HOL-CSPM - Architectural operators for HOL-CSP

Benoît Ballenghien Safouan Taha Burkhart Wolff

March 17, 2025

Abstract

Recently, a modern version of Roscoes and Brookes [1] Failure-Divergence Semantics for CSP has been formalized in Isabelle [4]. On top of this theory, we develop the so-called “architectural operators”, i.e. generalizations of basic non-deterministic choices, synchronized producs and sequetializations, as has been introduced in the well-known FDR4 model-checker for CSP.

While FDR4 uses these architectural operators as handy macros that help to structure the specifications, they are basically macro-expanded before the Labelled Transition Systems were generated. In contrast, we develop the formal theory of these operators in themselves which paves the way for a more structured approach to reasoning in HOL-CSP. Our generalizations will take commutativity and idempotence into account, such that they become fully-abstract wrt. to index-sets, index-multi-sets or lists, respectively.

Additionally, the theory of some more exotic — but in the CSP literature discussed — operators have been developed; in particular throw and interupt.

For these "architectural operators", we will prove the properties of refinement, monotonicity and continuity and the laws of interaction in order to simplify their use.

Finally, we will give examples of their usefulness when trying to model complex systems.

Contents

Chapter 1

Introduction

1.1 Motivations

`HOL-CSP` [4] 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" [2] and the semantic details in a joint Paper of Roscoe and Brooks "An improved failures model for communicating processes" [1].

In the session `HOL-CSP` are introduced the type $('a, 'r)\ process_{ptick}$, several classic CSP operators and number of laws that govern their interactions.

Four of them are binary operators: the non-deterministic choice $P \sqcap Q$, the deterministic choice $P \square Q$, the synchronization $P [\![S]\!] Q$ and the sequential composition $P \mathbin{;} Q$.

Analogously to the finite sum $\sum_{i\,=\,0}^{n} a\ i$ which is generalization of the addition $a + b$, we define generalisations of the binary operators of CSP.

The most straight-forward way to do so would be a fold on a list of processes. However, in many cases, we have additional properties, like commutativity, idempotency, etc. that allow for stronger/more abstract constructions. In particular, in several cases, generalization to unbounded and even infinite index-sets are possible.

The notations we choose are widely inspired by the CSP_M syntax of FDR: https://cocotec.io/fdr/manual/cspm.html.

For the non-deterministic choice $P \sqcap Q$, this is already done in `HOL-CSP`. In this session we therefore introduce the multi-operators:

- the global deterministic choice, written $\square\ a \in A.\ P\ a$, generalizing $P \square Q$
- the multi-synchronization product, written $[\![S]\!]\ m \in\!\#\ M.\ P\ m$, gen-

eralizing P $[\![S]\!]$ Q with the two special cases $|||$ $m \in\# M$. $P\ m$ and $||$ $m \in\# M$. $P\ m$

- the multi-sequential composition, written *SEQ* $l \in@ L$. $P\ l$, generalizing P **;** Q. We prove their continuity and refinements rules, as well as some laws governing their interactions.

We also provide the definitions of the POTS and Dining Philosophers examples, which greatly benefit from the newly introduced generalized operators. Since they appear naturally when modeling complex architectures, we may call them *architectural operators*: these multi-operators represent the heart of the architectural composition principles of CSP.

Additionally, we developed the theory of the interrupt operators *Sliding*, *Throw* and *Interrupt* [3]. This part of the present theory reintroduces denotational semantics for these operators and constructs on this basis the algebraic laws for them.

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.

Finally, his session includes a very powerful result about *deadlock-free* and *Sync*: the interleaving P $|||$ Q is *deadlock-free* if P and Q are, and so is the multi-interleaving of processes $P\ m$ for $m \in\# M$.

1.2 The Global Architecture of HOL-CSPM

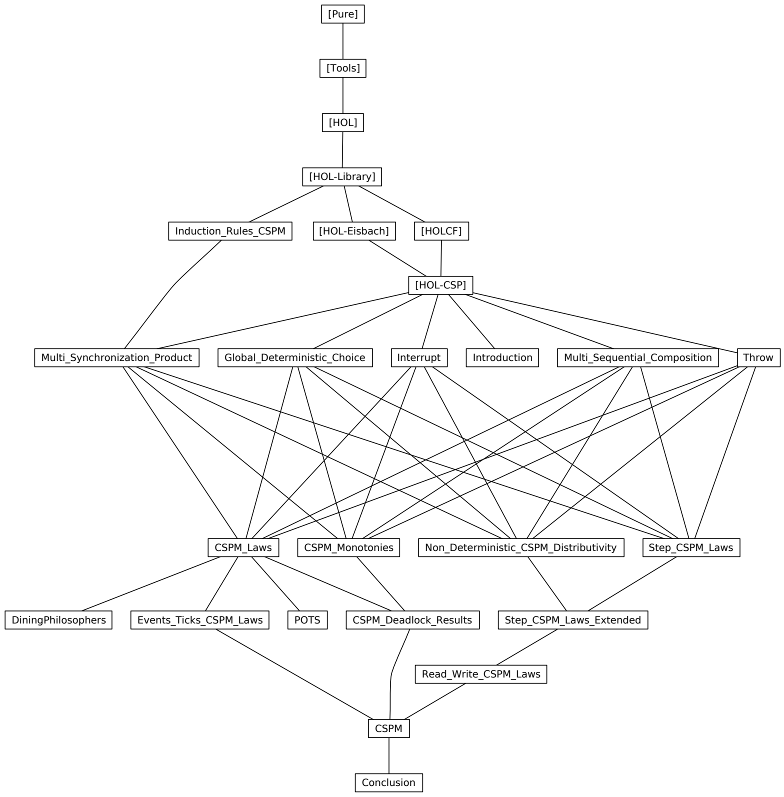

Figure 1.1: The overall architecture

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

1. `HOL-Eisbach` from the Isabelle/HOL distribution,

2. `HOLCF` from the Isabelle/HOL distribution, and

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

Chapter 2

Preliminary Work

2.1 Induction Rules for *'a set*

```
lemma finite-subset-induct-singleton
  [consumes 3, case-names singleton insertion]:
  ‹⟦a ∈ A; finite F; F ⊆ A; P {a};
    ⋀x F. finite F ⟹ x ∈ A ⟹ x ∉ (insert a F) ⟹ P (insert a F)
          ⟹ P (insert x (insert a F))⟧ ⟹ P (insert a F)›
  apply (erule Finite-Set.finite-subset-induct, simp-all)
  by (metis insert-absorb2 insert-commute)
```

```
lemma finite-set-induct-nonempty
  [consumes 2, case-names singleton insertion]:
  assumes ‹A ≠ {}› and ‹finite A›
    and singleton: ‹⋀a. a ∈ A ⟹ P {a}›
    and insert: ‹⋀x F. ⟦F ≠ {}; finite F; x ∈ A; x ∉ F;  P F⟧
                      ⟹ P (insert x F)›
  shows ‹P A›
proof−
  obtain a A' where ‹a ∈ A› ‹finite A'› ‹A' ⊆ A› ‹A = insert a A'›
    using ‹A ≠ {}› ‹finite A› by fastforce
  show ‹P A›
    apply (subst ‹A = insert a A'›, rule finite-subset-induct-singleton[of a A])
    by (simp-all add: ‹a ∈ A› ‹finite A'› ‹A' ⊆ A› singleton insert)
qed
```

```
lemma finite-subset-induct-singleton'
  [consumes 3, case-names singleton insertion]:
  ‹⟦a ∈ A; finite F; F ⊆ A; P {a};
    ⋀x F. ⟦finite F; x ∈ A; insert a F ⊆ A; x ∉ insert a F; P (insert a F)⟧
          ⟹ P (insert x (insert a F))⟧
   ⟹ P (insert a F)›
  apply (erule Finite-Set.finite-subset-induct', simp-all)
```

by (*metis insert-absorb2 insert-commute*)

lemma *induct-subset-empty-single*[*consumes 1*]:
‹⟦*finite A*; *P* {}; ⋀*a*. *a* ∈ *A* ⟹ *P* {*a*};
⋀*F a*. ⟦*a* ∈ *A*; *a* ∉ *F*; *finite F*; *F* ⊆ *A*; *F* ≠ {}; *P F*⟧ ⟹ *P* (*insert a F*)⟧ ⟹
P A›
by (*rule finite-subset-induct′*) *auto*

2.2 Induction Rules for ′α *multiset*

The following rule comes directly from *HOL−Library.Multiset* but is written with *consumes 2* instead of *consumes 1*. I rewrite here a correct version.

lemma *msubset-induct* [*consumes 1*, *case-names empty add*]:
‹⟦*F* ⊆# *A*; *P* {#}; ⋀*a F*. ⟦*a* ∈# *A*; *P F*⟧ ⟹ *P* (*add-mset a F*)⟧ ⟹ *P F*›
by (*fact multi-subset-induct*)

lemma *msubset-induct-singleton* [*consumes 2*, *case-names m-singleton add*]:
‹⟦*a* ∈# *A*; *F* ⊆# *A*; *P* {#*a*#};
⋀*x F*. ⟦*x* ∈# *A*; *P* (*add-mset a F*)⟧ ⟹ *P* (*add-mset x* (*add-mset a F*))⟧
⟹ *P* (*add-mset a F*)›
by (*erule msubset-induct*, *simp-all add*: *add-mset-commute*)

lemma *mset-induct-nonempty* [*consumes 1*, *case-names m-singleton add*]:
assumes ‹*A* ≠ {#}›
and *m-singleton*: ‹⋀*a*. *a* ∈# *A* ⟹ *P* {#*a*#}›
and *add*: ‹⋀*x F*. ⟦*F* ≠ {#}; *x* ∈# *A*; *P F*⟧ ⟹ *P* (*add-mset x F*)›
shows ‹*P A*›
proof−
obtain *a A′* **where** ‹*a* ∈# *A*› ‹*A′* ⊆# *A*› ‹*A* = *add-mset a A′*›
by (*metis* ‹*A* ≠ {#}› *diff-subset-eq-self insert-DiffM multiset-nonemptyE*)
show ‹*P A*›
apply (*subst* ‹*A* = *add-mset a A′*›, *rule msubset-induct-singleton*[*of a A*])
by (*simp-all add*: ‹*a* ∈# *A*› ‹*A′* ⊆# *A*› *m-singleton add*)
qed

lemma *msubset-induct′* [*consumes 2*, *case-names empty add*]:
assumes ‹*F* ⊆# *A*›
and *empty*: ‹*P* {#}›
and *insert*: ‹⋀*a F*. ⟦*a* ∈# *A* − *F*; *F* ⊆# *A*; *P F*⟧ ⟹ *P* (*add-mset a F*)›
shows ‹*P F*›
proof −
from ‹*F* ⊆# *A*›
show *?thesis*
proof (*induct F*)

show ‹P {#}› **by** (*simp add*: *assms*(*2*))
next
case (*add x F*)
then show ‹P (*add-mset x F*)›
using *Diff-eq-empty-iff-mset add-diff-cancel-left add-diff-cancel-left′*
add-mset-add-single local.insert mset-subset-eq-insertD
subset-mset.le-iff-add subset-mset.less-imp-le **by** *fastforce*
qed
qed

lemma *msubset-induct-singleton′* [*consumes 2*, *case-names m-singleton add*]:
‹⟦a ∈# A − F; F ⊆# A; P {#a#};
⋀x F. ⟦x ∈# A − F; F ⊆# A; P (*add-mset a F*)⟧
⟹ P (*add-mset x* (*add-mset a F*))⟧
⟹ P (*add-mset a F*)›
by (*erule msubset-induct′*, *simp-all add*: *add-mset-commute*)

lemma *msubset-induct-singleton″* [*consumes 1*, *case-names m-singleton add*]:
‹⟦*add-mset a F* ⊆# A; P {#a#};
⋀x F. ⟦*add-mset x* (*add-mset a F*) ⊆# A; P (*add-mset a F*)⟧
⟹ P (*add-mset x* (*add-mset a F*))⟧
⟹ P (*add-mset a F*)›
apply (*induct F*, *simp*)
by (*metis add-mset-commute diff-subset-eq-self subset-mset.trans union-single-eq-diff*)

lemma *mset-induct-nonempty′* [*consumes 1*, *case-names m-singleton add*]:
assumes *nonempty*: ‹A ≠ {#}› **and** *m-singleton*: ‹⋀a. a ∈# A ⟹ P {#a#}›
and *hyp*: ‹⋀a x F. ⟦a ∈# A; x ∈# A − *add-mset a F*; *add-mset a F* ⊆# A;
P (*add-mset a F*)⟧ ⟹ P (*add-mset x* (*add-mset a F*))›
shows ‹P A›
proof−
obtain a A' **where** ‹A = *add-mset a A′*› ‹*add-mset a A′* ⊆# A›
using *nonempty multiset-cases subset-mset-def* **by** *auto*
show ‹P A›
apply (*subst* ‹A = *add-mset a A′*›)
using ‹*add-mset a A′* ⊆# A›
proof (*induct A′ rule*: *msubset-induct-singleton″*)
show ‹P {#a#}› **using** ‹A = *add-mset a A′*› *m-singleton* **by** *force*
next
case (*add x F*)
show ‹P (*add-mset x* (*add-mset a F*))›
apply (*subst hyp*)
apply (*simp add*: ‹A = *add-mset a A′*›)
apply (*metis* ‹*add-mset x* (*add-mset a F*) ⊆# A› *add-mset-add-single*
mset-subset-eq-insertD subset-mset.add-diff-inverse
subset-mset.add-le-cancel-left subset-mset-def)

apply (*meson* ‹*add-mset x* (*add-mset a F*) ⊆# *A*› *mset-subset-eq-insertD* *subset-mset.dual-order.strict-implies-order*)
by (*simp-all add*: ‹*P* (*add-mset a F*)›)
qed
qed

lemma *induct-subset-mset-empty-single*:
‹⟦*P* {#}; ⋀*a*. *a* ∈# *M* ⟹ *P* {#*a*#};
⋀*N a*. ⟦*a* ∈# *M*; *N* ⊆# *M*; *N* ≠ {#}; *P N*⟧ ⟹ *P* (*add-mset a N*)⟧ ⟹ *P M*›
by (*metis in-diffD mset-induct-nonempty′*)

2.3 Strong Induction for *nat*

lemma *strong-nat-induct*[*consumes 0*, *case-names 0 Suc*]:
‹⟦*P 0*; ⋀*n*. (⋀*m*. *m* ≤ *n* ⟹ *P m*) ⟹ *P* (*Suc n*)⟧ ⟹ *P n*›
by (*induct n rule*: *nat-less-induct*) (*metis gr0-implies-Suc gr-zeroI less-Suc-eq-le*)

lemma *strong-nat-induct-non-zero*[*consumes 1*, *case-names 1 Suc*]:
‹⟦*0* < *n*; *P 1*; ⋀*n*. *0* < *n* ⟹ (⋀*m*. *0* < *m* ∧ *m* ≤ *n* ⟹ *P m*) ⟹ *P* (*Suc n*)⟧
⟹ *P n*›
by (*induct n rule*: *nat-less-induct*) (*metis One-nat-def gr0-implies-Suc gr-zeroI less-Suc-eq-le*)

2.4 Useful Results for Cartesian Products

lemma *prem-Multi-cartprod*:
‹(λ(*x*, *y*). *x* @ *y*) ‘ (*A* × *B*) = {*s* @ *t* |*s t*. (*s*, *t*) ∈ *A* × *B* }›
‹(λ(*x*, *y*). *x* # *y*) ‘ (*A′* × *B*) = {*s* # *t* |*s t*. (*s*, *t*) ∈ *A′* × *B* }›
‹(λ(*x*, *y*). [*x*, *y*]) ‘ (*A′* × *B′*) = {[*s*, *t*] |*s t*. (*s*, *t*) ∈ *A′* × *B′*}›
by *auto*

Chapter 3

Definitions of the Architectural Operators

3.1 The Global Deterministic Choice

3.1.1 Definition

This is an experimental generalization of the deterministic choice. In previous versions, this was done by folding the binary operator (□), but the set was of course necessarily finite. Now we give an abstract definition with the failures and the divergences.

lift-definition *GlobalDet* :: ‹['b set, 'b ⇒ ('a, 'r) $process_{ptick}$] ⇒ ('a, 'r) $process_{ptick}$›
  is ‹λA P. ({(s, X). s = [] ∧ (s, X) ∈ (⋂a∈A. 𝓕 (P a))} ∪
        {(s, X). s ≠ [] ∧ (s, X) ∈ (⋃a∈A. 𝓕 (P a))} ∪
        {(s, X). s = [] ∧ s ∈ (⋃a∈A. 𝓓 (P a))} ∪
        {(s, X). ∃r. s = [] ∧ ✓(r) ∉ X ∧ [✓(r)] ∈ (⋃a∈A. 𝓣 (P a))},
        ⋃a∈A. 𝓓 (P a))›
proof –
  show ‹*?thesis A P*› (**is** ‹*is-process* (*?f*, ⋃a∈A. 𝓓 (P a))›) **for** *A P*
  proof (*unfold is-process-def DIVERGENCES-def FAILURES-def fst-conv snd-conv, intro conjI allI impI*)
    show ‹([], {}) ∈ *?f*› **by** (*simp add*: *is-processT1*)
  next
    show ‹(s, X) ∈ *?f* ⟹ *ftF s*› **for** *s X* **by** (*auto intro*: *is-processT2*)
  next
    show ‹(s @ t, {}) ∈ *?f* ⟹ (s, {}) ∈ *?f*› **for** *s t*
      by (*auto intro*!: *is-processT1 dest*: *is-processT3*)
  next
    fix *s X Y*
    assume *assm* : ‹(s, Y) ∈ *?f* ∧ X ⊆ Y›
    then consider ‹s = []› ‹⋀a. a ∈ A ⟹ (s, Y) ∈ 𝓕 (P a)›
      | *e s' a* **where** ‹a ∈ A› ‹s = e # s'› ‹(s, Y) ∈ 𝓕 (P a)›
      | *a* **where** ‹a ∈ A› ‹s = []› ‹s ∈ 𝓓 (P a)›

| a r **where** ‹$a \in A$› ‹$s = []$› ‹$✓(r) \notin Y$› ‹$[✓(r)] \in \mathcal{T}\ (P\ a)$›
by (*cases s*) *auto*
thus ‹$(s,\ X) \in\ ?f$›
proof *cases*
assume ‹$s = []$› ‹$\bigwedge a.\ a \in A \Longrightarrow (s,\ Y) \in \mathcal{F}\ (P\ a)$›
from *this*(*2*) *assm* **have** ‹$a \in A \Longrightarrow (s,\ X) \in \mathcal{F}\ (P\ a)$› **for** a
by (*meson is-processT4*)
with ‹$s = []$› **show** ‹$(s,\ X) \in\ ?f$› **by** *fast*
next
fix $e\ s'\ a$ **assume** ‹$a \in A$› ‹$s = e\ \#\ s'$› ‹$(s,\ Y) \in \mathcal{F}\ (P\ a)$›
from ‹$(s,\ Y) \in \mathcal{F}\ (P\ a)$› *assm*[*THEN conjunct2*]
have ‹$(s,\ X) \in \mathcal{F}\ (P\ a)$› **by** (*fact is-processT4*)
with ‹$a \in A$› ‹$s = e\ \#\ s'$› **show** ‹$(s,\ X) \in\ ?f$› **by** *blast*
next
show ‹$a \in A \Longrightarrow s = [] \Longrightarrow s \in \mathcal{D}\ (P\ a) \Longrightarrow (s,\ X) \in\ ?f$› **for** a **by** *blast*
next
fix $a\ r$ **assume** ‹$a \in A$› ‹$s = []$› ‹$✓(r) \notin Y$› ‹$[✓(r)] \in \mathcal{T}\ (P\ a)$›
from ‹$✓(r) \notin Y$› *assm*[*THEN conjunct2*] **have** ‹$✓(r) \notin X$› **by** *blast*
with ‹$a \in A$› ‹$s = []$› ‹$[✓(r)] \in \mathcal{T}\ (P\ a)$› **show** ‹$(s,\ X) \in\ ?f$› **by** *blast*
qed
next
fix $s\ X\ Y$
assume *assm* : ‹$(s,\ X) \in\ ?f \wedge (\forall c.\ c \in Y \longrightarrow (s\ @\ [c],\ \{\}) \notin\ ?f)$›
then consider ‹$s = []$› ‹$\bigwedge a.\ a \in A \Longrightarrow (s,\ X) \in \mathcal{F}\ (P\ a)$›
| $e\ s'\ a$ **where** ‹$a \in A$› ‹$s = e\ \#\ s'$› ‹$(s,\ X) \in \mathcal{F}\ (P\ a)$›
| a **where** ‹$a \in A$› ‹$s = []$› ‹$s \in \mathcal{D}\ (P\ a)$›
| $a\ r$ **where** ‹$a \in A$› ‹$s = []$› ‹$✓(r) \notin X$› ‹$[✓(r)] \in \mathcal{T}\ (P\ a)$›
by (*cases s*) *auto*
thus ‹$(s,\ X \cup Y) \in\ ?f$›
proof *cases*
assume ‹$s = []$› ‹$\bigwedge a.\ a \in A \Longrightarrow (s,\ X) \in \mathcal{F}\ (P\ a)$›
from *this*(*2*) *assm*[*THEN conjunct2*]
have ‹$a \in A \Longrightarrow (s,\ X \cup Y) \in \mathcal{F}\ (P\ a)$› **for** a
by (*simp add*: *is-processT5*)
with ‹$s = []$› **show** ‹$(s,\ X \cup Y) \in\ ?f$› **by** *blast*
next
fix $e\ s'\ a$ **assume** ‹$a \in A$› ‹$s = e\ \#\ s'$› ‹$(s,\ X) \in \mathcal{F}\ (P\ a)$›
from ‹$(s,\ X) \in \mathcal{F}\ (P\ a)$› *assm*[*THEN conjunct2*]
have ‹$(s,\ X \cup Y) \in \mathcal{F}\ (P\ a)$› **by** (*simp add*: ‹$a \in A$› *is-processT5*)
with ‹$a \in A$› ‹$s = e\ \#\ s'$› **show** ‹$(s,\ X \cup Y) \in\ ?f$› **by** *blast*
next
show ‹$a \in A \Longrightarrow s = [] \Longrightarrow s \in \mathcal{D}\ (P\ a) \Longrightarrow (s,\ X \cup Y) \in\ ?f$› **for** a **by** *blast*
next
fix $a\ r$ **assume** ‹$a \in A$› ‹$s = []$› ‹$✓(r) \notin X$› ‹$[✓(r)] \in \mathcal{T}\ (P\ a)$›
with *assm*[*THEN conjunct2*] *T-F* **show** ‹$(s,\ X \cup Y) \in\ ?f$› **by** *simp blast*
qed
next
fix $s\ r\ X$

assume ‹$(s\ @\ [✓(r)],\ \{\}) \in\ ?f$›
then obtain a **where** ‹$a \in A$› ‹$(s\ @\ [✓(r)],\ \{\}) \in \mathcal{F}\ (P\ a)$› **by** *blast*
from *this*(*2*) **have** ‹$(s,\ X - \{✓(r)\}) \in \mathcal{F}\ (P\ a)$› **by** (*fact is-processT6*)
show ‹$(s,\ X - \{✓(r)\}) \in\ ?f$›
proof (*cases* ‹$s = []$›)
show ‹$s = [] \Longrightarrow (s,\ X - \{✓(r)\}) \in\ ?f$›
by *simp* (*metis F-T* ‹$(s\ @\ [✓(r)],\ \{\}) \in \mathcal{F}\ (P\ a)$› ‹$a \in A$› *append-Nil*)
next
assume ‹$s \neq []$›
with ‹$a \in A$› ‹$(s,\ X - \{✓(r)\}) \in \mathcal{F}\ (P\ a)$›
show ‹$(s,\ X - \{✓(r)\}) \in\ ?f$› **by** *blast*
qed
next
show ‹$s \in (\bigcup a{\in}A.\ \mathcal{D}\ (P\ a)) \wedge tF\ s \wedge ftF\ t \Longrightarrow s\ @\ t \in (\bigcup a{\in}A.\ \mathcal{D}\ (P\ a))$›
for $s\ t$
by (*blast intro*: *is-processT7*)
next
show ‹$s \in (\bigcup a{\in}A.\ \mathcal{D}\ (P\ a)) \Longrightarrow (s,\ X) \in\ ?f$› **for** $s\ X$
by (*blast intro*: *is-processT8*)
next
show ‹$s\ @\ [✓(r)] \in (\bigcup a{\in}A.\ \mathcal{D}\ (P\ a)) \Longrightarrow s \in (\bigcup a{\in}A.\ \mathcal{D}\ (P\ a))$› **for** $s\ r$
by (*blast intro*: *is-processT9*)
qed
qed

syntax *-GlobalDet* :: ‹[*pttrn*,′*b set*,(′*a*, ′*r*) $process_{ptick}$] ⇒ (′*a*, ′*r*) $process_{ptick}$›
(‹(*3*□((-)/∈(-))./ (-))› [*78*,*78*,*77*] *77*)
syntax-consts *-GlobalDet* ⇌ *GlobalDet*
translations □ $p \in A.\ P$ ⇌ *CONST GlobalDet A* ($\lambda p.\ P$)

3.1.2 The projections

lemma *F-GlobalDet*:
‹$\mathcal{F}\ (\square\ x \in A.\ P\ x) =$
$\{(s,\ X).\ s = [] \wedge (s,\ X) \in (\bigcap a{\in}A.\ \mathcal{F}\ (P\ a))\} \cup$
$\{(s,\ X).\ s \neq [] \wedge (s,\ X) \in (\bigcup a{\in}A.\ \mathcal{F}\ (P\ a))\} \cup$
$\{(s,\ X).\ s = [] \wedge s \in (\bigcup a{\in}A.\ \mathcal{D}\ (P\ a))\} \cup$
$\{(s,\ X).\ \exists r.\ s = [] \wedge ✓(r) \notin X \wedge [✓(r)] \in (\bigcup a{\in}A.\ \mathcal{T}\ (P\ a))\}$›
by (*simp add*: *Failures.rep-eq FAILURES-def GlobalDet.rep-eq*)

lemma *F-GlobalDet*′:
‹$\mathcal{F}\ (\square\ x \in A.\ P\ x) =$
$\{([],\ X)|\ X.\ (\exists a{\in}A.\ P\ a = \bot) \vee (\forall a{\in}A.\ ([],\ X) \in \mathcal{F}\ (P\ a)) \vee$
$(\exists a{\in}A.\ \exists r.\ ✓(r) \notin X \wedge [✓(r)] \in \mathcal{T}\ (P\ a))\} \cup$
$\{(s,\ X)|\ a\ s\ X.\ a \in A \wedge s \neq [] \wedge (s,\ X) \in \mathcal{F}\ (P\ a)\}$›
(**is** ‹$\mathcal{F}\ (\square\ x \in A.\ P\ x) =\ ?rhs$›)
proof (*intro subset-antisym subsetI*)
fix *sX* **assume** ‹$sX \in \mathcal{F}\ (\square\ x \in A.\ P\ x)$›

obtain *s X* **where** ‹*sX* = (*s*, *X*)› **using** *prod.exhaust-sel* **by** *blast*
with ‹*sX* ∈ $\mathcal{F}$ (□ *x* ∈ *A*. *P x*)› **show** ‹*sX* ∈ *?rhs*›
by (*auto simp add*: *F-GlobalDet BOT-iff-Nil-D*)
next
fix *sX* **assume** ‹*sX* ∈ *?rhs*›
obtain *s X* **where** ‹*sX* = (*s*, *X*)› **using** *prod.exhaust-sel* **by** *blast*
with ‹*sX* ∈ *?rhs*› **show** ‹*sX* ∈ $\mathcal{F}$ (□ *x* ∈ *A*. *P x*)›
by (*auto simp add*: *F-GlobalDet BOT-iff-Nil-D*)
qed

lemma *D-GlobalDet*: ‹$\mathcal{D}$ (□ *x* ∈ *A*. *P x*) = (⋃ *a*∈*A*. $\mathcal{D}$ (*P a*))›
by (*simp add*: *Divergences.rep-eq DIVERGENCES-def GlobalDet.rep-eq*)

lemma *T-GlobalDet*:
‹$\mathcal{T}$ (□ *x* ∈ *A*. *P x*) = (*if A* = {} *then* {[]} *else* (⋃ *x*∈*A*. $\mathcal{T}$ (*P x*)))›
by (*auto simp add*: *Traces.rep-eq TRACES-def Failures.rep-eq*[*symmetric*] *F-GlobalDet*
intro: *is-processT1 is-processT8*)

lemma *T-GlobalDet'*: ‹$\mathcal{T}$ (□ *x* ∈ *A*. *P x*) = (*insert* [] (⋃ *x*∈*A*. $\mathcal{T}$ (*P x*)))›
by (*simp add*: *T-GlobalDet*)
(*metis T-GlobalDet insert-absorb is-processT1-TR*)

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

lemma *mono-GlobalDet-eq*:
‹(⋀*x*. *x* ∈ *A* ⟹ *P x* = *Q x*) ⟹ *GlobalDet A P* = *GlobalDet A Q*›
by (*subst Process-eq-spec*, *simp add*: *F-GlobalDet D-GlobalDet*)

lemma *mono-GlobalDet-eq2*:
‹(⋀*x*. *x* ∈ *A* ⟹ *P* (*f x*) = *Q x*) ⟹ *GlobalDet* (*f* ‘ *A*) *P* = *GlobalDet A Q*›
by (*subst Process-eq-spec*, *simp add*: *F-GlobalDet D-GlobalDet*)

3.1.3 Factorization of (□) in front of *GlobalDet*

lemma *Process-eq-optimized-bisI* :
assumes ‹⋀*s*. *s* ∈ $\mathcal{D}$ *P* ⟹ *s* ∈ $\mathcal{D}$ *Q*› ‹⋀*s*. *s* ∈ $\mathcal{D}$ *Q* ⟹ *s* ∈ $\mathcal{D}$ *P*›
and ‹⋀*X*. $\mathcal{D}$ *P* = $\mathcal{D}$ *Q* ⟹ ([], *X*) ∈ $\mathcal{F}$ *P* ⟹ ([], *X*) ∈ $\mathcal{F}$ *Q*›
and ‹⋀*X*. $\mathcal{D}$ *Q* = $\mathcal{D}$ *P* ⟹ ([], *X*) ∈ $\mathcal{F}$ *Q* ⟹ ([], *X*) ∈ $\mathcal{F}$ *P*›
and ‹⋀*a s X*. $\mathcal{D}$ *P* = $\mathcal{D}$ *Q* ⟹ (*a* # *s*, *X*) ∈ $\mathcal{F}$ *P* ⟹ (*a* # *s*, *X*) ∈ $\mathcal{F}$ *Q*›
and ‹⋀*a s X*. $\mathcal{D}$ *Q* = $\mathcal{D}$ *P* ⟹ (*a* # *s*, *X*) ∈ $\mathcal{F}$ *Q* ⟹ (*a* # *s*, *X*) ∈ $\mathcal{F}$ *P*›
shows ‹*P* = *Q*›
proof (*subst Process-eq-spec-optimized*, *safe*)
show ‹*s* ∈ $\mathcal{D}$ *P* ⟹ *s* ∈ $\mathcal{D}$ *Q*› **for** *s* **by** (*fact assms*(*1*))
next
show ‹*s* ∈ $\mathcal{D}$ *Q* ⟹ *s* ∈ $\mathcal{D}$ *P*› **for** *s* **by** (*fact assms*(*2*))
next
show ‹$\mathcal{D}$ *P* = $\mathcal{D}$ *Q* ⟹ (*s*, *X*) ∈ $\mathcal{F}$ *P* ⟹ (*s*, *X*) ∈ $\mathcal{F}$ *Q*› **for** *s X*

by (*metis assms*(*3*, *5*) *neq-Nil-conv*)
next
show ‹$\mathcal{D}$ *P* = $\mathcal{D}$ *Q* ⟹ (*s*, *X*) ∈ $\mathcal{F}$ *Q* ⟹ (*s*, *X*) ∈ $\mathcal{F}$ *P*› **for** *s X*
by (*metis assms*(*4*, *6*) *neq-Nil-conv*)
qed

lemma *GlobalDet-factorization-union*:
‹(□ *p* ∈ *A*. *P p*) □ (□ *p* ∈ *B*. *P p*) = □ *p* ∈ (*A* ∪ *B*) . *P p*›
by (*rule Process-eq-optimized-bisI*)
(*auto simp add*: *D-Det D-GlobalDet F-Det F-GlobalDet T-GlobalDet split*: *if-split-asm*)

lemma *GlobalDet-Union* :
‹(□*a* ∈ (⋃*i* ∈ *I*. *A i*). *P a*) = □*i* ∈ *I*. □*a* ∈ *A i*. *P a*› (**is** ‹*?lhs* = *?rhs*›)
proof (*subst Process-eq-spec*, *safe*)
show ‹*s* ∈ $\mathcal{D}$ *?lhs* ⟹ *s* ∈ $\mathcal{D}$ *?rhs*›
and ‹*s* ∈ $\mathcal{D}$ *?rhs* ⟹ *s* ∈ $\mathcal{D}$ *?lhs*› **for** *s*
by (*auto simp add*: *D-GlobalDet*)
next
show ‹(*s*, *X*) ∈ $\mathcal{F}$ *?lhs* ⟹ (*s*, *X*) ∈ $\mathcal{F}$ *?rhs*› **for** *s X*
by (*cases s*) (*auto simp add*: *GlobalDet-projs*)
next
show ‹(*s*, *X*) ∈ $\mathcal{F}$ *?rhs* ⟹ (*s*, *X*) ∈ $\mathcal{F}$ *?lhs*› **for** *s X*
by (*cases s*; *simp add*: *GlobalDet-projs split*: *if-split-asm*) *blast+*
qed

3.1.4 First properties

lemma *GlobalDet-id* [*simp*] : ‹*A* ≠ {} ⟹ (□ *p* ∈ *A*. *P*) = *P*›
by (*auto simp add*: *Process-eq-spec F-GlobalDet D-GlobalDet*
intro: *is-processT8 is-processT6-TR-notin*)

lemma *GlobalDet-unit*[*simp*] : ‹(□ *x* ∈ {*a*}. *P x*) = *P a*›
by (*auto simp add*: *Process-eq-spec F-GlobalDet D-GlobalDet*
intro: *is-processT8 is-processT6-TR-notin*)

lemma *GlobalDet-empty*[*simp*] : ‹(□*a* ∈ {}. *P a*) = *STOP*›
by (*simp add*: *STOP-iff-T T-GlobalDet*)

lemma *GlobalDet-distrib-unit*:
‹(□ *x* ∈ *insert a A*. *P x*) = *P a* □ (□ *x* ∈ (*A* − {*a*}). *P x*)›
by (*metis GlobalDet-factorization-union GlobalDet-unit Un-Diff-cancel insert-is-Un*)

lemma *GlobalDet-distrib-unit-bis* :
‹*a* ∉ *A* ⟹ (□ *x* ∈ *insert a A*. *P x*) = *P a* □ (□ *x* ∈ *A*. *P x*)›

by (*simp add*: *GlobalDet-distrib-unit*)

3.1.5 Behaviour of *GlobalDet* with (□)

lemma *GlobalDet-Det-GlobalDet*:
‹(□ $a \in A$. $P\ a$) □ (□ $a \in A$. $Q\ a$) = □ $a \in A$. $P\ a$ □ $Q\ a$›
(**is** ‹*?G1* □ *?G2* = *?G*›)
proof (*subst Process-eq-spec*, *safe*)
show ‹$s \in \mathcal{D}$ (*?G1* □ *?G2*) $\Longrightarrow s \in \mathcal{D}$ *?G*›
and ‹$s \in \mathcal{D}$ *?G* $\Longrightarrow s \in \mathcal{D}$ (*?G1* □ *?G2*)› **for** *s*
by (*auto simp add*: *D-Det D-GlobalDet*)
next
show ‹(s, X) $\in \mathcal{F}$ (*?G1* □ *?G2*) $\Longrightarrow$ (s, X) $\in \mathcal{F}$ *?G*› **for** *s X*
by (*cases s*) (*auto simp add*: *F-Det D-Det T-Det D-GlobalDet T-GlobalDet′ F-GlobalDet*)
next
show ‹(s, X) $\in \mathcal{F}$ *?G* $\Longrightarrow$ (s, X) $\in \mathcal{F}$ (*?G1* □ *?G2*)› **for** *s X*
by (*cases s*; *simp add*: *F-Det D-Det T-Det D-GlobalDet T-GlobalDet′ F-GlobalDet*) *blast+*
qed

3.1.6 Commutativity

lemma *GlobalDet-sets-commute*:
‹(□ $a \in A$. □ $b \in B$. $P\ a\ b$) = □ $b \in B$. □ $a \in A$. $P\ a\ b$› (**is** ‹*?lhs* = *?rhs*›)
proof (*subst Process-eq-spec*, *safe*)
show ‹$s \in \mathcal{D}$ *?lhs* $\Longrightarrow s \in \mathcal{D}$ *?rhs*›
and ‹$s \in \mathcal{D}$ *?rhs* $\Longrightarrow s \in \mathcal{D}$ *?lhs*› **for** *s*
by (*auto simp add*: *D-GlobalDet*)
next
show ‹(s, X) $\in \mathcal{F}$ *?lhs* $\Longrightarrow$ (s, X) $\in \mathcal{F}$ *?rhs*› **for** *s X*
by (*cases s*; *simp add*: *F-GlobalDet T-GlobalDet′ D-GlobalDet split*: *if-split-asm*, *blast*)
next
show ‹(s, X) $\in \mathcal{F}$ *?rhs* $\Longrightarrow$ (s, X) $\in \mathcal{F}$ *?lhs*› **for** *s X*
by (*cases s*; *simp add*: *F-GlobalDet T-GlobalDet′ D-GlobalDet split*: *if-split-asm*, *blast*)
qed

3.1.7 Behaviour with injectivity

lemma *inj-on-mapping-over-GlobalDet*:
‹*inj-on f A* $\Longrightarrow$ (□ $x \in A$. $P\ x$) = □ $x \in f$ ‘ A. P (*inv-into A f x*)›
by (*simp add*: *Process-eq-spec F-GlobalDet D-GlobalDet*)

3.1.8 Cartesian product results

lemma *GlobalDet-cartprod-σs-set-σs-set*:
‹(□ (s, t) $\in A \times B$. P (s @ t)) = □ $u \in \{s$ @ t |$s\ t$. (s, t) $\in A \times B\}$. $P\ u$›
(**is** ‹*?lhs* = *?rhs*›)

proof (*subst Process-eq-spec*, *safe*)
show ‹s ∈ 𝒟 ?lhs ⟹ s ∈ 𝒟 ?rhs›
and ‹s ∈ 𝒟 ?rhs ⟹ s ∈ 𝒟 ?lhs› **for** *s*
by (*auto simp add*: *D-GlobalDet*)
next
show ‹(s, X) ∈ ℱ ?lhs ⟹ (s, X) ∈ ℱ ?rhs› **for** *s X*
by (*cases s*; *simp add*: *F-GlobalDet*, *blast*)
next
show ‹(s, X) ∈ ℱ ?rhs ⟹ (s, X) ∈ ℱ ?lhs› **for** *s X*
by (*cases s*; *simp add*: *F-GlobalDet*, *blast*)
qed

lemma *GlobalDet-cartprod-s-set-σs-set*:
‹(□ (s, t) ∈ A × B. P (s # t)) = □ u ∈ {s # t |s t. (s, t) ∈ A × B}. P u›
(**is** ‹?lhs = ?rhs›)
proof (*subst Process-eq-spec*, *safe*)
show ‹s ∈ 𝒟 ?lhs ⟹ s ∈ 𝒟 ?rhs›
and ‹s ∈ 𝒟 ?rhs ⟹ s ∈ 𝒟 ?lhs› **for** *s*
by (*auto simp add*: *D-GlobalDet*)
next
show ‹(s, X) ∈ ℱ ?lhs ⟹ (s, X) ∈ ℱ ?rhs› **for** *s X*
by (*cases s*; *simp add*: *F-GlobalDet*, *blast*)
next
show ‹(s, X) ∈ ℱ ?rhs ⟹ (s, X) ∈ ℱ ?lhs› **for** *s X*
by (*cases s*; *simp add*: *F-GlobalDet*, *blast*)
qed

lemma *GlobalDet-cartprod-s-set-s-set*:
‹(□ (s, t) ∈ A × B. P [s, t]) = □ u ∈ {[s, t] |s t. (s, t) ∈ A × B}. P u›
(**is** ‹?lhs = ?rhs›)
proof (*subst Process-eq-spec*, *safe*)
show ‹s ∈ 𝒟 ?lhs ⟹ s ∈ 𝒟 ?rhs›
and ‹s ∈ 𝒟 ?rhs ⟹ s ∈ 𝒟 ?lhs› **for** *s*
by (*auto simp add*: *D-GlobalDet*)
next
show ‹(s, X) ∈ ℱ ?lhs ⟹ (s, X) ∈ ℱ ?rhs› **for** *s X*
by (*cases s*; *simp add*: *F-GlobalDet*, *blast*)
next
show ‹(s, X) ∈ ℱ ?rhs ⟹ (s, X) ∈ ℱ ?lhs› **for** *s X*
by (*cases s*; *simp add*: *F-GlobalDet*, *blast*)
qed

lemma *GlobalDet-cartprod*: ‹(□(s, t) ∈ A × B. P s t) = □s ∈ A. □t ∈ B. P s t›
(**is** ‹?lhs = ?rhs›)
proof (*subst Process-eq-spec*, *safe*)

show ‹$s \in \mathcal{D}$ *?lhs* $\Longrightarrow$ $s \in \mathcal{D}$ *?rhs*›
and ‹$s \in \mathcal{D}$ *?rhs* $\Longrightarrow$ $s \in \mathcal{D}$ *?lhs*› **for** *s*
by (*auto simp add*: *D-GlobalDet*)
next
show ‹(s, X) $\in \mathcal{F}$ *?lhs* $\Longrightarrow$ (s, X) $\in \mathcal{F}$ *?rhs*› **for** *s X*
by (*cases s*) (*auto simp add*: *F-GlobalDet T-GlobalDet D-GlobalDet*)
next
show ‹(s, X) $\in \mathcal{F}$ *?rhs* $\Longrightarrow$ (s, X) $\in \mathcal{F}$ *?lhs*› **for** *s X*
by (*cases s*; *simp add*: *F-GlobalDet T-GlobalDet D-GlobalDet*
split: *if-split-asm*) *blast*
qed

3.1.9 Link with *Mprefix*

This is a trick to make proof of *Mprefix* using *GlobalDet* as it has an easier denotational definition.

lemma *Mprefix-GlobalDet*: ‹$\square$ $a \in A \rightarrow P\ a = \square\ a \in A.\ a \rightarrow P\ a$›
by (*simp add*: *Process-eq-spec write0-projs GlobalDet-projs Mprefix-projs*, *safe*, *auto*)

lemma *read-is-GlobalDet-write0* :
‹c**?**$a{\in}A \rightarrow P\ a = \square b{\in}c$ ‘ $A.\ b \rightarrow P$ (*inv-into A c b*)›
by (*simp add*: *read-def Mprefix-GlobalDet*)

lemma *read-is-GlobalDet-write* :
‹*inj-on c A* $\Longrightarrow$ c**?**$a{\in}A \rightarrow P\ a = \square a{\in}A.\ c$**!**$a \rightarrow P\ a$›
by (*auto simp add*: *read-is-GlobalDet-write0 write-def write0-def*
intro: *mono-GlobalDet-eq2*)

3.1.10 Properties

lemma *GlobalDet-Det*: ‹($\square$ $a \in A.\ P\ a$) $\square$ Q = (*if* A = {} *then* Q *else* $\square$ $a \in A.$ $P\ a \ \square\ Q$)›
(**is** ‹*?lhs* = (*if* A = {} *then* Q *else* *?rhs*)›)
proof (*split if-split*, *intro conjI impI*)
show ‹A = {} $\Longrightarrow$ *?lhs* = Q›
by (*auto simp add*: *Process-eq-spec F-Det F-STOP D-STOP T-STOP D-Det*
intro: *is-processT8 is-processT6-TR-notin*)
next
show ‹*?lhs* = *?rhs*› **if** ‹$A \neq$ {}›
proof (*subst Process-eq-spec*, *safe*)
show ‹$s \in \mathcal{D}$ *?lhs* $\Longrightarrow$ $s \in \mathcal{D}$ *?rhs*›
and ‹$s \in \mathcal{D}$ *?rhs* $\Longrightarrow$ $s \in \mathcal{D}$ *?lhs*› **for** *s*
by (*auto simp add*: ‹$A \neq$ {}› *D-Det D-GlobalDet*)
next
from ‹$A \neq$ {}› **show** ‹(s, X) $\in \mathcal{F}$ *?lhs* $\Longrightarrow$ (s, X) $\in \mathcal{F}$ *?rhs*› **for** *s X*
by (*cases s*) (*auto simp add*: *F-Det F-GlobalDet D-Det T-Det D-GlobalDet T-GlobalDet*)
next

from ‹$A \neq \{\}$› **show** ‹$(s, X) \in \mathcal{F}$ *?rhs* $\Longrightarrow (s, X) \in \mathcal{F}$ *?lhs*› **for** s X
by (*cases s*; *simp add*: *F-Det F-GlobalDet D-Det T-Det D-GlobalDet T-GlobalDet*, *blast*)
qed
qed

lemma *Mndetprefix-Sync-Mprefix-strong-subset*:
‹$[\![A \subseteq B; B \subseteq C]\!] \Longrightarrow \sqcap\ a \in A \to P\ a\ [\![\ C\ ]\!]\ \square\ b \in B \to Q\ b = \sqcap\ a \in A \to (P\ a\ [\![\ C\ ]\!]\ Q\ a)$›
by (*simp add*: *Mndetprefix-Sync-Mprefix-subset STOP-Sync-Mprefix Mprefix-is-STOP-iff*)

lemma *Mprefix-Sync-Mndetprefix-strong-subset*:
‹$[\![A \subseteq C; B \subseteq A]\!] \Longrightarrow \square\ a \in A \to P\ a\ [\![\ C\ ]\!] \sqcap\ b \in B \to Q\ b = \sqcap\ b \in B \to (P\ b\ [\![\ C\ ]\!]\ Q\ b)$›
by (*subst* (*1 2*) *Sync-commute*, *simp add*: *Mndetprefix-Sync-Mprefix-strong-subset*)

corollary *Mndetprefix-Par-Mprefix-strong-subset*:
‹$A \subseteq B \Longrightarrow \sqcap\ a \in A \to P\ a \mid\mid \square\ b \in B \to Q\ b = \sqcap\ a \in A \to (P\ a \mid\mid Q\ a)$›
by (*simp add*: *Mndetprefix-Sync-Mprefix-strong-subset*)

corollary *Mprefix-Par-Mndetprefix-strong-subset*:
‹$B \subseteq A \Longrightarrow \square\ a \in A \to P\ a \mid\mid \sqcap\ b \in B \to Q\ b = \sqcap\ b \in B \to (P\ b \mid\mid Q\ b)$›
by (*simp add*: *Mprefix-Sync-Mndetprefix-strong-subset*)

3.1.11 Continuity

lemma *mono-GlobalDet* : ‹$(\square a \in A.\ P\ a) \sqsubseteq \square a \in A.\ Q\ a$› **if** ‹$\bigwedge x.\ x \in A \Longrightarrow P\ x \sqsubseteq Q\ x$›
proof (*unfold le-approx-def*, *safe*)
show ‹$s \in \mathcal{D}\ (\square a \in A.\ Q\ a) \Longrightarrow s \in \mathcal{D}\ (\square a \in A.\ P\ a)$› **for** s
using *that*[*THEN le-approx1*] **by** (*auto simp add*: *D-GlobalDet*)
next
fix s X **assume** ‹$s \notin \mathcal{D}\ (\square a \in A.\ P\ a)$› ‹$X \in \mathcal{R}_a\ (\square a \in A.\ P\ a)\ s$›
from ‹$s \notin \mathcal{D}\ (\square a \in A.\ P\ a)$› **have** $*$: ‹$\forall\, a{\in}A.\ s \notin \mathcal{D}\ (P\ a)$›
by (*simp add*: *D-GlobalDet*)
with *that le-approx2*
have $**$: ‹$a \in A \Longrightarrow (s, X) \in \mathcal{F}\ (Q\ a) \longleftrightarrow (s, X) \in \mathcal{F}\ (P\ a)$› **for** a X **by** *blast*
from ‹$X \in \mathcal{R}_a\ (\square a \in A.\ P\ a)\ s$› $*$
consider ‹$s = []$› ‹$\bigwedge a.\ a \in A \Longrightarrow (s, X) \in \mathcal{F}\ (P\ a)$›
| e s' a **where** ‹$a \in A$› ‹$s = e \# s'$› ‹$(s, X) \in \mathcal{F}\ (P\ a)$›
| a r **where** ‹$a \in A$› ‹$s = []$› ‹$\checkmark(r) \notin X$› ‹$[\checkmark(r)] \in \mathcal{T}\ (P\ a)$›
by (*cases s*) (*auto simp add*: *Refusals-after-def F-GlobalDet*)
thus ‹$X \in \mathcal{R}_a\ (\square a \in A.\ Q\ a)\ s$›
proof *cases*
assume ‹$s = []$› ‹$\bigwedge a.\ a \in A \Longrightarrow (s, X) \in \mathcal{F}\ (P\ a)$›
from *this*(*2*) $**$ **have** ‹$\bigwedge a.\ a \in A \Longrightarrow (s, X) \in \mathcal{F}\ (Q\ a)$› **by** *simp*

with ‹$s = []$› **show** ‹$X \in \mathcal{R}_a$ ($\square a \in A.\ Q\ a$) s›
by (*simp add*: *Refusals-after-def F-GlobalDet*)
next
fix $e\ s'\ a$ **assume** ‹$a \in A$› ‹$s = e \# s'$› ‹$(s, X) \in \mathcal{F}\ (P\ a)$›
from ‹$(s, X) \in \mathcal{F}\ (P\ a)$› $**$ **have** ‹$(s, X) \in \mathcal{F}\ (Q\ a)$› **by** (*simp add*: ‹$a \in A$›)
with ‹$a \in A$› ‹$s = e \# s'$› **show** ‹$X \in \mathcal{R}_a$ ($\square a \in A.\ Q\ a$) s›
by (*auto simp add*: *Refusals-after-def F-GlobalDet*)
next
fix $a\ r$ **assume** ‹$a \in A$› ‹$s = []$› ‹✔(r) $\notin X$› ‹[✔(r)] $\in \mathcal{T}\ (P\ a)$›
from ‹$a \in A$› ‹[✔(r)] $\in \mathcal{T}\ (P\ a)$› **have** ‹[✔(r)] $\in \mathcal{T}\ (Q\ a)$›
by (*fold T-F-spec*, *simp add*: $**$[*OF* ‹$a \in A$›])
(*metis* $*$ ‹$s = []$› *is-processT9 proc-ord2a self-append-conv2 that*)
with ‹$a \in A$› ‹$s = []$› ‹✔(r) $\notin X$› **show** ‹$X \in \mathcal{R}_a$ ($\square a \in A.\ Q\ a$) s›
by (*auto simp add*: *Refusals-after-def F-GlobalDet*)
qed
next
fix $s\ X$ **assume** ‹$s \notin \mathcal{D}$ ($\square a \in A.\ P\ a$)› ‹$X \in \mathcal{R}_a$ ($\square a \in A.\ Q\ a$) s›
from ‹$s \notin \mathcal{D}$ ($\square a \in A.\ P\ a$)› **have** $*$: ‹$\forall\, a{\in}A.\ s \notin \mathcal{D}\ (P\ a)$›
by (*simp add*: *D-GlobalDet*)
with *that le-approx2*
have $**$: ‹$a \in A \Longrightarrow (s, X) \in \mathcal{F}\ (Q\ a) \longleftrightarrow (s, X) \in \mathcal{F}\ (P\ a)$› **for** $a\ X$ **by** *blast*
from ‹$X \in \mathcal{R}_a$ ($\square a \in A.\ Q\ a$) s›
consider ‹$s = []$› ‹$\bigwedge a.\ a \in A \Longrightarrow (s, X) \in \mathcal{F}\ (Q\ a)$›
| $e\ s'\ a$ **where** ‹$a \in A$› ‹$s = e \# s'$› ‹$(s, X) \in \mathcal{F}\ (Q\ a)$›
| a **where** ‹$a \in A$› ‹$s = []$› ‹$s \in \mathcal{D}\ (Q\ a)$›
| $a\ r$ **where** ‹$a \in A$› ‹$s = []$› ‹✔(r) $\notin X$› ‹[✔(r)] $\in \mathcal{T}\ (Q\ a)$›
by (*cases s*) (*auto simp add*: *Refusals-after-def F-GlobalDet*)
thus ‹$X \in \mathcal{R}_a$ ($\square a \in A.\ P\ a$) s›
proof *cases*
assume ‹$s = []$› ‹$\bigwedge a.\ a \in A \Longrightarrow (s, X) \in \mathcal{F}\ (Q\ a)$›
from *this*(*2*) $**$ **have** ‹$\bigwedge a.\ a \in A \Longrightarrow (s, X) \in \mathcal{F}\ (P\ a)$› **by** *simp*
with ‹$s = []$› **show** ‹$X \in \mathcal{R}_a$ ($\square a \in A.\ P\ a$) s›
by (*simp add*: *Refusals-after-def F-GlobalDet*)
next
fix $e\ s'\ a$ **assume** ‹$a \in A$› ‹$s = e \# s'$› ‹$(s, X) \in \mathcal{F}\ (Q\ a)$›
from ‹$(s, X) \in \mathcal{F}\ (Q\ a)$› $**$ **have** ‹$(s, X) \in \mathcal{F}\ (P\ a)$› **by** (*simp add*: ‹$a \in A$›)
with ‹$a \in A$› ‹$s = e \# s'$› **show** ‹$X \in \mathcal{R}_a$ ($\square a \in A.\ P\ a$) s›
by (*auto simp add*: *Refusals-after-def F-GlobalDet*)
next
show ‹$a \in A \Longrightarrow s = [] \Longrightarrow s \in \mathcal{D}\ (Q\ a) \Longrightarrow X \in \mathcal{R}_a$ ($\square a \in A.\ P\ a$) s› **for** a
by (*simp add*: *Refusals-after-def F-GlobalDet*)
(*meson le-approx1 subsetD that*)
next
fix $a\ r$ **assume** ‹$a \in A$› ‹$s = []$› ‹✔(r) $\notin X$› ‹[✔(r)] $\in \mathcal{T}\ (Q\ a)$›
from ‹$a \in A$› ‹[✔(r)] $\in \mathcal{T}\ (Q\ a)$› **have** ‹[✔(r)] $\in \mathcal{T}\ (P\ a)$›
by (*fold T-F-spec*, *simp add*: $**$[*OF* ‹$a \in A$›])
(*metis* $*$ ‹$s = []$› *is-processT9 proc-ord2a self-append-conv2 that*)
with ‹$a \in A$› ‹$s = []$› ‹✔(r) $\notin X$› **show** ‹$X \in \mathcal{R}_a$ ($\square a \in A.\ P\ a$) s›
by (*auto simp add*: *Refusals-after-def F-GlobalDet*)

qed
next
from *that*[*THEN le-approx3*]
show ‹$s \in$ *min-elems* ($\mathcal{D}$ ($\square a \in A.\ P\ a$)) $\Longrightarrow s \in \mathcal{T}$ ($\square a \in A.\ Q\ a$)› **for** *s*
by (*auto simp add*: *min-elems-def subset-iff less-list-def less-eq-list-def prefix-def D-GlobalDet T-GlobalDet*) *blast*
qed

lemma *chain-GlobalDet* : ‹*chain Y* $\Longrightarrow$ *chain* ($\lambda i.\ \square a \in A.\ Y\ i\ a$)›
by (*simp add*: *ch2ch-monofun fun-belowD mono-GlobalDet monofunI*)

lemma *GlobalDet-cont* [*simp*] : ‹⟦*finite A*; $\bigwedge a.\ a \in A \Longrightarrow$ *cont* ($P\ a$)⟧ $\Longrightarrow$ *cont* ($\lambda y.\ \square\ z{\in}A.\ P\ z\ y$)›
by (*induct A rule*: *finite-induct*)
(*simp-all add*: *GlobalDet-distrib-unit*)

end

3.2 Multiple Synchronization Product

3.2.1 Definition

As in the (⊓) case, we have no neutral element so we will also have to go through lists first. But the binary operator *Sync* is not idempotent either, so the generalization will be done on *'b multiset* and not on *'b set*.

Note that a *'b multiset* is by construction finite (cf. theorem *finite* (*set-mset M*)).

fun *MultiSync-list* :: ‹[*'a set*, *'b list*, *'b* ⇒ (*'a*, *'r*) $process_{ptick}$] ⇒ (*'a*, *'r*) $process_{ptick}$›
where ‹*MultiSync-list S* [] *P* = *STOP*›
| ‹*MultiSync-list S* (*l* # *L*) *P* = *fold* ($\lambda x\ r.\ r$ ⟦*S*⟧ *P x*) *L* (*P l*)›

interpretation *MultiSync*: *comp-fun-commute* **where** *f* = ‹$\lambda x\ r.\ r$ ⟦*E*⟧ *P x*›
unfolding *comp-fun-commute-def comp-fun-idem-axioms-def comp-def*
by (*metis Sync-assoc Sync-commute*)

lemma *MultiSync-list-mset*:
‹*mset L* = *mset L'* $\Longrightarrow$ *MultiSync-list S L P* = *MultiSync-list S L' P*›
proof (*cases L*, *simp-all*)
fix *a l*
assume ∗ : ‹*add-mset a* (*mset l*) = *mset L'*› **and** ∗∗ : ‹*L* = *a* # *l*›

then obtain *a′ l′* **where** ∗∗∗ : ‹ *L′* = *a′* # *l′*›
  by (*metis list.exhaust mset.simps*(*2*) *mset-zero-iff*)
 note ∗∗∗∗ = ∗[*simplified* ∗∗∗, *simplified*]
 have *a0*: ‹*a* ≠ *a′* ⟹ *MultiSync-list S L P* =
   fold (λ*x r*. *r* ⟦*S*⟧ *P x*) (*a′* # (*remove1 a′ l*)) (*P a*)›
  apply (*subst fold-multiset-equiv*[**where** *ys* = ‹*l*›])
   apply (*metis MultiSync.comp-fun-commute-axioms comp-fun-commute-def*)
  apply (*simp-all add*: ∗ ∗∗ ∗∗∗)
  by (*metis* ∗∗∗∗ *insert-DiffM insert-noteq-member*)
 have *a1*: ‹*a* ≠ *a′* ⟹ *MultiSync-list S L′ P* =
   fold (λ*x r*. *r* ⟦*S*⟧ *P x*) (*a* # (*remove1 a l′*)) (*P a′*)›
  apply (*subst fold-multiset-equiv*[**where** *ys* = ‹*l′*›])
   apply (*metis MultiSync.comp-fun-commute-axioms comp-fun-commute-def*)
  apply (*simp-all add*: ∗ ∗∗ ∗∗∗)
  by (*metis* ∗∗∗∗ *insert-DiffM insert-noteq-member*)
 from ∗∗∗∗ ∗∗ ∗∗∗ *a0 a1*
 show ‹*fold* (λ*x r*. *r* ⟦*S*⟧ *P x*) *l* (*P a*) = *MultiSync-list S L′ P*›
  apply (*cases* ‹*a* = *a′*›, *simp*)
   apply (*subst fold-multiset-equiv*[**where** *ys* = *l′*])
    apply (*metis MultiSync.comp-fun-commute-axioms comp-fun-commute-def*)
   apply (*simp-all*)
  apply (*subst fold-multiset-equiv*[**where** *ys* = ‹*remove1 a l′*›],
   simp-all add: *Sync-commute*)
   apply (*metis MultiSync.comp-fun-commute-axioms*
   comp-fun-commute.comp-fun-commute)
  by (*metis add-mset-commute add-mset-diff-bothsides*)
qed

definition *MultiSync* :: ‹[*′a set*, *′b multiset*, *′b* ⇒ (*′a*, *′r*) $process_{ptick}$] ⇒ (*′a*, *′r*) $process_{ptick}$›
 where ‹*MultiSync S M P* = *MultiSync-list S* (*SOME L*. *mset L* = *M*) *P*›

syntax *-MultiSync* :: ‹[*′a set*,*pttrn*,*′b multiset*,(*′a*, *′r*) $process_{ptick}$] ⇒ (*′a*, *′r*) *pro-*$cess_{ptick}$›
 (‹(*3*⟦-⟧ -∈#-./ -)› [*78*,*78*,*78*,*77*] *77*)
syntax-consts *-MultiSync* ⇌ *MultiSync*
translations ⟦*S*⟧ *p* ∈# *M*. *P* ⇌ *CONST MultiSync S M* (λ*p*. *P*)

Special case of *MultiSync E P* when *E* = {}.

abbreviation *MultiInter* :: ‹[*′b multiset*, *′b* ⇒ (*′a*, *′r*) $process_{ptick}$] ⇒ (*′a*, *′r*) $process_{ptick}$›
 where ‹*MultiInter M P* ≡ *MultiSync* {} *M P*›

syntax *-MultiInter* :: ‹[*pttrn*, *′b multiset*, (*′a*, *′r*) $process_{ptick}$] ⇒ (*′a*, *′r*) *pro-*$cess_{ptick}$›
 (‹(*3*||| -∈#-./ -)› [*78*,*78*,*77*] *77*)
syntax-consts *-MultiInter* ⇌ *MultiInter*
translations ||| *p* ∈# *M*. *P* ⇌ *CONST MultiInter M* (λ*p*. *P*)

Special case of *MultiSync E P* when *E* = *UNIV*.

abbreviation *MultiPar* :: ‹['b multiset, 'b ⇒ ('a, 'r) $process_{ptick}$] ⇒ ('a, 'r) $process_{ptick}$›
where ‹*MultiPar M P* ≡ *MultiSync UNIV M P*›

syntax *-MultiPar* :: ‹[pttrn, 'b multiset, ('a, 'r) $process_{ptick}$] ⇒ ('a, 'r) $process_{ptick}$›
(‹(3|| -∈#-. / -)› [78,78,77] 77)
syntax-consts *-MultiPar* ⇌ *MultiPar*
translations || *p* ∈# *M*. *P* ⇌ *CONST MultiPar M* (λ*p*. *P*)

3.2.2 First properties

lemma *MultiSync-rec0*[*simp*]: ‹(⟦*S*⟧ *p* ∈# {#}. *P p*) = *STOP*›
unfolding *MultiSync-def* **by** *simp*

lemma *MultiSync-rec1*[*simp*]: ‹(⟦*S*⟧ *p* ∈# {#*a*#}. *P p*) = *P a*›
unfolding *MultiSync-def* **apply**(*rule someI2-ex*) **by** *simp-all*

lemma *MultiSync-add*[*simp*]:
‹*M* ≠ {#} ⟹ (⟦*S*⟧ *p* ∈# *add-mset m M*. *P p*) = *P m* ⟦*S*⟧ (⟦*S*⟧ *p* ∈# *M*. *P p*)›
unfolding *MultiSync-def*
apply(*rule someI2-ex*, *simp add*: *ex-mset*)+
proof(*goal-cases*)
case (*1 x x'*)
thus ‹*MultiSync-list S x' P* = *P m* ⟦*S*⟧ *MultiSync-list S x P*›
apply (*subst MultiSync-list-mset*[**where** *L* = ‹*x'*› **and** *L'* = ‹*x* @ [*m*]›], *simp*)
by (*cases x*) (*simp-all add*: *Sync-commute*)
qed

lemma *mono-MultiSync-eq*:
‹(⋀*x*. *x* ∈# *M* ⟹ *P x* = *Q x*) ⟹ *MultiSync S M P* = *MultiSync S M Q*›
by (*cases* ‹*M* = {#}›, *simp*, *induct-tac rule*: *mset-induct-nonempty*) *auto*

lemma *mono-MultiSync-eq2*:
‹(⋀*x*. *x* ∈# *M* ⟹ *P* (*f x*) = *Q x*) ⟹ *MultiSync S* (*image-mset f M*) *P* = *MultiSync S M Q*›
by (*cases* ‹*M* = {#}›, *simp*, *induct-tac rule*: *mset-induct-nonempty*) *auto*

lemmas *MultiInter-rec0* = *MultiSync-rec0*[**where** *S* = ‹{}›]
and *MultiPar-rec0* = *MultiSync-rec0*[**where** *S* = ‹*UNIV*›]
and *MultiInter-rec1* = *MultiSync-rec1*[**where** *S* = ‹{}›]
and *MultiPar-rec1* = *MultiSync-rec1*[**where** *S* = ‹*UNIV*›]
and *MultiInter-add* = *MultiSync-add*[**where** *S* = ‹{}›]
and *MultiPar-add* = *MultiSync-add*[**where** *S* = ‹*UNIV*›]

and *mono-MultiInter-eq* = *mono-MultiSync-eq*[**where** *S* = ‹{}›]
and *mono-MultiPar-eq* = *mono-MultiSync-eq*[**where** *S* = ‹*UNIV*›]
and *mono-MultiInter-eq2* = *mono-MultiSync-eq2*[**where** *S* = ‹{}›]
and *mono-MultiPar-eq2* = *mono-MultiSync-eq2*[**where** *S* = ‹*UNIV*›]

3.2.3 Some Tests

lemma ‹*MultiSync-list S* [] *P* = *STOP*›
and ‹*MultiSync-list S* [*a*] *P* = *P a*›
and ‹*MultiSync-list S* [*a*, *b*] *P* = *P a* ⟦*S*⟧ *P b*›
and ‹*MultiSync-list S* [*a*, *b*, *c*] *P* = *P a* ⟦*S*⟧ *P b* ⟦*S*⟧ *P c*›
by *simp+*

lemma *test-MultiSync*:
‹(⟦*S*⟧ *p* ∈# *mset* []. *P p*) = *STOP*›
‹(⟦*S*⟧ *p* ∈# *mset* [*a*]. *P p*) = *P a*›
‹(⟦*S*⟧ *p* ∈# *mset* [*a*, *b*]. *P p*) = *P a* ⟦*S*⟧ *P b*›
‹(⟦*S*⟧ *p* ∈# *mset* [*a*, *b*, *c*]. *P p*) = *P a* ⟦*S*⟧ *P b* ⟦*S*⟧ *P c*›
by (*simp-all add*: *Sync-assoc*)

lemma *MultiSync-set1*: ‹*MultiSync S* (*mset-set* {*k*::*nat*..<*k*}) *P* = *STOP*›
by *fastforce*

lemma *MultiSync-set2*: ‹*MultiSync S* (*mset-set* {*k*..<*Suc k*}) *P* = *P k*›
by *fastforce*

lemma *MultiSync-set3*:
‹*l* < *k* ⟹ *MultiSync S* (*mset-set* {*l* ..< *Suc k*}) *P* =
P l ⟦*S*⟧ (*MultiSync S* (*mset-set* {*Suc l* ..< *Suc k*}) *P*)›
by (*simp add*: *Icc-eq-insert-lb-nat atLeastLessThanSuc-atLeastAtMost*)

lemma *test-MultiSync′*:
‹(⟦*S*⟧ *p* ∈# *mset-set* {*1*::*int* .. *3*}. *P p*) = *P 1* ⟦*S*⟧ *P 2* ⟦*S*⟧ *P 3*›
proof −
have ‹{*1*::*int* .. *3*} = *insert 1* (*insert 2* (*insert 3* {}))› **by** *fastforce*
thus ‹(⟦*S*⟧ *p* ∈# *mset-set* {*1*::*int* .. *3*}. *P p*) = *P 1* ⟦*S*⟧ *P 2* ⟦*S*⟧ *P 3*› **by** (*simp add*: *Sync-assoc*)
qed

lemma *test-MultiSync″*:
‹(⟦*S*⟧ *p* ∈# *mset-set* {*0*::*nat* .. *a*}. *P p*) =
⟦*S*⟧ *p* ∈# *mset-set* ({*a*} ∪ {*1* .. *a*} ∪ {*0*}) . *P p*›
by (*metis Un-insert-right atMost-atLeast0 boolean-algebra-cancel.sup0*

image-Suc-lessThan insert-absorb2 insert-is-Un lessThan-Suc lessThan-Suc-atMost lessThan-Suc-eq-insert-0)

lemmas *test-MultiInter* = *test-MultiSync*[**where** S = ‹{}›]
and *test-MultiPar* = *test-MultiSync*[**where** S = ‹*UNIV*›]
and *MultiInter-set1* = *MultiSync-set1*[**where** S = ‹{}›]
and *MultiPar-set1* = *MultiSync-set1*[**where** S = ‹*UNIV*›]
and *MultiInter-set2* = *MultiSync-set2*[**where** S = ‹{}›]
and *MultiPar-set2* = *MultiSync-set2*[**where** S = ‹*UNIV*›]
and *MultiInter-set3* = *MultiSync-set3*[**where** S = ‹{}›]
and *MultiPar-set3* = *MultiSync-set3*[**where** S = ‹*UNIV*›]
and *test-MultiInter′* = *test-MultiSync′*[**where** S = ‹{}›]
and *test-MultiPar′* = *test-MultiSync′*[**where** S = ‹*UNIV*›]
and *test-MultiInter″* = *test-MultiSync″*[**where** S = ‹{}›]
and *test-MultiPar″* = *test-MultiSync″*[**where** S = ‹*UNIV*›]

3.2.4 Continuity

lemma *mono-MultiSync* :
‹($\bigwedge x.\ x \in\# M \Longrightarrow P\ x \sqsubseteq Q\ x) \Longrightarrow$ (⟦*S*⟧ $m \in\# M.\ P\ m) \sqsubseteq$ (⟦*S*⟧ $m \in\# M.\ Q\ m)$›
by (*cases* ‹M = {#}›, *simp*, *erule mset-induct-nonempty*, *simp-all add*: *mono-Sync*)

lemmas *mono-MultiInter* = *mono-MultiSync*[**where** S = ‹{}›]
and *mono-MultiPar* = *mono-MultiSync*[**where** S = *UNIV*]

lemma *MultiSync-cont*[*simp*]:
‹($\bigwedge x.\ x \in\# M \Longrightarrow$ *cont* $(P\ x)) \Longrightarrow$ *cont* ($\lambda y.$ ⟦*S*⟧ $z \in\# M.\ P\ z\ y$)›
by (*cases* ‹M = {#}›, *simp*, *erule mset-induct-nonempty*, *simp*+)

lemmas *MultiInter-cont*[*simp*] = *MultiSync-cont*[**where** S = ‹{}›]
and *MultiPar-cont*[*simp*] = *MultiSync-cont*[**where** S = ‹*UNIV*›]

3.2.5 Factorization of *Sync* in front of *MultiSync*

lemma *MultiSync-factorization-union*:
‹⟦$M \neq$ {#}; $N \neq$ {#}⟧ $\Longrightarrow$
(⟦*S*⟧ $z \in\# M.\ P\ z$) ⟦*S*⟧ (⟦*S*⟧ $z \in\# N.\ P\ z$) = ⟦*S*⟧ $z \in\# (M + N).\ P\ z$›
by (*erule mset-induct-nonempty*, *simp-all add*: *Sync-assoc*[*symmetric*])

lemmas *MultiInter-factorization-union* =
MultiSync-factorization-union[**where** S = ‹{}›]
and *MultiPar-factorization-union* =
MultiSync-factorization-union[**where** S = ‹*UNIV*›]

3.2.6 ⊥ Absorbtion

lemma *MultiSync-BOT-absorb*:
‹$m \in\# M \Longrightarrow P\ m = \bot \Longrightarrow ([\![S]\!]\ z \in\# M.\ P\ z) = \bot$›
by (*metis MultiSync-add MultiSync-rec1 mset-add Sync-BOT Sync-commute*)

lemmas *MultiInter-BOT-absorb* = *MultiSync-BOT-absorb*[**where** S = ‹{}›]
and *MultiPar-BOT-absorb* = *MultiSync-BOT-absorb*[**where** S = ‹*UNIV*›]

lemma *MultiSync-is-BOT-iff*:
‹$([\![S]\!]\ m \in\# M.\ P\ m) = \bot \longleftrightarrow (\exists\, m \in\# M.\ P\ m = \bot)$›
apply (*cases* ‹$M = \{\#\}$›, *simp add*: *BOT-iff-Nil-D D-STOP*)
by (*rotate-tac*, *induct M rule*: *mset-induct-nonempty*, *auto simp add*: *Sync-is-BOT-iff*)

lemmas *MultiInter-is-BOT-iff* = *MultiSync-is-BOT-iff*[**where** S = ‹{}›]
and *MultiPar-is-BOT-iff* = *MultiSync-is-BOT-iff*[**where** S = ‹*UNIV*›]

3.2.7 Other Properties

lemma *MultiSync-SKIP-id*:
‹$([\![S]\!]\ r \in\# M.\ SKIP\ r)$ = (*if* $\exists\, r.$ *set-mset* $M = \{r\}$ *then SKIP* (*THE* r. *set-mset* $M = \{r\}$) *else STOP*)›
apply (*cases* ‹$M = \{\#\}$›, *simp*)
apply (*induct M rule*: *mset-induct-nonempty*, *simp*)
by (*simp add*: *subset-singleton-iff split*: *if-splits*)

lemmas *MultiInter-SKIP-id* = *MultiSync-SKIP-id*[**where** S = ‹{}›]
and *MultiPar-SKIP-id* = *MultiSync-SKIP-id*[**where** S = ‹*UNIV*›]

lemma *MultiPar-prefix-two-distincts-STOP*:
assumes ‹$m \in\# M$› **and** ‹$m' \in\# M$› **and** ‹*fst* $m \neq$ *fst* m'›
shows ‹($||$ $a \in\# M$. (*fst* $a \to P$ (*snd* a))) = *STOP*›
proof −
obtain M' **where** *f2*: ‹M = *add-mset* m (*add-mset* m' M')›
by (*metis diff-union-swap insert-DiffM assms*)
show ‹($||$ $x \in\# M$. (*fst* $x \to P$ (*snd* x))) = *STOP*›
apply (*simp add*: *f2*, *cases* ‹$M' = \{\#\}$›, *simp add*: *assms*(*3*) *write0-Par-write0*)
apply (*induct* M' *rule*: *mset-induct-nonempty*)
apply (*simp add*: *Sync-commute assms*(*3*) *write0-Par-write0*)
by *simp* (*metis* (*no-types*, *lifting*) *STOP-Sync-write0 Sync-assoc Sync-commute UNIV-I*)
qed

lemma *MultiPar-prefix-two-distincts-STOP′*:
‹⟦(m, n) ∈# M; (m′, n′) ∈# M; m ≠ m′⟧ ⟹
(|| (m, n) ∈# M. (m → P n)) = STOP›
apply (*subst cond-case-prod-eta*[**where** g = ‹λ x. (*fst* x → P (*snd* x))›])
by (*simp-all add*: *MultiPar-prefix-two-distincts-STOP*)

3.2.8 Behaviour of *MultiSync* with *Sync*

lemma *MultiSync-Sync*:
‹(⟦S⟧ z ∈# M. P z) ⟦S⟧ (⟦S⟧ z ∈# M. P′ z) = ⟦S⟧ z ∈# M. (P z ⟦S⟧ P′ z)›
apply (*cases* ‹M = {#}›, *simp*)
apply (*induct M rule*: *mset-induct-nonempty*)
by *simp-all* (*metis* (*no-types*, *lifting*) *Sync-assoc Sync-commute*)

lemmas *MultiInter-Inter* = *MultiSync-Sync*[**where** S = ‹{}›]
and *MultiPar-Par* = *MultiSync-Sync*[**where** S = ‹UNIV›]

3.2.9 Commutativity

lemma *MultiSync-sets-commute*:
‹(⟦S⟧ a ∈# M. ⟦S⟧ b ∈# N. P a b) = ⟦S⟧ b ∈# N. ⟦S⟧ a ∈# M. P a b›
apply (*cases* ‹N = {#}›, *induct M*, *simp-all*,
metis MultiSync-add MultiSync-rec1 STOP-Sync-STOP)
apply (*induct N rule*: *mset-induct-nonempty*, *fastforce*)
by *simp* (*metis MultiSync-Sync*)

lemmas *MultiInter-sets-commute* = *MultiSync-sets-commute*[**where** S = ‹{}›]
and *MultiPar-sets-commute* = *MultiSync-sets-commute*[**where** S = ‹UNIV›]

3.2.10 Behaviour with Injectivity

lemma *inj-on-mapping-over-MultiSync*:
‹*inj-on* f (*set-mset* M) ⟹
(⟦S⟧ x ∈# M. P x) = ⟦S⟧ x ∈# *image-mset* f M. P (*inv-into* (*set-mset* M) f x)›
proof (*induct M rule*: *induct-subset-mset-empty-single*, *simp*, *simp*)
case (*3 N a*)
hence *f1*: ‹*inv-into* (*insert* a (*set-mset* N)) f (f a) = a› **by** *force*
show *?case*
apply (*simp add*: *3.hyps*(*2*) *3.hyps*(*3*) *f1*,
rule arg-cong[**where** f = ‹λx. P a ⟦S⟧ x›])
apply (*subst 3.hyps*(*4*), *rule inj-on-subset*[*OF 3.prems*],
simp add: *subset-insertI*)
apply (*rule mono-MultiSync-eq*)
using *3.prems* **by** *fastforce*
qed

lemmas *inj-on-mapping-over-MultiInter* =

inj-on-mapping-over-MultiSync[**where** $S = \langle\{\}\rangle$]
and *inj-on-mapping-over-MultiPar* =
inj-on-mapping-over-MultiSync[**where** $S = \langle UNIV \rangle$]

3.3 Multiple Sequential Composition

Because of the fact that *SKIP r* is not exactly a neutral element for *Seq* (cf *SKIP ?r* **;** *?P* = *?P*
?P **;** *Skip* = *?P*), we do the folding on the reversed list.

3.3.1 Definition

fun *MultiSeq-rev* :: ‹['b list, 'b ⇒ ('a, 'r) $process_{ptick}$] ⇒ ('a, 'r) $process_{ptick}$›
where *MultiSeq-rev-Nil* : ‹*MultiSeq-rev* [] *P* = *SKIP undefined*›
| *MultiSeq-rev-Cons* : ‹*MultiSeq-rev* (*l* # *L*) *P* = *MultiSeq-rev L P* **;** *P l*›

definition *MultiSeq* :: ‹['b list, 'b ⇒ ('a, 'r) $process_{ptick}$] ⇒ ('a, 'r) $process_{ptick}$›
where ‹*MultiSeq L P* ≡ *MultiSeq-rev* (*rev L*) *P*›

lemma *MultiSeq-Nil* [*simp*] : ‹*MultiSeq* [] *P* = *SKIP undefined*›
and *MultiSeq-snoc* [*simp*] : ‹*MultiSeq* (*L* @ [*l*]) *P* = *MultiSeq L P* **;** *P l*›
by (*simp-all add*: *MultiSeq-def*)

lemma *MultiSeq-elims* :
‹*MultiSeq L P* = *Q* ⟹
(⋀*P'*. *L* = [] ⟹ *P* = *P'* ⟹ *Q* = *SKIP undefined* ⟹ *thesis*) ⟹
(⋀*l L' P'*. *L* = *L'* @ [*l*] ⟹ *P* = *P'* ⟹ *Q* = *MultiSeq L' P'* **;** *P' l* ⟹ *thesis*)
⟹ *thesis*›
by (*simp add*: *MultiSeq-def*, *erule MultiSeq-rev.elims*, *simp-all*)

syntax *-MultiSeq* :: ‹[pttrn, 'b list, 'b ⇒ 'r ⇒ ('a, 'r) $process_{ptick}$, 'r] ⇒ ('a, 'r) $process_{ptick}$›
(‹(3SEQ -∈@-./ -)› [78,78,77] 77)
syntax-consts *-MultiSeq* ⇌ *MultiSeq*
translations *SEQ p* ∈@ *L*. *P* ⇌ *CONST MultiSeq L* (λ*p*. *P*)

3.3.2 First Properties

lemma ‹*SEQ p* ∈@ []. *P p* = *SKIP undefined*› **by** (*fact MultiSeq-Nil*)

lemma ‹*SEQ i* ∈@ (*L* @ [*l*]). *P i* = *SEQ i* ∈@ *L*. *P i* **;** *P l*› **by** (*fact MultiSeq-snoc*)

lemma *MultiSeq-singl* [*simp*] : ‹*SEQ l* ∈@ [*l*]. *P l* = *P l*› **by** (*simp add*: *MultiSeq-def*)

3.3.3 Some Tests

lemma ‹*SEQ p* ∈@ []. *P p* = *SKIP undefined*›
 and ‹*SEQ p* ∈@ [*a*]. *P p* = *P a*›
 and ‹*SEQ p* ∈@ [*a*, *b*]. *P p* = *P a* **;** *P b*›
 and ‹*SEQ p* ∈@ [*a*, *b*, *c*]. *P p* = *P a* **;** *P b* **;** *P c*›
 by (*simp-all add*: *MultiSeq-def*)

lemma *test-MultiSeq*: ‹(*SEQ p* ∈@ [*1*::*int* .. *3*]. *P p*) = *P 1* **;** *P 2* **;** *P 3*›
 by (*simp add*: *upto.simps MultiSeq-def*)

3.3.4 Continuity

lemma *mono-MultiSeq* :
 ‹(⋀*x*. *x* ∈ *set L* ⟹ *P x* ⊑ *Q x*) ⟹ *SEQ l* ∈@ *L*. *P l* ⊑ *SEQ l* ∈@ *L*. *Q l*›
 by (*induct L rule*: *rev-induct*, *simp-all add*: *fun-belowI mono-Seq*)

lemma *MultiSeq-cont*[*simp*]:
 ‹(⋀*x*. *x* ∈ *set L* ⟹ *cont* (*P x*)) ⟹ *cont* (λ*y*. *SEQ z* ∈@ *L*. *P z y*)›
 by (*induct L rule*: *rev-induct*) *simp-all*

3.3.5 Factorization of (;) in front of *MultiSeq*

lemma *MultiSeq-factorization-append*:
 ‹*L2* ≠ [] ⟹ *SEQ p* ∈@ *L1*. *P p* **;** *SEQ p* ∈@ *L2*. *P p* = *SEQ p* ∈@ (*L1* @ *L2*). *P p*›
 by (*induct L2 rule*: *rev-induct*, *simp-all*)
  (*metis* (*no-types*, *lifting*) *MultiSeq-singl MultiSeq-snoc*
   Seq-assoc append-assoc append-self-conv2)

3.3.6 ⊥ Absorbtion

lemma *MultiSeq-BOT-absorb*:
 ‹*SEQ z* ∈@ (*L1* @ *a* # *L2*). *P z* = *SEQ z* ∈@ *L1*. *P z* **;** ⊥› **if** ‹*P a* = ⊥›
proof (*cases* ‹*L2* = []›)
 from ‹*P a* = ⊥› **show** ‹*L2* = [] ⟹ *MultiSeq* (*L1* @ *a* # *L2*) *P* = *MultiSeq L1 P* **;** ⊥› **by** *simp*
next
 show ‹*L2* ≠ [] ⟹ *MultiSeq* (*L1* @ *a* # *L2*) *P* = *MultiSeq L1 P* **;** ⊥›
  by (*simp add*: ‹*P a* = ⊥› *flip*: *Seq-assoc MultiSeq-factorization-append*
   [*of L2* ‹*L1* @ [*a*]›, *simplified*])
qed

3.3.7 First Properties

lemma *MultiSeq-SKIP-neutral*:

‹*SEQ z* ∈@ (*L1* @ *a* # *L2*). *P z* =
(*if L2* = [] *then SEQ z* ∈@ *L1*. *P z* **;** *SKIP r*
else SEQ z ∈@ (*L1* @ *L2*). *P z*)› **if** ‹*P a* = *SKIP r*›
proof (*split if-split*, *intro conjI impI*)
show ‹*L2* = [] ⟹ *MultiSeq* (*L1* @ *a* # *L2*) *P* = *MultiSeq L1 P* **;** *SKIP r*›
by (*simp add*: ‹*P a* = *SKIP r*›)
next
assume ‹*L2* ≠ []›
have ‹*MultiSeq* (*L1* @ *a* # *L2*) *P* = *MultiSeq L1 P* **;** *P a* **;** *MultiSeq L2 P*›
by (*metis* (*mono-tags*, *opaque-lifting*) *Cons-eq-appendI MultiSeq-factorization-append*
MultiSeq-snoc ‹*L2* ≠ []› *append-eq-appendI self-append-conv2*)
also have ‹... = *MultiSeq L1 P* **;** *MultiSeq L2 P*›
by (*simp add*: ‹*P a* = *SKIP r*› *flip*: *Seq-assoc*)
also have ‹... = *MultiSeq* (*L1* @ *L2*) *P*›
by (*simp add*: *MultiSeq-factorization-append* ‹*L2* ≠ []›)
finally show ‹*MultiSeq* (*L1* @ *a* # *L2*) *P* = *MultiSeq* (*L1* @ *L2*) *P*› **.**
qed

lemma *MultiSeq-STOP-absorb*:
‹*SEQ z* ∈@ (*L1* @ *a* # *L2*). *P z* = *SEQ z* ∈@ *L1*. *P z* **;** *STOP*› **if** ‹*P a* = *STOP*›
proof (*cases* ‹*L2* = []›)
show ‹*L2* = [] ⟹ *MultiSeq* (*L1* @ *a* # *L2*) *P* = *MultiSeq L1 P* **;** *STOP*›
by (*simp add*: ‹*P a* = *STOP*›)
next
show ‹*L2* ≠ [] ⟹ *MultiSeq* (*L1* @ *a* # *L2*) *P* = *MultiSeq L1 P* **;** *STOP*›
by (*simp add*: ‹*P a* = *STOP*› *flip*: *Seq-assoc MultiSeq-factorization-append*
[*of L2* ‹*L1* @ [*a*]›, *simplified*])
qed

lemma *mono-MultiSeq-eq*:
‹(⋀*x*. *x* ∈ *set L* ⟹ *P x* = *Q x*) ⟹ *MultiSeq L P* = *MultiSeq L Q*›
by (*induct L rule*: *rev-induct*) *simp-all*

3.3.8 Commutativity

Of course, since the sequential composition *P* **;** *Q* is not commutative, the result here is negative: the order of the elements of list *L* does matter in *SEQ z*∈@*L*. *P z*.

3.3.9 Behaviour with Injectivity

lemma *inj-on-mapping-over-MultiSeq*:
‹*inj-on f* (*set C*) ⟹
SEQ x ∈@ *C*. *P x* = *SEQ x* ∈@ *map f C*. *P* (*inv-into* (*set C*) *f x*)›
proof (*induct C rule*: *rev-induct*)
show ‹*inj-on f* (*set* []) ⟹ *MultiSeq* [] *P* =
SEQ x∈@*map f* []. *P* (*inv-into* (*set* []) *f x*)› **by** *simp*

next
  case (*snoc a C*)
  hence *f1*: ‹*inv-into* (*insert a* (*set C*)) *f* (*f a*) = *a*› **by** *force*
  show *?case*
    apply (*simp add*: *f1*, *intro ext arg-cong*[**where** *f* = ‹λ*x*. *x* **;** *P a*›])
    apply (*subst snoc.hyps*(*1*), *rule inj-on-subset*[*OF snoc.prems*],
        simp add: *subset-insertI*)
    using *snoc.prems* **by** (*auto intro*!: *mono-MultiSeq-eq*)
qed

3.3.10 Definition of *first-elem*

primrec *first-elem* :: ‹['*a* ⇒ *bool*, '*a list*] ⇒ *nat*›
  where ‹*first-elem P* [] = *0*›
  | ‹*first-elem P* (*x* # *L*) = (*if P x then 0 else Suc* (*first-elem P L*))›

first-elem returns the first index i such that $P\ (L\ !\ i) = True$ if it exists, *length L* otherwise.

This will be very useful later.

value ‹*first-elem* (λ*x*. *4* < *x*) [*0*::*nat*, *2*, *5*]›
lemma ‹*first-elem* (λ*x*. *5* < *x*) [*0*::*nat*, *2*, *5*] = *3*› **by** *simp*
lemma ‹*P* ‘ *set L* ⊆ {*False*} ⟹ *first-elem P L* = *length L*› **by** (*induct L*; *simp*)

3.4 The Throw Operator

3.4.1 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 [3, p.140]).

lift-definition *Throw* :: ‹[('*a*, '*r*) $process_{ptick}$, '*a set*, '*a* ⇒ ('*a*, '*r*) $process_{ptick}$] ⇒ ('*a*, '*r*) $process_{ptick}$›
  is ‹λ *P A Q*.
  ({(*t1*, *X*). (*t1*, *X*) ∈ 𝓕 *P* ∧ *set t1* ∩ *ev* ‘ *A* = {}} ∪
   {(*t1* @ *t2*, *X*) |*t1 t2 X*. *t1* ∈ 𝓓 *P* ∧ *tF t1* ∧ *set t1* ∩ *ev* ‘ *A* = {} ∧ *ftF 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* ∧ *tF t1* ∧ *set t1* ∩ *ev* ‘ *A* = {} ∧ *ftF 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 ⟹ ftF s› **for** s X
      apply (simp, elim disjE exE)
        apply (metis is-processT)
       apply (simp add: front-tickFree-append)
       by (metis F-imp-front-tickFree T-nonTickFree-imp-decomp append1-eq-conv $event_{ptick}$.distinct(1)
        front-tickFree-Cons-iff front-tickFree-append tickFree-Cons-iff tickFree-append-iff)
  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 **where** ‹s @ t @ [b] = t1 @ t2› ‹t1 ∈ 𝓓 P› ‹tF t1› ‹set t1 ∩ ev ‘ A = {}› ‹ftF t2›
        | t1 a t2 **where** ‹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** auto
      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
        show ‹⟦s @ t @ [b] = t1 @ t2; t1 ∈ 𝓓 P; tF t1; set t1 ∩ ev ‘ A = {}; ftF t2⟧
            ⟹ (s, {}) ∈ ?f› **for** t1 t2
          by (rule snoc.hyps, cases t2 rule: rev-cases, simp-all)
          (metis (no-types, opaque-lifting) Int-Un-distrib2 append-assoc is-processT3
            is-processT8 set-append sup.idem sup-bot.right-neutral,
            metis front-tickFree-dw-closed)
      next
        show ‹⟦s @ t @ [b] = t1 @ ev a # t2; t1 @ [ev a] ∈ 𝓣 P; set t1 ∩ ev ‘ A = {};
             a ∈ A; (t2, {}) ∈ 𝓕 (Q a)⟧ ⟹ (s, {}) ∈ ?f› **for** t1 a t2
          by (rule snoc.hyps, cases t2 rule: rev-cases, simp-all)
            (metis T-F is-processT3, metis is-processT3)
      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   where ‹s = t1 @ t2› ‹t1 ∈ 𝒟 P› ‹tF t1› ‹set t1 ∩ ev ‘ A = {}› ‹ftF
t2›
      | 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)›
      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
      show ‹⟦s = t1 @ t2; t1 ∈ 𝒟 P; tF t1; set t1 ∩ ev ‘ A = {}; ftF t2⟧
         ⟹ (s, X ∪ Y) ∈ ?f› for t1 t2 by blast
    next
      fix t1 a t2
      assume * : ‹s = t1 @ ev a # t2› ‹t1 @ [ev a] ∈ 𝒯 P›
        ‹set t1 ∩ ev ‘ A = {}› ‹a ∈ A› ‹(t2, X) ∈ ℱ (Q a)›
      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 r. s @ [✓(r)] = t1 @ ev a # t2 ⟹ ∃ t2'. t2 = t2' @
[✓(r)]›
      by (simp add: snoc-eq-iff-butlast split: if-split-asm)
        (metis append-butlast-last-id)
    show ‹(s @ [✓(r)], {}) ∈ ?f ⟹ (s, X − {✓(r)}) ∈ ?f› for s r X
      apply (simp, elim disjE exE conjE)
        apply (solves ‹simp add: is-processT6›)
       apply (metis append1-eq-conv append-assoc front-tickFree-dw-closed
          nonTickFree-n-frontTickFree non-tickFree-tick tickFree-append-iff)
```

by (*frule* ∗, *elim exE*, *simp*, *metis is-processT6*)
next
show ‹⟦s ∈ *?d*; *tF s*; *ftF t*⟧ ⟹ s @ t ∈ *?d*› **for** *s t*
by (*simp*, *elim disjE*)
(*meson append-assoc front-tickFree-append tickFree-append-iff*,
use append-self-conv2 is-processT7 tickFree-append-iff **in** *fastforce*)
next
show ‹s ∈ *?d* ⟹ (s, X) ∈ *?f*› **for** *s X*
by *simp* (*metis is-processT8*)
next
show ‹s @ [✓(r)] ∈ *?d* ⟹ s ∈ *?d*› **for** *s r*
by (*simp*, *elim disjE*)
(*metis butlast-append butlast-snoc front-tickFree-iff-tickFree-butlast non-tickFree-tick tickFree-Nil tickFree-append-iff tickFree-imp-front-tickFree*,
metis (*no-types*, *lifting*) *append-butlast-last-id butlast.simps*(*2*) *butlast-append butlast-snoc* $event_{ptick}$*.distinct*(*1*) *is-processT9 last.simps last-appendR list.distinct*(*1*))
qed
qed

We add some syntactic sugar.

syntax *-Throw* :: ‹[('*a*, '*r*) $process_{ptick}$, *pttrn*, '*a set*, '*a* ⇒ ('*a*, '*r*) $process_{ptick}$] ⇒ ('*a*, '*r*) $process_{ptick}$›
(‹((-) Θ (-∈-)./ (-))› [*78*,*78*,*78*,*77*] *77*)
syntax-consts *-Throw* ⇌ *Throw*
translations *P* Θ *a* ∈ *A*. *Q* ⇌ *CONST Throw P A* (λ*a*. *Q*)

3.4.2 Projections

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

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

lemma *T-Throw*:
‹$\mathcal{T}$ (*P* Θ *a* ∈ *A*. *Q a*) =
{*t1* ∈ $\mathcal{T}$ *P*. *set t1* ∩ *ev* ‘ *A* = {}} ∪

{t1 @ t2 |t1 t2. t1 ∈ 𝒟 P ∧ tF t1 ∧ set t1 ∩ ev ‘ A = {} ∧ ftF 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.rep-eq TRACES-def Failures.rep-eq*[*symmetric*] *F-Throw*)
blast+

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

lemma *Throw-T-third-clause-breaker* :
‹⟦*set t* ∩ *S* = {}; *set t′* ∩ *S* = {}; *e* ∈ *S*; *e′* ∈ *S*⟧ ⟹
t @ *e* # *u* = *t′* @ *e′* # *u′* ⟷ *t* = *t′* ∧ *e* = *e′* ∧ *u* = *u′*›
proof (*induct t arbitrary*: *t′*)
case *Nil* **thus** *?case*
by (*metis append-Nil disjoint-iff hd-append2 list.sel*(*1*, *3*) *list.set-sel*(*1*))
next
case (*Cons a t*)
show *?case*
proof (*rule iffI*)
assume ‹(*a* # *t*) @ *e* # *u* = *t′* @ *e′* # *u′*›
then obtain *t″* **where** ‹*t′* = *a* # *t″*›
by (*metis Cons.prems*(*1*, *4*) *append-Cons append-Nil disjoint-iff*
list.exhaust-sel list.sel(*1*) *list.set-intros*(*1*))
with *Cons.hyps Cons.prems* ‹(*a* # *t*) @ *e* # *u* = *t′* @ *e′* # *u′*›
show ‹*a* # *t* = *t′* ∧ *e* = *e′* ∧ *u* = *u′*› **by** *auto*
next
show ‹*a* # *t* = *t′* ∧ *e* = *e′* ∧ *u* = *u′* ⟹ (*a* # *t*) @ *e* # *u* = *t′* @ *e′* # *u′*› **by**
simp
qed
qed

3.4.3 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* Θ *a* ∈ *A*. *Q a* ⊑ *P′* Θ *a* ∈ *A*. *Q′ a*›
if ‹*P* ⊑ *P′*› **and** ‹⋀*a*. *a* ∈ *A* ⟹ *a* ∈ α(*P*) ⟹ *Q a* ⊑ *Q′ a*›
proof (*unfold le-approx-def Refusals-after-def*, *safe*)
from *le-approx1*[*OF that*(*1*)] *le-approx-lemma-T*[*OF that*(*1*)]
le-approx1[*OF that*(*2*)[*rule-format*]]
show ‹*s* ∈ 𝒟 (*P′* Θ *a* ∈ *A*. *Q′ a*) ⟹ *s* ∈ 𝒟 (*P* Θ *a* ∈ *A*. *Q a*)› **for** *s*
by (*simp add*: *D-Throw subset-iff*)
(*metis events-of-memI in-set-conv-decomp*)
next
fix *s X*
assume *assms* : ‹*s* ∉ 𝒟 (*P* Θ *a* ∈ *A*. *Q a*)› ‹(*s*, *X*) ∈ ℱ (*P* Θ *a* ∈ *A*. *Q a*)›
from *assms*(*2*) **consider** ‹(*s*, *X*) ∈ ℱ *P*› ‹*set s* ∩ *ev* ‘ *A* = {}›

| *t1 t2* **where** ‹*s* = *t1* @ *t2*› ‹*t1* ∈ 𝒟 *P*› ‹*tF t1*› ‹*set t1* ∩ *ev* ‘ *A* = {}› ‹*ftF t2*›
| *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 (*simp add*: *F-Throw*) *blast*
thus ‹(*s*, *X*) ∈ ℱ (*P′* Θ *a* ∈ *A*. *Q′ a*)›
proof *cases*
assume ∗ : ‹(*s*, *X*) ∈ 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* ∉ 𝒟 *P*›
using ∗(*2*) **apply** (*cases* ‹*tF s*›, *auto simp add*: *disjoint-iff*)
by (*metis* ∗(*1*) *D-imp-front-tickFree F-T T-nonTickFree-imp-decomp butlast-snoc*
front-tickFree-append-iff in-set-butlastD is-processT9 list.distinct(*1*))
show ‹(*s*, *X*) ∈ ℱ *P* ⟹ *set s* ∩ *ev* ‘ *A* = {} ⟹ (*s*, *X*) ∈ ℱ (*Throw P′ A Q′*)›
by (*simp add*: *F-Throw le-approx2*[*OF that*(*1*) ∗∗])
next
from *assms*(*1*) **show** ‹⟦*s* = *t1* @ *t2*; *t1* ∈ 𝒟 *P*; *tF t1*; *set t1* ∩ *ev* ‘ *A* = {}; *ftF t2*⟧
⟹ (*s*, *X*) ∈ ℱ (*Throw P′ A Q′*)› **for** *t1 t2*
by (*simp add*: *F-Throw D-Throw*)
next
fix *t1 a t2* **assume** ∗ : ‹*s* = *t1* @ *ev a* # *t2*› ‹*t1* @ [*ev a*] ∈ 𝒯 *P*›
‹*set t1* ∩ *ev* ‘ *A* = {}› ‹*a* ∈ *A*› ‹(*t2*, *X*) ∈ ℱ (*Q a*)›
from ∗(*2*) **have** ∗∗ : ‹*tF t1*› **by** (*simp add*: *append-T-imp-tickFree*)
have ∗∗∗ : ‹(*t2*, *X*) ∈ ℱ (*Q′ a*)›
using *assms*(*1*)[*simplified D-Throw*, *simplified*, *THEN conjunct2*, *rule-format*, *OF* ∗(*4*, *3*, *2*, *1*)]
by (*metis* ∗(*2*, *4*, *5*) *events-of-memI last-in-set le-approx2 snoc-eq-iff-butlast that*(*2*))
have ∗∗∗∗ : ‹*t1* ∉ 𝒟 *P*›
apply (*rule ccontr*, *simp*,
drule assms(*1*)[*simplified D-Throw*, *simplified*, *THEN conjunct1*, *rule-format*,
OF ∗(*3*) ∗∗, *of* ‹*ev a* # *t2*›, *simplified* ∗(*1*), *simplified*])
by (*metis* ∗(*1*) *F-imp-front-tickFree assms*(*2*) *front-tickFree-append-iff list.discI*)
show ‹(*s*, *X*) ∈ ℱ (*Throw P′ A Q′*)›
by (*simp add*: *F-Throw D-Throw* ∗(*1*))
(*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*
by (*simp add*: *F-Throw D-Throw subset-eq*, *safe*, *simp-all*)
(*metis is-processT8*, (*metis D-T events-of-memI in-set-conv-decomp*)+)

next
  define *S-left*
    where ‹*S-left* ≡ {*t1* @ *t2* |*t1 t2*. *t1* ∈ 𝒟 *P* ∧ *tF t1* ∧
                  set t1 ∩ *ev* ‘ *A* = {} ∧ *ftF 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 less-list-def less-eq-list-def prefix-def*
    apply (*simp*, *safe*)
        apply (*solves* ‹*meson is-processT7*›)
      apply (*metis* (*no-types*, *lifting*) *append.right-neutral front-tickFree-Nil front-tickFree-append*
      front-tickFree-nonempty-append-imp inf-bot-right inf-sup-absorb inf-sup-aci(*2*)
set-append)
      apply (*metis Int-iff Un-iff append.right-neutral front-tickFree-Nil image-eqI*
set-append)
    apply (*metis D-T prefixI same-prefix-nil T-nonTickFree-imp-decomp append.right-neutral*
front-tickFree-Nil is-processT9 list.distinct(*1*))
    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*
      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*›
        apply (*simp add*: *S-left-def*)
        by (*metis D-imp-front-tickFree T-nonTickFree-imp-decomp append1-eq-conv*
assms(*1*)
          assms(*2*, *4*) $event_{ptick}$.*distinct*(*1*) *front-tickFree-Cons-iff tickFree-Cons-iff*
tickFree-append-iff)
      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*] ∈ $\mathcal{T}$ *P* − $\mathcal{D}$ *P* ∧
set t1 ∩ *ev* ' *A* = {} ∧ *a* ∈ *A* ∧ *t2* ∈ *min-elems* ($\mathcal{D}$ (*Q a*))} ∪
{*t1* @ *ev a* # *t2* |*t1 a t2*. *t1* @ [*ev a*] ∈ *min-elems* ($\mathcal{D}$ *P*) ∧
set t1 ∩ *ev* ' *A* = {} ∧ *a* ∈ *A* ∧ *t2* ∈ *min-elems* ($\mathcal{D}$ (*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** *fast*

fix *s*
assume *assm*: ‹*s* ∈ *min-elems* ($\mathcal{D}$ (*P* Θ *a* ∈ *A*. *Q a*))›
from *set-mp*[*OF* ∗, *OF this*]
consider ‹*s* ∈ *min-elems* ($\mathcal{D}$ *P*)› ‹*set s* ∩ *ev* ' *A* = {}›
| *t1 a t2* **where** ‹*s* = *t1* @ *ev a* # *t2*› ‹*set t1* ∩ *ev* ' *A* = {}› ‹*a* ∈ *A*› ‹*t2* ∈ *min-elems* ($\mathcal{D}$ (*Q a*))›
‹*t1* @ [*ev a*] ∈ *min-elems* ($\mathcal{D}$ *P*) ∨ *t1* @ [*ev a*] ∈ $\mathcal{T}$ *P* ∧ *t1* @ [*ev a*] ∉ $\mathcal{D}$ *P*›
using $****$ **by** (*simp add*: ∗∗) *blast*
thus ‹*s* ∈ $\mathcal{T}$ (*P'* Θ *a* ∈ *A*. *Q' a*)›
proof *cases*
show ‹*s* ∈ *min-elems* ($\mathcal{D}$ *P*) ⟹ *set s* ∩ *ev* ' *A* = {} ⟹ *s* ∈ $\mathcal{T}$ (*Throw P' A Q'*)›
by (*drule set-mp*[*OF le-approx3*[*OF that*(*1*)]], *simp add*: *T-Throw*)
next
fix *t1 a t2*
assume $*****$: ‹*s* = *t1* @ *ev a* # *t2*› ‹*set t1* ∩ *ev* ' *A* = {}› ‹*a* ∈ *A*› ‹*t2* ∈ *min-elems* ($\mathcal{D}$ (*Q a*))›
‹*t1* @ [*ev a*] ∈ *min-elems* ($\mathcal{D}$ *P*) ∨ *t1* @ [*ev a*] ∈ $\mathcal{T}$ *P* ∧ *t1* @ [*ev a*] ∉ $\mathcal{D}$ *P*›
have ‹*t1* @ [*ev a*] ∈ $\mathcal{T}$ *P'* ∧ *t2* ∈ $\mathcal{T}$ (*Q' a*)›
by (*meson* $*****$(*3*−*5*) *D-T events-of-memI in-set-conv-decomp le-approx2T le-approx-def subsetD that*)
with $*****$ **show** ‹*s* ∈ $\mathcal{T}$ (*Throw P' A Q'*)›
by (*simp add*: *T-Throw*) *blast*
qed
qed

lemma *mono-Throw-eq* :
‹(⋀*a*. *a* ∈ *A* ⟹ *a* ∈ α(*P*) ⟹ *Q a* = *Q' a*) ⟹
P Θ *a* ∈ *A*. *Q a* = *P* Θ *a* ∈ *A*. *Q' a*›
by (*subst Process-eq-spec*) (*auto simp add*: *Throw-projs events-of-memI*)

3.4.4 Properties

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

lemma *Throw-is-STOP-iff* : ‹*P* Θ *a* ∈ *A*. *Q a* = *STOP* ⟷ *P* = *STOP*›
proof (*rule iffI*)

show ‹*P* = *STOP*› **if** ‹*P* Θ *a* ∈ *A*. *Q a* = *STOP*›
proof (*rule ccontr*)
assume ‹*P* ≠ *STOP*›
then obtain *t* **where** ‹*t* ≠ []› ‹*t* ∈ 𝒯 *P*› **by** (*auto simp add*: *STOP-iff-T*)
hence ‹[*hd t*] ∈ 𝒯 *P*›
by (*metis append-Cons append-Nil is-processT3-TR-append list.sel*(*1*) *neq-Nil-conv*)
hence ‹[*hd t*] ∈ 𝒯 (*P* Θ *a* ∈ *A*. *Q a*)› **by** (*auto simp add*: *T-Throw Cons-eq-append-conv*)
with ‹*P* Θ *a* ∈ *A*. *Q a* = *STOP*› **show** *False* **by** (*simp add*: *STOP-iff-T*)
qed
next
show ‹*P* = *STOP* ⟹ *P* Θ *a* ∈ *A*. *Q a* = *STOP*› **by** *simp*
qed

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

lemma *Throw-BOT* [*simp*] : ‹⊥ Θ *a* ∈ *A*. *Q a* = ⊥›
by (*simp add*: *BOT-iff-Nil-D D-Throw D-BOT*)

lemma *Throw-is-BOT-iff*: ‹*P* Θ *a* ∈ *A*. *Q a* = ⊥ ⟷ *P* = ⊥›
by (*simp add*: *BOT-iff-Nil-D D-Throw*)

lemma *Throw-empty-set* [*simp*] : ‹*P* Θ *a* ∈ {}. *Q a* = *P*›
by (*auto simp add*: *Process-eq-spec F-Throw D-Throw intro*: *is-processT7 is-processT8*)
(*metis append.right-neutral front-tickFree-nonempty-append-imp*
nonTickFree-n-frontTickFree process-charn snoc-eq-iff-butlast)

lemma *Throw-is-restrictable-on-events-of* :
‹*P* Θ *a* ∈ *A*. *Q a* = *P* Θ *a* ∈ (*A* ∩ α(*P*)). *Q a*› (**is** ‹*?lhs* = *?rhs*›)
— A stronger version where α(*P*) is replaced by **α**(*P*) ∪ {*a*. ∃ *t*. *t* @ [*ev a*] ∈ *min-elems* (𝒟 *P*)} is probably true.
proof (*cases* ‹𝒟 *P* = {}›)
show ‹*?lhs* = *?rhs*› **if** ‹𝒟 *P* = {}›
proof (*rule Process-eq-optimizedI*)
fix *t* **assume** ‹*t* ∈ 𝒟 *?lhs*›
with ‹𝒟 *P* = {}› **obtain** *t1 a t2*
where ∗ : ‹*t* = *t1* @ *ev a* # *t2*› ‹*t1* @ [*ev a*] ∈ 𝒯 *P*›
‹*set t1* ∩ *ev* ‘ *A* = {}› ‹*a* ∈ *A*› ‹*t2* ∈ 𝒟 (*Q a*)›
unfolding *D-Throw* **by** *blast*
from ∗(*3*) **have** ‹*set t1* ∩ *ev* ‘ (*A* ∩ α(*P*)) = {}› **by** *blast*
moreover from ∗(*2*, *4*) **have** ‹*a* ∈ *A* ∩ α(*P*)› **by** (*simp add*: *events-of-memI*)
ultimately show ‹*t* ∈ 𝒟 *?rhs*› **using** ∗(*1*, *2*, *5*) **by** (*auto simp add*: *D-Throw*)
next
fix *t* **assume** ‹*t* ∈ 𝒟 *?rhs*›
with ‹𝒟 *P* = {}› **obtain** *t1 a t2*
where ∗ : ‹*t* = *t1* @ *ev a* # *t2*› ‹*t1* @ [*ev a*] ∈ 𝒯 *P*›

‹set t1 ∩ ev ‘ (A ∩ α(P)) = {}› ‹a ∈ A ∩ α(P)› ‹t2 ∈ 𝒟 (Q a)›
unfolding *D-Throw* **by** *blast*
from ∗(*2*, *3*) *events-of-memI* **have** ‹*set t1* ∩ *ev* ‘ *A* = {}› **by** *fastforce*
with ∗(*1*, *2*, *4*, *5*) **show** ‹*t* ∈ 𝒟 *?lhs*› **by** (*auto simp add*: *D-Throw*)
next
fix *t X* **assume** ‹(*t*, *X*) ∈ ℱ *?lhs*›
with ‹𝒟 *P* = {}› **consider** ‹(*t*, *X*) ∈ ℱ *P*› ‹*set t* ∩ *ev* ‘ *A* = {}›
| (*failR*) *t1 a t2* **where** ‹*t* = *t1* @ *ev a* # *t2*› ‹*t1* @ [*ev a*] ∈ 𝒯 *P*›
‹*set t1* ∩ *ev* ‘ *A* = {}› ‹*a* ∈ *A*› ‹(*t2*, *X*) ∈ ℱ (*Q a*)›
unfolding *F-Throw* **by** *blast*
thus ‹(*t*, *X*) ∈ ℱ *?rhs*›
proof *cases*
show ‹(*t*, *X*) ∈ ℱ *P* ⟹ *set t* ∩ *ev* ‘ *A* = {} ⟹ (*t*, *X*) ∈ ℱ *?rhs*›
by (*simp add*: *F-Throw disjoint-iff image-iff*)
next
case *failR*
from *failR*(*3*) **have** ‹*set t1* ∩ *ev* ‘ (*A* ∩ α(*P*)) = {}› **by** *blast*
moreover from *failR*(*2*, *4*) **have** ‹*a* ∈ *A* ∩ α(*P*)› **by** (*simp add*: *events-of-memI*)
ultimately show ‹(*t*, *X*) ∈ ℱ *?rhs*› **using** *failR*(*1*, *2*, *5*) **by** (*auto simp add*:
F-Throw)
qed
next
fix *t X* **assume** ‹(*t*, *X*) ∈ ℱ *?rhs*›
with ‹𝒟 *P* = {}› **consider** ‹(*t*, *X*) ∈ ℱ *P*› ‹*set t* ∩ *ev* ‘ (*A* ∩ α(*P*)) = {}›
| (*failR*) *t1 a t2* **where** ‹*t* = *t1* @ *ev a* # *t2*› ‹*t1* @ [*ev a*] ∈ 𝒯 *P*›
‹*set t1* ∩ *ev* ‘ (*A* ∩ α(*P*)) = {}› ‹*a* ∈ *A*›
‹*a* ∈ α(*P*)› ‹(*t2*, *X*) ∈ ℱ (*Q a*)›
unfolding *F-Throw* **by** *blast*
thus ‹(*t*, *X*) ∈ ℱ *?lhs*›
proof *cases*
assume ‹(*t*, *X*) ∈ ℱ *P*› ‹*set t* ∩ *ev* ‘ (*A* ∩ α(*P*)) = {}›
from ‹(*t*, *X*) ∈ ℱ *P*› **have** ‹*t* ∈ 𝒯 *P*› **by** (*simp add*: *F-T*)
with ‹*set t* ∩ *ev* ‘ (*A* ∩ α(*P*)) = {}› *events-of-memI*
have ‹*set t* ∩ *ev* ‘ *A* = {}› **by** *fast*
with ‹(*t*, *X*) ∈ ℱ *P*› **show** ‹(*t*, *X*) ∈ ℱ *?lhs*› **by** (*simp add*: *F-Throw*)
next
case *failR*
from *failR*(*2*, *3*) *events-of-memI* **have** ‹*set t1* ∩ *ev* ‘ *A* = {}› **by** *fastforce*
with *failR*(*1*, *2*, *4*−*6*) **show** ‹(*t*, *X*) ∈ ℱ *?lhs*› **by** (*auto simp add*: *F-Throw*)
qed
qed
next
assume ‹𝒟 *P* ≠ {}›
hence ‹α(*P*) = *UNIV*› **by** (*simp add*: *events-of-is-strict-events-of-or-UNIV*)
thus ‹*?lhs* = *?rhs*› **by** *simp*
qed

lemma *Throw-disjoint-events-of*: ‹*A* ∩ α(*P*) = {} ⟹ *P* Θ *a* ∈ *A*. *Q a* = *P*›

by (*metis Throw-empty-set Throw-is-restrictable-on-events-of*)

3.4.5 Continuity

context begin

private lemma *chain-Throw-left* : ‹*chain* $Y \Longrightarrow$ *chain* ($\lambda i.\ Y\ i\ \Theta\ a \in A.\ Q\ a$)›
by (*simp add*: *chain-def*)

private lemma *chain-Throw-right* : ‹*chain* $Y \Longrightarrow$ *chain* ($\lambda i.\ P\ \Theta\ a \in A.\ Y\ i\ a$)›
by (*simp add*: *chain-def fun-belowD*)

private lemma *cont-left-prem-Throw* :
‹$(\bigsqcup\ i.\ Y\ i)\ \Theta\ a \in A.\ Q\ a = (\bigsqcup\ i.\ Y\ i\ \Theta\ a \in A.\ Q\ a)$›
(**is** ‹*?lhs* = *?rhs*›) **if** ‹*chain* Y›
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*: *limproc-is-thelub* ‹*chain* Y› *chain-Throw-left D-Throw T-LUB D-LUB*)
next
fix s
define S
where ‹$S\ i \equiv \{t1.\ \exists\, t2.\ s = t1\ @\ t2 \wedge t1 \in \mathcal{D}\ (Y\ i) \wedge$ *tickFree* $t1\ \wedge$
set $t1 \cap ev$ ‘ $A = \{\} \wedge$ *front-tickFree* $t2\} \cup$
$\{t1.\ \exists\, a\ t2.\ s = t1\ @\ ev\ a\ \#\ t2 \wedge t1\ @\ [ev\ a] \in \mathcal{T}\ (Y\ i) \wedge$ *tickFree* $t1\ \wedge$
set $t1 \cap ev$ ‘ $A = \{\} \wedge a \in A \wedge t2 \in \mathcal{D}\ (Q\ a)\}$› **for** i
assume ‹$s \in \mathcal{D}$ *?rhs*›
hence *ftF*: ‹*front-tickFree s*› **using** *D-imp-front-tickFree* **by** *blast*
from ‹$s \in \mathcal{D}$ *?rhs*› **have** ‹$s \in \mathcal{D}\ (Y\ i\ \Theta\ a \in A.\ Q\ a)$› **for** i
by (*simp add*: *limproc-is-thelub D-LUB chain-Throw-left* ‹*chain* Y›)
hence ‹$S\ i \neq \{\}$› **for** i **by** (*simp add*: *S-def D-Throw*)
(*metis append-T-imp-tickFree not-Cons-self2*)
moreover have ‹*finite* (S *0*)›
unfolding *S-def* **by** (*prove-finite-subset-of-prefixes s*)
moreover have ‹$S\ (Suc\ i) \subseteq S\ i$› **for** i
unfolding *S-def* **apply** (*intro allI Un-mono subsetI*; *simp*)
by (*metis in-mono le-approx1 po-class.chainE* ‹*chain* Y›)
(*metis le-approx-lemma-T po-class.chain-def subset-eq* ‹*chain* Y›)
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* ‹$\exists\, j\ a\ t2.\ s = t1\ @\ ev\ a\ \#\ t2 \wedge t1\ @\ [ev\ a] \in \mathcal{T}\ (Y\ j) \wedge a \in A \wedge t2 \in \mathcal{D}\ (Q\ a)$›)
case *True*
then obtain $j\ a\ t2$ **where** $**$: ‹$s = t1\ @\ ev\ a\ \#\ t2$› ‹$t1\ @\ [ev\ a] \in \mathcal{T}\ (Y\ j)$›

```
    ‹a ∈ A› ‹t2 ∈ 𝒟 (Q a)› by blast
  from * **(1) have ‹∀ i. t1 @ [ev a] ∈ 𝒯 (Y i)›
    by (simp add: S-def) (meson D-T front-tickFree-single is-processT7)
  with * **(1, 3, 4) show ‹s ∈ 𝒟 ?lhs›
    by (simp add: S-def D-Throw limproc-is-thelub ‹chain Y› T-LUB) blast
 next
  case False
  with * have ‹∀ i. ∃ t2. s = t1 @ t2 ∧ t1 ∈ 𝒟 (Y i) ∧ front-tickFree t2›
    by (simp add: S-def) blast
  hence ‹∃ t2. s = t1 @ t2 ∧ (∀ i. t1 ∈ 𝒟 (Y i)) ∧ front-tickFree t2› by blast
  with * show ‹s ∈ 𝒟 ?lhs›
    by (simp add: S-def D-Throw limproc-is-thelub ‹chain Y› D-LUB) blast
 qed
next
 show ‹(s, X) ∈ ℱ ?lhs ⟹ (s, X) ∈ ℱ ?rhs› for s X
   by (auto simp add: limproc-is-thelub ‹chain Y› chain-Throw-left F-Throw
F-LUB T-LUB D-LUB)
next
 assume same-div : ‹𝒟 ?lhs = 𝒟 ?rhs›
 fix s X assume ‹(s, X) ∈ ℱ ?rhs›
 show ‹(s, X) ∈ ℱ ?lhs›
 proof (cases ‹s ∈ 𝒟 ?rhs›)
  show ‹s ∈ 𝒟 ?rhs ⟹ (s, X) ∈ ℱ ?lhs› by (simp add: is-processT8 same-div)
 next
  assume ‹s ∉ 𝒟 ?rhs›
  have ‹∀ a∈A. Q a ⊑ Q a› by simp
  moreover from ‹s ∉ 𝒟 ?rhs› obtain j where ‹s ∉ 𝒟 (Y j Θ a ∈ A. Q a)›
    by (auto simp add: limproc-is-thelub chain-Throw-left ‹chain Y› D-LUB)
  moreover from ‹(s, X) ∈ ℱ ?rhs› have ‹(s, X) ∈ ℱ (Y j Θ a ∈ A. Q a)›
    by (simp add: limproc-is-thelub chain-Throw-left ‹chain Y› F-LUB)
  ultimately show ‹(s, X) ∈ ℱ ?lhs›
    by (meson is-ub-thelub le-approx2 mono-Throw ‹chain Y›)
 qed
qed


private lemma cont-right-prem-Throw :
 ‹P Θ a ∈ A. (⨆ i. Y i a) = (⨆ i. P Θ a ∈ A. Y i a)›
 (is ‹?lhs = ?rhs›) if ‹chain Y›
proof (subst Process-eq-spec-optimized, safe)
 show ‹s ∈ 𝒟 ?lhs ⟹ s ∈ 𝒟 ?rhs› for s
   by (simp add: limproc-is-thelub ‹chain Y› chain-Throw-right ch2ch-fun[OF
‹chain Y›] 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 ∧ tF t1 ∧
```

```
                        set t1 ∩ ev ` A = {} ∧ ftF 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 Θ a ∈ A. Y i a)› for i
    by (simp add: limproc-is-thelub D-LUB chain-Throw-right ‹chain Y›)
  hence ‹S i ≠ {}› for i by (simp add: S-def D-Throw) metis
  moreover have ‹finite (S 0)›
    unfolding S-def by (prove-finite-subset-of-prefixes s)
  moreover have ‹S (Suc i) ⊆ S i› for i
    unfolding S-def apply (intro allI Un-mono subsetI; simp)
    by (metis fun-belowD le-approx1 po-class.chainE subset-iff ‹chain Y›)
  ultimately have ‹(⋂ i. S i) ≠ {}›
    by (rule Inter-nonempty-finite-chained-sets)
  then obtain t1 where ‹∀ i. t1 ∈ S i›
    by (meson INT-iff ex-in-conv iso-tuple-UNIV-I)
  then consider ‹t1 ∈ 𝒟 P› ‹tF t1›
    ‹set t1 ∩ ev ` A = {}› ‹∃ t2. s = t1 @ t2 ∧ ftF 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 Y› 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 Y› chain-Throw-right ch2ch-fun[OF
‹chain Y›] F-Throw F-LUB T-LUB D-LUB) blast
next
  assume same-div : ‹𝒟 ?lhs = 𝒟 ?rhs›
  fix s X assume ‹(s, X) ∈ ℱ ?rhs›
  show ‹(s, X) ∈ ℱ ?lhs›
  proof (cases ‹s ∈ 𝒟 ?rhs›)
    show ‹s ∈ 𝒟 ?rhs ⟹ (s, X) ∈ ℱ ?lhs› by (simp add: is-processT8 same-div)
  next
    assume ‹s ∉ 𝒟 ?rhs›
```

have ‹$\forall a{\in}A.\ Y\ i\ a \sqsubseteq (\bigsqcup i.\ Y\ i\ a)$› **for** i **by** (*metis ch2ch-fun is-ub-thelub* ‹*chain* Y›)
moreover from ‹$s \notin \mathcal{D}$ *?rhs*› **obtain** j **where** ‹$s \notin \mathcal{D}\ (P\ \Theta\ a \in A.\ Y\ j\ a)$›
by (*auto simp add*: *limproc-is-thelub chain-Throw-right* ‹*chain* Y› *D-LUB*)
moreover from ‹$(s, X) \in \mathcal{F}$ *?rhs*› **have** ‹$(s, X) \in \mathcal{F}\ (P\ \Theta\ a \in A.\ Y\ j\ a)$›
by (*simp add*: *limproc-is-thelub chain-Throw-right* ‹*chain* Y› *F-LUB*)
find-theorems ‹*chain* ($\lambda a.$ *?P*)›
ultimately show ‹$(s, X) \in \mathcal{F}$ *?lhs*›
by (*metis* (*mono-tags*, *lifting*) *below-refl le-approx2 mono-Throw*)
qed
qed

lemma *Throw-cont*[*simp*] :
assumes *cont-f* : ‹*cont f*› **and** *cont-g* : ‹$\forall a.$ *cont* ($g\ a$)›
shows ‹*cont* ($\lambda x.\ f\ x\ \Theta\ a \in A.\ g\ a\ x$)›
proof −
have $*$: ‹*cont* ($\lambda y.\ y\ \Theta\ a \in A.\ g\ a\ x$)› **for** x
by (*rule contI2*, *rule monofunI*, *solves simp*, *simp add*: *cont-left-prem-Throw*)
have ‹$\bigwedge y.$ *cont* (*Throw* $y\ A$)›
by (*simp add*: *contI2 cont-right-prem-Throw fun-belowD lub-fun monofunI*)
hence $**$: ‹*cont* ($\lambda x.\ y\ \Theta\ a \in A.\ g\ a\ x$)› **for** y
by (*rule cont-compose*) (*simp add*: *cont-g*)
show *?thesis* **by** (*fact cont-apply*[*OF cont-f* $*$ $**$])
qed

end

3.5 The Interrupt Operator

3.5.1 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 [3, 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* :: ‹$[('a, 'r)\ process_{ptick}, ('a, 'r)\ process_{ptick}] \Rightarrow ('a, 'r)\ process_{ptick}$› (**infixl** ‹△› *81*)
is ‹$\lambda P\ Q.$
$(\{(t\ @\ [✓(r)], X)\ |t\ r\ X.\ t\ @\ [✓(r)] \in \mathcal{T}\ P\} \cup$
$\{(t, X - \{✓(r)\})\ |t\ r\ X.\ t\ @\ [✓(r)] \in \mathcal{T}\ P\} \cup$
$\{(t, X).\ (t, X) \in \mathcal{F}\ P \wedge tF\ t \wedge ([], X) \in \mathcal{F}\ Q\} \cup$

{(t @ u, X) |t u X. t ∈ 𝒯 P ∧ tF t ∧ (u, X) ∈ ℱ Q ∧ u ≠ []} ∪
{(t, X − {✓(r)}) |t r X. t ∈ 𝒯 P ∧ tF t ∧ [✓(r)] ∈ 𝒯 Q} ∪
{(t, X). t ∈ 𝒟 P} ∪
{(t @ u, X) |t u X. t ∈ 𝒯 P ∧ tF t ∧ u ∈ 𝒟 Q},
𝒟 P ∪ {t @ u |t u. t ∈ 𝒯 P ∧ tF t ∧ u ∈ 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 ‹(t, X) ∈ *?f1* ∪ *?f2* ∪ *?f3* ∪ *?f4* ∪ *?f5* ∪ *?f6* ∪ *?f7* ⟹ *ftF t*› **for** *t X*
by (*simp add*: *is-processT2 D-imp-front-tickFree front-tickFree-append*)
(*meson front-tickFree-append front-tickFree-dw-closed is-processT2-TR process-charn*)
next
fix *t u*
show ‹(t @ u, {}) ∈ *?f1* ∪ *?f2* ∪ *?f3* ∪ *?f4* ∪ *?f5* ∪ *?f6* ∪ *?f7* ⟹
(t, {}) ∈ *?f1* ∪ *?f2* ∪ *?f3* ∪ *?f4* ∪ *?f5* ∪ *?f6* ∪ *?f7*›
proof (*induct u rule*: *rev-induct*)
show ‹(t @ [], {}) ∈ *?f1* ∪ *?f2* ∪ *?f3* ∪ *?f4* ∪ *?f5* ∪ *?f6* ∪ *?f7* ⟹
(t, {}) ∈ *?f1* ∪ *?f2* ∪ *?f3* ∪ *?f4* ∪ *?f5* ∪ *?f6* ∪ *?f7*› **by** *simp*
next
fix *a u*
assume *assm* : ‹(t @ u @ [a], {}) ∈ *?f1* ∪ *?f2* ∪ *?f3* ∪ *?f4* ∪ *?f5* ∪ *?f6* ∪ *?f7*›
and *hyp* : ‹(t @ u, {}) ∈ *?f1* ∪ *?f2* ∪ *?f3* ∪ *?f4* ∪ *?f5* ∪ *?f6* ∪ *?f7* ⟹
(t, {}) ∈ *?f1* ∪ *?f2* ∪ *?f3* ∪ *?f4* ∪ *?f5* ∪ *?f6* ∪ *?f7*›
from *assm* **have** ‹(t @ u @ [a], {}) ∈ *?f1* ∨ (t @ u @ [a], {}) ∈ *?f2* ∨
(t @ u @ [a], {}) ∈ *?f3* ∨ (t @ u @ [a], {}) ∈ *?f4* ∨ (t @ u @ [a], {}) ∈ *?f5* ∨
(t @ u @ [a], {}) ∈ *?f6* ∨ (t @ u @ [a], {}) ∈ *?f7*› **by** *fast*
thus ‹(t, {}) ∈ *?f1* ∪ *?f2* ∪ *?f3* ∪ *?f4* ∪ *?f5* ∪ *?f6* ∪ *?f7*›
proof (*elim disjE*)
assume ‹(t @ u @ [a], {}) ∈ *?f1*›
hence ‹(t, {}) ∈ *?f3*›
by *simp* (*meson T-F append-T-imp-tickFree is-processT snoc-eq-iff-butlast*)

thus ‹(t, {}) ∈ *?f1* ∪ *?f2* ∪ *?f3* ∪ *?f4* ∪ *?f5* ∪ *?f6* ∪ *?f7*› **by** *blast*
next
assume ‹(t @ u @ [a], {}) ∈ *?f2*›
hence ‹(t, {}) ∈ *?f3*›
by *simp* (*metis T-F Nil-is-append-conv append-T-imp-tickFree is-processT list.discI*)
thus ‹(t, {}) ∈ *?f1* ∪ *?f2* ∪ *?f3* ∪ *?f4* ∪ *?f5* ∪ *?f6* ∪ *?f7*› **by** *blast*
next

assume ‹$(t @ u @ [a], \{\}) \in$ *?f3*›
with *is-processT3* **have** ‹$(t, \{\}) \in$ *?f3*› **by** *simp blast*
thus ‹$(t, \{\}) \in$ *?f1* ∪ *?f2* ∪ *?f3* ∪ *?f4* ∪ *?f5* ∪ *?f6* ∪ *?f7*› **by** *blast*
next
assume ‹$(t @ u @ [a], \{\}) \in$ *?f4*›
then obtain t' u'
where ∗ : ‹$t @ u = t' @ u'$› ‹$t' \in \mathcal{T}\ P$› ‹$tF\ t'$› ‹$(u' @ [a], \{\}) \in \mathcal{F}\ Q$›
by *simp* (*metis butlast-append last-appendR snoc-eq-iff-butlast*)
show ‹$(t, \{\}) \in$ *?f1* ∪ *?f2* ∪ *?f3* ∪ *?f4* ∪ *?f5* ∪ *?f6* ∪ *?f7*›
proof (*cases* ‹$u' = []$›)
assume ‹$u' = []$›
with ∗(*1*, *2*, *3*) **have** ‹$(t, \{\}) \in$ *?f3*›
by *simp* (*metis T-F process-charn tickFree-append-iff*)
thus ‹$(t, \{\}) \in$ *?f1* ∪ *?f2* ∪ *?f3* ∪ *?f4* ∪ *?f5* ∪ *?f6* ∪ *?f7*› **by** *blast*
next
assume ‹$u' \neq []$›
with ∗ *is-processT3* **have** ‹$(t @ u, \{\}) \in$ *?f4*› **by** *simp blast*
thus ‹$(t, \{\}) \in$ *?f1* ∪ *?f2* ∪ *?f3* ∪ *?f4* ∪ *?f5* ∪ *?f6* ∪ *?f7*› **by** (*intro hyp*)
blast
qed
next
assume ‹$(t @ u @ [a], \{\}) \in$ *?f5*›
hence ‹$(t, \{\}) \in$ *?f3*› **by** *simp* (*metis T-F process-charn*)
thus ‹$(t, \{\}) \in$ *?f1* ∪ *?f2* ∪ *?f3* ∪ *?f4* ∪ *?f5* ∪ *?f6* ∪ *?f7*› **by** *blast*
next
assume ‹$(t @ u @ [a], \{\}) \in$ *?f6*›
hence ‹$(t, \{\}) \in$ *?f3*›
by *simp* (*meson D-T append-T-imp-tickFree process-charn snoc-eq-iff-butlast*)
thus ‹$(t, \{\}) \in$ *?f1* ∪ *?f2* ∪ *?f3* ∪ *?f4* ∪ *?f5* ∪ *?f6* ∪ *?f7*› **by** *blast*
next
assume ‹$(t @ u @ [a], \{\}) \in$ *?f7*›
then obtain t' u'
where ∗ : ‹$t @ u @ [a] = t' @ u'$› ‹$t' \in \mathcal{T}\ P$› ‹$tF\ t'$› ‹$u' \in \mathcal{D}\ Q$› **by** *blast*
hence ‹$(t @ u, \{\}) \in$ (*if length* $u' \leq 1$ *then ?f3 else ?f4*)›
apply (*cases* u' *rule*: *rev-cases*; *simp*)
by (*metis T-F append-assoc process-charn tickFree-append-iff*)
(*metis D-T T-F is-processT3*)
thus ‹$(t, \{\}) \in$ *?f1* ∪ *?f2* ∪ *?f3* ∪ *?f4* ∪ *?f5* ∪ *?f6* ∪ *?f7*›
by (*intro hyp*) (*meson UnI1 UnI2*)
qed
qed
next
fix *t X Y*
assume *assm* : ‹$(t, Y) \in$ *?f1* ∪ *?f2* ∪ *?f3* ∪ *?f4* ∪ *?f5* ∪ *?f6* ∪ *?f7* $\wedge X \subseteq Y$›
hence ‹$(t, Y) \in$ *?f1* $\vee (t, Y) \in$ *?f2* $\vee (t, Y) \in$ *?f3* $\vee (t, Y) \in$ *?f4* $\vee$
$(t, Y) \in$ *?f5* $\vee (t, Y) \in$ *?f6* $\vee (t, Y) \in$ *?f7*› **by** *fast*
thus ‹$(t, X) \in$ *?f1* ∪ *?f2* ∪ *?f3* ∪ *?f4* ∪ *?f5* ∪ *?f6* ∪ *?f7*›
proof (*elim disjE*)
assume ‹$(t, Y) \in$ *?f1*›

hence ‹$(t, X) \in$ *?f1*› **by** *simp*
thus ‹$(t, X) \in$ *?f1* $\cup$ *?f2* $\cup$ *?f3* $\cup$ *?f4* $\cup$ *?f5* $\cup$ *?f6* $\cup$ *?f7*› **by** *blast*
next
assume ‹$(t, Y) \in$ *?f2*›
with *assm*[*THEN conjunct2*] **have** ‹$(t, X) \in$ *?f2*› **by** *simp blast*
thus ‹$(t, X) \in$ *?f1* $\cup$ *?f2* $\cup$ *?f3* $\cup$ *?f4* $\cup$ *?f5* $\cup$ *?f6* $\cup$ *?f7*› **by** *blast*
next
assume ‹$(t, Y) \in$ *?f3*›
with *assm*[*THEN conjunct2*] *is-processT4* **have** ‹$(t, X) \in$ *?f3*› **by** *simp blast*
thus ‹$(t, X) \in$ *?f1* $\cup$ *?f2* $\cup$ *?f3* $\cup$ *?f4* $\cup$ *?f5* $\cup$ *?f6* $\cup$ *?f7*› **by** *blast*
next
assume ‹$(t, Y) \in$ *?f4*›
with *assm*[*THEN conjunct2*] *is-processT4* **have** ‹$(t, X) \in$ *?f4*› **by** *simp blast*
thus ‹$(t, X) \in$ *?f1* $\cup$ *?f2* $\cup$ *?f3* $\cup$ *?f4* $\cup$ *?f5* $\cup$ *?f6* $\cup$ *?f7*› **by** *blast*
next
assume ‹$(t, Y) \in$ *?f5*›
with *assm*[*THEN conjunct2*] **have** ‹$(t, X) \in$ *?f5*› **by** *simp blast*
thus ‹$(t, X) \in$ *?f1* $\cup$ *?f2* $\cup$ *?f3* $\cup$ *?f4* $\cup$ *?f5* $\cup$ *?f6* $\cup$ *?f7*› **by** *blast*
next
assume ‹$(t, Y) \in$ *?f6*›
hence ‹$(t, X) \in$ *?f6*› **by** *simp*
thus ‹$(t, X) \in$ *?f1* $\cup$ *?f2* $\cup$ *?f3* $\cup$ *?f4* $\cup$ *?f5* $\cup$ *?f6* $\cup$ *?f7*› **by** *blast*
next
assume ‹$(t, Y) \in$ *?f7*›
hence ‹$(t, X) \in$ *?f7*› **by** *simp*
thus ‹$(t, X) \in$ *?f1* $\cup$ *?f2* $\cup$ *?f3* $\cup$ *?f4* $\cup$ *?f5* $\cup$ *?f6* $\cup$ *?f7*› **by** *blast*
qed
next
fix *t X Y*
assume *assm* : ‹$(t, X) \in$ *?f1* $\cup$ *?f2* $\cup$ *?f3* $\cup$ *?f4* $\cup$ *?f5* $\cup$ *?f6* $\cup$ *?f7* $\wedge$
($\forall c.\ c \in Y \longrightarrow (t\ @\ [c], \{\}) \notin$ *?f1* $\cup$ *?f2* $\cup$ *?f3* $\cup$ *?f4* $\cup$ *?f5* $\cup$ *?f6*
$\cup$ *?f7*)›
have ‹$(t, X) \in$ *?f1* $\vee$ $(t, X) \in$ *?f2* $\vee$ $(t, X) \in$ *?f3* $\vee$ $(t, X) \in$ *?f4* $\vee$
$(t, X) \in$ *?f5* $\vee$ $(t, X) \in$ *?f6* $\vee$ $(t, X) \in$ *?f7*› **using** *assm*[*THEN conjunct1*]
by *fast*
thus ‹$(t, X \cup Y) \in$ *?f1* $\cup$ *?f2* $\cup$ *?f3* $\cup$ *?f4* $\cup$ *?f5* $\cup$ *?f6* $\cup$ *?f7*›
proof (*elim disjE*)
assume ‹$(t, X) \in$ *?f1*›
hence ‹$(t, X \cup Y) \in$ *?f1*› **by** *simp*
thus ‹$(t, X \cup Y) \in$ *?f1* $\cup$ *?f2* $\cup$ *?f3* $\cup$ *?f4* $\cup$ *?f5* $\cup$ *?f6* $\cup$ *?f7*› **by** *blast*
next
assume ‹$(t, X) \in$ *?f2*›
with *assm*[*THEN conjunct2*] **have** ‹$(t, X \cup Y) \in$ *?f2*›
by *simp* (*metis Diff-insert-absorb Un-Diff*)
thus ‹$(t, X \cup Y) \in$ *?f1* $\cup$ *?f2* $\cup$ *?f3* $\cup$ *?f4* $\cup$ *?f5* $\cup$ *?f6* $\cup$ *?f7*› **by** *blast*
next
assume ‹$(t, X) \in$ *?f3*›
with *assm*[*THEN conjunct2*] **have** ‹$(t, X \cup Y) \in$ *?f3*›
by *simp* (*metis F-T T-F T-nonTickFree-imp-decomp append1-eq-conv ap-*

pend-Nil event$_{ptick}$*.distinct-disc*(*2*)
is-processT5-S7′ list.distinct(*1*) *tickFree-Cons-iff tickFree-append-iff*)
thus ‹(*t*, *X* ∪ *Y*) ∈ *?f1* ∪ *?f2* ∪ *?f3* ∪ *?f4* ∪ *?f5* ∪ *?f6* ∪ *?f7*› **by** *blast*
next
assume ‹(*t*, *X*) ∈ *?f4*›
with *assm*[*THEN conjunct2*] **have** ‹(*t*, *X* ∪ *Y*) ∈ *?f4*›
by *simp* (*metis append.assoc append-is-Nil-conv is-processT5*)
thus ‹(*t*, *X* ∪ *Y*) ∈ *?f1* ∪ *?f2* ∪ *?f3* ∪ *?f4* ∪ *?f5* ∪ *?f6* ∪ *?f7*› **by** *blast*
next
assume ‹(*t*, *X*) ∈ *?f5*›
with *assm*[*THEN conjunct2*] **have** ‹(*t*, *X* ∪ *Y*) ∈ *?f5*›
by *simp* (*metis Diff-empty Diff-insert0 T-F Un-Diff not-Cons-self*)
thus ‹(*t*, *X* ∪ *Y*) ∈ *?f1* ∪ *?f2* ∪ *?f3* ∪ *?f4* ∪ *?f5* ∪ *?f6* ∪ *?f7*› **by** *blast*
next
assume ‹(*t*, *X*) ∈ *?f6*›
hence ‹(*t*, *X* ∪ *Y*) ∈ *?f6*› **by** *simp*
thus ‹(*t*, *X* ∪ *Y*) ∈ *?f1* ∪ *?f2* ∪ *?f3* ∪ *?f4* ∪ *?f5* ∪ *?f6* ∪ *?f7*› **by** *blast*
next
assume ‹(*t*, *X*) ∈ *?f7*›
hence ‹(*t*, *X* ∪ *Y*) ∈ *?f7*› **by** *simp*
thus ‹(*t*, *X* ∪ *Y*) ∈ *?f1* ∪ *?f2* ∪ *?f3* ∪ *?f4* ∪ *?f5* ∪ *?f6* ∪ *?f7*› **by** *blast*
qed
next
fix *t r X*
have ∗ : ‹(*t* @ [✔(*r*)], {}) ∉ *?f2* ∪ *?f3* ∪ *?f5*›
by *simp* (*metis T-imp-front-tickFree front-tickFree-Cons-iff front-tickFree-append-iff non-tickFree-tick tickFree-Cons-iff tickFree-Nil*)
assume ‹(*t* @ [✔(*r*)], {}) ∈ *?f1* ∪ *?f2* ∪ *?f3* ∪ *?f4* ∪ *?f5* ∪ *?f6* ∪ *?f7*›
with ∗ **have** ‹(*t* @ [✔(*r*)], {}) ∈ *?f1* ∨ (*t* @ [✔(*r*)], {}) ∈ *?f4* ∨
(*t* @ [✔(*r*)], {}) ∈ *?f6* ∨ (*t* @ [✔(*r*)], {}) ∈ *?f7*› **by** *fast*
thus ‹(*t*, *X* − {✔(*r*)}) ∈ *?f1* ∪ *?f2* ∪ *?f3* ∪ *?f4* ∪ *?f5* ∪ *?f6* ∪ *?f7*›
proof (*elim disjE*)
assume ‹(*t* @ [✔(*r*)], {}) ∈ *?f1*›
hence ‹(*t*, *X* − {✔(*r*)}) ∈ *?f2*› **by** *blast*
thus ‹(*t*, *X* − {✔(*r*)}) ∈ *?f1* ∪ *?f2* ∪ *?f3* ∪ *?f4* ∪ *?f5* ∪ *?f6* ∪ *?f7*› **by** *blast*
next
assume ‹(*t* @ [✔(*r*)], {}) ∈ *?f4*›
then obtain *t′ u′*
where ∗∗ : ‹*t* = *t′* @ *u′*› ‹*t′* ∈ 𝒯 *P*› ‹*tF t′*› ‹(*u′* @ [✔(*r*)], {}) ∈ ℱ *Q*›
by *simp* (*metis butlast-append last-appendR snoc-eq-iff-butlast*)
hence ‹(*t*, *X* − {✔(*r*)}) ∈ (*if u′* = [] *then ?f5 else ?f4*)›
by *simp* (*metis F-T process-charn self-append-conv2*)
thus ‹(*t*, *X* − {✔(*r*)}) ∈ *?f1* ∪ *?f2* ∪ *?f3* ∪ *?f4* ∪ *?f5* ∪ *?f6* ∪ *?f7*› **by** (*meson UnCI*)
next
assume ‹(*t* @ [✔(*r*)], {}) ∈ *?f6*›
with *is-processT9* **have** ‹*t* ∈ 𝒟 *P*› **by** *fast*
thus ‹(*t*, *X* − {✔(*r*)}) ∈ *?f1* ∪ *?f2* ∪ *?f3* ∪ *?f4* ∪ *?f5* ∪ *?f6* ∪ *?f7*› **by** *blast*
next

assume ‹$(t \mathbin{@} [✓(r)], \{\}) \in$ *?f7*›
then obtain t' u'
where $**$: ‹$t \mathbin{@} [✓(r)] = t' \mathbin{@} u'$› ‹$t' \in \mathcal{T}\ P$› ‹*tF* t'› ‹$u' \in \mathcal{D}\ Q$› **by** *blast*
from $**(1, 3, 4)$ **obtain** u'' **where** ‹$u' = u'' \mathbin{@} [✓(r)]$› ‹$u'' \mathbin{@} [✓(r)] \in \mathcal{D}\ Q$›
by (*cases* u' *rule*: *rev-cases*) *auto*
with $**(1)$ *is-processT9* **have** ‹$t = t' \mathbin{@} u'' \wedge u'' \in \mathcal{D}\ Q$› **by** *force*
with $**(2, 3)$ **have** ‹$(t, X - \{✓(r)\}) \in$ *?f7*› **by** *simp blast*
thus ‹$(t, X - \{✓(r)\}) \in$ *?f1* $\cup$ *?f2* $\cup$ *?f3* $\cup$ *?f4* $\cup$ *?f5* $\cup$ *?f6* $\cup$ *?f7*› **by** *blast*
qed
next
show ‹$t \in$ *?d1* $\cup$ *?d2* $\wedge$ *tF* t $\wedge$ *ftF* $u \Longrightarrow t \mathbin{@} u \in$ *?d1* $\cup$ *?d2*› **for** t u
apply (*simp*, *elim conjE disjE exE*)
by (*solves* ‹*simp add*: *is-processT7*›)
(*meson append.assoc is-processT7 tickFree-append-iff*)
next
show ‹$t \in$ *?d1* $\cup$ *?d2* $\Longrightarrow (t, X) \in$ *?f1* $\cup$ *?f2* $\cup$ *?f3* $\cup$ *?f4* $\cup$ *?f5* $\cup$ *?f6* $\cup$ *?f7*›
for t X
by *blast*
next
fix t r
assume ‹$t \mathbin{@} [✓(r)] \in$ *?d1* $\cup$ *?d2*›
then consider ‹$t \mathbin{@} [✓(r)] \in$ *?d1*› | ‹$t \mathbin{@} [✓(r)] \in$ *?d2*› **by** *blast*
thus ‹$t \in$ *?d1* $\cup$ *?d2*›
proof *cases*
assume ‹$t \mathbin{@} [✓(r)] \in$ *?d1*›
hence ‹$t \in$ *?d1*› **by** (*fact is-processT9*)
thus ‹$t \in$ *?d1* $\cup$ *?d2*› **by** *blast*
next
assume ‹$t \mathbin{@} [✓(r)] \in$ *?d2*›
then obtain t' u'
where $**$: ‹$t \mathbin{@} [✓(r)] = t' \mathbin{@} u'$› ‹$t' \in \mathcal{T}\ P$› ‹*tF* t'› ‹$u' \in \mathcal{D}\ Q$› **by** *blast*
from $**(1, 3, 4)$ **obtain** u'' **where** ‹$u' = u'' \mathbin{@} [✓(r)]$› ‹$u'' \mathbin{@} [✓(r)] \in \mathcal{D}\ Q$›
by (*cases* u' *rule*: *rev-cases*) *auto*
with $**(1)$ *is-processT9* **have** ‹$t = t' \mathbin{@} u'' \wedge u'' \in \mathcal{D}\ Q$› **by** *force*
with $**(2, 3)$ **have** ‹$t \in$ *?d2*› **by** *simp blast*
thus ‹$t \in$ *?d1* $\cup$ *?d2*› **by** *blast*
qed
qed
qed

3.5.2 Projections

lemma *F-Interrupt* :
‹$\mathcal{F}\ (P \triangle Q) =$
$\{(t \mathbin{@} [✓(r)], X) \mid t\ r\ X.\ t \mathbin{@} [✓(r)] \in \mathcal{T}\ P\} \cup$
$\{(t, X - \{✓(r)\}) \mid t\ r\ X.\ t \mathbin{@} [✓(r)] \in \mathcal{T}\ P\} \cup$
$\{(t, X).\ (t, X) \in \mathcal{F}\ P \wedge tF\ t \wedge ([], X) \in \mathcal{F}\ Q\} \cup$
$\{(t \mathbin{@} u, X) \mid t\ u\ X.\ t \in \mathcal{T}\ P \wedge tF\ t \wedge (u, X) \in \mathcal{F}\ Q \wedge u \neq []\} \cup$
$\{(t, X - \{✓(r)\}) \mid t\ r\ X.\ t \in \mathcal{T}\ P \wedge tF\ t \wedge [✓(r)] \in \mathcal{T}\ Q\} \cup$

$\{(t, X).\ t \in \mathcal{D}\ P\} \cup$
$\{(t\ @\ u, X)\ |t\ u\ X.\ t \in \mathcal{T}\ P \wedge tF\ t \wedge u \in \mathcal{D}\ Q\}$›
by (*simp add*: *Failures.rep-eq FAILURES-def Interrupt.rep-eq*)

lemma *D-Interrupt* :
‹$\mathcal{D}\ (P \bigtriangleup Q) = \mathcal{D}\ P \cup \{t\ @\ u\ |t\ u.\ t \in \mathcal{T}\ P \wedge tF\ t \wedge u \in \mathcal{D}\ Q\}$›
by (*simp add*: *Divergences.rep-eq DIVERGENCES-def Interrupt.rep-eq*)

lemma *T-Interrupt* :
‹$\mathcal{T}\ (P \bigtriangleup Q) = \mathcal{T}\ P \cup \{t\ @\ u\ |t\ u.\ t \in \mathcal{T}\ P \wedge tF\ t \wedge u \in \mathcal{T}\ Q\}$›
apply (*simp add*: *Traces.rep-eq TRACES-def F-Interrupt flip*: *Failures.rep-eq*)
apply (*safe*, *simp-all add*: *is-processT8*)
apply (*meson is-processT4-empty is-processT6*)
apply *auto*[*2*]
apply (*metis is-processT8*)
apply (*metis is-processT4-empty nonTickFree-n-frontTickFree process-charn*)
by (*metis append.right-neutral is-processT4-empty tickFree-Nil*)

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

3.5.3 Monotony

lemma *mono-Interrupt* : ‹$P \bigtriangleup Q \sqsubseteq P' \bigtriangleup Q'$› **if** ‹$P \sqsubseteq P'$› **and** ‹$Q \sqsubseteq Q'$›
proof (*unfold le-approx-def*, *intro conjI allI impI subsetI*)
show ‹$s \in \mathcal{D}\ (P' \bigtriangleup Q') \Longrightarrow s \in \mathcal{D}\ (P \bigtriangleup Q)$› **for** *s*
using ‹$P \sqsubseteq P'$›[*THEN le-approx1*] ‹$Q \sqsubseteq Q'$›[*THEN le-approx1*]
‹$P \sqsubseteq P'$›[*THEN le-approx2T*] *D-T* **by** (*simp add*: *D-Interrupt*) *blast*
next
show ‹$s \notin \mathcal{D}\ (P \bigtriangleup Q) \Longrightarrow \mathcal{R}_a\ (P \bigtriangleup Q)\ s = \mathcal{R}_a\ (P' \bigtriangleup Q')\ s$› **for** *s*
apply (*simp add*: *D-Interrupt Refusals-after-def F-Interrupt*,
intro subset-antisym subsetI; *simp*, *elim disjE*)
apply (*metis le-approx2T* ‹$P \sqsubseteq P'$›)
apply (*metis is-processT9 le-approx2T* ‹$P \sqsubseteq P'$›)
apply (*metis F-T append.right-neutral le-approx2* ‹$P \sqsubseteq P'$› ‹$Q \sqsubseteq Q'$›)
apply (*metis is-processT2 is-processT7 le-approx2T proc-ord2a* ‹$P \sqsubseteq P'$› ‹$Q \sqsubseteq Q'$›)
apply (*metis append-Nil2 is-processT9 le-approx2T self-append-conv2* ‹$P \sqsubseteq P'$› ‹$Q \sqsubseteq Q'$›)
apply *metis*
apply (*metis le-approx2T* ‹$P \sqsubseteq P'$›)
apply (*metis le-approx-lemma-T subset-eq* ‹$P \sqsubseteq P'$›)
apply (*metis is-processT8 le-approx2* ‹$P \sqsubseteq P'$› ‹$Q \sqsubseteq Q'$›)
apply (*metis is-processT2 is-processT7 le-approx2 le-approx2T* ‹$P \sqsubseteq P'$› ‹$Q \sqsubseteq Q'$›)
apply (*metis D-T le-approx2T* ‹$P \sqsubseteq P'$› ‹$Q \sqsubseteq Q'$›)
apply (*metis in-mono le-approx1* ‹$P \sqsubseteq P'$›)
by (*metis le-approx1 le-approx2T process-charn subsetD* ‹$P \sqsubseteq P'$› ‹$Q \sqsubseteq Q'$›)
next

show ‹$s \in$ *min-elems* ($\mathcal{D}$ ($P \triangle Q$)) $\Longrightarrow s \in \mathcal{T}$ ($P' \triangle Q'$)› **for** *s*
apply (*rule set-mp*[*of* ‹*min-elems* ($\mathcal{D}$ *P*) $\cup$ {*t1* @ *t2*| *t1 t2*. *t1* $\in \mathcal{T}$ P' $\wedge$ *tickFree t1* $\wedge$ *t2* $\in$ *min-elems* ($\mathcal{D}$ *Q*)}›])

apply (*rule Un-least*)
apply (*simp add*: *T-Interrupt le-approx3 le-supI1* ‹$P \sqsubseteq P'$›)
apply (*simp add*: *T-Interrupt subset-iff*, *metis le-approx-def subset-iff* ‹$Q \sqsubseteq Q'$›)
apply (*simp add*: *min-elems-def D-Interrupt less-list-def*)

by (*smt* (*verit*, *ccfv-threshold*) *D-imp-front-tickFree same-prefix-prefix Un-iff is-processT7 le-approx2T mem-Collect-eq same-append-eq that*(*1*))
qed

3.5.4 Properties

lemma *Interrupt-STOP* [*simp*] : ‹$P \triangle STOP = P$›
proof (*subst Process-eq-spec*, *safe*)
show ‹$t \in \mathcal{D}$ ($P \triangle STOP$) $\Longrightarrow t \in \mathcal{D}$ *P*› **for** *t*
by (*simp add*: *D-Interrupt D-STOP*)
next
show ‹$t \in \mathcal{D}$ $P \Longrightarrow t \in \mathcal{D}$ ($P \triangle STOP$)› **for** *t*
by (*simp add*: *D-Interrupt D-STOP*)
next
show ‹(t, X) $\in \mathcal{F}$ ($P \triangle STOP$) $\Longrightarrow$ (t, X) $\in \mathcal{F}$ *P*› **for** *t X*
by (*simp add*: *F-Interrupt STOP-projs*)
(*meson is-processT6-TR is-processT8 tick-T-F*)
next
show ‹(t, X) $\in \mathcal{F}$ $P \Longrightarrow$ (t, X) $\in \mathcal{F}$ ($P \triangle STOP$)› **for** *t X*
by (*simp add*: *F-Interrupt STOP-projs*)
(*metis F-T T-nonTickFree-imp-decomp*)
qed

lemma *STOP-Interrupt* [*simp*] : ‹$STOP \triangle P = P$›
proof (*subst Process-eq-spec*, *safe*)
show ‹$t \in \mathcal{D}$ ($STOP \triangle P$) $\Longrightarrow t \in \mathcal{D}$ *P*› **for** *t*
by (*simp add*: *D-Interrupt STOP-projs*)
next
show ‹$t \in \mathcal{D}$ $P \Longrightarrow t \in \mathcal{D}$ ($STOP \triangle P$)› **for** *t*
by (*simp add*: *D-Interrupt STOP-projs*)
next
show ‹(t, X) $\in \mathcal{F}$ ($STOP \triangle P$) $\Longrightarrow$ (t, X) $\in \mathcal{F}$ *P*› **for** *t X*
by (*simp add*: *F-Interrupt STOP-projs*)
(*metis is-processT6-TR is-processT8 self-append-conv2*)
next
show ‹(t, X) $\in \mathcal{F}$ $P \Longrightarrow$ (t, X) $\in \mathcal{F}$ ($STOP \triangle P$)› **for** *t X*
by (*auto simp add*: *F-Interrupt STOP-projs*)
qed

lemma *Interrupt-is-STOP-iff* : ‹$P \triangle Q = STOP \longleftrightarrow P = STOP \wedge Q = STOP$›
  by (*simp add*: *STOP-iff-T T-Interrupt set-eq-iff*)
    (*metis append-self-conv2 is-processT1-TR tickFree-Nil*)

lemma *Interrupt-BOT* [*simp*] : ‹$P \triangle \bot = \bot$›
  and *BOT-Interrupt* [*simp*] : ‹$\bot \triangle P = \bot$›
  by (*simp-all add*: *BOT-iff-Nil-D D-Interrupt D-BOT*)

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

lemma *SKIP-Interrupt-is-SKIP-Det* : ‹$SKIP\ r \triangle P = SKIP\ r \square P$›
proof (*subst Process-eq-spec*, *safe*)
  show ‹$t \in \mathcal{D}\ (SKIP\ r \triangle P) \Longrightarrow t \in \mathcal{D}\ (SKIP\ r \square P)$› **for** *t*
    by (*auto simp add*: *D-Interrupt D-Det SKIP-projs*)
next
  show ‹$t \in \mathcal{D}\ (SKIP\ r \square P) \Longrightarrow t \in \mathcal{D}\ (SKIP\ r \triangle P)$› **for** *t*
    by (*auto simp add*: *D-Interrupt D-Det SKIP-projs intro*: *tickFree-Nil*)
next
  show ‹$(t, X) \in \mathcal{F}\ (SKIP\ r \triangle P) \Longrightarrow (t, X) \in \mathcal{F}\ (SKIP\ r \square P)$› **for** *t X*
    by (*cases t*) (*auto simp add*: *F-Interrupt SKIP-projs F-Det intro*: *is-processT8*)
next
  show ‹$(t, X) \in \mathcal{F}\ (SKIP\ r \square P) \Longrightarrow (t, X) \in \mathcal{F}\ (SKIP\ r \triangle P)$› **for** *t X*
    by (*cases t*) (*auto simp add*: *F-Interrupt SKIP-projs F-Det intro*: *tickFree-Nil*)
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 ‹$s \in \mathcal{D}\ P$›
      | ‹$\exists t1\ t2.\ s = t1\ @\ t2 \wedge t1 \in \mathcal{T}\ P \wedge tickFree\ t1 \wedge t2 \in \mathcal{D}\ (Q \triangle R)$›
      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 ‹$\exists t1\ t2.\ s = t1\ @\ t2 \wedge t1 \in \mathcal{T}\ P \wedge tickFree\ t1 \wedge t2 \in \mathcal{D}\ (Q \triangle R)$›
      then obtain *t1 t2* **where** ∗ : ‹$s = t1\ @\ t2$› ‹$t1 \in \mathcal{T}\ P$›
        ‹*tickFree t1*› ‹$t2 \in \mathcal{D}\ (Q \triangle R)$› **by** *blast*
      from ∗(*4*) **consider** ‹$t2 \in \mathcal{D}\ Q$›
        | ‹$\exists u1\ u2.\ t2 = u1\ @\ u2 \wedge u1 \in \mathcal{T}\ Q \wedge tickFree\ u1 \wedge u2 \in \mathcal{D}\ R$›
        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 ‹$\exists\, u1\ u2.\ t2 = u1\ @\ u2 \wedge u1 \in \mathcal{T}\ Q \wedge$ *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-iff*)
qed
qed
next
fix *s*
assume ‹$s \in \mathcal{D}$ *?rhs*›
then consider ‹$s \in \mathcal{D}\ (P \bigtriangleup Q)$›
| ‹$\exists\, t1\ t2.\ s = t1\ @\ t2 \wedge t1 \in \mathcal{T}\ (P \bigtriangleup Q) \wedge$ *tickFree* $t1 \wedge t2 \in \mathcal{D}\ R$›
by (*simp add*: *D-Interrupt*) *blast*
thus ‹$s \in \mathcal{D}$ *?lhs*›
proof *cases*
show ‹$s \in \mathcal{D}\ (P \bigtriangleup Q) \Longrightarrow s \in \mathcal{D}$ *?lhs*› **by** (*simp add*: *D-Interrupt*) *blast*
next
assume ‹$\exists\, t1\ t2.\ s = t1\ @\ t2 \wedge t1 \in \mathcal{T}\ (P \bigtriangleup Q) \wedge$ *tickFree* $t1 \wedge t2 \in \mathcal{D}\ R$›
then obtain *t1 t2* **where** $*$: ‹$s = t1\ @\ t2$› ‹$t1 \in \mathcal{T}\ (P \bigtriangleup Q)$›
‹*tickFree t1*› ‹$t2 \in \mathcal{D}\ R$› **by** *blast*
from $*(2)$ **consider** ‹$t1 \in \mathcal{T}\ P$›
| ‹$\exists\, u1\ u2.\ t1 = u1\ @\ u2 \wedge u1 \in \mathcal{T}\ P \wedge$ *tickFree* $u1 \wedge u2 \in \mathcal{T}\ Q$›
by (*simp add*: *T-Interrupt*) *blast*
thus ‹$s \in \mathcal{D}$ *?lhs*›
proof *cases*
show ‹$t1 \in \mathcal{T}\ P \Longrightarrow s \in \mathcal{D}$ *?lhs*›
by (*simp add*: *D-Interrupt* $*(1)$)
(*metis* $*(3, 4)$ *Nil-elem-T append-Nil tickFree-Nil*)
next
show ‹$\exists\, u1\ u2.\ t1 = u1\ @\ u2 \wedge u1 \in \mathcal{T}\ P \wedge$ *tickFree* $u1 \wedge u2 \in \mathcal{T}\ Q \Longrightarrow s \in \mathcal{D}$ *?lhs*›
by (*simp add*: *D-Interrupt* $*(1)$)
(*metis* $*(3, 4)$ *append.assoc tickFree-append-iff*)
qed
qed
next
fix *s X*
assume *same-div*: ‹$\mathcal{D}$ *?lhs* $= \mathcal{D}$ *?rhs*›
assume ‹$(s, X) \in \mathcal{F}$ *?lhs*›
then consider ‹$s \in \mathcal{D}$ *?lhs*›
| ‹$\exists\, t1\ r.\ s = t1\ @\ [\checkmark(r)] \wedge t1\ @\ [\checkmark(r)] \in \mathcal{T}\ P$›
| ‹$\exists\, r.\ s\ @\ [\checkmark(r)] \in \mathcal{T}\ P \wedge \checkmark(r) \notin X$›
| ‹$(s, X) \in \mathcal{F}\ P \wedge$ *tickFree* $s \wedge ([], X) \in \mathcal{F}\ (Q \bigtriangleup R)$›
| ‹$\exists\, t1\ t2.\ s = t1\ @\ t2 \wedge t1 \in \mathcal{T}\ P \wedge$ *tickFree* $t1 \wedge (t2, X) \in \mathcal{F}\ (Q \bigtriangleup R) \wedge t2 \neq []$›

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

by (*auto simp add*: *F-Interrupt T-Interrupt*)
next
from $*$(*1*, *2*, *3*) **show** ‹*t2* @ [✓(*r*)] ∈ $\mathcal{T}$ *Q* ⟹ ✓(*r*) ∉ *X* ⟹ (*s*, *X*) ∈ $\mathcal{F}$ *?rhs*› **for** *r*
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** ‹*t2* = *u1* @ *u2* ⟹ *u1* ∈ $\mathcal{T}$ *Q* ⟹ *tickFree u1* ⟹ (*u2*, *X*) ∈ $\mathcal{F}$ *R* ⟹ *u2* ≠ [] ⟹ (*s*, *X*) ∈ $\mathcal{F}$ *?rhs*› **for** *u1 u2*
by (*simp add*: *F-Interrupt T-Interrupt*)
(*metis* (*mono-tags*, *lifting*) *append-assoc tickFree-append-iff*)
next
from $*$(*1*, *2*, *3*) **show** ‹*t2* ∈ $\mathcal{T}$ *Q* ⟹ *tickFree t2* ⟹
[✓(*r*)] ∈ $\mathcal{T}$ *R* ⟹ ✓(*r*) ∉ *X* ⟹ (*s*, *X*) ∈ $\mathcal{F}$ *?rhs*› **for** *r*
by (*simp add*: *F-Interrupt T-Interrupt*) (*metis Diff-insert-absorb*)
qed
next
show ‹∃ *r*. *s* ∈ $\mathcal{T}$ *P* ∧ *tickFree s* ∧ [✓(*r*)] ∈ $\mathcal{T}$ (*Q* △ *R*) ∧ ✓(*r*) ∉ *X* ⟹ (*s*, *X*) ∈ $\mathcal{F}$ *?rhs*›
by (*simp add*: *F-Interrupt T-Interrupt*)
(*metis Diff-insert-absorb append-eq-Cons-conv non-tickFree-tick tick-Free-append-iff*)
qed
next
fix *s X*
assume *same-div* : ‹$\mathcal{D}$ *?lhs* = $\mathcal{D}$ *?rhs*›
assume ‹(*s*, *X*) ∈ $\mathcal{F}$ *?rhs*›
then consider ‹*s* ∈ $\mathcal{D}$ *?rhs*›
| ‹∃ *t1 r*. *s* = *t1* @ [✓(*r*)] ∧ *t1* @ [✓(*r*)] ∈ $\mathcal{T}$ (*P* △ *Q*)›
| *r* **where** ‹*s* @ [✓(*r*)] ∈ $\mathcal{T}$ (*P* △ *Q*)› ‹✓(*r*) ∉ *X*›
| ‹(*s*, *X*) ∈ $\mathcal{F}$ (*P* △ *Q*) ∧ *tickFree s* ∧ ([], *X*) ∈ $\mathcal{F}$ *R*›
| ‹∃ *t1 t2*. *s* = *t1* @ *t2* ∧ *t1* ∈ $\mathcal{T}$ (*P* △ *Q*) ∧ *tickFree t1* ∧ (*t2*, *X*) ∈ $\mathcal{F}$ *R* ∧ *t2* ≠ []›
| ‹∃ *r*. *s* ∈ $\mathcal{T}$ (*P* △ *Q*) ∧ *tickFree s* ∧ [✓(*r*)] ∈ $\mathcal{T}$ *R* ∧ ✓(*r*) ∉ *X*›
by (*subst* (*asm*) *F-Interrupt*, *simp add*: *D-Interrupt*) *blast*
thus ‹(*s*, *X*) ∈ $\mathcal{F}$ *?lhs*›
proof *cases*
from *same-div D-F* **show** ‹*s* ∈ $\mathcal{D}$ *?rhs* ⟹ (*s*, *X*) ∈ $\mathcal{F}$ *?lhs*› **by** *blast*
next
show ‹∃ *t1 r*. *s* = *t1* @ [✓(*r*)] ∧ *t1* @ [✓(*r*)] ∈ $\mathcal{T}$ (*P* △ *Q*) ⟹ (*s*, *X*) ∈ $\mathcal{F}$ *?lhs*›
by (*simp add*: *F-Interrupt T-Interrupt*)
(*metis last-append self-append-conv snoc-eq-iff-butlast*)
next
fix *r* **assume** ‹*s* @ [✓(*r*)] ∈ $\mathcal{T}$ (*P* △ *Q*)› ‹✓(*r*) ∉ *X*›

from *this*(*1*) **consider** ‹*s* @ [✓(*r*)] ∈ $\mathcal{T}$ *P*›
| *t1 t2* **where** ‹*s* @ [✓(*r*)] = *t1* @ *t2*› ‹*t1* ∈ $\mathcal{T}$ *P*› ‹*tickFree t1*› ‹*t2* ∈ $\mathcal{T}$ *Q*›
by (*simp add*: *T-Interrupt*) *blast*
thus ‹(*s*, *X*) ∈ $\mathcal{F}$ *?lhs*›
proof *cases*
show ‹*s* @ [✓(*r*)] ∈ $\mathcal{T}$ *P* ⟹ (*s*, *X*) ∈ $\mathcal{F}$ *?lhs*›
by (*simp add*: *F-Interrupt*) (*metis Diff-insert-absorb* ‹✓(*r*) ∉ *X*›)
next
show ‹*s* @ [✓(*r*)] = *t1* @ *t2* ⟹ *t1* ∈ $\mathcal{T}$ *P* ⟹ *tickFree t1* ⟹ *t2* ∈ $\mathcal{T}$ *Q* ⟹ (*s*, *X*) ∈ $\mathcal{F}$ *?lhs*› **for** *t1 t2*

apply (*simp add*: *F-Interrupt T-Interrupt*, *safe*, *simp-all*)
apply (*smt* (*z3*) *Diff-insert-absorb T-nonTickFree-imp-decomp* ‹✓(*r*) ∉ *X*› *append.assoc append1-eq-conv append-self-conv2 non-tickFree-tick tickFree-append-iff*)

apply (*metis* ‹*s* @ [✓(*r*)] ∈ $\mathcal{T}$ (*P* △ *Q*)› *append-T-imp-tickFree list.discI*)
apply (*smt* (*z3*) *Diff-insert-absorb T-nonTickFree-imp-decomp* ‹✓(*r*) ∉ *X*› *append1-eq-conv append-assoc is-processT6-TR non-tickFree-tick tickFree-append-iff*)
apply (*smt* (*z3*) *Diff-insert-absorb T-nonTickFree-imp-decomp* ‹✓(*r*) ∉ *X*› *append1-eq-conv append-assoc non-tickFree-tick self-append-conv2 tickFree-append-iff*)
done
qed
next
assume *assm* : ‹(*s*, *X*) ∈ $\mathcal{F}$ (*P* △ *Q*) ∧ *tickFree s* ∧ ([], *X*) ∈ $\mathcal{F}$ *R*›
from *assm*[*THEN conjunct1*] **consider** ‹*s* ∈ $\mathcal{D}$ (*P* △ *Q*)›
| *t1 r* **where** ‹*s* = *t1* @ [✓(*r*)]› ‹*t1* @ [✓(*r*)] ∈ $\mathcal{T}$ *P*›
| *r* **where** ‹*s* @ [✓(*r*)] ∈ $\mathcal{T}$ *P*› ‹✓(*r*) ∉ *X*›
| ‹(*s*, *X*) ∈ $\mathcal{F}$ *P*› ‹*tickFree s*› ‹([], *X*) ∈ $\mathcal{F}$ *Q*›
| *t1 t2* **where** ‹*s* = *t1* @ *t2*› ‹*t1* ∈ $\mathcal{T}$ *P*› ‹*tickFree t1*› ‹(*t2*, *X*) ∈ $\mathcal{F}$ *Q*› ‹*t2* ≠ []›
| *r* **where** ‹*s* ∈ $\mathcal{T}$ *P*› ‹*tickFree s*› ‹[✓(*r*)] ∈ $\mathcal{T}$ *Q*› ‹✓(*r*) ∉ *X*›
by (*simp add*: *F-Interrupt D-Interrupt*) *blast*
thus ‹(*s*, *X*) ∈ $\mathcal{F}$ *?lhs*›
proof *cases*
assume ‹*s* ∈ $\mathcal{D}$ (*P* △ *Q*)›
hence ‹*s* ∈ $\mathcal{D}$ *?rhs*› **by** (*simp add*: *D-Interrupt*)
with *same-div D-F* **show** ‹(*s*, *X*) ∈ $\mathcal{F}$ *?lhs*› **by** *blast*
next
show ‹*s* = *t1* @ [✓(*r*)] ⟹ *t1* @ [✓(*r*)] ∈ $\mathcal{T}$ *P* ⟹ (*s*, *X*) ∈ $\mathcal{F}$ *?lhs*› **for** *t1 r*
by (*simp add*: *F-Interrupt*)
next
show ‹*s* @ [✓(*r*)] ∈ $\mathcal{T}$ *P* ⟹ ✓(*r*) ∉ *X* ⟹ (*s*, *X*) ∈ $\mathcal{F}$ *?lhs*› **for** *r*
by (*simp add*: *F-Interrupt*) (*metis Diff-insert-absorb*)
next
show ‹(*s*, *X*) ∈ $\mathcal{F}$ *P* ⟹ *tickFree s* ⟹ ([], *X*) ∈ $\mathcal{F}$ *Q* ⟹ (*s*, *X*) ∈ $\mathcal{F}$ *?lhs*›
by (*simp add*: *F-Interrupt assm*[*THEN conjunct2*])
next

show ‹$s = t1\ @\ t2 \Longrightarrow t1 \in \mathcal{T}\ P \Longrightarrow$ *tickFree t1* $\Longrightarrow (t2, X) \in \mathcal{F}\ Q \Longrightarrow$
$t2 \neq [] \Longrightarrow (s, X) \in \mathcal{F}$ *?lhs*› **for** *t1 t2*
by (*simp add*: *F-Interrupt*) (*metis assm*[*THEN conjunct2*] *tickFree-append-iff*)
next
show ‹$s \in \mathcal{T}\ P \Longrightarrow$ *tickFree s* $\Longrightarrow [\checkmark(r)] \in \mathcal{T}\ Q \Longrightarrow \checkmark(r) \notin X \Longrightarrow (s, X)$
$\in \mathcal{F}$ *?lhs*› **for** *r*
by (*simp add*: *F-Interrupt T-Interrupt*) (*metis Diff-insert-absorb*)
qed
next
assume ‹$\exists\, t1\ t2.\ s = t1\ @\ t2 \wedge t1 \in \mathcal{T}\ (P \triangle Q) \wedge$
tickFree t1 $\wedge (t2, X) \in \mathcal{F}\ R \wedge t2 \neq []$›
then obtain *t1 t2* **where** $*$: ‹$s = t1\ @\ t2$› ‹$t1 \in \mathcal{T}\ (P \triangle Q)$›
‹*tickFree t1*› ‹$(t2, X) \in \mathcal{F}\ R$› ‹$t2 \neq []$› **by** *blast*
from $*(2)$ **consider** ‹$t1 \in \mathcal{T}\ P$›
| ‹$\exists\, u1\ u2.\ t1 = u1\ @\ u2 \wedge u1 \in \mathcal{T}\ P \wedge$ *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** ‹$\exists\, u1\ u2.\ t1 = u1\ @\ u2 \wedge u1 \in \mathcal{T}\ P \wedge$
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 ‹$\exists\, r.\ s \in \mathcal{T}\ (P \triangle Q) \wedge$ *tickFree s* $\wedge [\checkmark(r)] \in \mathcal{T}\ R \wedge \checkmark(r) \notin X \Longrightarrow (s,$
$X) \in \mathcal{F}$ *?lhs*›
by (*simp add*: *F-Interrupt T-Interrupt*)
(*metis Diff-insert-absorb Nil-elem-T append.right-neutral*
append-Nil tickFree-append-iff)
qed
qed

thus ‹*?lhs* = *?rhs*›
by (*cases* ‹$P = \bot$›; *cases* ‹$Q = \bot$›; *cases* ‹$R = \bot$›) *simp-all*
qed

3.5.5 Continuity

context begin

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

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

private lemma *cont-left-prem-Interrupt* : ‹(⨆ *i*. *Y i*) △ *Q* = (⨆ *i*. *Y i* △ *Q*)›
  (**is** ‹*?lhs* = *?rhs*›) **if** *chain* : ‹*chain Y*›
proof (*subst Process-eq-spec-optimized*, *safe*)
  show ‹*s* ∈ 𝒟 *?lhs* ⟹ *s* ∈ 𝒟 *?rhs*› **for** *s*
    by (*simp add*: *limproc-is-thelub chain chain-Interrupt-left*
        D-Interrupt T-LUB D-LUB) *blast*
next
  fix *s*
  define *S*
    where ‹*S i* ≡ {*t1*. *s* = *t1* ∧ *t1* ∈ 𝒟 (*Y i*)} ∪
                {*t1*. ∃ *t2*. *s* = *t1* @ *t2* ∧ *t1* ∈ 𝒯 (*Y i*) ∧ *tickFree t1* ∧ *t2* ∈ 𝒟 *Q*}›
for *i*
  assume ‹*s* ∈ 𝒟 *?rhs*›
  hence ‹*s* ∈ 𝒟 (*Y i* △ *Q*)› **for** *i*
    by (*simp add*: *limproc-is-thelub D-LUB chain-Interrupt-left chain*)
  hence ‹*S i* ≠ {}› **for** *i* **by** (*simp add*: *S-def D-Interrupt*) *blast*
  moreover have ‹*finite* (*S 0*)›
    unfolding *S-def* **by** (*prove-finite-subset-of-prefixes s*)
  moreover have ‹*S* (*Suc i*) ⊆ *S i*› **for** *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 ‹(⋂ *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* ∈ 𝒟 *?lhs*›
  proof (*cases* ‹∀ *i*. *s* ∈ 𝒟 (*Y i*)›)
    case *True*
    thus ‹*s* ∈ 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* ∈ 𝒯 (*Y j*)› ‹ *tickFree t1*› ‹*t2*
∈ 𝒟 *Q*›
      by (*simp add*: *S-def*) *blast*
    from ∗ *D-T* **have** ‹∀ *i*. *t1* ∈ 𝒯 (*Y i*)› **by** (*simp add*: *S-def*) *blast*
    with ∗∗(*1*, *3*, *4*) **show** ‹*s* ∈ 𝒟 *?lhs*›
      by (*simp add*: *D-Interrupt limproc-is-thelub T-LUB chain*) *blast*
  qed
next
  show ‹(*s*, *X*) ∈ ℱ *?lhs* ⟹ (*s*, *X*) ∈ ℱ *?rhs*› **for** *s X*
    by (*simp add*: *limproc-is-thelub chain chain-Interrupt-left*
        F-Interrupt F-LUB T-LUB D-LUB, *safe*; *simp*; *metis*)
next
  assume *same-div* : ‹𝒟 ((⨆ *i*. *Y i*) △ *Q*) = 𝒟 (⨆ *i*. *Y i* △ *Q*)›
  fix *s X* **assume** ‹(*s*, *X*) ∈ ℱ (⨆ *i*. *Y i* △ *Q*)›
  show ‹(*s*, *X*) ∈ ℱ ((⨆ *i*. *Y i*) △ *Q*)›
  proof (*cases* ‹*s* ∈ 𝒟 (⨆ *i*. *Y i* △ *Q*)›)

show ‹$s \in \mathcal{D}$ ($\bigsqcup i.$ Y i △ Q) $\Longrightarrow$ (s, X) $\in \mathcal{F}$ (($\bigsqcup i.$ Y i) △ Q)›
by (*simp add*: *is-processT8 same-div*)
next
assume ‹$s \notin \mathcal{D}$ ($\bigsqcup i.$ Y i △ Q)›
then obtain j **where** ‹$s \notin \mathcal{D}$ (Y j △ Q)›
by (*auto simp add*: *limproc-is-thelub chain-Interrupt-left* ‹*chain Y*› *D-LUB*)
moreover from ‹(s, X) $\in \mathcal{F}$ ($\bigsqcup i.$ Y i △ Q)› **have** ‹(s, X) $\in \mathcal{F}$ (Y j △ Q)›
by (*simp add*: *limproc-is-thelub chain-Interrupt-left* ‹*chain Y*› *F-LUB*)
ultimately show ‹(s, X) $\in \mathcal{F}$ (($\bigsqcup i.$ Y i) △ Q)›
by (*fact le-approx2*[*OF mono-Interrupt*[*OF is-ub-thelub*[*OF* ‹*chain Y*›] *below-refl*], *THEN iffD2*])
qed
qed

private lemma *cont-right-prem-Interrupt* : ‹S △ ($\bigsqcup i.$ Y i) = ($\bigsqcup i.$ S △ Y i)› **if** ‹*chain Y*›
proof (*subst Process-eq-spec-optimized*, *safe*)
show ‹$s \in \mathcal{D}$ (S △ ($\bigsqcup i.$ Y i)) $\Longrightarrow$ $s \in \mathcal{D}$ ($\bigsqcup i.$ S △ Y i)› **for** s
by (*auto simp add*: *D-Interrupt limproc-is-thelub* ‹*chain Y*› *chain-Interrupt-right D-LUB*)
next
fix s **assume** ‹$s \in \mathcal{D}$ ($\bigsqcup i.$ S △ Y i)›
show ‹$s \in \mathcal{D}$ (S △ ($\bigsqcup i.$ Y i))›
proof (*cases* ‹$s \in \mathcal{D}$ S›)
show ‹$s \in \mathcal{D}$ S $\Longrightarrow$ $s \in \mathcal{D}$ (S △ ($\bigsqcup i.$ Y i))› **by** (*simp add*: *D-Interrupt*)
next
assume ‹$s \notin \mathcal{D}$ S›
thm *D-Interrupt*
define T **where** ‹T i $\equiv$ {$t1$. $\exists$ $t2$ r. s = $t1$ @ $t2$ $\wedge$ $t1$ $\in \mathcal{T}$ S $\wedge$ *tickFree* $t1$ $\wedge$ $t2 \in \mathcal{D}$ (Y i)}› **for** i
from ‹$s \notin \mathcal{D}$ S› ‹$s \in \mathcal{D}$ ($\bigsqcup i.$ S △ Y i)› **have** ‹T i $\neq$ {}› **for** i
by (*simp add*: *T-def limproc-is-thelub chain-Interrupt-right* ‹*chain Y*› *D-LUB D-Interrupt*) *blast*
moreover have ‹*finite* (T *0*)›
unfolding *T-def* **by** (*prove-finite-subset-of-prefixes s*)
moreover have ‹T (*Suc* i) $\subseteq$ T i› **for** i
unfolding *T-def* **apply** (*intro allI Un-mono subsetI*; *simp*)
by (*metis le-approx1 po-class.chainE subset-iff* ‹*chain Y*›)
ultimately have ‹($\bigcap i.$ T i) $\neq$ {}› **by** (*rule Inter-nonempty-finite-chained-sets*)
then obtain *t1* **where** ‹$\forall i.$ $t1$ $\in$ T i› **by** *auto*
then obtain *t2* **where** $*$: ‹s = $t1$ @ $t2$› ‹$t1 \in \mathcal{T}$ S› ‹*tickFree t1*› ‹$\forall i.$ $t2 \in \mathcal{D}$ (Y i)›
by (*simp add*: *T-def*) *blast*
thus ‹$s \in \mathcal{D}$ (S △ ($\bigsqcup i.$ Y i))›
by (*simp add*: *D-Interrupt limproc-is-thelub* ‹*chain Y*› *D-LUB*) *blast*
qed
next

show ‹$(s, X) \in \mathcal{F}$ $(S \triangle (\bigsqcup i.\ Y\ i)) \Longrightarrow (s, X) \in \mathcal{F}$ $(\bigsqcup i.\ S \triangle Y\ i)$› **for** s X
  by (*simp add*: *F-Interrupt limproc-is-thelub* ‹*chain Y*› *chain-Interrupt-right F-LUB T-LUB D-LUB*)
  (*elim disjE exE conjE*; *metis*)
next
  assume *same-div* : ‹$\mathcal{D}$ $(S \triangle (\bigsqcup i.\ Y\ i)) = \mathcal{D}$ $(\bigsqcup i.\ S \triangle Y\ i)$›
  fix s X **assume** ‹$(s, X) \in \mathcal{F}$ $(\bigsqcup i.\ S \triangle Y\ i)$›
  show ‹$(s, X) \in \mathcal{F}$ $(S \triangle (\bigsqcup i.\ Y\ i))$›
  proof (*cases* ‹$s \in \mathcal{D}$ $(\bigsqcup i.\ S \triangle Y\ i)$›)
    show ‹$s \in \mathcal{D}$ $(\bigsqcup i.\ S \triangle Y\ i) \Longrightarrow (s, X) \in \mathcal{F}$ $(S \triangle (\bigsqcup i.\ Y\ i))$›
      by (*simp add*: *is-processT8 same-div*)
  next
    assume ‹$s \notin \mathcal{D}$ $(\bigsqcup i.\ S \triangle Y\ i)$›
    then obtain j **where** ‹$s \notin \mathcal{D}$ $(S \triangle Y\ j)$›
      by (*auto simp add*: *limproc-is-thelub chain-Interrupt-right* ‹*chain Y*› *D-LUB*)
    moreover from ‹$(s, X) \in \mathcal{F}$ $(\bigsqcup i.\ S \triangle Y\ i)$› **have** ‹$(s, X) \in \mathcal{F}$ $(S \triangle Y\ j)$›
      by (*simp add*: *limproc-is-thelub chain-Interrupt-right* ‹*chain Y*› *F-LUB*)
    ultimately show ‹$(s, X) \in \mathcal{F}$ $(S \triangle (\bigsqcup i.\ Y\ i))$›
      by (*fact le-approx2*[*OF mono-Interrupt*[*OF below-refl is-ub-thelub*[*OF* ‹*chain Y*›]], *THEN iffD2*])
  qed
qed

lemma *Interrupt-cont* [*simp*] :
  ‹*cont* $(\lambda x.\ f\ x \triangle g\ x)$› **if** ‹*cont f*› **and** ‹*cont g*›
proof (*rule cont-apply*[**where** f = ‹$\lambda x\ y.\ f\ x \triangle y$›])
  show ‹*cont g*› **by** (*fact* ‹*cont g*›)
next
  show ‹*cont* $((\triangle)\ (f\ x))$› **for** x
  proof (*rule contI2*)
    show ‹*monofun* $((\triangle)\ (f\ x))$› **by** (*simp add*: *mono-Interrupt monofunI*)
  next
    show ‹*chain* $Y \Longrightarrow f\ x \triangle (\bigsqcup i.\ Y\ i) \sqsubseteq (\bigsqcup i.\ f\ x \triangle Y\ i)$› **for** Y
      by (*simp add*: *cont-right-prem-Interrupt*)
  qed
next
  show ‹*cont* $(\lambda x.\ f\ x \triangle y)$› **for** y
  proof (*rule contI2*)
    show ‹*monofun* $(\lambda x.\ f\ x \triangle y)$› **by** (*simp add*: *cont2monofunE mono-Interrupt monofunI* ‹*cont f*›)
  next
    show ‹*chain* $Y \Longrightarrow f\ (\bigsqcup i.\ Y\ i) \triangle y \sqsubseteq (\bigsqcup i.\ f\ (Y\ i) \triangle y)$› **for** Y
      by (*simp add*: *ch2ch-cont cont2contlubE cont-left-prem-Interrupt* ‹*cont f*›)
  qed
qed

end

3.6 Monotonies

3.6.1 The Throw Operator

lemma *mono-Throw-F-right* :
‹($\bigwedge a.\ a \in A \Longrightarrow a \in \alpha(P) \Longrightarrow Q\ a \sqsubseteq_F Q'\ a) \Longrightarrow P\ \Theta\ a \in A.\ Q\ a \sqsubseteq_F P\ \Theta\ a \in A.\ Q'\ a$›
unfolding *failure-refine-def* **by** (*simp add*: *F-Throw subset-iff disjoint-iff*)
(*metis events-of-memI in-set-conv-decomp*)

lemma *mono-Throw-T-right* :
‹($\bigwedge a.\ a \in A \Longrightarrow a \in \alpha(P) \Longrightarrow Q\ a \sqsubseteq_T Q'\ a) \Longrightarrow P\ \Theta\ a \in A.\ Q\ a \sqsubseteq_T P\ \Theta\ a \in A.\ Q'\ a$›
unfolding *trace-refine-def* **by** (*simp add*: *T-Throw subset-iff disjoint-iff*)
(*metis events-of-memI in-set-conv-decomp*)

lemma *mono-Throw-D-right* :
‹($\bigwedge a.\ a \in A \Longrightarrow a \in \alpha(P) \Longrightarrow Q\ a \sqsubseteq_D Q'\ a) \Longrightarrow P\ \Theta\ a \in A.\ Q\ a \sqsubseteq_D P\ \Theta\ a \in A.\ Q'\ a$›
unfolding *divergence-refine-def* **by** (*simp add*: *D-Throw subset-iff disjoint-iff*)
(*metis events-of-memI in-set-conv-decomp*)

lemma *mono-Throw-FD* : ‹$P\ \Theta\ a \in A.\ Q\ a \sqsubseteq_{FD} P'\ \Theta\ a \in A.\ Q'\ a$›
if ‹$P \sqsubseteq_{FD} P'$› **and** ‹$\bigwedge a.\ a \in A \Longrightarrow a \in \alpha(P) \Longrightarrow Q\ a \sqsubseteq_{FD} Q'\ a$›
proof (*rule trans-FD*)
from ‹$P \sqsubseteq_{FD} P'$› **show** ‹$P\ \Theta\ a \in A.\ Q\ a \sqsubseteq_{FD} P'\ \Theta\ a \in A.\ Q\ a$›
by (*simp add*: *refine-defs Throw-projs subset-iff*, *safe*, *simp-all flip*: *T-F-spec*, *blast*)
next
show ‹$P'\ \Theta\ a \in A.\ Q\ a \sqsubseteq_{FD} P'\ \Theta\ a \in A.\ Q'\ a$›
by (*meson anti-mono-events-of-FD failure-divergence-refine-def*
mono-Throw-D-right mono-Throw-F-right subsetD that)
qed

lemma *mono-Throw-DT* : ‹$P\ \Theta\ a \in A.\ Q\ a \sqsubseteq_{DT} P'\ \Theta\ a \in A.\ Q'\ a$›
if ‹$P \sqsubseteq_{DT} P'$› **and** ‹$\bigwedge a.\ a \in A \Longrightarrow a \in \alpha(P) \Longrightarrow Q\ a \sqsubseteq_{DT} Q'\ a$›
proof (*rule trans-DT*)
from ‹$P \sqsubseteq_{DT} P'$› **show** ‹$P\ \Theta\ a \in A.\ Q\ a \sqsubseteq_{DT} P'\ \Theta\ a \in A.\ Q\ a$›
by (*simp add*: *refine-defs Throw-projs subset-iff*, *safe*, *auto*)
next
show ‹$P'\ \Theta\ a \in A.\ Q\ a \sqsubseteq_{DT} P'\ \Theta\ a \in A.\ Q'\ a$›
by (*meson anti-mono-events-of-DT leDT-imp-leD leDT-imp-leT leD-leT-imp-leDT*
mono-Throw-D-right mono-Throw-T-right subsetD that)
qed

lemmas *monos-Throw* = *mono-Throw mono-Throw-FD mono-Throw-DT mono-Throw-F-right mono-Throw-D-right mono-Throw-T-right*

3.6.2 The Interrupt Operator

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-Interrupt-D-right* : ‹$Q \sqsubseteq_D Q' \Longrightarrow P \triangle Q \sqsubseteq_D P \triangle Q'$›
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 failure-refine-def divergence-refine-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*)

lemmas *monos-Interrupt* = *mono-Interrupt mono-Interrupt-FD mono-Interrupt-DT mono-Interrupt-D-right mono-Interrupt-T*

3.6.3 Global Deterministic Choice

lemma *mono-GlobalDet-DT* : ‹$(\bigwedge a.\ a \in A \Longrightarrow P\ a \sqsubseteq_{DT} Q\ a) \Longrightarrow (\square a \in A.\ P\ a) \sqsubseteq_{DT} (\square a \in A.\ Q\ a)$›
and *mono-GlobalDet-T* : ‹$(\bigwedge a.\ a \in A \Longrightarrow P\ a \sqsubseteq_T Q\ a) \Longrightarrow (\square a \in A.\ P\ a) \sqsubseteq_T (\square a \in A.\ Q\ a)$›
and *mono-GlobalDet-D* : ‹$(\bigwedge a.\ a \in A \Longrightarrow P\ a \sqsubseteq_D Q\ a) \Longrightarrow (\square a \in A.\ P\ a) \sqsubseteq_D (\square a \in A.\ Q\ a)$›
by (*auto simp add*: *refine-defs GlobalDet-projs*)

lemma *mono-GlobalDet-FD* : ‹$(\bigwedge a.\ a \in A \Longrightarrow P\ a \sqsubseteq_{FD} Q\ a) \Longrightarrow (\square a \in A.\ P\ a) \sqsubseteq_{FD} (\square a \in A.\ Q\ a)$›
by (*simp add*: *refine-defs GlobalDet-projs subset-iff*) (*meson F-T T-F in-mono*)

lemmas *monos-GlobalDet* = *mono-GlobalDet mono-GlobalDet-FD mono-GlobalDet-DT mono-GlobalDet-T mono-GlobalDet-D*

lemma *GlobalNdet-FD-GlobalDet* : ‹$(\sqcap a \in A.\ P\ a) \sqsubseteq_{FD} (\square a \in A.\ P\ a)$›
and *GlobalNdet-DT-GlobalDet* : ‹$(\sqcap a \in A.\ P\ a) \sqsubseteq_{DT} (\square a \in A.\ P\ a)$›
and *GlobalNdet-F-GlobalDet* : ‹$(\sqcap a \in A.\ P\ a) \sqsubseteq_F (\square a \in A.\ P\ a)$›
and *GlobalNdet-T-GlobalDet* : ‹$(\sqcap a \in A.\ P\ a) \sqsubseteq_T (\square a \in A.\ P\ a)$›

and *GlobalNdet-D-GlobalDet* : ‹($\sqcap a \in A.\ P\ a) \sqsubseteq_D (\square a \in A.\ P\ a)$›
by (*simp-all add*: *refine-defs GlobalDet-projs GlobalNdet-projs subset-iff*, *safe*)
(*blast*, *blast intro*: *is-processT8*, *metis append-Nil is-processT6-TR-notin*)+

lemmas *GlobalNdet-le-GlobalDet* = *GlobalNdet-FD-GlobalDet GlobalNdet-DT-GlobalDet GlobalNdet-F-GlobalDet GlobalNdet-T-GlobalDet GlobalNdet-D-GlobalDet*

3.6.4 Multiple Synchronization Product

lemma *mono-MultiSync-FD* :
‹$(\bigwedge m.\ m \in\#\ M \Longrightarrow P\ m \sqsubseteq_{FD} Q\ m) \Longrightarrow ([\![S]\!]\ m \in\#\ M.\ P\ m) \sqsubseteq_{FD} ([\![S]\!]\ m \in\#\ M.\ Q\ m)$›
and *mono-MultiSync-DT* :
‹$(\bigwedge m.\ m \in\#\ M \Longrightarrow P\ m \sqsubseteq_{DT} Q\ m) \Longrightarrow ([\![S]\!]\ m \in\#\ M.\ P\ m) \sqsubseteq_{DT} ([\![S]\!]\ m \in\#\ M.\ Q\ m)$›
by (*cases* ‹$M = \{\#\}$›, *simp*, *erule mset-induct-nonempty*, *simp-all add*: *monos-Sync*)+

lemmas *mono-MultiInter-FD* = *mono-MultiSync-FD*[**where** S = ‹{}›]
and *mono-MultiInter-DT* = *mono-MultiSync-DT*[**where** S = ‹{}›]
and *mono-MultiPar-FD* = *mono-MultiSync-FD*[**where** S = ‹*UNIV*›]
and *mono-MultiPar-DT* = *mono-MultiSync-DT*[**where** S = ‹*UNIV*›]

lemmas *monos-MultiSync* = *mono-MultiSync mono-MultiSync-FD mono-MultiSync-DT*
and *monos-MultiPar* = *mono-MultiPar mono-MultiPar-FD mono-MultiPar-DT*
and *monos-MultiInter* = *mono-MultiInter mono-MultiInter-FD mono-MultiInter-DT*

Monotony doesn't hold for ($\sqsubseteq_F$), ($\sqsubseteq_T$) and ($\sqsubseteq_D$).

3.6.5 Multiple Sequential Composition

lemma *mono-MultiSeq-FD* :
‹$(\bigwedge x.\ x \in set\ L \Longrightarrow P\ x \sqsubseteq_{FD} Q\ x) \Longrightarrow SEQ\ l \in@\ L.\ P\ l \sqsubseteq_{FD} SEQ\ l \in@\ L.\ Q\ l$›
and *mono-MultiSeq-DT* :
‹$(\bigwedge x.\ x \in set\ L \Longrightarrow P\ x \sqsubseteq_{DT} Q\ x) \Longrightarrow SEQ\ l \in@\ L.\ P\ l \sqsubseteq_{DT} SEQ\ l \in@\ L.\ Q\ l$›
by (*induct L rule*: *rev-induct*, *simp-all add*: *monos-Seq*)

lemmas *monos-MultiSeq* = *mono-MultiSeq mono-MultiSeq-FD mono-MultiSeq-FD*

3.6.6 The Throw Operator

lemma *Throw-distrib-Ndet-right* :
‹$P \sqcap P'\ \Theta\ a \in A.\ Q\ a = (P\ \Theta\ a \in A.\ Q\ a) \sqcap (P'\ \Theta\ a \in A.\ Q\ a)$›
and *Throw-distrib-Ndet-left* :

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

lemma *Throw-distrib-GlobalNdet-right* :
‹$(\sqcap a \in A.\ P\ a)\ \Theta\ b \in B.\ Q\ b = \sqcap a \in A.\ (P\ a\ \Theta\ b \in B.\ Q\ b)$›
and *Throw-distrib-GlobalNdet-left* :
‹$P'\ \Theta\ a \in A.\ (\sqcap b \in B.\ Q'\ a\ b) =$
$(\mathit{if}\ B = \{\}\ \mathit{then}\ P'\ \Theta\ a \in A.\ \mathit{STOP}\ \mathit{else}\ \sqcap b \in B.\ (P'\ \Theta\ a \in A.\ Q'\ a\ b))$›

by (*simp add*: *Process-eq-spec Throw-projs GlobalNdet-projs*, *safe*, *simp-all*; *blast*)
(*simp add*: *Process-eq-spec Throw-projs GlobalNdet-projs STOP-projs*; *blast*)

3.6.7 The Interrupt Operator

lemma *Interrupt-distrib-GlobalNdet-left* :
‹$P \triangle (\sqcap a \in A.\ Q\ a) = (\mathit{if}\ A = \{\}\ \mathit{then}\ P\ \mathit{else}\ \sqcap a \in A.\ P \triangle Q\ a)$›
(**is** ‹*?lhs* = (*if - then - else ?rhs*)›)
proof (*split if-split*, *intro conjI impI*)
show ‹$A = \{\} \Longrightarrow$ *?lhs* = P› **by** *simp*
next
show ‹*?lhs* = *?rhs*› **if** ‹$A \neq \{\}$›
proof (*rule Process-eq-optimizedI*)
show ‹$t \in \mathcal{D}$ *?lhs* $\Longrightarrow t \in \mathcal{D}$ *?rhs*› **for** t
by (*auto simp add*: ‹$A \neq \{\}$› *D-Interrupt D-GlobalNdet*)
next
show ‹$t \in \mathcal{D}$ *?rhs* $\Longrightarrow t \in \mathcal{D}$ *?lhs*› **for** t
by (*auto simp add*: ‹$A \neq \{\}$› *D-Interrupt D-GlobalNdet*)
next
fix $t\ X$ **assume** ‹$(t, X) \in \mathcal{F}$ *?lhs*› ‹$t \notin \mathcal{D}$ *?lhs*›
with ‹$A \neq \{\}$› **consider** $r\ u$ **where** ‹$t = u\ @\ [\checkmark(r)]$› ‹$u\ @\ [\checkmark(r)] \in \mathcal{T}\ P$›
| r **where** ‹$\checkmark(r) \notin X$› ‹$t\ @\ [\checkmark(r)] \in \mathcal{T}\ P$›
| a **where** ‹$a \in A$› ‹$(t, X) \in \mathcal{F}\ P$› ‹$tF\ t$› ‹$([], X) \in \mathcal{F}\ (Q\ a)$›
| $a\ u\ v$ **where** ‹$a \in A$› ‹$t = u\ @\ v$› ‹$u \in \mathcal{T}\ P$› ‹$tF\ u$› ‹$(v, X) \in \mathcal{F}\ (Q\ a)$› ‹$v \neq []$›
| $a\ r$ **where** ‹$a \in A$› ‹$\checkmark(r) \notin X$› ‹$t \in \mathcal{T}\ P$› ‹$tF\ t$› ‹$[\checkmark(r)] \in \mathcal{T}\ (Q\ a)$›
unfolding *Interrupt-projs GlobalNdet-projs* **by** *force*
thus ‹$(t, X) \in \mathcal{F}$ *?rhs*›
proof *cases*
from ‹$A \neq \{\}$› **show** ‹$t = u\ @\ [\checkmark(r)] \Longrightarrow u\ @\ [\checkmark(r)] \in \mathcal{T}\ P \Longrightarrow (t, X) \in \mathcal{F}$ *?rhs*› **for** $r\ u$
by (*auto simp add*: *F-GlobalNdet F-Interrupt*)
next
show ‹$\checkmark(r) \notin X \Longrightarrow t\ @\ [\checkmark(r)] \in \mathcal{T}\ P \Longrightarrow (t, X) \in \mathcal{F}$ *?rhs*› **for** r
by (*simp add*: *F-GlobalNdet F-Interrupt*)
(*metis Diff-insert-absorb all-not-in-conv* ‹$A \neq \{\}$›)
next
show ‹$a \in A \Longrightarrow (t, X) \in \mathcal{F}\ P \Longrightarrow tF\ t \Longrightarrow ([], X) \in \mathcal{F}\ (Q\ a) \Longrightarrow (t, X)$

$\in \mathcal{F}$ *?rhs*› **for** *a*
by (*auto simp add*: *F-GlobalNdet F-Interrupt*)
next
show ‹⟦$a \in A$; $t = u$ @ v; $u \in \mathcal{T}$ P; *tF u*; $(v, X) \in \mathcal{F}$ $(Q\ a)$; $v \neq []$⟧
$\Longrightarrow$ $(t, X) \in \mathcal{F}$ *?rhs*› **for** *a u v* **by** (*auto simp add*: *F-GlobalNdet F-Interrupt*)
next
show ‹⟦$a \in A$; ✓$(r) \notin X$; $t \in \mathcal{T}$ P; *tF t*; [✓(r)] $\in \mathcal{T}$ $(Q\ a)$⟧ $\Longrightarrow$ $(t, X) \in \mathcal{F}$ *?rhs*› **for** *a r*
by (*simp add*: *F-GlobalNdet F-Interrupt*) (*metis Diff-insert-absorb* ‹$A \neq \{\}$›)
qed
next
fix *t X* **assume** ‹$(t, X) \in \mathcal{F}$ *?rhs*› ‹$t \notin \mathcal{D}$ *?rhs*›
from ‹$(t, X) \in \mathcal{F}$ *?rhs*› **obtain** *a* **where** ‹$a \in A$› ‹$(t, X) \in \mathcal{F}$ $(P \triangle Q\ a)$›
by (*auto simp add*: ‹$A \neq \{\}$› *F-GlobalNdet*)
with ‹$t \notin \mathcal{D}$ *?rhs*› **consider** *u r* **where** ‹$t = u$ @ [✓(r)]› ‹u @ [✓(r)] $\in \mathcal{T}$ P›
| *r* **where** ‹✓$(r) \notin X$› ‹t @ [✓(r)] $\in \mathcal{T}$ P›
| ‹$(t, X) \in \mathcal{F}$ P› ‹*tF t*› ‹$([], X) \in \mathcal{F}$ $(Q\ a)$›
| *u v* **where** ‹$t = u$ @ v› ‹$u \in \mathcal{T}$ P› ‹*tF u*› ‹$(v, X) \in \mathcal{F}$ $(Q\ a)$› ‹$v \neq []$›
| *r* **where** ‹✓$(r) \notin X$› ‹$t \in \mathcal{T}$ P› ‹*tF t*› ‹[✓(r)] $\in \mathcal{T}$ $(Q\ a)$›
unfolding *D-GlobalNdet Interrupt-projs* **by** *blast*
thus ‹$(t, X) \in \mathcal{F}$ *?lhs*›
proof *cases*
show ‹$t = u$ @ [✓(r)] $\Longrightarrow$ u @ [✓(r)] $\in \mathcal{T}$ P $\Longrightarrow$ $(t, X) \in \mathcal{F}$ *?lhs*› **for** *u r*
by (*simp add*: *F-Interrupt*)
next
show ‹✓$(r) \notin X$ $\Longrightarrow$ t @ [✓(r)] $\in \mathcal{T}$ P $\Longrightarrow$ $(t, X) \in \mathcal{F}$ *?lhs*› **for** *r*
by (*auto simp add*: *F-Interrupt*)
next
from ‹$a \in A$› **show** ‹⟦$(t, X) \in \mathcal{F}$ P; *tF t*; $([], X) \in \mathcal{F}$ $(Q\ a)$⟧ $\Longrightarrow$ $(t, X) \in \mathcal{F}$ *?lhs*›
by (*auto simp add*: *F-Interrupt F-GlobalNdet*)
next
from ‹$a \in A$› **show** ‹⟦$t = u$ @ v; $u \in \mathcal{T}$ P; *tF u*; $(v, X) \in \mathcal{F}$ $(Q\ a)$; $v \neq []$⟧ $\Longrightarrow$ $(t, X) \in \mathcal{F}$ *?lhs*› **for** *u v*
by (*simp add*: ‹$A \neq \{\}$› *F-Interrupt F-GlobalNdet*) *blast*
next
from ‹$a \in A$› **show** ‹⟦✓$(r) \notin X$; $t \in \mathcal{T}$ P; *tF t*; [✓(r)] $\in \mathcal{T}$ $(Q\ a)$⟧ $\Longrightarrow$ $(t, X) \in \mathcal{F}$ *?lhs*› **for** *r*
by (*simp add*: *F-Interrupt GlobalNdet-projs*) *blast*
qed
qed
qed

lemma *Interrupt-distrib-GlobalNdet-right* :
‹$(\sqcap a \in A.\ P\ a) \triangle Q$ = (*if* $A = \{\}$ *then* Q *else* $\sqcap a \in A.\ P\ a \triangle Q$)›
(**is** ‹*?lhs* = (*if* - *then* - *else* *?rhs*)›)
proof (*split if-split*, *intro conjI impI*)

```
  show ‹A = {} ⟹ ?lhs = Q› by simp
next
  show ‹?lhs = ?rhs› if ‹A ≠ {}›
  proof (rule Process-eq-optimizedI)
    show ‹t ∈ 𝒟 ?lhs ⟹ t ∈ 𝒟 ?rhs› for t
      by (simp add: GlobalNdet-projs D-Interrupt)
        (metis ex-in-conv is-processT1-TR ‹A ≠ {}›)
  next
    show ‹t ∈ 𝒟 ?rhs ⟹ t ∈ 𝒟 ?lhs› for t
      by (auto simp add: GlobalNdet-projs D-Interrupt)
  next
    fix t X assume ‹(t, X) ∈ ℱ ?lhs› ‹t ∉ 𝒟 ?lhs›
    then consider u r where ‹t = u @ [✓(r)]› ‹u @ [✓(r)] ∈ 𝒯 (⊓a ∈ A. P a)›
      | r where ‹✓(r) ∉ X› ‹t @ [✓(r)] ∈ 𝒯 (⊓a ∈ A. P a)›
      | ‹(t, X) ∈ ℱ (⊓a ∈ A. P a)› ‹tF t› ‹([], X) ∈ ℱ Q›
      | u v where ‹t = u @ v› ‹u ∈ 𝒯 (⊓a ∈ A. P a)› ‹tF u› ‹(v, X) ∈ ℱ Q› ‹v
≠ []›
      | r where ‹✓(r) ∉ X› ‹t ∈ 𝒯 (⊓a ∈ A. P a)› ‹tF t› ‹[✓(r)] ∈ 𝒯 Q›
      unfolding Interrupt-projs by blast
    thus ‹(t, X) ∈ ℱ ?rhs›
    proof cases
      show ‹t = u @ [✓(r)] ⟹ u @ [✓(r)] ∈ 𝒯 (⊓a ∈ A. P a) ⟹ (t, X) ∈ ℱ
?rhs› for u r
        by (auto simp add: ‹A ≠ {}› GlobalNdet-projs F-Interrupt)
    next
      show ‹✓(r) ∉ X ⟹ t @ [✓(r)] ∈ 𝒯 (⊓a ∈ A. P a) ⟹ (t, X) ∈ ℱ ?rhs›
for r
      by (simp add: ‹A ≠ {}› GlobalNdet-projs F-Interrupt) (metis Diff-insert-absorb)
    next
      show ‹(t, X) ∈ ℱ (⊓a ∈ A. P a) ⟹ tF t ⟹ ([], X) ∈ ℱ Q ⟹ (t, X) ∈
ℱ ?rhs›
        by (auto simp add: ‹A ≠ {}› F-GlobalNdet F-Interrupt)
    next
      show ‹⟦t = u @ v; u ∈ 𝒯 (⊓a ∈ A. P a); tF u; (v, X) ∈ ℱ Q; v ≠ []⟧
          ⟹ (t, X) ∈ ℱ ?rhs› for u v
        by (simp add: ‹A ≠ {}› GlobalNdet-projs F-Interrupt)
          (metis ex-in-conv is-processT1-TR ‹A ≠ {}›)
    next
      show ‹✓(r) ∉ X ⟹ t ∈ 𝒯 (⊓a ∈ A. P a) ⟹ tF t ⟹ [✓(r)] ∈ 𝒯 Q ⟹
(t, X) ∈ ℱ ?rhs› for r
        by (simp add: ‹A ≠ {}› GlobalNdet-projs F-Interrupt)
          (metis Diff-insert-absorb equals0I is-processT1-TR ‹A ≠ {}›)
    qed
  next
    fix t X assume ‹(t, X) ∈ ℱ ?rhs› ‹t ∉ 𝒟 ?rhs›
    from ‹(t, X) ∈ ℱ ?rhs› obtain a where ‹a ∈ A› ‹(t, X) ∈ ℱ (P a △ Q)›
      by (auto simp add: ‹A ≠ {}› F-GlobalNdet)
    with ‹t ∉ 𝒟 ?rhs› consider u r where ‹t = u @ [✓(r)]› ‹u @ [✓(r)] ∈ 𝒯 (P
a)›
```

| r **where** ‹✓(r) ∉ X› ‹t @ [✓(r)] ∈ $\mathcal{T}$ (P a)›
| ‹(t, X) ∈ $\mathcal{F}$ (P a)› ‹tF t› ‹([], X) ∈ $\mathcal{F}$ Q›
| u v **where** ‹t = u @ v› ‹u ∈ $\mathcal{T}$ (P a)› ‹tF u› ‹(v, X) ∈ $\mathcal{F}$ Q› ‹v ≠ []›
| r **where** ‹✓(r) ∉ X› ‹t ∈ $\mathcal{T}$ (P a)› ‹tF t› ‹[✓(r)] ∈ $\mathcal{T}$ Q›
unfolding *D-GlobalNdet Interrupt-projs* **by** *blast*
thus ‹(t, X) ∈ $\mathcal{F}$ *?lhs*›
proof *cases*
from ‹a ∈ A› **show** ‹t = u @ [✓(r)] ⟹ u @ [✓(r)] ∈ $\mathcal{T}$ (P a) ⟹ (t, X) ∈ $\mathcal{F}$ *?lhs*› **for** u r
by (*auto simp add*: *F-Interrupt T-GlobalNdet*)
next
from ‹a ∈ A› **show** ‹✓(r) ∉ X ⟹ t @ [✓(r)] ∈ $\mathcal{T}$ (P a) ⟹ (t, X) ∈ $\mathcal{F}$ *?lhs*› **for** r
by (*auto simp add*: *F-Interrupt GlobalNdet-projs*)
next
from ‹a ∈ A› **show** ‹(t, X) ∈ $\mathcal{F}$ (P a) ⟹ tF t ⟹ ([], X) ∈ $\mathcal{F}$ Q ⟹ (t, X) ∈ $\mathcal{F}$ *?lhs*›
by (*auto simp add*: *F-Interrupt F-GlobalNdet*)
next
from ‹a ∈ A› **show** ‹⟦t = u @ v; u ∈ $\mathcal{T}$ (P a); tF u; (v, X) ∈ $\mathcal{F}$ Q; v ≠ []⟧ ⟹ (t, X) ∈ $\mathcal{F}$ *?lhs*› **for** u v
by (*simp add*: *F-Interrupt GlobalNdet-projs*) *blast*
next
from ‹a ∈ A› **show** ‹⟦✓(r) ∉ X; t ∈ $\mathcal{T}$ (P a); tF t; [✓(r)] ∈ $\mathcal{T}$ Q⟧ ⟹ (t, X) ∈ $\mathcal{F}$ *?lhs*› **for** r
by (*simp add*: *F-Interrupt GlobalNdet-projs*) *blast*
qed
qed
qed

corollary *Interrupt-distrib-Ndet-left* : ‹P △ $Q1$ ⊓ $Q2$ = (P △ $Q1$) ⊓ (P △ $Q2$)›
proof –
have ‹P △ $Q1$ ⊓ $Q2$ = P △ (⊓n ∈ {*0::nat*, *1*}. (*if* n = *0 then Q1 else Q2*))›
by (*simp add*: *GlobalNdet-distrib-unit*)
also have ‹... = (⊓n ∈ {*0::nat*, *1*}. P △ (*if* n = *0 then Q1 else Q2*))›
by (*simp add*: *Interrupt-distrib-GlobalNdet-left*)
also have ‹... = (P △ $Q1$) ⊓ (P △ $Q2$)›
by (*simp add*: *GlobalNdet-distrib-unit*)
finally show *?thesis* **.**
qed

corollary *Interrupt-distrib-Ndet-right* : ‹$P1$ ⊓ $P2$ △ Q = ($P1$ △ Q) ⊓ ($P2$ △ Q)›
proof –
have ‹$P1$ ⊓ $P2$ △ Q = (⊓n ∈ {*0::nat*, *1*}. (*if* n = *0 then P1 else P2*)) △ Q›
by (*simp add*: *GlobalNdet-distrib-unit*)
also have ‹... = (⊓n ∈ {*0::nat*, *1*}. (*if* n = *0 then P1 else P2*) △ Q)›

by (*simp add*: *Interrupt-distrib-GlobalNdet-right*)
also have ‹. . . = (*P1* △ *Q*) ⊓ (*P2* △ *Q*)›
by (*simp add*: *GlobalNdet-distrib-unit*)
finally show *?thesis* **.**
qed

3.6.8 Global Deterministic Choice

lemma *GlobalDet-distrib-Ndet-left* :
‹(□*a* ∈ *A*. *P a* ⊓ *Q*) = (*if A* = {} *then STOP else* (□*a* ∈ *A*. *P a*) ⊓ *Q*)›
by (*auto simp add*: *Process-eq-spec Ndet-projs GlobalDet-projs F-STOP D-STOP*
intro: *is-processT8 is-processT6-TR-notin*)

lemma *GlobalDet-distrib-Ndet-right* :
‹(□*a* ∈ *A*. *P* ⊓ *Q a*) = (*if A* = {} *then STOP else P* ⊓ (□*a* ∈ *A*. *Q a*))›
by (*subst* (*1 2*) *Ndet-commute*) (*fact GlobalDet-distrib-Ndet-left*)

lemma *Ndet-distrib-GlobalDet-left* :
‹*P* ⊓ (□*a* ∈ *A*. *Q a*) = (*if A* = {} *then P* ⊓ *STOP else* □*a* ∈ *A*. *P* ⊓ *Q a*)›
by (*simp add*: *GlobalDet-distrib-Ndet-right*)

lemma *Ndet-distrib-GlobalDet-right* :
‹(□*a* ∈ *A*. *P a*) ⊓ *Q* = (*if A* = {} *then Q* ⊓ *STOP else* □*a* ∈ *A*. *P a* ⊓ *Q*)›
by (*metis* (*no-types*) *GlobalDet-distrib-Ndet-left GlobalDet-empty Ndet-commute*)

3.7 The Step-Laws

The step-laws describe the behaviour of the operators wrt. the multi-prefix choice.

3.7.1 The Throw Operator

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* **where** ‹*s* = *t1* @ *t2*› ‹*t1* ∈ 𝒟 (□*a* ∈ *A* → *P a*)› ‹*tF t1*›
‹*set t1* ∩ *ev* ‘ *B* = {}› ‹*ftF t2*›
| *t1 b t2* **where** ‹*s* = *t1* @ *ev b* # *t2*› ‹*t1* @ [*ev b*] ∈ 𝒯 (□*a*∈*A* → *P a*)›
‹*set t1* ∩ *ev* ‘ *B* = {}› ‹*b* ∈ *B*› ‹*t2* ∈ 𝒟 (*Q b*)›
by (*simp add*: *D-Throw*) *blast*
thus ‹*s* ∈ 𝒟 *?rhs*›
proof *cases*

fix *t1 t2* **assume** ∗ : ‹$s = t1 @ t2$› ‹$t1 \in \mathcal{D}$ (□$a{\in}A \rightarrow P\ a$)› ‹*tF t1*›
‹*set t1* ∩ *ev* ‘ $B = \{\}$› ‹*ftF t2*›
from ∗(*2*) **obtain** *a t1′* **where** ∗∗ : ‹$t1 = ev\ a \# t1'$› ‹$a \in A$› ‹$t1' \in \mathcal{D}$ (*P a*)›
by (*auto simp add*: *D-Mprefix*)
from ∗(*4*) ∗∗(*1*) **have** ∗∗∗ : ‹$a \notin B$› **by** (*simp add*: *image-iff*)
have ‹$t1' @ t2 \in \mathcal{D}$ (*Throw* (*P a*) *B Q*)›
using ∗(*3*, *4*, *5*) ∗∗(*1*, *3*) **by** (*auto simp add*: *D-Throw*)
with ∗∗∗ **show** ‹$s \in \mathcal{D}$ *?rhs*›
by (*simp add*: *D-Mprefix* ∗(*1*) ∗∗(*1*, *2*))
next
fix *t1 b t2* **assume** ∗ : ‹$s = t1 @ ev\ b \# t2$› ‹$t1 @ [ev\ b] \in \mathcal{T}$ (□$a{\in}A \rightarrow P\ a$)›
‹*set t1* ∩ *ev* ‘ $B = \{\}$› ‹$b \in B$› ‹$t2 \in \mathcal{D}$ (*Q b*)›
show ‹$s \in \mathcal{D}$ *?rhs*›
proof (*cases* ‹*t1*›)
from ∗(*2*) **show** ‹$t1 = [] \Longrightarrow s \in \mathcal{D}$ *?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-Cons append-Nil append-T-imp-tickFree*
$event_{ptick}$*.exhaust non-tickFree-tick not-Cons-self tickFree-append-iff*)
with ∗(*2*, *3*, *4*, *5*) **show** ‹$s \in \mathcal{D}$ *?rhs*›
by (*auto simp add*: ∗(*1*) *D-Mprefix T-Mprefix D-Throw*)
qed
qed
next
fix *s*
assume ‹$s \in \mathcal{D}$ *?rhs*›
then obtain *a s′* **where** ∗ : ‹$a \in A$› ‹$s = ev\ a \# s'$›
‹$s' \in \mathcal{D}$ (*if* $a \in B$ *then Q a else Throw* (*P a*) *B Q*)›
by (*auto simp add*: *D-Mprefix*)
show ‹$s \in \mathcal{D}$ *?lhs*›
proof (*cases* ‹$a \in B$›)
assume ‹$a \in B$›
hence ∗∗ : ‹$[] @ [ev\ a] \in \mathcal{T}$ (□$a{\in}A \rightarrow P\ a$) ∧ *set* [] ∩ *ev* ‘ $B = \{\}$ ∧ $s' \in \mathcal{D}$
(*Q a*)›
using ∗(*3*) **by** (*simp add*: *T-Mprefix* ∗(*1*))
show ‹$s \in \mathcal{D}$ *?lhs*›
by (*simp add*: *D-Throw*) (*metis* ∗(*2*) ∗∗ ‹$a \in B$› *append-Nil*)
next
assume ‹$a \notin B$›
with ∗(*2*, *3*)
consider *t1 t2* **where** ‹$s = ev\ a \# t1 @ t2$› ‹$t1 \in \mathcal{D}$ (*P a*)› ‹*tF t1*›
‹*set t1* ∩ *ev* ‘ $B = \{\}$› ‹*ftF t2*›
| *t1 b t2* **where** ‹$s = ev\ a \# t1 @ ev\ b \# t2$› ‹$t1 @ [ev\ b] \in \mathcal{T}$ (*P a*)›
‹*set t1* ∩ *ev* ‘ $B = \{\}$› ‹$b \in B$› ‹$t2 \in \mathcal{D}$ (*Q b*)›
by (*simp add*: *D-Throw*) *blast*
thus ‹$s \in \mathcal{D}$ *?lhs*›

```
    proof cases
      fix t1 t2 assume ** : ‹s = ev a # t1 @ t2› ‹t1 ∈ 𝒟 (P a)› ‹tF t1›
        ‹set t1 ∩ ev ‘ B = {}› ‹ftF t2›
      have *** : ‹ev a # t1 ∈ 𝒟 (□a∈A → P a) ∧ 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
      fix t1 b t2
      assume ** :  ‹s = ev a # t1 @ ev b # t2› ‹t1 @ [ev b] ∈ 𝒯 (P a)›
        ‹set t1 ∩ ev ‘ B = {}› ‹b ∈ B› ‹t2 ∈ 𝒟 (Q b)›
     have *** : ‹(ev a # t1) @ [ev b] ∈ 𝒯 (□a∈A → P a) ∧ 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 (□a∈A → P a)› ‹set s ∩ ev ‘ B = {}›
    | ‹s ∈ 𝒟 ?lhs›
    | t1 b t2 where ‹s = t1 @ ev b # t2› ‹t1 @ [ev b] ∈ 𝒯 (□a∈A → 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 ?rhs›
  proof cases
     show ‹(s, X) ∈ ℱ (□a∈A → P a) ⟹ set s ∩ ev ‘ B = {} ⟹ (s, X) ∈ ℱ
?rhs›
      by (simp add: F-Mprefix F-Throw)
        (metis image-eqI insert-disjoint(1) list.simps(15))
  next
    show ‹s ∈ 𝒟 ?lhs ⟹ (s, X) ∈ ℱ ?rhs›
      using same-div D-F by blast
  next
    fix t1 b t2 assume * : ‹s = t1 @ ev b # t2› ‹t1 @ [ev b] ∈ 𝒯 (□a∈A → P a)›
      ‹set t1 ∩ ev ‘ B = {}› ‹b ∈ B› ‹(t2, X) ∈ ℱ (Q b)›
    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-Cons append-Nil append-T-imp-tickFree
```

```
      eventptick.exhaust non-tickFree-tick not-Cons-self tickFree-append-iff)
    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 (□a∈A → P a) ∧
           set [] ∩ ev ‘ B = {} ∧ (s', X) ∈ ℱ (Q a')›
      using ∗(3) by (simp add: T-Mprefix ∗(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  where ‹s' = t1 @ t2› ‹t1 ∈ 𝒟 (P a')› ‹tF t1›
        ‹set t1 ∩ ev ‘ B = {}› ‹ftF t2›
      | t1 b t2 where ‹s' = t1 @ ev b # t2› ‹t1 @ [ev b] ∈ 𝒯 (P a')›
        ‹set t1 ∩ ev ‘ B = {}› ‹b ∈ B› ‹(t2, 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
      fix t1 t2 assume ∗∗ : ‹s' = t1 @ t2› ‹t1 ∈ 𝒟 (P a')› ‹tF t1›
        ‹set t1 ∩ ev ‘ B = {}› ‹ftF t2›
      have ∗∗∗ : ‹s = (ev a' # t1) @ t2 ∧ ev a' # t1 ∈ 𝒟 (□a∈A → P a) ∧
             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
      fix t1 b t2
      assume ∗∗ : ‹s' = t1 @ ev b # t2› ‹t1 @ [ev b] ∈ 𝒯 (P a')›
        ‹set t1 ∩ ev ‘ B = {}› ‹b ∈ B› ‹(t2, X) ∈ ℱ (Q b)›
```

have $*{*}*$: ‹$s = (ev\ a' \# t1) @ ev\ b \# t2 \wedge set\ (ev\ a' \# t1) \cap ev\ `\ B = \{\}$
$\wedge$
$(ev\ a' \# t1) @ [ev\ b] \in \mathcal{T}\ (\Box a{\in}A \to P\ a)$›
by (*simp add*: *T-Mprefix* ‹$a' \notin B$› *image-iff* $*(1, 2)$ $**(1, 2, 3)$)
show ‹$(s, X) \in \mathcal{F}$ *?lhs*›
by (*simp add*: *F-Throw F-Mprefix*) (*metis* $**(4, 5)$ $*{*}*$)
qed
qed
qed
qed

3.7.2 The Interrupt Operator

lemma *Interrupt-Mprefix*:
‹$(\Box a \in A \to P\ a) \triangle Q = Q \Box (\Box a \in A \to P\ a \triangle Q)$› (**is** ‹*?lhs* = *?rhs*›)
proof (*subst Process-eq-spec-optimized*, *safe*)
fix *s*
assume ‹$s \in \mathcal{D}$ *?lhs*›
then consider ‹$s \in \mathcal{D}\ (\Box a \in A \to P\ a)$›
| ‹$\exists\, t1\ t2.\ s = t1 @ t2 \wedge t1 \in \mathcal{T}\ (\Box a \in A \to P\ a) \wedge tF\ t1 \wedge t2 \in \mathcal{D}\ Q$›
by (*simp add*: *D-Interrupt*) *blast*
thus ‹$s \in \mathcal{D}$ *?rhs*›
proof *cases*
show ‹$s \in \mathcal{D}\ (\Box a \in A \to P\ a) \Longrightarrow s \in \mathcal{D}$ *?rhs*›
by (*auto simp add*: *D-Det D-Mprefix D-Interrupt*)
next
assume ‹$\exists\, t1\ t2.\ s = t1 @ t2 \wedge t1 \in \mathcal{T}\ (\Box a \in A \to P\ a) \wedge tF\ t1 \wedge t2 \in \mathcal{D}\ Q$›
then obtain *t1 t2*
where ‹$s = t1 @ t2$› ‹$t1 \in \mathcal{T}\ (\Box a \in A \to P\ a)$› ‹$tF\ t1$› ‹$t2 \in \mathcal{D}\ Q$› **by** *blast*
thus ‹$s \in \mathcal{D}$ *?rhs*› **by** (*fastforce simp add*: *D-Det Mprefix-projs D-Interrupt*)
qed
next
fix *s*
assume ‹$s \in \mathcal{D}$ *?rhs*›
then consider ‹$s \in \mathcal{D}\ Q$› | *a s'* **where** ‹$s = ev\ a \# s'$› ‹$a \in A$› ‹$s' \in \mathcal{D}\ (P\ a \triangle$
$Q)$›
by (*auto simp add*: *D-Det D-Mprefix image-iff*)
thus ‹$s \in \mathcal{D}$ *?lhs*›
proof *cases*
show ‹$s \in \mathcal{D}\ Q \Longrightarrow s \in \mathcal{D}$ *?lhs*›
by (*simp add*: *D-Interrupt*) (*use Nil-elem-T tickFree-Nil* **in** *blast*)
next
fix *a s'* **assume** ‹$s = ev\ a \# s'$› ‹$a \in A$› ‹$s' \in \mathcal{D}\ (P\ a \triangle Q)$›
from *this*(3) **consider** ‹$s' \in \mathcal{D}\ (P\ a)$›
| *t1 t2* **where** ‹$s' = t1 @ t2$› ‹$t1 \in \mathcal{T}\ (P\ a)$› ‹$tF\ t1$› ‹$t2 \in \mathcal{D}\ Q$›
by (*auto simp add*: *D-Interrupt*)
thus ‹$s \in \mathcal{D}$ *?lhs*›
proof *cases*
show ‹$s' \in \mathcal{D}\ (P\ a) \Longrightarrow s \in \mathcal{D}$ *?lhs*›

by (*simp add*: *D-Interrupt D-Mprefix* ‹$a \in A$› ‹$s = ev\ a \# s'$›)
next
show ‹$⟦s' = t1 @ t2; t1 \in \mathcal{T}\ (P\ a); tF\ t1; t2 \in \mathcal{D}\ Q⟧ \Longrightarrow s \in \mathcal{D}\ ?lhs$› **for** *t1 t2*
by (*simp add*: ‹$s = ev\ a \# s'$› *D-Interrupt T-Mprefix*)
(*metis Cons-eq-appendI* ‹$a \in A$› $event_{ptick}$*.disc*(*1*) *tickFree-Cons-iff*)
qed
qed
next
fix *s X*
assume *same-div* : ‹$\mathcal{D}\ ?lhs = \mathcal{D}\ ?rhs$›
assume ‹$(s, X) \in \mathcal{F}\ ?lhs$›
then consider ‹$s \in \mathcal{D}\ ?lhs$›
| *t1 r* **where** ‹$s = t1 @ [✓(r)]$› ‹$t1 @ [✓(r)] \in \mathcal{T}\ (Mprefix\ A\ P)$›
| *r* **where** ‹$s @ [✓(r)] \in \mathcal{T}\ (Mprefix\ A\ P)$› ‹$✓(r) \notin X$›
| ‹$(s, X) \in \mathcal{F}\ (Mprefix\ A\ P)$› ‹*tickFree s*› ‹$([], X) \in \mathcal{F}\ Q$›
| *t1 t2* **where** ‹$s = t1 @ t2$› ‹$t1 \in \mathcal{T}\ (Mprefix\ A\ P)$› ‹*tickFree t1*› ‹$(t2, X) \in \mathcal{F}\ Q$› ‹$t2 \neq []$›
| *r* **where** ‹$s \in \mathcal{T}\ (Mprefix\ A\ P)$› ‹*tickFree s*› ‹$[✓(r)] \in \mathcal{T}\ Q$› ‹$✓(r) \notin X$›
by (*simp add*: *F-Interrupt D-Interrupt*) *blast*
thus ‹$(s, X) \in \mathcal{F}\ ?rhs$›
proof *cases*
from *D-F same-div* **show** ‹$s \in \mathcal{D}\ ?lhs \Longrightarrow (s, X) \in \mathcal{F}\ ?rhs$› **by** *blast*
next
show ‹$s = t1 @ [✓(r)] \Longrightarrow t1 @ [✓(r)] \in \mathcal{T}\ (Mprefix\ A\ P) \Longrightarrow (s, X) \in \mathcal{F}\ ?rhs$› **for** *t1 r*
by (*simp add*: *F-Det T-Mprefix F-Mprefix F-Interrupt image-iff*)
(*metis append-Nil* $event_{ptick}$*.distinct*(*1*) *list.inject list.sel*(*3*) *tl-append2*)
next
show ‹$s @ [✓(r)] \in \mathcal{T}\ (Mprefix\ A\ P) \Longrightarrow ✓(r) \notin X \Longrightarrow (s, X) \in \mathcal{F}\ ?rhs$› **for** *r*
by (*simp add*: *F-Det T-Mprefix F-Mprefix F-Interrupt image-iff*)
(*metis* (*no-types*, *opaque-lifting*) *Diff-insert-absorb append-Nil*
$event_{ptick}$*.distinct*(*1*) *hd-append2 list.sel*(*1*, *3*) *neq-Nil-conv tl-append2*)
next
show ‹$(s, X) \in \mathcal{F}\ (Mprefix\ A\ P) \Longrightarrow tickFree\ s \Longrightarrow ([], X) \in \mathcal{F}\ Q \Longrightarrow (s, X) \in \mathcal{F}\ ?rhs$›
by (*simp add*: *F-Det F-Mprefix F-Interrupt image-iff*) (*metis tickFree-Cons-iff*)
next
show ‹$s = t1 @ t2 \Longrightarrow t1 \in \mathcal{T}\ (Mprefix\ A\ P) \Longrightarrow tickFree\ t1 \Longrightarrow (t2, X) \in \mathcal{F}\ Q \Longrightarrow$
$t2 \neq [] \Longrightarrow (s, X) \in \mathcal{F}\ ?rhs$› **for** *t1 t2*
by (*simp add*: *F-Det T-Mprefix F-Mprefix F-Interrupt image-iff*)
(*metis append-Cons append-Nil tickFree-Cons-iff*)
next
show ‹$s \in \mathcal{T}\ (Mprefix\ A\ P) \Longrightarrow tickFree\ s \Longrightarrow [✓(r)] \in \mathcal{T}\ Q \Longrightarrow$
$✓(r) \notin X \Longrightarrow (s, X) \in \mathcal{F}\ ?rhs$› **for** *r*
by (*simp add*: *F-Det T-Mprefix F-Mprefix F-Interrupt image-iff*)
(*metis Diff-insert-absorb tickFree-Cons-iff*)

qed
next
fix *s X*
assume *same-div* : ‹𝒟 *?lhs* = 𝒟 *?rhs*›
assume *assm* : ‹(*s*, *X*) ∈ ℱ *?rhs*›
show ‹(*s*, *X*) ∈ ℱ *?lhs*›
proof (*cases* ‹*s* = []›)
from *assm* **show** ‹*s* = [] ⟹ (*s*, *X*) ∈ ℱ *?lhs*›
by (*simp add*: *F-Det F-Mprefix F-Interrupt*) *blast*
next
assume ‹*s* ≠ []›
with *assm* **consider** ‹(*s*, *X*) ∈ ℱ *Q*›
| ‹∃ *a s*′. *s* = *ev a* # *s*′ ∧ *a* ∈ *A* ∧ (*s*′, *X*) ∈ ℱ (*P a* △ *Q*)›
by (*auto simp add*: *F-Det F-Mprefix image-iff*)
thus ‹(*s*, *X*) ∈ ℱ *?lhs*›
proof *cases*
show ‹(*s*, *X*) ∈ ℱ *Q* ⟹ (*s*, *X*) ∈ ℱ *?lhs*›
by (*simp add*: *F-Interrupt*)
(*metis Nil-elem-T* ‹*s* ≠ []› *append-Nil tickFree-Nil*)
next
assume ‹∃ *a s*′. *s* = *ev a* # *s*′ ∧ *a* ∈ *A* ∧ (*s*′, *X*) ∈ ℱ (*P a* △ *Q*)›
then obtain *a s*′
where ∗ : ‹*s* = *ev a* # *s*′› ‹*a* ∈ *A*› ‹(*s*′, *X*) ∈ ℱ (*P a* △ *Q*)› **by** *blast*
from ∗(*3*) **consider** ‹*s*′ ∈ 𝒟 (*P a* △ *Q*)›
| *t1 r* **where** ‹*s*′ = *t1* @ [✔(*r*)]› ‹*t1* @ [✔(*r*)] ∈ 𝒯 (*P a*)›
| *r* **where** ‹*s*′ @ [✔(*r*)] ∈ 𝒯 (*P a*)› ‹✔(*r*) ∉ *X*›
| ‹(*s*′, *X*) ∈ ℱ (*P a*)› ‹*tickFree s*′› ‹([], *X*) ∈ ℱ *Q*›
| *t1 t2* **where** ‹*s*′ = *t1* @ *t2*› ‹*t1* ∈ 𝒯 (*P a*)› ‹*tickFree t1*› ‹(*t2*, *X*) ∈ ℱ *Q*›
‹*t2* ≠ []›
| *r* **where** ‹*s*′ ∈ 𝒯 (*P a*)› ‹*tickFree s*′› ‹[✔(*r*)] ∈ 𝒯 *Q*› ‹✔(*r*) ∉ *X*›
by (*simp add*: *F-Interrupt D-Interrupt*) *blast*
thus ‹(*s*, *X*) ∈ ℱ *?lhs*›
proof *cases*
assume ‹*s*′ ∈ 𝒟 (*P a* △ *Q*)›
hence ‹*s* ∈ 𝒟 *?lhs*›
apply (*simp add*: *D-Interrupt D-Mprefix T-Mprefix* ∗(*1*, *2*) *image-iff*)
apply (*elim disjE exE conjE*; *simp*)
by (*metis* ∗(*2*) *Cons-eq-appendI* $event_{ptick}$.*disc*(*1*) *tickFree-Cons-iff*)
with *D-F same-div* **show** ‹(*s*, *X*) ∈ ℱ *?lhs*› **by** *blast*
next
show ‹*s*′ = *t1* @ [✔(*r*)] ⟹ *t1* @ [✔(*r*)] ∈ 𝒯 (*P a*) ⟹ (*s*, *X*) ∈ ℱ *?lhs*›
for *t1 r*
by (*simp add*: ∗(*1*, *2*) *F-Interrupt T-Mprefix*)
next
show ‹*s*′ @ [✔(*r*)] ∈ 𝒯 (*P a*) ⟹ ✔(*r*) ∉ *X* ⟹ (*s*, *X*) ∈ ℱ *?lhs*› **for** *r*
by (*simp add*: ∗(*1*, *2*) *F-Interrupt T-Mprefix*) *blast*
next
show ‹(*s*′, *X*) ∈ ℱ (*P a*) ⟹ *tickFree s*′ ⟹ ([], *X*) ∈ ℱ *Q* ⟹ (*s*, *X*) ∈
ℱ *?lhs*›

by (*simp add*: ∗(*1*, *2*) *F-Interrupt F-Mprefix image-iff*)
next
show ‹$s' = t1 @ t2 \Longrightarrow t1 \in \mathcal{T}\ (P\ a) \Longrightarrow tickFree\ t1 \Longrightarrow (t2, X) \in \mathcal{F}\ Q$
$\Longrightarrow$
$t2 \neq [] \Longrightarrow (s, X) \in \mathcal{F}\ ?lhs$› **for** *t1 t2*
apply (*simp add*: *F-Interrupt T-Mprefix* ∗(*1*))
by (*metis* (*no-types*, *lifting*) ∗(*1*, *2*) *Cons-eq-appendI F-imp-front-tickFree append-is-Nil-conv assm front-tickFree-Cons-iff tickFree-Cons-iff*)
next
show ‹$s' \in \mathcal{T}\ (P\ a) \Longrightarrow tickFree\ s' \Longrightarrow [\checkmark(r)] \in \mathcal{T}\ Q \Longrightarrow \checkmark(r) \notin X \Longrightarrow$
$(s, X) \in \mathcal{F}\ ?lhs$› **for** *r*
by (*simp add*: *F-Interrupt T-Mprefix* ∗(*1*, *2*) *image-iff*) *blast*
qed
qed
qed
qed

3.7.3 Global Deterministic Choice

lemma *GlobalDet-Mprefix* :
‹$(\Box a \in A.\ \Box b \in B\ a \to P\ a\ b) =$
$\Box b \in (\bigcup a \in A.\ B\ a) \to \sqcap a \in \{a \in A.\ b \in B\ a\}.\ P\ a\ b$› (**is** ‹?lhs = ?rhs›)
proof (*subst Process-eq-spec*, *safe*)
show ‹$s \in \mathcal{D}\ ?lhs \Longrightarrow s \in \mathcal{D}\ ?rhs$›
and ‹$s \in \mathcal{D}\ ?rhs \Longrightarrow s \in \mathcal{D}\ ?lhs$› **for** *s*
by (*auto simp add*: *D-Mprefix D-GlobalDet D-GlobalNdet*)
next
show ‹$(s, X) \in \mathcal{F}\ ?lhs \Longrightarrow (s, X) \in \mathcal{F}\ ?rhs$› **for** *s X*
by (*simp add*: *F-Mprefix F-GlobalDet F-GlobalNdet D-Mprefix*) *blast*
next
show ‹$(s, X) \in \mathcal{F}\ ?rhs \Longrightarrow (s, X) \in \mathcal{F}\ ?lhs$› **for** *s X*
by (*auto simp add*: *F-Mprefix F-GlobalDet F-GlobalNdet split*: *if-split-asm*)
qed

3.7.4 Multiple Synchronization Product

lemma *MultiSync-Mprefix-pseudo-distrib*:
‹$([\![S]\!]\ B \in\#\ M.\ \Box\ x \in B \to P\ B\ x) =$
$\Box\ x \in (\bigcap B \in set\text{-}mset\ M.\ B) \to ([\![S]\!]\ B \in\#\ M.\ P\ B\ x)$›
if *nonempty*: ‹$M \neq \{\#\}$› **and** *hyp*: ‹$\bigwedge B.\ B \in\#\ M \Longrightarrow B \subseteq S$›
proof−
from *nonempty* **obtain** *b M′* **where** ‹$b \in\#\ M - M'$›
‹ $M = add\text{-}mset\ b\ M'$› ‹$M' \subseteq\#\ M$›
by (*metis add-diff-cancel-left′ diff-subset-eq-self insert-DiffM insert-DiffM2 multi-member-last multiset-nonemptyE*)
show *?thesis*
apply (*subst* (*1 2 3*) ‹$M = add\text{-}mset\ b\ M'$›)
using ‹$b \in\#\ M - M'$› ‹$M' \subseteq\#\ M$›
proof (*induct rule*: *msubset-induct-singleton′*)
case *m-singleton* **show** *?case* **by** *fastforce*

next
case (*add x F*) **show** *?case*
apply (*simp*, *subst Mprefix-Sync-Mprefix-subset*[*symmetric*])
apply (*meson add.hyps*(*1*) *hyp in-diffD*,
metis ‹$b \in\# M - M'$› *hyp in-diffD le-infI1*)
using *add.hyps*(*3*) **by** *fastforce*
qed
qed

lemmas *MultiPar-Mprefix-pseudo-distrib* =
MultiSync-Mprefix-pseudo-distrib[**where** S = ‹*UNIV*›, *simplified*]

3.7.5 The Throw Operator

lemma *Throw-Mndetprefix*:
‹$(\sqcap a \in A \rightarrow P\ a)\ \Theta\ b \in B.\ Q\ b =$
$\sqcap a \in A \rightarrow (\mathit{if}\ a \in B\ \mathit{then}\ Q\ a\ \mathit{else}\ P\ a\ \Theta\ b \in B.\ Q\ b)$›
by (*auto simp add*: *Mndetprefix-GlobalNdet Throw-distrib-GlobalNdet-right*
write0-def Throw-Mprefix
intro: *mono-GlobalNdet-eq mono-Mprefix-eq*)

3.7.6 The Interrupt Operator

lemma *Interrupt-Mndetprefix*:
‹$(\sqcap a \in A \rightarrow P\ a) \triangle Q = Q \,\Box\, (\sqcap a \in A \rightarrow P\ a \triangle Q)$›
by (*simp add*: *Mndetprefix-GlobalNdet Interrupt-distrib-GlobalNdet-right*
write0-def Interrupt-Mprefix Det-distrib-GlobalNdet-left)

Chapter 4

CSPM Laws

4.0.1 The Throw Operator

lemma *Throw-read* :
‹*inj-on c A* $\Longrightarrow$ $(c?a \in A \to P\ a)\ \Theta\ a \in B.\ Q\ a =$
$c?a \in A \to (\textit{if } c\ a \in B \textit{ then } Q\ (c\ a) \textit{ else } P\ a\ \Theta\ a \in B.\ Q\ a)$›
by (*auto simp add*: *read-def Throw-Mprefix intro*: *mono-Mprefix-eq*)

lemma *Throw-ndet-write* :
‹*inj-on c A* $\Longrightarrow$ $(c!!a \in A \to P\ a)\ \Theta\ a \in B.\ Q\ a =$
$c!!a \in A \to (\textit{if } c\ a \in B \textit{ then } Q\ (c\ a) \textit{ else } P\ a\ \Theta\ a \in B.\ Q\ a)$›
by (*auto simp add*: *ndet-write-def Throw-Mndetprefix intro*: *mono-Mndetprefix-eq*)

lemma *Throw-write* :
‹$(c!a \to P)\ \Theta\ a \in B.\ Q\ a = c!a \to (\textit{if } c\ a \in B \textit{ then } Q\ (c\ a) \textit{ else } P\ \Theta\ a \in B.\ Q\ a)$›
by (*auto simp add*: *write-def Throw-Mprefix intro*: *mono-Mprefix-eq*)

lemma *Throw-write0* :
‹$(a \to P)\ \Theta\ a \in B.\ Q\ a = a \to (\textit{if } a \in B \textit{ then } Q\ a \textit{ else } P\ \Theta\ a \in B.\ Q\ a)$›
by (*auto simp add*: *write0-def Throw-Mprefix intro*: *mono-Mprefix-eq*)

4.0.2 The Interrupt Operator

lemma *Interrupt-read* :
‹$(c?a \in A \to P\ a) \triangle Q = Q \square (c?a \in A \to P\ a \triangle Q)$›
by (*auto simp add*: *read-def Interrupt-Mprefix*
intro!: *mono-Mprefix-eq arg-cong*[**where** f = ‹$\lambda P.\ Q \square P$›])

lemma *Interrupt-ndet-write* :
‹$(c!!a \in A \to P\ a) \triangle Q = Q \square (c!!a \in A \to P\ a \triangle Q)$›
by (*auto simp add*: *ndet-write-def Interrupt-Mndetprefix*
intro!: *mono-Mndetprefix-eq arg-cong*[**where** f = ‹$\lambda P.\ Q \square P$›])

lemma *Interrupt-write* : ‹$(c!a \to P) \triangle Q = Q \square (c!a \to P \triangle Q)$›
by (*auto simp add*: *write-def Interrupt-Mprefix intro*: *mono-Mprefix-eq*)

lemma *Interrupt-write0* : ‹$(a \to P) \triangle Q = Q \square (a \to P \triangle Q)$›
by (*auto simp add*: *write0-def Interrupt-Mprefix intro*: *mono-Mprefix-eq*)

4.0.3 Global Deterministic Choice

lemma *GlobalDet-read* :
‹$\square a \in A.\ c?b \in B\ a \to P\ a\ b = c?b \in (\bigcup a{\in}A.\ B\ a) \to \sqcap a{\in}\{a \in A.\ b \in B\ a\}.$ $P\ a\ b$›
if ‹*inj-on* $c\ (\bigcup a{\in}A.\ B\ a)$›
proof −
have ∗ : ‹$a \in A \Longrightarrow b \in B\ a \Longrightarrow$
$\{a \in A.\ \textit{inv-into}\ (\bigcup\ (B\ `\ A))\ c\ (c\ b) \in B\ a\} = \{a \in A.\ c\ b \in c\ `\ B\ a\}$›
for $a\ b$
by (*metis* (*no-types*, *opaque-lifting*) *SUP-upper UN-iff*
inj-on-image-mem-iff inv-into-f-eq ‹*inj-on* $c\ (\bigcup a{\in}A.\ B\ a)$›)
have ‹$\square a \in A.\ c?b \in B\ a \to P\ a\ b =$
$\square b{\in}(\bigcup x{\in}A.\ c\ `\ B\ x) \to \sqcap a{\in}\{a \in A.\ b \in c\ `\ B\ a\}.\ P\ a\ (\textit{inv-into}\ (B\ a)\ c\ b)$›
by (*simp add*: *read-def GlobalDet-Mprefix*)
also have ‹$(\bigcup x{\in}A.\ c\ `\ B\ x) = c\ `\ (\bigcup a{\in}A.\ B\ a)$› **by** *blast*
finally show ‹$\square a \in A.\ c?b \in B\ a \to P\ a\ b = c?b \in (\bigcup a{\in}A.\ B\ a) \to \sqcap a{\in}\{a \in$ $A.\ b \in B\ a\}.\ P\ a\ b$›
by (*auto simp add*: *read-def* ∗ *intro*!: *mono-Mprefix-eq mono-GlobalNdet-eq*)
(*metis* (*lifting*) *SUP-upper UN-I inv-into-f-eq subset-inj-on* ‹*inj-on* $c\ (\bigcup a{\in}A.$ $B\ a)$›)
qed

lemma *GlobalDet-write* :
‹$\square a \in A.\ c!(b\ a) \to P\ a = c?x \in b\ `\ A \to \sqcap a{\in}\{a \in A.\ x = b\ a\}.\ P\ a$› **if** ‹*inj-on* $c\ (b\ `\ A)$›
proof −
from ‹*inj-on* $c\ (b\ `\ A)$› **have** ∗ : ‹$x \in A \Longrightarrow \{a \in A.\ \textit{inv-into}\ (b\ `\ A)\ c\ (c\ (b\ x))$ $= b\ a\} =$
$\{a \in A.\ c\ (b\ x) = c\ (b\ a)\}$› **for** x
by (*auto simp add*: *inj-on-eq-iff*)
have ‹$\square a \in A.\ c!(b\ a) \to P\ a = \square x{\in}(\bigcup a{\in}A.\ \{c\ (b\ a)\}) \to \textit{GlobalNdet}\ \{a \in A.$ $x = c\ (b\ a)\}\ P$›
by (*simp add*: *write-def GlobalDet-Mprefix*)
also have ‹$(\bigcup a{\in}A.\ \{c\ (b\ a)\}) = c\ `\ b\ `\ A$› **by** *blast*
finally show ‹$\square a \in A.\ c!(b\ a) \to P\ a = c?x \in b\ `\ A \to \sqcap a{\in}\{a \in A.\ x = b\ a\}.$ $P\ a$›
by (*auto simp add*: *read-def* ∗ *intro*: *mono-Mprefix-eq*)
qed

lemma *GlobalDet-write0* :
‹$\square a{\in}A.\ b\ a \to P\ a = \square x \in (b\ `\ A) \to \sqcap a \in \{a \in A.\ x = b\ a\}.\ P\ a$›
by (*auto simp add*: *GlobalDet-write*[**where** c = ‹$\lambda x.\ x$›, *simplified write-is-write0*]

read-def
intro!: *mono-Mprefix-eq*) (*metis* (*lifting*) *f-inv-into-f image-eqI*)

4.0.4 Multiple Synchronization Product

4.1 Results for Throw

4.1.1 Laws for Throw

lemma *Throw-GlobalDet* :
‹(□ a ∈ A. P a) Θ b ∈ B. Q b = □ a ∈ A. P a Θ b ∈ B. Q b› (**is** ‹?lhs = ?rhs›)
proof (*rule Process-eq-optimizedI*)
show ‹t ∈ 𝒟 ?lhs ⟹ t ∈ 𝒟 ?rhs› **for** t
by (*simp add*: *D-Throw GlobalDet-projs split*: *if-split-asm*) *blast*
next
show ‹t ∈ 𝒟 ?rhs ⟹ t ∈ 𝒟 ?lhs› **for** t
by (*simp add*: *D-Throw GlobalDet-projs*) (*meson empty-iff*)
next
fix t X **assume** ‹(t, X) ∈ ℱ ?lhs› ‹t ∉ 𝒟 ?lhs›
then consider ‹(t, X) ∈ ℱ (□a ∈ A. P a)› ‹set t ∩ ev ‘ B = {}›
| (*failR*) t1 b t2 **where** ‹t = t1 @ ev b # t2› ‹t1 @ [ev b] ∈ 𝒯 (□a ∈ A. P a)›
‹set t1 ∩ ev ‘ B = {}› ‹b ∈ B› ‹(t2, X) ∈ ℱ (Q b)›
unfolding *Throw-projs* **by** *blast*
thus ‹(t, X) ∈ ℱ ?rhs›
proof *cases*
show ‹(t, X) ∈ ℱ (□a ∈ A. P a) ⟹ set t ∩ ev ‘ B = {} ⟹ (t, X) ∈ ℱ ?rhs›
by (*cases t*) (*auto simp add*: *F-GlobalDet Throw-projs*)
next
case *failR*
from *failR*(2) **obtain** a **where** ‹a ∈ A› ‹t1 @ [ev b] ∈ 𝒯 (P a)›
by (*auto simp add*: *T-GlobalDet split*: *if-split-asm*)
with *failR*(3−5) **show** ‹(t, X) ∈ ℱ ?rhs›
by (*simp add*: *F-GlobalDet F-Throw failR*(1)) *blast*
qed
next
fix t X **assume** ‹(t, X) ∈ ℱ ?rhs› ‹t ∉ 𝒟 ?rhs›
then consider ‹t = []› ‹∀ a∈A. (t, X) ∈ ℱ (P a Θ b ∈ B. Q b)›
| a **where** ‹a ∈ A› ‹t ≠ []› ‹(t, X) ∈ ℱ (P a Θ b ∈ B. Q b)›
| a r **where** ‹a ∈ A› ‹t = []› ‹✓(r) ∉ X› ‹[✓(r)] ∈ 𝒯 (P a Θ b ∈ B. Q b)›
by (*auto simp add*: *GlobalDet-projs*)
thus ‹(t, X) ∈ ℱ ?lhs›
proof *cases*
show ‹t = [] ⟹ ∀ a∈A. (t, X) ∈ ℱ (P a Θ b ∈ B. Q b) ⟹ (t, X) ∈ ℱ ?lhs›
by (*auto simp add*: *F-Throw F-GlobalDet*)
next
show ‹a ∈ A ⟹ t ≠ [] ⟹ (t, X) ∈ ℱ (P a Θ b ∈ B. Q b) ⟹ (t, X) ∈ ℱ ?lhs› **for** a
by (*simp add*: *F-Throw GlobalDet-projs*) (*metis empty-iff*)
next

show ‹⟦$a \in A$; $t = []$; ✔(r) $\notin X$; [✔(r)] $\in \mathcal{T}$ (P a Θ $b \in B$. Q b)⟧ $\Longrightarrow$ (t, X) $\in \mathcal{F}$ *?lhs*› **for** *a r*
by (*simp add*: *Throw-projs F-GlobalDet Cons-eq-append-conv*) (*metis is-processT9-tick*)
qed
qed

lemma *Throw-GlobalNdetR* :
‹P Θ $a \in A$. $\sqcap b \in B$. Q a b =
(*if* $B = \{\}$ *then* P Θ $a \in A$. *STOP else* $\Box b \in B$. P Θ $a \in A$. Q a b)›
(**is** ‹*?lhs* = (*if - then - else ?rhs*)›)
proof (*split if-split*, *intro conjI impI*)
show ‹$B = \{\}$ $\Longrightarrow$ *?lhs* = P Θ $a \in A$. *STOP*› **by** *simp*
next
show ‹*?lhs* = *?rhs*› **if** ‹$B \neq \{\}$›
proof (*subst Process-eq-spec*, *safe*)
show ‹$t \in \mathcal{D}$ *?lhs* $\Longrightarrow$ $t \in \mathcal{D}$ *?rhs*› **for** *t*
by (*auto simp add*: ‹$B \neq \{\}$› *D-Throw D-GlobalNdet D-GlobalDet*)
next
show ‹$t \in \mathcal{D}$ *?rhs* $\Longrightarrow$ $t \in \mathcal{D}$ *?lhs*› **for** *t*
by (*auto simp add*: ‹$B \neq \{\}$› *D-Throw D-GlobalNdet D-GlobalDet*)
next
show ‹(t, X) $\in \mathcal{F}$ *?lhs* $\Longrightarrow$ (t, X) $\in \mathcal{F}$ *?rhs*› **for** *t X*
by (*cases t*) (*auto simp add*: ‹$B \neq \{\}$› *F-Throw F-GlobalNdet F-GlobalDet*)
next
show ‹(t, X) $\in \mathcal{F}$ *?rhs* $\Longrightarrow$ (t, X) $\in \mathcal{F}$ *?lhs*› **for** *t X*
by (*auto simp add*: ‹$B \neq \{\}$› *Throw-projs F-GlobalNdet F-GlobalDet D-T is-processT7 Cons-eq-append-conv intro*!: *is-processT6-TR-notin*)
qed
qed

corollary *Throw-Det* : ‹$P \Box P'$ Θ $a \in A$. Q a = (P Θ $a \in A$. Q a) $\Box$ (P' Θ $a \in A$. Q a)›
proof –
have ‹$P \Box P'$ Θ $a \in A$. Q a = ($\Box a \in \{0 :: nat, 1\}$. (*if* $a = 0$ *then* P *else* P')) Θ $a \in A$. Q a›
by (*simp add*: *GlobalDet-distrib-unit*)
also have ‹... = $\Box a \in \{0 :: nat, 1\}$. (*if* $a = 0$ *then* P *else* P') Θ $a \in A$. Q a›
by (*fact Throw-GlobalDet*)
also have ‹... = (P Θ $a \in A$. Q a) $\Box$ (P' Θ $a \in A$. Q a)›
by (*simp add*: *GlobalDet-distrib-unit*)
finally show *?thesis* **.**
qed

corollary *Throw-NdetR* : ‹P Θ $a \in A$. Q $a \sqcap Q'$ a = (P Θ $a \in A$. Q a) $\Box$ (P Θ $a \in A$. Q' a)›
proof –
have ‹P Θ $a \in A$. Q $a \sqcap Q'$ a = P Θ $a \in A$. $\sqcap b \in \{0 :: nat, 1\}$. (*if* $b = 0$ *then*

Q a else Q′ a)›
  by (*simp add*: *GlobalNdet-distrib-unit*)
 also have ‹. . . = □*b* ∈ {*0* :: *nat*, *1*}. *P* Θ *a* ∈ *A*. (*if b* = *0 then Q a else Q′ a*)›
  by (*simp add*: *Throw-GlobalNdetR*)
 also have ‹. . . = (*P* Θ *a* ∈ *A*. *Q a*) □ (*P* Θ *a* ∈ *A*. *Q′ a*)›
  by (*simp add*: *GlobalDet-distrib-unit*)
 finally show *?thesis* **.**
qed

4.1.2 Laws for Sync

lemma *Sync-GlobalNdet-cartprod*:
 ‹(⊓ (*a*, *b*) ∈ *A* × *B*. (*P a* ⟦*S*⟧ *Q b*)) =
  (*if A* = {} ∨ *B* = {} *then STOP else* (⊓*a* ∈ *A*. *P a*) ⟦*S*⟧ (⊓*b* ∈ *B*. *Q b*))›
 by (*simp add*: *GlobalNdet-cartprod Sync-distrib-GlobalNdet-left*
   Sync-distrib-GlobalNdet-right GlobalNdet-sets-commute[*of A*])

lemmas *Inter-GlobalNdet-cartprod* = *Sync-GlobalNdet-cartprod*[**where** *S* = ‹{}›]
 and *Par-GlobalNdet-cartprod* = *Sync-GlobalNdet-cartprod*[**where** *S* = *UNIV*]

lemma *MultiSync-Hiding-pseudo-distrib*:
 ‹*finite A* ⟹ *A* ∩ *S* = {} ⟹ (⟦*S*⟧ *p* ∈# *M*. (*P p* \ *A*)) = (⟦*S*⟧ *p* ∈# *M*. *P p*)
\ *A*›
 by (*induct M*, *simp*) (*metis MultiSync-add MultiSync-rec1 Hiding-Sync*)

lemma *MultiSync-prefix-pseudo-distrib*:
 ‹*M* ≠ {#} ⟹ *a* ∈ *S* ⟹ (⟦*S*⟧ *p* ∈# *M*. (*a* → *P p*)) = (*a* → (⟦*S*⟧ *p* ∈# *M*. *P*
p))›
 by (*induct M rule*: *mset-induct-nonempty*)
  (*simp-all add*: *write0-Sync-write0-subset*)

lemmas *MultiInter-Hiding-pseudo-distrib* =
 MultiSync-Hiding-pseudo-distrib[**where** *S* = ‹{}›, *simplified*]
 and *MultiPar-prefix-pseudo-distrib* =
 MultiSync-prefix-pseudo-distrib[**where** *S* = ‹*UNIV*›, *simplified*]

A result on Mndetprefix and Sync.

lemma *Mndetprefix-Sync-distr*: ‹*A* ≠ {} ⟹ *B* ≠ {} ⟹
  (⊓ *a* ∈ *A* → *P a*) ⟦*S*⟧ (⊓ *b* ∈ *B* → *Q b*) =
  ⊓ *a*∈*A*. ⊓ *b*∈*B*. (□*c* ∈ ({*a*} − *S*) → (*P a* ⟦*S*⟧ (*b* → *Q b*))) □
      (□*d* ∈ ({*b*} − *S*) → ((*a* → *P a*) ⟦*S*⟧ *Q b*)) □
      (□*c*∈({*a*} ∩ {*b*} ∩ *S*) → (*P a* ⟦*S*⟧ *Q b*))›
 apply (*subst* (*1 2*) *Mndetprefix-GlobalNdet*)
 apply (*subst Sync-distrib-GlobalNdet-right*, *simp*)
 apply (*subst Sync-commute*)

apply (*subst Sync-distrib-GlobalNdet-right*, *simp*)
apply (*subst Sync-commute*)
apply (*unfold write0-def*)
apply (*subst Mprefix-Sync-Mprefix*)
by (*fold write0-def*, *rule refl*)

lemma ‹$A \neq \{\} \Longrightarrow B \neq \{\} \Longrightarrow (\sqcap\ a \in A \rightarrow P\ a)\ [\![S]\!]\ (\sqcap\ b \in B \rightarrow Q\ b) =$
$\sqcap\ a{\in}A.\ \sqcap\ b{\in}B.\ (\mathit{if}\ a \in S\ \mathit{then}\ STOP\ \mathit{else}\ (a \rightarrow (P\ a\ [\![S]\!]\ (b \rightarrow Q\ b)))) \square$
$(\mathit{if}\ b \in S\ \mathit{then}\ STOP\ \mathit{else}\ (b \rightarrow ((a \rightarrow P\ a)\ [\![S]\!]\ Q\ b))) \square$
$(\mathit{if}\ a = b \wedge a \in S\ \mathit{then}\ (a \rightarrow (P\ a\ [\![S]\!]\ Q\ a))\ \mathit{else}\ STOP)$›
apply (*subst Mndetprefix-Sync-distr*, *assumption+*)
apply (*intro mono-GlobalNdet-eq*)
apply (*intro arg-cong2*[**where** *f* = ‹(□)›])
by (*simp-all add*: *Mprefix-singl insert-Diff-if Int-insert-left*)

4.1.3 GlobalDet, GlobalNdet and write0

lemma *GlobalDet-write0-is-GlobalNdet-write0*:
‹$(\square\ p \in A.\ (a \rightarrow P\ p)) = \sqcap\ p \in A.\ (a \rightarrow P\ p)$› (**is** ‹*?lhs* = *?rhs*›)
proof (*subst Process-eq-spec*, *safe*)
show ‹$s \in \mathcal{D}$ *?lhs* $\Longrightarrow s \in \mathcal{D}$ *?rhs*›
and ‹$s \in \mathcal{D}$ *?rhs* $\Longrightarrow s \in \mathcal{D}$ *?lhs*› **for** *s*
by (*simp-all add*: *D-GlobalDet write0-def D-Mprefix D-GlobalNdet*)
next
show ‹$(s, X) \in \mathcal{F}$ *?lhs* $\Longrightarrow (s, X) \in \mathcal{F}$ *?rhs*›
and ‹$(s, X) \in \mathcal{F}$ *?rhs* $\Longrightarrow (s, X) \in \mathcal{F}$ *?lhs*› **for** *s X*
by (*auto simp add*: *F-GlobalDet write0-def F-Mprefix F-GlobalNdet split*: *if-split-asm*)
qed

lemma *write0-GlobalNdet-bis*:
‹$A \neq \{\} \Longrightarrow (a \rightarrow (\sqcap\ p \in A.\ P\ p) = \square\ p \in A.\ (a \rightarrow P\ p))$›
by (*simp add*: *GlobalDet-write0-is-GlobalNdet-write0 write0-GlobalNdet*)

4.2 Some Results on Renaming

lemma *Renaming-GlobalNdet*:
‹*Renaming* $(\sqcap\ a \in A.\ P\ (f\ a))\ f\ g = \sqcap\ b \in f\ `\ A.$ *Renaming* $(P\ b)\ f\ g$›
by (*metis Renaming-distrib-GlobalNdet mono-GlobalNdet-eq2*)

lemma *Renaming-GlobalNdet-inj-on*:
‹*Renaming* $(\sqcap\ a \in A.\ P\ a)\ f\ g =$
$\sqcap\ b \in f\ `\ A.$ *Renaming* $(P\ (THE\ a.\ a \in A \wedge f\ a = b))\ f\ g$›
if *inj-on-f*: ‹*inj-on f A*›
by (*smt* (*verit*, *ccfv-SIG*) *Renaming-distrib-GlobalNdet inj-on-def mono-GlobalNdet-eq2 that the-equality*)

corollary *Renaming-GlobalNdet-inj*:
‹*Renaming* $(\sqcap\ a \in A.\ P\ a)\ f\ g =$
$\sqcap\ b \in f\ `\ A.$ *Renaming* $(P\ (THE\ a.\ f\ a = b))\ f\ g$› **if** *inj-f*: ‹*inj f*›

apply (*subst Renaming-GlobalNdet-inj-on*, *metis inj-eq inj-onI inj-f*)
apply (*rule mono-GlobalNdet-eq*[*rule-format*])
by (*metis imageE inj-eq*[*OF inj-f*])

lemma *Renaming-distrib-GlobalDet* :
‹*Renaming* (□*a* ∈ *A*. *P a*) *f g* = □*a* ∈ *A*. *Renaming* (*P a*) *f g*› (**is** ‹*?lhs* = *?rhs*›)
proof (*subst Process-eq-spec-optimized*, *safe*)
show ‹*s* ∈ 𝒟 *?lhs* ⟹ *s* ∈ 𝒟 *?rhs*›
and ‹*s* ∈ 𝒟 *?rhs* ⟹ *s* ∈ 𝒟 *?lhs*› **for** *s*
by (*auto simp add*: *D-Renaming D-GlobalDet*)
next
assume *same-div* : ‹𝒟 *?lhs* = 𝒟 *?rhs*›
fix *s X* **assume** ‹(*s*, *X*) ∈ ℱ *?lhs*›
then consider ‹*s* ∈ 𝒟 *?lhs*›
| *t* **where** ‹*s* = *map* (*map-event*$_{ptick}$ *f g*) *t*› ‹(*t*, *map-event*$_{ptick}$ *f g* −‘ *X*) ∈ ℱ (□*a* ∈ *A*. *P a*)›
unfolding *Renaming-projs* **by** *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* = *map* (*map-event*$_{ptick}$ *f g*) *t* ⟹ (*t*, *map-event*$_{ptick}$ *f g* −‘ *X*) ∈ ℱ (□*a* ∈ *A*. *P a*)
⟹ (*s*, *X*) ∈ ℱ *?rhs*› **for** *t*
by (*cases t*; *simp add*: *F-GlobalDet Renaming-projs*)
(*force*, *metis list.simps*(*9*))
qed
next
assume *same-div* : ‹𝒟 *?lhs* = 𝒟 *?rhs*›
fix *s X* **assume** ‹(*s*, *X*) ∈ ℱ *?rhs*›
then consider ‹*s* = []› ‹∀ *a*∈*A*. (*s*, *X*) ∈ ℱ (*Renaming* (*P a*) *f g*)›
| *a* **where** ‹*a* ∈ *A*› ‹*s* ≠ []› ‹(*s*, *X*) ∈ ℱ (*Renaming* (*P a*) *f g*)›
| *a* **where** ‹*a* ∈ *A*› ‹*s* = []› ‹*s* ∈ 𝒟 (*Renaming* (*P a*) *f g*)›
| *a r* **where** ‹*a* ∈ *A*› ‹*s* = []› ‹✓(*r*) ∉ *X*› ‹[✓(*r*)] ∈ 𝒯 (*Renaming* (*P a*) *f g*)›
unfolding *F-GlobalDet* **by** *blast*
thus ‹(*s*, *X*) ∈ ℱ *?lhs*›
proof *cases*
show ‹*s* = [] ⟹ ∀ *a*∈*A*. (*s*, *X*) ∈ ℱ (*Renaming* (*P a*) *f g*) ⟹ (*s*, *X*) ∈ ℱ *?lhs*›
by (*auto simp add*: *F-Renaming F-GlobalDet*)
next
show ‹*a* ∈ *A* ⟹ *s* ≠ [] ⟹ (*s*, *X*) ∈ ℱ (*Renaming* (*P a*) *f g*) ⟹ (*s*, *X*) ∈ ℱ *?lhs*› **for** *a*
by (*simp add*: *F-Renaming GlobalDet-projs*) (*metis list.simps*(*8*))
next
show ‹*a* ∈ *A* ⟹ *s* = [] ⟹ *s* ∈ 𝒟 (*Renaming* (*P a*) *f g*) ⟹ (*s*, *X*) ∈ ℱ *?lhs*› **for** *a*
by (*auto simp add*: *Renaming-projs D-GlobalDet*)

next
fix *a r* **assume** ∗ : ‹$a \in A$› ‹$s = []$› ‹✔(r) ∉ X› ‹[✔(r)] ∈ $\mathcal{T}$ (*Renaming* (*P a*) *f g*)›
from ∗(*4*) **consider** *s1* **where** ‹[✔(r)] = *map* (*map-event$_{ptick}$ f g*) *s1*› ‹*s1* ∈ $\mathcal{T}$ (*P a*)›
| *s1 s2* **where** ‹[✔(r)] = *map* (*map-event$_{ptick}$ f g*) *s1* @ *s2*›
‹*tickFree s1*› ‹*front-tickFree s2*› ‹*s1* ∈ $\mathcal{D}$ (*P a*)›
by (*simp add*: *T-Renaming*) *meson*
thus ‹(*s*, *X*) ∈ $\mathcal{F}$ *?lhs*›
proof *cases*
fix *s1* **assume** ‹[✔(r)] = *map* (*map-event$_{ptick}$ f g*) *s1*› ‹*s1* ∈ $\mathcal{T}$ (*P a*)›
from ‹[✔(r)] = *map* (*map-event$_{ptick}$ f g*) *s1*› **obtain** r' **where** ‹$r = g\ r'$› ‹*s1* = [✔(r')]›
by (*metis map-map-event$_{ptick}$-eq-tick-iff*)
with ∗(*1*, *2*, *3*) ‹*s1* ∈ $\mathcal{T}$ (*P a*)›
show ‹(*s*, *X*) ∈ $\mathcal{F}$ *?lhs*› **by** (*auto simp add*: *F-Renaming F-GlobalDet*)
next
fix *s1 s2* **assume** ‹[✔(r)] = *map* (*map-event$_{ptick}$ f g*) *s1* @ *s2*›
‹*tickFree s1*› ‹*front-tickFree s2*› ‹*s1* ∈ $\mathcal{D}$ (*P a*)›
from ‹[✔(r)] = *map* (*map-event$_{ptick}$ f g*) *s1* @ *s2*› ‹*tickFree s1*›
have ‹*s1* = [] ∧ *s2* = [✔(r)]›
by (*cases s1*; *simp*) (*metis event$_{ptick}$.disc*(*2*) *event$_{ptick}$.map-disc-iff*(*1*))
with ∗(*1*, *2*) ‹*s1* ∈ $\mathcal{D}$ (*P a*)› **show** ‹(*s*, *X*) ∈ $\mathcal{F}$ *?lhs*›
by (*auto simp add*: *F-Renaming F-GlobalDet*)
qed
qed
qed

lemma *Renaming-Mprefix-bis* :
‹*Renaming* (□*a* ∈ *A* → *P a*) *f g* = □*a* ∈ *A*. (*f a* → *Renaming* (*P a*) *f g*)›
by (*simp add*: *Mprefix-GlobalDet Renaming-distrib-GlobalDet Renaming-write0*)

lemma *Renaming-GlobalDet-alt*:
‹*Renaming* (□ *a* ∈ *A*. *P* (*f a*)) *f g* = □ *b* ∈ *f* ‘ *A*. *Renaming* (*P b*) *f g*›
(**is** ‹*?lhs* = *?rhs*›)
by (*simp add*: *Renaming-distrib-GlobalDet mono-GlobalDet-eq2*)

lemma *Renaming-GlobalDet-inj-on*:
‹*inj-on f A* ⟹ *Renaming* (□ *a* ∈ *A*. *P a*) *f g* =
□ *b* ∈ *f* ‘ *A*. *Renaming* (*P* (*THE a*. *a* ∈ *A* ∧ *f a* = *b*)) *f g*›
by (*simp add*: *Renaming-distrib-GlobalDet inj-on-def mono-GlobalDet-eq2 the-equality*)

corollary *Renaming-GlobalDet-inj*:
‹*inj f* ⟹ *Renaming* (□ *a* ∈ *A*. *P a*) *f g* = □ *b* ∈ *f* ‘ *A*. *Renaming* (*P* (*THE a*. *f a* = *b*)) *f g*›

by (*subst Renaming-GlobalDet-inj-on*, *metis inj-eq inj-onI*)
(*rule mono-GlobalDet-eq*, *metis imageE inj-eq*)

lemma *Renaming-Interrupt* :
‹*Renaming* (*P* △ *Q*) *f g* = *Renaming P f g* △ *Renaming Q f g*› (**is** ‹*?lhs* = *?rhs*›)
proof (*subst Process-eq-spec-optimized*, *safe*)
fix *t* **assume** ‹*t* ∈ 𝒟 *?lhs*›
then obtain *t1 t2*
where ∗ : ‹*t* = *map* (*map-event*$_{ptick}$ *f g*) *t1* @ *t2*› ‹*tF t1*› ‹*ftF t2*› ‹*t1* ∈ 𝒟 (*P* △ *Q*)›
unfolding *D-Renaming* **by** *blast*
from ∗(*4*) **consider** ‹*t1* ∈ 𝒟 *P*›
| *u1 u2* **where** ‹*t1* = *u1* @ *u2*› ‹*u1* ∈ 𝒯 *P*› ‹*tF u1*› ‹*u2* ∈ 𝒟 *Q*›
unfolding *D-Interrupt* **by** *blast*
thus ‹*t* ∈ 𝒟 *?rhs*›
proof *cases*
from ∗(*1*−*3*) **show** ‹*t1* ∈ 𝒟 *P* ⟹ *t* ∈ 𝒟 *?rhs*›
by (*auto simp add*: *D-Interrupt D-Renaming*)
next
show ‹*t1* = *u1* @ *u2* ⟹ *u1* ∈ 𝒯 *P* ⟹ *tF u1* ⟹ *u2* ∈ 𝒟 *Q* ⟹ *t* ∈ 𝒟 *?rhs*› **for** *u1 u2*
by (*simp add*: *D-Interrupt Renaming-projs* ∗(*1*))
(*metis* ∗(*2*, *3*) *map-event*$_{ptick}$*-tickFree tickFree-append-iff*)
qed
next
fix *t* **assume** ‹*t* ∈ 𝒟 *?rhs*›
then consider ‹*t* ∈ 𝒟 (*Renaming P f g*)›
| *t1 t2* **where** ‹*t* = *t1* @ *t2*› ‹*t1* ∈ 𝒯 (*Renaming P f g*)› ‹*tF t1*› ‹*t2* ∈ 𝒟 (*Renaming Q f g*)›
unfolding *D-Interrupt* **by** *blast*
thus ‹*t* ∈ 𝒟 *?lhs*›
proof *cases*
show ‹*t* ∈ 𝒟 (*Renaming P f g*) ⟹ *t* ∈ 𝒟 *?lhs*›
by (*auto simp add*: *D-Renaming D-Interrupt*)
next
show ‹*t* = *t1* @ *t2* ⟹ *t1* ∈ 𝒯 (*Renaming P f g*) ⟹ *tF t1* ⟹ *t2* ∈ 𝒟 (*Renaming Q f g*) ⟹ *t* ∈ 𝒟 *?lhs*› **for** *t1 t2*
by (*auto simp add*: *Renaming-projs D-Interrupt append.assoc map-event*$_{ptick}$*-tickFree*)
(*metis* (*no-types*, *opaque-lifting*) *append.assoc map-append tickFree-append-iff*,
metis front-tickFree-append map-event$_{ptick}$*-tickFree*)
qed
next
fix *t X* **assume** *same-div* : ‹𝒟 *?lhs* = 𝒟 *?rhs*›
assume ‹(*t*, *X*) ∈ ℱ *?lhs*›
then consider ‹*t* ∈ 𝒟 *?lhs*›
| *u* **where** ‹*t* = *map* (*map-event*$_{ptick}$ *f g*) *u*› ‹(*u*, *map-event*$_{ptick}$ *f g* −‘ *X*) ∈ ℱ (*P* △ *Q*)›
unfolding *Renaming-projs* **by** *blast*

thus ‹$(t, X) \in \mathcal{F}$ *?rhs*›
proof *cases*
from *same-div D-F* **show** ‹$t \in \mathcal{D}$ *?lhs* $\Longrightarrow (t, X) \in \mathcal{F}$ *?rhs*› **by** *blast*
next
fix *u* **assume** $*$: ‹$t = map\ (map\text{-}event_{ptick}\ f\ g)\ u$› ‹$(u, map\text{-}event_{ptick}\ f\ g\ -`\ X) \in \mathcal{F}\ (P \triangle Q)$›
from $*(2)$ **consider** ‹$u \in \mathcal{D}\ (P \triangle Q)$›
| $u'\ r$ **where** ‹$u = u' @ [✓(r)]$› ‹$u' @ [✓(r)] \in \mathcal{T}\ P$›
| $X'\ r$ **where** ‹$map\text{-}event_{ptick}\ f\ g\ -`\ X = X' - \{✓(r)\}$› ‹$u @ [✓(r)] \in \mathcal{T}\ P$›
| ‹$(u, map\text{-}event_{ptick}\ f\ g\ -`\ X) \in \mathcal{F}\ P$› ‹$tF\ u$› ‹$([], map\text{-}event_{ptick}\ f\ g\ -`\ X) \in \mathcal{F}\ Q$›
| $u1\ u2$ **where** ‹$u = u1 @ u2$› ‹$u1 \in \mathcal{T}\ P$› ‹$tF\ u1$› ‹$(u2, map\text{-}event_{ptick}\ f\ g\ -`\ X) \in \mathcal{F}\ Q$› ‹$u2 \neq []$›
| $X'\ r$ **where** ‹$map\text{-}event_{ptick}\ f\ g\ -`\ X = X' - \{✓(r)\}$› ‹$u \in \mathcal{T}\ P$› ‹$tF\ u$› ‹$[✓(r)] \in \mathcal{T}\ Q$›
unfolding *Interrupt-projs* **by** *safe auto*
thus ‹$(t, X) \in \mathcal{F}$ *?rhs*›
proof *cases*
assume ‹$u \in \mathcal{D}\ (P \triangle Q)$›
hence ‹$t \in \mathcal{D}$ *?lhs*›
by (*simp add*: $*(1)$ *D-Renaming*)
(*metis* (*no-types*, *opaque-lifting*) *D-imp-front-tickFree append-Nil2 snoc-eq-iff-butlast butlast.simps*(*1*) *div-butlast-when-non-tickFree-iff front-tickFree-Nil front-tickFree-iff-tickFree-butlast front-tickFree-single map-butlast*)
with *same-div D-F* **show** ‹$(t, X) \in \mathcal{F}$ *?rhs*› **by** *blast*
next
show ‹$u = u' @ [✓(r)] \Longrightarrow u' @ [✓(r)] \in \mathcal{T}\ P \Longrightarrow (t, X) \in \mathcal{F}$ *?rhs*› **for** $u'\ r$
by (*auto simp add*: $*(1)$ *F-Interrupt T-Renaming*)
next
fix $X'\ r$ **assume** $**$: ‹$map\text{-}event_{ptick}\ f\ g\ -`\ X = X' - \{✓(r)\}$› ‹$u @ [✓(r)] \in \mathcal{T}\ P$›
from $**$ **obtain** X'' **where** ‹$X = X'' - \{✓(g\ r)\}$›
by (*metis DiffD2 Diff-insert-absorb* $event_{ptick}$*.simps*(*10*) *insertI1 vimage-eq*)
moreover from $**(2)$ **have** ‹$t @ [✓(g\ r)] \in \mathcal{T}$ (*Renaming P f g*)›
by (*auto simp add*: $*(1)$ *T-Renaming*)
ultimately show ‹$(t, X) \in \mathcal{F}$ *?rhs*› **by** (*auto simp add*: *F-Interrupt*)
next
show ‹$(u, map\text{-}event_{ptick}\ f\ g\ -`\ X) \in \mathcal{F}\ P \Longrightarrow tF\ u \Longrightarrow$
$([], map\text{-}event_{ptick}\ f\ g\ -`\ X) \in \mathcal{F}\ Q \Longrightarrow (t, X) \in \mathcal{F}$ *?rhs*›
using $map\text{-}event_{ptick}$*-tickFree* **by** (*auto simp add*: $*(1)$ *F-Interrupt F-Renaming*)
next
fix $u1\ u2$ **assume** ‹$u = u1 @ u2$› ‹$u1 \in \mathcal{T}\ P$› ‹$tF\ u1$›
‹$(u2, map\text{-}event_{ptick}\ f\ g\ -`\ X) \in \mathcal{F}\ Q$› ‹$u2 \neq []$›
hence ‹$t = map\ (map\text{-}event_{ptick}\ f\ g)\ u1 @ map\ (map\text{-}event_{ptick}\ f\ g)\ u2$›
‹$map\ (map\text{-}event_{ptick}\ f\ g)\ u1 \in \mathcal{T}$ (*Renaming P f g*)›
‹$tF\ (map\ (map\text{-}event_{ptick}\ f\ g)\ u1)$›
‹$(map\ (map\text{-}event_{ptick}\ f\ g)\ u2, X) \in \mathcal{F}$ (*Renaming Q f g*)›
‹$map\ (map\text{-}event_{ptick}\ f\ g)\ u2 \neq []$›
by (*auto simp add*: $*(1)$ *Renaming-projs* $map\text{-}event_{ptick}$*-tickFree*)

thus ‹$(t, X) \in \mathcal{F}$ *?rhs*› **by** (*simp add*: *F-Interrupt*) *blast*
next
fix X' *r* **assume** ∗∗ : ‹*map-event*$_{ptick}$ *f g* −‘ $X = X' - \{\checkmark(r)\}$› ‹$u \in \mathcal{T}$ *P*› ‹*tF u*› ‹$[\checkmark(r)] \in \mathcal{T}$ *Q*›
from ∗∗(*1*, *2*) **obtain** X'' **where** ‹$X = X'' - \{\checkmark(g\ r)\}$›
by (*metis DiffD2 Diff-insert-absorb event*$_{ptick}$*.simps*(*10*) *insertI1 vimage-eq*)
moreover from ∗∗(*2*−*4*) **have** ‹$t \in \mathcal{T}$ (*Renaming P f g*)› ‹*tF t*›
‹$[\checkmark(g\ r)] \in \mathcal{T}$ (*Renaming Q f g*)›
by (*auto simp add*: ∗(*1*) *T-Renaming map-event*$_{ptick}$*-tickFree*)
ultimately show ‹$(t, X) \in \mathcal{F}$ *?rhs*› **by** (*simp add*: *F-Interrupt*) *blast*
qed
qed
next
fix *t X* **assume** *same-div* : ‹$\mathcal{D}$ *?lhs* = $\mathcal{D}$ *?rhs*›
assume ‹$(t, X) \in \mathcal{F}$ *?rhs*›
then consider ‹$t \in \mathcal{D}$ *?rhs*›
| t' *s* **where** ‹$t = t'$ @ $[\checkmark(s)]$› ‹t' @ $[\checkmark(s)] \in \mathcal{T}$ (*Renaming P f g*)›
| X' *s* **where** ‹$X = X' - \{\checkmark(s)\}$› ‹*t* @ $[\checkmark(s)] \in \mathcal{T}$ (*Renaming P f g*)›
| ‹$(t, X) \in \mathcal{F}$ (*Renaming P f g*)› ‹*tF t*› ‹$([], X) \in \mathcal{F}$ (*Renaming Q f g*)›
| *t1 t2* **where** ‹*t* = *t1* @ *t2*› ‹*t1* $\in \mathcal{T}$ (*Renaming P f g*)› ‹*tF t1*›
‹$(t2, X) \in \mathcal{F}$ (*Renaming Q f g*)› ‹$t2 \neq []$›
| X' *s* **where** ‹$X = X' - \{\checkmark(s)\}$› ‹$t \in \mathcal{T}$ (*Renaming P f g*)› ‹*tF t*› ‹$[\checkmark(s)] \in \mathcal{T}$ (*Renaming Q f g*)›
by (*simp add*: *Interrupt-projs*) *blast*
thus ‹$(t, X) \in \mathcal{F}$ *?lhs*›
proof *cases*
from *same-div D-F* **show** ‹$t \in \mathcal{D}$ *?rhs* $\Longrightarrow (t, X) \in \mathcal{F}$ *?lhs*› **by** *blast*
next
show ‹$[\![t = t'$ @ $[\checkmark(s)]$; t' @ $[\checkmark(s)] \in \mathcal{T}$ (*Renaming P f g*)$]\!] \Longrightarrow (t, X) \in \mathcal{F}$ *?lhs*› **for** t' *s*
by (*simp add*: *Renaming-projs Interrupt-projs*)
(*metis T-nonTickFree-imp-decomp map-event*$_{ptick}$*-tickFree non-tickFree-tick tickFree-append-iff*)
next
fix X' *s* **assume** ∗ : ‹$X = X' - \{\checkmark(s)\}$› ‹*t* @ $[\checkmark(s)] \in \mathcal{T}$ (*Renaming P f g*)›
from ∗(*2*) **consider** *u1 u2* **where**
‹*t* @ $[\checkmark(s)]$ = *map* (*map-event*$_{ptick}$ *f g*) *u1* @ *u2*› ‹*tF u1*› ‹*ftF u2*› ‹*u1* $\in \mathcal{D}$ *P*›
| *u r* **where** ‹*s* = *g r*› ‹*t* = *map* (*map-event*$_{ptick}$ *f g*) *u*› ‹*u* @ $[\checkmark(r)] \in \mathcal{T}$ *P*›
by (*simp add*: *T-Renaming*)
(*metis* (*no-types*, *opaque-lifting*) *T-nonTickFree-imp-decomp event*$_{ptick}$*.disc*(*4*)
event$_{ptick}$*.map-sel*(*2*) *event*$_{ptick}$*.sel*(*2*) *last-map map-butlast map-event*$_{ptick}$*-tickFree*
non-tickFree-tick snoc-eq-iff-butlast tickFree-append-iff)
thus ‹$(t, X) \in \mathcal{F}$ *?lhs*›
proof *cases*
fix *u1 u2* **assume** ‹*t* @ $[\checkmark(s)]$ = *map* (*map-event*$_{ptick}$ *f g*) *u1* @ *u2*› ‹*tF u1*› ‹*ftF u2*› ‹*u1* $\in \mathcal{D}$ *P*›
hence ‹$t \in \mathcal{D}$ *?lhs*›
by (*cases u2 rule*: *rev-cases*)

(*auto simp add*: *D-Interrupt D-Renaming intro*: *front-tickFree-dw-closed*, *metis map-event*$_{ptick}$*-tickFree non-tickFree-tick tickFree-append-iff*)
with *D-F* **show** ‹(*t*, *X*) ∈ $\mathcal{F}$ *?lhs*› **by** *blast*
next
fix *u r* **assume** ‹*s* = *g r*› ‹*t* = *map* (*map-event*$_{ptick}$ *f g*) *u*› ‹*u* @ [✓(*r*)] ∈ $\mathcal{T}$ *P*›
moreover from ∗(*1*) ‹*s* = *g r*› **obtain** *X''* **where** ‹*map-event*$_{ptick}$ *f g* −‘ *X* = *X''* − {✓(*r*)}›
by (*metis Diff-iff Diff-insert-absorb event*$_{ptick}$*.simps*(*10*) *vimage-eq vimage-singleton-eq*)
ultimately show ‹(*t*, *X*) ∈ $\mathcal{F}$ *?lhs*› **by** (*simp add*: *F-Renaming F-Interrupt*) *metis*
qed
next
show ‹⟦(*t*, *X*) ∈ $\mathcal{F}$ (*Renaming P f g*); *tF t*; ([], *X*) ∈ $\mathcal{F}$ (*Renaming Q f g*)⟧ ⟹ (*t*, *X*) ∈ $\mathcal{F}$ *?lhs*›
by (*simp add*: *Renaming-projs Interrupt-projs*)
(*metis is-processT8 map-event*$_{ptick}$*-tickFree*)
next
fix *t1 t2* **assume** ∗ : ‹*t* = *t1* @ *t2*› ‹*t1* ∈ $\mathcal{T}$ (*Renaming P f g*)› ‹*tF t1*›
‹(*t2*, *X*) ∈ $\mathcal{F}$ (*Renaming Q f g*)› ‹*t2* ≠ []›
from ∗(*2*) **consider** *u1 u2* **where**
‹*t1* = *map* (*map-event*$_{ptick}$ *f g*) *u1* @ *u2*› ‹*tF u1*› ‹*ftF u2*› ‹*u1* ∈ $\mathcal{D}$ *P*›
| *u1* **where** ‹*t1* = *map* (*map-event*$_{ptick}$ *f g*) *u1*› ‹*u1* ∈ $\mathcal{T}$ *P*›
unfolding *T-Renaming* **by** *blast*
thus ‹(*t*, *X*) ∈ $\mathcal{F}$ *?lhs*›
proof *cases*
fix *u1 u2* **assume** ‹*t1* = *map* (*map-event*$_{ptick}$ *f g*) *u1* @ *u2*› ‹*tF u1*› ‹*ftF u2*› ‹*u1* ∈ $\mathcal{D}$ *P*›
hence ‹*t1* ∈ $\mathcal{D}$ *?lhs*› **by** (*auto simp add*: *D-Interrupt D-Renaming*)
with ∗(*1*, *3*, *4*) *F-imp-front-tickFree is-processT7* **have** ‹*t* ∈ $\mathcal{D}$ *?lhs*› **by** *blast*
with *D-F* **show** ‹(*t*, *X*) ∈ $\mathcal{F}$ *?lhs*› **by** *blast*
next
fix *u1* **assume** ∗∗ : ‹*t1* = *map* (*map-event*$_{ptick}$ *f g*) *u1*› ‹*u1* ∈ $\mathcal{T}$ *P*›
from ∗(*4*) **consider** *u2 u3* **where**
‹*t2* = *map* (*map-event*$_{ptick}$ *f g*) *u2* @ *u3*› ‹*tF u2*› ‹*ftF u3*› ‹*u2* ∈ $\mathcal{D}$ *Q*›
| *u2* **where** ‹*t2* = *map* (*map-event*$_{ptick}$ *f g*) *u2*› ‹(*u2*, *map-event*$_{ptick}$ *f g* −‘ *X*) ∈ $\mathcal{F}$ *Q*›
unfolding *F-Renaming* **by** *blast*
thus ‹(*t*, *X*) ∈ $\mathcal{F}$ *?lhs*›
proof *cases*
fix *u2 u3* **assume** ‹*t2* = *map* (*map-event*$_{ptick}$ *f g*) *u2* @ *u3*› ‹*tF u2*› ‹*ftF u3*› ‹*u2* ∈ $\mathcal{D}$ *Q*›
hence ‹*t* ∈ $\mathcal{D}$ *?lhs*›
by (*simp add*: ∗(*1*) ∗∗(*1*) *D-Renaming D-Interrupt flip*: *map-append append.assoc*)
(*metis* ∗(*3*) ∗∗(*1*, *2*) *map-event*$_{ptick}$*-tickFree tickFree-append-iff*)
with *D-F* **show** ‹(*t*, *X*) ∈ $\mathcal{F}$ *?lhs*› **by** *blast*
next

show ‹$t2$ = *map* (*map-event*$_{ptick}$ *f g*) *u2* ⟹ (*u2*, *map-event*$_{ptick}$ *f g* −‘ X) ∈ $\mathcal{F}$ Q
⟹ (t, X) ∈ $\mathcal{F}$ *?lhs*› **for** *u2*
by (*simp add*: *F-Renaming F-Interrupt* ∗(*1*) ∗∗(*1*) *flip*: *map-append*)
(*metis* ∗(*3*, *5*) ∗∗(*1*, *2*) *list.map-disc-iff map-event*$_{ptick}$*-tickFree*)
qed
qed
next
fix X′ s **assume** ∗ : ‹X = X′ − {✓(s)}› ‹t ∈ $\mathcal{T}$ (*Renaming P f g*)›
‹*tF t*› ‹[✓(s)] ∈ $\mathcal{T}$ (*Renaming Q f g*)›
from ∗(*2*) **consider** *u1 u2* **where**
‹t = *map* (*map-event*$_{ptick}$ *f g*) *u1* @ *u2*› ‹*tF u1*› ‹*ftF u2*› ‹*u1* ∈ $\mathcal{D}$ P›
| u **where** ‹t = *map* (*map-event*$_{ptick}$ *f g*) u› ‹u ∈ $\mathcal{T}$ P›
by (*auto simp add*: *T-Renaming*)
thus ‹(t, X) ∈ $\mathcal{F}$ *?lhs*›
proof *cases*
fix *u1 u2* **assume** ‹t = *map* (*map-event*$_{ptick}$ *f g*) *u1* @ *u2*› ‹*tF u1*› ‹*ftF u2*› ‹*u1* ∈ $\mathcal{D}$ P›
hence ‹t ∈ $\mathcal{D}$ *?lhs*› **by** (*auto simp add*: *D-Interrupt D-Renaming*)
with *D-F* **show** ‹(t, X) ∈ $\mathcal{F}$ *?lhs*› **by** *blast*
next
fix u **assume** ∗∗ : ‹t = *map* (*map-event*$_{ptick}$ *f g*) u› ‹u ∈ $\mathcal{T}$ P›
from ∗(*4*) **consider** ‹*Renaming Q f g* = ⊥› | r **where** ‹s = g r› ‹[✓(r)] ∈ $\mathcal{T}$ Q›
by (*simp add*: *Renaming-projs BOT-iff-tick-D*)
(*metis map-map-event*$_{ptick}$*-eq-tick-iff*)
thus ‹(t, X) ∈ $\mathcal{F}$ *?lhs*›
proof *cases*
assume ‹*Renaming Q f g* = ⊥›
hence ‹Q = ⊥› **by** (*simp add*: *Renaming-is-BOT-iff*)
hence ‹*Renaming* (P △ Q) *f g* = ⊥› **by** *simp*
thus ‹(t, X) ∈ $\mathcal{F}$ *?lhs*› **by** (*simp add*: *F-BOT* ∗(*3*))
next
fix r **assume** ‹s = g r› ‹[✓(r)] ∈ $\mathcal{T}$ Q›
moreover from ∗(*1*) ‹s = g r› **obtain** X′′
where ‹*map-event*$_{ptick}$ *f g* −‘ X = X′′ − {✓(r)}›
by (*metis DiffD2 Diff-empty Diff-insert0 event*$_{ptick}$*.simps*(*10*) *insertI1 vimage-eq*)
ultimately show ‹(t, X) ∈ $\mathcal{F}$ *?lhs*›
by (*simp add*: ∗∗(*1*) *F-Renaming F-Interrupt*)
(*metis* ∗(*3*) ∗∗(*1*, *2*) *map-event*$_{ptick}$*-tickFree*)
qed
qed
qed
qed

lemma *inj-on-Renaming-Throw* :
‹*Renaming* (P Θ a ∈ A. Q a) *f g* =

Renaming P f g Θ *b* ∈ *f* ' *A*. *Renaming* (*Q* (*inv-into A f b*)) *f g*›
(**is** ‹*?lhs* = *?rhs*›) **if** *inj-on-f* : ‹*inj-on f* (*events-of P* ∪ *A*)›
proof −
have $: ‹*set* (*map* (*map-event*$_{ptick}$ *f g*) *t*) ∩ *ev* ' *f* ' *A* = {}
⟷ *set t* ∩ *ev* ' *A* = {}› **if** ‹*t* ∈ 𝒯 *P*› **for** *t*
proof −
from ‹*t* ∈ 𝒯 *P*› *inj-on-f* **have** ‹*inj-on f* ({*a*. *ev a* ∈ *set t*} ∪ *A*)›
by (*auto simp add*: *inj-on-def events-of-memI*)
thus ‹*set* (*map* (*map-event*$_{ptick}$ *f g*) *t*) ∩ *ev* ' *f* ' *A* = {}
⟷ *set t* ∩ *ev* ' *A* = {}›
by (*auto simp add*: *disjoint-iff image-iff inj-on-def map-event*$_{ptick}$*-eq-ev-iff*)
(*metis event*$_{ptick}$*.simps*(*9*), *blast*)
qed
show ‹*?lhs* = *?rhs*›
proof (*subst Process-eq-spec-optimized*, *safe*)
fix *t* **assume** ‹*t* ∈ 𝒟 *?lhs*›
then obtain *t1 t2* **where** ∗ : ‹*t* = *map* (*map-event*$_{ptick}$ *f g*) *t1* @ *t2*› ‹*tF t1*›
‹*ftF t2*› ‹*t1* ∈ 𝒟 (*P* Θ *a* ∈ *A*. *Q a*)›
unfolding *D-Renaming* **by** *blast*
from ∗(*4*) **consider** *u1 u2* **where** ‹*t1* = *u1* @ *u2*› ‹*u1* ∈ 𝒟 *P*› ‹*tF u1*›
‹*set u1* ∩ *ev* ' *A* = {}› ‹*ftF u2*›
| *u1 a u2* **where** ‹*t1* = *u1* @ *ev a* # *u2*› ‹*u1* @ [*ev a*] ∈ 𝒯 *P*›
‹*set u1* ∩ *ev* ' *A* = {}› ‹*a* ∈ *A*› ‹*u2* ∈ 𝒟 (*Q a*)›
unfolding *D-Throw* **by** *blast*
thus ‹*t* ∈ 𝒟 *?rhs*›
proof *cases*
fix *u1 u2* **assume** ∗∗ : ‹*t1* = *u1* @ *u2*› ‹*u1* ∈ 𝒟 *P*› ‹*tF u1*›
‹*set u1* ∩ *ev* ' *A* = {}› ‹*ftF u2*›
from $ ∗∗(*2*) ∗∗(*4*) *D-T*
have ∗∗∗ : ‹*set* (*map* (*map-event*$_{ptick}$ *f g*) *u1*) ∩ *ev* ' *f* ' *A* = {}› **by** *blast*
have ‹*t* = *map* (*map-event*$_{ptick}$ *f g*) *u1* @ (*map* (*map-event*$_{ptick}$ *f g*) *u2* @ *t2*)›
by (*simp add*: ∗(*1*) ∗∗(*1*))
moreover from ∗(*2*, *3*) ∗∗(*1*) **have** ‹*ftF* (*map* (*map-event*$_{ptick}$ *f g*) *u2* @ *t2*)›
by (*simp add*: *front-tickFree-append map-event*$_{ptick}$*-tickFree*)
moreover have ‹*tF* (*map* (*map-event*$_{ptick}$ *f g*) *u1*)›
by (*simp add*: ∗∗(*3*) *map-event*$_{ptick}$*-tickFree*)
ultimately show ‹*t* ∈ 𝒟 *?rhs*›
by (*simp add*: *D-Throw D-Renaming*)
(*use* ∗∗(*2*) ∗∗(*3*) ∗∗∗ *front-tickFree-Nil* **in** *blast*)
next
fix *u1 a u2* **assume** ∗∗ : ‹*t1* = *u1* @ *ev a* # *u2*› ‹*u1* @ [*ev a*] ∈ 𝒯 *P*›
‹*set u1* ∩ *ev* ' *A* = {}› ‹*a* ∈ *A*› ‹*u2* ∈ 𝒟 (*Q a*)›
have ∗∗∗ : ‹*set* (*map* (*map-event*$_{ptick}$ *f g*) *u1*) ∩ *ev* ' *f* ' *A* = {}›
by (*meson* $ ∗∗(*2*) ∗∗(*3*) *T-F-spec is-processT3*)
have ‹*tF u2*› **using** ∗(*2*) ∗∗(*1*) **by** *auto*
moreover have ‹*t* = *map* (*map-event*$_{ptick}$ *f g*) *u1* @ *ev* (*f a*) # *map* (*map-event*$_{ptick}$ *f g*) *u2* @ *t2*›

by (*simp add*: ∗(*1*) ∗∗(*1*))
moreover from ∗∗(*2*) **have** ‹*map* (*map-event*$_{ptick}$ *f g*) *u1* @ [*ev* (*f a*)] ∈ $\mathcal{T}$ (*Renaming P f g*)›
by (*auto simp add*: *T-Renaming*)
moreover have ‹*inv-into A f* (*f a*) = *a*›
by (*meson* ∗∗(*4*) *inj-on-Un inv-into-f-eq inj-on-f*)
ultimately show ‹*t* ∈ $\mathcal{D}$ *?rhs*›
by (*simp add*: *D-Throw D-Renaming*)
(*metis* ∗(*3*) ∗∗(*4*) ∗∗(*5*) ∗∗∗ *imageI*)
qed
next
fix *t* **assume** ‹*t* ∈ $\mathcal{D}$ *?rhs*›
then consider *t1 t2* **where** ‹*t* = *t1* @ *t2*› ‹*t1* ∈ $\mathcal{D}$ (*Renaming P f g*)›
‹*tF t1*› ‹*set t1* ∩ *ev* ‘ *f* ‘ *A* = {}› ‹*ftF t2*›
| *t1 b t2* **where** ‹*t* = *t1* @ *ev b* # *t2*› ‹*t1* @ [*ev b*] ∈ $\mathcal{T}$ (*Renaming P f g*)›
‹*set t1* ∩ *ev* ‘ *f* ‘ *A* = {}› ‹*b* ∈ *f* ‘ *A*›
‹*t2* ∈ $\mathcal{D}$ (*Renaming* (*Q* (*inv-into A f b*)) *f g*)›
unfolding *D-Throw* **by** *blast*
thus ‹*t* ∈ $\mathcal{D}$ *?lhs*›
proof *cases*
fix *t1 t2* **assume** ∗ : ‹*t* = *t1* @ *t2*› ‹*t1* ∈ $\mathcal{D}$ (*Renaming P f g*)›
‹*tF t1*› ‹*set t1* ∩ *ev* ‘ *f* ‘ *A* = {}› ‹*ftF t2*›
from ∗(*2*) **obtain** *u1 u2*
where ∗∗ : ‹*t1* = *map* (*map-event*$_{ptick}$ *f g*) *u1* @ *u2*› ‹*tF u1*› ‹*ftF u2*› ‹*u1* ∈ $\mathcal{D}$ *P*›
unfolding *D-Renaming* **by** *blast*
from ∗(*4*) ∗∗(*1*) **have** ‹*set u1* ∩ *ev* ‘ *A* = {}› **by** *auto*
moreover have ‹*t* = *map* (*map-event*$_{ptick}$ *f g*) *u1* @ (*u2* @ *t2*)›
by (*simp add*: ∗(*1*) ∗∗(*1*))
moreover from ∗(*3*, *5*) ∗∗(*1*) *front-tickFree-append tickFree-append-iff*
have ‹*ftF* (*u2* @ *t2*)› **by** *blast*
ultimately show ‹*t* ∈ $\mathcal{D}$ *?lhs*›
by (*simp add*: *D-Renaming D-Throw*)
(*use* ∗∗(*2*, *4*) *front-tickFree-Nil* **in** *blast*)
next
fix *t1 b t2* **assume** ∗ : ‹*t* = *t1* @ *ev b* # *t2*› ‹*t1* @ [*ev b*] ∈ $\mathcal{T}$ (*Renaming P f g*)›
‹*set t1* ∩ *ev* ‘ *f* ‘ *A* = {}› ‹*b* ∈ *f* ‘ *A*›
‹*t2* ∈ $\mathcal{D}$ (*Renaming* (*Q* (*inv-into A f b*)) *f g*)›
from ‹*b* ∈ *f* ‘ *A*› **obtain** *a* **where** ‹*a* ∈ *A*› ‹*b* = *f a*› **by** *blast*
hence ‹*inv-into A f b* = *a*› **by** (*meson inj-on-Un inv-into-f-f inj-on-f*)
from ∗(*2*) **consider** *u1 u2* **where**
‹*t1* @ [*ev b*] = *map* (*map-event*$_{ptick}$ *f g*) *u1* @ *u2*› ‹*u2* ≠ []› ‹*tF u1*› ‹*ftF u2*› ‹*u1* ∈ $\mathcal{D}$ *P*›
| *u1* **where** ‹*t1* @ [*ev b*] = *map* (*map-event*$_{ptick}$ *f g*) *u1*› ‹*u1* ∈ $\mathcal{T}$ *P*›
by (*simp add*: *D-T T-Renaming*)
(*metis* (*no-types*, *opaque-lifting*) *D-T append.right-neutral*)
thus ‹*t* ∈ $\mathcal{D}$ *?lhs*›
proof *cases*

fix *u1 u2*
assume ∗∗ : ‹*t1* @ [*ev b*] = *map* (*map-event*$_{ptick}$ *f g*) *u1* @ *u2*› ‹*u2* ≠ []› ‹*tF u1*› ‹*ftF u2*› ‹*u1* ∈ 𝒟 *P*›
from ∗∗(*1*, *2*) **obtain** *u2′* **where** ∗∗∗ : ‹*t1* = *map* (*map-event*$_{ptick}$ *f g*) *u1* @ *u2′*›
by (*metis butlast-append butlast-snoc*)
from ∗(*3*) ∗∗∗ **have** ∗∗∗∗ : ‹*set u1* ∩ *ev* ‘ *A* = {}› **by** *auto*
have ∗∗∗∗∗ : ‹*t* = *map* (*map-event*$_{ptick}$ *f g*) *u1* @ (*u2′* @ *ev b* # *t2*)› ‹*ftF* (*u2′* @ *ev b* # *t2*)›
by (*simp-all add*: ∗(*1*) ∗∗∗ ∗∗∗∗ *front-tickFree-append-iff*)
(*metis* ∗(*2*, *5*) ∗∗∗ *D-imp-front-tickFree append-T-imp-tickFree event*$_{ptick}$.*disc*(*1*) *front-tickFree-Cons-iff not-Cons-self tickFree-append-iff*)
show ‹*t* ∈ 𝒟 *?lhs*›
by (*simp add*: *D-Renaming D-Throw*)
(*metis* ∗∗(*3*) ∗∗(*5*) ∗∗∗∗ ∗∗∗∗∗ *append-Nil2 front-tickFree-Nil*)
next
fix *u1* **assume** ‹*t1* @ [*ev b*] = *map* (*map-event*$_{ptick}$ *f g*) *u1*› ‹*u1* ∈ 𝒯 *P*›
then obtain *u1′* **where** ∗∗ : ‹*t1* = *map* (*map-event*$_{ptick}$ *f g*) *u1′*› ‹*u1′* @ [*ev a*] ∈ 𝒯 *P*›
by (*cases u1 rule*: *rev-cases*, *simp-all add*: ‹*b* = *f a*› *ev-eq-map-event*$_{ptick}$*-iff*)
(*metis Nil-is-append-conv Un-iff* ‹*a* ∈ *A*› *events-of-memI inj-onD inj-on-f last-in-set last-snoc list.distinct*(*1*))
from ∗(*3*) ∗∗(*1*) **have** ∗∗∗ : ‹*set u1′* ∩ *ev* ‘ *A* = {}› **by** *auto*
from ∗(*5*) ‹*inv-into A f b* = *a*› **obtain** *u2 u3* **where**
∗∗∗∗ : ‹*t2* = *map* (*map-event*$_{ptick}$ *f g*) *u2* @ *u3*› ‹*tF u2*› ‹*ftF u3*› ‹*u2* ∈ 𝒟 (*Q a*)›
unfolding *Renaming-projs* **by** *blast*
have ∗∗∗∗∗ : ‹*t* = *map* (*map-event*$_{ptick}$ *f g*) (*u1′* @ *ev a* # *u2*) @ *u3*› ‹*tF* (*u1′* @ *ev a* # *u2*)›
by (*simp-all add*: ∗(*1*) ∗∗(*1*) ‹*b* = *f a*› ∗∗∗∗(*1*))
(*metis* ∗∗(*2*) ∗∗∗∗(*2*) *T-imp-front-tickFree butlast-snoc front-tickFree-iff-tickFree-butlast*)
show ‹*t* ∈ 𝒟 *?lhs*›
by (*simp add*: *D-Renaming D-Throw*)
(*metis* ∗∗(*2*) ∗∗∗ ∗∗∗∗(*3*, *4*) ∗∗∗∗∗(*1*, *2*) ‹*a* ∈ *A*›)
qed
qed
next
fix *t X* **assume** *same-div* : ‹𝒟 *?lhs* = 𝒟 *?rhs*›
assume ‹(*t*, *X*) ∈ ℱ *?lhs*›
then consider ‹*t* ∈ 𝒟 *?lhs*›
| *u* **where** ‹*t* = *map* (*map-event*$_{ptick}$ *f g*) *u*› ‹(*u*, *map-event*$_{ptick}$ *f g* −‘ *X*) ∈ ℱ (*P* Θ *a* ∈ *A*. *Q a*)›
unfolding *Renaming-projs* **by** *blast*
thus ‹(*t*, *X*) ∈ ℱ *?rhs*›
proof *cases*
from *same-div D-F* **show** ‹*t* ∈ 𝒟 *?lhs* ⟹ (*t*, *X*) ∈ ℱ *?rhs*› **by** *blast*
next
fix *u* **assume** ∗ : ‹*t* = *map* (*map-event*$_{ptick}$ *f g*) *u*›

‹(u, map-event$_{ptick}$ f g −‘ X) ∈ $\mathcal{F}$ (P Θ a ∈ A. Q a)›
then consider ‹(u, map-event$_{ptick}$ f g −‘ X) ∈ $\mathcal{F}$ P› ‹set u ∩ ev ‘ A = {}›
| u1 u2 **where** ‹u = u1 @ u2› ‹u1 ∈ $\mathcal{D}$ P› ‹tF u1› ‹set u1 ∩ ev ‘ A = {}›
‹ftF u2›
| u1 a u2 **where** ‹u = u1 @ ev a # u2› ‹u1 @ [ev a] ∈ $\mathcal{T}$ P› ‹set u1 ∩ ev
‘ A = {}›
‹a ∈ A› ‹(u2, map-event$_{ptick}$ f g −‘ X) ∈ $\mathcal{F}$ (Q a)›
unfolding F-Throw **by** blast
thus ‹(t, X) ∈ $\mathcal{F}$?rhs›
proof cases
show ‹(u, map-event$_{ptick}$ f g −‘ X) ∈ $\mathcal{F}$ P ⟹ set u ∩ ev ‘ A = {} ⟹ (t,
X) ∈ $\mathcal{F}$?rhs›
by (simp add: F-Throw F-Renaming) (metis $ ∗(1) F-T)
next
fix u1 u2 **assume** ‹u = u1 @ u2› ‹u1 ∈ $\mathcal{D}$ P› ‹tF u1› ‹set u1 ∩ ev ‘ A =
{}› ‹ftF u2›
hence ‹t ∈ $\mathcal{D}$?lhs›
by (simp add: ∗(1) D-Renaming D-Throw)
(metis append-Nil2 front-tickFree-Nil map-event$_{ptick}$-front-tickFree)
with same-div D-F **show** ‹(t, X) ∈ $\mathcal{F}$?rhs› **by** blast
next
fix u1 a u2
assume ∗∗ : ‹u = u1 @ ev a # u2› ‹u1 @ [ev a] ∈ $\mathcal{T}$ P› ‹set u1 ∩ ev ‘ A
= {}›
‹a ∈ A› ‹(u2, map-event$_{ptick}$ f g −‘ X) ∈ $\mathcal{F}$ (Q a)›
have ∗∗∗ : ‹set (map (map-event$_{ptick}$ f g) u1) ∩ ev ‘ f ‘ A = {}›
by (meson $ ∗∗(2, 3) T-F-spec is-processT3)
have ‹t = map (map-event$_{ptick}$ f g) u1 @ ev (f a) # map (map-event$_{ptick}$
f g) u2›
by (simp add: ∗(1) ∗∗(1))
moreover from ∗∗(2) **have** ‹map (map-event$_{ptick}$ f g) u1 @ [ev (f a)] ∈
$\mathcal{T}$ (Renaming P f g)›
by (auto simp add: T-Renaming)
moreover have ‹inv-into A f (f a) = a›
by (meson ∗∗(4) inj-on-Un inv-into-f-f inj-on-f)
moreover from ∗∗(5) **have** ‹(map (map-event$_{ptick}$ f g) u2, X) ∈ $\mathcal{F}$
(Renaming (Q a) f g)›
by (auto simp add: F-Renaming)
ultimately show ‹(t, X) ∈ $\mathcal{F}$?rhs›
by (simp add: F-Throw) (metis ∗∗(4) ∗∗∗ image-eqI)
qed
qed
next
fix t X **assume** same-div : ‹$\mathcal{D}$?lhs = $\mathcal{D}$?rhs›
assume ‹(t, X) ∈ $\mathcal{F}$?rhs›
then consider ‹t ∈ $\mathcal{D}$?rhs›
| ‹(t, X) ∈ $\mathcal{F}$ (Renaming P f g)› ‹set t ∩ ev ‘ f ‘ A = {}›
| t1 b t2 **where** ‹t = t1 @ ev b # t2› ‹t1 @ [ev b] ∈ $\mathcal{T}$ (Renaming P f g)›
‹set t1 ∩ ev ‘ f ‘ A = {}› ‹b ∈ f ‘ A›

‹(t2, X) ∈ $\mathcal{F}$ (*Renaming* (*Q* (*inv-into A f b*)) *f g*)›
unfolding *Throw-projs* **by** *auto*
thus ‹(t, X) ∈ $\mathcal{F}$ *?lhs*›
proof *cases*
from *same-div D-F* **show** ‹t ∈ $\mathcal{D}$ *?rhs* ⟹ (t, X) ∈ $\mathcal{F}$ *?lhs*› **by** *blast*
next
assume ∗ : ‹(t, X) ∈ $\mathcal{F}$ (*Renaming P f g*)› ‹*set t* ∩ *ev* ‘ *f* ‘ *A* = {}›
from ∗(*1*) **consider** ‹t ∈ $\mathcal{D}$ (*Renaming P f g*)›
| *u* **where** ‹t = *map* (*map-event*$_{ptick}$ *f g*) *u*› ‹(u, *map-event*$_{ptick}$ *f g* −‘ *X*) ∈ $\mathcal{F}$ *P*›
unfolding *Renaming-projs* **by** *blast*
thus ‹(t, X) ∈ $\mathcal{F}$ *?lhs*›
proof *cases*
assume ‹t ∈ $\mathcal{D}$ (*Renaming P f g*)›
hence ‹t ∈ $\mathcal{D}$ *?lhs*›
by (*simp add*: *D-Renaming D-Throw*)
(*metis* (*no-types*, *lifting*) $ ∗(*2*) *D-T Un-Int-eq*(*3*) *append-Nil2*
front-tickFree-Nil inf-bot-right inf-sup-aci(*2*) *set-append*)
with *D-F* **show** ‹(t, X) ∈ $\mathcal{F}$ *?lhs*› **by** *blast*
next
show ‹t = *map* (*map-event*$_{ptick}$ *f g*) *u* ⟹ (u, *map-event*$_{ptick}$ *f g* −‘ *X*) ∈ $\mathcal{F}$ *P*
⟹ (t, X) ∈ $\mathcal{F}$ *?lhs*› **for** *u*
by (*simp add*: *F-Renaming F-Throw*) (*metis* $ ∗(*2*) *F-T*)
qed
next
fix *t1 b t2*
assume ∗ : ‹t = t1 @ *ev b* # t2› ‹t1 @ [*ev b*] ∈ $\mathcal{T}$ (*Renaming P f g*)›
‹*set t1* ∩ *ev* ‘ *f* ‘ *A* = {}› ‹b ∈ f ‘ A›
‹(t2, X) ∈ $\mathcal{F}$ (*Renaming* (*Q* (*inv-into A f b*)) *f g*)›
from ∗(*4*) **obtain** *a* **where** ‹a ∈ A› ‹b = f a› **by** *blast*
hence ‹*inv-into A f b* = a› **by** (*meson inj-on-Un inv-into-f-f inj-on-f*)
from ∗(*2*) **consider** *u1 u2* **where**
‹t1 @ [*ev b*] = *map* (*map-event*$_{ptick}$ *f g*) u1 @ u2› ‹u2 ≠ []› ‹*tF u1*› ‹*ftF u2*› ‹u1 ∈ $\mathcal{D}$ P›
| *u1* **where** ‹t1 @ [*ev b*] = *map* (*map-event*$_{ptick}$ *f g*) u1› ‹u1 ∈ $\mathcal{T}$ P›
by (*simp add*: *D-T T-Renaming*)
(*metis* (*no-types*, *opaque-lifting*) *D-T append.right-neutral*)
thus ‹(t, X) ∈ $\mathcal{F}$ *?lhs*›
proof *cases*
fix *u1 u2*
assume ∗∗ : ‹t1 @ [*ev b*] = *map* (*map-event*$_{ptick}$ *f g*) u1 @ u2› ‹u2 ≠ []› ‹*tF u1*› ‹*ftF u2*› ‹u1 ∈ $\mathcal{D}$ P›
from ∗∗(*1*, *2*) **obtain** *u2′* **where** ∗∗∗ : ‹t1 = *map* (*map-event*$_{ptick}$ *f g*) u1 @ u2′›
by (*metis butlast-append butlast-snoc*)
from ∗(*3*) ∗∗∗ **have** ‹*set u1* ∩ *ev* ‘ *A* = {}› **by** *auto*
with ∗∗(*3*−*5*) ∗∗∗ **have** ‹t ∈ $\mathcal{D}$ *?rhs*›
by (*simp add*: *D-Renaming D-Throw*)

(*metis* ∗(*1*, *3*) *F-imp-front-tickFree* ‹(*t*, *X*) ∈ $\mathcal{F}$ *?rhs*› *front-tickFree-Nil* *front-tickFree-append-iff front-tickFree-dw-closed list.discI*)
with *same-div D-F* **show** ‹(*t*, *X*) ∈ $\mathcal{F}$ *?lhs*› **by** *blast*
next
fix *u1* **assume** ‹*t1* @ [*ev b*] = *map* (*map-event*$_{ptick}$ *f g*) *u1*› ‹*u1* ∈ $\mathcal{T}$ *P*›
then obtain *u1′* **where** ∗∗ : ‹*t1* = *map* (*map-event*$_{ptick}$ *f g*) *u1′*› ‹*u1′* @ [*ev a*] ∈ $\mathcal{T}$ *P*›
by (*cases u1 rule*: *rev-cases*, *simp-all add*: ‹*b* = *f a*› *ev-eq-map-event*$_{ptick}$*-iff*)
(*metis Nil-is-append-conv Un-iff* ‹*a* ∈ *A*› *events-of-memI inj-onD* *inj-on-f last-in-set last-snoc list.distinct*(*1*))
from ∗(*3*) ∗∗(*1*) **have** ∗∗∗ : ‹*set u1′* ∩ *ev* ‘ *A* = {}› **by** *auto*
from ∗(*5*) ‹*inv-into A f b* = *a*› **consider** ‹*t2* ∈ $\mathcal{D}$ (*Renaming* (*Q a*) *f g*)›
| *u2* **where** ‹*t2* = *map* (*map-event*$_{ptick}$ *f g*) *u2*› ‹(*u2*, *map-event*$_{ptick}$ *f g* −‘ *X*) ∈ $\mathcal{F}$ (*Q a*)›
unfolding *Renaming-projs* **by** *blast*
thus ‹(*t*, *X*) ∈ $\mathcal{F}$ *?lhs*›
proof *cases*
assume ‹*t2* ∈ $\mathcal{D}$ (*Renaming* (*Q a*) *f g*)›
with ∗(*1*−*4*) ‹*inv-into A f b* = *a*› **have** ‹*t* ∈ $\mathcal{D}$ *?rhs*›
by (*auto simp add*: *D-Throw*)
with *same-div D-F* **show** ‹(*t*, *X*) ∈ $\mathcal{F}$ *?lhs*› **by** *blast*
next
fix *u2* **assume** ∗∗∗∗ : ‹*t2* = *map* (*map-event*$_{ptick}$ *f g*) *u2*›
‹(*u2*, *map-event*$_{ptick}$ *f g* −‘ *X*) ∈ $\mathcal{F}$ (*Q a*)›
from ∗∗∗∗(*1*) **have** ∗∗∗∗∗ : ‹*t* = *map* (*map-event*$_{ptick}$ *f g*) (*u1′* @ *ev a* # *u2*)›
by (*simp add*: ∗(*1*) ∗∗(*1*) ∗∗∗∗ ‹*b* = *f a*›)
show ‹(*t*, *X*) ∈ $\mathcal{F}$ *?lhs*›
by (*simp add*: *F-Renaming F-Throw*)
(*use* ∗∗(*2*) ∗∗∗ ∗∗∗∗(*2*) ∗∗∗∗∗ ‹*a* ∈ *A*› **in** *blast*)
qed
qed
qed
qed
qed

4.2.1 *Renaming* and (\)

When *f* is one to one, *Renaming* (*P* \ *S*) *f* will behave like we expect it to do.

lemma *strict-mono-map*: ‹*strict-mono g* ⟹ *strict-mono* (λ*i*. *map f* (*g i*))›
unfolding *strict-mono-def less-eq-list-def less-list-def prefix-def* **by** *fastforce*

lemma *trace-hide-map-map-event*$_{ptick}$:
‹*inj-on* (*map-event*$_{ptick}$ *f g*) (*set s* ∪ *ev* ‘ *S*) ⟹
trace-hide (*map* (*map-event*$_{ptick}$ *f g*) *s*) (*ev* ‘ *f* ‘ *S*) =
map (*map-event*$_{ptick}$ *f g*) (*trace-hide s* (*ev* ‘ *S*))›

proof (*induct s*)
  case *Nil*
  show *?case* **by** *simp*
next
  case (*Cons e s*)
  hence ∗ : ‹*trace-hide* (*map* (*map-event*$_{ptick}$ *f g*) *s*) (*ev* ‘ *f* ‘ *S*) =
            map (*map-event*$_{ptick}$ *f g*) (*trace-hide s* (*ev* ‘ *S*))› **by** *fastforce*
  from *Cons.prems*[*unfolded inj-on-def*, *rule-format*, *of e*, *simplified*] **show** *?case*
    apply (*simp add*: ∗)
    apply (*simp add*: *image-iff*)
    by (*metis event*$_{ptick}$*.simps*(*9*))
qed

lemma *inj-on-map-event*$_{ptick}$*-set-T*:
  ‹*inj-on* (*map-event*$_{ptick}$ *f g*) (*set s*)› **if** ‹*inj-on f* (*events-of P*)› ‹*s* ∈ $\mathcal{T}$ *P*›
proof (*rule inj-onI*)
  show ‹*e* ∈ *set s* ⟹ *e*′ ∈ *set s* ⟹ *map-event*$_{ptick}$ *f g e* = *map-event*$_{ptick}$ *f g e*′
⟹ *e* = *e*′› **for** *e e*′
    by (*cases e*; *cases e*′; *simp*)
      (*meson events-of-memI inj-onD that*(*1*, *2*),
      metis T-imp-front-tickFree event$_{ptick}$*.disc*(*2*) *event*$_{ptick}$*.simps*(*2*) *front-tickFree-Cons-iff*
that(*2*)
      front-tickFree-nonempty-append-imp list.distinct(*1*) *snoc-eq-iff-butlast split-list-last*)
qed

theorem *bij-Renaming-Hiding*: ‹*Renaming* (*P* \ *S*) *f g* = *Renaming P f g* \ *f* ‘ *S*›
  (**is** ‹*?lhs* = *?rhs*›) **if** *bij-f*: ‹*bij f*› **and** *bij-g* : ‹*bij g*›
proof −
  have *inj-on-map-event*$_{ptick}$: ‹*inj-on* (*map-event*$_{ptick}$ *f g*) *X*› **for** *X*
  proof (*rule inj-onI*)
    show ‹*e* ∈ *X* ⟹ *e*′ ∈ *X* ⟹ *map-event*$_{ptick}$ *f g e* = *map-event*$_{ptick}$ *f g e*′ ⟹
e = *e*′› **for** *e e*′
      by (*cases e*; *cases e*′; *simp*)
        (*metis bij-f bij-pointE*, *metis bij-g bij-pointE*)
  qed
  have *inj-on-map-event*$_{ptick}$*-inv* : ‹*inj-on* (*map-event*$_{ptick}$ (*inv f*) (*inv g*)) *X*› **for**
X
  proof (*rule inj-onI*)
    show ‹*e* ∈ *X* ⟹ *e*′ ∈ *X* ⟹ *map-event*$_{ptick}$ (*inv f*) (*inv g*) *e* = *map-event*$_{ptick}$
(*inv f*) (*inv g*) *e*′
          ⟹ *e* = *e*′› **for** *e e*′
      by (*cases e*; *cases e*′, *simp-all*)
        (*metis bij-f bij-inv-eq-iff*, *metis bij-g bij-inv-eq-iff*)
  qed
  show ‹*?lhs* = *?rhs*›
  proof (*subst Process-eq-spec-optimized*, *safe*)
    fix *s*

assume ‹$s \in \mathcal{D}$ *?lhs*›
then obtain *s1 s2* **where** $*$: ‹*tickFree s1*› ‹*front-tickFree s2*›
‹s = *map* (*map-event*$_{ptick}$ *f g*) *s1* @ *s2*› ‹*s1* $\in \mathcal{D}$ ($P \setminus S$)›
by (*simp add*: *D-Renaming*) *blast*
from $*(4)$ **obtain** *t u*
where $**$: ‹*front-tickFree u*› ‹*tickFree t*› ‹*s1* = *trace-hide t* (*ev* ‘ *S*) @ *u*›
‹$t \in \mathcal{D}$ P ∨ (∃ *h*. *isInfHiddenRun h P S* ∧ t ∈ *range h*)›
by (*simp add*: *D-Hiding*) *blast*
from $**(4)$ **show** ‹$s \in \mathcal{D}$ *?rhs*›
proof (*elim disjE*)
assume ‹$t \in \mathcal{D}$ P›
hence ‹*front-tickFree* (*map* (*map-event*$_{ptick}$ *f g*) *u* @ *s2*) ∧ *tickFree* (*map* (*map-event*$_{ptick}$ *f g*) *t*) ∧
s = *trace-hide* (*map* (*map-event*$_{ptick}$ *f g*) *t*) (*ev* ‘ *f* ‘ *S*) @ *map* (*map-event*$_{ptick}$ *f g*) *u* @ *s2* ∧
map (*map-event*$_{ptick}$ *f g*) $t \in \mathcal{D}$ (*Renaming P f g*)›
apply (*simp add*: $*(3)$ $**(2, 3)$ *map-event*$_{ptick}$*-tickFree*, *intro conjI*)
apply (*metis* $*(1, 2)$ $**(1)$ $**(3)$ *front-tickFree-append-iff*
map-event$_{ptick}$*-front-tickFree map-event*$_{ptick}$*-tickFree tickFree-append-iff*)
apply (*simp add*: *trace-hide-map-map-event*$_{ptick}$ *inj-on-map-event*$_{ptick}$)
by (*metis* (*mono-tags*, *lifting*) $**(2)$ *CollectI D-Renaming append.right-neutral front-tickFree-Nil*)
thus ‹$s \in \mathcal{D}$ *?rhs*› **by** (*simp add*: *D-Hiding*) *blast*
next
assume ‹∃ *h*. *isInfHiddenRun h P S* ∧ t ∈ *range h*›
then obtain *h* **where** ‹*isInfHiddenRun h P S*› ‹t ∈ *range h*› **by** *blast*
hence ‹*front-tickFree* (*map* (*map-event*$_{ptick}$ *f g*) *u* @ *s2*) ∧
tickFree (*map* (*map-event*$_{ptick}$ *f g*) *t*) ∧
s = *trace-hide* (*map* (*map-event*$_{ptick}$ *f g*) *t*) (*ev* ‘ *f* ‘ *S*) @ *map* (*map-event*$_{ptick}$ *f g*) *u* @ *s2* ∧
isInfHiddenRun (λ*i*. *map* (*map-event*$_{ptick}$ *f g*) (*h i*)) (*Renaming P f g*) (*f* ‘ *S*) ∧
map (*map-event*$_{ptick}$ *f g*) t ∈ *range* (λ*i*. *map* (*map-event*$_{ptick}$ *f g*) (*h i*))›
apply (*simp add*: $*(3)$ $**(2, 3)$ *map-event*$_{ptick}$*-tickFree*, *intro conjI*)
apply (*metis* $*(1, 2)$ $**(3)$ *front-tickFree-append map-event*$_{ptick}$*-tickFree tickFree-append-iff*)
apply (*rule trace-hide-map-map-event*$_{ptick}$[*OF inj-on-map-event*$_{ptick}$, *symmetric*])
apply (*solves* ‹*rule strict-mono-map*, *simp*›)
apply (*solves* ‹*auto simp add*: *T-Renaming*›)
apply (*subst* (*1 2*) *trace-hide-map-map-event*$_{ptick}$[*OF inj-on-map-event*$_{ptick}$])
by *metis blast*
thus ‹$s \in \mathcal{D}$ *?rhs*› **by** (*simp add*: *D-Hiding*) *blast*
qed
next
fix *s*
assume ‹$s \in \mathcal{D}$ *?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*) ∨
(∃ *h*. *isInfHiddenRun h* (*Renaming P f g*) (*f* ‘ *S*) ∧ *t* ∈ *range h*)›
by (*simp add*: *D-Hiding*) *blast*
from ∗(*4*) **show** ‹*s* ∈ 𝒟 *?lhs*›
proof (*elim disjE*)
assume ‹*t* ∈ 𝒟 (*Renaming P f g*)›
then obtain *t1 t2* **where** ∗∗ : ‹*tickFree t1*› ‹*front-tickFree t2*›
‹*t* = *map* (*map-event*$_{ptick}$ *f g*) *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* (*map-event*$_{ptick}$ *f g*) *t1*) (*ev* ‘ *f* ‘ *S*) @ *trace-hide t2* (*ev* ‘ *f* ‘ *S*) @ *u* =
map (*map-event*$_{ptick}$ *f g*) (*trace-hide t1* (*ev* ‘ *S*)) @ *trace-hide t2* (*ev* ‘ *f* ‘ *S*) @ *u* ∧
trace-hide t1 (*ev* ‘ *S*) ∈ 𝒟 (*P* \ *S*)›
apply (*simp*, *intro conjI*)
using ∗∗(*1*) *Hiding-tickFree* **apply** *blast*
using ∗(*1*, *2*) ∗∗(*3*) *Hiding-tickFree front-tickFree-append tickFree-append-iff* **apply** *blast*
apply (*rule trace-hide-map-map-event*$_{ptick}$[*OF inj-on-map-event*$_{ptick}$])
using ∗∗(*4*) *mem-D-imp-mem-D-Hiding* **by** *blast*
thus ‹*s* ∈ 𝒟 *?lhs*› **by** (*simp add*: *D-Renaming* ∗(*3*) ∗∗(*3*)) *blast*
next
have *inv-S*: ‹*S* = *inv f* ‘ *f* ‘ *S*› **by** (*simp add*: *bij-is-inj bij-f*)
have *inj-inv-f*: ‹*inj* (*inv f*)›
by (*simp add*: *bij-imp-bij-inv bij-is-inj bij-f*)
have ∗∗ : ‹*map* (*map-event*$_{ptick}$ (*inv f*) (*inv g*) ∘ *map-event*$_{ptick}$ *f g*) *v* = *v*› **for** *v*
by (*induct v*, *simp-all*)
(*metis bij-f bij-g bij-inv-eq-iff event*$_{ptick}$.*exhaust event*$_{ptick}$.*simps*(*9*) *map-event*$_{ptick}$-*eq-tick-iff*)
assume ‹∃ *h*. *isInfHiddenRun h* (*Renaming P f g*) (*f* ‘ *S*) ∧ *t* ∈ *range h*›
then obtain *h*
where ∗∗∗ : ‹*isInfHiddenRun h* (*Renaming P f g*) (*f* ‘ *S*)› ‹*t* ∈ *range h*› **by** *blast*
then consider *t1* **where** ‹*t1* ∈ 𝒯 *P*› ‹*t* = *map* (*map-event*$_{ptick}$ *f g*) *t1*›
| *t1 t2* **where** ‹*tickFree t1*› ‹*front-tickFree t2*›
‹*t* = *map* (*map-event*$_{ptick}$ *f g*) *t1* @ *t2*› ‹*t1* ∈ 𝒟 *P*›
by (*simp add*: *T-Renaming*) *blast*
thus ‹*s* ∈ 𝒟 *?lhs*›
proof *cases*
fix *t1* **assume** ∗∗∗∗ : ‹*t1* ∈ 𝒯 *P*› ‹*t* = *map* (*map-event*$_{ptick}$ *f g*) *t1*›
have ∗∗∗∗∗ : ‹*t1* = *map* (*map-event*$_{ptick}$ (*inv f*) (*inv g*)) *t*› **by** (*simp add*: ∗∗∗∗(*2*) ∗∗)
have ∗∗∗∗∗∗ : ‹*trace-hide t1* (*ev* ‘ *S*) = *trace-hide t1* (*ev* ‘ *S*) ∧
isInfHiddenRun (λ*i*. *map* (*map-event*$_{ptick}$ (*inv f*) (*inv g*)) (*h i*)) *P S* ∧

t1 ∈ *range* (λ*i*. *map* (*map-event*$_{ptick}$ (*inv f*) (*inv g*)) (*h i*))›
apply (*simp add*: ∗∗∗(*1*) *strict-mono-map*, *intro conjI*)
apply (*subst Renaming-inv*[**where** *f* = *f* **and** *g* = *g*, *symmetric*])
apply (*solves* ‹*simp add*: *bij-is-inj bij-f*›)
apply (*solves* ‹*simp add*: *bij-is-inj bij-g*›)

using ∗∗∗(*1*) **apply** (*subst T-Renaming*, *blast*)
apply (*subst* (*1 2*) *inv-S*, *subst* (*1 2*) *trace-hide-map-map-event*$_{ptick}$[*OF inj-on-map-event*$_{ptick}$*-inv*])
apply (*metis* ∗∗∗(*1*))
using ∗∗∗(*2*) ∗∗∗∗∗ **by** *blast*
have ‹*tickFree* (*trace-hide t1* (*ev* ‘ *S*)) ∧ *front-tickFree t1* ∧
trace-hide (*map* (*map-event*$_{ptick}$ *f g*) *t1*) (*ev* ‘ *f* ‘ *S*) @ *u* =
map (*map-event*$_{ptick}$ *f g*) (*trace-hide t1* (*ev* ‘ *S*)) @ *u* ∧
trace-hide t1 (*ev* ‘ *S*) ∈ 𝒟 (*P* \ *S*)›
apply (*simp*, *intro conjI*)
using ∗(*2*) ∗∗∗∗(*2*) *map-event*$_{ptick}$*-tickFree Hiding-tickFree* **apply** *blast*
using ∗∗∗∗(*1*) *is-processT2-TR* **apply** *blast*
apply (*rule trace-hide-map-map-event*$_{ptick}$[*OF inj-on-map-event*$_{ptick}$])
apply (*simp add*: *D-Renaming D-Hiding*)
using ∗(*2*) ∗∗∗∗∗ ∗∗∗∗∗∗ *map-event*$_{ptick}$*-tickFree front-tickFree-Nil* **by** *blast*
with ∗(*1*) **show** ‹*s* ∈ 𝒟 *?lhs*› **by** (*simp add*: *D-Renaming* ∗(*3*) ∗∗∗∗(*2*)) *blast*
next
fix *t1 t2* **assume** ∗∗∗∗ : ‹*tickFree t1*› ‹*front-tickFree t2*›
‹*t* = *map* (*map-event*$_{ptick}$ *f g*) *t1* @ *t2*› ‹*t1* ∈ 𝒟 *P*›
have ‹*tickFree* (*trace-hide t1* (*ev* ‘ *S*)) ∧
front-tickFree (*trace-hide t2* (*ev* ‘ *f* ‘ *S*) @ *u*) ∧
trace-hide (*map* (*map-event*$_{ptick}$ *f g*) *t1*) (*ev* ‘ *f* ‘ *S*) @ *trace-hide t2* (*ev* ‘ *f* ‘ *S*) @ *u* =
map (*map-event*$_{ptick}$ *f g*) (*trace-hide t1* (*ev* ‘ *S*)) @ *trace-hide t2* (*ev* ‘ *f* ‘ *S*) @ *u* ∧
trace-hide t1 (*ev* ‘ *S*) ∈ 𝒟 (*P* \ *S*)›
apply (*simp*, *intro conjI*)
using ∗∗∗∗(*1*) *Hiding-tickFree* **apply** *blast*
using ∗(*1*, *2*) ∗∗∗∗(*3*) *Hiding-tickFree front-tickFree-append tickFree-append-iff* **apply** *blast*
apply (*rule trace-hide-map-map-event*$_{ptick}$[*OF inj-on-map-event*$_{ptick}$])
using ∗∗∗∗(*4*) *mem-D-imp-mem-D-Hiding* **by** *blast*
thus ‹*s* ∈ 𝒟 *?lhs*› **by** (*simp add*: *D-Renaming* ∗(*3*) ∗∗∗∗(*3*)) *blast*
qed
qed
next
fix *s X*
assume *same-div* : ‹𝒟 *?lhs* = 𝒟 *?rhs*›
assume ‹(*s*, *X*) ∈ ℱ *?lhs*›
then consider ‹*s* ∈ 𝒟 *?lhs*›
| *s1* **where** ‹(*s1*, *map-event*$_{ptick}$ *f g* −‘ *X*) ∈ ℱ (*P* \ *S*)› ‹*s* = *map*

(*map-event*$_{ptick}$ *f g*) *s1*›
by (*simp add*: *F-Renaming D-Renaming*) *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
fix *s1* **assume** ∗ : ‹(*s1*, *map-event*$_{ptick}$ *f g* −‘ *X*) ∈ $\mathcal{F}$ (*P* \ *S*)›
‹*s* = *map* (*map-event*$_{ptick}$ *f g*) *s1*›
from *this*(*1*) **consider** ‹*s1* ∈ $\mathcal{D}$ (*P* \ *S*)›
| *t* **where** ‹*s1* = *trace-hide t* (*ev* ‘ *S*)› ‹(*t*, *map-event*$_{ptick}$ *f g* −‘ *X* ∪ *ev* ‘ *S*) ∈ $\mathcal{F}$ *P*›
by (*simp add*: *F-Hiding D-Hiding*) *blast*
thus ‹(*s*, *X*) ∈ $\mathcal{F}$ *?rhs*›
proof *cases*
assume ‹*s1* ∈ $\mathcal{D}$ (*P* \ *S*)›
then obtain *t u*
where ∗∗ : ‹*front-tickFree u*› ‹*tickFree t*› ‹*s1* = *trace-hide t* (*ev* ‘ *S*) @ *u*›
‹*t* ∈ $\mathcal{D}$ *P* ∨ (∃ *g*. *isInfHiddenRun g P S* ∧ *t* ∈ *range g*)›
by (*simp add*: *D-Hiding*) *blast*
have ∗∗∗ : ‹*front-tickFree* (*map* (*map-event*$_{ptick}$ *f g*) *u*) ∧ *tickFree* (*map* (*map-event*$_{ptick}$ *f g*) *t*) ∧
map (*map-event*$_{ptick}$ *f g*) (*trace-hide t* (*ev* ‘ *S*)) @ *map* (*map-event*$_{ptick}$ *f g*) *u* =
trace-hide (*map* (*map-event*$_{ptick}$ *f g*) *t*) (*ev* ‘ *f* ‘ *S*) @ (*map* (*map-event*$_{ptick}$ *f g*) *u*)›
by (*simp add*: *map-event*$_{ptick}$*-front-tickFree map-event*$_{ptick}$*-tickFree* ∗∗(*1*, *2*))
(*rule trace-hide-map-map-event*$_{ptick}$[*OF inj-on-map-event*$_{ptick}$, *symmetric*])
from ∗∗(*4*) **show** ‹(*s*, *X*) ∈ $\mathcal{F}$ *?rhs*›
proof (*elim disjE exE*)
assume ‹*t* ∈ $\mathcal{D}$ *P*›
hence $: ‹*map* (*map-event*$_{ptick}$ *f g*) *t* ∈ $\mathcal{D}$ (*Renaming P f g*)›
apply (*simp add*: *D-Renaming*)
using ∗∗(*2*) *front-tickFree-Nil* **by** *blast*
show ‹(*s*, *X*) ∈ $\mathcal{F}$ *?rhs*›
by (*simp add*: *F-Hiding*) (*metis* $ ∗(*2*) ∗∗(*3*) ∗∗∗ *map-append*)
next
fix *h* **assume** ‹*isInfHiddenRun h P S* ∧ *t* ∈ *range h*›
hence $: ‹*isInfHiddenRun* (λ*i*. *map* (*map-event*$_{ptick}$ *f g*) (*h i*)) (*Renaming P f g*) (*f* ‘ *S*) ∧
map (*map-event*$_{ptick}$ *f g*) *t* ∈ *range* (λ*i*. *map* (*map-event*$_{ptick}$ *f g*) (*h i*))›
apply (*subst* (*1 2*) *trace-hide-map-map-event*$_{ptick}$[*OF inj-on-map-event*$_{ptick}$])
by (*simp add*: *strict-mono-map T-Renaming image-iff*) (*metis* (*mono-tags*, *lifting*))
show ‹(*s*, *X*) ∈ $\mathcal{F}$ *?rhs*›
apply (*simp add*: *F-Hiding*)

by (*smt* (*verit*, *del-insts*) $ ∗(*2*) ∗∗(*3*) ∗∗∗ *map-append*)
qed
next
fix *t* **assume** ∗∗ : ‹*s1* = *trace-hide t* (*ev* ‘ *S*)›
‹(*t*, *map-event*$_{ptick}$ *f g* −‘ *X* ∪ *ev* ‘ *S*) ∈ $\mathcal{F}$ *P*›
have ∗∗∗ : ‹*map-event*$_{ptick}$ *f g* −‘ *X* ∪ *map-event*$_{ptick}$ *f g* −‘ *ev* ‘ *f* ‘ *S* = *map-event*$_{ptick}$ *f g* −‘ *X* ∪ *ev* ‘ *S*›
by (*auto simp add*: *image-iff map-event*$_{ptick}$*-eq-ev-iff*) (*metis bij-f bij-pointE*)
have ‹*map* (*map-event*$_{ptick}$ *f g*) (*trace-hide t* (*ev* ‘ *S*)) =
trace-hide (*map* (*map-event*$_{ptick}$ *f g*) *t*) (*ev* ‘ *f* ‘ *S*) ∧
(*map* (*map-event*$_{ptick}$ *f g*) *t*, *X* ∪ *ev* ‘ *f* ‘ *S*) ∈ $\mathcal{F}$ (*Renaming P f g*)›
apply (*intro conjI*)
apply (*rule trace-hide-map-map-event*$_{ptick}$[*OF inj-on-map-event*$_{ptick}$, *symmetric*])
apply (*simp add*: *F-Renaming*)
using ∗∗(*2*) ∗∗∗ **by** *auto*
show ‹(*s*, *X*) ∈ $\mathcal{F}$ *?rhs*›
apply (*simp add*: *F-Hiding* ∗(*2*) ∗∗(*1*))
using ‹*?this*› **by** *blast*
qed
qed
next
fix *s X*
assume *same-div* : ‹$\mathcal{D}$ *?lhs* = $\mathcal{D}$ *?rhs*›
assume ‹(*s*, *X*) ∈ $\mathcal{F}$ *?rhs*›
then consider ‹*s* ∈ $\mathcal{D}$ *?rhs*›
| *t* **where** ‹*s* = *trace-hide t* (*ev* ‘ *f* ‘ *S*)› ‹(*t*, *X* ∪ *ev* ‘ *f* ‘ *S*) ∈ $\mathcal{F}$ (*Renaming P f g*)›
by (*simp add*: *F-Hiding D-Hiding*) *blast*
thus ‹(*s*, *X*) ∈ $\mathcal{F}$ *?lhs*›
proof *cases*
from *D-F same-div* **show** ‹*s* ∈ $\mathcal{D}$ *?rhs* ⟹ (*s*, *X*) ∈ $\mathcal{F}$ *?lhs*› **by** *blast*
next
fix *t* **assume** ‹*s* = *trace-hide t* (*ev* ‘ *f* ‘ *S*)› ‹(*t*, *X* ∪ *ev* ‘ *f* ‘ *S*) ∈ $\mathcal{F}$ (*Renaming P f g*)›
then obtain *t*
where ∗ : ‹*s* = *trace-hide t* (*ev* ‘ *f* ‘ *S*)›
‹(*t*, *X* ∪ *ev* ‘ *f* ‘ *S*) ∈ $\mathcal{F}$ (*Renaming P f g*)› **by** *blast*
have ∗∗ : ‹*map-event*$_{ptick}$ *f g* −‘ *X* ∪ *map-event*$_{ptick}$ *f g* −‘ *ev* ‘ *f* ‘ *S* = *map-event*$_{ptick}$ *f g* −‘ *X* ∪ *ev* ‘ *S*›
by (*auto simp add*: *image-iff map-event*$_{ptick}$*-eq-ev-iff*) (*metis bij-f bij-pointE*)
have ‹(∃ *s1*. (*s1*, *map-event*$_{ptick}$ *f g* −‘ *X* ∪ *map-event*$_{ptick}$ *f g* −‘ *ev* ‘ *f* ‘ *S*) ∈ $\mathcal{F}$ *P* ∧ *t* = *map* (*map-event*$_{ptick}$ *f g*) *s1*) ∨
(∃ *s1 s2*. *tickFree s1* ∧ *front-tickFree s2* ∧ *t* = *map* (*map-event*$_{ptick}$ *f g*) *s1* @ *s2* ∧ *s1* ∈ $\mathcal{D}$ *P*)›
using ∗(*2*) **by** (*auto simp add*: *F-Renaming*)
thus ‹(*s*, *X*) ∈ $\mathcal{F}$ *?lhs*›
proof (*elim disjE exE conjE*)

fix *s1*
assume ‹(*s1*, *map-event*$_{ptick}$ *f g* −‘ *X* ∪ *map-event*$_{ptick}$ *f g* −‘ *ev* ‘ *f* ‘ *S*) ∈ $\mathcal{F}$ *P*› ‹*t* = *map* (*map-event*$_{ptick}$ *f g*) *s1*›
hence ‹(*trace-hide s1* (*ev* ‘ *S*), *map-event*$_{ptick}$ *f g* −‘ *X*) ∈ $\mathcal{F}$ (*P* \ *S*) ∧
s = *map* (*map-event*$_{ptick}$ *f g*) (*trace-hide s1* (*ev* ‘ *S*))›
apply (*simp add*: ∗(*1*) *F-Hiding* ∗∗, *intro conjI*)
by *blast* (*rule trace-hide-map-map-event*$_{ptick}$[*OF inj-on-map-event*$_{ptick}$])
show ‹(*s*, *X*) ∈ $\mathcal{F}$ *?lhs*›
apply (*simp add*: *F-Renaming*)
using ‹*?this*› **by** *blast*
next
fix *s1 s2*
assume ‹*tickFree s1*› ‹*front-tickFree s2*› ‹*t* = *map* (*map-event*$_{ptick}$ *f g*) *s1* @ *s2*› ‹*s1* ∈ $\mathcal{D}$ *P*›
hence ‹*tickFree* (*trace-hide s1* (*ev* ‘ *S*)) ∧
front-tickFree (*trace-hide s2* (*ev* ‘ *f* ‘ *S*)) ∧
s = *map* (*map-event*$_{ptick}$ *f g*) (*trace-hide s1* (*ev* ‘ *S*)) @ *trace-hide s2* (*ev* ‘ *f* ‘ *S*) ∧
trace-hide s1 (*ev* ‘ *S*) ∈ $\mathcal{D}$ (*P* \ *S*)›
apply (*simp add*: *F-Renaming* ∗(*1*), *intro conjI*)
using *Hiding-tickFree* **apply** *blast*
using *Hiding-front-tickFree* **apply** *blast*
apply (*rule trace-hide-map-map-event*$_{ptick}$[*OF inj-on-map-event*$_{ptick}$])
using *mem-D-imp-mem-D-Hiding* **by** *blast*
show ‹(*s*, *X*) ∈ $\mathcal{F}$ *?lhs*›
apply (*simp add*: *F-Renaming*)
using ‹*?this*› **by** *blast*
qed
qed
qed
qed

4.2.2 *Renaming* and *Sync*

Idem for the synchronization: when *f* is one to one, *Renaming* (*P* ⟦*S*⟧ *Q*) will behave as expected.

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 ‹$f\ y \notin f$ ‘ S› **by** (*metis bij-betw-imp-inj-on inj-image-mem-iff bij-f*)
with *2.hyps*[*OF* ‹$y \notin S$›, *of* ‹*tl r*›] **show** *?case*
by (*cases r*; *simp add*: ‹$y \notin S$›) (*metis bij-pointE bij-f*)
qed
next
case (*3 x t*)
show *?case*
proof (*cases* ‹$x \in S$›)
show ‹$x \in S \Longrightarrow$ *?case*› **by** *simp*
next
assume ‹$x \notin S$›
hence ‹$f\ x \notin f$ ‘ S› **by** (*metis bij-betw-imp-inj-on inj-image-mem-iff bij-f*)
with *3.hyps*[*OF* ‹$x \notin S$›, *of* ‹*tl r*›] **show** *?case*
by (*cases r*; *simp add*: ‹$x \notin S$›) (*metis bij-pointE bij-f*)
qed
next
case (*4 x t y u*)
have $*$: ‹$x \neq y \Longrightarrow f\ x \neq f\ y$› **by** (*metis bij-pointE bij-f*)
have $**$: ‹$f\ z \in f$ ‘ $S \longleftrightarrow z \in S$› **for** z
by (*meson bij-betw-def inj-image-mem-iff bij-f*)
show *?case*
proof (*cases* ‹$x \in S$›; *cases* ‹$y \in S$›)
from *4.hyps*(*1*)[*of* ‹*tl r*›] **show** ‹$x \in S \Longrightarrow y \in S \Longrightarrow$ *?case*›
by (*cases r*; *simp add*: $*$) (*metis bij-pointE bij-f*)
next
from *4.hyps*(*2*)[*of* ‹*tl r*›] **show** ‹$x \in S \Longrightarrow y \notin S \Longrightarrow$ *?case*›
by (*cases r*; *simp add*: $**$) (*metis bij-pointE bij-f*)
next
from *4.hyps*(*5*)[*of* ‹*tl r*›] **show** ‹$x \notin S \Longrightarrow y \in S \Longrightarrow$ *?case*›
by (*cases r*; *simp add*: $**$) (*metis bij-pointE bij-f*)
next
from *4.hyps*(*3*, *4*)[*of* ‹*tl r*›] **show** ‹$x \notin S \Longrightarrow y \notin S \Longrightarrow$ *?case*›
by (*cases r*; *simp add*: $**$) (*metis bij-pointE bij-f*)
qed
qed

theorem *bij-Renaming-Sync*:
‹*Renaming* (*P* ⟦*S*⟧ *Q*) *f g* = *Renaming P f g* ⟦*f* ‘ *S*⟧ *Renaming Q f g*›
(**is** ‹*?lhs P Q* = *?rhs P Q*›) **if** *bij-f*: ‹*bij f*› **and** *bij-g* : ‹*bij g*›
proof −
— Four intermediate results.
have *bij-map-event*$_{ptick}$: ‹*bij* (*map-event*$_{ptick}$ *f g*)›
proof (*rule bijI*)
show ‹*inj* (*map-event*$_{ptick}$ *f g*)›
proof (*rule injI*)
show ‹*map-event*$_{ptick}$ *f g e* = *map-event*$_{ptick}$ *f g e*$'$ $\Longrightarrow e = e'$› **for** $e\ e'$
by (*cases e*; *cases e*$'$; *simp*)
(*metis bij-f bij-pointE*, *metis bij-g bij-pointE*)

qed
next
show ‹*surj* (*map-event*$_{ptick}$ *f g*)›
proof (*rule surjI*)
show ‹*map-event*$_{ptick}$ *f g* (*map-event*$_{ptick}$ (*inv f*) (*inv g*) *e*) = *e*› **for** *e*
by (*cases e*, *simp-all*)
(*meson bij-f bij-inv-eq-iff*, *meson bij-g bij-inv-eq-iff*)
qed
qed
moreover have ‹*map-event*$_{ptick}$ (*inv f*) (*inv g*) ∘ *map-event*$_{ptick}$ *f g* = *id*›
proof (*rule ext*)
show ‹(*map-event*$_{ptick}$ (*inv f*) (*inv g*) ∘ *map-event*$_{ptick}$ *f g*) *e* = *id e*› **for** *e*
by (*cases e*, *simp-all*)
(*meson bij-betw-def bij-f inv-f-eq*, *meson bij-betw-def bij-g inv-f-eq*)
qed
ultimately have *inv-map-event*$_{ptick}$*-is-map-event*$_{ptick}$*-inv* :
‹*inv* (*map-event*$_{ptick}$ *f g*) = *map-event*$_{ptick}$ (*inv f*) (*inv g*)›
by (*metis bij-betw-imp-inj-on bij-betw-imp-surj-on inv-o-cancel surj-fun-eq*)
have *sets-S-eq* : ‹*map-event*$_{ptick}$ *f g* ‘ (*range tick* ∪ *ev* ‘ *S*) = *range tick* ∪ *ev* ‘ *f* ‘ *S*›
by (*auto simp add*: *image-iff*)
(*metis Un-iff bij-g bij-pointE event*$_{ptick}$*.simps*(*10*) *rangeI*,
metis Un-iff event$_{ptick}$*.simps*(*9*) *imageI*)
have *inj-map-event*$_{ptick}$: ‹*inj* (*map-event*$_{ptick}$ *f g*)›
and *inj-inv-map-event*$_{ptick}$: ‹*inj* (*inv* (*map-event*$_{ptick}$ *f g*))›
by (*use bij-betw-imp-inj-on bij-map-event*$_{ptick}$ **in** *blast*)
(*meson bij-betw-imp-inj-on bij-betw-inv-into bij-map-event*$_{ptick}$)
show ‹*?lhs P Q* = *?rhs P Q*›
proof (*subst Process-eq-spec-optimized*, *safe*)
fix *s*
assume ‹*s* ∈ 𝒟 (*?lhs P Q*)›
then obtain *s1 s2* **where** ∗ : ‹*tickFree s1*› ‹*front-tickFree s2*›
‹*s* = *map* (*map-event*$_{ptick}$ *f g*) *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*), *range 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*), *range tick* ∪ *ev* ‘ *S*)›
‹*t* ∈ 𝒟 *P*› ‹*u* ∈ 𝒯 *Q*›
have ‹*map* (*map-event*$_{ptick}$ *f g*) *r setinterleaves*
((*map* (*map-event*$_{ptick}$ *f g*) *t*, *map* (*map-event*$_{ptick}$ *f g*) *u*), *range tick* ∪ *ev* ‘ *f* ‘ *S*)›
by (*metis assms*(*1*) *bij-map-setinterleaving-iff-setinterleaving bij-map-event*$_{ptick}$ *sets-S-eq*)
moreover have ‹*map* (*map-event*$_{ptick}$ *f g*) *t* ∈ 𝒟 (*Renaming P f g*)›
apply (*cases* ‹*tickFree t*›; *simp add*: *D-Renaming*)

using *assms*(*2*) *front-tickFree-Nil* **apply** *blast*
by (*metis D-T D-imp-front-tickFree append-T-imp-tickFree assms*(*2*) *front-tickFree-Cons-iff is-processT9 list.simps*(*3*) *map-append nonTickFree-n-frontTickFree map-event$_{ptick}$-front-tickFree*)
moreover have ‹*map* (*map-event$_{ptick}$ f g*) *u* ∈ $\mathcal{T}$ (*Renaming Q f g*)›
using *assms*(*3*) **by** (*simp add*: *T-Renaming*) *blast*
ultimately have ‹*s* ∈ $\mathcal{D}$ (*?rhs P Q*)›
by (*simp add*: *D-Sync* ∗(*3*) ∗∗(*3*))
(*metis* ∗(*1*, *2*) ∗∗(*3*) *map-event$_{ptick}$-tickFree front-tickFree-append tickFree-append-iff*)
} **note** ∗∗∗ = *this*

from ∗∗(*4*, *5*) ∗∗∗ **show** ‹*s* ∈ $\mathcal{D}$ (*?rhs P Q*)›
apply (*elim disjE*)
using ∗∗(*4*) ∗∗∗ **apply** *blast*
using ∗∗(*4*) ∗∗∗ **by** (*subst Sync-commute*) *blast*
next
fix *s*
assume ‹*s* ∈ $\mathcal{D}$ (*?rhs P Q*)›
then obtain *t u r v*
where ∗ : ‹*front-tickFree v*› ‹*tickFree r* ∨ *v* = []› ‹*s* = *r* @ *v*›
‹*r setinterleaves* ((*t*, *u*), *range tick* ∪ *ev* ‘ *f* ‘ *S*)›
‹*t* ∈ $\mathcal{D}$ (*Renaming P f g*) ∧ *u* ∈ $\mathcal{T}$ (*Renaming Q f g*) ∨
t ∈ $\mathcal{D}$ (*Renaming Q f g*) ∧ *u* ∈ $\mathcal{T}$ (*Renaming P f g*)›
by (*simp add*: *D-Sync*) *blast*

{ **fix** *t u P Q*
assume *assms* : ‹*r setinterleaves* ((*t*, *u*), *range tick* ∪ *ev* ‘ *f* ‘ *S*)›
‹*t* ∈ $\mathcal{D}$ (*Renaming P f g*)› ‹*u* ∈ $\mathcal{T}$ (*Renaming Q f g*)›
have ‹*inv* (*map-event$_{ptick}$ f g*) ‘ (*range tick* ∪ *ev* ‘ *f* ‘ *S*) =
inv (*map-event$_{ptick}$ f g*) ‘ *map-event$_{ptick}$ f g* ‘ (*range tick* ∪ *ev* ‘ *S*)›
using *sets-S-eq* **by** *presburger*
from *bij-map-setinterleaving-iff-setinterleaving*
[*THEN iffD2*, *OF* - *assms*(*1*), *of* ‹*inv* (*map-event$_{ptick}$ f g*)›,
simplified this image-inv-f-f[*OF inj-map-event$_{ptick}$*]]
have ∗∗ : ‹(*map* (*inv* (*map-event$_{ptick}$ f g*)) *r*) *setinterleaves*
((*map* (*inv* (*map-event$_{ptick}$ f g*)) *t*, *map* (*inv* (*map-event$_{ptick}$ f g*)) *u*), *range tick* ∪ *ev* ‘ *S*)›
using *bij-betw-inv-into bij-map-event$_{ptick}$* **by** *blast*
from *assms*(*2*) **obtain** *s1 s2*
where ‹*t* = *map* (*map-event$_{ptick}$ f g*) *s1* @ *s2*› ‹*tickFree s1*› ‹*front-tickFree s2*› ‹*s1* ∈ $\mathcal{D}$ *P*›
by (*auto simp add*: *D-Renaming*)
hence ‹*map* (*map-event$_{ptick}$* (*inv f*) (*inv g*)) *t* ∈ $\mathcal{D}$ (*Renaming* (*Renaming P f g*) (*inv f*) (*inv g*))›
apply (*simp add*: *D-Renaming*)
apply (*rule-tac x* = ‹*map* (*map-event$_{ptick}$ f g*) *s1*› **in** *exI*)
apply (*rule-tac x* = ‹*map* (*map-event$_{ptick}$* (*inv f*) (*inv g*)) *s2*› **in** *exI*)
by *simp* (*metis append-Nil2 front-tickFree-Nil map-event$_{ptick}$-front-tickFree*

map-event$_{ptick}$-tickFree)
hence ∗∗∗ : ‹*map* (*inv* (*map-event$_{ptick}$ f g*)) *t* ∈ $\mathcal{D}$ *P*›
by (*metis Renaming-inv bij-def bij-f bij-g inv-map-event$_{ptick}$-is-map-event$_{ptick}$-inv*)
have ‹*map* (*map-event$_{ptick}$* (*inv f*) (*inv g*)) *u* ∈ $\mathcal{T}$ (*Renaming* (*Renaming Q f g*) (*inv f*) (*inv g*))›
using *assms*(*3*) **by** (*subst T-Renaming*, *simp*) *blast*
hence ∗∗∗∗ : ‹*map* (*inv* (*map-event$_{ptick}$ f g*)) *u* ∈ $\mathcal{T}$ *Q*›
by (*simp add*: *Renaming-inv bij-f bij-g bij-is-inj inv-map-event$_{ptick}$-is-map-event$_{ptick}$-inv*)
have ∗∗∗∗∗ : ‹*map* (*map-event$_{ptick}$ f g* ∘ *inv* (*map-event$_{ptick}$ f g*)) *r* = *r*›
by (*metis* (*no-types*, *lifting*) *bij-betw-imp-inj-on bij-betw-inv-into bij-map-event$_{ptick}$ inj-iff list.map-comp list.map-id*)
have ‹*s* ∈ $\mathcal{D}$ (*?lhs P Q*)›
proof (*cases* ‹*tickFree r*›)
assume ‹*tickFree r*›
have \$: ‹*r* @ *v* = *map* (*map-event$_{ptick}$ f g*) (*map* (*inv* (*map-event$_{ptick}$ f g*)) *r*) @ *v*›
by (*simp add*: ∗∗∗∗∗)
show ‹*s* ∈ $\mathcal{D}$ (*?lhs P Q*)›
apply (*simp add*: *D-Renaming D-Sync* ∗(*3*))
by (*metis* \$ ∗(*1*) ∗∗ ∗∗∗ ∗∗∗∗ *map-event$_{ptick}$-tickFree* ‹*tickFree r*› *append.right-neutral append-same-eq front-tickFree-Nil*)
next
assume ‹¬ *tickFree r*›
then obtain *r′ res* **where** \$: ‹*r* = *r′* @ [✓(*res*)]› ‹*tickFree r′*›
by (*metis D-imp-front-tickFree assms butlast-snoc front-tickFree-charn front-tickFree-single ftf-Sync is-processT2-TR list.distinct*(*1*) *nonTickFree-n-frontTickFree self-append-conv2*)
then obtain *t′ u′*
where \$\$: ‹*t* = *t′* @ [✓(*res*)]› ‹*u* = *u′* @ [✓(*res*)]›
by (*metis D-imp-front-tickFree SyncWithTick-imp-NTF T-imp-front-tickFree assms*)
hence \$\$\$: ‹(*map* (*inv* (*map-event$_{ptick}$ f g*)) *r′*) *setinterleaves* ((*map* (*inv* (*map-event$_{ptick}$ f g*)) *t′*, *map* (*inv* (*map-event$_{ptick}$ f g*)) *u′*), *range tick* ∪ *ev* ‘ *S*)›
by (*metis* \$(*1*) *append-same-eq assms*(*1*) *bij-betw-imp-surj-on bij-map-setinterleaving-iff-setinterleaving bij-map-event$_{ptick}$ ftf-Sync32 inj-imp-bij-betw-inv inj-map-event$_{ptick}$ sets-S-eq*)
from ∗∗∗ \$\$(*1*) **have** ∗∗∗ : ‹*map* (*inv* (*map-event$_{ptick}$ f g*)) *t′* ∈ $\mathcal{D}$ *P*›
by *simp* (*use inv-map-event$_{ptick}$-is-map-event$_{ptick}$-inv is-processT9* **in** *force*)
from ∗∗∗∗ \$\$(*2*) **have** ∗∗∗∗ : ‹*map* (*inv* (*map-event$_{ptick}$ f g*)) *u′* ∈ $\mathcal{T}$ *Q*›
using *is-processT3-TR prefixI* **by** *simp blast*
have \$\$\$\$: ‹*r* = *map* (*map-event$_{ptick}$ f g*) (*map* (*inv* (*map-event$_{ptick}$ f g*)) *r′*) @ [✓(*res*)]›
using \$ ∗∗∗∗∗ **by** *auto*
show ‹*s* ∈ $\mathcal{D}$ (*?lhs P Q*)›
by (*simp add*: *D-Renaming D-Sync* ∗(*3*) \$\$\$)
(*metis* \$(*1*) \$(*2*) \$\$\$ \$\$\$\$ ∗(*2*) ∗∗∗ ∗∗∗∗ *map-event$_{ptick}$-tickFree* ‹¬

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*) ∈ $\mathcal{F}$ (*?lhs P Q*)›
then consider ‹$s \in \mathcal{D}$ (*?lhs P Q*)›
| *s1* **where** ‹(*s1*, *map-event*$_{ptick}$ *f g* −‘ *X*) ∈ $\mathcal{F}$ (*P* ⟦*S*⟧ *Q*)› ‹*s* = *map* (*map-event*$_{ptick}$ *f g*) *s1*›
by (*simp add*: *F-Renaming D-Renaming*) *blast*
thus ‹(*s*, *X*) ∈ $\mathcal{F}$ (*?rhs P Q*)›
proof *cases*
from *same-div D-F* **show** ‹$s \in \mathcal{D}$ (*?lhs P Q*) ⟹ (*s*, *X*) ∈ $\mathcal{F}$ (*?rhs P Q*)› **by** *blast*
next
fix *s1* **assume** ∗ : ‹(*s1*, *map-event*$_{ptick}$ *f g* −‘ *X*) ∈ $\mathcal{F}$ (*P* ⟦*S*⟧ *Q*)›
‹*s* = *map* (*map-event*$_{ptick}$ *f g*) *s1*›
from ∗(*1*) **consider** ‹*s1* ∈ $\mathcal{D}$ (*P* ⟦*S*⟧ *Q*)›
| *t-P t-Q X-P X-Q*
where ‹(*t-P*, *X-P*) ∈ $\mathcal{F}$ *P*› ‹(*t-Q*, *X-Q*) ∈ $\mathcal{F}$ *Q*›
‹*s1 setinterleaves* ((*t-P*, *t-Q*), *range tick* ∪ *ev* ‘ *S*)›
‹*map-event*$_{ptick}$ *f g* −‘ *X* = (*X-P* ∪ *X-Q*) ∩ (*range tick* ∪ *ev* ‘ *S*) ∪ *X-P* ∩ *X-Q*›
by (*auto simp add*: *F-Sync D-Sync*)
thus ‹(*s*, *X*) ∈ $\mathcal{F}$ (*?rhs P Q*)›
proof *cases*
assume ‹*s1* ∈ $\mathcal{D}$ (*P* ⟦*S*⟧ *Q*)›
hence ‹$s \in \mathcal{D}$ (*?lhs P Q*)›
apply (*cases* ‹*tickFree s1*›; *simp add*: *D-Renaming* ∗(*2*))
using *front-tickFree-Nil* **apply** *blast*
by (*metis* (*no-types*, *lifting*) *map-event*$_{ptick}$*-front-tickFree butlast-snoc front-tickFree-iff-tickFree-butlast*
front-tickFree-single map-butlast nonTickFree-n-frontTickFree process-charn)
with *same-div D-F* **show** ‹(*s*, *X*) ∈ $\mathcal{F}$ (*?rhs P Q*)› **by** *blast*
next
fix *t-P t-Q X-P X-Q*
assume ∗∗ : ‹(*t-P*, *X-P*) ∈ $\mathcal{F}$ *P*› ‹(*t-Q*, *X-Q*) ∈ $\mathcal{F}$ *Q*›
‹*s1 setinterleaves* ((*t-P*, *t-Q*), *range tick* ∪ *ev* ‘ *S*)›
‹*map-event*$_{ptick}$ *f g* −‘ *X* = (*X-P* ∪ *X-Q*) ∩ (*range tick* ∪ *ev* ‘ *S*) ∪ *X-P* ∩ *X-Q*›
have ‹(*map* (*map-event*$_{ptick}$ *f g*) *t-P*, (*map-event*$_{ptick}$ *f g*) ‘ *X-P*) ∈ $\mathcal{F}$ (*Renaming P f g*)›
by (*simp add*: *F-Renaming*)
(*metis* ∗∗(*1*) *bij-betw-def bij-map-event*$_{ptick}$ *inj-vimage-image-eq*)
moreover have ‹(*map* (*map-event*$_{ptick}$ *f g*) *t-Q*, (*map-event*$_{ptick}$ *f g*) ‘

X-Q) ∈ ℱ (Renaming Q f g)›
by (simp add: F-Renaming)
(metis **(2) bij-betw-imp-inj-on bij-map-event$_{ptick}$ inj-vimage-image-eq)
moreover have ‹s setinterleaves ((map (map-event$_{ptick}$ f g) t-P, map (map-event$_{ptick}$ f g) t-Q),
range tick ∪ ev ‘ f ‘ S)›
by (metis *(2) **(3) bij-map-event$_{ptick}$ sets-S-eq
bij-map-setinterleaving-iff-setinterleaving)
moreover have ‹X = ((map-event$_{ptick}$ f g) ‘ X-P ∪ (map-event$_{ptick}$ f g) ‘ X-Q) ∩ (range tick ∪ ev ‘ f ‘ S) ∪
(map-event$_{ptick}$ f g) ‘ X-P ∩ (map-event$_{ptick}$ f g) ‘ X-Q›
apply (rule inj-image-eq-iff[THEN iffD1, OF inj-inv-map-event$_{ptick}$])
apply (subst bij-vimage-eq-inv-image[OF bij-map-event$_{ptick}$, symmetric])
apply (subst **(4), fold image-Un sets-S-eq)
apply (subst (1 2) image-Int[OF inj-map-event$_{ptick}$, symmetric])
apply (fold image-Un)
apply (subst image-inv-f-f[OF inj-map-event$_{ptick}$])
by simp
ultimately show ‹(s, X) ∈ 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 **where**
‹(t-P, X-P) ∈ ℱ (Renaming P f g)› ‹(t-Q, X-Q) ∈ ℱ (Renaming Q f g)›
‹s setinterleaves ((t-P, t-Q), range tick ∪ ev ‘ f ‘ S)›
‹X = (X-P ∪ X-Q) ∩ (range tick ∪ ev ‘ f ‘ S) ∪ X-P ∩ X-Q›
by (simp add: F-Sync D-Sync) blast
thus ‹(s, X) ∈ 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
fix t-P t-Q X-P X-Q
assume * : ‹(t-P, X-P) ∈ ℱ (Renaming P f g)› ‹(t-Q, X-Q) ∈ ℱ (Renaming Q f g)›
‹s setinterleaves ((t-P, t-Q), range tick ∪ ev ‘ f ‘ S)›
‹X = (X-P ∪ X-Q) ∩ (range tick ∪ ev ‘ f ‘ S) ∪ X-P ∩ X-Q›
from *(1, 2) **consider** ‹t-P ∈ 𝒟 (Renaming P f g) ∨ t-Q ∈ 𝒟 (Renaming Q f g)›
| t-P1 t-Q1 **where** ‹(t-P1, map-event$_{ptick}$ f g –‘ X-P) ∈ ℱ P› ‹t-P = map (map-event$_{ptick}$ f g) t-P1›
‹(t-Q1, map-event$_{ptick}$ f g –‘ X-Q) ∈ ℱ Q› ‹t-Q = map (map-event$_{ptick}$ f g) t-Q1›
by (simp add: F-Renaming D-Renaming) blast

thus ‹$(s, X) \in \mathcal{F}$ (*?lhs P Q*)›
proof *cases*
assume ‹*t-P* $\in \mathcal{D}$ (*Renaming P f g*) $\vee$ *t-Q* $\in \mathcal{D}$ (*Renaming Q f g*)›
hence ‹$s \in \mathcal{D}$ (*?rhs P Q*)›
apply (*simp add*: *D-Sync*)
using $*$(*1*, *2*, *3*) *F-T setinterleaving-sym front-tickFree-Nil* **by** *blast*
with *same-div D-F* **show** ‹$(s, X) \in \mathcal{F}$ (*?lhs P Q*)› **by** *blast*
next
fix *t-P1 t-Q1*
assume $**$: ‹(*t-P1*, *map-event*$_{ptick}$ *f g* −‘ *X-P*) $\in \mathcal{F}$ *P*› ‹*t-P* = *map* (*map-event*$_{ptick}$ *f g*) *t-P1*›
‹(*t-Q1*, *map-event*$_{ptick}$ *f g* −‘ *X-Q*) $\in \mathcal{F}$ *Q*› ‹*t-Q* = *map* (*map-event*$_{ptick}$ *f g*) *t-Q1*›
from $**$(*2*, *4*) **have** $***$: ‹*t-P1* = *map* (*inv* (*map-event*$_{ptick}$ *f g*)) *t-P*›
‹*t-Q1* = *map* (*inv* (*map-event*$_{ptick}$ *f g*)) *t-Q*›
by (*simp-all add*: *inj-map-event*$_{ptick}$)
have $****$: ‹*map* (*map-event*$_{ptick}$ *f g*) (*map* (*inv* (*map-event*$_{ptick}$ *f g*)) *s*) = *s*›
by (*metis bij-betw-imp-surj bij-map-event*$_{ptick}$ *list.map-comp list.map-id surj-iff*)
from *bij-map-setinterleaving-iff-setinterleaving*
[*of* ‹*inv* (*map-event*$_{ptick}$ *f g*)› *s t-P* ‹*range tick* $\cup$ *ev* ‘ *f* ‘ *S*› *t-Q*, *simplified* $*$(*3*)]
have ‹*map* (*inv* (*map-event*$_{ptick}$ *f g*)) *s setinterleaves* ((*t-P1*, *t-Q1*), *range tick* $\cup$ *ev* ‘ *S*)›
by (*metis* $***$ *bij-betw-def bij-map-event*$_{ptick}$ *inj-imp-bij-betw-inv sets-S-eq*)
moreover have ‹*map-event*$_{ptick}$ *f g* −‘ *X* = (*map-event*$_{ptick}$ *f g* −‘ *X-P* $\cup$ *map-event*$_{ptick}$ *f g* −‘ *X-Q*) $\cap$ (*range tick* $\cup$ *ev* ‘ *S*) $\cup$
map-event$_{ptick}$ *f g* −‘ *X-P* $\cap$ *map-event*$_{ptick}$ *f g* −‘ *X-Q*›
by (*metis* (*no-types*, *lifting*) $*$(*4*) *inj-map-event*$_{ptick}$ *inj-vimage-image-eq sets-S-eq vimage-Int vimage-Un*)
ultimately show ‹$(s, X) \in \mathcal{F}$ (*?lhs P Q*)›
by (*simp add*: *F-Renaming F-Sync*) (*metis* $**$(*1*, *3*) $****$)
qed
qed
qed
qed

end

Chapter 5

Results on *events-of* and *ticks-of*

5.1 Events

lemma *events-of-GlobalDet* :
‹$\alpha(\square a \in A.\ P\ a) = (\bigcup a{\in}A.\ \alpha(P\ a))$›
by (*simp add*: *events-of-def T-GlobalDet*)

lemma *strict-events-of-GlobalDet-subset* : ‹$\boldsymbol{\alpha}(\square a \in A.\ P\ a) \subseteq (\bigcup a{\in}A.\ \boldsymbol{\alpha}(P\ a))$›
by (*auto simp add*: *strict-events-of-def GlobalDet-projs*)

lemma *events-of-MultiSync-subset* :
‹$\alpha([\![S]\!]\ a \in\#\ M.\ P\ a) \subseteq (\bigcup a \in \textit{set-mset}\ M.\ \alpha(P\ a))$›
by (*induct M rule*: *induct-subset-mset-empty-single*, *simp-all*)
(*meson Diff-subset-conv dual-order.trans events-of-Sync-subset*)

lemma *events-of-MultiInter* :
‹$\alpha(|||\ a \in\#\ M.\ P\ a) = (\bigcup a \in \textit{set-mset}\ M.\ \alpha(P\ a))$›
by (*induct M rule*: *induct-subset-mset-empty-single*)
(*simp-all add*: *events-of-Inter*)

lemma *strict-events-of-MultiSync-subset* :
‹$\boldsymbol{\alpha}([\![S]\!]\ a \in\#\ M.\ P\ a) \subseteq (\bigcup a \in \textit{set-mset}\ M.\ \boldsymbol{\alpha}(P\ a))$›
by (*induct M rule*: *induct-subset-mset-empty-single*, *simp-all*)
(*metis* (*no-types*, *lifting*) *inf-sup-aci*(*7*) *le-supI2 strict-events-of-Sync-subset sup.orderE*)

lemma *events-of-Throw-subset* :
‹$\alpha(P\ \Theta\ a \in A.\ Q\ a) \subseteq \alpha(P) \cup (\bigcup a \in A \cap \alpha(P).\ \alpha(Q\ a))$›
proof (*intro subsetI*)
fix *e* **assume** ‹$e \in \alpha(P\ \Theta\ a \in A.\ Q\ a)$›
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* ∩ *ev* ‘ *A* = {}›
| *t1 t2* **where** ‹*s* = *t1* @ *t2*› ‹$t1 \in \mathcal{D}\ P$› ‹*tF t1*› ‹*set t1* ∩ *ev* ‘ *A* = {}› ‹*ftF t2*›
| *t1 a t2* **where** ‹*s* = *t1* @ *ev a* # *t2*› ‹*t1* @ [*ev a*] ∈ $\mathcal{T}$ *P*›
‹*set t1* ∩ *ev* ‘ *A* = {}› ‹*a* ∈ *A*› ‹*t2* ∈ $\mathcal{T}$ (*Q a*)›
by (*simp add*: *T-Throw*) *blast*
thus ‹$e \in \alpha(P) \cup (\bigcup a \in A \cap \alpha(P).\ \alpha(Q\ a))$›
proof *cases*
from ∗(*1*) **show** ‹$s \in \mathcal{T}\ P \Longrightarrow$ *set s* ∩ *ev* ‘ *A* = {} ⟹
$e \in \alpha(P) \cup (\bigcup a \in A \cap \alpha(P).\ \alpha(Q\ a))$›
by (*simp add*: *events-of-def*) *blast*
next
show ‹⟦*s* = *t1* @ *t2*; $t1 \in \mathcal{D}\ P$; *tF t1*; *set t1* ∩ *ev* ‘ *A* = {}; *ftF t2*⟧ ⟹
$e \in \alpha(P) \cup (\bigcup a \in A \cap \alpha(P).\ \alpha(Q\ a))$› **for** *t1 t2*
by (*metis* ∗(*1*) *D-T UnI1 events-of-memI is-processT7*)
next
fix *t1 a t2*
assume ∗∗ : ‹*s* = *t1* @ *ev a* # *t2*› ‹*t1* @ [*ev a*] ∈ $\mathcal{T}$ *P*›
‹*set t1* ∩ *ev* ‘ *A* = {}› ‹*a* ∈ *A*› ‹*t2* ∈ $\mathcal{T}$ (*Q a*)›
from ∗(*1*) ∗∗(*1*) **have** ‹*ev e* ∈ *set* (*t1* @ [*ev a*]) ∨ *ev e* ∈ *set t2*› **by** *simp*
thus ‹$e \in \alpha(P) \cup (\bigcup a \in A \cap \alpha(P).\ \alpha(Q\ a))$›
proof (*elim disjE*)
show ‹*ev e* ∈ *set* (*t1* @ [*ev a*]) ⟹ $e \in \alpha(P) \cup (\bigcup a \in A \cap \alpha(P).\ \alpha(Q\ a))$›
by (*metis* ∗∗(*2*) *UnI1 events-of-memI*)
next
show ‹*ev e* ∈ *set t2* ⟹ $e \in \alpha(P) \cup (\bigcup a \in A \cap \alpha(P).\ \alpha(Q\ a))$›
by (*metis* (*no-types*, *lifting*) ∗∗(*2*, *4*, *5*) *Int-iff UN-iff UnI2*
events-of-memI list.set-intros(*1*) *set-append*)
qed
qed
qed

lemma *events-of-Interrupt* : ‹$\alpha(P \bigtriangleup Q) = \alpha(P) \cup \alpha(Q)$›
by (*safe elim*!: *events-of-memE*,
auto simp add: *events-of-def Interrupt-projs*)
(*metis append-Nil is-processT1-TR tickFree-Nil*)

lemma *strict-events-of-Interrupt-subset* : ‹$\boldsymbol{\alpha}(P \bigtriangleup Q) \subseteq \boldsymbol{\alpha}(P) \cup \boldsymbol{\alpha}(Q)$›
by (*safe elim*!:*strict-events-of-memE*,
auto simp add: *strict-events-of-def Interrupt-projs*)
(*metis DiffI T-imp-front-tickFree is-processT7*)

5.2 Ticks

lemma *ticks-of-GlobalDet*:
‹*ticks-of* (□*a* ∈ *A*. *P a*) = (⋃ *a*∈*A*. *ticks-of* (*P a*))›

by (*auto simp add*: *ticks-of-def T-GlobalDet*)

lemma *strict-ticks-of-GlobalDet-subset* : ‹$\checkmark$**s**($\square a \in A.\ P\ a$) $\subseteq$ ($\bigcup a{\in}A.$ $\checkmark$**s**($P\ a$))›
by (*auto simp add*: *strict-ticks-of-def GlobalDet-projs*)

lemma *ticks-of-MultiSync-subset* :
‹$\checkmark s$($[\![S]\!]\ a \in\#\ M.\ P\ a$) $\subseteq$ ($\bigcup a \in$ *set-mset* $M.$ $\checkmark s(P\ a)$)›
by (*induct M rule*: *induct-subset-mset-empty-single*, *simp-all*)
(*meson Diff-subset-conv dual-order.trans ticks-of-Sync-subset*)

lemma *strict-ticks-of-MultiSync-subset* :
‹$\checkmark$**s**($[\![S]\!]\ a \in\#\ M.\ P\ a$) $\subseteq$ ($\bigcap a \in$ *set-mset* $M.$ $\checkmark$**s**($P\ a$))›
by (*induct M rule*: *induct-subset-mset-empty-single*, *simp-all*)
(*use strict-ticks-of-Sync-subset* **in** *fastforce*)

lemma *ticks-Throw-subset* :
‹$\checkmark s(P\ \Theta\ a{\in}A.\ Q\ a) \subseteq \checkmark s(P) \cup (\bigcup a{\in}A \cap \alpha(P).\ \checkmark s(Q\ a))$›
proof (*rule subsetI*, *elim ticks-of-memE*)
fix *t r* **assume** ‹$t\ @\ [\checkmark(r)] \in \mathcal{T}\ (P\ \Theta\ a{\in}A.\ Q\ a)$›
from ‹$t\ @\ [\checkmark(r)] \in \mathcal{T}\ (P\ \Theta\ a{\in}A.\ Q\ a)$› **consider** ‹$t\ @\ [\checkmark(r)] \in \mathcal{T}\ P$›
| *t1 t2* **where** ‹$t\ @\ [\checkmark(r)] = t1\ @\ t2$› ‹$t1 \in \mathcal{D}\ P$› ‹*tF t1*› ‹*ftF t2*›
| *t1 a t2* **where** ‹$t\ @\ [\checkmark(r)] = t1\ @\ ev\ a\ \#\ t2$› ‹$t1\ @\ [ev\ a] \in \mathcal{T}\ P$› ‹$a \in A$›
‹$t2 \in \mathcal{T}\ (Q\ a)$›
unfolding *T-Throw* **by** *blast*
thus ‹$r \in \checkmark s(P) \cup (\bigcup a{\in}A \cap \alpha(P).\ \checkmark s(Q\ a))$›
proof *cases*
show ‹$t\ @\ [\checkmark(r)] \in \mathcal{T}\ P \Longrightarrow r \in \checkmark s(P) \cup (\bigcup a{\in}A \cap \alpha(P).\ \checkmark s(Q\ a))$›
by (*simp add*: *ticks-of-memI*)
next
show ‹$[\![t\ @\ [\checkmark(r)] = t1\ @\ t2;\ t1 \in \mathcal{D}\ P;\ tF\ t1;\ ftF\ t2]\!]$
$\Longrightarrow r \in \checkmark s(P) \cup (\bigcup a{\in}A \cap \alpha(P).\ \checkmark s(Q\ a))$› **for** *t1 t2*
by (*cases t2 rule*: *rev-cases*, *auto*)
(*metis D-T append-assoc is-processT7 ticks-of-memI*)
next
show ‹$[\![t\ @\ [\checkmark(r)] = t1\ @\ ev\ a\ \#\ t2;\ t1\ @\ [ev\ a] \in \mathcal{T}\ P;\ a \in A;\ t2 \in \mathcal{T}\ (Q\ a)]\!]$
$\Longrightarrow r \in \checkmark s(P) \cup (\bigcup a{\in}A \cap \alpha(P).\ \checkmark s(Q\ a))$› **for** *t1 a t2*
by (*cases t2 rule*: *rev-cases*, *simp-all*)
(*meson IntI events-of-memI in-set-conv-decomp ticks-of-memI*)
qed
qed

lemma *ticks-of-Interrupt* : ‹$\checkmark s(P \triangle Q) = \checkmark s(P) \cup \checkmark s(Q)$›
  by (*safe elim*!: *ticks-of-memE*,
      auto simp add: *ticks-of-def Interrupt-projs*)
    (*metis append.right-neutral last-appendR snoc-eq-iff-butlast*,
      metis append-Nil is-processT1-TR tickFree-Nil)

lemma *strict-ticks-of-Interrupt-subset* : ‹$\checkmark \mathbf{s}(P \triangle Q) \subseteq \checkmark \mathbf{s}(P) \cup \checkmark \mathbf{s}(Q)$›
  by (*safe elim*!: *strict-ticks-of-memE*,
      auto simp add: *strict-ticks-of-def Interrupt-projs*)
    (*meson is-processT9*,
      metis (*no-types*, *opaque-lifting*) *Nil-is-append-conv append-assoc*
      append-butlast-last-id butlast-snoc is-processT9 last-appendR list.distinct(*1*))

events-of and *deadlock-free*

lemma *nonempty-events-of-if-deadlock-free*: ‹*deadlock-free* $P \Longrightarrow \alpha(P) \neq \{\}$›
  unfolding *deadlock-free-def events-of-def failure-divergence-refine-def*
    failure-refine-def divergence-refine-def
  apply (*simp add*: *div-free-DF*, *subst* (*asm*) *DF-unfold*)
  apply (*auto simp add*: *F-Mndetprefix write0-def F-Mprefix subset-iff*)
  by (*metis* (*full-types*) *Nil-elem-T T-F is-processT5-S7*
      list.set-intros(*1*) *rangeI snoc-eq-iff-butlast*)

lemma *nonempty-strict-events-of-if-deadlock-free*: ‹*deadlock-free* $P \Longrightarrow \boldsymbol{\alpha}(P) \neq \{\}$›
  by (*metis deadlock-free-implies-div-free events-of-is-strict-events-of-or-UNIV nonempty-events-of-if-dead*

lemma *events-of-in-DF*: ‹$DF\ A \sqsubseteq_{FD} P \Longrightarrow \alpha(P) \subseteq A$›
  by (*metis anti-mono-events-of-FD events-of-DF*)

lemma *nonempty-events-of-if-deadlock-free*$_{SKIP}$:
  ‹*deadlock-free*$_{SKIPS}$ $P \Longrightarrow (\exists\, r.\ [\checkmark(r)] \in \mathcal{T}\ P) \lor \alpha(P) \neq \{\}$›
  unfolding *deadlock-free*$_{SKIPS}$*-def events-of-def failure-refine-def*
  apply (*subst* (*asm*) *DF*$_{SKIPS}$*-unfold*)
  apply (*auto simp add*: *F-Mndetprefix write0-def F-Mprefix subset-iff F-Ndet F-SKIPS*)
  by (*metis event*$_{ptick}$*.exhaust is-processT1-TR is-processT5-S7 iso-tuple-UNIV-I list.set-intros*(*1*) *self-append-conv2*)

lemma *events-of-in-DF*$_{SKIP}$: ‹$DF_{SKIPS}\ A\ R \sqsubseteq_{FD} P \Longrightarrow \alpha(P) \subseteq A$›
  by (*metis anti-mono-events-of-FD events-of-DF*$_{SKIPS}$)

lemma ‹$\neg\ \alpha(P) \subseteq A \Longrightarrow \neg\ DF\ A \sqsubseteq_{FD} P$›
  and ‹$\neg\ \alpha(P) \subseteq A \Longrightarrow \neg\ DF_{SKIPS}\ A\ R \sqsubseteq_{FD} P$›

by (*metis anti-mono-events-of-FD events-of-DF*)
(*metis anti-mono-events-of-FD events-of-DF*$_{SKIPS}$)

lemma ‹*chain Y* $\Longrightarrow$ $\alpha(\bigsqcup i.\ Y\ i) = (\bigcup i.\ \alpha(Y\ i))$›
apply (*simp add*: *events-of-def limproc-is-thelub T-LUB D-LUB*)
apply *auto*

oops

lemma *f1* : ‹*chain Y* $\Longrightarrow$ $\boldsymbol{\alpha}(\bigsqcup i.\ Y\ i) = (\bigcup i.\ \boldsymbol{\alpha}(Y\ i))$›
apply (*simp add*: *strict-events-of-def limproc-is-thelub T-LUB D-LUB*)
apply *auto*

by (*smt* (*verit*, *ccfv-threshold*) *D-T DiffI INT-iff Inter-iff le-approx2T lim-proc-is-ub rangeI ub-rangeD*)

find-theorems *Lub*

lemma *f2* : ‹*chain Y* $\Longrightarrow$ $\mathcal{D}\ (Y\ i) = \{\} \Longrightarrow (\bigcup i.\ \boldsymbol{\alpha}(Y\ i)) = \boldsymbol{\alpha}(Y\ i)$›
apply (*auto simp add*: *strict-events-of-def*)
by (*meson ND-F-dir2′ chain-lemma*)

Chapter 6

Deadlock results

When working with the interleaving $P \mid\mid\mid Q$, we intuitively expect it to be *deadlock-free* when both P and Q are.

This chapter contains several results about deadlock notion, and concludes with a proof of the theorem we just mentioned.

6.1 Unfolding lemmas for the projections of DF and DF_{SKIPS}

DF and DF_{SKIPS} naturally appear when we work around *deadlock-free* and *deadlock-free*$_{SKIPS}$ notions (because

deadlock-free $P \equiv DF\ UNIV \sqsubseteq_{FD} P$

deadlock-free$_{SKIPS}$ $P \equiv DF_{SKIPS}\ UNIV\ UNIV \sqsubseteq_{F} P$).

It is therefore convenient to have the following rules for unfolding the projections.

lemma *F-DF*:
‹$\mathcal{F}$ $(DF\ A)$ =
$(if\ A = \{\}\ then\ \{(s, X).\ s = []\}$
$else\ (\bigcup a{\in}A.\ \{[]\} \times \{X.\ ev\ a \notin X\} \cup \{(ev\ a \# s, X)|\ s\ X.\ (s, X) \in \mathcal{F}\ (DF\ A)\}))$›
by (*subst DF-unfold*) (*auto simp add*: *F-STOP F-Mndetprefix write0-def F-Mprefix*)

lemma *F-DF*$_{SKIPS}$:
‹$\mathcal{F}$ $(DF_{SKIPS}\ A\ R)$ =
$(\ if\ A = \{\}\ then\ \{(s, X).\ s = [] \lor (\exists r{\in}R.\ s = [✓(r)])\}$
$else\ (\bigcup a{\in}A.\ \{[]\} \times \{X.\ ev\ a \notin X\} \cup$
$\{(ev\ a \# s, X)|\ s\ X.\ (s, X) \in \mathcal{F}\ (DF_{SKIPS}\ A\ R)\}) \cup$
$(\ if\ R = \{\}\ then\ \{(s, X).\ s = []\}$
$else\ \{([], X)\ |X.\ \exists r{\in}R.\ ✓(r) \notin X\} \cup \{(s, X).\ \exists r{\in}R.\ s = [✓(r)]\}))$›
by (*subst DF*$_{SKIPS}$*-unfold*, *simp add*: *F-Ndet F-STOP F-SKIPS F-Mndetprefix*

write0-def F-Mprefix, safe, simp-all)

corollary *Cons-F-DF*:
‹$(x \# t, X) \in \mathcal{F}$ (*DF A*) $\Longrightarrow (t, X) \in \mathcal{F}$ (*DF A*)›
and *Cons-F-DF*$_{SKIPS}$:
‹$x \notin$ *tick* ‘ $R \Longrightarrow (x \# t, X) \in \mathcal{F}$ (*DF*$_{SKIPS}$ *A R*) $\Longrightarrow (t, X) \in \mathcal{F}$ (*DF*$_{SKIPS}$ *A R*)›
by (*subst* (*asm*) *F-DF F-DF*$_{SKIPS}$; *auto split*: *if-split-asm*)+

lemma *D-DF*: ‹$\mathcal{D}$ (*DF A*) = (*if* $A = \{\}$ *then* $\{\}$ *else* $\{ev\ a \# s |\ a\ s.\ a \in A \wedge s \in \mathcal{D}$ (*DF A*)$\}$)›
and *D-DF*$_{SKIPS}$: ‹$\mathcal{D}$ (*DF*$_{SKIPS}$ *A R*) = (*if* $A = \{\}$ *then* $\{\}$ *else* $\{ev\ a \# s |\ a\ s.\ a \in A \wedge s \in \mathcal{D}$ (*DF*$_{SKIPS}$ *A R*)$\}$)›
by (*subst DF-unfold DF*$_{SKIPS}$*-unfold*;
auto simp add: *D-Mndetprefix D-Mprefix write0-def D-Ndet D-SKIPS*)+

thm *T-SKIPS*[*of R*]
lemma *T-DF*:
‹$\mathcal{T}$ (*DF A*) = (*if* $A = \{\}$ *then* $\{[]\}$ *else insert* [] $\{ev\ a \# s |\ a\ s.\ a \in A \wedge s \in \mathcal{T}$ (*DF A*)$\}$)›
and *T-DF*$_{SKIPS}$:
‹$\mathcal{T}$ (*DF*$_{SKIPS}$ *A R*) = (*if* $A = \{\}$ *then insert* [] $\{[✓(r)]\ |r.\ r \in R\}$
else $\{s.\ s = [] \vee (\exists r{\in}R.\ s = [✓(r)]) \vee$
$s \neq [] \wedge (\exists a{\in}A.$ *hd* $s = ev\ a \wedge$ *tl* $s \in \mathcal{T}$ (*DF*$_{SKIPS}$ *A R*))$\}$)›
by (*subst DF-unfold DF*$_{SKIPS}$*-unfold*;
auto simp add: *T-STOP T-Mndetprefix write0-def T-Mprefix T-Ndet T-SKIPS*)+
(*metis list.collapse*)

6.2 Characterizations for *deadlock-free*, *deadlock-free*$_{SKIPS}$

We want more results like *deadlock-free* $(P \sqcap Q)$ = (*deadlock-free* $P \wedge$ *deadlock-free* Q), and we want to add the reciprocal when possible.

The first thing we notice is that we only have to care about the failures

lemma ‹*deadlock-free*$_{SKIPS}$ $P \equiv$ *DF*$_{SKIPS}$ *UNIV UNIV* $\sqsubseteq_F P$›
by (*fact deadlock-free*$_{SKIPS}$*-def*)

lemma *deadlock-free-F*: ‹*deadlock-free* $P \longleftrightarrow$ *DF UNIV* $\sqsubseteq_F P$›
by (*auto simp add*: *deadlock-free-def refine-defs F-subset-imp-T-subset*
non-terminating-refine-DF nonterminating-implies-div-free)

lemma *deadlock-free-Mprefix-iff*: ‹*deadlock-free* ($\square\ a \in A \rightarrow P\ a$) $\longleftrightarrow$
$A \neq \{\} \wedge (\forall a \in A.$ *deadlock-free* $(P\ a))$›
and *deadlock-free*$_{SKIPS}$*-Mprefix-iff*: ‹*deadlock-free*$_{SKIPS}$ (*Mprefix A P*) $\longleftrightarrow$

$A \neq \{\} \wedge (\forall\, a \in A.\ \textit{deadlock-free}_{SKIPS}\ (P\ a))$›
unfolding *deadlock-free-F deadlock-free*$_{SKIPS}$*-def failure-refine-def*
apply (*all* ‹*subst F-DF F-DF*$_{SKIPS}$›,
auto simp add: *div-free-DF F-Mprefix D-Mprefix subset-iff*)
by *blast+*

lemma *deadlock-free-read-iff* :
‹*deadlock-free* $(c?a{\in}A \rightarrow P\ a) \longleftrightarrow A \neq \{\} \wedge (\forall\, a{\in}c\ `\ A.\ \textit{deadlock-free}\ ((P \circ \textit{inv-into}\ A\ c)\ a))$›
and *deadlock-free*$_{SKIPS}$*-read-iff* :
‹*deadlock-free*$_{SKIPS}$ $(c?a{\in}A \rightarrow P\ a) \longleftrightarrow A \neq \{\} \wedge (\forall\, a{\in}c\ `\ A.\ \textit{deadlock-free}_{SKIPS}\ ((P \circ \textit{inv-into}\ A\ c)\ a))$›
by (*simp-all add*: *read-def deadlock-free-Mprefix-iff deadlock-free*$_{SKIPS}$*-Mprefix-iff*)

lemma *deadlock-free-read-inj-on-iff* :
‹*inj-on c A* $\Longrightarrow$ *deadlock-free* $(c?a{\in}A \rightarrow P\ a) \longleftrightarrow A \neq \{\} \wedge (\forall\, a{\in}A.\ \textit{deadlock-free}\ (P\ a))$›
and *deadlock-free*$_{SKIPS}$*-read-inj-on-iff* :
‹*inj-on c A* $\Longrightarrow$ *deadlock-free*$_{SKIPS}$ $(c?a{\in}A \rightarrow P\ a) \longleftrightarrow A \neq \{\} \wedge (\forall\, a{\in}A.\ \textit{deadlock-free}_{SKIPS}\ (P\ a))$›
by (*simp-all add*: *deadlock-free-read-iff deadlock-free*$_{SKIPS}$*-read-iff*)

lemma *deadlock-free-write-iff* :
‹*deadlock-free* $(c!a \rightarrow P) \longleftrightarrow$ *deadlock-free P*›
and *deadlock-free*$_{SKIPS}$*-write-iff* :
‹*deadlock-free*$_{SKIPS}$ $(c!a \rightarrow P) \longleftrightarrow$ *deadlock-free*$_{SKIPS}$ *P*›
by (*simp-all add*: *deadlock-free-Mprefix-iff deadlock-free*$_{SKIPS}$*-Mprefix-iff write-def*)

lemma *deadlock-free-write0-iff* :
‹*deadlock-free* $(a \rightarrow P) \longleftrightarrow$ *deadlock-free P*›
and *deadlock-free*$_{SKIPS}$*-write0-iff* :
‹*deadlock-free*$_{SKIPS}$ $(a \rightarrow P) \longleftrightarrow$ *deadlock-free*$_{SKIPS}$ *P*›
by (*simp-all add*: *deadlock-free-Mprefix-iff deadlock-free*$_{SKIPS}$*-Mprefix-iff write0-def*)

lemma *deadlock-free-GlobalNdet-iff*: ‹*deadlock-free* $(\sqcap\ a \in A.\ P\ a) \longleftrightarrow$
$A \neq \{\} \wedge (\forall\ a \in A.\ \textit{deadlock-free}\ (P\ a))$›
and *deadlock-free*$_{SKIPS}$*-GlobalNdet-iff*: ‹*deadlock-free*$_{SKIPS}$ $(\sqcap\ a \in A.\ P\ a) \longleftrightarrow$
$A \neq \{\} \wedge (\forall\ a \in A.\ \textit{deadlock-free}_{SKIPS}\ (P\ a))$›
by (*metis* (*mono-tags*, *lifting*) *GlobalNdet-refine-FD deadlock-free-def trans-FD mono-GlobalNdet-FD-const non-deadlock-free-STOP GlobalNdet-empty*)
(*metis* (*mono-tags*, *lifting*) *GlobalNdet-refine-FD deadlock-free*$_{SKIPS}$*-FD trans-FD mono-GlobalNdet-FD-const non-deadlock-free*$_{SKIPS}$*-STOP GlobalNdet-empty*)

lemma *deadlock-free-Mndetprefix-iff*: ‹*deadlock-free* $(\sqcap\ a \in A \rightarrow P\ a) \longleftrightarrow$
$A \neq \{\} \wedge (\forall\, a{\in}A.$ *deadlock-free* $(P\ a))$›
and *deadlock-free*$_{SKIPS}$*-Mndetprefix-iff*: ‹*deadlock-free*$_{SKIPS}$ $(\sqcap\ a \in A \rightarrow P\ a)$ $\longleftrightarrow$
$A \neq \{\} \wedge (\forall\, a{\in}A.$ *deadlock-free*$_{SKIPS}$ $(P\ a))$›
by (*simp-all add*: *Mndetprefix-GlobalNdet*
deadlock-free-GlobalNdet-iff deadlock-free$_{SKIPS}$*-GlobalNdet-iff*
deadlock-free-write0-iff deadlock-free$_{SKIPS}$*-write0-iff*)

lemma *deadlock-free-Ndet-iff*: ‹*deadlock-free* $(P \sqcap Q) \longleftrightarrow$
deadlock-free $P \wedge$ *deadlock-free* Q›
and *deadlock-free*$_{SKIPS}$*-Ndet-iff*: ‹*deadlock-free*$_{SKIPS}$ $(P \sqcap Q) \longleftrightarrow$
deadlock-free$_{SKIPS}$ $P \wedge$ *deadlock-free*$_{SKIPS}$ Q›
unfolding *deadlock-free-F deadlock-free*$_{SKIPS}$*-def failure-refine-def*
by (*simp-all add*: *F-Ndet*)

lemma *deadlock-free-is-right*:
‹*deadlock-free* $(P :: ('a,\ 'r)\ process_{ptick}) \longleftrightarrow (\forall\, s \in \mathcal{T}\ P.$ *tickFree* $s \wedge (s,$ *UNIV*$) \notin \mathcal{F}\ P)$›
‹*deadlock-free* $P \longleftrightarrow (\forall\, s \in \mathcal{T}\ P.$ *tickFree* $s \wedge (s,$ *ev* ‘ *UNIV*$) \notin \mathcal{F}\ P)$›
oops

lemma ‹*deadlock-free* $(P \;\square\; Q) \longleftrightarrow P = STOP \wedge$ *deadlock-free* $Q \vee$ *deadlock-free* $P \wedge Q = STOP$›

oops

lemma *deadlock-free-GlobalDet-iff* :
‹$[\![A \neq \{\};$ *finite* $A;\ \forall\, a{\in}A.$ *deadlock-free* $(P\ a)]\!] \Longrightarrow$ *deadlock-free* $(\square a \in A.\ P\ a)$›
and *deadlock-free*$_{SKIPS}$*-MultiDet*:
‹$[\![A \neq \{\};$ *finite* $A;\ \forall\, a{\in}A.$ *deadlock-free*$_{SKIPS}$ $(P\ a)]\!] \Longrightarrow$ *deadlock-free*$_{SKIPS}$ $(\square a \in A.\ P\ a)$›
by (*metis GlobalNdet-FD-GlobalDet deadlock-free-GlobalNdet-iff deadlock-free-def trans-FD*)
(*metis GlobalNdet-FD-GlobalDet deadlock-free*$_{SKIPS}$*-FD deadlock-free*$_{SKIPS}$*-GlobalNdet-iff trans-FD*)

lemma *deadlock-free-Det*:
‹*deadlock-free* $P \Longrightarrow$ *deadlock-free* $Q \Longrightarrow$ *deadlock-free* $(P \,\square\, Q)$›
and *deadlock-free*$_{SKIPS}$*-Det*:
‹*deadlock-free*$_{SKIPS}$ $P \Longrightarrow$ *deadlock-free*$_{SKIPS}$ $Q \Longrightarrow$ *deadlock-free*$_{SKIPS}$ $(P \,\square\, Q)$›
by (*metis deadlock-free-Ndet-iff Ndet-FD-Det deadlock-free-def trans-FD*)
(*metis deadlock-free*$_{SKIPS}$*-Ndet-iff Ndet-F-Det deadlock-free*$_{SKIPS}$*-def trans-F*)

For $P \,\square\, Q$, we can not expect more:

lemma
‹$\exists\, P\, Q.$ *deadlock-free* $P \wedge \neg$ *deadlock-free* $Q \wedge$
deadlock-free $(P \,\square\, Q)$›
‹$\exists\, P\, Q.$ *deadlock-free*$_{SKIPS}$ $P \wedge \neg$ *deadlock-free*$_{SKIPS}$ $Q \wedge$
deadlock-free$_{SKIPS}$ $(P \,\square\, Q)$›
by (*rule-tac* $x =$ ‹*DF UNIV*› **in** *exI*, *rule-tac* $x = STOP$ **in** *exI*,
simp add: *non-deadlock-free-STOP*, *simp add*: *deadlock-free-def*)
(*rule-tac* $x =$ ‹DF_{SKIPS} *UNIV UNIV*› **in** *exI*, *rule-tac* $x = STOP$ **in** *exI*,
simp add: *non-deadlock-free*$_{SKIPS}$*-STOP*, *simp add*: *deadlock-free*$_{SKIPS}$*-FD*)

lemma *FD-Mndetprefix-iff*:
‹$A \neq \{\} \Longrightarrow P \sqsubseteq_{FD} \sqcap\, a \in A \rightarrow Q \longleftrightarrow (\forall\, a \in A.\ P \sqsubseteq_{FD} (a \rightarrow Q))$›
by (*auto simp*: *failure-divergence-refine-def failure-refine-def divergence-refine-def subset-iff D-Mndetprefix F-Mndetprefix write0-def D-Mprefix F-Mprefix*)

lemma *Mndetprefix-FD*: ‹$(\exists\, a \in A.\ (a \rightarrow Q) \sqsubseteq_{FD} P) \Longrightarrow \sqcap\, a \in A \rightarrow Q \sqsubseteq_{FD} P$›
by (*metis FD-Mndetprefix-iff ex-in-conv idem-FD trans-FD*)

Mprefix, *Sync* and *deadlock-free*

lemma *Mprefix-Sync-deadlock-free*:
assumes *not-all-empty*: ‹$A \neq \{\} \vee B \neq \{\} \vee A' \cap B' \neq \{\}$›
and ‹$A \cap S = \{\}$› **and** ‹$A' \subseteq S$› **and** ‹$B \cap S = \{\}$› **and** ‹$B' \subseteq S$›
and ‹$\forall\, x{\in}A.$ *deadlock-free* $(P\ x\ [\![S]\!]$ *Mprefix* $(B \cup B')\ Q)$›
and ‹$\forall\, y{\in}B.$ *deadlock-free* (*Mprefix* $(A \cup A')\ P\ [\![S]\!]\ Q\ y)$›
and ‹$\forall\, x{\in}A' \cap B'.$ *deadlock-free* $((P\ x\ [\![S]\!]\ Q\ x))$›
shows ‹*deadlock-free* (*Mprefix* $(A \cup A')\ P\ [\![S]\!]$ *Mprefix* $(B \cup B')\ Q)$›
proof −
have ‹$A = \{\} \wedge B \neq \{\} \wedge A' \cap B' \neq \{\} \vee A \neq \{\} \wedge B = \{\} \wedge A' \cap B' = \{\} \vee$
$A \neq \{\} \wedge B = \{\} \wedge A' \cap B' \neq \{\} \vee A = \{\} \wedge B \neq \{\} \wedge A' \cap B' = \{\} \vee$
$A \neq \{\} \wedge B \neq \{\} \wedge A' \cap B' = \{\} \vee A = \{\} \wedge B = \{\} \wedge A' \cap B' \neq \{\} \vee$
$A \neq \{\} \wedge B \neq \{\} \wedge A' \cap B' \neq \{\}$› **by** (*meson not-all-empty*)
thus *?thesis*
by (*elim disjE*, *all* ‹*subst Mprefix-Sync-Mprefix-bis*[*OF assms*$(2{-}5)$]›)

(*use assms*(*6−8*) **in** ‹*auto intro*!: *deadlock-free-Det deadlock-free-Mprefix-iff* [*THEN iffD2*]›)
qed

lemmas *Mprefix-Sync-subset-deadlock-free* = *Mprefix-Sync-deadlock-free*
[**where** *A* = ‹{}› **and** *B* = ‹{}›, *simplified*]
and *Mprefix-Sync-indep-deadlock-free* = *Mprefix-Sync-deadlock-free*
[**where** A' = ‹{}› **and** B' = ‹{}›, *simplified*]
and *Mprefix-Sync-right-deadlock-free* = *Mprefix-Sync-deadlock-free*
[**where** *A* = ‹{}› **and** B' = ‹{}›, *simplified*]
and *Mprefix-Sync-left-deadlock-free* = *Mprefix-Sync-deadlock-free*
[**where** A' = ‹{}› **and** *B* = ‹{}›, *simplified*]

6.3 Results on *Renaming*

The *Renaming* operator is new (release of 2023), so here are its properties on reference processes from *HOL−CSP.CSP-Assertions*, and deadlock notion.

6.3.1 Behaviour with references processes

For *DF*

lemma *DF-FD-Renaming-DF*: ‹*DF* (*f* ‘ *A*) $\sqsubseteq_{FD}$ *Renaming* (*DF A*) *f g*›
proof (*subst DF-def*, *induct rule*: *fix-ind*)
show ‹*adm* (λ*a*. *a* $\sqsubseteq_{FD}$ *Renaming* (*DF A*) *f g*)› **by** (*simp add*: *monofun-def*)
next
show ‹⊥ $\sqsubseteq_{FD}$ *Renaming* (*DF A*) *f g*› **by** *simp*
next
show ‹(Λ *x*. ⊓*a* ∈ *f* ‘ *A* → *x*)·*x* $\sqsubseteq_{FD}$ *Renaming* (*DF A*) *f g*›
if ‹*x* $\sqsubseteq_{FD}$ *Renaming* (*DF A*) *f g*› **for** *x*
apply *simp*
apply (*subst DF-unfold*)
apply (*subst Renaming-Mndetprefix*)
by (*auto simp add*: *that intro*!: *mono-Mndetprefix-FD*)
qed

lemma *Renaming-DF-FD-DF*: ‹*Renaming* (*DF A*) *f g* $\sqsubseteq_{FD}$ *DF* (*f* ‘ *A*)›
if *finitary*: ‹*finitary f*› ‹*finitary g*›
proof (*subst DF-def*, *induct rule*: *fix-ind*)
show ‹*adm* (λ*a*. *Renaming a f g* $\sqsubseteq_{FD}$ *DF* (*f* ‘ *A*))›
by (*simp add*: *finitary monofun-def*)
next
show ‹*Renaming* ⊥ *f g* $\sqsubseteq_{FD}$ *DF* (*f* ‘ *A*)› **by** *simp*
next
show ‹*Renaming* ((Λ *x*. ⊓*a* ∈ *A* → *x*)·*x*) *f g* $\sqsubseteq_{FD}$ *DF* (*f* ‘ *A*)›
if ‹*Renaming x f g* $\sqsubseteq_{FD}$ *DF* (*f* ‘ *A*)› **for** *x*

apply *simp*
apply (*subst Renaming-Mndetprefix*)
apply (*subst DF-unfold*)
by (*auto simp add*: *that intro*!: *mono-Mndetprefix-FD*)
qed

For DF_{SKIPS}

lemma *Renaming-SKIPS* [*simp*] : ‹*Renaming* (*SKIPS R*) *f g* = *SKIPS* (*g* ‘ *R*)›
by (*simp add*: *SKIPS-def Renaming-distrib-GlobalNdet*)
(*metis mono-GlobalNdet-eq2*)

lemma DF_{SKIPS}*-FD-Renaming-*DF_{SKIPS}:
‹DF_{SKIPS} (*f* ‘ *A*) (*g* ‘ *R*) $\sqsubseteq_{FD}$ *Renaming* (DF_{SKIPS} *A R*) *f g*›
proof (*subst* DF_{SKIPS}*-def*, *induct rule*: *fix-ind*)
show ‹*adm* (λ*a*. *a* $\sqsubseteq_{FD}$ *Renaming* (DF_{SKIPS} *A R*) *f g*)› **by** (*simp add*: *monofun-def*)
next
show ‹⊥ $\sqsubseteq_{FD}$ *Renaming* (DF_{SKIPS} *A R*) *f g*› **by** *simp*
next
show ‹(Λ *x*. (⊓ *a*∈*f* ‘ *A* → *x*) ⊓ *SKIPS* (*g* ‘ *R*))·*x* $\sqsubseteq_{FD}$ *Renaming* (DF_{SKIPS} *A R*) *f g*›
if ‹*x* $\sqsubseteq_{FD}$ *Renaming* (DF_{SKIPS} *A R*) *f g*› **for** *x*
by (*subst* DF_{SKIPS}*-unfold*)
(*auto simp add*: *Renaming-Ndet Renaming-Mndetprefix*
intro!: *mono-Ndet-FD mono-Mndetprefix-FD that*)
qed

lemma *Renaming-*DF_{SKIPS}*-FD-*DF_{SKIPS}:
‹*Renaming* (DF_{SKIPS} *A R*) *f g* $\sqsubseteq_{FD}$ DF_{SKIPS} (*f* ‘ *A*) (*g* ‘ *R*)›
if *finitary*: ‹*finitary f*› ‹*finitary g*›
proof (*subst* DF_{SKIPS}*-def*, *induct rule*: *fix-ind*)
show ‹*adm* (λ*a*. *Renaming a f g* $\sqsubseteq_{FD}$ DF_{SKIPS} (*f* ‘ *A*) (*g* ‘ *R*))›
by (*simp add*: *finitary monofun-def*)
next
show ‹*Renaming* ⊥ *f g* $\sqsubseteq_{FD}$ DF_{SKIPS} (*f* ‘ *A*) (*g* ‘ *R*)› **by** *simp*
next
show ‹*Renaming* ((Λ *x*. (⊓*a* ∈ *A* → *x*) ⊓ *SKIPS R*)·*x*) *f g* $\sqsubseteq_{FD}$ DF_{SKIPS} (*f* ‘ *A*) (*g* ‘ *R*)›
if ‹*Renaming x f g* $\sqsubseteq_{FD}$ DF_{SKIPS} (*f* ‘ *A*) (*g* ‘ *R*)› **for** *x*
by (*subst* DF_{SKIPS}*-unfold*)
(*auto simp add*: *Renaming-Ndet Renaming-Mndetprefix*
intro!: *mono-Ndet-FD mono-Mndetprefix-FD that*)
qed

For *RUN*

lemma *RUN-FD-Renaming-RUN*: ‹*RUN* (*f* ‘ *A*) $\sqsubseteq_{FD}$ *Renaming* (*RUN A*) *f g*›
proof (*subst RUN-def*, *induct rule*: *fix-ind*)
show ‹*adm* (λ*a*. *a* $\sqsubseteq_{FD}$ *Renaming* (*RUN A*) *f g*)› **by** (*simp add*: *monofun-def*)

```
next
  show ‹⊥ ⊑FD Renaming (RUN A) f g› by simp
next
  show ‹(Λ x. □a ∈ f ‘ A →  x)·x ⊑FD Renaming (RUN A) f g›
    if ‹x ⊑FD Renaming (RUN A) f g› for x
    by (subst RUN-unfold)
      (auto simp add: Renaming-Mprefix intro!: mono-Mprefix-FD that)
qed

lemma Renaming-RUN-FD-RUN: ‹Renaming (RUN A) f g ⊑FD RUN (f ‘ A)›
  if finitary: ‹finitary f› ‹finitary g›
proof (subst RUN-def, induct rule: fix-ind)
  show ‹adm (λa. Renaming a f g ⊑FD RUN (f ‘ A))›
    by (simp add: finitary monofun-def)
next
  show ‹Renaming ⊥ f g ⊑FD RUN (f ‘ A)› by simp
next
  show ‹Renaming ((Λ x. □a ∈ A →  x)·x) f g ⊑FD RUN (f ‘ A)›
    if ‹Renaming x f g ⊑FD RUN (f ‘ A)› for x
    by (subst RUN-unfold)
      (auto simp add: Renaming-Mprefix intro!: mono-Mprefix-FD that)
qed
```

For *CHAOS*

```
lemma CHAOS-FD-Renaming-CHAOS:
  ‹CHAOS (f ‘ A) ⊑FD Renaming (CHAOS A) f g›
proof (subst CHAOS-def, induct rule: fix-ind)
 show ‹adm (λa. a ⊑FD Renaming (CHAOS A) f g)› by (simp add: monofun-def)
next
  show ‹⊥ ⊑FD Renaming (CHAOS A) f g› by simp
next
  show ‹(Λ x. STOP ⊓ (□a∈f ‘ A → x))·x ⊑FD Renaming (CHAOS A) f g›
    if ‹x ⊑FD Renaming (CHAOS A) f g› for x
    by (subst CHAOS-unfold)
      (auto simp add: Renaming-Mprefix Renaming-Ndet
        intro!: mono-Ndet-FD[OF idem-FD] mono-Mprefix-FD that)
qed

lemma Renaming-CHAOS-FD-CHAOS:
  ‹Renaming (CHAOS A) f g ⊑FD CHAOS (f ‘ A)›
  if finitary: ‹finitary f› ‹finitary g›
proof (subst CHAOS-def, induct rule: fix-ind)
  show ‹adm (λa. Renaming a f g ⊑FD CHAOS (f ‘ A))›
    by (simp add: finitary monofun-def)
next
  show ‹Renaming ⊥ f g ⊑FD CHAOS (f ‘ A)› by simp
next
  show ‹Renaming ((Λ x. STOP ⊓ (□xa∈A → x))·x) f g ⊑FD CHAOS (f ‘ A)›
    if ‹Renaming x f g ⊑FD CHAOS (f ‘ A)› for x
```

by (*subst CHAOS-unfold*)
(*auto simp add*: *Renaming-Mprefix Renaming-Ndet*
intro!: *mono-Ndet-FD*[*OF idem-FD*] *mono-Mprefix-FD that*)
qed

For $CHAOS_{SKIPS}$

lemma $CHAOS_{SKIPS}$-*FD-Renaming*-$CHAOS_{SKIPS}$:
‹$CHAOS_{SKIPS}$ (*f* ‘ *A*) (*g* ‘ *R*) $\sqsubseteq_{FD}$ *Renaming* ($CHAOS_{SKIPS}$ *A R*) *f g*›
proof (*subst* $CHAOS_{SKIPS}$-*def*, *induct rule*: *fix-ind*)
show ‹*adm* (λ*a*. *a* $\sqsubseteq_{FD}$ *Renaming* ($CHAOS_{SKIPS}$ *A R*) *f g*)›
by (*simp add*: *monofun-def*)
next
show ‹⊥ $\sqsubseteq_{FD}$ *Renaming* ($CHAOS_{SKIPS}$ *A R*) *f g*› **by** *simp*
next
show ‹(Λ *x*. *SKIPS* (*g* ‘ *R*) ⊓ *STOP* ⊓ (□*xa*∈*f* ‘ *A* → *x*))·*x* $\sqsubseteq_{FD}$
Renaming ($CHAOS_{SKIPS}$ *A R*) *f g*›
if ‹*x* $\sqsubseteq_{FD}$ *Renaming* ($CHAOS_{SKIPS}$ *A R*) *f g*› **for** *x*
by (*subst* $CHAOS_{SKIPS}$-*unfold*)
(*auto simp add*: *Renaming-Ndet Renaming-Mprefix*
intro!: *mono-Ndet-FD mono-Mprefix-FD that*)
qed

lemma *Renaming*-$CHAOS_{SKIPS}$-*FD*-$CHAOS_{SKIPS}$:
‹*Renaming* ($CHAOS_{SKIPS}$ *A R*) *f g* $\sqsubseteq_{FD}$ $CHAOS_{SKIPS}$ (*f* ‘ *A*) (*g* ‘ *R*)›
if *finitary*: ‹*finitary f*› ‹*finitary g*›
proof (*subst* $CHAOS_{SKIPS}$-*def*, *induct rule*: *fix-ind*)
show ‹*adm* (λ*a*. *Renaming a f g* $\sqsubseteq_{FD}$ $CHAOS_{SKIPS}$ (*f* ‘ *A*) (*g* ‘ *R*))›
by (*simp add*: *finitary monofun-def*)
next
show ‹*Renaming* ⊥ *f g* $\sqsubseteq_{FD}$ $CHAOS_{SKIPS}$ (*f* ‘ *A*) (*g* ‘ *R*)› **by** *simp*
next
show ‹ *Renaming* ((Λ *x*. *SKIPS R* ⊓ *STOP* ⊓ (□*xa*∈*A* → *x*))·*x*) *f g* $\sqsubseteq_{FD}$
$CHAOS_{SKIPS}$ (*f* ‘ *A*) (*g* ‘ *R*)›
if ‹*Renaming x f g* $\sqsubseteq_{FD}$ $CHAOS_{SKIPS}$ (*f* ‘ *A*) (*g* ‘ *R*)› **for** *x*
by (*subst* $CHAOS_{SKIPS}$-*unfold*)
(*auto simp add*: *Renaming-Ndet Renaming-Mprefix*
intro!: *mono-Ndet-FD mono-Mprefix-FD that*)
qed

6.3.2 Corollaries on *deadlock-free* and *deadlock-free*$_{SKIPS}$

lemmas *Renaming-DF* =
FD-antisym[*OF Renaming-DF-FD-DF DF-FD-Renaming-DF*]
and *Renaming*-DF_{SKIPS} =
FD-antisym[*OF Renaming*-DF_{SKIPS}-*FD*-DF_{SKIPS} DF_{SKIPS}-*FD-Renaming*-DF_{SKIPS}]
and *Renaming-RUN* =
FD-antisym[*OF Renaming-RUN-FD-RUN RUN-FD-Renaming-RUN*]
and *Renaming-CHAOS* =
FD-antisym[*OF Renaming-CHAOS-FD-CHAOS CHAOS-FD-Renaming-CHAOS*]

and *Renaming-CHAOS*$_{SKIPS}$ =
FD-antisym[*OF Renaming-CHAOS*$_{SKIPS}$*-FD-CHAOS*$_{SKIPS}$
CHAOS$_{SKIPS}$*-FD-Renaming-CHAOS*$_{SKIPS}$]

lemma *deadlock-free-imp-deadlock-free-Renaming*: ‹*deadlock-free* (*Renaming P f g*)›
if ‹*deadlock-free P*›
apply (*rule DF-Univ-freeness*[*of* ‹*range f*›], *simp*)
apply (*rule trans-FD*[*OF DF-FD-Renaming-DF*])
apply (*rule mono-Renaming-FD*)
by (*rule that*[*unfolded deadlock-free-def*])

lemma *deadlock-free-Renaming-imp-deadlock-free*: ‹*deadlock-free P*›
if ‹*inj f*› **and** ‹*inj g*› **and** ‹*deadlock-free* (*Renaming P f g*)›
apply (*subst Renaming-inv*[*OF that*(*1*, *2*), *symmetric*])
apply (*rule deadlock-free-imp-deadlock-free-Renaming*)
by (*rule that*(*3*))

corollary *deadlock-free-Renaming-iff*:
‹*inj f* $\Longrightarrow$ *inj g* $\Longrightarrow$ *deadlock-free* (*Renaming P f g*) $\longleftrightarrow$ *deadlock-free P*›
using *deadlock-free-Renaming-imp-deadlock-free*
deadlock-free-imp-deadlock-free-Renaming **by** *blast*

lemma *deadlock-free*$_{SKIPS}$*-imp-deadlock-free*$_{SKIPS}$*-Renaming*:
‹*deadlock-free*$_{SKIPS}$ *P* $\Longrightarrow$ *deadlock-free*$_{SKIPS}$ (*Renaming P f g*)›
unfolding *deadlock-free*$_{SKIPS}$*-FD*
apply (*rule trans-FD*[*of -* ‹*DF*$_{SKIPS}$ (*f* ‘ *UNIV*) (*g* ‘ *UNIV*)›])
by (*simp add*: *DF*$_{SKIPS}$*-subset*) (*meson DF*$_{SKIPS}$*-FD-Renaming-DF*$_{SKIPS}$ *mono-Renaming-FD trans-FD*)

lemma *deadlock-free*$_{SKIPS}$*-Renaming-imp-deadlock-free*$_{SKIPS}$:
‹*deadlock-free*$_{SKIPS}$ *P*› **if** ‹*inj f*› **and** ‹*inj g*› **and** ‹*deadlock-free*$_{SKIPS}$ (*Renaming P f g*)›
apply (*subst Renaming-inv*[*OF that*(*1*, *2*), *symmetric*])
apply (*rule deadlock-free*$_{SKIPS}$*-imp-deadlock-free*$_{SKIPS}$*-Renaming*)
by (*rule that*(*3*))

corollary *deadlock-free*$_{SKIPS}$*-Renaming-iff*:
‹*inj f* $\Longrightarrow$ *inj g* $\Longrightarrow$ *deadlock-free*$_{SKIPS}$ (*Renaming P f g*) $\longleftrightarrow$ *deadlock-free*$_{SKIPS}$ *P*›
using *deadlock-free*$_{SKIPS}$*-Renaming-imp-deadlock-free*$_{SKIPS}$
deadlock-free$_{SKIPS}$*-imp-deadlock-free*$_{SKIPS}$*-Renaming* **by** *blast*

6.4 The big results

6.4.1 An interesting equivalence

lemma *deadlock-free-of-Sync-iff-DF-FD-DF-Sync-DF*:
‹($\forall$ *P Q*. *deadlock-free* (*P*::('*a*, '*r*) $process_{ptick}$) $\longrightarrow$ *deadlock-free Q* $\longrightarrow$
deadlock-free (*P* ⟦*S*⟧ *Q*))
$\longleftrightarrow$ (*DF UNIV* :: ('*a*, '*r*) $process_{ptick}$) $\sqsubseteq_{FD}$ (*DF UNIV* ⟦*S*⟧ *DF UNIV*)› (**is** ‹*?lhs* $\longleftrightarrow$ *?rhs*›)
proof (*rule iffI*)
assume *?lhs*
show *?rhs* **by** (*fold deadlock-free-def*, *rule* ‹*?lhs*›[*rule-format*])
(*simp-all add*: *deadlock-free-def*)
next
assume *?rhs*
show *?lhs* **unfolding** *deadlock-free-def*
by (*intro allI impI trans-FD*[*OF* ‹*?rhs*›]) (*rule mono-Sync-FD*)
qed

From this general equivalence on *Sync*, we immediately obtain the equivalence on *A* ||| *B*: ($\forall$ *P Q*. *deadlock-free P* $\longrightarrow$ *deadlock-free Q* $\longrightarrow$ *deadlock-free* (*P* ||| *Q*)) = (*DF UNIV* $\sqsubseteq_{FD}$ *DF UNIV* ||| *DF UNIV*).

6.4.2 *STOP* and *SKIP* synchronized with *DF A*

lemma *DF-FD-DF-Sync-STOP-or-SKIP-iff*:
‹(*DF A* $\sqsubseteq_{FD}$ *DF A* ⟦*S*⟧ *P*) $\longleftrightarrow$ *A* $\cap$ *S* = {}›
if *P-disj*: ‹*P* = *STOP* $\vee$ *P* = *SKIP r*›
proof
{ assume *a1*: ‹*DF A* $\sqsubseteq_{FD}$ *DF A* ⟦*S*⟧ *P*› **and** *a2*: ‹*A* $\cap$ *S* $\neq$ {}›
from *a2* **obtain** *x* **where** *f1*: ‹*x* $\in$ *A*› **and** *f2*: ‹*x* $\in$ *S*› **by** *blast*
have ‹*DF A* ⟦*S*⟧ *P* $\sqsubseteq_{FD}$ *DF* {*x*} ⟦*S*⟧ *P*›
by (*intro mono-Sync-FD*[*OF - idem-FD*]) (*simp add*: *DF-subset f1*)
also have ‹... = *STOP*›
apply (*subst DF-unfold*)
using *P-disj* **apply** (*rule disjE*; *simp*)

apply (*simp add*: *write0-def*, *subst Mprefix-empty*[*symmetric*])
apply (*subst Mprefix-Sync-Mprefix-right*, (*simp-all add*: *f2*)[*3*])
by (*subst DF-unfold*, *simp add*: *f2 write0-Sync-SKIP*)
finally have *False*
by (*metis DF-Univ-freeness a1 empty-not-insert f1*
insert-absorb non-deadlock-free-STOP trans-FD)
}
thus ‹*DF A* $\sqsubseteq_{FD}$ *DF A* ⟦*S*⟧ *P* $\Longrightarrow$ *A* $\cap$ *S* = {}› **by** *blast*
next
have *D-P*: ‹$\mathcal{D}$ *P* = {}› **using** *D-SKIP*[*of r*] *D-STOP P-disj* **by** *blast*
note *F-T-P* = *F-STOP T-STOP F-SKIP D-SKIP*
{ assume *a1*: ‹$\neg$ *DF A* $\sqsubseteq_{FD}$ *DF A* ⟦*S*⟧ *P*› **and** *a2*: ‹*A* $\cap$ *S* = {}›
have *False*

```
    proof (cases ‹A = {}›)
      assume ‹A = {}›
      with a1[unfolded DF-def] that show ?thesis
        by (auto simp add: fix-const)
    next
      assume a3: ‹A ≠ {}›
      from a1 show ?thesis
       unfolding failure-divergence-refine-def failure-refine-def divergence-refine-def
      proof (auto simp add: F-Sync D-Sync D-P div-free-DF subset-iff, goal-cases)
        case (1 a t u X Y)
        then show ?case
        proof (induct t arbitrary: a)
          case Nil
          show ?case by (rule disjE[OF P-disj], insert Nil a2)
              (subst (asm) F-DF, auto simp add: a3 F-T-P)+
        next
          case (Cons x t)
          from Cons(4) have f1: ‹u = []›
            apply (subst disjE[OF P-disj], simp-all add: F-T-P)
           by (metis Cons.prems(1, 2, 4) F-T F-imp-front-tickFree Int-iff TickLeft-
Sync
                append-T-imp-tickFree inf-sup-absorb is-processT5-S7 list.distinct(1)
             non-tickFree-tick rangeI setinterleaving-sym tickFree-Cons-iff tickFree-Nil
tickFree-butlast)
          from Cons(2, 3) show False
            apply (subst (asm) (1 2) F-DF, auto simp add: a3)
            by (metis Cons.hyps Cons.prems(3, 4) setinterleaving-sym
                SyncTlEmpty emptyLeftProperty f1 list.sel(3))
        qed
      qed
    qed
  }
  thus ‹A ∩ S = {} ⟹ DF A ⊑FD DF A [S] P› by blast
qed


lemma DF-Sync-STOP-or-SKIP-FD-DF: ‹DF A [S] P ⊑FD DF A›
  if P-disj: ‹P = STOP ∨ P = SKIP r› and empty-inter: ‹A ∩ S = {}›
proof (cases ‹A = {}›)
  from P-disj show ‹A = {} ⟹ DF A [S] P ⊑FD DF A›
    by (rule disjE) (simp-all add: DF-def fix-const)
next
  assume ‹A ≠ {}›
  show ?thesis
  proof (subst DF-def, induct rule: fix-ind)
    show ‹adm (λa. a [S] P ⊑FD DF A)› by (simp add: cont2mono)
  next
    show ‹⊥ [S] P ⊑FD DF A› by (metis BOT-leFD Sync-BOT Sync-commute)
```

next
  case (*3 x*)
  have ‹(⊓*a* ∈ *A* → *x*) ⟦*S*⟧ *P* ⊑$_{FD}$ (*a* → *DF A*)› **if** ‹*a* ∈ *A*› **for** *a*
    find-theorems *Mndetprefix name*: *set*
    apply (*rule trans-FD*[*OF mono-Sync-FD*
          [*OF Mndetprefix-FD-subset*
            [*of* ‹{*a*}›, *simplified*, *OF that*] *idem-FD*]])
    apply (*rule disjE*[*OF P-disj*], *simp-all*)
     apply (*subst Mprefix-Sync-Mprefix-left*
        [*of* ‹{*a*}› - ‹{}› ‹λ*a*. *x*›, *simplified*, *folded write0-def*])
    using *empty-inter that* **apply** *blast*
    using *3 mono-write0-FD* **apply** *fast*
    by (*metis 3 disjoint-iff empty-inter mono-write0-FD that write0-Sync-SKIP*)
  thus *?case* **by** (*subst DF-unfold*, *subst FD-Mndetprefix-iff*; *simp add*: ‹*A* ≠ {}›)
qed
qed

lemmas *DF-FD-DF-Sync-STOP-iff* =
DF-FD-DF-Sync-STOP-or-SKIP-iff[*of STOP*, *simplified*]
and *DF-FD-DF-Sync-SKIP-iff* =
DF-FD-DF-Sync-STOP-or-SKIP-iff[*of* ‹*SKIP r*›, *simplified*]
and *DF-Sync-STOP-FD-DF* =
DF-Sync-STOP-or-SKIP-FD-DF[*of STOP*, *simplified*]
and *DF-Sync-SKIP-FD-DF* =
DF-Sync-STOP-or-SKIP-FD-DF[*of* ‹*SKIP r*›, *simplified*] **for** *r*

6.4.3 Finally, *deadlock-free* (*P* ||| *Q*)

theorem *DF-F-DF-Sync-DF*: ‹(*DF* (*A* ∪ *B*) :: ('*a*, '*r*) $process_{ptick}$) ⊑$_F$ *DF A* ⟦*S*⟧ *DF B*›
  if *nonempty*: ‹*A* ≠ {} ∧ *B* ≠ {}›
    and *intersect-hyp*: ‹*B* ∩ *S* = {} ∨ (∃ *y*. *B* ∩ *S* = {*y*} ∧ *A* ∩ *S* ⊆ {*y*})›
proof −
  let *?Z* = ‹*range tick* ∪ *ev* ' *S* :: ('*a*, '*r*) $event_{ptick}$ *set*›
  have ‹⟦(*t*, *X*) ∈ ℱ (*DF A*); (*u*, *Y*) ∈ ℱ (*DF B*); *v setinterleaves* ((*t*, *u*), *?Z*)⟧
      ⟹ (*v*, (*X* ∪ *Y*) ∩ *?Z* ∪ *X* ∩ *Y*) ∈ ℱ (*DF* (*A* ∪ *B*))› **for** *v t u* :: ‹('*a*, '*r*) $trace_{ptick}$› **and** *X Y*
  proof (*induct* ‹(*t*, *?Z*, *u*)› *arbitrary*: *t u v rule*: *setinterleaving.induct*)
    case (*1 v*)
    from *1.prems*(*3*) *emptyLeftProperty* **have** ‹*v* = []› **by** *blast*
    with *intersect-hyp 1.prems*(*1*, *2*) **show** *?case*
      by (*subst* (*asm*) (*1 2*) *F-DF*, *subst F-DF*)
        (*simp add*: *nonempty image-iff subset-iff*, *metis IntI empty-iff insertE*)
  next
    case (*2 y u*)
    from *2.prems*(*3*) *emptyLeftProperty* **obtain** *b*
      where ‹*b* ∉ *S*› ‹*y* = *ev b*› ‹*v* = *y* # *u*› ‹*u setinterleaves* (([], *u*), *?Z*)›

by (*cases y*) (*auto simp add*: *image-iff split*: *if-split-asm*)
from *2.prems*(*2*) **have** ‹b ∈ B› ‹(u, Y) ∈ ℱ (DF B)›
by (*subst* (*asm*) *F-DF*; *simp add*: ‹y = ev b› *nonempty*)+
have ‹(u, (X ∪ Y) ∩ ?Z ∪ X ∩ Y) ∈ ℱ (DF (A ∪ B))›
by (*rule 2.hyps*)
(*simp-all add*: *2.prems*(*1*) ‹(u, Y) ∈ ℱ (DF B)› ‹b ∉ S› ‹y = ev b›
‹u setinterleaves (([], u), ?Z)› *image-iff*)
thus *?case* **by** (*subst F-DF*) (*simp add*: *nonempty* ‹v = y # u› ‹y = ev b› ‹b ∈ B›)
next
case (*3 x t*)
from *3.prems*(*3*) *emptyRightProperty* **obtain** *a*
where ‹a ∉ S› ‹x = ev a› ‹v = x # t› ‹t setinterleaves ((t, []), ?Z)›
by (*cases x*) (*auto simp add*: *image-iff split*: *if-split-asm*)
from *3.prems*(*1*) **have** ‹a ∈ A› ‹(t, X) ∈ ℱ (DF A)›
by (*subst* (*asm*) *F-DF*; *simp add*: ‹x = ev a› *nonempty*)+
have ‹(t, (X ∪ Y) ∩ ?Z ∪ X ∩ Y) ∈ ℱ (DF (A ∪ B))›
by (*rule 3.hyps*)
(*simp-all add*: *3.prems*(*2*) ‹(t, X) ∈ ℱ (DF A)› ‹a ∉ S› ‹x = ev a›
‹t setinterleaves ((t, []), ?Z)› *image-iff*)
thus *?case* **by** (*subst F-DF*) (*simp add*: *nonempty* ‹v = x # t› ‹x = ev a› ‹a ∈ A›)
next
case (*4 x t y u*)
from *4.prems*(*1*) **obtain** *a* **where** ‹a ∈ A› ‹x = ev a› ‹(t, X) ∈ ℱ (DF A)›
by (*subst* (*asm*) *F-DF*) (*auto simp add*: *nonempty*)
from *4.prems*(*2*) **obtain** *b* **where** ‹b ∈ B› ‹y = ev b› ‹(u, Y) ∈ ℱ (DF B)›
by (*subst* (*asm*) *F-DF*) (*auto simp add*: *nonempty*)
consider ‹x ∈ ?Z› ‹y ∈ ?Z› | ‹x ∈ ?Z› ‹y ∉ ?Z›
| ‹x ∉ ?Z› ‹y ∈ ?Z› | ‹x ∉ ?Z› ‹y ∉ ?Z› **by** *blast*
thus *?case*
proof *cases*
assume ‹x ∈ ?Z› ‹y ∈ ?Z›
with *4.prems*(*3*) **obtain** *v′*
where ‹x = y› ‹v = y # v′› ‹v′ setinterleaves ((t, u), ?Z)›
by (*simp split*: *if-split-asm*) *blast*
from *4.hyps*(*1*)[*OF* ‹x ∈ ?Z› ‹y ∈ ?Z› ‹x = y›
‹(t, X) ∈ ℱ (DF A)› ‹(u, Y) ∈ ℱ (DF B)› *this*(*3*)]
have ‹(v′, (X ∪ Y) ∩ ?Z ∪ X ∩ Y) ∈ ℱ (DF (A ∪ B))› **.**
thus *?case* **by** (*subst F-DF*) (*simp add*: *nonempty* ‹v = y # v′› ‹y = ev b› ‹b ∈ B›)
next
assume ‹x ∈ ?Z› ‹y ∉ ?Z›
with *4.prems*(*3*) **obtain** *v′*
where ‹v = y # v′› ‹v′ setinterleaves ((x # t, u), ?Z)›
by (*simp split*: *if-split-asm*) *blast*
from *4.hyps*(*2*)[*OF* ‹x ∈ ?Z› ‹y ∉ ?Z› *4.prems*(*1*) ‹(u, Y) ∈ ℱ (DF B)› *this*(*2*)]
have ‹(v′, (X ∪ Y) ∩ ?Z ∪ X ∩ Y) ∈ ℱ (DF (A ∪ B))› **.**

thus *?case* **by** (*subst F-DF*) (*simp add*: *nonempty* ‹$v = y \# v'$› ‹$y = ev\ b$› ‹$b \in B$›)
next
assume ‹$x \notin ?Z$› ‹$y \in ?Z$›
with *4.prems*(*3*) **obtain** v'
where ‹$v = x \# v'$› ‹v' *setinterleaves* ((t, $y \# u$), $?Z$)›
by (*simp split*: *if-split-asm*) *blast*
from *4.prems*(*2*) *4.hyps*(*5*) ‹$x \notin ?Z$› ‹$y \in ?Z$› ‹(t, X) $\in \mathcal{F}$ (*DF A*)› *this*(*2*)
have ‹(v', ($X \cup Y$) $\cap$ $?Z \cup X \cap Y$) $\in \mathcal{F}$ (*DF* ($A \cup B$))› **by** *simp*
thus *?case* **by** (*subst F-DF*) (*simp add*: *nonempty* ‹$v = x \# v'$› ‹$x = ev\ a$› ‹$a \in A$›)
next
assume ‹$x \notin ?Z$› ‹$y \notin ?Z$›
with *4.prems*(*3*) **obtain** v'
where ‹$v = x \# v' \wedge v'$ *setinterleaves* ((t, $y \# u$), $?Z$) $\vee$
$v = y \# v' \wedge v'$ *setinterleaves* (($x \# t$, u), $?Z$)› **by** *auto*
thus *?case*
proof (*elim disjE conjE*)
assume ‹$v = x \# v'$› ‹v' *setinterleaves* ((t, $y \# u$), $?Z$)›
from *4.hyps*(*3*)[*OF* ‹$x \notin ?Z$› ‹$y \notin ?Z$› ‹(t, X) $\in \mathcal{F}$ (*DF A*)› *4.prems*(*2*) *this*(*2*)]
have ‹(v', ($X \cup Y$) $\cap$ $?Z \cup X \cap Y$) $\in \mathcal{F}$ (*DF* ($A \cup B$))› **.**
thus *?case* **by** (*subst F-DF*) (*simp add*: *nonempty* ‹$v = x \# v'$› ‹$x = ev\ a$› ‹$a \in A$›)
next
assume ‹$v = y \# v'$› ‹v' *setinterleaves* (($x \# t$, u), $?Z$)›
from *4.hyps*(*4*)[*OF* ‹$x \notin ?Z$› ‹$y \notin ?Z$› *4.prems*(*1*) ‹(u, Y) $\in \mathcal{F}$ (*DF B*)› *this*(*2*)]
have ‹(v', ($X \cup Y$) $\cap$ $?Z \cup X \cap Y$) $\in \mathcal{F}$ (*DF* ($A \cup B$))› **.**
thus *?case* **by** (*subst F-DF*) (*simp add*: *nonempty* ‹$v = y \# v'$› ‹$y = ev\ b$› ‹$b \in B$›)
qed
qed
qed

thus ‹*DF* ($A \cup B$) :: ($'a$, $'r$) $process_{ptick}$) $\sqsubseteq_F$ *DF A* ⟦*S*⟧ *DF B*›
by (*auto simp add*: *failure-refine-def F-Sync div-free-DF*)
qed

lemma *DF-FD-DF-Sync-DF*:
‹$A \neq \{\} \wedge B \neq \{\} \Longrightarrow B \cap S = \{\} \vee (\exists y.\ B \cap S = \{y\} \wedge A \cap S \subseteq \{y\}) \Longrightarrow$
DF ($A \cup B$) $\sqsubseteq_{FD}$ *DF A* ⟦*S*⟧ *DF B*›
unfolding *failure-divergence-refine-def failure-refine-def divergence-refine-def*
by (*simp add*: *div-free-DF D-Sync*)
(*simp add*: *DF-F-DF-Sync-DF*[*unfolded failure-refine-def*])

theorem *DF-FD-DF-Sync-DF-iff*:
‹*DF* ($A \cup B$) $\sqsubseteq_{FD}$ *DF A* ⟦*S*⟧ *DF B* $\longleftrightarrow$

(*if* $A = \{\}$ *then* $B \cap S = \{\}$
else if $B = \{\}$ *then* $A \cap S = \{\}$
else $A \cap S = \{\} \lor (\exists a.\ A \cap S = \{a\} \land B \cap S \subseteq \{a\}) \lor$
$B \cap S = \{\} \lor (\exists b.\ B \cap S = \{b\} \land A \cap S \subseteq \{b\})$)›
(**is** ‹*?FD-ref* ⟷ (*if* $A = \{\}$ *then* $B \cap S = \{\}$
else if $B = \{\}$ *then* $A \cap S = \{\}$
else ?cases)›)

apply (*cases* ‹$A = \{\}$›, *simp*,
metis DF-FD-DF-Sync-STOP-iff DF-unfold Sync-commute Mndetprefix-empty)
apply (*cases* ‹$B = \{\}$›, *simp*,
metis DF-FD-DF-Sync-STOP-iff DF-unfold Sync-commute Mndetprefix-empty)
proof (*simp*, *intro iffI*)
{ assume ‹$A \neq \{\}$› **and** ‹$B \neq \{\}$› **and** *?FD-ref* **and** ‹¬ *?cases*›
from ‹¬ *?cases*›[*simplified*]
obtain *a* **and** *b* **where** ‹$a \in A$› ‹$a \in S$› ‹$b \in B$› ‹$b \in S$› ‹$a \neq b$› **by** *blast*
have ‹*DF A* ⟦*S*⟧ *DF B* $\sqsubseteq_{FD}$ ($a \rightarrow$ *DF A*) ⟦*S*⟧ ($b \rightarrow$ *DF B*)›
by (*intro mono-Sync-FD*; *subst DF-unfold*, *meson Mndetprefix-FD-write0* ‹*a*
$\in A$› ‹$b \in B$›)
also have ‹... = *STOP*› **by** (*simp add*: ‹$a \in S$› ‹$a \neq b$› ‹$b \in S$› *write0-Sync-write0-subset*)
finally have *False*
by (*metis DF-Univ-freeness Un-empty* ‹$A \neq \{\}$›
trans-FD[*OF* ‹*?FD-ref*›] *non-deadlock-free-STOP*)**}**
thus ‹$A \neq \{\} \Longrightarrow B \neq \{\} \Longrightarrow$ *?FD-ref* ⟹ *?cases*› **by** *fast*
qed (*metis Sync-commute Un-commute DF-FD-DF-Sync-DF*)

lemma
‹$(\forall a \in A.\ X\ a \cap S = \{\} \lor (\forall b \in A.\ \exists y.\ X\ a \cap S = \{y\} \land X\ b \cap S \subseteq \{y\}))$
$\longleftrightarrow (\forall a \in A.\ \forall b \in A.\ \exists y.\ (X\ a \cup X\ b) \cap S \subseteq \{y\})$›
— this is the reason we write ugly_hyp this way
apply (*subst Int-Un-distrib2*, *auto*)
by (*metis subset-insertI*) *blast*

lemma *DF-FD-DF-MultiSync-DF*:
‹(*DF* ($\bigcup$ $x \in$ (*insert a A*). *X x*) :: ('*a*, '*r*) $process_{ptick}$) $\sqsubseteq_{FD}$ ⟦*S*⟧ $x \in\#$ *mset-set*
(*insert a A*). *DF* (*X x*)›
if *fin*: ‹*finite A*› **and** *nonempty*: ‹$X\ a \neq \{\}$› ‹$\forall b \in A.\ X\ b \neq \{\}$›
and *ugly-hyp*: ‹$\forall b \in A.\ X\ b \cap S = \{\} \lor (\exists y.\ X\ b \cap S = \{y\} \land X\ a \cap S \subseteq$
$\{y\})$›
‹$\forall b \in A.\ \forall c \in A.\ \exists y.\ (X\ b \cup X\ c) \cap S \subseteq \{y\}$›

apply (*rule conjunct1*[**where** Q = ‹$\forall b \in A.\ X\ b \cap S = \{\} \lor$
$(\exists y.\ X\ b \cap S = \{y\} \land \bigcup (X$ ‘ *insert a A*$) \cap S \subseteq \{y\})$›])

```
proof (induct rule: finite-subset-induct-singleton'
    [of a ‹insert a A›, simplified, OF fin,
      simplified subset-insertI, simplified])
  case 1
  show ?case by (simp add: ugly-hyp)
next
  case (2 b A')
  show ?case
  proof (rule conjI; subst image-insert, subst Union-insert)
    show ‹DF (X b ∪ ⋃ (X ‘ insert a A')) ⊑FD
        ⟦S⟧ a∈#mset-set (insert b (insert a A')). (DF (X a) :: ('a, 'r) process_ptick)›
      apply (subst Un-commute)
      apply (rule trans-FD[OF DF-FD-DF-Sync-DF[where S = S]])
        apply (simp add: 2.hyps(2) nonempty ugly-hyp(1))
      using 2.hyps(2, 5) apply blast
      apply (subst Sync-commute,
          rule trans-FD[OF mono-Sync-FD
            [OF idem-FD 2.hyps(5)[THEN conjunct1]]])
      by (simp add: 2.hyps(1, 4) mset-set-empty-iff)
  next
    show ‹∀ c ∈ A. X c ∩ S = {} ∨ (∃ y. X c ∩ S = {y} ∧
                (X b ∪ ⋃ (X ‘ insert a A')) ∩ S ⊆ {y})›
      apply (subst Int-Un-distrib2, subst Un-subset-iff)
      by (metis 2.hyps(2, 5) Int-Un-distrib2 Un-subset-iff
          subset-singleton-iff ugly-hyp(2))
  qed
qed
```

```
lemma DF-FD-DF-MultiSync-DF':
  ‹⟦finite A; ∀ a ∈ A. X a ≠ {}; ∀ a ∈ A. ∀ b ∈ A. ∃ y. (X a ∪ X b) ∩ S ⊆ {y}⟧
  ⟹ DF (⋃ a ∈ A. X a) ⊑FD ⟦S⟧ a ∈# mset-set A. DF (X a)›
  apply (cases A rule: finite.cases, assumption)
   apply (subst DF-unfold, simp)
  apply clarify
  apply (rule DF-FD-DF-MultiSync-DF)
  by simp-all (metis Int-Un-distrib2 Un-subset-iff subset-singleton-iff)
```

```
lemmas DF-FD-DF-MultiInter-DF =
  DF-FD-DF-MultiSync-DF'[where S = ‹{}›, simplified]
  and   DF-FD-DF-MultiPar-DF =
  DF-FD-DF-MultiSync-DF [where S = UNIV, simplified]
  and   DF-FD-DF-MultiPar-DF' =
  DF-FD-DF-MultiSync-DF'[where S = UNIV, simplified]
```

lemma ‹$DF\ \{a\} = DF\ \{a\}\ [\![S]\!]\ STOP \longleftrightarrow a \notin S$›
by (*metis DF-FD-DF-Sync-STOP-iff DF-Sync-STOP-FD-DF Diff-disjoint Diff-insert-absorb FD-antisym insert-disjoint*(*2*))

lemma ‹$DF\ \{a\}\ [\![S]\!]\ STOP = STOP \longleftrightarrow a \in S$›
by (*metis* (*no-types*, *lifting*) *DF-unfold Diff-disjoint Diff-eq-empty-iff Int-commute Int-insert-left Mndetprefix-Sync-STOP Mndetprefix-is-STOP-iff Ndet-is-STOP-iff empty-not-insert inf-le2*)

corollary *DF-FD-DF-Inter-DF*: ‹$DF\ A \sqsubseteq_{FD} DF\ A\ |||\ DF\ A$›
by (*metis DF-FD-DF-Sync-DF-iff inf-bot-right sup.idem*)

corollary *DF-UNIV-FD-DF-UNIV-Inter-DF-UNIV*:
‹$DF\ UNIV \sqsubseteq_{FD} DF\ UNIV\ |||\ DF\ UNIV$›
by (*fact DF-FD-DF-Inter-DF*)

corollary *Inter-deadlock-free*:
‹*deadlock-free* $P \Longrightarrow$ *deadlock-free* $Q \Longrightarrow$ *deadlock-free* ($P\ |||\ Q$)›
using *DF-FD-DF-Inter-DF deadlock-free-of-Sync-iff-DF-FD-DF-Sync-DF* **by** *blast*

theorem *MultiInter-deadlock-free*:
‹$[\![M \neq \{\#\}; \bigwedge m.\ m \in\# M \Longrightarrow$ *deadlock-free* $(P\ m)]\!] \Longrightarrow$
deadlock-free ($|||\ p \in\# M.\ P\ p$)›
proof (*induct rule*: *mset-induct-nonempty*)
case (*m-singleton a*) **thus** *?case* **by** *simp*
next
case (*add x F*) **with** *Inter-deadlock-free* **show** *?case* **by** *auto*
qed

Chapter 7

The Main Entry Point

This is the theory `HOL-CSPM` should be imported from.

Chapter 8

Example: Dining Philosophers

8.1 Classic version

We formalize here the Dining Philosophers problem with a locale.

locale *DiningPhilosophers* =

fixes *N*::*nat*
assumes *N-g1*[*simp*] : ‹*N* > *1*›
— We assume that we have at least one right handed philosophers (so at least two philosophers with the left handed one).

begin

We use a datatype for representing the dinner's events.

datatype *dining-event* = *picks* (*phil*:*nat*) (*fork*:*nat*)
| *putsdown* (*phil*:*nat*) (*fork*:*nat*)

We introduce the right handed philosophers, the left handed philosopher and the forks.

definition *RPHIL*:: ‹*nat* ⇒ *dining-event process*›
where ‹*RPHIL i* ≡ *μ X*. (*picks i i* → (*picks i* ((*i*−*1*) *mod N*) →
(*putsdown i* ((*i*−*1*) *mod N*) → (*putsdown i i* → *X*))))›

definition *LPHIL0*:: ‹*dining-event process*›
where ‹*LPHIL0* ≡ *μ X*. (*picks 0* (*N*−*1*) → (*picks 0 0* →
(*putsdown 0 0* → (*putsdown 0* (*N*−*1*) → *X*))))›

definition *FORK* :: ‹*nat* ⇒ *dining-event process*›
where ‹*FORK i* ≡ *μ X*. (*picks i i* → (*putsdown i i* → *X*)) □
(*picks* ((*i*+*1*) *mod N*) *i* → (*putsdown* ((*i*+*1*) *mod N*) *i* → *X*))›

Now we use the architectural operators for modelling the interleaving of the philosophers, and the interleaving of the forks.

definition ‹*PHILS* ≡ ||| *P* ∈# *add-mset LPHIL0* (*mset* (*map RPHIL* [*1*..< *N*])). *P*›
definition ‹*FORKS* ≡ ||| *P* ∈# *mset* (*map FORK* [*0*..< *N*]). *P*›

corollary ‹*N* = *3* ⟹ *PHILS* = (*LPHIL0* ||| *RPHIL 1* ||| *RPHIL 2*)›
— just a test
unfolding *PHILS-def* **by** (*simp add*: *eval-nat-numeral upt-rec Sync-assoc*)

Finally, the dinner is obtained by putting forks and philosophers in parallel.

definition *DINING* :: ‹*dining-event process*›
where ‹*DINING* = (*FORKS* || *PHILS*)›

end

8.2 Formalization with fixrec package

The fixrec package of HOLCF provides a more readable syntax (essentially, it allows us to "get rid of μ" in equations like μ *x*. *P x*).

First, we need to see *nat* as *cpo*.

instantiation *nat* :: *discrete-cpo*
begin

definition *below-nat-def*:
(*x*::*nat*) ⊑ *y* ⟷ *x* = *y*

instance proof
qed (*rule below-nat-def*)

end

locale *DiningPhilosophers-fixrec* =

fixes *N*::*nat*
assumes *N-g1*[*simp*] : ‹*N* > *1*›
— We assume that we have at least one right handed philosophers (so at least two philosophers with the left handed one).

begin

We use a datatype for representing the dinner's events.

datatype *dining-event* = *picks* (*phil*:*nat*) (*fork*:*nat*)
| *putsdown* (*phil*:*nat*) (*fork*:*nat*)

We introduce the right handed philosophers, the left handed philosopher and the forks.

fixrec *RPHIL* :: ‹*nat* → *dining-event process*›
and *LPHIL0* :: ‹*dining-event process*›
and *FORK* :: ‹*nat* → *dining-event process*›
where
RPHIL-rec [*simp del*] :
‹*RPHIL*·*i* = (*picks i i* → (*picks i* (*i*−*1*) →
(*putsdown i* (*i*−*1*) → (*putsdown i i* → *RPHIL*·*i*))))›
| *LPHIL0-rec*[*simp del*] :
‹*LPHIL0* = (*picks 0* (*N*−*1*) → (*picks 0 0* →
(*putsdown 0 0* → (*putsdown 0* (*N*−*1*) → *LPHIL0*))))›
| *FORK-rec* [*simp del*] :
‹*FORK*·*i* = (*picks i i* → (*putsdown i i* → *FORK*·*i*)) □
(*picks* ((*i*+*1*) *mod N*) *i* → (*putsdown* ((*i*+*1*) *mod N*) *i* → *FORK*·*i*))›

Now we use the architectural operators for modelling the interleaving of the philosophers, and the interleaving of the forks.

definition ‹*PHILS* ≡ ||| *P* ∈# *add-mset LPHIL0* (*mset* (*map* (λ*i*. *RPHIL*·*i*) [*1*..<*N*])). *P*›
definition ‹*FORKS* ≡ ||| *P* ∈# *mset* (*map* (λ*i*. *FORK*·*i*) [*0*..<*N*]). *P*›

corollary ‹*N* = *3* ⟹ *PHILS* = (*LPHIL0* ||| *RPHIL*·*1* ||| *RPHIL*·*2*)›
— just a test
unfolding *PHILS-def* **by** (*simp add*: *eval-nat-numeral upt-rec Sync-assoc*)

Finally, the dinner is obtained by putting forks and philosophers in parallel.

definition *DINING* :: ‹*dining-event process*›
where ‹*DINING* = (*FORKS* || *PHILS*)›

end

Chapter 9

Example: Plain Old Telephone System

The "Plain Old Telephone Service is a standard medium-size example for architectural modeling of a concurrent system.

Plain old telephone service (POTS), or plain ordinary telephone system,[1] is a retronym for voice-grade telephone service employing analog signal transmission over copper loops. POTS was the standard service offering from telephone companies from 1876 until 1988[2] in the United States when the Integrated Services Digital Network (ISDN) Basic Rate Interface (BRI) was introduced, followed by cellular telephone systems, and voice over IP (VoIP). POTS remains the basic form of residential and small business service connection to the telephone network in many parts of the world. The term reflects the technology that has been available since the introduction of the public telephone system in the late 19th century, in a form mostly unchanged despite the introduction of Touch-Tone dialing, electronic telephone exchanges and fiber-optic communication into the public switched telephone network (PSTN).

C.f. wikipedia https://en.wikipedia.org/wiki/Plain_old_telephone_service.

We need to see *int* as a *cpo*.

instantiation *int* :: *discrete-cpo*
begin

definition *below-int-def*:
  $(x::int) \sqsubseteq y \longleftrightarrow x = y$

instance proof
qed (*rule below-int-def*)

end

9.1 The Alphabet and Basic Types of POTS

Underlying terminology apparent in the acronyms:

1. T-side (target side, callee side)
2. O-side (originator (?) side, caller side)

datatype *MtcO* = *Osetup* | *Odiscon-o*
datatype *MctO* = *Obusy* | *Oalert* | *Oconnect* | *Odiscon-t*
datatype *MtcT* = *Tbusy* | *Talert* | *Tconnect* | *Tdiscon-t*
datatype *MctT* = *Tsetup* | *Tdiscon-o*

type-synonym *Phones* = ‹*int*›

datatype *channels* = *tcO* ‹*Phones* × *MtcO*› —
| *ctO* ‹*Phones* × *MctO*›
| *tcT* ‹*Phones* × *MtcT* × *Phones*›
| *ctT* ‹*Phones* × *MctT* × *Phones*›
| *tcOdial* ‹*Phones* × *Phones*›
| *StartReject* *Phones* — phone x rejects from now on to be called
| *EndReject* *Phones* — phone x accepts from now on to be called
| *terminal* *Phones*
| *off-hook* *Phones*
| *on-hook* *Phones*
| *digits* ‹*Phones* × *Phones*› — communication relation: x calls y
| *tone-ring* *Phones*
| *tone-quiet* *Phones*
| *tone-busy* *Phones*
| *tone-dial* *Phones*
| *connected* *Phones*

locale *POTS* =
fixes *min-phones* :: *int*
and *max-phones* :: *int*
and *VisibleEvents* :: ‹*channels set*›
assumes *min-phones-g-1*[*simp*] : ‹*1* ≤ *min-phones*›
and *max-phones-g-min-phones*[*simp*] : ‹*min-phones* < *max-phones*›
begin

definition *phones* :: ‹*Phones set*› **where** ‹*phones* ≡ {*min-phones* .. *max-phones*}›

lemma *nonempty-phones*[*simp*]: ‹*phones* ≠ {}›
and *finite-phones*[*simp*]: ‹*finite phones*›
and *at-least-two-phones*[*simp*]: ‹*2* ≤ *card phones*›
and *not-singl-phone*[*simp*]: ‹*phones* − {*p*} ≠ {}›
apply (*simp-all add*: *phones-def*)

using *max-phones-g-min-phones* **apply** *linarith+*
by (*metis atLeastAtMost-iff less-le-not-le max-phones-g-min-phones order-refl singletonD subsetD*)

definition *EventsIPhone* :: ‹*Phones* ⇒ *channels set*›
where ‹*EventsIPhone u* ≡ {*tone-ring u*, *tone-quiet u*, *tone-busy u*, *tone-dial u*, *connected u*}›
definition *EventsUser* :: ‹*Phones* ⇒ *channels set*›
where ‹*EventsUser u* ≡ {*off-hook u*, *on-hook u*} ∪ {*x* . ∃ *n*. *x* = *digits* (*u*, *n*)}›

9.2 Auxilliaries to Substructure the Specification

abbreviation
Tside-connected :: ‹*Phones* ⇒ *Phones* ⇒ *channels process*›
where ‹*Tside-connected ts os* ≡
(*ctT*!(*ts*,*Tdiscon-o*,*os*) → *tcT*!(*ts*,*Tdiscon-t*,*os*) → *EndReject*!*ts*→*Skip*)
▷ (*tcT*!(*ts*,*Tdiscon-t*,*os*) → *ctT*!(*ts*,*Tdiscon-o*,*os*) → *EndReject*!*ts*→*Skip*)›

abbreviation
Oside-connected :: ‹*Phones* ⇒ *channels process*›
where ‹*Oside-connected ts* ≡
(*ctO*!(*ts*,*Odiscon-t*) → *tcO*!(*ts*,*Odiscon-o*) → *EndReject*!*ts*→*Skip*)
▷ (*tcO*!(*ts*,*Odiscon-o*) → *ctO*!(*ts*,*Odiscon-t*) → *EndReject*!*ts*→*Skip*)›

abbreviation
Oside1 :: ‹[*Phones*, *Phones*] ⇒ *channels process*›
where
‹*Oside1 ts p* ≡ *tcOdial*!(*ts*,*p*)
→ (*ctO*!(*ts*,*Oalert*)
→ *ctO*!(*ts*,*Oconnect*)
→ (*Oside-connected ts*))
□(*ctO*!(*ts*,*Oconnect*) →(*Oside-connected ts*))
□(*ctO*!(*ts*,*Obusy*) → *tcO*!(*ts*,*Odiscon-o*) → *EndReject*!*ts* → *Skip*)›

definition
ITside-connected :: ‹[*Phones*,*Phones*,*channels process*] ⇒ *channels process*›
where
‹*ITside-connected ts os IT* ≡ (*ctT*(*ts*,*Tdiscon-o*,*os*)
→((*tone-busy*!*ts*
→ *on-hook*!*ts*
→ *tcT*!(*ts*,*Tdiscon-t*,*os*)

→ *EndReject*!*ts*
→ *IT*)
□ (*on-hook*!*ts*
→ *tcT*!(*ts*, *Tdiscon-t*, *os*)
→ *EndReject*!*ts*
→ *IT*)
))
□ (*on-hook*!*ts*
→ *tcT*!(*ts*, *Tdiscon-t*, *os*)
→ *ctT*!(*ts*, *Tdiscon-o*, *os*)
→ *EndReject*!*ts*
→*IT*)›

9.3 A Telephone

fixrec *T* :: ‹*Phones* → *channels process*›
and *Oside* :: ‹*Phones* → *channels process*›
and *Tside* :: ‹*Phones* → *channels process*›
and *NoReject* :: ‹*Phones* → *channels process*›
and *Reject* :: ‹*Phones* → *channels process*›
where
T-rec [*simp del*]: ‹*T*·*ts* = (*Tside*·*ts* **;** *T*·*ts*) ▷ (*Oside*·*ts* **;** *T*·*ts*)›
| *Oside-rec* [*simp del*]: ‹*Oside*·*ts* = *StartReject*!*ts*
→ *tcO*!(*ts*, *Osetup*)
→ (⊓ *p* ∈ *phones*. *Oside1 ts p*)›
| *Tside-rec* [*simp del*]: ‹*Tside*·*ts* = *ctT*?(*y*,*z*,*os*)|((*y*,*z*)=(*ts*, *Tsetup*))
→ *StartReject*!*ts*
→ (*tcT*!(*ts*, *Talert*, *os*)
→ *tcT*!(*ts*, *Tconnect*, *os*)
→(*Tside-connected ts os*)
⊓ (*tcT*!(*ts*, *Tconnect*, *os*)
→ (*Tside-connected ts os*)))›
| *NoReject-rec* [*simp del*]: ‹*NoReject*·*ts* = *StartReject*!*ts* → *Reject*·*ts*›
| *Reject-rec* [*simp del*]: ‹*Reject*·*ts* = *ctT*?(*y*,*z*,*os*)|(*y*=*ts* ∧ *z*=*Tsetup* ∧ *os*∈*phones* ∧ *os*≠*ts*)
→ (*tcT*!(*ts*, *Tbusy*, *os*) → *Reject*·*ts*)
□ (*EndReject*!*ts* → *NoReject*·*ts*)›

definition *Tel*:: ‹*Phones* ⇒ *channels process*›
where ‹*Tel p* ≡ (*T*·*p* ⟦{*StartReject p*, *EndReject p*}⟧ *NoReject*·*p*) \ {*StartReject p*, *EndReject p*}›

9.4 A Connector with the Network

```
fixrec    Call      :: ‹Phones → channels process›
  and BUSY      :: ‹Phones → Phones → channels process›
  and Connected :: ‹Phones → Phones → channels process›
  where
    Call-rec  [simp del]: ‹Call·os     = (tcO!  (os,Osetup) → tcOdial?(x,ts)|(x=os)
→ (BUSY·os·ts)) ; Call·os›
  | BUSY-rec  [simp del]: ‹BUSY·os·ts  = (if ts = os
                              then ctO!(os,Obusy) → tcO!(os,Odiscon-o) → Skip
                              else ctT!(ts,Tsetup,os)
                                   →( (tcT!(ts,Tbusy,os)
                                         → ctO!(os,Obusy)
                                         → tcO!(os,Odiscon-o) → Skip)
                                     □
                                       (tcT ! (ts,Talert,os)
                                         → ctO!(os,Oalert)
                                         → tcT!(ts,Tconnect,os)
                                         → ctO!(os,Oconnect)
                                         → Connected·os·ts)
                                     □
                                       (tcT!(ts,Tconnect,os)
                                         → ctO!(os,Oconnect)
                                         → Connected·os·ts)))›
  | Connected-rec [simp del]: ‹Connected·os·ts =  (tcO!(os,Odiscon-o) →
                             (( (ctT!(ts,Tdiscon-o,os) → tcT!(ts,Tdiscon-t,os) → Skip)
                                 □
                                (tcT!(ts,Tdiscon-t,os)→ ctT!(ts,Tdiscon-o,os) → Skip)
                              )
                              ; (ctO!(os,Odiscon-t) → Skip)))
                              □
                             (tcT!(ts,Tdiscon-t,os) →
                                   (  (ctO!(os,Odiscon-t)
                                       → ctT!(ts,Tdiscon-o,os)
                                       → tcO!(os,Odiscon-o)
                                       → Skip )
                                      □
                                      (tcO!(os,Odiscon-o)
                                       → ctT!(ts,Tdiscon-o,os)
                                       → ctO!(os,Odiscon-t)
                                       → Skip)
                                   )
                             )›
```

9.5 Combining NETWORK and TELEPHONES to a SYSTEM

```
definition  NETWORK     :: ‹channels process›
```

where ⟨*NETWORK* ≡ (||| *os* ∈# (*mset-set phones*). *Call*·*os*)⟩

definition *TELEPHONES* :: ⟨*channels process*⟩
where ⟨*TELEPHONES* ≡ (||| *ts* ∈# (*mset-set phones*). *Tel ts*)⟩

definition *SYSTEM* :: ⟨*channels process*⟩
where ⟨*SYSTEM* ≡ *NETWORK* ⟦*VisibleEvents*⟧ *TELEPHONES*⟩

We underline here the usefulness of the architectural operators, especially *MultiSync* but also *GlobalNdet* which appears in *Oside* recursive definition.

9.6 A simple Model of a User

fixrec *User* :: ⟨*Phones* → *channels process*⟩
and *UserSCon* :: ⟨*Phones* → *channels process*⟩
where
User-rec[*simp del*] : ⟨*User*·*u* = (*off-hook*!*u* →
(*tone-dial*!*u* →
(⊓ *p* ∈ *phones*. *digits*!(*u*,*p*)→*tone-quiet*!*u*→
((*tone-ring*!*u*→*connected*!*u*→*UserSCon*·*u*)
□ (*connected*!*u*→*UserSCon*·*u*)
□ (*tone-busy*!*u*→*on-hook*!*u*→*User*·*u*)
)
)
)
□ (*connected*!*u* → *UserSCon*·*u*)
)
□ (*tone-ring*!*u*→*off-hook*!*u*→*connected*!*u* →*UserSCon*·*u*)⟩
| *UserSCon-rec*[*simp del*]: ⟨*UserSCon*·*u* = (*tone-busy*!*u* → *on-hook*!*u* → *User*·*u*) ▷ (*on-hook*!*u* → *User*·*u*)⟩

fixrec *User-Ndet* :: ⟨*Phones* → *channels process*⟩
and *UserSCon-Ndet* :: ⟨*Phones* → *channels process*⟩
where
User-Ndet-rec[*simp del*] : ⟨*User-Ndet*·*u* = (*off-hook*!*u* →
(*tone-dial*!*u* →
(⊓ *p* ∈ *phones*. *digits*!(*u*,*p*)→*tone-quiet*!*u*→
((*tone-ring*!*u*→*connected*!*u*→*UserSCon-Ndet*·*u*)
⊓ (*connected*!*u*→*UserSCon-Ndet*·*u*)
⊓ (*tone-busy*!*u*→*on-hook*!*u*→*User-Ndet*·*u*)
)
)
)
⊓ (*connected*!*u* → *UserSCon-Ndet*·*u*)
)
⊓ (*tone-ring*!*u*→*off-hook*!*u*→*connected*!*u* →*UserSCon-Ndet*·*u*)⟩

| *UserSCon-Ndet-rec*[*simp del*]: ‹*UserSCon-Ndet*·*u* = (*tone-busy*!*u* → *on-hook*!*u* → *User-Ndet*·*u*) ⊓ (*on-hook*!*u* → *User-Ndet*·*u*)›

definition *ImplementT* :: ‹*Phones* ⇒ *channels process*›
where ‹*ImplementT ts* ≡ ((*Tel ts*) ⟦*EventsIPhone ts* ∪ *EventsUser ts*⟧ (*User*·*ts*)) \ (*EventsIPhone ts* ∪ *EventsUser ts*)›

9.7 Toplevel Proof-Goals

This has been proven in an ancient FDR model for *max-phones* = *5*...

lemma ‹∀ *p* ∈ *phones*. *deadlock-free* (*Tel p*)› **oops**
lemma ‹∀ *p* ∈ *phones*. *deadlock-free-v2* (*Call*·*p*)› **oops**
lemma ‹*deadlock-free-v2 NETWORK*› **oops**
lemma ‹*deadlock-free-v2 SYSTEM*› **oops**
lemma ‹*lifelock-free SYSTEM*› **oops**
lemma ‹∀ *p* ∈ *phones*. *lifelock-free* (*ImplementT p*)› **oops**
lemma ‹∀ *p* ∈ *phones*. *Tel p* $\sqsubseteq_{FD}$ *ImplementT p*› **oops**

lemma ‹∀ *p* ∈ *phones*. *Tel*′·*p* $\sqsubseteq_{F}$ *RUN UNIV*› **oops**

this should represent "deterministic" in process-algebraic terms. . .

end

Chapter 10

Conclusion

In this session, we defined three architectural operators: *GlobalDet*, *MultiSync*, and *MultiSeq* as respective generalizations of $P \square Q$, $P [\![S]\!] Q$, and $P ; Q$. The generalization of $P \sqcap Q$, *GlobalNdet*, is already in `HOL-CSP` since it is required for some algebraic laws.

We did this in a fully-abstract way, that is:

- $(\square)$ is commutative, idempotent and admits *STOP* as a neutral element so we defined *GlobalDet* on a *'a set A* by making it equal to *STOP* when $A = \emptyset$. Continuity only holds for finite cases, while the operator is always defined.

- $(\sqcap)$ is also commutative and idempotent so in `HOL-CSP` *GlobalNdet* has been defined on a *'a set A* by making it equal to *STOP* when $A = \emptyset$. Beware of the fact that *STOP* is not the neutral element for $(\sqcap)$ (this operator does not admit a neutral element) so we **do not have** the equality

  $\sqcap p{\in}\{a\}.\ P\ p = P\ a \sqcap (\sqcap p{\in}\emptyset.\ P\ p)$

  while this holds for $(\square)$ and *GlobalDet*). Again, continuity only holds for finite cases.

- *Sync* is commutative but is not idempotent so we defined *MultiSync* on a *'a multiset M* to keep the multiplicity of the processes. We made it equal to *STOP* when $M = \{\#\}$ but like $(\sqcap)$, *Sync* does not admit a neutral element so beware of the fact that in general

  $[\![S]\!]\ p{\in}\#\{\#a\#\}.\ P\ p \neq P\ a\ [\![S]\!]\ [\![S]\!]\ p{\in}\#\{\#\}.\ P\ p$

  . By construction, multiset are finite and therefore continuity holds.

- $(;)$ is neither commutative nor idempotent, and *SKIP r* is neutral only on the left hand side (note if the second type $'r$ of $('a, 'r)$ $process_{ptick}$ is actually *unit*, that is to say we go back to the old version without parameterized termination, it is neutral element on both sides, see *?P* ; *Skip* = *?P*

  SKIP ?r ; *?P* = *?P*). Therefore we defined *MultiSeq* on a *'a list L* to keep the multiplicity and the order of the processes, and the folding is done on the reversed list in order to enjoy the neutrality of *SKIP r* on the left hand side. For example, proving *SEQ p*∈@*L1*. *P p* ; *SEQ p*∈@*L2*. *P p* = *SEQ p*∈@(*L1* @ *L2*). *P p* in general requires $L2 \neq []$.

We presented two examples: Dining philosophers, and POTS.

In both, we underlined the usefulness of the architectural operators for modeling complex systems.

Finally we provided powerful results on *events-of* and *deadlock-free* among which the most important is undoubtedly:

$[\![M \neq \{\#\}; \bigwedge m.\ m \in\# M \Longrightarrow \textit{deadlock-free}\ (P\ m)]\!]$
$\Longrightarrow \textit{deadlock-free}\ (|||\ p{\in}\#M.\ P\ p)$

This theorem allows, for example, to establish:

$0 < n \Longrightarrow \textit{deadlock-free}\ (|||\ m{\in}\#\textit{mset}\ [0..{<}n].\ P\ m)$

under the assumption that a family of processes parameterized by *m* :: *nat* verifies $\forall\, m{<}n.\ \textit{deadlock-free}\ (P\ m)$.

More recently, two operators *Throw* and $(\triangle)$ have been added. The corresponding continuities and algebraic laws can also be found in this session.

Bibliography

[1] 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.

[2] A. Roscoe. *Theory and Practice of Concurrency.* Prentice Hall, 1998.

[3] A. W. Roscoe. Understanding concurrent systems. In *Texts in Computer Science*, 2010.

[4] 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.