev\ e)\ Q'$›

by (*fact ev-trans-SeqR*)

$P \setminus B$ laws

lemma *τ-trans-Hiding*: ‹$P\ {}_{DT}\!\!\rightsquigarrow_{\tau}\ P' \Longrightarrow P \setminus B\ {}_{DT}\!\!\rightsquigarrow_{\tau}\ P' \setminus B$›
by (*fact mono-Hiding-DT*)

lemma *ev-trans-Hiding-notin*:
‹$P\ {}_{DT}\!\!\rightsquigarrow(ev\ e)\ P' \Longrightarrow e \notin B \Longrightarrow P \setminus B\ {}_{DT}\!\!\rightsquigarrow(ev\ e)\ P' \setminus B$›
by (*meson AfterExt-Hiding-DT-Hiding-AfterExt-if-ready-notin*
mono-Hiding-DT ready-notin-imp-ready-Hiding trans-DT)

lemma ‹$P\ {}_{DT}\!\!\rightsquigarrow\checkmark\ P' \Longrightarrow P \setminus B\ {}_{DT}\!\!\rightsquigarrow\checkmark\ STOP$›
by (*fact tick-trans-Hiding*)

lemma *ev-trans-Hiding-inside*:
‹$P\ {}_{DT}\!\!\rightsquigarrow(ev\ e)\ P' \Longrightarrow e \in B \Longrightarrow P \setminus B\ {}_{DT}\!\!\rightsquigarrow_{\tau}\ P' \setminus B$›
by (*meson Hiding-DT-Hiding-AfterExt-if-ready-inside mono-Hiding-DT trans-DT*)

Renaming P f laws

lemma *τ-trans-Renaming*:
‹$P\ {}_{DT}\!\!\rightsquigarrow_{\tau}\ P' \Longrightarrow Renaming\ P\ f\ {}_{DT}\!\!\rightsquigarrow_{\tau}\ Renaming\ P'\ f$›
by (*fact mono-Renaming-DT*)

lemma *tick-trans-Renaming*: ‹$P\ {}_{DT}\!\!\rightsquigarrow\checkmark\ P' \Longrightarrow Renaming\ P\ f\ {}_{DT}\!\!\rightsquigarrow\checkmark\ STOP$›
by (*simp add*: *AfterExt-def ready-set-Renaming tick-eq-EvExt*)

lemma *ev-trans-Renaming*:
‹$f\ a = b \Longrightarrow P\ {}_{DT}\!\!\rightsquigarrow(ev\ a)\ P' \Longrightarrow Renaming\ P\ f\ {}_{DT}\!\!\rightsquigarrow(ev\ b)\ (Renaming\ P'\ f)$›
apply (*simp add*: *AfterExt-Renaming Renaming-BOT ready-set-BOT ready-set-Renaming*)
apply (*intro conjI impI*)
apply (*meson ev-elem-anteced1 imageI vimageI2*)
apply (*rule τ-trans-transitivity*[*of* - ‹*Renaming* (*P afterExt ev a*) *f*›])
apply (*solves* ‹*rule τ-trans-GlobalNdet*, *simp*›)
by (*simp add*: *τ-trans-Renaming*)

lemma ‹$P\ {}_{DT}\!\!\rightsquigarrow_{\tau}\ P' \Longrightarrow P\ [\![a := b]\!]\ {}_{DT}\!\!\rightsquigarrow_{\tau}\ P'\ [\![a := b]\!]$›
by (*fact τ-trans-Renaming*)

lemma ‹$P\ {}_{DT}\!\!\rightsquigarrow\checkmark\ P' \Longrightarrow P\ [\![a := b]\!]\ {}_{DT}\!\!\rightsquigarrow\checkmark\ STOP$›
by (*fact tick-trans-Renaming*)

lemma *ev-trans-RenamingF*:
‹$P\ {}_{DT}\!\!\rightsquigarrow(ev\ a)\ P' \Longrightarrow P\ [\![a := b]\!]\ {}_{DT}\!\!\rightsquigarrow(ev\ b)\ P'\ [\![a := b]\!]$›
by (*metis ev-trans-Renaming fun-upd-same*)

$P\ [\![S]\!]\ Q$ laws

lemma *τ-trans-SyncL*: ‹$P\ {}_{DT}\!\!\rightsquigarrow_{\tau}\ P' \Longrightarrow P\ [\![S]\!]\ Q\ {}_{DT}\!\!\rightsquigarrow_{\tau}\ P'\ [\![S]\!]\ Q$ ›
and *τ-trans-SyncR*: ‹$Q\ {}_{DT}\!\!\rightsquigarrow_{\tau}\ Q' \Longrightarrow P\ [\![S]\!]\ Q\ {}_{DT}\!\!\rightsquigarrow_{\tau}\ P\ \ [\![S]\!]\ Q'$›
by *simp-all*

lemma *ev-trans-SyncL*:
‹$e \notin S \Longrightarrow P\ {}_{DT}\!\rightsquigarrow\!(ev\ e)\ P' \Longrightarrow P\ [\![S]\!]\ Q\ {}_{DT}\!\rightsquigarrow\!(ev\ e)\ P'\ [\![S]\!]\ Q$›
and *ev-trans-SyncR*:
‹$e \notin S \Longrightarrow Q\ {}_{DT}\!\rightsquigarrow\!(ev\ e)\ Q' \Longrightarrow P\ [\![S]\!]\ Q\ {}_{DT}\!\rightsquigarrow\!(ev\ e)\ P\ [\![S]\!]\ Q'$›
by (*simp-all add*: *AfterExt-Sync ready-set-Sync image-iff*)

lemma *ev-trans-SyncLR*:
‹$[\![e \in S;\ P\ {}_{DT}\!\rightsquigarrow\!(ev\ e)\ P';\ Q\ {}_{DT}\!\rightsquigarrow\!(ev\ e)\ Q']\!] \Longrightarrow$
$P\ [\![S]\!]\ Q\ {}_{DT}\!\rightsquigarrow\!(ev\ e)\ P'\ [\![S]\!]\ Q'$›
by (*simp add*: *AfterExt-Sync ready-set-Sync*)

From here we slightly defer from Roscoe's laws for *Sync*: we obtain the following rules for *SKIP* instead of *STOP*.

lemma *tick-trans-SyncL*: ‹$P\ {}_{DT}\!\rightsquigarrow\!\checkmark\ P' \Longrightarrow P\ [\![S]\!]\ Q\ {}_{DT}\!\rightsquigarrow_\tau\ SKIP\ [\![S]\!]\ Q$›
and *tick-trans-SyncR*: ‹$Q\ {}_{DT}\!\rightsquigarrow\!\checkmark\ Q' \Longrightarrow P\ [\![S]\!]\ Q\ {}_{DT}\!\rightsquigarrow_\tau\ P\ [\![S]\!]\ SKIP$›
by (*simp-all add*: *ready-tick-imp-τ-trans-SKIP*)

lemma *tick-trans-SKIP-Sync-SKIP*: ‹$SKIP\ [\![S]\!]\ SKIP\ {}_{DT}\!\rightsquigarrow\!\checkmark\ STOP$›
by (*simp add*: *SKIP-trans-tick Sync-SKIP-SKIP*)

lemma *τ-trans-SKIP-Sync-SKIP*: ‹$SKIP\ [\![S]\!]\ SKIP\ {}_{DT}\!\rightsquigarrow_\tau\ SKIP$›
by (*simp add*: *Sync-SKIP-SKIP*)

$P \rhd Q$ laws

lemma *Sliding-τ-trans-left*: ‹$P\ {}_{DT}\!\rightsquigarrow_\tau\ P' \Longrightarrow P \rhd Q\ {}_{DT}\!\rightsquigarrow_\tau\ P' \rhd Q$›
unfolding *Sliding-def* **by** *simp*

lemma ‹$P\ {}_{DT}\!\rightsquigarrow e\ P' \Longrightarrow P \rhd Q\ {}_{DT}\!\rightsquigarrow e\ P'$›
by (*fact Sliding-event-transL*)

lemma ‹$P \rhd Q\ {}_{DT}\!\rightsquigarrow_\tau\ Q$›
by (*fact Sliding-τ-transR*)

$P \triangle Q$ laws

lemma *Interrupt-τ-trans-left*: ‹$P\ {}_{DT}\!\rightsquigarrow_\tau\ P' \Longrightarrow P \triangle Q\ {}_{DT}\!\rightsquigarrow_\tau\ P' \triangle Q$›
by (*simp add*: *mono-Interrupt-DT*)

lemma *Interrupt-τ-trans-right*: ‹$Q\ {}_{DT}\!\rightsquigarrow_\tau\ Q' \Longrightarrow P \triangle Q\ {}_{DT}\!\rightsquigarrow_\tau\ P \triangle Q'$›
by (*simp add*: *mono-Interrupt-DT*)

lemma *Interrupt-ev-trans-left*:
‹$P\ {}_{DT}\!\rightsquigarrow\!(ev\ e)\ P' \Longrightarrow P \triangle Q\ {}_{DT}\!\rightsquigarrow\!(ev\ e)\ P' \triangle Q$›
by (*simp add*: *AfterExt-def After-Interrupt Interrupt-τ-trans-left ready-set-Interrupt*)

lemma *Interrupt-ev-trans-right*: ‹$Q\ {}_{DT}\!\rightsquigarrow\!(ev\ e)\ Q' \Longrightarrow P \triangle Q\ {}_{DT}\!\rightsquigarrow\!(ev\ e)\ Q'$›
by (*simp add*: *AfterExt-def After-Interrupt ready-set-Interrupt*)

Throw P A Q laws

lemma *Throw-τ-trans-left*:

$\langle P \ {}_{DT}\!\rightsquigarrow_{\tau} \ P' \Longrightarrow P \ \Theta \ a \in A. \ Q \ a \ {}_{DT}\!\rightsquigarrow_{\tau} \ P' \ \Theta \ a \in A. \ Q \ a \rangle$
by (*simp add*: *mono-Throw-DT*)

lemma *Throw-τ-trans-right*:
$\langle \forall \, a \in A. \ Q \ a \ {}_{DT}\!\rightsquigarrow_{\tau} \ Q' \ a \Longrightarrow P \ \Theta \ a \in A. \ Q \ a \ {}_{DT}\!\rightsquigarrow_{\tau} \ P \ \Theta \ a \in A. \ Q' \ a \rangle$
by (*simp add*: *mono-Throw-DT*)

lemma *Throw-event-trans-left*:
$\langle P \ {}_{DT}\!\rightsquigarrow e \ P' \Longrightarrow e \notin ev \ `\ A \Longrightarrow P \ \Theta \ a \in A. \ Q \ a \ {}_{DT}\!\rightsquigarrow e \ (P' \ \Theta \ a \in A. \ Q \ a) \rangle$
apply (*simp add*: *AfterExt-Throw ready-set-Throw image-iff split*: *event.split*)
apply (*intro conjI impI*)
by (*metis AfterExt-def Throw-τ-trans-left event.simps*(4))
(*solves* ‹*simp add*: *Throw-STOP tick-trans-iff*›)

lemma *Throw-ev-trans-right*:
$\langle P \ {}_{DT}\!\rightsquigarrow (ev \ e) \ P' \Longrightarrow e \in A \Longrightarrow P \ \Theta \ a \in A. \ Q \ a \ {}_{DT}\!\rightsquigarrow (ev \ e) \ (Q \ e) \rangle$
by (*simp add*: *AfterExt-Throw ready-set-Throw split*: *event.split*)

lemma ‹*front-tickFree s* $\Longrightarrow \bot \ {}_{DT}\!\rightsquigarrow^* s \ P$›
by (*fact BOT-trace-trans-anything*)

9.2 Reality Checks

lemma $\langle STOP \ {}_{DT}\!\rightsquigarrow^* s \ P \longleftrightarrow s = [] \wedge P = STOP \rangle$
by (*fact STOP-trace-trans-iff*)

lemma *T-iff-exists-trans* : $\langle s \in \mathcal{T} \ P \longleftrightarrow (\exists \, P'. \ P \ {}_{DT}\!\rightsquigarrow^* s \ P') \rangle$
using *T-imp-exists-trace-trans leDT-imp-leT trace-trans-imp-T-if-τ-trans-imp-leT*
by *blast*

lemma *D-iff-trace-trans-BOT*: $\langle s \in \mathcal{D} \ P \longleftrightarrow P \ {}_{DT}\!\rightsquigarrow^* s \ \bot \rangle$
using *D-imp-trace-trans-BOT leDT-imp-leD trace-trans-BOT-imp-D-if-τ-trans-imp-leD*
by *blast*

9.3 Other Results

lemma *trace-trans-ready-set-subset-ready-set-AfterTrace*:
$\langle P \ {}_{DT}\!\rightsquigarrow^* s \ Q \Longrightarrow ready\text{-}set \ Q \subseteq ready\text{-}set \ (P \ afterTrace \ s) \rangle$
by (*metis T-iff-exists-trans T-imp-trace-trans-iff-AfterTrace-τ-trans τ-trans-anti-mono-ready-set*)

lemma *trace-trans-imp-ready-set*:
$\langle P \ {}_{DT}\!\rightsquigarrow^* (s \ @ \ e \ \# \ t) \ Q \Longrightarrow e \in ready\text{-}set \ (P \ afterTrace \ s) \rangle$
using *T-iff-exists-trans ready-set-AfterTrace* **by** *blast*

lemma *AfterTrace-τ-trans-if-τ-trans-imp-leT* :
‹$(P\ {}_{DT}\!\rightsquigarrow^{*} s\ Q) \longleftrightarrow s \in \mathcal{T}\ P \wedge P\ afterTrace\ s\ {}_{DT}\!\rightsquigarrow_{\tau}\ Q$›
using *T-iff-exists-trans T-imp-trace-trans-iff-AfterTrace-τ-trans* **by** *blast*

9.4 Summary: Operational Rules

In this section, we will just write down the operational laws that we have proven in a fancy way.

Absorbance rules

$$\frac{?P\ {}_{DT}\!\rightsquigarrow ?e\ ?P' \qquad ?P'\ {}_{DT}\!\rightsquigarrow_{\tau}\ ?P''}{?P\ {}_{DT}\!\rightsquigarrow ?e\ ?P''}$$

$$\frac{?P\ {}_{DT}\!\rightsquigarrow_{\tau}\ ?P' \qquad ?P'\ {}_{DT}\!\rightsquigarrow ?e\ ?P''}{?P\ {}_{DT}\!\rightsquigarrow ?e\ ?P''}$$

SKIP **rule**

$$\frac{}{SKIP\ {}_{DT}\!\rightsquigarrow\checkmark\ STOP}$$

$e \rightarrow P$ **rules**

$$\frac{?e \in ?A}{\Box a{\in}?A \rightarrow ?P\ a\ {}_{DT}\!\rightsquigarrow ev\ ?e\ ?P\ ?e} \qquad \frac{?e \in ?A}{\sqcap a{\in}?A \rightarrow\ ?P\ a\ {}_{DT}\!\rightsquigarrow ev\ ?e\ ?P\ ?e}$$

$$\frac{}{?e \rightarrow ?P\ {}_{DT}\!\rightsquigarrow ev\ ?e\ ?P}$$

$(\sqcap)$ **rules**

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

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

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

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

(□) **rules (more powerful than in OpSemFD)**

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

(;) **rules**

$$\frac{?P \;{}_{DT}\!\rightsquigarrow_{\tau}\; ?P'}{?P \;;\; ?Q \;{}_{DT}\!\rightsquigarrow_{\tau}\; ?P' \;;\; ?Q} \qquad \frac{?P \;{}_{DT}\!\rightsquigarrow ev\; ?e\; ?P'}{?P \;;\; ?Q \;{}_{DT}\!\rightsquigarrow ev\; ?e\; ?P' \;;\; ?Q}$$

$$\frac{\exists\, P'.\ ?P \;{}_{DT}\!\rightsquigarrow\checkmark\; P'}{?P \;;\; ?Q \;{}_{DT}\!\rightsquigarrow_{\tau}\; ?Q}$$

Hiding **rules**

$$\frac{?P \;{}_{DT}\!\rightsquigarrow_{\tau}\; ?P'}{?P \setminus ?B \;{}_{DT}\!\rightsquigarrow_{\tau}\; ?P' \setminus ?B} \qquad \frac{?P \;{}_{DT}\!\rightsquigarrow\checkmark\; ?P'}{?P \setminus ?B \;{}_{DT}\!\rightsquigarrow\checkmark\; STOP}$$

$$\frac{?P \;{}_{DT}\!\rightsquigarrow ev\; ?e\; ?P' \qquad ?e \notin ?B}{?P \setminus ?B \;{}_{DT}\!\rightsquigarrow ev\; ?e\; ?P' \setminus ?B} \qquad \frac{?P \;{}_{DT}\!\rightsquigarrow ev\; ?e\; ?P' \qquad ?e \in ?B}{?P \setminus ?B \;{}_{DT}\!\rightsquigarrow_{\tau}\; ?P' \setminus ?B}$$

Renaming **rules**

$$\frac{?P \;{}_{DT}\!\rightsquigarrow_{\tau}\; ?P'}{Renaming\; ?P\; ?f \;{}_{DT}\!\rightsquigarrow_{\tau}\; Renaming\; ?P'\; ?f}$$

$$\frac{?P \;{}_{DT}\!\rightsquigarrow\checkmark\; ?P'}{Renaming\; ?P\; ?f \;{}_{DT}\!\rightsquigarrow\checkmark\; STOP}$$

$$\frac{?f\; ?a = ?b \qquad ?P \;{}_{DT}\!\rightsquigarrow ev\; ?a\; ?P'}{Renaming\; ?P\; ?f \;{}_{DT}\!\rightsquigarrow ev\; ?b\; Renaming\; ?P'\; ?f}$$

Sync **rules**

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

$$\frac{?e \notin ?S \qquad ?P \;{}_{DT}\!\rightsquigarrow ev\; ?e\; ?P'}{?P \;[\![?S]\!]\; ?Q \;{}_{DT}\!\rightsquigarrow ev\; ?e\; ?P' \;[\![?S]\!]\; ?Q}$$

$$\frac{?e \notin ?S \qquad ?Q \;{}_{DT}\!\rightsquigarrow ev\; ?e\; ?Q'}{?P \;[\![?S]\!]\; ?Q \;{}_{DT}\!\rightsquigarrow ev\; ?e\; ?P \;[\![?S]\!]\; ?Q'}$$

$$\frac{?e \in ?S \qquad ?P \;{}_{DT}\!\rightsquigarrow ev\; ?e\; ?P' \qquad ?Q \;{}_{DT}\!\rightsquigarrow ev\; ?e\; ?Q'}{?P \;[\![?S]\!]\; ?Q \;{}_{DT}\!\rightsquigarrow ev\; ?e\; ?P' \;[\![?S]\!]\; ?Q'}$$

$$\frac{?P \ {}_{DT}\!\rightsquigarrow\!\checkmark \ ?P'}{?P \ [\![?S]\!] \ ?Q \ {}_{DT}\!\rightsquigarrow_{\tau} \ SKIP \ [\![?S]\!] \ ?Q} \qquad \frac{?Q \ {}_{DT}\!\rightsquigarrow\!\checkmark \ ?Q'}{?P \ [\![?S]\!] \ ?Q \ {}_{DT}\!\rightsquigarrow_{\tau} \ ?P \ [\![?S]\!] \ SKIP}$$

$$\frac{}{SKIP \ [\![?S]\!] \ SKIP \ {}_{DT}\!\rightsquigarrow_{\tau} \ SKIP}$$

($\triangleright$) **rules**

$$\frac{?P \ {}_{DT}\!\rightsquigarrow_{\tau} \ ?P'}{?P \triangleright ?Q \ {}_{DT}\!\rightsquigarrow_{\tau} \ ?P' \triangleright ?Q} \qquad \frac{?P \ {}_{DT}\!\rightsquigarrow ?e \ ?P'}{?P \triangleright ?Q \ {}_{DT}\!\rightsquigarrow ?e \ ?P'}$$

$$\frac{}{?P \triangleright ?Q \ {}_{DT}\!\rightsquigarrow_{\tau} \ ?Q}$$

($\triangle$) **rules**

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

$$\frac{?P \ {}_{DT}\!\rightsquigarrow ev \ ?e \ ?P'}{?P \triangle ?Q \ {}_{DT}\!\rightsquigarrow ev \ ?e \ ?P' \triangle ?Q} \qquad \frac{?Q \ {}_{DT}\!\rightsquigarrow ev \ ?e \ ?Q'}{?P \triangle ?Q \ {}_{DT}\!\rightsquigarrow ev \ ?e \ ?Q'}$$

Throw **rules**

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

$$\frac{\forall \, a{\in}?A. \ ?Q \ a \ {}_{DT}\!\rightsquigarrow_{\tau} \ ?Q' \ a}{?P \ \Theta \ a{\in}?A. \ ?Q \ a \ {}_{DT}\!\rightsquigarrow_{\tau} \ ?P \ \Theta \ a{\in}?A. \ ?Q' \ a}$$

$$\frac{?P \ {}_{DT}\!\rightsquigarrow ?e \ ?P' \qquad ?e \notin ev \ `\ ?A}{?P \ \Theta \ a{\in}?A. \ ?Q \ a \ {}_{DT}\!\rightsquigarrow ?e \ ?P' \ \Theta \ a{\in}?A. \ ?Q \ a}$$

$$\frac{?P \ {}_{DT}\!\rightsquigarrow ev \ ?e \ ?P' \qquad ?e \in ?A}{?P \ \Theta \ a{\in}?A. \ ?Q \ a \ {}_{DT}\!\rightsquigarrow ev \ ?e \ ?Q \ ?e}$$

end

Chapter 10

Extension of the After Operator, bis

theory *AfterExtBis*
imports *After*
begin

10.1 The AfterExt Operator, bis

10.1.1 Definition

The refinements ($\sqsubseteq_F$) and ($\sqsubseteq_T$) are not verifying the locale axioms. In order to make the constructions available for these refinements, we will slightly modify the definition of AfterExt.

If the event is ✓ we obtain *STOP* anyway, even if the process was $\perp$.

At first this appears a little weird, but can be interpreted as the fact that even if a process if diverging, after accepting ✓ it has to stop.

definition *AfterExt* :: ‹['α *process*, 'α *event*] ⇒ 'α *process*› (**infixl** ‹*afterExt*› 77)
where ‹*P afterExt e* ≡ *case e of ev x* ⇒ *P after x* | ✓ ⇒ *STOP*›

lemma *not-ready-AfterExt*: ‹*e* ∉ *ready-set P* ⟹ *P afterExt e* = *STOP*›
by (*auto simp add*: *AfterExt-def intro*: *not-ready-After split*: *event.split*)

lemma *ready-set-AfterExt*:
‹*ready-set* (*P afterExt e*) = (*if e* = ✓ *then* {} *else* {*a*. *e* # [*a*] ∈ $\mathcal{T}$ *P*})›
by (*simp add*: *AfterExt-def ready-set-After ready-set-STOP*
split: *event.split*)

10.1.2 Projections

lemma *F-AfterExt*:
‹$\mathcal{F}$ (*P afterExt e*) =

(*if* $e \in$ *ready-set* P *then* $\{(tl\ s,\ X) |\ s\ X.\ (s,\ X) \in \mathcal{F}\ P \wedge s \neq [] \wedge hd\ s = e\}$
  else $\{(s,\ X).\ s = []\})$›
 (**is** ‹- = *?rhs*›)
proof (*unfold AfterExt-def*, *split event.split*, *intro conjI allI impI*)
 show ‹$e = ev\ x \Longrightarrow \mathcal{F}\ (P\ after\ x) =$ *?rhs*› **for** x
  by (*simp add*: *F-After*)
next
 show ‹$e = \checkmark \Longrightarrow \mathcal{F}$ *STOP* = *?rhs*›
  by (*simp add*: *F-UU F-STOP BOT-iff-D ready-set-def set-eq-iff*)
   (*metis F-T T-nonTickFree-imp-decomp append-single-T-imp-tickFree hd-append2*
       hd-in-set list.discI list.sel(*1*) *list.sel*(*3*) *tickFree-def tick-T-F*)
qed

lemma *D-AfterExt*: ‹$\mathcal{D}\ (P\ afterExt\ e) = (\ $ *if* $e = \checkmark \wedge P = \bot$ *then* $\{\}$
  else $\{tl\ s |\ s\ .\ s \in \mathcal{D}\ P \wedge s \neq [] \wedge hd\ s = e\})$›
 (**is** ‹- = *?rhs*›)
proof (*unfold AfterExt-def*, *split event.split*, *intro conjI allI impI*)
 show ‹$e = ev\ x \Longrightarrow \mathcal{D}\ (P\ after\ x) =$ *?rhs*› **for** x
  by (*simp add*: *D-After*)
   (*metis Cons-in-T-imp-elem-ready-set D-T list.exhaust-sel*)
next
 show ‹$e = \checkmark \Longrightarrow \mathcal{D}$ *STOP* = *?rhs*›
  by (*simp add*: *D-UU D-STOP BOT-iff-D*)
   (*metis D-imp-front-tickFree front-tickFree-implies-tickFree hd-append2*
       hd-in-set is-processT9 nonTickFree-n-frontTickFree tickFree-def)
qed

lemma *T-AfterExt*: ‹$\mathcal{T}\ (P\ afterExt\ e) = (\ $ *if* $e = \checkmark \wedge P = \bot$ *then* $\{[]\}$
  else insert $[]$ $\{tl\ s |\ s\ .\ s \in \mathcal{T}\ P \wedge s \neq [] \wedge hd\ s = e\})$›
 (**is** ‹- = *?rhs*›)
proof (*unfold AfterExt-def*, *split event.split*, *intro conjI allI impI*)
 show ‹$e = ev\ x \Longrightarrow \mathcal{T}\ (P\ after\ x) =$ *?rhs*› **for** x
  by (*simp add*: *T-After set-eq-iff subset-iff*)
   (*metis Cons-in-T-imp-elem-ready-set list.collapse*
       list.distinct(*1*) *list.sel*(*1*, *3*) *mem-Collect-eq ready-set-def*)
next
 show ‹$e = \checkmark \Longrightarrow \mathcal{T}$ *STOP* = *?rhs*›
 by (*simp add*: *T-After T-UU T-STOP subset-iff*)
  (*metis front-tickFree-charn hd-append2 hd-in-set*
      is-processT2-TR list.sel(*3*) *tickFree-def tl-append-if*)
qed

10.1.3 Monotony

lemma *mono-AfterExt* : ‹$P \sqsubseteq Q \Longrightarrow P\ afterExt\ e \sqsubseteq Q\ afterExt\ e$›
 by (*auto simp add*: *AfterExt-def mono-After split*: *event.split*)

lemma *mono-AfterExt-T* : ‹$P \sqsubseteq_T Q \Longrightarrow P$ *afterExt* $e \sqsubseteq_T Q$ *afterExt* e›
by (*auto simp add*: *AfterExt-def mono-After-T split*: *event.split*)

lemma *mono-AfterExt-F* :
‹$P \sqsubseteq_F Q \Longrightarrow ev\ e \notin$ *ready-set* $P \vee ev\ e \in$ *ready-set* $Q \Longrightarrow$
P *afterExt* $ev\ e \sqsubseteq_F Q$ *afterExt* $ev\ e$›
by (*simp add*: *AfterExt-def mono-After-F*)

lemma *mono-AfterExt-D* : ‹$P \sqsubseteq_D Q \Longrightarrow P$ *afterExt* $e \sqsubseteq_D Q$ *afterExt* e›
by (*simp add*: *AfterExt-def mono-After-D split*: *event.split*)

lemma *mono-AfterExt-FD* :
‹$P \sqsubseteq_{FD} Q \Longrightarrow e \notin$ *ready-set* $P \vee e \in$ *ready-set* $Q \Longrightarrow$
P *afterExt* $e \sqsubseteq_{FD} Q$ *afterExt* e›
by (*auto simp add*: *AfterExt-def mono-After-FD split*: *event.split*)

lemma *mono-AfterExt-DT* : ‹$P \sqsubseteq_{DT} Q \Longrightarrow P$ *afterExt* $e \sqsubseteq_{DT} Q$ *afterExt* e›
by (*simp add*: *AfterExt-def mono-After-DT split*: *event.split*)

10.1.4 Behaviour of *AfterExt* with *STOP*, *SKIP* and $\bot$

lemma *AfterExt-STOP*: ‹*STOP afterExt* e = *STOP*›
by (*simp add*: *STOP-neq-BOT STOP-iff-T T-AfterExt ready-set-STOP T-STOP*)

lemma *AfterExt-is-STOP-iff*:
‹P *afterExt* e = *STOP* $\longleftrightarrow P = \bot \wedge e = \checkmark \vee (\forall s.\ e\ \#\ s \in \mathcal{T}\ P \longrightarrow s = [])$›
by (*simp add*: *AfterExt-def After-is-STOP-iff split*: *event.split*)
(*metis T-nonTickFree-imp-decomp append-single-T-imp-tickFree*
butlast.simps(*2*) *butlast-snoc tickFree-Cons*)

lemma *AfterExt-SKIP*: ‹*SKIP afterExt* e = *STOP*›
by (*auto simp add*: *SKIP-neq-BOT STOP-iff-T T-AfterExt ready-set-SKIP T-SKIP*)

lemma *AfterExt-BOT* : ‹$\bot$ *afterExt* e = (*if* $e = \checkmark$ *then STOP else* $\bot$)›
by (*metis AfterExt-def After-BOT event.case event.exhaust*)

lemma *AfterExt-is-BOT-iff*: ‹P *afterExt* $e = \bot \longleftrightarrow e \neq \checkmark \wedge [e] \in \mathcal{D}\ P$›
by (*simp add*: *AfterExt-def After-is-BOT-iff STOP-neq-BOT split*: *event.split*)

10.1.5 Behaviour of *AfterExt* with Operators of HOL-CSP

Here again, we lose determinism.

lemma *AfterExt-Mprefix-is-AfterExt-Mndetprefix*:
‹$(\Box a \in A \rightarrow P\ a)$ *afterExt* $e = (\sqcap a \in A \rightarrow P\ a)$ *afterExt* e›
by (*simp add*: *AfterExt-def After-Mprefix-is-After-Mndetprefix*
Mprefix-neq-BOT Mndetprefix-neq-BOT split: *event.split*)

lemma *AfterExt-Det-is-AfterExt-Ndet*: ‹*P* □ *Q afterExt e* = *P* ⊓ *Q afterExt e*›
by (*simp add*: *AfterExt-def After-Det-is-After-Ndet Det-is-BOT-iff Ndet-is-BOT-iff split*: *event.split*)

lemma *AfterExt-Ndet*:
‹*P* ⊓ *Q afterExt e* = (*case e of ev x* ⇒ *if ev x* ∈ *ready-set P* ∩ *ready-set Q*
then (*P afterExt ev x*) ⊓ (*Q afterExt ev x*)
else if ev x ∈ *ready-set P*
then P afterExt ev x
else Q afterExt ev x
| ✓ ⇒ *STOP*)›
by (*auto simp add*: *AfterExt-def After-Ndet not-ready-After split*: *event.split*)

lemma *AfterExt-Mprefix*:
‹(□ *a* ∈ *A* → *P a*) *afterExt e* =
(*case e of ev x* ⇒ *if x* ∈ *A then P x else STOP* | ✓ ⇒ *STOP*)›
by (*simp add*: *AfterExt-def Mprefix-neq-BOT After-Mprefix split*: *event.split*)

corollary *AfterExt-prefix*:
‹(*a* → *P*) *afterExt e* =
(*case e of ev x* ⇒ *if x* = *a then P else STOP* | ✓ ⇒ *STOP*)›
unfolding *write0-def* **by** (*simp add*: *AfterExt-Mprefix split*: *event.split*)

lemmas *AfterExt-Det* = *AfterExt-Ndet*[*folded AfterExt-Det-is-AfterExt-Ndet*]
and *AfterExt-Mndetprefix* = *AfterExt-Mprefix*[*unfolded AfterExt-Mprefix-is-AfterExt-Mndetprefix*]

lemma *Renaming-is-BOT-iff*: ‹*Renaming P f* = ⊥ ⟷ *P* = ⊥›
apply (*intro iffI*)
apply (*simp add*: *BOT-iff-D D-Renaming*)
by (*simp add*: *Renaming-BOT*)

lemma *Renaming-is-STOP-iff*: ‹*Renaming P f* = *STOP* ⟷ *P* = *STOP*›
apply (*intro iffI*, *simp-all add*: *Renaming-STOP*)
by (*auto simp add*: *STOP-iff-T T-Renaming intro*: *Nil-elem-T*)

lemma *AfterExt-Renaming*:
‹*Renaming P f afterExt e* =
(*case e of* ✓ ⇒ *STOP*
| *ev a* ⇒ *if P* = ⊥ *then* ⊥ *else*
⊓*a* ∈ {*a*. *ev a* ∈ *ready-set P* ∧ *ev* (*f a*) = *e*}. *Renaming* (*P afterExt ev a*) *f*)›
by (*simp add*: *AfterExt-def After-Renaming Renaming-is-BOT-iff split*: *event.split*)

— Move this result in HOL-CSP
lemma *Seq-is-BOT-iff*: ‹P ; $Q = \bot \longleftrightarrow P = \bot \vee ([\checkmark] \in \mathcal{T}\ P \wedge Q = \bot)$›
by (*auto simp add*: *BOT-iff-D D-Seq D-UU*)

lemma *AfterExt-Seq*:
‹(P ; Q) *afterExt* e =
(*if* $e \notin$ *ready-set* $P \wedge e \notin$ *ready-set* Q *then* *STOP*
else if $e \notin$ *ready-set* Q *then* P *afterExt* e ; Q
else if $e \notin$ *ready-set* P *then if* ✓ $\in$ *ready-set* P *then* Q *afterExt* e *else* *STOP*
else if ✓ $\in$ *ready-set* P *then* (P *afterExt* e ; Q) $\sqcap$ (Q *afterExt* e)
else P *afterExt* e ; Q)›
by (*simp add*: *AfterExt-def After-Seq Ndet-id Ndet-is-BOT-iff*
Seq-is-BOT-iff ready-set-def T-UU STOP-Seq split: *event.split*)

theorem *AfterExt-Sync*:
‹(P $[\![S]\!]$ Q) *afterExt* e =
(*case* e *of* ✓ ⇒ *STOP*
| *ev* x ⇒ *if* $P = \bot \vee Q = \bot$ *then* $\bot$
else if $e \in$ *ready-set* $P \wedge e \in$ *ready-set* Q
then *if* $x \in S$
then P *afterExt* e $[\![S]\!]$ Q *afterExt* e
else (P *afterExt* e $[\![S]\!]$ Q) $\sqcap$ (P $[\![S]\!]$ Q *afterExt* e)
else *if* $e \in$ *ready-set* P
then *if* $x \in S$ *then* *STOP* *else* P *afterExt* e $[\![S]\!]$ Q
else *if* $e \in$ *ready-set* Q
then *if* $x \in S$ *then* *STOP* *else* P $[\![S]\!]$ Q *afterExt* e
else *STOP*)›
by (*simp add*: *AfterExt-def After-Sync split*: *event.split*)

lemma *Hiding-FD-Hiding-AfterExt-if-ready-inside*:
‹$e \in B \Longrightarrow (P \setminus B) \sqsubseteq_{FD} (P$ *afterExt* *ev* $e \setminus B)$›
and *AfterExt-Hiding-FD-Hiding-AfterExt-if-ready-notin*:
‹$e \notin B \Longrightarrow (P \setminus B)$ *afterExt* *ev* $e \sqsubseteq_{FD} (P$ *afterExt* *ev* $e \setminus B)$›
if *ready*: ‹*ev* $e \in$ *ready-set* P›
by (*simp add*: *AfterExt-def Hiding-FD-Hiding-After-if-ready-inside that*)
(*simp add*: *AfterExt-def After-Hiding-FD-Hiding-After-if-ready-notin that*)

lemmas *Hiding-F-Hiding-AfterExt-if-ready-inside* =
Hiding-FD-Hiding-AfterExt-if-ready-inside[*THEN leFD-imp-leF*]
and *AfterExt-Hiding-F-Hiding-AfterExt-if-ready-notin* =
AfterExt-Hiding-FD-Hiding-AfterExt-if-ready-notin[*THEN leFD-imp-leF*]

and *Hiding-D-Hiding-AfterExt-if-ready-inside* =
Hiding-FD-Hiding-AfterExt-if-ready-inside[*THEN leFD-imp-leD*]
and *AfterExt-Hiding-D-Hiding-AfterExt-if-ready-notin* =
AfterExt-Hiding-FD-Hiding-AfterExt-if-ready-notin[*THEN leFD-imp-leD*]
and *Hiding-T-Hiding-AfterExt-if-ready-inside* =
Hiding-FD-Hiding-AfterExt-if-ready-inside[*THEN leFD-imp-leF*, *THEN leF-imp-leT*]

and *AfterExt-Hiding-T-Hiding-AfterExt-if-ready-notin* =
AfterExt-Hiding-FD-Hiding-AfterExt-if-ready-notin[*THEN leFD-imp-leF*, *THEN leF-imp-leT*]

corollary *Hiding-DT-Hiding-AfterExt-if-ready-inside*:
‹⟦*ev e* ∈ *ready-set P*; *e* ∈ *B*⟧ ⟹ (*P* \ *B*) ⊑$_{DT}$ (*P afterExt ev e* \ *B*)›
and *AfterExt-Hiding-DT-Hiding-AfterExt-if-ready-notin*:
‹⟦*ev e* ∈ *ready-set P*; *e* ∉ *B*⟧ ⟹ (*P* \ *B*) *afterExt ev e* ⊑$_{DT}$ (*P afterExt ev e* \ *B*)›
by (*simp add*: *Hiding-D-Hiding-AfterExt-if-ready-inside*
Hiding-T-Hiding-AfterExt-if-ready-inside leD-leT-imp-leDT)
(*simp add*: *AfterExt-Hiding-D-Hiding-AfterExt-if-ready-notin*
AfterExt-Hiding-T-Hiding-AfterExt-if-ready-notin leD-leT-imp-leDT)

10.1.6 Behaviour of *AfterExt* with Operators of `HOL-CSPM`

lemma *AfterExt-MultiDet-is-AfterExt-MultiNdet*:
‹*finite A* ⟹ (□ *a* ∈ *A*. *P a*) *afterExt e* = (⨅ *a* ∈ *A*. *P a*) *afterExt e*›
by (*auto simp add*: *AfterExt-def After-MultiDet-is-After-MultiNdet*
MultiDet-is-BOT-iff MultiNdet-is-BOT-iff split: *event.split*)

lemma *AfterExt-GlobalNdet*:
‹(⊓ *a* ∈ *A*. *P a*) *afterExt e* =
(*if e* = ✓ ∨ *e* ∉ (⋃ *a* ∈ *A*. *ready-set* (*P a*)) *then STOP*
else (⊓ *a* ∈ {*a* ∈ *A*. *e* ∈ *ready-set* (*P a*)}. *P a*) *afterExt e*)›
and *AfterExt-MultiNdet*:
‹*finite A* ⟹ (⨅ *a* ∈ *A*. *P a*) *afterExt e* =
(*if e* = ✓ ∨ *e* ∉ (⋃ *a* ∈ *A*. *ready-set* (*P a*)) *then STOP*
else (⨅ *a* ∈ {*a* ∈ *A*. *e* ∈ *ready-set* (*P a*)}. *P a*) *afterExt e*)›
and *AfterExt-MultiDet*:
‹*finite A* ⟹ (□ *a* ∈ *A*. *P a*) *afterExt e* =
(*if e* = ✓ ∨ *e* ∉ (⋃ *a* ∈ *A*. *ready-set* (*P a*)) *then STOP*
else (⨅ *a* ∈ {*a* ∈ *A*. *e* ∈ *ready-set* (*P a*)}. *P a*) *afterExt e*)›
by (*simp-all add*: *AfterExt-def split*: *event.split*)
(*simp add*: *After-GlobalNdet*, *simp add*: *After-MultiNdet*, *simp add*: *After-MultiDet*)

10.1.7 Behaviour of *AfterExt* with Operators of `HOL-CSP_OpSem`

lemma *AfterExt-Sliding-is-AfterExt-Ndet*:
‹*P* ▷ *Q afterExt e* = *P* ⊓ *Q afterExt e*›
by (*simp add*: *AfterExt-def After-Sliding-is-After-Ndet Sliding-is-BOT-iff Ndet-is-BOT-iff*
split: *event.split*)

lemmas *AfterExt-Sliding* = *AfterExt-Ndet*[*folded AfterExt-Sliding-is-AfterExt-Ndet*]

lemma *AfterExt-Throw*:
‹(*P* ⊖ *a* ∈ *A*. *Q a*) *afterExt e* =
(*case e of* ✓ ⇒ *STOP*
| *ev x* ⇒ *if P* = ⊥ *then* ⊥
else if ev x ∈ *ready-set P then if x* ∈ *A then Q x*
else (*P after x*) ⊖ *a* ∈ *A*. *Q a else STOP*)›
by (*simp add*: *AfterExt-def After-Throw Throw-is-BOT-iff split*: *event.split*)

lemma *AfterExt-Interrupt*:
‹(*P* △ *Q*) *afterExt e* =
(*case e of* ✓ ⇒ *STOP*
| *ev x* ⇒ *if P* = ⊥ ∨ *Q* = ⊥ *then* ⊥
else if ev x ∈ *ready-set P* ∧ *ev x* ∈ *ready-set Q*
then (*Q after x*) ⊓ (*P after x* △ *Q*)
else if ev x ∈ *ready-set P* ∧ *ev x* ∉ *ready-set Q*
then P after x △ *Q*
else if ev x ∉ *ready-set P* ∧ *ev x* ∈ *ready-set Q*
then Q after x
else STOP)›
by (*simp add*: *AfterExt-def After-Interrupt Interrupt-is-BOT-iff*
Ndet-is-BOT-iff ready-set-BOT After-is-BOT-iff D-UU split: *event.split*)

10.1.8 Behaviour of *AfterExt* with Reference Processes

lemma *AfterExt-DF*:
‹*DF A afterExt e* =
(*case e of ev x* ⇒ *if x* ∈ *A then DF A else STOP* | ✓ ⇒ *STOP*)›
by (*cases e*) (*simp-all add*: *AfterExt-def After-DF BOT-iff-D div-free-DF*)

lemma *AfterExt-DF*$_{SKIP}$:
‹*DF*$_{SKIP}$ *A afterExt e* =
(*case e of ev x* ⇒ *if x* ∈ *A then DF*$_{SKIP}$ *A else STOP* | ✓ ⇒ *STOP*)›
by (*cases e*) (*simp-all add*: *AfterExt-def After-DF*$_{SKIP}$ *BOT-iff-D div-free-DF*$_{SKIP}$)

lemma *AfterExt-RUN*:
‹*RUN A afterExt e* =
(*case e of ev x* ⇒ *if x* ∈ *A then RUN A else STOP* | ✓ ⇒ *STOP*)›
by (*cases e*) (*simp-all add*: *AfterExt-def After-RUN BOT-iff-D div-free-RUN*)

lemma *AfterExt-CHAOS*:
‹*CHAOS A afterExt e* =
(*case e of ev x* ⇒ *if x* ∈ *A then CHAOS A else STOP* | ✓ ⇒ *STOP*)›
by (*cases e*) (*simp-all add*: *AfterExt-def After-CHAOS BOT-iff-D div-free-CHAOS*)

lemma *AfterExt-CHAOS$_{SKIP}$*:
‹*CHAOS$_{SKIP}$ A afterExt e* =
(*case e of ev x* ⇒ *if x* ∈ *A then CHAOS$_{SKIP}$ A else STOP* | ✓ ⇒ *STOP*)›
by (*cases e*) (*simp-all add*: *AfterExt-def After-CHAOS$_{SKIP}$ BOT-iff-D div-free-CHAOS$_{SKIP}$*)

lemma *DF-FD-AfterExt*:
‹*DF A* ⊑$_{FD}$ *P* ⟹ *e* ∈ *ready-set P* ⟹ *DF A* ⊑$_{FD}$ *P afterExt e*›
apply (*cases e*, *simp add*: *AfterExt-def DF-FD-After*)
by (*metis anti-mono-ready-set-FD event.distinct*(*1*) *image-iff ready-set-DF subsetD*)

lemma *DF$_{SKIP}$-FD-AfterExt*:
‹*DF$_{SKIP}$ A* ⊑$_{FD}$ *P* ⟹ *ev e* ∈ *ready-set P* ⟹ *DF$_{SKIP}$ A* ⊑$_{FD}$ *P afterExt ev e*›
by (*simp add*: *AfterExt-def DF$_{SKIP}$-FD-After*)

lemma *deadlock-free-AfterExt*:
‹*deadlock-free P* ⟹ *deadlock-free* (*P afterExt e*) ⟷
(*if e* ∈ *ready-set P* ∧ *e* ≠ ✓ *then True else False*)›
by (*cases e*)
(*simp add*: *AfterExt-def deadlock-free-After*,
simp add: *AfterExt-def BOT-iff-D deadlock-free-implies-div-free non-deadlock-free-STOP*)

lemma *deadlock-free$_{SKIP}$-AfterExt*:
‹*deadlock-free$_{SKIP}$ P* ⟹ *deadlock-free$_{SKIP}$* (*P afterExt e*) ⟷
(*if e* ∈ *ready-set P* ∧ *e* ≠ ✓ *then True else False*)›
by (*cases e*)
(*simp add*: *AfterExt-def deadlock-free$_{SKIP}$-After*,
simp add: *AfterExt-def BOT-iff-D deadlock-free$_{SKIP}$-implies-div-free non-deadlock-free$_{SKIP}$-STOP*)

end

10.2 The AfterTrace Operator, bis

theory *AfterTraceBis*
imports *AfterExtBis HOL−CSPM.DeadlockResults*
begin

10.2.1 Definition

We just defined *P afterExt e* for *P* an *'α process* and *e* of type *'α event*. Since traces of an *'α process* are *'α event list*, the following inductive definition is

natural.

fun *AfterTrace* :: ‹'α *process* ⇒ 'α *trace* ⇒ 'α *process*› (**infixl** ‹*afterTrace*› 77)
where ‹*P afterTrace* [] = *P*›
| ‹*P afterTrace* (*e* # *s*) = *P afterExt e afterTrace s*›

We can also induct backward.

lemma *AfterTrace-append*: ‹*P afterTrace* (*s* @ *t*) = *P afterTrace s afterTrace t*›
apply (*induct t rule*: *rev-induct*, *simp*)
apply (*induct s rule*: *rev-induct*, *simp*)
by (*metis AfterTrace.simps append.assoc append.right-neutral append-Cons append-Nil*)

lemma *AfterTrace-snoc* : ‹*P afterTrace* (*s* @ [*e*]) = *P afterTrace s afterExt e*›
by (*simp add*: *AfterTrace-append*)

We have some immediate properties.

lemma *AfterTrace-STOP* : ‹*STOP afterTrace s* = *STOP*›
by (*induct s*) (*simp-all add*: *AfterExt-STOP*)

lemma *AfterTrace-SKIP* : ‹*SKIP afterTrace s* = (*if s* = [] *then SKIP else STOP*)›
by (*induct s*) (*simp-all add*: *AfterExt-SKIP AfterTrace-STOP*)

lemma *AfterTrace-BOT* : ‹⊥ *afterTrace s* = (*if tickFree s then* ⊥ *else STOP*)›
by (*induct s*) (*simp-all add*: *AfterExt-BOT AfterTrace-STOP*)

10.2.2 Projections

lemma *F-AfterTrace* : ‹(*s* @ *t*, *X*) ∈ ℱ *P* ⟹ (*t*, *X*) ∈ ℱ (*P afterTrace s*)›
apply (*induct s arbitrary*: *t rule*: *rev-induct*, *simp*)
apply (*simp add*: *AfterTrace-snoc F-AfterExt*)
by (*metis Cons-in-T-imp-elem-ready-set F-T list.discI list.sel*(*1*, *3*))

lemma *D-AfterTrace* :
‹*tickFree* (*s* @ *t*) ⟹ *s* @ *t* ∈ 𝒟 *P* ⟹ *t* ∈ 𝒟 (*P afterTrace s*)›
apply (*induct s arbitrary*: *t rule*: *rev-induct*, *solves* ‹*simp*›)
by (*simp add*: *AfterTrace-snoc D-AfterExt*,
metis list.discI list.sel(*1*, *3*) *tickFree-Cons*)

lemma *T-AfterTrace* : ‹*s* @ *t* ∈ 𝒯 *P* ⟹ *t* ∈ 𝒯 (*P afterTrace s*)›
using *F-AfterTrace T-F-spec* **by** *blast*

corollary *ready-set-AfterTrace* :
‹*s* @ *e* # *t* ∈ 𝒯 *P* ⟹ *e* ∈ *ready-set* (*P afterTrace s*)›
by (*metis Cons-in-T-imp-elem-ready-set T-AfterTrace*)

corollary *F-imp-refusal-AfterTrace*:
‹(*s*, *X*) ∈ ℱ *P* ⟹ ([], *X*) ∈ ℱ (*P afterTrace s*)›
by (*simp add*: *F-AfterTrace*)

corollary *D-imp-AfterTrace-is-BOT*:
‹*tickFree s* ⟹ *s* ∈ $\mathcal{D}$ *P* ⟹ *P afterTrace s* = ⊥›
by (*simp add*: *BOT-iff-D D-AfterTrace*)

10.2.3 Monotony

lemma *mono-AfterTrace* : ‹*P* ⊑ *Q* ⟹ *P afterTrace s* ⊑ *Q afterTrace s*›
by (*induct s rule*: *rev-induct*) (*simp-all add*: *mono-AfterExt AfterTrace-snoc*)

lemma *mono-AfterTrace-T* :
‹*P* ⊑$_T$ *Q* ⟹ *s* ∈ $\mathcal{T}$ *Q* ⟹ *tickFree s* ⟹ *P afterTrace s* ⊑$_T$ *Q afterTrace s*›
by (*induct s rule*: *rev-induct*; *simp add*: *AfterTrace-snoc*)
(*rule mono-AfterExt-T*; *use is-processT3-ST* **in** *blast*)

lemma *mono-AfterTrace-F* :
‹*P* ⊑$_F$ *Q* ⟹ *s* ∈ $\mathcal{T}$ *Q* ⟹ *tickFree s* ⟹ *P afterTrace s* ⊑$_F$ *Q afterTrace s*›
by (*induct s rule*: *rev-induct*; *simp add*: *AfterTrace-snoc*)
(*metis event.exhaust is-processT3-ST mono-AfterExt-F ready-set-AfterTrace*)

lemma *mono-AfterTrace-D* : ‹*P* ⊑$_D$ *Q* ⟹ *P afterTrace s* ⊑$_D$ *Q afterTrace s*›
by (*induct s rule*: *rev-induct*) (*simp-all add*: *mono-AfterExt-D AfterTrace-snoc*)

lemma *mono-AfterTrace-FD* :
‹*P* ⊑$_{FD}$ *Q* ⟹ *s* ∈ $\mathcal{T}$ *Q* ⟹ *P afterTrace s* ⊑$_{FD}$ *Q afterTrace s*›
by (*induct s rule*: *rev-induct*; *simp add*: *AfterTrace-snoc*)
(*meson Cons-in-T-imp-elem-ready-set T-AfterTrace is-processT3-ST mono-AfterExt-FD*)

lemma *mono-AfterTrace-DT* :
‹*P* ⊑$_{DT}$ *Q* ⟹ *P afterTrace s* ⊑$_{DT}$ *Q afterTrace s*›
by (*induct s rule*: *rev-induct*; *simp add*: *AfterTrace-snoc mono-AfterExt-DT*)

10.2.4 Another Definition of *events-of*

inductive *reachable-event* :: ‹'α *process* ⇒ 'α ⇒ *bool*›
where ‹*ev e* ∈ *ready-set P* ⟹ *reachable-event P e*›
| ‹*reachable-event* (*P after f*) *e* ⟹ *reachable-event P e*›

lemma *events-of-iff-reachable-event*: ‹*e* ∈ *events-of P* ⟷ *reachable-event P e*›
proof (*intro iffI*)
show ‹*reachable-event P e* ⟹ *e* ∈ *events-of P*›
apply (*induct rule*: *reachable-event.induct*;
simp add: *T-After events-of-def ready-set-def split*: *if-split-asm*)
by (*meson list.set-intros*(*1*)) (*meson list.set-sel*(*2*))
next
assume ‹*e* ∈ *events-of P*›
then obtain *s* **where** ∗ : ‹*s* ∈ $\mathcal{T}$ *P*› ‹*ev e* ∈ *set s*› **unfolding** *events-of-def* **by** *blast*
thus ‹*reachable-event P e*›

```
  proof (induct s arbitrary: P)
    show ⟨⋀P e. ev e ∈ set [] ⟹ reachable-event P e⟩ by simp
  next
    case (Cons f s)
    from Cons.prems(1) is-processT3-ST
    have * : ⟨f ∈ ready-set P⟩ unfolding ready-set-def by force
    from Cons.prems(2) consider ⟨f = ev e⟩ | ⟨ev e ∈ set s⟩ by auto
    thus ⟨reachable-event P e⟩
    proof cases
      assume ⟨f = ev e⟩
      from *[simplified this]
      show ⟨reachable-event P e⟩ by (rule reachable-event.intros(1))
    next
      assume ⟨ev e ∈ set s⟩
      show ⟨reachable-event P e⟩
      proof (cases f)
        fix f'
        assume ⟨f = ev f'⟩
          with * Cons.prems(1) have ⟨s ∈ 𝒯 (P after f')⟩ by (force simp add:
T-After)
        show ⟨reachable-event P e⟩
          apply (rule reachable-event.intros(2))
          by (rule Cons.hyps[OF ⟨s ∈ 𝒯 (P after f')⟩ ⟨ev e ∈ set s⟩])
      next
        from Cons.prems(1) have ⟨f = ✓ ⟹ s = []⟩
            by simp (metis butlast.simps(2) front-tickFree-butlast is-processT2-TR
tickFree-Cons)
        thus ⟨f = ✓ ⟹ reachable-event P e⟩
          using ⟨ev e ∈ set s⟩ by force
      qed
    qed
  qed
qed


lemma reachable-event-BOT: ⟨reachable-event ⊥ e⟩
  by (simp add: reachable-event.intros(1) ready-set-BOT)

lemma not-reachable-event-STOP: ⟨¬ reachable-event STOP e⟩
  by (subst events-of-iff-reachable-event[symmetric])
     (unfold events-of-def, simp add: T-STOP)

lemma reachable-tick-SKIP: ⟨¬ reachable-event SKIP ✓⟩
  by (subst events-of-iff-reachable-event[symmetric])
     (unfold events-of-def, simp add: T-SKIP)



lemma reachable-event-iff-in-ready-set-AfterTrace:
```

```
  ‹reachable-event P e ⟷ e ∈ {e| e s. ev e ∈ ready-set (P afterTrace s)}›
proof −
  have ‹reachable-event P e ⟹ ∃ s. ev e ∈ ready-set (P afterTrace s)›
  proof (induct rule: reachable-event.induct)
    case (1 e P)
    thus ?case by (metis AfterTrace.simps(1))
  next
    case (2 P f e)
    from 2.hyps(2) obtain s where ‹ev e ∈ ready-set (P after f afterTrace s)› by
blast
    hence ‹ev e ∈ ready-set (P afterTrace (ev f # s))› by (simp add: AfterExt-def)
    thus ?case by blast
  qed
  also have ‹ev e ∈ ready-set (P afterTrace s) ⟹ reachable-event P e› for s
  proof (induct s arbitrary: P)
    show ‹ev e ∈ ready-set (P afterTrace []) ⟹ reachable-event P e› for P
      by (simp add: reachable-event.intros(1))
  next
    fix f s P
    assume  hyp : ‹ev e ∈ ready-set (P afterTrace s) ⟹ reachable-event P e› for
P
    assume prem : ‹ev e ∈ ready-set (P afterTrace (f # s))›
    show ‹reachable-event P e›
    proof (cases f)
      fix f'
      assume ‹f = ev f'›
      show ‹reachable-event P e› by (rule reachable-event.intros(2)[OF hyp])
                                    (use prem in ‹simp add: AfterExt-def ‹f = ev f'››)
    next
      case tick
      with prem show ‹f = ✓ ⟹ reachable-event P e›
      by (simp add: AfterExt-def reachable-event-BOT AfterTrace-STOP ready-set-STOP

              split: if-split-asm)
    qed
  qed
  ultimately show ?thesis by blast
qed
```

10.2.5 Characterizations for Deadlock Freeness

```
lemma deadlock-free-AfterExt-characterization:
  ‹deadlock-free P ⟷ range ev ∉ ℛ P ∧
                      (∀ e ∈ ready-set P. deadlock-free (P afterExt e))›
  (is ‹deadlock-free P ⟷ ?rhs›)
proof (intro iffI)
  have ‹range ev ∉ ℛ P ⟷ UNIV − {✓} ∉ ℛ P›
    by (metis Diff-insert-absorb UNIV-eq-I event.simps(3) event-set image-iff)
  thus ‹deadlock-free P ⟹ ?rhs›
```

```
    by (metis DF-FD-AfterExt Diff-Diff-Int Diff-partition Diff-subset F-T
          deadlock-free_SKIP-is-right deadlock-free-def
        deadlock-free-implies-non-terminating deadlock-free-is-deadlock-free_SKIP
      inf-top-left is-processT5-S2a non-tickFree-tick Refusals-iff self-append-conv2)
next
  assume assm : ‹?rhs›
  with BOT-iff-D process-charn have non-BOT : ‹P ≠ ⊥› by (metis Refusals-iff)
  show ‹deadlock-free P›
  proof (unfold deadlock-free-F failure-refine-def, safe)
    from assm have * : ‹range ev ∉ ℛ P›
      ‹e ∈ ready-set P ⟹
       {(tl s, X) |s X. (s, X) ∈ ℱ P ∧ s ≠ [] ∧ hd s = e} ⊆ ℱ (DF UNIV)› for e
      by (simp-all add: deadlock-free-F failure-refine-def F-AfterExt non-BOT)

    fix s X
    assume ** : ‹(s, X) ∈ ℱ P›
    show ‹(s, X) ∈ 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'›
      with ** Cons-in-T-imp-elem-ready-set F-T have ‹e ∈ ready-set P› by blast
      with *(2)[OF this] show ‹(s, X) ∈ ℱ (DF UNIV)›
        by (subst F-DF, simp add: subset-iff)
           (metis (no-types, lifting) *(1) ** *** CollectD Refusals-iff
                 event-set hd-append2 hd-in-set in-set-conv-decomp-first insert-iff
                 is-processT6-S2 list.sel(1) list.set-intros(1) rangeE ready-set-def)
    qed
  qed
qed


lemma deadlock-free_SKIP-AfterExt-characterization:
  ‹deadlock-free_SKIP P ⟷
   UNIV ∉ ℛ P ∧ (∀ e ∈ ready-set P − {✓}. deadlock-free_SKIP (P afterExt e))›
  (is ‹deadlock-free_SKIP P ⟷ ?rhs›)
proof (intro iffI)
  show ‹deadlock-free_SKIP P ⟹ ?rhs›
    by (metis Diff-iff Nil-elem-T deadlock-free_SKIP-AfterExt
          deadlock-free_SKIP-is-right insertI1 Refusals-iff tickFree-Nil)
next
  assume assm : ‹?rhs›
  with BOT-iff-D process-charn have non-BOT : ‹P ≠ ⊥› by (metis Refusals-iff)
  show ‹deadlock-free_SKIP P›
  proof (unfold deadlock-free_SKIP-def failure-refine-def, safe)
    from assm have * : ‹UNIV ∉ ℛ P›
```

```
    ‹e ∈ ready-set P ∧ e ≠ ✓ ⟹
     {(tl s, X) |s X. (s, X) ∈ ℱ P ∧ s ≠ [] ∧ hd s = e} ⊆ ℱ (DF_SKIP UNIV)›
for e
  by (simp-all add: deadlock-free_SKIP-def failure-refine-def F-AfterExt non-BOT)

  fix s X
  assume ** : ‹(s, X) ∈ ℱ P›
  show ‹(s, X) ∈ ℱ (DF_SKIP UNIV)›
  proof (cases s)
    show ‹s = [] ⟹ (s, X) ∈ ℱ (DF_SKIP UNIV)›
      by (subst F-DF_SKIP, simp)
         (metis *(1) ** UNIV-eq-I event.exhaust Refusals-iff)
  next
    fix e s'
    assume *** : ‹s = e # s'›
    show ‹(s, X) ∈ ℱ (DF_SKIP UNIV)›
    proof (cases e)
      fix e'
      assume ‹e = ev e'›
      with ** *** Cons-in-T-imp-elem-ready-set F-T
      have ‹e ∈ ready-set P ∧ e ≠ ✓› by blast
      with *(2)[OF this] show ‹(s, X) ∈ ℱ (DF_SKIP UNIV)›
        by (subst F-DF_SKIP, simp add: subset-iff)
           (metis ** *** ‹e = ev e'› list.distinct(1) list.sel(1))
    next
      assume ‹e = ✓›
      hence ‹s = [✓]›
        by (metis ** *** F-T append-butlast-last-id append-single-T-imp-tickFree
                  butlast.simps(2) list.distinct(1) tickFree-Cons)
      thus ‹(s, X) ∈ ℱ (DF_SKIP UNIV)›
        by (subst F-DF_SKIP, simp)
    qed
  qed
 qed
qed



end
```

Chapter 11

Generic Operational Semantics as a Locale, bis

theory *OpSemGenericBis*
imports *AfterTraceBis HOL−CSPM.DeadlockResults*
begin

11.1 Definition

locale *OpSemGeneric* =
fixes τ-*trans* :: ‹$'\alpha$ *process* ⇒ $'\alpha$ *process* ⇒ *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-ready-set*: ‹$P \rightsquigarrow_\tau Q \Longrightarrow$ *ready-set* $Q \subseteq$ *ready-set* P›
and τ-*trans-mono-AfterExt*:
‹$e \in$ *ready-set* $Q \Longrightarrow P \rightsquigarrow_\tau Q \Longrightarrow P$ *afterExt* $e \rightsquigarrow_\tau Q$ *afterExt* e›
begin

This locale needs to be instantiated with a binary relation ($\rightsquigarrow_\tau$) which:

- is compatible with ($\sqcap$)
- is transitive
- makes *ready-set* anti-monotonic
- makes *AfterExt* monotonic.

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

abbreviation *event-trans* :: ‹$'\alpha$ *process* ⇒ $'\alpha$ *event* ⇒ $'\alpha$ *process* ⇒ *bool*› (‹-/ ↝-/ -› [*50*, *3*, *51*] *50*)
where ‹$P \rightsquigarrow e\ Q \equiv e \in$ *ready-set* $P \wedge P$ *afterExt* $e \rightsquigarrow_\tau Q$›

From idempotence, commutativity and $\bot$ absorbance of $(\sqcap)$, we get the following for free.

lemma *τ-trans-eq*: ‹$P \rightsquigarrow_\tau P$›
and *τ-trans-NdetR*: ‹$P \sqcap Q \rightsquigarrow_\tau Q$›
and *BOT-τ-trans-anything*: ‹$\bot \rightsquigarrow_\tau P$›
and *BOT-ev-trans-anything*: ‹$\bot \rightsquigarrow (ev\ e)\ P$›
by (*metis Ndet-id τ-trans-NdetL*,
metis Ndet-commute τ-trans-NdetL,
metis Ndet-BOT τ-trans-NdetL,
metis AfterExt-BOT Ndet-BOT UNIV-I τ-trans-NdetL event.simps(*3*) *ready-set-BOT*)

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

lemma *event-trans-τ-trans*: ‹$P \rightsquigarrow e\ P' \Longrightarrow P' \rightsquigarrow_\tau P'' \Longrightarrow P \rightsquigarrow e\ P''$›
by (*meson τ-trans-transitivity*)

lemma *τ-trans-event-trans*: ‹$P \rightsquigarrow_\tau P' \Longrightarrow P' \rightsquigarrow e\ P'' \Longrightarrow P \rightsquigarrow e\ P''$›
using *τ-trans-mono-AfterExt τ-trans-transitivity τ-trans-anti-mono-ready-set* **by** *blast*

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

inductive *trace-trans* :: ‹$'\alpha$ *process* ⇒ $'\alpha$ *trace* ⇒ $'\alpha$ *process* ⇒ *bool*› (‹-/ $\rightsquigarrow^*$-/ -› [*50*, *3*, *51*] *50*)
where *trace-τ-trans* : ‹$P \rightsquigarrow_\tau P' \Longrightarrow P \rightsquigarrow^* [] P'$›
| *trace-tick-trans* : ‹$P \rightsquigarrow ✓\ P' \Longrightarrow P \rightsquigarrow^* [✓]\ P'$›
| *trace-Cons-ev-trans* :
‹$P \rightsquigarrow (ev\ e)\ P' \Longrightarrow P' \rightsquigarrow^* s\ P'' \Longrightarrow P \rightsquigarrow^* (ev\ e) \# s\ P''$›

lemma *trace-trans-τ-trans*: ‹$P \rightsquigarrow^* s\ P' \Longrightarrow P' \rightsquigarrow_\tau P'' \Longrightarrow P \rightsquigarrow^* s\ P''$›
apply (*induct rule*: *trace-trans.induct*)
subgoal using *τ-trans-transitivity trace-τ-trans* **by** *blast*
subgoal using *event-trans-τ-trans trace-tick-trans* **by** *blast*
using *trace-Cons-ev-trans* **by** *blast*

lemma *τ-trans-trace-trans*: ‹$P \rightsquigarrow_\tau P' \Longrightarrow P' \rightsquigarrow^* s\ P'' \Longrightarrow P \rightsquigarrow^* s\ P''$›
apply (*rotate-tac*, *induct rule*: *trace-trans.induct*)
subgoal using *τ-trans-transitivity trace-τ-trans* **by** *blast*
subgoal using *τ-trans-event-trans trace-tick-trans* **by** *blast*
using *τ-trans-event-trans trace-Cons-ev-trans* **by** *blast*

lemma *tickFree-if-trans-trans-not-STOP* :
‹$P \rightsquigarrow^* s\ Q \Longrightarrow Q \neq STOP \Longrightarrow tickFree\ s$›
by (*induct rule*: *trace-trans.induct*, *simp-all*)
(*metis τ-trans-anti-mono-ready-set bot.extremum-uniqueI*

ready-set-AfterExt ready-set-empty-iff-STOP)

lemma *BOT-trace-trans-tickFree-anything* : ‹*tickFree s* $\Longrightarrow$ $\bot \rightsquigarrow^* s$ *P*›
proof (*induct s arbitrary*: *P*)
show ‹$\bigwedge$*P*. *tickFree* [] $\Longrightarrow$ $\bot \rightsquigarrow^*$[] *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* $\Longrightarrow$ $\bot \rightsquigarrow^* s$ *P*› **for** *P*
have ∗ : ‹*tickFree s*› **using** *prem* **by** *auto*
obtain *e′* **where** ‹*e* = *ev e′*› **using** *event.exhaust prem* **by** *auto*
thus ‹$\bot \rightsquigarrow^* e$ # *s P*›
by *simp* (*rule trace-Cons-ev-trans*[*OF - hyp*];
simp add: ∗ *AfterExt-BOT BOT-τ-trans-anything ready-set-BOT*)
qed

11.2 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* $\rightsquigarrow$✓ *Q* $\Longrightarrow$ *X* $\in \mathcal{R}$ *Q* $\Longrightarrow$ ([✓], *X*) $\in \mathcal{F}$ *P*› **for** *P Q*
by (*metis append-Nil mem-Collect-eq ready-set-def tick-T-F*)
next
fix *P e Q s Q′*
assume ∗ : ‹*P* $\rightsquigarrow$(*ev e*) *Q*› ‹*X* $\in \mathcal{R}$ *Q′* $\Longrightarrow$ (*s*, *X*) $\in \mathcal{F}$ *Q*› ‹*X* $\in \mathcal{R}$ *Q′*›
have ‹*P afterExt ev e* $\sqsubseteq_F$ *Q*› **using** ∗(*1*) *that* **by** *blast*
hence ‹(*s*, *X*) $\in \mathcal{F}$ (*P afterExt ev e*)› **by** (*simp add*: *failure-refine-def subsetD* ∗(*2*, *3*))
thus ‹(*ev e* # *s*, *X*) $\in \mathcal{F}$ *P*› **by** (*simp add*: *F-AfterExt* ∗(*1*)) (*metis list.exhaust-sel*)
qed

lemma *trace-trans-imp-T-if-τ-trans-imp-leT*: ‹*P* $\rightsquigarrow^* s$ *Q* $\Longrightarrow$ *s* $\in \mathcal{T}$ *P*›
if ‹$\forall$ *P Q*. *P* $\rightsquigarrow_\tau$ *Q* $\longrightarrow$ *P* $\sqsubseteq_T$ *Q*›
proof (*induct rule*: *trace-trans.induct*)
show ‹*P* $\rightsquigarrow_\tau$ *Q* $\Longrightarrow$ [] $\in \mathcal{T}$ *P*› **for** *P Q*
by (*simp add*: *Nil-elem-T*)
next
show ‹*P* $\rightsquigarrow$✓ *Q* $\Longrightarrow$ [✓] $\in \mathcal{T}$ *P*› **for** *P Q*
by (*simp add*: *ready-set-def*)
next
fix *P e Q s Q′*
assume ∗ : ‹*P* $\rightsquigarrow$(*ev e*) *Q*› ‹*s* $\in \mathcal{T}$ *Q*›

have ‹*P afterExt ev e* $\sqsubseteq_T$ *Q*› **using** ∗(*1*) *that* **by** *blast*
hence ‹*s* ∈ $\mathcal{T}$ (*P afterExt ev e*)› **by** (*simp add*: ∗(*2*) *subsetD trace-refine-def*)
with ∗(*1*) *list.collapse* **show** ‹*ev e* # *s* ∈ $\mathcal{T}$ *P*›
by (*force simp add*: *T-AfterExt ready-set-def*)
qed

lemma *trace-trans-BOT-imp-D-if-τ-trans-imp-leD*: ‹*P* $\rightsquigarrow^*$*s* ⊥ ⟹ *s* ∈ $\mathcal{D}$ *P*›
if ‹∀ *P Q*. *P* $\rightsquigarrow_\tau$ *Q* ⟶ *P* $\sqsubseteq_D$ *Q*›
proof (*induct s arbitrary*: *P*)
show ‹⋀*P*. *P* $\rightsquigarrow^*$[] ⊥ ⟹ [] ∈ $\mathcal{D}$ *P*›
by (*subst* (*asm*) *trace-trans.simps*, *auto*)
(*meson BOT-iff-D divergence-refine-def subsetD that*)
next
fix *e s P*
assume *prem* : ‹*P* $\rightsquigarrow^*$*e* # *s* ⊥›
assume *hyp*: ‹*P* $\rightsquigarrow^*$*s* ⊥ ⟹ *s* ∈ $\mathcal{D}$ *P*› **for** *P*
have ‹*P afterExt e* $\rightsquigarrow^*$*s* ⊥›
using *prem* **by** (*cases rule*: *trace-trans.cases*)
(*auto simp add*: *trace-τ-trans intro*: *τ-trans-trace-trans*)
from *hyp*[*OF this*] **show** ‹*e* # *s* ∈ $\mathcal{D}$ *P*›
by (*auto simp add*: *D-AfterExt D-UU split*: *if-split-asm*)
qed

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

The following results require a lot of work, but will be very useful.

lemma *trace-trans-iff* :
‹*P* $\rightsquigarrow^*$ [] *Q* ⟷ *P* $\rightsquigarrow_\tau$ *Q*›
‹*P* $\rightsquigarrow^*$ [✓] *Q* ⟷ *P* $\rightsquigarrow$✓ *Q*›
‹*P* $\rightsquigarrow^*$ (*ev e*) # *s Q*′ ⟷ (∃ *Q*. *P* $\rightsquigarrow$(*ev e*) *Q* ∧ *Q* $\rightsquigarrow^*$ *s Q*′)›
‹*tickFree s* ⟹ (*P* $\rightsquigarrow^*$ *s* @ [*f*] *Q*′) ⟷ (∃ *Q*. *P* $\rightsquigarrow^*$*s Q* ∧ *Q* $\rightsquigarrow$*f Q*′)›
‹*front-tickFree* (*s* @ *t*) ⟹ (*P* $\rightsquigarrow^*$*s* @ *t Q*′) ⟷ (∃ *Q*. *P* $\rightsquigarrow^*$*s Q* ∧ *Q* $\rightsquigarrow^*$*t Q*′)›
proof −
show *f1* : ‹⋀*P Q*. *P* $\rightsquigarrow^*$ [] *Q* ⟷ *P* $\rightsquigarrow_\tau$ *Q*›
and *f2* : ‹⋀*P Q*. *P* $\rightsquigarrow^*$ [✓] *Q* ⟷ *P* $\rightsquigarrow$✓ *Q*›
and *f3* : ‹⋀*P Q*′ *e*. *P* $\rightsquigarrow^*$ (*ev e*) # *s Q*′ ⟷ (∃ *Q*. *P* $\rightsquigarrow$(*ev e*) *Q* ∧ *Q* $\rightsquigarrow^*$ *s Q*′)›
by ((*subst trace-trans.simps*, *auto*)[*1*])+

show *f4* : ‹*tickFree s* ⟹ (*P* $\rightsquigarrow^*$ *s* @ [*f*] *Q*′) ⟷ (∃ *Q*. *P* $\rightsquigarrow^*$*s Q* ∧ *Q* $\rightsquigarrow$*f Q*′)›
for *s f P Q*′
proof *safe*
show ‹*P* $\rightsquigarrow^*$ *s* @ [*f*] *Q*′ ⟹ ∃ *Q*. *P* $\rightsquigarrow^*$*s Q* ∧ *Q* $\rightsquigarrow$*f Q*′›
proof (*induct s arbitrary*: *P*)
case *Nil*
thus *?case*
apply (*subst* (*asm*) *trace-trans.simps*, *simp*)
using *τ-trans-eq τ-trans-transitivity f1* **by** *blast*

```
    next
      case (Cons e s)
      from Cons.prems have * : ‹e ∈ ready-set P ∧ P afterExt e ⇝* s @ [f] Q'›
        by (subst (asm) trace-trans.simps)
           (auto simp add:  intro: τ-trans-trace-trans)
      with Cons.hyps obtain Q where ** : ‹P afterExt e ⇝* s Q› ‹Q ⇝f 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 = ✓ ⟹ s = []› by (subst (asm) trace-trans.simps)
auto
        thus ‹e = ✓ ⟹ P ⇝* e # s Q› using * **(1) f1 f2 by blast
      qed
      with **(2) show ‹∃ Q. P ⇝* e # s Q ∧ Q ⇝f Q'› by blast
    qed
  next
    show ‹tickFree s ⟹ P ⇝* s Q ⟹ f ∈ ready-set Q ⟹ Q afterExt f ⇝τ Q'
⟹
          P ⇝* s @ [f] Q'› for Q
    proof (induct s arbitrary: P Q)
      show ‹tickFree [] ⟹ P ⇝* [] Q ⟹ f ∈ ready-set Q ⟹ Q afterExt f ⇝τ
Q' ⟹
            P ⇝* [] @ [f] Q'› for P Q
        by (simp add: f1)
           (metis (full-types) τ-trans-eq τ-trans-event-trans event.exhaust
                          trace-Cons-ev-trans trace-τ-trans trace-tick-trans)
    next
      case (Cons e s)
      from Cons.prems(2) have * : ‹e ∈ ready-set P ∧ P afterExt 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'›
          apply simp
          by (rule trace-Cons-ev-trans
                [OF - Cons.hyps[OF tickFree-tl[OF Cons.prems(1), simplified]
                                *[THEN conjunct2] Cons.prems(3)]])
             (use * τ-trans-eq Cons.prems(4) in ‹blast+›)
      next
        show ‹e = ✓ ⟹ P ⇝* (e # s) @ [f] Q'› using Cons.prems(1) by auto
```

qed
qed
qed

show ‹*front-tickFree* (*s* @ *t*) ⟹ *P* ⇝*s* @ *t Q'* ⟷ (∃ *Q*. *P* ⇝*s Q* ∧ *Q* ⇝*t Q'*)›
proof (*induct t arbitrary*: *Q'* *rule*: *rev-induct*)
show ‹*P* ⇝*s* @ [] *Q'* ⟷ (∃ *Q*. *P* ⇝*s Q* ∧ *Q* ⇝**[] *Q'*)› **for** *Q'*
by (*metis τ-trans-eq append.right-neutral trace-trans-τ-trans f1*)
next
case (*snoc e t*)
show ‹*P* ⇝*s* @ *t* @ [*e*] *Q'* ⟷ (∃ *Q*. *P* ⇝*s Q* ∧ *Q* ⇝*t* @ [*e*] *Q'*)›
proof (*intro iffI*)
assume *assm* : ‹*P* ⇝*s* @ *t* @ [*e*] *Q'*›
from *assm* **obtain** *Q* **where** ‹*P* ⇝*s* @ *t Q*› ‹*Q* ⇝*e Q'*›
by (*metis append.assoc f4 front-tickFree-implies-tickFree snoc.prems*)
also obtain *R* **where** ∗∗ : ‹*P* ⇝*s R*› ‹*R* ⇝*t Q*›
by (*metis calculation*(*1*) *append.assoc front-tickFree-dw-closed snoc.hyps snoc.prems*)
ultimately show ‹∃ *Q*. *P* ⇝*s Q* ∧ *Q* ⇝*t* @ [*e*] *Q'*›
by (*metis append-is-Nil-conv f4 front-tickFree-mono list.distinct*(*1*) *snoc.prems*)
next
assume ‹∃ *Q*. *P* ⇝*s Q* ∧ *Q* ⇝*t* @ [*e*] *Q'*›
then obtain *Q* **where** ‹*P* ⇝*s Q*› ‹*Q* ⇝*t* @ [*e*] *Q'*› **by** *blast*
also obtain *R* **where** ‹*Q* ⇝*t R*› ‹*R* ⇝*e Q'*›
by (*metis append-is-Nil-conv calculation*(*2*) *f4*
front-tickFree-mono list.distinct(*1*) *snoc.prems*)
ultimately show ‹*P* ⇝*s* @ *t* @ [*e*] *Q'*›
by (*metis append-assoc f4 front-tickFree-implies-tickFree snoc.hyps*
snoc.prems tickFree-implies-front-tickFree)
qed
qed
qed

11.4 Finally: *P* ⇝*s Q* is *P afterTrace s* ⇝*τ Q*

theorem *T-imp-trace-trans-iff-AfterTrace-τ-trans* :
‹*s* ∈ *T P* ⟹ (*P* ⇝*s Q*) ⟷ *P afterTrace s* ⇝*τ Q*›
proof (*intro iffI*)
show ‹*P* ⇝*s Q* ⟹ *s* ∈ *T P* ⟹ *P afterTrace s* ⇝*τ Q*›
proof (*induct s arbitrary*: *P Q rule*: *rev-induct*)
show ‹⋀*P Q*. *P* ⇝**[] *Q* ⟹ [] ∈ *T P* ⟹ *P afterTrace* [] ⇝*τ Q*›
using *trace-trans.cases* **by** *auto*
next
fix *s e P Q*
assume *hyp* : ‹*P* ⇝*s Q* ⟹ *s* ∈ *T P* ⟹ *P afterTrace s* ⇝*τ Q*› **for** *P Q*
assume *prems* : ‹*P* ⇝*s* @ [*e*] *Q*› ‹*s* @ [*e*] ∈ *T P*›
have ∗ : ‹*e* ∈ *ready-set* (*P afterTrace s*)›
using *prems*(*2*) *ready-set-AfterTrace* **by** *blast*

```
    show ‹P afterTrace (s @ [e]) ⇝τ Q›
      by (metis AfterTrace-snoc τ-trans-event-trans append-single-T-imp-tickFree
            hyp is-processT3-ST prems trace-trans-iff(4))
  qed
next
  show ‹P afterTrace s ⇝τ Q ⟹ s ∈ 𝒯 P ⟹ P ⇝*s Q›
  proof (induct arbitrary: P Q rule: AfterTrace.induct)
    show ‹⋀P Q. P afterTrace [] ⇝τ Q ⟹ [] ∈ 𝒯 P ⟹ P ⇝*[] Q›
      by (simp add: trace-τ-trans)
  next
    fix e and s :: ‹'α trace› and Q P
    assume  hyp : ‹P afterTrace s ⇝τ Q ⟹ s ∈ 𝒯 P ⟹ P ⇝*s Q› for P Q
    assume prems : ‹P afterTrace (e # s) ⇝τ Q› ‹e # s ∈ 𝒯 P›
    have ∗ : ‹e ∈ ready-set P ∧ s ∈ 𝒯 (P afterExt e)›
      by (metis AfterTrace.simps(1, 2) Cons-in-T-imp-elem-ready-set
            T-AfterTrace append-Cons append-Nil prems(2))
    show ‹P ⇝*e # s Q›
    proof (cases e)
      fix e'
      assume ∗∗ : ‹e = ev e'›
      from ∗ ∗∗ have ‹P ⇝(ev e') P afterExt (ev e')›
        by (simp add: τ-trans-eq)
      thus ‹P ⇝*e # s Q›
        by (subst ∗∗, rule trace-Cons-ev-trans[OF - hyp[OF prems(1)[simplified]
                                                 ∗[THEN conjunct2], simplified ∗∗]])
    next
      have ‹e = ✓ ⟹ s = []› by (metis butlast.simps(2) front-tickFree-butlast
                                   is-processT2-TR tickFree-Cons prems(2))
      thus ‹e = ✓ ⟹ P ⇝*e # s Q›
        using ∗ prems(1) trace-tick-trans by force
    qed
  qed
qed
```

As corollaries we obtain the reciprocal results of

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

$[\![\forall P\ Q.\ P \rightsquigarrow_\tau Q \longrightarrow P \sqsubseteq_T Q;\ ?P \rightsquigarrow^* ?s\ ?Q]\!] \Longrightarrow ?s \in \mathcal{T}\ ?P$

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

```
lemma F-imp-exists-trace-trans: ‹(s, X) ∈ 𝓕 P ⟹ ∃Q. (P ⇝*s Q) ∧ X ∈ ℛ Q›
  by (meson F-T F-imp-refusal-AfterTrace Refusals-iff
        T-imp-trace-trans-iff-AfterTrace-τ-trans τ-trans-eq)

lemma T-imp-exists-trace-trans: ‹s ∈ 𝒯 P ⟹ ∃Q. P ⇝*s Q›
  using F-imp-exists-trace-trans T-F by blast

lemma D-imp-trace-trans-BOT: ‹tickFree s ⟹ s ∈ 𝒟 P ⟹ P ⇝*s ⊥›
  by (simp add: D-T D-imp-AfterTrace-is-BOT
```

T-imp-trace-trans-iff-AfterTrace-τ-trans τ-trans-eq)

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

lemma *τ-trans-imp-leT-imp-STOP-trace-trans-iff*:
‹∀ *P Q*. $P \rightsquigarrow_\tau Q \longrightarrow P \sqsubseteq_T Q \Longrightarrow STOP \rightsquigarrow^* s\ P \longleftrightarrow s = [] \wedge P = STOP$›
using *STOP-T-iff* **by** (*subst trace-trans.simps*)
(*auto simp add*: *ready-set-STOP τ-trans-eq*)

lemma *τ-trans-imp-leF-imp-SKIP-trace-trans-iff*:
‹∀ *P Q*. $P \rightsquigarrow_\tau Q \longrightarrow P \sqsubseteq_F Q \Longrightarrow$
SKIP $\rightsquigarrow^* s\ P \longleftrightarrow s = [] \wedge P = SKIP \vee s = [✓] \wedge P = STOP$›
using *SKIP-F-iff STOP-F-iff* **by** (*subst trace-trans.simps*)
(*auto simp add*: *AfterExt-SKIP ready-set-SKIP τ-trans-eq*)

lemma *τ-trans-imp-leT-imp-trace-trans-ready-set-subset-ready-set-AfterTrace*:
‹∀ *P Q*. $P \rightsquigarrow_\tau Q \longrightarrow P \sqsubseteq_T Q \Longrightarrow P \rightsquigarrow^* s\ Q \Longrightarrow$
ready-set Q ⊆ *ready-set* (*P afterTrace s*)›
using *T-imp-trace-trans-iff-AfterTrace-τ-trans*
anti-mono-ready-set-T trace-trans-imp-T-if-τ-trans-imp-leT **by** *blast*

lemma *τ-trans-imp-leT-imp-trace-trans-imp-ready-set*:
‹∀ *P Q*. $P \rightsquigarrow_\tau Q \longrightarrow P \sqsubseteq_T Q \Longrightarrow P \rightsquigarrow^* (s\ @\ e\ \#\ t)\ Q \Longrightarrow$
e ∈ *ready-set* (*P afterTrace s*)›
using *ready-set-AfterTrace trace-trans-imp-T-if-τ-trans-imp-leT* **by** *blast*

lemma *trace-trans-iff-T-and-AfterTrace-τ-trans-if-τ-trans-imp-leT*:
‹∀ *P Q*. $P \rightsquigarrow_\tau Q \longrightarrow P \sqsubseteq_T Q \Longrightarrow$
$(P \rightsquigarrow^* s\ Q) \longleftrightarrow s \in \mathcal{T}\ P \wedge P\ afterTrace\ s \rightsquigarrow_\tau Q$›
using *T-imp-trace-trans-iff-AfterTrace-τ-trans trace-trans-imp-T-if-τ-trans-imp-leT*
by *blast*

11.5 General Rules of Operational Semantics

Some rules of operational semantics are consequences of *OpSemGeneric*'s axioms without needing to specify more what $P \rightsquigarrow_\tau Q$ is.

lemma *SKIP-trans-tick*: ‹*SKIP* $\rightsquigarrow$✓ *STOP*›
by (*simp add*: *AfterExt-SKIP τ-trans-eq ready-set-SKIP*)

lemma *tick-trans-imp-BOT-L-or-STOP-R*: ‹$P \rightsquigarrow$✓ $Q \Longrightarrow P = \bot \vee Q = STOP$›
by (*metis τ-trans-anti-mono-ready-set ready-set-AfterExt ready-set-empty-iff-STOP subset-empty*)

lemma *STOP-trace-trans-iff* : ‹*STOP* $\rightsquigarrow^* s\ P \longleftrightarrow s = [] \wedge P = STOP$›
by (*metis AfterExt-SKIP SKIP-neq-BOT SKIP-trans-tick empty-iff trace-trans.cases*

ready-set-STOP tick-trans-imp-BOT-L-or-STOP-R trace-trans-iff(1))

lemma *ready-tick-imp-τ-trans-SKIP*: ‹$P \rightsquigarrow_\tau SKIP$› **if** ‹$✓ \in$ *ready-set* P›
proof −
  from *that* **have** ‹$P \sqsubseteq_{FD} SKIP$›
    unfolding *failure-divergence-refine-def le-ref-def*
    by (*auto simp add*: *F-SKIP D-SKIP subset-iff ready-set-def is-processT6-S2*)
      (*metis append-Nil tick-T-F*)
  then obtain Q **where** ‹$P = Q \sqcap SKIP$›
    by (*metis mono-Ndet-FD mono-Ndet-FD-left FD-antisym Ndet-id idem-FD*)
  thus ‹$P \rightsquigarrow_\tau SKIP$› **by** (*simp add*: *τ-trans-NdetR*)
qed

lemma *exists-tick-trans-is-ready-tick*: ‹$(\exists\, P'.\ P \rightsquigarrow ✓\ P') \longleftrightarrow ✓ \in$ *ready-set* P›
  using *τ-trans-eq* **by** *blast*

lemma *tick-trans-iff* : ‹$P \rightsquigarrow ✓\ P' \longleftrightarrow P \rightsquigarrow_\tau SKIP \wedge P' = STOP$›
  by (*metis AfterExt-BOT AfterExt-SKIP SKIP-neq-BOT SKIP-trans-tick*
      τ-trans-event-trans ready-tick-imp-τ-trans-SKIP tick-trans-imp-BOT-L-or-STOP-R)

lemma *SKIP-cant-ev-trans*: ‹$\neg$ *SKIP* $\rightsquigarrow$(*ev e*) *STOP*›
  by (*simp add*: *ready-set-SKIP*)

lemma *STOP-cant-event-trans*: ‹$\neg$ *STOP* $\rightsquigarrow e$ *P*›
  by (*simp add*: *ready-set-STOP*)

lemma *ev-trans-Mprefix*: ‹$e \in A \Longrightarrow \Box a \in A \to P\ a \rightsquigarrow$(*ev e*) (*P e*)›
  by (*simp add*: *AfterExt-def After-Mprefix τ-trans-eq ready-set-Mprefix*)

lemma *ev-trans-Mndetprefix*: ‹$e \in A \Longrightarrow \sqcap a \in A \to P\ a \rightsquigarrow$(*ev e*) (*P e*)›
  by (*simp add*: *AfterExt-def After-Mndetprefix τ-trans-eq ready-set-Mndetprefix*)

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

lemma *τ-trans-MultiNdet*: ‹*finite* $A \Longrightarrow x \in A \Longrightarrow (\bigsqcap a \in A.\ P\ a) \rightsquigarrow_\tau P\ x$›
  by (*metis MultiNdet-insert′ τ-trans-NdetL emptyE insert-absorb*)

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

lemma *fix-point-*τ-*trans*: ‹*cont* $f \Longrightarrow P = (\mu\ X.\ f\ X) \Longrightarrow P \rightsquigarrow_\tau f\ P$›
by (*metis* τ-*trans-eq cont-process-rec*)

lemma *event-trans-DetL*: ‹$P \rightsquigarrow e\ P' \Longrightarrow P \Box Q \rightsquigarrow e\ P'$›
by (*metis AfterExt-Det-is-AfterExt-Ndet Un-iff* τ-*trans-NdetL* τ-*trans-event-trans ready-set-Det*)

lemma *event-trans-DetR*: ‹$Q \rightsquigarrow e\ Q' \Longrightarrow P \Box Q \rightsquigarrow e\ Q'$›
by (*metis Det-commute event-trans-DetL*)

lemma *event-trans-MultiDet*:
‹*finite* $A \Longrightarrow a \in A \Longrightarrow P\ a \rightsquigarrow e\ Q \Longrightarrow (\Box a \in A.\ P\ a) \rightsquigarrow e\ Q$›
by (*metis MultiDet-insert*$'$ *event-trans-DetL insert-absorb*)

lemma *Sliding-event-transL*: ‹$P \rightsquigarrow e\ P' \Longrightarrow P \rhd Q \rightsquigarrow e\ P'$›
unfolding *Sliding-def*
apply (*drule event-trans-DetL*[*of e P P*$'$ *Q*])
using τ-*trans-NdetL* τ-*trans-event-trans* **by** *blast*

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

lemma ‹$\exists P\ P'\ Q.\ P \rightsquigarrow ✓\ P' \wedge \neg P\ ;\ Q \rightsquigarrow ✓\ P'\ ;\ Q$›
by (*metis SKIP-Seq SKIP-trans-tick STOP-cant-event-trans*)

lemma *ev-trans-SeqR*:
‹$✓ \in$ *ready-set* $P \Longrightarrow Q \rightsquigarrow (ev\ e)\ Q' \Longrightarrow P\ ;\ Q \rightsquigarrow (ev\ e)\ Q'$›
by (*simp add*: *AfterExt-Seq ready-set-Seq AfterExt-BOT BOT-Seq*)
(*metis Ndet-commute* τ-*trans-NdetL* τ-*trans-transitivity*)

lemma ‹*SKIP* ⟦*S*⟧ *SKIP* ⇝✓ *STOP*›
by (*simp add*: *SKIP-trans-tick Sync-SKIP-SKIP*)

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

lemma *tick-trans-Hiding*: ‹*P* \ *B* ⇝✓ *STOP*› **if** ‹*P* ⇝✓ *P'*›
proof −
have ‹(*P* \ *B*) *afterExt* ✓ = *STOP* ⊓ (*P* \ *B*) *afterExt* ✓›
by (*simp add*: *AfterExt-def*)
thus ‹*P* \ *B* ⇝✓ *STOP*›
by (*simp add*: *AfterExt-def τ-trans-eq ready-tick-imp-ready-tick-Hiding that*)
qed

The following lemma is useless since the locale mechanism forces f to be of type $'\alpha \Rightarrow '\alpha$ while it could be $'\alpha \Rightarrow '\beta$. We will have to prove it again on each instantiation.

lemma ‹*Renaming P f* ⇝✓ *STOP*› **if** ‹*P* ⇝✓ *P'*›
proof −
have ‹*Renaming P f afterExt* ✓ = *STOP* ⊓ *Renaming P f afterExt* ✓›
by (*metis τ-trans-eq not-ready-AfterExt tick-trans-iff*)
thus ‹*Renaming P f* ⇝✓ *STOP*›
by (*simp add*: *that tick-eq-EvExt AfterExt-def τ-trans-eq*
ready-set-Renaming BOT-τ-trans-anything)
qed

end

end

Chapter 12

Failure Divergence Operational Semantics, bis

theory *OpSemFDBis*
imports *OpSemGenericBis HOL−Library.LaTeXsugar*
begin

As announced in the motivations, the first definition we want to try is with $(\sqsubseteq_{FD})$.

abbreviation τ-*trans* :: $\langle 'a\ process \Rightarrow 'a\ process \Rightarrow bool\rangle$ (**infixl** $\langle {}_{FD}\rightsquigarrow_{\tau}\rangle$ *50*)
where $\langle P\ {}_{FD}\rightsquigarrow_{\tau}\ Q \equiv P \sqsubseteq_{FD} Q\rangle$

We now instantiate the locale of *HOL−CSP-OpSem.OpSemGenericBis*.

interpretation *OpSemGeneric* $\langle({}_{FD}\rightsquigarrow_{\tau})\rangle$
using *trans-FD* **by** *unfold-locales*
(*auto simp add*: *anti-mono-ready-set-FD mono-AfterExt-FD*)

notation *event-trans* ($\langle$-/ ${}_{FD}\rightsquigarrow$-/ -$\rangle$ [*50*, *3*, *51*] *50*)
notation *trace-trans* ($\langle$-/ ${}_{FD}\rightsquigarrow^{*}$-/ -$\rangle$ [*50*, *3*, *51*] *50*)

lemma $\langle P\ {}_{FD}\rightsquigarrow\ e\ P' \Longrightarrow P'\ {}_{FD}\rightsquigarrow_{\tau}\ P'' \Longrightarrow P\ {}_{FD}\rightsquigarrow\ e\ P''\rangle$
$\langle P\ {}_{FD}\rightsquigarrow_{\tau}\ P' \Longrightarrow P'\ {}_{FD}\rightsquigarrow\ e\ P'' \Longrightarrow P\ {}_{FD}\rightsquigarrow\ e\ P''\rangle$
by (*fact event-trans-τ-trans τ-trans-event-trans*)+

12.1 Operational Semantics Laws

SKIP law

lemma $\langle SKIP\ {}_{FD}\rightsquigarrow\checkmark\ STOP\rangle$
by (*fact SKIP-trans-tick*)

$e \rightarrow P$ laws

lemma $\langle e \in A \Longrightarrow \Box a \in A \rightarrow P\ a\ {}_{FD}\rightsquigarrow(ev\ e)\ (P\ e)\rangle$

by (*fact ev-trans-Mprefix*)

lemma ‹$e \in A \Longrightarrow \sqcap a \in A \rightarrow P\ a\ {}_{FD}\!\!\rightsquigarrow (ev\ e)\ (P\ e)$›
by (*fact ev-trans-Mndetprefix*)

lemma ‹$e \rightarrow P\ {}_{FD}\!\!\rightsquigarrow\ (ev\ e)\ P$›
by (*fact ev-trans-prefix*)

$P \sqcap Q$ laws

lemma ‹$P \sqcap Q\ {}_{FD}\!\!\rightsquigarrow_{\tau}\ P$›
and ‹$P \sqcap Q\ {}_{FD}\!\!\rightsquigarrow_{\tau}\ Q$›
by (*fact* τ*-trans-NdetL* τ*-trans-NdetR*)+

lemma ‹$a \in A \Longrightarrow (\sqcap a \in A.\ P\ a)\ {}_{FD}\!\!\rightsquigarrow_{\tau}\ P\ a$›
by (*fact* τ*-trans-GlobalNdet*)

lemma ‹*finite* $A \Longrightarrow a \in A \Longrightarrow (\bigsqcap a \in A.\ P\ a)\ {}_{FD}\!\!\rightsquigarrow_{\tau}\ P\ a$›
by (*fact* τ*-trans-MultiNdet*)

$\mu\ x.\ f\ x$ law

lemma ‹*cont* $f \Longrightarrow P = (\mu\ X.\ f\ X) \Longrightarrow P\ {}_{FD}\!\!\rightsquigarrow_{\tau}\ f\ P$›
by (*fact fix-point-*τ*-trans*)

$P \Box Q$ laws

lemma τ*-trans-DetL*: ‹$P\ {}_{FD}\!\!\rightsquigarrow_{\tau}\ P' \Longrightarrow P \Box Q\ {}_{FD}\!\!\rightsquigarrow_{\tau}\ P' \Box Q$ ›
and τ*-trans-DetR*: ‹$Q\ {}_{FD}\!\!\rightsquigarrow_{\tau}\ Q' \Longrightarrow P \Box Q\ {}_{FD}\!\!\rightsquigarrow_{\tau}\ P \Box Q'$›
by *simp-all*

lemma τ*-trans-MultiDet*:
‹*finite* $A \Longrightarrow \forall\, a \in A.\ P\ a\ {}_{FD}\!\!\rightsquigarrow_{\tau}\ P'\ a \Longrightarrow$
$(\Box a \in A.\ P\ a)\ {}_{FD}\!\!\rightsquigarrow_{\tau}\ (\Box a \in A.\ P'\ a)$›
by (*fact mono-MultiDet-FD*)

lemma ‹$P\ {}_{FD}\!\!\rightsquigarrow e\ P' \Longrightarrow P \Box Q\ {}_{FD}\!\!\rightsquigarrow e\ P'$›
and ‹$Q\ {}_{FD}\!\!\rightsquigarrow e\ Q' \Longrightarrow P \Box Q\ {}_{FD}\!\!\rightsquigarrow e\ Q'$›
by (*fact event-trans-DetL event-trans-DetR*)+

lemma ‹*finite* $A \Longrightarrow a \in A \Longrightarrow P\ a\ {}_{FD}\!\!\rightsquigarrow e\ Q \Longrightarrow (\Box a \in A.\ P\ a)\ {}_{FD}\!\!\rightsquigarrow e\ Q$›
by (*fact event-trans-MultiDet*)

$P\ ;\ Q$ laws

lemma τ*-trans-SeqL*: ‹$P\ {}_{FD}\!\!\rightsquigarrow_{\tau}\ P' \Longrightarrow P\ ;\ Q\ {}_{FD}\!\!\rightsquigarrow_{\tau}\ P'\ ;\ Q$›
by *simp*

lemma *ev-trans-SeqL*: ‹$P\ {}_{FD}\!\!\rightsquigarrow (ev\ e)\ P' \Longrightarrow P\ ;\ Q\ {}_{FD}\!\!\rightsquigarrow (ev\ e)\ P'\ ;\ Q$›

by (*auto simp add*: *ready-set-Seq AfterExt-Seq*)

lemma *τ-trans-SeqR*: ‹∃ *P′*. *P* $_{FD}\!\rightsquigarrow\!\checkmark$ *P′* ⟹ *P* ; *Q* $_{FD}\!\rightsquigarrow_{\tau}$ *Q*›
by (*metis mono-Seq-FD SKIP-Seq τ-trans-eq ready-tick-imp-τ-trans-SKIP*)

lemma ‹✓ ∈ *ready-set P* ⟹ *Q* $_{FD}\!\rightsquigarrow$(*ev e*) *Q′* ⟹ *P* ; *Q* $_{FD}\!\rightsquigarrow$(*ev e*) *Q′*›

by (*fact ev-trans-SeqR*)

P \ *B* laws

lemma *τ-trans-Hiding*: ‹*P* $_{FD}\!\rightsquigarrow_{\tau}$ *P′* ⟹ *P* \ *B* $_{FD}\!\rightsquigarrow_{\tau}$ *P′* \ *B*›
by (*fact mono-Hiding-FD*)

lemma *ev-trans-Hiding-notin*:
‹*P* $_{FD}\!\rightsquigarrow$(*ev e*) *P′* ⟹ *e* ∉ *B* ⟹ *P* \ *B* $_{FD}\!\rightsquigarrow$(*ev e*) *P′* \ *B*›
by (*metis AfterExt-def After-Hiding-FD-Hiding-After-if-ready-notin mono-Hiding-FD*

event-trans-τ-trans event.simps(*4*) *ready-notin-imp-ready-Hiding*)

lemma ‹*P* $_{FD}\!\rightsquigarrow\!\checkmark$ *P′* ⟹ *P* \ *B* $_{FD}\!\rightsquigarrow\!\checkmark$ *STOP*›
by (*fact tick-trans-Hiding*)

lemma *ev-trans-Hiding-inside*:
‹*P* $_{FD}\!\rightsquigarrow$(*ev e*) *P′* ⟹ *e* ∈ *B* ⟹ *P* \ *B* $_{FD}\!\rightsquigarrow_{\tau}$ *P′* \ *B*›
by (*metis AfterExt-def Hiding-FD-Hiding-After-if-ready-inside*
mono-Hiding-FD event.simps(*4*) *trans-FD*)

Renaming P f laws

lemma *τ-trans-Renaming*: ‹*P* $_{FD}\!\rightsquigarrow_{\tau}$ *P′* ⟹ *Renaming P f* $_{FD}\!\rightsquigarrow_{\tau}$ *Renaming P′ f*›
by (*fact mono-Renaming-FD*)

lemma *tick-trans-Renaming*: ‹*P* $_{FD}\!\rightsquigarrow\!\checkmark$ *P′* ⟹ *Renaming P f* $_{FD}\!\rightsquigarrow\!\checkmark$ *STOP*›
by (*simp add*: *AfterExt-def ready-set-Renaming tick-eq-EvExt*)

lemma *ev-trans-Renaming*:
‹*f a* = *b* ⟹ *P* $_{FD}\!\rightsquigarrow$(*ev a*) *P′* ⟹ *Renaming P f* $_{FD}\!\rightsquigarrow$(*ev b*) (*Renaming P′ f*)›
apply (*simp add*: *AfterExt-Renaming Renaming-BOT ready-set-BOT ready-set-Renaming*)
apply (*intro conjI impI*)
apply (*meson ev-elem-anteced1 imageI vimageI2*)
apply (*rule τ-trans-transitivity*[*of* - ‹*Renaming* (*P afterExt ev a*) *f*›])
apply (*solves* ‹*rule τ-trans-GlobalNdet*, *simp*›)
by (*simp add*: *τ-trans-Renaming*)

lemma ‹*P* $_{FD}\!\rightsquigarrow_{\tau}$ *P′* ⟹ *P* ⟦*a* := *b*⟧ $_{FD}\!\rightsquigarrow_{\tau}$ *P′* ⟦*a* := *b*⟧›
by (*fact τ-trans-Renaming*)

lemma $\langle P \ {}_{FD}\!\rightsquigarrow\!\checkmark \ P' \Longrightarrow P \ [\![a := b]\!] \ {}_{FD}\!\rightsquigarrow\!\checkmark \ STOP\rangle$
by (*fact tick-trans-Renaming*)

lemma *ev-trans-RenamingF*:
$\langle P \ {}_{FD}\!\rightsquigarrow\!(ev\ a) \ P' \Longrightarrow P \ [\![a := b]\!] \ {}_{FD}\!\rightsquigarrow\!(ev\ b) \ P' \ [\![a := b]\!]\rangle$
by (*metis ev-trans-Renaming fun-upd-same*)

$P \ [\![S]\!] \ Q$ laws

lemma τ*-trans-SyncL*: $\langle P \ {}_{FD}\!\rightsquigarrow_\tau \ P' \Longrightarrow P \ [\![S]\!] \ Q \ {}_{FD}\!\rightsquigarrow_\tau \ P' \ [\![S]\!] \ Q\rangle$
and τ*-trans-SyncR*: $\langle Q \ {}_{FD}\!\rightsquigarrow_\tau \ Q' \Longrightarrow P \ [\![S]\!] \ Q \ {}_{FD}\!\rightsquigarrow_\tau \ P \ [\![S]\!] \ Q'\rangle$
by *simp-all*

lemma *ev-trans-SyncL*:
$\langle e \notin S \Longrightarrow P \ {}_{FD}\!\rightsquigarrow\!(ev\ e) \ P' \Longrightarrow P \ [\![S]\!] \ Q \ {}_{FD}\!\rightsquigarrow\!(ev\ e) \ P' \ [\![S]\!] \ Q \ \rangle$
and *ev-trans-SyncR*:
$\langle e \notin S \Longrightarrow Q \ {}_{FD}\!\rightsquigarrow\!(ev\ e) \ Q' \Longrightarrow P \ [\![S]\!] \ Q \ {}_{FD}\!\rightsquigarrow\!(ev\ e) \ P \ \ [\![S]\!] \ Q'\rangle$
by (*simp-all add*: *AfterExt-Sync ready-set-Sync image-iff*)

lemma *ev-trans-SyncLR*:
$\langle [\![e \in S;\ P \ {}_{FD}\!\rightsquigarrow\!(ev\ e) \ P';\ Q \ {}_{FD}\!\rightsquigarrow\!(ev\ e) \ Q']\!] \Longrightarrow$
$P \ [\![S]\!] \ Q \ {}_{FD}\!\rightsquigarrow\!(ev\ e) \ P' \ [\![S]\!] \ Q'\rangle$
by (*simp add*: *AfterExt-Sync ready-set-Sync*)

From here we slightly defer from Roscoe's laws for *Sync*: we obtain the following rules for *SKIP* instead of *STOP*.

lemma *tick-trans-SyncL*: $\langle P \ {}_{FD}\!\rightsquigarrow\!\checkmark \ P' \Longrightarrow P \ [\![S]\!] \ Q \ {}_{FD}\!\rightsquigarrow_\tau \ SKIP \ [\![S]\!] \ Q\rangle$
and *tick-trans-SyncR*: $\langle Q \ {}_{FD}\!\rightsquigarrow\!\checkmark \ Q' \Longrightarrow P \ [\![S]\!] \ Q \ {}_{FD}\!\rightsquigarrow_\tau \ P \ [\![S]\!] \ SKIP\rangle$
by (*simp-all add*: *ready-tick-imp-*τ*-trans-SKIP*)

lemma *tick-trans-SKIP-Sync-SKIP*: $\langle SKIP \ [\![S]\!] \ SKIP \ {}_{FD}\!\rightsquigarrow\!\checkmark \ STOP\rangle$
by (*simp add*: *SKIP-trans-tick Sync-SKIP-SKIP*)

lemma τ*-trans-SKIP-Sync-SKIP*: $\langle SKIP \ [\![S]\!] \ SKIP \ {}_{FD}\!\rightsquigarrow_\tau \ SKIP\rangle$
by (*simp add*: *Sync-SKIP-SKIP*)

$P \rhd Q$ laws

lemma *Sliding-*τ*-trans-left*: $\langle P \ {}_{FD}\!\rightsquigarrow_\tau \ P' \Longrightarrow P \rhd Q \ {}_{FD}\!\rightsquigarrow_\tau \ P' \rhd Q\rangle$
unfolding *Sliding-def* **by** *simp*

lemma $\langle P \ {}_{FD}\!\rightsquigarrow e \ P' \Longrightarrow P \rhd Q \ {}_{FD}\!\rightsquigarrow e \ P'\rangle$
by (*fact Sliding-event-transL*)

lemma $\langle P \rhd Q \ {}_{FD}\!\rightsquigarrow_\tau \ Q\rangle$
by (*fact Sliding-*τ*-transR*)

$P \triangle Q$ laws

lemma *Interrupt-*τ*-trans-left*: $\langle P \ {}_{FD}\!\rightsquigarrow_\tau \ P' \Longrightarrow P \triangle Q \ {}_{FD}\!\rightsquigarrow_\tau \ P' \triangle Q\rangle$

by (*simp add*: *mono-Interrupt-FD*)

lemma *Interrupt-τ-trans-right*: ‹$Q\ {}_{FD}\!\!\rightsquigarrow_{\tau}\ Q' \Longrightarrow P \triangle Q\ {}_{FD}\!\!\rightsquigarrow_{\tau}\ P \triangle Q'$›
by (*simp add*: *mono-Interrupt-FD*)

lemma *Interrupt-ev-trans-left*:
‹$P\ {}_{FD}\!\!\rightsquigarrow(ev\ e)\ P' \Longrightarrow P \triangle Q\ {}_{FD}\!\!\rightsquigarrow(ev\ e)\ P' \triangle Q$›
by (*simp add*: *AfterExt-def After-Interrupt Interrupt-τ-trans-left ready-set-Interrupt*)

lemma *Interrupt-ev-trans-right*: ‹$Q\ {}_{FD}\!\!\rightsquigarrow(ev\ e)\ Q' \Longrightarrow P \triangle Q\ {}_{FD}\!\!\rightsquigarrow(ev\ e)\ Q'$›
by (*simp add*: *AfterExt-def After-Interrupt ready-set-Interrupt*)

Throw P A Q laws

lemma *Throw-τ-trans-left*:
‹$P\ {}_{FD}\!\!\rightsquigarrow_{\tau}\ P' \Longrightarrow P\ \Theta\ a \in A.\ Q\ a\ {}_{FD}\!\!\rightsquigarrow_{\tau}\ P'\ \Theta\ a \in A.\ Q\ a$›
by (*simp add*: *mono-Throw-FD*)

lemma *Throw-τ-trans-right*:
‹$\forall a \in A.\ Q\ a\ {}_{FD}\!\!\rightsquigarrow_{\tau}\ Q'\ a \Longrightarrow P\ \Theta\ a \in A.\ Q\ a\ {}_{FD}\!\!\rightsquigarrow_{\tau}\ P\ \Theta\ a \in A.\ Q'\ a$›
by (*simp add*: *mono-Throw-FD*)

lemma *Throw-event-trans-left*:
‹$P\ {}_{FD}\!\!\rightsquigarrow e\ P' \Longrightarrow e \notin ev\ `\ A \Longrightarrow P\ \Theta\ a \in A.\ Q\ a\ {}_{FD}\!\!\rightsquigarrow e\ (P'\ \Theta\ a \in A.\ Q\ a)$›
apply (*simp add*: *AfterExt-Throw ready-set-Throw image-iff split*: *event.split*)
apply (*intro conjI impI*)
by (*metis AfterExt-def Throw-τ-trans-left event.simps*(*4*))
(*solves* ‹*simp add*: *Throw-STOP tick-trans-iff*›)

lemma *Throw-ev-trans-right*:
‹$P\ {}_{FD}\!\!\rightsquigarrow(ev\ e)\ P' \Longrightarrow e \in A \Longrightarrow P\ \Theta\ a \in A.\ Q\ a\ {}_{FD}\!\!\rightsquigarrow(ev\ e)\ (Q\ e)$›
by (*simp add*: *AfterExt-Throw ready-set-Throw split*: *event.split*)

lemma ‹*tickFree s* $\Longrightarrow \bot\ {}_{FD}\!\!\rightsquigarrow^{*} s\ P$›
by (*fact BOT-trace-trans-tickFree-anything*)

12.2 Reality Checks

lemma ‹$STOP\ {}_{FD}\!\!\rightsquigarrow^{*} s\ P \longleftrightarrow s = [] \wedge P = STOP$›
by (*fact STOP-trace-trans-iff*)

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

lemma *F-iff-exists-trans* :
‹$(s,\ X) \in \mathcal{F}\ P \longleftrightarrow (\exists\, P'.\ (P\ {}_{FD}\!\rightsquigarrow^{*} s\ P') \wedge X \in \mathcal{R}\ P')$›
using *F-imp-exists-trace-trans leFD-imp-leF trace-trans-imp-F-if-τ-trans-imp-leF*
by *blast*

lemma *T-iff-exists-trans* : ‹$s \in \mathcal{T}\ P \longleftrightarrow (\exists\, P'.\ P\ {}_{FD}\!\rightsquigarrow^{*} s\ P')$›
by (*meson T-imp-exists-trace-trans leFD-imp-leF leF-imp-leT trace-trans-imp-T-if-τ-trans-imp-leT*)

lemma *tickFree-imp-D-iff-trace-trans-BOT*:
‹*tickFree* $s \Longrightarrow s \in \mathcal{D}\ P \longleftrightarrow P\ {}_{FD}\!\rightsquigarrow^{*} s\ \bot$›
using *D-imp-trace-trans-BOT leFD-imp-leD trace-trans-BOT-imp-D-if-τ-trans-imp-leD*
by *blast*

lemma *D-iff-trace-trans-BOT*:
‹$s \in \mathcal{D}\ P \longleftrightarrow$ (*if tickFree s then* $P\ {}_{FD}\!\rightsquigarrow^{*} s\ \bot$ *else* $P\ {}_{FD}\!\rightsquigarrow^{*}$(*butlast s*) $\bot$)›
by (*metis tickFree-if-trans-trans-not-STOP STOP-neq-BOT*
append-butlast-last-id front-tickFree-butlast
front-tickFree-single is-processT tickFree-butlast
tickFree-imp-D-iff-trace-trans-BOT)

12.3 Other Results

lemma *trace-trans-ready-set-subset-ready-set-AfterTrace*:
‹$P\ {}_{FD}\!\rightsquigarrow^{*} s\ Q \Longrightarrow$ *ready-set* $Q \subseteq$ *ready-set* (P *afterTrace* s)›
by (*metis T-iff-exists-trans T-imp-trace-trans-iff-AfterTrace-τ-trans τ-trans-anti-mono-ready-set*)

lemma *trace-trans-imp-ready-set*:
‹$P\ {}_{FD}\!\rightsquigarrow^{*}(s\ @\ e\ \#\ t)\ Q \Longrightarrow e \in$ *ready-set* (P *afterTrace* s)›
using *T-iff-exists-trans ready-set-AfterTrace* **by** *blast*

lemma *AfterTrace-τ-trans-if-τ-trans-imp-leT* :
‹$(P\ {}_{FD}\!\rightsquigarrow^{*} s\ Q) \longleftrightarrow s \in \mathcal{T}\ P \wedge P$ *afterTrace* $s\ {}_{FD}\!\rightsquigarrow_{\tau}\ Q$›
using *T-iff-exists-trans T-imp-trace-trans-iff-AfterTrace-τ-trans* **by** *blast*

lemma ‹*deadlock-free* $P \longleftrightarrow DF\ UNIV\ {}_{FD}\!\rightsquigarrow_{\tau}\ P$›
by (*simp add*: *deadlock-free-def*)

lemma ‹$deadlock\text{-}free_{SKIP}\ P \longleftrightarrow DF_{SKIP}\ UNIV\ {}_{FD}\!\rightsquigarrow_{\tau}\ P$›
by (*fact* $deadlock\text{-}free_{SKIP}$*-FD*)

12.4 Summary: Operational Rules

In this section, we will just write down the operational laws that we have proven in a fancy way.

Absorbance rules

$$\frac{?P \ {}_{FD}\!\rightsquigarrow ?e \ ?P' \qquad ?P' \ {}_{FD}\!\rightsquigarrow_{\tau} \ ?P''}{?P \ {}_{FD}\!\rightsquigarrow ?e \ ?P''}$$
$$\frac{?P \ {}_{FD}\!\rightsquigarrow_{\tau} \ ?P' \qquad ?P' \ {}_{FD}\!\rightsquigarrow ?e \ ?P''}{?P \ {}_{FD}\!\rightsquigarrow ?e \ ?P''}$$

SKIP **rule**

$$\frac{}{SKIP \ {}_{FD}\!\rightsquigarrow \checkmark \ STOP}$$

$e \rightarrow P$ **rules**

$$\frac{?e \in ?A}{\Box a{\in}?A \rightarrow \ ?P \ a \ {}_{FD}\!\rightsquigarrow ev \ ?e \ ?P \ ?e} \qquad \frac{?e \in ?A}{\sqcap a{\in}?A \rightarrow \ \ ?P \ a \ {}_{FD}\!\rightsquigarrow ev \ ?e \ ?P \ ?e}$$

$$\frac{}{?e \rightarrow \ ?P \ {}_{FD}\!\rightsquigarrow ev \ ?e \ ?P}$$

$(\sqcap)$ **rules**

$$\frac{}{?P \sqcap \ ?Q \ {}_{FD}\!\rightsquigarrow_{\tau} \ ?P} \qquad \frac{}{?P \sqcap \ ?Q \ {}_{FD}\!\rightsquigarrow_{\tau} \ ?Q}$$
$$\frac{?e \in \ ?A}{\sqcap \ a{\in}?A. \ \ ?P \ a \ {}_{FD}\!\rightsquigarrow_{\tau} \ ?P \ ?e}$$

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

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

$(\Box)$ **rules**

$$\frac{?P \ {}_{FD}\!\rightsquigarrow_{\tau} \ ?P'}{?P \ \Box \ ?Q \ {}_{FD}\!\rightsquigarrow_{\tau} \ ?P' \ \Box \ ?Q} \qquad \frac{?Q \ {}_{FD}\!\rightsquigarrow_{\tau} \ ?Q'}{?P \ \Box \ ?Q \ {}_{FD}\!\rightsquigarrow_{\tau} \ ?P \ \Box \ ?Q'}$$
$$\frac{?P \ {}_{FD}\!\rightsquigarrow ?e \ ?P'}{?P \ \Box \ ?Q \ {}_{FD}\!\rightsquigarrow ?e \ ?P'} \qquad \frac{?Q \ {}_{FD}\!\rightsquigarrow ?e \ ?Q'}{?P \ \Box \ ?Q \ {}_{FD}\!\rightsquigarrow ?e \ ?Q'}$$

(;) **rules**

$$\frac{?P \ {}_{FD}\!\rightsquigarrow_{\tau} \ ?P'}{?P \ ; \ ?Q \ {}_{FD}\!\rightsquigarrow_{\tau} \ ?P' \ ; \ ?Q} \qquad \frac{?P \ {}_{FD}\!\rightsquigarrow ev \ ?e \ ?P'}{?P \ ; \ ?Q \ {}_{FD}\!\rightsquigarrow ev \ ?e \ ?P' \ ; \ ?Q}$$

$$\frac{\exists\, P'.\ ?P \ {}_{FD}\!\rightsquigarrow \checkmark \ P'}{?P \ ; \ ?Q \ {}_{FD}\!\rightsquigarrow_{\tau} \ ?Q}$$

Hiding **rules**

$$\frac{?P \ {}_{FD}\!\rightsquigarrow_{\tau} \ ?P'}{?P \setminus ?B \ {}_{FD}\!\rightsquigarrow_{\tau} \ ?P' \setminus ?B} \qquad \frac{?P \ {}_{FD}\!\rightsquigarrow \checkmark \ ?P'}{?P \setminus ?B \ {}_{FD}\!\rightsquigarrow \checkmark \ STOP}$$

$$\frac{?P \ {}_{FD}\!\rightsquigarrow ev \ ?e \ ?P' \qquad ?e \notin ?B}{?P \setminus ?B \ {}_{FD}\!\rightsquigarrow ev \ ?e \ ?P' \setminus ?B} \qquad \frac{?P \ {}_{FD}\!\rightsquigarrow ev \ ?e \ ?P' \qquad ?e \in ?B}{?P \setminus ?B \ {}_{FD}\!\rightsquigarrow_{\tau} \ ?P' \setminus ?B}$$

Renaming **rules**

$$\frac{?P \ {}_{FD}\!\rightsquigarrow_{\tau} \ ?P'}{Renaming \ ?P \ ?f \ {}_{FD}\!\rightsquigarrow_{\tau} \ Renaming \ ?P' \ ?f}$$

$$\frac{?P \ {}_{FD}\!\rightsquigarrow \checkmark \ ?P'}{Renaming \ ?P \ ?f \ {}_{FD}\!\rightsquigarrow \checkmark \ STOP}$$

$$\frac{?f \ ?a = ?b \qquad ?P \ {}_{FD}\!\rightsquigarrow ev \ ?a \ ?P'}{Renaming \ ?P \ ?f \ {}_{FD}\!\rightsquigarrow ev \ ?b \ Renaming \ ?P' \ ?f}$$

Sync **rules**

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

$$\frac{?e \notin ?S \qquad ?P \ {}_{FD}\!\rightsquigarrow ev \ ?e \ ?P'}{?P \ [\![?S]\!] \ ?Q \ {}_{FD}\!\rightsquigarrow ev \ ?e \ ?P' \ [\![?S]\!] \ ?Q}$$

$$\frac{?e \notin ?S \qquad ?Q \ {}_{FD}\!\rightsquigarrow ev \ ?e \ ?Q'}{?P \ [\![?S]\!] \ ?Q \ {}_{FD}\!\rightsquigarrow ev \ ?e \ ?P \ [\![?S]\!] \ ?Q'}$$

$$\frac{?e \in ?S \qquad ?P \ {}_{FD}\!\rightsquigarrow ev \ ?e \ ?P' \qquad ?Q \ {}_{FD}\!\rightsquigarrow ev \ ?e \ ?Q'}{?P \ [\![?S]\!] \ ?Q \ {}_{FD}\!\rightsquigarrow ev \ ?e \ ?P' \ [\![?S]\!] \ ?Q'}$$

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

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

$$\frac{}{SKIP\ [\![?S]\!]\ SKIP\ {}_{FD}\!\rightsquigarrow_{\tau}\ SKIP}$$

($\rhd$) **rules**

$$\frac{?P\ {}_{FD}\!\rightsquigarrow_{\tau}\ ?P'}{?P \rhd ?Q\ {}_{FD}\!\rightsquigarrow_{\tau}\ ?P' \rhd ?Q} \qquad \frac{?P\ {}_{FD}\!\rightsquigarrow ?e\ ?P'}{?P \rhd ?Q\ {}_{FD}\!\rightsquigarrow ?e\ ?P'}$$

$$\frac{}{?P \rhd ?Q\ {}_{FD}\!\rightsquigarrow_{\tau}\ ?Q}$$

($\triangle$) **rules**

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

$$\frac{?P\ {}_{FD}\!\rightsquigarrow ev\ ?e\ ?P'}{?P \triangle ?Q\ {}_{FD}\!\rightsquigarrow ev\ ?e\ ?P' \triangle ?Q} \qquad \frac{?Q\ {}_{FD}\!\rightsquigarrow ev\ ?e\ ?Q'}{?P \triangle ?Q\ {}_{FD}\!\rightsquigarrow ev\ ?e\ ?Q'}$$

Throw **rules**

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

$$\frac{\forall\, a{\in}?A.\ ?Q\ a\ {}_{FD}\!\rightsquigarrow_{\tau}\ ?Q'\ a}{?P\ \Theta\ a{\in}?A.\ ?Q\ a\ {}_{FD}\!\rightsquigarrow_{\tau}\ ?P\ \Theta\ a{\in}?A.\ ?Q'\ a}$$

$$\frac{?P\ {}_{FD}\!\rightsquigarrow ?e\ ?P' \qquad ?e \notin ev\ `\ ?A}{?P\ \Theta\ a{\in}?A.\ ?Q\ a\ {}_{FD}\!\rightsquigarrow ?e\ ?P'\ \Theta\ a{\in}?A.\ ?Q\ a}$$

$$\frac{?P\ {}_{FD}\!\rightsquigarrow ev\ ?e\ ?P' \qquad ?e \in ?A}{?P\ \Theta\ a{\in}?A.\ ?Q\ a\ {}_{FD}\!\rightsquigarrow ev\ ?e\ ?Q\ ?e}$$

end

Chapter 13

Trace Divergence Operational Semantics, bis

theory *OpSemDTBis*
imports *OpSemGenericBis HOL−Library.LaTeXsugar*
begin

The definition with ($\sqsubseteq_{FD}$) motivates us to try with other refinements.

abbreviation *τ-trans* :: ‹'α *process* ⇒ 'α *process* ⇒ *bool*› (**infixl** ‹$_{DT}\!\rightsquigarrow_{\tau}$› *50*)
where ‹*P* $_{DT}\!\rightsquigarrow_{\tau}$ *Q* ≡ *P* $\sqsubseteq_{DT}$ *Q*›

We now instantiate the locale of *HOL−CSP-OpSem.OpSemGenericBis*.

interpretation *OpSemGeneric* ‹($_{DT}\!\rightsquigarrow_{\tau}$)›
using *trans-DT* **by** *unfold-locales*
(*auto simp add*: *anti-mono-ready-set-DT mono-AfterExt-DT*)

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

lemma ‹*P* $_{DT}\!\rightsquigarrow$ *e* *P'* ⟹ *P'* $_{DT}\!\rightsquigarrow_{\tau}$ *P''* ⟹ *P* $_{DT}\!\rightsquigarrow$ *e* *P''*›
‹*P* $_{DT}\!\rightsquigarrow_{\tau}$ *P'* ⟹ *P'* $_{DT}\!\rightsquigarrow$ *e* *P''* ⟹ *P* $_{DT}\!\rightsquigarrow$ *e* *P''*›
by (*fact event-trans-τ-trans τ-trans-event-trans*)+

13.1 Operational Semantics Laws

SKIP law

lemma ‹*SKIP* $_{DT}\!\rightsquigarrow$✓ *STOP*›
by (*fact SKIP-trans-tick*)

$e \rightarrow P$ laws

lemma ‹*e* ∈ *A* ⟹ □*a* ∈ *A* → *P a* $_{DT}\!\rightsquigarrow$(*ev e*) (*P e*)›
by (*fact ev-trans-Mprefix*)

lemma ‹$e \in A \Longrightarrow \sqcap a \in A \rightarrow P\ a\ {}_{DT}\!\rightsquigarrow (ev\ e)\ (P\ e)$›
by (*fact ev-trans-Mndetprefix*)

lemma ‹$e \rightarrow P\ {}_{DT}\!\rightsquigarrow\ (ev\ e)\ P$›
by (*fact ev-trans-prefix*)

$P \sqcap Q$ laws

lemma ‹$P \sqcap Q\ {}_{DT}\!\rightsquigarrow_{\tau}\ P$›
and ‹$P \sqcap Q\ {}_{DT}\!\rightsquigarrow_{\tau}\ Q$›
by (*fact τ-trans-NdetL τ-trans-NdetR*)+

lemma ‹$a \in A \Longrightarrow (\sqcap a \in A.\ P\ a)\ {}_{DT}\!\rightsquigarrow_{\tau}\ P\ a$›
by (*fact τ-trans-GlobalNdet*)

lemma ‹*finite* $A \Longrightarrow a \in A \Longrightarrow (\bigsqcap a \in A.\ P\ a)\ {}_{DT}\!\rightsquigarrow_{\tau}\ P\ a$›
by (*fact τ-trans-MultiNdet*)

$\mu\ x.\ f\ x$ law

lemma ‹*cont* $f \Longrightarrow P = (\mu\ X.\ f\ X) \Longrightarrow P\ {}_{DT}\!\rightsquigarrow_{\tau}\ f\ P$›
by (*fact fix-point-τ-trans*)

$P \Box Q$ laws

lemma *τ-trans-DetL*: ‹$P\ {}_{DT}\!\rightsquigarrow_{\tau}\ P' \Longrightarrow P \Box Q\ {}_{DT}\!\rightsquigarrow_{\tau}\ P' \Box Q$ ›
and *τ-trans-DetR*: ‹$Q\ {}_{DT}\!\rightsquigarrow_{\tau}\ Q' \Longrightarrow P \Box Q\ {}_{DT}\!\rightsquigarrow_{\tau}\ P\ \Box\ Q'$›
by *simp-all*

lemma *τ-trans-MultiDet*:
‹*finite* $A \Longrightarrow \forall\, a \in A.\ P\ a\ {}_{DT}\!\rightsquigarrow_{\tau}\ P'\ a \Longrightarrow$
$(\Box a \in A.\ P\ a)\ {}_{DT}\!\rightsquigarrow_{\tau}\ (\Box a \in A.\ P'\ a)$›
by (*fact mono-MultiDet-DT*)

lemma ‹$P\ {}_{DT}\!\rightsquigarrow e\ P' \Longrightarrow P \Box Q\ {}_{DT}\!\rightsquigarrow e\ P'$›
and ‹$Q\ {}_{DT}\!\rightsquigarrow e\ Q' \Longrightarrow P \Box Q\ {}_{DT}\!\rightsquigarrow e\ Q'$›
by (*fact event-trans-DetL event-trans-DetR*)+

lemma ‹*finite* $A \Longrightarrow a \in A \Longrightarrow P\ a\ {}_{DT}\!\rightsquigarrow e\ Q \Longrightarrow (\Box a \in A.\ P\ a)\ {}_{DT}\!\rightsquigarrow e\ Q$›
by (*fact event-trans-MultiDet*)

$P\ ;\ Q$ laws

lemma *τ-trans-SeqL*: ‹$P\ {}_{DT}\!\rightsquigarrow_{\tau}\ P' \Longrightarrow P\ ;\ Q\ {}_{DT}\!\rightsquigarrow_{\tau}\ P'\ ;\ Q$›
by *simp*

lemma *ev-trans-SeqL*: ‹$P\ {}_{DT}\!\rightsquigarrow (ev\ e)\ P' \Longrightarrow P\ ;\ Q\ {}_{DT}\!\rightsquigarrow (ev\ e)\ P'\ ;\ Q$›

by (*auto simp add*: *ready-set-Seq AfterExt-Seq*)

lemma *τ-trans-SeqR*: ‹∃ *P′*. *P* $_{DT}\!\rightsquigarrow\!\checkmark$ *P′* ⟹ *P* ; *Q* $_{DT}\!\rightsquigarrow_{\tau}$ *Q*›
by (*metis mono-Seq-DT SKIP-Seq τ-trans-eq ready-tick-imp-τ-trans-SKIP*)

lemma ‹✓ ∈ *ready-set P* ⟹ *Q* $_{DT}\!\rightsquigarrow$(*ev e*) *Q′* ⟹ *P* ; *Q* $_{DT}\!\rightsquigarrow$(*ev e*) *Q′*›

by (*fact ev-trans-SeqR*)

P \ *B* laws

lemma *τ-trans-Hiding*: ‹*P* $_{DT}\!\rightsquigarrow_{\tau}$ *P′* ⟹ *P* \ *B* $_{DT}\!\rightsquigarrow_{\tau}$ *P′* \ *B*›
by (*fact mono-Hiding-DT*)

lemma *ev-trans-Hiding-notin*:
‹*P* $_{DT}\!\rightsquigarrow$(*ev e*) *P′* ⟹ *e* ∉ *B* ⟹ *P* \ *B* $_{DT}\!\rightsquigarrow$(*ev e*) *P′* \ *B*›
by (*metis AfterExt-def After-Hiding-DT-Hiding-After-if-ready-notin mono-Hiding-DT*

event-trans-τ-trans event.simps(*4*) *ready-notin-imp-ready-Hiding*)

lemma ‹*P* $_{DT}\!\rightsquigarrow\!\checkmark$ *P′* ⟹ *P* \ *B* $_{DT}\!\rightsquigarrow\!\checkmark$ *STOP*›
by (*fact tick-trans-Hiding*)

lemma *ev-trans-Hiding-inside*:
‹*P* $_{DT}\!\rightsquigarrow$(*ev e*) *P′* ⟹ *e* ∈ *B* ⟹ *P* \ *B* $_{DT}\!\rightsquigarrow_{\tau}$ *P′* \ *B*›
by (*metis AfterExt-def Hiding-DT-Hiding-After-if-ready-inside*
mono-Hiding-DT event.simps(*4*) *trans-DT*)

Renaming P f laws

lemma *τ-trans-Renaming*:
‹*P* $_{DT}\!\rightsquigarrow_{\tau}$ *P′* ⟹ *Renaming P f* $_{DT}\!\rightsquigarrow_{\tau}$ *Renaming P′ f*›
by (*fact mono-Renaming-DT*)

lemma *tick-trans-Renaming*: ‹*P* $_{DT}\!\rightsquigarrow\!\checkmark$ *P′* ⟹ *Renaming P f* $_{DT}\!\rightsquigarrow\!\checkmark$ *STOP*›
by (*simp add*: *AfterExt-def ready-set-Renaming tick-eq-EvExt*)

lemma *ev-trans-Renaming*:
‹*f a* = *b* ⟹ *P* $_{DT}\!\rightsquigarrow$(*ev a*) *P′* ⟹ *Renaming P f* $_{DT}\!\rightsquigarrow$(*ev b*) (*Renaming P′ f*)›
apply (*simp add*: *AfterExt-Renaming Renaming-BOT ready-set-BOT ready-set-Renaming*)
apply (*intro conjI impI*)
apply (*meson ev-elem-anteced1 imageI vimageI2*)
apply (*rule τ-trans-transitivity*[*of* - ‹*Renaming* (*P afterExt ev a*) *f*›])
apply (*solves* ‹*rule τ-trans-GlobalNdet*, *simp*›)
by (*simp add*: *τ-trans-Renaming*)

lemma ‹*P* $_{DT}\!\rightsquigarrow_{\tau}$ *P′* ⟹ *P* ⟦*a* := *b*⟧ $_{DT}\!\rightsquigarrow_{\tau}$ *P′* ⟦*a* := *b*⟧›
by (*fact τ-trans-Renaming*)

lemma $\langle P \ {}_{DT}\!\rightsquigarrow\!\checkmark \ P' \Longrightarrow P \ [\![a := b]\!] \ {}_{DT}\!\rightsquigarrow\!\checkmark \ STOP\rangle$
by (*fact tick-trans-Renaming*)

lemma *ev-trans-RenamingF*:
$\langle P \ {}_{DT}\!\rightsquigarrow\!(ev\ a)\ P' \Longrightarrow P \ [\![a := b]\!] \ {}_{DT}\!\rightsquigarrow\!(ev\ b)\ P' \ [\![a := b]\!]\rangle$
by (*metis ev-trans-Renaming fun-upd-same*)

$P \ [\![S]\!] \ Q$ laws

lemma τ*-trans-SyncL*: $\langle P \ {}_{DT}\!\rightsquigarrow_\tau \ P' \Longrightarrow P \ [\![S]\!] \ Q \ {}_{DT}\!\rightsquigarrow_\tau \ P' \ [\![S]\!] \ Q\rangle$
and τ*-trans-SyncR*: $\langle Q \ {}_{DT}\!\rightsquigarrow_\tau \ Q' \Longrightarrow P \ [\![S]\!] \ Q \ {}_{DT}\!\rightsquigarrow_\tau \ P \ [\![S]\!] \ Q'\rangle$
by *simp-all*

lemma *ev-trans-SyncL*:
$\langle e \notin S \Longrightarrow P \ {}_{DT}\!\rightsquigarrow\!(ev\ e)\ P' \Longrightarrow P \ [\![S]\!] \ Q \ {}_{DT}\!\rightsquigarrow\!(ev\ e)\ P' \ [\![S]\!] \ Q \ \rangle$
and *ev-trans-SyncR*:
$\langle e \notin S \Longrightarrow Q \ {}_{DT}\!\rightsquigarrow\!(ev\ e)\ Q' \Longrightarrow P \ [\![S]\!] \ Q \ {}_{DT}\!\rightsquigarrow\!(ev\ e)\ P \ \ [\![S]\!] \ Q'\rangle$
by (*simp-all add*: *AfterExt-Sync ready-set-Sync image-iff*)

lemma *ev-trans-SyncLR*:
$\langle [\![e \in S;\ P \ {}_{DT}\!\rightsquigarrow\!(ev\ e)\ P';\ Q \ {}_{DT}\!\rightsquigarrow\!(ev\ e)\ Q']\!] \Longrightarrow$
$P \ [\![S]\!] \ Q \ {}_{DT}\!\rightsquigarrow\!(ev\ e)\ P' \ [\![S]\!] \ Q'\rangle$
by (*simp add*: *AfterExt-Sync ready-set-Sync*)

From here we slightly defer from Roscoe's laws for *Sync*: we obtain the following rules for *SKIP* instead of *STOP*.

lemma *tick-trans-SyncL*: $\langle P \ {}_{DT}\!\rightsquigarrow\!\checkmark \ P' \Longrightarrow P \ [\![S]\!] \ Q \ {}_{DT}\!\rightsquigarrow_\tau \ SKIP \ [\![S]\!] \ Q\rangle$
and *tick-trans-SyncR*: $\langle Q \ {}_{DT}\!\rightsquigarrow\!\checkmark \ Q' \Longrightarrow P \ [\![S]\!] \ Q \ {}_{DT}\!\rightsquigarrow_\tau \ P \ [\![S]\!] \ SKIP\rangle$
by (*simp-all add*: *ready-tick-imp-*τ*-trans-SKIP*)

lemma *tick-trans-SKIP-Sync-SKIP*: $\langle SKIP \ [\![S]\!] \ SKIP \ {}_{DT}\!\rightsquigarrow\!\checkmark \ STOP\rangle$
by (*simp add*: *SKIP-trans-tick Sync-SKIP-SKIP*)

lemma τ*-trans-SKIP-Sync-SKIP*: $\langle SKIP \ [\![S]\!] \ SKIP \ {}_{DT}\!\rightsquigarrow_\tau \ SKIP\rangle$
by (*simp add*: *Sync-SKIP-SKIP*)

$P \rhd Q$ laws

lemma *Sliding-*τ*-trans-left*: $\langle P \ {}_{DT}\!\rightsquigarrow_\tau \ P' \Longrightarrow P \rhd Q \ {}_{DT}\!\rightsquigarrow_\tau \ P' \rhd Q\rangle$
unfolding *Sliding-def* **by** *simp*

lemma $\langle P \ {}_{DT}\!\rightsquigarrow e \ P' \Longrightarrow P \rhd Q \ {}_{DT}\!\rightsquigarrow e \ P'\rangle$
by (*fact Sliding-event-transL*)

lemma $\langle P \rhd Q \ {}_{DT}\!\rightsquigarrow_\tau \ Q\rangle$
by (*fact Sliding-*τ*-transR*)

$P \triangle Q$ laws

lemma *Interrupt-*τ*-trans-left*: $\langle P \ {}_{DT}\!\rightsquigarrow_\tau \ P' \Longrightarrow P \triangle Q \ {}_{DT}\!\rightsquigarrow_\tau \ P' \triangle Q\rangle$
by (*simp add*: *mono-Interrupt-DT*)

lemma *Interrupt-τ-trans-right*: ‹$Q\ {}_{DT}\!\rightsquigarrow_{\tau}\ Q' \Longrightarrow P \triangle Q\ {}_{DT}\!\rightsquigarrow_{\tau}\ P \triangle Q'$›
by (*simp add*: *mono-Interrupt-DT*)

lemma *Interrupt-ev-trans-left*:
‹$P\ {}_{DT}\!\rightsquigarrow(ev\ e)\ P' \Longrightarrow P \triangle Q\ {}_{DT}\!\rightsquigarrow(ev\ e)\ P' \triangle Q$›
by (*simp add*: *AfterExt-def After-Interrupt Interrupt-τ-trans-left ready-set-Interrupt*)

lemma *Interrupt-ev-trans-right*: ‹$Q\ {}_{DT}\!\rightsquigarrow(ev\ e)\ Q' \Longrightarrow P \triangle Q\ {}_{DT}\!\rightsquigarrow(ev\ e)\ Q'$›
by (*simp add*: *AfterExt-def After-Interrupt ready-set-Interrupt*)

Throw P A Q laws

lemma *Throw-τ-trans-left*:
‹$P\ {}_{DT}\!\rightsquigarrow_{\tau}\ P' \Longrightarrow P\ \Theta\ a \in A.\ Q\ a\ {}_{DT}\!\rightsquigarrow_{\tau}\ P'\ \Theta\ a \in A.\ Q\ a$›
by (*simp add*: *mono-Throw-DT*)

lemma *Throw-τ-trans-right*:
‹$\forall a \in A.\ Q\ a\ {}_{DT}\!\rightsquigarrow_{\tau}\ Q'\ a \Longrightarrow P\ \Theta\ a \in A.\ Q\ a\ {}_{DT}\!\rightsquigarrow_{\tau}\ P\ \Theta\ a \in A.\ Q'\ a$›
by (*simp add*: *mono-Throw-DT*)

lemma *Throw-event-trans-left*:
‹$P\ {}_{DT}\!\rightsquigarrow e\ P' \Longrightarrow e \notin ev\ `\ A \Longrightarrow P\ \Theta\ a \in A.\ Q\ a\ {}_{DT}\!\rightsquigarrow e\ (P'\ \Theta\ a \in A.\ Q\ a)$›
apply (*simp add*: *AfterExt-Throw ready-set-Throw image-iff split*: *event.split*)
apply (*intro conjI impI*)
by (*metis AfterExt-def Throw-τ-trans-left event.simps*(*4*))
(*solves* ‹*simp add*: *Throw-STOP tick-trans-iff*›)

lemma *Throw-ev-trans-right*:
‹$P\ {}_{DT}\!\rightsquigarrow(ev\ e)\ P' \Longrightarrow e \in A \Longrightarrow P\ \Theta\ a \in A.\ Q\ a\ {}_{DT}\!\rightsquigarrow(ev\ e)\ (Q\ e)$›
by (*simp add*: *AfterExt-Throw ready-set-Throw split*: *event.split*)

lemma ‹*tickFree s* $\Longrightarrow \bot\ {}_{DT}\!\rightsquigarrow^{*} s\ P$›
by (*fact BOT-trace-trans-tickFree-anything*)

13.2 Reality Checks

lemma ‹$STOP\ {}_{DT}\!\rightsquigarrow^{*} s\ P \longleftrightarrow s = [] \wedge P = STOP$›
by (*fact STOP-trace-trans-iff*)

lemma *T-iff-exists-trans* : ‹$s \in \mathcal{T}\ P \longleftrightarrow (\exists P'.\ P\ {}_{DT}\!\rightsquigarrow^{*} s\ P')$›

using *T-imp-exists-trace-trans leDT-imp-leT trace-trans-imp-T-if-τ-trans-imp-leT*
by *blast*

lemma *tickFree-imp-D-iff-trace-trans-BOT*:
‹*tickFree* $s \Longrightarrow s \in \mathcal{D}\ P \longleftrightarrow P\ {}_{DT}\!\rightsquigarrow^{*} s\ \bot$›
using *D-imp-trace-trans-BOT leDT-imp-leD trace-trans-BOT-imp-D-if-τ-trans-imp-leD*
by *blast*

lemma *D-iff-trace-trans-BOT*:
‹$s \in \mathcal{D}\ P \longleftrightarrow$ (*if tickFree* s *then* $P\ {}_{DT}\!\rightsquigarrow^{*} s\ \bot$ *else* $P\ {}_{DT}\!\rightsquigarrow^{*}$(*butlast* s) $\bot$)›
by (*metis tickFree-if-trans-trans-not-STOP STOP-neq-BOT*
append-butlast-last-id front-tickFree-butlast
front-tickFree-single is-processT tickFree-butlast
tickFree-imp-D-iff-trace-trans-BOT)

13.3 Other Results

lemma *trace-trans-ready-set-subset-ready-set-AfterTrace*:
‹$P\ {}_{DT}\!\rightsquigarrow^{*} s\ Q \Longrightarrow$ *ready-set* $Q \subseteq$ *ready-set* (P *afterTrace* s)›
by (*metis T-iff-exists-trans T-imp-trace-trans-iff-AfterTrace-τ-trans τ-trans-anti-mono-ready-set*)

lemma *trace-trans-imp-ready-set*:
‹$P\ {}_{DT}\!\rightsquigarrow^{*}(s\ @\ e\ \#\ t)\ Q \Longrightarrow e \in$ *ready-set* (P *afterTrace* s)›
using *T-iff-exists-trans ready-set-AfterTrace* **by** *blast*

lemma *AfterTrace-τ-trans-if-τ-trans-imp-leT* :
‹$(P\ {}_{DT}\!\rightsquigarrow^{*} s\ Q) \longleftrightarrow s \in \mathcal{T}\ P \wedge P$ *afterTrace* $s\ {}_{DT}\!\rightsquigarrow_{\tau}\ Q$›
using *T-iff-exists-trans T-imp-trace-trans-iff-AfterTrace-τ-trans* **by** *blast*

13.4 Summary: Operational Rules

In this section, we will just write down the operational laws that we have proven in a fancy way.

Absorbance rules

$$\frac{?P\ {}_{DT}\!\rightsquigarrow ?e\ ?P' \qquad ?P'\ {}_{DT}\!\rightsquigarrow_{\tau}\ ?P''}{?P\ {}_{DT}\!\rightsquigarrow ?e\ ?P''}$$

$$\frac{?P\ {}_{DT}\!\rightsquigarrow_{\tau}\ ?P' \qquad ?P'\ {}_{DT}\!\rightsquigarrow ?e\ ?P''}{?P\ {}_{DT}\!\rightsquigarrow ?e\ ?P''}$$

SKIP **rule**

$$\frac{}{SKIP\ {}_{DT}\!\rightsquigarrow\checkmark\ STOP}$$

$e \to P$ **rules**

$$\frac{?e \in ?A}{\Box a{\in}?A \to ?P\ a\ {}_{DT}{\rightsquigarrow} ev\ ?e\ ?P\ ?e} \qquad \frac{?e \in ?A}{\sqcap a{\in}?A \to\ ?P\ a\ {}_{DT}{\rightsquigarrow} ev\ ?e\ ?P\ ?e}$$

$$\frac{}{?e \to ?P\ {}_{DT}{\rightsquigarrow} ev\ ?e\ ?P}$$

$(\sqcap)$ **rules**

$$\frac{}{?P \sqcap ?Q\ {}_{DT}{\rightsquigarrow}_{\tau}\ ?P} \qquad \frac{}{?P \sqcap ?Q\ {}_{DT}{\rightsquigarrow}_{\tau}\ ?Q}$$

$$\frac{?e \in ?A}{\sqcap\ a{\in}?A.\ \ ?P\ a\ {}_{DT}{\rightsquigarrow}_{\tau}\ ?P\ ?e}$$

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

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

$(\Box)$ **rules (more powerful than in OpSemFD)**

$$\frac{?P\ {}_{DT}{\rightsquigarrow}_{\tau}\ ?P'}{?P \,\Box\, ?Q\ {}_{DT}{\rightsquigarrow}_{\tau}\ ?P' \,\Box\, ?Q} \qquad \frac{?Q\ {}_{DT}{\rightsquigarrow}_{\tau}\ ?Q'}{?P \,\Box\, ?Q\ {}_{DT}{\rightsquigarrow}_{\tau}\ ?P \,\Box\, ?Q'}$$

$(;)$ **rules**

$$\frac{?P\ {}_{DT}{\rightsquigarrow}_{\tau}\ ?P'}{?P\ ;\ ?Q\ {}_{DT}{\rightsquigarrow}_{\tau}\ ?P'\ ;\ ?Q} \qquad \frac{?P\ {}_{DT}{\rightsquigarrow} ev\ ?e\ ?P'}{?P\ ;\ ?Q\ {}_{DT}{\rightsquigarrow} ev\ ?e\ ?P'\ ;\ ?Q}$$

$$\frac{\exists\, P'.\ ?P\ {}_{DT}{\rightsquigarrow}\checkmark\ P'}{?P\ ;\ ?Q\ {}_{DT}{\rightsquigarrow}_{\tau}\ ?Q}$$

Hiding **rules**

$$\frac{?P\ {}_{DT}{\rightsquigarrow}_{\tau}\ ?P'}{?P \setminus ?B\ {}_{DT}{\rightsquigarrow}_{\tau}\ ?P' \setminus ?B} \qquad \frac{?P\ {}_{DT}{\rightsquigarrow}\checkmark\ ?P'}{?P \setminus ?B\ {}_{DT}{\rightsquigarrow}\checkmark\ STOP}$$

$$\frac{?P\ {}_{DT}{\rightsquigarrow} ev\ ?e\ ?P' \qquad ?e \notin ?B}{?P \setminus ?B\ {}_{DT}{\rightsquigarrow} ev\ ?e\ ?P' \setminus ?B} \qquad \frac{?P\ {}_{DT}{\rightsquigarrow} ev\ ?e\ ?P' \qquad ?e \in ?B}{?P \setminus ?B\ {}_{DT}{\rightsquigarrow}_{\tau}\ ?P' \setminus ?B}$$

Renaming **rules**

$$\frac{?P \ {}_{DT}\!\rightsquigarrow_{\tau} \ ?P'}{Renaming \ ?P \ ?f \ {}_{DT}\!\rightsquigarrow_{\tau} \ Renaming \ ?P' \ ?f}$$

$$\frac{?P \ {}_{DT}\!\rightsquigarrow\checkmark \ ?P'}{Renaming \ ?P \ ?f \ {}_{DT}\!\rightsquigarrow\checkmark \ STOP}$$

$$\frac{?f \ ?a = ?b \qquad ?P \ {}_{DT}\!\rightsquigarrow ev \ ?a \ ?P'}{Renaming \ ?P \ ?f \ {}_{DT}\!\rightsquigarrow ev \ ?b \ Renaming \ ?P' \ ?f}$$

Sync **rules**

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

$$\frac{?e \notin ?S \qquad ?P \ {}_{DT}\!\rightsquigarrow ev \ ?e \ ?P'}{?P \ [\![?S]\!] \ ?Q \ {}_{DT}\!\rightsquigarrow ev \ ?e \ ?P' \ [\![?S]\!] \ ?Q}$$

$$\frac{?e \notin ?S \qquad ?Q \ {}_{DT}\!\rightsquigarrow ev \ ?e \ ?Q'}{?P \ [\![?S]\!] \ ?Q \ {}_{DT}\!\rightsquigarrow ev \ ?e \ ?P \ [\![?S]\!] \ ?Q'}$$

$$\frac{?e \in ?S \qquad ?P \ {}_{DT}\!\rightsquigarrow ev \ ?e \ ?P' \qquad ?Q \ {}_{DT}\!\rightsquigarrow ev \ ?e \ ?Q'}{?P \ [\![?S]\!] \ ?Q \ {}_{DT}\!\rightsquigarrow ev \ ?e \ ?P' \ [\![?S]\!] \ ?Q'}$$

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

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

$$\frac{}{SKIP \ [\![?S]\!] \ SKIP \ {}_{DT}\!\rightsquigarrow_{\tau} \ SKIP}$$

($\triangleright$) **rules**

$$\frac{?P \ {}_{DT}\!\rightsquigarrow_{\tau} \ ?P'}{?P \triangleright ?Q \ {}_{DT}\!\rightsquigarrow_{\tau} \ ?P' \triangleright ?Q} \qquad \frac{?P \ {}_{DT}\!\rightsquigarrow ?e \ ?P'}{?P \triangleright ?Q \ {}_{DT}\!\rightsquigarrow ?e \ ?P'}$$

$$\frac{}{?P \triangleright ?Q \ {}_{DT}\!\rightsquigarrow_{\tau} \ ?Q}$$

($\triangle$) **rules**

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

$$\frac{?P \ {}_{DT}\!\rightsquigarrow ev \ ?e \ ?P'}{?P \ \triangle \ ?Q \ {}_{DT}\!\rightsquigarrow ev \ ?e \ ?P' \ \triangle \ ?Q} \qquad \frac{?Q \ {}_{DT}\!\rightsquigarrow ev \ ?e \ ?Q'}{?P \ \triangle \ ?Q \ {}_{DT}\!\rightsquigarrow ev \ ?e \ ?Q'}$$

Throw **rules**

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

$$\frac{\forall \, a{\in}?A. \ ?Q \ a \ {}_{DT}\!\rightsquigarrow_{\tau} \ ?Q' \ a}{?P \ \Theta \ a{\in}?A. \ ?Q \ a \ {}_{DT}\!\rightsquigarrow_{\tau} \ ?P \ \Theta \ a{\in}?A. \ ?Q' \ a}$$

$$\frac{?P \ {}_{DT}\!\rightsquigarrow ?e \ ?P' \qquad ?e \notin ev \ `\ ?A}{?P \ \Theta \ a{\in}?A. \ ?Q \ a \ {}_{DT}\!\rightsquigarrow ?e \ ?P' \ \Theta \ a{\in}?A. \ ?Q \ a}$$

$$\frac{?P \ {}_{DT}\!\rightsquigarrow ev \ ?e \ ?P' \qquad ?e \in ?A}{?P \ \Theta \ a{\in}?A. \ ?Q \ a \ {}_{DT}\!\rightsquigarrow ev \ ?e \ ?Q \ ?e}$$

end

Chapter 14

Failure Operational Semantics, bis

theory *OpSemFBis*
imports *OpSemGenericBis HOL−Library.LaTeXsugar*
begin

The definition with ($\sqsubseteq_{FD}$) motivates us to try with other refinements.

abbreviation *τ-trans* :: ‹'α *process* ⇒ 'α *process* ⇒ *bool*› (**infixl** ‹$_F\rightsquigarrow_\tau$› *50*)
where ‹*P* $_F\rightsquigarrow_\tau$ *Q* ≡ *P* $\sqsubseteq_F$ *Q*›

We now instantiate the locale of *HOL−CSP-OpSem.OpSemGenericBis*.

interpretation *OpSemGeneric* ‹($_F\rightsquigarrow_\tau$) :: 'α *process* ⇒ 'α *process* ⇒ *bool*›
proof *unfold-locales*
show ‹(*P* :: 'α *process*) ⊓ *Q* $_F\rightsquigarrow_\tau$ *P*› **for** *P Q* **by** *simp*
next
show ‹(*P* :: 'α *process*) $_F\rightsquigarrow_\tau$ *Q* ⟹ *Q* $_F\rightsquigarrow_\tau$ *R* ⟹ *P* $_F\rightsquigarrow_\tau$ *R*› **for** *P Q R*
by (*rule trans-F*)
next
show ‹(*P* :: 'α *process*) $_F\rightsquigarrow_\tau$ *Q* ⟹ *ready-set Q* ⊆ *ready-set P*› **for** *P Q*
by (*simp add*: *anti-mono-ready-set-F*)
next
show ‹(*e*::'α *event*) ∈ *ready-set Q* ⟹ *P* $_F\rightsquigarrow_\tau$ *Q* ⟹
P afterExt e $_F\rightsquigarrow_\tau$ *Q afterExt e*› **for** *e P Q*
by (*auto simp add*: *AfterExt-def mono-After-F split*: *event.split*)
qed

notation *event-trans* (‹-/ $_F\rightsquigarrow$-/ -› [*50*, *3*, *51*] *50*)
notation *trace-trans* (‹-/ $_F\rightsquigarrow^*$-/ -› [*50*, *3*, *51*] *50*)

lemma ‹*P* $_F\rightsquigarrow$ *e P'* ⟹ *P'* $_F\rightsquigarrow_\tau$ *P''* ⟹ *P* $_F\rightsquigarrow$ *e P''*›
‹*P* $_F\rightsquigarrow_\tau$ *P'* ⟹ *P'* $_F\rightsquigarrow$ *e P''* ⟹ *P* $_F\rightsquigarrow$ *e P''*›
by (*fact event-trans-τ-trans τ-trans-event-trans*)+

14.1 Operational Semantics Laws

SKIP law

lemma ‹*SKIP* $_F\!\rightsquigarrow\!\checkmark$ *STOP*›
by (*fact SKIP-trans-tick*)

$e \rightarrow P$ laws

lemma ‹$e \in A \Longrightarrow \Box a \in A \rightarrow P\ a\ {}_F\!\rightsquigarrow\!(ev\ e)\ (P\ e)$›
by (*fact ev-trans-Mprefix*)

lemma ‹$e \in A \Longrightarrow \sqcap a \in A \rightarrow P\ a\ {}_F\!\rightsquigarrow\!(ev\ e)\ (P\ e)$›
by (*fact ev-trans-Mndetprefix*)

lemma ‹$e \rightarrow P\ {}_F\!\rightsquigarrow\ (ev\ e)\ P$›
by (*fact ev-trans-prefix*)

$P \sqcap Q$ laws

lemma ‹$P \sqcap Q\ {}_F\!\rightsquigarrow_\tau\ P$›
and ‹$P \sqcap Q\ {}_F\!\rightsquigarrow_\tau\ Q$›
by (*fact* τ*-trans-NdetL* τ*-trans-NdetR*)+

lemma ‹$a \in A \Longrightarrow (\sqcap a \in A.\ P\ a)\ {}_F\!\rightsquigarrow_\tau\ P\ a$›
by (*fact* τ*-trans-GlobalNdet*)

lemma ‹*finite* $A \Longrightarrow a \in A \Longrightarrow (\bigsqcap a \in A.\ P\ a)\ {}_F\!\rightsquigarrow_\tau\ P\ a$›
by (*fact* τ*-trans-MultiNdet*)

$\mu\ x.\ f\ x$ law

lemma ‹*cont* $f \Longrightarrow P = (\mu\ X.\ f\ X) \Longrightarrow P\ {}_F\!\rightsquigarrow_\tau\ f\ P$›
by (*fact fix-point-*τ*-trans*)

$P \Box Q$ laws

lemma ‹$P\ {}_F\!\rightsquigarrow e\ P' \Longrightarrow P \Box Q\ {}_F\!\rightsquigarrow e\ P'$›
and ‹$Q\ {}_F\!\rightsquigarrow e\ Q' \Longrightarrow P \Box Q\ {}_F\!\rightsquigarrow e\ Q'$›
by (*fact event-trans-DetL event-trans-DetR*)+

lemma ‹*finite* $A \Longrightarrow a \in A \Longrightarrow P\ a\ {}_F\!\rightsquigarrow e\ Q \Longrightarrow (\Box a \in A.\ P\ a)\ {}_F\!\rightsquigarrow e\ Q$›
by (*fact event-trans-MultiDet*)

P **;** Q laws

lemma τ*-trans-SeqR*: ‹$\exists\, P'.\ P\ {}_F\!\rightsquigarrow\!\checkmark\ P' \Longrightarrow P\ ;\ Q\ {}_F\!\rightsquigarrow_\tau\ Q$›
apply (*elim exE conjE*, *drule ready-tick-imp-*τ*-trans-SKIP*)
by (*metis* (*no-types*, *lifting*) *CSP-Laws.mono-Seq-FD D-Seq SKIP-Seq divergence-refine-def*
dual-order.refl failure-divergence-refine-def leFD-imp-leF leF-leD-imp-leFD le-sup-iff)

— not in the Roscoe's because direct consequence of $\exists P'.\ ?P\ {}_F\!\rightsquigarrow\!\checkmark\ P' \Longrightarrow ?P$; $?Q\ {}_F\!\rightsquigarrow_\tau\ ?Q$
lemma $\langle\checkmark \in \mathit{ready\text{-}set}\ P \Longrightarrow Q\ {}_F\!\rightsquigarrow(\mathit{ev}\ e)\ Q' \Longrightarrow P\ ;\ Q\ {}_F\!\rightsquigarrow(\mathit{ev}\ e)\ Q'\rangle$

by (*fact ev-trans-SeqR*)

$P \setminus B$ laws

lemma *τ-trans-Hiding*: $\langle P\ {}_F\!\rightsquigarrow_\tau\ P' \Longrightarrow P \setminus B\ {}_F\!\rightsquigarrow_\tau\ P' \setminus B\rangle$
by (*fact mono-Hiding-F*)

lemma *ev-trans-Hiding-notin*:
$\langle P\ {}_F\!\rightsquigarrow(\mathit{ev}\ e)\ P' \Longrightarrow e \notin B \Longrightarrow P \setminus B\ {}_F\!\rightsquigarrow(\mathit{ev}\ e)\ P' \setminus B\rangle$
by (*metis AfterExt-def After-Hiding-F-Hiding-After-if-ready-notin mono-Hiding-F*

event-trans-τ-trans event.simps(4) *ready-notin-imp-ready-Hiding*)

lemma $\langle P\ {}_F\!\rightsquigarrow\!\checkmark\ P' \Longrightarrow P \setminus B\ {}_F\!\rightsquigarrow\!\checkmark\ \mathit{STOP}\rangle$
by (*fact tick-trans-Hiding*)

lemma *ev-trans-Hiding-inside*:
$\langle P\ {}_F\!\rightsquigarrow(\mathit{ev}\ e)\ P' \Longrightarrow e \in B \Longrightarrow P \setminus B\ {}_F\!\rightsquigarrow_\tau\ P' \setminus B\rangle$
by (*metis AfterExt-def Hiding-F-Hiding-After-if-ready-inside*
mono-Hiding-F event.simps(4) *trans-F*)

Renaming P f laws

lemma *tick-trans-Renaming*: $\langle P\ {}_F\!\rightsquigarrow\!\checkmark\ P' \Longrightarrow \mathit{Renaming}\ P\ f\ {}_F\!\rightsquigarrow\!\checkmark\ \mathit{STOP}\rangle$
by (*simp add*: *AfterExt-def ready-set-Renaming tick-eq-EvExt*)

lemma $\langle P\ {}_F\!\rightsquigarrow\!\checkmark\ P' \Longrightarrow P\ [\![a := b]\!]\ {}_F\!\rightsquigarrow\!\checkmark\ \mathit{STOP}\rangle$
using *tick-trans-Renaming* **by** *blast*

$P\ [\![S]\!]\ Q$ laws

From here we slightly defer from Roscoe's laws for *Sync*: we obtain the following rules for *SKIP* instead of *STOP*.

lemma *tick-trans-SyncL*: $\langle P\ {}_F\!\rightsquigarrow\!\checkmark\ P' \Longrightarrow P\ [\![S]\!]\ Q\ {}_F\!\rightsquigarrow_\tau\ \mathit{SKIP}\ [\![S]\!]\ Q\rangle$
and *tick-trans-SyncR*: $\langle Q\ {}_F\!\rightsquigarrow\!\checkmark\ Q' \Longrightarrow P\ [\![S]\!]\ Q\ {}_F\!\rightsquigarrow_\tau\ P\ [\![S]\!]\ \mathit{SKIP}\rangle$
by (*simp-all add*: *D-SKIP ready-tick-imp-τ-trans-SKIP*
divergence-refine-def leFD-imp-leF leF-leD-imp-leFD)

lemma *tick-trans-SKIP-Sync-SKIP*: $\langle \mathit{SKIP}\ [\![S]\!]\ \mathit{SKIP}\ {}_F\!\rightsquigarrow\!\checkmark\ \mathit{STOP}\rangle$
by (*simp add*: *SKIP-trans-tick Sync-SKIP-SKIP*)

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

$P \rhd Q$ laws

lemma ‹P $_F\!\rightsquigarrow e$ $P' \Longrightarrow P \rhd Q$ $_F\!\rightsquigarrow e$ P'›
by (*fact Sliding-event-transL*)

lemma ‹$P \rhd Q$ $_F\!\rightsquigarrow_\tau$ Q›
by (*fact Sliding-τ-transR*)

$P \triangle Q$ laws

lemma *Interrupt-ev-trans-right*: ‹Q $_F\!\rightsquigarrow$(*ev* e) $Q' \Longrightarrow P \triangle Q$ $_F\!\rightsquigarrow$(*ev* e) Q'›
by (*simp add*: *AfterExt-def After-Interrupt ready-set-Interrupt*)

Throw P A Q laws

lemma *Throw-τ-trans-right*:
‹$\forall a \in A.\ Q\ a$ $_F\!\rightsquigarrow_\tau$ $Q'\ a \Longrightarrow P \ominus a \in A.\ Q\ a$ $_F\!\rightsquigarrow_\tau$ $P \ominus a \in A.\ Q'\ a$›
by (*simp add*: *mono-right-Throw-F*)

lemma *Throw-ev-trans-right*:
‹P $_F\!\rightsquigarrow$(*ev* e) $P' \Longrightarrow e \in A \Longrightarrow P \ominus a \in A.\ Q\ a$ $_F\!\rightsquigarrow$(*ev* e) (Q e)›
by (*simp add*: *AfterExt-Throw ready-set-Throw*)

lemma ‹*tickFree* $s \Longrightarrow \bot$ $_F\!\rightsquigarrow^* s$ P›
by (*fact BOT-trace-trans-tickFree-anything*)

14.2 Reality Checks

lemma ‹*STOP* $_F\!\rightsquigarrow^* s$ $P \longleftrightarrow s = [] \wedge P = STOP$›
by (*fact STOP-trace-trans-iff*)

lemma *SKIP-trace-trans-iff* :
‹*SKIP* $_F\!\rightsquigarrow^* s$ $P \longleftrightarrow s = [] \wedge P = SKIP \vee s = [\checkmark] \wedge P = STOP$›
by (*simp add*: *τ-trans-imp-leF-imp-SKIP-trace-trans-iff*)

lemma *F-iff-exists-trans* :
‹$(s, X) \in \mathcal{F}\ P \longleftrightarrow (\exists P'.\ (P$ $_F\!\rightsquigarrow^* s$ $P') \wedge X \in \mathcal{R}\ P')$›
using *F-imp-exists-trace-trans trace-trans-imp-F-if-τ-trans-imp-leF* **by** *blast*

lemma *T-iff-exists-trans* : ‹$s \in \mathcal{T}\ P \longleftrightarrow (\exists P'.\ P$ $_F\!\rightsquigarrow^* s$ $P')$›
by (*meson T-imp-exists-trace-trans leF-imp-leT trace-trans-imp-T-if-τ-trans-imp-leT*)

14.3 Other Results

lemma *trace-trans-ready-set-subset-ready-set-AfterTrace*:
‹$P\ {}_F\!\rightsquigarrow^*s\ Q \Longrightarrow$ *ready-set* $Q \subseteq$ *ready-set* $(P$ *afterTrace* $s)$›
by (*metis T-iff-exists-trans T-imp-trace-trans-iff-AfterTrace-τ-trans τ-trans-anti-mono-ready-set*)

lemma *trace-trans-imp-ready-set*:
‹$P\ {}_F\!\rightsquigarrow^*(s\ @\ e\ \#\ t)\ Q \Longrightarrow e \in$ *ready-set* $(P$ *afterTrace* $s)$›
using *T-iff-exists-trans ready-set-AfterTrace* **by** *blast*

lemma *AfterTrace-τ-trans-if-τ-trans-imp-leT* :
‹$(P\ {}_F\!\rightsquigarrow^*s\ Q) \longleftrightarrow s \in \mathcal{T}\ P \wedge P$ *afterTrace* $s\ {}_F\!\rightsquigarrow_\tau\ Q$›
using *T-iff-exists-trans T-imp-trace-trans-iff-AfterTrace-τ-trans* **by** *blast*

lemma ‹*deadlock-free* $P \longleftrightarrow$ *DF UNIV* $\ {}_F\!\rightsquigarrow_\tau\ P$›
by (*fact deadlock-free-F*)

lemma ‹*deadlock-free*$_{SKIP}$ $P \longleftrightarrow$ *DF*$_{SKIP}$ *UNIV* $\ {}_F\!\rightsquigarrow_\tau\ P$›
by (*simp add*: *deadlock-free*$_{SKIP}$*-def*)

14.4 Nicely written operational rules

In this section, we will just write down the operational laws that we have proven in a fancy way.

Absorbance rules

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

SKIP **rule**

$$\frac{}{SKIP\ {}_F\!\rightsquigarrow\checkmark\ STOP}$$

$e \rightarrow P$ **rules**

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

$$\frac{}{?e \rightarrow\ ?P\ {}_F\!\rightsquigarrow ev\ ?e\ ?P}$$

($\sqcap$) **rules**

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

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

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

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

($\Box$) **rules**

$$\frac{?P \ {}_F\!\rightsquigarrow ?e \ ?P'}{?P \ \Box \ ?Q \ {}_F\!\rightsquigarrow ?e \ ?P'} \qquad \frac{?Q \ {}_F\!\rightsquigarrow ?e \ ?Q'}{?P \ \Box \ ?Q \ {}_F\!\rightsquigarrow ?e \ ?Q'}$$

(;) **rule**

$$\frac{\exists\, P'.\ ?P \ {}_F\!\rightsquigarrow\checkmark \ P'}{?P \mathbin{;} ?Q \ {}_F\!\rightsquigarrow_\tau \ ?Q}$$

Hiding **rules**

$$\frac{?P \ {}_F\!\rightsquigarrow_\tau \ ?P'}{?P \setminus ?B \ {}_F\!\rightsquigarrow_\tau \ ?P' \setminus ?B} \qquad \frac{?P \ {}_F\!\rightsquigarrow\checkmark \ ?P'}{?P \setminus ?B \ {}_F\!\rightsquigarrow\checkmark \ STOP}$$

$$\frac{?P \ {}_F\!\rightsquigarrow ev \ ?e \ ?P' \qquad ?e \notin ?B}{?P \setminus ?B \ {}_F\!\rightsquigarrow ev \ ?e \ ?P' \setminus ?B} \qquad \frac{?P \ {}_F\!\rightsquigarrow ev \ ?e \ ?P' \qquad ?e \in ?B}{?P \setminus ?B \ {}_F\!\rightsquigarrow_\tau \ ?P' \setminus ?B}$$

Renaming **rule**

$$\frac{?P \ {}_F\!\rightsquigarrow\checkmark \ ?P'}{Renaming\ ?P\ ?f \ {}_F\!\rightsquigarrow\checkmark \ STOP}$$

Sync **rules**

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

$$\frac{}{SKIP \ [\![?S]\!] \ SKIP \ {}_F\!\rightsquigarrow_\tau \ SKIP}$$

($\triangleright$) **rules**

$$\frac{?P \ {}_F\!\rightsquigarrow ?e \ ?P'}{?P \triangleright ?Q \ {}_F\!\rightsquigarrow ?e \ ?P'}$$

$$\frac{}{?P \triangleright ?Q \ {}_F\!\rightsquigarrow_\tau \ ?Q}$$

($\triangle$) **rule**

$$\frac{?Q \ {}_F\!\rightsquigarrow ev \ ?e \ ?Q'}{?P \triangle ?Q \ {}_F\!\rightsquigarrow ev \ ?e \ ?Q'}$$

Throw **rules**

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

$$\frac{?P \ {}_F\!\rightsquigarrow ev \ ?e \ ?P' \qquad ?e \in ?A}{?P \ \Theta \ a{\in}?A.\ ?Q \ a \ {}_F\!\rightsquigarrow ev \ ?e \ ?Q \ ?e}$$

end

Chapter 15

Trace Operational Semantics, bis

theory *OpSemTBis*
imports *OpSemGenericBis HOL−Library.LaTeXsugar*
begin

The definition with ($\sqsubseteq_{FD}$) motivates us to try with other refinements.

abbreviation *τ-trans* :: ‹'α *process* ⇒ 'α *process* ⇒ *bool*› (**infixl** ‹$_T\rightsquigarrow_\tau$› *50*)
where ‹*P* $_T\rightsquigarrow_\tau$ *Q* ≡ *P* $\sqsubseteq_T$ *Q*›

We now instantiate the locale of *HOL−CSP-OpSem.OpSemGenericBis*.

interpretation *OpSemGeneric* ‹($_T\rightsquigarrow_\tau$)›
using *trans-T* **by** *unfold-locales*
(*auto simp add*: *anti-mono-ready-set-T mono-AfterExt-T*)

notation *event-trans* (‹-/ $_T\rightsquigarrow$-/ -› [*50*, *3*, *51*] *50*)
notation *trace-trans* (‹-/ $_T\rightsquigarrow^*$-/ -› [*50*, *3*, *51*] *50*)

lemma ‹*P* $_T\rightsquigarrow$ *e P'* ⟹ *P'* $_T\rightsquigarrow_\tau$ *P''* ⟹ *P* $_T\rightsquigarrow$ *e P''*›
‹*P* $_T\rightsquigarrow_\tau$ *P'* ⟹ *P'* $_T\rightsquigarrow$ *e P''* ⟹ *P* $_T\rightsquigarrow$ *e P''*›
by (*fact event-trans-τ-trans τ-trans-event-trans*)+

15.1 Operational Semantics Laws

SKIP law

lemma ‹*SKIP* $_T\rightsquigarrow$✓ *STOP*›
by (*fact SKIP-trans-tick*)

$e \rightarrow P$ laws

lemma ‹*e* ∈ *A* ⟹ □*a* ∈ *A* → *P a* $_T\rightsquigarrow$(*ev e*) (*P e*)›
by (*fact ev-trans-Mprefix*)

lemma ‹$e \in A \Longrightarrow \sqcap a \in A \rightarrow P\ a\ {}_T\!\rightsquigarrow (ev\ e)\ (P\ e)$›
by (*fact ev-trans-Mndetprefix*)

lemma ‹$e \rightarrow P\ {}_T\!\rightsquigarrow\ (ev\ e)\ P$›
by (*fact ev-trans-prefix*)

$P \sqcap Q$ laws

lemma ‹$P \sqcap Q\ {}_T\!\rightsquigarrow_\tau\ P$›
and ‹$P \sqcap Q\ {}_T\!\rightsquigarrow_\tau\ Q$›
by (*fact τ-trans-NdetL τ-trans-NdetR*)+

lemma ‹$a \in A \Longrightarrow (\sqcap a \in A.\ P\ a)\ {}_T\!\rightsquigarrow_\tau\ P\ a$›
by (*fact τ-trans-GlobalNdet*)

lemma ‹*finite* $A \Longrightarrow a \in A \Longrightarrow (\bigsqcap a \in A.\ P\ a)\ {}_T\!\rightsquigarrow_\tau\ P\ a$›
by (*fact τ-trans-MultiNdet*)

$\mu\ x.\ f\ x$ law

lemma ‹*cont* $f \Longrightarrow P = (\mu\ X.\ f\ X) \Longrightarrow P\ {}_T\!\rightsquigarrow_\tau\ f\ P$›
by (*fact fix-point-τ-trans*)

$P \Box Q$ laws

lemma *τ-trans-DetL*: ‹$P\ {}_T\!\rightsquigarrow_\tau\ P' \Longrightarrow P \Box Q\ {}_T\!\rightsquigarrow_\tau\ P' \Box Q$ ›
and *τ-trans-DetR*: ‹$Q\ {}_T\!\rightsquigarrow_\tau\ Q' \Longrightarrow P \Box Q\ {}_T\!\rightsquigarrow_\tau\ P\ \Box\ Q'$›
by *simp-all*

lemma *τ-trans-MultiDet*:
‹*finite* $A \Longrightarrow \forall\ a \in A.\ P\ a\ {}_T\!\rightsquigarrow_\tau\ P'\ a \Longrightarrow (\Box a \in A.\ P\ a)\ {}_T\!\rightsquigarrow_\tau\ (\Box a \in A.\ P'\ a)$›
by (*fact mono-MultiDet-T*)

lemma ‹$P\ {}_T\!\rightsquigarrow e\ P' \Longrightarrow P \Box Q\ {}_T\!\rightsquigarrow e\ P'$›
and ‹$Q\ {}_T\!\rightsquigarrow e\ Q' \Longrightarrow P \Box Q\ {}_T\!\rightsquigarrow e\ Q'$›
by (*fact event-trans-DetL event-trans-DetR*)+

lemma ‹*finite* $A \Longrightarrow a \in A \Longrightarrow P\ a\ {}_T\!\rightsquigarrow e\ Q \Longrightarrow (\Box a \in A.\ P\ a)\ {}_T\!\rightsquigarrow e\ Q$›
by (*fact event-trans-MultiDet*)

$P\ ;\ Q$ laws

lemma *τ-trans-SeqR*: ‹$\exists\ P'.\ P\ {}_T\!\rightsquigarrow\checkmark\ P' \Longrightarrow P\ ;\ Q\ {}_T\!\rightsquigarrow_\tau\ Q$›
by (*metis D-SKIP D-STOP SKIP-Seq divergence-refine-def leDT-imp-leT*
leD-STOP leD-leT-imp-leDT mono-Seq-DT-left ready-tick-imp-τ-trans-SKIP)

lemma ‹$Q\ {}_T\!\rightsquigarrow_\tau\ Q' \Longrightarrow P\ ;\ Q\ {}_T\!\rightsquigarrow_\tau\ P\ ;\ Q'$›
by (*fact mono-Seq-T-right*)

lemma ‹✓ ∈ *ready-set P* ⟹ *Q* $_T\!\rightsquigarrow$(*ev e*) *Q′* ⟹ *P* ; *Q* $_T\!\rightsquigarrow$(*ev e*) *Q′*›

by (*fact ev-trans-SeqR*)

P \ *B* laws

lemma *τ-trans-Hiding*: ‹*P* $_T\!\rightsquigarrow_\tau$ *P′* ⟹ *P* \ *B* $_T\!\rightsquigarrow_\tau$ *P′* \ *B*›
by (*fact mono-Hiding-T*)

lemma *ev-trans-Hiding-notin*:
‹*P* $_T\!\rightsquigarrow$(*ev e*) *P′* ⟹ *e* ∉ *B* ⟹ *P* \ *B* $_T\!\rightsquigarrow$(*ev e*) *P′* \ *B*›
by (*metis AfterExt-def After-Hiding-T-Hiding-After-if-ready-notin mono-Hiding-T*

event-trans-τ-trans event.simps(*4*) *ready-notin-imp-ready-Hiding*)

lemma ‹*P* $_T\!\rightsquigarrow$✓ *P′* ⟹ *P* \ *B* $_T\!\rightsquigarrow$✓ *STOP*›
by (*fact tick-trans-Hiding*)

lemma *ev-trans-Hiding-inside*:
‹*P* $_T\!\rightsquigarrow$(*ev e*) *P′* ⟹ *e* ∈ *B* ⟹ *P* \ *B* $_T\!\rightsquigarrow_\tau$ *P′* \ *B*›
by (*metis AfterExt-def Hiding-T-Hiding-After-if-ready-inside*
mono-Hiding-T event.simps(*4*) *trans-T*)

Renaming P f laws

lemma *tick-trans-Renaming*: ‹*P* $_T\!\rightsquigarrow$✓ *P′* ⟹ *Renaming P f* $_T\!\rightsquigarrow$✓ *STOP*›
by (*simp add*: *AfterExt-def ready-set-Renaming tick-eq-EvExt*)

lemma ‹*P* $_T\!\rightsquigarrow$✓ *P′* ⟹ *P* ⟦*a* := *b*⟧ $_T\!\rightsquigarrow$✓ *STOP*›
by (*fact tick-trans-Renaming*)

P ⟦*S*⟧ *Q* laws

From here we slightly defer from Roscoe's laws for *Sync*: we obtain the following rules for *SKIP* instead of *STOP*.

lemma *tick-trans-SyncL*: ‹*P* $_T\!\rightsquigarrow$✓ *P′* ⟹ *P* ⟦*S*⟧ *Q* $_T\!\rightsquigarrow_\tau$ *SKIP* ⟦*S*⟧ *Q*›
and *tick-trans-SyncR*: ‹*Q* $_T\!\rightsquigarrow$✓ *Q′* ⟹ *P* ⟦*S*⟧ *Q* $_T\!\rightsquigarrow_\tau$ *P* ⟦*S*⟧ *SKIP*›
by (*simp-all add*: *D-SKIP divergence-refine-def leDT-imp-leT leD-leT-imp-leDT ready-tick-imp-τ-trans-SKIP*)

lemma *tick-trans-SKIP-Sync-SKIP*: ‹*SKIP* ⟦*S*⟧ *SKIP* $_T\!\rightsquigarrow$✓ *STOP*›
by (*simp add*: *SKIP-trans-tick Sync-SKIP-SKIP*)

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

$P \rhd Q$ laws

lemma *Sliding-τ-trans-left*: $\langle P \ {}_T\!\rightsquigarrow_\tau \ P' \Longrightarrow P \rhd Q \ {}_T\!\rightsquigarrow_\tau \ P' \rhd Q\rangle$
unfolding *Sliding-def* **by** *simp*

lemma $\langle P \ {}_T\!\rightsquigarrow e \ P' \Longrightarrow P \rhd Q \ {}_T\!\rightsquigarrow e \ P'\rangle$
by (*fact Sliding-event-transL*)

lemma $\langle P \rhd Q \ {}_T\!\rightsquigarrow_\tau \ Q\rangle$
by (*fact Sliding-τ-transR*)

$P \triangle Q$ laws

lemma *Interrupt-τ-trans-left*: $\langle P \ {}_T\!\rightsquigarrow_\tau \ P' \Longrightarrow P \triangle Q \ {}_T\!\rightsquigarrow_\tau \ P' \triangle Q\rangle$
by (*simp add*: *mono-Interrupt-T*)

lemma *Interrupt-τ-trans-right*: $\langle Q \ {}_T\!\rightsquigarrow_\tau \ Q' \Longrightarrow P \triangle Q \ {}_T\!\rightsquigarrow_\tau \ P \triangle Q'\rangle$
by (*simp add*: *mono-Interrupt-T*)

lemma *Interrupt-ev-trans-left*: $\langle P \ {}_T\!\rightsquigarrow(ev\ e) \ P' \Longrightarrow P \triangle Q \ {}_T\!\rightsquigarrow(ev\ e) \ P' \triangle Q\rangle$
by (*simp add*: *AfterExt-def After-Interrupt Interrupt-τ-trans-left ready-set-Interrupt*)

lemma *Interrupt-ev-trans-right*: $\langle Q \ {}_T\!\rightsquigarrow(ev\ e) \ Q' \Longrightarrow P \triangle Q \ {}_T\!\rightsquigarrow(ev\ e) \ Q'\rangle$
by (*simp add*: *AfterExt-def After-Interrupt ready-set-Interrupt*)

Throw P A Q laws

lemma *Throw-τ-trans-right*:
$\langle \forall\, a \in A.\ Q\ a \ {}_T\!\rightsquigarrow_\tau \ Q'\ a \Longrightarrow P \ \Theta\ a \in A.\ Q\ a \ {}_T\!\rightsquigarrow_\tau \ P \ \Theta\ a \in A.\ Q'\ a\rangle$
by (*fact mono-right-Throw-T*)

lemma *Throw-ev-trans-right*:
$\langle P \ {}_T\!\rightsquigarrow(ev\ e) \ P' \Longrightarrow e \in A \Longrightarrow P \ \Theta\ a \in A.\ Q\ a \ {}_T\!\rightsquigarrow(ev\ e) \ (Q\ e)\rangle$
by (*simp add*: *AfterExt-Throw ready-set-Throw split*: *event.split*)

lemma $\langle tickFree\ s \Longrightarrow \bot \ {}_T\!\rightsquigarrow^* s \ P\rangle$
by (*fact BOT-trace-trans-tickFree-anything*)

15.2 Reality Checks

lemma $\langle STOP \ {}_T\!\rightsquigarrow^* s \ P \longleftrightarrow s = [] \wedge P = STOP\rangle$
by (*fact STOP-trace-trans-iff*)

lemma *T-iff-exists-trans* : $\langle s \in \mathcal{T}\ P \longleftrightarrow (\exists\, P'.\ P \ {}_T\!\rightsquigarrow^* s \ P')\rangle$

by (*meson T-imp-exists-trace-trans trace-trans-imp-T-if-τ-trans-imp-leT*)

15.3 Other Results

lemma *trace-trans-ready-set-subset-ready-set-AfterTrace*:
‹$P\ {}_T\!\rightsquigarrow^*s\ Q \Longrightarrow$ *ready-set* $Q \subseteq$ *ready-set* $(P$ *afterTrace* $s)$›
by (*metis T-iff-exists-trans T-imp-trace-trans-iff-AfterTrace-τ-trans τ-trans-anti-mono-ready-set*)

lemma *trace-trans-imp-ready-set*:
‹$P\ {}_T\!\rightsquigarrow^*(s\ @\ e\ \#\ t)\ Q \Longrightarrow e \in$ *ready-set* $(P$ *afterTrace* $s)$›
using *T-iff-exists-trans ready-set-AfterTrace* **by** *blast*

lemma *AfterTrace-τ-trans-if-τ-trans-imp-leT* :
‹$(P\ {}_T\!\rightsquigarrow^*s\ Q) \longleftrightarrow s \in \mathcal{T}\ P \wedge P$ *afterTrace* $s\ {}_T\!\rightsquigarrow_\tau\ Q$›
using *T-iff-exists-trans T-imp-trace-trans-iff-AfterTrace-τ-trans* **by** *blast*

15.4 Summary: Operational Rules

In this section, we will just write down the operational laws that we have proven in a fancy way.

Absorbance rules

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

SKIP **rule**

$$\frac{}{SKIP\ {}_T\!\rightsquigarrow\checkmark\ STOP}$$

$e \rightarrow P$ **rules**

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

$$\frac{}{?e \rightarrow\ ?P\ {}_T\!\rightsquigarrow ev\ ?e\ ?P}$$

($\sqcap$) **rules**

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

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

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

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

($\Box$) **rules**

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

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

(;) **rule**

$$\frac{\exists\, P'.\ ?P \ {}_T\!\rightsquigarrow\!\checkmark \ P'}{?P\ ;\ ?Q \ {}_T\!\rightsquigarrow_\tau \ ?Q}$$

Hiding **rules**

$$\frac{?P \ {}_T\!\rightsquigarrow_\tau \ ?P'}{?P \setminus ?B \ {}_T\!\rightsquigarrow_\tau \ ?P' \setminus ?B} \qquad \frac{?P \ {}_T\!\rightsquigarrow\!\checkmark \ ?P'}{?P \setminus ?B \ {}_T\!\rightsquigarrow\!\checkmark \ STOP}$$

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

Renaming **rule**

$$\frac{?P \ {}_T\!\rightsquigarrow\!\checkmark \ ?P'}{Renaming\ ?P\ ?f \ {}_T\!\rightsquigarrow\!\checkmark \ STOP}$$

Sync **rules**

$$\frac{?P\ {}_T{\rightsquigarrow}\checkmark\ ?P'}{?P\ [\![?S]\!]\ ?Q\ {}_T{\rightsquigarrow}_\tau\ SKIP\ [\![?S]\!]\ ?Q} \qquad \frac{?Q\ {}_T{\rightsquigarrow}\checkmark\ ?Q'}{?P\ [\![?S]\!]\ ?Q\ {}_T{\rightsquigarrow}_\tau\ ?P\ [\![?S]\!]\ SKIP}$$

$$\frac{}{SKIP\ [\![?S]\!]\ SKIP\ {}_T{\rightsquigarrow}_\tau\ SKIP}$$

$(\triangleright)$ **rules**

$$\frac{?P\ {}_T{\rightsquigarrow}_\tau\ ?P'}{?P \triangleright ?Q\ {}_T{\rightsquigarrow}_\tau\ ?P' \triangleright ?Q} \qquad \frac{?P\ {}_T{\rightsquigarrow}?e\ ?P'}{?P \triangleright ?Q\ {}_T{\rightsquigarrow}?e\ ?P'}$$

$$\frac{}{?P \triangleright ?Q\ {}_T{\rightsquigarrow}_\tau\ ?Q}$$

$(\triangle)$ **rules**

$$\frac{?P\ {}_T{\rightsquigarrow}_\tau\ ?P'}{?P \triangle ?Q\ {}_T{\rightsquigarrow}_\tau\ ?P' \triangle ?Q} \qquad \frac{?Q\ {}_T{\rightsquigarrow}_\tau\ ?Q'}{?P \triangle ?Q\ {}_T{\rightsquigarrow}_\tau\ ?P \triangle ?Q'}$$

$$\frac{?P\ {}_T{\rightsquigarrow}ev\ ?e\ ?P'}{?P \triangle ?Q\ {}_T{\rightsquigarrow}ev\ ?e\ ?P' \triangle ?Q} \qquad \frac{?Q\ {}_T{\rightsquigarrow}ev\ ?e\ ?Q'}{?P \triangle ?Q\ {}_T{\rightsquigarrow}ev\ ?e\ ?Q'}$$

Throw **rules**

$$\frac{\forall\, a{\in}?A.\ ?Q\ a\ {}_T{\rightsquigarrow}_\tau\ ?Q'\ a}{?P\ \Theta\ a{\in}?A.\ ?Q\ a\ {}_T{\rightsquigarrow}_\tau\ ?P\ \Theta\ a{\in}?A.\ ?Q'\ a}$$

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

end

Chapter 16

Bonus: powerful new Laws

theory *NewLaws*
imports *Interrupt Sliding Throw*
begin

16.1 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.

16.1.1 Some Generalizations

For *Renaming*, we can obtain generalizations of the following results:

Renaming $(\Box a{\in}A \rightarrow P\ (f\ a))\ f = \Box b{\in}f\ `\ A \rightarrow Renaming\ (P\ b)\ f$

Renaming $(\sqcap a{\in}A \rightarrow\ \ P\ (f\ a))\ f = \sqcap b{\in}f\ `\ A \rightarrow\ \ Renaming\ (P\ b)\ f$

lemma *Renaming-Mprefix*:
‹*Renaming* $(\Box a \in A \rightarrow P\ a)\ f =$
$\Box y \in f\ `\ A \rightarrow \sqcap a \in \{x \in A.\ y = f\ x\}.$ *Renaming* $(P\ a)\ f$› (**is** ‹*?lhs* = *?rhs*›)
proof (*subst Process-eq-spec-optimized*, *safe*)
show ‹$s \in \mathcal{D}$ *?lhs* $\Longrightarrow$ $s \in \mathcal{D}$ *?rhs*› **for** *s*
by (*auto simp add*: *D-Renaming D-Mprefix D-GlobalNdet hd-map-EvExt*)
(*use list.map-sel*(*2*) *tickFree-tl* **in** *blast*)
next
fix *s*
assume ‹$s \in \mathcal{D}$ *?rhs*›
then obtain *a s'* **where** $*$: ‹$a \in A$› ‹$s = EvExt\ f\ (ev\ a)\ \#\ s'$›
‹$s' \in \mathcal{D}\ (Renaming\ (P\ a)\ f)$›
by (*simp add*: *D-Mprefix EvExt-def D-GlobalNdet split*: *if-split-asm*)
(*metis list.collapse*)
from $*$(*3*) **obtain** *s1 s2*

```
    where ** : ‹tickFree s1› ‹front-tickFree s2›
          ‹s' = map (EvExt f) s1 @ s2› ‹s1 ∈ 𝒟 (P a)›
    by (simp add: D-Renaming) blast
  have *** : ‹tickFree (ev a # s1) ∧
        s = EvExt f (ev a) # map (EvExt f) s1 @ s2 ∧ ev a # s1 ∈ 𝒟 (Mprefix
A P)›
    by (simp add: D-Mprefix *(1, 2) **(1, 3, 4))
  show ‹s ∈ 𝒟 ?lhs›
    by (simp add: D-Renaming )
      (metis **(2) *** append-Cons list.simps(9))
next
  fix s X
  assume same-div: ‹𝒟 ?lhs = 𝒟 ?rhs›
  assume ‹(s, X) ∈ ℱ ?lhs›
  then consider ‹s ∈ 𝒟 ?lhs›
    | ‹∃ t. (t, EvExt f −‘ X) ∈ ℱ (Mprefix A P) ∧ s = map (EvExt f) t›
    by (simp add: F-Renaming D-Renaming) blast
  thus ‹(s, X) ∈ ℱ ?rhs›
  proof cases
    from D-F same-div show ‹s ∈ 𝒟 ?lhs ⟹ (s, X) ∈ ℱ ?rhs› by blast
  next
    assume ‹∃ t. (t, EvExt f −‘ X) ∈ ℱ (Mprefix A P) ∧ s = map (EvExt f) t›
    then obtain t where * : ‹(t, EvExt f −‘ X) ∈ ℱ (Mprefix A P)›
                            ‹s = map (EvExt f) t› by blast
    show ‹(s, X) ∈ ℱ ?rhs›
    proof (cases ‹t = []›)
      from * show ‹t = [] ⟹ (s, X) ∈ ℱ ?rhs›
        by (simp add: F-Mprefix disjoint-iff-not-equal)
          (metis ev-elem-anteced1)
    next
      assume ‹t ≠ []›
      with *(1) obtain a t'
        where ** : ‹a ∈ A› ‹t = ev a # t'› ‹(t', EvExt f −‘ X) ∈ ℱ (P a)›
        by (simp add: F-Mprefix) (metis event.inject imageE list.exhaust-sel)
      have *** : ‹s ≠ [] ∧ hd s ∈ ev ‘ f ‘ A›
        using *(2) **(1, 2) hd-map-EvExt by fastforce
      with ** have ‹ev (f a) = hd s ∧
                    (tl s, X) ∈ ℱ (⊓a ∈ {x ∈ A. f a = f x}. Renaming (P a) f)›
        by (simp add: F-GlobalNdet *(2) F-Renaming)
          (metis (no-types, opaque-lifting) ‹t ≠ []› hd-map hd-map-EvExt list.sel(1))
      with *** show ‹(s, X) ∈ ℱ ?rhs› by (simp add: F-Mprefix) blast
    qed
  qed
next
  show ‹(s, X) ∈ ℱ ?rhs ⟹ (s, X) ∈ ℱ ?lhs› for s X
  proof (cases s)
    show ‹s = [] ⟹ (s, X) ∈ ℱ ?rhs ⟹ (s, X) ∈ ℱ ?lhs›
      by (simp add: F-Mprefix F-Renaming disjoint-iff-not-equal)
        (metis ev-elem-anteced1)
```

```
  next
    fix b s'
    assume ‹s = b # s'› ‹(s, X) ∈ F ?rhs›
    then obtain a
      where * : ‹a ∈ A› ‹s = (EvExt f) (ev a) # s'›
              ‹(s', X) ∈ F (⊓a ∈ {x ∈ A. f a = f x}. Renaming (P a) f)›
      by (auto simp add: F-Mprefix EvExt-def)
    from *(1, 3) obtain a'
      where ** : ‹a' ∈ A› ‹f a' = f a› ‹(s', X) ∈ F (Renaming (P a') f)›
      by (auto simp add: F-GlobalNdet split: if-split-asm)
    from **(3) consider
      ‹∃ t. (t, EvExt f -‘ X) ∈ F (P a') ∧ s' = map (EvExt f) t›
      | ‹∃ s1 s2. tickFree s1 ∧ front-tickFree s2 ∧
                  s' = map (EvExt f) s1 @ s2 ∧ s1 ∈ D (P a')›
      by (simp add: F-Renaming) blast
    thus ‹(s, X) ∈ F ?lhs›
    proof cases
      assume ‹∃ t. (t, EvExt f -‘ X) ∈ F (P a') ∧ s' = map (EvExt f) t›
      then obtain t where *** : ‹(t, EvExt f -‘ X) ∈ F (P a')›
                               ‹s' = map (EvExt f) t› by blast
      have **** : ‹(ev a' # t, EvExt f -‘ X) ∈ F (Mprefix A P) ∧
                   s = map (EvExt f) (ev a' # t)›
        by (simp add: F-Mprefix *(2) **(1) ***(1, 2))
           (metis **(2) ev-elem-anteced1 vimage-singleton-eq)
      show ‹(s, X) ∈ F ?lhs›
        apply (simp add: F-Renaming)
        using **** by blast
    next
      assume ‹∃ s1 s2. tickFree s1 ∧ front-tickFree s2 ∧
                       s' = map (EvExt f) s1 @ s2 ∧ s1 ∈ D (P a')›
      then obtain s1 s2
        where *** : ‹tickFree s1› ‹front-tickFree s2›
                    ‹s' = map (EvExt f) s1 @ s2› ‹s1 ∈ D (P a')› by blast
      have ‹tickFree (ev a' # s1) ∧
            s = EvExt f (ev a') # map (EvExt f) s1 @ s2 ∧
            ev a' # s1 ∈ D (Mprefix A P)›
        by (simp add: *(2) **(1) *** D-Mprefix)
           (metis **(2) ev-elem-anteced1 vimage-singleton-eq)
      thus ‹(s, X) ∈ F ?lhs› by (auto simp add: ***(2) F-Renaming)
    qed
  qed
qed


lemma Renaming-Mprefix-Sliding:
  ‹Renaming ((□a ∈ A → P a) ▷ Q) f =
   (□y ∈ f ‘ A → ⊓a ∈ {x ∈ A. y = f x}. Renaming (P a) f) ▷ Renaming Q f›
  unfolding Sliding-def
  by (simp add: Renaming-Ndet Renaming-Det Renaming-Mprefix)
```

lemma *Renaming-GlobalNdet*:
‹*Renaming* (⊓*a* ∈ *A*. *P a*) *f* = ⊓*b* ∈ *f* ‘ *A*. ⊓*a* ∈ {*a* ∈ *A*. *b* = *f a*}. *Renaming* (*P a*) *f*›
by (*simp add*: *Process-eq-spec F-Renaming F-GlobalNdet D-Renaming D-GlobalNdet*) *blast*

lemma *Renaming-Mndetprefix*:
‹*Renaming* (⊓*a* ∈ *A* → *P a*) *f* =
⊓*y* ∈ *f* ‘ *A* → ⊓*a* ∈ {*x* ∈ *A*. *y* = *f x*}. *Renaming* (*P a*) *f*› (**is** ‹*?lhs* = *?rhs*›)
apply (*subst* (*1 2*) *Mndetprefix-GlobalNdet*)
apply (*simp add*: *Renaming-GlobalNdet Renaming-prefix*)
apply (*intro mono-GlobalNdet-eq ballI*)
apply (*subst write0-GlobalNdet*[*symmetric*], *blast*)
by (*simp add*: *mono-GlobalNdet-eq*)

16.1.2 *Renaming* and *Hiding*

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* ⟹ *strict-mono* (λ*i*. *map f* (*g i*))›
unfolding *strict-mono-def less-list-def le-list-def*
by (*metis list.map-disc-iff map-append self-append-conv*)

lemma *trace-hide-map-EvExt* :
‹*inj-on* (*EvExt f*) (*set s* ∪ *ev* ‘ *S*) ⟹
trace-hide (*map* (*EvExt f*) *s*) (*ev* ‘ *f* ‘ *S*) =
map (*EvExt f*) (*trace-hide s* (*ev* ‘ *S*))›
proof (*induct s*)
case *Nil*
show *?case* **by** *simp*
next
case (*Cons a s*)
hence ∗ : ‹*trace-hide* (*map* (*EvExt f*) *s*) (*ev* ‘ *f* ‘ *S*) =
map (*EvExt f*) (*trace-hide s* (*ev* ‘ *S*))› **by** *fastforce*
from *Cons.prems*[*unfolded inj-on-def*, *rule-format*, *of a*, *simplified*] **show** *?case*
apply (*simp add*: ∗)
apply (*simp add*: *image-iff split*: *event.split*)
by (*metis ev-elem-anteced1 singletonI vimage-singleton-eq*)
qed

lemma *inj-on-EvExt-iff*: ‹*inj-on* (*EvExt f*) (*insert tick* (*ev* ‘ *S*)) ⟷ *inj-on f S*›
unfolding *inj-on-def EvExt-def* **by** (*auto split*: *event.split*)

lemma *inj-on-EvExt-set-T*:

```
  ‹inj-on f (events-of P) ⟹ s ∈ 𝒯 P ⟹ inj-on (EvExt f) (insert tick (set s))›
  unfolding inj-on-def EvExt-def events-of-def by (auto split: event.split)


theorem bij-Renaming-Hiding: ‹Renaming (P \ S) f = Renaming P f \ f ‘ S›
  (is ‹?lhs = ?rhs›) if bij-f: ‹bij f›
proof (subst Process-eq-spec-optimized, safe)
  fix s
  assume ‹s ∈ 𝒟 ?lhs›
  then obtain s1 s2 where ∗ : ‹tickFree s1› ‹front-tickFree s2›
                          ‹s = map (EvExt f) s1 @ s2› ‹s1 ∈ 𝒟 (P \ S)›
    by (simp add: D-Renaming) blast
  from ∗(4) obtain t u
    where ∗∗ : ‹front-tickFree u› ‹tickFree t› ‹s1 = trace-hide t (ev ‘ S) @ u›
               ‹t ∈ 𝒟 P ∨ (∃ g. isInfHiddenRun g P S ∧ t ∈ range g)›
    by (simp add: D-Hiding) blast
  from ∗∗(4) show ‹s ∈ 𝒟 ?rhs›
  proof (elim disjE)
    assume ‹t ∈ 𝒟 P›
    hence ‹front-tickFree (map (EvExt f) u @ s2) ∧ tickFree (map (EvExt f) t) ∧
          s = trace-hide (map (EvExt f) t) (ev ‘ f ‘ S) @ map (EvExt f) u @ s2 ∧
          map (EvExt f) t ∈ 𝒟 (Renaming P f)›
      apply (simp add: ∗(3) ∗∗(2, 3) EvExt-tF, intro conjI)
      subgoal using ∗(1, 2) ∗∗(3) EvExt-tF front-tickFree-append tickFree-append
by blast
      subgoal by (rule trace-hide-map-EvExt[symmetric];
            use bij-f in ‹unfold bij-def inj-on-def EvExt-def, auto split: event.split›)
      using ∗∗(2) D-Renaming front-tickFree-Nil by blast
    thus ‹s ∈ 𝒟 ?rhs› by (simp add: D-Hiding) blast
  next
    assume ‹∃ g. isInfHiddenRun g P S ∧ t ∈ range g›
    then obtain g where ‹isInfHiddenRun g P S› ‹t ∈ range g› by blast
    hence ‹front-tickFree (map (EvExt f) u @ s2) ∧
          tickFree (map (EvExt f) t) ∧
          s = trace-hide (map (EvExt f) t) (ev ‘ f ‘ S) @ map (EvExt f) u @ s2 ∧
          isInfHiddenRun (λi. map (EvExt f) (g i)) (Renaming P f) (f ‘ S) ∧
          map (EvExt f) t ∈ range (λi. map (EvExt f) (g i))›
      apply (simp add: ∗(3) ∗∗(2, 3) EvExt-tF, intro conjI)
      subgoal using ∗(1, 2) ∗∗(3) EvExt-tF front-tickFree-append tickFree-append
by blast
      subgoal by (rule trace-hide-map-EvExt[symmetric];
            use bij-f in ‹unfold bij-def inj-on-def EvExt-def, auto split: event.split›)
      subgoal using strict-mono-map by blast
      subgoal by (solves ‹auto simp add: T-Renaming›)
      subgoal apply (subst (1 2) trace-hide-map-EvExt)
        by (use bij-f in ‹unfold bij-def inj-on-def EvExt-def, auto split: event.split›)
metis
      subgoal by blast
      done
```

```
    thus ‹s ∈ 𝒟 ?rhs› by (simp add: D-Hiding) blast
  qed
next
  fix s
  assume ‹s ∈ 𝒟 ?rhs›
  then obtain t u
    where ∗ : ‹front-tickFree u› ‹tickFree t› ‹s = trace-hide t (ev ‘ f ‘ S) @ u›
            ‹t ∈ 𝒟 (Renaming P f) ∨
             (∃ g. isInfHiddenRun g (Renaming P f) (f ‘ S) ∧ t ∈ range g)›
    by (simp add: D-Hiding) blast
  from ∗(4) show ‹s ∈ 𝒟 ?lhs›
  proof (elim disjE)
    assume ‹t ∈ 𝒟 (Renaming P f)›
    then obtain t1 t2 where ∗∗ : ‹tickFree t1› ‹front-tickFree t2›
                                 ‹t = map (EvExt f) t1 @ t2› ‹t1 ∈ 𝒟 P›
      by (simp add: D-Renaming) blast
    have ‹tickFree (trace-hide t1 (ev ‘ S)) ∧
          front-tickFree (trace-hide t2 (ev ‘ f ‘ S) @ u) ∧
          trace-hide (map (EvExt f) t1) (ev ‘ f ‘ S) @ trace-hide t2 (ev ‘ f ‘ S) @ u
=
          map (EvExt f) (trace-hide t1 (ev ‘ S)) @ trace-hide t2 (ev ‘ f ‘ S) @ u ∧
          trace-hide t1 (ev ‘ S) ∈ 𝒟 (P \ S)›
      apply (simp, intro conjI)
      subgoal using ∗∗(1) Hiding-tickFree by blast
        subgoal using ∗(1, 2) ∗∗(3) Hiding-tickFree front-tickFree-append tick-
Free-append by blast
      subgoal by (rule trace-hide-map-EvExt;
            use bij-f in ‹unfold bij-def inj-on-def EvExt-def, simp split: event.split›)
      subgoal using ∗∗(4) elemDIselemHD by blast
      done
    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 (EvExt (inv f) ∘ EvExt f) v = v› for v
      by (induct v, unfold EvExt-def, auto split: event.split)
        (metis bij-inv-eq-iff bij-f)
    assume ‹∃ g. isInfHiddenRun g (Renaming P f) (f ‘ S) ∧ t ∈ range g›
    then obtain g
      where ∗∗∗ : ‹isInfHiddenRun g (Renaming P f) (f ‘ S)› ‹t ∈ range g› by
blast
    then consider ‹∃ t1. t1 ∈ 𝒯 P ∧ t = map (EvExt f) t1›
      | ‹∃ t1 t2. tickFree t1 ∧ front-tickFree t2 ∧
                  t = map (EvExt f) t1 @ t2 ∧ t1 ∈ 𝒟 P›
      by (simp add: T-Renaming) blast
    thus ‹s ∈ 𝒟 ?lhs›
    proof cases
      assume ‹∃ t1. t1 ∈ 𝒯 P ∧ t = map (EvExt f) t1›
```

then obtain *t1* **where** ∗∗∗∗ : ‹*t1* ∈ $\mathcal{T}$ *P*› ‹*t* = *map* (*EvExt f*) *t1*› **by** *blast*
have ∗∗∗∗∗ : ‹*t1* = *map* (*EvExt* (*inv f*)) *t*› **by** (*simp add*: ∗∗∗∗(*2*) ∗∗)
have ∗∗∗∗∗∗ : ‹*trace-hide t1* (*ev* ‘ *S*) = *trace-hide t1* (*ev* ‘ *S*) ∧
isInfHiddenRun (λ*i*. *map* (*EvExt* (*inv f*)) (*g i*)) *P S* ∧
t1 ∈ *range* (λ*i*. *map* (*EvExt* (*inv f*)) (*g i*))›
apply (*simp add*: ∗∗∗(*1*) *strict-mono-map*, *intro conjI*)
subgoal apply (*subst Renaming-inv*[**where** *f* = *f*, *symmetric*])
apply (*solves* ‹*simp add*: *bij-is-inj bij-f*›)
using ∗∗∗(*1*) **by** (*subst T-Renaming*, *blast*)
subgoal apply (*subst* (*1 2*) *inv-S*, *subst* (*1 2*) *trace-hide-map-EvExt*)
by (*use bij-f* **in** ‹*unfold bij-def inj-on-def EvExt-def*,
auto simp add: *inj-eq inj-inv-f split*: *event.split*›)
(*metis* ∗∗∗(*1*))
subgoal using ∗∗∗(*2*) ∗∗∗∗∗ **by** *blast*
done
have ‹*tickFree* (*trace-hide t1* (*ev* ‘ *S*)) ∧ *front-tickFree t1* ∧
trace-hide (*map* (*EvExt f*) *t1*) (*ev* ‘ *f* ‘ *S*) @ *u* =
map (*EvExt f*) (*trace-hide t1* (*ev* ‘ *S*)) @ *u* ∧
trace-hide t1 (*ev* ‘ *S*) ∈ $\mathcal{D}$ (*P* \ *S*)›
apply (*simp*, *intro conjI*)
subgoal using ∗(*2*) ∗∗∗∗(*2*) *EvExt-tF Hiding-tickFree* **by** *blast*

subgoal using ∗∗∗∗(*1*) *is-processT2-TR* **by** *blast*
apply (*rule trace-hide-map-EvExt*,
use bij-f **in** ‹*unfold bij-def inj-on-def EvExt-def*, *auto split*: *event.split*›)[*1*]
apply (*simp add*: *D-Renaming D-Hiding*)
using ∗(*2*) ∗∗∗∗∗ ∗∗∗∗∗∗ *EvExt-tF front-tickFree-Nil* **by** *blast*
with ∗(*1*) **show** ‹*s* ∈ $\mathcal{D}$ *?lhs*› **by** (*simp add*: *D-Renaming* ∗(*3*) ∗∗∗∗(*2*)) *blast*
next
assume ‹∃ *t1 t2*. *tickFree t1* ∧ *front-tickFree t2* ∧
t = *map* (*EvExt f*) *t1* @ *t2* ∧ *t1* ∈ $\mathcal{D}$ *P*›
then obtain *t1 t2*
where ∗∗∗∗ : ‹*tickFree t1*› ‹*front-tickFree t2*›
‹*t* = *map* (*EvExt f*) *t1* @ *t2*› ‹*t1* ∈ $\mathcal{D}$ *P*› **by** *blast*
have ‹*tickFree* (*trace-hide t1* (*ev* ‘ *S*)) ∧
front-tickFree (*trace-hide t2* (*ev* ‘ *f* ‘ *S*) @ *u*) ∧
trace-hide (*map* (*EvExt f*) *t1*) (*ev* ‘ *f* ‘ *S*) @ *trace-hide t2* (*ev* ‘ *f* ‘ *S*) @ *u* =
map (*EvExt f*) (*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*)
subgoal using ∗∗∗∗(*1*) *Hiding-tickFree* **by** *blast*
subgoal using ∗(*1*, *2*) ∗∗∗∗(*3*) *Hiding-tickFree front-tickFree-append tick-Free-append* **by** *blast*
subgoal by (*rule trace-hide-map-EvExt*;
use bij-f **in** ‹*unfold bij-def inj-on-def EvExt-def*, *simp split*: *event.split*›)
subgoal using ∗∗∗∗(*4*) *elemDIselemHD* **by** *blast*
done
thus ‹*s* ∈ $\mathcal{D}$ *?lhs*› **by** (*simp add*: *D-Renaming* ∗(*3*) ∗∗∗∗(*3*)) *blast*

```
    qed
  qed
next
  fix s X
  assume same-div : ‹𝒟 ?lhs = 𝒟 ?rhs›
  assume ‹(s, X) ∈ ℱ ?lhs›
  then consider ‹s ∈ 𝒟 ?lhs›
    | ‹∃ s1. (s1, EvExt f −‘ X) ∈ ℱ (P \ S) ∧ s = map (EvExt f) s1›
    by (simp add: F-Renaming D-Renaming) blast
  thus ‹(s, X) ∈ ℱ ?rhs›
  proof cases
    from D-F same-div show ‹s ∈ 𝒟 ?lhs ⟹ (s, X) ∈ ℱ ?rhs› by blast
  next
    assume ‹∃ s1. (s1, EvExt f −‘ X) ∈ ℱ (P \ S) ∧ s = map (EvExt f) s1›
    then obtain s1 where ∗ : ‹(s1, EvExt f −‘ X) ∈ ℱ (P \ S)›
                           ‹s = map (EvExt f) s1› by blast
    from this(1) consider ‹s1 ∈ 𝒟 (P \ S)›
      | ‹∃ t. s1 = trace-hide t (ev ‘ S) ∧ (t, EvExt f −‘ X ∪ ev ‘ S) ∈ ℱ P›
      by (simp add: F-Hiding D-Hiding) blast
    thus ‹(s, X) ∈ ℱ ?rhs›
    proof cases
      assume ‹s1 ∈ 𝒟 (P \ S)›
      then obtain t u
        where ∗∗ : ‹front-tickFree u› ‹tickFree t› ‹s1 = trace-hide t (ev ‘ S) @ u›
                   ‹t ∈ 𝒟 P ∨ (∃ g. isInfHiddenRun g P S ∧ t ∈ range g)›
        by (simp add: D-Hiding) blast
      have ∗∗∗ : ‹front-tickFree (map (EvExt f) u) ∧ tickFree (map (EvExt f) t) ∧
                  map (EvExt f) (trace-hide t (ev ‘ S)) @ map (EvExt f) u =
                  trace-hide (map (EvExt f) t) (ev ‘ f ‘ S) @ (map (EvExt f) u)›
        by (simp add: EvExt-ftF EvExt-tF ∗∗(1, 2))
           (rule trace-hide-map-EvExt[symmetric];
            use bij-f in ‹unfold bij-def inj-on-def EvExt-def, simp split: event.split›)
      from ∗∗(4) show ‹(s, X) ∈ ℱ ?rhs›
      proof (elim disjE)
        assume ‹t ∈ 𝒟 P›
        hence $ : ‹map (EvExt f) t ∈ 𝒟 (Renaming P f)›
          apply (simp add: D-Renaming)
          using ∗∗(2) front-tickFree-Nil by blast
        show ‹(s, X) ∈ ℱ ?rhs›
          by (simp add: F-Hiding) (metis $ ∗(2) ∗∗(3) ∗∗∗ map-append)
      next
        assume ‹∃ g. isInfHiddenRun g P S ∧ t ∈ range g›
        then obtain g where ‹isInfHiddenRun g P S› ‹t ∈ range g› by blast
        hence $ : ‹isInfHiddenRun (λi. map (EvExt f) (g i)) (Renaming P f) (f ‘
S) ∧
                   map (EvExt f) t ∈ range (λi. map (EvExt f) (g i))›
          apply (subst (1 2) trace-hide-map-EvExt,
                 (use bij-f in ‹unfold bij-def inj-on-def EvExt-def, auto split:
event.split›)[2])
```

```
      by (simp add: strict-mono-map T-Renaming image-iff) (metis (mono-tags,
lifting))
    show ‹(s, X) ∈ F ?rhs›
     by (simp add: F-Hiding)
       (metis (mono-tags, lifting) $ ∗(2) ∗∗(3) ∗∗∗ map-append)
  qed
 next
  assume ‹∃ t. s1 = trace-hide t (ev ‘ S) ∧ (t, EvExt f −‘ X ∪ ev ‘ S) ∈ F P›
  then obtain t where ∗∗ : ‹s1 = trace-hide t (ev ‘ S)›
                          ‹(t, EvExt f −‘ X ∪ ev ‘ S) ∈ F P› by blast
  have ∗∗∗ : ‹EvExt f −‘ X ∪ EvExt f −‘ ev ‘ f ‘ S = EvExt f −‘ X ∪ ev ‘ S›
   by (use bij-f in ‹unfold bij-def inj-on-def EvExt-def, auto simp add: image-iff
split: event.split›)
     (metis (no-types, opaque-lifting) event.distinct(1) event.exhaust event.simps(4,
5) o-apply)
  have ‹map (EvExt f) (trace-hide t (ev ‘ S)) =
       trace-hide (map (EvExt f) t) (ev ‘ f ‘ S) ∧
       (map (EvExt f) t, X ∪ ev ‘ f ‘ S) ∈ F (Renaming P f)›
    apply (intro conjI)
     apply (rule trace-hide-map-EvExt[symmetric];
       use bij-f in ‹unfold bij-def inj-on-def EvExt-def, simp split: event.split›)
    apply (simp add: F-Renaming)
    using ∗∗(2) ∗∗∗ by auto
  show ‹(s, X) ∈ F ?rhs›
    apply (simp add: F-Hiding ∗(2) ∗∗(1))
    using ‹?this› by blast
 qed
qed
next
 fix s X
 assume same-div : ‹D ?lhs = D ?rhs›
 assume ‹(s, X) ∈ F ?rhs›
 then consider ‹s ∈ D ?rhs›
  | ‹∃ t. s = trace-hide t (ev ‘ f ‘ S) ∧
       (t, X ∪ ev ‘ f ‘ S) ∈ F (Renaming P f)›
  by (simp add: F-Hiding D-Hiding) blast
 thus ‹(s, X) ∈ F ?lhs›
 proof cases
  from D-F same-div show ‹s ∈ D ?rhs ⟹ (s, X) ∈ F ?lhs› by blast
 next
  assume ‹∃ t. s = trace-hide t (ev ‘ f ‘ S) ∧
            (t, X ∪ ev ‘ f ‘ S) ∈ F (Renaming P f)›
  then obtain t
   where ∗ : ‹s = trace-hide t (ev ‘ f ‘ S)›
             ‹(t, X ∪ ev ‘ f ‘ S) ∈ F (Renaming P f)› by blast
  have ∗∗ : ‹EvExt f −‘ X ∪ EvExt f −‘ ev ‘ f ‘ S = EvExt f −‘ X ∪ ev ‘ S›
   by (use bij-f in ‹unfold bij-def inj-on-def EvExt-def, auto simp add: image-iff
split: event.split›)
     (metis (no-types, opaque-lifting) event.distinct(1) event.exhaust event.simps(4,
```

```
5) o-apply)
    have ‹(∃ s1. (s1, EvExt f −‘ X ∪ EvExt f −‘ ev ‘ f ‘ S) ∈ ℱ P ∧ t = map
(EvExt f) s1) ∨
          (∃ s1 s2. tickFree s1 ∧ front-tickFree s2 ∧ t = map (EvExt f) s1 @ s2 ∧
s1 ∈ 𝒟 P)›
      using ∗(2) by (simp add: F-Renaming)
    thus ‹(s, X) ∈ ℱ ?lhs›
    proof (elim disjE exE conjE)
      fix s1
     assume ‹(s1, EvExt f −‘ X ∪ EvExt f −‘ ev ‘ f ‘ S) ∈ ℱ P› ‹t = map (EvExt
f) s1›
      hence ‹(trace-hide s1 (ev ‘ S), EvExt f −‘ X) ∈ ℱ (P \ S) ∧
            s = map (EvExt f) (trace-hide s1 (ev ‘ S))›
        apply (simp add: ∗(1) F-Hiding ∗∗, intro conjI)
         apply blast
        by (rule trace-hide-map-EvExt;
            use bij-f in ‹unfold bij-def inj-on-def EvExt-def, simp split: event.split›)
      show ‹(s, X) ∈ ℱ ?lhs›
        apply (simp add: F-Renaming)
        using ‹?this› by blast
    next
      fix s1 s2
     assume ‹tickFree s1› ‹front-tickFree s2› ‹t = map (EvExt f) s1 @ s2› ‹s1 ∈
𝒟 P›
      hence ‹tickFree (trace-hide s1 (ev ‘ S)) ∧
             front-tickFree (trace-hide s2 (ev ‘ f ‘ S)) ∧
            s = map (EvExt f) (trace-hide s1 (ev ‘ S)) @ trace-hide s2 (ev ‘ f ‘ S) ∧
             trace-hide s1 (ev ‘ S) ∈ 𝒟 (P \ S)›
        apply (simp add: F-Renaming ∗(1), intro conjI)
        subgoal using Hiding-tickFree by blast
        subgoal using Hiding-fronttickFree by blast
         apply (rule trace-hide-map-EvExt;
              use bij-f in ‹unfold bij-def inj-on-def EvExt-def, simp split: event.split›)
        using elemDIselemHD by blast
      show ‹(s, X) ∈ ℱ ?lhs›
        apply (simp add: F-Renaming)
        using ‹?this› by blast
    qed
  qed
qed
```

16.1.3 *Renaming* and *Sync*

Idem for the synchronization: when f is one to one, *Renaming* $(P \llbracket S \rrbracket Q)$ will behave as expected.

```
lemma exists-map-antecedent-if-subset-range:
  ‹set u ⊆ range f ⟹ ∃ t. u = map f t›
  — In particular, when f is surjective or bijective.
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


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 = Renaming P f ⟦f ‘ S⟧ Renaming Q f›
  (is ‹?lhs P Q = ?rhs P Q›) if bij-f: ‹bij f›
proof −
  — Three intermediate results.
  have bij-EvExt-f : ‹bij (EvExt f)›
  proof (unfold bij-iff EvExt-def, intro allI ex1I)
    show ‹(case (EvExt (inv f)) e of ev x ⇒ (ev ∘ f) x | tick ⇒ tick) = e› for e
      by (simp add: EvExt-def split: event.split)
         (meson bij-inv-eq-iff bij-f)
  next
    show ‹(case e′ of ev x ⇒ (ev ∘ f) x | tick ⇒ tick) = e ⟹
          e′ = EvExt (inv f) e› for e e′
      by (simp add: EvExt-def split: event.splits)
         (metis bij-inv-eq-iff event.inject event.simps(3) bij-f)
  qed
  have EvExt-inv-f-is-inv-EvExt-f : ‹inv (EvExt f) = EvExt (inv f)›
  proof −
    have ‹EvExt (inv f) ∘ EvExt f = id›
      by (rule ext, simp add: EvExt-def split: event.split)
         (meson bij-betw-imp-inj-on inv-f-eq bij-f)
    thus ‹inv (EvExt f) = EvExt (inv f)›
      by (metis EvExt-comp EvExt-id bij-betw-def inv-unique-comp surj-iff that)
  qed
  have sets-S-eq : ‹(EvExt f) ‘ (insert tick (ev ‘ S)) = insert tick (ev ‘ f ‘ S)›
    unfolding EvExt-def image-def by auto

  show ‹?lhs P Q = ?rhs P Q›
  proof (subst Process-eq-spec-optimized, safe)
    fix s
    assume ‹s ∈ 𝒟 (?lhs P Q)›
    then obtain s1 s2 where ∗ : ‹tickFree s1› ‹front-tickFree s2›
                             ‹s = map (EvExt f) s1 @ s2› ‹s1 ∈ 𝒟 (P ⟦S⟧ Q)›
```

```
    by (simp add: D-Renaming) blast
  from *(4) obtain t u r v
    where ** : ‹front-tickFree v› ‹tickFree r ∨ v = []›
          ‹s1 = r @ v› ‹r setinterleaves ((t, u), insert tick (ev ‘ S))›
          ‹t ∈ 𝒟 P ∧ u ∈ 𝒯 Q ∨ t ∈ 𝒟 Q ∧ u ∈ 𝒯 P›
    by (simp add: D-Sync) blast
  { fix t u P Q
    assume assms : ‹r setinterleaves ((t, u), insert tick (ev ‘ S))›
                   ‹t ∈ 𝒟 P› ‹u ∈ 𝒯 Q›
    have ‹map (EvExt f) r setinterleaves
        ((map (EvExt f) t, map (EvExt f) u), insert tick (ev ‘ f ‘ S))›
      by (metis assms(1) bij-EvExt-f
          bij-map-setinterleaving-iff-setinterleaving sets-S-eq)
    moreover have ‹map (EvExt f) t ∈ 𝒟 (Renaming P f)›
      apply (cases ‹tickFree t›; simp add: D-Renaming)
      using assms(2) front-tickFree-Nil apply blast
      by (metis (no-types, lifting) assms(2) butlast-snoc front-tickFree-butlast
                front-tickFree-single map-EvExt-tick map-append
                nonTickFree-n-frontTickFree process-charn)
    moreover have ‹map (EvExt f) u ∈ 𝒯 (Renaming Q f)›
      using assms(3) by (simp add: T-Renaming) blast
    ultimately have ‹s ∈ 𝒟 (?rhs P Q)›
      by (simp add: D-Sync *(3) **(3))
        (metis *(1, 2) **(3) EvExt-tF front-tickFree-append tickFree-append)
  } note *** = this

  from **(4, 5) *** show ‹s ∈ 𝒟 (?rhs P Q)›
    apply (elim disjE)
    using **(4) *** apply blast
    using **(4) *** by (subst Sync-commute) blast
next
  fix s
  assume ‹s ∈ 𝒟 (?rhs P Q)›
  then obtain t u r v
    where * : ‹front-tickFree v› ‹tickFree r ∨ v = []› ‹s = r @ v›
              ‹r setinterleaves ((t, u), insert tick (ev ‘ f ‘ S))›
              ‹t ∈ 𝒟 (Renaming P f) ∧ u ∈ 𝒯 (Renaming Q f) ∨
               t ∈ 𝒟 (Renaming Q f) ∧ u ∈ 𝒯 (Renaming P f)›
    by (simp add: D-Sync) blast

  { fix t u P Q
    assume assms : ‹r setinterleaves ((t, u), insert tick (ev ‘ f ‘ S))›
                   ‹t ∈ 𝒟 (Renaming P f)› ‹u ∈ 𝒯 (Renaming Q f)›
    have ** : ‹(map (inv (EvExt f)) r) setinterleaves
                ((map (inv (EvExt f)) t, map (inv (EvExt f)) u), insert tick (ev ‘
S))›
      by (metis (no-types) assms(1) bij-EvExt-f bij-betw-imp-inj-on
                           bij-betw-inv-into  image-inv-f-f sets-S-eq
                           bij-map-setinterleaving-iff-setinterleaving)
```

have ‹*map* (*EvExt* (*inv f*)) *t* ∈ $\mathcal{D}$ (*Renaming* (*Renaming P f*) (*inv f*))›
by (*subst D-Renaming*, *simp*)
(*metis EvExt-ftF append.right-neutral assms*(*2*) *front-tickFree-implies-tickFree*

front-tickFree-single map-append nonTickFree-n-frontTickFree
process-charn)
hence ∗∗∗ : ‹*map* (*inv* (*EvExt f*)) *t* ∈ $\mathcal{D}$ *P*›
by (*simp add*: *Renaming-inv bij-is-inj bij-f EvExt-inv-f-is-inv-EvExt-f*)
have ‹*map* (*EvExt* (*inv f*)) *u* ∈ $\mathcal{T}$ (*Renaming* (*Renaming Q f*) (*inv f*))›
using *assms*(*3*) **by** (*subst T-Renaming*, *simp*) *blast*
hence ∗∗∗∗ : ‹*map* (*inv* (*EvExt f*)) *u* ∈ $\mathcal{T}$ *Q*›
by (*simp add*: *Renaming-inv bij-is-inj bij-f EvExt-inv-f-is-inv-EvExt-f*)
have ∗∗∗∗∗ : ‹*map* (*EvExt f* ∘ *inv* (*EvExt f*)) *r* = *r*›
by (*metis bij-EvExt-f bij-betw-imp-surj list.map-id surj-iff*)
have ‹*s* ∈ $\mathcal{D}$ (*?lhs P Q*)›
proof (*cases* ‹*tickFree r*›)
assume ‹*tickFree r*›
have $: ‹*r* @ *v* = *map* (*EvExt f*) (*map* (*inv* (*EvExt f*)) *r*) @ *v*›
by (*simp add*: ∗∗∗∗∗)
show ‹*s* ∈ $\mathcal{D}$ (*?lhs P Q*)›
apply (*simp add*: *D-Renaming D-Sync* ∗(*3*))
by (*metis* $ ∗(*1*) ∗∗ ∗∗∗ ∗∗∗∗ *EvExt-tF* ‹*tickFree r*›
append.right-neutral append-same-eq front-tickFree-Nil)
next
assume ‹¬ *tickFree r*›
then obtain *r′* **where** $: ‹*r* = *r′* @ [*tick*]› ‹*tickFree r′*›
by (*metis D-imp-front-tickFree assms front-tickFree-implies-tickFree ftf-Sync*
is-processT2-TR nonTickFree-n-frontTickFree)
then obtain *t′ u′*
where $$: ‹*t* = *t′* @ [*tick*]› ‹*u* = *u′* @ [*tick*]›
using *SyncWithTick-imp-NTF D-T assms* **by** *blast*
hence $$$: ‹(*map* (*inv* (*EvExt f*)) *r′*) *setinterleaves*
((*map* (*inv* (*EvExt f*)) *t′*, *map* (*inv* (*EvExt f*)) *u′*),
insert tick (*ev* ‘ *S*))›
by (*metis* $(*1*) *assms*(*1*) *bij-EvExt-f bij-betw-imp-inj-on*
bij-betw-inv-into bij-map-setinterleaving-iff-setinterleaving
butlast-snoc ftf-Sync32 image-inv-f-f sets-S-eq)
from ∗∗∗ $$(*1*) **have** ∗∗∗ : ‹*map* (*inv* (*EvExt f*)) *t′* ∈ $\mathcal{D}$ *P*›
by *simp* (*metis EvExt-inv-f-is-inv-EvExt-f is-processT9 tick-eq-EvExt*)
from ∗∗∗∗ $$(*2*) **have** ∗∗∗∗ : ‹*map* (*inv* (*EvExt f*)) *u′* ∈ $\mathcal{T}$ *Q*›
using *is-processT3-ST* **by** *simp blast*
have $$$$: ‹*r* = *map* (*EvExt f*) (*map* (*inv* (*EvExt f*)) *r′*) @ [*tick*]›
using $ ∗∗∗∗∗ **by** *auto*
show ‹*s* ∈ $\mathcal{D}$ (*?lhs P Q*)›
by (*simp add*: *D-Renaming D-Sync* ∗(*3*) $$$)
(*metis* $(*1*) $(*2*) $$$ $$$$ ∗(*2*) ∗∗∗ ∗∗∗∗ *EvExt-tF* ‹¬ *tickFree r*›
append.right-neutral append-same-eq front-tickFree-Nil front-tickFree-single)
qed
} **note** ∗∗ = *this*

show ‹$s \in \mathcal{D}$ (*?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 ‹$(s, X) \in \mathcal{F}$ (*?lhs P Q*)›
then consider ‹$s \in \mathcal{D}$ (*?lhs P Q*)›
| ‹$\exists$ *s1*. (*s1*, *EvExt f* −‘ *X*) $\in \mathcal{F}$ (*P* ⟦*S*⟧ *Q*) $\wedge$ *s* = *map* (*EvExt f*) *s1*›
by (*simp add*: *F-Renaming D-Renaming*) *blast*
thus ‹$(s, X) \in \mathcal{F}$ (*?rhs P Q*)›
proof *cases*
from *same-div D-F* **show** ‹$s \in \mathcal{D}$ (*?lhs P Q*) $\Longrightarrow$ $(s, X) \in \mathcal{F}$ (*?rhs P Q*)› **by** *blast*
next
assume ‹$\exists$ *s1*. (*s1*, *EvExt f* −‘ *X*) $\in \mathcal{F}$ (*P* ⟦*S*⟧ *Q*) $\wedge$ *s* = *map* (*EvExt f*) *s1*›
then obtain *s1* **where** $*$: ‹(*s1*, *EvExt f* −‘ *X*) $\in \mathcal{F}$ (*P* ⟦*S*⟧ *Q*)›
‹*s* = *map* (*EvExt f*) *s1*› **by** *blast*
from $*(1)$ **consider** ‹*s1* $\in \mathcal{D}$ (*P* ⟦*S*⟧ *Q*)›
| ‹$\exists$ *t-P t-Q X-P X-Q*.
(*t-P*, *X-P*) $\in \mathcal{F}$ *P* $\wedge$ (*t-Q*, *X-Q*) $\in \mathcal{F}$ *Q* $\wedge$
s1 setinterleaves ((*t-P*, *t-Q*), *insert tick* (*ev* ‘ *S*)) $\wedge$
EvExt f −‘ *X* = (*X-P* $\cup$ *X-Q*) $\cap$ *insert tick* (*ev* ‘ *S*) $\cup$ *X-P* $\cap$ *X-Q*›
by (*simp add*: *F-Sync D-Sync*) *blast*
thus ‹$(s, X) \in \mathcal{F}$ (*?rhs P Q*)›
proof *cases*
assume ‹*s1* $\in \mathcal{D}$ (*P* ⟦*S*⟧ *Q*)›
hence ‹$s \in \mathcal{D}$ (*?lhs P Q*)›
apply (*cases* ‹*tickFree s1*›; *simp add*: *D-Renaming* $*(2)$)
using *front-tickFree-Nil* **apply** *blast*
by (*metis* (*no-types*, *lifting*) *EvExt-ftF butlast-snoc front-tickFree-butlast front-tickFree-single map-butlast nonTickFree-n-frontTickFree process-charn*)
with *same-div D-F* **show** ‹$(s, X) \in \mathcal{F}$ (*?rhs P Q*)› **by** *blast*
next
assume ‹$\exists$ *t-P t-Q X-P X-Q*.
(*t-P*, *X-P*) $\in \mathcal{F}$ *P* $\wedge$ (*t-Q*, *X-Q*) $\in \mathcal{F}$ *Q* $\wedge$
s1 setinterleaves ((*t-P*, *t-Q*), *insert tick* (*ev* ‘ *S*)) $\wedge$
EvExt f −‘ *X* = (*X-P* $\cup$ *X-Q*) $\cap$ *insert tick* (*ev* ‘ *S*) $\cup$ *X-P* $\cap$ *X-Q*›
then obtain *t-P t-Q X-P X-Q*
where $**$: ‹(*t-P*, *X-P*) $\in \mathcal{F}$ *P*› ‹(*t-Q*, *X-Q*) $\in \mathcal{F}$ *Q*›
‹*s1 setinterleaves* ((*t-P*, *t-Q*), *insert tick* (*ev* ‘ *S*))›
‹*EvExt f* −‘ *X* = (*X-P* $\cup$ *X-Q*) $\cap$ *insert tick* (*ev* ‘ *S*) $\cup$ *X-P* $\cap$ *X-Q*› **by** *blast*
have ‹(*map* (*EvExt f*) *t-P*, (*EvExt f*) ‘ *X-P*) $\in \mathcal{F}$ (*Renaming P f*)›
by (*simp add*: *F-Renaming*)
(*metis* $**(1)$ *bij-EvExt-f bij-betw-imp-inj-on inj-vimage-image-eq*)
moreover have ‹(*map* (*EvExt f*) *t-Q*, (*EvExt f*) ‘ *X-Q*) $\in \mathcal{F}$ (*Renaming Q f*)›
by (*simp add*: *F-Renaming*)
(*metis* $**(2)$ *bij-EvExt-f bij-betw-imp-inj-on inj-vimage-image-eq*)

moreover have ‹*s setinterleaves* ((*map* (*EvExt f*) *t-P*, *map* (*EvExt f*) *t-Q*), *insert tick* (*ev* ‘ *f* ‘ *S*))›
by (*metis* ∗(*2*) ∗∗(*3*) *bij-EvExt-f sets-S-eq bij-map-setinterleaving-iff-setinterleaving*)
moreover have ‹*X* = ((*EvExt f*) ‘ *X-P* ∪ (*EvExt f*) ‘ *X-Q*) ∩ *insert tick* (*ev* ‘ *f* ‘ *S*) ∪ (*EvExt f*) ‘ *X-P* ∩ (*EvExt f*) ‘ *X-Q*›
by (*metis* ∗∗(*4*) *bij-EvExt-f bij-betw-def image-Int image-Un image-vimage-eq inf-top.right-neutral sets-S-eq*)
ultimately show ‹(*s*, *X*) ∈ *F* (*?rhs P Q*)›
by (*simp add*: *F-Sync*) *blast*
qed
qed
next
fix *s X*
assume *same-div* : ‹*D* (*?lhs P Q*) = *D* (*?rhs P Q*)›
assume ‹(*s*, *X*) ∈ *F* (*?rhs P Q*)›
then consider ‹*s* ∈ *D* (*?rhs P Q*)›
| ‹∃ *t-P t-Q X-P X-Q*.
(*t-P*, *X-P*) ∈ *F* (*Renaming P f*) ∧ (*t-Q*, *X-Q*) ∈ *F* (*Renaming Q f*) ∧
s setinterleaves ((*t-P*, *t-Q*), *insert tick* (*ev* ‘ *f* ‘ *S*)) ∧
X = (*X-P* ∪ *X-Q*) ∩ *insert tick* (*ev* ‘ *f* ‘ *S*) ∪ *X-P* ∩ *X-Q*›
by (*simp add*: *F-Sync D-Sync*) *blast*
thus ‹(*s*, *X*) ∈ *F* (*?lhs P Q*)›
proof *cases*
from *same-div D-F* **show** ‹*s* ∈ *D* (*?rhs P Q*) ⟹ (*s*, *X*) ∈ *F* (*?lhs P Q*)› **by** *blast*
next
assume ‹∃ *t-P t-Q X-P X-Q*.
(*t-P*, *X-P*) ∈ *F* (*Renaming P f*) ∧ (*t-Q*, *X-Q*) ∈ *F* (*Renaming Q f*) ∧
s setinterleaves ((*t-P*, *t-Q*), *insert tick* (*ev* ‘ *f* ‘ *S*)) ∧
X = (*X-P* ∪ *X-Q*) ∩ *insert tick* (*ev* ‘ *f* ‘ *S*) ∪ *X-P* ∩ *X-Q*›
then obtain *t-P t-Q X-P X-Q*
where ∗ : ‹(*t-P*, *X-P*) ∈ *F* (*Renaming P f*)› ‹(*t-Q*, *X-Q*) ∈ *F* (*Renaming Q f*)›
‹*s setinterleaves* ((*t-P*, *t-Q*), *insert tick* (*ev* ‘ *f* ‘ *S*))›
‹*X* = (*X-P* ∪ *X-Q*) ∩ *insert tick* (*ev* ‘ *f* ‘ *S*) ∪ *X-P* ∩ *X-Q*› **by** *blast*
from ∗(*1*, *2*) **consider** ‹*t-P* ∈ *D* (*Renaming P f*) ∨ *t-Q* ∈ *D* (*Renaming Q f*)›
| ‹∃ *t-P1 t-Q1*. (*t-P1*, *EvExt f* −‘ *X-P*) ∈ *F P* ∧ *t-P* = *map* (*EvExt f*) *t-P1* ∧ (*t-Q1*, *EvExt f* −‘ *X-Q*) ∈ *F Q* ∧ *t-Q* = *map* (*EvExt f*) *t-Q1*›
by (*simp add*: *F-Renaming D-Renaming*) *blast*
thus ‹(*s*, *X*) ∈ *F* (*?lhs P Q*)›
proof *cases*
assume ‹*t-P* ∈ *D* (*Renaming P f*) ∨ *t-Q* ∈ *D* (*Renaming Q f*)›
hence ‹*s* ∈ *D* (*?rhs P Q*)›
apply (*simp add*: *D-Sync*)
using ∗(*1*, *2*, *3*) *F-T Sync.sym front-tickFree-Nil* **by** *blast*

with *same-div D-F* **show** ‹$(s,\ X) \in \mathcal{F}\ (?lhs\ P\ Q)$› **by** *blast*
next
assume ‹$\exists$ *t-P1 t-Q1*. $(t\text{-}P1,\ EvExt\ f\ -`\ X\text{-}P) \in \mathcal{F}\ P \wedge t\text{-}P = map\ (EvExt\ f)\ t\text{-}P1 \wedge$
$(t\text{-}Q1,\ EvExt\ f\ -`\ X\text{-}Q) \in \mathcal{F}\ Q \wedge t\text{-}Q = map\ (EvExt\ f)\ t\text{-}Q1$›
then obtain *t-P1 t-Q1*
where ** : ‹$(t\text{-}P1,\ EvExt\ f\ -`\ X\text{-}P) \in \mathcal{F}\ P$› ‹$t\text{-}P = map\ (EvExt\ f)\ t\text{-}P1$›
‹$(t\text{-}Q1,\ EvExt\ f\ -`\ X\text{-}Q) \in \mathcal{F}\ Q$› ‹$t\text{-}Q = map\ (EvExt\ f)\ t\text{-}Q1$›
by *blast*
from **(*2*, *4*) **have** *** : ‹$t\text{-}P1 = map\ (inv\ (EvExt\ f))\ t\text{-}P$›
‹$t\text{-}Q1 = map\ (inv\ (EvExt\ f))\ t\text{-}Q$›
by (*simp*, *metis bij-EvExt-f bij-betw-imp-inj-on inv-o-cancel list.map-id*)+
have **** : ‹$map\ (EvExt\ f)\ (map\ (inv\ (EvExt\ f))\ s) = s$›
by (*metis bij-EvExt-f bij-betw-imp-surj list.map-comp list.map-id surj-iff*)
from *bij-map-setinterleaving-iff-setinterleaving*
[*of* ‹$inv\ (EvExt\ f)$› *s t-P* ‹$insert\ tick\ (ev\ `\ f\ `\ S)$› *t-Q*, *simplified* *(*3*)]
have ‹$map\ (inv\ (EvExt\ f))\ s\ setinterleaves\ ((t\text{-}P1,\ t\text{-}Q1),\ insert\ tick\ (ev\ `\ S))$›
by (*metis **** *bij-EvExt-f bij-betw-imp-inj-on*
bij-betw-inv-into image-inv-f-f sets-S-eq)
moreover have ‹$EvExt\ f\ -`\ X = (EvExt\ f\ -`\ X\text{-}P \cup EvExt\ f\ -`\ X\text{-}Q) \cap insert\ tick\ (ev\ `\ S) \cup$
$EvExt\ f\ -`\ X\text{-}P \cap EvExt\ f\ -`\ X\text{-}Q$›
by (*metis **(*4*) *bij-EvExt-f bij-betw-imp-inj-on*
inj-vimage-image-eq sets-S-eq vimage-Int vimage-Un)
ultimately show ‹$(s,\ X) \in \mathcal{F}\ (?lhs\ P\ Q)$›
by (*simp add*: *F-Renaming F-Sync*)
(*metis **(*1*, *3*) ****)
qed
qed
qed
qed

16.2 *Hiding* and *Mprefix*

We already have a way to distribute the *Hiding* operator on the *Mprefix* operator with $S \cap A = \{\} \Longrightarrow Mprefix\ A\ ?P \setminus S = \Box x{\in}A \rightarrow (?P\ x \setminus S)$. But this is only usable when $A \cap S = \{\}$. With the $(\triangleright)$ operator, we can now handle the case $A \cap S \neq \{\}$.

16.2.1 Two intermediate Results

lemma *D-Hiding-Mprefix-dir2*:
assumes ‹$s \in \mathcal{D}\ (\Box a \in A - S \rightarrow (P\ a \setminus S))$›
shows ‹$s \in \mathcal{D}\ (\Box a \in A \rightarrow P\ a \setminus S)$›
proof −
from *assms* **obtain** *a s′*

```
    where * : ‹a ∈ A› ‹a ∉ S› ‹s = ev a # s'› ‹s' ∈ 𝒟 (P a \ S)›
    by (auto simp add: D-Mprefix) (metis list.collapse)
  from *(4) obtain t u
    where ** : ‹front-tickFree u› ‹tickFree t› ‹s' = trace-hide t (ev ‘ S) @ u›
          ‹t ∈ 𝒟 (P a) ∨ (∃ f. isInfHiddenRun f (P a) S ∧ t ∈ range f)›
    by (simp add: D-Hiding) blast
  from **(4) show ‹s ∈ 𝒟 (□a ∈ A → P a \ S)›
  proof (elim disjE)
    assume ‹t ∈ 𝒟 (P a)›
    hence ‹tickFree (ev a # t) ∧ s = trace-hide (ev a # t) (ev ‘ S) @ u ∧
        ev a # t ∈ 𝒟 (Mprefix A P)›
      by (simp add: *(1, 2, 3) **(2, 3) image-iff[of ‹ev a›] D-Mprefix)
    show ‹s ∈ 𝒟 (Mprefix A P \ S)›
      apply (simp add: D-Hiding)
      using **(1) ‹?this› by blast
  next
    assume ‹∃f. isInfHiddenRun f (P a) S ∧ t ∈ range f›
    then obtain f where *** : ‹isInfHiddenRun f (P a) S› ‹t ∈ range f› by blast
    hence ‹tickFree (ev a # t) ∧ s = trace-hide (ev a # t) (ev ‘ S) @ u ∧
        isInfHiddenRun (λi. ev a # f i) (Mprefix A P) S ∧
        ev a # t ∈ range (λi. ev a # f i)›
      by (auto simp add: *(1, 2, 3) **(2, 3) image-iff[of ‹ev a›]
                         T-Mprefix less-cons strict-mono-Suc-iff)
    show ‹s ∈ 𝒟 (Mprefix A P \ S)›
      apply (simp add: D-Hiding)
      using **(1) ‹?this› by blast
  qed
qed


lemma F-Hiding-Mprefix-dir2:
  assumes ‹s ≠ []› and ‹(s, X) ∈ ℱ (□a ∈ A − S → (P a \ S))›
  shows ‹(s, X) ∈ ℱ (□a ∈ A → P a \ S)›
proof −
  from assms obtain a s'
    where * : ‹a ∈ A› ‹a ∉ S› ‹s = ev a # s'› ‹(s', X) ∈ ℱ (P a \ S)›
    by (auto simp add: F-Mprefix) (metis list.collapse)
  from *(4) consider ‹s' ∈ 𝒟 (P a \ S)›
    | ‹∃t. s' = trace-hide t (ev ‘ S) ∧ (t, X ∪ ev ‘ S) ∈ ℱ (P a)›
    by (simp add: F-Hiding D-Hiding) blast
  thus ‹(s, X) ∈ ℱ (Mprefix A P \ S)›
  proof cases
    assume ‹s' ∈ 𝒟 (P a \ S)›
    then obtain t u
      where ** : ‹front-tickFree u› ‹tickFree t›
            ‹s' = trace-hide t (ev ‘ S) @ u›
            ‹t ∈ 𝒟 (P a) ∨ (∃ f. isInfHiddenRun f (P a) S ∧ t ∈ range f)›
      by (simp add: D-Hiding) blast
    from **(4) show ‹(s, X) ∈ ℱ (Mprefix A P \ S)›
```

```
    proof (elim disjE)
      assume ‹t ∈ 𝒟 (P a)›
      hence ‹tickFree (ev a # t) ∧ s = trace-hide (ev a # t) (ev ‘ S) @ u ∧
            ev a # t ∈ 𝒟 (Mprefix A P)›
        by (simp add: ∗(1, 2, 3) ∗∗(2, 3) D-Mprefix image-iff[of ‹ev a›])
      show ‹(s, X) ∈ ℱ (Mprefix A P \ S)›
        apply (simp add: F-Hiding)
        using ∗∗(1) ‹?this› by blast
    next
      assume ‹∃f. isInfHiddenRun f (P a) S ∧ t ∈ range f›
      then obtain f where ‹isInfHiddenRun f (P a) S› ‹t ∈ range f› by blast
      hence ‹tickFree (ev a # t) ∧ s = trace-hide (ev a # t) (ev ‘ S) @ u ∧
            (isInfHiddenRun (λi. ev a # f i) (Mprefix A P) S ∧
            ev a # t ∈ range (λi. ev a # f i))›
       by (auto simp add: ∗(1, 2, 3) ∗∗(2, 3) less-cons monotone-on-def T-Mprefix)
      show ‹(s, X) ∈ ℱ (Mprefix A P \ S)›
        apply (simp add: F-Hiding)
        using ∗∗(1) ‹?this› by blast
    qed
  next
    assume ‹∃t. s' = trace-hide t (ev ‘ S) ∧ (t, X ∪ ev ‘ S) ∈ ℱ (P a)›
    then obtain t where ∗∗ : ‹s' = trace-hide t (ev ‘ S)›
                             ‹(t, X ∪ ev ‘ S) ∈ ℱ (P a)› by blast
    have ‹s = trace-hide (ev a # t) (ev ‘ S) ∧ (ev a # t, X ∪ ev ‘ S) ∈ ℱ (Mprefix
A P)›
      by (simp add: ∗(1, 2, 3) ∗∗ F-Mprefix image-iff)
    show ‹(s, X) ∈ ℱ (Mprefix A P \ S)›
      apply (simp add: F-Hiding)
      using ‹?this› by blast
  qed
qed
```

16.2.2 *Hiding* and *Mprefix* for disjoint Sets

Now we can give a more readable proof of the following result (already proven in *HOL−CSP.CSP-Laws*).

```
theorem Hiding-Mprefix-disjoint:
  ‹□a ∈ A → P a \ S = □a ∈ A → (P a \ S)›
  (is ‹?lhs = ?rhs›) if disjoint: ‹A ∩ S = {}›
proof (subst Process-eq-spec-optimized, safe)
  fix s
  assume ‹s ∈ 𝒟 ?lhs›
  then obtain t u
    where ∗ : ‹front-tickFree u› ‹tickFree t› ‹s = trace-hide t (ev ‘ S) @ u›
              ‹t ∈ 𝒟 (Mprefix A P) ∨
               (∃ f. isInfHiddenRun f (Mprefix A P) S ∧ t ∈ range f)›
  by (simp add: D-Hiding) blast
  from ∗(4) show ‹s ∈ 𝒟 ?rhs›
  proof (elim disjE)
```

```
    assume ‹t ∈ 𝒟 (Mprefix A P)›
    then obtain a t' where ** : ‹a ∈ A› ‹a ∉ S› ‹t = ev a # t'› ‹t' ∈ 𝒟 (P a)›
      by (auto simp add: D-Mprefix)
         (metis list.exhaust-sel disjoint-iff disjoint)
    have ‹front-tickFree u ∧ tickFree t' ∧
          trace-hide t' (ev ` S) @ u = trace-hide t' (ev ` S) @ u ∧ t' ∈ 𝒟 (P a)›
      apply (simp add: *(1) **(4))
      using *(2) **(3) tickFree-Cons by blast
    show ‹s ∈ 𝒟 ?rhs›
      apply (simp add: D-Mprefix *(3) **(1, 2, 3) image-iff[of ‹ev -›] D-Hiding)
      using ‹?this› by blast
  next
    assume ‹∃ f. isInfHiddenRun f (Mprefix A P) S ∧ t ∈ range f›
    then obtain f where ** : ‹isInfHiddenRun f (Mprefix A P) S›
                             ‹t ∈ range f› by blast
    from **(1) T-Mprefix obtain a
      where *** : ‹a ∈ A› ‹a ∉ S› ‹f (Suc 0) ≠ []› ‹hd (f (Suc 0)) = ev a›
      by (simp add: T-Mprefix)
         (metis disjoint-iff disjoint nil-less strict-mono-Suc-iff)
    from **(1)[THEN conjunct2, THEN conjunct2, rule-format, of 1]
         **(1)[simplified isInfHiddenRun-1] ***(1, 4) disjoint
    have **** : ‹f j ≠ [] ∧ hd (f j) = ev a› for j
      using **(1)[THEN conjunct2, THEN conjunct1, rule-format, of j]
      apply (cases ‹f 1›; simp add: T-Mprefix ***(3) split: if-split-asm)
      by (metis Nil-is-append-conv filter.simps(1)
                hd-append2 list.distinct(1) list.sel(1)) blast
    then obtain t' where ‹t = ev a # t'›
      by (metis **(2) list.exhaust-sel rangeE)
    hence ‹tickFree t' ∧ trace-hide t' (ev ` S) @ u = trace-hide t' (ev ` S) @ u ∧
          isInfHiddenRun (λi. tl (f i)) (P a) S ∧ t' ∈ range (λi. tl (f i))›
      apply (simp, intro conjI)
      subgoal using *(2) tickFree-Cons by blast
      subgoal by (meson **(1) **** less-tail strict-mono-Suc-iff)
      subgoal using **(1) by (simp add: T-Mprefix ****)
     subgoal by (metis (no-types, lifting) **(1) **** filter.simps(2) list.exhaust-sel
list.sel(3))
      by (metis (no-types, lifting) **(2) image-iff list.sel(3))
    show ‹s ∈ 𝒟 ?rhs›
      apply (simp add: D-Mprefix *(3) image-iff[of ‹ev a›] ***(1, 2) D-Hiding ‹t
= ev a # t'›)
      using *(1) ‹?this› by blast
  qed
next
  show ‹s ∈ 𝒟 ?rhs ⟹ s ∈ 𝒟 ?lhs› for s
    by (simp add: D-Hiding-Mprefix-dir2 Diff-triv that)
next
  fix s X
  assume same-div : ‹𝒟 ?lhs = 𝒟 ?rhs›
  assume ‹(s, X) ∈ ℱ ?lhs›
```

```
  then consider ‹s ∈ 𝒟 ?lhs›
    | ‹∃ 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 ?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)›
    then obtain t where ∗ : ‹s = trace-hide t (ev ‘ S)›
                            ‹(t, X ∪ ev ‘ S) ∈ ℱ (Mprefix A P)› by blast
    show ‹(s, X) ∈ ℱ ?rhs›
    proof (cases ‹t = []›)
      from ∗ show ‹t = [] ⟹ (s, X) ∈ ℱ ?rhs›
        by (auto simp add: F-Mprefix)
    next
      assume ‹t ≠ []›
      with ∗(2) disjoint obtain a t′
        where ∗∗ : ‹a ∈ A› ‹a ∉ S› ‹t = ev a # t′› ‹(t′, X ∪ ev ‘ S) ∈ ℱ (P a)›
        by (cases t, auto simp add: F-Mprefix)
      show ‹(s, X) ∈ ℱ ?rhs›
        apply (simp add: F-Mprefix ∗(1) ∗∗(1, 2, 3) image-iff[of ‹ev a›])
        by (simp add: F-Hiding, rule disjI1, auto simp add: ∗∗(4))
    qed
  qed
next
  from disjoint show ‹(s, X) ∈ ℱ ?rhs ⟹ (s, X) ∈ ℱ ?lhs› for s X
    apply (cases ‹s = []›)
     apply (simp add: F-Mprefix F-Hiding disjoint-iff,
           metis event.inject filter.simps(1) imageE)
    by (simp add: Diff-triv F-Hiding-Mprefix-dir2)
qed
```

And we obtain a similar result when adding 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 ∈ 𝒟 (Mprefix A P ⊓ Q \ S)›
      by (simp add: D-Hiding D-Sliding D-Ndet T-Sliding T-Ndet)
    thus ‹s ∈ 𝒟 ?rhs›
      by (rule set-rev-mp)
         (simp add: D-Ndet D-Sliding Hiding-Ndet subsetI)
  next
    fix s
```

```
  assume ‹s ∈ 𝒟 ?rhs›
  hence ‹s ∈ 𝒟 (Q \ S) ∨ s ∈ 𝒟 (□a ∈ A → P a \ S)›
    by (simp add: Hiding-Mprefix-disjoint[OF disjoint]
                  D-Ndet D-Sliding) blast
  thus ‹s ∈ 𝒟 ?lhs›
    by (auto simp add: D-Hiding D-Sliding T-Sliding)
next
  fix s X
  assume same-div : ‹𝒟 ?lhs = 𝒟 ?rhs›
  assume ‹(s, X) ∈ ℱ ?lhs›
  then consider ‹s ∈ 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 ≠ []› ‹(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 assms : ‹t ≠ []› ‹(t, X ∪ ev ‘ S) ∈ ℱ (Mprefix A P)›
      with disjoint have ‹trace-hide t (ev ‘ S) ≠ []›
        by (cases t, auto simp add: F-Mprefix)
      also have ‹(s, X) ∈ ℱ (Mprefix A P \ S)›
        using assms by (auto simp: F-Hiding ∗(1))
      ultimately show ‹(s, X) ∈ ℱ ?rhs›
        by (simp add: F-Sliding ∗(1))
    qed
  qed
next
  have ∗ : ‹t ∈ 𝒯 (Mprefix A P) ⟹ trace-hide t (ev ‘ S) = [] ⟷ 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)
    by (metis (mono-tags, lifting) ∗ NT-ND Nil-Nin-D-Mprefix
              UNIV-I f-inv-into-f lessI nless-le strict-mono-on-eqD)
  fix s X
```

assume *same-div* : ‹$\mathcal{D}$ *?lhs* = $\mathcal{D}$ *?rhs*›
assume ‹(*s*, *X*) ∈ $\mathcal{F}$ *?rhs*›
with ∗∗ **consider** ‹(*s*, *X*) ∈ $\mathcal{F}$ (*Q* \ *S*)›
| ‹*s* ≠ []› ‹(*s*, *X*) ∈ $\mathcal{F}$ (*Mprefix A P* \ *S*)›
by (*simp add*: *Hiding-Mprefix-disjoint*[*OF disjoint*] *F-Sliding D-Mprefix*) *blast*
thus ‹(*s*, *X*) ∈ $\mathcal{F}$ *?lhs*›
proof *cases*
assume ‹(*s*, *X*) ∈ $\mathcal{F}$ (*Q* \ *S*)›
then consider ‹*s* ∈ $\mathcal{D}$ (*Q* \ *S*)›
| ‹∃ *t*. *s* = *trace-hide t* (*ev* ‘ *S*) ∧ (*t*, *X* ∪ *ev* ‘ *S*) ∈ $\mathcal{F}$ *Q*›
by (*simp add*: *F-Hiding D-Hiding*) *blast*
thus ‹(*s*, *X*) ∈ $\mathcal{F}$ *?lhs*›
proof *cases*
assume ‹*s* ∈ $\mathcal{D}$ (*Q* \ *S*)›
hence ‹*s* ∈ $\mathcal{D}$ *?rhs*› **by** (*simp add*: *D-Ndet D-Sliding*)
with *same-div D-F* **show** ‹(*s*, *X*) ∈ $\mathcal{F}$ *?lhs*› **by** *blast*
next
assume ‹∃ *t*. *s* = *trace-hide t* (*ev* ‘ *S*) ∧ (*t*, *X* ∪ *ev* ‘ *S*) ∈ $\mathcal{F}$ *Q*›
thus ‹(*s*, *X*) ∈ $\mathcal{F}$ *?lhs*›
by (*simp add*: *F-Hiding F-Sliding*) *blast*
qed
next
assume *assms* : ‹*s* ≠ []› ‹(*s*, *X*) ∈ $\mathcal{F}$ (*Mprefix A P* \ *S*)›
then consider ‹*s* ∈ $\mathcal{D}$ (*Mprefix A P* \ *S*)›
| ‹∃ *t*. *t* ≠ [] ∧ *s* = *trace-hide t* (*ev* ‘ *S*) ∧ (*t*, *X* ∪ *ev* ‘ *S*) ∈ $\mathcal{F}$ (*Mprefix A P*)›
by (*simp add*: *F-Hiding D-Hiding*) (*metis filter.simps*(*1*))
thus ‹(*s*, *X*) ∈ $\mathcal{F}$ *?lhs*›
proof *cases*
assume ‹*s* ∈ $\mathcal{D}$ (*Mprefix A P* \ *S*)›
hence ‹*s* ∈ $\mathcal{D}$ *?rhs*› **by** (*simp add*: *D-Ndet D-Sliding*)
with *same-div D-F* **show** ‹(*s*, *X*) ∈ $\mathcal{F}$ *?lhs*› **by** *blast*
next
show ‹∃ *t*. *t* ≠ [] ∧ *s* = *trace-hide t* (*ev* ‘ *S*) ∧
(*t*, *X* ∪ *ev* ‘ *S*) ∈ $\mathcal{F}$ (*Mprefix A P*) ⟹ (*s*, *X*) ∈ $\mathcal{F}$ *?lhs*›
by (*auto simp add*: *F-Sliding F-Hiding*)
qed
qed
qed
qed

16.2.3 *Hiding* and *Mprefix* for non-disjoint Sets

Finally the new version, when $A \cap S \neq \{\}$.

theorem *Hiding-Mprefix-non-disjoint*:
— Rework this proof
‹□*a* ∈ *A* → *P a* \ *S* = (□*a* ∈ *A* − *S* → (*P a* \ *S*)) ▷ (⊓*a* ∈ *A* ∩ *S*. (*P a* \ *S*))›
(**is** ‹*?lhs* = *?rhs*›) **if** *non-disjoint*: ‹*A* ∩ *S* ≠ {}›
proof (*subst Process-eq-spec-optimized*, *safe*)

fix s
assume ‹$s \in \mathcal{D}$ *?lhs*›
then obtain t u
  where $*$: ‹*front-tickFree* u› ‹*tickFree* t› ‹s = *trace-hide* t (ev ‘ S) @ u›
        ‹$t \in \mathcal{D}$ (*Mprefix* A P) ∨
         (∃ f. *isInfHiddenRun* f (*Mprefix* A P) $S \wedge t \in$ *range* f)›
  by (*simp add*: *D-Hiding*) *blast*
from $*(4)$ **show** ‹$s \in \mathcal{D}$ *?rhs*›
proof (*elim disjE*)
  assume ‹$t \in \mathcal{D}$ (*Mprefix* A P)›
  then obtain a t' **where** $**$: ‹$a \in A$› ‹$t = ev\ a \# t'$› ‹$t' \in \mathcal{D}$ (P a)›
    by (*simp add*: *D-Mprefix*)
      (*metis event.inject image-iff list.exhaust-sel*)
  have ‹*tickFree* $t' \wedge$
      (*if* $a \in S$ *then* s *else tl* s) = *trace-hide* t' (ev ‘ S) @ $u \wedge$
      ($t' \in \mathcal{D}$ (P a) ∨ (∃f. *isInfHiddenRun* f (P a) $S \wedge t' \in$ *range* f))›
    using $*(2)$ $**(2,\ 3)$ **by** (*auto simp add*: $*(1,\ 3)$ $**(2)$ *image-iff*)
  with $*(1)$ **have** $***$: ‹(*if* $a \in S$ *then* s *else tl* s) $\in \mathcal{D}$ (P $a \setminus S$)›
    by (*simp add*: *D-Hiding*) *blast*
  show ‹$s \in \mathcal{D}$ *?rhs*›
  proof (*cases* ‹$a \in S$›)
    assume ‹$a \in S$›
    hence ‹$a \in A \cap S$› **by** (*simp add*: $**(1)$)
    with $***$ **show** ‹$s \in \mathcal{D}$ *?rhs*›
      by (*auto simp add*: *D-Sliding D-GlobalNdet*)
  next
    assume ‹$a \notin S$›
    hence ‹$a \in A - S$› **by** (*simp add*: $**(1)$)
    with $***$ **show** ‹$s \in \mathcal{D}$ *?rhs*›
      by (*auto simp add*: *D-Sliding D-Mprefix* $*(3)$ $**(2)$ *image-iff*)
  qed
next
  assume ‹∃ f. *isInfHiddenRun* f (*Mprefix* A P) $S \wedge t \in$ *range* f›
  then obtain f
    where $**$: ‹*isInfHiddenRun* f (*Mprefix* A P) S› ‹$t \in$ *range* f› **by** *blast*
  obtain k **where** ‹$t = f\ k$› **using** $**(2)$ **by** *blast*
  show ‹$s \in \mathcal{D}$ *?rhs*›
  proof (*cases* ‹$f\ 0 = []$›)
    assume ‹$f\ 0 = []$›
    hence ‹$f\ 1 \neq []$›
      by (*metis* $**(1)$ *One-nat-def monotoneD nil-less zero-less-Suc*)
    with $**(1)$[*THEN conjunct2*, *THEN conjunct1*, *rule-format*, *of 1*]
    obtain a **where** $***$: ‹$a \in A$› ‹$f\ 1 \neq []$› ‹*hd* (f 1) = ev a›
      by (*simp add*: *T-Mprefix*) *blast*
    have $****$: ‹$0 < j \Longrightarrow f\ j \neq [] \wedge$ *hd* (f j) = ev a› **for** j
    proof (*induct j rule*: *nat-induct-non-zero*)
      from $***(2,\ 3)$ **show** ‹$f\ 1 \neq [] \wedge$ *hd* (f 1) = ev a› **by** *blast*
    next
      case (*Suc j*)

```
    have ‹j < Suc j› by simp
    from **(1)[THEN conjunct1, THEN strict-monoD, OF this]
    obtain v where ‹f (Suc j) = f j @ v›
      by (metis le-list-def order-less-imp-le)
    thus ?case by (simp add: Suc.hyps(2))
  qed
  from ***(1) have *****: ‹a ∈ A ∩ S›
    by simp (metis (no-types, lifting) **(1) ***(2, 3)
                ‹f 0 = []› empty-filter-conv event.inject
                filter.simps(1) image-iff list.set-sel(1))
  have ‹(if i = 0 ∧ t = [] then [] else tl (f (Suc i))) ∈ 𝒯 (P a)› for i
  proof −
    from **(1) ****[of ‹Suc i›] have ‹tl (f (Suc i)) ∈ 𝒯 (P a)›
      by (simp add: T-Mprefix) (metis event.inject)
    thus ‹(if i = 0 ∧ t = [] then [] else tl (f (Suc i))) ∈ 𝒯 (P a)›
      by (simp add: Nil-elem-T)
  qed
  hence ****** :
    ‹front-tickFree u ∧ tickFree (tl t) ∧
     s = trace-hide (tl t) (ev ‘ S) @ u ∧
     isInfHiddenRun (λi. if i = 0 ∧ t = [] then [] else tl (f (Suc i))) (P a) S ∧
     tl t ∈ range (λi. if i = 0 ∧ t = [] then [] else tl (f (Suc i)))›
    apply (intro conjI)
    subgoal by (solves ‹simp add: *(1)›)
    subgoal by (metis *(2) list.sel(2) tickFree-tl)
    subgoal by (metis *(3) **(1) ‹f 0 = []› ‹t = f k› empty-filter-conv
          filter.simps(1) list.sel(2) list.set-sel(2))
    subgoal by (simp add: monotone-on-def,
          metis **(1) Suc-less-eq less-list-def less-tail monotoneD
          nil-le nil-less zero-less-Suc)
    subgoal by blast
    subgoal by (simp, metis **(1) **** ‹f 0 = []› empty-filter-conv
          filter.simps(1) list.set-sel(2) zero-less-Suc)
    by (simp add: image-iff,
        metis Suc-pred ‹f 0 = []› ‹t = f k› bot-nat-0.not-eq-extremum)
  have ‹a ∈ A ∩ S ∧ s ∈ 𝒟 (P a \ S)›
    apply (simp add: D-Hiding *****[simplified])
    using ****** by blast
  hence ‹s ∈ 𝒟 (⊓a ∈ A ∩ S. (P a \ S))› by (simp add: D-GlobalNdet) blast
  thus ‹s ∈ 𝒟 ?rhs› by (simp add: D-Sliding)
next
  assume ‹f 0 ≠ []›
  with **(1)[THEN conjunct2, THEN conjunct1, rule-format, of 0]
  obtain a where *** : ‹a ∈ A› ‹f 0 ≠ []› ‹hd (f 0) = ev a›
    by (simp add: T-Mprefix) blast
  have **** : ‹f j ≠ [] ∧ hd (f j) = ev a› for j
  proof (induct j)
    from ***(2, 3) show ‹f 0 ≠ [] ∧ hd (f 0) = ev a› by blast
  next
```

```
      case (Suc j)
      have ‹j < Suc j› by simp
      from **(1)[THEN conjunct1, THEN strict-monoD, OF this]
      obtain v where ‹f (Suc j) = f j @ v›
        by (metis le-list-def order-less-imp-le)
      thus ?case by (simp add: Suc.hyps(1))
    qed
    show ‹s ∈ 𝒟 ?rhs›
    proof (cases ‹a ∈ S›)
      assume ‹a ∈ S›
      hence ‹a ∈ A ∩ S› by (simp add: ***(1))
      have ‹tickFree (tl t) ∧ s = trace-hide (tl t) (ev ‘ S) @ u ∧
          isInfHiddenRun (λi. tl (f i)) (P a) S ∧ tl t ∈ range (λi. tl (f i))›
        apply (simp add: *(3), intro conjI)
        subgoal by (metis *(2) list.sel(2) tickFree-tl)
      subgoal by (cases t; simp; metis **** ‹a ∈ S› ‹t = f k› image-iff list.sel(1))
        subgoal by (meson **(1) **** less-tail strict-mono-Suc-iff)
        subgoal using **(1) by (simp add: T-Mprefix ****)
     subgoal by (metis (no-types, lifting) **(1) **** filter.simps(2) list.exhaust-sel
list.sel(3))
        using **(2) by blast
      have ‹s ∈ 𝒟 (⊓a ∈ A ∩ S. (P a \ S))›
        apply (simp add: D-GlobalNdet D-Hiding)
        using *(1) ‹a ∈ A ∩ S› ‹?this› by blast
      thus ‹s ∈ 𝒟 ?rhs› by (simp add: D-Sliding)
    next
      assume ‹a ∉ S›
      have ‹tickFree (tl t) ∧
          trace-hide (tl t) (ev ‘ S) @ u = trace-hide (tl t) (ev ‘ S) @ u ∧
          isInfHiddenRun (λi. tl (f i)) (P a) S ∧ tl t ∈ range (λi. tl (f i))›
        apply (simp add: *(3), intro conjI)
        subgoal by (metis *(2) list.sel(2) tickFree-tl)
        subgoal by (meson **(1) **** less-tail strict-mono-Suc-iff)
        subgoal using **(1) by (simp add: T-Mprefix ****)
     subgoal by (metis (no-types, lifting) **(1) **** filter.simps(2) list.exhaust-sel
list.sel(3))
        using **(2) by blast
      from ‹a ∉ S› have ‹s ∈ 𝒟 (□a∈A − S → (P a \ S))›
        apply (simp add: D-Mprefix *(3) ‹t = f k›)
        using ***(1) ****[of k]
        apply (cases ‹f k›; simp add: ‹a ∉ S› image-iff[of ‹ev a›] D-Hiding)
        using ‹?this› by (metis *(1) ‹t = f k› list.sel(3))
      thus ‹s ∈ 𝒟 ?rhs› by (simp add: D-Sliding)
      qed
    qed
  qed
next
  fix s
  assume ‹s ∈ 𝒟 ?rhs›
```

```
  then consider ‹s ∈ 𝒟 (□a ∈ A − S → (P a \ S))›
    | ‹s ∈ 𝒟 (⊓a ∈ A ∩ S. (P a \ S))› by (simp add: D-Sliding) blast
  thus ‹s ∈ 𝒟 ?lhs›
  proof cases
    show ‹s ∈ 𝒟 (□a ∈ A − S → (P a \ S)) ⟹ s ∈ 𝒟 ?lhs›
      by (rule D-Hiding-Mprefix-dir2)
  next
    assume ‹s ∈ 𝒟 (⊓a ∈ A ∩ S. (P a \ S))›
    then obtain a where ∗ : ‹a ∈ A› ‹a ∈ S› ‹s ∈ 𝒟 (P a \ S)›
      by (simp add: D-GlobalNdet)
         (metis (no-types, lifting) IntD1 IntD2 UN-iff empty-iff)
    from ∗(3) obtain t u
      where ∗∗ : ‹front-tickFree u› ‹tickFree t› ‹s = trace-hide t (ev ‘ S) @ u›
                 ‹t ∈ 𝒟 (P a) ∨ (∃ f. isInfHiddenRun f (P a) S ∧ t ∈ range f)›
      by (simp add: D-Hiding) blast
    from ∗∗(4) show ‹s ∈ 𝒟 ?lhs›
    proof (elim disjE)
      assume ‹t ∈ 𝒟 (P a)›
      hence $ : ‹tickFree (ev a # t) ∧ ev a # t ∈ 𝒟 (Mprefix A P) ∧
                 s = trace-hide (ev a # t) (ev ‘ S) @ u›
        by (simp add: ∗(1, 2) ∗∗(2, 3) D-Mprefix)
      show ‹s ∈ 𝒟 ?lhs›
        apply (simp add: D-Hiding)
        using $ ∗∗(1) by blast
    next
      assume ‹∃ f. isInfHiddenRun f (P a) S ∧ t ∈ range f›
      then obtain f where ‹isInfHiddenRun f (P a) S› ‹t ∈ range f› by blast
      hence $ : ‹tickFree (ev a # t) ∧
                 s = trace-hide (ev a # t) (ev ‘ S) @ u ∧
                 isInfHiddenRun (λi. ev a # f i) (Mprefix A P) S ∧
                 ev a # t ∈ range (λi. ev a # f i)›
        by (auto simp add: ∗(1, 2) ∗∗(2, 3) less-cons monotone-on-def T-Mprefix)
      show ‹s ∈ 𝒟 ?lhs›
        apply (simp add: D-Hiding)
        using $ ∗∗(1) by blast
    qed
  qed
next
  fix s X
  assume same-div : ‹𝒟 ?lhs = 𝒟 ?rhs›
  assume ‹(s, X) ∈ ℱ ?lhs›
  then consider ‹s ∈ 𝒟 ?lhs›
    | ‹∃ 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 ?rhs›
  proof cases
    from D-F same-div 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)›
```

```
    then obtain t where ∗ : ‹s = trace-hide t (ev ‘ S)›
                           ‹(t, X ∪ ev ‘ S) ∈ ℱ (Mprefix A P)› by blast
    from ∗(2) consider ‹t = []› ‹(X ∪ ev ‘ S) ∩ ev ‘ A = {}›
      | ‹∃ a t'. a ∈ A ∧ t = ev a # t' ∧ (t', X ∪ ev ‘ S) ∈ ℱ (P a)›
      by (simp add: F-Mprefix) (metis event.inject image-iff list.collapse)
    thus ‹(s, X) ∈ ℱ ?rhs›
    proof cases
      show ‹t = [] ⟹ (X ∪ ev ‘ S) ∩ ev ‘ A = {} ⟹ (s, X) ∈ ℱ ?rhs›
        using ∗(1) by (auto simp add: F-Sliding F-GlobalNdet)
    next
      assume ‹∃ a t'. a ∈ A ∧ t = ev a # t' ∧ (t', X ∪ ev ‘ S) ∈ ℱ (P a)›
      then obtain a t' where ∗∗ : ‹a ∈ A› ‹t = ev a # t'›
                                 ‹(t', X ∪ ev ‘ S) ∈ ℱ (P a)› by blast
      show ‹(s, X) ∈ ℱ ?rhs›
      proof (cases ‹a ∈ S›)
        show ‹a ∈ S ⟹ (s, X) ∈ ℱ ?rhs›
          using ∗(1) ∗∗ by (auto simp add: F-Sliding F-GlobalNdet F-Hiding)
      next
        show ‹a ∉ S ⟹ (s, X) ∈ ℱ ?rhs›
          using ∗(1) ∗∗(1, 2, 3) by (auto simp add: F-Sliding F-Mprefix F-Hiding)
      qed
    qed
  qed
next
  fix s X
  assume same-div : ‹𝒟 ?lhs = 𝒟 ?rhs›
  assume ‹(s, X) ∈ ℱ ?rhs›
  then consider ‹(s, X) ∈ ℱ (⊓a ∈ A ∩ S. (P a \ S))›
    | ‹s ≠ []› ‹(s, X) ∈ ℱ (□a ∈ A − S → (P a \ S))›
    by (auto simp add: F-Sliding)
  thus ‹(s, X) ∈ ℱ ?lhs›
  proof cases
    assume ‹(s, X) ∈ ℱ (⊓a ∈ A ∩ S. (P a \ S))›
    then obtain a where ∗ : ‹a ∈ A› ‹a ∈ S› ‹(s, X) ∈ ℱ (P a \ S)›
      by (simp add: F-GlobalNdet non-disjoint) blast
    from ∗(3) consider ‹s ∈ 𝒟 (P a \ S)›
      | ‹∃ t. s = trace-hide t (ev ‘ S) ∧ (t, X ∪ ev ‘ S) ∈ ℱ (P a)›
      by (simp add: F-Hiding D-Hiding) blast
    thus ‹(s, X) ∈ ℱ ?lhs›
    proof cases
      assume ‹s ∈ 𝒟 (P a \ S)›
      then obtain t u
        where ∗∗ : ‹front-tickFree u› ‹tickFree t›
                   ‹s = trace-hide t (ev ‘ S) @ u›
                   ‹t ∈ 𝒟 (P a) ∨ (∃ f. isInfHiddenRun f (P a) S ∧ t ∈ range f)›
        by (simp add: D-Hiding) blast
      from ∗∗(4) show ‹(s, X) ∈ ℱ ?lhs›
      proof (elim disjE)
        assume ‹t ∈ 𝒟 (P a)›
```

```
    hence $ : ‹tickFree (ev a # t) ∧ s = trace-hide (ev a # t) (ev ‘ S) @ u ∧
              ev a # t ∈ 𝒟 (Mprefix A P)›
      by (simp add: D-Mprefix ∗(1, 2) ∗∗(2, 3) image-iff[of ‹ev a›])
    show ‹(s, X) ∈ ℱ ?lhs›
      apply (simp add: F-Hiding)
      using $ ∗∗(1) by blast
  next
    assume ‹∃f. isInfHiddenRun f (P a) S ∧ t ∈ range f›
    then obtain f where ‹isInfHiddenRun f (P a) S› ‹t ∈ range f› by blast
    hence $ : ‹tickFree (ev a # t) ∧ s = trace-hide (ev a # t) (ev ‘ S) @ u ∧
              isInfHiddenRun (λi. ev a # f i) (Mprefix A P) S ∧
              ev a # t ∈ range (λi. ev a # f i)›
      by (simp add: T-Mprefix ∗(1, 2) ∗∗(2, 3)
                    image-iff[of ‹ev a›] less-cons monotone-on-def) blast
    show ‹(s, X) ∈ ℱ ?lhs›
      apply (simp add: F-Hiding)
      using $ ∗∗(1) by blast
  qed
 next
  assume ‹∃t. s = trace-hide t (ev ‘ S) ∧ (t, X ∪ ev ‘ S) ∈ ℱ (P a)›
  then obtain t where ‹s = trace-hide t (ev ‘ S)›
                      ‹(t, X ∪ ev ‘ S) ∈ ℱ (P a)› by blast
  hence ‹s = trace-hide (ev a # t) (ev ‘ S) ∧
        (ev a # t, X ∪ ev ‘ S) ∈ ℱ (Mprefix A P)›
    by (simp add: ∗(1, 2) F-Mprefix)
  show ‹(s, X) ∈ ℱ ?lhs›
    apply (simp add: F-Hiding, rule disjI1)
    using ‹?this› by blast
 qed
next
 show ‹s ≠ [] ⟹ (s, X) ∈ ℱ (□a∈A − S → (P a \ S)) ⟹ (s, X) ∈ ℱ ?lhs›
  by (rule F-Hiding-Mprefix-dir2)
qed
qed
```

— Just a small lemma to understand why the nonempty hypothesis is necessary.

```
lemma ‹∃A P S. A ∩ S = {} ∧
            □a ∈ A::nat set → P a \ S ≠
            (□a ∈ A − S → (P a \ S)) ▷ (⊓a ∈ A ∩ S. (P a \ S))›
proof (intro exI)
 show ‹{0} ∩ {Suc 0} = {} ∧
      □a ∈ {0}::nat set → SKIP \ {Suc 0} ≠
     (□a ∈ {0} − {Suc 0} → (SKIP \ {Suc 0})) ▷ (⊓a ∈ {0} ∩ {Suc 0}. (SKIP
\ {Suc 0}))›
   apply (simp add: write0-def[symmetric] no-Hiding-write0 Hiding-set-SKIP)
   by (simp add: Process-eq-spec write0-def
                 F-Mprefix F-Sliding F-STOP set-eq-iff) blast
qed
```

And we obtain a similar result when adding a (▷) in the expression.

lemma *Hiding-Mprefix-Sliding-non-disjoint*:
‹((□$a \in A \to P\ a$) ▷ Q) \ S = (□$a \in A - S \to (P\ a \setminus S)$) ▷
($Q \setminus S$) ⊓ (⊓$a \in A \cap S.\ (P\ a \setminus S)$)›
if *non-disjoint*: ‹$A \cap S \neq \{\}$›
proof (*subst Sliding-Ndet*(*2*),
subst Hiding-Mprefix-non-disjoint[*OF non-disjoint*, *symmetric*])
show ‹*Mprefix A P* ▷ $Q \setminus S$ =
((□$a \in A - S \to (P\ a \setminus S)$) ▷ ($Q \setminus S$)) ⊓ (□$a \in A \to P\ a \setminus S$)›
(**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* ⊓ $Q \setminus S$)›
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-Ndet subsetI*)
next
fix *s*
assume ‹$s \in \mathcal{D}$ *?rhs*›
hence ‹$s \in \mathcal{D}$ ($Q \setminus S$) ∨ $s \in \mathcal{D}$ (□$a \in A \to P\ a \setminus S$)›
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) ∈ $\mathcal{F}$ *?lhs*›
then consider ‹$s \in \mathcal{D}$ *?lhs*›
|‹∃ t. s = *trace-hide t* (*ev* ‘ S) ∧ (t, $X \cup$ *ev* ‘ S) ∈ $\mathcal{F}$ (*Mprefix A P* ▷ Q)›
by (*simp add*: *F-Hiding D-Hiding*) *blast*
thus ‹(s, X) ∈ $\mathcal{F}$ *?rhs*›
proof *cases*
from *same-div D-F* **show** ‹$s \in \mathcal{D}$ *?lhs* ⟹ (s, X) ∈ $\mathcal{F}$ *?rhs*› **by** *blast*
next
assume ‹∃ t. s = *trace-hide t* (*ev* ‘ S) ∧
(t, $X \cup$ *ev* ‘ S) ∈ $\mathcal{F}$ (*Mprefix A P* ▷ Q)›
then obtain *t*
where ∗ : ‹s = *trace-hide t* (*ev* ‘ S)›
‹(t, $X \cup$ *ev* ‘ S) ∈ $\mathcal{F}$ (*Mprefix A P* ▷ Q)› **by** *blast*
from ∗(*2*) **consider** ‹(t, $X \cup$ *ev* ‘ S) ∈ $\mathcal{F}$ Q›
| ‹(t, $X \cup$ *ev* ‘ S) ∈ $\mathcal{F}$ (*Mprefix A P*)›
by (*simp add*: *F-Sliding D-Mprefix*) *blast*
thus ‹(s, X) ∈ $\mathcal{F}$ *?rhs*›
proof *cases*
have ‹(t, $X \cup$ *ev* ‘ S) ∈ $\mathcal{F}$ Q ⟹ (s, X) ∈ $\mathcal{F}$ ($Q \setminus S$)›
by (*auto simp add*: *F-Hiding* ∗(*1*))
thus ‹(t, $X \cup$ *ev* ‘ S) ∈ $\mathcal{F}$ Q ⟹ (s, X) ∈ $\mathcal{F}$ *?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
qed

16.3 (▷) 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-distr*)
have ‹*Mprefix B Q* ⊓ (*Mprefix A P* □ *Mprefix B Q*) ▷ *R* = *Mprefix A P* □ *Mprefix B Q* ▷ *R*›
by (*metis Det-STOP Ndet-commute Ndet-distrib Sliding-STOP-Det Sliding-assoc Sliding-def*)
thus ‹*?term* ⊓ *Mprefix B Q* ▷ *R* = *?term* ▷ *R*›
by (*simp add*: *Mprefix-Det-distr 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 (*auto simp add*: *D-Sliding D-Seq T-Sliding*)
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*
apply (*simp add*: *F-Seq*, *elim disjE exE*)
apply (*solves* ‹*auto simp add*: *F-Sliding F-Seq D-Mprefix*›)
apply (*solves* ‹*auto simp add*: *F-Sliding T-Sliding F-Seq*›)
by (*solves* ‹*auto simp add*: *D-Sliding D-Seq F-Sliding F-Seq*›)
next
show ‹(*s*, *X*) ∈ *F ?rhs* ⟹ (*s*, *X*) ∈ *F ?lhs*› **for** *s X*
apply (*simp add*: *F-Sliding*, *elim disjE*)
apply (*solves* ‹*auto simp add*: *F-Seq D-Sliding F-Sliding T-Sliding*›)
apply (*solves* ‹*auto simp add*: *F-Seq D-Sliding F-Sliding T-Sliding*›)
by (*simp add*: *D-Seq D-Mprefix T-Seq T-Mprefix*)
(*metis append-is-Nil-conv event.simps*(*3*) *hd-append list.sel*(*1*))

```
  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. s = t1 @ t2 ∧ t1 ∈ 𝒟 (Mprefix A P ▷ P') ∧
                         tickFree t1 ∧ set t1 ∩ ev ‘ B = {} ∧ front-tickFree t2›
    | ‹∃t1 b t2. 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
    assume ‹∃t1 t2. s = t1 @ t2 ∧ t1 ∈ 𝒟 (Mprefix A P ▷ P') ∧
                    tickFree t1 ∧ set t1 ∩ ev ‘ B = {} ∧ front-tickFree t2›
    then obtain t1 t2
      where * : ‹s = t1 @ t2› ‹t1 ∈ 𝒟 (Mprefix A P ▷ P')› ‹tickFree t1›
                ‹set t1 ∩ ev ‘ B = {}› ‹front-tickFree t2› by blast
    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 (simp add: D-Mprefix image-iff)
           (metis event.inject list.collapse)
      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
    assume ‹∃t1 b t2. s = t1 @ ev b # t2 ∧ t1 @ [ev b] ∈ 𝒯 (Mprefix A P ▷ P')
∧
                      set t1 ∩ ev ‘ B = {} ∧ b ∈ B ∧ t2 ∈ 𝒟 (Q b)›
    then obtain 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 blast
    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 (simp add: T-Mprefix)
          (metis list.exhaust-sel snoc-eq-iff-butlast)
      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 list.sel(1, 3) self-append-conv2)
    next
      assume ‹a ∉ B›
      with ∗(3) consider
        ‹∃ t1 t2. s' = t1 @ t2 ∧ t1 ∈ 𝒟 (P a) ∧ tickFree t1 ∧
              set t1 ∩ ev ‘ B = {} ∧ front-tickFree t2›
        | ‹∃ t1 b t2. 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
        assume ‹∃ t1 t2. s' = t1 @ t2 ∧ t1 ∈ 𝒟 (P a) ∧ tickFree t1 ∧
                    set t1 ∩ ev ‘ B = {} ∧ front-tickFree t2›
        then obtain t1 t2
          where ∗∗ : ‹s' = t1 @ t2› ‹t1 ∈ 𝒟 (P a)› ‹tickFree t1›
                ‹set t1 ∩ ev ‘ B = {}› ‹front-tickFree t2›
```

```
      by blast
    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.distinct(1) list.discI list.sel(1, 3) tickFree-Cons)
  next
    assume ‹∃ t1 b t2. s' = t1 @ ev b # t2 ∧ t1 @ [ev b] ∈ 𝒯 (P a) ∧
                  set t1 ∩ ev ‘ B = {} ∧ b ∈ B ∧ t2 ∈ 𝒟 (Q b)›
    then obtain 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 blast
    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 list.sel(1, 3))
  qed
 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. 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 (simp add: F-Throw D-Throw) blast
 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
  assume ‹∃ t1 b t2. s = t1 @ ev b # t2 ∧ t1 @ [ev b] ∈ 𝒯 (Mprefix A P ▷ P')
∧
                  set t1 ∩ ev ‘ B = {} ∧ b ∈ B ∧ (t2, X) ∈ ℱ (Q b)›
  then obtain 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 blast
  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 (simp add: T-Mprefix)
        (metis list.exhaust-sel snoc-eq-iff-butlast)
    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 (simp add: F-Mprefix image-iff) (metis event.inject list.collapse)
  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) Nil-elem-T)

   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. s' = t1 @ ev b # t2 ∧ t1 @ [ev b] ∈ 𝒯 (P a) ∧
                 set t1 ∩ ev ‘ B = {} ∧ b ∈ B ∧ (t2, X) ∈ ℱ (Q b)›
    by (simp add: F-Throw D-Throw) blast
   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
      assume ‹∃ t1 b t2. s' = t1 @ ev b # t2 ∧ t1 @ [ev b] ∈ 𝒯 (P a) ∧
                         set t1 ∩ ev ‘ B = {} ∧ b ∈ B ∧ (t2, X) ∈ ℱ (Q b)›
      then obtain 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 blast
      from ∗∗ have ∗∗∗ : ‹(ev a # t1) @ [ev b] ∈ 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) ∈ ℱ ?lhs›
        by (simp add: F-Throw T-Sliding) (metis ∗∗∗ Cons-eq-appendI)
    qed
  qed
 qed
qed



end
```

Chapter 17

Conclusion

theory *Conclusion*
imports *OpSemFD OpSemDT OpSemFDBis OpSemDTBis OpSemFBis OpSemT-Bis NewLaws*

begin

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 *ready-set* notion, and described its behaviour with the reference processes and the operators of CSP (which is already a minor contribution).

As main contribution, we defined the *After* operator which represents a bridge between the denotational and the versions of operational semantics for CSP. 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 [3, 5].
We only have a slight variation for the *Sync* operator: *STOP* is replaced by *SKIP*, probably because of the operator definition in `HOL-CSP`.
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 ($_{DT}\rightsquigarrow_{\tau}$) 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 allowing ($_{F}\rightsquigarrow_{\tau}$) and

$(_T\rightsquigarrow_\tau)$ orders to be used operational construction by modifying the definition of *AfterExt.AfterExt*.

But even if $(_{FD}\rightsquigarrow_\tau)$ and $(_{DT}\rightsquigarrow_\tau)$ constructions are (almost) not impacted by this change, this remains a bit disappointing because the monotony of $(_F\rightsquigarrow_\tau)$ and $(_T\rightsquigarrow_\tau)$ w.r.t. to some operators does not allow to recover all the laws of [3, 5].

As a bonus we provided in *HOL−CSP-OpSem.NewLaws* some powerful laws for CSP. Here, we recall only the most important ones:

bij ?f $\Longrightarrow$ *Renaming* (*?P* $\setminus$ *?S*) *?f* = *Renaming ?P ?f* $\setminus$ *?f* ‘ *?S*

bij ?f $\Longrightarrow$ *Renaming* (*?P* $[\![$*?S*$]\!]$ *?Q*) *?f* = *Renaming ?P ?f* $[\![$*?f* ‘ *?S*$]\!]$ *Renaming ?Q ?f*

?A $\cap$ *?S* $\neq \emptyset \Longrightarrow \Box a{\in}?A \rightarrow ?P\ a \setminus ?S = (\Box a{\in}?A - ?S \rightarrow (?P\ a \setminus ?S)) \rhd (\sqcap\ a{\in}?A \cap ?S.\ \ ?P\ a \setminus ?S)$

end

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] 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.

[3] A. Roscoe. *Theory and Practice of Concurrency*. Prentice Hall, 1998.

[4] A. W. Roscoe. Understanding concurrent systems. In *Texts in Computer Science*, 2010.

[5] 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.

[6] 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.