HOL-CSPM - Architectural operators for HOL-CSP

Benoît Ballenghien Safouan Taha Burkhart Wolff

April 18, 2024

Abstract

Recently, a modern version of Roscoes and Brookes [1] Failure-Divergence Semantics for CSP has been formalized in Isabelle [3].

The session HOL-CSP introduces among other things some binary operators on processes that we will here generalize in a fully-abstract way.

On these "architectural operators", we will prove the properties of refinement, the rules of continuity and the laws of interaction so that they can be easily used.

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

Contents

Chapter 1

Introduction

1.1 Motivations

`HOL-CSP` [3] 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 $'\alpha$ *process*, 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.

In this session we therefore introduce the multi-operators associated respectively with $P \sqcap Q$, $P \square Q$, $P [\![S]\!] Q$ and $P \mathbin{;} Q$. We prove their continuity and refinements rules, as well as some laws governing their interactions.

We also give 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* of CSP.

Finally this session also includes results on the notion of *events-of*, and a very powerful result about *deadlock-free* and *Sync*: the interleaving $P|||Q$ is *deadlock-free* if P and Q are.

1.2 The Global Architecture of HOL-CSPM

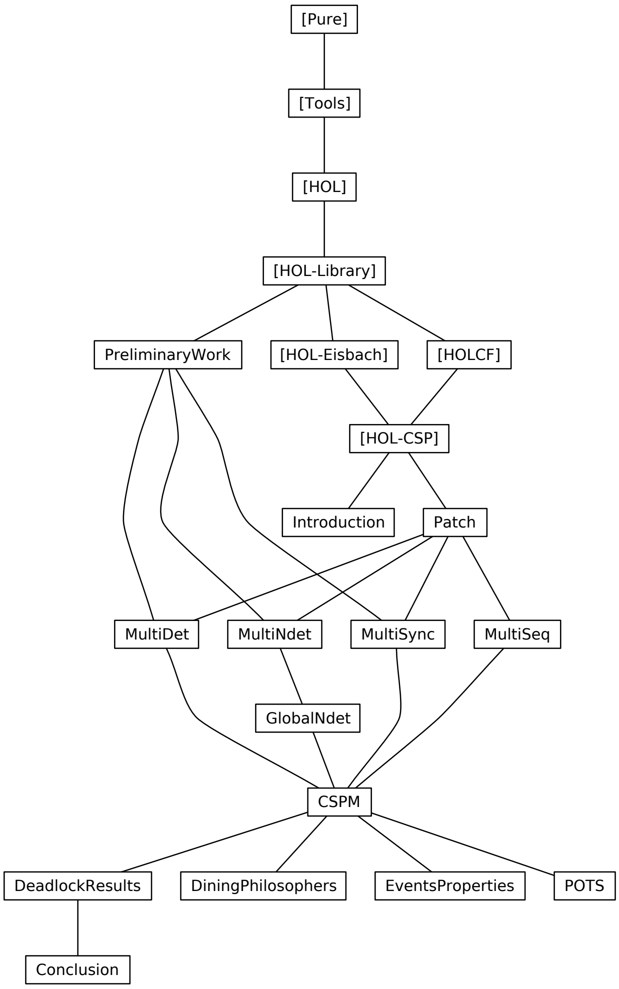

Figure 1.1: The overall architecture

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

Some Preliminary Work

theory *PreliminaryWork*
  imports *HOL−Library.Multiset*
begin

2.1 Induction Rules for $'\alpha$ *set*

lemma *finite-subset-induct-singleton*
      [*consumes 3*, *case-names singleton insertion*]:
  ‹⟦$a \in A$; *finite F*; $F \subseteq A$; $P\ \{a\}$;
    $\bigwedge x\ F.$ *finite F* $\Longrightarrow x \in A \Longrightarrow x \notin$ (*insert a F*) $\Longrightarrow P$ (*insert a F*)
        $\Longrightarrow P$ (*insert x* (*insert a F*))⟧ $\Longrightarrow P$ (*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 ‹$A \neq \{\}$› **and** ‹*finite A*›
    and *singleton*: ‹$\bigwedge a.\ a \in A \Longrightarrow P\ \{a\}$›
    and *insert*: ‹$\bigwedge x\ F.$ ⟦$F \neq \{\}$; *finite F*; $x \in A$; $x \notin F$; $P\ F$⟧
                $\Longrightarrow P$ (*insert x F*)›
    shows ‹$P\ A$›
proof−
  obtain $a\ A'$ **where** ‹$a \in A$› ‹*finite* A'› ‹$A' \subseteq A$› ‹$A =$ *insert a* A'›
    using ‹$A \neq \{\}$› ‹*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*: ‹$a \in A$› ‹*finite* A'› ‹$A' \subseteq A$› *singleton insert*)
qed

lemma *finite-subset-induct-singleton'*
      [*consumes 3*, *case-names singleton insertion*]:
  ‹⟦$a \in A$; *finite F*; $F \subseteq A$; $P\ \{a\}$;

⋀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. P {a};
⋀F a. ⟦a ∈ A; *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 ‹$F \subseteq\# A$›
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*]:
‹⟦$a \in\# A - F$; $F \subseteq\# A$; P {#a#};
$\bigwedge x\ F$. ⟦$x \in\# A - F$; $F \subseteq\# A$; P (*add-mset a F*)⟧
$\Longrightarrow P$ (*add-mset x* (*add-mset a F*))⟧
$\Longrightarrow P$ (*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* $\subseteq\# A$; P {#a#};
$\bigwedge x\ F$. ⟦*add-mset x* (*add-mset a F*) $\subseteq\# A$; P (*add-mset a F*)⟧
$\Longrightarrow P$ (*add-mset x* (*add-mset a F*))⟧
$\Longrightarrow P$ (*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*: ‹$A \neq$ {#}› **and** *m-singleton*: ‹$\bigwedge a$. $a \in\# A \Longrightarrow P$ {#a#}›
and *hyp*: ‹$\bigwedge a\ x\ F$. ⟦$a \in\# A$; $x \in\# A -$ *add-mset a F*; *add-mset a F* $\subseteq\# A$;
P (*add-mset a F*)⟧ $\Longrightarrow P$ (*add-mset x* (*add-mset a F*))›
shows ‹$P\ A$›
proof−
obtain $a\ A'$ **where** ‹$A =$ *add-mset a A′*› ‹*add-mset a A′* $\subseteq\# A$›
using *nonempty multiset-cases subset-mset-def* **by** *auto*
show ‹$P\ A$›
apply (*subst* ‹$A =$ *add-mset a A′*›)
using ‹*add-mset a A′* $\subseteq\# A$›
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* ∈# *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 Preliminaries for Cartesian Product Results

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*

end

Chapter 3

Patch for Compatibility

```
theory Patch
  imports HOL−CSP.Assertions
begin
```

HOL-CSP significantly changed during the past months, but not all the modifications appear in the current version on the AFP. This theory fixes the incompatibilities and will be removed in the next release.

3.1 Results

```
lemma Mprefix-Det-distr:
  ‹(□ a ∈ A → P a) □ (□ b ∈ B → Q b) =
   □ x ∈ A ∪ B → (  if x ∈ A ∩ B then P x ⊓ Q x
                    else if x ∈ A then P x else Q x)›
  (is ‹?lhs = ?rhs›)
proof (subst Process-eq-spec, safe)
  show ‹(s, X) ∈ ℱ ?lhs ⟹ (s, X) ∈ ℱ ?rhs› for s X
    by (simp add: F-Det F-Mprefix F-Ndet disjoint-iff, safe, auto)
next
  show ‹(s, X) ∈ ℱ ?rhs ⟹ (s, X) ∈ ℱ ?lhs› for s X
    by (auto simp add: F-Mprefix F-Ndet F-Det split: if-split-asm)
next
  show ‹s ∈ 𝒟 ?lhs ⟹ s ∈ 𝒟 ?rhs› for s
    by (simp add: D-Det D-Mprefix D-Ndet, safe, auto)
next
  show ‹s ∈ 𝒟 ?rhs ⟹ s ∈ 𝒟 ?lhs› for s
    by (auto simp add: D-Mprefix D-Ndet D-Det split: if-split-asm)
qed

lemma D-expand :
  ‹𝒟 P = {t1 @ t2| t1 t2. t1 ∈ 𝒟 P ∧ tickFree t1 ∧ front-tickFree t2}›
  (is ‹𝒟 P = ?rhs›)
proof (intro subset-antisym subsetI)
  show ‹s ∈ 𝒟 P ⟹ s ∈ ?rhs› for s
```

apply (*simp*, *cases* ‹*tickFree s*›)
apply (*rule-tac x* = *s* **in** *exI*, *rule-tac x* = ‹[]› **in** *exI*, *simp*)
apply (*rule-tac x* = ‹*butlast s*› **in** *exI*, *rule-tac x* = ‹[*tick*]› **in** *exI*, *simp*)
by (*metis front-tickFree-implies-tickFree nonTickFree-n-frontTickFree process-charn snoc-eq-iff-butlast*)
next
show ‹*s* ∈ *?rhs* ⟹ *s* ∈ 𝒟 *P*› **for** *s*
using *is-processT7* **by** *blast*
qed

3.1.1 Continuity Rule

Monotonicity of ***Renaming*****.**

lemma *mono-Renaming*[*simp*] : ‹(*Renaming P f*) ⊑ (*Renaming Q f*)› **if** ‹*P* ⊑ *Q*›
proof (*unfold le-approx-def*, *intro conjI subset-antisym subsetI allI impI*)
show ‹*s* ∈ 𝒟 (*Renaming Q f*) ⟹ *s* ∈ 𝒟 (*Renaming P f*)› **for** *s*
using *that*[*THEN le-approx1*] **by** (*auto simp add*: *D-Renaming*)
next
show ‹*s* ∉ 𝒟 (*Renaming P f*) ⟹
$X \in \mathcal{R}_a$ (*Renaming P f*) *s* ⟹ $X \in \mathcal{R}_a$ (*Renaming Q f*) *s*› **for** *s X*
using *that*[*THEN le-approx2*] **apply** (*simp add*: *Ra-def D-Renaming F-Renaming*)
by (*metis* (*no-types*, *opaque-lifting*) *append.right-neutral butlast.simps*(*2*)
front-tickFree-butlast front-tickFree-mono list.distinct(*1*)
map-EvExt-tick map-append nonTickFree-n-frontTickFree process-charn)
next
show ‹*s* ∉ 𝒟 (*Renaming P f*) ⟹
$X \in \mathcal{R}_a$ (*Renaming Q f*) *s* ⟹ $X \in \mathcal{R}_a$ (*Renaming P f*) *s*› **for** *s X*
using *that*[*THEN le-approx2*] *that*[*THEN le-approx1*]
by (*simp add*: *Ra-def D-Renaming F-Renaming subset-iff*)
(*metis is-processT8-S*)
next
fix *s*
assume ∗ : ‹*s* ∈ *min-elems* (𝒟 (*Renaming P f*))›
from *elem-min-elems*[*OF* ∗] **obtain** *s1 s2*
where ∗∗ : ‹*tickFree s1*› ‹*front-tickFree s2*›
‹*s* = *map* (*EvExt f*) *s1* @ *s2*› ‹*s1* ∈ 𝒟 *P*›
‹*front-tickFree s*›
using *D-imp-front-tickFree D-Renaming* **by** *blast*
with *min-elems-no*[*OF* ∗, *of* ‹*butlast s*›] **have** ‹*s2* = []›
by (*cases s rule*: *rev-cases*; *simp add*: *D-Renaming*)
(*metis butlast-append butlast-snoc front-tickFree-butlast less-self nless-le tickFree-implies-front-tickFree*)
with ∗∗ *min-elems-no*[*OF* ∗, *of* ‹*butlast s*›] **have** ‹*s1* ∈ *min-elems* (𝒟 *P*)›
apply (*cases s*; *simp add*: *D-Renaming Nil-min-elems*)
by (*metis* (*no-types*, *lifting*) *D-imp-front-tickFree append.right-neutral butlast-snoc*
front-tickFree-charn less-self list.discI
list.map-disc-iff map-butlast min-elems1 nless-le)
hence ‹*s1* ∈ 𝒯 *Q*› **using** *that*[*THEN le-approx3*] **by** *blast*

show ‹$s \in \mathcal{T}$ (*Renaming Q f*)›
  apply (*simp add*: ∗∗(*3*) ‹*s2* = []› *T-Renaming*)
  using ‹*s1* $\in \mathcal{T}$ *Q*› **by** *blast*
qed

Useful Results about *finitary*, and Preliminaries Lemmas for Continuity.

lemma *le-snoc-is* : ‹$t \leq s$ @ [x] $\longleftrightarrow$ $t = s$ @ [x] $\vee$ $t \leq s$›
 by (*metis append-eq-first-pref-spec le-list-def order.trans self-append-conv*)

lemma *Cont-RenH5*: ‹*finite* ($\bigcup t \in \{t.\ t \leq (s :: {'}\alpha\ trace)\}.\ \{s.\ t{=}map\ (EvExt\ f)\ s\}$)› **if** ‹*finitary f*›
 apply (*rule finite-UN-I*)
  apply (*induct s rule*: *rev-induct*)
   apply (*simp*; *fail*)
  apply (*simp add*: *le-snoc-is*; *fail*)
 using *Cont-RenH2 Cont-RenH4 that* **by** *blast*

lemma *Cont-RenH7*:
 ‹*finite* $\{t.\ \exists t2.\ tickFree\ t \wedge front\text{-}tickFree\ t2 \wedge s = map\ (EvExt\ f)\ t\ @\ t2\}$›
 if ‹*finitary f*›
proof −
 have *f1*: ‹$\{y.\ map\ (EvExt\ f)\ y \leq s\}$ =
   ($\bigcup z \in \{z.\ z \leq s\}.\ \{y.\ z = map\ (EvExt\ f)\ y\}$)› **by** *fast*
 show *?thesis*
  apply (*rule finite-subset*[*OF Cont-RenH6*])
  apply (*simp add*: *f1*)
  apply (*rule finite-UN-I*)
   apply (*induct s rule*: *rev-induct*)
    apply (*simp*; *fail*)
   apply (*simp add*: *le-snoc-is*; *fail*)
  using *Cont-RenH2 Cont-RenH4 that* **by** *blast*
qed

Finally, Continuity !

lemma *Cont-Renaming-prem*:
 ‹($\bigsqcup$ *i*. *Renaming* (*Y i*) *f*) = (*Renaming* ($\bigsqcup$ *i*. *Y i*) *f*)› **if** *finitary*: ‹*finitary f*›
  and *chain*: ‹*chain Y*›
proof (*subst Process-eq-spec*, *safe*)
 fix *s*
 define *S* **where** ‹$S\ i \equiv \{s1.\ \exists t2.\ tickFree\ s1 \wedge front\text{-}tickFree\ t2 \wedge$
   $s = map\ (EvExt\ f)\ s1\ @\ t2 \wedge s1 \in \mathcal{D}\ (Y\ i)\}$› **for** *i*
 assume ‹$s \in \mathcal{D}$ ($\bigsqcup$*i*. *Renaming* (*Y i*) *f*)›
 hence ‹$s \in \mathcal{D}$ (*Renaming* (*Y i*) *f*)› **for** *i*
  by (*simp add*: *limproc-is-thelub chain chain-Renaming D-LUB*)

 hence ‹$\forall i.\ S\ i \neq \{\}$› **by** (*simp add*: *S-def D-Renaming*) *blast*

moreover have ‹*finite* (*S 0*)›
  unfolding *S-def*
  by (*rule finite-subset*[*OF - Cont-RenH7*[*OF finitary*, *of s*]], *blast*)
moreover have ‹∀ *i*. *S* (*Suc i*) ⊆ *S i*›
  unfolding *S-def* **using** *NF-ND po-class.chainE*[*OF chain*]
    proc-ord2a le-approx-def **by** *blast*
ultimately have ‹(⋂ *i*. *S i*) ≠ {}›
  by (*rule Inter-nonempty-finite-chained-sets*)

thus ‹*s* ∈ 𝒟 (*Renaming* (*Lub Y*) *f*)›
  by (*simp add*: *limproc-is-thelub chain D-Renaming*
    D-LUB ex-in-conv[*symmetric*] *S-def*) *blast*
next
  show ‹*s* ∈ 𝒟 (*Renaming* (*Lub Y*) *f*) ⟹ *s* ∈ 𝒟 (⨆ *i*. *Renaming* (*Y i*) *f*)› **for** *s*
  by (*auto simp add*: *limproc-is-thelub chain chain-Renaming D-Renaming D-LUB*)
next
  fix *s X*
  define *S* **where** ‹*S i* ≡ {*s1*. (*s1*, *EvExt f* −‘ *X*) ∈ ℱ (*Y i*) ∧ *s* = *map* (*EvExt f*) *s1*} ∪
    {*s1*. ∃ *t2*. *tickFree s1* ∧ *front-tickFree t2* ∧
      s = *map* (*EvExt f*) *s1* @ *t2* ∧ *s1* ∈ 𝒟 (*Y i*)}› **for** *i*
  assume ‹(*s*, *X*) ∈ ℱ (⨆ *i*. *Renaming* (*Y i*) *f*)›
  hence ‹(*s*, *X*) ∈ ℱ (*Renaming* (*Y i*) *f*)› **for** *i*
    by (*simp add*: *limproc-is-thelub chain-Renaming*[*OF chain*] *F-LUB*)

  hence ‹∀ *i*. *S i* ≠ {}› **by** (*auto simp add*: *S-def F-Renaming*)
  moreover have ‹*finite* (*S 0*)›
    unfolding *S-def*
    apply (*subst finite-Un*, *intro conjI*)
     apply (*rule finite-subset*[*of -* ‹{*s1*. *s* = *map* (*EvExt f*) *s1*}›], *blast*)
     apply (*insert Cont-RenH2 Cont-RenH4 finitary*, *blast*)
    by (*rule finite-subset*[*OF - Cont-RenH7*[*OF finitary*, *of s*]], *blast*)
  moreover have ‹∀ *i*. *S* (*Suc i*) ⊆ *S i*›
    unfolding *S-def* **using** *NF-ND po-class.chainE*[*OF chain*]
      proc-ord2a le-approx-def **by** *safe blast+*
  ultimately have ‹(⋂ *i*. *S i*) ≠ {}›
    by (*rule Inter-nonempty-finite-chained-sets*)

  thus ‹(*s*, *X*) ∈ ℱ (*Renaming* (*Lub Y*) *f*)›
    by (*simp add*: *F-Renaming limproc-is-thelub chain D-LUB*
      F-LUB ex-in-conv[*symmetric*] *S-def*) (*meson NF-ND*)
next
  show ‹(*s*, *X*) ∈ ℱ (*Renaming* (*Lub Y*) *f*) ⟹ (*s*, *X*) ∈ ℱ (⨆ *i*. *Renaming* (*Y i*) *f*)› **for** *s X*
   by (*auto simp add*: *limproc-is-thelub chain chain-Renaming F-Renaming D-LUB F-LUB*)
qed

3.1.2 Nice Properties

lemma *Renaming-inv*: ‹*Renaming* (*Renaming P f*) (*inv f*) = *P*› **if** ‹*inj f*›
  apply (*subst Renaming-comp*[*symmetric*])
  apply (*subst inv-o-cancel*[*OF that*])
  by (*fact Renaming-id*)

3.1.3 Renaming Laws

lemma *Renaming-Mprefix-inj-on*:
  ‹*Renaming* (*Mprefix A P*) *f* = □ *b* ∈ *f* ‘ *A* → *Renaming* (*P* (*THE a. a* ∈ *A* ∧ *f a* = *b*)) *f*›
  if *inj-on-f*: ‹*inj-on f A*›
  apply (*subst Renaming-Mprefix*[*symmetric*])
  apply (*intro arg-cong*[**where** *f* = ‹λ*Q*. *Renaming Q f*›] *mono-Mprefix-eq*)
  by (*metis that the-inv-into-def the-inv-into-f-f*)

corollary *Renaming-Mprefix-inj*:
  ‹*Renaming* (*Mprefix A P*) *f* = □ *b* ∈ *f* ‘ *A* → *Renaming* (*P* (*THE a. f a* = *b*)) *f*› **if** *inj-f*: ‹*inj f*›
  apply (*subst Renaming-Mprefix-inj-on*, *metis inj-eq inj-onI that*)
  apply (*rule mono-Mprefix-eq*[*rule-format*])
  by (*metis imageE inj-eq*[*OF inj-f*])

corollary *Renaming-Mndetprefix-inj-on*:
  ‹*Renaming* (*Mndetprefix A P*) *f* = ⊓ *b* ∈ *f* ‘ *A* → *Renaming* (*P* (*THE a. a* ∈ *A* ∧ *f a* = *b*)) *f*›
  if *inj-on-f*: ‹*inj-on f A*›
  apply (*subst Renaming-Mndetprefix*[*symmetric*])
  apply (*intro arg-cong*[**where** *f* = ‹λ*Q*. *Renaming Q f*›] *mono-Mndetprefix-eq*)
  by (*metis that the-inv-into-def the-inv-into-f-f*)

corollary *Renaming-Mndetprefix-inj*:
  ‹*Renaming* (*Mndetprefix A P*) *f* = ⊓ *b* ∈ *f* ‘ *A* → *Renaming* (*P* (*THE a. f a* = *b*)) *f*›
  if *inj-f*: ‹*inj f*›
  apply (*subst Renaming-Mndetprefix-inj-on*, *metis inj-eq inj-onI that*)
  apply (*rule mono-Mndetprefix-eq*[*rule-format*])
  by (*metis imageE inj-eq*[*OF inj-f*])

3.2 Assertions

abbreviation *deadlock-free$_{SKIP}$* :: *'a process* ⇒ *bool*
where *deadlock-free$_{SKIP}$* ≡ *deadlock-free-v2*

lemma *deadlock-free-implies-lifelock-free*: ‹*deadlock-free P* ⟹ *lifelock-free P*›
unfolding *deadlock-free-def lifelock-free-def*
using *CHAOS-DF-refine-FD trans-FD* **by** *blast*

lemmas *deadlock-free$_{SKIP}$-def* = *deadlock-free-v2-def*
and *deadlock-free$_{SKIP}$-is-right* = *deadlock-free-v2-is-right*
and *deadlock-free$_{SKIP}$-implies-div-free* = *deadlock-free-v2-implies-div-free*
and *deadlock-free$_{SKIP}$-FD* = *deadlock-free-v2-FD*
and *deadlock-free$_{SKIP}$-is-right-wrt-events* = *deadlock-free-v2-is-right-wrt-events*
and *deadlock-free-is-deadlock-free$_{SKIP}$* = *deadlock-free-is-deadlock-free-v2*
and *deadlock-free$_{SKIP}$-SKIP* = *deadlock-free-v2-SKIP*
and *non-deadlock-free$_{SKIP}$-STOP* = *non-deadlock-free-v2-STOP*

3.3 Lifelock Freeness

definition *lifelock-free$_{SKIP}$* :: *'a process* ⇒ *bool*
where *lifelock-free$_{SKIP}$ P* ≡ *CHAOS$_{SKIP}$ UNIV* $\sqsubseteq_{FD}$ *P*

lemma *div-free-is-lifelock-free$_{SKIP}$*: *lifelock-free$_{SKIP}$ P* ⟷ $\mathcal{D}$ *P* = {}
using *CHAOS$_{SKIP}$-has-all-failures-Un leFD-imp-leD leF-leD-imp-leFD*
div-free-divergence-refine(*1*) *lifelock-free$_{SKIP}$-def*
by *blast*

lemma *lifelock-free-is-lifelock-free$_{SKIP}$*: *lifelock-free P* ⟹ *lifelock-free$_{SKIP}$ P*
by (*simp add*: *leFD-imp-leD div-free-divergence-refine*(*2*) *div-free-is-lifelock-free$_{SKIP}$*
lifelock-free-def)

corollary *deadlock-free$_{SKIP}$-is-lifelock-free$_{SKIP}$*: *deadlock-free$_{SKIP}$ P* ⟹ *life-lock-free$_{SKIP}$ P*
by (*simp add*: *deadlock-free$_{SKIP}$-implies-div-free div-free-is-lifelock-free$_{SKIP}$*)

3.4 New Laws

lemma *non-terminating-Sync*:
‹*non-terminating P* ⟹ *lifelock-free$_{SKIP}$ Q* ⟹ *non-terminating* (*P* ⟦*A*⟧ *Q*)›
apply (*simp add*: *non-terminating-is-right div-free-is-lifelock-free$_{SKIP}$ T-Sync*)
apply (*intro ballI*, *simp add*: *non-terminating-is-right nonterminating-implies-div-free*)
by (*metis empty-iff ftf-Sync1 ftf-Sync21 insertI1 tickFree-def*)

lemmas *non-terminating-Par* = *non-terminating-Sync*[**where** *A* = ‹*UNIV*›]
and *non-terminating-Inter* = *non-terminating-Sync*[**where** *A* = ‹{}›]

syntax
-writeS :: ['b ⇒ 'a, pttrn, 'b set, 'a process] => 'a process ((4(-!-|-) /→ -) [0,0,50,78] 50)
translations
-writeS c p b P => *CONST Mndetprefix* (c ‘ {p. b}) (λ-. P)

end

Chapter 4

The MultiDet Operator

```
theory MultiDet
  imports Patch PreliminaryWork
begin
```

4.1 Definition

```
definition  MultiDet :: ‹['a set, 'a ⇒ 'b process] ⇒ 'b process›
  where     ‹MultiDet A P = Finite-Set.fold (λa r. r □ P a) STOP A›


syntax -MultiDet :: ‹[pttrn, 'a set, 'b process] ⇒ 'b process›  (‹(3□-∈-. / -)› 75)
translations □ p ∈ A. P  ⇌  CONST MultiDet A (λp. P)
```

4.2 First Properties

```
lemma MultiDet-rec0[simp]: ‹(□ p ∈ {}. P p) = STOP›
  by(simp add: MultiDet-def)

lemma MultiDet-rec1[simp]: ‹(□ p ∈ {a}. P p) = P a›
  unfolding MultiDet-def
  apply (subst comp-fun-commute-on.fold-insert-remove[where S = ‹{a}›])
  by (simp-all add: comp-fun-commute-on-def
                    Det-commute[of ‹STOP›, simplified Det-STOP])


lemma MultiDet-in-id[simp]:
  ‹a ∈ A ⟹ (□ p ∈ insert a A. P p) = □ p ∈ A. P p›
  unfolding MultiDet-def by (simp add: insert-absorb)


lemma MultiDet-insert[simp]:
  ‹finite A ⟹ (□ p ∈ insert a A. P p) = P a □ (□ p ∈ A − {a}. P p)›
  unfolding MultiDet-def
```

apply (*subst comp-fun-commute-on.fold-insert-remove*[**where** S = ‹*insert a A*›])
unfolding *comp-fun-commute-on-def comp-def*
apply (*metis Det-assoc Det-commute*)
by (*auto simp*: *comp-fun-commute-on-def Det-commute Det-assoc comp-def*)

lemma *MultiDet-insert′*[*simp*]:
‹*finite A* ⟹ (□ *p* ∈ *insert a A*. *P p*) = (*P a* □ (□ *p* ∈ *A*. *P p*))›
by (*cases* ‹*a* ∈ *A*›, *metis MultiDet-insert Det-assoc Det-id insert-absorb*, *simp*)

lemma *mono-MultiDet-eq*:
‹*finite A* ⟹ ∀ *x* ∈ *A*. *P x* = *Q x* ⟹ *MultiDet A P* = *MultiDet A Q*›
by (*induct A rule*: *induct-subset-empty-single*, *simp*, *simp*)
(*metis MultiDet-insert′ insertCI*)

4.3 Some Tests

lemma *test-MultiDet*: ‹(□ *p* ∈ {*1*::*int* .. *3*}. *P p*) = *P 1* □ *P 2* □ *P 3*›
proof −
have ‹{*1*::*int* .. *3*} = *insert 1* (*insert 2* (*insert 3* {}))› **by** *fastforce*
thus ‹(□ *p* ∈ {*1*::*int* .. *3*}. *P p*) = *P 1* □ *P 2* □ *P 3*› **by** (*simp add*: *Det-assoc*)
qed

lemma *test-MultiDet′*:
‹(□ *p* ∈ {*0*::*nat* .. *a*}. *P p*) = (□ *p* ∈ {*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)

4.4 Continuity

lemma *MultiDet-cont*[*simp*]:
‹⟦*finite A*; ∀ *x* ∈ *A*. *cont* (*P x*)⟧ ⟹ *cont* (λ*y*. □ *z*∈*A*. *P z y*)›
by (*rule Finite-Set.finite-subset-induct*[*of A A*], *simp*+)

4.5 Factorization of (□) in front of *MultiDet*

lemma *MultiDet-factorization-union*:
‹⟦*finite A*; *finite B*⟧ ⟹ (□ *p* ∈ *A*. *P p*) □ (□ *p* ∈ *B*. *P p*) = □ *p* ∈ *A* ∪ *B* . *P p*›
apply (*erule finite-induct*, *simp-all*)
by (*metis Det-commute Det-STOP*)
(*metis MultiDet-insert MultiDet-insert′ Det-assoc finite-UnI*)

4.6 ⊥ Absorbance

lemma *MultiDet-BOT-absorb*:
  assumes *fin*: ‹*finite A*› **and** *bot*: ‹*P a* = ⊥› **and** *dom*: ‹*a* ∈ *A*›
  shows ‹(□ *x* ∈ *A*. *P x*) = ⊥›
  apply(*rule rev-mp*[*OF dom*], *rule rev-mp*[*OF bot*])
  by (*metis Det-commute MultiDet-insert′ Det-BOT fin insert-absorb*)

lemma *MultiDet-is-BOT-iff*:
  ‹*finite A* ⟹ *MultiDet A P* = ⊥ ⟷ (∃ *a* ∈ *A*. *P a* = ⊥)›
  by (*induct A rule*: *finite-induct*) (*auto simp add*: *STOP-neq-BOT Det-is-BOT-iff*)

4.7 First Properties

lemma *MultiDet-id*: ‹*A* ≠ {} ⟹ *finite A* ⟹ (□*p* ∈ *A*. *P*) = *P*›
  by (*erule finite-set-induct-nonempty*, *simp-all add*: *Det-id*)

lemma *MultiDet-STOP-id*: ‹*finite A* ⟹ (□*p* ∈ *A*. *STOP*) = *STOP*›
  by (*cases* ‹*A* = {}›) (*simp-all add*: *MultiDet-id*)

lemma *MultiDet-STOP-neutral*:
  ‹*finite A* ⟹ *P a* = *STOP* ⟹ (□ *z* ∈ *insert a A*. *P z*) = □ *z* ∈ *A*. *P z*›
  by (*metis Det-commute MultiDet-insert′ Det-STOP*)

lemma *MultiDet-is-STOP-iff*:
  ‹*finite A* ⟹ (□ *a* ∈ *A*. *P a*) = *STOP* ⟷ *A* = {} ∨ (∀ *a* ∈ *A*. *P a* = *STOP*)›
  by (*induct rule*: *finite-induct*) (*auto simp add*: *Det-is-STOP-iff*)

4.8 Behaviour of *MultiDet* with (□)

lemma *MultiDet-Det*:
  ‹*finite A* ⟹ (□ *a* ∈ *A*. *P a*) □ (□ *a* ∈ *A*. *Q a*) = □ *a* ∈ *A*. *P a* □ *Q a*›
proof (*induct A rule*: *finite-induct*)
  case *empty* **show** *?case* **by** (*simp add*: *Det-id*)
next
  case (*insert x F*)
  hence ‹*MultiDet* (*insert x F*) *P* □ *MultiDet* (*insert x F*) *Q* =
        P x □ *MultiDet F P* □ *Q x* □ *MultiDet F Q*› **by** (*simp add*: *Det-assoc*)
  also have ‹. . . = (*P x* □ *Q x*) □ (□ *a*∈*F*. *P a* □ *Q a*)›
    by (*metis* (*no-types*, *lifting*) *Det-assoc Det-commute insert.hyps*(*3*))
  ultimately show ‹*MultiDet* (*insert x F*) *P* □ *MultiDet* (*insert x F*) *Q* =
        (□ *a*∈*insert x F*. *P a* □ *Q a*)›
    by (*simp add*: ‹*finite F*› ‹*x* ∉ *F*›)
qed

4.9 Commutativity

lemma *MultiDet-sets-commute*:
‹⟦*finite A*; *finite B*⟧ ⟹ (□ *a* ∈ *A*. □ *b* ∈ *B*. *P a b*) = □ *b* ∈ *B*. □ *a* ∈ *A*. *P a b*›
by (*induct A rule*: *finite-induct*) (*simp-all add*: *MultiDet-STOP-id MultiDet-Det*)

4.10 Behaviour with Injectivity

lemma *inj-on-mapping-over-MultiDet*:
‹⟦*finite A*; *inj-on f A*⟧ ⟹ (□ *x* ∈ *A*. *P x*) = □ *x* ∈ *f* ‘ *A*. *P* (*inv-into A f x*)›
proof (*induct A rule*: *induct-subset-empty-single*)
case *1*
thus *?case* **by** *force*
next
case *2*
thus *?case* **by** *force*
next
case (*3 F a*)
hence *f1*: ‹*inv-into* (*insert a F*) *f* (*f a*) = *a*› **by** *force*
show *?case*
apply (*simp add*: *3.hyps*(*2*) *3.hyps*(*4*) *f1 del*: *MultiDet-insert*)
apply (*rule arg-cong*[**where** *f* = ‹λ*x*. *P a* □ *x*›])
apply (*subst 3.hyps*(*5*), *rule inj-on-subset*[*OF 3.prems subset-insertI*])
apply (*rule mono-MultiDet-eq*, *simp add*: *3.hyps*(*2*))
using *3.prems* **by** *fastforce*
qed

4.11 The Projections

lemma *D-MultiDet*: ‹*finite A* ⟹ 𝒟 (□ *x* ∈ *A*. *P x*) = (⋃ *p* ∈ *A*. 𝒟 (*P p*))›
by (*induct rule*: *finite-induct*) (*simp-all add*: *D-Det D-STOP*)

lemma *T-MultiDet*:
‹*finite A* ⟹ 𝒯 (□ *x* ∈ *A*. *P x*) = (*if A* = {} *then* {[]} *else* ⋃ *p* ∈ *A*. 𝒯 (*P p*))›
apply (*simp add*: *T-STOP*, *intro impI*, *rotate-tac*)
by (*induct rule*: *finite-set-induct-nonempty*) (*simp-all add*: *T-Det T-STOP*)

4.12 Cartesian Product Results

lemma *MultiDet-cartprod-σs-set-σs-set*:

‹⟦*finite A*; *finite B*; ∀ *s* ∈ *A*. *length s* = len_1⟧ ⟹
(□ (*s*, *t*) ∈ *A* × *B*. *P* (*s* @ *t*)) = □ *u* ∈ {*s* @ *t* |*s t*. (*s*, *t*) ∈ *A* × *B*}. *P u*›
apply (*subst inj-on-mapping-over-MultiDet*[**where** *f* = ‹λ (*s*, *t*). *s* @ *t*›],
simp-all add: *inj-on-def*)
apply (*subst prem-Multi-cartprod*(*1*)[*simplified*, *symmetric*])
apply (*rule mono-MultiDet-eq*, *simp add*: *finite-image-set2*)
by (*metis* (*no-types*, *lifting*) *case-prod-unfold f-inv-into-f*)

```
lemma MultiDet-cartprod-s-set-σs-set:
  ‹⟦finite A; finite B⟧ ⟹
   (□ (s, t) ∈ A × B. P (s # t)) = □ u ∈ {s # t |s t. (s, t) ∈ A × B}. P u›
  apply (subst inj-on-mapping-over-MultiDet[where f = ‹λ (s, t). s # t›],
        simp-all add: inj-on-def)
  apply (subst prem-Multi-cartprod(2)[simplified, symmetric])
  apply (rule mono-MultiDet-eq, simp add: finite-image-set2)
  by (metis (no-types, lifting) case-prod-unfold f-inv-into-f)


lemma MultiDet-cartprod-s-set-s-set:
  ‹⟦finite A; finite B⟧ ⟹
   (□ (s, t) ∈ A × B. P [s, t]) = □ u ∈ {[s, t] |s t. (s, t) ∈ A × B}. P u›
  apply (subst inj-on-mapping-over-MultiDet[where f = ‹λ (s, t). [s, t]›],
        simp-all add: inj-on-def)
  apply (subst prem-Multi-cartprod(3)[simplified, symmetric])
  apply (rule mono-MultiDet-eq, simp add: finite-image-set2)
  by (metis (no-types, lifting) case-prod-unfold f-inv-into-f)


lemma MultiDet-cartprod:
  ‹finite A ⟹ finite B ⟹ (□ (s, t) ∈ A × B. P s t) = □ s∈A. □ t∈B. P s t›
  supply arg-cong-Det = arg-cong[where f = ‹λQ. - □ Q›]
  supply MultiDet-insert[simp del]
proof (induct ‹card A› arbitrary: A B rule: nat-less-induct)
  case (1 A B)
  from ‹finite A› ‹finite B› consider ‹A = {}› | ‹B = {}› |
     ‹∃ mA mB a b A' B'. A = insert a A' ∧ B = insert b B' ∧ mA = card A' ∧
                        mB = card B' ∧ mA < card A ∧ mB < card B›
    by (metis card-Diff1-less-iff ex-in-conv insert-Diff)
  thus ‹(□(x, y)∈A × B.  P x y) = □s∈A.  MultiDet B (P s)›
  proof cases
    show ‹A = {} ⟹ (□(x, y)∈A × B. P x y) = □s∈A. MultiDet B (P s)›
      by simp
  next
    show ‹B = {} ⟹ (□(x, y)∈A × B. P x y) = □s∈A. MultiDet B (P s)›
      by (simp add: MultiDet-STOP-id[OF 1.prems(1)])
  next
    assume ‹∃ mA mB a b A' B'. A = insert a A' ∧ B = insert b B' ∧
           mA = card A' ∧ mB = card B' ∧ mA < card A ∧ mB < card B›
    then obtain mA mB a b A' B'
      where ∗ : ‹A = insert a A'› ‹B = insert b B'› ‹mA = card A'›
            ‹mB = card B'› ‹mA < card A› ‹mB < card B› by blast
    have ∗∗ : ‹Pair a ‘ B' = {a} × B'› unfolding image-def by blast
    show ‹(□(x, y)∈A × B.  P x y) = □s∈A.  MultiDet B (P s)›
      using ∗(1, 2) ‹finite A› ‹finite B› apply simp
      apply (subst MultiDet-factorization-union[symmetric], simp-all)
```

apply (*subst 1(1)*[*rule-format*, *OF* ∗(*5*, *3*)], *simp-all*)
apply (*simp add*: *MultiDet-Det*[*symmetric*])
apply (*subst Det-assoc*, *rule arg-cong-Det*)
apply (*subst* (*3*) *Det-commute*, *rule arg-cong-Det*)
apply (*subst inj-on-mapping-over-MultiDet*[*of B′* ‹λ*b*. (*a*, *b*)›],
simp-all add: *inj-on-def* ∗∗)
apply (*rule mono-MultiDet-eq*)
apply (*simp*; *fail*)
by (*metis* ∗∗ *case-prod-conv f-inv-into-f*)
qed
qed

end

Chapter 5

The MultiNdet Operator

theory *MultiNdet*
  imports *Patch PreliminaryWork*
begin

5.1 Definition

Defining the multi operator of (⊓) requires more work than with (□) since there is no neutral element. We will first build a version on *'α list* that we will generalize to *'α set*.

fun *MultiNdet-list* :: ‹['a list, 'a ⇒ 'b process] ⇒ 'b process›
  where ‹*MultiNdet-list* [] $P = STOP$›
  | ‹*MultiNdet-list* $(a \# l)$ $P = fold\ (\lambda x\ r.\ r \sqcap P\ x)\ l\ (P\ a)$›

syntax -*MultiNdet-list* :: ‹[pttrn,'a set,'b process] ⇒ 'b process›
  (‹(3⨅$_l$ -∈-. / -)› 76)
translations $\bigsqcap_l\ p \in l.\ P \rightleftharpoons$ *CONST MultiNdet-list* l $(\lambda p.\ P)$

interpretation *MultiNdet*: *comp-fun-idem* **where** f=‹$\lambda x\ r.\ r \sqcap P\ x$›
  unfolding *comp-fun-idem-def comp-fun-commute-def*
    comp-fun-idem-axioms-def comp-def
  by (*metis Ndet-commute Ndet-assoc Ndet-id*)

lemma *MultiNdet-list-set*:
  ‹$set\ L = set\ L' \Longrightarrow$ *MultiNdet-list* $L\ P$ = *MultiNdet-list* $L'\ P$›
  apply (*cases* L, *simp-all*)
proof −
  fix $a\ \ l$
  assume $*$: ‹$insert\ a\ (set\ l) = set\ L'$› **and** $**$: ‹$L = a \# l$›
  then obtain $a'\ l'$ **where** $***$: ‹$L' = a' \# l'$› **by** (*metis insertI1 list.set-cases*)
  note $* = *$[*simplified* $***$, *simplified*]

```
  have a0: ‹MultiNdet-list L P =
           Finite-Set.fold (λx r. r ⊓ P x) (P a) (set L − {a})›
    by (metis ** List.finite-set MultiNdet.fold-fun-left-comm
             MultiNdet.fold-insert-idem2 MultiNdet.fold-rec
             MultiNdet.fold-set-fold MultiNdet-list.simps(2)
             insert-iff list.simps(15) Ndet-id set-removeAll)
  have a11: ‹a′ ∈ set L›
   and a12: ‹a ≠ a′ ⟹ insert a′ (set L − {a, a′}) = set L − {a}›
    by (auto simp add: * **)
  hence a2: ‹Finite-Set.fold (λx r. r ⊓ P x) (P a) (insert a′ (set L − {a, a′})) =
            Finite-Set.fold (λx r. r ⊓ P x) (P a ⊓ P a′) (set L − {a, a′})›
    by (subst MultiNdet.fold-insert-idem2, simp-all)
  have a3:‹MultiNdet-list L′ P =
          Finite-Set.fold (λx r. r ⊓ P x) (P a′) (set L′ − {a′})›
    by (metis *** List.finite-set MultiNdet.fold-fun-left-comm
             MultiNdet.fold-insert-idem2 MultiNdet.fold-rec
             MultiNdet.fold-set-fold MultiNdet-list.simps(2)
             insert-iff list.simps(15) Ndet-id set-removeAll)
  have a41: ‹a ∈ set L′›
   and a42: ‹a ≠ a′ ⟹ insert a (set L′ − {a, a′}) = set L′ − {a′}›
    using * *** by auto
  hence a5: ‹Finite-Set.fold (λx r. r ⊓ P x) (P a′) (insert a (set L′ − {a, a′}))
          = Finite-Set.fold (λx r. r ⊓ P x) (P a ⊓ P a′) (set L′ − {a, a′})›
    by (subst MultiNdet.fold-insert-idem2, simp-all add: Ndet-commute)
  have a6: ‹(set L − {a, a′}) = (set L′ − {a, a′})›
    using * ** *** by force
  from * ** *** a0 a11 a12 a2 a3 a41 a42 a5 a6
  show ‹fold (λx r. r ⊓ P x) l (P a) = MultiNdet-list L′ P›
    by (metis MultiNdet-list.simps(2) list.simps(15))
qed


definition MultiNdet :: ‹['a set, 'a ⇒ 'b process] ⇒ 'b process›
  where   ‹MultiNdet A P = MultiNdet-list (SOME L. set L = A) P›


syntax -MultiNdet :: ‹[pttrn, 'a set, 'b process] ⇒ 'b process› (‹(3⨅ -∈-. / -)› 76)
translations ⨅ p ∈ A. P ⇌ CONST MultiNdet A (λp. P)
```

5.2 First Properties

```
lemma MultiNdet-rec0[simp]: ‹(⨅ p ∈ {}. P p) = STOP›
  by(simp add: MultiNdet-def)


lemma MultiNdet-rec1[simp]: ‹(⨅ p ∈ {a}. P p) = P a›
  unfolding MultiNdet-def apply (rule someI2[of - ‹[a]›], simp)
  by (rule MultiNdet-list-set[where L′ = ‹[a]›, simplified])
```

```
lemma MultiNdet-in-id[simp]:
  ‹a ∈ A ⟹ (⨅ p ∈ insert a A. P p) = ⨅ p ∈ A. P p›
  unfolding MultiNdet-def by (simp add: insert-absorb)


lemma MultiNdet-insert[simp]:
  assumes fin: ‹finite A› and notempty: ‹A ≠ {}› and notin: ‹a ∉ A›
  shows ‹(⨅ p ∈ insert a A. P p) = P a ⊓ (⨅ p ∈ A. P p)›
  unfolding MultiNdet-def
  apply (rule someI2-ex, simp add: fin finite-list)+
proof−
  fix l l'
  assume ‹set l = A› and ‹set l' = insert a A›
  from notempty and ‹set l = A› have ‹l ≠ []› by fastforce
  then have ‹MultiNdet-list (a # l) P = P a ⊓ MultiNdet-list l P›
    proof(induct l rule: List.list-nonempty-induct)
    case (single x)
    show ‹MultiNdet-list [a, x] P = P a ⊓ MultiNdet-list [x] P› by simp
  next
    case (cons x xs)
    have ‹MultiNdet-list (a # x # xs) P = P a ⊓ ((MultiNdet-list xs P) ⊓ P x)›
      by (metis List.finite-set MultiNdet.fold-insert-idem MultiNdet.fold-set-fold
              MultiNdet-list.simps(2) cons.hyps(2) list.simps(15) Ndet-assoc)
    thus ‹MultiNdet-list (a # x # xs) P = P a ⊓ MultiNdet-list (x # xs) P›
    proof −
      have f1: ‹MultiNdet-list (a # x # xs) P =
              Finite-Set.fold (λa p. p ⊓ P a) (P x ⊓ P a) (set xs)›
        by (simp add: MultiNdet.fold-set-fold Ndet-commute)
      have ‹MultiNdet-list (x # xs) P =
          Finite-Set.fold (λa p. p ⊓ P a) (P x) (set xs)›
        by (simp add: MultiNdet.fold-set-fold)
      hence ‹MultiNdet-list (a # x # xs) P = MultiNdet-list (x # xs) P ⊓ P a›
        using f1 by (simp add: MultiNdet.fold-fun-left-comm)
      thus ‹MultiNdet-list (a # x # xs) P = P a ⊓ MultiNdet-list (x # xs) P›
        by (simp add: Ndet-commute)
    qed
  qed
  moreover have ‹set l' = set (a # l)›
    by (simp add: ‹set l = A› ‹set l' = insert a A›)
  ultimately show ‹MultiNdet-list l' P = P a ⊓ MultiNdet-list l P›
    by (metis MultiNdet-list-set)
qed


lemma MultiNdet-insert'[simp]:
  ‹⟦finite A; A ≠ {}⟧ ⟹ (⨅ p ∈ insert a A. P p) = P a ⊓ (⨅ p ∈ A. P p)›
  apply (cases ‹a ∈ A›, subst Set.insert-absorb, simp-all)
  apply (cases ‹A = {a}›, simp add: Ndet-id)
```

proof −
  assume ‹*finite* A› **and** ‹$a \in A$› **and** ‹$A \neq \{a\}$›
  then obtain A' **where** ‹$A' \neq \{\}$› ‹$A = insert\ a\ A'$› ‹$a \notin A'$› ‹*finite* A'›
    by (*metis Set.set-insert finite-insert*)
  hence ‹$MultiNdet\ A\ P = P\ a \sqcap MultiNdet\ A'\ P$› **by** *simp*
  hence ‹$MultiNdet\ A\ P = P\ a \sqcap P\ a \sqcap MultiNdet\ A'\ P$› **by** (*simp add*: *Ndet-id*)
  thus ‹$MultiNdet\ A\ P = P\ a \sqcap MultiNdet\ A\ P$› **by** (*metis Ndet-id Ndet-assoc*)
qed

lemma *mono-MultiNdet-eq*:
  ‹$finite\ A \Longrightarrow \forall x \in A.\ P\ x = Q\ x \Longrightarrow MultiNdet\ A\ P = MultiNdet\ A\ Q$›
  by (*induct A rule*: *induct-subset-empty-single*; *simp*)

5.3 Some Tests

lemma ‹$(\bigsqcap_l\ p \in [].\ P\ p) = STOP$›
  and ‹$(\bigsqcap_l\ p \in [a].\ P\ p) = P\ a$›
  and ‹$(\bigsqcap_l\ p \in [a,\ b].\ P\ p) = P\ a \sqcap P\ b$›
  and ‹$(\bigsqcap_l\ p \in [a,\ b,\ c].\ P\ p) = P\ a \sqcap P\ b \sqcap P\ c$›
  by *auto*

lemma ‹$(\bigsqcap\ p \in \{\}.\ P\ p) = STOP$›
  and ‹$(\bigsqcap\ p \in \{a\}.\ P\ p) = P\ a$›
  and ‹$(\bigsqcap\ p \in \{a,\ b\}.\ P\ p) = P\ a \sqcap P\ b$›
  and ‹$(\bigsqcap\ p \in \{a,\ b,\ c\}.\ P\ p) = P\ a \sqcap P\ b \sqcap P\ c$›
  by (*simp add*: *Ndet-assoc*)+

lemma *test-MultiNdet*: ‹$(\bigsqcap\ p \in \{1{::}int\ ..\ 3\}.\ P\ p) = P\ 1 \sqcap P\ 2 \sqcap P\ 3$›
proof −
  have ‹$\{1{::}int\ ..\ 3\} = insert\ 1\ (insert\ 2\ (insert\ 3\ \{\}))$› **by** *fastforce*
  thus ‹$(\bigsqcap\ p \in \{1{::}int\ ..\ 3\}.\ P\ p) = P\ 1 \sqcap P\ 2 \sqcap P\ 3$› **by** (*simp add*: *Ndet-assoc*)
qed

lemma *test-MultiNdet'*:
  ‹$(\bigsqcap\ p \in \{0{::}nat\ ..\ a\}.\ P\ p) = (\bigsqcap\ p \in \{a\} \cup \{1\ ..\ a\} \cup \{0\}\ .\ P\ p)$›
  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)

5.4 Continuity

lemma *MultiNdet-cont*[*simp*]:
  ‹$[\![finite\ A;\ \forall x \in A.\ cont\ (P\ x)]\!] \Longrightarrow cont\ (\lambda y.\ \bigsqcap\ z{\in}A.\ P\ z\ y)$›
  by (*cases* ‹$A = \{\}$›, *simp*, *erule finite-set-induct-nonempty*; *simp*)

5.5 Factorization of $(\sqcap)$ in front of *MultiNdet*

lemma *MultiNdet-factorization-union*:
‹⟦$A \neq \{\}$; *finite* A; $B \neq \{\}$; *finite* B⟧ $\Longrightarrow$
$(\prod\ p \in A.\ P\ p) \sqcap (\prod\ p \in B.\ P\ p) = \prod\ p \in A \cup B\ .\ P\ p$›
by (*erule finite-set-induct-nonempty*, *simp-all add*: *Ndet-assoc*)

5.6 ⊥ Absorbance

lemma *MultiNdet-BOT-absorb*:
assumes *fin*: ‹*finite* A› **and** *bot*: ‹$P\ a = \bot$› **and** *dom*: ‹$a \in A$›
shows ‹$(\prod\ x \in A.\ P\ x) = \bot$›
apply(*rule rev-mp*[*OF dom*], *rule rev-mp*[*OF bot*])
by (*metis MultiNdet-insert MultiNdet-rec1 Ndet-commute fin finite-insert mk-disjoint-insert Ndet-BOT*)

lemma *MultiNdet-is-BOT-iff*:
‹*finite* $A \Longrightarrow (\prod\ p \in A.\ P\ p) = \bot \longleftrightarrow (\exists\, a \in A.\ P\ a = \bot)$›
apply (*cases* ‹$A = \{\}$›, *simp add*: *STOP-neq-BOT*)
by (*rotate-tac*, *induct A rule*: *finite-set-induct-nonempty*) (*simp-all add*: *Ndet-is-BOT-iff*)

5.7 First Properties

lemma *MultiNdet-id*: ‹$A \neq \{\} \Longrightarrow$ *finite* $A \Longrightarrow (\prod\ p \in A.\ P) = P$›
by (*erule finite-set-induct-nonempty*, (*simp-all add*: *Ndet-id*))

lemma *MultiNdet-STOP-id*: ‹*finite* $A \Longrightarrow (\prod\ p \in A.\ STOP) = STOP$›
by (*cases* ‹$A = \{\}$›) (*simp-all add*: *MultiNdet-id*)

lemma *MultiNdet-is-STOP-iff*:
‹*finite* $A \Longrightarrow (\prod\ p \in A.\ P\ p) = STOP \longleftrightarrow A = \{\} \lor (\forall\, a \in A.\ P\ a = STOP)$›
apply (*cases* ‹$A = \{\}$›, *simp*)
by (*rotate-tac*, *induct A rule*: *finite-set-induct-nonempty*) (*simp-all add*: *Ndet-is-STOP-iff*)

5.8 Behaviour of *MultiNdet* with $(\sqcap)$

lemma *MultiNdet-Ndet*:
‹*finite* $A \Longrightarrow (\prod\ a \in A.\ P\ a) \sqcap (\prod\ a \in A.\ Q\ a) = \prod\ a \in A.\ P\ a \sqcap Q\ a$›
apply (*cases* ‹$A = \{\}$›, *simp add*: *Ndet-id*)
apply (*rotate-tac*, *induct A rule*: *finite-set-induct-nonempty*)
by *simp-all* (*metis* (*no-types*, *lifting*) *Ndet-commute Ndet-assoc*)

5.9 Commutativity

```
lemma MultiNdet-sets-commute:
  ‹⟦finite A; finite B⟧ ⟹
    (⨅ a ∈ A. ⨅ b ∈ B. P a b) = ⨅ b ∈ B. ⨅ a ∈ A. P a b›
proof (cases ‹A = {}›)
  show ‹finite A ⟹ finite B ⟹ A = {} ⟹
        (⨅ a∈A.  MultiNdet B (P a)) = ⨅ b∈B.  ⨅ a∈A.  P a b›
    by (simp add: MultiNdet-STOP-id)
next
  assume ‹A ≠ {}› and ‹finite A› and ‹finite B›
  thus ‹(⨅ a∈A.  MultiNdet B (P a)) = ⨅ b∈B.  ⨅ a∈A.  P a b›
    apply (induct A rule: finite-set-induct-nonempty)
    by (simp-all add: MultiNdet-Ndet)
qed
```

5.10 Behaviour with Injectivity

```
lemma inj-on-mapping-over-MultiNdet:
  ‹⟦finite A; inj-on f A⟧ ⟹ (⨅ x ∈ A. P x) = ⨅ x ∈ f ‘ A. P (inv-into A f x)›
proof (induct A rule: induct-subset-empty-single)
  show ‹MultiNdet {} P = ⨅ x∈f ‘ {}.  P (inv-into {} f x)› by force
next
  case 2
  show ?case by force
next
  case (3 F a)
  hence f1: ‹inv-into (insert a F) f (f a) = a› by force
  show ?case
    apply (simp add: 3.hyps(2) 3.hyps(4) f1)
    apply (rule arg-cong[where f = ‹λx. P a ⊓ x›])
    apply (subst 3.hyps(5), rule inj-on-subset[OF 3.prems subset-insertI])
    apply (rule mono-MultiNdet-eq, simp add: 3.hyps(2))
    using 3.prems by fastforce
qed
```

5.11 The Projections

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

lemma F-MultiNdet:
  ‹finite A ⟹ ℱ (⨅ x ∈ A. P x) =
              (if A = {} then {(s, X). s = []} else ⋃ p ∈ A. ℱ (P p))›
  apply (simp add: F-STOP, intro impI, rotate-tac)
  by (induct rule: finite-set-induct-nonempty) (simp-all add: F-Ndet)
```

lemma *T-MultiNdet*:
‹*finite A* ⟹ 𝒯 (⨅ *x* ∈ *A*. *P x*) =
(*if A* = {} *then* {[]} *else* ⋃ *p* ∈ *A*. 𝒯 (*P p*))›
apply (*simp add*: *T-STOP*, *intro impI*, *rotate-tac*)
by (*induct rule*: *finite-set-induct-nonempty*) (*simp-all add*: *T-Ndet*)

5.12 Cartesian Product Results

lemma *MultiNdet-cartprod-σs-set-σs-set*:

‹⟦*finite A*; *finite B*; ∀ *s* ∈ *A*. *length s* = len_1⟧ ⟹
(⨅ (*s*, *t*) ∈ *A* × *B*. *P* (*s* @ *t*)) = ⨅ *u* ∈ {*s* @ *t* |*s t*. (*s*, *t*) ∈ *A* × *B*}. *P u*›
apply (*subst inj-on-mapping-over-MultiNdet*[**where** *f* = ‹λ (*s*, *t*). *s* @ *t*›],
simp-all add: *inj-on-def*)
apply (*subst prem-Multi-cartprod*(*1*)[*simplified*, *symmetric*])
apply (*rule mono-MultiNdet-eq*, *simp add*: *finite-image-set2*)
by (*metis* (*no-types*, *lifting*) *case-prod-unfold f-inv-into-f*)

lemma *MultiNdet-cartprod-s-set-σs-set*:
‹⟦*finite A*; *finite B*⟧ ⟹
(⨅ (*s*, *t*) ∈ *A* × *B*. *P* (*s* # *t*)) = ⨅ *u* ∈ {*s* # *t* |*s t*. (*s*, *t*) ∈ *A* × *B*}. *P u*›
apply (*subst inj-on-mapping-over-MultiNdet*[**where** *f* = ‹λ (*s*, *t*). *s* # *t*›],
simp-all add: *inj-on-def*)
apply (*subst prem-Multi-cartprod*(*2*)[*simplified*, *symmetric*])
apply (*rule mono-MultiNdet-eq*, *simp add*: *finite-image-set2*)
by (*metis* (*no-types*, *lifting*) *case-prod-unfold f-inv-into-f*)

lemma *MultiNdet-cartprod-s-set-s-set*:
‹⟦*finite A*; *finite B*⟧ ⟹
(⨅ (*s*, *t*) ∈ *A* × *B*. *P* [*s*, *t*]) = ⨅ *u* ∈ {[*s*, *t*] |*s t*. (*s*, *t*) ∈ *A* × *B*}. *P u*›
apply (*subst inj-on-mapping-over-MultiNdet*[**where** *f* = ‹λ (*s*, *t*). [*s*, *t*]›],
simp-all add: *inj-on-def*)
apply (*subst prem-Multi-cartprod*(*3*)[*simplified*, *symmetric*])
apply (*rule mono-MultiNdet-eq*, *simp add*: *finite-image-set2*)
by (*metis* (*no-types*, *lifting*) *case-prod-unfold f-inv-into-f*)

lemma *MultiNdet-cartprod*:
‹⟦*finite A*; *finite B*⟧ ⟹ (⨅ (*s*, *t*) ∈ *A* × *B*. *P s t*) = ⨅ *s*∈*A*. ⨅ *t*∈*B*. *P s t*›
supply *arg-cong-Ndet* = *arg-cong*[**where** *f* = ‹λ*Q*. - ⊓ *Q*›]
proof (*induct* ‹*card A*› *arbitrary*: *A B rule*: *nat-less-induct*)
case (*1 A B*)
from ‹*finite A*› ‹*finite B*› **consider** ‹*A* = {}› | ‹*B* = {}› |
‹∃ *mA mB a b A′ B′*. *A* = *insert a A′* ∧ *B* = *insert b B′* ∧ *mA* = *card A′* ∧
mB = *card B′* ∧ *mA* < *card A* ∧ *mB* < *card B*›
by (*metis card-Diff1-less-iff ex-in-conv insert-Diff*)
thus ‹(⨅(*x*, *y*) ∈ *A* × *B*. *P x y*) = ⨅ *s*∈*A*. *MultiNdet B* (*P s*)›

```
  proof cases
    show ‹A = {} ⟹ (⨅ (x, y)∈A × B. P x y) = ⨅ s∈A. MultiNdet B (P s)›
      by simp
  next
    show ‹B = {} ⟹ (⨅ (x, y)∈A × B. P x y) = ⨅ s∈A. MultiNdet B (P s)›
      by (simp add: MultiNdet-STOP-id[OF 1.prems(1)])
  next
    assume ‹∃ mA mB a b A' B'. A = insert a A' ∧ B = insert b B' ∧
            mA = card A' ∧ mB = card B' ∧ mA < card A ∧ mB < card B›
    then obtain mA mB a b A' B'
      where ∗ : ‹A = insert a A'› ‹B = insert b B'› ‹mA = card A'›
                ‹mB = card B'› ‹mA < card A› ‹mB < card B› by blast
    have ∗∗  : ‹Pair a ‘ B' = {a} × B'›
     and ∗∗∗ : ‹(λa. (a, b)) ‘ A' = A' × {b}› unfolding image-def by blast+
    show ‹(⨅(x, y) ∈ A × B.  P x y) = ⨅ s∈A.  MultiNdet B (P s)›
      using ∗(1, 2) ‹finite A› ‹finite B›
      apply (cases ‹A' = {}›; cases ‹B' = {}›; simp-all)
        apply (rule arg-cong-Ndet)
        apply (subst inj-on-mapping-over-MultiNdet[of B' ‹λb. (a, b)›],
               simp-all add: inj-on-def ∗∗)
        apply (rule mono-MultiNdet-eq, simp-all)
        apply (metis Pair-inject f-inv-into-f image-eqI)
       apply (rule arg-cong-Ndet)
       apply (subst inj-on-mapping-over-MultiNdet[of A' ‹λa. (a, b)›],
              simp-all add: inj-on-def ∗∗∗)
       apply (rule mono-MultiNdet-eq, simp-all)
       apply (metis (no-types, lifting) f-inv-into-f image-eqI prod.inject)

      apply (subst MultiNdet-factorization-union[symmetric], simp-all)
      apply (subst 1(1)[rule-format, OF ∗(5, 3)], simp-all)
      apply (simp add: MultiNdet-Ndet[symmetric])
      apply (subst Ndet-assoc, rule arg-cong-Ndet)
      apply (subst (3) Ndet-commute, rule arg-cong-Ndet)
      apply (subst inj-on-mapping-over-MultiNdet[of B' ‹λb. (a, b)›],
             simp-all add: inj-on-def ∗∗)
      apply (rule mono-MultiNdet-eq)
       apply (simp; fail)
      by (metis ∗∗ case-prod-conv f-inv-into-f)
  qed
qed


end
```

Chapter 6

The MultiSync Operator

theory *MultiSync*
  imports *HOL−Library.Multiset PreliminaryWork Patch*
begin

6.1 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 *′α multiset* and not on *′α set*.

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

fun *MultiSync-list* :: ‹[′b set, ′a list, ′a ⇒ ′b process] ⇒ ′b process›
  where ‹*MultiSync-list S* [] *P* = *STOP*›
  | ‹*MultiSync-list S* (*l* # *L*) *P* = *fold* (λ*x r*. *r* ⟦*S*⟧ *P x*) *L* (*P l*)›

syntax *-MultiSync-list* :: ‹[*pttrn*, ′*b set*, ′*a list*, ′*b process*] ⇒ ′*b process*›
  (‹(*3*⟦-⟧$_l$-∈-./ -)› *63*)
translations ⟦*S*⟧$_l$ *p* ∈ *L*. *P* ⇌ *CONST MultiSync-list S L* (λ*p*. *P*)

interpretation *MultiSync*: *comp-fun-commute* **where** *f* = ‹λ*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*′ ⟹ *MultiSync-list S L P* = *MultiSync-list S L*′ *P*›
  apply (*cases L*; *simp*)
proof −
  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 (λx 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 (λx 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 (λx 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 :: ‹['b set, 'a multiset, 'a ⇒ 'b process] ⇒ 'b process›
where ‹MultiSync S M P = MultiSync-list S (SOME L. mset L = M) P›

syntax -MultiSync :: ‹[pttrn,'b set,'a multiset,'b process] ⇒ 'b process›
(‹(3⟦-⟧ -∈#-. / -)› 63)
translations ⟦S⟧ p ∈# M. P ⇌ CONST MultiSync S M (λp. P)

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

abbreviation MultiInter :: ‹['a multiset, 'a ⇒ 'b process] ⇒ 'b process›
where ‹MultiInter M P ≡ MultiSync {} M P›

syntax -MultiInter :: ‹[pttrn, 'a multiset, 'b process] ⇒ 'b process›
(‹(3||| -∈#-. / -)› 77)
translations ||| p ∈# M. P ⇌ CONST MultiInter M (λp. P)

Special case of MultiSync E P when E = UNIV.

abbreviation MultiPar :: ‹['a multiset, 'a ⇒ 'b process] ⇒ 'b process›
where ‹MultiPar M P ≡ MultiSync UNIV M P›

syntax -MultiPar :: ‹[pttrn, 'a multiset, 'b process] ⇒ 'b process›

(‹(*3*|| -∈#-. / -)› *77*)
translations || *p* ∈# *M*. *P* ⇌ *CONST MultiPar M* (*λp*. *P*)

6.2 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 −
  fix *L L′*
  assume ‹*M* ≠ {#}› ‹*mset L* = *M*› ‹*mset L′* = *add-mset m M*›
  thus ‹*MultiSync-list S L′ P* = *P m* ⟦*S*⟧ *MultiSync-list S L P*›
    apply (*subst MultiSync-list-mset*[**where** *L* = ‹*L′*› **and** *L′* = ‹*L* @ [*m*]›], *simp*)
    by (*cases L*; *simp add*: *Sync-commute*)
qed

lemma *mono-MultiSync-eq*:
  ‹∀ *x* ∈# *M*. *P x* = *Q x* ⟹ *MultiSync S M P* = *MultiSync S M Q*›
  by (*induct M rule*: *induct-subset-mset-empty-single*; *simp*)

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*›]

6.3 Some Tests

lemma ‹(⟦*S*⟧$_l$ *p* ∈ []. *P p*) = *STOP*›
  and ‹(⟦*S*⟧$_l$ *p* ∈ [*a*]. *P p*) = *P a*›
  and ‹(⟦*S*⟧$_l$ *p* ∈ [*a*, *b*]. *P p*) = *P a* ⟦*S*⟧ *P b*›
  and ‹(⟦*S*⟧$_l$ *p* ∈ [*a*, *b*, *c*]. *P 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*›]

6.4 Continuity

lemma *MultiSync-cont*[*simp*]:
‹∀ *x* ∈# *M*. *cont* (*P x*) ⟹ *cont* (λ*y*. ⟦*S*⟧ *z* ∈# *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*›]

6.5 Factorization of *Sync* in front of *MultiSync*

lemma *MultiSync-factorization-union*:
‹⟦*M* ≠ {#}; *N* ≠ {#}⟧ ⟹
(⟦*S*⟧ *z* ∈# *M*. *P z*) ⟦*S*⟧ (⟦*S*⟧ *z* ∈# *N*. *P z*) = ⟦*S*⟧ *z*∈# *M* + *N*. *P z*›
by (*erule mset-induct-nonempty*, *simp-all add*: *Sync-assoc*)

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

6.6 ⊥ Absorbance

lemma *MultiSync-BOT-absorb*:
‹*m* ∈# *M* ⟹ *P m* = ⊥ ⟹ (⟦*S*⟧ *z* ∈# *M*. *P z*) = ⊥›
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* ∈# *M*. *P m*) = ⊥ ⟷ (∃ *m* ∈# *M*. *P m* = ⊥)›
apply (*cases* ‹*M* = {#}›, *simp add*: *BOT-iff-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*›]

6.7 Other Properties

lemma *MultiSync-SKIP-id*: ‹$M \neq \{\#\} \Longrightarrow (⟦S⟧\ z \in\# M.\ SKIP) = SKIP$›
  by (*rule mset-induct-nonempty*, *simp-all add*: *Sync-SKIP-SKIP*)

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*: ‹$M = add\text{-}mset\ m\ (add\text{-}mset\ m'\ M')$›
    by (*metis diff-union-swap insert-DiffM assms*)
  show ‹$(||\ x \in\# M.\ (fst\ x \to P\ (snd\ x))) = STOP$›
    apply (*cases* ‹$M' = \{\#\}$›,
        simp-all add: *f2 prefix-Par1*[*rotated*, *rotated*, *OF assms*(*3*)])
    apply (*induct* M' *rule*: *mset-induct-nonempty*, *simp*)
     apply (*metis* (*no-types*, *opaque-lifting*) *Sync-BOT Par-STOP prefix-Par2*
                   prefix-Par1 assms(*3*))
   by (*metis* (*no-types*, *lifting*) *MultiPar-add add-mset-commute empty-not-add-mset*
                Par-BOT Par-STOP prefix-Par-SKIP Par-commute)
qed

lemma *MultiPar-prefix-two-distincts-STOP'*:
  ‹$⟦(m, n) \in\# M;\ (m', n') \in\# M;\ m \neq m'⟧ \Longrightarrow$
   $(||\ (m, n) \in\# M.\ (m \to P\ n)) = STOP$›
  apply (*subst cond-case-prod-eta*[**where** g = ‹$\lambda\ x.\ (fst\ x \to P\ (snd\ x))$›])
  by (*simp-all add*: *MultiPar-prefix-two-distincts-STOP*)

6.8 Behaviour of *MultiSync* with *Sync*

lemma *Sync-STOP-STOP*: ‹$STOP\ ⟦S⟧\ STOP = STOP$›
  by (*fact Mprefix-Sync-distr-subset*[*of* ‹{}› S ‹{}›, *simplified*,
            simplified Mprefix-STOP])

lemma *MultiSync-Sync*:
  ‹$(⟦S⟧\ z \in\# M.\ P\ z)\ ⟦S⟧\ (⟦S⟧\ z \in\# M.\ P'\ z) = ⟦S⟧\ z \in\# M.\ P\ z\ ⟦S⟧\ P'\ z$›
  apply (*cases* ‹$M = \{\#\}$›, *simp add*: *Sync-STOP-STOP*)
  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*›]

6.9 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 Sync-STOP-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›]

6.10 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*)
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 *auto*

lemmas *inj-on-mapping-over-MultiInter* =
inj-on-mapping-over-MultiSync[**where** S = ‹{}›]
and *inj-on-mapping-over-MultiPar* =
inj-on-mapping-over-MultiSync[**where** S = ‹UNIV›]

end

Chapter 7

The MultiSeq Operator

theory *MultiSeq*
  imports *Patch*
begin

7.1 Definition

definition *MultiSeq* :: ‹[′*a list*, ′*a* ⇒ ′*b process*] ⇒ ′*b process*›
  where ‹*MultiSeq S P* = *foldr* (λ*a r*. (*P a*) **;** *r*) *S SKIP*›

syntax -*MultiSeq* :: ‹[*pttrn*,′*a list*, ′*b process*] ⇒ ′*b process*›
  (‹(3*SEQ* -∈@-./ -)› 73)
translations *SEQ i* ∈@ *A*. *P* ⇌ *CONST MultiSeq A* (λ*i*. *P*)

7.2 First Properties

lemma *MultiSeq-rec0*[*simp*]: ‹(*SEQ p* ∈@ []. *P p*) = *SKIP*›
  by (*simp add*: *MultiSeq-def*)

lemma *MultiSeq-rec1*[*simp*]: ‹(*SEQ p* ∈@ [*a*]. *P p*) = *P a*›
  by (*simp add*: *MultiSeq-def Seq-SKIP*)

lemma *MultiSeq-Cons*[*simp*]: ‹(*SEQ i* ∈@ *a* # *L*. *P i*) = *P a* **;** (*SEQ i* ∈@ *L*. *P* *i*)›
  by (*simp add*: *MultiSeq-def*)

7.3 Some Tests

lemma ‹(*SEQ p* ∈@ []. *P p*) = *SKIP*›
  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*: *Seq-SKIP Seq-assoc*)

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

7.4 Continuity

lemma *MultiSeq-cont*[*simp*]:
‹∀ *x* ∈ *set L*. *cont* (*P x*) ⟹ *cont* (λ*y*. *SEQ z* ∈@ *L*. *P z y*)›
by (*induct L*) *force+*

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

lemma *MultiSeq-factorization-append*:
‹(*SEQ p* ∈@ *A*. *P p*) **;** (*SEQ p* ∈@ *B*. *P p*) = (*SEQ p* ∈@ *A* @ *B* . *P p*)›
by (*induct A rule*: *list.induct*, *simp-all add*: *SKIP-Seq*, *metis Seq-assoc*)

7.6 ⊥ Absorbance

lemma *MultiSeq-BOT-absorb*:
‹*P a* = ⊥ ⟹ (*SEQ z* ∈@ *l1* @ [*a*] @ *l2*. *P z*) = (*SEQ z* ∈@ *l1*. *P z*) **;** ⊥›
by (*metis BOT-Seq MultiSeq-Cons MultiSeq-factorization-append*)

7.7 First Properties

lemma *MultiSeq-SKIP-neutral*:
‹*P a* = *SKIP* ⟹ (*SEQ z* ∈@ *l1* @ [*a*] @ *l2*. *P z*) = *SEQ z* ∈@ *l1* @ *l2*. *P z*›
by (*simp add*: *MultiSeq-def SKIP-Seq*)

lemma *MultiSeq-STOP-absorb*:
‹*P a* = *STOP* ⟹
(*SEQ z* ∈@ *l1* @ [*a*] @ *l2*. *P z*) = (*SEQ z* ∈@ *l1*. *P z*) **;** *STOP*›
by (*metis STOP-Seq MultiSeq-Cons MultiSeq-factorization-append*)

lemma *mono-MultiSeq-eq*:
‹∀ *x* ∈ *set L*. *P x* = *Q x* ⟹ *MultiSeq L P* = *MultiSeq L Q*›
by (*induct L*) *fastforce+*

lemma *MultiSeq-is-SKIP-iff*:
‹*MultiSeq L P* = *SKIP* ⟷ (∀ *a* ∈ *set L*. *P a* = *SKIP*)›
by (*induct L*, *simp-all add*: *Seq-is-SKIP-iff*)

7.8 Commutativity

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

7.9 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*)
  case *Nil*
  show *?case* **by** *simp*
next
  case (*Cons a C*)
  hence *f1*: ‹*inv-into* (*insert a* (*set C*)) *f* (*f a*) = *a*› **by** *force*
  show *?case*
    apply (*simp add*: *f1*, *rule arg-cong*[**where** *f* = ‹λ*x*. *P a* **;** *x*›])
    apply (*subst Cons.hyps*(*1*), *rule inj-on-subset*[*OF Cons.prems*],
          simp add: *subset-insertI*)
    apply (*rule mono-MultiSeq-eq*)
    using *Cons.prems* **by** *fastforce*
qed

7.10 Definition of *first-elem*

primrec *first-elem* :: ‹[′α ⇒ *bool*, ′α *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*)

end

Chapter 8

The Global Non-Deterministic Choice

theory *GlobalNdet*
  imports *MultiNdet*
begin

8.1 General Non-Deterministic Choice Definition

This is an experimental definition of a generalized non-deterministic choice $a \sqcap b$ for an arbitrary set. The present version is "totalised" for the case of $A = \{\}$ by *STOP*, which is not the neutral element of the $(\sqcap)$ operator (because there is no neutral element for $(\sqcap)$).

```
lemma ‹∄ P. ∀ Q. (P :: 'α process) ⊓ Q = Q›
proof −
  { fix P :: ‹'α process›
    assume ∗ : ‹∀ Q. P ⊓ Q = Q›
    hence ‹P = STOP›
      by (erule-tac x = STOP in allE) (simp add: Ndet-is-STOP-iff)
    with ∗ have False
      by (erule-tac x = SKIP in allE)
        (metis mono-Ndet-FD-right Ndet-commute
              SKIP-FD-iff SKIP-Neq-STOP idem-FD)
  }
  thus ?thesis by blast
qed
```

```
lift-definition GlobalNdet :: ‹['α set, 'α ⇒ 'β process] ⇒ 'β process›
  is  ‹λA P.   if A = {}
              then ({(s, X). s = []}, {})
              else (⋃ a∈A. ℱ (P a), ⋃ a∈A. 𝒟 (P a))›
```

```
proof -
  show ‹?thesis A P›
   (is ‹is-process (  if A = {} then ({(s, X). s = []}, {}) else (?f, ?d))›) for A P
  proof (split if-split, intro conjI impI)
    show ‹is-process ({(s, X). s = []}, {})›
      by (metis STOP.rsp eq-onp-def)
  next
    show ‹is-process (⋃ a∈A. ℱ (P a), ⋃ a∈A. 𝒟 (P a))› if nonempty: ‹A ≠ {}›
     unfolding is-process-def FAILURES-def DIVERGENCES-def fst-conv snd-conv
    proof (intro conjI allI impI)
      show ‹([], {}) ∈ ?f› using is-processT1 nonempty by blast
    next
      show ‹(s, X) ∈ ?f ⟹ front-tickFree s› for s X
        using is-processT2 by blast
    next
      show ‹(s @ t, {}) ∈ ?f ⟹ (s, {}) ∈ ?f› for s t
        using is-processT3 by blast
    next
      show ‹(s, Y) ∈ ?f ∧ X ⊆ Y ⟹ (s, X) ∈ ?f› for s X Y
        using is-processT4 by blast
    next
      show ‹(s, X) ∈ ?f ∧ (∀ c. c ∈ Y ⟶ (s @ [c], {}) ∉ ?f) ⟹ (s, X ∪ Y) ∈
?f› for s X Y
        using is-processT5 by simp blast
    next
      show ‹(s @ [tick], {}) ∈ ?f ⟹ (s, X − {tick}) ∈ ?f› for s X
        using is-processT6 by blast
    next
      show ‹s ∈ ?d ∧ tickFree s ∧ front-tickFree t ⟹ s @ t ∈ ?d› for s t
        using is-processT7 by blast
    next
      show ‹s ∈ ?d ⟹ (s, X) ∈ ?f› for s X
        using is-processT8 by blast
    next
      show ‹s @ [tick] ∈ ?d ⟹ s ∈ ?d› for s
        using is-processT9 by blast
    qed
  qed
qed
```

```
syntax -GlobalNdet :: ‹[pttrn,'a set,'b process] ⇒ 'b process› (‹(3⊓ -∈-. / -)› 76)
translations ⊓ p ∈ A. P ⇌ CONST GlobalNdet A (λp. P)
```

Note that the global non-deterministic choice $\sqcap\ p{\in}A.\ \ P\ p$ is different from the multi-non-deterministic prefix $\sqcap p{\in}A \rightarrow\ \ P\ p$ which guarantees continuity even when A is *infinite* due to the fact that it communicates its choice via an internal prefix operator.

It is also subtly different from the multi-non-deterministic choice $\prod\ p{\in}A.\ P\ p$ which is only defined when A is *finite*.

lemma *empty-GlobalNdet*[*simp*] : ‹*GlobalNdet* {} *P* = *STOP*›
by (*simp add*: *GlobalNdet.abs-eq STOP-def*)

8.2 The Projections

lemma *F-GlobalNdet*:
‹$\mathcal{F}$ (⊓ *x* ∈ *A*. *P x*) = (*if A* = {} *then* {(*s*, *X*). *s* = []} *else* (⋃ *x*∈*A*. $\mathcal{F}$ (*P x*)))›
by (*simp add*: *Failures-def FAILURES-def GlobalNdet.rep-eq*)

lemma *D-GlobalNdet*:
‹$\mathcal{D}$ (⊓ *x* ∈ *A*. *P x*) = (*if A* = {} *then* {} *else* (⋃ *x*∈*A*. $\mathcal{D}$ (*P x*)))›
by (*simp add*: *Divergences-def DIVERGENCES-def GlobalNdet.rep-eq*)

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

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

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

8.3 Factorization of (⊓) in front of *GlobalNdet*

lemma *GlobalNdet-factorization-union*:
‹⟦*A* ≠ {}; *B* ≠ {}⟧ ⟹
(⊓ *p* ∈ *A*. *P p*) ⊓ (⊓ *p* ∈ *B*. *P p*) = (⊓ *p* ∈ *A* ∪ *B* . *P p*)›
by (*subst Process-eq-spec*) (*simp add*: *F-GlobalNdet D-GlobalNdet F-Ndet D-Ndet*)

8.4 ⊥ Absorbance

lemma *GlobalNdet-BOT-absorb*: ‹*P a* = ⊥ ⟹ *a* ∈ *A* ⟹ (⊓ *x* ∈ *A*. *P x*) = ⊥›
using *is-processT2*
by (*subst Process-eq-spec*)
(*auto simp add*: *F-GlobalNdet D-GlobalNdet F-UU D-UU D-imp-front-tickFree*)

lemma *GlobalNdet-is-BOT-iff*: ‹(⊓ *x* ∈ *A*. *P x*) = ⊥ ⟷ (∃ *a* ∈ *A*. *P a* = ⊥)›
by (*simp add*: *BOT-iff-D D-GlobalNdet*)

8.5 First Properties

lemma *GlobalNdet-id*: $\langle A \neq \{\} \Longrightarrow (\sqcap\ p \in A.\ P) = P\rangle$
by (*subst Process-eq-spec*) (*simp add*: *F-GlobalNdet D-GlobalNdet*)

lemma *GlobalNdet-STOP-id*: $\langle(\sqcap\ p \in A.\ STOP) = STOP\rangle$
by (*cases* $\langle A = \{\}\rangle$) (*simp-all add*: *GlobalNdet-id*)

lemma *GlobalNdet-unit*[*simp*] : $\langle(\sqcap\ x \in \{a\}.\ P\ x) = P\ a\rangle$
by(*auto simp* : *Process-eq-spec F-GlobalNdet D-GlobalNdet*)

lemma *GlobalNdet-distrib-unit*:
$\langle A - \{a\} \neq \{\} \Longrightarrow (\sqcap\ x \in insert\ a\ A.\ P\ x) = P\ a \sqcap (\sqcap\ x \in A - \{a\}.\ P\ x)\rangle$
by (*metis GlobalNdet-factorization-union GlobalNdet-unit*
empty-not-insert insert-Diff-single insert-is-Un)

8.6 Behaviour of *GlobalNdet* with ($\sqcap$)

lemma *GlobalNdet-Ndet*:
$\langle(\sqcap\ a \in A.\ P\ a) \sqcap (\sqcap\ a \in A.\ Q\ a) = \sqcap\ a \in A.\ P\ a \sqcap Q\ a\rangle$
by (*auto simp add*: *Process-eq-spec F-GlobalNdet D-GlobalNdet F-Ndet D-Ndet*)

8.7 Commutativity

lemma *GlobalNdet-sets-commute*:
$\langle(\sqcap\ a \in A.\ \sqcap\ b \in B.\ P\ a\ b) = \sqcap\ b \in B.\ \sqcap\ a \in A.\ P\ a\ b\rangle$
by (*auto simp add*: *Process-eq-spec F-GlobalNdet D-GlobalNdet*
F-Ndet D-Ndet F-STOP D-STOP)

8.8 Behaviour with Injectivity

lemma *inj-on-mapping-over-GlobalNdet*:
$\langle inj\text{-}on\ f\ A \Longrightarrow (\sqcap\ x \in A.\ P\ x) = \sqcap\ x \in f\ `\ A.\ P\ (inv\text{-}into\ A\ f\ x)\rangle$
by (*simp add*: *Process-eq-spec F-GlobalNdet D-GlobalNdet*
F-Ndet D-Ndet F-STOP D-STOP)

8.9 Cartesian Product Results

lemma *GlobalNdet-cartprod-σs-set-σs-set*:
$\langle(\sqcap\ (s,\ t) \in A \times B.\ P\ (s\ @\ t)) = \sqcap\ u \in \{s\ @\ t\ |s\ t.\ (s,\ t) \in A \times B\}.\ P\ u\rangle$
apply (*subst Process-eq-spec*, *simp add*: *F-GlobalNdet D-GlobalNdet*)
by *safe auto*

lemma *GlobalNdet-cartprod-s-set-σs-set*:
‹($\sqcap$ (s, t) ∈ A × B. P (s # t)) = $\sqcap$ u ∈ {s # t |s t. (s, t) ∈ A × B}. P u›
apply (*subst Process-eq-spec*, *simp add*: *F-GlobalNdet D-GlobalNdet*)
by *safe auto*

lemma *GlobalNdet-cartprod-s-set-s-set*:
‹($\sqcap$ (s, t) ∈ A × B. P [s, t]) = $\sqcap$ u ∈ {[s, t] |s t. (s, t) ∈ A × B}. P u›
apply (*subst Process-eq-spec*, *simp add*: *F-GlobalNdet D-GlobalNdet*)
by *safe auto*

lemma *GlobalNdet-cartprod*: ‹($\sqcap$(s, t) ∈ A × B. P s t) = $\sqcap$s ∈ A. $\sqcap$t ∈ B. P s t›
apply (*subst Process-eq-spec*, *simp add*: *F-GlobalNdet D-GlobalNdet*)
by *safe auto*

8.10 Link with *MultiNdet*

This operator is in fact an extension of *MultiNdet* to arbitrary sets: when *A* is *finite*, we have $\sqcap$ a∈A. P a = $\bigsqcap$ a∈A. P a.

lemma *finite-GlobalNdet-is-MultiNdet*:
‹*finite A* ⟹ ($\sqcap$ p ∈ A. P p) = $\bigsqcap$ p ∈ A. P p›
by (*simp add*: *Process-eq-spec F-GlobalNdet F-MultiNdet D-GlobalNdet D-MultiNdet*)

We obtain immediately the continuity when *A* is *finite* (and this is a necessary hypothesis for continuity).

lemma *GlobalNdet-cont*[*simp*]:
‹⟦*finite A*; ∀ x. *cont* (f x)⟧ ⟹ *cont* (λy. ($\sqcap$ z ∈ A. (f z y)))›
by (*simp add*: *finite-GlobalNdet-is-MultiNdet*)

8.11 Link with *Mndetprefix*

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

lemma *Mndetprefix-GlobalNdet*: ‹$\sqcap$ x ∈ A → P x = $\sqcap$ x ∈ A. (x → P x)›
apply (*cases* ‹A = {}›, *simp*)
by (*subst Process-eq-spec-optimized*)
(*simp-all add*: *F-Mndetprefix D-Mndetprefix F-GlobalNdet D-GlobalNdet*)

lemma *write0-GlobalNdet*:
‹A ≠ {} ⟹ ($\sqcap$ x∈A. (a → P x)) = (a → ($\sqcap$ x∈A. P x))›
by (*auto simp add*: *Process-eq-spec write0-def*
F-GlobalNdet D-GlobalNdet F-Mprefix D-Mprefix)

8.12 Properties

lemma *GlobalNdet-Det*:
‹$A \neq \{\} \Longrightarrow (\sqcap\ a \in A.\ P\ a)\ \Box\ Q = \sqcap\ a \in A.\ P\ a\ \Box\ Q$›
by (*auto simp add*: *Process-eq-spec F-GlobalNdet D-GlobalNdet*
F-Det D-Det Un-def T-GlobalNdet)

lemma *Mndetprefix-STOP*: ‹$A \subseteq C \Longrightarrow (\sqcap\ a \in A \to P\ a)\ [\![C]\!]\ STOP = STOP$›
proof (*subst STOP-iff-T*, *intro subset-antisym subsetI*)
show ‹$s \in \{[]\} \Longrightarrow s \in \mathcal{T}\ (Mndetprefix\ A\ P\ [\![C]\!]\ STOP)$› **for** *s*
by (*simp add*: *Nil-elem-T*)
next
show ‹$A \subseteq C \Longrightarrow s \in \mathcal{T}\ (Mndetprefix\ A\ P\ [\![C]\!]\ STOP) \Longrightarrow s \in \{[]\}$› **for** *s*
by (*auto simp add*: *STOP-iff-T T-Sync T-Mndetprefix D-Mndetprefix*
T-STOP D-STOP write0-def T-Mprefix D-Mprefix subset-iff
split: *if-split-asm*)
(*metis Sync.sym emptyLeftNonSync hd-in-set imageI insert-iff*)+
qed

lemma *GlobalNdet-Sync-distr*:
‹$A \neq \{\} \Longrightarrow (\sqcap\ x \in A.\ P\ x)\ [\![\ C\ ]\!]\ Q = \sqcap\ x \in A.\ (P\ x\ [\![\ C\ ]\!]\ Q)$›
apply (*auto simp*: *Process-eq-spec T-GlobalNdet*
F-GlobalNdet D-GlobalNdet D-Sync F-Sync) — takes some seconds
using *front-tickFree-Nil* **by** *blast*+

lemma *Mndetprefix-Mprefix-Sync-distr*:
‹$[\![A \subseteq B;\ B \subseteq C]\!] \Longrightarrow (\sqcap\ a \in A \to P\ a)\ [\![\ C\ ]\!]\ (\Box\ b \in B \to Q\ b) =$
$\sqcap\ a \in A \to (P\ a\ [\![\ C\ ]\!]\ Q\ a)$›
— does not hold in general when $A \subseteq C$
apply (*cases* ‹$A = \{\}$›, *simp*,
metis (*no-types*, *lifting*) *Mprefix-STOP Mprefix-Sync-distr-subset*
empty-subsetI inf-bot-left)
apply (*cases* ‹$B = \{\}$›, *simp add*: *Mprefix-STOP Mndetprefix-STOP*)
apply (*subst Mndetprefix-GlobalNdet*, *subst GlobalNdet-Sync-distr*, *assumption*)
apply (*subst Mndetprefix-GlobalNdet*, *subst Mprefix-singl*[*symmetric*])
apply (*unfold write0-def*, *rule mono-GlobalNdet-eq*[*rule-format*])
apply (*subst Mprefix-Sync-distr-subset*[*of - C B P Q*], *blast*, *blast*)
by (*metis* (*no-types*, *lifting*) *in-mono inf-le1 insert-disjoint*(*1*)
Mprefix-singl subset-singletonD)

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

lemma *Mndetprefix-Sync-Det-distr*:
‹$(\sqcap\ a \in A \to (P\ a\ [\![\ C\ ]\!]\ (\sqcap\ b \in B \to Q\ b)))\ \Box$

(⊓ b ∈ B → ((⊓ a ∈ A → P a) ⟦ C ⟧ Q b))
⊑_{FD} (⊓ a ∈ A → P a) ⟦ C ⟧ (⊓ b ∈ B → Q b)›
if *set-hyps* : ‹A ≠ {}› ‹B ≠ {}› ‹A ∩ C = {}› ‹B ∩ C = {}›
— both surprising: equality does not hold + deterministic choice
proof −
have *mono-GlobalNdet-FD*:
‹⋀P Q A. ∀ x ∈ A. P x ⊑$_{FD}$ Q x ⟹ GlobalNdet A P ⊑$_{FD}$ GlobalNdet A Q›
by (*auto simp*: *failure-divergence-refine-def le-ref-def F-GlobalNdet D-GlobalNdet*)

have ∗ : ‹a ∈ A ⟹ b ∈ B ⟹
(⊓b ∈ B. (a → (P a ⟦C⟧ (b → Q b)))) □
(⊓a ∈ A. (b → ((a → P a) ⟦C⟧ Q b))) ⊑$_{FD}$
(a → P a) ⟦C⟧ (b → Q b)› **for** a b
apply (*subst Mprefix-Sync-distr-indep*[*of* ‹{a}› C ‹{b}›, *unfolded Mprefix-singl*])
using *that*(*3*)
apply (*simp add*: *disjoint-iff*; *fail*)
using *that*(*4*)
apply (*simp add*: *disjoint-iff*; *fail*)
apply (*rule mono-Det-FD*)
unfolding *failure-divergence-refine-def le-ref-def*
by (*auto simp add*: *D-GlobalNdet F-GlobalNdet*)

have ‹(⊓a ∈ A. ⊓b ∈ B. (a → (P a ⟦C⟧ (b → Q b)))) □
(⊓b ∈ B. ⊓a ∈ A. (b → ((a → P a) ⟦C⟧ Q b))) ⊑$_{FD}$
⊓b ∈ B. ⊓a ∈ A. ((a → P a) ⟦C⟧ (b → Q b))›
apply (*subst Det-commute*, *subst GlobalNdet-Det*, *simp add*: *set-hyps*(*2*))
apply (*subst Det-commute*, *subst GlobalNdet-Det*, *simp add*: *set-hyps*(*1*))
apply (*intro ballI impI mono-GlobalNdet-FD*)
using ∗ **by** *blast*

thus *?thesis*
apply (*simp add*: *Mndetprefix-GlobalNdet GlobalNdet-Sync-distr*)
apply (*subst* (*1 2 3*) *Sync-commute*, *simp add*: *GlobalNdet-Sync-distr set-hyps*(*2*))
apply (*subst* (*1 2 3*) *Sync-commute*, *simp add*: *GlobalNdet-Sync-distr set-hyps*(*1*))
by (*simp add*: *set-hyps*(*1*, *2*) *write0-GlobalNdet*)
qed

lemma *GlobalNdet-Mprefix-distr*:
‹A ≠ {} ⟹ (⊓ a ∈ A. □ b ∈ B → P a b) = □ b ∈ B → (⊓ a ∈ A. P a b)›
by (*auto simp add*: *Process-eq-spec F-GlobalNdet D-GlobalNdet F-Mprefix D-Mprefix*)

lemma *GlobalNdet-Det-distrib*:
‹(⊓ a ∈ A. P a □ Q a) = (⊓ a ∈ A. P a) □ (⊓ a ∈ A. Q a)›

if ‹∃ Q' b. ∀ a. Q a = (b → Q' a)›
proof −
from *that* **obtain** b Q' **where** ‹∀ a. (Q a = (b → Q' a))› **by** *blast*
thus *?thesis*
apply (*cases* ‹A = {}›, *simp add*: *Det-STOP*)
apply (*simp add*: *Process-eq-spec F-Det D-Det write0-def*
F-GlobalNdet D-GlobalNdet T-GlobalNdet, *safe*)
by (*auto simp add*: *F-Mprefix D-Mprefix T-Mprefix*)
qed

end

Chapter 9

CSPM

theory *CSPM*
imports *MultiDet MultiNdet MultiSync MultiSeq GlobalNdet HOL−CSP.Assertions*
begin

From the binary laws of `HOL-CSP`, we immediately obtain refinement results and lemmas about the combination of multi-operators.

9.1 Refinements Results

lemma *mono-MultiDet-F*:
‹*finite A* $\Longrightarrow \forall x \in A.\ P\ x \sqsubseteq_F Q\ x \Longrightarrow$ *MultiDet A P* $\sqsubseteq_F$ *MultiDet A Q*›
apply (*induct A rule*: *induct-subset-empty-single*; *simp*)
oops

lemma *mono-MultiDet-D*[*simp*, *elim*]:
‹*finite A* $\Longrightarrow \forall x \in A.\ P\ x \sqsubseteq_D Q\ x \Longrightarrow$ *MultiDet A P* $\sqsubseteq_D$ *MultiDet A Q*›
and *mono-MultiDet-T*[*simp*, *elim*]:
‹*finite A* $\Longrightarrow \forall x \in A.\ P\ x \sqsubseteq_T Q\ x \Longrightarrow$ *MultiDet A P* $\sqsubseteq_T$ *MultiDet A Q*›
and *mono-MultiDet-DT*[*simp*, *elim*]:
‹*finite A* $\Longrightarrow \forall x \in A.\ P\ x \sqsubseteq_{DT} Q\ x \Longrightarrow$ *MultiDet A P* $\sqsubseteq_{DT}$ *MultiDet A Q*›
and *mono-MultiDet-FD*[*simp*, *elim*]:
‹*finite A* $\Longrightarrow \forall x \in A.\ P\ x \sqsubseteq_{FD} Q\ x \Longrightarrow$ *MultiDet A P* $\sqsubseteq_{FD}$ *MultiDet A Q*›
by (*induct A rule*: *induct-subset-empty-single*; *simp del*: *MultiDet-insert*)+

lemma *mono-MultiNdet-F*[*simp*, *elim*]:
‹*finite A* $\Longrightarrow \forall x \in A.\ P\ x \sqsubseteq_F Q\ x \Longrightarrow$ *MultiNdet A P* $\sqsubseteq_F$ *MultiNdet A Q*›
and *mono-MultiNdet-D*[*simp*, *elim*]:
‹*finite A* $\Longrightarrow \forall x \in A.\ P\ x \sqsubseteq_D Q\ x \Longrightarrow$ *MultiNdet A P* $\sqsubseteq_D$ *MultiNdet A Q*›
and *mono-MultiNdet-T*[*simp*, *elim*]:
‹*finite A* $\Longrightarrow \forall x \in A.\ P\ x \sqsubseteq_T Q\ x \Longrightarrow$ *MultiNdet A P* $\sqsubseteq_T$ *MultiNdet A Q*›
and *mono-MultiNdet-DT*[*simp*, *elim*]:

‹*finite A* ⟹ ∀ *x* ∈ *A*. *P x* $\sqsubseteq_{DT}$ *Q x* ⟹ *MultiNdet A P* $\sqsubseteq_{DT}$ *MultiNdet A Q*›
and *mono-MultiNdet-FD*[*simp*, *elim*]:
‹*finite A* ⟹ ∀ *x* ∈ *A*. *P x* $\sqsubseteq_{FD}$ *Q x* ⟹ *MultiNdet A P* $\sqsubseteq_{FD}$ *MultiNdet A Q*›
by (*induct A rule*: *induct-subset-empty-single*; *simp*)+

lemma *mono-MultiNdet-F-single*:
‹*A* ≠ {} ⟹ *finite A* ⟹ ∀ *a* ∈ *A*. *P* $\sqsubseteq_F$ *Q a* ⟹ *P* $\sqsubseteq_F$ *MultiNdet A Q*›
and *mono-MultiNdet-D-single*:
‹*A* ≠ {} ⟹ *finite A* ⟹ ∀ *a* ∈ *A*. *P* $\sqsubseteq_D$ *Q a* ⟹ *P* $\sqsubseteq_D$ *MultiNdet A Q*›
and *mono-MultiNdet-T-single*:
‹*A* ≠ {} ⟹ *finite A* ⟹ ∀ *a* ∈ *A*. *P* $\sqsubseteq_T$ *Q a* ⟹ *P* $\sqsubseteq_T$ *MultiNdet A Q*›
and *mono-MultiNdet-DT-single*:
‹*A* ≠ {} ⟹ *finite A* ⟹ ∀ *a* ∈ *A*. *P* $\sqsubseteq_{DT}$ *Q a* ⟹ *P* $\sqsubseteq_{DT}$ *MultiNdet A Q*›
and *mono-MultiNdet-FD-single*:
‹*A* ≠ {} ⟹ *finite A* ⟹ ∀ *a* ∈ *A*. *P* $\sqsubseteq_{FD}$ *Q a* ⟹ *P* $\sqsubseteq_{FD}$ *MultiNdet A Q*›
by (*subst MultiNdet-id*[**where** *A* = *A*, *symmetric*], *simp-all*)+

lemma
assumes ‹*A* ≠ {}› **and** ‹*finite B*› **and** ‹*A* ⊆ *B*›
shows *mono-MultiNdet-F-left-absorb-subset*:
‹∀ *x* ∈ *A*. *P x* $\sqsubseteq_F$ *Q x* ⟹ *MultiNdet B P* $\sqsubseteq_F$ *MultiNdet A Q*›
and *mono-MultiNdet-D-left-absorb-subset*:
‹∀ *x* ∈ *A*. *P x* $\sqsubseteq_D$ *Q x* ⟹ *MultiNdet B P* $\sqsubseteq_D$ *MultiNdet A Q*›
and *mono-MultiNdet-T-left-absorb-subset*:
‹∀ *x* ∈ *A*. *P x* $\sqsubseteq_T$ *Q x* ⟹ *MultiNdet B P* $\sqsubseteq_T$ *MultiNdet A Q*›
and *mono-MultiNdet-FD-left-absorb-subset*:
‹∀ *x* ∈ *A*. *P x* $\sqsubseteq_{FD}$ *Q x* ⟹ *MultiNdet B P* $\sqsubseteq_{FD}$ *MultiNdet A Q*›
and *mono-MultiNdet-DT-left-absorb-subset*:
‹∀ *x* ∈ *A*. *P x* $\sqsubseteq_{DT}$ *Q x* ⟹ *MultiNdet B P* $\sqsubseteq_{DT}$ *MultiNdet A Q*›
supply *finiteA* = *finite-subset*[*OF* ‹*A* ⊆ *B*› ‹*finite B*›]
and *B-eq* = *Un-absorb1*[*OF* ‹*A* ⊆ *B*›, *symmetric*,
simplified Un-Diff-cancel[*of A B*, *symmetric*]]
and *results* = *Diff-cancel MultiNdet-factorization-union Un-Diff-cancel assms*(*1*, *2*)
apply (*metis mono-MultiNdet-F mono-Ndet-F-left results finiteA B-eq*)
apply (*metis mono-MultiNdet-D mono-Ndet-D-left results finiteA B-eq*)
apply (*metis mono-MultiNdet-T mono-Ndet-T-left results finiteA B-eq*)
apply (*metis mono-MultiNdet-FD mono-Ndet-FD-left results finiteA B-eq*)
by (*metis mono-MultiNdet-DT mono-Ndet-DT-left results finiteA B-eq*)

corollary *mono-MultiNdet-F-left-absorb*[*simp*]:
‹*finite A* ⟹ *x* ∈ *A* ⟹ *P x* $\sqsubseteq_F$ *Q* ⟹ *MultiNdet A P* $\sqsubseteq_F$ *Q*›
and *mono-MultiNdet-D-left-absorb*[*simp*]:
‹*finite A* ⟹ *x* ∈ *A* ⟹ *P x* $\sqsubseteq_D$ *Q* ⟹ *MultiNdet A P* $\sqsubseteq_D$ *Q*›

and *mono-MultiNdet-T-left-absorb*[*simp*]:
‹*finite A* ⟹ *x* ∈ *A* ⟹ *P x* $\sqsubseteq_T$ *Q* ⟹ *MultiNdet A P* $\sqsubseteq_T$ *Q*›
and *mono-MultiNdet-FD-left-absorb*[*simp*]:
‹*finite A* ⟹ *x* ∈ *A* ⟹ *P x* $\sqsubseteq_{FD}$ *Q* ⟹ *MultiNdet A P* $\sqsubseteq_{FD}$ *Q*›
and *mono-MultiNdet-DT-left-absorb*[*simp*]:
‹*finite A* ⟹ *x* ∈ *A* ⟹ *P x* $\sqsubseteq_{DT}$ *Q* ⟹ *MultiNdet A P* $\sqsubseteq_{DT}$ *Q*›
apply (*rule trans-F* [*OF mono-MultiNdet-F-left-absorb-subset*
[**where** *A* = ‹{*x*}›, *simplified*]]; *simp*)
apply (*rule trans-D* [*OF mono-MultiNdet-D-left-absorb-subset*
[**where** *A* = ‹{*x*}›, *simplified*]]; *simp*)
apply (*rule trans-T* [*OF mono-MultiNdet-T-left-absorb-subset*
[**where** *A* = ‹{*x*}›, *simplified*]]; *simp*)
apply (*rule trans-FD*[*OF mono-MultiNdet-FD-left-absorb-subset*
[**where** *A* = ‹{*x*}›, *simplified*]]; *simp*)
by (*rule trans-DT*[*OF mono-MultiNdet-DT-left-absorb-subset*
[**where** *A* = ‹{*x*}›, *simplified*]]; *simp*)

lemma *mono-MultiNdet-MultiDet-F*[*simp*, *elim*]:
‹*finite A* ⟹ *MultiNdet A P* $\sqsubseteq_F$ *MultiDet A P*›
and *mono-MultiNdet-MultiDet-D*[*simp*, *elim*]:
‹*finite A* ⟹ *MultiNdet A P* $\sqsubseteq_D$ *MultiDet A P*›
and *mono-MultiNdet-MultiDet-T*[*simp*, *elim*]:
‹*finite A* ⟹ *MultiNdet A P* $\sqsubseteq_T$ *MultiDet A P*›
and *mono-MultiNdet-MultiDet-FD*[*simp*, *elim*]:
‹*finite A* ⟹ *MultiNdet A P* $\sqsubseteq_{FD}$ *MultiDet A P*›
and *mono-MultiNdet-MultiDet-DT*[*simp*, *elim*]:
‹*finite A* ⟹ *MultiNdet A P* $\sqsubseteq_{DT}$ *MultiDet A P*›
by (*induct A rule*: *induct-subset-empty-single*;
simp del: *MultiDet-insert*;
meson idem-F mono-Ndet-F mono-Ndet-Det-F trans-F
idem-D mono-Ndet-D mono-Ndet-Det-D trans-D
idem-T mono-Ndet-T mono-Ndet-Det-T trans-T
idem-FD mono-Ndet-FD mono-Ndet-Det-FD trans-FD
idem-DT mono-Ndet-DT mono-Ndet-Det-DT trans-DT)+

lemma *mono-MultiSync-F*: ‹∀ *x* ∈# *M*. *P x* $\sqsubseteq_F$ *Q x* ⟹ *MultiSync S M P* $\sqsubseteq_F$ *MultiSync S M Q*›
apply (*induct M rule*: *induct-subset-mset-empty-single*; *simp*)
oops

lemma *mono-MultiSync-D*: ‹∀ *x* ∈# *M*. *P x* $\sqsubseteq_D$ *Q x* ⟹ *MultiSync S M P* $\sqsubseteq_D$ *MultiSync S M Q*›
apply (*induct M rule*: *induct-subset-mset-empty-single*; *simp*)
oops

lemma *mono-MultiSync-T*: ‹$\forall x \in\# M$. $P\ x \sqsubseteq_T Q\ x \Longrightarrow$ *MultiSync* $S\ M\ P \sqsubseteq_T$ *MultiSync* $S\ M\ Q$›
apply (*induct M rule*: *induct-subset-mset-empty-single*; *simp*)
oops

lemma *mono-MultiSync-DT*[*simp*, *elim*]:
‹$\forall x \in\# M$. $P\ x \sqsubseteq_{DT} Q\ x \Longrightarrow$ *MultiSync* $S\ M\ P \sqsubseteq_{DT}$ *MultiSync* $S\ M\ Q$›
and *mono-MultiSync-FD*[*simp*, *elim*]:
‹$\forall x \in\# M$. $P\ x \sqsubseteq_{FD} Q\ x \Longrightarrow$ *MultiSync* $S\ M\ P \sqsubseteq_{FD}$ *MultiSync* $S\ M\ Q$›
by (*induct M rule*: *induct-subset-mset-empty-single*; *simp*)+

find-theorems *name*: *mset name*: *ind*
lemmas *mono-MultiInter-DT*[*simp*, *elim*] = *mono-MultiSync-DT*[**where** S = ‹{}›]
and *mono-MultiInter-FD*[*simp*, *elim*] = *mono-MultiSync-FD*[**where** S = ‹{}›]
and *mono-MultiPar-DT*[*simp*, *elim*] = *mono-MultiSync-DT*[**where** S = ‹*UNIV*›]
and *mono-MultiPar-FD*[*simp*, *elim*] = *mono-MultiSync-FD*[**where** S = ‹*UNIV*›]

lemma *mono-MultiSeq-F*:
‹$\forall x \in$ *set* L. $P\ x \sqsubseteq_F Q\ x \Longrightarrow$ *MultiSeq* $L\ P \sqsubseteq_F$ *MultiSeq* $L\ Q$›
apply (*induct L*, *fastforce*) **apply** *simp* **oops**

lemma *mono-MultiSeq-D*:
‹$\forall x \in$ *set* L. $P\ x \sqsubseteq_D Q\ x \Longrightarrow$ *MultiSeq* $L\ P \sqsubseteq_D$ *MultiSeq* $L\ Q$›
apply (*induct L*, *fastforce*) **apply** *simp* **oops**

lemma *mono-MultiSeq-T*:
‹$\forall x \in$ *set* L. $P\ x \sqsubseteq_T Q\ x \Longrightarrow$ *MultiSeq* $L\ P \sqsubseteq_T$ *MultiSeq* $L\ Q$›
apply (*induct L*, *fastforce*) **apply** *simp* **oops**

lemma *mono-MultiSeq-DT*[*simp*, *elim*]:
‹$\forall x \in$ *set* L. $P\ x \sqsubseteq_{DT} Q\ x \Longrightarrow$ *MultiSeq* $L\ P \sqsubseteq_{DT}$ *MultiSeq* $L\ Q$›
and *mono-MultiSeq-FD*[*simp*, *elim*]:
‹$\forall x \in$ *set* L. $P\ x \sqsubseteq_{FD} Q\ x \Longrightarrow$ *MultiSeq* $L\ P \sqsubseteq_{FD}$ *MultiSeq* $L\ Q$›
by (*induct L*) *fastforce*+

lemma *mono-GlobalNdet*[*simp*] : ‹*GlobalNdet* $A\ P \sqsubseteq$ *GlobalNdet* $A\ Q$›
if ‹$\forall x \in A$. $P\ x \sqsubseteq Q\ x$›
proof (*cases* ‹$A = \{\}$›)
show ‹$A = \{\} \Longrightarrow$ *GlobalNdet* $A\ P \sqsubseteq$ *GlobalNdet* $A\ Q$› **by** *simp*
next
assume ‹$A \neq \{\}$›
show ‹*GlobalNdet* $A\ P \sqsubseteq$ *GlobalNdet* $A\ Q$›
unfolding *le-approx-def*

proof (*intro conjI impI allI subsetI*)
  show ‹$s \in \mathcal{D}$ (*GlobalNdet A Q*) $\Longrightarrow s \in \mathcal{D}$ (*GlobalNdet A P*)› **for** *s*
    using *that*[*rule-format*, *THEN le-approx1*] **by** (*auto simp add*: *D-GlobalNdet* ‹$A \neq \{\}$›)
next
  show ‹$s \notin \mathcal{D}$ (*GlobalNdet A P*) $\Longrightarrow \mathcal{R}_a$ (*GlobalNdet A P*) $s = \mathcal{R}_a$ (*GlobalNdet A Q*) *s*› **for** *s*
    using *that*[*rule-format*, *THEN le-approx2*]
    by (*auto simp add*: *D-GlobalNdet Ra-def F-GlobalNdet* ‹$A \neq \{\}$›)
next
  show ‹$s \in$ *min-elems* ($\mathcal{D}$ (*GlobalNdet A P*)) $\Longrightarrow s \in \mathcal{T}$ (*GlobalNdet A Q*)› **for** *s*
    using *that*[*rule-format*, *THEN le-approx3*]
    by (*simp add*: *D-GlobalNdet T-GlobalNdet* ‹$A \neq \{\}$› *min-elems-def*) *blast*
qed
qed

lemma *mono-GlobalNdet-F*[*simp*, *elim*]:
  ‹$\forall x \in A$. $P\ x \sqsubseteq_F Q\ x \Longrightarrow$ *GlobalNdet A P* $\sqsubseteq_F$ *GlobalNdet A Q*›
  and *mono-GlobalNdet-D*[*simp*, *elim*]:
  ‹$\forall x \in A$. $P\ x \sqsubseteq_D Q\ x \Longrightarrow$ *GlobalNdet A P* $\sqsubseteq_D$ *GlobalNdet A Q*›
  and *mono-GlobalNdet-T*[*simp*, *elim*]:
  ‹$\forall x \in A$. $P\ x \sqsubseteq_T Q\ x \Longrightarrow$ *GlobalNdet A P* $\sqsubseteq_T$ *GlobalNdet A Q*›
  and *mono-GlobalNdet-DT*[*simp*, *elim*]:
  ‹$\forall x \in A$. $P\ x \sqsubseteq_{DT} Q\ x \Longrightarrow$ *GlobalNdet A P* $\sqsubseteq_{DT}$ *GlobalNdet A Q*›
  and *mono-GlobalNdet-FD*[*simp*, *elim*]:
  ‹$\forall x \in A$. $P\ x \sqsubseteq_{FD} Q\ x \Longrightarrow$ *GlobalNdet A P* $\sqsubseteq_{FD}$ *GlobalNdet A Q*›
  unfolding *failure-refine-def divergence-refine-def trace-refine-def*
    trace-divergence-refine-def failure-divergence-refine-def le-ref-def
  by (*auto simp add*: *F-GlobalNdet D-GlobalNdet T-GlobalNdet*)

lemma *GlobalNdet-refine-FD-subset*:
  ‹$A \neq \{\} \Longrightarrow A \subseteq B \Longrightarrow$ *GlobalNdet B P* $\sqsubseteq_{FD}$ *GlobalNdet A P*›
  by (*metis mono-Ndet-FD-left GlobalNdet-factorization-union*
        bot.extremum-uniqueI idem-FD le-iff-sup)

lemma *GlobalNdet-refine-F-subset*:
  ‹$A \neq \{\} \Longrightarrow A \subseteq B \Longrightarrow$ *GlobalNdet B P* $\sqsubseteq_F$ *GlobalNdet A P*›
  by (*simp add*: *GlobalNdet-refine-FD-subset leFD-imp-leF*)

lemma *GlobalNdet-refine-FD*: ‹$a \in A \Longrightarrow$ *GlobalNdet A P* $\sqsubseteq_{FD}$ *P a*›
  using *GlobalNdet-refine-FD-subset*[*of* ‹$\{a\}$› *A*] **by** *simp*

lemma *GlobalNdet-refine-F*: ‹$a \in A \Longrightarrow$ *GlobalNdet A P* $\sqsubseteq_F$ *P a*›
  by (*simp add*: *GlobalNdet-refine-FD leFD-imp-leF*)

lemma *mono-GlobalNdet-FD-const*:
  ‹$A \neq \{\} \Longrightarrow \forall x \in A$. $P \sqsubseteq_{FD} Q\ x \Longrightarrow P \sqsubseteq_{FD}$ *GlobalNdet A Q*›

by (*metis GlobalNdet-id mono-GlobalNdet-FD*)

lemma *mono-GlobalNdet-F-const*:
‹$A \neq \{\} \Longrightarrow \forall x \in A.\ P \sqsubseteq_F Q\ x \Longrightarrow P \sqsubseteq_F GlobalNdet\ A\ Q$›
by (*metis GlobalNdet-id mono-GlobalNdet-F*)

9.2 Combination of Multi-Operators Laws

lemma *finite-Mprefix-is-MultiDet*:
‹$finite\ A \Longrightarrow (\Box\ p \in A \rightarrow P\ p) = \Box\ p \in A.\ (p \rightarrow P\ p)$›
by (*induct rule*: *finite-induct*, *simp-all add*: *Mprefix-STOP*)
(*metis Mprefix-Un-distrib Mprefix-singl insert-is-Un*)

lemma *finite-Mndetprefix-is-MultiNdet*:
‹$finite\ A \Longrightarrow Mndetprefix\ A\ P = \bigsqcap\ p \in A.\ (p \rightarrow P\ p)$›
by (*cases* ‹$A = \{\}$›, *simp*, *rotate-tac*, *induct rule*: *finite-set-induct-nonempty*)
(*simp-all add*: *Mndetprefix-unit Mndetprefix-distrib-unit*)

lemma ‹$Q \Box (\bigsqcap\ p \in \{\}.\ P\ p) = \bigsqcap\ p \in \{\}.\ (Q \Box P\ p) \Longrightarrow Q = STOP$›
by (*simp add*: *Det-STOP*)

lemma *Det-MultiNdet-distrib*:
‹$A \neq \{\} \Longrightarrow finite\ A \Longrightarrow M \Box (\bigsqcap\ p \in A.\ P\ p) = \bigsqcap\ p \in A.\ M \Box P\ p$›
by (*erule finite-set-induct-nonempty*, *simp-all add*: *Det-distrib*)

lemma ‹$M \sqcap (\Box\ p \in \{\}.\ P\ p) = \Box\ p \in \{\}.\ (M \sqcap P\ p) \Longrightarrow M \sqcap STOP = STOP$›
by *simp*

lemma *Ndet-MultiDet-distrib*:
‹$A \neq \{\} \Longrightarrow finite\ A \Longrightarrow M \sqcap (\Box\ p \in A.\ P\ p) = \Box\ p \in A.\ M \sqcap P\ p$›
by (*erule finite-set-induct-nonempty*, *simp-all add*: *Ndet-distrib*)

lemma *MultiNdet-Sync-left-distrib*:
‹$B \neq \{\} \Longrightarrow finite\ B \Longrightarrow (\bigsqcap\ a \in B.\ P\ a)\ [\![S]\!]\ M = \bigsqcap\ a \in B.\ (P\ a\ [\![S]\!]\ M)$›
by (*induct rule*: *finite-set-induct-nonempty*)
(*simp-all add*: *Sync-Ndet-left-distrib*)

lemma *MultiNdet-Sync-right-distrib*:
‹$B \neq \{\} \Longrightarrow finite\ B \Longrightarrow M\ [\![S]\!]\ MultiNdet\ B\ P = \bigsqcap\ a{\in}B.\ (M\ [\![S]\!]\ P\ a)$›
by (*subst Sync-commute*, *subst MultiNdet-Sync-left-distrib*)
(*simp-all add*: *Sync-commute*)

lemma *Sync-MultiNdet-cartprod*:
‹$A \neq \{\} \Longrightarrow$ *finite* $A \Longrightarrow B \neq \{\} \Longrightarrow$ *finite* $B \Longrightarrow$
$(\bigsqcap\ (s,\ t) \in A \times B.\ (x\ s\ [\![S]\!]\ y\ t)) = (\bigsqcap\ s \in A.\ x\ s)\ [\![S]\!]\ (\bigsqcap\ t \in B.\ y\ t)$›
apply (*subst MultiNdet-cartprod*, *assumption*+)
apply (*subst MultiNdet-Sync-left-distrib*, *assumption*+)
apply (*subst MultiNdet-Sync-right-distrib*, *assumption*+)
by *presburger*

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

lemmas *MultiNdet-Inter-left-distrib* =
MultiNdet-Sync-left-distrib[**where** S = ‹{}›]
and *MultiNdet-Par-left-distrib* =
MultiNdet-Sync-left-distrib[**where** S = ‹*UNIV*›]

lemma *Seq-MultiNdet-distribR*:
‹*finite* $A \Longrightarrow (\bigsqcap\ p \in A.\ P\ p)\ \mathbf{;}\ S = (\bigsqcap\ p \in A.\ (P\ p\ \mathbf{;}\ S))$›
by (*induct A rule*: *finite-induct*, *simp add*: *STOP-Seq*)
(*metis MultiNdet-insert′ MultiNdet-rec1 Seq-Ndet-distrR*)

lemma *Seq-MultiNdet-distribL*:
‹$A \neq \{\} \Longrightarrow$ *finite* $A \Longrightarrow S\ \mathbf{;}\ (\bigsqcap\ p \in A.\ P\ p) = (\bigsqcap\ p \in A.\ (S\ \mathbf{;}\ P\ p))$›
by (*induct A rule*: *finite-set-induct-nonempty*, *simp-all add*: *Seq-Ndet-distrL*)

lemma *prefix-MultiNdet-distrib*:
‹$A \neq \{\} \Longrightarrow$ *finite* $A \Longrightarrow (a \rightarrow (\bigsqcap\ p \in A.\ P\ p) = \bigsqcap\ p \in A.\ (a \rightarrow P\ p))$›
by (*induct A rule*: *finite-set-induct-nonempty*, *simp-all add*: *write0-Ndet*)

lemma *Mndetprefix-MultiNdet-distrib*:
‹$(\sqcap\ q \in B \rightarrow (\bigsqcap\ p \in A.\ P\ p\ q)) = \bigsqcap\ p \in A.\ \sqcap\ q \in B \rightarrow P\ p\ q$›
if *finB*: ‹*finite B*› **and** *nonemptyA*: ‹$A \neq \{\}$› **and** *finA*: ‹*finite A*›
proof (*cases* ‹$B = \{\}$›)
case *True* **thus** *?thesis* **by** (*simp add*: *MultiNdet-STOP-id finA*)
next
case *False* **thus** *?thesis*
proof (*insert finB*, *induct B rule*: *finite-set-induct-nonempty*)
case (*singleton a*)
thus *?case*
by (*simp add*: *Mndetprefix-unit finA prefix-MultiNdet-distrib nonemptyA*)
next

case (*insertion x F*)
show *?case*
apply (*subst Mndetprefix-Un-distrib*[*of* ‹{*x*}›, *simplified*, *OF* ‹*F* ≠ {}›])
apply (*subst Mndetprefix-unit*,
subst prefix-MultiNdet-distrib[*OF nonemptyA finA*])
apply (*subst insertion.hyps*(*5*))
apply (*subst MultiNdet-Ndet*[*OF finA*])
by (*insert* ‹*F* ≠ {}› ‹*x* ∉ *F*›, *subst Mndetprefix-distrib-unit*) *force+*
qed
qed

lemma *MultiDet-Mprefix*:
‹*finite A* ⟹ (□*a* ∈ *A*. □*x* ∈ *S a* → *P a x*) =
□*x* ∈ (⋃ *a* ∈ *A*. *S a*) → ⨅ *a* ∈ {*a* ∈ *A*. *x* ∈ *S a*}. *P a x*›
proof (*induct A rule*: *induct-subset-empty-single*)
case *1*
show *?case* **by** (*simp add*: *Mprefix-STOP*)
next
case *2*
show *?case*
by (*simp*, *intro ballI mono-Mprefix-eq*)
(*simp add*: *Collect-conv-if*)
next
case (*3 F a*)
show *?case*
apply (*simp del*: *MultiDet-insert add*: ‹*finite F*›)
apply (*subst 3.hyps*)
apply (*subst Mprefix-Det-distr*)
apply (*intro mono-Mprefix-eq ballI*)
using ‹*finite F*› **by** (*auto simp add*: *Process-eq-spec F-MultiNdet F-Ndet D-MultiNdet D-Ndet*)
qed

lemma *MultiDet-prefix-is-MultiNdet-prefix*:
‹*finite A* ⟹ (□ *p* ∈ *A*. (*a* → *P p*)) = ⨅ *p* ∈ *A*. (*a* → *P p*)›
by (*induct A rule*: *induct-subset-empty-single*, *simp*, *simp*)
(*metis MultiDet-insert′ MultiNdet-insert′ prefix-MultiNdet-distrib write0-Det-Ndet*)

lemma *prefix-MultiNdet-is-MultiDet-prefix*:
‹*A* ≠ {} ⟹ *finite A* ⟹ (*a* → (⨅ *p* ∈ *A*. *P p*) = □ *p* ∈ *A*. (*a* → *P p*))›
by (*simp add*: *MultiDet-prefix-is-MultiNdet-prefix prefix-MultiNdet-distrib*)

lemma *Mprefix-MultiNdet-distrib′*:
‹*finite B* ⟹ *A* ≠ {} ⟹ *finite A* ⟹

(□ q ∈ B → ⨅ p ∈ A. P p q) = □ p ∈ A. □ q ∈ B → P p q›
proof (*induct B rule*: *finite-induct*)
  case *empty*
  thus *?case* **by** (*simp add*: *Mprefix-STOP MultiDet-STOP-id*)
next
  case (*insert x F*)
  show *?case*
    apply (*subst* (*1 2*) *Mprefix-Un-distrib*[*of* ‹{x}› *F*, *simplified*])
    apply (*subst Mprefix-singl*, *subst prefix-MultiNdet-distrib*[*OF insert.prems*])
    apply (*subst MultiDet-prefix-is-MultiNdet-prefix*[*symmetric*, *OF* ‹*finite A*›])
    apply (*subst insert.hyps*(*3*)[*OF insert.prems*])
    apply (*subst MultiDet-Det*[*OF* ‹*finite A*›],
          rule mono-MultiDet-eq[*OF* ‹*finite A*›])
    by (*subst Mprefix-singl*) *fast*
qed

lemma *MultiSync-Hiding-pseudo-distrib*:
  ‹*finite B* ⟹ B ∩ S = {} ⟹
   (⟦S⟧ p ∈# M. ((P p) \ B)) = (⟦S⟧ p ∈# M. P p) \ B›
  by (*induct M*, *simp add*: *Hiding-set-STOP*)
     (*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*: *prefix-Sync2*)

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*]

lemma *Hiding-MultiNdet-distrib*:
  ‹*finite A* ⟹ (⨅ p ∈ A. P p) \ B = (⨅ p ∈ A. (P p \ B))›
  by (*cases* ‹A = {}›, *simp add*: *Hiding-set-STOP*, *rotate-tac*)
     (*induct A rule*: *finite-set-induct-nonempty*, *simp-all add*: *Hiding-Ndet*)

lemma *Mndetprefix-Hiding-is-MultiNdet-prefix-Hiding*:
  ‹*finite A* ⟹ ⊓ p ∈ A − B → ((P p) \ B) = ⨅ p ∈ A − B. (p → ((P p) \ B))›
proof (*induct A rule*: *finite-induct*)
  case *empty*
  thus *?case* **by** *fastforce*
next
  show ‹*finite F* ⟹ x ∉ F ⟹

$\sqcap$ *p* ∈ *F* − *B* → (*P p* \ *B*) = $\bigsqcap$ *p* ∈ *F* − *B*. (*p* → (*P p* \ *B*)) ⟹
$\sqcap$ *p* ∈ *insert x F* − *B* → (*P p* \ *B*) =
$\bigsqcap$ *p* ∈ *insert x F* − *B*. (*p* → (*P p* \ *B*))› **for** *x F*
apply (*cases* ‹*x* ∈ *B*›, *simp*)
apply (*cases* ‹*F* − *B* = {}›,
metis (*no-types*, *lifting*) *Mndetprefix-unit MultiNdet-rec1 insert-Diff-if*)
by (*simp add*: *Mndetprefix-distrib-unit insert-Diff-if*)
qed

lemma *Hiding-Mndetprefix-is-MultiNdet-Hiding*:
‹*finite A* ⟹ *A* ⊆ *B* ⟹ ($\sqcap$ *a* ∈ *A* → *P*) \ *B* = $\bigsqcap$ *a* ∈ *A*. (*P* \ *B*)›
by (*cases* ‹*A* = {}›, *simp add*: *Hiding-set-STOP*, *rotate-tac 2*)
(*induct A rule*: *finite-set-induct-nonempty*,
simp-all add: *Mndetprefix-unit Hiding-Ndet Hiding-write0*
Mndetprefix-distrib-unit)

lemma *MultiSync-Mprefix-pseudo-distrib*:
‹(⟦*S*⟧ *B* ∈# *M*. □ *x* ∈ *B* → *P B x*) =
□ *x* ∈ ($\bigcap$*B* ∈ *set-mset M*. *B*) → (⟦*S*⟧ *B* ∈# *M*. *P B x*)›
if *nonempty*: ‹*M* ≠ {#}› **and** *hyp*: ‹∀ *B* ∈# *M*. *B* ⊆ *S*›
proof−
from *nonempty* **obtain** *b M′* **where** ‹*b* ∈# *M* − *M′*›
‹ *M* = *add-mset b M′*› ‹*M′* ⊆# *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-mset b M′*›)
using ‹*b* ∈# *M* − *M′*› ‹*M′* ⊆# *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-distr-subset*[*symmetric*])
apply (*meson add.hyps*(*1*) *hyp in-diffD*,
metis ‹*b* ∈# *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*]

A result on Mndetprefix and Sync.

lemma *Mndetprefix-Sync-distr*: ‹*A* ≠ {} ⟹ *B* ≠ {} ⟹

$(\sqcap\ a \in A \rightarrow P\ a)\ [\![S]\!]\ (\sqcap\ b \in B \rightarrow Q\ b) =$
$\sqcap\ a{\in}A.\ \sqcap\ b{\in}B.\ (\square c \in \{a\} - S \rightarrow (P\ a\ [\![S]\!]\ (b \rightarrow Q\ b)))\ \square$
$(\square d \in \{b\} - S \rightarrow ((a \rightarrow P\ a)\ [\![S]\!]\ Q\ b))\ \square$
$(\square c{\in}\{a\} \cap \{b\} \cap S \rightarrow (P\ a\ [\![S]\!]\ Q\ b))$›
apply (*subst* (*1 2*) *Mndetprefix-GlobalNdet*)
apply (*subst GlobalNdet-Sync-distr*, *assumption*)
apply (*subst Sync-commute*)
apply (*subst GlobalNdet-Sync-distr*, *assumption*)
apply (*subst Sync-commute*)
apply (*unfold write0-def*)
apply (*subst Mprefix-Sync-distr-bis*)
by (*fold write0-def*) *blast*

A result on *MultiSeq* with *non-terminating*.

lemma *non-terminating-MultiSeq*:
‹(*SEQ a* ∈@ *L*. *P a*) =
SEQ a ∈@ *take* (*Suc* (*first-elem* (λ*a*. *non-terminating* (*P a*)) *L*)) *L*. *P a*›
by (*induct L*; *simp add*: *non-terminating-Seq*)

9.3 Results on *Renaming*

lemma *Renaming-GlobalNdet*:
‹*Renaming* (⊓ *a* ∈ *A*. *P* (*f a*)) *f* = ⊓ *b* ∈ *f* ‘ *A*. *Renaming* (*P b*) *f*›
by (*subst Process-eq-spec*)
(*auto simp add*: *F-Renaming D-Renaming F-GlobalNdet D-GlobalNdet*)

lemma *Renaming-GlobalNdet-inj-on*:
‹*Renaming* (⊓ *a* ∈ *A*. *P a*) *f* =
⊓ *b* ∈ *f* ‘ *A*. *Renaming* (*P* (*THE a*. *a* ∈ *A* ∧ *f a* = *b*)) *f*›
if *inj-on-f*: ‹*inj-on f A*›
apply (*subst Renaming-GlobalNdet*[*symmetric*])
apply (*intro arg-cong*[**where** *f* = ‹λ*Q*. *Renaming Q f*›] *mono-GlobalNdet-eq*)
by (*metis inj-on-f the-inv-into-def the-inv-into-f-f*)

corollary *Renaming-GlobalNdet-inj*:
‹*Renaming* (⊓ *a* ∈ *A*. *P a*) *f* =
⊓ *b* ∈ *f* ‘ *A*. *Renaming* (*P* (*THE a*. *f a* = *b*)) *f*› **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-MultiNdet*: ‹*finite A* ⟹ *Renaming* (⨅ *a* ∈ *A*. *P* (*f a*)) *f* =
⨅ *b* ∈ *f* ‘ *A*. *Renaming* (*P b*) *f*›
by (*subst* (*1 2*) *finite-GlobalNdet-is-MultiNdet*[*symmetric*])
(*simp-all add*: *Renaming-GlobalNdet*)

lemma *Renaming-MultiNdet-inj-on*:
‹*finite A* ⟹ *inj-on f A* ⟹
Renaming (⨅ *a* ∈ *A*. *P a*) *f* =
⨅ *b* ∈ *f* ‘ *A*. *Renaming* (*P* (*THE a*. *a* ∈ *A* ∧ *f a* = *b*)) *f*›
by (*subst* (*1 2*) *finite-GlobalNdet-is-MultiNdet*[*symmetric*])
(*simp-all add*: *Renaming-GlobalNdet-inj-on*)

corollary *Renaming-MultiNdet-inj*:
‹*finite A* ⟹ *inj f* ⟹
Renaming (⨅ *a* ∈ *A*. *P a*) *f* = ⨅ *b* ∈ *f* ‘ *A*. *Renaming* (*P* (*THE a*. *f a* = *b*)) *f*›
by (*subst* (*1 2*) *finite-GlobalNdet-is-MultiNdet*[*symmetric*])
(*simp-all add*: *Renaming-GlobalNdet-inj*)

lemma *Renaming-MultiDet*:
‹*finite A* ⟹ *Renaming* (□ *a* ∈ *A*. *P* (*f a*)) *f* =
□ *b* ∈ *f* ‘ *A*. *Renaming* (*P b*) *f*›
apply (*induct A rule*: *finite-induct*)
by (*simp-all add*: *Renaming-STOP Renaming-Det del*: *MultiDet-insert*)

lemma *Renaming-MultiDet-inj-on*:
‹*Renaming* (□ *a* ∈ *A*. *P a*) *f* =
□ *b* ∈ *f* ‘ *A*. *Renaming* (*P* (*THE a*. *a* ∈ *A* ∧ *f a* = *b*)) *f*›
if *finite-A*: ‹*finite A*› **and** *inj-on-f*: ‹*inj-on f A*›
apply (*subst Renaming-MultiDet*[*OF finite-A*, *symmetric*])
apply (*intro arg-cong*[**where** *f* = ‹λ*Q*. *Renaming Q f*›]
mono-MultiDet-eq finite-A)
by (*metis inj-on-f the-inv-into-def the-inv-into-f-f*)

corollary *Renaming-MultiDet-inj*:
‹*Renaming* (□ *a* ∈ *A*. *P a*) *f* = □ *b* ∈ *f* ‘ *A*. *Renaming* (*P* (*THE a*. *f a* = *b*)) *f*›
if *finite-A*: ‹*finite A*› **and** *inj-f*: ‹*inj f*›
apply (*subst Renaming-MultiDet-inj-on*[*OF finite-A*], *metis inj-eq inj-onI inj-f*)
apply (*rule mono-MultiDet-eq*[*rule-format*], *fact finite-imageI*[*OF finite-A*])
by (*metis imageE inj-eq*[*OF inj-f*])

lemma *Renaming-MultiSeq*:
‹*Renaming* (*SEQ l* ∈@ *L*. *P* (*f l*)) *f* = *SEQ m* ∈@ *map f L*. *Renaming* (*P m*) *f*›
by (*induct L*, *simp-all add*: *Renaming-SKIP Renaming-Seq*)

lemma *Renaming-MultiSeq-inj-on*:
‹*Renaming* (*SEQ l* ∈@ *L*. *P l*) *f* =
SEQ m ∈@ *map f L*. *Renaming* (*P* (*THE l*. *l* ∈ *set L* ∧ *f l* = *m*)) *f*›
if *inj-on-f*: ‹*inj-on f* (*set L*)›
apply (*subst Renaming-MultiSeq*[*symmetric*])

apply (*intro arg-cong*[**where** *f* = ‹λ*Q*. *Renaming Q f*›] *mono-MultiSeq-eq*)
by (*metis that the-inv-into-def the-inv-into-f-f*)

corollary *Renaming-MultiSeq-inj*:
‹*Renaming* (*SEQ l* ∈@ *L*. *P l*) *f* =
SEQ m ∈@ *map f L*. *Renaming* (*P* (*THE l*. *f l* = *m*)) *f*› **if** *inj-f*: ‹*inj f*›
apply (*subst Renaming-MultiSeq-inj-on*, *metis inj-eq inj-onI inj-f*)
apply (*rule mono-MultiSeq-eq*[*rule-format*])
by (*metis* (*mono-tags*, *opaque-lifting*) *inj-image-mem-iff list*.*set-map inj-f*)

end

Chapter 10

Example: Dining Philosophers

theory *DiningPhilosophers*
imports *CSPM*
begin

10.1 Classic Version

We formalize here the Dining Philosophers problem with a locale.

locale *DiningPhilosophers* =

fixes $N::nat$
assumes $N\text{-}g1[simp] : \langle N > 1 \rangle$
— 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 $dining\text{-}event = picks\ (phil{:}nat)\ (fork{:}nat)$
$\quad\quad | \ putsdown\ (phil{:}nat)\ (fork{:}nat)$

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

definition $RPHIL{::}\ \langle nat \Rightarrow dining\text{-}event\ process\rangle$
where $\langle RPHIL\ i \equiv \mu\ X.\ (picks\ i\ i \rightarrow (picks\ i\ ((i-1)\ mod\ N) \rightarrow$
$(putsdown\ i\ ((i-1)\ mod\ N) \rightarrow (putsdown\ i\ i \rightarrow X))))\rangle$

definition $LPHIL0{::}\ \langle dining\text{-}event\ process\rangle$
where $\langle LPHIL0 \equiv \mu\ X.\ (picks\ 0\ (N-1) \rightarrow (picks\ 0\ 0 \rightarrow$
$(putsdown\ 0\ 0 \rightarrow (putsdown\ 0\ (N-1) \rightarrow X))))\rangle$

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

10.2 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 $\mu\ x.\ P\ x$).

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

end

Chapter 11

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.

theory *POTS*
  imports *CSPM*
begin

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

11.1 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*)}›

11.2 Auxilliaries to Substructure the Specification

abbreviation *Sliding* :: ‹*'α process* ⇒ *'α process* ⇒ *'α process*› (**infixl** ‹▷› *78*)
where ‹*P* ▷ *Q* ≡ (*P* □ *Q*) ⊓ *Q*›
— This operator is also called Timeout, more studied in future theories.

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*)›

11.3 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*}›

11.4 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)
                  )
                )›
```

11.5 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 *MultiNdet* which appears in *Oside* recursive definition.

11.6 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*)›

11.7 Toplevel Proof-Goals

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

lemma ‹$\forall$ *p* ∈ *phones*. *deadlock-free* (*Tel p*)› **oops**
lemma ‹$\forall$ *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 ‹$\forall$ *p* ∈ *phones*. *lifelock-free* (*ImplementT p*)› **oops**
lemma ‹$\forall$ *p* ∈ *phones*. *Tel p* $\sqsubseteq_{FD}$ *ImplementT p*› **oops**

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

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

end

end

Chapter 12

Results on *events-of*

theory *EventsProperties*
  imports *CSPM*
begin

12.1 With Operators of HOL-CSP

lemma *events-of-def-tickFree*:
  ‹*events-of P* = (⋃ *t*∈{*t* ∈ 𝒯 *P*. *tickFree t*}. {*a*. *ev a* ∈ *set t*})›
proof −
  have ‹*s* ∈ 𝒯 *P* ⟹ ¬ *tickFree s* ⟹ *ev e* ∈ *set s* ⟹ *ev e* ∈ *set* (*butlast s*)› **for** *e s*
    by (*cases s rule*: *rev-cases*) (*simp-all add*: *append-single-T-imp-tickFree*)
  thus *?thesis*
    by (*auto simp add*: *events-of-def*)
      (*metis butlast-snoc front-tickFree-butlast is-processT2-TR*
        is-processT3-ST nonTickFree-n-frontTickFree)
qed

lemma *events-BOT*: ‹*events-of* ⊥ = *UNIV*›
  and *events-SKIP*: ‹*events-of SKIP* = {}›
  and *events-STOP*: ‹*events-of STOP* = {}›
  by (*auto simp add*: *events-of-def T-UU T-SKIP T-STOP*)
    (*meson front-tickFree-single list.set-intros*(*1*))

lemma *anti-mono-events-T*: ‹*P* $\sqsubseteq_T$ *Q* ⟹ *events-of Q* ⊆ *events-of P*›
  unfolding *trace-refine-def events-of-def* **by** *fast*

lemma *anti-mono-events-F*: ‹*P* $\sqsubseteq_F$ *Q* ⟹ *events-of Q* ⊆ *events-of P*›
  by (*intro anti-mono-events-T leF-imp-leT*)

lemma *anti-mono-events-FD*: ‹*P* $\sqsubseteq_{FD}$ *Q* ⟹ *events-of Q* ⊆ *events-of P*›
  by (*intro anti-mono-events-F leFD-imp-leF*)

lemmas *events-fix-prefix* =
events-DF[*of* ‹{*a*}›, *simplified DF-def Mndetprefix-unit*] **for** *a*

lemma *events-Mndetprefix*:
‹*events-of* (*Mndetprefix A P*) = *A* ∪ (⋃ *a*∈*A*. *events-of* (*P a*))›
apply (*cases* ‹*A* = {}›, *simp add*: *events-STOP*)
unfolding *events-of-def*
apply (*simp add*: *T-Mndetprefix T-Mprefix write0-def*, *safe*; *simp*)
apply (*metis event.inject list.exhaust-sel set-ConsD*)
by (*metis Nil-elem-T list.sel*(*1*, *3*) *list.set-intros*(*1*))
(*metis list.sel*(*1*, *3*) *list.set-intros*(*2*))

lemma *events-Mprefix*:
‹*events-of* (*Mprefix A P*) = *A* ∪ (⋃ *a*∈*A*. *events-of* (*P a*))›
apply (*rule subset-antisym*)
apply (*rule subset-trans*[*OF anti-mono-events-FD*[*OF Mprefix-refines-Mndetprefix-FD*]],
simp add: *events-Mndetprefix*)
unfolding *events-of-def*
apply (*simp add*: *T-Mprefix*, *safe*; *simp*)
by (*metis Nil-elem-T list.sel*(*1*, *3*) *list.set-intros*(*1*))
(*metis list.sel*(*1*, *3*) *list.set-intros*(*2*))

lemma *events-prefix*: ‹*events-of* (*a* → *P*) = *insert a* (*events-of P*)›
unfolding *write0-def* **by** (*simp add*: *events-Mprefix*)

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

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

lemma *events-Renaming*:
‹*events-of* (*Renaming P f*) = (*if* 𝒟 *P* = {} *then f* ‘ *events-of P else UNIV*)›
proof (*split if-split*, *intro conjI impI*)
show ‹𝒟 *P* ≠ {} ⟹ *events-of* (*Renaming P f*) = *UNIV*›
by (*rule events-div*, *simp add*: *D-Renaming*)
(*metis D-imp-front-tickFree ex-in-conv front-tickFree-charn*
front-tickFree-implies-tickFree is-processT9-S-swap nonTickFree-n-frontTickFree)
next
show ‹*events-of* (*Renaming P f*) = *f* ‘ *events-of P*› **if** *div-free* : ‹𝒟 *P* = {}›
proof (*intro subset-antisym subsetI*)

fix *e*
assume ‹*e* ∈ *events-of* (*Renaming P f*)›
then obtain *s t* **where** ∗ : ‹*t* ∈ $\mathcal{T}$ *P*› ‹*s* = *map* (*EvExt f*) *t*› ‹*ev e* ∈ *set s*›
by (*auto simp add*: *events-of-def T-Renaming div-free*)
from ∗(*2*, *3*) **obtain** *e′* **where** ‹*e* = *f e′*› ‹*ev e′* ∈ *set t*›
by (*auto simp add*: *EvExt-def split*: *event.split-asm*)
with ∗(*1*) **show** ‹*e* ∈ *f* ‘ *events-of P*›
unfolding *events-of-def* **by** *blast*
next
fix *e*
assume ‹*e* ∈ *f* ‘ *events-of P*›
then obtain *e′* **where** ∗ : ‹*e* = *f e′*› ‹*e′* ∈ *events-of P*› **by** *blast*
from ∗(*2*) **obtain** *t* **where** ‹*t* ∈ $\mathcal{T}$ *P*› ‹*ev e′* ∈ *set t*›
unfolding *events-of-def* **by** *blast*
thus ‹*e* ∈ *events-of* (*Renaming P f*)›
apply (*simp add*: *events-of-def T-Renaming*)
apply (*rule disjI1*)
apply (*rule-tac x* = ‹*map* (*EvExt f*) *t*› **in** *exI*)
using ∗(*1*) **by** (*auto simp add*: *EvExt-def image-iff split*: *event.split*)
qed
qed

lemma *events-Seq*:
‹*events-of* (*P* **;** *Q*) =
(*if non-terminating P then events-of P else events-of P* ∪ *events-of Q*)›
proof (*split if-split*, *intro conjI impI*)
show ‹*non-terminating P* ⟹ *events-of* (*P* **;** *Q*) = *events-of P*›
by (*simp add*: *non-terminating-Seq*)
next
show ‹*events-of* (*P* **;** *Q*) = *events-of P* ∪ *events-of Q*› **if** ‹¬ *non-terminating P*›
proof (*intro subset-antisym subsetI*)
show ‹*e* ∈ *events-of* (*P* **;** *Q*) ⟹ *e* ∈ *events-of P* ∪ *events-of Q*› **for** *e*
by (*auto simp add*: *events-of-def T-Seq F-T D-T intro*: *is-processT3-ST*)
next
fix *s*
assume ‹*s* ∈ *events-of P* ∪ *events-of Q*›
then consider ‹*s* ∈ *events-of P*› | ‹*s* ∈ *events-of Q*› **by** *fast*
thus ‹*s* ∈ *events-of* (*P* **;** *Q*)›
proof *cases*
show ‹*s* ∈ *events-of P* ⟹ *s* ∈ *events-of* (*P* **;** *Q*)›
by (*simp add*: *events-of-def-tickFree T-Seq*)
(*metis Nil-elem-T append.right-neutral is-processT5-S7 singletonD*)
next
from *non-terminating-refine-DF that*
obtain *t1* **where** ∗ : ‹*t1* ∈ $\mathcal{T}$ *P*› ‹*t1* ∉ $\mathcal{T}$ (*DF UNIV*)›
by (*metis subsetI trace-refine-def*)

```
      then obtain t1' where ‹t1 = t1' @ [tick]›
        using DF-all-tickfree-traces2 T-nonTickFree-imp-decomp is-processT3-ST
by blast
      hence ‹t2 ∈ 𝒯 Q ⟹ t1' @ t2 ∈ 𝒯 (P ; Q)› for t2
        using ∗(1) T-Seq by blast
      thus ‹s ∈ events-of Q ⟹ s ∈ events-of (P ; Q)›
        by (simp add: events-of-def, elim bexE)
          (metis UnCI set-append)
    qed
  qed
qed


lemma events-Sync: ‹events-of (P ⟦S⟧ Q) ⊆ events-of P ∪ events-of Q›
  apply (subst events-of-def, subst T-Sync, simp add: subset-iff)
proof (intro allI impI conjI, goal-cases)
  case (1 a)
  thus ?case by (metis (no-types, lifting) UN-I events-of-def ftf-Sync1 mem-Collect-eq)

next
  case (2 a)
  then obtain t u where ‹t ∈ 𝒟 P ∧ u ∈ 𝒯 Q ∨ t ∈ 𝒟 Q ∧ u ∈ 𝒯 P› by blast
  thus ?case using events-div by blast
qed


lemma events-Inter:
  ‹events-of ((P :: 'α process) ||| Q) = events-of P ∪ events-of Q›
proof (rule subset-antisym[OF events-Sync])
  have ‹⟦tickFree s; s ∈ 𝒯 (P :: 'α process)⟧ ⟹ s ∈ 𝒯 (P ||| Q)› for s P Q
    apply (simp add: T-Sync)
    apply (rule disjI1)
    apply (rule-tac x = s in exI, simp)
    apply (rule-tac x = ‹[]› in exI, simp add: Nil-elem-T)
    by (metis Sync.sym emptyLeftSelf singletonD tickFree-def)
  hence ‹events-of (P :: 'α process) ⊆ events-of (P ||| Q)› for P Q
    unfolding events-of-def-tickFree by fast
  thus ‹events-of P ∪ events-of Q ⊆ events-of (P ||| Q)›
    by (metis Inter-commute Un-least)
qed


lemma empty-div-hide-events-Hiding: ‹events-of (P \ B) ⊆ events-of P − B›
  if ‹div-hide P B = {}›
  unfolding events-of-def T-Hiding
  apply (subst that, simp)
  using F-T by auto blast
```

lemma *not-empty-div-hide-events-Hiding*:
‹*div-hide P B* ≠ {} ⟹ *events-of* (*P* \ *B*) = *UNIV*›
using *D-Hiding events-div* **by** *blast*

events-of and *deadlock-free*

lemma *nonempty-events-if-deadlock-free*: ‹*deadlock-free P* ⟹ *events-of P* ≠ {}›
unfolding *deadlock-free-def events-of-def failure-divergence-refine-def le-ref-def*
apply (*simp add*: *div-free-DF*, *subst* (*asm*) *DF-unfold*)
apply (*simp add*: *F-Mndetprefix write0-def F-Mprefix subset-iff*)
by (*metis Nil-elem-T T-F-spec UNIV-I hd-in-set is-processT5-S7*
list.distinct(*1*) *self-append-conv2*)

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

lemma *nonempty-events-if-deadlock-free*$_{SKIP}$:
‹*deadlock-free*$_{SKIP}$ *P* ⟹ [*tick*] ∈ $\mathcal{T}$ *P* ∨ *events-of P* ≠ {}›
unfolding *deadlock-free*$_{SKIP}$*-def events-of-def failure-refine-def le-ref-def*
apply (*subst* (*asm*) *DF*$_{SKIP}$*-unfold*)
apply (*simp add*: *F-Mndetprefix write0-def F-Mprefix subset-iff F-Ndet F-SKIP*)
by (*metis Nil-elem-T T-F-spec UNIV-I hd-in-set is-processT5-S7*
list.distinct(*1*) *self-append-conv2*)

lemma *events-in-DF*$_{SKIP}$: ‹$DF_{SKIP}\ A \sqsubseteq_{FD} P$ ⟹ *events-of P* ⊆ *A*›
by (*metis anti-mono-events-FD events-DF*$_{SKIP}$)

lemma ‹¬ *events-of P* ⊆ *A* ⟹ ¬ $DF\ A \sqsubseteq_{FD} P$›
and ‹¬ *events-of P* ⊆ *A* ⟹ ¬ $DF_{SKIP}\ A \sqsubseteq_{FD} P$›
by (*metis anti-mono-events-FD events-DF*)
(*metis anti-mono-events-FD events-DF*$_{SKIP}$)

12.2 With Architectural Operators of HOL-CSPM

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

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

lemma *events-MultiSeq*:
‹*events-of* (*SEQ a* ∈@ *L*. *P a*) =
(⋃ *a* ∈ *set* (*take* (*Suc* (*first-elem* (λ*a*. *non-terminating* (*P a*)) *L*)) *L*).

events-of (*P a*))›
by (*subst non-terminating-MultiSeq*, *induct L*; *simp add*: *events-SKIP events-Seq*)

lemma *events-MultiSeq-subset*:
‹*events-of* (*SEQ a* ∈@ *L*. *P a*) ⊆ (⋃ *a* ∈ *set L*. *events-of* (*P a*))›
using *in-set-takeD* **by** (*subst events-MultiSeq*) *fastforce*

lemma *events-MultiSync*:
‹*events-of* (⟦*S*⟧ *a* ∈# *M*. *P a*) ⊆ (⋃ *a* ∈ *set-mset M*. *events-of* (*P a*))›
by (*induct M rule*: *induct-subset-mset-empty-single*; *simp add*: *events-STOP*)
(*meson Diff-subset-conv dual-order.trans events-Sync*)

lemma *events-MultiInter*:
‹*events-of* (||| *a* ∈# *M*. *P a*) = (⋃ *a* ∈ *set-mset M*. *events-of* (*P a*))›
by (*induct M rule*: *induct-subset-mset-empty-single*)
(*simp-all add*: *events-STOP events-Inter*)

end

Chapter 13

Deadlock Results

theory *DeadlockResults*
  imports *CSPM*
begin

When working with the interleaving $P|||Q$, 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.

13.1 Unfolding Lemmas for the Projections of DF and DF_{SKIP}

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

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

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

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 ($\bigcup$*x*∈*A*. {[]} × {*ref*. *ev x* ∉ *ref*} ∪
               {(*tr*, *ref*). *tr* ≠ [] ∧ *hd tr* = *ev x* ∧ (*tl tr*, *ref*) ∈ $\mathcal{F}$ (*DF A*)}))›
 and *F-DF*$_{SKIP}$:
  ‹$\mathcal{F}$ (DF_{SKIP} *A*) =
   (*if A* = {} *then* {(*s*, *X*). *s* = [] ∨ *s* = [*tick*]}
    else {(*s*, *X*)| *s X*. *s* = [] ∧ *tick* ∉ *X* ∨ *s* = [*tick*]} ∪
        ($\bigcup$*x*∈*A*. {[]} × {*ref*. *ev x* ∉ *ref*} ∪
               {(*tr*, *ref*). *tr* ≠ [] ∧ *hd tr* = *ev x* ∧ (*tl tr*, *ref*) ∈ $\mathcal{F}$ (DF_{SKIP} *A*)}))›
 by (*subst DF-unfold* DF_{SKIP}*-unfold*;
    auto simp add: *F-STOP F-Mprefix F-Mndetprefix write0-def F-SKIP F-Ndet*)+

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

lemma *D-DF*: ‹$\mathcal{D}$ (*DF A*) = (*if* $A = \{\}$ *then* $\{\}$
else $\{s.\ s \neq [] \wedge (\exists x \in A.\ hd\ s = ev\ x \wedge tl\ s \in \mathcal{D}\ (DF\ A))\}$)›
and *D-DF$_{SKIP}$*: ‹$\mathcal{D}$ (DF_{SKIP} *A*) = (*if* $A = \{\}$ *then* $\{\}$
else $\{s.\ s \neq [] \wedge (\exists x \in A.\ hd\ s = ev\ x \wedge tl\ s \in \mathcal{D}\ (DF_{SKIP}\ A))\}$)›
by (*subst DF-unfold DF$_{SKIP}$-unfold*;
auto simp add: *D-Mndetprefix D-Mprefix write0-def D-Ndet D-SKIP*)+

lemma *T-DF*:
‹$\mathcal{T}$ (*DF A*) =
(*if* $A = \{\}$ *then* $\{[]\}$
else $\{s.\ s = [] \vee s \neq [] \wedge (\exists x \in A.\ hd\ s = ev\ x \wedge tl\ s \in \mathcal{T}\ (DF\ A))\}$)›
and *T-DF$_{SKIP}$*:
‹$\mathcal{T}$ (DF_{SKIP} *A*) =
(*if* $A = \{\}$ *then* $\{[], [tick]\}$
else $\{s.\ s = [] \vee s = [tick] \vee$
$s \neq [] \wedge (\exists x \in A.\ hd\ s = ev\ x \wedge tl\ s \in \mathcal{T}\ (DF_{SKIP}\ A))\}$)›
by (*subst DF-unfold DF$_{SKIP}$-unfold*;
auto simp add: *T-STOP T-Mndetprefix write0-def T-Mprefix T-Ndet T-SKIP*)+

13.2 Characterizations for *deadlock-free*, *deadlock-free$_{SKIP}$*

We want more results like ⟦*deadlock-free P*; *deadlock-free Q*⟧ $\Longrightarrow$ *deadlock-free* ($P \sqcap Q$), 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$_{SKIP}$ P* $\equiv$ DF_{SKIP} *UNIV* $\sqsubseteq_F$ *P*›
by (*fact deadlock-free$_{SKIP}$-def*)

lemma *deadlock-free-F*: ‹*deadlock-free P* $\longleftrightarrow$ *DF UNIV* $\sqsubseteq_F$ *P*›
by (*metis deadlock-free-def div-free-divergence-refine*(*5*) *leFD-imp-leF*
leF-imp-leT leF-leD-imp-leFD non-terminating-refine-DF
nonterminating-implies-div-free)

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

$A \neq \{\} \wedge (\forall\, a \in A.\ \textit{deadlock-free}_{SKIP}\ (P\ a))$›
unfolding *deadlock-free-F deadlock-free$_{SKIP}$-def failure-refine-def*
apply (*all* ‹*subst F-DF F-DF$_{SKIP}$*›,
auto simp add: *div-free-DF F-Mprefix D-Mprefix subset-iff*)
by (*metis imageI list.distinct*(*1*) *list.sel*(*1*, *3*))
(*metis* (*no-types*, *lifting*) *event.distinct*(*1*) *image-eqI list.sel*(*1*, *3*) *neq-Nil-conv*)

lemmas *deadlock-free-prefix-iff* =
deadlock-free-Mprefix-iff [*of* ‹$\{a\}$› ‹$\lambda a.\ P$›, *folded write0-def*, *simplified*]
and *deadlock-free$_{SKIP}$-prefix-iff* =
deadlock-free$_{SKIP}$-Mprefix-iff[*of* ‹$\{a\}$› ‹$\lambda a.\ P$›, *folded write0-def*, *simplified*]
for *a P*

lemma *deadlock-free-Mndetprefix-iff*: ‹*deadlock-free* $(\sqcap\ a \in A \rightarrow P\ a) \longleftrightarrow$
$A \neq \{\} \wedge (\forall\, a{\in}A.\ \textit{deadlock-free}\ (P\ a))$›
and *deadlock-free$_{SKIP}$-Mndetprefix-iff*: ‹*deadlock-free$_{SKIP}$* $(\sqcap\ a \in A \rightarrow P\ a)$
$\longleftrightarrow$
$A \neq \{\} \wedge (\forall\, a{\in}A.\ \textit{deadlock-free}_{SKIP}\ (P\ a))$›
apply (*all* ‹*intro iffI conjI*›)
using *non-deadlock-free-STOP*
apply *force*
apply (*meson Mprefix-refines-Mndetprefix-FD*
deadlock-free-Mprefix-iff deadlock-free-def trans-FD)
apply (*metis* (*no-types*, *lifting*) *Mndetprefix-GlobalNdet*
deadlock-free-def deadlock-free-prefix-iff mono-GlobalNdet-FD-const)
using *non-deadlock-free-v2-STOP*
apply *force*
apply (*meson Mprefix-refines-Mndetprefix-FD deadlock-free$_{SKIP}$-FD*
deadlock-free$_{SKIP}$-Mprefix-iff trans-FD)
by (*metis* (*no-types*, *lifting*) *Mndetprefix-GlobalNdet deadlock-free$_{SKIP}$-prefix-iff*
deadlock-free-v2-FD mono-GlobalNdet-FD-const)

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

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

lemma *deadlock-free-MultiNdet-iff*: ‹*deadlock-free* ($\prod$ $a \in A$. P a) $\longleftrightarrow$
$A \neq \{\} \wedge (\forall\ a \in A.$ *deadlock-free* (P a))›
and *deadlock-free*$_{SKIP}$*-MultiNdet-iff*: ‹*deadlock-free*$_{SKIP}$ ($\prod$ $a \in A$. P a) $\longleftrightarrow$
$A \neq \{\} \wedge (\forall\ a \in A.$ *deadlock-free*$_{SKIP}$ (P a))›
if *fin*: ‹*finite A*›
by (*metis deadlock-free-GlobalNdet-iff finite-GlobalNdet-is-MultiNdet that*)
(*metis deadlock-free*$_{SKIP}$*-GlobalNdet-iff finite-GlobalNdet-is-MultiNdet that*)

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

lemma *deadlock-free-Det*:
‹*deadlock-free* $P \Longrightarrow$ *deadlock-free* $Q \Longrightarrow$ *deadlock-free* ($P \Box Q$)›
and *deadlock-free*$_{SKIP}$*-Det*:
‹*deadlock-free*$_{SKIP}$ $P \Longrightarrow$ *deadlock-free*$_{SKIP}$ $Q \Longrightarrow$ *deadlock-free*$_{SKIP}$ ($P \Box Q$)›
by (*meson deadlock-free-Ndet deadlock-free-def mono-Ndet-Det-FD trans-FD*)
(*metis deadlock-free*$_{SKIP}$*-FD deadlock-free*$_{SKIP}$*-Ndet-iff mono-Ndet-Det-FD*
trans-FD)

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

lemma
‹$\exists\, P\ Q$. *deadlock-free* $P \wedge \neg$ *deadlock-free* $Q \wedge$
deadlock-free ($P \Box Q$)›
‹$\exists\, P\ Q$. *deadlock-free*$_{SKIP}$ $P \wedge \neg$ *deadlock-free*$_{SKIP}$ $Q \wedge$
deadlock-free$_{SKIP}$ ($P \Box Q$)›
by (*metis Det-STOP deadlock-free-def idem-FD non-deadlock-free-STOP*)
(*metis Det-STOP deadlock-free*$_{SKIP}$*-FD idem-FD non-deadlock-free*$_{SKIP}$*-STOP*)

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 le-ref-def subset-iff D-Mndetprefix F-Mndetprefix write0-def D-Mprefix F-Mprefix*) *blast*

lemma *Mndetprefix-FD*: ‹$(\exists\, a \in A.\ (a \rightarrow Q) \sqsubseteq_{FD} P) \Longrightarrow \sqcap\, a \in A \rightarrow Q \sqsubseteq_{FD} P$›
by (*meson Mndetprefix-refine-FD failure-divergence-refine-def 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*
apply (*subst Mprefix-Sync-distr*; *simp add*: *assms Mprefix-STOP*)
by (*metis* (*no-types*, *lifting*) *Det-STOP Det-commute Mprefix-STOP assms*(*6*, *7*, *8*) *deadlock-free-Det deadlock-free-Mprefix-iff*)
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*]

13.3 Results on *Renaming*

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

13.3.1 Behaviour with References Processes

For *DF*

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

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

For DF_{SKIP}

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

lemma *Renaming-*DF_{SKIP}*-FD-*DF_{SKIP}:

‹*Renaming* (DF_{SKIP} *A*) *f* $\sqsubseteq_{FD}$ DF_{SKIP} (*f* ‘ *A*)›
if *finitary*: ‹*finitary f*›
proof (*subst* DF_{SKIP}*-def*, *induct rule*: *fix-ind*)
show ‹*adm* (λ*a*. *Renaming a f* $\sqsubseteq_{FD}$ DF_{SKIP} (*f* ‘ *A*))›
by (*simp add*: *finitary monofun-def*)
next
show ‹*Renaming* ⊥ *f* $\sqsubseteq_{FD}$ DF_{SKIP} (*f* ‘ *A*)› **by** (*simp add*: *Renaming-BOT*)
next
show ‹*Renaming* ((Λ *x*. (⊓*a* ∈ *A* → *x*) ⊓ *SKIP*)·*x*) *f* $\sqsubseteq_{FD}$ DF_{SKIP} (*f* ‘ *A*)›
if ‹*Renaming x f* $\sqsubseteq_{FD}$ DF_{SKIP} (*f* ‘ *A*)› **for** *x*
apply *simp*
apply (*simp add*: *Renaming-Ndet Renaming-SKIP*)
apply (*subst Renaming-Mndetprefix*)
apply (*subst* DF_{SKIP}*-unfold*)
apply (*rule mono-Ndet-FD* [*OF* - *idem-FD*])
by (*rule mono-Mndetprefix-FD*[*rule-format*, *OF that*])
qed

For *RUN*

lemma *RUN-FD-Renaming-RUN*: ‹*RUN* (*f* ‘ *A*) $\sqsubseteq_{FD}$ *Renaming* (*RUN A*) *f*›
proof (*subst RUN-def*, *induct rule*: *fix-ind*)
show ‹*adm* (λ*a*. *a* $\sqsubseteq_{FD}$ *Renaming* (*RUN A*) *f*)› **by** (*simp add*: *monofun-def*)
next
show ‹⊥ $\sqsubseteq_{FD}$ *Renaming* (*RUN A*) *f*› **by** *simp*
next
show ‹(Λ *x*. □*a* ∈ *f* ‘ *A* → *x*)·*x* $\sqsubseteq_{FD}$ *Renaming* (*RUN A*) *f*›
if ‹*x* $\sqsubseteq_{FD}$ *Renaming* (*RUN A*) *f*› **for** *x*
apply *simp*
apply (*subst RUN-unfold*)
apply (*subst Renaming-Mprefix*)
by (*rule mono-Mprefix-FD*[*rule-format*, *OF that*])
qed

lemma *Renaming-RUN-FD-RUN*: ‹*Renaming* (*RUN A*) *f* $\sqsubseteq_{FD}$ *RUN* (*f* ‘ *A*)›
if *finitary*: ‹*finitary f*›
proof (*subst RUN-def*, *induct rule*: *fix-ind*)
show ‹*adm* (λ*a*. *Renaming a f* $\sqsubseteq_{FD}$ *RUN* (*f* ‘ *A*))›
by (*simp add*: *finitary monofun-def*)
next
show ‹*Renaming* ⊥ *f* $\sqsubseteq_{FD}$ *RUN* (*f* ‘ *A*)› **by** (*simp add*: *Renaming-BOT*)
next
show ‹*Renaming* ((Λ *x*. □*a* ∈ *A* → *x*)·*x*) *f* $\sqsubseteq_{FD}$ *RUN* (*f* ‘ *A*)›
if ‹*Renaming x f* $\sqsubseteq_{FD}$ *RUN* (*f* ‘ *A*)› **for** *x*
apply *simp*
apply (*subst Renaming-Mprefix*)
apply (*subst RUN-unfold*)
by (*rule mono-Mprefix-FD*[*rule-format*, *OF that*])
qed

For *CHAOS*

lemma *CHAOS-FD-Renaming-CHAOS*:
‹*CHAOS* (*f* ‘ *A*) $\sqsubseteq_{FD}$ *Renaming* (*CHAOS A*) *f*›
proof (*subst CHAOS-def*, *induct rule*: *fix-ind*)
show ‹*adm* (λ*a*. *a* $\sqsubseteq_{FD}$ *Renaming* (*CHAOS A*) *f*)› **by** (*simp add*: *monofun-def*)
next
show ‹⊥ $\sqsubseteq_{FD}$ *Renaming* (*CHAOS A*) *f*› **by** *simp*
next
show ‹(Λ *x*. *STOP* ⊓ (□*a*∈*f* ‘ *A* → *x*))·*x* $\sqsubseteq_{FD}$ *Renaming* (*CHAOS A*) *f*›
if ‹*x* $\sqsubseteq_{FD}$ *Renaming* (*CHAOS A*) *f*› **for** *x*
apply *simp*
apply (*subst CHAOS-unfold*)
apply (*simp add*: *Renaming-Ndet Renaming-STOP*)
apply (*rule mono-Ndet-FD*[*OF idem-FD*])
apply (*subst Renaming-Mprefix*)
by (*rule mono-Mprefix-FD*[*rule-format*, *OF that*])
qed

lemma *Renaming-CHAOS-FD-CHAOS*:
‹*Renaming* (*CHAOS A*) *f* $\sqsubseteq_{FD}$ *CHAOS* (*f* ‘ *A*)›
if *finitary*: ‹*finitary f*›
proof (*subst CHAOS-def*, *induct rule*: *fix-ind*)
show ‹*adm* (λ*a*. *Renaming a f* $\sqsubseteq_{FD}$ *CHAOS* (*f* ‘ *A*))›
by (*simp add*: *finitary monofun-def*)
next
show ‹*Renaming* ⊥ *f* $\sqsubseteq_{FD}$ *CHAOS* (*f* ‘ *A*)› **by** (*simp add*: *Renaming-BOT*)
next
show ‹*Renaming* ((Λ *x*. *STOP* ⊓ (□*xa*∈*A* → *x*))·*x*) *f* $\sqsubseteq_{FD}$ *CHAOS* (*f* ‘ *A*)›
if ‹*Renaming x f* $\sqsubseteq_{FD}$ *CHAOS* (*f* ‘ *A*)› **for** *x*
apply *simp*
apply (*simp add*: *Renaming-Ndet Renaming-STOP*)
apply (*subst CHAOS-unfold*)
apply (*subst Renaming-Mprefix*)
apply (*rule mono-Ndet-FD*[*OF idem-FD*])
by (*rule mono-Mprefix-FD*[*rule-format*, *OF that*])
qed

For $CHAOS_{SKIP}$

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

apply (*subst* $CHAOS_{SKIP}$*-unfold*)
apply (*simp add*: *Renaming-Ndet Renaming-STOP Renaming-SKIP*)
apply (*rule mono-Ndet-FD*[*OF idem-FD*])
apply (*subst Renaming-Mprefix*)
by (*rule mono-Mprefix-FD*[*rule-format*, *OF that*])
qed

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

13.3.2 Corollaries on *deadlock-free* and *deadlock-free*$_{SKIP}$

lemmas *Renaming-DF* =
FD-antisym[*OF Renaming-DF-FD-DF DF-FD-Renaming-DF*]
and *Renaming-*DF_{SKIP} =
FD-antisym[*OF Renaming-*DF_{SKIP}*-FD-*DF_{SKIP} DF_{SKIP}*-FD-Renaming-*DF_{SKIP}]
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-*$CHAOS_{SKIP}$ =
FD-antisym[*OF Renaming-*$CHAOS_{SKIP}$*-FD-*$CHAOS_{SKIP}$
$CHAOS_{SKIP}$*-FD-Renaming-*$CHAOS_{SKIP}$]

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

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

lemma *deadlock-free*$_{SKIP}$*-imp-deadlock-free*$_{SKIP}$*-Renaming*:
‹*deadlock-free*$_{SKIP}$ (*Renaming P f*)› **if** ‹*deadlock-free*$_{SKIP}$ *P*›
unfolding *deadlock-free*$_{SKIP}$*-FD*
apply (*rule trans-FD*[*OF DF*$_{SKIP}$*-subset-FD*[*of* ‹*range f*›]], *simp-all*)
apply (*rule trans-FD*[*OF DF*$_{SKIP}$*-FD-Renaming-DF*$_{SKIP}$])
by (*rule mono-Renaming-FD*[*OF that*[*unfolded deadlock-free*$_{SKIP}$*-FD*]])

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

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

13.4 Big Results

13.4.1 Interesting Equivalence

lemma *deadlock-free-of-Sync-iff-DF-FD-DF-Sync-DF*:
‹($\forall$ *P Q*. *deadlock-free* (*P*::'α *process*) $\longrightarrow$ *deadlock-free Q* $\longrightarrow$
deadlock-free (*P* $[\![S]\!]$ *Q*))
$\longleftrightarrow$ (*DF* (*UNIV*::'α *set*) $\sqsubseteq_{FD}$ (*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 $A|||B$: $(\forall\, P\ Q.\ \textit{deadlock-free}\ P \longrightarrow \textit{deadlock-free}\ Q \longrightarrow \textit{deadlock-free}\ (P|||Q)) = \textit{DF UNIV} \sqsubseteq_{FD} \textit{DF UNIV}|||\textit{DF UNIV}$.

13.4.2 *STOP* and *SKIP* Synchronized with *DF A*

```
lemma DF-FD-DF-Sync-STOP-or-SKIP-iff:
  ‹(DF A ⊑FD DF A ⟦S⟧ P) ⟷ A ∩ S = {}›
  if P-disj: ‹P = STOP ∨ P = SKIP›
proof
  { assume a1: ‹DF A ⊑FD DF A ⟦S⟧ P› and a2: ‹A ∩ S ≠ {}›
    from a2 obtain x where f1: ‹x ∈ A› and f2: ‹x ∈ S› by blast
    have ‹DF A ⟦S⟧ P ⊑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 add: Mndetprefix-unit)
       apply (simp add: write0-def, subst Mprefix-STOP[symmetric],
              subst Mprefix-Sync-distr-right, (simp-all add: f2 Mprefix-STOP)[3])
      by (subst DF-unfold, simp add: Mndetprefix-unit f2 prefix-Sync-SKIP2)
      finally have False
        by (metis DF-Univ-freeness a1 empty-not-insert f1
                  insert-absorb non-deadlock-free-STOP trans-FD)
  }
  thus ‹DF A ⊑FD DF A ⟦S⟧ P ⟹ A ∩ S = {}› by blast
next
  have D-P: ‹𝒟 P = {}› using D-SKIP D-STOP P-disj by blast
  note F-T-P = F-STOP T-STOP F-SKIP D-SKIP
  { assume a1: ‹¬ DF A ⊑FD DF A ⟦S⟧ P› and a2: ‹A ∩ S = {}›
    have False
    proof (cases ‹A = {}›)
      assume ‹A = {}›
      with a1[unfolded DF-def] show False
        by (simp add: fix-const)
          (metis Sync-SKIP-STOP Sync-STOP-STOP Sync-commute idem-FD that)
    next
      assume a3: ‹A ≠ {}›
      have falsify: ‹(a, (X ∪ Y) ∩ insert tick (ev ‘ S) ∪ X ∩ Y) ∉ ℱ (DF A) ⟹
                    (t, X) ∈ ℱ (DF A) ⟹ (u, Y) ∈ ℱ P ⟹
                     a setinterleaves ((t, u), insert tick (ev ‘ S)) ⟹ False› for a t u
X Y
      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(2, 3, 5) F-T Sync.sym TickLeftSync empty-iff
              ftf-Sync21 insertI1 nonTickFree-n-frontTickFree
              non-tickFree-tick process-charn tickFree-def tick-T-F)
    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) Sync.sym SyncTlEmpty
              emptyLeftProperty f1 list.distinct(1) list.sel(1, 3))
   qed
   from a1 show False
     unfolding failure-divergence-refine-def le-ref-def
     by (auto simp add: F-Sync D-Sync D-P div-free-DF subset-iff intro: falsify)
  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› 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 Sync-SKIP-STOP
                              Sync-STOP-STOP Sync-commute)
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
     apply (rule trans-FD[OF mono-Sync-FD
               [OF mono-Mndetprefix-FD-set
                   [of ‹{a}›, simplified, OF that] idem-FD]])
     apply (rule disjE[OF P-disj], simp-all add: Mndetprefix-unit)
      apply (subst Mprefix-Sync-distr-left
               [of ‹{a}› - ‹{}› ‹λa. x›,
                simplified Mprefix-STOP, folded write0-def],
           (insert empty-inter that 3, auto)[3])
     by (subst prefix-Sync-SKIP1, (insert empty-inter that 3, auto)[2])
   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*, *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*, *simplified*]

13.4.3 Finally, *deadlock-free* $(P|||Q)$

```
theorem DF-F-DF-Sync-DF: ‹DF (A ∪ B::'α set) ⊑F DF A ⟦S⟧ DF B›
  if  nonempty: ‹A ≠ {} ∧ B ≠ {}›
 and intersect-hyp: ‹B ∩ S = {} ∨ (∃ y. B ∩ S = {y} ∧ A ∩ S ⊆ {y})›
  unfolding failure-refine-def apply (simp add: F-Sync div-free-DF, safe)
proof −
  fix v t u X Y
  assume ∗ : ‹(t, X) ∈ ℱ (DF A)› ‹(u, Y) ∈ ℱ (DF B)›
        ‹v setinterleaves ((t, u), insert tick (ev ‘ S))›
  define β where ‹β ≡ (t, insert tick (ev ‘ S), u)›
  with ∗ have ‹(fst β, X) ∈ ℱ (DF A)› ‹(snd (snd β), Y) ∈ ℱ (DF B)›
           ‹v ∈ setinterleaving β› by simp-all

  thus ‹(v, (X ∪ Y) ∩ insert tick (ev ‘ S) ∪ X ∩ Y) ∈ ℱ (DF (A ∪ B))›
    apply (subst F-DF, simp add: nonempty)
  proof (induct arbitrary: v rule: setinterleaving.induct)

    case (1 Z)
    hence mt-a: ‹v = []› using emptyLeftProperty by blast
    from intersect-hyp
    consider ‹B ∩ S = {}› | ‹∃ y. B ∩ S = {y} ∧ A ∩ S ⊆ {y}› by blast
    thus ?case
    proof cases
      case 11: 1
      with 1(2) show ?thesis by (subst (asm) F-DF)
                          (auto simp add: nonempty mt-a)
    next
      case 12: 2
      then obtain y where f12: ‹B ∩ S = {y}› and ‹A ∩ S ⊆ {y}› by blast
      from this(2) consider ‹A ∩ S = {}› | ‹A ∩ S = {y}› by blast
      thus ?thesis
      proof cases
        case 121: 1
        with 1(1) show ?thesis by (subst (asm) F-DF)
                            (auto simp add: nonempty mt-a)
      next
        case 122: 2
        with 1(1, 2) f12 nonempty mt-a mk-disjoint-insert show ?thesis
```

```
        by (subst (asm) (1 2) F-DF) (auto, fastforce)
    qed
  qed
next

  case (2 Z y u)
  have * : ‹y ∉ Z› ‹([], X) ∈ ℱ (DF A)› ‹(u, Y) ∈ ℱ (DF B)› ‹v = y # u›
    using 2.prems Cons-F-DF by (auto simp add: emptyLeftProperty)
  have ** : ‹u setinterleaves (([], u), Z)›
    by (metis *(4) 2.prems(3) SyncTlEmpty list.sel(3))
  from 2.prems(2) obtain b where  *** : ‹b ∈ B› ‹y = ev b›
    by (subst (asm) F-DF, simp split: if-split-asm) blast
  show ?case
    apply (rule disjI2, rule-tac x = b in bexI)
    using 2.hyps[simplified, OF *(1, 2, 3) **]
    by (subst F-DF) (auto simp add: *(4) ***)
next

  case (3 x t Z)
  have * : ‹x ∉ Z› ‹(t, X) ∈ ℱ (DF A)› ‹([], Y) ∈ ℱ (DF B)› ‹v = x # t›
    using 3.prems Cons-F-DF by (auto simp add: Sync.sym emptyLeftProperty)
  have ** : ‹t setinterleaves ((t, []), Z)›
    by (metis *(4) 3.prems(3) Sync.sym SyncTlEmpty list.sel(3))
  from 3.prems(1) obtain a where  *** : ‹a ∈ A› ‹x = ev a›
    by (subst (asm) F-DF, simp split: if-split-asm) blast
  show ?case
    apply (rule disjI1, rule-tac x = a in bexI)
    using 3.hyps[simplified, OF *(1, 2, 3) **]
    by (subst F-DF) (auto simp add: *(4) ***)
next

  case (4 x t Z y u)
  consider ‹x ∈ Z› ‹y ∈ Z› | ‹x ∈ Z› ‹y ∉ Z›
      |  ‹x ∉ Z› ‹y ∈ Z› | ‹x ∉ Z› ‹y ∉ Z› by blast
  then show ?case
  proof (cases)
    assume hyps: ‹x ∈ Z› ‹y ∈ Z›
    obtain v' where * : ‹x = y› ‹(t, X) ∈ ℱ (DF A)›
                      ‹(u, Y) ∈ ℱ (DF B)› ‹v = x # v'›
      using 4.prems Cons-F-DF by (simp add: hyps split: if-split-asm) blast
    have ** : ‹v' setinterleaves ((t, u), Z)›
      using *(1, 4) 4.prems(3) hyps(1) by force
    from 4.prems(1) obtain a where  *** : ‹a ∈ A› ‹x = ev a›
      by (subst (asm) F-DF, simp split: if-split-asm) blast
    show ?case
      apply (rule disjI1, rule-tac x = a in bexI)
      using 4.hyps(1)[simplified, OF hyps(2) *(1, 2, 3) **]
      by (subst F-DF) (auto simp add: *(4) ***)
  next
```

```
      assume hyps: ‹x ∈ Z› ‹y ∉ Z›
      obtain v′ where ∗ : ‹(x # t, X) ∈ ℱ (DF A)› ‹(u, Y) ∈ ℱ (DF B)›
                          ‹v = y # v′› ‹v′ setinterleaves ((x # t, u), Z)›
        using 4.prems Cons-F-DF by (simp add: hyps split: if-split-asm) blast
      from 4.prems(2) 4.hyps(2)[simplified, OF hyps ∗(1, 2, 4)]
      show ?case by (subst (asm) F-DF, subst F-DF)
                    (auto simp add: nonempty ∗(3))
    next
      assume hyps: ‹x ∉ Z› ‹y ∈ Z›
      obtain v′ where ∗ : ‹(t, X) ∈ ℱ (DF A)› ‹(y # u, Y) ∈ ℱ (DF B)›
                          ‹v = x # v′› ‹v′ setinterleaves ((t, y # u), Z)›
        using 4.prems Cons-F-DF by (simp add: hyps split: if-split-asm) blast
      from 4.prems(1) 4.hyps(5)[simplified, OF hyps ∗(1, 2, 4)]
      show ?case by (subst (asm) F-DF, subst F-DF)
                    (auto simp add: nonempty ∗(3))
    next
      assume hyps: ‹x ∉ Z› ‹y ∉ Z›
      note f4 = 4[simplified, simplified hyps, simplified]
      from f4(8) obtain v′ v″
        where ‹v = x # v′ ∧ v′ setinterleaves ((t, y # u), Z) ∨
               v = y # v″ ∧ v″ setinterleaves ((x # t, u), Z)›
              (is ‹?left ∨ ?right›) by blast
      then consider ‹?left› | ‹?right› by fast
      then show ?thesis
      proof cases
        assume ‹?left›
        from f4(6) f4(3)[OF f4(6)[THEN Cons-F-DF] f4(7)
             ‹?left›[THEN conjunct2]]
        show ?thesis by (subst (asm) F-DF, subst F-DF)
                        (auto simp add: ‹?left› nonempty)
      next
        assume ‹?right›
        from f4(7) f4(4)[OF f4(6) f4(7)[THEN Cons-F-DF]
             ‹?right›[THEN conjunct2]]
        show ?thesis by (subst (asm) F-DF, subst F-DF)
                        (auto simp add: ‹?right› nonempty)
      qed
    qed
  qed
qed


lemma DF-FD-DF-Sync-DF:
  ‹A ≠ {} ∧ B ≠ {} ⟹ B ∩ S = {} ∨ (∃ y. B ∩ S = {y} ∧ A ∩ S ⊆ {y}) ⟹
   DF (A ∪ B) ⊑FD DF A ⟦S⟧ DF B›
  unfolding failure-divergence-refine-def le-ref-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*:
‹$DF\ (A \cup B) \sqsubseteq_{FD} DF\ A\ [\![S]\!]\ DF\ B \longleftrightarrow$
$(\quad if\ A = \{\}\ then\ B \cap S = \{\}$
$else\ if\ B = \{\}\ then\ A \cap S = \{\}$
$else\ A \cap S = \{\} \lor (\exists a.\ A \cap S = \{a\} \land B \cap S \subseteq \{a\}) \lor$
$B \cap S = \{\} \lor (\exists b.\ B \cap S = \{b\} \land A \cap S \subseteq \{b\}))$›
(**is** ‹*?FD-ref* $\longleftrightarrow$ $(\quad if\ A = \{\}\ then\ B \cap S = \{\}$
$else\ if\ B = \{\}\ then\ A \cap S = \{\}$
else ?cases)›)
proof −
{ 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 ‹$DF\ A\ [\![S]\!]\ DF\ B \sqsubseteq_{FD} (a \to DF\ A)\ [\![S]\!]\ (b \to DF\ B)$›
by (*intro mono-Sync-FD*; *subst DF-unfold*,
subst Mndetprefix-unit[*symmetric*], *simp add*: ‹$a \in A$› ‹$b \in B$›)
also have ‹$\ldots = STOP$› **by** (*simp add*: ‹$a \in S$› ‹$a \neq b$› ‹$b \in S$› *prefix-Sync1*)
finally have *False*
by (*metis DF-Univ-freeness Un-empty* ‹$A \neq \{\}$›
trans-FD[*OF* ‹*?FD-ref*›] *non-deadlock-free-STOP*)
}
thus *?thesis*
apply (*cases* ‹$A = \{\}$›, *simp*,
metis DF-FD-DF-Sync-STOP-iff DF-unfold Sync-commute mt-Mndetprefix)
apply (*cases* ‹$B = \{\}$›, *simp*,
metis DF-FD-DF-Sync-STOP-iff DF-unfold Sync-commute mt-Mndetprefix)
by (*metis Sync-commute Un-commute DF-FD-DF-Sync-DF*)
qed

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*:
‹$DF\ (\bigcup x \in (insert\ a\ A).\ X\ x) \sqsubseteq_{FD} [\![S]\!]\ x \in\#\ mset\text{-}set\ (insert\ a\ A).\ DF\ (X\ x)$›
if *fin*: ‹*finite A*› **and** *nonempty*: ‹$X\ a \neq \{\}$› ‹$\forall b \in A.\ X\ b \neq \{\}$›
and *ugly-hyp*: ‹$\forall b \in A.\ X\ b \cap S = \{\} \lor (\exists y.\ X\ b \cap S = \{y\} \land X\ a \cap S \subseteq \{y\})$›
‹$\forall b \in A.\ \forall c \in A.\ \exists y.\ (X\ b \cup X\ c) \cap S \subseteq \{y\}$›
proof −
have ‹$DF\ (\bigcup\ (X\ `\ insert\ a\ A)) \sqsubseteq_{FD} ([\![S]\!]\ x \in\#\ mset\text{-}set\ (insert\ a\ A).\ DF\ (X\ x)) \land$

```
      (∀ b∈A. X b ∩ S = {} ∨ (∃ y. X b ∩ S = {y} ∧ ⋃ (X ‘ insert a A) ∩ S ⊆
{y}))›
  — We need to add this in our induction
  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)›
        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
  thus ?thesis by (rule conjunct1)
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 (metis DF-unfold MultiSync-rec0 UN-empty idem-FD
              mset-set.empty mt-Mndetprefix)
  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* ⟷ *a* ∉ *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* ⟷ *a* ∈ *S*›
by (*metis DF-FD-DF-Sync-STOP-iff DF-unfold Diff-disjoint Sync-SKIP-STOP Diff-insert-absorb Mndetprefix-unit Sync-STOP-STOP Sync-assoc UNIV-I insert-disjoint*(*2*) *prefix-Sync-SKIP2*)

corollary *DF-FD-DF-Inter-DF*: ‹*DF* (*A*::'α *set*) $\sqsubseteq_{FD}$ *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* ⟹ *deadlock-free Q* ⟹ *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* ≠ {#} ⟹ ∀ *p* ∈# *M*. *deadlock-free* (*P p*) ⟹
deadlock-free (||| *p* ∈# *M*. *P p*)›
proof (*induct rule*: *mset-induct-nonempty*)
case (*m-singleton a*)
thus *?case* **by** *simp*
next
case (*add x F*)
thus *?case* **using** *Inter-deadlock-free* **by** *auto*
qed

end

Chapter 14

Conclusion

In this session, we defined five architectural operators: *MultiDet*, *MultiNdet* and *GlobalNdet*, *MultiSync*, and *MultiSeq* as respective generalizations of $P \square Q$, $P \sqcap Q$, $P [\![S]\!] Q$, and $P \,;\, Q$.

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

- $(\square)$ is commutative, idempotent and admits *STOP* as a neutral element so we defined *MultiDet* on a *finite* $'\alpha$ *set* A by making it equal to *STOP* when $A = \emptyset$.
- $(\sqcap)$ is also commutative and idempotent so we defined *MultiNdet* on a *finite* $'\alpha$ *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

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

  while this holds for $(\square)$ and *MultiDet*).

  As its failures and divergences can easily be generalized to the infinite case, we also defined *GlobalNdet* verifying

  $\textit{finite}\ A \Longrightarrow \sqcap\ p{\in}A.\ \ P\ p = \bigsqcap\ p{\in}A.\ \ P\ p$

- *Sync* is commutative but is not idempotent so we defined *MultiSync* on a $'\alpha$ *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

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

- $(;)$ is neither commutative nor idempotent, so we defined *MultiSeq* on a $'\alpha$ *list* L to keep the multiplicity and the order of the processes. Since *SKIP* is the neutral element for $(;)$, we have

$$SEQ\ p{\in}@[a].\ P\ p = (SEQ\ p{\in}@[].\ P\ p)\ \mathbf{;}\ P\ a$$

$$SEQ\ p{\in}@[a].\ P\ p = P\ a\ \mathbf{;}\ (SEQ\ p{\in}@[].\ P\ p)$$

On our architectural operators we proved continuity (under weakest liberal assumptions), wrote refinements rules and obtained results about the behaviour with other operators inherited from the binary rules.

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 :

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

This theorem allows, for example, to establish:

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

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

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] 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.