HOL-CSPM - Architectural operators for HOL-CSP

Benoît Ballenghien Safouan Taha Burkhart Wolff

March 17, 2025

Abstract

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

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

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

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

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

Contents

Chapter 1

Introduction

1.1 Motivations

HOL-CSP [4] is a formalization in Isabelle/HOL of the work of Hoare and Roscoe on the denotational semantics of the Failure/Divergence Model of CSP. It follows essentially the presentation of CSP in Roscoe's Book "Theory and Practice of Concurrency" [2] and the semantic details in a joint Paper of Roscoe and Brooks "An improved failures model for communicating processes" [1].

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

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

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

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

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

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

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

eralizing $P \llbracket S \rrbracket Q$ with the two special cases $|||\ m \in\#\ M.\ P\ m$ and $||\ m \in\#\ M.\ P\ m$

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

We also provide the definitions of the POTS and Dining Philosophers examples, which greatly benefit from the newly introduced generalized operators.

Since they appear naturally when modeling complex architectures, we may call them *architectural operators*: these multi-operators represent the heart of the architectural composition principles of CSP.

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

In several places, our formalization efforts led to slight modifications of the original definitions in order to achieve the goal of a combined integrated theory. In some cases – in particular in connection with the *Interrupt* operator definition – some corrections have been necessary since the fundamental invariants were not respected.

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

1.2 The Global Architecture of HOL-CSPM

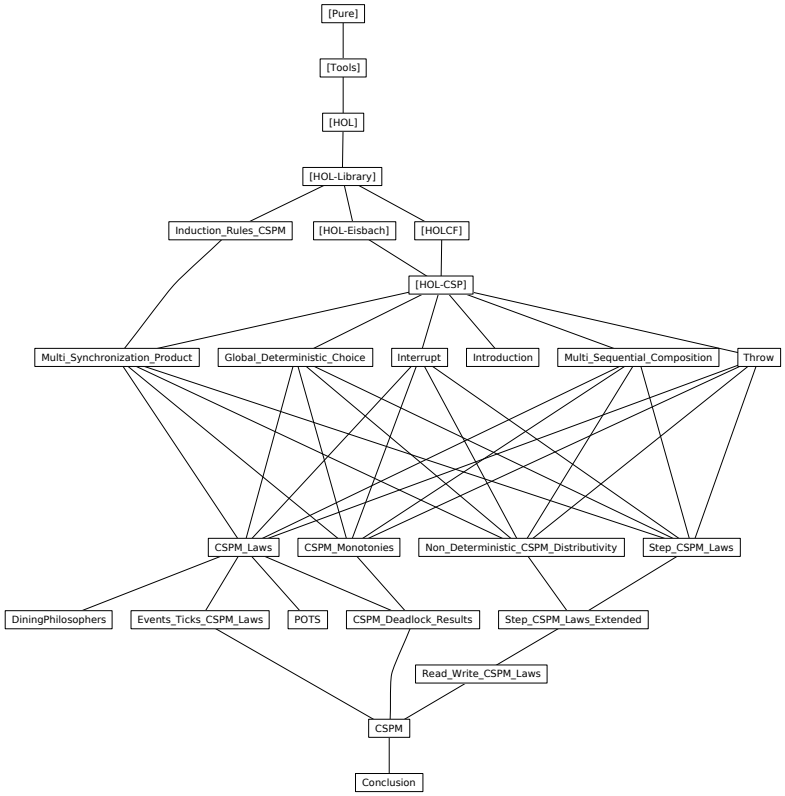

Figure 1.1: The overall architecture

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

1. `HOL-Eisbach` from the Isabelle/HOL distribution,
2. `HOLCF` from the Isabelle/HOL distribution, and
3. `HOL-CSP` 2.0 from the Isabelle Archive of Formal Proofs.

Chapter 2

Preliminary Work

2.1 Induction Rules for *'a set*

lemma *finite-subset-induct-singleton*
[*consumes 3*, *case-names singleton insertion*]:
‹⟦$a \in A$; *finite* F; $F \subseteq A$; $P\ \{a\}$;
⋀$x\ F$. *finite* $F \Longrightarrow x \in A \Longrightarrow x \notin (\mathit{insert}\ a\ F) \Longrightarrow P\ (\mathit{insert}\ a\ F)$
$\Longrightarrow P\ (\mathit{insert}\ x\ (\mathit{insert}\ a\ F))$⟧ $\Longrightarrow P\ (\mathit{insert}\ a\ F)$›
⟨*proof*⟩

lemma *finite-set-induct-nonempty*
[*consumes 2*, *case-names singleton insertion*]:
assumes ‹$A \neq \{\}$› **and** ‹*finite* A›
and *singleton*: ‹⋀a. $a \in A \Longrightarrow P\ \{a\}$›
and *insert*: ‹⋀$x\ F$. ⟦$F \neq \{\}$; *finite* F; $x \in A$; $x \notin F$; $P\ F$⟧
$\Longrightarrow P\ (\mathit{insert}\ x\ F)$›
shows ‹$P\ A$›
⟨*proof*⟩

lemma *finite-subset-induct-singleton'*
[*consumes 3*, *case-names singleton insertion*]:
‹⟦$a \in A$; *finite* F; $F \subseteq A$; $P\ \{a\}$;
⋀$x\ F$. ⟦*finite* F; $x \in A$; *insert* $a\ F \subseteq A$; $x \notin \mathit{insert}\ a\ F$; $P\ (\mathit{insert}\ a\ F)$⟧
$\Longrightarrow P\ (\mathit{insert}\ x\ (\mathit{insert}\ a\ F))$⟧
$\Longrightarrow P\ (\mathit{insert}\ a\ F)$›
⟨*proof*⟩

lemma *induct-subset-empty-single*[*consumes 1*]:
‹⟦*finite* A; $P\ \{\}$; ⋀a. $a \in A \Longrightarrow P\ \{a\}$;
⋀$F\ a$. ⟦$a \in A$; $a \notin F$; *finite* F; $F \subseteq A$; $F \neq \{\}$; $P\ F$⟧ $\Longrightarrow P\ (\mathit{insert}\ a\ F)$⟧ $\Longrightarrow$
$P\ A$›
⟨*proof*⟩

2.2 Induction Rules for *'α multiset*

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

lemma *msubset-induct* [*consumes 1*, *case-names empty add*]:
‹⟦$F \subseteq\# A$; $P\ \{\#\}$; $\bigwedge a\ F.\ ⟦a \in\# A;\ P\ F⟧ \Longrightarrow P\ (\text{add-mset}\ a\ F)⟧ \Longrightarrow P\ F$›
⟨*proof*⟩

lemma *msubset-induct-singleton* [*consumes 2*, *case-names m-singleton add*]:
‹⟦$a \in\# A$; $F \subseteq\# A$; $P\ \{\#a\#\}$;
$\bigwedge x\ F.\ ⟦x \in\# A;\ P\ (\text{add-mset}\ a\ F)⟧ \Longrightarrow P\ (\text{add-mset}\ x\ (\text{add-mset}\ a\ F))⟧$
$\Longrightarrow P\ (\text{add-mset}\ a\ F)$›
⟨*proof*⟩

lemma *mset-induct-nonempty* [*consumes 1*, *case-names m-singleton add*]:
assumes ‹$A \neq \{\#\}$›
and *m-singleton*: ‹$\bigwedge a.\ a \in\# A \Longrightarrow P\ \{\#a\#\}$›
and *add*: ‹$\bigwedge x\ F.\ ⟦F \neq \{\#\};\ x \in\# A;\ P\ F⟧ \Longrightarrow P\ (\text{add-mset}\ x\ F)$›
shows ‹$P\ A$›
⟨*proof*⟩

lemma *msubset-induct'* [*consumes 2*, *case-names empty add*]:
assumes ‹$F \subseteq\# A$›
and *empty*: ‹$P\ \{\#\}$›
and *insert*: ‹$\bigwedge a\ F.\ ⟦a \in\# A - F;\ F \subseteq\# A;\ P\ F⟧ \Longrightarrow P\ (\text{add-mset}\ a\ F)$›
shows ‹$P\ F$›
⟨*proof*⟩

lemma *msubset-induct-singleton'* [*consumes 2*, *case-names m-singleton add*]:
‹⟦$a \in\# A - F$; $F \subseteq\# A$; $P\ \{\#a\#\}$;
$\bigwedge x\ F.\ ⟦x \in\# A - F;\ F \subseteq\# A;\ P\ (\text{add-mset}\ a\ F)⟧$
$\Longrightarrow P\ (\text{add-mset}\ x\ (\text{add-mset}\ a\ F))⟧$
$\Longrightarrow P\ (\text{add-mset}\ a\ F)$›
⟨*proof*⟩

lemma *msubset-induct-singleton''* [*consumes 1*, *case-names m-singleton add*]:
‹⟦$\text{add-mset}\ a\ F \subseteq\# A$; $P\ \{\#a\#\}$;
$\bigwedge x\ F.\ ⟦\text{add-mset}\ x\ (\text{add-mset}\ a\ F) \subseteq\# A;\ P\ (\text{add-mset}\ a\ F)⟧$
$\Longrightarrow P\ (\text{add-mset}\ x\ (\text{add-mset}\ a\ F))⟧$
$\Longrightarrow P\ (\text{add-mset}\ a\ F)$›
⟨*proof*⟩

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

assumes *nonempty*: ‹$A \neq \{\#\}$› **and** *m-singleton*: ‹$\bigwedge a.\ a \in\# A \Longrightarrow P\ \{\#a\#\}$›
and *hyp*: ‹$\bigwedge a\ x\ F.\ [\![a \in\# A;\ x \in\# A - \textit{add-mset}\ a\ F;\ \textit{add-mset}\ a\ F \subseteq\# A;$
$P\ (\textit{add-mset}\ a\ F)]\!] \Longrightarrow P\ (\textit{add-mset}\ x\ (\textit{add-mset}\ a\ F))$›
shows ‹$P\ A$›
⟨*proof*⟩

lemma *induct-subset-mset-empty-single*:
‹$[\![P\ \{\#\};\ \bigwedge a.\ a \in\# M \Longrightarrow P\ \{\#a\#\};$
$\bigwedge N\ a.\ [\![a \in\# M;\ N \subseteq\# M;\ N \neq \{\#\};\ P\ N]\!] \Longrightarrow P\ (\textit{add-mset}\ a\ N)]\!] \Longrightarrow P\ M$›
⟨*proof*⟩

2.3 Strong Induction for *nat*

lemma *strong-nat-induct*[*consumes 0*, *case-names 0 Suc*]:
‹$[\![P\ 0;\ \bigwedge n.\ (\bigwedge m.\ m \leq n \Longrightarrow P\ m) \Longrightarrow P\ (\textit{Suc}\ n)]\!] \Longrightarrow P\ n$›
⟨*proof*⟩

lemma *strong-nat-induct-non-zero*[*consumes 1*, *case-names 1 Suc*]:
‹$[\![0 < n;\ P\ 1;\ \bigwedge n.\ 0 < n \Longrightarrow (\bigwedge m.\ 0 < m \wedge m \leq n \Longrightarrow P\ m) \Longrightarrow P\ (\textit{Suc}\ n)]\!]$
$\Longrightarrow P\ n$›
⟨*proof*⟩

2.4 Useful Results for Cartesian Products

lemma *prem-Multi-cartprod*:
‹$(\lambda(x,\ y).\ x\ @\ y\ )\ `\ (A\ \times\ B\ ) = \{s\ @\ t\ \ |s\ t.\ (s,\ t) \in A\ \times\ B\ \}$›
‹$(\lambda(x,\ y).\ x\ \#\ y\ )\ `\ (A' \times\ B\ ) = \{s\ \#\ t\ \ |s\ t.\ (s,\ t) \in A' \times\ B\ \}$›
‹$(\lambda(x,\ y).\ [x,\ y])\ `\ (A' \times\ B') = \{[s,\ t]\ |s\ t.\ (s,\ t) \in A' \times\ B'\}$›
⟨*proof*⟩

Chapter 3

Definitions of the Architectural Operators

3.1 The Global Deterministic Choice

3.1.1 Definition

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

lift-definition *GlobalDet* :: ‹['b set, 'b ⇒ ('a, 'r) process$_{ptick}$] ⇒ ('a, 'r) process$_{ptick}$›
  is ‹λA P. ({(s, X). s = [] ∧ (s, X) ∈ (⋂a∈A. ℱ (P a))} ∪
        {(s, X). s ≠ [] ∧ (s, X) ∈ (⋃a∈A. ℱ (P a))} ∪
        {(s, X). s = [] ∧ s ∈ (⋃a∈A. 𝒟 (P a))} ∪
        {(s, X). ∃r. s = [] ∧ ✓(r) ∉ X ∧ [✓(r)] ∈ (⋃a∈A. 𝒯 (P a))},
        ⋃a∈A. 𝒟 (P a))›
⟨*proof*⟩

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

3.1.2 The projections

lemma *F-GlobalDet*:
  ‹ℱ (□ x ∈ A. P x) =
   {(s, X). s = [] ∧ (s, X) ∈ (⋂a∈A. ℱ (P a))} ∪
   {(s, X). s ≠ [] ∧ (s, X) ∈ (⋃a∈A. ℱ (P a))} ∪
   {(s, X). s = [] ∧ s ∈ (⋃a∈A. 𝒟 (P a))} ∪
   {(s, X). ∃r. s = [] ∧ ✓(r) ∉ X ∧ [✓(r)] ∈ (⋃a∈A. 𝒯 (P a))}›

⟨*proof*⟩

lemma *F-GlobalDet'*:
‹$\mathcal{F}$ ($\Box$ $x \in A.\ P\ x$) =
{([], X)| X. ($\exists\, a{\in}A.\ P\ a = \bot$) $\vee$ ($\forall\, a{\in}A.$ ([], X) $\in \mathcal{F}$ ($P\ a$)) $\vee$
($\exists\, a{\in}A.\ \exists\, r.$ ✓(r) $\notin X \wedge$ [✓(r)] $\in \mathcal{T}$ ($P\ a$))} $\cup$
{(s, X)| $a\ s\ X.\ a \in A \wedge s \neq [] \wedge (s, X) \in \mathcal{F}\ (P\ a)$}›
(**is** ‹$\mathcal{F}$ ($\Box$ $x \in A.\ P\ x$) = *?rhs*›)
⟨*proof*⟩

lemma *D-GlobalDet*: ‹$\mathcal{D}$ ($\Box$ $x \in A.\ P\ x$) = ($\bigcup a{\in}A.\ \mathcal{D}\ (P\ a)$)›
⟨*proof*⟩

lemma *T-GlobalDet*:
‹$\mathcal{T}$ ($\Box$ $x \in A.\ P\ x$) = (*if* $A = \{\}$ *then* $\{[]\}$ *else* ($\bigcup$ $x{\in}A.\ \mathcal{T}\ (P\ x)$))›
⟨*proof*⟩

lemma *T-GlobalDet'*: ‹$\mathcal{T}$ ($\Box$ $x \in A.\ P\ x$) = (*insert* [] ($\bigcup$ $x{\in}A.\ \mathcal{T}\ (P\ x)$))›
⟨*proof*⟩

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

lemma *mono-GlobalDet-eq*:
‹($\bigwedge x.\ x \in A \Longrightarrow P\ x = Q\ x$) $\Longrightarrow$ *GlobalDet A P* = *GlobalDet A Q*›
⟨*proof*⟩

lemma *mono-GlobalDet-eq2*:
‹($\bigwedge x.\ x \in A \Longrightarrow P\ (f\ x) = Q\ x$) $\Longrightarrow$ *GlobalDet* (f ‘ A) P = *GlobalDet A Q*›
⟨*proof*⟩

3.1.3 Factorization of ($\Box$) in front of *GlobalDet*

lemma *Process-eq-optimized-bisI* :
assumes ‹$\bigwedge s.\ s \in \mathcal{D}\ P \Longrightarrow s \in \mathcal{D}\ Q$› ‹$\bigwedge s.\ s \in \mathcal{D}\ Q \Longrightarrow s \in \mathcal{D}\ P$›
and ‹$\bigwedge X.\ \mathcal{D}\ P = \mathcal{D}\ Q \Longrightarrow ([], X) \in \mathcal{F}\ P \Longrightarrow ([], X) \in \mathcal{F}\ Q$›
and ‹$\bigwedge X.\ \mathcal{D}\ Q = \mathcal{D}\ P \Longrightarrow ([], X) \in \mathcal{F}\ Q \Longrightarrow ([], X) \in \mathcal{F}\ P$›
and ‹$\bigwedge a\ s\ X.\ \mathcal{D}\ P = \mathcal{D}\ Q \Longrightarrow (a\ \#\ s, X) \in \mathcal{F}\ P \Longrightarrow (a\ \#\ s, X) \in \mathcal{F}\ Q$›
and ‹$\bigwedge a\ s\ X.\ \mathcal{D}\ Q = \mathcal{D}\ P \Longrightarrow (a\ \#\ s, X) \in \mathcal{F}\ Q \Longrightarrow (a\ \#\ s, X) \in \mathcal{F}\ P$›
shows ‹$P = Q$›
⟨*proof*⟩

lemma *GlobalDet-factorization-union*:
‹($\Box$ $p \in A.\ P\ p$) $\Box$ ($\Box$ $p \in B.\ P\ p$) = $\Box$ $p \in (A \cup B)\ .\ P\ p$›
⟨*proof*⟩

lemma *GlobalDet-Union* :

‹(□a ∈ (⋃ i ∈ I. A i). P a) = □i ∈ I. □a ∈ A i. P a› (**is** ‹?lhs = ?rhs›)
⟨*proof*⟩

3.1.4 First properties

lemma *GlobalDet-id* [*simp*] : ‹A ≠ {} ⟹ (□ p ∈ A. P) = P›
⟨*proof*⟩

lemma *GlobalDet-unit*[*simp*] : ‹(□ x ∈ {a}. P x) = P a›
⟨*proof*⟩

lemma *GlobalDet-empty*[*simp*] : ‹(□a ∈ {}. P a) = STOP›
⟨*proof*⟩

lemma *GlobalDet-distrib-unit*:
‹(□ x ∈ *insert* a A. P x) = P a □ (□ x ∈ (A − {a}). P x)›
⟨*proof*⟩

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

3.1.5 Behaviour of *GlobalDet* with (□)

lemma *GlobalDet-Det-GlobalDet*:
‹(□ a ∈ A. P a) □ (□ a ∈ A. Q a) = □ a ∈ A. P a □ Q a›
(**is** ‹?G1 □ ?G2 = ?G›)
⟨*proof*⟩

3.1.6 Commutativity

lemma *GlobalDet-sets-commute*:
‹(□ a ∈ A. □ b ∈ B. P a b) = □ b ∈ B. □ a ∈ A. P a b› (**is** ‹?lhs = ?rhs›)
⟨*proof*⟩

3.1.7 Behaviour with injectivity

lemma *inj-on-mapping-over-GlobalDet*:
‹*inj-on* f A ⟹ (□ x ∈ A. P x) = □ x ∈ f ‘ A. P (*inv-into* A f x)›
⟨*proof*⟩

3.1.8 Cartesian product results

lemma *GlobalDet-cartprod-σs-set-σs-set*:
‹(□ (s, t) ∈ A × B. P (s @ t)) = □ u ∈ {s @ t |s t. (s, t) ∈ A × B}. P u›
(**is** ‹?lhs = ?rhs›)
⟨*proof*⟩

lemma *GlobalDet-cartprod-s-set-σs-set*:
‹(□ (*s*, *t*) ∈ *A* × *B*. *P* (*s* # *t*)) = □ *u* ∈ {*s* # *t* |*s t*. (*s*, *t*) ∈ *A* × *B*}. *P u*›
(**is** ‹*?lhs* = *?rhs*›)
⟨*proof*⟩

lemma *GlobalDet-cartprod-s-set-s-set*:
‹(□ (*s*, *t*) ∈ *A* × *B*. *P* [*s*, *t*]) = □ *u* ∈ {[*s*, *t*] |*s t*. (*s*, *t*) ∈ *A* × *B*}. *P u*›
(**is** ‹*?lhs* = *?rhs*›)
⟨*proof*⟩

lemma *GlobalDet-cartprod*: ‹(□(*s*, *t*) ∈ *A* × *B*. *P s t*) = □*s* ∈ *A*. □*t* ∈ *B*. *P s t*›
(**is** ‹*?lhs* = *?rhs*›)
⟨*proof*⟩

3.1.9 Link with *Mprefix*

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

lemma *Mprefix-GlobalDet*: ‹□ *a* ∈ *A* → *P a* = □ *a* ∈ *A*. *a* → *P a*›
⟨*proof*⟩

lemma *read-is-GlobalDet-write0* :
‹*c***?***a*∈*A* → *P a* = □*b*∈*c* ‘ *A*. *b* → *P* (*inv-into A c b*)›
⟨*proof*⟩

lemma *read-is-GlobalDet-write* :
‹*inj-on c A* ⟹ *c***?***a*∈*A* → *P a* = □*a*∈*A*. *c***!***a* → *P a*›
⟨*proof*⟩

3.1.10 Properties

lemma *GlobalDet-Det*: ‹(□ *a* ∈ *A*. *P a*) □ *Q* = (*if A* = {} *then Q else* □ *a* ∈ *A*. *P a* □ *Q*)›
(**is** ‹*?lhs* = (*if A* = {} *then Q else ?rhs*)›)
⟨*proof*⟩

lemma *Mndetprefix-Sync-Mprefix-strong-subset*:
‹⟦*A* ⊆ *B*; *B* ⊆ *C*⟧ ⟹ ⊓ *a* ∈ *A* → *P a* ⟦ *C* ⟧ □ *b* ∈ *B* → *Q b* = ⊓ *a* ∈ *A* → (*P a* ⟦ *C* ⟧ *Q a*)›
⟨*proof*⟩

lemma *Mprefix-Sync-Mndetprefix-strong-subset*:

‹$[\![A \subseteq C;\ B \subseteq A]\!] \Longrightarrow \Box\ a \in A \rightarrow P\ a\ [\![\ C\ ]\!] \sqcap b \in B \rightarrow Q\ b = \sqcap\ b \in B \rightarrow (P\ b\ [\![\ C\ ]\!]\ Q\ b)$›
⟨*proof*⟩

corollary *Mndetprefix-Par-Mprefix-strong-subset*:
‹$A \subseteq B \Longrightarrow \sqcap\ a \in A \rightarrow P\ a\ ||\ \Box\ b \in B \rightarrow Q\ b = \sqcap\ a \in A \rightarrow (P\ a\ ||\ Q\ a)$›
⟨*proof*⟩

corollary *Mprefix-Par-Mndetprefix-strong-subset*:
‹$B \subseteq A \Longrightarrow \Box\ a \in A \rightarrow P\ a\ ||\ \sqcap\ b \in B \rightarrow Q\ b = \sqcap\ b \in B \rightarrow (P\ b\ ||\ Q\ b)$›
⟨*proof*⟩

3.1.11 Continuity

lemma *mono-GlobalDet* : ‹$(\Box a \in A.\ P\ a) \sqsubseteq \Box a \in A.\ Q\ a$› **if** ‹$\bigwedge x.\ x \in A \Longrightarrow P\ x \sqsubseteq Q\ x$›
⟨*proof*⟩

lemma *chain-GlobalDet* : ‹*chain* $Y \Longrightarrow$ *chain* $(\lambda i.\ \Box a \in A.\ Y\ i\ a)$›
⟨*proof*⟩

lemma *GlobalDet-cont* [*simp*] : ‹$[\![$*finite* A; $\bigwedge a.\ a \in A \Longrightarrow$ *cont* $(P\ a)]\!] \Longrightarrow$ *cont* $(\lambda y.\ \Box\ z{\in}A.\ P\ z\ y)$›
⟨*proof*⟩

end

3.2 Multiple Synchronization Product

3.2.1 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 *'b multiset* and not on *'b set*.

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

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

interpretation *MultiSync*: *comp-fun-commute* **where** $f = \langle \lambda x\ r.\ r\ [\![E]\!]\ P\ x \rangle$
⟨*proof*⟩

lemma *MultiSync-list-mset*:
$\langle mset\ L = mset\ L' \Longrightarrow MultiSync\text{-}list\ S\ L\ P = MultiSync\text{-}list\ S\ L'\ P \rangle$
⟨*proof*⟩

definition *MultiSync* :: $\langle ['a\ set,\ 'b\ multiset,\ 'b \Rightarrow ('a,\ 'r)\ process_{ptick}] \Rightarrow ('a,\ 'r)\ process_{ptick} \rangle$
where $\langle MultiSync\ S\ M\ P = MultiSync\text{-}list\ S\ (SOME\ L.\ mset\ L = M)\ P \rangle$

syntax *-MultiSync* :: $\langle ['a\ set, pttrn, 'b\ multiset, ('a,\ 'r)\ process_{ptick}] \Rightarrow ('a,\ 'r)\ process_{ptick} \rangle$
$(\langle (3[\![\text{-}]\!]\ \text{-}\in\#\text{-}./\ \text{-}) \rangle\ [78,78,78,77]\ 77)$
syntax-consts *-MultiSync* ⇌ *MultiSync*
translations $[\![S]\!]\ p \in\#\ M.\ P \rightleftharpoons CONST\ MultiSync\ S\ M\ (\lambda p.\ P)$

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

abbreviation *MultiInter* :: $\langle ['b\ multiset,\ 'b \Rightarrow ('a,\ 'r)\ process_{ptick}] \Rightarrow ('a,\ 'r)\ process_{ptick} \rangle$
where $\langle MultiInter\ M\ P \equiv MultiSync\ \{\}\ M\ P \rangle$

syntax *-MultiInter* :: $\langle [pttrn,\ 'b\ multiset,\ ('a,\ 'r)\ process_{ptick}] \Rightarrow ('a,\ 'r)\ process_{ptick} \rangle$
$(\langle (3||| \text{-}\in\#\text{-}./\ \text{-}) \rangle\ [78,78,77]\ 77)$
syntax-consts *-MultiInter* ⇌ *MultiInter*
translations $||| p \in\#\ M.\ P \rightleftharpoons CONST\ MultiInter\ M\ (\lambda p.\ P)$

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

abbreviation *MultiPar* :: $\langle ['b\ multiset,\ 'b \Rightarrow ('a,\ 'r)\ process_{ptick}] \Rightarrow ('a,\ 'r)\ process_{ptick} \rangle$
where $\langle MultiPar\ M\ P \equiv MultiSync\ UNIV\ M\ P \rangle$

syntax *-MultiPar* :: $\langle [pttrn,\ 'b\ multiset,\ ('a,\ 'r)\ process_{ptick}] \Rightarrow ('a,\ 'r)\ process_{ptick} \rangle$
$(\langle (3|| \text{-}\in\#\text{-}.\ /\ \text{-}) \rangle\ [78,78,77]\ 77)$
syntax-consts *-MultiPar* ⇌ *MultiPar*
translations $|| p \in\#\ M.\ P \rightleftharpoons CONST\ MultiPar\ M\ (\lambda p.\ P)$

3.2.2 First properties

lemma *MultiSync-rec0*[*simp*]: $\langle ([\![S]\!]\ p \in\#\ \{\#\}.\ P\ p) = STOP \rangle$
⟨*proof*⟩

lemma *MultiSync-rec1*[*simp*]: $\langle ([\![S]\!]\ p \in\#\ \{\#a\#\}.\ P\ p) = P\ a \rangle$
⟨*proof*⟩

lemma *MultiSync-add*[*simp*]:
‹$M \neq \{\#\} \Longrightarrow (\llbracket S \rrbracket\ p \in\# \textit{add-mset}\ m\ M.\ P\ p) = P\ m\ \llbracket S \rrbracket\ (\llbracket S \rrbracket\ p \in\# M.\ P\ p)$›
⟨*proof*⟩

lemma *mono-MultiSync-eq*:
‹$(\bigwedge x.\ x \in\# M \Longrightarrow P\ x = Q\ x) \Longrightarrow \textit{MultiSync}\ S\ M\ P = \textit{MultiSync}\ S\ M\ Q$›
⟨*proof*⟩

lemma *mono-MultiSync-eq2*:
‹$(\bigwedge x.\ x \in\# M \Longrightarrow P\ (f\ x) = Q\ x) \Longrightarrow \textit{MultiSync}\ S\ (\textit{image-mset}\ f\ M)\ P = \textit{MultiSync}\ S\ M\ Q$›
⟨*proof*⟩

lemmas *MultiInter-rec0* = *MultiSync-rec0*[**where** S = ‹{}›]
and *MultiPar-rec0* = *MultiSync-rec0*[**where** S = ‹*UNIV*›]
and *MultiInter-rec1* = *MultiSync-rec1*[**where** S = ‹{}›]
and *MultiPar-rec1* = *MultiSync-rec1*[**where** S = ‹*UNIV*›]
and *MultiInter-add* = *MultiSync-add*[**where** S = ‹{}›]
and *MultiPar-add* = *MultiSync-add*[**where** S = ‹*UNIV*›]
and *mono-MultiInter-eq* = *mono-MultiSync-eq*[**where** S = ‹{}›]
and *mono-MultiPar-eq* = *mono-MultiSync-eq*[**where** S = ‹*UNIV*›]
and *mono-MultiInter-eq2* = *mono-MultiSync-eq2*[**where** S = ‹{}›]
and *mono-MultiPar-eq2* = *mono-MultiSync-eq2*[**where** S = ‹*UNIV*›]

3.2.3 Some Tests

lemma ‹*MultiSync-list* S [] P = *STOP*›
and ‹*MultiSync-list* S [a] P = $P\ a$›
and ‹*MultiSync-list* S [a, b] P = $P\ a\ \llbracket S \rrbracket\ P\ b$›
and ‹*MultiSync-list* S [a, b, c] P = $P\ a\ \llbracket S \rrbracket\ P\ b\ \llbracket S \rrbracket\ P\ c$›
⟨*proof*⟩

lemma *test-MultiSync*:
‹$(\llbracket S \rrbracket\ p \in\# \textit{mset}\ [].\ P\ p) = \textit{STOP}$›
‹$(\llbracket S \rrbracket\ p \in\# \textit{mset}\ [a].\ P\ p) = P\ a$›
‹$(\llbracket S \rrbracket\ p \in\# \textit{mset}\ [a, b].\ P\ p) = P\ a\ \llbracket S \rrbracket\ P\ b$›
‹$(\llbracket S \rrbracket\ p \in\# \textit{mset}\ [a, b, c].\ P\ p) = P\ a\ \llbracket S \rrbracket\ P\ b\ \llbracket S \rrbracket\ P\ c$›
⟨*proof*⟩

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

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

lemma *MultiSync-set3*:
‹*l* < *k* ⟹ *MultiSync S* (*mset-set* {*l* ..< *Suc k*}) *P* =
P l ⟦*S*⟧ (*MultiSync S* (*mset-set* {*Suc l* ..< *Suc k*}) *P*)›
⟨*proof*⟩

lemma *test-MultiSync′*:
‹(⟦*S*⟧ *p* ∈# *mset-set* {*1*::*int* .. *3*}. *P p*) = *P 1* ⟦*S*⟧ *P 2* ⟦*S*⟧ *P 3*›
⟨*proof*⟩

lemma *test-MultiSync″*:
‹(⟦*S*⟧ *p* ∈# *mset-set* {*0*::*nat* .. *a*}. *P p*) =
⟦*S*⟧ *p* ∈# *mset-set* ({*a*} ∪ {*1* .. *a*} ∪ {*0*}) . *P p*›
⟨*proof*⟩

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

3.2.4 Continuity

lemma *mono-MultiSync* :
‹(⋀*x*. *x* ∈# *M* ⟹ *P x* ⊑ *Q x*) ⟹ (⟦*S*⟧ *m* ∈# *M*. *P m*) ⊑ (⟦*S*⟧ *m* ∈# *M*. *Q m*)›
⟨*proof*⟩

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

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

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

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

lemma *MultiSync-factorization-union*:
$\langle[\![M \neq \{\#\};\ N \neq \{\#\}]\!] \Longrightarrow$
$([\![S]\!]\ z \in\# M.\ P\ z)\ [\![S]\!]\ ([\![S]\!]\ z \in\# N.\ P\ z) = [\![S]\!]\ z{\in\#}\ (M + N).\ P\ z\rangle$
⟨*proof*⟩

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

3.2.6 ⊥ Absorbtion

lemma *MultiSync-BOT-absorb*:
$\langle m \in\# M \Longrightarrow P\ m = \bot \Longrightarrow ([\![S]\!]\ z \in\# M.\ P\ z) = \bot\rangle$
⟨*proof*⟩

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

lemma *MultiSync-is-BOT-iff*:
$\langle([\![S]\!]\ m \in\# M.\ P\ m) = \bot \longleftrightarrow (\exists\, m \in\# M.\ P\ m = \bot)\rangle$
⟨*proof*⟩

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

3.2.7 Other Properties

lemma *MultiSync-SKIP-id*:
$\langle([\![S]\!]\ r \in\# M.\ SKIP\ r) = (if\ \exists\, r.\ set\text{-}mset\ M = \{r\}\ then\ SKIP\ (THE\ r.\ set\text{-}mset\ M = \{r\})\ else\ STOP)\rangle$
⟨*proof*⟩

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

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

assumes ‹$m \in\# M$› **and** ‹$m' \in\# M$› **and** ‹$fst\ m \neq fst\ m'$›
shows ‹$(||\ a \in\# M.\ (fst\ a \rightarrow P\ (snd\ a))) = STOP$›
⟨*proof*⟩

lemma *MultiPar-prefix-two-distincts-STOP'*:
‹$[\![(m,\ n) \in\# M;\ (m',\ n') \in\# M;\ m \neq m']\!] \Longrightarrow$
$(||\ (m,\ n) \in\# M.\ (m \rightarrow P\ n)) = STOP$›
⟨*proof*⟩

3.2.8 Behaviour of *MultiSync* with *Sync*

lemma *MultiSync-Sync*:
‹$([\![S]\!]\ z \in\# M.\ P\ z)\ [\![S]\!]\ ([\![S]\!]\ z \in\# M.\ P'\ z) = [\![S]\!]\ z \in\# M.\ (P\ z\ [\![S]\!]\ P'\ z)$›
⟨*proof*⟩

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

3.2.9 Commutativity

lemma *MultiSync-sets-commute*:
‹$([\![S]\!]\ a \in\# M.\ [\![S]\!]\ b \in\# N.\ P\ a\ b) = [\![S]\!]\ b \in\# N.\ [\![S]\!]\ a \in\# M.\ P\ a\ b$›
⟨*proof*⟩

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

3.2.10 Behaviour with Injectivity

lemma *inj-on-mapping-over-MultiSync*:
‹*inj-on f* (*set-mset M*) $\Longrightarrow$
$([\![S]\!]\ x \in\# M.\ P\ x) = [\![S]\!]\ x \in\#$ *image-mset f M*. P (*inv-into* (*set-mset M*) *f x*)›
⟨*proof*⟩

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

3.3 Multiple Sequential Composition

Because of the fact that *SKIP r* is not exactly a neutral element for *Seq* (cf *SKIP ?r* **;** *?P* = *?P*

?P **;** *Skip* = *?P*), we do the folding on the reversed list.

3.3.1 Definition

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

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

lemma *MultiSeq-Nil* [*simp*] : ‹*MultiSeq* [] *P* = *SKIP undefined*›
  and *MultiSeq-snoc* [*simp*] : ‹*MultiSeq* (*L* @ [*l*]) *P* = *MultiSeq L P* **;** *P l*›
  ⟨*proof*⟩

lemma *MultiSeq-elims* :
  ‹*MultiSeq L P* = *Q* ⟹
   (⋀*P'*. *L* = [] ⟹ *P* = *P'* ⟹ *Q* = *SKIP undefined* ⟹ *thesis*) ⟹
   (⋀*l L' P'*. *L* = *L'* @ [*l*] ⟹ *P* = *P'* ⟹ *Q* = *MultiSeq L' P'* **;** *P' l* ⟹ *thesis*)
⟹ *thesis*›
  ⟨*proof*⟩

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

3.3.2 First Properties

lemma ‹*SEQ p* ∈@ []. *P p* = *SKIP undefined*› ⟨*proof*⟩

lemma ‹*SEQ i* ∈@ (*L* @ [*l*]). *P i* = *SEQ i* ∈@ *L*. *P i* **;** *P l*› ⟨*proof*⟩

lemma *MultiSeq-singl* [*simp*] : ‹*SEQ l* ∈@ [*l*]. *P l* = *P l*› ⟨*proof*⟩

3.3.3 Some Tests

lemma ‹*SEQ p* ∈@ []. *P p* = *SKIP undefined*›
  and ‹*SEQ p* ∈@ [*a*]. *P p* = *P a*›
  and ‹*SEQ p* ∈@ [*a*, *b*]. *P p* = *P a* **;** *P b*›
  and ‹*SEQ p* ∈@ [*a*, *b*, *c*]. *P p* = *P a* **;** *P b* **;** *P c*›
  ⟨*proof*⟩

lemma *test-MultiSeq*: ‹(*SEQ p* ∈@ [*1*::*int* .. *3*]. *P p*) = *P 1* ; *P 2* ; *P 3*›
⟨*proof*⟩

3.3.4 Continuity

lemma *mono-MultiSeq* :
‹(⋀*x*. *x* ∈ *set L* ⟹ *P x* ⊑ *Q x*) ⟹ *SEQ l* ∈@ *L*. *P l* ⊑ *SEQ l* ∈@ *L*. *Q l*›
⟨*proof*⟩

lemma *MultiSeq-cont*[*simp*]:
‹(⋀*x*. *x* ∈ *set L* ⟹ *cont* (*P x*)) ⟹ *cont* (λ*y*. *SEQ z* ∈@ *L*. *P z y*)›
⟨*proof*⟩

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

lemma *MultiSeq-factorization-append*:
‹*L2* ≠ [] ⟹ *SEQ p* ∈@ *L1*. *P p* ; *SEQ p* ∈@ *L2*. *P p* = *SEQ p* ∈@ (*L1* @ *L2*). *P p*›
⟨*proof*⟩

3.3.6 ⊥ Absorbtion

lemma *MultiSeq-BOT-absorb*:
‹*SEQ z* ∈@ (*L1* @ *a* # *L2*). *P z* = *SEQ z* ∈@ *L1*. *P z* ; ⊥› **if** ‹*P a* = ⊥›
⟨*proof*⟩

3.3.7 First Properties

lemma *MultiSeq-SKIP-neutral*:
‹*SEQ z* ∈@ (*L1* @ *a* # *L2*). *P z* =
(*if L2* = [] *then SEQ z* ∈@ *L1*. *P z* ; *SKIP r*
else SEQ z ∈@ (*L1* @ *L2*). *P z*)› **if** ‹*P a* = *SKIP r*›
⟨*proof*⟩

lemma *MultiSeq-STOP-absorb*:
‹*SEQ z* ∈@ (*L1* @ *a* # *L2*). *P z* = *SEQ z* ∈@ *L1*. *P z* ; *STOP*› **if** ‹*P a* = *STOP*›
⟨*proof*⟩

lemma *mono-MultiSeq-eq*:
‹(⋀*x*. *x* ∈ *set L* ⟹ *P x* = *Q x*) ⟹ *MultiSeq L P* = *MultiSeq L Q*›
⟨*proof*⟩

3.3.8 Commutativity

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

3.3.9 Behaviour with Injectivity

lemma *inj-on-mapping-over-MultiSeq*:
‹*inj-on f* (*set C*) ⟹
SEQ x ∈@ *C*. *P x* = *SEQ x* ∈@ *map f C*. *P* (*inv-into* (*set C*) *f x*)›
⟨*proof*⟩

3.3.10 Definition of *first-elem*

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

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

This will be very useful later.

value ‹*first-elem* (λ*x*. *4* < *x*) [*0*::*nat*, *2*, *5*]›
lemma ‹*first-elem* (λ*x*. *5* < *x*) [*0*::*nat*, *2*, *5*] = *3*› ⟨*proof*⟩
lemma ‹*P* ' *set L* ⊆ {*False*} ⟹ *first-elem P L* = *length L*› ⟨*proof*⟩

3.4 The Throw Operator

3.4.1 Definition

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

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

This is a relatively new addition to CSP (see [3, p.140]).

lift-definition *Throw* :: ‹[('*a*, '*r*) *process*$_{ptick}$, '*a set*, '*a* ⇒ ('*a*, '*r*) *process*$_{ptick}$]
⇒ ('*a*, '*r*) *process*$_{ptick}$›
is ‹λ *P A Q*.
({(*t1*, *X*). (*t1*, *X*) ∈ ℱ *P* ∧ *set t1* ∩ *ev* ' *A* = {}} ∪
{(*t1* @ *t2*, *X*) |*t1 t2 X*. *t1* ∈ 𝒟 *P* ∧ *tF t1* ∧ *set t1* ∩ *ev* ' *A* = {} ∧ *ftF t2*} ∪
{(*t1* @ *ev a* # *t2*, *X*) |*t1 a t2 X*.
t1 @ [*ev a*] ∈ 𝒯 *P* ∧ *set t1* ∩ *ev* ' *A* = {} ∧ *a* ∈ *A* ∧ (*t2*, *X*) ∈ ℱ (*Q a*)},
{*t1* @ *t2* |*t1 t2*. *t1* ∈ 𝒟 *P* ∧ *tF t1* ∧ *set t1* ∩ *ev* ' *A* = {} ∧ *ftF t2*} ∪
{*t1* @ *ev a* # *t2* |*t1 a t2*. *t1* @ [*ev a*] ∈ 𝒯 *P* ∧ *set t1* ∩ *ev* ' *A* = {} ∧ *a* ∈ *A* ∧
t2 ∈ 𝒟 (*Q a*)})›
⟨*proof*⟩

We add some syntactic sugar.

syntax -*Throw* :: ‹[('*a*, '*r*) *process*$_{ptick}$, *pttrn*, '*a set*, '*a* ⇒ ('*a*, '*r*) *process*$_{ptick}$] ⇒
('*a*, '*r*) *process*$_{ptick}$›

(‹((-) Θ (-∈-)./ (-))› [78,78,78,77] 77)
syntax-consts -*Throw* ⇌ *Throw*
translations *P* Θ *a* ∈ *A*. *Q* ⇌ *CONST Throw P A* (λ*a*. *Q*)

3.4.2 Projections

lemma *F-Throw*:
‹$\mathcal{F}$ (*P* Θ *a* ∈ *A*. *Q a*) =
{(*t1*, *X*). (*t1*, *X*) ∈ $\mathcal{F}$ *P* ∧ *set t1* ∩ *ev* ‘ *A* = {}} ∪
{(*t1* @ *t2*, *X*) |*t1 t2 X*. *t1* ∈ $\mathcal{D}$ *P* ∧ *tF t1* ∧ *set t1* ∩ *ev* ‘ *A* = {} ∧ *ftF t2*} ∪
{(*t1* @ *ev a* # *t2*, *X*) |*t1 a t2 X*.
t1 @ [*ev a*] ∈ $\mathcal{T}$ *P* ∧ *set t1* ∩ *ev* ‘ *A* = {} ∧ *a* ∈ *A* ∧ (*t2*, *X*) ∈ $\mathcal{F}$ (*Q a*)}›
⟨*proof*⟩

lemma *D-Throw*:
‹$\mathcal{D}$ (*P* Θ *a* ∈ *A*. *Q a*) =
{*t1* @ *t2* |*t1 t2*. *t1* ∈ $\mathcal{D}$ *P* ∧ *tF t1* ∧ *set t1* ∩ *ev* ‘ *A* = {} ∧ *ftF t2*} ∪
{*t1* @ *ev a* # *t2* |*t1 a t2*. *t1* @ [*ev a*] ∈ $\mathcal{T}$ *P* ∧ *set t1* ∩ *ev* ‘ *A* = {} ∧ *a* ∈ *A* ∧
t2 ∈ $\mathcal{D}$ (*Q a*)}›
⟨*proof*⟩

lemma *T-Throw*:
‹$\mathcal{T}$ (*P* Θ *a* ∈ *A*. *Q a*) =
{*t1* ∈ $\mathcal{T}$ *P*. *set t1* ∩ *ev* ‘ *A* = {}} ∪
{*t1* @ *t2* |*t1 t2*. *t1* ∈ $\mathcal{D}$ *P* ∧ *tF t1* ∧ *set t1* ∩ *ev* ‘ *A* = {} ∧ *ftF t2*} ∪
{*t1* @ *ev a* # *t2* |*t1 a t2*. *t1* @ [*ev a*] ∈ $\mathcal{T}$ *P* ∧ *set t1* ∩ *ev* ‘ *A* = {} ∧ *a* ∈ *A* ∧
t2 ∈ $\mathcal{T}$ (*Q a*)}›
⟨*proof*⟩

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

lemma *Throw-T-third-clause-breaker* :
‹⟦*set t* ∩ *S* = {}; *set t′* ∩ *S* = {}; *e* ∈ *S*; *e′* ∈ *S*⟧ ⟹
t @ *e* # *u* = *t′* @ *e′* # *u′* ⟷ *t* = *t′* ∧ *e* = *e′* ∧ *u* = *u′*›
⟨*proof*⟩

3.4.3 Monotony

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

lemma *mono-Throw*[*simp*] : ‹*P* Θ *a* ∈ *A*. *Q a* ⊑ *P′* Θ *a* ∈ *A*. *Q′ a*›
if ‹*P* ⊑ *P′*› **and** ‹⋀*a*. *a* ∈ *A* ⟹ *a* ∈ α(*P*) ⟹ *Q a* ⊑ *Q′ a*›
⟨*proof*⟩

lemma *mono-Throw-eq* :
‹($\bigwedge a.\ a \in A \Longrightarrow a \in \alpha(P) \Longrightarrow Q\ a = Q'\ a) \Longrightarrow$
$P\ \Theta\ a \in A.\ Q\ a = P\ \Theta\ a \in A.\ Q'\ a$›
⟨*proof*⟩

3.4.4 Properties

lemma *Throw-STOP* [*simp*] : ‹$STOP\ \Theta\ a \in A.\ Q\ a = STOP$›
⟨*proof*⟩

lemma *Throw-is-STOP-iff* : ‹$P\ \Theta\ a \in A.\ Q\ a = STOP \longleftrightarrow P = STOP$›
⟨*proof*⟩

lemma *Throw-SKIP* [*simp*] : ‹$SKIP\ r\ \Theta\ a \in A.\ Q\ a = SKIP\ r$›
⟨*proof*⟩

lemma *Throw-BOT* [*simp*] : ‹$\bot\ \Theta\ a \in A.\ Q\ a = \bot$›
⟨*proof*⟩

lemma *Throw-is-BOT-iff*: ‹$P\ \Theta\ a \in A.\ Q\ a = \bot \longleftrightarrow P = \bot$›
⟨*proof*⟩

lemma *Throw-empty-set* [*simp*] : ‹$P\ \Theta\ a \in \{\}.\ Q\ a = P$›
⟨*proof*⟩

lemma *Throw-is-restrictable-on-events-of* :
‹$P\ \Theta\ a \in A.\ Q\ a = P\ \Theta\ a \in (A \cap \alpha(P)).\ Q\ a$› (**is** ‹*?lhs* = *?rhs*›)
— A stronger version where $\alpha(P)$ is replaced by $\boldsymbol{\alpha}(P) \cup \{a.\ \exists t.\ t\ @\ [ev\ a] \in min\text{-}elems\ (\mathcal{D}\ P)\}$ is probably true.
⟨*proof*⟩

lemma *Throw-disjoint-events-of*: ‹$A \cap \alpha(P) = \{\} \Longrightarrow P\ \Theta\ a \in A.\ Q\ a = P$›
⟨*proof*⟩

3.4.5 Continuity

context begin

private lemma *chain-Throw-left* : ‹$chain\ Y \Longrightarrow chain\ (\lambda i.\ Y\ i\ \Theta\ a \in A.\ Q\ a)$›
⟨*proof*⟩ **lemma** *chain-Throw-right* : ‹$chain\ Y \Longrightarrow chain\ (\lambda i.\ P\ \Theta\ a \in A.\ Y\ i\ a)$›
⟨*proof*⟩ **lemma** *cont-left-prem-Throw* :
‹$(\bigsqcup\ i.\ Y\ i)\ \Theta\ a \in A.\ Q\ a = (\bigsqcup\ i.\ Y\ i\ \Theta\ a \in A.\ Q\ a)$›
(**is** ‹*?lhs* = *?rhs*›) **if** ‹$chain\ Y$›
⟨*proof*⟩ **lemma** *cont-right-prem-Throw* :
‹$P\ \Theta\ a \in A.\ (\bigsqcup\ i.\ Y\ i\ a) = (\bigsqcup\ i.\ P\ \Theta\ a \in A.\ Y\ i\ a)$›
(**is** ‹*?lhs* = *?rhs*›) **if** ‹$chain\ Y$›

⟨*proof*⟩

lemma *Throw-cont*[*simp*] :
  assumes *cont-f* : ‹*cont f*› **and** *cont-g* : ‹∀ *a*. *cont* (*g a*)›
  shows ‹*cont* (λ*x*. *f x* Θ *a* ∈ *A*. *g a x*)›
⟨*proof*⟩

end

3.5 The Interrupt Operator

3.5.1 Definition

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

The definition provided by Roscoe [3, p.239] does not respect the invariant *is-process*: it seems like *tick* is not handled.

We propose here our corrected version.

lift-definition *Interrupt* :: ‹[('*a*, '*r*) *process*$_{ptick}$, ('*a*, '*r*) *process*$_{ptick}$] ⇒ ('*a*, '*r*) *process*$_{ptick}$› (**infixl** ‹△› *81*)
  is ‹λ*P Q*.
  ({(*t* @ [✓(*r*)], *X*) |*t r X*. *t* @ [✓(*r*)] ∈ $\mathcal{T}$ *P*} ∪
   {(*t*, *X* − {✓(*r*)}) |*t r X*. *t* @ [✓(*r*)] ∈ $\mathcal{T}$ *P*} ∪
   {(*t*, *X*). (*t*, *X*) ∈ $\mathcal{F}$ *P* ∧ *tF t* ∧ ([], *X*) ∈ $\mathcal{F}$ *Q*} ∪
   {(*t* @ *u*, *X*) |*t u X*. *t* ∈ $\mathcal{T}$ *P* ∧ *tF t* ∧ (*u*, *X*) ∈ $\mathcal{F}$ *Q* ∧ *u* ≠ []} ∪
   {(*t*, *X* − {✓(*r*)}) |*t r X*. *t* ∈ $\mathcal{T}$ *P* ∧ *tF t* ∧ [✓(*r*)] ∈ $\mathcal{T}$ *Q*} ∪
   {(*t*, *X*). *t* ∈ $\mathcal{D}$ *P*} ∪
   {(*t* @ *u*, *X*) |*t u X*. *t* ∈ $\mathcal{T}$ *P* ∧ *tF t* ∧ *u* ∈ $\mathcal{D}$ *Q*},
   $\mathcal{D}$ *P* ∪ {*t* @ *u* |*t u*. *t* ∈ $\mathcal{T}$ *P* ∧ *tF t* ∧ *u* ∈ $\mathcal{D}$ *Q*})›
⟨*proof*⟩

3.5.2 Projections

lemma *F-Interrupt* :
  ‹$\mathcal{F}$ (*P* △ *Q*) =
   {(*t* @ [✓(*r*)], *X*) |*t r X*. *t* @ [✓(*r*)] ∈ $\mathcal{T}$ *P*} ∪
   {(*t*, *X* − {✓(*r*)}) |*t r X*. *t* @ [✓(*r*)] ∈ $\mathcal{T}$ *P*} ∪
   {(*t*, *X*). (*t*, *X*) ∈ $\mathcal{F}$ *P* ∧ *tF t* ∧ ([], *X*) ∈ $\mathcal{F}$ *Q*} ∪
   {(*t* @ *u*, *X*) |*t u X*. *t* ∈ $\mathcal{T}$ *P* ∧ *tF t* ∧ (*u*, *X*) ∈ $\mathcal{F}$ *Q* ∧ *u* ≠ []} ∪
   {(*t*, *X* − {✓(*r*)}) |*t r X*. *t* ∈ $\mathcal{T}$ *P* ∧ *tF t* ∧ [✓(*r*)] ∈ $\mathcal{T}$ *Q*} ∪
   {(*t*, *X*). *t* ∈ $\mathcal{D}$ *P*} ∪
   {(*t* @ *u*, *X*) |*t u X*. *t* ∈ $\mathcal{T}$ *P* ∧ *tF t* ∧ *u* ∈ $\mathcal{D}$ *Q*}›

⟨*proof*⟩

lemma *D-Interrupt* :
‹$\mathcal{D}\ (P \mathbin{\triangle} Q) = \mathcal{D}\ P \cup \{t \mathbin{@} u \mid t\ u.\ t \in \mathcal{T}\ P \wedge tF\ t \wedge u \in \mathcal{D}\ Q\}$›
⟨*proof*⟩

lemma *T-Interrupt* :
‹$\mathcal{T}\ (P \mathbin{\triangle} Q) = \mathcal{T}\ P \cup \{t \mathbin{@} u \mid t\ u.\ t \in \mathcal{T}\ P \wedge tF\ t \wedge u \in \mathcal{T}\ Q\}$›
⟨*proof*⟩

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

3.5.3 Monotony

lemma *mono-Interrupt* : ‹$P \mathbin{\triangle} Q \sqsubseteq P' \mathbin{\triangle} Q'$› **if** ‹$P \sqsubseteq P'$› **and** ‹$Q \sqsubseteq Q'$›
⟨*proof*⟩

3.5.4 Properties

lemma *Interrupt-STOP* [*simp*] : ‹$P \mathbin{\triangle} STOP = P$›
⟨*proof*⟩

lemma *STOP-Interrupt* [*simp*] : ‹$STOP \mathbin{\triangle} P = P$›
⟨*proof*⟩

lemma *Interrupt-is-STOP-iff* : ‹$P \mathbin{\triangle} Q = STOP \longleftrightarrow P = STOP \wedge Q = STOP$›
⟨*proof*⟩

lemma *Interrupt-BOT* [*simp*] : ‹$P \mathbin{\triangle} \bot = \bot$›
and *BOT-Interrupt* [*simp*] : ‹$\bot \mathbin{\triangle} P = \bot$›
⟨*proof*⟩

lemma *Interrupt-is-BOT-iff* : ‹$P \mathbin{\triangle} Q = \bot \longleftrightarrow P = \bot \vee Q = \bot$›
⟨*proof*⟩

lemma *SKIP-Interrupt-is-SKIP-Det* : ‹$SKIP\ r \mathbin{\triangle} P = SKIP\ r \mathbin{\Box} P$›
⟨*proof*⟩

lemma *Interrupt-assoc*: ‹$P \mathbin{\triangle} (Q \mathbin{\triangle} R) = P \mathbin{\triangle} Q \mathbin{\triangle} R$› (**is** ‹*?lhs* = *?rhs*›)
⟨*proof*⟩

3.5.5 Continuity

context begin

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

⟨*proof*⟩ **lemma** *chain-Interrupt-right*: ‹*chain* $Y \Longrightarrow$ *chain* $(\lambda i.\ P \triangle Y\ i)$›
⟨*proof*⟩ **lemma** *cont-left-prem-Interrupt* : ‹$(\bigsqcup\ i.\ Y\ i) \triangle Q = (\bigsqcup\ i.\ Y\ i \triangle Q)$›
(**is** ‹*?lhs* = *?rhs*›) **if** *chain* : ‹*chain* Y›
⟨*proof*⟩ **lemma** *cont-right-prem-Interrupt* : ‹$S \triangle (\bigsqcup i.\ Y\ i) = (\bigsqcup i.\ S \triangle Y\ i)$› **if** ‹*chain* Y›
⟨*proof*⟩

lemma *Interrupt-cont* [*simp*] :
‹*cont* $(\lambda x.\ f\ x \triangle g\ x)$› **if** ‹*cont* f› **and** ‹*cont* g›
⟨*proof*⟩

end

3.6 Monotonies

3.6.1 The Throw Operator

lemma *mono-Throw-F-right* :
‹$(\bigwedge a.\ a \in A \Longrightarrow a \in \alpha(P) \Longrightarrow Q\ a \sqsubseteq_F Q'\ a) \Longrightarrow P\ \Theta\ a \in A.\ Q\ a \sqsubseteq_F P\ \Theta\ a \in A.\ Q'\ a$›
⟨*proof*⟩

lemma *mono-Throw-T-right* :
‹$(\bigwedge a.\ a \in A \Longrightarrow a \in \alpha(P) \Longrightarrow Q\ a \sqsubseteq_T Q'\ a) \Longrightarrow P\ \Theta\ a \in A.\ Q\ a \sqsubseteq_T P\ \Theta\ a \in A.\ Q'\ a$›
⟨*proof*⟩

lemma *mono-Throw-D-right* :
‹$(\bigwedge a.\ a \in A \Longrightarrow a \in \alpha(P) \Longrightarrow Q\ a \sqsubseteq_D Q'\ a) \Longrightarrow P\ \Theta\ a \in A.\ Q\ a \sqsubseteq_D P\ \Theta\ a \in A.\ Q'\ a$›
⟨*proof*⟩

lemma *mono-Throw-FD* : ‹$P\ \Theta\ a \in A.\ Q\ a \sqsubseteq_{FD} P'\ \Theta\ a \in A.\ Q'\ a$›
if ‹$P \sqsubseteq_{FD} P'$› **and** ‹$\bigwedge a.\ a \in A \Longrightarrow a \in \alpha(P) \Longrightarrow Q\ a \sqsubseteq_{FD} Q'\ a$›
⟨*proof*⟩

lemma *mono-Throw-DT* : ‹$P\ \Theta\ a \in A.\ Q\ a \sqsubseteq_{DT} P'\ \Theta\ a \in A.\ Q'\ a$›
if ‹$P \sqsubseteq_{DT} P'$› **and** ‹$\bigwedge a.\ a \in A \Longrightarrow a \in \alpha(P) \Longrightarrow Q\ a \sqsubseteq_{DT} Q'\ a$›
⟨*proof*⟩

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

3.6.2 The Interrupt Operator

lemma *mono-Interrupt-T*: $\langle P \sqsubseteq_T P' \Longrightarrow Q \sqsubseteq_T Q' \Longrightarrow P \triangle Q \sqsubseteq_T P' \triangle Q' \rangle$
⟨*proof*⟩

lemma *mono-Interrupt-D-right* : $\langle Q \sqsubseteq_D Q' \Longrightarrow P \triangle Q \sqsubseteq_D P \triangle Q' \rangle$
⟨*proof*⟩

lemma *mono-Interrupt-FD*:
$\langle P \sqsubseteq_{FD} P' \Longrightarrow Q \sqsubseteq_{FD} Q' \Longrightarrow P \triangle Q \sqsubseteq_{FD} P' \triangle Q' \rangle$
⟨*proof*⟩

lemma *mono-Interrupt-DT*:
$\langle P \sqsubseteq_{DT} P' \Longrightarrow Q \sqsubseteq_{DT} Q' \Longrightarrow P \triangle Q \sqsubseteq_{DT} P' \triangle Q' \rangle$
⟨*proof*⟩

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

3.6.3 Global Deterministic Choice

lemma *mono-GlobalDet-DT* : $\langle (\bigwedge a.\ a \in A \Longrightarrow P\ a \sqsubseteq_{DT} Q\ a) \Longrightarrow (\Box a \in A.\ P\ a) \sqsubseteq_{DT} (\Box a \in A.\ Q\ a) \rangle$
and *mono-GlobalDet-T* : $\langle (\bigwedge a.\ a \in A \Longrightarrow P\ a \sqsubseteq_T Q\ a) \Longrightarrow (\Box a \in A.\ P\ a) \sqsubseteq_T (\Box a \in A.\ Q\ a) \rangle$
and *mono-GlobalDet-D* : $\langle (\bigwedge a.\ a \in A \Longrightarrow P\ a \sqsubseteq_D Q\ a) \Longrightarrow (\Box a \in A.\ P\ a) \sqsubseteq_D (\Box a \in A.\ Q\ a) \rangle$
⟨*proof*⟩

lemma *mono-GlobalDet-FD* : $\langle (\bigwedge a.\ a \in A \Longrightarrow P\ a \sqsubseteq_{FD} Q\ a) \Longrightarrow (\Box a \in A.\ P\ a) \sqsubseteq_{FD} (\Box a \in A.\ Q\ a) \rangle$
⟨*proof*⟩

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

lemma *GlobalNdet-FD-GlobalDet* : $\langle (\sqcap a \in A.\ P\ a) \sqsubseteq_{FD} (\Box a \in A.\ P\ a) \rangle$
and *GlobalNdet-DT-GlobalDet* : $\langle (\sqcap a \in A.\ P\ a) \sqsubseteq_{DT} (\Box a \in A.\ P\ a) \rangle$
and *GlobalNdet-F-GlobalDet* : $\langle (\sqcap a \in A.\ P\ a) \sqsubseteq_F (\Box a \in A.\ P\ a) \rangle$
and *GlobalNdet-T-GlobalDet* : $\langle (\sqcap a \in A.\ P\ a) \sqsubseteq_T (\Box a \in A.\ P\ a) \rangle$
and *GlobalNdet-D-GlobalDet* : $\langle (\sqcap a \in A.\ P\ a) \sqsubseteq_D (\Box a \in A.\ P\ a) \rangle$
⟨*proof*⟩

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

3.6.4 Multiple Synchronization Product

lemma *mono-MultiSync-FD* :

‹($\bigwedge m.\ m \in\# M \Longrightarrow P\ m \sqsubseteq_{FD} Q\ m$) $\Longrightarrow$ ($[\![S]\!]\ m \in\# M.\ P\ m$) $\sqsubseteq_{FD}$ ($[\![S]\!]\ m \in\# M.\ Q\ m$)›
and *mono-MultiSync-DT* :
‹($\bigwedge m.\ m \in\# M \Longrightarrow P\ m \sqsubseteq_{DT} Q\ m$) $\Longrightarrow$ ($[\![S]\!]\ m \in\# M.\ P\ m$) $\sqsubseteq_{DT}$ ($[\![S]\!]\ m \in\# M.\ Q\ m$)›
⟨*proof*⟩

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

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

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

3.6.5 Multiple Sequential Composition

lemma *mono-MultiSeq-FD* :
‹($\bigwedge x.\ x \in set\ L \Longrightarrow P\ x \sqsubseteq_{FD} Q\ x$) $\Longrightarrow$ *SEQ* $l \in@ L.\ P\ l \sqsubseteq_{FD}$ *SEQ* $l \in@ L.\ Q\ l$›
and *mono-MultiSeq-DT* :
‹($\bigwedge x.\ x \in set\ L \Longrightarrow P\ x \sqsubseteq_{DT} Q\ x$) $\Longrightarrow$ *SEQ* $l \in@ L.\ P\ l \sqsubseteq_{DT}$ *SEQ* $l \in@ L.\ Q\ l$›
⟨*proof*⟩

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

3.6.6 The Throw Operator

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

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

⟨*proof*⟩

3.6.7 The Interrupt Operator

lemma *Interrupt-distrib-GlobalNdet-left* :
‹$P \triangle (\sqcap a \in A.\ Q\ a) = (\mathit{if}\ A = \{\}\ \mathit{then}\ P\ \mathit{else} \sqcap a \in A.\ P \triangle Q\ a)$›
(**is** ‹*?lhs* = (*if* - *then* - *else* *?rhs*)›)
⟨*proof*⟩

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

corollary *Interrupt-distrib-Ndet-left* : ‹$P \triangle Q1 \sqcap Q2 = (P \triangle Q1) \sqcap (P \triangle Q2)$›
⟨*proof*⟩

corollary *Interrupt-distrib-Ndet-right* : ‹$P1 \sqcap P2 \triangle Q = (P1 \triangle Q) \sqcap (P2 \triangle Q)$›
⟨*proof*⟩

3.6.8 Global Deterministic Choice

lemma *GlobalDet-distrib-Ndet-left* :
‹$(\Box a \in A.\ P\ a \sqcap Q) = (\mathit{if}\ A = \{\}\ \mathit{then}\ \mathit{STOP}\ \mathit{else}\ (\Box a \in A.\ P\ a) \sqcap Q)$›
⟨*proof*⟩

lemma *GlobalDet-distrib-Ndet-right* :
‹$(\Box a \in A.\ P \sqcap Q\ a) = (\mathit{if}\ A = \{\}\ \mathit{then}\ \mathit{STOP}\ \mathit{else}\ P \sqcap (\Box a \in A.\ Q\ a))$›
⟨*proof*⟩

lemma *Ndet-distrib-GlobalDet-left* :
‹$P \sqcap (\Box a \in A.\ Q\ a) = (\mathit{if}\ A = \{\}\ \mathit{then}\ P \sqcap \mathit{STOP}\ \mathit{else} \Box a \in A.\ P \sqcap Q\ a)$›
⟨*proof*⟩

lemma *Ndet-distrib-GlobalDet-right* :
‹$(\Box a \in A.\ P\ a) \sqcap Q = (\mathit{if}\ A = \{\}\ \mathit{then}\ Q \sqcap \mathit{STOP}\ \mathit{else} \Box a \in A.\ P\ a \sqcap Q)$›
⟨*proof*⟩

3.7 The Step-Laws

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

3.7.1 The Throw Operator

lemma *Throw-Mprefix*:
‹$(\Box a \in A \rightarrow P\ a)\ \Theta\ b \in B.\ Q\ b =$
$\Box a \in A \rightarrow (\mathit{if}\ a \in B\ \mathit{then}\ Q\ a\ \mathit{else}\ P\ a\ \Theta\ b \in B.\ Q\ b)$›
(**is** ‹*?lhs* = *?rhs*›)
⟨*proof*⟩

3.7.2 The Interrupt Operator

lemma *Interrupt-Mprefix*:
‹$(\Box a \in A \rightarrow P\ a)\ \triangle\ Q = Q\ \Box\ (\Box a \in A \rightarrow P\ a\ \triangle\ Q)$› (**is** ‹*?lhs* = *?rhs*›)
⟨*proof*⟩

3.7.3 Global Deterministic Choice

lemma *GlobalDet-Mprefix* :
‹$(\Box a \in A.\ \Box b \in B\ a \rightarrow P\ a\ b) =$
$\Box b \in (\bigcup a \in A.\ B\ a) \rightarrow \sqcap a \in \{a \in A.\ b \in B\ a\}.\ P\ a\ b$› (**is** ‹*?lhs* = *?rhs*›)
⟨*proof*⟩

3.7.4 Multiple Synchronization Product

lemma *MultiSync-Mprefix-pseudo-distrib*:
‹$([\![S]\!]\ B \in\# M.\ \Box\ x \in B \rightarrow P\ B\ x) =$
$\Box\ x \in (\bigcap B \in \mathit{set\text{-}mset}\ M.\ B) \rightarrow ([\![S]\!]\ B \in\# M.\ P\ B\ x)$›
if *nonempty*: ‹$M \neq \{\#\}$› **and** *hyp*: ‹$\bigwedge B.\ B \in\# M \Longrightarrow B \subseteq S$›
⟨*proof*⟩

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

3.7.5 The Throw Operator

lemma *Throw-Mndetprefix*:
‹$(\sqcap a \in A \rightarrow P\ a)\ \Theta\ b \in B.\ Q\ b =$
$\sqcap a \in A \rightarrow (\mathit{if}\ a \in B\ \mathit{then}\ Q\ a\ \mathit{else}\ P\ a\ \Theta\ b \in B.\ Q\ b)$›
⟨*proof*⟩

3.7.6 The Interrupt Operator

lemma *Interrupt-Mndetprefix*:

$\langle (\sqcap a \in A \rightarrow P\ a) \ \triangle\ Q = Q \ \Box\ (\sqcap a \in A \rightarrow P\ a \ \triangle\ Q) \rangle$
$\langle proof \rangle$

Chapter 4

CSPM Laws

4.0.1 The Throw Operator

lemma *Throw-read* :
‹*inj-on c A* $\Longrightarrow$ ($c?a \in A \rightarrow P\ a$) $\Theta\ a \in B.\ Q\ a =$
$c?a \in A \rightarrow$ (*if c a* $\in B$ *then Q* (*c a*) *else P a* $\Theta\ a \in B.\ Q\ a$)›
⟨*proof*⟩

lemma *Throw-ndet-write* :
‹*inj-on c A* $\Longrightarrow$ ($c!!a \in A \rightarrow P\ a$) $\Theta\ a \in B.\ Q\ a =$
$c!!a \in A \rightarrow$ (*if c a* $\in B$ *then Q* (*c a*) *else P a* $\Theta\ a \in B.\ Q\ a$)›
⟨*proof*⟩

lemma *Throw-write* :
‹($c!a \rightarrow P$) $\Theta\ a \in B.\ Q\ a = c!a \rightarrow$ (*if c a* $\in B$ *then Q* (*c a*) *else P* $\Theta\ a \in B.\ Q$ *a*)›
⟨*proof*⟩

lemma *Throw-write0* :
‹($a \rightarrow P$) $\Theta\ a \in B.\ Q\ a = a \rightarrow$ (*if a* $\in B$ *then Q a else P* $\Theta\ a \in B.\ Q\ a$)›
⟨*proof*⟩

4.0.2 The Interrupt Operator

lemma *Interrupt-read* :
‹($c?a \in A \rightarrow P\ a$) $\triangle\ Q = Q \Box (c?a \in A \rightarrow P\ a \triangle\ Q)$›
⟨*proof*⟩

lemma *Interrupt-ndet-write* :
‹($c!!a \in A \rightarrow P\ a$) $\triangle\ Q = Q \Box (c!!a \in A \rightarrow P\ a \triangle\ Q)$›
⟨*proof*⟩

lemma *Interrupt-write* : ‹($c!a \rightarrow P$) $\triangle\ Q = Q \Box (c!a \rightarrow P \triangle\ Q)$›
⟨*proof*⟩

lemma *Interrupt-write0* : ‹($a \rightarrow P$) $\triangle\ Q = Q \Box (a \rightarrow P \triangle\ Q)$›

⟨*proof*⟩

4.0.3 Global Deterministic Choice

lemma *GlobalDet-read* :
‹$\Box a \in A.\ c?b \in B\ a \to P\ a\ b = c?b \in (\bigcup a{\in}A.\ B\ a) \to \sqcap a{\in}\{a \in A.\ b \in B\ a\}.\ P\ a\ b$›
if ‹*inj-on* c $(\bigcup a{\in}A.\ B\ a)$›
⟨*proof*⟩

lemma *GlobalDet-write* :
‹$\Box a \in A.\ c!(b\ a) \to P\ a = c?x \in b\ `\ A \to \sqcap a{\in}\{a \in A.\ x = b\ a\}.\ P\ a$› **if** ‹*inj-on* $c$ $(b\ `\ A)$›
⟨*proof*⟩

lemma *GlobalDet-write0* :
‹$\Box a{\in}A.\ b\ a \to P\ a = \Box x \in (b\ `\ A) \to \sqcap a \in \{a \in A.\ x = b\ a\}.\ P\ a$›
⟨*proof*⟩

4.0.4 Multiple Synchronization Product

4.1 Results for Throw

4.1.1 Laws for Throw

lemma *Throw-GlobalDet* :
‹$(\Box\ a \in A.\ P\ a)\ \Theta\ b \in B.\ Q\ b = \Box\ a \in A.\ P\ a\ \Theta\ b \in B.\ Q\ b$› (**is** ‹*?lhs* = *?rhs*›)
⟨*proof*⟩

lemma *Throw-GlobalNdetR* :
‹$P\ \Theta\ a \in A.\ \sqcap b \in B.\ Q\ a\ b =$
(*if* $B = \{\}$ *then* $P\ \Theta\ a \in A.\ STOP$ *else* $\Box b \in B.\ P\ \Theta\ a \in A.\ Q\ a\ b$)›
(**is** ‹*?lhs* = (*if - then - else ?rhs*)›)
⟨*proof*⟩

corollary *Throw-Det* : ‹$P \Box P'\ \Theta\ a \in A.\ Q\ a = (P\ \Theta\ a \in A.\ Q\ a) \Box (P'\ \Theta\ a \in A.\ Q\ a)$›
⟨*proof*⟩

corollary *Throw-NdetR* : ‹$P\ \Theta\ a \in A.\ Q\ a \sqcap Q'\ a = (P\ \Theta\ a \in A.\ Q\ a) \Box (P\ \Theta\ a \in A.\ Q'\ a)$›
⟨*proof*⟩

4.1.2 Laws for Sync

lemma *Sync-GlobalNdet-cartprod*:

‹(⊓ (a, b) ∈ A × B. (P a ⟦S⟧ Q b)) =
(if A = {} ∨ B = {} then STOP else (⊓a ∈ A. P a) ⟦S⟧ (⊓b ∈ B. Q b))›
⟨proof⟩

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

lemma *MultiSync-Hiding-pseudo-distrib*:
‹*finite A* ⟹ *A* ∩ *S* = {} ⟹ (⟦*S*⟧ *p* ∈# *M*. (*P p* \ *A*)) = (⟦*S*⟧ *p* ∈# *M*. *P p*) \ *A*›
⟨*proof*⟩

lemma *MultiSync-prefix-pseudo-distrib*:
‹*M* ≠ {#} ⟹ *a* ∈ *S* ⟹ (⟦*S*⟧ *p* ∈# *M*. (*a* → *P p*)) = (*a* → (⟦*S*⟧ *p* ∈# *M*. *P p*))›
⟨*proof*⟩

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

A result on Mndetprefix and Sync.

lemma *Mndetprefix-Sync-distr*: ‹*A* ≠ {} ⟹ *B* ≠ {} ⟹
(⊓ *a* ∈ *A* → *P a*) ⟦*S*⟧ (⊓ *b* ∈ *B* → *Q b*) =
⊓ *a*∈*A*. ⊓ *b*∈*B*. (□*c* ∈ ({*a*} − *S*) → (*P a* ⟦*S*⟧ (*b* → *Q b*))) □
(□*d* ∈ ({*b*} − *S*) → ((*a* → *P a*) ⟦*S*⟧ *Q b*)) □
(□*c*∈({*a*} ∩ {*b*} ∩ *S*) → (*P a* ⟦*S*⟧ *Q b*))›
⟨*proof*⟩

lemma ‹*A* ≠ {} ⟹ *B* ≠ {} ⟹ (⊓ *a* ∈ *A* → *P a*) ⟦*S*⟧ (⊓ *b* ∈ *B* → *Q b*) =
⊓ *a*∈*A*. ⊓ *b*∈*B*. (*if a* ∈ *S then STOP else* (*a* → (*P a* ⟦*S*⟧ (*b* → *Q b*)))) □
(*if b* ∈ *S then STOP else* (*b* → ((*a* → *P a*) ⟦*S*⟧ *Q b*))) □
(*if a* = *b* ∧ *a* ∈ *S then* (*a* → (*P a* ⟦*S*⟧ *Q a*)) *else STOP*)›
⟨*proof*⟩

4.1.3 GlobalDet, GlobalNdet and write0

lemma *GlobalDet-write0-is-GlobalNdet-write0*:
‹(□ *p* ∈ *A*. (*a* → *P p*)) = ⊓ *p* ∈ *A*. (*a* → *P p*)› (**is** ‹*?lhs* = *?rhs*›)
⟨*proof*⟩

lemma *write0-GlobalNdet-bis*:
‹*A* ≠ {} ⟹ (*a* → (⊓ *p* ∈ *A*. *P p*) = □ *p* ∈ *A*. (*a* → *P p*))›
⟨*proof*⟩

4.2 Some Results on Renaming

lemma *Renaming-GlobalNdet*:
‹*Renaming* (⊓ *a* ∈ *A*. *P* (*f a*)) *f g* = ⊓ *b* ∈ *f* ‘ *A*. *Renaming* (*P b*) *f g›*
⟨*proof*⟩

lemma *Renaming-GlobalNdet-inj-on*:
‹*Renaming* (⊓ *a* ∈ *A*. *P a*) *f g* =
⊓ *b* ∈ *f* ‘ *A*. *Renaming* (*P* (*THE a*. *a* ∈ *A* ∧ *f a* = *b*)) *f g›*
if *inj-on-f*: ‹*inj-on f A›*
⟨*proof*⟩

corollary *Renaming-GlobalNdet-inj*:
‹*Renaming* (⊓ *a* ∈ *A*. *P a*) *f g* =
⊓ *b* ∈ *f* ‘ *A*. *Renaming* (*P* (*THE a*. *f a* = *b*)) *f g›* **if** *inj-f*: ‹*inj f›*
⟨*proof*⟩

lemma *Renaming-distrib-GlobalDet* :
‹*Renaming* (□*a* ∈ *A*. *P a*) *f g* = □*a* ∈ *A*. *Renaming* (*P a*) *f g›* (**is** ‹*?lhs* = *?rhs›*)
⟨*proof*⟩

lemma *Renaming-Mprefix-bis* :
‹*Renaming* (□*a* ∈ *A* → *P a*) *f g* = □*a* ∈ *A*. (*f a* → *Renaming* (*P a*) *f g*)›
⟨*proof*⟩

lemma *Renaming-GlobalDet-alt*:
‹*Renaming* (□ *a* ∈ *A*. *P* (*f a*)) *f g* = □ *b* ∈ *f* ‘ *A*. *Renaming* (*P b*) *f g›*
(**is** ‹*?lhs* = *?rhs›*)
⟨*proof*⟩

lemma *Renaming-GlobalDet-inj-on*:
‹*inj-on f A* ⟹ *Renaming* (□ *a* ∈ *A*. *P a*) *f g* =
□ *b* ∈ *f* ‘ *A*. *Renaming* (*P* (*THE a*. *a* ∈ *A* ∧ *f a* = *b*)) *f g›*
⟨*proof*⟩

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

lemma *Renaming-Interrupt* :
‹*Renaming* (*P* △ *Q*) *f g* = *Renaming P f g* △ *Renaming Q f g›* (**is** ‹*?lhs* = *?rhs›*)
⟨*proof*⟩

lemma *inj-on-Renaming-Throw* :
‹*Renaming* (*P* Θ *a* ∈ *A*. *Q a*) *f g* =
Renaming P f g Θ *b* ∈ *f* ‘ *A*. *Renaming* (*Q* (*inv-into A f b*)) *f g*›
(**is** ‹*?lhs* = *?rhs*›) **if** *inj-on-f* : ‹*inj-on f* (*events-of P* ∪ *A*)›
⟨*proof*⟩

4.2.1 *Renaming* and (\)

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

lemma *strict-mono-map*: ‹*strict-mono g* ⟹ *strict-mono* (λ*i*. *map f* (*g i*))›
⟨*proof*⟩

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

lemma *inj-on-map-event*$_{ptick}$*-set-T*:
‹*inj-on* (*map-event*$_{ptick}$ *f g*) (*set s*)› **if** ‹*inj-on f* (*events-of P*)› ‹*s* ∈ *T P*›
⟨*proof*⟩

theorem *bij-Renaming-Hiding*: ‹*Renaming* (*P* \ *S*) *f g* = *Renaming P f g* \ *f* ‘ *S*›
(**is** ‹*?lhs* = *?rhs*›) **if** *bij-f*: ‹*bij f*› **and** *bij-g* : ‹*bij g*›
⟨*proof*⟩

4.2.2 *Renaming* and *Sync*

Idem for the synchronization: when f is one to one, *Renaming* $(P \ [\![S]\!] \ Q)$ will behave as expected.

lemma *bij-map-setinterleaving-iff-setinterleaving* :
‹*map f r setinterleaves* ((*map f t*, *map f u*), *f* ‘ *S*) ⟷
r setinterleaves ((*t*, *u*), *S*)› **if** *bij-f* : ‹*bij f*›
⟨*proof*⟩

theorem *bij-Renaming-Sync*:
‹*Renaming* (*P* [*S*] *Q*) *f g* = *Renaming P f g* [*f* ‘ *S*] *Renaming Q f g*›
(**is** ‹*?lhs P Q* = *?rhs P Q*›) **if** *bij-f*: ‹*bij f*› **and** *bij-g* : ‹*bij g*›
⟨*proof*⟩

end

Chapter 5

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

5.1 Events

lemma *events-of-GlobalDet* :
‹$\alpha(\Box a \in A.\ P\ a) = (\bigcup a{\in}A.\ \alpha(P\ a))$›
⟨*proof*⟩

lemma *strict-events-of-GlobalDet-subset* : ‹$\boldsymbol{\alpha}(\Box a \in A.\ P\ a) \subseteq (\bigcup a{\in}A.\ \boldsymbol{\alpha}(P\ a))$›
⟨*proof*⟩

lemma *events-of-MultiSync-subset* :
‹$\alpha([\![S]\!]\ a \in\# M.\ P\ a) \subseteq (\bigcup a \in$ *set-mset* $M.\ \alpha(P\ a))$›
⟨*proof*⟩

lemma *events-of-MultiInter* :
‹$\alpha(|||\ a \in\# M.\ P\ a) = (\bigcup a \in$ *set-mset* $M.\ \alpha(P\ a))$›
⟨*proof*⟩

lemma *strict-events-of-MultiSync-subset* :
‹$\boldsymbol{\alpha}([\![S]\!]\ a \in\# M.\ P\ a) \subseteq (\bigcup a \in$ *set-mset* $M.\ \boldsymbol{\alpha}(P\ a))$›
⟨*proof*⟩

lemma *events-of-Throw-subset* :
‹$\alpha(P\ \Theta\ a \in A.\ Q\ a) \subseteq \alpha(P) \cup (\bigcup a \in A \cap \alpha(P).\ \alpha(Q\ a))$›
⟨*proof*⟩

lemma *events-of-Interrupt* : ‹$\alpha(P \triangle Q) = \alpha(P) \cup \alpha(Q)$›
⟨*proof*⟩

lemma *strict-events-of-Interrupt-subset* : ‹$\boldsymbol{\alpha}(P \triangle Q) \subseteq \boldsymbol{\alpha}(P) \cup \boldsymbol{\alpha}(Q)$›
⟨*proof*⟩

5.2 Ticks

lemma *ticks-of-GlobalDet*:
‹*ticks-of* $(\Box a \in A.\ P\ a) = (\bigcup a{\in}A.$ *ticks-of* $(P\ a))$›
⟨*proof*⟩

lemma *strict-ticks-of-GlobalDet-subset* : ‹$\checkmark\mathbf{s}(\Box a \in A.\ P\ a) \subseteq (\bigcup a{\in}A.\ \checkmark\mathbf{s}(P\ a))$›
⟨*proof*⟩

lemma *ticks-of-MultiSync-subset* :
‹$\checkmark s([\![S]\!]\ a \in\# M.\ P\ a) \subseteq (\bigcup a \in$ *set-mset* $M.\ \checkmark s(P\ a))$›
⟨*proof*⟩

lemma *strict-ticks-of-MultiSync-subset* :
‹$\checkmark\mathbf{s}([\![S]\!]\ a \in\# M.\ P\ a) \subseteq (\bigcap a \in$ *set-mset* $M.\ \checkmark\mathbf{s}(P\ a))$›
⟨*proof*⟩

lemma *ticks-Throw-subset* :
‹$\checkmark s(P\ \Theta\ a{\in}A.\ Q\ a) \subseteq \checkmark s(P) \cup (\bigcup a{\in}A \cap \alpha(P).\ \checkmark s(Q\ a))$›
⟨*proof*⟩

lemma *ticks-of-Interrupt* : ‹$\checkmark s(P \triangle Q) = \checkmark s(P) \cup \checkmark s(Q)$›
⟨*proof*⟩

lemma *strict-ticks-of-Interrupt-subset* : ‹$\checkmark\mathbf{s}(P \triangle Q) \subseteq \checkmark\mathbf{s}(P) \cup \checkmark\mathbf{s}(Q)$›
⟨*proof*⟩

events-of and *deadlock-free*

lemma *nonempty-events-of-if-deadlock-free*: ‹*deadlock-free* $P \Longrightarrow \alpha(P) \neq \{\}$›
⟨*proof*⟩

lemma *nonempty-strict-events-of-if-deadlock-free*: ‹*deadlock-free* $P \Longrightarrow \boldsymbol{\alpha}(P) \neq \{\}$›
⟨*proof*⟩

lemma *events-of-in-DF*: ‹$DF\ A \sqsubseteq_{FD}\ P \Longrightarrow \alpha(P) \subseteq A$›
⟨*proof*⟩

lemma *nonempty-events-of-if-deadlock-free*$_{SKIP}$:
‹*deadlock-free*$_{SKIPS}\ P \Longrightarrow (\exists\, r.\ [\checkmark(r)] \in \mathcal{T}\ P) \vee \alpha(P) \neq \{\}$›
⟨*proof*⟩

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

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

lemma ‹$chain\ Y \Longrightarrow \alpha(\bigsqcup i.\ Y\ i) = (\bigcup i.\ \alpha(Y\ i))$›
⟨*proof*⟩

lemma *f1* : ‹$chain\ Y \Longrightarrow \boldsymbol{\alpha}(\bigsqcup i.\ Y\ i) = (\bigcup i.\ \boldsymbol{\alpha}(Y\ i))$›
⟨*proof*⟩

find-theorems *Lub*

lemma *f2* : ‹$chain\ Y \Longrightarrow \mathcal{D}\ (Y\ i) = \{\} \Longrightarrow (\bigcup i.\ \boldsymbol{\alpha}(Y\ i)) = \boldsymbol{\alpha}(Y\ i)$›
⟨*proof*⟩

Chapter 6

Deadlock results

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

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

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

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

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

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

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

lemma *F-DF*:
‹$\mathcal{F}\ (DF\ A) =$
($if\ A = \{\}\ then\ \{(s, X).\ s = []\}$
$else\ (\bigcup a{\in}A.\ \{[]\} \times \{X.\ ev\ a \notin X\} \cup \{(ev\ a\ \#\ s, X) |\ s\ X.\ (s, X) \in \mathcal{F}\ (DF\ A)\})$›
⟨*proof*⟩

lemma *F-DF*$_{SKIPS}$:
‹$\mathcal{F}\ (DF_{SKIPS}\ A\ R) =$
($if\ A = \{\}\ then\ \{(s, X).\ s = [] \vee (\exists\, r{\in}R.\ s = [\checkmark(r)])\}$
$else\ (\bigcup a{\in}A.\ \{[]\} \times \{X.\ ev\ a \notin X\} \cup$
$\{(ev\ a\ \#\ s, X) |\ s\ X.\ (s, X) \in \mathcal{F}\ (DF_{SKIPS}\ A\ R)\}) \cup$
($if\ R = \{\}\ then\ \{(s, X).\ s = []\}$
$else\ \{([], X)\ |X.\ \exists\, r{\in}R.\ \checkmark(r) \notin X\} \cup \{(s, X).\ \exists\, r{\in}R.\ s = [\checkmark(r)]\})$›
⟨*proof*⟩

corollary *Cons-F-DF*:
‹$(x \# t,\ X) \in \mathcal{F}\ (DF\ A) \Longrightarrow (t,\ X) \in \mathcal{F}\ (DF\ A)$›
and *Cons-F-DF*$_{SKIPS}$:
‹$x \notin tick\ `\ R \Longrightarrow (x \# t,\ X) \in \mathcal{F}\ (DF_{SKIPS}\ A\ R) \Longrightarrow (t,\ X) \in \mathcal{F}\ (DF_{SKIPS}\ A\ R)$›
⟨*proof*⟩

lemma *D-DF*: ‹$\mathcal{D}\ (DF\ A) = (if\ A = \{\}\ then\ \{\}\ else\ \{ev\ a \# s |\ a\ s.\ a \in A \wedge s \in \mathcal{D}\ (DF\ A)\})$›
and *D-DF*$_{SKIPS}$: ‹$\mathcal{D}\ (DF_{SKIPS}\ A\ R) = (if\ A = \{\}\ then\ \{\}\ else\ \{ev\ a \# s |\ a\ s.\ a \in A \wedge s \in \mathcal{D}\ (DF_{SKIPS}\ A\ R)\})$›
⟨*proof*⟩

thm *T-SKIPS*[*of R*]
lemma *T-DF*:
‹$\mathcal{T}\ (DF\ A) = (if\ A = \{\}\ then\ \{[]\}\ else\ insert\ []\ \{ev\ a \# s |\ a\ s.\ a \in A \wedge s \in \mathcal{T}\ (DF\ A)\})$›
and *T-DF*$_{SKIPS}$:
‹$\mathcal{T}\ (DF_{SKIPS}\ A\ R) = (if\ A = \{\}\ then\ insert\ []\ \{[\checkmark(r)]\ |r.\ r \in R\}$
else $\{s.\ s = [] \vee (\exists\, r{\in}R.\ s = [\checkmark(r)]) \vee$
$s \neq [] \wedge (\exists\, a{\in}A.\ hd\ s = ev\ a \wedge tl\ s \in \mathcal{T}\ (DF_{SKIPS}\ A\ R))\})$›
⟨*proof*⟩

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

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

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

lemma ‹*deadlock-free*$_{SKIPS}$ $P \equiv DF_{SKIPS}\ UNIV\ UNIV \sqsubseteq_F P$›
⟨*proof*⟩

lemma *deadlock-free-F*: ‹*deadlock-free* $P \longleftrightarrow DF\ UNIV \sqsubseteq_F P$›
⟨*proof*⟩

lemma *deadlock-free-Mprefix-iff*: ‹*deadlock-free* $(\Box\ a \in A \rightarrow P\ a) \longleftrightarrow$
$A \neq \{\} \wedge (\forall\, a \in A.\ deadlock\text{-}free\ (P\ a))$›
and *deadlock-free*$_{SKIPS}$*-Mprefix-iff*: ‹*deadlock-free*$_{SKIPS}$ $(Mprefix\ A\ P) \longleftrightarrow$
$A \neq \{\} \wedge (\forall\, a \in A.\ deadlock\text{-}free_{SKIPS}\ (P\ a))$›
⟨*proof*⟩

lemma *deadlock-free-read-iff* :

‹*deadlock-free* ($c?a{\in}A \to P\ a$) $\longleftrightarrow$ $A \neq \{\} \wedge (\forall\, a{\in}c\ `\ A.$ *deadlock-free* ((*P* ∘ *inv-into A c*) *a*))›
and *deadlock-free*$_{SKIPS}$*-read-iff* :
‹*deadlock-free*$_{SKIPS}$ ($c?a{\in}A \to P\ a$) $\longleftrightarrow$ $A \neq \{\} \wedge (\forall\, a{\in}c\ `\ A.$ *deadlock-free*$_{SKIPS}$ ((*P* ∘ *inv-into A c*) *a*))›
⟨*proof*⟩

lemma *deadlock-free-read-inj-on-iff* :
‹*inj-on c A* $\Longrightarrow$ *deadlock-free* ($c?a{\in}A \to P\ a$) $\longleftrightarrow$ $A \neq \{\} \wedge (\forall\, a{\in}A.$ *deadlock-free* (*P a*))›
and *deadlock-free*$_{SKIPS}$*-read-inj-on-iff* :
‹*inj-on c A* $\Longrightarrow$ *deadlock-free*$_{SKIPS}$ ($c?a{\in}A \to P\ a$) $\longleftrightarrow$ $A \neq \{\} \wedge (\forall\, a{\in}A.$ *deadlock-free*$_{SKIPS}$ (*P a*))›
⟨*proof*⟩

lemma *deadlock-free-write-iff* :
‹*deadlock-free* ($c!a \to P$) $\longleftrightarrow$ *deadlock-free P*›
and *deadlock-free*$_{SKIPS}$*-write-iff* :
‹*deadlock-free*$_{SKIPS}$ ($c!a \to P$) $\longleftrightarrow$ *deadlock-free*$_{SKIPS}$ *P*›
⟨*proof*⟩

lemma *deadlock-free-write0-iff* :
‹*deadlock-free* ($a \to P$) $\longleftrightarrow$ *deadlock-free P*›
and *deadlock-free*$_{SKIPS}$*-write0-iff* :
‹*deadlock-free*$_{SKIPS}$ ($a \to P$) $\longleftrightarrow$ *deadlock-free*$_{SKIPS}$ *P*›
⟨*proof*⟩

lemma *deadlock-free-GlobalNdet-iff*: ‹*deadlock-free* ($\sqcap\ a \in A.\ P\ a$) $\longleftrightarrow$
$A \neq \{\} \wedge (\forall\ a \in A.$ *deadlock-free* (*P a*))›
and *deadlock-free*$_{SKIPS}$*-GlobalNdet-iff*: ‹*deadlock-free*$_{SKIPS}$ ($\sqcap\ a \in A.\ P\ a$) $\longleftrightarrow$
$A \neq \{\} \wedge (\forall\ a \in A.$ *deadlock-free*$_{SKIPS}$ (*P a*))›
⟨*proof*⟩

lemma *deadlock-free-Mndetprefix-iff*: ‹*deadlock-free* ($\sqcap\ a \in A \to P\ a$) $\longleftrightarrow$
$A \neq \{\} \wedge (\forall\, a{\in}A.$ *deadlock-free* (*P a*))›
and *deadlock-free*$_{SKIPS}$*-Mndetprefix-iff*: ‹*deadlock-free*$_{SKIPS}$ ($\sqcap\ a \in A \to P\ a$) $\longleftrightarrow$
$A \neq \{\} \wedge (\forall\, a{\in}A.$ *deadlock-free*$_{SKIPS}$ (*P a*))›
⟨*proof*⟩

lemma *deadlock-free-Ndet-iff*: ‹*deadlock-free* ($P \sqcap Q$) $\longleftrightarrow$

$\mathit{deadlock\text{-}free}\ P \wedge \mathit{deadlock\text{-}free}\ Q\rangle$
and $\mathit{deadlock\text{-}free}_{SKIPS}$*-Ndet-iff*: $\langle \mathit{deadlock\text{-}free}_{SKIPS}\ (P \sqcap Q) \longleftrightarrow \mathit{deadlock\text{-}free}_{SKIPS}\ P \wedge \mathit{deadlock\text{-}free}_{SKIPS}\ Q\rangle$
⟨*proof*⟩

lemma *deadlock-free-is-right*:
$\langle \mathit{deadlock\text{-}free}\ (P :: ('a, 'r)\ \mathit{process}_{ptick}) \longleftrightarrow (\forall s \in \mathcal{T}\ P.\ \mathit{tickFree}\ s \wedge (s, \mathit{UNIV}) \notin \mathcal{F}\ P)\rangle$
$\langle \mathit{deadlock\text{-}free}\ P \longleftrightarrow (\forall s \in \mathcal{T}\ P.\ \mathit{tickFree}\ s \wedge (s, \mathit{ev}\ `\ \mathit{UNIV}) \notin \mathcal{F}\ P)\rangle$
⟨*proof*⟩

lemma $\langle \mathit{deadlock\text{-}free}\ (P \Box Q) \longleftrightarrow P = \mathit{STOP} \wedge \mathit{deadlock\text{-}free}\ Q \vee \mathit{deadlock\text{-}free}\ P \wedge Q = \mathit{STOP}\rangle$

⟨*proof*⟩

lemma *deadlock-free-GlobalDet-iff* :
$\langle [\![A \neq \{\}; \mathit{finite}\ A; \forall a{\in}A.\ \mathit{deadlock\text{-}free}\ (P\ a)]\!] \Longrightarrow \mathit{deadlock\text{-}free}\ (\Box a \in A.\ P\ a)\rangle$
and $\mathit{deadlock\text{-}free}_{SKIPS}$*-MultiDet*:
$\langle [\![A \neq \{\}; \mathit{finite}\ A; \forall a{\in}A.\ \mathit{deadlock\text{-}free}_{SKIPS}\ (P\ a)]\!] \Longrightarrow \mathit{deadlock\text{-}free}_{SKIPS}\ (\Box a \in A.\ P\ a)\rangle$
⟨*proof*⟩

lemma *deadlock-free-Det*:
$\langle \mathit{deadlock\text{-}free}\ P \Longrightarrow \mathit{deadlock\text{-}free}\ Q \Longrightarrow \mathit{deadlock\text{-}free}\ (P \Box Q)\rangle$
and $\mathit{deadlock\text{-}free}_{SKIPS}$*-Det*:
$\langle \mathit{deadlock\text{-}free}_{SKIPS}\ P \Longrightarrow \mathit{deadlock\text{-}free}_{SKIPS}\ Q \Longrightarrow \mathit{deadlock\text{-}free}_{SKIPS}\ (P \Box Q)\rangle$
⟨*proof*⟩

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

lemma
$\langle \exists P\ Q.\ \mathit{deadlock\text{-}free}\ P \wedge \neg\ \mathit{deadlock\text{-}free}\ Q \wedge \mathit{deadlock\text{-}free}\ (P \Box Q)\rangle$
$\langle \exists P\ Q.\ \mathit{deadlock\text{-}free}_{SKIPS}\ P \wedge \neg\ \mathit{deadlock\text{-}free}_{SKIPS}\ Q \wedge \mathit{deadlock\text{-}free}_{SKIPS}\ (P \Box Q)\rangle$
⟨*proof*⟩

lemma *FD-Mndetprefix-iff*:
‹$A \neq \{\} \Longrightarrow P \sqsubseteq_{FD} \sqcap a \in A \rightarrow Q \longleftrightarrow (\forall\, a \in A.\ P \sqsubseteq_{FD} (a \rightarrow Q))$›
⟨*proof*⟩

lemma *Mndetprefix-FD*: ‹$(\exists\, a \in A.\ (a \rightarrow Q) \sqsubseteq_{FD} P) \Longrightarrow \sqcap a \in A \rightarrow Q \sqsubseteq_{FD} P$›
⟨*proof*⟩

Mprefix, *Sync* and *deadlock-free*

lemma *Mprefix-Sync-deadlock-free*:
assumes *not-all-empty*: ‹$A \neq \{\} \vee B \neq \{\} \vee A' \cap B' \neq \{\}$›
and ‹$A \cap S = \{\}$› **and** ‹$A' \subseteq S$› **and** ‹$B \cap S = \{\}$› **and** ‹$B' \subseteq S$›
and ‹$\forall\, x{\in}A.$ *deadlock-free* $(P\ x\ [\![S]\!]$ *Mprefix* $(B \cup B')\ Q)$›
and ‹$\forall\, y{\in}B.$ *deadlock-free* $(\mathit{Mprefix}\ (A \cup A')\ P\ [\![S]\!]\ Q\ y)$›
and ‹$\forall\, x{\in}A' \cap B'.$ *deadlock-free* $((P\ x\ [\![S]\!]\ Q\ x))$›
shows ‹*deadlock-free* $(\mathit{Mprefix}\ (A \cup A')\ P\ [\![S]\!]\ \mathit{Mprefix}\ (B \cup B')\ Q)$›
⟨*proof*⟩

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

6.3 Results on *Renaming*

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

6.3.1 Behaviour with references processes

For *DF*

lemma *DF-FD-Renaming-DF*: ‹$\mathit{DF}\ (f\ `\ A) \sqsubseteq_{FD} \mathit{Renaming}\ (\mathit{DF}\ A)\ f\ g$›
⟨*proof*⟩

lemma *Renaming-DF-FD-DF*: ‹$\mathit{Renaming}\ (\mathit{DF}\ A)\ f\ g \sqsubseteq_{FD} \mathit{DF}\ (f\ `\ A)$›
if *finitary*: ‹*finitary* f› ‹*finitary* g›
⟨*proof*⟩

For DF_{SKIPS}

lemma *Renaming-SKIPS* [*simp*] : ‹*Renaming* (*SKIPS R*) *f g* = *SKIPS* (*g* ‘ *R*)›
⟨*proof*⟩

lemma DF_{SKIPS}-*FD-Renaming-*DF_{SKIPS}:
‹DF_{SKIPS} (*f* ‘ *A*) (*g* ‘ *R*) $\sqsubseteq_{FD}$ *Renaming* (DF_{SKIPS} *A R*) *f g*›
⟨*proof*⟩

lemma *Renaming-*DF_{SKIPS}-*FD-*DF_{SKIPS}:
‹*Renaming* (DF_{SKIPS} *A R*) *f g* $\sqsubseteq_{FD}$ DF_{SKIPS} (*f* ‘ *A*) (*g* ‘ *R*)›
if *finitary*: ‹*finitary f*› ‹*finitary g*›
⟨*proof*⟩

For *RUN*

lemma *RUN-FD-Renaming-RUN*: ‹*RUN* (*f* ‘ *A*) $\sqsubseteq_{FD}$ *Renaming* (*RUN A*) *f g*›
⟨*proof*⟩

lemma *Renaming-RUN-FD-RUN*: ‹*Renaming* (*RUN A*) *f g* $\sqsubseteq_{FD}$ *RUN* (*f* ‘ *A*)›
if *finitary*: ‹*finitary f*› ‹*finitary g*›
⟨*proof*⟩

For *CHAOS*

lemma *CHAOS-FD-Renaming-CHAOS*:
‹*CHAOS* (*f* ‘ *A*) $\sqsubseteq_{FD}$ *Renaming* (*CHAOS A*) *f g*›
⟨*proof*⟩

lemma *Renaming-CHAOS-FD-CHAOS*:
‹*Renaming* (*CHAOS A*) *f g* $\sqsubseteq_{FD}$ *CHAOS* (*f* ‘ *A*)›
if *finitary*: ‹*finitary f*› ‹*finitary g*›
⟨*proof*⟩

For $CHAOS_{SKIPS}$

lemma $CHAOS_{SKIPS}$-*FD-Renaming-*$CHAOS_{SKIPS}$:
‹$CHAOS_{SKIPS}$ (*f* ‘ *A*) (*g* ‘ *R*) $\sqsubseteq_{FD}$ *Renaming* ($CHAOS_{SKIPS}$ *A R*) *f g*›
⟨*proof*⟩

lemma *Renaming-*$CHAOS_{SKIPS}$-*FD-*$CHAOS_{SKIPS}$:
‹*Renaming* ($CHAOS_{SKIPS}$ *A R*) *f g* $\sqsubseteq_{FD}$ $CHAOS_{SKIPS}$ (*f* ‘ *A*) (*g* ‘ *R*)›
if *finitary*: ‹*finitary f*› ‹*finitary g*›
⟨*proof*⟩

6.3.2 Corollaries on *deadlock-free* and $deadlock\text{-}free_{SKIPS}$

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

and *Renaming-CHAOS* =
FD-antisym[*OF Renaming-CHAOS-FD-CHAOS CHAOS-FD-Renaming-CHAOS*]
and $Renaming\text{-}CHAOS_{SKIPS}$ =
$FD\text{-}antisym[OF\ Renaming\text{-}CHAOS_{SKIPS}\text{-}FD\text{-}CHAOS_{SKIPS}\ CHAOS_{SKIPS}\text{-}FD\text{-}Renaming\text{-}CHAOS_{SKIPS}]$

lemma *deadlock-free-imp-deadlock-free-Renaming*: ‹*deadlock-free* (*Renaming P f g*)›
if ‹*deadlock-free P*›
⟨*proof*⟩

lemma *deadlock-free-Renaming-imp-deadlock-free*: ‹*deadlock-free P*›
if ‹*inj f*› **and** ‹*inj g*› **and** ‹*deadlock-free* (*Renaming P f g*)›
⟨*proof*⟩

corollary *deadlock-free-Renaming-iff*:
‹*inj f* ⟹ *inj g* ⟹ *deadlock-free* (*Renaming P f g*) ⟷ *deadlock-free P*›
⟨*proof*⟩

lemma $deadlock\text{-}free_{SKIPS}\text{-}imp\text{-}deadlock\text{-}free_{SKIPS}\text{-}Renaming$:
‹$deadlock\text{-}free_{SKIPS}$ *P* ⟹ $deadlock\text{-}free_{SKIPS}$ (*Renaming P f g*)›
⟨*proof*⟩

lemma $deadlock\text{-}free_{SKIPS}\text{-}Renaming\text{-}imp\text{-}deadlock\text{-}free_{SKIPS}$:
‹$deadlock\text{-}free_{SKIPS}$ *P*› **if** ‹*inj f*› **and** ‹*inj g*› **and** ‹$deadlock\text{-}free_{SKIPS}$ (*Renaming P f g*)›
⟨*proof*⟩

corollary $deadlock\text{-}free_{SKIPS}\text{-}Renaming\text{-}iff$:
‹*inj f* ⟹ *inj g* ⟹ $deadlock\text{-}free_{SKIPS}$ (*Renaming P f g*) ⟷ $deadlock\text{-}free_{SKIPS}$ *P*›
⟨*proof*⟩

6.4 The big results

6.4.1 An interesting equivalence

lemma *deadlock-free-of-Sync-iff-DF-FD-DF-Sync-DF*:
‹(∀ *P Q*. *deadlock-free* (*P*::('*a*, '*r*) $process_{ptick}$) ⟶ *deadlock-free Q* ⟶
deadlock-free (*P* ⟦*S*⟧ *Q*))
⟷ (*DF UNIV* :: ('*a*, '*r*) $process_{ptick}$) ⊑$_{FD}$ (*DF UNIV* ⟦*S*⟧ *DF UNIV*)› (**is** ‹*?lhs* ⟷ *?rhs*›)
⟨*proof*⟩

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

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

lemma *DF-FD-DF-Sync-STOP-or-SKIP-iff*:
$\langle(\textit{DF}\ A \sqsubseteq_{FD} \textit{DF}\ A\ [\![S]\!]\ P) \longleftrightarrow A \cap S = \{\}\rangle$
if *P-disj*: $\langle P = \textit{STOP} \vee P = \textit{SKIP}\ r\rangle$
⟨*proof*⟩

lemma *DF-Sync-STOP-or-SKIP-FD-DF*: $\langle \textit{DF}\ A\ [\![S]\!]\ P \sqsubseteq_{FD} \textit{DF}\ A\rangle$
if *P-disj*: $\langle P = \textit{STOP} \vee P = \textit{SKIP}\ r\rangle$ **and** *empty-inter*: $\langle A \cap S = \{\}\rangle$
⟨*proof*⟩

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

6.4.3 Finally, *deadlock-free* $(P \mid\mid\mid Q)$

theorem *DF-F-DF-Sync-DF*: $\langle(\textit{DF}\ (A \cup B) :: ('a, 'r)\ \textit{process}_{ptick}) \sqsubseteq_F \textit{DF}\ A\ [\![S]\!]\ \textit{DF}\ B\rangle$
if *nonempty*: $\langle A \neq \{\} \wedge B \neq \{\}\rangle$
and *intersect-hyp*: $\langle B \cap S = \{\} \vee (\exists\, y.\ B \cap S = \{y\} \wedge A \cap S \subseteq \{y\})\rangle$
⟨*proof*⟩

lemma *DF-FD-DF-Sync-DF*:
$\langle A \neq \{\} \wedge B \neq \{\} \Longrightarrow B \cap S = \{\} \vee (\exists\, y.\ B \cap S = \{y\} \wedge A \cap S \subseteq \{y\}) \Longrightarrow$
$\textit{DF}\ (A \cup B) \sqsubseteq_{FD} \textit{DF}\ A\ [\![S]\!]\ \textit{DF}\ B\rangle$
⟨*proof*⟩

theorem *DF-FD-DF-Sync-DF-iff*:
$\langle \textit{DF}\ (A \cup B) \sqsubseteq_{FD} \textit{DF}\ A\ [\![S]\!]\ \textit{DF}\ B \longleftrightarrow$
(*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** ‹*?FD-ref* $\longleftrightarrow$ (*if* $A = \{\}$ *then* $B \cap S = \{\}$

else if $B = \{\}$ *then* $A \cap S = \{\}$
else ?cases)›)

⟨*proof*⟩

lemma
‹$(\forall\, a \in A.\ X\ a \cap S = \{\} \vee (\forall\, b \in A.\ \exists\, y.\ X\ a \cap S = \{y\} \wedge X\ b \cap S \subseteq \{y\}))$
$\longleftrightarrow (\forall\, a \in A.\ \forall\, b \in A.\ \exists\, y.\ (X\ a \cup X\ b) \cap S \subseteq \{y\})$›
— this is the reason we write ugly_hyp this way
⟨*proof*⟩

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

⟨*proof*⟩

lemma *DF-FD-DF-MultiSync-DF′*:
‹$[\![$*finite A*; $\forall\, a \in A.\ X\ a \neq \{\}$; $\forall\, a \in A.\ \forall\, b \in A.\ \exists\, y.\ (X\ a \cup X\ b) \cap S \subseteq \{y\}]\!]$
$\Longrightarrow DF\ (\bigcup\ a \in A.\ X\ a) \sqsubseteq_{FD} [\![S]\!]\ a \in\#\ mset\text{-}set\ A.\ DF\ (X\ a)$›
⟨*proof*⟩

lemmas *DF-FD-DF-MultiInter-DF* =
DF-FD-DF-MultiSync-DF′[**where** $S = \{\}$, *simplified*]
and *DF-FD-DF-MultiPar-DF* =
DF-FD-DF-MultiSync-DF [**where** $S = UNIV$, *simplified*]
and *DF-FD-DF-MultiPar-DF′* =
DF-FD-DF-MultiSync-DF′[**where** $S = UNIV$, *simplified*]

lemma ‹$DF\ \{a\} = DF\ \{a\}\ [\![S]\!]\ STOP \longleftrightarrow a \notin S$›
⟨*proof*⟩

lemma ‹$DF\ \{a\}\ [\![S]\!]\ STOP = STOP \longleftrightarrow a \in S$›
⟨*proof*⟩

corollary *DF-FD-DF-Inter-DF*: ‹*DF A* $\sqsubseteq_{FD}$ *DF A* ||| *DF A*›
⟨*proof*⟩

corollary *DF-UNIV-FD-DF-UNIV-Inter-DF-UNIV*:
‹*DF UNIV* $\sqsubseteq_{FD}$ *DF UNIV* ||| *DF UNIV*›
⟨*proof*⟩

corollary *Inter-deadlock-free*:
‹*deadlock-free P* ⟹ *deadlock-free Q* ⟹ *deadlock-free* (*P* ||| *Q*)›
⟨*proof*⟩

theorem *MultiInter-deadlock-free*:
‹⟦*M* ≠ {#}; ⋀*m*. *m* ∈# *M* ⟹ *deadlock-free* (*P m*)⟧ ⟹
deadlock-free (||| *p* ∈# *M*. *P p*)›
⟨*proof*⟩

Chapter 7

The Main Entry Point

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

Chapter 8

Example: Dining Philosophers

8.1 Classic version

We formalize here the Dining Philosophers problem with a locale.

locale *DiningPhilosophers* =

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

begin

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

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

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

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

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

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

Now we use the architectural operators for modelling the interleaving of the

philosophers, and the interleaving of the forks.

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

corollary ‹*N* = *3* ⟹ *PHILS* = (*LPHIL0* ||| *RPHIL 1* ||| *RPHIL 2*)›
— just a test
⟨*proof*⟩

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

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

end

8.2 Formalization with fixrec package

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

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

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

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

instance ⟨*proof*⟩

end

locale *DiningPhilosophers-fixrec* =

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

begin

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

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

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

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

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

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

corollary ‹*N* = *3* ⟹ *PHILS* = (*LPHIL0* ||| *RPHIL*·*1* ||| *RPHIL*·*2*)›
— just a test
⟨*proof*⟩

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

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

end

Chapter 9

Example: Plain Old Telephone System

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

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

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

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

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

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

instance ⟨*proof*⟩

end

9.1 The Alphabet and Basic Types of POTS

Underlying terminology apparent in the acronyms:

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

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

type-synonym *Phones* = ‹*int*›

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

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

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

lemma *nonempty-phones*[*simp*]: ‹*phones* ≠ {}›
and *finite-phones*[*simp*]: ‹*finite phones*›
and *at-least-two-phones*[*simp*]: ‹*2* ≤ *card phones*›
and *not-singl-phone*[*simp*]: ‹*phones* − {*p*} ≠ {}›
⟨*proof*⟩

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

9.2 Auxilliaries to Substructure the Specification

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

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

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

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

$\rightarrow$ *tcT*!(*ts*,*Tdiscon-t*,*os*)
$\rightarrow$ *EndReject*!*ts*
$\rightarrow$ *IT*)
))
$\Box$ (*on-hook*!*ts*
$\rightarrow$ *tcT*!(*ts*,*Tdiscon-t*,*os*)
$\rightarrow$ *ctT*!(*ts*,*Tdiscon-o*,*os*)
$\rightarrow$ *EndReject*!*ts*
$\rightarrow$*IT*)›

9.3 A Telephone

fixrec *T* :: ‹*Phones* $\rightarrow$ *channels process*›
and *Oside* :: ‹*Phones* $\rightarrow$ *channels process*›
and *Tside* :: ‹*Phones* $\rightarrow$ *channels process*›
and *NoReject* :: ‹*Phones* $\rightarrow$ *channels process*›
and *Reject* :: ‹*Phones* $\rightarrow$ *channels process*›
where
T-rec [*simp del*]: ‹*T*·*ts* = (*Tside*·*ts* ; *T*·*ts*) $\triangleright$ (*Oside*·*ts* ; *T*·*ts*)›
| *Oside-rec* [*simp del*]: ‹*Oside*·*ts* = *StartReject*!*ts*
$\rightarrow$ *tcO*!(*ts*,*Osetup*)
$\rightarrow$ ($\sqcap$ *p* $\in$ *phones*. *Oside1 ts p*)›
| *Tside-rec* [*simp del*]: ‹*Tside*·*ts* = *ctT*?(*y*,*z*,*os*)|((*y*,*z*)=(*ts*,*Tsetup*))
$\rightarrow$ *StartReject*!*ts*
$\rightarrow$ (*tcT*!(*ts*,*Talert*,*os*)
$\rightarrow$ *tcT*!(*ts*,*Tconnect*,*os*)
$\rightarrow$(*Tside-connected ts os*)
$\sqcap$ (*tcT*!(*ts*,*Tconnect*,*os*)
$\rightarrow$ (*Tside-connected ts os*)))›
| *NoReject-rec* [*simp del*]: ‹*NoReject*·*ts* = *StartReject*!*ts* $\rightarrow$ *Reject*·*ts*›
| *Reject-rec* [*simp del*]: ‹*Reject*·*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*·*ts*)
$\Box$ (*EndReject*!*ts* $\rightarrow$ *NoReject*·*ts*)›

definition *Tel*:: ‹*Phones* $\Rightarrow$ *channels process*›
where ‹*Tel p* $\equiv$ (*T*·*p* $[\![\{$*StartReject p*, *EndReject p*$\}]\!]$ *NoReject*·*p*) $\setminus$ {*StartReject p*, *EndReject p*}›

9.4 A Connector with the Network

fixrec *Call* :: ‹*Phones* $\rightarrow$ *channels process*›
and *BUSY* :: ‹*Phones* $\rightarrow$ *Phones* $\rightarrow$ *channels process*›
and *Connected* :: ‹*Phones* $\rightarrow$ *Phones* $\rightarrow$ *channels process*›

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

9.5 Combining NETWORK and TELEPHONES to a SYSTEM

definition *NETWORK* :: ‹*channels process*›
where ‹*NETWORK* ≡ (||| *os* ∈# (*mset-set phones*). *Call*·*os*)›

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

definition *SYSTEM* :: ‹*channels process*›
where ‹*SYSTEM* ≡ *NETWORK* ⟦*VisibleEvents*⟧ *TELEPHONES*›

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

9.6 A simple Model of a User

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

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

definition *ImplementT* :: ‹*Phones* ⇒ *channels process*›

where $\langle ImplementT\ ts \equiv ((\ Tel\ ts)\ [\![EventsIPhone\ ts \cup EventsUser\ ts]\!]\ (User{\cdot}ts)) \setminus (EventsIPhone\ ts \cup EventsUser\ ts)\rangle$

9.7 Toplevel Proof-Goals

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

lemma $\langle \forall\ p \in phones.\ deadlock\text{-}free\ (Tel\ p)\rangle\ \langle proof\rangle$
lemma $\langle \forall\ p \in phones.\ deadlock\text{-}free\text{-}v2\ (Call{\cdot}p)\rangle\ \langle proof\rangle$
lemma $\langle deadlock\text{-}free\text{-}v2\ NETWORK\rangle\ \langle proof\rangle$
lemma $\langle deadlock\text{-}free\text{-}v2\ SYSTEM\rangle\ \langle proof\rangle$
lemma $\langle lifelock\text{-}free\ SYSTEM\rangle\ \langle proof\rangle$
lemma $\langle \forall\ p \in phones.\ lifelock\text{-}free\ (ImplementT\ p)\rangle\ \langle proof\rangle$
lemma $\langle \forall\ p \in phones.\ Tel\ p \sqsubseteq_{F\,D}\ ImplementT\ p\rangle\ \langle proof\rangle$

lemma $\langle \forall\ p \in phones.\ Tel'{\cdot}p\ \sqsubseteq_F\ RUN\ UNIV\rangle\ \langle proof\rangle$

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

end

Chapter 10

Conclusion

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

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

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

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

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

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

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

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

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

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

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

We presented two examples: Dining philosophers, and POTS.
In both, we underlined the usefulness of the architectural operators for modeling complex systems.

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

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

This theorem allows, for example, to establish:

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

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

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

Bibliography

[1] S. D. Brookes and A. W. Roscoe. An improved failures model for communicating processes. In S. D. Brookes, A. W. Roscoe, and G. Winskel, editors, *Seminar on Concurrency*, pages 281–305, Berlin, Heidelberg, 1985. Springer Berlin Heidelberg.

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

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

[4] S. Taha, L. Ye, and B. Wolff. Hol-csp version 2.0. *Archive of Formal Proofs*, April 2019. https://isa-afp.org/entries/HOL-CSP.html, Formal proof development.