

HOL-CSPM - Architectural operators for HOL-CSP

Benoît Ballenghien Safouan Taha Burkhart Wolff

May 26, 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

1	Introduction	9
1.1	Motivations	9
1.2	The Global Architecture of HOL-CSPM	11
2	Some Preliminary Work	13
2.1	Induction Rules for ' α set'	13
2.2	Induction Rules for ' α multiset'	14
2.3	Strong Induction for <i>nat</i>	16
2.4	Preliminaries for Cartesian Product Results	16
3	Patch for Compatibility	17
3.1	Results	17
3.1.1	Continuity Rule	18
3.1.2	Nice Properties	21
3.1.3	Renaming Laws	21
3.2	Assertions	22
3.3	Liflock Freeness	22
3.4	New Laws	22
4	The MultiDet Operator	25
4.1	Definition	25
4.2	First Properties	25
4.3	Some Tests	26
4.4	Continuity	26
4.5	Factorization of (\square) in front of <i>MultiDet</i>	26
4.6	\perp Absorbance	27
4.7	First Properties	27
4.8	Behaviour of <i>MultiDet</i> with (\square)	27
4.9	Commutativity	28
4.10	Behaviour with Injectivity	28
4.11	The Projections	28
4.12	Cartesian Product Results	28

5	The MultiNdet Operator	31
5.1	Definition	31
5.2	First Properties	32
5.3	Some Tests	34
5.4	Continuity	34
5.5	Factorization of (\sqcap) in front of <i>MultiNdet</i>	35
5.6	\perp Absorbance	35
5.7	First Properties	35
5.8	Behaviour of <i>MultiNdet</i> with (\sqcap)	35
5.9	Commutativity	36
5.10	Behaviour with Injectivity	36
5.11	The Projections	36
5.12	Cartesian Product Results	37
6	The MultiSync Operator	39
6.1	Definition	39
6.2	First Properties	41
6.3	Some Tests	41
6.4	Continuity	43
6.5	Factorization of <i>Sync</i> in front of <i>MultiSync</i>	43
6.6	\perp Absorbance	43
6.7	Other Properties	44
6.8	Behaviour of <i>MultiSync</i> with <i>Sync</i>	44
6.9	Commutativity	45
6.10	Behaviour with Injectivity	45
7	The MultiSeq Operator	47
7.1	Definition	47
7.2	First Properties	47
7.3	Some Tests	47
7.4	Continuity	48
7.5	Factorization of $(;)$ in front of <i>MultiSeq</i>	48
7.6	\perp Absorbance	48
7.7	First Properties	48
7.8	Commutativity	49
7.9	Behaviour with Injectivity	49
7.10	Definition of <i>first-elem</i>	49
8	The Global Non-Deterministic Choice	51
8.1	General Non-Deterministic Choice Definition	51
8.2	The Projections	53
8.3	Factorization of (\sqcap) in front of <i>GlobalNdet</i>	53
8.4	\perp Absorbance	53
8.5	First Properties	54

8.6	Behaviour of <i>GlobalNdet</i> with (\sqcap)	54
8.7	Commutativity	54
8.8	Behaviour with Injectivity	54
8.9	Cartesian Product Results	54
8.10	Link with <i>MultiNdet</i>	55
8.11	Link with <i>Mndetprefix</i>	55
8.12	Properties	56
9	CSPM	59
9.1	Refinements Results	59
9.2	Combination of Multi-Operators Laws	64
9.3	Results on <i>Renaming</i>	69
10	Example: Dining Philosophers	73
10.1	Classic Version	73
10.2	Formalization with <code>.-{ }fixrec</code> Package	74
11	Example: Plain Old Telephone System	77
11.1	The Alphabet and Basic Types of POTS	78
11.2	Auxilliaris to Substructure the Specification	79
11.3	A Telephone	80
11.4	A Connector with the Network	81
11.5	Combining NETWORK and TELEPHONES to a SYSTEM	82
11.6	Simple Model of a User	82
11.7	Toplevel Proof-Goals	83
12	Results on <i>events-of</i>	85
12.1	With Operators of HOL-CSP	85
12.2	With Architectural Operators of HOL-CSPM	89
13	Deadlock Results	91
13.1	Unfolding Lemmas for the Projections of <i>DF</i> and <i>DF_{SKIP}</i>	91
13.2	Characterizations for <i>deadlock-free</i> , <i>deadlock-free_{SKIP}</i>	92
13.3	Results on <i>Renaming</i>	95
13.3.1	Behaviour with References Processes	95
13.3.2	Corollaries on <i>deadlock-free</i> and <i>deadlock-free_{SKIP}</i>	99
13.4	Big Results	100
13.4.1	Interesting Equivalence	100
13.4.2	<i>STOP</i> and <i>SKIP</i> Synchronized with <i>DF A</i>	101
13.4.3	Finally, <i>deadlock-free</i> ($P Q$)	103
14	Conclusion	109

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 α 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 $P \sqcap Q$, the synchronization $P \llbracket S \rrbracket Q$ and the sequential composition $P ; Q$.

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 $P \sqcap Q$, $P \sqcap Q$, $P \llbracket S \rrbracket Q$ and $P ; Q$. 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

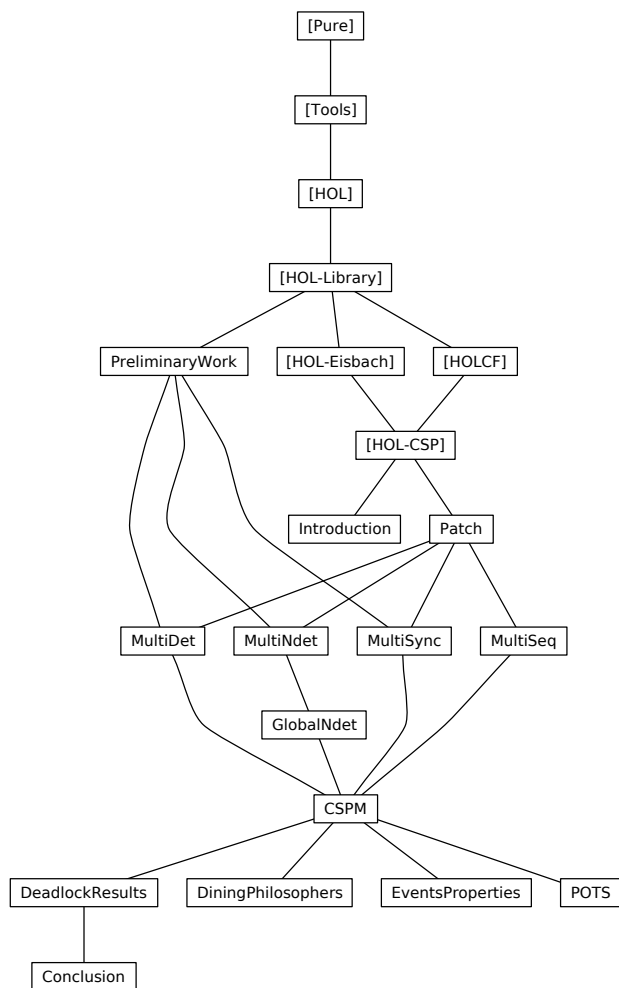


Figure 1.1: The overall architecture

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 α set

```
lemma finite-subset-induct-singleton
  [consumes 3, case-names singleton insertion]:
   $\langle [a \in A; \text{finite } F; F \subseteq A; P \{a\};$ 
   $\bigwedge x F. \text{finite } F \implies x \in A \implies x \notin (\text{insert } a F) \implies P (\text{insert } a F)$ 
   $\implies P (\text{insert } x (\text{insert } a F))] \implies P (\text{insert } a F) \rangle$ 
  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  $\langle A \neq \{\} \rangle$  and  $\langle \text{finite } A \rangle$ 
  and singleton:  $\langle \bigwedge a. a \in A \implies P \{a\} \rangle$ 
  and insert:  $\langle \bigwedge x F. [F \neq \{\}; \text{finite } F; x \in A; x \notin F; P F]$ 
   $\implies P (\text{insert } x F) \rangle$ 
  shows  $\langle P A \rangle$ 
proof -
  obtain  $a A'$  where  $\langle a \in A \rangle \langle \text{finite } A' \rangle \langle A' \subseteq A \rangle \langle A = \text{insert } a A' \rangle$ 
  using  $\langle A \neq \{\} \rangle \langle \text{finite } A \rangle$  by fastforce
  show  $\langle P A \rangle$ 
  apply (subst  $\langle A = \text{insert } a A' \rangle$ , rule finite-subset-induct-singleton[of a A])
  by (simp-all add:  $\langle a \in A \rangle \langle \text{finite } A' \rangle \langle A' \subseteq A \rangle$  singleton insert)
qed
```

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

$\wedge x F. \llbracket \text{finite } F; x \in A; \text{insert } a F \subseteq A; x \notin \text{insert } a F; P (\text{insert } a F) \rrbracket$
 $\implies P (\text{insert } x (\text{insert } a F))$
 $\implies P (\text{insert } a F)$
apply (*erule Finite-Set.finite-subset-induct', simp-all*)
by (*metis insert-absorb2 insert-commute*)

lemma *induct-subset-empty-single*[*consumes 1*]:
 $\langle \llbracket \text{finite } A; P \{\#\}; \forall a \in A. P \{a\};$
 $\wedge F a. \llbracket a \in A; \text{finite } F; F \subseteq A; F \neq \{\#\}; P F \rrbracket \implies P (\text{insert } a F) \rrbracket \implies P A \rangle$
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*]:
 $\langle \llbracket F \subseteq\# A; P \{\#\}; \wedge a F. \llbracket a \in\# A; P F \rrbracket \implies P (\text{add-mset } a F) \rrbracket \implies P F \rangle$
by (*fact multi-subset-induct*)

lemma *msubset-induct-singleton* [*consumes 2, case-names m-singleton add*]:
 $\langle \llbracket a \in\# A; F \subseteq\# A; P \{\#a\#\};$
 $\wedge x F. \llbracket x \in\# A; P (\text{add-mset } a F) \rrbracket \implies P (\text{add-mset } x (\text{add-mset } a F)) \rrbracket$
 $\implies P (\text{add-mset } a F) \rangle$
by (*erule msubset-induct, simp-all add: add-mset-commute*)

lemma *mset-induct-nonempty* [*consumes 1, case-names m-singleton add*]:
assumes $\langle A \neq \{\#\} \rangle$
and *m-singleton*: $\langle \wedge a. a \in\# A \implies P \{\#a\#\} \rangle$
and *add*: $\langle \wedge x F. \llbracket F \neq \{\#\}; x \in\# A; P F \rrbracket \implies P (\text{add-mset } x F) \rangle$
shows $\langle P A \rangle$
proof –
obtain $a A'$ **where** $\langle a \in\# A \rangle \langle A' \subseteq\# A \rangle \langle A = \text{add-mset } a A' \rangle$
by (*metis* $\langle A \neq \{\#\} \rangle$ *diff-subset-eq-self insert-DiffM multiset-nonemptyE*)
show $\langle P A \rangle$
apply (*subst* $\langle A = \text{add-mset } a A' \rangle$, *rule msubset-induct-singleton*[*of a A*])
by (*simp-all add:* $\langle a \in\# A \rangle \langle A' \subseteq\# A \rangle$ *m-singleton add*)
qed

lemma *msubset-induct'* [*consumes 2, case-names empty add*]:
assumes $\langle F \subseteq\# A \rangle$
and *empty*: $\langle P \{\#\} \rangle$
and *insert*: $\langle \wedge a F. \llbracket a \in\# A - F; F \subseteq\# A; P F \rrbracket \implies P (\text{add-mset } a F) \rangle$
shows $\langle P F \rangle$
proof –

```

from  $\langle F \subseteq\# A \rangle$ 
show ?thesis
proof (induct F)
  show  $\langle P \{\#\} \rangle$  by (simp add: assms(2))
next
  case (add x F)
  then show  $\langle P (add\text{-}mset\ x\ F) \rangle$ 
    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*]:

```

 $\langle \llbracket a \in\# A - F; F \subseteq\# A; P \{\#a\# \};$ 
 $\wedge x F. \llbracket x \in\# A - F; F \subseteq\# A; P (add\text{-}mset\ a\ F) \rrbracket$ 
 $\implies P (add\text{-}mset\ x (add\text{-}mset\ a\ F)) \rrbracket$ 
 $\implies P (add\text{-}mset\ a\ F) \rangle$ 
by (erule msubset-induct', simp-all add: add-mset-commute)

```

lemma *msubset-induct-singleton''* [*consumes 1, case-names m-singleton add*]:

```

 $\langle \llbracket add\text{-}mset\ a\ F \subseteq\# A; P \{\#a\# \};$ 
 $\wedge x F. \llbracket add\text{-}mset\ x (add\text{-}mset\ a\ F) \subseteq\# A; P (add\text{-}mset\ a\ F) \rrbracket$ 
 $\implies P (add\text{-}mset\ x (add\text{-}mset\ a\ F)) \rrbracket$ 
 $\implies P (add\text{-}mset\ a\ F) \rangle$ 
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:  $\langle A \neq \{\#\} \rangle$  and m-singleton:  $\langle \wedge a. a \in\# A \implies P \{\#a\# \} \rangle$ 
and hyp:  $\langle \wedge a\ x F. \llbracket a \in\# A; x \in\# A - add\text{-}mset\ a\ F; add\text{-}mset\ a\ F \subseteq\# A;$ 
 $P (add\text{-}mset\ a\ F) \rrbracket \implies P (add\text{-}mset\ x (add\text{-}mset\ a\ F)) \rangle$ 
shows  $\langle P A \rangle$ 
proof–
obtain a A' where  $\langle A = add\text{-}mset\ a\ A' \rangle \langle add\text{-}mset\ a\ A' \subseteq\# A \rangle$ 
using nonempty multiset-cases subset-mset-def by auto
show  $\langle P A \rangle$ 
apply (subst  $\langle A = add\text{-}mset\ a\ A' \rangle$ )
using  $\langle add\text{-}mset\ a\ A' \subseteq\# A \rangle$ 
proof (induct A' rule: msubset-induct-singleton'')
show  $\langle P \{\#a\# \} \rangle$  using  $\langle A = add\text{-}mset\ a\ A' \rangle$  m-singleton by force
next
case (add x F)
show  $\langle P (add\text{-}mset\ x (add\text{-}mset\ a\ F)) \rangle$ 
apply (subst hyp)
apply (simp add:  $\langle A = add\text{-}mset\ a\ A' \rangle$ )

```

apply (*metis* $\langle \text{add-mset } x \ (\text{add-mset } a \ F) \subseteq\# \ A \rangle \text{ add-mset-add-single}$
 $\text{mset-subset-eq-insertD}$ *subset-mset.add-diff-inverse*
 $\text{subset-mset.add-le-cancel-left}$ *subset-mset-def*)
apply (*meson* $\langle \text{add-mset } x \ (\text{add-mset } a \ F) \subseteq\# \ A \rangle \text{ mset-subset-eq-insertD}$
 $\text{subset-mset.dual-order.strict-implies-order}$)
by (*simp-all* $\text{add: } \langle P \ (\text{add-mset } a \ F) \rangle$)
qed
qed

lemma *induct-subset-mset-empty-single*:
 $\langle [P \ \{\#\}; \forall a \in\# \ M. \ P \ \{\#a\# \};$
 $\bigwedge N \ a. \ [a \in\# \ M; \ N \subseteq\# \ M; \ N \neq \{\#\}; \ P \ N] \implies P \ (\text{add-mset } a \ N)] \implies P \ M \rangle$
by (*metis in-diffD mset-induct-nonempty'*)

2.3 Strong Induction for *nat*

lemma *strong-nat-induct*[*consumes 0, case-names 0 Suc*]:
 $\langle [P \ 0; \bigwedge n. (\bigwedge m. m \leq n \implies P \ m) \implies P \ (\text{Suc } n)] \implies P \ n \rangle$
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*]:
 $\langle [0 < n; \ P \ 1; \bigwedge n. 0 < n \implies (\bigwedge m. 0 < m \wedge m \leq n \implies P \ m) \implies P \ (\text{Suc } n)]$
 $\implies P \ n \rangle$
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*:
 $\langle (\lambda(x, y). \ x \ @ \ y) \ ' (A \times B) = \{s \ @ \ t \ | \ s \ t. \ (s, t) \in A \times B \} \rangle$
 $\langle (\lambda(x, y). \ x \ \# \ y) \ ' (A' \times B) = \{s \ \# \ t \ | \ s \ t. \ (s, t) \in A' \times B \} \rangle$
 $\langle (\lambda(x, y). \ [x, y]) \ ' (A' \times B') = \{[s, t] \ | \ s \ t. \ (s, t) \in A' \times B' \} \rangle$
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-chan snoc-eq-iff-butlast)
next
show ⟨s ∈ ?rhs ⇒ s ∈  $\mathcal{D}$  P⟩ for s
using is-processT7 by blast
qed

```

3.1.1 Continuity Rule

Monotonicity of Renaming.

lemma *mono-Renaming*[simp] : ⟨(Renaming P f) \sqsubseteq (Renaming Q f)⟩ **if** ⟨P \sqsubseteq Q⟩

proof (unfold le-approx-def, intro conjI subset-antisym subsetI allI impI)

```

show ⟨s ∈  $\mathcal{D}$  (Renaming Q f) ⇒ s ∈  $\mathcal{D}$  (Renaming P f)⟩ for s
using that[THEN le-approx1] by (auto simp add: D-Renaming)

```

next

```

show ⟨s ∉  $\mathcal{D}$  (Renaming P f) ⇒
      X ∈  $\mathcal{R}_a$  (Renaming P f) s ⇒ X ∈  $\mathcal{R}_a$  (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-chan)

```

next

```

show ⟨s ∉  $\mathcal{D}$  (Renaming P f) ⇒
      X ∈  $\mathcal{R}_a$  (Renaming Q f) s ⇒ X ∈  $\mathcal{R}_a$  (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 ( $\mathcal{D}$  (Renaming P f))⟩
from elem-min-elems[OF *] obtain s1 s2
where ** : ⟨tickFree s1⟩ ⟨front-tickFree s2⟩
      ⟨s = map (EvExt f) s1 @ s2⟩ ⟨s1 ∈  $\mathcal{D}$  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 ( $\mathcal{D}$  P)⟩
apply (cases s; simp add: D-Renaming Nil-min-elems)
by (metis (no-types, lifting) D-imp-front-tickFree append.right-neutral but-
last-snoc
      front-tickFree-chan less-self list.discI
      list.map-disc-iff map-butlast min-elems1 nless-le)
hence ⟨s1 ∈  $\mathcal{T}$  Q⟩ using that[THEN le-approx3] by blast

```

show $\langle s \in \mathcal{T} (\text{Renaming } Q f) \rangle$
apply (*simp add: **(β) $\langle s2 = [] \rangle T\text{-Renaming}$*)
using $\langle s1 \in \mathcal{T} Q \rangle$ **by** *blast*
qed

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

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

lemma *Cont-RenH5*: $\langle \text{finite } (\bigcup t \in \{t. t \leq (s :: 'a \text{ trace})\}. \{s. t = \text{map } (\text{EvExt } f) s\}) \rangle$ **if** $\langle \text{finitary } f \rangle$
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*:
 $\langle \text{finite } \{t. \exists t2. \text{tickFree } t \wedge \text{front-tickFree } t2 \wedge s = \text{map } (\text{EvExt } f) t @ t2\} \rangle$
if $\langle \text{finitary } f \rangle$
proof –
have *f1*: $\langle \{y. \text{map } (\text{EvExt } f) y \leq s \} = (\bigcup z \in \{z. z \leq s\}. \{y. z = \text{map } (\text{EvExt } f) y\}) \rangle$ **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*:
 $\langle (\bigsqcup i. \text{Renaming } (Y i) f) = (\text{Renaming } (\bigsqcup i. Y i) f) \rangle$ **if** *finitary*: $\langle \text{finitary } f \rangle$
and *chain*: $\langle \text{chain } Y \rangle$
proof (*subst Process-eq-spec, safe*)
fix *s*
define *S* **where** $\langle S i \equiv \{s1. \exists t2. \text{tickFree } s1 \wedge \text{front-tickFree } t2 \wedge s = \text{map } (\text{EvExt } f) s1 @ t2 \wedge s1 \in \mathcal{D} (Y i)\} \rangle$ **for** *i*
assume $\langle s \in \mathcal{D} (\bigsqcup i. \text{Renaming } (Y i) f) \rangle$
hence $\langle s \in \mathcal{D} (\text{Renaming } (Y i) f) \rangle$ **for** *i*
by (*simp add: limproc-is-thelub chain chain-Renaming D-LUB*)

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

moreover have $\langle \text{finite } (S\ 0) \rangle$
unfolding $S\text{-def}$
by (*rule finite-subset*[*OF - Cont-RenH7*[*OF finitary, of s*]], *blast*)
moreover have $\langle \forall i. S\ (Suc\ i) \subseteq S\ i \rangle$
unfolding $S\text{-def}$ **using** *NF-ND po-class.chainE*[*OF chain*]
proc-ord2a le-approx-def **by** *blast*
ultimately have $\langle (\bigcap i. S\ i) \neq \{\} \rangle$
by (*rule Inter-nonempty-finite-chained-sets*)

thus $\langle s \in \mathcal{D}\ (\text{Renaming } (Lub\ Y)\ f) \rangle$
by (*simp add: limproc-is-thelub chain D-Renaming*
D-LUB ex-in-conv[symmetric] S-def) *blast*

next
show $\langle s \in \mathcal{D}\ (\text{Renaming } (Lub\ Y)\ f) \implies s \in \mathcal{D}\ (\bigsqcup i. \text{Renaming } (Y\ i)\ f) \rangle$ **for** s
by (*auto simp add: limproc-is-thelub chain chain-Renaming D-Renaming D-LUB*)

next
fix $s\ X$
define S **where** $\langle S\ i \equiv \{s1. (s1, \text{EvExt } f\ -' X) \in \mathcal{F}\ (Y\ i) \wedge s = \text{map } (\text{EvExt } f)\ s1\} \cup$
 $\{s1. \exists t2. \text{tickFree } s1 \wedge \text{front-tickFree } t2 \wedge$
 $s = \text{map } (\text{EvExt } f)\ s1\ @\ t2 \wedge s1 \in \mathcal{D}\ (Y\ i)\} \rangle$ **for** i

assume $\langle (s, X) \in \mathcal{F}\ (\bigsqcup i. \text{Renaming } (Y\ i)\ f) \rangle$
hence $\langle (s, X) \in \mathcal{F}\ (\text{Renaming } (Y\ i)\ f) \rangle$ **for** i
by (*simp add: limproc-is-thelub chain-Renaming*[*OF chain*] *F-LUB*)

hence $\langle \forall i. S\ i \neq \{\} \rangle$ **by** (*auto simp add: S-def F-Renaming*)
moreover have $\langle \text{finite } (S\ 0) \rangle$
unfolding $S\text{-def}$
apply (*subst finite-Un, intro conjI*)
apply (*rule finite-subset*[*of -* $\langle \{s1. s = \text{map } (\text{EvExt } f)\ s1\} \rangle$], *blast*)
apply (*insert Cont-RenH2 Cont-RenH4 finitary, blast*)
by (*rule finite-subset*[*OF - Cont-RenH7*[*OF finitary, of s*]], *blast*)

moreover have $\langle \forall i. S\ (Suc\ i) \subseteq S\ i \rangle$
unfolding $S\text{-def}$ **using** *NF-ND po-class.chainE*[*OF chain*]
proc-ord2a le-approx-def **by** *safe blast+*

ultimately have $\langle (\bigcap i. S\ i) \neq \{\} \rangle$
by (*rule Inter-nonempty-finite-chained-sets*)

thus $\langle (s, X) \in \mathcal{F}\ (\text{Renaming } (Lub\ Y)\ f) \rangle$
by (*simp add: F-Renaming limproc-is-thelub chain D-LUB*
F-LUB ex-in-conv[symmetric] S-def) (*meson NF-ND*)

next
show $\langle (s, X) \in \mathcal{F}\ (\text{Renaming } (Lub\ Y)\ f) \implies (s, X) \in \mathcal{F}\ (\bigsqcup i. \text{Renaming } (Y\ i)\ f) \rangle$ **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*: $\langle \text{Renaming } (\text{Renaming } P f) (\text{inv } f) = P \rangle$ **if** $\langle \text{inj } f \rangle$
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*:
 $\langle \text{Renaming } (\text{Mprefix } A P) f = \square b \in f \text{ ' } A \rightarrow \text{Renaming } (P (\text{THE } a. a \in A \wedge f a = b)) f \rangle$
if *inj-on-f*: $\langle \text{inj-on } f A \rangle$
apply (*subst Renaming-Mprefix[symmetric]*)
apply (*intro arg-cong[where f = $\langle \lambda Q. \text{Renaming } Q f \rangle$ mono-Mprefix-eq]*)
by (*metis that the-inv-into-def the-inv-into-f-f*)

corollary *Renaming-Mprefix-inj*:
 $\langle \text{Renaming } (\text{Mprefix } A P) f = \square b \in f \text{ ' } A \rightarrow \text{Renaming } (P (\text{THE } a. f a = b)) f \rangle$ **if** *inj-f*: $\langle \text{inj } f \rangle$
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*:
 $\langle \text{Renaming } (\text{Mndetprefix } A P) f = \square b \in f \text{ ' } A \rightarrow \text{Renaming } (P (\text{THE } a. a \in A \wedge f a = b)) f \rangle$
if *inj-on-f*: $\langle \text{inj-on } f A \rangle$
apply (*subst Renaming-Mndetprefix[symmetric]*)
apply (*intro arg-cong[where f = $\langle \lambda Q. \text{Renaming } Q f \rangle$ mono-Mndetprefix-eq]*)
by (*metis that the-inv-into-def the-inv-into-f-f*)

corollary *Renaming-Mndetprefix-inj*:
 $\langle \text{Renaming } (\text{Mndetprefix } A P) f = \square b \in f \text{ ' } A \rightarrow \text{Renaming } (P (\text{THE } a. f a = b)) f \rangle$
if *inj-f*: $\langle \text{inj } f \rangle$
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 $deadlock-free_{SKIP} :: 'a\ process \Rightarrow bool$
where $deadlock-free_{SKIP} \equiv deadlock-free-v2$

lemma $deadlock-free-implies-lifelock-free: \langle deadlock-free\ P \Longrightarrow lifelock-free\ P \rangle$
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 $lifelock-free_{SKIP} :: 'a\ process \Rightarrow bool$
where $lifelock-free_{SKIP}\ P \equiv CHAOS_{SKIP}\ UNIV \sqsubseteq_{FD}\ P$

lemma $div-free-is-lifelock-free_{SKIP}: lifelock-free_{SKIP}\ P \longleftrightarrow \mathcal{D}\ P = \{\}$
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 $lifelock-free-is-lifelock-free_{SKIP}: lifelock-free\ P \Longrightarrow lifelock-free_{SKIP}\ P$
by ($simp\ add: leFD-imp-leD\ div-free-divergence-refine(2)\ div-free-is-lifelock-free_{SKIP}$
 $lifelock-free-def$)

corollary $deadlock-free_{SKIP}-is-lifelock-free_{SKIP}: deadlock-free_{SKIP}\ P \Longrightarrow lifelock-free_{SKIP}\ P$
by ($simp\ add: deadlock-free_{SKIP}-implies-div-free\ div-free-is-lifelock-free_{SKIP}$)

3.4 New Laws

lemma $non-terminating-Sync:$
 $\langle non-terminating\ P \Longrightarrow lifelock-free_{SKIP}\ Q \Longrightarrow non-terminating\ (P\ \llbracket A \rrbracket\ Q) \rangle$
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 $non-terminating-Par = non-terminating-Sync[\mathbf{where}\ A = \langle UNIV \rangle]$
and $non-terminating-Inter = non-terminating-Sync[\mathbf{where}\ A = \langle \{\} \rangle]$

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 (⟨(λ□-∈-. / -) 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** $S = \langle \text{insert } a \ A \rangle$])
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]*:
 $\langle \text{finite } A \implies (\Box p \in \text{insert } a \ A. P \ p) = (P \ a \ \Box (p \in A. P \ p)) \rangle$
by (*cases* $\langle a \in A \rangle$, *metis MultiDet-insert Det-assoc Det-id insert-absorb, simp*)

lemma *mono-MultiDet-eq*:
 $\langle \text{finite } A \implies \forall x \in A. P \ x = Q \ x \implies \text{MultiDet } A \ P = \text{MultiDet } A \ Q \rangle$
by (*induct* A *rule: induct-subset-empty-single, simp, simp*)
(metis MultiDet-insert' insertCI)

4.3 Some Tests

lemma *test-MultiDet*: $\langle (\Box p \in \{1::\text{int} \ .. \ 3\}. P \ p) = P \ 1 \ \Box P \ 2 \ \Box P \ 3 \rangle$
proof –
have $\langle \{1::\text{int} \ .. \ 3\} = \text{insert } 1 \ (\text{insert } 2 \ (\text{insert } 3 \ \{\})) \rangle$ **by** *fastforce*
thus $\langle (\Box p \in \{1::\text{int} \ .. \ 3\}. P \ p) = P \ 1 \ \Box P \ 2 \ \Box P \ 3 \rangle$ **by** (*simp add: Det-assoc*)
qed

lemma *test-MultiDet'*:
 $\langle (\Box p \in \{0::\text{nat} \ .. \ a\}. P \ p) = (\Box p \in \{a\} \cup \{1 \ .. \ a\} \cup \{0\}. P \ p) \rangle$
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]*:
 $\langle \llbracket \text{finite } A; \forall x \in A. \text{cont } (P \ x) \rrbracket \implies \text{cont } (\lambda y. \Box z \in A. P \ z \ y) \rangle$
by (*rule Finite-Set.finite-subset-induct[of A A], simp+*)

4.5 Factorization of (\Box) in front of *MultiDet*

lemma *MultiDet-factorization-union*:
 $\langle \llbracket \text{finite } A; \text{finite } B \rrbracket \implies (\Box p \in A. P \ p) \ \Box (\Box p \in B. P \ p) = \Box p \in A \cup B. P \ p \rangle$
apply (*erule finite-induct, simp-all*)
by (*metis Det-commute Det-STOP*)
(metis MultiDet-insert MultiDet-insert' Det-assoc finite-UnI)

4.6 \perp Absorbance

lemma *MultiDet-BOT-absorb*:

assumes *fin*: $\langle \text{finite } A \rangle$ **and** *bot*: $\langle P \ a = \perp \rangle$ **and** *dom*: $\langle a \in A \rangle$

shows $\langle (\Box x \in A. P \ x) = \perp \rangle$

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*:

$\langle \text{finite } A \implies \text{MultiDet } A \ P = \perp \iff (\exists a \in A. P \ a = \perp) \rangle$

by (*induct A rule: finite-induct*) (*auto simp add: STOP-neq-BOT Det-is-BOT-iff*)

4.7 First Properties

lemma *MultiDet-id*: $\langle A \neq \{\} \implies \text{finite } A \implies (\Box p \in A. P) = P \rangle$

by (*erule finite-set-induct-nonempty, simp-all add: Det-id*)

lemma *MultiDet-STOP-id*: $\langle \text{finite } A \implies (\Box p \in A. \text{STOP}) = \text{STOP} \rangle$

by (*cases $\langle A = \{\} \rangle$*) (*simp-all add: MultiDet-id*)

lemma *MultiDet-STOP-neutral*:

$\langle \text{finite } A \implies P \ a = \text{STOP} \implies (\Box z \in \text{insert } a \ A. P \ z) = \Box z \in A. P \ z \rangle$

by (*metis Det-commute MultiDet-insert' Det-STOP*)

lemma *MultiDet-is-STOP-iff*:

$\langle \text{finite } A \implies (\Box a \in A. P \ a) = \text{STOP} \iff A = \{\} \vee (\forall a \in A. P \ a = \text{STOP}) \rangle$

by (*induct rule: finite-induct*) (*auto simp add: Det-is-STOP-iff*)

4.8 Behaviour of *MultiDet* with (\Box)

lemma *MultiDet-Det*:

$\langle \text{finite } A \implies (\Box a \in A. P \ a) \Box (\Box a \in A. Q \ a) = \Box a \in A. P \ a \Box Q \ a \rangle$

proof (*induct A rule: finite-induct*)

case empty show ?case by (*simp add: Det-id*)

next

case (*insert x F*)

hence $\langle \text{MultiDet } (\text{insert } x \ F) \ P \Box \text{MultiDet } (\text{insert } x \ F) \ Q =$

$P \ x \Box \text{MultiDet } F \ P \Box Q \ x \Box \text{MultiDet } F \ Q \rangle$ **by** (*simp add: Det-assoc*)

also have $\langle \dots = (P \ x \Box Q \ x) \Box (\Box a \in F. P \ a \Box Q \ a) \rangle$

by (*metis (no-types, lifting) Det-assoc Det-commute insert.hyps(3)*)

ultimately show $\langle \text{MultiDet } (\text{insert } x \ F) \ P \Box \text{MultiDet } (\text{insert } x \ F) \ Q =$

$(\Box a \in \text{insert } x \ F. P \ a \Box Q \ a) \rangle$

by (*simp add: $\langle \text{finite } F \rangle \langle x \notin F \rangle$*)

qed

4.9 Commutativity

lemma *MultiDet-sets-commute*:

$\langle \llbracket \text{finite } A; \text{ finite } B \rrbracket \implies (\Box a \in A. \Box b \in B. P a b) = \Box b \in B. \Box a \in A. P a b \rangle$
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*:

$\langle \llbracket \text{finite } A; \text{ inj-on } f A \rrbracket \implies (\Box x \in A. P x) = \Box x \in f ' A. P (\text{inv-into } A f x) \rangle$

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: $\langle \text{inv-into } (\text{insert } a F) f (f a) = a \rangle$ **by force**

show ?case

apply (*simp add: 3.hyps(2) 3.hyps(4) f1 del: MultiDet-insert*)

apply (*rule arg-cong[where f = $\langle \lambda x. P a \Box x \rangle$]*)

apply (*subst 3.hyps(5), rule inj-on-subset[OF 3.prem subset-insertI]*)

apply (*rule mono-MultiDet-eq, simp add: 3.hyps(2)*)

using 3.prem **by fastforce**

qed

4.11 The Projections

lemma *D-MultiDet*: $\langle \text{finite } A \implies \mathcal{D} (\Box x \in A. P x) = (\bigcup p \in A. \mathcal{D} (P p)) \rangle$

by (*induct rule: finite-induct*) (*simp-all add: D-Det D-STOP*)

lemma *T-MultiDet*:

$\langle \text{finite } A \implies \mathcal{T} (\Box x \in A. P x) = (\text{if } A = \{\} \text{ then } \{\} \text{ else } \bigcup p \in A. \mathcal{T} (P p)) \rangle$

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- σ s-set- σ s-set*:

$\langle \llbracket \text{finite } A; \text{ finite } B; \forall s \in A. \text{ length } s = \text{len}_1 \rrbracket \implies$

$(\Box (s, t) \in A \times B. P (s @ t)) = \Box u \in \{s @ t \mid s t. (s, t) \in A \times B\}. P u \rangle$

apply (*subst inj-on-mapping-over-MultiDet[where f = $\langle \lambda (s, t). s @ t \rangle$, 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-σ s-set*:

$\langle \llbracket \text{finite } A; \text{ finite } B \rrbracket \implies$
 $(\Box (s, t) \in A \times B. P (s \# t)) = \Box u \in \{s \# t \mid s t. (s, t) \in A \times B\}. P u \rangle$
apply (*subst inj-on-mapping-over-MultiDet*[**where** $f = \langle \lambda (s, t). s \# t \rangle$],
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*:

$\langle \llbracket \text{finite } A; \text{ finite } B \rrbracket \implies$
 $(\Box (s, t) \in A \times B. P [s, t]) = \Box u \in \{[s, t] \mid s t. (s, t) \in A \times B\}. P u \rangle$
apply (*subst inj-on-mapping-over-MultiDet*[**where** $f = \langle \lambda (s, t). [s, t] \rangle$],
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*:

$\langle \text{finite } A \implies \text{finite } B \implies (\Box (s, t) \in A \times B. P s t) = \Box s \in A. \Box t \in B. P s t \rangle$
supply *arg-cong-Det = arg-cong*[**where** $f = \langle \lambda Q. - \Box Q \rangle$]
supply *MultiDet-insert*[*simp del*]
proof (*induct* $\langle \text{card } A \rangle$ *arbitrary: A B rule: nat-less-induct*)
case ($1 A B$)
from $\langle \text{finite } A \rangle$ **consider** $\langle A = \{\} \mid \langle B = \{\} \mid$
 $\langle \exists mA mB a b A' B'. A = \text{insert } a A' \wedge B = \text{insert } b B' \wedge mA = \text{card } A' \wedge$
 $mB = \text{card } B' \wedge mA < \text{card } A \wedge mB < \text{card } B \rangle$
by (*metis card-Diff1-less-iff ex-in-conv insert-Diff*)
thus $\langle (\Box (x, y) \in A \times B. P x y) = \Box s \in A. \text{MultiDet } B (P s) \rangle$
proof cases
show $\langle A = \{\} \implies (\Box (x, y) \in A \times B. P x y) = \Box s \in A. \text{MultiDet } B (P s) \rangle$
by *simp*
next
show $\langle B = \{\} \implies (\Box (x, y) \in A \times B. P x y) = \Box s \in A. \text{MultiDet } B (P s) \rangle$
by (*simp add: MultiDet-STOP-id*[*OF 1.prem(1)*])
next
assume $\langle \exists mA mB a b A' B'. A = \text{insert } a A' \wedge B = \text{insert } b B' \wedge$
 $mA = \text{card } A' \wedge mB = \text{card } B' \wedge mA < \text{card } A \wedge mB < \text{card } B \rangle$
then obtain $\text{insert } a A' B'$
where $*$: $\langle A = \text{insert } a A' \rangle \langle B = \text{insert } b B' \rangle \langle mA = \text{card } A' \rangle$
 $\langle mB = \text{card } B' \rangle \langle mA < \text{card } A \rangle \langle mB < \text{card } B \rangle$ **by** *blast*
have $**$: $\langle \text{Pair } a ' B' = \{a\} \times B' \rangle$ **unfolding** *image-def* **by** *blast*
show $\langle (\Box (x, y) \in A \times B. P x y) = \Box s \in A. \text{MultiDet } B (P s) \rangle$
using $(1, 2) \langle \text{finite } A \rangle \langle \text{finite } B \rangle$ **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' <math>\langle \lambda b. (a, b) \rangle</math>],
  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 (\sqcap) requires more work than with (\sqcup) 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  $\Rightarrow$  'b process]  $\Rightarrow$  '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]  $\Rightarrow$  'b process>
              (<( $\exists \sqcap_l \in \cdot. / \cdot$ )> 76)
```

```
translations  $\sqcap_l p \in l. P \Rightarrow \text{CONST MultiNdet-list } l (\lambda p. P)$ 
```

```
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]:

$\langle a \in A \implies (\prod p \in \text{insert } a \ A. P \ p) = \prod p \in A. P \ p \rangle$
unfolding *MultiNdet-def* **by** (*simp add: insert-absorb*)

lemma *MultiNdet-insert*[simp]:

assumes *fin*: $\langle \text{finite } A \rangle$ **and** *notempty*: $\langle A \neq \{\} \rangle$ **and** *notin*: $\langle a \notin A \rangle$
shows $\langle (\prod p \in \text{insert } a \ A. P \ p) = P \ a \ \sqcap \ (\prod p \in A. P \ p) \rangle$

unfolding *MultiNdet-def*

apply (*rule someI2-ex, simp add: fin finite-list*)**+**

proof –

fix *l l'*

assume $\langle \text{set } l = A \rangle$ **and** $\langle \text{set } l' = \text{insert } a \ A \rangle$

from *notempty* **and** $\langle \text{set } l = A \rangle$ **have** $\langle l \neq [] \rangle$ **by** *fastforce*

then have $\langle \text{MultiNdet-list } (a \ \# \ l) \ P = P \ a \ \sqcap \ \text{MultiNdet-list } l \ P \rangle$

proof (*induct l rule: List.list-nonempty-induct*)

case (*single x*)

show $\langle \text{MultiNdet-list } [a, x] \ P = P \ a \ \sqcap \ \text{MultiNdet-list } [x] \ P \rangle$ **by** *simp*

next

case (*cons x xs*)

have $\langle \text{MultiNdet-list } (a \ \# \ x \ \# \ xs) \ P = P \ a \ \sqcap \ ((\text{MultiNdet-list } xs \ P) \ \sqcap \ P \ x) \rangle$

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 $\langle \text{MultiNdet-list } (a \ \# \ x \ \# \ xs) \ P = P \ a \ \sqcap \ \text{MultiNdet-list } (x \ \# \ xs) \ P \rangle$

proof –

have *f1*: $\langle \text{MultiNdet-list } (a \ \# \ x \ \# \ xs) \ P =$

$\text{Finite-Set.fold } (\lambda a \ p. p \ \sqcap \ P \ a) \ (P \ x \ \sqcap \ P \ a) \ (\text{set } xs) \rangle$

by (*simp add: MultiNdet.fold-set-fold Ndet-commute*)

have $\langle \text{MultiNdet-list } (x \ \# \ xs) \ P =$

$\text{Finite-Set.fold } (\lambda a \ p. p \ \sqcap \ P \ a) \ (P \ x) \ (\text{set } xs) \rangle$

by (*simp add: MultiNdet.fold-set-fold*)

hence $\langle \text{MultiNdet-list } (a \ \# \ x \ \# \ xs) \ P = \text{MultiNdet-list } (x \ \# \ xs) \ P \ \sqcap \ P \ a \rangle$

using *f1* **by** (*simp add: MultiNdet.fold-fun-left-comm*)

thus $\langle \text{MultiNdet-list } (a \ \# \ x \ \# \ xs) \ P = P \ a \ \sqcap \ \text{MultiNdet-list } (x \ \# \ xs) \ P \rangle$

by (*simp add: Ndet-commute*)

qed

qed

moreover have $\langle \text{set } l' = \text{set } (a \ \# \ l) \rangle$

by (*simp add: set l = A set l' = insert a A*)

ultimately show $\langle \text{MultiNdet-list } l' \ P = P \ a \ \sqcap \ \text{MultiNdet-list } l \ P \rangle$

by (*metis MultiNdet-list-set*)

qed

lemma *MultiNdet-insert'*[simp]:

$\langle [\text{finite } A; A \neq \{\}] \implies (\prod p \in \text{insert } a \ A. P \ p) = P \ a \ \sqcap \ (\prod p \in A. P \ p) \rangle$

apply (*cases a ∈ A, subst Set.insert-absorb, simp-all*)

apply (*cases A = {a}, simp add: Ndet-id*)

proof –

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

lemma *mono-MultiNdet-eq*:

$\langle \text{finite } A \implies \forall x \in A. P \ x = Q \ x \implies \text{MultiNdet } A \ P = \text{MultiNdet } A \ Q \rangle$
by (*induct A rule: induct-subset-empty-single; simp*)

5.3 Some Tests

lemma $\langle (\prod_l p \in []. P \ p) = \text{STOP} \rangle$
and $\langle (\prod_l p \in [a]. P \ p) = P \ a \rangle$
and $\langle (\prod_l p \in [a, b]. P \ p) = P \ a \ \sqcap \ P \ b \rangle$
and $\langle (\prod_l p \in [a, b, c]. P \ p) = P \ a \ \sqcap \ P \ b \ \sqcap \ P \ c \rangle$
by *auto*

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

lemma *test-MultiNdet*: $\langle (\prod p \in \{1::\text{int} .. 3\}. P \ p) = P \ 1 \ \sqcap \ P \ 2 \ \sqcap \ P \ 3 \rangle$

proof –

have $\langle \{1::\text{int} .. 3\} = \text{insert } 1 \ (\text{insert } 2 \ (\text{insert } 3 \ \{\})) \rangle$ **by** *fastforce*
thus $\langle (\prod p \in \{1::\text{int} .. 3\}. P \ p) = P \ 1 \ \sqcap \ P \ 2 \ \sqcap \ P \ 3 \rangle$ **by** (*simp add: Ndet-assoc*)
qed

lemma *test-MultiNdet'*:

$\langle (\prod p \in \{0::\text{nat} .. a\}. P \ p) = (\prod p \in \{a\} \cup \{1 .. a\} \cup \{0\}. P \ p) \rangle$
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*]*:

$\langle [\text{finite } A; \forall x \in A. \text{cont } (P \ x)] \implies \text{cont } (\lambda y. \prod z \in A. P \ z \ y) \rangle$
by (*cases A = \{\}, simp, erule finite-set-induct-nonempty; simp*)

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

lemma *MultiNdet-factorization-union*:

$\langle [A \neq \{\}; \text{finite } A; B \neq \{\}; \text{finite } B] \implies$
 $(\sqcap p \in A. P p) \sqcap (\sqcap p \in B. P p) = \sqcap p \in A \cup B. P p \rangle$
by (*erule finite-set-induct-nonempty, simp-all add: Ndet-assoc*)

5.6 \perp Absorbance

lemma *MultiNdet-BOT-absorb*:

assumes *fin*: $\langle \text{finite } A \rangle$ **and** *bot*: $\langle P a = \perp \rangle$ **and** *dom*: $\langle a \in A \rangle$
shows $\langle (\sqcap x \in A. P x) = \perp \rangle$
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*:

$\langle \text{finite } A \implies (\sqcap p \in A. P p) = \perp \iff (\exists a \in A. P a = \perp) \rangle$
apply (*cases $\langle A = \{\} \rangle$, 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*: $\langle A \neq \{\} \implies \text{finite } A \implies (\sqcap p \in A. P) = P \rangle$

by (*erule finite-set-induct-nonempty, (simp-all add: Ndet-id)*)

lemma *MultiNdet-STOP-id*: $\langle \text{finite } A \implies (\sqcap p \in A. STOP) = STOP \rangle$

by (*cases $\langle A = \{\} \rangle$) (simp-all add: MultiNdet-id)*

lemma *MultiNdet-is-STOP-iff*:

$\langle \text{finite } A \implies (\sqcap p \in A. P p) = STOP \iff A = \{\} \vee (\forall a \in A. P a = STOP) \rangle$
apply (*cases $\langle A = \{\} \rangle$, 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*:

$\langle \text{finite } A \implies (\sqcap a \in A. P a) \sqcap (\sqcap a \in A. Q a) = \sqcap a \in A. P a \sqcap Q a \rangle$
apply (*cases $\langle A = \{\} \rangle$, 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*:

$\langle \llbracket \text{finite } A; \text{finite } B \rrbracket \implies$

$(\prod a \in A. \prod b \in B. P a b) = \prod b \in B. \prod a \in A. P a b \rangle$

proof $\langle \text{cases } \langle A = \{\} \rangle \rangle$

show $\langle \text{finite } A \implies \text{finite } B \implies A = \{\} \implies$

$(\prod a \in A. \text{MultiNdet } B (P a)) = \prod b \in B. \prod a \in A. P a b \rangle$

by $\langle \text{simp add: MultiNdet-STOP-id} \rangle$

next

assume $\langle A \neq \{\} \rangle$ **and** $\langle \text{finite } A \rangle$ **and** $\langle \text{finite } B \rangle$

thus $\langle (\prod a \in A. \text{MultiNdet } B (P a)) = \prod b \in B. \prod a \in A. P a b \rangle$

apply $\langle \text{induct } A \text{ rule: finite-set-induct-nonempty} \rangle$

by $\langle \text{simp-all add: MultiNdet-Ndet} \rangle$

qed

5.10 Behaviour with Injectivity

lemma *inj-on-mapping-over-MultiNdet*:

$\langle \llbracket \text{finite } A; \text{inj-on } f A \rrbracket \implies (\prod x \in A. P x) = \prod x \in f ' A. P (\text{inv-into } A f x) \rangle$

proof $\langle \text{induct } A \text{ rule: induct-subset-empty-single} \rangle$

show $\langle \text{MultiNdet } \{\} P = \prod x \in f ' \{\}. P (\text{inv-into } \{\} f x) \rangle$ **by force**

next

case 2

show $\langle ?\text{case} \rangle$ **by force**

next

case $\langle \exists F a \rangle$

hence $f1: \langle \text{inv-into } (\text{insert } a F) f (f a) = a \rangle$ **by force**

show $\langle ?\text{case} \rangle$

apply $\langle \text{simp add: } \mathcal{I}.\text{hyps}(2) \mathcal{I}.\text{hyps}(4) f1 \rangle$

apply $\langle \text{rule arg-cong}[\text{where } f = \langle \lambda x. P a \sqcap x \rangle] \rangle$

apply $\langle \text{subst } \mathcal{I}.\text{hyps}(5), \text{rule inj-on-subset}[OF \mathcal{I}.\text{prems subset-insertI}] \rangle$

apply $\langle \text{rule mono-MultiNdet-eq, simp add: } \mathcal{I}.\text{hyps}(2) \rangle$

using $\mathcal{I}.\text{prems}$ **by fastforce**

qed

5.11 The Projections

lemma *D-MultiNdet*: $\langle \text{finite } A \implies \mathcal{D} (\prod x \in A. P x) = (\bigcup p \in A. \mathcal{D} (P p)) \rangle$

apply $\langle \text{cases } \langle A = \{\} \rangle, \text{simp add: D-STOP, rotate-tac} \rangle$

by $\langle \text{induct rule: finite-set-induct-nonempty} \rangle$ $\langle \text{simp-all add: D-Ndet} \rangle$

lemma *F-MultiNdet*:

$\langle \text{finite } A \implies \mathcal{F} (\prod x \in A. P x) =$

$(\text{if } A = \{\} \text{ then } \{(s, X). s = []\} \text{ else } \bigcup p \in A. \mathcal{F} (P p)) \rangle$

apply $\langle \text{simp add: F-STOP, intro impI, rotate-tac} \rangle$

by $\langle \text{induct rule: finite-set-induct-nonempty} \rangle$ $\langle \text{simp-all add: F-Ndet} \rangle$

lemma *T-MultiNdet*:

$\langle \llbracket \text{finite } A \implies \mathcal{T} (\prod x \in A. P x) =$
 $(\text{if } A = \{\} \text{ then } \{\{\}\} \text{ else } \bigcup p \in A. \mathcal{T} (P p)) \rangle$
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- σ s-set- σ s-set*:

$\langle \llbracket \text{finite } A; \text{finite } B; \forall s \in A. \text{length } s = \text{len}_1 \rrbracket \implies$
 $(\prod (s, t) \in A \times B. P (s @ t)) = \prod u \in \{s @ t \mid s t. (s, t) \in A \times B\}. P u \rangle$
apply (*subst inj-on-mapping-over-MultiNdet[where f = $\langle \lambda (s, t). s @ t \rangle$],*
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- σ s-set*:

$\langle \llbracket \text{finite } A; \text{finite } B \rrbracket \implies$
 $(\prod (s, t) \in A \times B. P (s \# t)) = \prod u \in \{s \# t \mid s t. (s, t) \in A \times B\}. P u \rangle$
apply (*subst inj-on-mapping-over-MultiNdet[where f = $\langle \lambda (s, t). s \# t \rangle$],*
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*:

$\langle \llbracket \text{finite } A; \text{finite } B \rrbracket \implies$
 $(\prod (s, t) \in A \times B. P [s, t]) = \prod u \in \{[s, t] \mid s t. (s, t) \in A \times B\}. P u \rangle$
apply (*subst inj-on-mapping-over-MultiNdet[where f = $\langle \lambda (s, t). [s, t] \rangle$],*
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*:

$\langle \llbracket \text{finite } A; \text{finite } B \rrbracket \implies (\prod (s, t) \in A \times B. P s t) = \prod s \in A. \prod t \in B. P s t \rangle$
supply *arg-cong-Ndet = arg-cong[where f = $\langle \lambda Q. - \prod Q \rangle$]*
proof (*induct $\langle \text{card } A \rangle$ arbitrary: A B rule: nat-less-induct*)
case (*1 A B*)
from $\langle \text{finite } A \rangle \langle \text{finite } B \rangle$ **consider** $\langle A = \{\} \rangle \mid \langle B = \{\} \rangle \mid$
 $\langle \exists mA mB a b A' B'. A = \text{insert } a A' \wedge B = \text{insert } b B' \wedge mA = \text{card } A' \wedge$
 $mB = \text{card } B' \wedge mA < \text{card } A \wedge mB < \text{card } B \rangle$
by (*metis card-Diff1-less-iff ex-in-conv insert-Diff*)
thus $\langle (\prod (x, y) \in A \times B. P x y) = \prod s \in A. \text{MultiNdet } B (P s) \rangle$

```

proof cases
  show  $\langle A = \{\} \implies (\prod (x, y) \in A \times B. P x y) = \prod s \in A. \text{MultiNdet } B (P s) \rangle$ 
    by simp
next
  show  $\langle B = \{\} \implies (\prod (x, y) \in A \times B. P x y) = \prod s \in A. \text{MultiNdet } B (P s) \rangle$ 
    by (simp add: MultiNdet-STOP-id[OF 1.prem1])
next
  assume  $\langle \exists mA mB a b A' B'. A = \text{insert } a A' \wedge B = \text{insert } b B' \wedge$ 
     $mA = \text{card } A' \wedge mB = \text{card } B' \wedge mA < \text{card } A \wedge mB < \text{card } B \rangle$ 
  then obtain  $mA mB a b A' B'$ 
    where  $*$  :  $\langle A = \text{insert } a A' \rangle \langle B = \text{insert } b B' \rangle \langle mA = \text{card } A' \rangle$ 
     $\langle mB = \text{card } B' \rangle \langle mA < \text{card } A \rangle \langle mB < \text{card } B \rangle$  by blast
  have  $**$  :  $\langle \text{Pair } a \text{ ' } B' = \{a\} \times B' \rangle$ 
  and  $***$  :  $\langle (\lambda a. (a, b)) \text{ ' } A' = A' \times \{b\} \rangle$  unfolding image-def by blast+
  show  $\langle (\prod (x, y) \in A \times B. P x y) = \prod s \in A. \text{MultiNdet } B (P s) \rangle$ 
    using  $*(1, 2) \langle \text{finite } A \rangle \langle \text{finite } B \rangle$ 
    apply (cases  $\langle A' = \{\} \rangle$ ; cases  $\langle B' = \{\} \rangle$ ; simp-all)
    apply (rule arg-cong-Ndet)
    apply (subst inj-on-mapping-over-MultiNdet[of  $B' \langle \lambda b. (a, b) \rangle$ ],
      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  $A' \langle \lambda a. (a, b) \rangle$ ],
      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  $B' \langle \lambda b. (a, b) \rangle$ ],
      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 (\sqcap) 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 (λx r. r [[S]] P x) L (P l)⟩
```

```
syntax -MultiSync-list :: ⟨[pttrn, 'b set, 'a list, 'b process] ⇒ 'b process⟩
      (⟨(β[-]l-∈-./ -)⟩ 63)
```

```
translations [[S]]l p ∈ L. P ⇒ CONST MultiSync-list S L (λp. P)
```

```
interpretation MultiSync: comp-fun-commute where f = ⟨λx 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** $*** : \langle L' = a' \# l' \rangle$
by (*metis list.exhaust mset.simps(2) mset-zero-iff*)
note $**** = *[\text{simplified } ***, \text{simplified}]$
have $a0 : \langle a \neq a' \implies \text{MultiSync-list } S L P =$
 $\text{fold } (\lambda x r. r \llbracket S \rrbracket P x) (a' \# (\text{remove1 } a' l)) (P a) \rangle$
apply (*subst fold-multiset-equiv*[**where** $ys = \langle l \rangle$])
apply (*metis MultiSync.comp-fun-commute-axioms comp-fun-commute-def*)
apply (*simp-all add: * ** *****)
by (*metis **** insert-DiffM insert-noteq-member*)
have $a1 : \langle a \neq a' \implies \text{MultiSync-list } S L' P =$
 $\text{fold } (\lambda x r. r \llbracket S \rrbracket P x) (a \# (\text{remove1 } a l')) (P a') \rangle$
apply (*subst fold-multiset-equiv*[**where** $ys = \langle l' \rangle$])
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 $\langle \text{fold } (\lambda x r. r \llbracket S \rrbracket P x) l (P a) = \text{MultiSync-list } S L' P \rangle$
apply (*cases* $\langle a = a' \rangle$, *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** $ys = \langle \text{remove1 } a l' \rangle$],
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* $:: \langle ['b \text{ set}, 'a \text{ multiset}, 'a \Rightarrow 'b \text{ process}] \Rightarrow 'b \text{ process} \rangle$
where $\langle \text{MultiSync } S M P = \text{MultiSync-list } S (\text{SOME } L. \text{mset } L = M) P \rangle$

syntax *-MultiSync* $:: \langle [\text{pttrn}, 'b \text{ set}, 'a \text{ multiset}, 'b \text{ process}] \Rightarrow 'b \text{ process} \rangle$
 $(\langle \mathfrak{A} \llbracket - \rrbracket - \in \# - . / - \rangle 63)$

translations $\llbracket S \rrbracket p \in \# M. P \Leftrightarrow \text{CONST } \text{MultiSync } S M (\lambda p. P)$

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

abbreviation *MultiInter* $:: \langle ['a \text{ multiset}, 'a \Rightarrow 'b \text{ process}] \Rightarrow 'b \text{ process} \rangle$
where $\langle \text{MultiInter } M P \equiv \text{MultiSync } \{\} M P \rangle$

syntax *-MultiInter* $:: \langle [\text{pttrn}, 'a \text{ multiset}, 'b \text{ process}] \Rightarrow 'b \text{ process} \rangle$
 $(\langle \mathfrak{A} \llbracket \llbracket - \rrbracket \rrbracket - \in \# - . / - \rangle 77)$

translations $\llbracket \llbracket - \rrbracket \rrbracket p \in \# M. P \Leftrightarrow \text{CONST } \text{MultiInter } M (\lambda p. P)$

Special case of *MultiSync* $E P$ when $E = \text{UNIV}$.

abbreviation *MultiPar* $:: \langle ['a \text{ multiset}, 'a \Rightarrow 'b \text{ process}] \Rightarrow 'b \text{ process} \rangle$
where $\langle \text{MultiPar } M P \equiv \text{MultiSync } \text{UNIV } M P \rangle$

syntax *-MultiPar* $:: \langle [\text{pttrn}, 'a \text{ multiset}, 'b \text{ process}] \Rightarrow 'b \text{ process} \rangle$

translations $\llbracket p \in\# M. P \equiv \text{CONST MultiPar } M (\lambda p. P) \rrbracket$ (37)

6.2 First Properties

lemma *MultiSync-rec0[simp]*: $\langle \llbracket S \rrbracket p \in\# \{\#\}. P p \rangle = \text{STOP}$
unfolding *MultiSync-def* **by** *simp*

lemma *MultiSync-rec1[simp]*: $\langle \llbracket S \rrbracket p \in\# \{\#a\#\}. P p \rangle = P a$
unfolding *MultiSync-def* **apply**(*rule someI2-ex*) **by** *simp-all*

lemma *MultiSync-add[simp]*:
 $\langle M \neq \{\#\} \implies (\llbracket S \rrbracket p \in\# \text{add-mset } m M. P p) = P m \llbracket S \rrbracket (\llbracket S \rrbracket p \in\# M. P p) \rangle$
unfolding *MultiSync-def*
apply (*rule someI2-ex, simp add: ex-mset*)+

proof –
fix $L L'$
assume $\langle M \neq \{\#\} \rangle \langle \text{mset } L = M \rangle \langle \text{mset } L' = \text{add-mset } m M \rangle$
thus $\langle \text{MultiSync-list } S L' P = P m \llbracket S \rrbracket \text{MultiSync-list } S L P \rangle$
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*:
 $\langle \forall x \in\# M. P x = Q x \implies \text{MultiSync } S M P = \text{MultiSync } S M Q \rangle$
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 $\langle \llbracket S \rrbracket_l p \in []. P p \rangle = \text{STOP}$
and $\langle \llbracket S \rrbracket_l p \in [a]. P p \rangle = P a$
and $\langle \llbracket S \rrbracket_l p \in [a, b]. P p \rangle = P a \llbracket S \rrbracket P b$
and $\langle \llbracket S \rrbracket_l p \in [a, b, c]. P p \rangle = P a \llbracket S \rrbracket P b \llbracket S \rrbracket P c$
by *simp+*

lemma *test-MultiSync*:

$\langle \llbracket S \rrbracket p \in \# \text{ mset } []. P p \rangle = \text{STOP}$
 $\langle \llbracket S \rrbracket p \in \# \text{ mset } [a]. P p \rangle = P a$
 $\langle \llbracket S \rrbracket p \in \# \text{ mset } [a, b]. P p \rangle = P a \llbracket S \rrbracket P b$
 $\langle \llbracket S \rrbracket p \in \# \text{ mset } [a, b, c]. P p \rangle = P a \llbracket S \rrbracket P b \llbracket S \rrbracket P c$
by (*simp-all add: Sync-assoc*)

lemma *MultiSync-set1*: $\langle \text{MultiSync } S (\text{mset-set } \{k::\text{nat}..<k\}) P \rangle = \text{STOP}$

by *fastforce*

lemma *MultiSync-set2*: $\langle \text{MultiSync } S (\text{mset-set } \{k..<\text{Suc } k\}) P \rangle = P k$

by *fastforce*

lemma *MultiSync-set3*:

$\langle l < k \implies \text{MultiSync } S (\text{mset-set } \{l ..< \text{Suc } k\}) P =$
 $P l \llbracket S \rrbracket (\text{MultiSync } S (\text{mset-set } \{\text{Suc } l ..< \text{Suc } k\}) P) \rangle$
by (*simp add: Icc-eq-insert-lb-nat atLeastLessThanSuc-atLeastAtMost*)

lemma *test-MultiSync'*:

$\langle \llbracket S \rrbracket p \in \# \text{ mset-set } \{1::\text{int} .. 3\}. P p \rangle = P 1 \llbracket S \rrbracket P 2 \llbracket S \rrbracket P 3$

proof –

have $\langle \{1::\text{int} .. 3\} = \text{insert } 1 (\text{insert } 2 (\text{insert } 3 \{\})) \rangle$ **by** *fastforce*

thus $\langle \llbracket S \rrbracket p \in \# \text{ mset-set } \{1::\text{int} .. 3\}. P p \rangle = P 1 \llbracket S \rrbracket P 2 \llbracket S \rrbracket P 3$ **by** (*simp add: Sync-assoc*)

qed

lemma *test-MultiSync''*:

$\langle \llbracket S \rrbracket p \in \# \text{ mset-set } \{0::\text{nat} .. a\}. P p \rangle =$
 $\llbracket S \rrbracket p \in \# \text{ mset-set } (\{a\} \cup \{1 .. a\} \cup \{0\}) . P p$
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*)

lemmas *test-MultiInter* = *test-MultiSync*[**where** $S = \langle \{\} \rangle$]
and *test-MultiPar* = *test-MultiSync*[**where** $S = \langle \text{UNIV} \rangle$]
and *MultiInter-set1* = *MultiSync-set1*[**where** $S = \langle \{\} \rangle$]
and *MultiPar-set1* = *MultiSync-set1*[**where** $S = \langle \text{UNIV} \rangle$]
and *MultiInter-set2* = *MultiSync-set2*[**where** $S = \langle \{\} \rangle$]
and *MultiPar-set2* = *MultiSync-set2*[**where** $S = \langle \text{UNIV} \rangle$]
and *MultiInter-set3* = *MultiSync-set3*[**where** $S = \langle \{\} \rangle$]
and *MultiPar-set3* = *MultiSync-set3*[**where** $S = \langle \text{UNIV} \rangle$]

and $test\text{-}MultiInter' = test\text{-}MultiSync'[\mathbf{where} S = \langle \{\} \rangle]$
and $test\text{-}MultiPar' = test\text{-}MultiSync'[\mathbf{where} S = \langle UNIV \rangle]$
and $test\text{-}MultiInter'' = test\text{-}MultiSync''[\mathbf{where} S = \langle \{\} \rangle]$
and $test\text{-}MultiPar'' = test\text{-}MultiSync''[\mathbf{where} S = \langle UNIV \rangle]$

6.4 Continuity

lemma $MultiSync\text{-}cont[simp]$:

$\langle \forall x \in \# M. cont (P x) \implies cont (\lambda y. \llbracket S \rrbracket z \in \# M. P z y) \rangle$
by ($cases \langle M = \{\# \} \rangle$, $simp$, $erule mset\text{-}induct\text{-}nonempty$, $simp+$)

lemmas $MultiInter\text{-}cont[simp] = MultiSync\text{-}cont[\mathbf{where} S = \langle \{\} \rangle]$
and $MultiPar\text{-}cont[simp] = MultiSync\text{-}cont[\mathbf{where} S = \langle UNIV \rangle]$

6.5 Factorization of $Sync$ in front of $MultiSync$

lemma $MultiSync\text{-}factorization\text{-}union$:

$\langle \llbracket M \neq \{\# \}; N \neq \{\# \} \rrbracket \implies$
 $(\llbracket S \rrbracket z \in \# M. P z) \llbracket S \rrbracket (\llbracket S \rrbracket z \in \# N. P z) = \llbracket S \rrbracket z \in \# M + N. P z \rangle$
by ($erule mset\text{-}induct\text{-}nonempty$, $simp\text{-}all$ $add: Sync\text{-}assoc$)

lemmas $MultiInter\text{-}factorization\text{-}union =$
 $MultiSync\text{-}factorization\text{-}union[\mathbf{where} S = \langle \{\} \rangle]$
and $MultiPar\text{-}factorization\text{-}union =$
 $MultiSync\text{-}factorization\text{-}union[\mathbf{where} S = \langle UNIV \rangle]$

6.6 \perp Absorbance

lemma $MultiSync\text{-}BOT\text{-}absorb$:

$\langle m \in \# M \implies P m = \perp \implies (\llbracket S \rrbracket z \in \# M. P z) = \perp \rangle$
by ($metis MultiSync\text{-}add MultiSync\text{-}rec1 mset\text{-}add Sync\text{-}BOT Sync\text{-}commute$)

lemmas $MultiInter\text{-}BOT\text{-}absorb = MultiSync\text{-}BOT\text{-}absorb[\mathbf{where} S = \langle \{\} \rangle]$
and $MultiPar\text{-}BOT\text{-}absorb = MultiSync\text{-}BOT\text{-}absorb[\mathbf{where} S = \langle UNIV \rangle]$

lemma $MultiSync\text{-}is\text{-}BOT\text{-}iff$:

$\langle (\llbracket S \rrbracket m \in \# M. P m) = \perp \longleftrightarrow (\exists m \in \# M. P m = \perp) \rangle$
apply ($cases \langle M = \{\# \} \rangle$, $simp$ $add: BOT\text{-}iff\text{-}D D\text{-}STOP$)
by ($rotate\text{-}tac$, $induct M$ $rule: mset\text{-}induct\text{-}nonempty$)
 $(auto simp add: Sync\text{-}is\text{-}BOT\text{-}iff)$

lemmas $MultiInter\text{-}is\text{-}BOT\text{-}iff = MultiSync\text{-}is\text{-}BOT\text{-}iff[\mathbf{where} S = \langle \{\} \rangle]$
and $MultiPar\text{-}is\text{-}BOT\text{-}iff = MultiSync\text{-}is\text{-}BOT\text{-}iff[\mathbf{where} S = \langle UNIV \rangle]$

6.7 Other Properties

lemma *MultiSync-SKIP-id*: $\langle M \neq \{\#\} \implies (\llbracket S \rrbracket z \in\# M. \text{SKIP}) = \text{SKIP} \rangle$
 by (*rule mset-induct-nonempty, simp-all add: Sync-SKIP-SKIP*)

lemmas *MultiInter-SKIP-id* = *MultiSync-SKIP-id*[**where** $S = \langle \{\} \rangle$]
and *MultiPar-SKIP-id* = *MultiSync-SKIP-id*[**where** $S = \langle \text{UNIV} \rangle$]

lemma *MultiPar-prefix-two-distincts-STOP*:

assumes $\langle m \in\# M \rangle$ **and** $\langle m' \in\# M \rangle$ **and** $\langle \text{fst } m \neq \text{fst } m' \rangle$
shows $\langle (\| a \in\# M. (\text{fst } a \rightarrow P (\text{snd } a))) = \text{STOP} \rangle$

proof –

obtain M' **where** $f2: \langle M = \text{add-mset } m (\text{add-mset } m' M') \rangle$

by (*metis diff-union-swap insert-DiffM assms*)

show $\langle (\| x \in\# M. (\text{fst } x \rightarrow P (\text{snd } x))) = \text{STOP} \rangle$

apply (*cases* $\langle M' = \{\#\} \rangle$,

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'*:

$\langle (\| (m, n) \in\# M; (m', n') \in\# M; m \neq m' \rangle \implies$
 $(\| (m, n) \in\# M. (m \rightarrow P n)) = \text{STOP} \rangle$

apply (*subst cond-case-prod-eta*[**where** $g = \langle \lambda x. (\text{fst } x \rightarrow P (\text{snd } x)) \rangle$])

by (*simp-all add: MultiPar-prefix-two-distincts-STOP*)

6.8 Behaviour of *MultiSync* with *Sync*

lemma *Sync-STOP-STOP*: $\langle \text{STOP } \llbracket S \rrbracket \text{STOP} = \text{STOP} \rangle$

by (*fact Mprefix-Sync-distr-subset[of* $\langle \{\} \rangle$ $S \langle \{\} \rangle$, *simplified,*
simplified Mprefix-STOP])

lemma *MultiSync-Sync*:

$\langle (\llbracket S \rrbracket z \in\# M. P z) \llbracket S \rrbracket (\llbracket S \rrbracket z \in\# M. P' z) = \llbracket S \rrbracket z \in\# M. P z \llbracket S \rrbracket P' z \rangle$

apply (*cases* $\langle M = \{\#\} \rangle$, *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** $S = \langle \{\} \rangle$]

and *MultiPar-Par* = *MultiSync-Sync*[**where** $S = \langle \text{UNIV} \rangle$]

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** $S = \langle\{\}\rangle$]
and *MultiPar-sets-commute* = *MultiSync-sets-commute*[**where** $S = \langle UNIV \rangle$]

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 (∃ N a)
hence f1: ⟨inv-into (insert a (set-mset N)) f (f a) = a⟩ by force
show ?case
apply (simp add: ∃.hyps(2) ∃.hyps(3) f1,
        rule arg-cong[where f = ⟨λx. P a ⟦S⟧ x⟩])
apply (subst ∃.hyps(4), rule inj-on-subset[OF ∃.prems],
        simp add: subset-insertI)
apply (rule mono-MultiSync-eq)
using ∃.prems by fastforce
qed auto

```

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

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 (λa r. (P a) ; r ) S SKIP⟩
```

```
syntax -MultiSeq :: ⟨[pttrn,'a list, 'b process] ⇒ 'b process⟩
        (⟨(3SEQ -∈@-./ -)⟩ 73)
```

```
translations SEQ i ∈@ A. P ⇒ CONST MultiSeq A (λi. 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 $\langle (SEQ\ p \in@ [a,b,c]. P\ p) = P\ a ; P\ b ; P\ c \rangle$
by (*simp-all add: Seq-SKIP Seq-assoc*)

lemma *test-MultiSeq*: $\langle (SEQ\ p \in@ [1::int .. 3]. P\ p) = P\ 1 ; P\ 2 ; P\ 3 \rangle$
by (*simp add: upto.simps Seq-SKIP Seq-assoc*)

7.4 Continuity

lemma *MultiSeq-cont[simp]*:
 $\langle \forall x \in set\ L. cont\ (P\ x) \implies cont\ (\lambda y. SEQ\ z \in@ L. P\ z\ y) \rangle$
by (*induct L force+*)

7.5 Factorization of (;) in front of MultiSeq

lemma *MultiSeq-factorization-append*:
 $\langle (SEQ\ p \in@ A. P\ p) ; (SEQ\ p \in@ B. P\ p) = (SEQ\ p \in@ A @ B . P\ p) \rangle$
by (*induct A rule: list.induct, simp-all add: SKIP-Seq, metis Seq-assoc*)

7.6 \perp Absorbance

lemma *MultiSeq-BOT-absorb*:
 $\langle P\ a = \perp \implies (SEQ\ z \in@ l1 @ [a] @ l2. P\ z) = (SEQ\ z \in@ l1. P\ z) ; \perp \rangle$
by (*metis BOT-Seq MultiSeq-Cons MultiSeq-factorization-append*)

7.7 First Properties

lemma *MultiSeq-SKIP-neutral*:
 $\langle P\ a = SKIP \implies (SEQ\ z \in@ l1 @ [a] @ l2. P\ z) = SEQ\ z \in@ l1 @ l2. P\ z \rangle$
by (*simp add: MultiSeq-def SKIP-Seq*)

lemma *MultiSeq-STOP-absorb*:
 $\langle P\ a = STOP \implies (SEQ\ z \in@ l1 @ [a] @ l2. P\ z) = (SEQ\ z \in@ l1. P\ z) ; STOP \rangle$
by (*metis STOP-Seq MultiSeq-Cons MultiSeq-factorization-append*)

lemma *mono-MultiSeq-eq*:
 $\langle \forall x \in set\ L. P\ x = Q\ x \implies MultiSeq\ L\ P = MultiSeq\ L\ Q \rangle$
by (*induct L fastforce+*)

lemma *MultiSeq-is-SKIP-iff*:
 $\langle MultiSeq\ L\ P = SKIP \iff (\forall a \in set\ L. P\ a = SKIP) \rangle$
by (*induct L, simp-all add: Seq-is-SKIP-iff*)

7.8 Commutativity

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

7.9 Behaviour with Injectivity

lemma *inj-on-mapping-over-MultiSeq*:
 $\langle inj\text{-}on\ f\ (set\ C) \implies$
 $(SEQ\ x \in @\ C.\ P\ x) = SEQ\ x \in @\ map\ f\ C.\ P\ (inv\text{-}into\ (set\ C)\ f\ x) \rangle$
proof (*induct C*)
 case *Nil*
 show *?case by simp*
next
 case (*Cons a C*)
 hence $f1: \langle inv\text{-}into\ (insert\ a\ (set\ C))\ f\ (f\ a) = a \rangle$ **by** *force*
 show *?case*
 apply (*simp add: f1, rule arg-cong[where f = $\langle \lambda x. P\ a ; x \rangle$]*)
 apply (*subst Cons.hyps(1), rule inj-on-subset[OF Cons.prem],*
 simp add: subset-insertI)
 apply (*rule mono-MultiSeq-eq*)
 using *Cons.prem* **by** *fastforce*
qed

7.10 Definition of *first-elem*

primrec *first-elem* :: $\langle [\alpha \Rightarrow bool, \alpha\ list] \Rightarrow nat \rangle$
 where $\langle first\text{-}elem\ P\ [] = 0 \rangle$
 | $\langle first\text{-}elem\ P\ (x \# L) = (if\ P\ x\ then\ 0\ else\ Suc\ (first\text{-}elem\ P\ L)) \rangle$

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 $\langle first\text{-}elem\ (\lambda x. 4 < x)\ [0::nat, 2, 5] \rangle$
lemma $\langle first\text{-}elem\ (\lambda x. 5 < x)\ [0::nat, 2, 5] = 3 \rangle$ **by** *simp*
lemma $\langle P\ \text{' set } L \subseteq \{False\} \implies first\text{-}elem\ P\ L = length\ L \rangle$ **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  $\langle \# P. \forall Q. (P :: \text{'}\alpha \text{ process}) \sqcap Q = Q \rangle$ 
```

```
proof -
```

```
  { fix P ::  $\langle \text{'}\alpha \text{ process} \rangle$ 
```

```
    assume * :  $\langle \forall Q. P \sqcap Q = Q \rangle$ 
```

```
    hence  $\langle P = \text{STOP} \rangle$ 
```

```
      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 ::  $\langle [\text{'}\alpha \text{ set}, \text{'}\alpha \Rightarrow \text{'}\beta \text{ process}] \Rightarrow \text{'}\beta \text{ process} \rangle$ 
```

```
is  $\langle \lambda A P. \text{ if } A = \{\}$ 
```

```
  then  $\{(s, X). s = []\}, \{\}$ 
```

```
  else  $(\bigcup_{a \in A}. \mathcal{F} (P a), \bigcup_{a \in A}. \mathcal{D} (P a)) \rangle$ 
```

```

proof –
  show  $\langle ?thesis\ A\ P \rangle$ 
  (is  $\langle is\text{-}process\ (if\ A = \{\}\ then\ (\{(s, X). s = \{\}\}, \{\})\ else\ (?f, ?d)) \rangle$  for  $A\ P$ )
  proof (split if-split, intro conjI impI)
    show  $\langle is\text{-}process\ (\{(s, X). s = \{\}\}, \{\}) \rangle$ 
    by (metis STOP.rsp eq-onp-def)
  next
  show  $\langle is\text{-}process\ (\bigcup_{a \in A} \mathcal{F}(P\ a), \bigcup_{a \in A} \mathcal{D}(P\ a)) \rangle$  if nonempty:  $\langle A \neq \{\} \rangle$ 
  unfolding is-process-def FAILURES-def DIVERGENCES-def fst-conv snd-conv
  proof (intro conjI allI impI)
    show  $\langle \{\}, \{\} \rangle \in ?f$  using is-processT1 nonempty by blast
  next
  show  $\langle (s, X) \in ?f \implies front\text{-}tickFree\ s \rangle$  for  $s\ X$ 
  using is-processT2 by blast
  next
  show  $\langle (s @ t, \{\}) \in ?f \implies (s, \{\}) \in ?f \rangle$  for  $s\ t$ 
  using is-processT3 by blast
  next
  show  $\langle (s, Y) \in ?f \wedge X \subseteq Y \implies (s, X) \in ?f \rangle$  for  $s\ X\ Y$ 
  using is-processT4 by blast
  next
  show  $\langle (s, X) \in ?f \wedge (\forall c. c \in Y \longrightarrow (s @ [c], \{\}) \notin ?f) \implies (s, X \cup Y) \in ?f \rangle$  for  $s\ X\ Y$ 
  using is-processT5 by simp blast
  next
  show  $\langle (s @ [tick], \{\}) \in ?f \implies (s, X - \{tick\}) \in ?f \rangle$  for  $s\ X$ 
  using is-processT6 by blast
  next
  show  $\langle s \in ?d \wedge tickFree\ s \wedge front\text{-}tickFree\ t \implies s @ t \in ?d \rangle$  for  $s\ t$ 
  using is-processT7 by blast
  next
  show  $\langle s \in ?d \implies (s, X) \in ?f \rangle$  for  $s\ X$ 
  using is-processT8 by blast
  next
  show  $\langle s @ [tick] \in ?d \implies s \in ?d \rangle$  for  $s$ 
  using is-processT9 by blast
  qed
qed
qed

```

syntax $-GlobalNdet :: \langle [pttrn, 'a\ set, 'b\ process] \Rightarrow 'b\ process \rangle (\langle (\exists \square - \in -. / -) \rangle 76)$
translations $\square p \in A. P \equiv CONST\ GlobalNdet\ A\ (\lambda p. P)$

Note that the global non-deterministic choice $\square p \in A. P\ p$ is different from the multi-non-deterministic prefix $\square p \in A \rightarrow P\ p$ 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 $\sqcap p \in A. P p$ which is only defined when A is *finite*.

lemma *empty-GlobalNdet[simp]* : $\langle \text{GlobalNdet } \{\} P = \text{STOP} \rangle$
by (*simp add: GlobalNdet.abs-eq STOP-def*)

8.2 The Projections

lemma *F-GlobalNdet*:

$\langle \mathcal{F} (\sqcap x \in A. P x) = (\text{if } A = \{\} \text{ then } \{(s, X). s = []\} \text{ else } (\bigcup x \in A. \mathcal{F} (P x))) \rangle$
by (*simp add: Failures-def FAILURES-def GlobalNdet.rep-eq*)

lemma *D-GlobalNdet*:

$\langle \mathcal{D} (\sqcap x \in A. P x) = (\text{if } A = \{\} \text{ then } \{\} \text{ else } (\bigcup x \in A. \mathcal{D} (P x))) \rangle$
by (*simp add: Divergences-def DIVERGENCES-def GlobalNdet.rep-eq*)

lemma *T-GlobalNdet*:

$\langle \mathcal{T} (\sqcap x \in A. P x) = (\text{if } A = \{\} \text{ then } \{[]\} \text{ else } (\bigcup x \in A. \mathcal{T} (P x))) \rangle$
by (*auto simp: Traces.rep-eq TRACES-def Failures.rep-eq[symmetric]*)
F-GlobalNdet intro: F-T T-F)

lemma *mono-GlobalNdet-eq*:

$\langle \forall x \in A. P x = Q x \implies \text{GlobalNdet } A P = \text{GlobalNdet } A Q \rangle$
by (*subst Process-eq-spec, simp add: F-GlobalNdet D-GlobalNdet*)

lemma *mono-GlobalNdet-eq2*:

$\langle \forall x \in A. P (f x) = Q x \implies \text{GlobalNdet } (f ' A) P = \text{GlobalNdet } A Q \rangle$
by (*subst Process-eq-spec, simp add: F-GlobalNdet D-GlobalNdet*)

8.3 Factorization of (\sqcap) in front of *GlobalNdet*

lemma *GlobalNdet-factorization-union*:

$\langle [A \neq \{\}; B \neq \{\}] \implies$
 $(\sqcap p \in A. P p) \sqcap (\sqcap p \in B. P p) = (\sqcap p \in A \cup B. P p) \rangle$
by (*subst Process-eq-spec*) (*simp add: F-GlobalNdet D-GlobalNdet F-Ndet D-Ndet*)

8.4 \perp Absorbance

lemma *GlobalNdet-BOT-absorb*: $\langle P a = \perp \implies a \in A \implies (\sqcap x \in A. P x) = \perp \rangle$
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*: $\langle (\sqcap x \in A. P x) = \perp \longleftrightarrow (\exists a \in A. P a = \perp) \rangle$

by (*simp add: BOT-iff-D D-GlobalNdet*)

8.5 First Properties

lemma *GlobalNdet-id*: $\langle A \neq \{\} \implies (\sqcap p \in A. P) = P \rangle$
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*:
 $\langle A - \{a\} \neq \{\} \implies (\sqcap x \in \text{insert } a \text{ } A. P x) = P a \sqcap (\sqcap x \in A - \{a\}. P x) \rangle$
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*:
 $\langle \text{inj-on } f \text{ } A \implies (\sqcap x \in A. P x) = \sqcap x \in f \text{ ' } A. P (\text{inv-into } A \text{ } f \text{ } x) \rangle$
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*:
 $\langle (\sqcap (s, t) \in A \times B. P (s @ t)) = \sqcap u \in \{s @ t \mid s \text{ } t. (s, t) \in A \times B\}. P u \rangle$
apply (*subst Process-eq-spec, simp add: F-GlobalNdet D-GlobalNdet*)
by *safe auto*

lemma *GlobalNdet-cartprod-s-set-σs-set*:

$\langle (\prod (s, t) \in A \times B. P (s \# t)) = \prod u \in \{s \# t \mid s t. (s, t) \in A \times B\}. P u \rangle$

apply (*subst Process-eq-spec, simp add: F-GlobalNdet D-GlobalNdet*)

by *safe auto*

lemma *GlobalNdet-cartprod-s-set-s-set*:

$\langle (\prod (s, t) \in A \times B. P [s, t]) = \prod u \in \{[s, t] \mid s t. (s, t) \in A \times B\}. P u \rangle$

apply (*subst Process-eq-spec, simp add: F-GlobalNdet D-GlobalNdet*)

by *safe auto*

lemma *GlobalNdet-cartprod*: $\langle (\prod (s, t) \in A \times B. P s t) = \prod s \in A. \prod t \in B. P s t \rangle$

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 $\prod a \in A. P a = \prod_{a \in A} P a$.

lemma *finite-GlobalNdet-is-MultiNdet*:

$\langle \text{finite } A \implies (\prod p \in A. P p) = \prod_{p \in A} P p \rangle$

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]*:

$\langle \llbracket \text{finite } A; \forall x. \text{cont } (f x) \rrbracket \implies \text{cont } (\lambda y. (\prod z \in A. (f z y))) \rangle$

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*: $\langle \prod x \in A \rightarrow P x = \prod x \in A. (x \rightarrow P x) \rangle$

apply (*cases $\langle A = \{\} \rangle$, simp*)

by (*subst Process-eq-spec-optimized*)

(*simp-all add: F-Mndetprefix D-Mndetprefix F-GlobalNdet D-GlobalNdet*)

lemma *write0-GlobalNdet*:

$\langle A \neq \{\} \implies (\prod x \in A. (a \rightarrow P x)) = (a \rightarrow (\prod x \in A. P x)) \rangle$

by (*auto simp add: Process-eq-spec write0-def*)

F-GlobalNdet D-GlobalNdet F-Mprefix D-Mprefix)

8.12 Properties

lemma *GlobalNdet-Det*:

$\langle A \neq \{\} \implies (\prod a \in A. P a) \sqcap Q = \prod a \in A. P a \sqcap Q \rangle$
 by (*auto simp add: Process-eq-spec F-GlobalNdet D-GlobalNdet*
F-Det D-Det Un-def T-GlobalNdet)

lemma *Mndetprefix-STOP*: $\langle A \subseteq C \implies (\prod a \in A \rightarrow P a) \llbracket C \rrbracket STOP = STOP \rangle$
proof (*subst STOP-iff-T, intro subset-antisym subsetI*)

show $\langle s \in \{\} \implies s \in \mathcal{T} (Mndetprefix A P \llbracket C \rrbracket STOP) \rangle$ **for** s
 by (*simp add: Nil-elem-T*)

next

show $\langle A \subseteq C \implies s \in \mathcal{T} (Mndetprefix A P \llbracket C \rrbracket STOP) \implies s \in \{\} \rangle$ **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*:

$\langle A \neq \{\} \implies (\prod x \in A. P x) \llbracket C \rrbracket Q = \prod x \in A. (P x \llbracket C \rrbracket Q) \rangle$
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*:

$\langle [A \subseteq B; B \subseteq C] \implies (\prod a \in A \rightarrow P a) \llbracket C \rrbracket (\prod b \in B \rightarrow Q b) =$
 $\prod a \in A \rightarrow (P a \llbracket C \rrbracket Q a) \rangle$

— does not hold in general when $A \subseteq C$

apply (*cases* $\langle A = \{\} \rangle$, *simp*,
metis (no-types, lifting) Mprefix-STOP Mprefix-Sync-distr-subset
empty-subsetI inf-bot-left)

apply (*cases* $\langle B = \{\} \rangle$, *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*:

$\langle A \subseteq B \implies ((\prod a \in A \rightarrow P a) \parallel (\prod b \in B \rightarrow Q b)) = \prod a \in A \rightarrow P a \parallel Q a \rangle$
by (*simp add: Mndetprefix-Mprefix-Sync-distr*)

lemma *Mndetprefix-Sync-Det-distr*:

$\langle (\prod a \in A \rightarrow (P a \llbracket C \rrbracket (\prod b \in B \rightarrow Q b))) \sqcap \rangle$

$(\sqcap b \in B \rightarrow ((\sqcap a \in A \rightarrow P a) \llbracket C \rrbracket Q b))$
 $\sqsubseteq_{FD} (\sqcap a \in A \rightarrow P a) \llbracket C \rrbracket (\sqcap b \in B \rightarrow Q b)$
if *set-hyps* : $\langle A \neq \{\} \rangle \langle B \neq \{\} \rangle \langle A \cap C = \{\} \rangle \langle B \cap C = \{\} \rangle$
 — both surprising: equality does not hold + deterministic choice
proof –
have *mono-GlobalNdet-FD*:
 $\langle \bigwedge P Q A. \forall x \in A. P x \sqsubseteq_{FD} Q x \implies GlobalNdet A P \sqsubseteq_{FD} GlobalNdet A Q \rangle$
by (*auto simp: failure-divergence-refine-def le-ref-def F-GlobalNdet D-GlobalNdet*)

have * : $\langle a \in A \implies b \in B \implies$
 $(\sqcap b \in B. (a \rightarrow (P a \llbracket C \rrbracket (b \rightarrow Q b)))) \sqcap$
 $(\sqcap a \in A. (b \rightarrow ((a \rightarrow P a) \llbracket C \rrbracket Q b))) \sqsubseteq_{FD}$
 $(a \rightarrow P a) \llbracket C \rrbracket (b \rightarrow Q b) \rangle$ **for** $a b$
apply (*subst Mprefix-Sync-distr-indep[of $\langle \{a\} \rangle C \langle \{b\} \rangle$, 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 $\langle (\sqcap a \in A. \sqcap b \in B. (a \rightarrow (P a \llbracket C \rrbracket (b \rightarrow Q b)))) \sqcap$
 $(\sqcap b \in B. \sqcap a \in A. (b \rightarrow ((a \rightarrow P a) \llbracket C \rrbracket Q b))) \sqsubseteq_{FD}$
 $\sqcap b \in B. \sqcap a \in A. ((a \rightarrow P a) \llbracket C \rrbracket (b \rightarrow Q b)) \rangle$
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*:
 $\langle A \neq \{\} \implies (\sqcap a \in A. \sqcap b \in B \rightarrow P a b) = \sqcap b \in B \rightarrow (\sqcap a \in A. P a b) \rangle$
by (*auto simp add: Process-eq-spec F-GlobalNdet D-GlobalNdet F-Mprefix D-Mprefix*)

lemma *GlobalNdet-Det-distrib*:
 $\langle (\sqcap a \in A. P a \sqcap Q a) = (\sqcap a \in A. P a) \sqcap (\sqcap a \in A. Q a) \rangle$

```

if  $\langle \exists Q' b. \forall a. Q a = (b \rightarrow Q' a) \rangle$ 
proof –
from that obtain b Q' where  $\langle \forall a. (Q a = (b \rightarrow Q' a)) \rangle$  by blast
thus ?thesis
apply (cases  $\langle A = \{\} \rangle$ , 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*:

$\langle \text{finite } A \implies \forall x \in A. P \sqsubseteq_F Q \implies \text{MultiDet } A \ P \sqsubseteq_F \text{MultiDet } A \ Q \rangle$
apply (*induct A rule: induct-subset-empty-single; simp*)
oops

lemma *mono-MultiDet-D[simp, elim]*:

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

lemma *mono-MultiNdet-F[simp, elim]*:

$\langle \text{finite } A \implies \forall x \in A. P \sqsubseteq_F Q \implies \text{MultiNdet } A \ P \sqsubseteq_F \text{MultiNdet } A \ Q \rangle$
and *mono-MultiNdet-D[simp, elim]*:
 $\langle \text{finite } A \implies \forall x \in A. P \sqsubseteq_D Q \implies \text{MultiNdet } A \ P \sqsubseteq_D \text{MultiNdet } A \ Q \rangle$
and *mono-MultiNdet-T[simp, elim]*:
 $\langle \text{finite } A \implies \forall x \in A. P \sqsubseteq_T Q \implies \text{MultiNdet } A \ P \sqsubseteq_T \text{MultiNdet } A \ Q \rangle$
and *mono-MultiNdet-DT[simp, elim]*:

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

lemma *mono-MultiNdet-MultiDet-F*[simp, elim]:
 $\langle \text{finite } A \implies \text{MultiNdet } A P \sqsubseteq_F \text{MultiDet } A P \rangle$
and *mono-MultiNdet-MultiDet-D*[simp, elim]:
 $\langle \text{finite } A \implies \text{MultiNdet } A P \sqsubseteq_D \text{MultiDet } A P \rangle$
and *mono-MultiNdet-MultiDet-T*[simp, elim]:
 $\langle \text{finite } A \implies \text{MultiNdet } A P \sqsubseteq_T \text{MultiDet } A P \rangle$
and *mono-MultiNdet-MultiDet-FD*[simp, elim]:
 $\langle \text{finite } A \implies \text{MultiNdet } A P \sqsubseteq_{FD} \text{MultiDet } A P \rangle$
and *mono-MultiNdet-MultiDet-DT*[simp, elim]:
 $\langle \text{finite } A \implies \text{MultiNdet } A P \sqsubseteq_{DT} \text{MultiDet } A P \rangle$
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*: $\langle \forall x \in \# M. P x \sqsubseteq_F Q x \implies \text{MultiSync } S M P \sqsubseteq_F \text{MultiSync } S M Q \rangle$
apply (*induct M* rule: *induct-subset-mset-empty-single*; *simp*)
oops

lemma *mono-MultiSync-D*: $\langle \forall x \in \# M. P x \sqsubseteq_D Q x \implies \text{MultiSync } S M P \sqsubseteq_D \text{MultiSync } S M Q \rangle$
apply (*induct M* rule: *induct-subset-mset-empty-single*; *simp*)
oops

lemma *mono-MultiSync-T*: $\langle \forall x \in \# M. P x \sqsubseteq_T Q x \implies \text{MultiSync } S M P \sqsubseteq_T \text{MultiSync } S M Q \rangle$
apply (*induct M rule: induct-subset-mset-empty-single; simp*)
oops

lemma *mono-MultiSync-DT*[*simp, elim*]:
 $\langle \forall x \in \# M. P x \sqsubseteq_{DT} Q x \implies \text{MultiSync } S M P \sqsubseteq_{DT} \text{MultiSync } S M Q \rangle$
and *mono-MultiSync-FD*[*simp, elim*]:
 $\langle \forall x \in \# M. P x \sqsubseteq_{FD} Q x \implies \text{MultiSync } S M P \sqsubseteq_{FD} \text{MultiSync } S M Q \rangle$
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** $S = \langle \{\} \rangle$]
and *mono-MultiInter-FD*[*simp, elim*] = *mono-MultiSync-FD*[**where** $S = \langle \{\} \rangle$]
and *mono-MultiPar-DT*[*simp, elim*] = *mono-MultiSync-DT*[**where** $S = \langle UNIV \rangle$]
and *mono-MultiPar-FD*[*simp, elim*] = *mono-MultiSync-FD*[**where** $S = \langle UNIV \rangle$]

lemma *mono-MultiSeq-F*:
 $\langle \forall x \in \text{set } L. P x \sqsubseteq_F Q x \implies \text{MultiSeq } L P \sqsubseteq_F \text{MultiSeq } L Q \rangle$
apply (*induct L, fastforce*) **apply** *simp* **oops**

lemma *mono-MultiSeq-D*:
 $\langle \forall x \in \text{set } L. P x \sqsubseteq_D Q x \implies \text{MultiSeq } L P \sqsubseteq_D \text{MultiSeq } L Q \rangle$
apply (*induct L, fastforce*) **apply** *simp* **oops**

lemma *mono-MultiSeq-T*:
 $\langle \forall x \in \text{set } L. P x \sqsubseteq_T Q x \implies \text{MultiSeq } L P \sqsubseteq_T \text{MultiSeq } L Q \rangle$
apply (*induct L, fastforce*) **apply** *simp* **oops**

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

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

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

lemma *mono-GlobalNdet-F[simp, elim]*:
 $\langle \forall x \in A. P \ x \sqsubseteq_F Q \ x \implies \text{GlobalNdet } A \ P \sqsubseteq_F \text{GlobalNdet } A \ Q \rangle$
and *mono-GlobalNdet-D[simp, elim]*:
 $\langle \forall x \in A. P \ x \sqsubseteq_D Q \ x \implies \text{GlobalNdet } A \ P \sqsubseteq_D \text{GlobalNdet } A \ Q \rangle$
and *mono-GlobalNdet-T[simp, elim]*:
 $\langle \forall x \in A. P \ x \sqsubseteq_T Q \ x \implies \text{GlobalNdet } A \ P \sqsubseteq_T \text{GlobalNdet } A \ Q \rangle$
and *mono-GlobalNdet-DT[simp, elim]*:
 $\langle \forall x \in A. P \ x \sqsubseteq_{DT} Q \ x \implies \text{GlobalNdet } A \ P \sqsubseteq_{DT} \text{GlobalNdet } A \ Q \rangle$
and *mono-GlobalNdet-FD[simp, elim]*:
 $\langle \forall x \in A. P \ x \sqsubseteq_{FD} Q \ x \implies \text{GlobalNdet } A \ P \sqsubseteq_{FD} \text{GlobalNdet } A \ Q \rangle$
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*:
 $\langle A \neq \{\} \implies A \subseteq B \implies \text{GlobalNdet } B \ P \sqsubseteq_{FD} \text{GlobalNdet } A \ P \rangle$
by (*metis mono-Ndet-FD-left GlobalNdet-factorization-union*
bot.extremum-uniqueI idem-FD le-iff-sup)

lemma *GlobalNdet-refine-F-subset*:
 $\langle A \neq \{\} \implies A \subseteq B \implies \text{GlobalNdet } B \ P \sqsubseteq_F \text{GlobalNdet } A \ P \rangle$
by (*simp add: GlobalNdet-refine-FD-subset leFD-imp-leF*)

lemma *GlobalNdet-refine-FD*: $\langle a \in A \implies \text{GlobalNdet } A \ P \sqsubseteq_{FD} P \ a \rangle$
using *GlobalNdet-refine-FD-subset[of* $\langle \{a\} \rangle$ $A]$ **by** *simp*

lemma *GlobalNdet-refine-F*: $\langle a \in A \implies \text{GlobalNdet } A \ P \sqsubseteq_F P \ a \rangle$
by (*simp add: GlobalNdet-refine-FD leFD-imp-leF*)

lemma *mono-GlobalNdet-FD-const*:
 $\langle A \neq \{\} \implies \forall x \in A. P \sqsubseteq_{FD} Q \ x \implies P \sqsubseteq_{FD} \text{GlobalNdet } A \ Q \rangle$

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

lemma *mono-GlobalNdet-F-const*:

$\langle A \neq \{\} \implies \forall x \in A. P \sqsubseteq_F Q \ x \implies P \sqsubseteq_F \text{GlobalNdet } A \ Q \rangle$

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

9.2 Combination of Multi-Operators Laws

lemma *finite-Mprefix-is-MultiDet*:

$\langle \text{finite } A \implies (\Box p \in A \rightarrow P \ p) = \Box p \in A. (p \rightarrow P \ p) \rangle$

by (*induct rule: finite-induct, simp-all add: Mprefix-STOP*)

(*metis Mprefix-Un-distrib Mprefix-singl insert-is-Un*)

lemma *finite-Mndetprefix-is-MultiNdet*:

$\langle \text{finite } A \implies \text{Mndetprefix } A \ P = \prod p \in A. (p \rightarrow P \ p) \rangle$

by (*cases* $\langle A = \{\} \rangle$, *simp, rotate-tac, induct rule: finite-set-induct-nonempty*)

(*simp-all add: Mndetprefix-unit Mndetprefix-distrib-unit*)

lemma $\langle Q \Box (\prod p \in \{\}. P \ p) = \prod p \in \{\}. (Q \Box P \ p) \implies Q = \text{STOP} \rangle$

by (*simp add: Det-STOP*)

lemma *Det-MultiNdet-distrib*:

$\langle A \neq \{\} \implies \text{finite } A \implies M \Box (\prod p \in A. P \ p) = \prod p \in A. M \Box P \ p \rangle$

by (*erule finite-set-induct-nonempty, simp-all add: Det-distrib*)

lemma $\langle M \prod (\Box p \in \{\}. P \ p) = \Box p \in \{\}. (M \prod P \ p) \implies M \prod \text{STOP} = \text{STOP} \rangle$

by *simp*

lemma *Ndet-MultiDet-distrib*:

$\langle A \neq \{\} \implies \text{finite } A \implies M \prod (\Box p \in A. P \ p) = \Box p \in A. M \prod P \ p \rangle$

by (*erule finite-set-induct-nonempty, simp-all add: Ndet-distrib*)

lemma *MultiNdet-Sync-left-distrib*:

$\langle B \neq \{\} \implies \text{finite } B \implies (\prod a \in B. P \ a) \llbracket S \rrbracket M = \prod a \in B. (P \ a \llbracket S \rrbracket M) \rangle$

by (*induct rule: finite-set-induct-nonempty*)

(*simp-all add: Sync-Ndet-left-distrib*)

lemma *MultiNdet-Sync-right-distrib*:

$\langle B \neq \{\} \implies \text{finite } B \implies M \llbracket S \rrbracket \text{MultiNdet } B \ P = \prod a \in B. (M \llbracket S \rrbracket P \ a) \rangle$

by (*subst Sync-commute, subst MultiNdet-Sync-left-distrib*)

(*simp-all add: Sync-commute*)

lemma *Sync-MultiNdet-cartprod*:

$\langle A \neq \{\} \implies \text{finite } A \implies B \neq \{\} \implies \text{finite } B \implies$
 $(\prod (s, t) \in A \times B. (x s \llbracket S \rrbracket y t)) = (\prod s \in A. x s) \llbracket S \rrbracket (\prod t \in B. y t) \rangle$
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** $S = \langle \{\} \rangle$]
and *Par-MultiNdet-cartprod* = *Sync-MultiNdet-cartprod*[**where** $S = UNIV$]

lemmas *MultiNdet-Inter-left-distrib* =
MultiNdet-Sync-left-distrib[**where** $S = \langle \{\} \rangle$]
and *MultiNdet-Par-left-distrib* =
MultiNdet-Sync-left-distrib[**where** $S = \langle UNIV \rangle$]

lemma *Seq-MultiNdet-distribR*:

$\langle \text{finite } A \implies (\prod p \in A. P p) ; S = (\prod p \in A. (P p ; S)) \rangle$
by (*induct A rule: finite-induct, simp add: STOP-Seq*)
(metis MultiNdet-insert' MultiNdet-rec1 Seq-Ndet-distrR)

lemma *Seq-MultiNdet-distribL*:

$\langle A \neq \{\} \implies \text{finite } A \implies S ; (\prod p \in A. P p) = (\prod p \in A. (S ; P p)) \rangle$
by (*induct A rule: finite-set-induct-nonempty, simp-all add: Seq-Ndet-distrL*)

lemma *prefix-MultiNdet-distrib*:

$\langle A \neq \{\} \implies \text{finite } A \implies (a \rightarrow (\prod p \in A. P p)) = \prod p \in A. (a \rightarrow P p) \rangle$
by (*induct A rule: finite-set-induct-nonempty, simp-all add: write0-Ndet*)

lemma *Mndetprefix-MultiNdet-distrib*:

$\langle (\prod q \in B \rightarrow (\prod p \in A. P p q)) = \prod p \in A. \prod q \in B \rightarrow P p q \rangle$
if *finB*: $\langle \text{finite } B \rangle$ **and** *nonemptyA*: $\langle A \neq \{\} \rangle$ **and** *finA*: $\langle \text{finite } A \rangle$
proof (*cases* $\langle B = \{\} \rangle$)
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  $\langle \{x\} \rangle$ , simplified, OF  $\langle F \neq \{\} \rangle$ ])
  apply (subst Mndetprefix-unit,
    subst prefix-MultiNdet-distrib[OF nonemptyA finA])
  apply (subst insertion.hyps(5))
  apply (subst MultiNdet-Ndet[OF finA])
  by (insert  $\langle F \neq \{\} \rangle \langle x \notin F \rangle$ , subst Mndetprefix-distrib-unit) force+
qed
qed

```

lemma *MultiDet-Mprefix*:

$$\langle \text{finite } A \implies (\Box a \in A. \Box x \in S a \rightarrow P a x) = \Box x \in (\bigcup a \in A. S a) \rightarrow \prod a \in \{a \in A. x \in S a\}. P a x \rangle$$

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: $\langle \text{finite } F \rangle$)

apply (subst 3.hyps)

apply (subst Mprefix-Det-distr)

apply (intro mono-Mprefix-eq ballI)

using $\langle \text{finite } F \rangle$ **by** (auto simp add: Process-eq-spec F-MultiNdet F-Ndet D-MultiNdet D-Ndet)

qed

lemma *MultiDet-prefix-is-MultiNdet-prefix*:

$$\langle \text{finite } A \implies (\Box p \in A. (a \rightarrow P p)) = \prod p \in A. (a \rightarrow P p) \rangle$$

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*:

$$\langle A \neq \{\} \implies \text{finite } A \implies (a \rightarrow (\prod p \in A. P p)) = \Box p \in A. (a \rightarrow P p) \rangle$$

by (simp add: MultiDet-prefix-is-MultiNdet-prefix prefix-MultiNdet-distrib)

lemma *Mprefix-MultiNdet-distrib'*:

$$\langle \text{finite } B \implies A \neq \{\} \implies \text{finite } A \implies$$

$(\Box q \in B \rightarrow \Box p \in A. P p q) = \Box p \in A. \Box q \in B \rightarrow P p q$
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.premis]*)
apply (*subst MultiDet-prefix-is-MultiNdet-prefix[symmetric, OF ⟨finite A⟩]*)
apply (*subst insert.hyps(3)[OF insert.premis]*)
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*:
 $\langle \text{finite } B \implies B \cap S = \{\} \implies$
 $(\llbracket S \rrbracket p \in \# M. ((P p) \setminus B)) = (\llbracket S \rrbracket p \in \# M. P p) \setminus B$
by (*induct M, simp add: Hiding-set-STOP*)
(metis MultiSync-add MultiSync-rec1 Hiding-Sync)

lemma *MultiSync-prefix-pseudo-distrib*:
 $\langle M \neq \{\#\} \implies a \in S \implies$
 $((\llbracket S \rrbracket p \in \# M. (a \rightarrow P p)) = (a \rightarrow (\llbracket S \rrbracket p \in \# M. P p))) \rangle$
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*:
 $\langle \text{finite } A \implies (\Box p \in A. P p) \setminus B = (\Box p \in A. (P p \setminus B)) \rangle$
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*:
 $\langle \text{finite } A \implies \Box p \in A - B \rightarrow ((P p) \setminus B) = \Box p \in A - B. (p \rightarrow ((P p) \setminus B)) \rangle$
proof (*induct A rule: finite-induct*)
case empty
thus ?*case* **by** *fastforce*
next
show $\langle \text{finite } F \implies x \notin F \implies$

$\sqcap p \in F - B \rightarrow (P p \setminus B) = \sqcap p \in F - B. (p \rightarrow (P p \setminus B)) \implies$
 $\sqcap p \in \text{insert } x F - B \rightarrow (P p \setminus B) =$
 $\sqcap p \in \text{insert } x F - B. (p \rightarrow (P p \setminus B)) \rangle \text{ for } x F$
apply (cases $\langle x \in B \rangle$, simp)
apply (cases $\langle F - B = \{\} \rangle$,
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*:
 $\langle \text{finite } A \implies A \subseteq B \implies (\sqcap a \in A \rightarrow P) \setminus B = \sqcap a \in A. (P \setminus B) \rangle$
by (cases $\langle A = \{\} \rangle$, 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*:
 $\langle (\llbracket S \rrbracket B \in \# M. \sqcap x \in B \rightarrow P B x) =$
 $\sqcap x \in (\bigcap B \in \text{set-mset } M. B) \rightarrow (\llbracket S \rrbracket B \in \# M. P B x) \rangle$
if nonempty: $\langle M \neq \{\# \} \rangle$ **and** hyp: $\langle \forall B \in \# M. B \subseteq S \rangle$
proof –
from nonempty **obtain** $b M'$ **where** $\langle b \in \# M - M' \rangle$
 $\langle M = \text{add-mset } b M' \rangle \langle M' \subseteq \# M \rangle$
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) $\langle M = \text{add-mset } b M' \rangle$)
using $\langle b \in \# M - M' \rangle \langle M' \subseteq \# M \rangle$
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 $\langle b \in \# M - M' \rangle$ hyp in-diffD le-infI1)
using add.hyps(3) **by** fastforce
qed
qed

lemmas *MultiPar-Mprefix-pseudo-distrib* =
MultiSync-Mprefix-pseudo-distrib[**where** $S = \langle UNIV \rangle$, simplified]

A result on Mndetprefix and Sync.

lemma *Mndetprefix-Sync-distr*: $\langle A \neq \{\} \implies B \neq \{\} \implies$

$$\begin{aligned}
& (\sqcap a \in A \rightarrow P a) \llbracket S \rrbracket (\sqcap b \in B \rightarrow Q b) = \\
& \sqcap a \in A. \sqcap b \in B. (\sqcap c \in \{a\} - S \rightarrow (P a \llbracket S \rrbracket (b \rightarrow Q b))) \square \\
& \quad (\sqcap d \in \{b\} - S \rightarrow ((a \rightarrow P a) \llbracket S \rrbracket Q b)) \square \\
& \quad (\sqcap c \in \{a\} \cap \{b\} \cap S \rightarrow (P a \llbracket S \rrbracket Q b)) \rangle
\end{aligned}$$

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*:

$\langle SEQ a \in @ L. P a \rangle =$
 $SEQ a \in @ take (Suc (first-elem (\lambda a. non-terminating (P a)) L)) L. P a \rangle$
by (*induct L; simp add: non-terminating-Seq*)

9.3 Results on Renaming

lemma *Renaming-GlobalNdet*:

$\langle Renaming (\sqcap a \in A. P (f a)) f = \sqcap b \in f ' A. Renaming (P b) f \rangle$
by (*subst Process-eq-spec*)
(auto simp add: F-Renaming D-Renaming F-GlobalNdet D-GlobalNdet)

lemma *Renaming-GlobalNdet-inj-on*:

$\langle Renaming (\sqcap a \in A. P a) f =$
 $\sqcap b \in f ' A. Renaming (P (THE a. a \in A \wedge f a = b)) f \rangle$
if *inj-on-f*: $\langle inj-on f A \rangle$
apply (*subst Renaming-GlobalNdet[symmetric]*)
apply (*intro arg-cong[where f = $\langle \lambda Q. Renaming Q f \rangle$ mono-GlobalNdet-eq]*)
by (*metis inj-on-f the-inv-into-def the-inv-into-f-f*)

corollary *Renaming-GlobalNdet-inj*:

$\langle Renaming (\sqcap a \in A. P a) f =$
 $\sqcap b \in f ' A. Renaming (P (THE a. f a = b)) f \rangle$ **if** *inj-f*: $\langle inj f \rangle$
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*: $\langle finite A \implies Renaming (\sqcap a \in A. P (f a)) f =$

$\sqcap b \in f ' A. Renaming (P b) f \rangle$
by (*subst* (1 2) *finite-GlobalNdet-is-MultiNdet[symmetric]*)
(simp-all add: Renaming-GlobalNdet)

lemma *Renaming-MultiNdet-inj-on*:
 $\langle \text{finite } A \implies \text{inj-on } f \ A \implies$
 $\text{Renaming } (\prod a \in A. P \ a) \ f =$
 $\prod b \in f \ ' \ A. \text{Renaming } (P \ (\text{THE } a. a \in A \wedge f \ a = b)) \ f \rangle$
by (*subst (1 2) finite-GlobalNdet-is-MultiNdet[symmetric]*)
(simp-all add: Renaming-GlobalNdet-inj-on)

corollary *Renaming-MultiNdet-inj*:
 $\langle \text{finite } A \implies \text{inj } f \implies$
 $\text{Renaming } (\prod a \in A. P \ a) \ f = \prod b \in f \ ' \ A. \text{Renaming } (P \ (\text{THE } a. f \ a = b)) \ f \rangle$
by (*subst (1 2) finite-GlobalNdet-is-MultiNdet[symmetric]*)
(simp-all add: Renaming-GlobalNdet-inj)

lemma *Renaming-MultiDet*:
 $\langle \text{finite } A \implies \text{Renaming } (\sqcap a \in A. P \ (f \ a)) \ f =$
 $\sqcap b \in f \ ' \ A. \text{Renaming } (P \ b) \ f \rangle$
apply (*induct A rule: finite-induct*)
by (*simp-all add: Renaming-STOP Renaming-Det del: MultiDet-insert*)

lemma *Renaming-MultiDet-inj-on*:
 $\langle \text{Renaming } (\sqcap a \in A. P \ a) \ f =$
 $\sqcap b \in f \ ' \ A. \text{Renaming } (P \ (\text{THE } a. a \in A \wedge f \ a = b)) \ f \rangle$
if *finite-A*: $\langle \text{finite } A \rangle$ **and** *inj-on-f*: $\langle \text{inj-on } f \ A \rangle$
apply (*subst Renaming-MultiDet[OF finite-A, symmetric]*)
apply (*intro arg-cong[where f = $\langle \lambda Q. \text{Renaming } Q \ f \rangle$*
mono-MultiDet-eq finite-A])
by (*metis inj-on-f the-inv-into-def the-inv-into-f-f*)

corollary *Renaming-MultiDet-inj*:
 $\langle \text{Renaming } (\sqcap a \in A. P \ a) \ f = \sqcap b \in f \ ' \ A. \text{Renaming } (P \ (\text{THE } a. f \ a = b)) \ f \rangle$
if *finite-A*: $\langle \text{finite } A \rangle$ **and** *inj-f*: $\langle \text{inj } f \rangle$
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*:
 $\langle \text{Renaming } (\text{SEQ } l \in @ \ L. P \ (f \ l)) \ f = \text{SEQ } m \in @ \ \text{map } f \ L. \text{Renaming } (P \ m) \ f \rangle$
by (*induct L, simp-all add: Renaming-SKIP Renaming-Seq*)

lemma *Renaming-MultiSeq-inj-on*:
 $\langle \text{Renaming } (\text{SEQ } l \in @ \ L. P \ l) \ f =$
 $\text{SEQ } m \in @ \ \text{map } f \ L. \text{Renaming } (P \ (\text{THE } l. l \in \text{set } L \wedge f \ l = m)) \ f \rangle$
if *inj-on-f*: $\langle \text{inj-on } f \ (\text{set } L) \rangle$
apply (*subst Renaming-MultiSeq[symmetric]*)

apply (*intro arg-cong*[**where** $f = \langle \lambda Q. \text{Renaming } Q \ f \rangle$] *mono-MultiSeq-eq*)
by (*metis that the-inv-into-def the-inv-into-f-f*)

corollary *Renaming-MultiSeq-inj*:

$\langle \text{Renaming } (\text{SEQ } l \in @ L. P \ l) \ f =$
 $\text{SEQ } m \in @ \text{ map } f \ L. \text{Renaming } (P \ (\text{THE } l. \ f \ l = m)) \ f \rangle$ **if** *inj-f*: $\langle \text{inj } f \rangle$
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  $N-g1[simp] : \langle N > 1 \rangle$ 
```

— 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-event = picks (phil:nat) (fork:nat)
  | putsdown (phil:nat) (fork:nat)
```

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

```
definition RPHIL::  $\langle nat \Rightarrow dining-event process \rangle$ 
```

```
  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 LPHILO::  $\langle dining-event process \rangle$ 
```

```
  where  $\langle LPHILO \equiv \mu X. (picks\ 0\ (N-1) \rightarrow (picks\ 0\ 0 \rightarrow$   
     $(putsdown\ 0\ 0 \rightarrow (putsdown\ 0\ (N-1) \rightarrow X))) \rangle$ 
```

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

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

definition $\langle \text{PHILS} \equiv ||| P \in\# \text{ add-mset } \text{LPHILO} \ (\text{mset} \ (\text{map } \text{RPHIL} \ [1..< N]))$.
 $P \rangle$

definition $\langle \text{FORKS} \equiv ||| P \in\# \text{ mset} \ (\text{map } \text{FORK} \ [0..< N])$. $P \rangle$

corollary $\langle N = 3 \implies \text{PHILS} = (\text{LPHILO} \ ||| \ \text{RPHIL } 1 \ ||| \ \text{RPHIL } 2) \rangle$

— 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* :: $\langle \text{dining-event process} \rangle$

where $\langle \text{DINING} = (\text{FORKS} \ || \ \text{PHILS}) \rangle$

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*:

$(x::\text{nat}) \sqsubseteq y \iff x = y$

instance **proof**

qed (*rule below-nat-def*)

end

locale *DiningPhilosophers-fixrec* =

fixes $N::\text{nat}$

assumes $N-g1[\text{simp}] : \langle N > 1 \rangle$

— 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-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* :: $\langle \text{nat} \rightarrow \text{dining-event process} \rangle$
and *LPHILO* :: $\langle \text{dining-event process} \rangle$
and *FORK* :: $\langle \text{nat} \rightarrow \text{dining-event process} \rangle$

where

RPHIL-rec [*simp del*] :
 $\langle \text{RPHIL} \cdot i = (\text{picks } i \ i \rightarrow (\text{picks } i \ (i-1) \rightarrow$
 $(\text{putsdown } i \ (i-1) \rightarrow (\text{putsdown } i \ i \rightarrow \text{RPHIL} \cdot i))) \rangle$
 | *LPHILO-rec* [*simp del*] :
 $\langle \text{LPHILO} = (\text{picks } 0 \ (N-1) \rightarrow (\text{picks } 0 \ 0 \rightarrow$
 $(\text{putsdown } 0 \ 0 \rightarrow (\text{putsdown } 0 \ (N-1) \rightarrow \text{LPHILO}))) \rangle$
 | *FORK-rec* [*simp del*] :
 $\langle \text{FORK} \cdot i = (\text{picks } i \ i \rightarrow (\text{putsdown } i \ i \rightarrow \text{FORK} \cdot i)) \square$
 $(\text{picks } ((i+1) \ \text{mod } N) \ i \rightarrow (\text{putsdown } ((i+1) \ \text{mod } N) \ i \rightarrow \text{FORK} \cdot i)) \rangle$

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

definition $\langle \text{PHILS} \equiv ||| P \in\# \text{add-mset } \text{LPHILO} \ (\text{mset} \ (\text{map} \ (\lambda i. \ \text{RPHIL} \cdot i) \ [1..<N])) . P \rangle$

definition $\langle \text{FORKS} \equiv ||| P \in\# \text{mset} \ (\text{map} \ (\lambda i. \ \text{FORK} \cdot i) \ [0..<N]) . P \rangle$

corollary $\langle N = 3 \implies \text{PHILS} = (\text{LPHILO} \ ||| \ \text{RPHIL} \cdot 1 \ ||| \ \text{RPHIL} \cdot 2) \rangle$

— 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* :: $\langle \text{dining-event process} \rangle$

where $\langle \text{DINING} = (\text{FORKS} \ ||| \ \text{PHILS}) \rangle$

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* = $\langle int \rangle$

datatype *channels* = *tcO* $\langle Phones \times MtcO \rangle$ —
| *ctO* $\langle Phones \times MctO \rangle$
| *tcT* $\langle Phones \times MctT \times Phones \rangle$
| *ctT* $\langle Phones \times MctT \times Phones \rangle$
| *tcOdial* $\langle Phones \times Phones \rangle$
| *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* $\langle Phones \times Phones \rangle$ — 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* :: $\langle channels \ set \rangle$

assumes *min-phones-g-1*[*simp*] : $\langle 1 \leq min-phones \rangle$

and *max-phones-g-min-phones*[*simp*] : $\langle min-phones < max-phones \rangle$

begin

definition $phones :: \langle Phones\ set \rangle$ **where** $\langle phones \equiv \{min\text{-phones} .. max\text{-phones}\} \rangle$

lemma $nonempty\text{-phones}[simp]: \langle phones \neq \{\} \rangle$
and $finite\text{-phones}[simp]: \langle finite\ phones \rangle$
and $at\text{-least}\text{-two}\text{-phones}[simp]: \langle 2 \leq card\ phones \rangle$
and $not\text{-singl}\text{-phone}[simp]: \langle phones - \{p\} \neq \{\} \rangle$
apply ($simp\text{-all}\ add: phones\text{-def}$)
using $max\text{-phones}\text{-g}\text{-min}\text{-phones}$ **apply** $linarith+$
by ($metis\ atLeastAtMost\text{-iff}\ less\text{-le}\text{-not}\text{-le}\ max\text{-phones}\text{-g}\text{-min}\text{-phones}\ order\text{-refl}\ singletonD\ subsetD$)

definition $EventsIPhone :: \langle Phones \Rightarrow channels\ set \rangle$
where $\langle EventsIPhone\ u \equiv \{tone\text{-ring}\ u, tone\text{-quiet}\ u, tone\text{-busy}\ u, tone\text{-dial}\ u, connected\ u\} \rangle$

definition $EventsUser :: \langle Phones \Rightarrow channels\ set \rangle$
where $\langle EventsUser\ u \equiv \{off\text{-hook}\ u, on\text{-hook}\ u\} \cup \{x . \exists n. x = digits\ (u, n)\} \rangle$

11.2 Auxilliaris to Substructure the Specification

abbreviation $Sliding :: \langle 'a\ process \Rightarrow 'a\ process \Rightarrow 'a\ process \rangle$ (**infixl** $\langle \triangleright \rangle$ 78)

where $\langle P \triangleright Q \equiv (P \square Q) \square Q \rangle$

— This operator is also called Timeout, more studied in future theories.

abbreviation

$Tside\text{-connected} :: \langle Phones \Rightarrow Phones \Rightarrow channels\ process \rangle$

where $\langle Tside\text{-connected}\ ts\ os \equiv$
 $(ctT!(ts, Tdiscon\text{-}o, os) \rightarrow tcT!(ts, Tdiscon\text{-}t, os) \rightarrow EndReject!ts \rightarrow SKIP)$
 $\triangleright (tcT!(ts, Tdiscon\text{-}t, os) \rightarrow ctT!(ts, Tdiscon\text{-}o, os) \rightarrow EndReject!ts \rightarrow SKIP) \rangle$

abbreviation

$Oside\text{-connected} :: \langle Phones \Rightarrow channels\ process \rangle$

where $\langle Oside\text{-connected}\ ts \equiv$
 $(ctO!(ts, Odiscon\text{-}t) \rightarrow tcO!(ts, Odiscon\text{-}o) \rightarrow EndReject!ts \rightarrow SKIP)$
 $\triangleright (tcO!(ts, Odiscon\text{-}o) \rightarrow ctO!(ts, Odiscon\text{-}t) \rightarrow EndReject!ts \rightarrow SKIP) \rangle$

abbreviation

$Oside1 :: \langle [Phones, Phones] \Rightarrow channels\ process \rangle$

where

$\langle Oside1\ ts\ p \equiv$
 $tcO!dial!(ts, p)$
 $\rightarrow (ctO!(ts, Oalert)$
 $\rightarrow ctO!(ts, Oconnect)$
 $\rightarrow (Oside\text{-connected}\ ts)) \rangle$

$$\langle \text{SKIP} \rangle$$

$$\square (ctO!(ts, Oconnect) \rightarrow (Oside\text{-}connected\ ts))$$

$$\square (ctO!(ts, Obusy) \rightarrow tcO!(ts, Odiscon\text{-}o) \rightarrow EndReject!ts \rightarrow$$

definition

$$ITside\text{-}connected \quad :: \langle [Phones, Phones, channels\ process] \Rightarrow channels\ process \rangle$$

where

$$\langle ITside\text{-}connected\ ts\ os\ IT \equiv (ctT(ts, Tdiscon\text{-}o, os)$$

$$\rightarrow (\text{tone-busy!}ts$$

$$\rightarrow on\text{-}hook!ts$$

$$\rightarrow tcT!(ts, Tdiscon\text{-}t, os)$$

$$\rightarrow EndReject!ts$$

$$\rightarrow IT)$$

$$\square (on\text{-}hook!ts$$

$$\rightarrow tcT!(ts, Tdiscon\text{-}t, os)$$

$$\rightarrow EndReject!ts$$

$$\rightarrow IT)$$

$$))$$

$$\square (on\text{-}hook!ts$$

$$\rightarrow tcT!(ts, Tdiscon\text{-}t, os)$$

$$\rightarrow ctT!(ts, Tdiscon\text{-}o, os)$$

$$\rightarrow EndReject!ts$$

$$\rightarrow IT) \rangle$$

11.3 A Telephone

fixrec $T \quad :: \langle Phones \rightarrow channels\ process \rangle$
and $Oside \quad :: \langle Phones \rightarrow channels\ process \rangle$
and $Tside \quad :: \langle Phones \rightarrow channels\ process \rangle$
and $NoReject \quad :: \langle Phones \rightarrow channels\ process \rangle$
and $Reject \quad :: \langle Phones \rightarrow channels\ process \rangle$

where

$$T\text{-}rec \quad [simp\ del]: \langle T \cdot ts = (Tside \cdot ts ; T \cdot ts) \triangleright (Oside \cdot ts ; T \cdot ts) \rangle$$

$$| Oside\text{-}rec \quad [simp\ del]: \langle Oside \cdot ts = StartReject!ts$$

$$\rightarrow tcO!(ts, Osetup)$$

$$\rightarrow (\prod p \in phones. Oside1\ ts\ p) \rangle$$

$$| Tside\text{-}rec \quad [simp\ del]: \langle Tside \cdot ts = ctT?(y, z, os) | ((y, z) = (ts, Tsetup))$$

$$\rightarrow StartReject!ts$$

$$\rightarrow (\quad tcT!(ts, Talert, os)$$

$$\rightarrow tcT!(ts, Tconnect, os)$$

$$\rightarrow (Tside\text{-}connected\ ts\ os)$$

$$\square (tcT!(ts, Tconnect, os)$$

$$\rightarrow (Tside\text{-}connected\ ts\ os))) \rangle$$

$$| NoReject\text{-}rec \quad [simp\ del]: \langle NoReject \cdot ts = StartReject!ts \rightarrow Reject \cdot ts \rangle$$

$$| Reject\text{-}rec \quad [simp\ del]: \langle Reject \cdot ts = ctT?(y, z, os) | (y = ts \wedge z = Tsetup \wedge os \in phones$$

$$\wedge os \neq ts)$$

$$\rightarrow (tcT!(ts, Tbusy, os) \rightarrow Reject \cdot ts)$$

$$\square (EndReject!ts \rightarrow NoReject \cdot ts) \rangle$$

definition $Tel :: \langle Phones \Rightarrow channels\ process \rangle$

where $\langle Tel\ p \equiv (T \cdot p \llbracket \{StartReject\ p, EndReject\ p\} \rrbracket NoReject \cdot p) \setminus \{StartReject\ p, EndReject\ p\} \rangle$

11.4 A Connector with the Network

fixrec $Call :: \langle Phones \rightarrow channels\ process \rangle$

and $BUSY :: \langle Phones \rightarrow Phones \rightarrow channels\ process \rangle$

and $Connected :: \langle Phones \rightarrow Phones \rightarrow channels\ process \rangle$

where

$Call-rec$ [simp del]: $\langle Call \cdot os = (tcO! (os, Osetup) \rightarrow tcO! \text{dial?}(x, ts) | (x=os)) \rightarrow (BUSY \cdot os \cdot ts) \rangle ; Call \cdot os$

| $BUSY-rec$ [simp del]: $\langle BUSY \cdot os \cdot ts = (if\ ts = os$
 $then\ ctO!(os, Obusy) \rightarrow tcO!(os, Odiscon-o) \rightarrow SKIP$
 $else\ ctT!(ts, Tsetup, os)$

$\rightarrow ((tcT!(ts, Tbusy, os)$
 $\rightarrow ctO!(os, Obusy)$
 $\rightarrow tcO!(os, Odiscon-o) \rightarrow SKIP)$

□

$(tcT!(ts, Talert, os)$
 $\rightarrow ctO!(os, Oalert)$
 $\rightarrow tcT!(ts, Tconnect, os)$
 $\rightarrow ctO!(os, Oconnect)$
 $\rightarrow Connected \cdot os \cdot ts)$

□

$(tcT!(ts, Tconnect, os)$
 $\rightarrow ctO!(os, Oconnect)$
 $\rightarrow Connected \cdot os \cdot ts)) \rangle$

| $Connected-rec$ [simp del]: $\langle Connected \cdot os \cdot ts = (tcO!(os, Odiscon-o) \rightarrow$
 $(((ctT!(ts, Tdiscon-o, os) \rightarrow tcT!(ts, Tdiscon-t, os) \rightarrow SKIP)$

□

$(tcT!(ts, Tdiscon-t, os) \rightarrow ctT!(ts, Tdiscon-o, os) \rightarrow SKIP)$
 $)$

$; (ctO!(os, Odiscon-t) \rightarrow SKIP)))$

□

$(tcT!(ts, Tdiscon-t, os) \rightarrow$

$((ctO!(os, Odiscon-t)$
 $\rightarrow ctT!(ts, Tdiscon-o, os)$
 $\rightarrow tcO!(os, Odiscon-o)$
 $\rightarrow SKIP)$

□

$(tcO!(os, Odiscon-o)$
 $\rightarrow ctT!(ts, Tdiscon-o, os)$
 $\rightarrow ctO!(os, Odiscon-t)$

$$\begin{aligned} & \rightarrow SKIP) \\ &) \\ &) \end{aligned}$$

11.5 Combining NETWORK and TELEPHONES to a SYSTEM

definition $NETWORK$ $:: \langle channels\ process \rangle$
where $\langle NETWORK \equiv (||| os \in \# (mset\text{-}set\ phones). Call\ os) \rangle$

definition $TELEPHONES$ $:: \langle channels\ process \rangle$
where $\langle TELEPHONES \equiv (||| ts \in \# (mset\text{-}set\ phones). Tel\ ts) \rangle$

definition $SYSTEM$ $:: \langle channels\ process \rangle$
where $\langle SYSTEM \equiv NETWORK \llbracket VisibleEvents \rrbracket TELEPHONES \rangle$

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 $User$ $:: \langle Phones \rightarrow channels\ process \rangle$
and $UserSCon$ $:: \langle Phones \rightarrow channels\ process \rangle$

where

$$\begin{aligned} User\text{-}rec[simp\ del] : \langle User \cdot u = & (off\text{-}hook!u \rightarrow \\ & (tone\text{-}dial!u \rightarrow \\ & (\sqcap p \in phones. digits!(u,p) \rightarrow tone\text{-}quiet!u \rightarrow \\ & \quad ((tone\text{-}ring!u \rightarrow connected!u \rightarrow UserSCon \cdot u) \\ & \quad \square (connected!u \rightarrow UserSCon \cdot u) \\ & \quad \square (tone\text{-}busy!u \rightarrow on\text{-}hook!u \rightarrow User \cdot u) \\ & \quad) \\ & \quad) \\ & \quad) \\ & \square (connected!u \rightarrow UserSCon \cdot u) \\ & \quad) \\ & \square (tone\text{-}ring!u \rightarrow off\text{-}hook!u \rightarrow connected!u \rightarrow UserSCon \cdot u) \rangle \\ | UserSCon\text{-}rec[simp\ del]: \langle UserSCon \cdot u = & (tone\text{-}busy!u \rightarrow on\text{-}hook!u \rightarrow User \cdot u) \\ \triangleright (on\text{-}hook!u \rightarrow User \cdot u) \rangle \end{aligned}$$

fixrec $User\text{-}Ndet$ $:: \langle Phones \rightarrow channels\ process \rangle$
and $UserSCon\text{-}Ndet$ $:: \langle Phones \rightarrow channels\ process \rangle$

where

$$\begin{aligned} User\text{-}Ndet\text{-}rec[simp\ del] : \langle User\text{-}Ndet \cdot u = & (off\text{-}hook!u \rightarrow \\ & (tone\text{-}dial!u \rightarrow \\ & (\sqcap p \in phones. digits!(u,p) \rightarrow tone\text{-}quiet!u \rightarrow \end{aligned}$$

$$\begin{aligned}
& ((\text{tone-ring!}u \rightarrow \text{connected!}u \rightarrow \text{UserSCon-Ndet}\cdot u) \\
& \quad \sqcap (\text{connected!}u \rightarrow \text{UserSCon-Ndet}\cdot u) \\
& \quad \sqcap (\text{tone-busy!}u \rightarrow \text{on-hook!}u \rightarrow \text{User-Ndet}\cdot u) \\
&) \\
&) \\
&) \\
& \sqcap (\text{connected!}u \rightarrow \text{UserSCon-Ndet}\cdot u) \\
&) \\
& \sqcap (\text{tone-ring!}u \rightarrow \text{off-hook!}u \rightarrow \text{connected!}u \rightarrow \text{UserSCon-Ndet}\cdot u) \rangle \\
| \text{UserSCon-Ndet-rec}[\text{simp del}]: \langle \text{UserSCon-Ndet}\cdot u = (\text{tone-busy!}u \rightarrow \text{on-hook!}u \\
\rightarrow \text{User-Ndet}\cdot u) \sqcap (\text{on-hook!}u \rightarrow \text{User-Ndet}\cdot u) \rangle
\end{aligned}$$

definition $\text{Implement}T \quad :: \langle \text{Phones} \Rightarrow \text{channels process} \rangle$
where $\langle \text{Implement}T \text{ ts} \equiv ((\text{Tel ts}) \llbracket \text{EventsIPhone ts} \cup \text{EventsUser ts} \rrbracket (\text{User}\cdot \text{ts}))$
 $\quad \setminus (\text{EventsIPhone ts} \cup \text{EventsUser ts}) \rangle$

11.7 Toplevel Proof-Goals

This has been proven in an ancient FDR model for $\text{max-phones} = 5\dots$

lemma $\langle \forall p \in \text{phones. deadlock-free} (\text{Tel } p) \rangle$ **oops**
lemma $\langle \forall p \in \text{phones. deadlock-free-v2} (\text{Call}\cdot p) \rangle$ **oops**
lemma $\langle \text{deadlock-free-v2 NETWORK} \rangle$ **oops**
lemma $\langle \text{deadlock-free-v2 SYSTEM} \rangle$ **oops**
lemma $\langle \text{lifelock-free SYSTEM} \rangle$ **oops**
lemma $\langle \forall p \in \text{phones. lifelock-free} (\text{Implement}T \text{ } p) \rangle$ **oops**
lemma $\langle \forall p \in \text{phones. Tel } p \sqsubseteq_{FD} \text{Implement}T \text{ } p \rangle$ **oops**

lemma $\langle \forall p \in \text{phones. Tel}\cdot p \sqsubseteq_F \text{RUN UNIV} \rangle$ **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*:

$\langle \text{events-of } P = (\bigcup t \in \{t \in \mathcal{T} P. \text{tickFree } t\}. \{a. \text{ev } a \in \text{set } t\}) \rangle$

proof –

have $\langle s \in \mathcal{T} P \implies \neg \text{tickFree } s \implies \text{ev } e \in \text{set } s \implies \text{ev } e \in \text{set } (\text{butlast } s) \rangle$ **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*: $\langle \text{events-of } \perp = \text{UNIV} \rangle$

and *events-SKIP*: $\langle \text{events-of } \text{SKIP} = \{\} \rangle$

and *events-STOP*: $\langle \text{events-of } \text{STOP} = \{\} \rangle$

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*: $\langle P \sqsubseteq_T Q \implies \text{events-of } Q \subseteq \text{events-of } P \rangle$

unfolding *trace-refine-def events-of-def* **by** *fast*

lemma *anti-mono-events-F*: $\langle P \sqsubseteq_F Q \implies \text{events-of } Q \subseteq \text{events-of } P \rangle$

by (*intro anti-mono-events-T leF-imp-leT*)

lemma *anti-mono-events-FD*: $\langle P \sqsubseteq_{FD} Q \implies \text{events-of } Q \subseteq \text{events-of } P \rangle$

by (*intro anti-mono-events-F leFD-imp-leF*)

lemmas *events-fix-prefix* =
events-DF[of $\langle \{a\} \rangle$, *simplified DF-def Mndetprefix-unit*] **for** a

lemma *events-Mndetprefix*:
 $\langle \text{events-of } (Mndetprefix\ A\ P) = A \cup (\bigcup_{a \in A} \text{events-of } (P\ a)) \rangle$
apply (*cases* $\langle A = \{\} \rangle$, *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*:
 $\langle \text{events-of } (Mprefix\ A\ P) = A \cup (\bigcup_{a \in A} \text{events-of } (P\ a)) \rangle$
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*: $\langle \text{events-of } (a \rightarrow P) = \text{insert } a\ (\text{events-of } P) \rangle$
unfolding *write0-def* **by** (*simp add: events-Mprefix*)

lemma *events-Ndet*: $\langle \text{events-of } (P \sqcap Q) = \text{events-of } P \cup \text{events-of } Q \rangle$
unfolding *events-of-def* **by** (*simp add: T-Ndet*)

lemma *events-Det*: $\langle \text{events-of } (P \sqcup Q) = \text{events-of } P \cup \text{events-of } Q \rangle$
unfolding *events-of-def* **by** (*simp add: T-Det*)

lemma *events-Renaming*:
 $\langle \text{events-of } (Renaming\ P\ f) = (\text{if } \mathcal{D}\ P = \{\} \text{ then } f\ \langle \text{events-of } P \text{ else } UNIV) \rangle$
proof (*split if-split, intro conjI impI*)
show $\langle \mathcal{D}\ P \neq \{\} \implies \text{events-of } (Renaming\ P\ f) = UNIV \rangle$
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 $\langle \text{events-of } (Renaming\ P\ f) = f\ \langle \text{events-of } P \rangle \text{ if } \text{div-free} : \langle \mathcal{D}\ P = \{\} \rangle$
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** $\langle t1 = t1' @ [tick] \rangle$
using *DF-all-tickfree-traces2 T-nonTickFree-imp-decomp is-processT3-ST*
by *blast*
hence $\langle t2 \in \mathcal{T} Q \implies t1' @ t2 \in \mathcal{T} (P ; Q) \rangle$ **for** $t2$
using $*(1)$ *T-Seq* **by** *blast*
thus $\langle s \in \text{events-of } Q \implies s \in \text{events-of } (P ; Q) \rangle$
by (*simp add: events-of-def, elim bexE*)
(metis UnCI set-append)
qed
qed
qed

lemma *events-Sync*: $\langle \text{events-of } (P \llbracket S \rrbracket Q) \subseteq \text{events-of } P \cup \text{events-of } Q \rangle$
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** $\langle t \in \mathcal{D} P \wedge u \in \mathcal{T} Q \vee t \in \mathcal{D} Q \wedge u \in \mathcal{T} P \rangle$ **by** *blast*
thus ?*case* **using** *events-div* **by** *blast*
qed

lemma *events-Inter*:
 $\langle \text{events-of } ((P :: '\alpha \text{ process}) \parallel Q) = \text{events-of } P \cup \text{events-of } Q \rangle$
proof (*rule subset-antisym[OF events-Sync]*)
have $\langle \llbracket \text{tickFree } s; s \in \mathcal{T} (P :: '\alpha \text{ process}) \rrbracket \implies s \in \mathcal{T} (P \parallel Q) \rangle$ **for** $s P Q$
apply (*simp add: T-Sync*)
apply (*rule disjI1*)
apply (*rule-tac x = s in exI, simp*)
apply (*rule-tac x = \langle \rangle in exI, simp add: Nil-elem-T*)
by (*metis Sync.sym emptyLeftSelf singletonD tickFree-def*)
hence $\langle \text{events-of } (P :: '\alpha \text{ process}) \subseteq \text{events-of } (P \parallel Q) \rangle$ **for** $P Q$
unfolding *events-of-def-tickFree* **by** *fast*
thus $\langle \text{events-of } P \cup \text{events-of } Q \subseteq \text{events-of } (P \parallel Q) \rangle$
by (*metis Inter-commute Un-least*)
qed

lemma *empty-div-hide-events-Hiding*: $\langle \text{events-of } (P \setminus B) \subseteq \text{events-of } P - B \rangle$
if $\langle \text{div-hide } P B = \{\} \rangle$
unfolding *events-of-def T-Hiding*
apply (*subst that, simp*)
using *F-T* **by** *auto blast*

lemma *not-empty-div-hide-events-Hiding*:
 $\langle \text{div-hide } P \ B \neq \{\} \implies \text{events-of } (P \setminus B) = \text{UNIV} \rangle$
using *D-Hiding events-div* **by** *blast*

events-of and *deadlock-free*

lemma *nonempty-events-if-deadlock-free*: $\langle \text{deadlock-free } P \implies \text{events-of } P \neq \{\} \rangle$
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-elim-T T-F-spec UNIV-I hd-in-set is-processT5-S7*
list.distinct(1) self-append-conv2)

lemma *events-in-DF*: $\langle \text{DF } A \sqsubseteq_{FD} P \implies \text{events-of } P \subseteq A \rangle$
by (*metis anti-mono-events-FD events-DF*)

lemma *nonempty-events-if-deadlock-free_SKIP*:
 $\langle \text{deadlock-free}_{SKIP} P \implies [\text{tick}] \in \mathcal{T} P \vee \text{events-of } P \neq \{\} \rangle$
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-elim-T T-F-spec UNIV-I hd-in-set is-processT5-S7*
list.distinct(1) self-append-conv2)

lemma *events-in-DF_{SKIP}*: $\langle \text{DF}_{SKIP} A \sqsubseteq_{FD} P \implies \text{events-of } P \subseteq A \rangle$
by (*metis anti-mono-events-FD events-DF_{SKIP}*)

lemma $\langle \neg \text{events-of } P \subseteq A \implies \neg \text{DF } A \sqsubseteq_{FD} P \rangle$
and $\langle \neg \text{events-of } P \subseteq A \implies \neg \text{DF}_{SKIP} A \sqsubseteq_{FD} P \rangle$
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*:
 $\langle \text{finite } A \implies \text{events-of } (\text{MultiNdet } A \ P) = (\bigcup_{a \in A} \text{events-of } (P \ a)) \rangle$
by (*cases* $\langle A = \{\} \rangle$, *simp add: events-STOP*)
(rotate-tac, induct A rule: finite-set-induct-nonempty; simp add: events-Ndet)

lemma *events-MultiDet*:
 $\langle \text{finite } A \implies \text{events-of } (\text{MultiDet } A \ P) = (\bigcup_{a \in A} \text{events-of } (P \ a)) \rangle$
by (*induct A rule: finite-induct*) (*simp-all add: events-STOP events-Det*)

lemma *events-MultiSeq*:
 $\langle \text{events-of } (\text{SEQ } a \in @ L. P \ a) =$
 $(\bigcup_{a \in \text{set } (\text{take } (\text{Suc } (\text{first-elim } (\lambda a. \text{non-terminating } (P \ a)) \ L)) \ L)} a) \rangle$

$\langle \text{events-of } (P a) \rangle$
by (*subst non-terminating-MultiSeq, induct L; simp add: events-SKIP events-Seq*)

lemma *events-MultiSeq-subset*:
 $\langle \text{events-of } (SEQ a \in@ L. P a) \subseteq (\bigcup a \in \text{set } L. \text{events-of } (P a)) \rangle$
using *in-set-takeD* **by** (*subst events-MultiSeq*) *fastforce*

lemma *events-MultiSync*:
 $\langle \text{events-of } ([S] a \in\# M. P a) \subseteq (\bigcup a \in \text{set-mset } M. \text{events-of } (P a)) \rangle$
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*:
 $\langle \text{events-of } (||| a \in\# M. P a) = (\bigcup a \in \text{set-mset } M. \text{events-of } (P a)) \rangle$
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*:

```
 $\langle \mathcal{F}\ (DF\ A) =$ 
  (if  $A = \{\}$  then  $\{(s, X). s = []\}$ 
  else  $(\bigcup_{x \in A}. \{[]\} \times \{ref. ev\ x \notin ref\}) \cup$ 
     $\{(tr, ref). tr \neq [] \wedge hd\ tr = ev\ x \wedge (tl\ tr, ref) \in \mathcal{F}\ (DF\ A)\}) \rangle$ 
```

and *F-DF_{SKIP}*:

```
 $\langle \mathcal{F}\ (DF_{SKIP}\ A) =$ 
  (if  $A = \{\}$  then  $\{(s, X). s = [] \vee s = [tick]\}$ 
  else  $\{(s, X) \mid s\ X. s = [] \wedge tick \notin X \vee s = [tick]\} \cup$ 
     $(\bigcup_{x \in A}. \{[]\} \times \{ref. ev\ x \notin ref\}) \cup$ 
     $\{(tr, ref). tr \neq [] \wedge hd\ tr = ev\ x \wedge (tl\ tr, ref) \in \mathcal{F}\ (DF_{SKIP}\ A)\}) \rangle$ 
```

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*:

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

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

lemma *T-DF*:

$\langle \mathcal{T} (DF A) =$
 $(if A = \{\} then \{[]\}$
 $else \{s. s = [] \vee s \neq [] \wedge (\exists x \in A. hd s = ev x \wedge tl s \in \mathcal{T} (DF A))\}) \rangle$
and *T-DF_{SKIP}*:
 $\langle \mathcal{T} (DF_{SKIP} A) =$
 $(if A = \{\} then \{[], [tick]\}$
 $else \{s. s = [] \vee s = [tick] \vee$
 $s \neq [] \wedge (\exists x \in A. hd s = ev x \wedge tl s \in \mathcal{T} (DF_{SKIP} A))\}) \rangle$
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 $\llbracket deadlock\text{-free } P; deadlock\text{-free } Q \rrbracket \implies deadlock\text{-free } (P \sqcap Q)$, 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 $\langle deadlock\text{-free}_{SKIP} P \equiv DF_{SKIP} UNIV \sqsubseteq_F P \rangle$
by (*fact deadlock-free_{SKIP}-def*)

lemma *deadlock-free-F*: $\langle deadlock\text{-free } P \longleftrightarrow DF UNIV \sqsubseteq_F P \rangle$
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*: $\langle deadlock\text{-free } (\square a \in A \rightarrow P a) \longleftrightarrow$
 $A \neq \{\} \wedge (\forall a \in A. deadlock\text{-free } (P a)) \rangle$
and *deadlock-free_{SKIP}-Mprefix-iff*: $\langle deadlock\text{-free}_{SKIP} (Mprefix A P) \longleftrightarrow$

$A \neq \{\} \wedge (\forall a \in A. \text{deadlock-free}_{SKIP} (P a))$

unfolding *deadlock-free-F deadlock-free_{SKIP}-def failure-refine-def*
apply (*all* $\langle \text{subst } F\text{-DF } F\text{-DF}_{SKIP} \rangle$,
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* $\langle \{a\} \rangle \langle \lambda a. P \rangle$, *folded write0-def, simplified*]
and *deadlock-free_{SKIP}-prefix-iff* =
deadlock-free_{SKIP}-Mprefix-iff [*of* $\langle \{a\} \rangle \langle \lambda a. P \rangle$, *folded write0-def, simplified*]
for $a P$

lemma *deadlock-free-Mndetprefix-iff*: $\langle \text{deadlock-free} (\sqcap a \in A \rightarrow P a) \longleftrightarrow$
 $A \neq \{\} \wedge (\forall a \in A. \text{deadlock-free} (P a)) \rangle$
and *deadlock-free_{SKIP}-Mndetprefix-iff*: $\langle \text{deadlock-free}_{SKIP} (\sqcap a \in A \rightarrow P a)$
 \longleftrightarrow
 $A \neq \{\} \wedge (\forall a \in A. \text{deadlock-free}_{SKIP} (P a)) \rangle$
apply (*all* $\langle \text{intro iffI conjI} \rangle$)
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*: $\langle \text{deadlock-free} (P \sqcap Q) \longleftrightarrow$
 $\text{deadlock-free } P \wedge \text{deadlock-free } Q \rangle$
and *deadlock-free_{SKIP}-Ndet-iff*: $\langle \text{deadlock-free}_{SKIP} (P \sqcap Q) \longleftrightarrow$
 $\text{deadlock-free}_{SKIP} P \wedge \text{deadlock-free}_{SKIP} Q \rangle$
unfolding *deadlock-free-F deadlock-free_{SKIP}-def failure-refine-def*
by (*simp-all add: F-Ndet*)

lemma *deadlock-free-GlobalNdet-iff*: $\langle \text{deadlock-free} (\sqcap a \in A. P a) \longleftrightarrow$
 $A \neq \{\} \wedge (\forall a \in A. \text{deadlock-free} (P a)) \rangle$

and *deadlock-free_{SKIP}-GlobalNdet-iff*: $\langle \text{deadlock-free}_{SKIP} (\prod a \in A. P a) \longleftrightarrow$
 $A \neq \{\} \wedge (\forall a \in A. \text{deadlock-free}_{SKIP} (P a)) \rangle$
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*: $\langle \text{deadlock-free} (\prod a \in A. P a) \longleftrightarrow$
 $A \neq \{\} \wedge (\forall a \in A. \text{deadlock-free} (P a)) \rangle$
and *deadlock-free_{SKIP}-MultiNdet-iff*: $\langle \text{deadlock-free}_{SKIP} (\prod a \in A. P a) \longleftrightarrow$
 $A \neq \{\} \wedge (\forall a \in A. \text{deadlock-free}_{SKIP} (P a)) \rangle$
if *fin*: $\langle \text{finite } A \rangle$
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*:
 $\langle \llbracket A \neq \{\}; \text{finite } A; \forall a \in A. \text{deadlock-free} (P a) \rrbracket \Longrightarrow \text{deadlock-free} (\sqcap a \in A. P a) \rangle$
and *deadlock-free_{SKIP}-MultiDet*:
 $\langle \llbracket A \neq \{\}; \text{finite } A; \forall a \in A. \text{deadlock-free}_{SKIP} (P a) \rrbracket \Longrightarrow \text{deadlock-free}_{SKIP} (\sqcap a \in A. P a) \rangle$
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*:
 $\langle \text{deadlock-free } P \Longrightarrow \text{deadlock-free } Q \Longrightarrow \text{deadlock-free } (P \sqcap Q) \rangle$
and *deadlock-free_{SKIP}-Det*:
 $\langle \text{deadlock-free}_{SKIP} P \Longrightarrow \text{deadlock-free}_{SKIP} Q \Longrightarrow \text{deadlock-free}_{SKIP} (P \sqcap Q) \rangle$
by (*metis 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 $P \sqcap Q$, we can not expect more:

lemma
 $\langle \exists P Q. \text{deadlock-free } P \wedge \neg \text{deadlock-free } Q \wedge$
 $\text{deadlock-free } (P \sqcap Q) \rangle$
 $\langle \exists P Q. \text{deadlock-free}_{SKIP} P \wedge \neg \text{deadlock-free}_{SKIP} Q \wedge$
 $\text{deadlock-free}_{SKIP} (P \sqcap Q) \rangle$
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*:

$\langle A \neq \{\} \implies P \sqsubseteq_{FD} \sqcap a \in A \rightarrow Q \longleftrightarrow (\forall a \in A. P \sqsubseteq_{FD} (a \rightarrow Q)) \rangle$
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*: $\langle (\exists a \in A. (a \rightarrow Q) \sqsubseteq_{FD} P) \implies \sqcap a \in A \rightarrow Q \sqsubseteq_{FD} P \rangle$
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*: $\langle A \neq \{\} \vee B \neq \{\} \vee A' \cap B' \neq \{\} \rangle$
and $\langle A \cap S = \{\} \rangle$ **and** $\langle A' \subseteq S \rangle$ **and** $\langle B \cap S = \{\} \rangle$ **and** $\langle B' \subseteq S \rangle$
and $\langle \forall x \in A. \text{deadlock-free } (P \ x \llbracket S \rrbracket \text{ Mprefix } (B \cup B') \ Q) \rangle$
and $\langle \forall y \in B. \text{deadlock-free } (\text{Mprefix } (A \cup A') \ P \llbracket S \rrbracket \ Q \ y) \rangle$
and $\langle \forall x \in A' \cap B'. \text{deadlock-free } ((P \ x \llbracket S \rrbracket \ Q \ x)) \rangle$
shows $\langle \text{deadlock-free } (\text{Mprefix } (A \cup A') \ P \llbracket S \rrbracket \text{ Mprefix } (B \cup B') \ Q) \rangle$

proof –

have $\langle A = \{\} \wedge B \neq \{\} \wedge A' \cap B' \neq \{\} \vee A \neq \{\} \wedge B = \{\} \wedge A' \cap B' = \{\} \vee$
 $A \neq \{\} \wedge B = \{\} \wedge A' \cap B' \neq \{\} \vee A = \{\} \wedge B \neq \{\} \wedge A' \cap B' = \{\} \vee$
 $A \neq \{\} \wedge B \neq \{\} \wedge A' \cap B' = \{\} \vee A = \{\} \wedge B = \{\} \wedge A' \cap B' \neq \{\} \vee$
 $A \neq \{\} \wedge B \neq \{\} \wedge A' \cap B' \neq \{\} \rangle$ **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** $A = \langle \{\} \rangle$ **and** $B = \langle \{\} \rangle$, *simplified*]

and *Mprefix-Sync-indep-deadlock-free = Mprefix-Sync-deadlock-free*

[**where** $A' = \langle \{\} \rangle$ **and** $B' = \langle \{\} \rangle$, *simplified*]

and *Mprefix-Sync-right-deadlock-free = Mprefix-Sync-deadlock-free*

[**where** $A = \langle \{\} \rangle$ **and** $B' = \langle \{\} \rangle$, *simplified*]

and *Mprefix-Sync-left-deadlock-free = Mprefix-Sync-deadlock-free*

[**where** $A' = \langle \{\} \rangle$ **and** $B = \langle \{\} \rangle$, *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*: $\langle DF (f \text{ ' } A) \sqsubseteq_{FD} Renaming (DF A) f \rangle$
proof (*subst DF-def, induct rule: fix-ind*)
 show $\langle adm (\lambda a. a \sqsubseteq_{FD} Renaming (DF A) f) \rangle$ **by** (*simp add: monofun-def*)
next
 show $\langle \perp \sqsubseteq_{FD} Renaming (DF A) f \rangle$ **by** *simp*
next
 show $\langle (\Lambda x. \Pi a \in f \text{ ' } A \rightarrow x) \cdot x \sqsubseteq_{FD} Renaming (DF A) f \rangle$
 if $\langle x \sqsubseteq_{FD} Renaming (DF A) f \rangle$ **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*: $\langle Renaming (DF A) f \sqsubseteq_{FD} DF (f \text{ ' } A) \rangle$
 if *finitary*: $\langle finitary f \rangle$
proof (*subst DF-def, induct rule: fix-ind*)
 show $\langle adm (\lambda a. Renaming a f \sqsubseteq_{FD} DF (f \text{ ' } A)) \rangle$
 by (*simp add: finitary monofun-def*)
next
 show $\langle Renaming \perp f \sqsubseteq_{FD} DF (f \text{ ' } A) \rangle$ **by** (*simp add: Renaming-BOT*)
next
 show $\langle Renaming ((\Lambda x. \Pi a \in A \rightarrow x) \cdot x) f \sqsubseteq_{FD} DF (f \text{ ' } A) \rangle$
 if $\langle Renaming x f \sqsubseteq_{FD} DF (f \text{ ' } A) \rangle$ **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}*:
 $\langle DF_{SKIP} (f \text{ ' } A) \sqsubseteq_{FD} Renaming (DF_{SKIP} A) f \rangle$
proof (*subst DF_{SKIP}-def, induct rule: fix-ind*)
 show $\langle adm (\lambda a. a \sqsubseteq_{FD} Renaming (DF_{SKIP} A) f) \rangle$ **by** (*simp add: monofun-def*)
next
 show $\langle \perp \sqsubseteq_{FD} Renaming (DF_{SKIP} A) f \rangle$ **by** *simp*
next
 show $\langle (\Lambda x. (\Pi a \in f \text{ ' } A \rightarrow x) \sqcap SKIP) \cdot x \sqsubseteq_{FD} Renaming (DF_{SKIP} A) f \rangle$
 if $\langle x \sqsubseteq_{FD} Renaming (DF_{SKIP} A) f \rangle$ **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}*:

$\langle \text{Renaming } (DF_{SKIP} A) f \sqsubseteq_{FD} DF_{SKIP} (f ' A) \rangle$
if *finitary*: $\langle \text{finitary } f \rangle$
proof (*subst* DF_{SKIP} -def, *induct rule*: *fix-ind*)
show $\langle \text{adm } (\lambda a. \text{Renaming } a f \sqsubseteq_{FD} DF_{SKIP} (f ' A)) \rangle$
by (*simp add*: *finitary monofun-def*)
next
show $\langle \text{Renaming } \perp f \sqsubseteq_{FD} DF_{SKIP} (f ' A) \rangle$ **by** (*simp add*: *Renaming-BOT*)
next
show $\langle \text{Renaming } ((\Lambda x. (\Box a \in A \rightarrow x) \sqcap SKIP) \cdot x) f \sqsubseteq_{FD} DF_{SKIP} (f ' A) \rangle$
if $\langle \text{Renaming } x f \sqsubseteq_{FD} DF_{SKIP} (f ' A) \rangle$ **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*: $\langle \text{RUN } (f ' A) \sqsubseteq_{FD} \text{Renaming } (RUN A) f \rangle$
proof (*subst* *RUN-def*, *induct rule*: *fix-ind*)
show $\langle \text{adm } (\lambda a. a \sqsubseteq_{FD} \text{Renaming } (RUN A) f) \rangle$ **by** (*simp add*: *monofun-def*)
next
show $\langle \perp \sqsubseteq_{FD} \text{Renaming } (RUN A) f \rangle$ **by** *simp*
next
show $\langle (\Lambda x. \Box a \in f ' A \rightarrow x) \cdot x \sqsubseteq_{FD} \text{Renaming } (RUN A) f \rangle$
if $\langle x \sqsubseteq_{FD} \text{Renaming } (RUN A) f \rangle$ **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*: $\langle \text{Renaming } (RUN A) f \sqsubseteq_{FD} \text{RUN } (f ' A) \rangle$
if *finitary*: $\langle \text{finitary } f \rangle$
proof (*subst* *RUN-def*, *induct rule*: *fix-ind*)
show $\langle \text{adm } (\lambda a. \text{Renaming } a f \sqsubseteq_{FD} \text{RUN } (f ' A)) \rangle$
by (*simp add*: *finitary monofun-def*)
next
show $\langle \text{Renaming } \perp f \sqsubseteq_{FD} \text{RUN } (f ' A) \rangle$ **by** (*simp add*: *Renaming-BOT*)
next
show $\langle \text{Renaming } ((\Lambda x. \Box a \in A \rightarrow x) \cdot x) f \sqsubseteq_{FD} \text{RUN } (f ' A) \rangle$
if $\langle \text{Renaming } x f \sqsubseteq_{FD} \text{RUN } (f ' A) \rangle$ **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*:
 $\langle \text{CHAOS } (f \text{ ' } A) \sqsubseteq_{FD} \text{Renaming } (\text{CHAOS } A) f \rangle$
proof (*subst CHAOS-def, induct rule: fix-ind*)
show $\langle \text{adm } (\lambda a. a \sqsubseteq_{FD} \text{Renaming } (\text{CHAOS } A) f) \rangle$ **by** (*simp add: monofun-def*)
next
show $\langle \perp \sqsubseteq_{FD} \text{Renaming } (\text{CHAOS } A) f \rangle$ **by** *simp*
next
show $\langle (\Lambda x. \text{STOP} \sqcap (\Box a \in f \text{ ' } A \rightarrow x)) \cdot x \sqsubseteq_{FD} \text{Renaming } (\text{CHAOS } A) f \rangle$
if $\langle x \sqsubseteq_{FD} \text{Renaming } (\text{CHAOS } A) f \rangle$ **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*:
 $\langle \text{Renaming } (\text{CHAOS } A) f \sqsubseteq_{FD} \text{CHAOS } (f \text{ ' } A) \rangle$
if *finitary*: $\langle \text{finitary } f \rangle$
proof (*subst CHAOS-def, induct rule: fix-ind*)
show $\langle \text{adm } (\lambda a. \text{Renaming } a f \sqsubseteq_{FD} \text{CHAOS } (f \text{ ' } A)) \rangle$
by (*simp add: finitary monofun-def*)
next
show $\langle \text{Renaming } \perp f \sqsubseteq_{FD} \text{CHAOS } (f \text{ ' } A) \rangle$ **by** (*simp add: Renaming-BOT*)
next
show $\langle \text{Renaming } ((\Lambda x. \text{STOP} \sqcap (\Box xa \in A \rightarrow x)) \cdot x) f \sqsubseteq_{FD} \text{CHAOS } (f \text{ ' } A) \rangle$
if $\langle \text{Renaming } x f \sqsubseteq_{FD} \text{CHAOS } (f \text{ ' } A) \rangle$ **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}*:
 $\langle \text{CHAOS}_{SKIP} (f \text{ ' } A) \sqsubseteq_{FD} \text{Renaming } (\text{CHAOS}_{SKIP} A) f \rangle$
proof (*subst CHAOS_{SKIP}-def, induct rule: fix-ind*)
show $\langle \text{adm } (\lambda a. a \sqsubseteq_{FD} \text{Renaming } (\text{CHAOS}_{SKIP} A) f) \rangle$
by (*simp add: monofun-def*)
next
show $\langle \perp \sqsubseteq_{FD} \text{Renaming } (\text{CHAOS}_{SKIP} A) f \rangle$ **by** *simp*
next
show $\langle (\Lambda x. \text{SKIP} \sqcap \text{STOP} \sqcap (\Box xa \in f \text{ ' } A \rightarrow x)) \cdot x \sqsubseteq_{FD} \text{Renaming } (\text{CHAOS}_{SKIP} A) f \rangle$
if $\langle x \sqsubseteq_{FD} \text{Renaming } (\text{CHAOS}_{SKIP} A) f \rangle$ **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}*:
 $\langle \text{Renaming } (CHAOS_{SKIP} A) f \sqsubseteq_{FD} CHAOS_{SKIP} (f ' A) \rangle$
if *finitary*: $\langle \text{finitary } f \rangle$
proof (*subst CHAOS_{SKIP}-def, induct rule: fix-ind*)
show $\langle \text{adm } (\lambda a. \text{Renaming } a f \sqsubseteq_{FD} CHAOS_{SKIP} (f ' A)) \rangle$
by (*simp add: finitary monofun-def*)
next
show $\langle \text{Renaming } \perp f \sqsubseteq_{FD} CHAOS_{SKIP} (f ' A) \rangle$ **by** (*simp add: Renaming-BOT*)
next
show $\langle \text{Renaming } ((\Lambda x. SKIP \sqcap STOP \sqcap (\Box xa \in A \rightarrow x)) \cdot x) f \sqsubseteq_{FD} CHAOS_{SKIP} (f ' A) \rangle$
if $\langle \text{Renaming } x f \sqsubseteq_{FD} CHAOS_{SKIP} (f ' A) \rangle$ **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 =*
 $FD\text{-antisym}[OF \text{Renaming-DF-FD-DF } DF\text{-FD-Renaming-DF}]$
and *Renaming-DF_{SKIP} =*
 $FD\text{-antisym}[OF \text{Renaming-DF}_{SKIP}\text{-FD-DF}_{SKIP} \text{DF}_{SKIP}\text{-FD-Renaming-DF}_{SKIP}]$
and *Renaming-RUN =*
 $FD\text{-antisym}[OF \text{Renaming-RUN-FD-RUN } RUN\text{-FD-Renaming-RUN}]$
and *Renaming-CHAOS =*
 $FD\text{-antisym}[OF \text{Renaming-CHAOS-FD-CHAOS } CHAOS\text{-FD-Renaming-CHAOS}]$
and *Renaming-CHAOS_{SKIP} =*
 $FD\text{-antisym}[OF \text{Renaming-CHAOS}_{SKIP}\text{-FD-CHAOS}_{SKIP} \text{CHAOS}_{SKIP}\text{-FD-Renaming-CHAOS}_{SKIP}]$

lemma *deadlock-free-imp-deadlock-free-Renaming*: $\langle \text{deadlock-free } (\text{Renaming } P f) \rangle$
if $\langle \text{deadlock-free } P \rangle$
apply (*rule DF-Univ-freeness[of $\langle \text{range } f \rangle$, 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*: $\langle \text{deadlock-free } P \rangle$
if $\langle \text{inj } f \rangle$ **and** $\langle \text{deadlock-free } (\text{Renaming } P f) \rangle$
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*:
 $\langle \text{inj } f \implies \text{deadlock-free } (\text{Renaming } P f) \iff \text{deadlock-free } P \rangle$
using *deadlock-free-Renaming-imp-deadlock-free*
deadlock-free-imp-deadlock-free-Renaming **by** *blast*

lemma *deadlock-free_{SKIP}-imp-deadlock-free_{SKIP}-Renaming*:
 $\langle \text{deadlock-free}_{\text{SKIP}} (\text{Renaming } P f) \rangle$ **if** $\langle \text{deadlock-free}_{\text{SKIP}} P \rangle$
unfolding *deadlock-free_{SKIP}-FD*
apply (*rule trans-FD*[*OF DF_{SKIP}-subset-FD*[*of* $\langle \text{range } f \rangle$]], *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}*:
 $\langle \text{deadlock-free}_{\text{SKIP}} P \rangle$ **if** $\langle \text{inj } f \rangle$ **and** $\langle \text{deadlock-free}_{\text{SKIP}} (\text{Renaming } P f) \rangle$
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*:
 $\langle \text{inj } f \implies \text{deadlock-free}_{\text{SKIP}} (\text{Renaming } P f) \iff \text{deadlock-free}_{\text{SKIP}} P \rangle$
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*:
 $\langle (\forall P Q. \text{deadlock-free } (P :: 'a \text{ process}) \longrightarrow \text{deadlock-free } Q \longrightarrow$
 $\text{deadlock-free } (P \llbracket S \rrbracket Q))$
 $\iff (DF (UNIV :: 'a \text{ set}) \sqsubseteq_{FD} (DF UNIV \llbracket S \rrbracket DF UNIV)) \rangle$ (**is** $\langle ?lhs \iff ?rhs \rangle$)

proof (*rule iffI*)
assume *?lhs*
show *?rhs* **by** (*fold deadlock-free-def*, *rule* $\langle ?lhs \rangle$ [*rule-format*])
(simp-all add: deadlock-free-def)
next
assume *?rhs*
show *?lhs* **unfolding** *deadlock-free-def*
by (*intro allI impI trans-FD*[*OF* $\langle ?rhs \rangle$]) (*rule mono-Sync-FD*)
qed

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

13.4.2 STOP and SKIP Synchronized with DF A

lemma *DF-FD-DF-Sync-STOP-or-SKIP-iff*:

$\langle (DF A \sqsubseteq_{FD} DF A \llbracket S \rrbracket P) \longleftrightarrow A \cap S = \{\} \rangle$
 if *P-disj*: $\langle P = STOP \vee P = SKIP \rangle$

proof

{ assume *a1*: $\langle DF A \sqsubseteq_{FD} DF A \llbracket S \rrbracket P \rangle$ **and** *a2*: $\langle A \cap S \neq \{\} \rangle$
from *a2* **obtain** *x* **where** *f1*: $\langle x \in A \rangle$ **and** *f2*: $\langle x \in S \rangle$ **by** *blast*
have $\langle DF A \llbracket S \rrbracket P \sqsubseteq_{FD} DF \{x\} \llbracket S \rrbracket P \rangle$
by (*intro mono-Sync-FD*[*OF - idem-FD*]) (*simp add: DF-subset f1*)
also have $\langle \dots = STOP \rangle$
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)[*β*])
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 $\langle DF A \sqsubseteq_{FD} DF A \llbracket S \rrbracket P \implies A \cap S = \{\} \rangle$ **by** *blast*

next

have *D-P*: $\langle \mathcal{D} P = \{\} \rangle$ **using** *D-SKIP D-STOP P-disj* **by** *blast*
note *F-T-P* = *F-STOP T-STOP F-SKIP D-SKIP*

{ assume *a1*: $\langle \neg DF A \sqsubseteq_{FD} DF A \llbracket S \rrbracket P \rangle$ **and** *a2*: $\langle A \cap S = \{\} \rangle$
have *False*

proof (*cases* $\langle A = \{\} \rangle$)

assume $\langle A = \{\} \rangle$

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*: $\langle A \neq \{\} \rangle$

have *falsify*: $\langle (a, (X \cup Y) \cap \text{insert tick } (ev \text{ ' } S) \cup X \cap Y) \notin \mathcal{F} (DF A) \implies$
 $(t, X) \in \mathcal{F} (DF A) \implies (u, Y) \in \mathcal{F} P \implies$
 $a \text{ setinterleaves } ((t, u), \text{insert tick } (ev \text{ ' } S)) \implies \text{False} \rangle$ **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*(*t*) **have** *f1*: $\langle u = [] \rangle$

```

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.prem(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  $\langle A \cap S = \{\} \implies DF A \sqsubseteq_{FD} DF A \llbracket S \rrbracket P \rangle$  by blast
qed

```

```

lemma DF-Sync-STOP-or-SKIP-FD-DF:  $\langle DF A \llbracket S \rrbracket P \sqsubseteq_{FD} DF A \rangle$ 
if P-disj:  $\langle P = STOP \vee P = SKIP \rangle$  and empty-inter:  $\langle A \cap S = \{\} \rangle$ 
proof (cases  $\langle A = \{\} \rangle$ )
from P-disj show  $\langle A = \{\} \implies DF A \llbracket S \rrbracket P \sqsubseteq_{FD} DF A \rangle$ 
by (rule disjE) (simp-all add: DF-def fix-const Sync-SKIP-STOP
      Sync-STOP-STOP Sync-commute)
next
assume  $\langle A \neq \{\} \rangle$ 
show ?thesis
proof (subst DF-def, induct rule: fix-ind)
show  $\langle adm (\lambda a. a \llbracket S \rrbracket P \sqsubseteq_{FD} DF A) \rangle$  by (simp add: cont2mono)
next
show  $\langle \perp \llbracket S \rrbracket P \sqsubseteq_{FD} DF A \rangle$  by (metis BOT-leFD Sync-BOT Sync-commute)
next
case (3 x)
have  $\langle (\bigwedge a \in A \rightarrow x) \llbracket S \rrbracket P \sqsubseteq_{FD} (a \rightarrow DF A) \rangle$  if  $\langle a \in A \rangle$  for a
apply (rule trans-FD[OF mono-Sync-FD
      [OF mono-Mndetprefix-FD-set
      [of  $\langle \{a \} \rangle$ , simplified, OF that] idem-FD]])
apply (rule disjE[OF P-disj], simp-all add: Mndetprefix-unit)
apply (subst Mprefix-Sync-distr-left
      [of  $\langle \{a \} \rangle - \langle \{\} \rangle \langle \lambda a. x \rangle$ ,
      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:  $\langle A \neq \{\} \rangle$ )
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 $DF-F-DF-Sync-DF: \langle DF (A \cup B::'\alpha \text{ set}) \sqsubseteq_F DF A \llbracket S \rrbracket DF B \rangle$
if $nonempty: \langle A \neq \{\} \wedge B \neq \{\} \rangle$
and $intersect-hyp: \langle B \cap S = \{\} \vee (\exists y. B \cap S = \{y\} \wedge A \cap S \subseteq \{y\}) \rangle$
unfolding $failure-refine-def$ **apply** ($simp \text{ add: } F-Sync \text{ div-free-DF, safe}$)

proof –

fix $v \ t \ u \ X \ Y$

assume $*$: $\langle (t, X) \in \mathcal{F} (DF A) \rangle \langle (u, Y) \in \mathcal{F} (DF B) \rangle$

$\langle v \text{ setinterleaves } ((t, u), \text{insert tick } (ev \ 'S)) \rangle$

define β **where** $\langle \beta \equiv (t, \text{insert tick } (ev \ 'S), u) \rangle$

with $*$ **have** $\langle fst \ \beta, X \rangle \in \mathcal{F} (DF A) \rangle \langle snd \ (snd \ \beta), Y \rangle \in \mathcal{F} (DF B) \rangle$

$\langle v \in \text{setinterleaving } \beta \rangle$ **by** $simp\text{-all}$

thus $\langle (v, (X \cup Y) \cap \text{insert tick } (ev \ 'S) \cup X \cap Y) \in \mathcal{F} (DF (A \cup B)) \rangle$

apply ($subst \ F-DF, simp \ \text{add: } nonempty$)

proof ($induct \ \text{arbitrary: } v \ \text{rule: } \text{setinterleaving.induct}$)

case ($1 \ Z$)

hence $mt\text{-a: } \langle v = [] \rangle$ **using** $emptyLeftProperty$ **by** $blast$

from $intersect-hyp$

consider $\langle B \cap S = \{\} \rangle \mid \langle \exists y. B \cap S = \{y\} \wedge A \cap S \subseteq \{y\} \rangle$ **by** $blast$

thus $?case$

proof $cases$

case $11: 1$

with $1(2)$ **show** $?thesis$ **by** ($subst \ (asm) \ F-DF$)

($auto \ simp \ \text{add: } nonempty \ mt\text{-a}$)

next

case $12: 2$

then obtain y **where** $f12: \langle B \cap S = \{y\} \rangle$ **and** $\langle A \cap S \subseteq \{y\} \rangle$ **by** $blast$

from $this(2)$ **consider** $\langle A \cap S = \{\} \rangle \mid \langle A \cap S = \{y\} \rangle$ **by** $blast$

thus $?thesis$

proof $cases$

case $121: 1$

with $1(1)$ **show** $?thesis$ **by** ($subst \ (asm) \ F-DF$)

($auto \ simp \ \text{add: } nonempty \ mt\text{-a}$)

next

case $122: 2$

with $1(1, 2)$ $f12$ $nonempty \ mt\text{-a} \ mk\text{-disjoint-insert}$ **show** $?thesis$

```

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

case (2 Z y u)
have * : ⟨y ∉ Z⟩ ⟨([], X) ∈  $\mathcal{F}$  (DF A)⟩ ⟨(u, Y) ∈  $\mathcal{F}$  (DF B)⟩ ⟨v = y # u⟩
  using 2.prem1 Cons-F-DF by (auto simp add: emptyLeftProperty)
have ** : ⟨u setinterleaves (([], u), Z)⟩
  by (metis *(4) 2.prem3 SyncTLEmpty list.sel(3))
from 2.prem2 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) ∈  $\mathcal{F}$  (DF A)⟩ ⟨([], Y) ∈  $\mathcal{F}$  (DF B)⟩ ⟨v = x # t⟩
  using 3.prem1 Cons-F-DF by (auto simp add: Sync.sym emptyLeftProperty)
have ** : ⟨t setinterleaves ((t, []), Z)⟩
  by (metis *(4) 3.prem3 Sync.sym SyncTLEmpty list.sel(3))
from 3.prem2 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) ∈  $\mathcal{F}$  (DF A)⟩
    ⟨(u, Y) ∈  $\mathcal{F}$  (DF B)⟩ ⟨v = x # v'⟩
  using 4.prem1 Cons-F-DF by (simp add: hyps split: if-split-asm) blast
  have ** : ⟨v' setinterleaves ((t, u), Z)⟩
    using *(1, 4) 4.prem3 hyps(1) by force
  from 4.prem2 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:  $\langle x \in Z \rangle \langle y \notin Z \rangle$ 
obtain  $v'$  where  $*$  :  $\langle (x \# t, X) \in \mathcal{F} (DF A) \rangle \langle (u, Y) \in \mathcal{F} (DF B) \rangle$ 
       $\langle v = y \# v' \rangle \langle v' \text{ setinterleaves } ((x \# t, u), Z) \rangle$ 
  using  $4.\text{prems } Cons\text{-}F\text{-}DF$  by (simp add: hyps split: if-split-asm) blast
from  $4.\text{prems}(2)$   $4.\text{hyps}(2)$  [simplified, OF hyps *(1, 2, 4)]
show  $?case$  by (subst (asm) F-DF, subst F-DF)
      (auto simp add: nonempty *(3))

next
assume hyps:  $\langle x \notin Z \rangle \langle y \in Z \rangle$ 
obtain  $v'$  where  $*$  :  $\langle (t, X) \in \mathcal{F} (DF A) \rangle \langle (y \# u, Y) \in \mathcal{F} (DF B) \rangle$ 
       $\langle v = x \# v' \rangle \langle v' \text{ setinterleaves } ((t, y \# u), Z) \rangle$ 
  using  $4.\text{prems } Cons\text{-}F\text{-}DF$  by (simp add: hyps split: if-split-asm) blast
from  $4.\text{prems}(1)$   $4.\text{hyps}(5)$  [simplified, OF hyps *(1, 2, 4)]
show  $?case$  by (subst (asm) F-DF, subst F-DF)
      (auto simp add: nonempty *(3))

next
assume hyps:  $\langle x \notin Z \rangle \langle y \notin Z \rangle$ 
note  $f_4 = 4$  [simplified, simplified hyps, simplified]
from  $f_4(8)$  obtain  $v' v''$ 
  where  $\langle v = x \# v' \wedge v' \text{ setinterleaves } ((t, y \# u), Z) \vee$ 
       $v = y \# v'' \wedge v'' \text{ setinterleaves } ((x \# t, u), Z) \rangle$ 
      (is  $\langle ?left \vee ?right \rangle$ ) by blast
then consider  $\langle ?left \rangle \mid \langle ?right \rangle$  by fast
then show  $?thesis$ 
proof cases
  assume  $\langle ?left \rangle$ 
from  $f_4(6)$   $f_4(3)$  [OF f_4(6) [THEN Cons-F-DF] f_4(7)]
       $\langle ?left \rangle$  [THEN conjunct2]
show  $?thesis$  by (subst (asm) F-DF, subst F-DF)
      (auto simp add:  $\langle ?left \rangle$  nonempty)

  next
  assume  $\langle ?right \rangle$ 
from  $f_4(7)$   $f_4(4)$  [OF f_4(6) f_4(7) [THEN Cons-F-DF]]
       $\langle ?right \rangle$  [THEN conjunct2]
show  $?thesis$  by (subst (asm) F-DF, subst F-DF)
      (auto simp add:  $\langle ?right \rangle$  nonempty)

qed
qed
qed
qed

```

lemma *DF-FD-DF-Sync-DF*:

$\langle A \neq \{\} \wedge B \neq \{\} \implies B \cap S = \{\} \vee (\exists y. B \cap S = \{y\} \wedge A \cap S \subseteq \{y\}) \implies$
 $DF (A \cup B) \sqsubseteq_{FD} DF A \llbracket S \rrbracket DF B \rangle$

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*:
 $\langle DF (A \cup B) \sqsubseteq_{FD} DF A \llbracket S \rrbracket DF B \longleftrightarrow$
 (if $A = \{\}$ then $B \cap S = \{\}$
 else if $B = \{\}$ then $A \cap S = \{\}$
 else $A \cap S = \{\} \vee (\exists a. A \cap S = \{a\} \wedge B \cap S \subseteq \{a\}) \vee$
 $B \cap S = \{\} \vee (\exists b. B \cap S = \{b\} \wedge A \cap S \subseteq \{b\}) \rangle$
 (is $\langle ?FD-ref \longleftrightarrow$ (if $A = \{\}$ then $B \cap S = \{\}$
 else if $B = \{\}$ then $A \cap S = \{\}$
 else $?cases \rangle$)

proof –

{ **assume** $\langle A \neq \{\} \rangle$ **and** $\langle B \neq \{\} \rangle$ **and** $?FD-ref$ **and** $\langle \neg ?cases \rangle$
from $\langle \neg ?cases \rangle$ *[simplified]*
obtain a **and** b **where** $\langle a \in A \rangle \langle a \in S \rangle \langle b \in B \rangle \langle b \in S \rangle \langle a \neq b \rangle$ **by** *blast*
have $\langle DF A \llbracket S \rrbracket DF B \sqsubseteq_{FD} (a \rightarrow DF A) \llbracket S \rrbracket (b \rightarrow DF B) \rangle$
by (*intro mono-Sync-FD; subst DF-unfold,*
subst Mndetprefix-unit[symmetric], simp add: $\langle a \in A \rangle \langle b \in B \rangle$)
also have $\langle \dots = STOP \rangle$ **by** (*simp add: $\langle a \in S \rangle \langle a \neq b \rangle \langle b \in S \rangle$ prefix-Sync1*)
finally have *False*
by (*metis DF-Univ-freeness Un-empty $\langle A \neq \{\} \rangle$*
trans-FD[OF $\langle ?FD-ref \rangle$] non-deadlock-free-STOP)
 }
thus *?thesis*
apply (*cases $\langle A = \{\} \rangle$, simp,*
metis DF-FD-DF-Sync-STOP-iff DF-unfold Sync-commute mt-Mndetprefix)
apply (*cases $\langle B = \{\} \rangle$, 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

$\langle (\forall a \in A. X a \cap S = \{\} \vee (\forall b \in A. \exists y. X a \cap S = \{y\} \wedge X b \cap S \subseteq \{y\}))$
 $\longleftrightarrow (\forall a \in A. \forall b \in A. \exists y. (X a \cup X b) \cap S \subseteq \{y\}) \rangle$

— 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*:

$\langle DF (\bigcup x \in (\text{insert } a \ A). X x) \sqsubseteq_{FD} \llbracket S \rrbracket x \in \# \text{ mset-set } (\text{insert } a \ A). DF (X x) \rangle$
if *fin*: $\langle \text{finite } A \rangle$ **and** *nonempty*: $\langle X a \neq \{\} \rangle \langle \forall b \in A. X b \neq \{\} \rangle$
and *ugly-hyp*: $\langle \forall b \in A. X b \cap S = \{\} \vee (\exists y. X b \cap S = \{y\} \wedge X a \cap S \subseteq \{y\}) \rangle$
 $\langle \forall b \in A. \forall c \in A. \exists y. (X b \cup X c) \cap S \subseteq \{y\} \rangle$

proof –

have $\langle DF (\bigcup (X \text{ 'insert } a \ A)) \sqsubseteq_{FD} (\llbracket S \rrbracket x \in \# \text{ mset-set } (\text{insert } a \ A). DF (X x)) \rangle \wedge$

$(\forall b \in A. X b \cap S = \{\} \vee (\exists y. X b \cap S = \{y\} \wedge \bigcup (X \text{ 'insert a A}) \cap S \subseteq \{y\}))$

— 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 $\langle DF (X b \cup \bigcup (X \text{ 'insert a A}')) \sqsubseteq_{FD}$

$\llbracket S \rrbracket a \in \# \text{mset-set (insert b (insert a A}')) \rangle. DF (X a) \rangle$

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 $\langle \forall c \in A. X c \cap S = \{\} \vee (\exists y. X c \cap S = \{y\} \wedge$

$(X b \cup \bigcup (X \text{ 'insert a A}')) \cap S \subseteq \{y\} \rangle$

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'*:

$\langle \llbracket \text{finite } A; \forall a \in A. X a \neq \{\}; \forall a \in A. \forall b \in A. \exists y. (X a \cup X b) \cap S \subseteq \{y\} \rrbracket$

$\implies DF (\bigcup a \in A. X a) \sqsubseteq_{FD} \llbracket S \rrbracket a \in \# \text{mset-set } A. DF (X a) \rangle$

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 $S = \langle \{\} \rangle$, *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 $\langle DF \{a\} = DF \{a\} \llbracket S \rrbracket STOP \longleftrightarrow a \notin S \rangle$
by (*metis* $DF-FD-DF-Sync-STOP$ -iff $DF-Sync-STOP-FD-DF$ *Diff-disjoint*
 $Diff-insert-absorb$ $FD-antisym$ *insert-disjoint*(2))

lemma $\langle DF \{a\} \llbracket S \rrbracket STOP = STOP \longleftrightarrow a \in S \rangle$
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-SKIP*2)

corollary $DF-FD-DF-Inter-DF$: $\langle DF (A::'\alpha \text{ set}) \sqsubseteq_{FD} DF A \parallel DF A \rangle$
by (*metis* $DF-FD-DF-Sync-DF$ -iff *inf-bot-right* *sup.idem*)

corollary $DF-UNIV-FD-DF-UNIV-Inter-DF-UNIV$:
 $\langle DF UNIV \sqsubseteq_{FD} DF UNIV \parallel DF UNIV \rangle$
by (*fact* $DF-FD-DF-Inter-DF$)

corollary *Inter-deadlock-free*:
 $\langle \text{deadlock-free } P \implies \text{deadlock-free } Q \implies \text{deadlock-free } (P \parallel Q) \rangle$
using $DF-FD-DF-Inter-DF$ *deadlock-free-of-Sync-iff-DF-FD-DF-Sync-DF* **by** *blast*

theorem *MultiInter-deadlock-free*:
 $\langle M \neq \{\#\} \implies \forall p \in \# M. \text{deadlock-free } (P p) \implies$
 $\text{deadlock-free } (\parallel p \in \# M. P p) \rangle$

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 $P \sqcap Q$, $P \sqcap Q$, $P \llbracket S \rrbracket Q$, and $P ; Q$.

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

- (\sqcap) is commutative, idempotent and admits *STOP* as a neutral element so we defined *MultiDet* on a *finite 'α set* A by making it equal to *STOP* when $A = \emptyset$.
- (\sqcap) is also commutative and idempotent so we defined *MultiNdet* on a *finite 'α set* A by making it equal to *STOP* when $A = \emptyset$. Beware of the fact that *STOP* is not the neutral element for (\sqcap) (this operator does not admit a neutral element) so we **do not have** the equality

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

while this holds for (\sqcap) and *MultiDet*.

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

$$\text{finite } A \implies \sqcap_{p \in A} P p = \sqcap_{p \in A} P p$$

- *Sync* is commutative but is not idempotent so we defined *MultiSync* on a *'α 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

$$\llbracket S \rrbracket p \in \#\{a\#\}. P p \neq P a \llbracket S \rrbracket (\llbracket S \rrbracket p \in \#\{a\#\}. P p)$$

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

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

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

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 :

$$\llbracket M \neq \{\#\}; \forall p \in \#M. \text{deadlock-free} (P\ p) \rrbracket \implies \text{deadlock-free} (\lll p \in \#M. P\ p \rrl)$$

This theorem allows, for example, to establish:

$$0 < n \implies \text{deadlock-free} (\lll i \in \#mset [0..<n]. P\ i \rrl)$$

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

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.