HOL-CSP Version 2.0

Burkhart Wolff

October 14, 2012

Contents

theory *Directed*
imports *HOLCF*
begin

0.1 Directed sets

default-sort *type*

definition *directed* :: *'a::po set* ⇒ *bool* **where**
directed S ⟷ (∃ *x*. *x* ∈ *S*) ∧ (∀ *x*∈*S*. ∀ *y*∈*S*. ∃ *z*∈*S*. *x* ⊑ *z* ∧ *y* ⊑ *z*)

lemma *directedI*:
assumes ∃ *z*. *z* ∈ *S*
assumes ⋀*x y*. ⟦*x* ∈ *S*; *y* ∈ *S*⟧ ⟹ ∃ *z*∈*S*. *x* ⊑ *z* ∧ *y* ⊑ *z*
shows *directed S*
unfolding *directed-def* **using** *assms* **by** *fast*

lemma *directedD1*: *directed S* ⟹ ∃ *z*. *z* ∈ *S*
unfolding *directed-def* **by** *fast*

lemma *directedD2*: ⟦*directed S*; *x* ∈ *S*; *y* ∈ *S*⟧ ⟹ ∃ *z*∈*S*. *x* ⊑ *z* ∧ *y* ⊑ *z*
unfolding *directed-def* **by** *fast*

lemma *directedE1*:
assumes *S*: *directed S*
obtains *z* **where** *z* ∈ *S*
by (*insert directedD1* [*OF S*], *fast*)

lemma *directedE2*:
assumes S: *directed* S
assumes x: $x \in S$ **and** y: $y \in S$
obtains z **where** $z \in S$ $x \sqsubseteq z$ $y \sqsubseteq z$
by (*insert directedD2* [*OF* S x y], *fast*)

lemma *directed-finiteI*:
assumes U: $\bigwedge U.\ [\![\mathit{finite}\ U;\ U \subseteq S]\!] \Longrightarrow \exists z{\in}S.\ U <| z$
shows *directed* S
proof (*rule directedI*)
have *finite* $\{\}$ **and** $\{\} \subseteq S$ **by** *simp-all*
hence $\exists z{\in}S.\ \{\} <| z$ **by** (*rule* U)
thus $\exists z.\ z \in S$ **by** *simp*
next
fix x y
assume $x \in S$ **and** $y \in S$
hence *finite* $\{x, y\}$ **and** $\{x, y\} \subseteq S$ **by** *simp-all*
hence $\exists z{\in}S.\ \{x, y\} <| z$ **by** (*rule* U)
thus $\exists z{\in}S.\ x \sqsubseteq z \wedge y \sqsubseteq z$ **by** *simp*
qed

lemma *directed-finiteD*:
assumes S: *directed* S
shows $[\![\mathit{finite}\ U;\ U \subseteq S]\!] \Longrightarrow \exists z{\in}S.\ U <| z$
proof (*induct* U *set*: *finite*)
case *empty*
from S **have** $\exists z.\ z \in S$ **by** (*rule directedD1*)
thus $\exists z{\in}S.\ \{\} <| z$ **by** *simp*
next
case (*insert* x F)
from ‹*insert* x $F \subseteq S$›
have xS: $x \in S$ **and** FS: $F \subseteq S$ **by** *simp-all*
from FS **have** $\exists y{\in}S.\ F <| y$ **by** *fact*
then obtain y **where** yS: $y \in S$ **and** Fy: $F <| y$ **..**
obtain z **where** zS: $z \in S$ **and** xz: $x \sqsubseteq z$ **and** yz: $y \sqsubseteq z$
using S xS yS **by** (*rule directedE2*)
from Fy yz **have** $F <| z$ **by** (*rule is-ub-upward*)
with xz **have** *insert* x $F <| z$ **by** *simp*
with zS **show** $\exists z{\in}S.$ *insert* x $F <| z$ **..**
qed

lemma *not-directed-empty* [*simp*]: ¬ *directed* $\{\}$
by (*rule notI*, *drule directedD1*, *simp*)

lemma *directed-singleton*: *directed* $\{x\}$
by (*rule directedI*, *auto*)

lemma *directed-bin*: $x \sqsubseteq y \Longrightarrow$ *directed* $\{x, y\}$

by (*rule directedI*, *auto*)

lemma *directed-chain*: *chain S* $\Longrightarrow$ *directed* (*range S*)
apply (*rule directedI*)
apply (*rule-tac x*=*S 0* **in** *exI*, *simp*)
apply (*clarify*, *rename-tac m n*)
apply (*rule-tac x*=*S* (*max m n*) **in** *bexI*)
apply (*simp add*: *chain-mono*)
apply *simp*
done

end

1 Hoare/Roscoe's Denotational Semantics for CSP The Notion of Processes

theory *Process*
imports *HOLCF Directed*
begin

ML$\langle\!\langle$ *quick-and-dirty*:=*true*$\rangle\!\rangle$

This 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 [1],and the semantic details in a joint Paper of Roscoe and Brooks "An improved failures model for communicating processes", in Proceedings of the Pittsburgh seminar on concurrency, Springer LNCS 197 (1985), 281-305. This work revealed minor, but omnipresent foundational errors in key concepts like the process invariant that were revealed by a first formalization in Isabelle/HOL, called HOL-CSP 1.0 [2].

In contrast to HOL-CSP 1.0, which came with an own fixpoint theory partly inspired by previous work of Franz Regensburger and developed by myself, it is the goal of this redesign of the HOL-CSP theory to reuse the HOLCF theory that emmerged from Franz'ens work. Thus, the footprint of this theory should be reduced drastically. Moreover, all proofs have been heavily revised or re-constructed to reflect the drastically improved state of the art of interactive theory development with Isabelle.

The following merely technical command has the purpose to undo a default setting of HOLCF.

default-sort *type*

1.1 Pre-Requisite: Basic Traces and tick-Freeness

The denotational semantics of CSP assumes a distinguishable special event, called `tick` and written ?, that is required to occur only in the end in order to signalize succesful termination of a process. (In the original text of Hoare, this treatment was more liberal and lead to foundational problems: the process invariant could not be established for the sequential composition operator of CSP; see [2] for details.)

datatype $'\alpha$ *event* = *ev* $'\alpha$ | *tick*

type-synonym $'\alpha$ *trace* = ($'\alpha$ *event*) *list*

We chose as standard ordering on traces the prefix ordxering.

instantiation *list* :: (*type*) *order*
begin

definition *le-list-def* : $s \leq t \longleftrightarrow (\exists\ r.\ s\ @\ r = t)$

definition *less-list-def*: (s::$'a$ *list*) < $t \longleftrightarrow s \leq t \wedge s \neq t$

instance

proof
  fix $x\ y$ ::$'\alpha$ *list*
  show $(x < y) = (x \leq y \wedge \neg\ y \leq x)$
    by(*auto simp*: *le-list-def less-list-def*)
 next
  fix x ::$'\alpha$ *list*
  show $x \leq x$ **by**(*simp add*: *le-list-def*)
 next
  fix $x\ y\ z$ :: $'\alpha$ *list*
  assume *A*:$x \leq y$ **and** *B*:$y \leq z$ **thus** $x \leq z$
  apply(*insert A B*, *simp add*: *le-list-def*, *safe*)
  apply(*rule-tac* x=r@*ra* **in** *exI*, *simp*)
  done
 next
  fix $x\ y$ :: $'\alpha$ *list*
  assume *A*:$x \leq y$ **and** *B*:$y \leq x$ **thus** $x = y$
  by(*insert A B*, *auto simp*: *le-list-def*)
qed

end

Some facts on the prefix ordering.

lemma *nil-le*[*simp*]: [] $\leq s$
by(*induct s*, *simp-all*, *auto simp*: *le-list-def*)

lemma *nil-le2*[*simp*]: $s \leq [] = (s = [])$

by(*induct s*, *auto simp*:*le-list-def*)

lemma *nil-less*[*simp*]: $\neg\ t < []$
by(*simp add*: *less-list-def*)

lemma *nil-less2*[*simp*]: $[] < t\ @\ [a]$
by(*simp add*: *less-list-def*)

lemma *less-self*[*simp*]: $t < t@[a]$
by(*simp add*:*less-list-def le-list-def*)

lemma *le-length-mono*: $s \leq t \Longrightarrow length\ s \leq length\ t$
by(*auto simp*: *le-list-def*)

lemma *less-length-mono*: $s < t \Longrightarrow length\ s < length\ t$
by(*auto simp*: *less-list-def le-list-def*)

lemma *list-nonMt-append*:
$s \neq [] \Longrightarrow \exists\ a\ t.\ s = t\ @\ [a]$
by(*erule rev-mp*,*induct s*,*simp-all*,*case-tac s* = [],*auto*)

lemma *append-eq-first-pref-spec*[*rule-format*]:
$s\ @\ t = r\ @\ [x] \wedge t \neq [] \longrightarrow s \leq r$
apply(*rule-tac x*=*s* **in** *spec*)
apply(*induct r*,*auto*)
apply(*erule rev-mp*)
apply(*rule-tac list*=*xa* **in** *list.induct*, *simp-all*)
apply(*simp add*: *le-list-def*)
apply(*drule list-nonMt-append*, *auto*)
done

For the process invariant, it is a key element to reduce the notion of traces to traces that may only contain one tick event at the very end. This is captured by the definition of the predicate `front_tickFree` and its stronger version `tickFree`. Here is the theory of this concept.

definition *tickFree* :: $'\alpha$ *trace* $\Rightarrow$ *bool*
where *tickFree s* = (*tick* $\notin$ *set s*)

definition *front-tickFree* :: $'\alpha$ *trace* $\Rightarrow$ *bool*
where *front-tickFree s* = (*s* =[] $\vee$ *tickFree*(*tl*(*rev s*)))

lemma *tickFree-Nil* [*simp*]: *tickFree* []
by(*simp add*: *tickFree-def*)

lemma *tickFree-Cons* [*simp*]: *tickFree* (*a* # *t*) = (*a* $\neq$ *tick* $\wedge$ *tickFree t*)
by(*auto simp add*: *tickFree-def*)

lemma *tickFree-tl* : [|s ~= [] ; *tickFree* s|] ==> *tickFree*(*tl* s)
by(*case-tac* s, *simp-all*)

lemma *tickFree-append*[*simp*]: *tickFree*(s@t) = (*tickFree* s ∧ *tickFree* t)
by(*simp add*: *tickFree-def member-def*)

lemma *non-tickFree-tick* [*simp*]: ¬ *tickFree* [*tick*]
by(*simp add*: *tickFree-def*)

lemma *non-tickFree-implies-nonMt*: ¬ *tickFree* s ⟹ $s \neq []$
by(*simp add*:*tickFree-def*,*erule rev-mp*, *induct* s, *simp-all*)

lemma *tickFree-rev* : *tickFree*(*rev* t) = (*tickFree* t)
by(*simp add*: *tickFree-def member-def*)

lemma *front-tickFree-Nil*[*simp*]: *front-tickFree* []
by(*simp add*: *front-tickFree-def*)

lemma *front-tickFree-single*[*simp*]:*front-tickFree* [a]
by(*simp add*: *front-tickFree-def*)

lemma *tickFree-implies-front-tickFree*:
tickFree s ⟹ *front-tickFree* s
apply(*simp add*: *tickFree-def front-tickFree-def member-def*,*safe*)
apply(*erule contrapos-np*, *simp*,(*erule rev-mp*)+)
apply(*rule-tac* xs=s **in** *List.rev-induct*,*simp-all*)
done

lemma *front-tickFree-charn*:
front-tickFree s = (s = [] ∨ (∃ a t. s = t @ [a] ∧ *tickFree* t))
apply(*simp add*: *front-tickFree-def*)
apply(*cases* s=[], *simp-all*)
apply(*drule list-nonMt-append*, *auto simp*: *tickFree-rev*)
done

lemma *front-tickFree-implies-tickFree*:
front-tickFree (t @ [a]) ⟹ *tickFree* t
by(*simp add*: *tickFree-def front-tickFree-def member-def*)

lemma *tickFree-implies-front-tickFree-single*:
tickFree t ⟹ *front-tickFree* (t @ [a])
by(*simp add*:*front-tickFree-charn*)

lemma *nonTickFree-n-frontTickFree*:
⟦¬ *tickFree* s; *front-tickFree* s ⟧ ⟹ ∃ t. s = t @ [*tick*]
apply(*frule non-tickFree-implies-nonMt*)

apply(*drule front-tickFree-charn*[*THEN iffD1*], *auto*)
done

lemma *front-tickFree-dw-closed* :
front-tickFree (*s* @ *t*) ⟹ *front-tickFree s*
apply(*erule rev-mp*, *rule-tac x*= *s* **in** *spec*)
apply(*rule-tac xs*=*t* **in** *List.rev-induct*, *simp*, *safe*)
apply(*simp only*: *append-assoc*[*symmetric*])
apply(*erule-tac x*=*xa* @ *xs* **in** *all-dupE*)
apply(*drule front-tickFree-implies-tickFree*)
apply(*erule-tac x*=*xa* **in** *allE*, *auto*)
apply(*auto dest*!:*tickFree-implies-front-tickFree*)
done

lemma *front-tickFree-append*:
⟦ *tickFree s*; *front-tickFree t*⟧ ⟹ *front-tickFree* (*s* @ *t*)
apply(*drule front-tickFree-charn*[*THEN iffD1*], *auto*)
apply(*erule tickFree-implies-front-tickFree*)
apply(*subst append-assoc*[*symmetric*])
apply(*rule tickFree-implies-front-tickFree-single*)
apply(*auto intro*: *tickFree-append*)
done

1.2 Basic Types, Traces, Failures and Divergences

type-synonym 'α *refusal* = ('α *event*) *set*
type-synonym 'α *failure* = 'α *trace* × 'α *refusal*
type-synonym 'α *divergence* = 'α *trace set*
type-synonym 'α *process-pre* = 'α *failure set* × 'α *divergence*

definition *FAILURES* :: 'α *process-pre* ⇒ ('α *failure set*)
where *FAILURES P* = *fst P*

definition *TRACES* :: 'α *process-pre* ⇒ ('α *trace set*)
where *TRACES P* = {*tr*. ∃ *a*. *a* ∈ *FAILURES P* ∧ *tr* = *fst a*}

definition *DIVERGENCES* :: 'α *process-pre* ⇒ 'α *divergence*
where *DIVERGENCES P* = *snd P*

definition *REFUSALS* :: 'α *process-pre* ⇒ ('α *refusal set*)
where *REFUSALS P* = {*ref*. ∃ *F*. *F* ∈ *FAILURES P* ∧ *F* = ([],*ref*)}

1.3 The Process Type Invariant

definition *is-process* :: 'α *process-pre* ⇒ *bool* **where**
is-process P =
(([],{}) ∈ *FAILURES P* ∧
(∀ *s X*. (*s*,*X*) ∈ *FAILURES P* ⟶ *front-tickFree s*) ∧
(∀ *s t* . (*s*@*t*,{}) ∈ *FAILURES P* ⟶ (*s*,{}) ∈ *FAILURES P*) ∧
(∀ *s X Y*. (*s*,*Y*) ∈ *FAILURES P* & *X* <= *Y* ⟶ (*s*,*X*) ∈ *FAILURES P*) ∧

$(\forall\ s\ X\ Y.\ (s,X) \in FAILURES\ P \wedge$
$(\forall\ c.\quad c \in Y \longrightarrow ((s@[c],\{\}) \notin FAILURES\ P)) \longrightarrow$
$(s,X \cup Y) \in FAILURES\ P) \wedge$
$(\forall\ s\ X.\ (s@[tick],\{\}) : FAILURES\ P \longrightarrow (s,X-\{tick\}) \in FAILURES\ P) \wedge$
$(\forall\ s\ t.\ s \in DIVERGENCES\ P \wedge tickFree\ s \wedge front\text{-}tickFree\ t$
$\longrightarrow s@t \in DIVERGENCES\ P) \wedge$
$(\forall\ s\ X.\ s \in DIVERGENCES\ P \longrightarrow (s,X) \in FAILURES\ P) \wedge$
$(\forall\ s.\ s\ @\ [tick] : DIVERGENCES\ P \longrightarrow s \in DIVERGENCES\ P))$

lemma *is-process-spec*:
is-process $P =$
$(([],\{\}) \in FAILURES\ P \wedge$
$(\forall\ s\ X.\ (s,X) \in FAILURES\ P \longrightarrow front\text{-}tickFree\ s) \wedge$
$(\forall\ s\ t\ .\ (s\ @\ t,\{\}) \notin FAILURES\ P \vee (s,\{\}) \in FAILURES\ P) \wedge$
$(\forall\ s\ X\ Y.\ (s,Y) \notin FAILURES\ P \vee \neg(X \subseteq Y) \mid (s,X) \in FAILURES\ P) \wedge$
$(\forall\ s\ X\ Y.(s,X) \in FAILURES\ P \wedge$
$(\forall\ c.\ c \in Y \longrightarrow ((s@[c],\{\}) \notin FAILURES\ P)) \longrightarrow (s,X \cup Y) \in FAILURES\ P) \wedge$
$(\forall\ s\ X.\ (s@[tick],\{\}) \in FAILURES\ P \longrightarrow (s,X - \{tick\}) \in FAILURES\ P) \wedge$
$(\forall\ s\ t.\ s \notin DIVERGENCES\ P \vee \neg tickFree\ s \vee \neg front\text{-}tickFree\ t$
$\vee\ s\ @\ t \in DIVERGENCES\ P) \wedge$
$(\forall\ s\ X.\ s \notin DIVERGENCES\ P \vee (s,X) \in FAILURES\ P) \wedge$
$(\forall\ s.\ s\ @\ [tick] \notin DIVERGENCES\ P \vee s \in DIVERGENCES\ P))$
by(*simp only*: *is-process-def HOL.nnf-simps*(*1*) *HOL.nnf-simps*(*3*) [*symmetric*]
HOL.imp-conjL[*symmetric*])

lemma *Process-eqI* :
assumes *A*: *FAILURES P* = *FAILURES Q*
assumes *B*: *DIVERGENCES P* = *DIVERGENCES Q*
shows (*P*::'α *process-pre*) = *Q*
apply(*insert A B*, *unfold FAILURES-def DIVERGENCES-def*)
apply(*rule-tac t*=*P* **in** *surjective-pairing*[*symmetric*,*THEN subst*])
apply(*rule-tac t*=*Q* **in** *surjective-pairing*[*symmetric*,*THEN subst*])
apply(*simp*)
done

lemma *process-eq-spec*:
((*P*::'*a process-pre*) = *Q*) =
(*FAILURES P* = *FAILURES Q* $\wedge$ *DIVERGENCES P* = *DIVERGENCES Q*)
apply(*auto simp*: *FAILURES-def DIVERGENCES-def*)
apply(*rule-tac t*=*P* **in** *surjective-pairing*[*symmetric*,*THEN subst*])
apply(*rule-tac t*=*Q* **in** *surjective-pairing*[*symmetric*,*THEN subst*])
apply(*simp*)
done

lemma *process-surj-pair*:
(*FAILURES P*,*DIVERGENCES P*) = *P*
by(*auto simp*:*FAILURES-def DIVERGENCES-def*)

lemma *Fa-eq-imp-Tr-eq*:
FAILURES P = *FAILURES Q* ⟹ *TRACES P* = *TRACES Q*
by(*auto simp*:*FAILURES-def DIVERGENCES-def TRACES-def*)

lemma *is-process1*:
is-process P ⟹ ([],{})∈ *FAILURES P*
by(*auto simp*: *is-process-def*)

lemma *is-process2*:
is-process P ⟹ ∀ *s X*. (*s*,*X*) ∈ *FAILURES P* ⟶ *front-tickFree s*
by(*simp only*: *is-process-spec*, *metis*)

lemma *is-process3*:
is-process P ⟹ ∀ *s t*. (*s* @ *t*,{}) ∈ *FAILURES P* ⟶ (*s*, {}) ∈ *FAILURES P*
by(*simp only*: *is-process-spec*, *metis*)

lemma *is-process3-S-pref*:
⟦*is-process P*; (*t*, {}) ∈ *FAILURES P*; *s* ≤ *t*⟧ ⟹ (*s*, {}) ∈ *FAILURES P*
by(*auto simp*: *le-list-def intro*: *is-process3* [*rule-format*])

lemma *is-process4*:
is-process P ⟹ ∀ *s X Y*. (*s*, *Y*) ∉ *FAILURES P* ∨ ¬ *X* ⊆ *Y* ∨ (*s*, *X*) ∈ *FAILURES P*
by(*simp only*: *is-process-spec*, *simp*)

lemma *is-process4-S*:
⟦*is-process P*; (*s*, *Y*) ∈ *FAILURES P*; *X* ⊆ *Y*⟧ ⟹ (*s*, *X*) ∈ *FAILURES P*
by(*drule is-process4*, *auto*)

lemma *is-process4-S1*:
⟦*is-process P*; *x* ∈ *FAILURES P*; *X* ⊆ *snd x*⟧ ⟹ (*fst x*, *X*) ∈ *FAILURES P*
by(*drule is-process4-S*, *auto*)

lemma *is-process5*:
is-process P ⟹
  ∀ *sa X Y*.
    (*sa*, *X*) ∈ *FAILURES P* ∧ (∀ *c*. *c* ∈ *Y* ⟶ (*sa* @ [*c*], {}) ∉ *FAILURES P*) ⟶
    (*sa*, *X* ∪ *Y*) ∈ *FAILURES P*
     using [[*show-sorts*]]
by(*drule is-process-spec*[*THEN iffD1*],*metis*)

lemma *is-process5-S*:
⟦*is-process P*; (*sa*, *X*) ∈ *FAILURES P*;
 ∀ *c*. *c* ∈ *Y* ⟶ (*sa* @ [*c*], {}) ∉ *FAILURES P*⟧

$\Longrightarrow (sa,\ X \cup Y) \in FAILURES\ P$
by(*drule is-process5*, *metis*)

lemma *is-process5-S1*:
$[\![is\text{-}process\ P;\ (sa,\ X) \in FAILURES\ P;\ (sa,\ X \cup Y) \notin FAILURES\ P]\!]$
$\Longrightarrow \exists\, c.\ c \in Y \wedge (sa\ @\ [c],\ \{\}) \in FAILURES\ P$
by(*erule contrapos-np*, *drule is-process5-S*, *simp-all*)

lemma *is-process6*:
is-process P $\Longrightarrow$
$\forall\ s\ X.\ (s@[tick],\{\}) \in FAILURES\ P \longrightarrow (s,X-\{tick\}) \in FAILURES\ P$
by(*drule is-process-spec*[*THEN iffD1*], *metis*)

lemma *is-process6-S*:
$[\![is\text{-}process\ P\ ;(s@[tick],\{\}) \in FAILURES\ P]\!] \Longrightarrow$
$(s,X-\{tick\}) \in FAILURES\ P$
by(*drule is-process6*, *metis*)

lemma *is-process7*:
is-process P $\Longrightarrow$
$\forall\ s\ t.\ s \notin DIVERGENCES\ P \vee$
$\neg\ tickFree\ s \vee$
$\neg\ front\text{-}tickFree\ t \vee$
$s\ @\ t \in DIVERGENCES\ P$
by(*drule is-process-spec*[*THEN iffD1*], *metis*)

lemma *is-process7-S*:
$[\![\ is\text{-}process\ P;s : DIVERGENCES\ P;tickFree\ s;front\text{-}tickFree\ t]\!]$
$\Longrightarrow\ s\ @\ t \in DIVERGENCES\ P$
by(*drule is-process7*, *metis*)

lemma *is-process8*:
is-process P $\Longrightarrow \forall\ \ s\ X.\ s \notin DIVERGENCES\ P \vee\ \ (s,X) \in FAILURES\ P$
by(*drule is-process-spec*[*THEN iffD1*], *metis*)

lemma *is-process8-S*:
$[\![\ is\text{-}process\ P;\ s \in DIVERGENCES\ P\ ]\!] \Longrightarrow (s,X)\ \in FAILURES\ P$
by(*drule is-process8*, *metis*)

lemma *is-process9*:
is-process P $\Longrightarrow \forall\ \ s.\ s@[tick] \notin DIVERGENCES\ P \vee\ \ s \in DIVERGENCES\ P$
by(*drule is-process-spec*[*THEN iffD1*], *metis*)

lemma *is-process9-S*:
$[\![\ is\text{-}process\ P;s@[tick] \in DIVERGENCES\ P\ ]\!] \Longrightarrow s \in DIVERGENCES\ P$
by(*drule is-process9*, *metis*)

lemma *Failures-implies-Traces*:
$[\![is\text{-}process\ P;\ (s,\ X) \in FAILURES\ P]\!] \Longrightarrow s \in TRACES\ P$

by(*simp add*: *TRACES-def*, *metis*)

lemma *is-process5-sing*:
⟦ *is-process P* ; (*s*,{*x*}) ∉ *FAILURES P*;(*s*,{}) ∈ *FAILURES P*⟧ ⟹
(*s* @ [*x*],{}) ∈ *FAILURES P*
by(*drule-tac X*={} **in** *is-process5-S1*, *auto*)

lemma *is-process5-singT*:
⟦ *is-process P* ; (*s*,{*x*}) ∉ *FAILURES P*;(*s*,{}) ∈ *FAILURES P*⟧
⟹ *s* @ [*x*] ∈ *TRACES P*
apply(*drule is-process5-sing*, *auto*)
by(*simp add*: *TRACES-def*, *auto*)

lemma *front-trace-is-tickfree*:
⟦ *is-process P*; (*t* @ [*tick*],*X*) ∈ *FAILURES P*⟧ ⟹ *tickFree t*
apply(*tactic subgoals-tac* @{*context*} [*front-tickFree*(*t* @ [*tick*])] *1*)
apply(*erule front-tickFree-implies-tickFree*)
apply(*drule is-process2*, *metis*)
done

lemma *trace-with-Tick-implies-tickFree-front* :
⟦ *is-process P*; *t* @ [*tick*] ∈ *TRACES P*⟧ ⟹ *tickFree t*
by(*auto simp*: *TRACES-def intro*: *front-trace-is-tickfree*)

1.4 The Abstraction to the process-Type

typedef (*Process*)
'α process = {*p* :: *'α process-pre* . *is-process p*}
proof −
have ({(*s*, *X*). *s* = []},{}) ∈ {*p*::*'α process-pre*. *is-process p*}
by(*simp add*: *is-process-def front-tickFree-def*
FAILURES-def TRACES-def DIVERGENCES-def)
thus *?thesis* **by** *auto*
qed

definition *F* :: *'α process* ⇒ (*'α failure set*)
where *F P* = *FAILURES* (*Rep-Process P*)

definition *T* :: *'α process* ⇒ (*'α trace set*)
where *T P* = *TRACES* (*Rep-Process P*)

definition *D* :: *'α process* ⇒ *'α divergence*
where *D P* = *DIVERGENCES* (*Rep-Process P*)

definition $R :: {'\alpha}\ process \Rightarrow ({'\alpha}\ refusal\ set)$
where $R\ P = REFUSALS\ (Rep\text{-}Process\ P)$

lemma *is-process-Rep* : *is-process* (*Rep-Process P*)
apply(*rule-tac P=is-process* **in** *CollectD*)
apply(*subst Process-def*[*symmetric*])
apply(*simp add*: *Rep-Process*)
done

lemma *Process-spec*: *Abs-Process*((*F P* , *D P*)) = *P*
by(*simp add*: *F-def FAILURES-def D-def*
DIVERGENCES-def Rep-Process-inverse)

theorem *Process-eq-spec*:
$(P = Q) = (F\ P = F\ Q \wedge D\ P = D\ Q)$
apply(*rule iffI*,*simp*)
apply(*rule-tac t=P* **in** *Process-spec*[*THEN subst*])
apply(*rule-tac t=Q* **in** *Process-spec*[*THEN subst*])
apply *simp*
done

theorem *is-processT*:
$([],\{\}) \in F\ P \wedge$
$(\forall\ s\ X.\ (s,X) \in F\ P \longrightarrow front\text{-}tickFree\ s) \wedge$
$(\forall\ s\ t\ .(s@t,\{\}) \in F\ P \longrightarrow (s,\{\}) \in F\ P) \wedge$
$(\forall\ s\ X\ Y.(s,Y) \in F\ P \wedge (X \subseteq Y) \longrightarrow (s,X) \in F\ P) \wedge$
$(\forall\ s\ X\ Y.(s,X) \in F\ P \wedge (\forall\ c.\ c \in Y \longrightarrow ((s@[c],\{\}) \notin F\ P)) \longrightarrow (s,X \cup Y) \in F\ P) \wedge$
$(\forall\ s\ X.\ (s@[tick],\{\}) \in F\ P \longrightarrow (s,\ X - \{tick\}) \in F\ P) \wedge$
$(\forall\ s\ t.\ s \in D\ P \wedge tickFree\ s \wedge front\text{-}tickFree\ t \longrightarrow s\ @\ t \in D\ P) \wedge$
$(\forall\ s\ X.\ s \in D\ P \longrightarrow (s,X) \in F\ P) \wedge$
$(\forall\ s.\ s@[tick] \in D\ P \longrightarrow s \in D\ P)$
apply(*simp only*: *F-def D-def T-def*)
apply(*rule is-process-def*[*THEN iffD1*])
apply(*rule is-process-Rep*)
done

theorem *process-charn*:
$([],\ \{\}) \in F\ P \wedge$
$(\forall s\ X.\ (s,\ X) \in F\ P \longrightarrow front\text{-}tickFree\ s) \wedge$
$(\forall s\ t.\ (s\ @\ t,\ \{\}) \notin F\ P \vee (s,\ \{\}) \in F\ P) \wedge$
$(\forall s\ X\ Y.\ (s,\ Y) \notin F\ P \vee \neg\ X \subseteq Y \vee (s,\ X) \in F\ P) \wedge$
$(\forall s\ X\ Y.\ (s,\ X) \in F\ P \wedge (\forall c.\ c \in Y \longrightarrow (s\ @\ [c],\ \{\}) \notin F\ P) \longrightarrow$
$(s,\ X \cup Y) \in F\ P) \wedge$
$(\forall s\ X.\ (s\ @\ [tick],\ \{\}) \in F\ P \longrightarrow (s,\ X - \{tick\}) \in F\ P) \wedge$
$(\forall s\ t.\ s \notin D\ P \vee \neg\ tickFree\ s \vee \neg\ front\text{-}tickFree\ t \vee s\ @\ t \in D\ P) \wedge$

$(\forall\, s\ X.\ s \notin D\ P \vee (s,\ X) \in F\ P) \wedge (\forall\, s.\ s\ @\ [tick] \notin D\ P \vee s \in D\ P)$
proof −
{
have *A*: $(\forall\, s\ t.\ (s\ @\ t,\ \{\}) \notin F\ P \vee\ (s,\ \{\}) \in F\ P) =$
$(\forall\, s\ t.\ (s\ @\ t,\ \{\}) \in F\ P \longrightarrow (s,\ \{\}) \in F\ P)$
by *metis*

have *B* : $(\forall\, s\ X\ Y.\ (s,\ Y) \notin F\ P \vee \neg\ X \subseteq Y \vee (s,\ X) \in F\ P)\ =$
$(\forall\, s\ X\ Y.\ (s,\ Y) \in F\ P \wedge X \subseteq Y \longrightarrow (s,\ X) \in F\ P)$
by *metis*

have *C* : $(\forall\, s\ t.\ s \notin D\ P \vee \neg\ tickFree\ s \vee$
$\neg\ front\text{-}tickFree\ t \vee s\ @\ t \in D\ P)\ =$
$(\forall\, s\ t.\ s \in D\ P \wedge tickFree\ s \wedge front\text{-}tickFree\ t \longrightarrow s\ @\ t \in D\ P)$
by *metis*

have *D*:$(\forall\, s\ X.\ s \notin D\ P \vee (s,\ X) \in F\ P) = (\forall\, s\ X.\ s \in D\ P \longrightarrow (s,\ X) \in F\ P)$
by *metis*

have *E*:$(\forall\, s.\ s\ @\ [tick] \notin D\ P \vee s \in D\ P) =$
$(\forall\, s.\ s\ @\ [tick] \in D\ P \longrightarrow s \in D\ P)$
by *metis*

note *A B C D E*
}
note $a = this$
then
show *?thesis*
apply(*simp only*: *a*)
apply(*rule is-processT*)
done
qed

split of `is_processT`:

lemma *is-processT1*: $([],\{\}) \in F\ P$
by(*simp add*:*process-charn*)

lemma *is-processT2*:
$\forall\, s\ X.\ (s,\ X) \in F\ P \longrightarrow front\text{-}tickFree\ s$
by(*simp add*:*process-charn*)

lemma *is-processT2-TR* : $\forall\, s.\ s \in T\ P \longrightarrow front\text{-}tickFree\ s$
apply(*simp add*: *F-def* [*symmetric*] *T-def TRACES-def*, *safe*)
apply (*drule is-processT2*[*rule-format*], *assumption*)
done

lemma *is-proT2*:
$[\![(s,\ X) \in F\ P;\ s \neq []]\!] \Longrightarrow tick \notin set\ (tl\ (rev\ s))$

apply(*tactic* ⟪ *subgoals-tac* @{*context*} [*front-tickFree s*] *1* ⟫)
apply(*simp add*: *tickFree-def front-tickFree-def*)
by(*simp add*: *is-processT2*)

lemma *is-processT3* :
$\forall\, s\ t.\ (s\ @\ t,\ \{\}) \in F\ P \longrightarrow (s,\ \{\}) \in F\ P$
by(*simp only*: *process-charn HOL.nnf-simps*(*3*), *simp*)

lemma *is-processT3-S-pref* :
$[\![(t,\ \{\}) \in F\ P;\ s \leq t]\!] \Longrightarrow (s,\ \{\}) \in F\ P$
apply(*simp only*: *le-list-def*, *safe*)
apply(*erule is-processT3*[*rule-format*])
done

lemma *is-processT4* :
$\forall\, s\ X\ Y.\ (s,\ Y) \in F\ P \wedge X \subseteq Y \longrightarrow (s,\ X) \in F\ P$
by(*insert process-charn* [*of P*], *metis*)

lemma *is-processT4-S1* :
$[\![x \in F\ P;\ X \subseteq snd\ x]\!] \Longrightarrow (fst\ x,\ X) \in F\ P$
apply(*rule-tac* $Y = snd\ x$ **in** *is-processT4*[*rule-format*])
apply *simp*
done

lemma *is-processT5*:
$\forall\, s\ X\ Y.(s,X) \in F\ P \wedge (\forall\, c.\ c{\in}Y \longrightarrow (s@[c],\{\}) \notin F\ P) \longrightarrow (s,X{\cup}Y){\in}F\ P$
by(*simp add*: *process-charn*)

lemma *is-processT5-S1*:
$[\![(s,\ X) \in F\ P;\ (s,\ X \cup Y) \notin F\ P]\!] \Longrightarrow \exists\, c.\ c \in Y \wedge (s\ @\ [c],\ \{\}) \in F\ P$
by(*erule contrapos-np*, *simp add*: *is-processT5*[*rule-format*])

lemma *is-processT5-S2*:
$[\![(s,\ X) \in F\ P;\ (s\ @\ [c],\ \{\}) \notin F\ P]\!] \Longrightarrow (s,\ X \cup \{c\}) \in F\ P$
by(*rule is-processT5*[*rule-format*,*OF conjI*], *metis*, *safe*)

lemma *is-processT5-S2a*:
$[\![(s,\ X) \in F\ P;\ (s,\ X \cup \{c\}) \notin F\ P]\!] \Longrightarrow (s\ @\ [c],\ \{\}) \in F\ P$
apply(*erule contrapos-np*)
apply(*rule is-processT5-S2*)
apply(*simp-all*)
done

lemma *is-processT5-S3*:
assumes *A*: $(s, \{\}) \in F\ P$
and *B*: $(s\ @\ [c], \{\}) \notin F\ P$
shows $(s, \{c\}) \in F\ P$
proof −
have *C* : $\{c\} = (\{\}\ Un\ \{c\})$ **by** *simp*
show *?thesis*
by(*subst C*, *rule is-processT5-S2*, *simp-all add*: *A B*)
qed

lemma *is-processT5-S4*:
$[\![(s, \{\}) \in F\ P; (s, \{c\}) \notin F\ P]\!] \Longrightarrow (s\ @\ [c], \{\}) \in F\ P$
by(*erule contrapos-np*, *simp add*: *is-processT5-S3*)

lemma *is-processT5-S5*:
$[\![(s, X) \in F\ P; \forall\, c.\ c \in Y \longrightarrow (s, X \cup \{c\}) \notin F\ P]\!]$
$\Longrightarrow \forall\, c.\ c \in Y \longrightarrow (s\ @\ [c], \{\}) \in F\ P$
by(*erule-tac Q* = $\forall\ x.$ *?Z x* **in** *contrapos-pp*, *metis is-processT5-S2*)

lemma *is-processT5-S6*:
$([], \{c\}) \notin F\ P \Longrightarrow ([c], \{\}) \in F\ P$
apply(*rule-tac t*=[*c*] **and** *s*=[]@[*c*] **in** *subst*, *simp*)
apply(*rule is-processT5-S4*, *simp-all add*: *is-processT1*)
done

lemma *is-processT6*:
$\forall\, s\ X.\ (s\ @\ [tick], \{\}) \in F\ P \longrightarrow (s, X - \{tick\}) \in F\ P$
by(*simp add*: *process-charn*)

lemma *is-processT7*:
$\forall\, s\ t.\ s \in D\ P \wedge tickFree\ s \wedge front\text{-}tickFree\ t \longrightarrow s\ @\ t \in D\ P$
by(*insert process-charn*[*of P*], *metis*)

lemmas *is-processT7-S* =
is-processT7[*rule-format*,*OF conjI*[*THEN conjI*,
THEN conj-commute[*THEN iffD1*]]]

lemma *is-processT8*:
$\forall\, s\ X.\ s \in D\ P \longrightarrow (s, X) \in F\ P$
by(*insert process-charn*[*of P*], *metis*)

lemmas *is-processT8-S* = *is-processT8*[*rule-format*]

lemma *is-processT8-Pair*: *fst s* ∈ *D P* ⟹ *s* ∈ *F P*
apply(*subst surjective-pairing*)
apply(*rule is-processT8-S*, *simp*)
done

lemma *is-processT9*:
∀ *s*. *s* @ [*tick*] ∈ *D P* ⟶ *s* ∈ *D P*
by(*insert process-charn*[*of P*], *metis*)

lemma *is-processT9-S-swap*: *s* ∉ *D P* ⟹ *s* @ [*tick*] ∉ *D P*
by(*erule contrapos-nn*,*simp add*: *is-processT9*[*rule-format*])

1.5 Some Consequences of the Process Characterization

lemma *no-Trace-implies-no-Failure*:
s ∉ *T P* ⟹ (*s*, {}) ∉ *F P*
by(*simp add*: *T-def TRACES-def F-def*)

lemmas *NT-NF* = *no-Trace-implies-no-Failure*

lemma *T-def-spec*:
T P = {*tr*. ∃ *a*. *a* ∈ *F P* ∧ *tr* = *fst a*}
by(*simp add*: *T-def TRACES-def F-def*)

lemma *F-T*:
(*s*, *X*) ∈ *F P* ⟹ *s* ∈ *T P*
by(*simp add*: *T-def-spec split-def*, *metis*)

lemma *F-T1*:
a ∈ *F P* ⟹ *fst a* ∈ *T P*
by(*rule-tac X*=*snd a* **in** *F-T*,*simp*)

lemma *T-F*:
s ∈ *T P* ⟹ (*s*, {}) ∈ *F P*
apply(*auto simp*: *T-def-spec*)
apply(*drule is-processT4-S1*, *simp-all*)
done

lemmas *is-processT4-empty* [*elim*!]= *F-T* [*THEN T-F*]

lemma *NF-NT*:
(*s*, {}) ∉ *F P* ⟹ *s* ∉ *T P*
by(*erule contrapos-nn*, *simp only*: *T-F*)

lemma *is-processT6-S1*:
$[\![\ tick \notin X;(s\ @\ [tick],\ \{\}) \in F\ P\ ]\!] \Longrightarrow (s::'a\ event\ list,\ X) \in F\ P$
by(*subst Diff-triv*[*of X* {*tick*}, *symmetric*],
simp, *erule is-processT6*[*rule-format*])

lemmas *is-processT3-ST* = *T-F* [*THEN is-processT3*[*rule-format*,*THEN F-T*]]

lemmas *is-processT3-ST-pref* = *T-F* [*THEN is-processT3-S-pref* [*THEN F-T*]]

lemmas *is-processT3-SR* = *F-T* [*THEN T-F* [*THEN is-processT3*[*rule-format*]]]

lemmas *D-T* = *is-processT8-S* [*THEN F-T*]

lemma *D-T-subset* : *D P* ⊆ *T P* **by**(*auto intro*!:*D-T*)

lemma *NF-ND* : (*s*, *X*) ∉ *F P* ⟹ *s* ∉ *D P*
by(*erule contrapos-nn*, *simp add*: *is-processT8-S*)

lemmas *NT-ND* = *D-T-subset*[*THEN Set.contra-subsetD*]

lemma *T-F-spec* : ((*t*, {}) ∈ *F P*) = (*t* ∈ *T P*)
by(*auto simp*:*T-F F-T*)

lemma *is-processT5-S7*:
⟦*t* ∈ *T P*; (*t*, *A*) ∉ *F P*⟧ ⟹ ∃*x*. *x* ∈ *A* ∧ *t* @ [*x*] ∈ *T P*
apply(*erule contrapos-np*, *simp*)
apply(*rule is-processT5*[*rule-format*, *OF conjI*,*of* - {}, *simplified*])
apply(*auto simp*: *T-F-spec*)
done

lemma *Nil-subset-T*: {[]} ⊆ *T P*
by(*auto simp*: *T-F-spec*[*symmetric*] *is-processT1*)

lemma *Nil-elem-T*: [] ∈ *T P*
by(*simp add*: *Nil-subset-T*[*THEN subsetD*])

lemmas *D-imp-front-tickFree* =
is-processT8-S[*THEN is-processT2*[*rule-format*]]

lemma *D-front-tickFree-subset* : *D P* ⊆ *Collect front-tickFree*
by(*auto simp*: *D-imp-front-tickFree*)

lemma *F-D-part*:
F P = {(*s*, *x*). *s* ∈ *D P*} ∪ {(*s*, *x*). *s* ∉ *D P* ∧ (*s*, *x*) ∈ *F P*}
by(*insert excluded-middle*[*of fst x* : *D P*],*auto intro*:*is-processT8-Pair*)

lemma *D-F* : $\{(s,\ x).\ s \in D\ P\} \subseteq F\ P$
by(*auto intro*:*is-processT8-Pair*)

lemma *append-T-imp-tickFree*:
$[\![t\ @\ s \in T\ P;\ s \neq []]\!] \Longrightarrow$ *tickFree t*
by(*frule is-processT2-TR*[*rule-format*],
simp add: *front-tickFree-def tickFree-rev*)

lemma *F-subset-imp-T-subset*:
$F\ P \subseteq F\ Q \Longrightarrow T\ P \subseteq T\ Q$
by(*auto simp*: *subsetD T-F-spec*[*symmetric*])

lemmas *append-single-T-imp-tickFree* =
append-T-imp-tickFree[*of* - [*a*], *simplified*]

lemma *is-processT6-S2*:
$[\![tick \notin X;\ [tick] \in T\ P]\!] \Longrightarrow ([],\ X) \in F\ P$
by(*erule is-processT6-S1*, *simp add*: *T-F-spec*)

lemma *is-processT9-tick*:
$[\![[tick] \in D\ P;\ front\text{-}tickFree\ s]\!] \Longrightarrow s \in D\ P$
apply(*rule append.simps*(*1*) [*THEN subst*, *of* - *s*])
apply(*rule is-processT7-S*, *simp-all*)
apply(*rule is-processT9* [*rule-format*], *simp*)
done

lemma *T-nonTickFree-imp-decomp*:
$[\![t \in T\ P;\ \neg\ tickFree\ t]\!] \Longrightarrow \exists s.\ t = s\ @\ [tick]$
by(*auto elim*: *is-processT2-TR*[*rule-format*] *nonTickFree-n-frontTickFree*)

1.6 Process Approximation is a Partial Ordering, a Cpo, and a Pcpo

The Failure/Divergence Model of CSP Semantics provides two orderings: The *approximation ordering* (also called *process ordering*) will be used for giving semantics to recursion (fixpoints) over processes, the *refinement ordering* captures our intuition that a more concrete process is more deterministic and more defined than an abstract one.

We start with the key-concepts of the approximation ordering, namely the predicates *min_elems* and *Ra* (abbreviating *refusals after*). The former provides just a set of minimal elements from a given set of elements of type-class *ord* ...

definition *min-elems* :: ('*s*::*ord*) *set* ⇒ '*s set*
where *min-elems X* = $\{s \in X.\ \forall t.\ t \in X \longrightarrow \neg\ (t < s)\}$

lemma *Nil-min-elems* : $[] \in A \Longrightarrow [] \in$ *min-elems A*
by(*simp add*: *min-elems-def*)

lemma *min-elems-le-self*[*simp*] : (*min-elems A*) ⊆ *A*
by(*auto simp*: *min-elems-def*)

lemmas *elem-min-elems* = *Set.set-mp*[*OF min-elems-le-self*]

lemma *min-elems-Collect-ftF-is-Nil* :
min-elems (*Collect front-tickFree*) = {[]}
apply(*auto simp*: *min-elems-def le-list-def*)
apply(*drule front-tickFree-charn*[*THEN iffD1*])
apply(*auto dest*!: *tickFree-implies-front-tickFree*)
done

lemma *min-elems5* :
assumes *A*: (*x*::'*a list*) ∈ *A*
shows ∃ *s*≤*x*. *s* ∈ *min-elems A*
proof −
have *core* : !! (*x*::'*a list*) (*A*::'*a list set*) (*n*::*nat*).
x ∈ *A* ∧ *length x* ≤ *n* ⟶ (∃ *s*≤*x*. *s* ∈ *min-elems A*)
apply(*rule-tac x*=*x* **in** *spec*)
apply(*rule-tac n*=*n* **in** *nat-induct*)
apply(*auto simp*: *Nil-min-elems*)
apply(*case-tac* ∃ *y*. *y* ∈ *A* ∧ *y* < *x*,*auto*)
apply(*erule-tac x*=*y* **in** *allE*, *simp*)
apply(*erule impE*,*drule less-length-mono*, *arith*)
apply(*safe*, *rule-tac x*=*s* **in** *exI*,*simp*)
apply(*rule-tac x*=*x* **in** *exI*, *simp add*:*min-elems-def*)
done
show *?thesis*
apply(*rule-tac n*=*length x* **in** *core*[*rule-format*],*simp add*:*A*)
done
qed

lemma *min-elems4*:
A ≠ {} ⟹ ∃ *s*. (*s* :: '*a trace*) ∈ *min-elems A*
by(*auto dest*: *min-elems5*)

lemma *min-elems-charn*:
t ∈ *A* ⟹ ∃ *t*' *r*. *t* = (*t*' @ *r*) ∧ *t*' ∈ *min-elems A*
by(*drule min-elems5*[*simplified le-list-def*], *auto*)

lemmas *min-elems-ex* = *min-elems-charn*

. . . while the second returns the set of possible refusal sets after a given trace s and a given process P:

definition *Ra* :: ['α *process*, 'α *trace*] ⇒ ('α *refusal set*)
where *Ra P s* = {*X*. (*s*, *X*) ∈ *F P*}

In the following, we link the process theory to the underlying fixpoint/domain theory of

HOLCF by identifying the approximation ordering with HOLCF's pcpo's.

instantiation
process :: (*type*) *below*
begin

declares approximation ordering _ $\sqsubseteq$ _ also written _ << _.

definition *le-approx-def* : $P \sqsubseteq Q \equiv D\ Q \subseteq D\ P \wedge$
$(\forall s.\ s \notin D\ P \longrightarrow Ra\ P\ s = Ra\ Q\ s) \wedge$
$min\text{-}elems\ (D\ P) \subseteq T\ Q$

The approximation ordering captures the fact that more concrete processes should be more defined by ordering the divergence sets appropriately. For defined positions in a process, the failure sets must coincide pointwise; moreover, the minimal elements (wrt. prefix ordering on traces, i.e. lists) must be contained in the trace set of the more concrete process.

instance ..

end

lemma *le-approx1*:
$P \sqsubseteq Q \Longrightarrow D\ Q \subseteq D\ P$
by(*simp add*: *le-approx-def*)

lemma *le-approx2*:
$[\![\ P \sqsubseteq Q;\ s \notin D\ P\ ]\!] \Longrightarrow (s,X) \in F\ Q = ((s,X) \in F\ P)$
by(*auto simp*: *Ra-def le-approx-def*)

lemma *le-approx3*:
$P \sqsubseteq Q \Longrightarrow min\text{-}elems(D\ P) \subseteq T\ Q$
by(*simp add*: *le-approx-def*)

lemma *le-approx2T*:
$[\![\ P \sqsubseteq Q;\ s \notin D\ P\ ]\!] \Longrightarrow\ \ s \in T\ Q = (s \in T\ P)$
by(*auto simp*: *le-approx2 T-F-spec*[*symmetric*])

lemma *le-approx-lemma-F* :
$P \sqsubseteq Q \Longrightarrow F\ Q \subseteq F\ P$
apply(*subst F-D-part*[*of Q*], *subst F-D-part*[*of P*])
apply(*auto simp*:*le-approx-def Ra-def min-elems-def*)
done

lemmas *order-lemma* = *le-approx-lemma-F*

lemma *le-approx-lemma-T*:
$P \sqsubseteq Q \Longrightarrow T\ Q \subseteq T\ P$

by(*auto dest*!:*le-approx-lemma-F simp*: *T-F-spec*[*symmetric*])

lemma *proc-ord2a* :
$[\![P \sqsubseteq Q;\ s \notin D\ P]\!] \Longrightarrow ((s,\ X) \in F\ P) = ((s,\ X) \in F\ Q)$
by(*auto simp*: *le-approx-def Ra-def*)

instance
process :: (*type*) *po*
proof
fix $P::'\alpha$ *process*
show $P \sqsubseteq P$ **by**(*auto simp*: *le-approx-def min-elems-def elim*: *Process.D-T*)
next
fix $P\ Q ::'\alpha$ *process*
assume A:$P \sqsubseteq Q$ **and** B:$Q \sqsubseteq P$ **thus** $P = Q$
apply(*insert A*[*THEN le-approx1*] *B*[*THEN le-approx1*])
apply(*insert A*[*THEN le-approx-lemma-F*] *B*[*THEN le-approx-lemma-F*])
by(*auto simp*: *Process-eq-spec*)
next
fix $P\ Q\ R ::'\alpha$ *process*
assume A: $P \sqsubseteq Q$ **and** B: $Q \sqsubseteq R$ **thus** $P \sqsubseteq R$
proof −
have C : $D\ R \subseteq D\ P$
by(*insert A*[*THEN le-approx1*] *B*[*THEN le-approx1*], *auto*)
have D : $\forall\ s.\ s \notin D\ P \longrightarrow \{X.\ (s,\ X) \in F\ P\} = \{X.\ (s,\ X) \in\ F\ R\}$
apply(*rule allI*, *rule impI*, *rule set-eqI*, *simp*)
apply(*frule A*[*THEN le-approx1*, *THEN Set.contra-subsetD*])
apply(*frule B*[*THEN le-approx1*, *THEN Set.contra-subsetD*])
apply(*drule A*[*THEN le-approx2*], *drule B*[*THEN le-approx2*])
apply *auto*
done
have E : *min-elems* $(D\ P) \subseteq T\ R$
apply(*insert B*[*THEN le-approx3*] *A*[*THEN le-approx3*])
apply(*insert B*[*THEN le-approx-lemma-T*] *A*[*THEN le-approx1*])
apply(*rule subsetI*, *simp add*: *min-elems-def*, *auto*)
apply(*case-tac* $x \in D\ Q$)
apply(*drule-tac* $B = T\ R$ **and** $t{=}x$
in *subset-iff*[*THEN iffD1*,*rule-format*], *auto*)
apply(*subst B* [*THEN le-approx2T*],*simp*)
apply(*drule-tac* $B = T\ Q$ **and** $t{=}x$
in *subset-iff*[*THEN iffD1*,*rule-format*],*auto*)
done
show *?thesis*
by(*insert C D E*, *simp add*: *le-approx-def Ra-def*)
qed
qed

At this point, we inherit quite a number of facts from the underlying HOLCF theory,

which comprises a library of facts such as `chain`, `directed`(sets), upper bounds and least upper bounds, etc.

find-theorems *name*:*Porder is-lub*

Some facts from the theory of complete partial orders:

- `Porder.chainE` : $chain\ ?Y \Longrightarrow ?Y\ ?i \sqsubseteq ?Y\ (Suc\ ?i)$
- `Porder.chain_mono` : $[\![chain\ ?Y;\ ?i \leq ?j]\!] \Longrightarrow ?Y\ ?i \sqsubseteq ?Y\ ?j$
- `Directed.directed_chain` : $chain\ ?S \Longrightarrow directed\ (range\ ?S)$
- `Directed.directed_def` :
  $directed\ ?S = ((\exists x.\ x \in ?S) \wedge (\forall x{\in}?S.\ \forall y{\in}?S.\ \exists z{\in}?S.\ x \sqsubseteq z \wedge y \sqsubseteq z))$
- `Directed.directedD1` : $directed\ ?S \Longrightarrow \exists z.\ z \in ?S$
- `Directed.directedD2` :
  $[\![directed\ ?S;\ ?x \in ?S;\ ?y \in ?S]\!] \Longrightarrow \exists z{\in}?S.\ ?x \sqsubseteq z \wedge ?y \sqsubseteq z$
- `Directed.directedI` : $[\![\exists z.\ z \in ?S;\ \bigwedge x\ y.\ [\![x \in ?S;\ y \in ?S]\!] \Longrightarrow \exists z{\in}?S.\ x \sqsubseteq z \wedge y \sqsubseteq z]\!] \Longrightarrow directed\ ?S$
- `Porder.is_ubD` : $[\![?S <|\ ?u;\ ?x \in ?S]\!] \Longrightarrow ?x \sqsubseteq ?u$
- `Porder.ub_rangeI` :
  $(\bigwedge i.\ ?S\ i \sqsubseteq ?x) \Longrightarrow range\ ?S <|\ ?x$
- `Porder.ub_imageD` : $[\![?f\ `\ ?S <|\ ?u;\ ?x \in ?S]\!] \Longrightarrow ?f\ ?x \sqsubseteq ?u$
- `Porder.is_ub_upward` : $[\![?S <|\ ?x;\ ?x \sqsubseteq ?y]\!] \Longrightarrow ?S <|\ ?y$
- `Porder.is_lubD1` : $?S <<|\ ?x \Longrightarrow ?S <|\ ?x$
- `Porder.is_lubI` : $[\![?S <|\ ?x;\ \bigwedge u.\ ?S <|\ u \Longrightarrow ?x \sqsubseteq u]\!] \Longrightarrow ?S <<|\ ?x$
- `Porder.is_lub_maximal` : $[\![?S <|\ ?x;\ ?x \in ?S]\!] \Longrightarrow ?S <<|\ ?x$
- `Porder.is_lub_lub` : $?M <<|\ ?x \Longrightarrow ?M <<|\ lub\ ?M$
- `Porder.is_lub_range_shift`:
  $chain\ ?S \Longrightarrow range\ (\lambda i.\ ?S\ (i + ?j)) <<|\ ?x = range\ ?S <<|\ ?x$
- `Porder.is_lub_rangeD1`: $range\ ?S <<|\ ?x \Longrightarrow ?S\ ?i \sqsubseteq ?x$
- `Porder.lub_eqI`: $?M <<|\ ?l \Longrightarrow lub\ ?M = ?l$
- `Porder.is_lub_unique`:$[\![?S <<|\ ?x;\ ?S <<|\ ?y]\!] \Longrightarrow ?x = ?y$

definition *lim-proc* :: $('\alpha\ process)\ set \Rightarrow '\alpha\ process$
  where $lim\text{-}proc\ (X) = Abs\text{-}Process\ (INTER\ X\ F,\ INTER\ X\ D)$

lemma *min-elems3*:
$[\![s\ @\ [c] \in D\ P;\ s\ @\ [c] \notin min\text{-}elems\ (D\ P)]\!] \Longrightarrow s \in D\ P$
apply(*auto simp*: *min-elems-def le-list-def less-list-def*)
apply(*subgoal-tac* $t \leq s$)
apply(*subgoal-tac* $r \neq []$)
apply(*simp add*: *le-list-def*)
apply(*auto intro*!: *is-processT7-S append-eq-first-pref-spec*)
apply(*auto dest*!: *D-T*)
apply(*simp-all only*: *append-assoc*[*symmetric*],

drule append-T-imp-tickFree,
simp-all add: *tickFree-implies-front-tickFree*)+
done

lemma *min-elems1* :
$[\![s \notin D\ P;\ s\ @\ [c] \in D\ P]\!] \Longrightarrow s\ @\ [c] \in \textit{min-elems}\ (D\ P)$
by(*erule contrapos-np*, *auto elim*!: *min-elems3*)

lemma *min-elems2*:
$[\![s \notin D\ P;\ s\ @\ [c] \in D\ P;\ P \sqsubseteq S;\ Q \sqsubseteq S]\!] \Longrightarrow (s\ @\ [c], \{\}) \in F\ Q$
apply(*frule-tac P=Q* **and** *Q=S* **in** *le-approx-lemma-T*)
apply(*simp add*: *T-F-spec*)
apply(*rule-tac A=T S* **in** *subsetD*, *assumption*)
apply(*rule-tac A=min-elems*(*D P*) **in** *subsetD*)
apply(*simp-all add*: *le-approx-def min-elems1*)
done

lemma *min-elems6*:
$[\![s \notin D\ P;\ s\ @\ [c] \in D\ P;\ P \sqsubseteq S]\!] \Longrightarrow (s\ @\ [c], \{\}) \in F\ S$
by(*auto intro*!: *min-elems2*)

lemma *ND-F-dir2*:
$[\![s \notin D\ P;\ (s, \{\}) \in F\ P;\ P \sqsubseteq S;\ Q \sqsubseteq S]\!] \Longrightarrow (s, \{\}) \in F\ Q$
apply(*frule-tac P=Q* **and** *Q=S* **in** *le-approx-lemma-T*)
apply(*simp add*: *le-approx-def Ra-def T-F-spec*, *safe*)
apply((*erule-tac x=s* **in** *allE*)+,*simp*)
apply(*drule-tac x={}* **in** *eqset-imp-iff*, *auto simp*: *T-F-spec*)
done

lemma *ND-F-dir2'*:
$[\![s \notin D\ P;\ s \in T\ P;\ P \sqsubseteq S;\ Q \sqsubseteq S]\!] \Longrightarrow s \in T\ Q$
apply(*frule-tac P=Q* **and** *Q=S* **in** *le-approx-lemma-T*)
apply(*simp add*: *le-approx-def Ra-def T-F-spec*, *safe*)
apply((*erule-tac x=s* **in** *allE*)+,*simp*)
apply(*drule-tac x={}* **in** *eqset-imp-iff*, *auto simp*: *T-F-spec*)
done

lemma *chain-lemma*:$[\![\textit{chain}\ S]\!] \Longrightarrow S\ i \sqsubseteq S\ k \vee S\ k \sqsubseteq S\ i$
by(*case-tac* $i \leq k$, *auto intro*:*chain-mono chain-mono-less*)

lemma *is-process-REP-LUB*:
assumes *chain*: *chain S*
shows *is-process*(*INTER* (*range S*) *F*,*INTER* (*range S*) *D*)

proof (*auto simp*: *is-process-def*)
show ([], {}) ∈ *FAILURES* ($\bigcap$ *a*::*nat*. *F* (*S a*), $\bigcap$ *a*::*nat*. *D* (*S a*))

by(*auto simp*: *FAILURES-def is-processT*)
next
fix *s*::′*a trace* **fix** *X*::′*a event set*
assume *A* : (*s*, *X*) ∈ (*FAILURES* (⋂ *a* :: *nat*. *F* (*S a*), ⋂ *a* :: *nat*. *D* (*S a*)))
thus *front-tickFree s*
by(*auto simp*: *DIVERGENCES-def FAILURES-def*
intro!: *is-processT2*[*rule-format*])
next
fix *s t*::′*a trace*
assume (*s* @ *t*, {}) ∈ *FAILURES* (⋂ *a*::*nat*. *F* (*S a*), ⋂ *a*::*nat*. *D* (*S a*))
thus (*s*, {}) ∈ *FAILURES* (⋂ *a*::*nat*. *F* (*S a*), ⋂ *a*::*nat*. *D* (*S a*))
by(*auto simp*: *DIVERGENCES-def FAILURES-def*
intro: *is-processT3*[*rule-format*])
next
fix *s*::′*a trace* **fix** *X Y* ::′*a event set*
assume (*s*, *Y*) ∈ *FAILURES* (⋂ *a*::*nat*. *F* (*S a*), ⋂ *a*::*nat*. *D* (*S a*)) **and** *X* ⊆ *Y*
thus (*s*, *X*) ∈ *FAILURES* (⋂ *a*::*nat*. *F* (*S a*), ⋂ *a*::*nat*. *D* (*S a*))
by(*auto simp*: *DIVERGENCES-def FAILURES-def*
intro: *is-processT4*[*rule-format*])
next
fix *s*::′*a trace* **fix** *X Y* :: ′*a event set*
assume *A*:(*s*, *X*) ∈ *FAILURES* (⋂ *a*::*nat*. *F* (*S a*), ⋂ *a*::*nat*. *D* (*S a*))
assume *B*:∀ *c*. *c*∈*Y* ⟶ (*s*@[*c*],{})∉*FAILURES*(⋂ *a*::*nat*. *F*(*S a*),⋂ *a*::*nat*. *D*(*S a*))
thus (*s*, *X* ∪ *Y*) ∈ *FAILURES* (⋂ *a*::*nat*. *F* (*S a*), ⋂ *a*::*nat*. *D* (*S a*))

What does this mean: All trace prolongations c in all Y, which are blocking in the limit, will also occur in the refusal set of the limit.

using *A B directed-chain*[*OF chain*] *chain*

proof (*auto simp*: *DIVERGENCES-def FAILURES-def directed-def*,
case-tac ∀ *x*. *x* ∈ (*range S*) ⟶ (*s*, *X* ∪ *Y*) ∈ *F x*,
simp-all add:*DIVERGENCES-def FAILURES-def directed-def*,
case-tac s ∉ *D* (*S a*),*simp-all add*: *is-processT8*)
fix *a*::*nat*
assume *X*: ∀ *a*. (*s*, *X*) ∈ *F* (*S a*)
have *X-ref-at-a*: (*s*, *X*) ∈ *F* (*S a*)
using *X* **by** *auto*
assume *Y*: ∀ *c*. *c* ∈ *Y* ⟶ (∃ *a*. (*s* @ [*c*], {}) ∉ *F* (*S a*))
assume *defined*: *s* ∉ *D* (*S a*)
show (*s*::′*a trace*, *X* ∪ *Y*) ∈ *F* (*S a*)
proof(*auto simp*:*X-ref-at-a*
intro!: *is-processT5*[*rule-format*],
frule Y[*THEN spec*, *THEN mp*], *erule exE*,
erule-tac Q=(*s* @ [*c*], {}) ∈ *F* (*S a*) **in** *contrapos-pp*)
fix *c*::′*a event* **fix** *a*′ :: *nat*
assume *s-c-trace-not-trace-somewhere*: (*s* @ [*c*], {}) ∉ *F* (*S a*′)
show (*s* @ [*c*], {}) ∉ *F* (*S a*)
proof(*insert chain-lemma*[*OF chain*, *of a a*′],*erule disjE*)
assume *before*: *S a* ⊑ *S a*′

```
        show (s @ [c], {}) ∉ F (S a)
          using s-c-trace-not-trace-somewhere  before
          apply(case-tac s @ [c] ∉ D (S a),
                simp-all add: T-F-spec before[THEN le-approx2T,symmetric])
          apply(erule contrapos-nn)
          apply(simp only: T-F-spec[symmetric])
          apply(auto dest!:min-elems6[OF defined])
          done
      next
        assume after:S a' ⊑ S a
        show (s @ [c], {}) ∉ F (S a)
          using s-c-trace-not-trace-somewhere
          by(simp add:T-F-spec after[THEN le-approx2T]
                    s-c-trace-not-trace-somewhere[THEN NF-ND])
      qed
    qed
  qed
next
  fix s::'a trace   fix X::'a event set
  assume (s @ [tick], {}) ∈ FAILURES (⋂ a::nat. F (S a), ⋂ a::nat. D (S a))
  thus   (s, X − {tick}) ∈ FAILURES (⋂ a::nat. F (S a), ⋂ a::nat. D (S a))
      by(auto simp: DIVERGENCES-def FAILURES-def
            intro! : is-processT6[rule-format])
next
  fix s t ::'a trace
  assume s : DIVERGENCES (⋂ a::nat. F (S a), ⋂ a::nat. D (S a))
  and    tickFree s and front-tickFree t
  thus   s @ t ∈ DIVERGENCES (⋂ a::nat. F (S a), ⋂ a::nat. D (S a))
      by(auto simp: DIVERGENCES-def FAILURES-def
            intro: is-processT7[rule-format])
next
  fix s::'a trace  fix X::'a event set
  assume s ∈ DIVERGENCES (⋂ a::nat. F (S a), ⋂ a::nat. D (S a))
  thus   (s, X) ∈ FAILURES (⋂ a::nat. F (S a), ⋂ a::nat. D (S a))
      by(auto simp: DIVERGENCES-def FAILURES-def
              intro: is-processT8[rule-format])
next
  fix s::'a trace
  assume s @ [tick] ∈ DIVERGENCES (⋂ a::nat. F (S a), ⋂ a::nat. D (S a))
  thus   s ∈ DIVERGENCES (⋂ a::nat. F (S a), ⋂ a::nat. D (S a))
      by(auto simp: DIVERGENCES-def FAILURES-def
            intro: is-processT9[rule-format])
qed
```

lemmas *Rep-Abs-LUB* = *Abs-Process-inverse*[*simplified Process-def*,
simplified, *OF is-process-REP-LUB*,
simplified]

lemma *F-LUB*: *chain S* $\Longrightarrow$ *F*(*lim-proc*(*range S*)) = *INTER* (*range S*) *F*
by(*simp add*: *lim-proc-def* , *subst F-def*, *auto simp*: *FAILURES-def Rep-Abs-LUB*)

lemma *D-LUB*: *chain S* $\Longrightarrow$ *D*(*lim-proc*(*range S*)) = *INTER* (*range S*) *D*
by(*simp add*: *lim-proc-def* , *subst D-def*, *auto simp*: *DIVERGENCES-def Rep-Abs-LUB*)

lemma *T-LUB*: *chain S* $\Longrightarrow$ *T*(*lim-proc*(*range S*)) = *INTER* (*range S*) *T*
apply(*simp add*: *lim-proc-def* , *subst T-def*)
apply(*simp add*: *TRACES-def FAILURES-def Rep-Abs-LUB*)
apply(*auto intro*: *F-T*, *rule-tac x*={} **in** *exI*, *auto intro*: *T-F*)
done

schematic-lemma *D-LUB-2*: *chain S* $\Longrightarrow$ *t* $\in$ *D*(*lim-proc*(*range S*)) = *?X*
apply(*subst D-LUB*, *simp*)
apply(*rule trans*, *simp*)
apply(*simp*)
done

schematic-lemma *T-LUB-2*: *chain S* $\Longrightarrow$ (*t* $\in$ *T* (*lim-proc* (*range S*))) = *?X*
apply(*subst T-LUB*, *simp*)
apply(*rule trans*, *simp*)
apply(*simp*)
done

instance
process :: (*type*) *cpo*
proof
fix *S* ::*nat* $\Rightarrow$ 'α *process*
assume *C*:*chain S* **thus** $\exists$ *x*. *range S* <<| *x*
proof −
have *lim-proc-is-ub* :*range S* <| *lim-proc* (*range S*)
proof (*auto simp*: *C is-ub-def le-approx-def*
F-LUB D-LUB T-LUB Ra-def)
fix *a a'* :: *nat* **fix** *s*::'α *trace* **fix** *X*::'α *event set*
assume *A*:*s* $\notin$ *D* (*S a'*)
assume *B*:(*s*, *X*) $\in$ *F* (*S a'*)
show (*s*, *X*) $\in$ *F* (*S a*)
apply(*insert A B chain-lemma*[*OF C*, *of a a'*],*erule disjE*)
apply(*drule le-approx-lemma-F*, *auto*)
apply(*subst le-approx2*[*OF - A*],*simp-all*)
done
next

```
        fix a a' :: nat fix s::'α trace
        assume A:s ∈ min-elems (D (S a'))
        show     s ∈ T (S a)
          using A chain-lemma[OF C, of a a']
          by(auto dest:elem-min-elems le-approx1 le-approx3 intro: D-T)
      qed
    have lim-proc-is-lub1:
        ∀ u . (range S <| u ⟶  D u ⊆ D (lim-proc (range S)))
        by(auto simp: C D-LUB, frule-tac i=a in Porder.ub-rangeD,
          auto dest: le-approx1)
    have lim-proc-is-lub2:
        ∀ u . range S <| u ⟶ (∀ s.  s ∉ D (lim-proc (range S))
                              ⟶ Ra (lim-proc (range S)) s = Ra u s)
        proof(auto simp: is-ub-def C D-LUB F-LUB Ra-def)
          fix u s fix a::nat fix X
          assume A: ∀ y. S y ⊑ u
          assume B: s ∉ D (S a)
          assume C: ∀ x. (s, X) ∈ F (S x)
          show  (s, X) ∈ F u
            apply(insert A B C)
            apply(erule-tac x=a in allE,simp add: le-approx2)
          done
        next
          fix u s a a' X
          assume A: ∀ y. S y ⊑ u
          assume B: s ∉ D (S a)
          assume C: (s, X) ∈ F u
          show  (s, X) ∈ F (S a')
            apply(insert A B C)
            apply(metis le-approx2 is-processT8)
          done
        qed

    have lim-proc-is-lub3:
        ∀ u. range S <| u ⟶  min-elems(D(lim-proc(range S))) ⊆ T u

        sorry

     show ?thesis
     apply(rule-tac x=lim-proc (S ‘ UNIV) in exI)
     apply(simp add: le-approx-def is-lub-def lim-proc-is-ub)
     apply(rule allI,rule impI,
          simp add: lim-proc-is-lub1 lim-proc-is-lub2 lim-proc-is-lub3)
    done
  qed
qed
```

instance
process :: (*type*) *pcpo*
proof
show ∃ *x*::'*a process*. ∀ *y*::'*a process*. *x* ⊑ *y*
proof −
have *is-process-witness* :
is-process({(*s*,*X*). *front-tickFree s*},{*d*. *front-tickFree d*})
apply(*auto simp*:*is-process-def FAILURES-def DIVERGENCES-def*)
apply(*auto elim*!: *tickFree-implies-front-tickFree front-tickFree-dw-closed*
front-tickFree-append)
done
have *bot-inverse* :
Rep-Process(*Abs-Process*({(*s*, *X*). *front-tickFree s*},*Collect front-tickFree*))=
({(*s*, *X*). *front-tickFree s*}, *Collect front-tickFree*)
by(*subst Abs-Process-inverse*, *simp-all add*: *Process-def is-process-witness*)
have *divergences-frontTickFree*:
⋀*y x*. *x* ∈ *snd* (*Rep-Process y*) ⟹ *front-tickFree x*
by(*rule D-imp-front-tickFree*, *simp add*: *D-def DIVERGENCES-def*)
have *failures-frontTickFree*:
⋀*y s x*. (*s*, *x*) ∈ *fst* (*Rep-Process y*) ⟹ *front-tickFree s*
by(*rule is-processT2*[*rule-format*],
simp add: *F-def FAILURES-def*)
have *minelems-contains-mt*:
⋀*y x*. *x* ∈ *min-elems* (*Collect front-tickFree*) ⟹ *x* = []
by(*simp add*: *min-elems-def front-tickFree-charn*,*safe*,
auto simp: *Nil-elem-T*)
show *?thesis*
apply(*rule-tac x*=*Abs-Process* ({(*s*,*X*). *front-tickFree s*},{*d*. *front-tickFree d*})
in *exI*)
apply(*auto simp*: *le-approx-def bot-inverse Ra-def*
F-def D-def FAILURES-def DIVERGENCES-def
divergences-frontTickFree failures-frontTickFree)
apply (*metis minelems-contains-mt Nil-elem-T*)
done
qed
qed

1.7 Process Refinement is a Partial Ordering

The following type instantiation declares the refinement order _ ≤ _ written `_ <= _`. It captures the intuition that more concrete processes should be more deterministic and more defined.

instantiation
process :: (*type*) *ord*
begin

definition *le-ref-def* : *P* ≤ *Q* ≡ *D Q* ⊆ *D P* ∧ *F Q* ⊆ *F P*

definition *less-ref-def* : $(P::'a\ process) < Q \equiv P \leq Q \wedge P \neq Q$

instance ..

end

lemma *le-approx-implies-le-ref*: $(P::'\alpha\ process) \sqsubseteq Q \Longrightarrow P \leq Q$
by(*simp add*: *le-ref-def le-approx1 le-approx-lemma-F*)

lemma *le-ref1*: $P \leq Q \Longrightarrow D\ Q \subseteq D\ P$
by(*simp add*: *le-ref-def*)

lemma *le-ref2*: $P \leq Q \Longrightarrow F\ Q \subseteq F\ P$
by(*simp add*: *le-ref-def*)

lemma *le-ref2T* : $P \leq Q \Longrightarrow T\ Q \subseteq T\ P$
by (*rule subsetI*) (*simp add*: *T-F-spec*[*symmetric*] *le-ref2*[*THEN subsetD*])

instance *process* :: (*type*) *order*
proof
fix $P\ Q\ R :: 'a\ process$
{
show $(P < Q) = (P \leq Q \wedge \neg\ Q \leq\ P)$
by(*auto simp*: *le-ref-def less-ref-def Process-eq-spec*)
next
show $P \leq P$ **by**(*simp add*: *le-ref-def*)
next
assume $P \leq Q$ **and** $Q \leq R$ **then show** $P \leq R$
by (*simp add*: *le-ref-def*, *auto*)
next
assume $P \leq Q$ **and** $Q \leq P$ **then show** $P = Q$
by(*auto simp*: *le-ref-def Process-eq-spec*)
}
qed

lemma *lim-proc-is-ub*:*chain S* $\Longrightarrow$ *range S* $<|$ *lim-proc* (*range S*)
apply (*auto simp*: *is-ub-def le-approx-def F-LUB D-LUB T-LUB Ra-def*)
apply(*frule-tac i*=*y* **and** *k*=*a* **in** *chain-lemma*, *erule disjE*)
apply(*frule-tac s*=*s* **and** *X*=*x* **in** *le-approx2*, *simp-all*)
apply(*drule le-approx-lemma-F*, *subst* (*asm*) *subset-iff*,
erule-tac x=(*s*, *x*) **in** *allE*, *auto*)
apply(*frule-tac i*=*y* **and** *k*=*a* **in** *chain-lemma*, *erule disjE*)
by(*auto dest*:*elem-min-elems le-approx1 le-approx3 intro*: *D-T*)

lemma *lim-proc-is-lub1*:
chain S $\Longrightarrow \forall\ u\ .\ (range\ S <|\ u \longrightarrow\ D\ u \subseteq D\ (lim\text{-}proc\ (range\ S)))$
by(*auto simp*: *D-LUB*, *frule-tac i*=*a* **in** *Porder.ub-rangeD*, *auto dest*: *le-approx1*)

lemma *lim-proc-is-lub2*:
chain S $\Longrightarrow \forall\ u\ .\ range\ S <|\ u \longrightarrow (\forall\ s.\ \ s \notin D\ (lim\text{-}proc\ (range\ S))$
$\longrightarrow Ra\ (lim\text{-}proc\ (range\ S))\ s = Ra\ u\ s)$
apply(*auto simp*: *is-ub-def D-LUB F-LUB Ra-def*)
apply(*erule-tac x*=*a* **in** *allE*,*simp add*: *le-approx2*)
apply(*metis le-approx2 is-processT8*)
done

lemma *legacy-imp-conj*: $(P \dashrightarrow Q\ \&\ R') = ((P \dashrightarrow Q)\ \&\ (P \dashrightarrow R'))$
by *auto*

lemma *legacy-all-conj-distr*: $(!\ x.\ p\ x\ \&\ q\ x) = ((!\ x.\ p\ x)\ \&\ (!\ x.\ q\ x))$
by *auto*

lemma *legacy-INTER-def*: $INTER\ A\ B == \{y.\ !\ x{:}A.\ y : B\ x\}$
sorry

lemma *lim-proc-is-lub3*:
assumes *A*: *directed X*
shows $\forall\ u.\ X <|\ u \longrightarrow\ min\text{-}elems(D(lim\text{-}proc\ X)) \subseteq T\ u$
apply(*insert A*)
apply(*auto simp*: *is-ub-def D-LUB Ra-def*)
apply(*auto simp*: *min-elems-def le-approx-def directed-def*
legacy-imp-conj legacy-all-conj-distr legacy-INTER-def Ball-def)

sorry

lemma *limproc-is-lub*: *chain S* $\Longrightarrow$ *range S* $<<|$ *lim-proc* (*range S*)
apply (*auto simp*: *is-lub-def lim-proc-is-ub*)
apply (*simp add*: *le-approx-def is-lub-def lim-proc-is-ub*)
sorry

lemma *limproc-is-thelub*: *chain S* $\Longrightarrow$ *Lub S* = *lim-proc* (*range S*)
by (*frule limproc-is-lub*,*frule Porder.po-class.lub-eqI*, *simp*)

end

theory *Bot*
imports *Process*
begin

definition *Bot* :: $'\alpha$ *process*
where *Bot* ≡ *Abs-Process* ({(*s*,*X*). *front-tickFree s*}, {*d*. *front-tickFree d*})

lemma *is-process-REP-Bot* :
is-process ({(*s*,*X*). *front-tickFree s*}, {*d*. *front-tickFree d*})
by(*auto simp*: *tickFree-implies-front-tickFree is-process-def*
FAILURES-def DIVERGENCES-def
elim: *Process.front-tickFree-dw-closed*
elim: *Process.front-tickFree-append*)

lemma *Rep-Abs-Bot* :*Rep-Process* (*Abs-Process* ({(*s*,*X*). *front-tickFree s*},{*d*. *front-tickFree d*})) =
({(*s*,*X*). *front-tickFree s*},{*d*. *front-tickFree d*})
by(*subst Abs-Process-inverse*, *simp-all only*: *CollectI Process-def is-process-REP-Bot*)

lemma *F-Bot*[*simp*]: *F Bot* = {(*s*,*X*). *front-tickFree s*}
by(*simp add*: *Bot-def FAILURES-def F-def Rep-Abs-Bot*)

lemma *D-Bot*[*simp*]: *D Bot* = {*d*. *front-tickFree d*}
by(*simp add*: *Bot-def DIVERGENCES-def D-def Rep-Abs-Bot*)

lemma *T-Bot*[*simp*]: *T Bot* = {*s*. *front-tickFree s*}
by(*simp add*: *Bot-def TRACES-def T-def FAILURES-def Rep-Abs-Bot*)

This is the key result: ⊥ — which we know to exist from the process instantiation — is equal Bot .

lemma *Bot-is-UU*: *Bot* = ⊥
apply(*auto simp*: *Pcpo.eq-bottom-iff Process.le-approx-def Ra-def*
min-elems-Collect-ftF-is-Nil Process.Nil-elem-T
elim: *D-imp-front-tickFree*)
apply(*metis Process.is-processT2*)
done

lemma *F-UU*[*simp*]: *F* ⊥ = {(*s*,*X*). *front-tickFree s*}
by(*simp add*: *Bot-is-UU*[*symmetric*])

lemma *D-UU*[*simp*]: *D* ⊥ = {*d*. *front-tickFree d*}
by(*simp add*: *Bot-is-UU*[*symmetric*])

lemma *T-UU*[*simp*]: *T* ⊥ = {*s*. *front-tickFree s*}
by(*simp add*: *Bot-is-UU*[*symmetric*])

end

theory *Skip*
imports *Process*

begin

definition *SKIP* :: *'a process*
where *SKIP* ≡ *Abs-Process* ({(*s*, *X*). *s* = [] ∧ *tick* ∉ *X*} ∪ {(*s*, *X*). *s* = [*tick*]}, {})

lemma *is-process-REP-Skip*:
is-process ({(*s*, *X*). *s* = [] ∧ *tick* ∉ *X*} ∪ {(*s*, *X*). *s* = [*tick*]}, {})
apply(*auto simp*: *FAILURES-def DIVERGENCES-def front-tickFree-def is-process-def*)
apply(*erule contrapos-np*,*drule neq-Nil-conv*[*THEN iffD1*], *auto*)
done

lemma *is-process-REP-Skip2*:
is-process ({[]} × {*X*. *tick* ∉ *X*} ∪ {(*s*, *X*). *s* = [*tick*]}, {})
apply(*insert is-process-REP-Skip*)
apply *auto* **done**

lemmas *process-prover* = *Process-def Abs-Process-inverse*
FAILURES-def TRACES-def
DIVERGENCES-def is-process-REP-Skip

lemma *F-SKIP*:
F SKIP = {(*s*, *X*). *s* = [] ∧ *tick* ∉ *X*} ∪ {(*s*, *X*). *s* = [*tick*]}
by(*simp add*: *process-prover SKIP-def F-def is-process-REP-Skip2*)

lemma *D-SKIP*: *D SKIP* = {}
by(*simp add*: *process-prover SKIP-def D-def is-process-REP-Skip2*)

lemma *T-SKIP*: *T SKIP* ={[],[*tick*]}
by(*auto simp*: *process-prover SKIP-def T-def is-process-REP-Skip2*)

end

theory *Legacy*
imports *Process*
begin

lemmas *tF-Nil* = *tickFree-Nil*
lemmas *tF-Cons* = *tickFree-Cons*
lemmas *NtF-tick* = *non-tickFree-tick*
lemmas *tF-rev* = *tickFree-rev*
lemmas *ftF-Nil* = *front-tickFree-Nil*
lemmas *tF-imp-ftF* = *tickFree-implies-front-tickFree*
lemmas *ftF-imp-f-is-tF* = *front-tickFree-implies-tickFree*
lemmas *NtF-ftF-ex* = *nonTickFree-n-frontTickFree*
lemmas *Nconj-eq-disjN* = *HOL.nnf-simps*(*1*)
lemmas *Ndisj-eq-conjN* = *HOL.nnf-simps*(*2*)
lemmas *imp-disj* = *HOL.nnf-simps*(*3*)
lemmas *conj-imp* = *HOL.imp-conjL*
lemmas *Pair-fst-snd-eq* = *surjective-pairing*
lemmas *t-F-T* = *Failures-implies-Traces*
lemmas *f-F-is-tF* = *front-trace-is-tickfree*
lemmas *f-T-is-tF* = *trace-with-Tick-implies-tickFree-front*
lemmas *D-ftF-subset* = *D-front-tickFree-subset*
lemmas *append-T-tF* = *append-T-imp-tickFree*
lemmas *T-tF* = *append-single-T-imp-tickFree*
lemmas *T-tF1* = *append-single-T-imp-tickFree*
lemmas *T-NtF-ex* = *T-nonTickFree-imp-decomp*

definition *member* :: *'a list* ⇒ *'a* ⇒ *bool* **where**
mem-iff [*code-post*]: *member xs x* ⟷ *x* ∈ *set xs*

lemmas *is-process3-S* = *is-process3* [*rule-format*]
lemmas *is-process2-S* = *is-process2* [*THEN spec*, *THEN spec*, *THEN mp*]
lemmas *ProcessT-eqI* = *Process-eq-spec*[*THEN iffD2*,*OF conjI*]
lemmas *is-processT-spec* = *process-charn*
lemmas *is-processT2-TR-S* = *is-processT2-TR*[*rule-format*]
lemmas *is-processT2-S* = *is-processT2*[*rule-format*]
lemmas *is-processT3-S* = *is-processT3*[*rule-format*]
lemmas *is-processT4-S* = *is-processT4*[*rule-format*]
lemmas *is-processT5-S* = *is-processT5*[*rule-format*, *OF conjI*]
lemmas *is-processT6-S* = *is-processT6*[*rule-format*]
lemmas *is-processT9-S* = *is-processT9* [*rule-format*]
lemmas *subsetND* = *Set.contra-subsetD*
lemmas *D-ftF* = *D-imp-front-tickFree*
lemmas *ftF-imp-f-is-tF1* = *front-tickFree-implies-tickFree*

lemmas *less-eq-process-def* = *Process.le-ref-def*

lemma *Collect-eq-spec*:
$\{x.\ P\ x\} = \{x.\ Q\ x\} = (\forall\ x.\ P\ x = Q\ x)$
by *auto*

lemmas *subset-spec* = *subset-iff*[*THEN iffD1*,*rule-format*]

lemmas *rec-ord-implies-ref-ord* = *le-approx-implies-le-ref*

lemmas *process-ref-ord-def* = *Process.le-ref-def*

lemmas *sq-eq-process* = *le-approx-def*
lemmas *process-ord-def* = *sq-eq-process*

lemmas *proc-ord1*=*le-approx1*
lemmas *proc-ord2*=*le-approx2*
lemmas *proc-ord3*=*le-approx3*
lemmas *proc-ord2T*=*le-approx2T*
lemmas *proc-ord-lemma-F*=*le-approx-lemma-F*
lemmas *proc-ord-lemma-T*=*le-approx-lemma-T*

lemmas *le-approx-implies-ref-ord* = *le-approx-implies-le-ref*
lemmas *ref-ord1* = *le-ref1*
lemmas *ref-ord2* = *le-ref2*
lemmas *ref-ord2T* = *le-ref2T*

end

2 The STOP Process Definition

theory *Stop*
imports *Process Legacy*
begin

definition *STOP* :: $'\alpha$ *process*
where *STOP* $\equiv$ *Abs-Process* $(\{(s,\ X).\ s = []\},\ \{\})$

lemma *is-process-REP-STOP*: *is-process* $(\{(s,\ X).\ s = []\},\{\})$

by(*simp add*: *is-process-def FAILURES-def DIVERGENCES-def*)

lemma *Rep-Abs-STOP* : *Rep-Process* (*Abs-Process* ({(*s*, *X*). *s* = []},{})) = ({(*s*, *X*). *s* = []},{})
by(*subst Abs-Process-inverse*, *simp add*: *Process-def is-process-REP-STOP*, *auto*)

lemma *F-STOP* : *F STOP* = {(*s*,*X*). *s* = []}
by(*simp add*: *STOP-def FAILURES-def F-def Rep-Abs-STOP*)

lemma *D-STOP*: *D STOP* = {}
by(*simp add*: *STOP-def DIVERGENCES-def D-def Rep-Abs-STOP*)

lemma *T-STOP*: *T STOP* = {[]}
by(*simp add*: *STOP-def TRACES-def FAILURES-def T-def Rep-Abs-STOP*)

end

3 The Multi-Prefix Operator Definition

theory *Mprefix*
imports *Process Legacy*
begin

4 Backpatch Isabelle 2009-1

definition
contlub :: ('*a*::*cpo* ⇒ '*b*::*cpo*) ⇒ *bool* — first cont. def **where**
contlub f = (∀ *Y*. *chain Y* ⟶ *f* (⨆ *i*. *Y i*) = (⨆ *i*. *f* (*Y i*)))

lemma *contlubE*:
⟦*contlub f*; *chain Y*⟧ ⟹ *f* (⨆ *i*. *Y i*) = (⨆ *i*. *f* (*Y i*))
by (*simp add*: *contlub-def*)

lemma *monocontlub2cont*: ⟦*monofun f*; *contlub f*⟧ ⟹ *cont f*
apply (*rule contI*)
apply (*rule thelubE*)
apply (*erule* (*1*) *ch2ch-monofun*)
apply (*erule* (*1*) *contlubE* [*symmetric*])
done

lemma *contlubI*:
(⋀*Y*. *chain Y* ⟹ *f* (⨆ *i*. *Y i*) = (⨆ *i*. *f* (*Y i*))) ⟹ *contlub f*
by (*simp add*: *contlub-def*)

lemma *cont2contlub*: *cont f* ⟹ *contlub f*

apply (*rule contlubI*)
apply (*rule Porder.po-class.lub-eqI* [*symmetric*])
apply (*erule* (*1*) *contE*)
done

5 The core of it . . .

definition *Mprefix* :: ['*a set*,'*a* => '*a process*] => '*a process* **where**
Mprefix A P ≡ *Abs-Process*(
{(*tr*,*ref*). *tr* = [] ∧ *ref Int* (*ev* ' *A*) = {}} ∪

{(*tr*,*ref*). *tr* ≠ [] ∧ *hd tr* ∈ (*ev* ' *A*) ∧
(∃ *a*. *ev a* = (*hd tr*)∧(*tl tr*,*ref*)∈*F*(*P a*))},
{*d*. *d* ≠ [] ∧ *hd d* ∈ (*ev* ' *A*) ∧
(∃ *a*. *ev a* = *hd d* ∧ *tl d* ∈ *D*(*P a*))})

syntax(*HOL*)
@*mprefix* :: [*pttrn*,'*a set*,'*a process*]=>'*a process* ((*3*[−]- : - −> -)[*0*,*0*,*64*]*64*)

syntax(*xsymbols*)
@*mprefix* :: [*pttrn*,'*a set*,'*a process*]=>'*a process* ((*3*□ - ∈ - → -)[*0*,*0*,*64*]*64*)

translations
□ *x* ∈ *A* → *P* == *CONST Mprefix A* (% *x* . *P*)

5.1 Well-foundedness of Mprefix

lemma *is-process-REP-Mp* :
is-process ({(*tr*,*ref*). *tr*=[] ∧ *ref* ∩ (*ev* ' *A*) = {}} ∪
{(*tr*,*ref*). *tr* ≠ [] ∧ *hd tr* ∈ (*ev* ' *A*) ∧
(∃ *a*. *ev a* = (*hd tr*) ∧ (*tl tr*,*ref*) ∈ *F*(*P a*))},
{*d*. *d* ≠ [] ∧ *hd d* ∈ (*ev* ' *A*) ∧
(∃ *a*. *ev a* = *hd d* ∧ *tl d* ∈ *D*(*P a*))})
(**is** *is-process*(*?f*, *?d*))
proof (*simp only*:*is-process-def FAILURES-def DIVERGENCES-def*
Product-Type.fst-conv Product-Type.snd-conv,
intro conjI allI impI)
case *goal1*
have *1*: ([],{}) ∈ *?f* **by** *simp*
show *?case* **by**(*simp add*: *1*)
next
case *goal2* **note** *asm2* = *goal2*
{
fix *s*:: '*a event list* **fix** *X*::'*a event set*
assume *H* : (*s*, *X*) ∈ *?f*
have *front-tickFree s*
apply(*insert H*, *auto simp*:*mem-iff front-tickFree-def tickFree-def*
dest!:*list-nonMt-append*)
apply(*case-tac ta*, *auto simp*: *front-tickFree-charn*

```
                    dest! : is-processT2[rule-format])
    apply(simp add: tickFree-def mem-iff)
    done
  } note 2 = this
  show ?case  by(rule 2[OF asm2])
next
  case goal3  note asm3 = goal3
  {
    fix s t :: 'a event list
    assume H : (s @ t, {}) ∈ ?f
    have       (s, {}) ∈ ?f
    using H  by(auto elim: is-processT3[rule-format])
  } note 3 = this
  show ?case by(rule 3[OF asm3])
next
  case goal4  note asm4 = goal4
  {
    fix    s:: 'a event list fix X Y::'a event set
    assume H1: (s, Y) ∈ ?f
    assume H2: X ⊆ Y
    have       (s, X) ∈ ?f
      using H1 H2 by(auto intro: is-processT4[rule-format])
   } note 4 = this
  show ?case by(rule 4 [where Ya2=Y])(simp-all only: asm4)
next
  case goal5  note asm5 = goal5
  {
    fix s:: 'a event list fix X Y::'a event set
    assume H1 : (s, X) ∈ ?f
    assume H2 : ∀ c. c∈Y ⟶ (s @ [c], {}) ∉ ?f
    have 5:     (s, X ∪ Y) ∈ ?f
      using H1 H2 by(auto intro!: is-processT1 is-processT5[rule-format])
   } note 5 = this
   show ?case  by(rule 5,simp only: asm5,
                  rule asm5[THEN conjunct2])
next
  case goal6  note asm6 = goal6
  {
    fix s:: 'a event list fix X::'a event set
    assume H : (s @ [tick], {}) ∈ ?f
    have 6:   (s, X − {tick}) ∈ ?f
      using H by(cases s, auto dest!: is-processT6[rule-format])
  } note 6 = this
  show ?case by(rule 6[OF asm6])
next
  case goal7  note asm7 = goal7
  {
    fix s t:: 'a event list fix X::'a event set
    assume H1 : s ∈  ?d
```

```
    assume H2 :  tickFree s
    assume H3 : front-tickFree t
    have 7:     s @ t ∈ ?d
      using H1 H2 H3 by(auto intro!: is-processT7-S, cases s, simp-all)
  } note 7 = this
  show ?case by(rule 7, insert asm7, auto)
next
  case goal8  note asm8 = goal8
  {
    fix s:: 'a event list fix X::'a event set
    assume H : s ∈ ?d
    have 8:    (s, X) ∈ ?f
      using H by(auto simp: is-processT8-S)
  } note 8 = this
  show ?case by(rule 8[OF asm8])
next
  case goal9  note asm9 = goal9
  {
  fix s:: 'a event list
  assume H: s @ [tick] ∈ ?d
  have 9:   s ∈ ?d
    using H apply(auto)
    apply(cases s, simp-all)
    apply(cases s, auto intro: is-processT9[rule-format])
    done
  } note 9 = this
  show ?case by(rule 9, rule asm9)
  qed


lemma Rep-Abs-Mp :
assumes H1 : f = {(tr, ref). tr = [] ∧ ref ∩ ev ‘ A = {}} ∪
                 {(tr, ref). tr ≠ [] ∧ hd tr ∈ ev ‘ A
                         ∧ (∃ a. ev a = hd tr ∧ (tl tr, ref) ∈ F (P a))}
    and H2 : d = {d. d ≠ [] ∧  hd d ∈ (ev ‘ A) ∧
                   (∃ a. ev a = hd d  ∧ tl d ∈ D(P a))}
shows Rep-Process (Abs-Process (f,d)) = (f,d)
by(subst Abs-Process-inverse,
   simp-all only: H1 H2 CollectI Process-def is-process-REP-Mp)
```

5.2 Projections in Prefix

```
lemma F-Mprefix :
F(□ x ∈ A → P x) = {(tr,ref). tr=[] ∧  ref ∩ (ev ‘ A) = {}} ∪
                   {(tr,ref). tr ≠ [] ∧ hd tr ∈ (ev ‘ A) ∧
                           (∃ a. ev a = (hd tr) ∧ (tl tr,ref) ∈ F(P a))}
by(simp add:Mprefix-def F-def Rep-Abs-Mp FAILURES-def)
```

lemma *D-Mprefix*:
$D(\Box\ x \in A \to P\ x) = \{d.\ d \neq\ [] \wedge\ \ hd\ d \in (ev\ `\ A) \wedge$
$(\exists\ a.\ ev\ a = hd\ d\ \wedge tl\ d \in D(P\ a))\}$
by(*simp add*:*Mprefix-def D-def Rep-Abs-Mp DIVERGENCES-def*)

lemma *T-Mprefix*:
$T(\Box\ x \in A \to P\ x)=\{s.\ s=[] \vee (\exists\ a.\ a \in A \wedge s \neq [] \wedge hd\ s = ev\ a \wedge tl\ s \in T(P\ a))\}$
by(*auto simp*: *T-F-spec*[*symmetric*] *F-Mprefix*)

5.3 Basic Properties

lemma *tick-T-Mprefix* [*simp*]: [*tick*] $\notin T(\Box\ x \in A \to P\ x)$
by(*simp add*:*T-Mprefix*)

lemma *Nil-Nin-D-Mprefix* [*simp*]: [] $\notin D(\Box\ x \in A \to P\ x)$
by(*simp add*: *D-Mprefix*)

5.4 Proof of Continuity Rule

lemma *mono-Mprefix1*:
$\forall\, a.\ P\ a \sqsubseteq Q\ a \Longrightarrow D\ (Mprefix\ A\ Q) \subseteq D\ (Mprefix\ A\ P)$
apply(*auto simp*: *D-Mprefix*)
apply(*erule-tac x*=*xa* **in** *allE*)
by(*auto elim*: *proc-ord1* [*THEN subsetD*])

lemma *mono-Mprefix2*:
$\forall\, x.\ P\ x \sqsubseteq Q\ x \Longrightarrow$
$\forall\, s.\ s \notin D\ (Mprefix\ A\ P) \longrightarrow Ra\ (Mprefix\ A\ P)\ s = Ra\ (Mprefix\ A\ Q)\ s$
apply(*auto simp*: *Ra-def D-Mprefix F-Mprefix*)
apply(*erule-tac x* = *xa* **in** *allE*, *simp add*: *proc-ord2a*)+
done

lemma *mono-Mprefix3* :
$\forall\, x.\ P\ x \sqsubseteq Q\ x \Longrightarrow min\text{-}elems\ (D\ (Mprefix\ A\ P)) \subseteq T\ (Mprefix\ A\ Q)$
apply(*auto simp*: *min-elems-def D-Mprefix T-Mprefix image-def*)
apply(*erule-tac x*=*xa* **in** *allE*)
apply(*auto simp*:*min-elems-def dest*!: *proc-ord3*)
sorry

lemma *mono-Mprefix0*:
$\forall\, x.\ P\ x \sqsubseteq Q\ x \Longrightarrow Mprefix\ A\ P \sqsubseteq Mprefix\ A\ Q$
apply(*simp add*: *process-ord-def mono-Mprefix1 mono-Mprefix3*)
apply(*rule mono-Mprefix2*)
apply(*auto simp*: *process-ord-def*)
done

lemma *mono-Mprefix* : *monofun*(*Mprefix A*)
by(*auto simp*: *Fun-Cpo.below-fun-def monofun-def mono-Mprefix0*)

lemma *contlub-Mprefix* : *contlub*(*Mprefix A*)
apply(*auto simp*: *contlub-def*)
sorry

lemma *cont-revert2cont-pointwise*:
$\bigwedge x.\ cont\ (f\ x) \Longrightarrow cont\ (\lambda x\ y.\ f\ y\ x)$
sorry

lemma *Mprefix-cont* [*simp*]:
$(\bigwedge x.\ cont\ (f\ x)) \Longrightarrow cont\ (\lambda y.\ \Box\ z \in\ A \rightarrow f\ z\ y)$
apply(*rule-tac* $f = \lambda z\ y.\ (f\ y\ z)$ **in** *Cont.cont-compose*)
apply(*rule monocontlub2cont*)
apply(*auto intro*: *mono-Mprefix contlub-Mprefix cont-revert2cont-pointwise*)
done

lemmas *proc-ord1D* = *proc-ord1* [*THEN subsetD*]

lemmas *proc-ord2b* = *proc-ord2a* [*THEN sym*]
lemmas *le-fun-def* = *Fun-Cpo.below-fun-def*
lemmas *cont-compose1* = *Cont.cont-compose*
lemmas *mono-contlub-imp-cont* = *monocontlub2cont*

5.5 High-level Syntax

definition *read* :: ['a=>'b,'a set, 'a => 'b process] => 'b process
where *read c A P* ≡ *Mprefix*(*c* ‘ *A*) (*P o* (*inv c*))
definition *write* :: ['a=>'b, 'a, 'b process] => 'b process
where *write c a P* ≡ *Mprefix* {*c a*} (λ *x*. *P*)
definition *write0* :: ['a, 'a process] => 'a process
where *write0 a P* ≡ *Mprefix* {*a*} (λ *x*. *P*)

syntax
-read :: [*id*, *pttrn*, 'a process] => 'a process
((3-‘?‘- /→ -) [0,0,28] 28)
-readX :: [*id*, *pttrn*, *bool*,'a process] => 'a process
((3-‘?‘-‘|‘- /→ -) [0,0,28] 28)
-readS :: [*id*, *pttrn*, 'b set,'a process] => 'a process
((3-‘?‘-‘:‘- /→ -) [0,0,28] 28)

-write :: [*id*, 'b, 'a process] => 'a process
((3-‘!‘- /→ -) [0,0,28] 28)
-writeS :: ['a, 'a process] => 'a process

((*3- /→ -*) [*0*,*28*] *28*)

translations
-read c p P == *CONST read c CONST UNIV* (λ*p*. *P*)
-write c p P == *CONST write c p P*
-readX c p b P => *CONST read c* {*p*. *b*} (λ*p*. *P*)
-writeS a P == *CONST write0 a P*

lemma *read-cont*[*simp*]:
(⋀*x*. *cont* (*f x*)) ⟹ *cont* (λ*y*. *c*‘*?*‘*x* → *f x y*)
by(*simp add*:*read-def o-def inv-def*)

lemma *write-cont*[*simp*]:
(⋀*x*. *cont* (*P*::(′*b*::*cpo* ⇒ ′*a process*)))
⟹ *cont*(λ*x*. (*c* ‘!‘ *a* → *P x*))
by(*simp add*:*write-def*)

lemma *write0-cont*[*simp*]:
cont (*P*::(′*b*::*cpo* ⇒ ′*a process*))
⟹ *cont*(λ*x*. (*a* → *P x*))
by(*simp add*:*write0-def*)

end

6 Deterministic Choice Operator Definition

theory *Det*
imports *Process*
begin

definition
det :: [′α *process*,′α *process*] ⇒ ′α *process* (**infixl** [+] *18*)
where *P* [+] *Q* ≡ *Abs-Process*({(*s*,*X*). *s* = [] ∧ (*s*,*X*) ∈ *F P* ∩ *F Q*}
∪ {(*s*,*X*). *s* ≠ [] ∧ (*s*,*X*) ∈ *F P* ∪ *F Q*}
∪ {(*s*,*X*). *s* = [] ∧ *s* ∈ *D P* ∪ *D Q*}
∪ {(*s*,*X*). *s* = [] ∧ *tick* ∉ *X* ∧ [*tick*] ∈ *T P* ∪ *T Q*},
D P ∪ *D Q*)

notation(*xsymbols*)
det (**infixl** □ *18*)

lemma *is-process-REP-D*:
is-process ({(*s*, *X*). *s* = [] ∧ (*s*, *X*) ∈ *F P* ∩ *F Q*} ∪
{(*s*, *X*). *s* ≠ [] ∧ (*s*, *X*) ∈ *F P* ∪ *F Q*} ∪
{(*s*, *X*). *s* = [] ∧ *s* ∈ *D P* ∪ *D Q*} ∪
{(*s*, *X*). *s* = [] ∧ *tick* ∉ *X* ∧ [*tick*] ∈ *T P* ∪ *T Q*},
D P ∪ *D Q*)
proof (*simp only*: *fst-conv snd-conv Process-def is-process-def*
DIVERGENCES-def FAILURES-def, *intro conjI*)
fix *P Q* :: *'α process*
let *?T* = {(*s*, *X*). *s* = [] ∧ (*s*, *X*) ∈ *F P* ∩ *F Q*} ∪
{(*s*, *X*). *s* ≠ [] ∧ (*s*, *X*) ∈ *F P* ∪ *F Q*} ∪
{(*s*, *X*). *s* = [] ∧ *s* ∈ *D P* ∪ *D Q*} ∪
{(*s*, *X*). *s* = [] ∧ *tick* ∉ *X* ∧ [*tick*] ∈ *T P* ∪ *T Q*}
{
show ([], {}) ∈ *?T*
by(*simp add*: *is-processT1*)
next
show ∀ *s X*. (*s*, *X*) ∈ *?T* ⟶ *front-tickFree s*
by(*auto simp*: *is-processT2*)
next
show ∀ *s t*. (*s* @ *t*, {}) ∈ *?T* ⟶ (*s*, {}) ∈ *?T*
by(*auto simp*: *is-processT1 dest*!: *is-processT3*[*rule-format*])
next
show ∀ *s X Y*. (*s*, *Y*) ∈ *?T* ∧ *X* ⊆ *Y* ⟶ (*s*, *X*) ∈ *?T*
by(*auto dest*: *is-processT4*[*rule-format*,*OF conj-commute*[*THEN iffD1*],*OF conjI*])
next
show ∀ *sa X Y*. (*sa*, *X*) ∈ *?T* ∧ (∀ *c*. *c* ∈ *Y* ⟶ (*sa* @ [*c*], {}) ∉ *?T*) ⟶ (*sa*, *X* ∪ *Y*) ∈ *?T*
by(*auto simp*: *is-processT5 T-F*)
next
show ∀ *s X*. (*s* @ [*tick*], {}) ∈ *?T* ⟶ (*s*, *X* − {*tick*}) ∈ *?T*
apply((*rule allI*)+, *rule impI*)
apply(*case-tac s*=[], *simp-all add*: *is-processT6*[*rule-format*] *T-F-spec*)
by(*auto simp*: *is-processT6*[*rule-format*] *T-F-spec*)
next
show ∀ *s t*. *s* ∈ *D P* ∪ *D Q* ∧ *tickFree s* ∧ *front-tickFree t* ⟶ *s* @ *t* ∈ *D P* ∪ *D Q*
by(*auto simp*: *is-processT7*)
next
show ∀ *s X*. *s* ∈ *D P* ∪ *D Q* ⟶ (*s*, *X*) ∈ *?T*
by(*auto simp*: *is-processT8*[*rule-format*])
next
show ∀ *s*. *s* @ [*tick*] ∈ *D P* ∪ *D Q* ⟶ *s* ∈ *D P* ∪ *D Q*
by(*auto intro*!:*is-processT9*[*rule-format*])
}
qed

lemma *Rep-Abs-D*:
Rep-Process

(*Abs-Process*
(\{(*s*, *X*). *s* = [] ∧ (*s*, *X*) ∈ *F P* ∩ *F Q*} ∪
\{(*s*, *X*). *s* ≠ [] ∧ (*s*, *X*) ∈ *F P* ∪ *F Q*} ∪
\{(*s*, *X*). *s* = [] ∧ *s* ∈ *D P* ∪ *D Q*} ∪
\{(*s*, *X*). *s* = [] ∧ *tick* ∉ *X* ∧ [*tick*] ∈ *T P* ∪ *T Q*},
D P ∪ *D Q*)) =
(\{(*s*, *X*). *s* = [] ∧ (*s*, *X*) ∈ *F P* ∩ *F Q*} ∪
\{(*s*, *X*). *s* ≠ [] ∧ (*s*, *X*) ∈ *F P* ∪ *F Q*} ∪
\{(*s*, *X*). *s* = [] ∧ *s* ∈ *D P* ∪ *D Q*} ∪
\{(*s*, *X*). *s* = [] ∧ *tick* ∉ *X* ∧ [*tick*] ∈ *T P* ∪ *T Q*},
D P ∪ *D Q*)
by(*simp only*:*CollectI Process-def Abs-Process-inverse is-process-REP-D*)

lemma *F-det* :
F(*P* □ *Q*) = \{(*s*,*X*). *s* = [] ∧ (*s*,*X*) ∈ *F P* ∩ *F Q*}
∪ \{(*s*,*X*). *s* ≠ [] ∧ (*s*,*X*) ∈ *F P* ∪ *F Q*}
∪ \{(*s*,*X*). *s* = [] ∧ *s* ∈ *D P* ∪ *D Q*}
∪ \{(*s*,*X*). *s* = [] ∧ *tick* ∉ *X* ∧ [*tick*] ∈ *T P* ∪ *T Q*}
by(*subst F-def*, *simp only*: *det-def Rep-Abs-D FAILURES-def fst-conv*)

lemma *D-det*: *D*(*P* □ *Q*) = *D P* ∪ *D Q*
by(*subst D-def*, *simp only*: *det-def Rep-Abs-D DIVERGENCES-def snd-conv*)

lemma *T-det*: *T*(*P* □ *Q*) = *T P* ∪ *T Q*
apply(*subst T-def*, *simp only*: *det-def Rep-Abs-D TRACES-def FAILURES-def*
fst-def snd-conv)
apply(*auto simp*: *T-F F-T is-processT1 Nil-elem-T*)
apply(*rule-tac x*={} **in** *exI*, *simp add*: *T-F F-T is-processT1 Nil-elem-T*)+
done

lemma *Det-commute*:(*P* □ *Q*) = (*Q* □ *P*)
by(*auto simp*: *Process-eq-spec F-det D-det*)

lemma *mono-D1*: *P* ⊑ *Q* ⟹ *D* (*Q* □ *S*) ⊆ *D* (*P* □ *S*)
apply (*drule le-approx1*)
by (*auto simp*: *Process-eq-spec F-det D-det*)

lemma *mono-D2*:
assumes *ordered*: *P* ⊑ *Q*
shows (∀ *s*. *s* ∉ *D* (*P* □ *S*) ⟶ *Ra* (*P* □ *S*) *s* = *Ra* (*Q* □ *S*) *s*)
proof −
have *A*:⋀*s t*. [] ∉ *D P* ⟹ [] ∉ *D S* ⟹ *s* = [] ⟹
([], *t*) ∈ *F P* ⟹ ([], *t*) ∈ *F S* ⟹ ([], *t*) ∈ *F Q*
by (*insert ordered*,*frule-tac X*=*t* **and** *s*=[] **in** *proc-ord2a*, *simp-all*)
have *B*:⋀*s t*. *s* ∉ *D P* ⟹ *s* ∉ *D S* ⟹
(*s* ≠ [] ∧ ((*s*, *t*) ∈ *F P* ∨ (*s*, *t*) ∈ *F S*) ⟶ (*s*, *t*) ∈ *F Q* ∨ (*s*, *t*) ∈ *F S*) ∧
(*s* = [] ∧ *tick* ∉ *t* ∧ ([*tick*] ∈ *T P* ∨ [*tick*] ∈ *T S*) ⟶

```
      ([], t) ∈ F Q ∧ ([], t) ∈ F S ∨ [] ∈ D Q ∨  [tick] ∈ T Q ∨  [tick] ∈ T S)
    apply(intro conjI impI, elim conjE disjE, rule disjI1)
    apply(simp-all add: proc-ord2a[OF ordered,symmetric])
    apply(elim conjE disjE,subst le-approx2T[OF ordered])
    apply(frule is-processT9-S-swap, simp-all)
    done
  have C: ⋀s. s ∉ D P ⟹ s ∉ D S ⟹
          {X. s = [] ∧ (s, X) ∈ F Q ∧ (s, X) ∈ F S ∨
              s ≠ [] ∧ ((s, X) ∈ F Q ∨ (s, X) ∈ F S) ∨
              s = [] ∧ s ∈ D Q ∨ s = [] ∧ tick ∉ X ∧
              ([tick] ∈ T Q ∨ [tick] ∈ T S)}
        ⊆ {X. s = [] ∧ (s, X) ∈ F P ∧ (s, X) ∈ F S ∨
              s ≠ [] ∧ ((s, X) ∈ F P ∨ (s, X) ∈ F S) ∨
              s = [] ∧ tick ∉ X ∧ ([tick] ∈ T P ∨ [tick] ∈ T S)}
    apply(intro subsetI, frule is-processT9-S-swap, simp)
    apply(elim conjE disjE, simp-all add: proc-ord2a[OF ordered,symmetric] is-processT8-S)
    apply(insert le-approx1[OF ordered] le-approx-lemma-T[OF ordered])
    by   (auto simp: proc-ord2a[OF ordered,symmetric])
  show ?thesis
  apply(simp add: Process-eq-spec F-det D-det Ra-def min-elems-def)
  apply(intro allI impI equalityI C, simp-all)
  apply(intro allI impI subsetI, simp)
  apply(metis A B)
  done
qed


lemma mono-D3 : P ⊑ Q ⟹ min-elems (D (P □ S)) ⊆ T (Q □ S)
apply (frule le-approx3)
apply (simp add: Process-eq-spec F-det T-det D-det Ra-def min-elems-def subset-iff)
apply (auto intro:D-T)
done

lemma mono-Det     : P ⊑ Q ⟹ (P □ S) ⊑ (Q □ S)
by (auto simp: le-approx-def mono-D1 mono-D2 mono-D3)


lemma mono-Det-sym : P ⊑ Q ⟹ (S □ P) ⊑ (S □ Q)
by (simp add: Det-commute mono-Det)

lemma all-conj-distrib1: ((∀ x. P x) ∧ (∀ x. Q x)) = (∀ x. P x ∧ Q x)
by (auto)

lemma all-conj-distrib2: ((∀ x. P x) ∧ Q) = (∀ x. P x ∧ Q)
by (auto)

lemma all-conj-distrib3: (P ∧ (∀ x. Q x)) = (∀ x. P ∧ Q x)
by (auto)
```

lemma *all-disj-distrib2*: $((\forall x.\ P\ x) \vee Q) = (\forall x.\ P\ x \vee Q)$
by (*auto*)

lemma *all-disj-distrib3*: $(P \vee (\forall x.\ Q\ x)) = (\forall x.\ P \vee Q\ x)$
by (*auto*)

lemma *cont-D* : *chain* $Y \Longrightarrow ((\bigsqcup\ i.\ Y\ i) \sqcap S) = (\bigsqcup\ i.\ (Y\ i \sqcap S))$
apply (*subst limproc-is-thelub*, *simp*)
apply (*subst limproc-is-thelub*)
apply (*rule chainI*)
apply (*frule-tac i=i* **in** *chainE*)
apply (*simp add*: *mono-Det mono-Det-sym*)
apply (*subst Process-eq-spec*)
apply (*simp add*: *D-det F-det*)
apply (*subst F-LUB*)
defer
apply (*subst D-LUB*)
defer
apply (*subst F-LUB*)
defer
apply (*subst D-LUB*)
defer
apply (*subst T-LUB*)
defer
apply (*subst D-LUB*)
defer
apply (*subst F-LUB*)
defer
apply (*simp add*: *D-det F-det T-det*)
apply (*rule equalityI*)
apply (*simp add*: *subset-iff*)
apply (*rule allI*)+
apply (*rule impI* | *rule conjI* | *rule allI* | *erule conjE*)+
apply (*erule-tac x=aa* **in** *allE*, *simp-all*)
apply (*rule impI* | *rule conjI* | *rule allI* | *erule conjE* | *erule disjE*)+
apply (*erule-tac x=aa* **in** *allE*, *simp-all*)
apply (*rule impI* | *rule conjI* | *rule allI* | *erule conjE* | *erule disjE*)+
apply (*erule-tac x=aa* **in** *allE*, *simp-all*)
apply (*simp add*: *subset-iff*)
apply (*rule allI*)+
apply (*rule impI* | *rule conjI* | *rule allI* | *erule conjE*)+
apply (*subst all-conj-distrib1* | *subst all-conj-distrib2* |
subst all-conj-distrib3 | *subst all-disj-distrib2* | *subst all-disj-distrib3*)+
apply (*rule allI*)+
apply (*erule-tac x=x + xa + xb + xc* **in** *allE*)
apply (*erule disjE*)
apply (*rule disjI1*, *erule conjE*, *simp*)
apply (*frule-tac i=x* **and** *j=x + xa + xb + xc* **in** *chain-mono*, *simp-all*)
apply (*erule conjE*)

apply (*frule order-lemma*, *simp add*: *subset-iff*)
apply (*erule disjE*)
apply (*rule disjI2*, *rule disjI1*, *erule conjE*, *simp*)
apply (*erule disjE*)
apply (*rule disjI1*)
apply (*frule-tac i=xa* **and** *j=x + xa + xb + xc* **in** *chain-mono*, *simp-all*)
apply (*frule order-lemma*, *simp add*: *subset-iff*)
apply (*erule disjE*)
apply (*rule disjI2*, *rule disjI2*, *rule disjI1*, *erule conjE*, *simp*)
apply (*erule disjE*)
apply (*rule disjI1*)
apply (*frule-tac i=xb* **and** *j=x + xa + xb + xc* **in** *chain-mono*, *simp-all*)
apply (*frule le-approx1*, *simp add*: *subset-iff*)
apply (*erule conjE* | *erule disjE*)+
apply (*rule disjI2*, *rule disjI2*, *rule disjI2*, *rule disjI1*)
apply (*frule-tac i=xc* **and** *j=x + xa + xb + xc* **in** *chain-mono*, *simp-all*)
apply (*frule le-approx-lemma-T*, *simp add*: *subset-iff*)
apply (*rule chainI*)
apply (*frule-tac i=i* **in** *chainE*)
by (*simp add*: *mono-Det*)

lemma *cont-D'* :
assumes *chain*:*chain Y*
shows $((\bigsqcup\ i.\ Y\ i) \sqcup S) = (\bigsqcup\ i.\ (Y\ i \sqcup S))$
proof −
  have *chain'*:*chain* $(\lambda i.\ Y\ i \sqcup S)$
    by(*auto intro*!:*chainI simp*: *mono-Det mono-Det-sym chainE*[*OF chain*])
  have *B*:*F* $(lim\text{-}proc\ (range\ Y) \sqcup S) \subseteq F\ (lim\text{-}proc\ (range\ (\lambda i.\ Y\ i \sqcup S)))$
    apply(*simp add*: *D-det F-det F-LUB T-LUB D-LUB chain chain'*)
    apply(*intro conjI subsetI*, *simp-all*)
    by(*auto split*:*prod.split prod.split-asm*)
  have *C*:*F* $(lim\text{-}proc\ (range\ (\lambda i.\ Y\ i \sqcup S))) \subseteq F(lim\text{-}proc\ (range\ Y) \sqcup S)$
    proof −
    have *C1* : $\bigwedge ba\ ab\ ac.\ \forall a.\ ([], ba) \in F\ (Y\ a) \wedge ([], ba) \in F\ S \vee [] \in D\ (Y\ a) \Longrightarrow$
        $[] \notin D\ (Y\ ab) \Longrightarrow [] \notin D\ S \Longrightarrow\ tick \in ba \Longrightarrow ([], ba) \in F\ (Y\ ac)$
      sorry
    have *C2* : $\bigwedge ba\ ab\ ac\ ad.\ \forall a.\ ([], ba) \in F\ (Y\ a) \wedge ([], ba) \in F\ S \vee [] \in D\ (Y\ a)$
        $\vee\ tick \notin ba \wedge [tick] \in T\ (Y\ a) \Longrightarrow$
      $[] \notin D(Y\ ab) \Longrightarrow [] \notin D\ S \Longrightarrow ([], ba) \notin F(Y\ ac) \Longrightarrow [tick] \notin T\ S \Longrightarrow [tick] \in T(Y\ ad)$
      sorry
    have *C3*: $\bigwedge ba\ ab\ ac.\ \forall a.\ [] \in D\ (Y\ a) \vee tick \notin ba \wedge [tick] \in T\ (Y\ a) \Longrightarrow$
        $[] \notin D\ (Y\ ab) \Longrightarrow [] \notin D\ S \Longrightarrow ([], ba) \notin F\ S \Longrightarrow$
        $[tick] \notin T\ S \Longrightarrow [tick] \in T\ (Y\ ac)$
      sorry
    show *?thesis*
      apply(*simp add*: *D-det F-det F-LUB T-LUB D-LUB chain chain'*)
      apply(*rule subsetI*, *simp*)
      apply(*simp split*:*prod.split prod.split-asm*)

```
        apply(intro allI impI,simp)
        apply(case-tac a=[], auto)
        apply(metis C1, metis C2, metis C3)
        done
    qed
  have D:D (lim-proc (range Y) □ S) = D (lim-proc (range (λi. Y i □ S)))
        by(simp add: D-det F-det F-LUB T-LUB D-LUB chain chain')
  show ?thesis
  by(simp add: chain chain' limproc-is-thelub Process-eq-spec equalityI B C D)
qed

lemma det-cont:
assumes f:cont f
and     g:cont g
shows     cont (λx. f x □ g x)
proof −
  have A : ⋀x. cont f ⟹ cont (λy. y □ f x)
     apply (rule contI2,rule monofunI)
     apply (auto simp add: mono-Det Det-commute)
     apply (subst Det-commute,subst cont-D)
     apply (auto simp: Det-commute)
     done
  have B : ⋀y. cont f ⟹ cont (λx. y □ f x)
     apply (rule-tac c=(λ x. y □ x) in cont-compose)
     apply (rule contI2,rule monofunI)
     apply (auto simp add: mono-Det Det-commute mono-Det-sym)
     apply (subst Det-commute,subst cont-D)
     by (simp-all add: Det-commute)
  show ?thesis using f g
    apply(rule-tac f=(λx. (λ g. f x □ g)) in cont-apply)
    apply(auto intro:contI2 monofunI simp:mono-Det Det-commute mono-Det-sym A B)
    done
qed


end
```

7 Nondeterministic Choice Operator Definition

```
theory  Ndet
imports Process Cont
begin

definition
      ndet    :: ['α process,'α process] ⇒ 'α process   (infixl |−| 16)
where   P |−| Q ≡ Abs-Process(F P ∪ F Q , D P ∪ D Q)

notation(xsymbols)
  ndet (infixl ⊓ 16)
```

lemma *is-process-REP-ND*:
is-process $(F\ P \cup F\ Q\ ,\ D\ P \cup D\ Q)$
proof(*simp only*: *fst-conv snd-conv Process-def is-process-def DIVERGENCES-def FAILURES-def*,
intro conjI)
fix $P\ Q :: 'a\ process$
{
show $([], \{\}) \in (F\ P \cup F\ Q)$
by(*simp add*: *is-processT1*)
next
show $\forall s\ X.\ (s,\ X) \in (F\ P \cup F\ Q) \longrightarrow$ *front-tickFree s*
by(*auto simp*: *is-processT2*)
next
show $\forall s\ t.\quad (s\ @\ t, \{\}) \in (F\ P \cup F\ Q) \longrightarrow (s, \{\}) \in (F\ P \cup F\ Q)$
by (*auto simp*: *is-processT1 dest*!: *is-processT3*[*rule-format*])
next
show $\forall s\ X\ Y.\ (s,\ Y) \in (F\ P \cup F\ Q) \wedge X \subseteq Y \longrightarrow (s,\ X) \in (F\ P \cup F\ Q)$
by(*auto dest*: *is-processT4*[*rule-format*,*OF conj-commute*[*THEN iffD1*], *OF conjI*])
next
show $\forall sa\ X\ Y.\ (sa,\ X) \in (F\ P \cup F\ Q) \wedge (\forall c.\ c \in Y \longrightarrow (sa\ @\ [c], \{\}) \notin (F\ P \cup F\ Q))$
$\longrightarrow (sa,\ X \cup Y) \in (F\ P \cup F\ Q)$
by(*auto simp*: *is-processT5 T-F*)
next
show $\forall s\ X.\ (s\ @\ [tick], \{\}) \in (F\ P \cup F\ Q) \longrightarrow (s,\ X - \{tick\}) \in (F\ P \cup F\ Q)$
apply((*rule allI*)+, *rule impI*)
apply(*case-tac s*=[], *simp-all add*: *is-processT6*[*rule-format*] *T-F-spec*)
apply(*erule disjE*,*simp-all add*: *is-processT6*[*rule-format*] *T-F-spec*)
apply(*erule disjE*,*simp-all add*: *is-processT6*[*rule-format*] *T-F-spec*)
done
next
show $\forall s\ t.\ s \in D\ P \cup D\ Q \wedge$ *tickFree s* $\wedge$ *front-tickFree t* $\longrightarrow s\ @\ t \in D\ P \cup D\ Q$
by(*auto simp*: *is-processT7*)
next
show $\forall s\ X.\ s \in D\ P \cup D\ Q \longrightarrow (s,\ X) \in (F\ P \cup F\ Q)$
by(*auto simp*: *is-processT8*[*rule-format*])
next
show $\forall s.\ s\ @\ [tick] \in D\ P \cup D\ Q \longrightarrow s \in D\ P \cup D\ Q$
by(*auto intro*!:*is-processT9*[*rule-format*])
}
qed

lemma *Rep-Abs-ND*:
Rep-Process(*Abs-Process*$(F\ P \cup F\ Q\ ,\ D\ P \cup D\ Q)) = (F\ P \cup F\ Q\ ,\ D\ P \cup D\ Q)$
by(*simp only*:*CollectI Process-def Abs-Process-inverse is-process-REP-ND*)

lemma *F-ndet* : $F(P \sqcap Q) = F\ P \cup F\ Q$

by(*subst F-def*, *simp only*: *ndet-def Rep-Abs-ND FAILURES-def fst-conv*)

lemma *D-ndet* : $D(P \sqcap Q) = D\ P \cup D\ Q$
by(*subst D-def*, *simp only*: *ndet-def Rep-Abs-ND DIVERGENCES-def snd-conv*)

lemma *T-ndet* : $T(P \sqcap Q) = T\ P \cup T\ Q$
apply(*subst T-def*, *simp only*: *ndet-def Rep-Abs-ND TRACES-def FAILURES-def*
fst-def snd-conv)
apply(*auto simp*: *T-F F-T is-processT1 Nil-elem-T*)
apply(*rule-tac x*=$\{\}$ **in** *exI*, *simp add*: *T-F F-T is-processT1 Nil-elem-T*)+
done

lemma *Ndet-commute*: $(P \sqcap Q) = (Q \sqcap P)$
by(*auto simp*: *Process-eq-spec F-ndet D-ndet*)

lemma *mono-Ndet1*: $P \sqsubseteq Q \Longrightarrow D\ (Q \sqcap S) \subseteq D\ (P \sqcap S)$
apply(*drule le-approx1*)
by (*auto simp*: *Process-eq-spec F-ndet D-ndet*)

lemma *mono-Ndet2*: $P \sqsubseteq Q \Longrightarrow (\forall\ s.\ s \notin D\ (P \sqcap S) \longrightarrow Ra\ (P \sqcap S)\ s = Ra\ (Q \sqcap S)\ s)$
apply(*subst* (*asm*) *le-approx-def*)
by (*auto simp*: *Process-eq-spec F-ndet D-ndet Ra-def*)

lemma *mono-Ndet3*: $P \sqsubseteq Q \Longrightarrow min\text{-}elems\ (D\ (P \sqcap S)) \subseteq T\ (Q \sqcap S)$
apply(*auto dest*!:*le-approx3 simp*: *min-elems-def subset-iff F-ndet D-ndet T-ndet*)
apply(*erule-tac x*=*t* **in** *allE*, *auto*)
by (*erule-tac x*=[] **in** *allE*, *auto simp*: *less-list-def Nil-elem-T D-T*)

lemma *mono-Ndet* : $P \sqsubseteq Q \Longrightarrow (P \sqcap S) \sqsubseteq (Q \sqcap S)$
by(*auto simp*:*le-approx-def mono-Ndet1 mono-Ndet2 mono-Ndet3*)

lemma *mono-Ndet-sym* : $P \sqsubseteq Q \Longrightarrow (S \sqcap P) \sqsubseteq (S \sqcap Q)$
by (*auto simp*: *mono-Ndet Ndet-commute*)

lemma *cont-Ndet1*:
assumes *chain*:*chain Y*
shows $((\bigsqcup\ i.\ Y\ i) \sqcap S) = (\bigsqcup\ i.\ (Y\ i \sqcap S))$
proof −
have *A* : *chain* $(\lambda i.\ Y\ i \sqcap S)$
apply(*insert chain*,*rule chainI*)
apply(*frule-tac i*=*i* **in** *chainE*)
by(*simp add*: *mono-Ndet mono-Ndet-sym*)
show *?thesis* **using** *chain*
by(*auto simp add*: *limproc-is-thelub Process-eq-spec D-ndet F-ndet F-LUB D-LUB A*)
qed

lemma *ndet-cont*:
assumes *f*: *cont f*
and *g*: *cont g*
shows *cont* (λ*x*. *f x* ⊓ *g x*)
proof −
have *A*:⋀*x*. *cont f* ⟹ *cont g* ⟹ *cont* (*op* ⊓ (*f x*))
apply(*rule contI2*, *rule monofunI*)
apply(*auto simp add*: *mono-Ndet mono-Ndet-sym*)
apply(*subst Ndet-commute*, *subst cont-Ndet1*)
by (*auto simp*:*Ndet-commute*)
have *B*:⋀*y*. *cont f* ⟹ *cont g* ⟹ *cont* (λ*x*. *f x* ⊓ *y*)
apply(*rule-tac c*=(λ *g*. *g* ⊓ *y*) **in** *cont-compose*)
apply(*rule contI2*,*rule monofunI*)
by (*simp-all add*: *cont-Ndet1 mono-Ndet*)
show *?thesis* **using** *f g*
by (*rule-tac f*=(λ *x*. (λ *g*. *f x* ⊓ *g*)) **in** *cont-apply*, *auto simp*: *A B*)
qed

end

8 The Sequence Operator

theory *Seq*
imports *Process*

begin

definition *seq* :: ['*a process*,'*a process*] => '*a process* (**infixl** ';' *24*)
where *P* ';' *Q* ≡ *Abs-Process*
({(*t*, *X*). (*t*, *X* ∪ {*tick*}) ∈ *F P* ∧ *tickFree t*} ∪
{(*t*, *X*). ∃*t1 t2*. *t* = *t1* @ *t2* ∧ *t1* @ [*tick*] ∈ *T P* ∧ (*t2*, *X*) ∈ *F Q*} ∪
{(*t*, *X*). ∃*t1 t2*. *t* = *t1* @ *t2* ∧ *t1* ∈ *D P* ∧ *tickFree t1* ∧ *front-tickFree t2*} ∪
{(*t*, *X*). ∃*t1 t2*. *t* = *t1* @ *t2* ∧ *t1* @ [*tick*] ∈ *T P* ∧ *t2* ∈ *D Q*},
{*t1* @ *t2* |*t1 t2*. *t1* ∈ *D P* ∧ *tickFree t1* ∧ *front-tickFree t2*} ∪
{*t1* @ *t2* |*t1 t2*. *t1* @ [*tick*] ∈ *T P* ∧ *t2* ∈ *D Q*})

axioms
F-seq : *F*(*P* ';' *Q*) = {(*t*, *X*). (*t*, *X* ∪ {*tick*}) ∈ *F P* ∧ *tickFree t*} ∪
{(*t*, *X*). ∃*t1 t2*. *t* = *t1* @ *t2* ∧ *t1* @ [*tick*] ∈ *T P* ∧ (*t2*, *X*) ∈ *F Q*} ∪
{(*t*, *X*). ∃*t1 t2*. *t* = *t1* @ *t2* ∧ *t1* ∈ *D P* ∧ *tickFree t1* ∧ *front-tickFree t2*} ∪
{(*t*, *X*). ∃*t1 t2*. *t* = *t1* @ *t2* ∧ *t1* @ [*tick*] ∈ *T P* ∧ *t2* ∈ *D Q*}

D-seq : *D*(*P* ';' *Q*) = {*t1* @ *t2* |*t1 t2*. *t1* ∈ *D P* ∧ *tickFree t1* ∧ *front-tickFree t2*} ∪

$$\{t1\ @\ t2\ |t1\ t2.\ t1\ @\ [tick] \in T\ P \wedge t2 \in D\ Q\}$$

T-seq : $T(P\ ;\ Q) = \{t.\ \exists\ X.\ (t,\ X \cup \{tick\}) \in F\ P \wedge tickFree\ t\} \cup$ (∗ *REALLY ???* ∗)

$$\{t.\ \exists\ t1\ t2.\ t = t1\ @\ t2 \wedge t1\ @\ [tick] \in T\ P \wedge t2 \in T\ Q\} \cup$$
$$\{t1\ @\ t2\ |t1\ t2.\ t1 \in D\ P \wedge tickFree\ t1 \wedge front\text{-}tickFree\ t2\} \cup$$
$$\{t1\ @\ t2\ |t1\ t2.\ t1\ @\ [tick] \in T\ P \wedge t2 \in D\ Q\}$$

seq-cont[*simp*]: $[\![cont\ f;\ cont\ g]\!] \Longrightarrow cont\ (\lambda\ x.\ f\ x\ ;\ g\ x)$

lemma *is-processT1-SEQ*: ([],{}) : {(t,X). (t, X Un {tick}) : F P & tickFree t} Un {(t,X). ? t1 t2. t = t1 @ t2 & t1 @ [tick] : T P & (t2, X) : F Q} Un {(t,X). ? t1 t2. t = t1@t2 & t1 : D P & tickFree t1 & front-tickFree t2} Un {(t,X). ? t1 t2. t = t1@t2 & t1@[tick] : T P & t2 : D Q}
apply (*simp*)
apply (*subst disj-commute*)
apply (*rule disjCI*)
apply (*rule disjI1*)
apply (*rule conjI*)
apply (*rule-tac X= {} in F-T*)
apply (*simp-all add*: *is-processT1 is-processT5-S6*)
done

lemma *is-processT2-SEQ*: ! s X. (s, X) :{(t,X). (t, X Un {tick}) : F P & tickFree t} Un {(t,X). ? t1 t2. t = t1 @ t2 & t1 @ [tick] : T P & (t2, X) : F Q} Un {(t,X). ? t1 t2. t = t1@t2 & t1 : D P & tickFree t1 & front-tickFree t2} Un {(t,X). ? t1 t2. t = t1@t2 & t1@[tick] : T P & t2 : D Q} −−> *front-tickFree s*
apply (*auto*)
apply (*rule-tac X=insert tick X* **and** *P=P* **in** *is-processT2*[*rule-format*])
apply (*simp*)
apply (*subst front-tickFree-append*)
apply (*rule-tac s=*[*tick*] **in** *append-T-imp-tickFree*)
apply (*simp-all add*: *is-processT2*[*rule-format*] *front-tickFree-append*)
apply (*subst front-tickFree-append*)
apply (*rule-tac s=*[*tick*] **in** *append-T-imp-tickFree*)
apply (*simp-all add*: *front-tickFree-append*)
apply (*rule-tac P=Q* **in** *is-processT2*[*rule-format*])
apply (*simp add*: *is-processT8*[*rule-format*])
done

lemma *F-D-SEQ-spec*: $F\ (P\ ;\ Q) =$
$\{(t,\ X).\ (t,\ X \cup \{tick\}) \in F\ P \wedge tickFree\ t\} \cup$
$\{(t,\ X).\ \exists\ t1\ t2.\ t = t1\ @\ t2 \wedge t1\ @\ [tick] \in T\ P \wedge (t2,\ X) \in F\ Q\} \cup$
$\{(t,\ x).\ t \in D\ (P\ ;\ Q)\}$
by (*auto simp*: *F-seq Un-def D-seq*)

lemma *F-SEQ-spec*: $F\ (P\ ;\ Q) =$
$\{(t,\ X).\ (t,\ X \cup \{tick\}) \in F\ P \wedge tickFree\ t\} \cup$
$\{(t,\ X).\ \exists\ t1\ t2.\ t = t1\ @\ t2 \wedge t1\ @\ [tick] \in T\ P \wedge (t2,\ X) \in F\ Q\} \cup$
$\{(t,\ x).\ \exists\ t1\ t2.\ t = t1\ @\ t2 \wedge t1 \in D\ P \wedge tickFree\ t1 \wedge front\text{-}tickFree\ t2\}$
apply (*simp-all add*: *Un-def F-seq*)
apply (*rule equalityI*)
apply (*rule subsetI*)
apply (*simp*)
apply (*erule disjE*)
apply (*rule disjI1*)
defer
apply (*erule disjE*)
apply (*rule disjI2*)
apply (*rule disjI1*)
defer
apply (*erule disjE*)
apply (*rule disjI2*)
apply (*rule disjI2*)
defer
apply (*rule disjI2*)
apply (*rule disjI1*)
defer
apply (*rule subsetI*)
apply (*simp*)
apply (*erule disjE*)
apply (*rule disjI1*)
defer
apply (*erule disjE*)
apply (*rule disjI2*)
apply (*rule disjI1*)
defer
apply (*rule disjI2*)
apply (*rule disjI2*)
apply (*rule disjI1*)
apply (*auto*)
apply (*rule-tac x=t1* **in** *exI*, *simp*)
apply (*simp add*: *is-processT8-S*)
done

end

9 The Hiding Operator

theory *Hide*
imports *Process*
begin

primrec *trace-hide* :: [$'\alpha$ *trace*,($'\alpha$ *event*) *set*] => $'\alpha$ *trace* **where**
trace-hide [] *A* = []
| *trace-hide* (*x* # *s*) *A* = (*if x* ∈ *A*
then trace-hide s A
else x # (*trace-hide s A*))

definition *IsChainOver* :: [*nat* => $'\alpha$ *list*, $'\alpha$ *list*] => *bool*
(**infixl** *IsChainOver 70*) **where**
f IsChainOver t = (*f 0* = *t* ∧ (∀ *i*. *f i* < *f* (*Suc i*)))

definition *CongruentModuloHide* :: [*nat* => $'\alpha$ *trace*,$'\alpha$ *trace* , $'\alpha$ *set*] => *bool*
(- *Congruent* - *ModuloHide* - *70*) **where**
f Congruent t ModuloHide A ≡
∀ *i*. *trace-hide* (*f i*) (*ev* ‘ *A*) = *trace-hide t* (*ev*‘*A*)

definition
hiding :: [$'\alpha$ *process* ,$'\alpha$ *set*] => $'\alpha$ *process* (- \ - [*73*,*72*] *72*) **where**
P \ *A* ≡ *Abs-Process*({(*s*,*X*). ∃ *t*. *s* = *trace-hide t* (*ev*‘*A*) ∧ (*t*,*X* ∪ (*ev*‘*A*)) ∈ *F P*} ∪
{(*s*,*X*). ∃ *t u*. *front-tickFree u* ∧ *tickFree t* ∧
s = *trace-hide t* (*ev*‘*A*) @ *u* ∧
(*t* ∈ *D P* ∨ (∃ *f*. (*f IsChainOver t*) ∧
(*f Congruent t ModuloHide A*) ∧
(∀ *i*. *f i* ∈ *T P*)))},
{*s*. ∃ *t u*. *front-tickFree u* ∧
tickFree t ∧ *s* = *trace-hide t* (*ev*‘*A*) @ *u* ∧
(*t* ∈ *D P* ∨ (∃ *f*. (*f IsChainOver t*) ∧
(*f Congruent t ModuloHide A*) ∧
(∀ *i*. *f i* ∈ *T P*)))})

axioms
F-hiding : *F*(*P* \ *A*) = {(*s*,*X*). ∃ *t*. *s* = *trace-hide t* (*ev*‘*A*) ∧ (*t*,*X* ∪ (*ev*‘*A*)) ∈ *F P*} ∪
{(*s*,*X*). ∃ *t u*. *front-tickFree u* ∧ *tickFree t* ∧
s = *trace-hide t* (*ev*‘*A*) @ *u* ∧
(*t* ∈ *D P* ∨ (∃ *f*. (*f IsChainOver t*) ∧
(*f Congruent t ModuloHide A*) ∧
(∀ *i*. *f i* ∈ *T P*))) }

D-hiding : *D*(*P* \ *A*) = {*s*. ∃ *t u*. *front-tickFree u* ∧ *tickFree t* ∧
s = *trace-hide t* (*ev*‘*A*) @ *u* ∧

$$(t \in D\ P \vee (\exists\ f.\ (f\ \mathit{IsChainOver}\ t) \wedge (f\ \mathit{Congruent}\ t\ \mathit{ModuloHide}\ A) \wedge (\forall\ i.\ f\ i \in T\ P)))\}$$

T-hiding : $T(P \setminus A) = \{s.\quad \exists\ t.\ s = \mathit{trace\text{-}hide}\ t\ (ev`A) \wedge t \in T\ P\}$

hiding-cont [*simp*]: $[\![\mathit{cont}\ f; \mathit{finite}\ A]\!] \Longrightarrow \mathit{cont}\ (\lambda x.\ f\ x \setminus A)$

lemmas *tr-hide-set-def* = *trace-hide-def*
lemmas *Hide-set-def* = *hiding-def*
lemmas *F-hide-set* = *F-hiding*
lemmas *D-hide-set* = *D-hiding*
lemmas *T-hide-set* = *T-hiding*
lemmas *hide-set-cont* = *hiding-cont*

end

theory *Sync*
imports *Process*
begin

fun *setinterleaving*::$'a\ \mathit{trace} \times ('a\ \mathit{event})\ \mathit{set} \times 'a\ \mathit{trace} \Rightarrow ('a\ \mathit{trace})\mathit{set}$
where

si-empty1: $\mathit{setinterleaving}([], X, []) = \{[]\}$
| *si-empty2*: $\mathit{setinterleaving}([], X, (y \# t)) =$
$(\mathit{if}\ (y \in X)$
$\mathit{then}\ \{\}$
$\mathit{else}\ \{z.\exists\ u.\ z = (y \# u) \wedge u \in \mathit{setinterleaving}\ ([], X, t)\})$
| *si-empty3*: $\mathit{setinterleaving}((x \# s), X, []) =$
$(\mathit{if}\ (x \in X)$
$\mathit{then}\ \{\}$
$\mathit{else}\ \{z.\exists\ u.\ z = (x \# u) \wedge u \in \mathit{setinterleaving}\ (s, X, [])\})$
| *si-neq* : $\mathit{setinterleaving}((x \# s), X, (y \# t)) =$
$(\mathit{if}\ (x \in X)$

then if $(y \in X)$
then if $(x = y)$
then $\{z.\exists\ u.\ z = (x\#u) \wedge u \in setinterleaving(s,\ X,\ t)\}$
else $\{\}$
else $\{z.\exists\ u.\ z = (y\#u) \wedge u \in setinterleaving\ ((x\#s),\ X,\ t)\}$
else if $(y \notin X)$
then $\{z.\exists\ u.\ z = (x\ \#\ u) \wedge u \in setinterleaving\ (s,\ X,\ (y\ \#\ t))\}$
$\cup\ \{z.\exists\ u.\ z = (y\ \#\ u) \wedge u \in setinterleaving((x\ \#\ s),\ X,\ t)\}$
else $\{z.\exists\ u.\ z = (x\ \#\ u) \wedge u \in setinterleaving\ (s,\ X,\ (y\ \#\ t))\})$

lemma *sym1* [*simp*]: $setinterleaving([],\ X,\ t) = setinterleaving(t,\ X\ ,[])$
by (*induct t*, *simp-all*)

lemma *sym2* [*simp*]:
$\forall\ s.\ setinterleaving\ (s,\ X,\ t) = setinterleaving\ (t,\ X,\ s)$
$\longrightarrow setinterleaving\ (a\ \#\ s,\ X,\ t) = setinterleaving\ (t,\ X,\ a\ \#\ s)$
apply (*induct t*)
apply (*simp-all*)
apply *auto*
apply (*case-tac t*,*simp*)
sorry

lemma *sym* [*simp*] : $setinterleaving(s,\ X,\ t) = setinterleaving(t,\ X,\ s)$
by (*induct s*, *simp-all*)

abbreviation *setinterleaves-syntax*
(- *setinterleaves* ′(()′(-, -′)(), -′) [60,0,0,0]70)
where
$u\ setinterleaves\ ((s,\ t),\ X) == (u \in setinterleaving(s,\ X,\ t))$

definition *sync* :: [′*a process*,′*a set*,′*a process*] => ′*a process*
((3- ⟦-⟧/ -) [14,0,15] 14)
where
$P \llbracket A \rrbracket Q ==$
Abs-Process($\{(s,R).\exists\ t\ u\ X\ Y.\ (t,X) \in F\ P \wedge (u,Y) \in F\ Q \wedge$
$(s\ setinterleaves\ ((t,u),(ev`A) \cup \{tick\})) \wedge$
$R = (X \cup Y) \cap ((ev`A) \cup \{tick\}) \cup X \cap Y\} \cup$
$\{(s,R).\exists\ t\ u\ r\ v.\ front\text{-}tickFree\ v \wedge (tickFree\ r \vee v=[]) \wedge$
$s = r@v \wedge$
$(r\ setinterleaves\ ((t,u),(ev`A) \cup \{tick\})) \wedge$
$(t \in D\ P \wedge u \in T\ Q \vee t \in D\ Q \wedge u \in T\ P)\},$
$\{s.\quad \exists\ t\ u\ r\ v.\ front\text{-}tickFree\ v \wedge (tickFree\ r \vee v=[]) \wedge$
$s = r@v \wedge$

$(r$ *setinterleaves* $((t,u),(ev`A) \cup \{tick\})) \wedge$
$(t \in D\ P \wedge u \in T\ Q \vee t \in D\ Q \wedge u \in T\ P)\})$

axioms
F-sync : $F(P \llbracket A \rrbracket Q) =$
$\{(s,R).\exists\ t\ u\ X\ Y.\ (t,X) \in F\ P \wedge$
$(u,Y) \in F\ Q \wedge$
s *setinterleaves* $((t,u),(ev`A) \cup \{tick\}) \wedge$
$R = (X \cup Y) \cap ((ev`A) \cup \{tick\}) \cup X \cap Y\} \cup$
$\{(s,R).\exists\ t\ u\ r\ v.$ *front-tickFree* $v \wedge$
$($*tickFree* $r \vee v = []) \wedge$
$s = r @ v \wedge$
r *setinterleaves* $((t,u),(ev`A) \cup \{tick\}) \wedge$
$(t \in D\ P \wedge u \in T\ Q \vee t \in D\ Q \wedge u \in T\ P)\}$

D-sync : $D(P \llbracket A \rrbracket Q) =$
$\{s.\quad \exists\ t\ u\ r\ v.$ *front-tickFree* $v \wedge (tickFree\ r \vee v = []) \wedge$
$s = r @ v \wedge r$ *setinterleaves* $((t,u),(ev`A) \cup \{tick\}) \wedge$
$(t \in D\ P \wedge u \in T\ Q \vee t \in D\ Q \wedge u \in T\ P)\}$

T-sync : $T(P \llbracket A \rrbracket Q) =$
$\{s.\quad \forall\ t\ u.\ t \in T\ P \wedge u \in T\ Q \wedge$
s *setinterleaves* $((t,u),(ev`A) \cup \{tick\})\}$

abbreviation *Inter-syntax* ((-|||-) [*14*,*15*] *14*)
where $P\ |||\ Q == (P \llbracket \{\} \rrbracket Q)$

abbreviation *Par-syntax* ((-||-) [*14*,*15*] *14*)
where $P\ ||\ Q == (P \llbracket UNIV \rrbracket Q)$

lemma *sync-cont*[*simp*]:
$\llbracket$ *cont* f; *cont* $g \rrbracket \Longrightarrow$ *cont* $(\%x.\ (f\ x) \llbracket A \rrbracket (g\ x))$
sorry

end

10 Toplevel Theory

theory *CSP*
imports *Bot Skip Stop Mprefix Det Ndet Seq Hide Sync Legacy*
begin

10.1 Refinement Proof Rules

10.2 The "Laws" of CSP

axioms

$$\begin{array}{ll}
\textit{hide-mprefix-distr} & : [\![(B \cap A) = \{\}]\!] \Longrightarrow \\
& ((\textit{Mprefix } A\ P) \setminus B) = (\textit{Mprefix } A\ (\%\ x.\ ((P\ x) \setminus B))) \\
\textit{hide-prefix-distr1} & : a : B \implies ((a \rightarrow P) \setminus B) = (P \setminus B) \\
\textit{hide-prefix-distr2} & : a\ {}^{\sim}\!: B \Longrightarrow ((a \rightarrow P) \setminus B) = (a \rightarrow (P \setminus B)) \\
\textit{hide-det} & : ((a \rightarrow P) \Box (b \rightarrow Q)) \setminus \{a\} = \\
& ((P \setminus \{a\}) \sqcap ((P \setminus \{a\}) \Box (b \rightarrow (Q \setminus \{a\}))))
\end{array}$$

lemma *mprefix-singl*: $(\textit{Mprefix } \{a\}\ P) = (a \rightarrow (P\ a))$
by (*auto simp*: *write0-def Process-eq-spec F-Mprefix D-Mprefix*)

lemma *mono-mprefix-ref*: $\forall x.\ P\ x \sqsubseteq Q\ x \Longrightarrow \textit{Mprefix } A\ P \sqsubseteq \textit{Mprefix } A\ Q$
by(*simp add*: *mono-Mprefix0*)

lemma *mono-prefix-ref*: $P \sqsubseteq Q \Longrightarrow (a \rightarrow P) \sqsubseteq (a \rightarrow Q)$
by (*simp add*: *write0-def mono-mprefix-ref*)

lemma *mono-ndet-ref*: $[\![P \sqsubseteq P';\ S \sqsubseteq S']\!] \Longrightarrow (P \sqcap S) \sqsubseteq (P' \sqcap S')$
by(*auto simp*: *le-approx-def F-ndet D-ndet Ra-def F-ndet min-elems-def T-ndet*)

lemma *mono-det-ref*:
$[\![P \sqsubseteq P';\ S \sqsubseteq S']\!] \Longrightarrow (P \Box S) \sqsubseteq (P' \Box S')$
apply (*simp add*: *le-approx-def*)
apply (*simp add*: *F-det D-det*)
apply (*rule conjI*)
apply (*rule subsetI*)
apply (*rule UnI1*)
apply (*erule conjE*)
apply (*drule conjunct1*)
apply (*drule conjunct1*)
apply (*simp add*: *subsetD*)
apply (*rule conjI*)
apply (*rule subsetI*)
apply (*rule UnI2*)
apply (*erule conjE*)
apply (*drule conjunct1*)
apply (*drule conjunct1*)
apply (*simp add*: *subsetD*)
apply (*rule conjI*)
apply (*rule allI*)

apply (*rule impI*)
apply (*simp add*: *Ra-def*)
apply (*simp add*: *F-det*)
apply (*drule conjunct2*)
apply (*drule conjunct2*)
apply (*drule conjunct1*)
back
apply (*drule conjunct1*)
back
apply (*rule equalityI*)
apply (*rule subsetI*)
apply (*simp*)
apply (*rule disjI1*)
apply (*rule conjI*)
prefer *3*
apply (*rule subsetI*)
apply (*simp*)
apply (*rule disjI1*)
apply (*rule conjI*)
prefer *5*
apply (*simp add*: *min-elems-def T-det*)
sorry

lemma *mono-hide-set-refD*:
$P \sqsubseteq Q \Longrightarrow D\ (Q \setminus A) \subseteq D\ (P \setminus A)$
apply (*simp add*: *le-approx-def*)
apply (*simp add*: *D-hiding*)
apply (*simp add*: *Ra-def min-elems-def*)
apply (*auto*)
apply (*rule-tac x*=*t* **in** *exI*)
apply (*rule-tac x*=*u* **in** *exI*)
apply (*rule conjI*)
apply (*simp*)
apply (*rule conjI*)
apply (*simp*)
apply (*rule conjI*)
apply (*simp*)
apply (*rule disjI2*)
apply (*rule-tac x*=*f* **in** *exI*)
apply (*rule conjI*)
apply (*simp*)
apply (*rule conjI*)
apply (*simp*)
apply (*rule allI*)
apply (*simp add*: *T-F-spec*[*symmetric*])
sorry

lemma *mono-hide-set-refF*: $P \sqsubseteq Q \Longrightarrow F\ (Q \setminus A) \subseteq F\ (P \setminus A)$
apply (*simp add*: *le-approx-def*)
apply (*simp add*: *F-hiding*)
apply (*simp add*: *Ra-def min-elems-def*)
apply (*auto*)
apply (*rule-tac x*=*t* **in** *exI*)
apply (*rule conjI*)
apply (*simp*)
sorry

lemma *mono-hide-set-ref*: $P \sqsubseteq Q \Longrightarrow P \setminus A \sqsubseteq Q \setminus A$
apply (*subst le-approx-def*)
apply (*simp add*: *Ra-def min-elems-def*)
apply (*simp add*: *mono-hide-set-refD*)
apply (*rule conjI*)
apply (*rule allI*)
apply (*rule impI*)
apply (*rule equalityI*)
apply (*auto*)
sorry

lemma *mono-HSI2a*: $[\![P \leq Q;\ s \notin D\ (P \setminus A)]\!] \Longrightarrow Ra\ (P \setminus A)\ s \subseteq Ra\ (Q \setminus A)\ s$
sorry

lemma *mono-HSI2b*: $[\![P \leq Q;\ s \notin D\ (P \setminus A)]\!] \Longrightarrow Ra\ (Q \setminus A)\ s \subseteq Ra\ (P \setminus A)\ s$
sorry

lemma *mono-HSI2*: $[\![P \leq Q;\ s \notin D\ (P \setminus A)]\!] \Longrightarrow Ra\ (P \setminus A)\ s = Ra\ (Q \setminus A)\ s$
sorry

lemma *mono-HSI31*:
$[\![tr\text{-}hide\text{-}set\ t\ (ev\ `\ A) = tr\text{-}hide\text{-}set\ s\ (ev\ `\ A);\ s \in T\ Q;$
$\forall s.\ tr\text{-}hide\text{-}set\ t\ (ev\ `\ A) = tr\text{-}hide\text{-}set\ s\ (ev\ `\ A) \longrightarrow$
$(s,\ ev\ `\ A) \notin F\ Q]\!]$
$\Longrightarrow \exists a.\ a \in ev\ `\ A \wedge s\ @\ [a] \in T\ Q$
sorry

lemma *help1*: [*a*,*b*]=[*a*]@[*b*]
by(*simp*)

lemma *help2*: [*a*]<[*a*,*b*]
by(*simp add*: *less-list-def le-list-def*)

lemma *mono-HSI32*:
$[\![tickFree\ t;\ t \in T\ Q;$
$\forall s.\ tr\text{-}hide\text{-}set\ t\ (ev\ `\ A) = tr\text{-}hide\text{-}set\ s\ (ev\ `\ A) \longrightarrow$
$(s,\ ev\ `\ A) \notin F\ Q]\!]$
$\Longrightarrow \exists f.\ f\ IsChainOver\ t \wedge f\ Congruent\ t\ ModuloHide\ A \wedge (\forall i.\ f\ i \in T\ Q)$
sorry

lemma *mono-HSI33*:
$\exists\, n\ s.\ length\ s = n \wedge s \leq t \wedge tr\text{-}hide\text{-}set\ s\ A = tr\text{-}hide\text{-}set\ t\ A$
by (*auto*)

lemma *mono-HSI34*:
$\exists\, s.\ tr\text{-}hide\text{-}set\ s\ A = tr\text{-}hide\text{-}set\ t\ A \wedge$
$s \leq t \wedge (\forall\, s1{<}s.\ tr\text{-}hide\text{-}set\ s1\ A \neq tr\text{-}hide\text{-}set\ t\ A)$
sorry

lemma *mono-HSI35*:
$[\![s < t;\ tr\text{-}hide\text{-}set\ s\ A \neq tr\text{-}hide\text{-}set\ t\ A]\!]$
$\Longrightarrow tr\text{-}hide\text{-}set\ s\ A < tr\text{-}hide\text{-}set\ t\ A$
sorry

lemma *mono-HSI36*:
$\forall\, ta.\ (\exists\, t\ u.\ front\text{-}tickFree\ u \wedge$
$tickFree\ t \wedge$
$ta = tr\text{-}hide\text{-}set\ t\ (ev\ `\ A)\ @\ u \wedge$
$(t \in D\ P \vee$
$(\exists f.\ f\ IsChainOver\ t \wedge$
$f\ Congruent\ t\ ModuloHide\ A \wedge (\forall\, i.\ f\ i \in T\ P)))) \longrightarrow$
$\neg\ ta < tr\text{-}hide\text{-}set\ t\ (ev\ `\ A) \Longrightarrow$
$\exists\, t1.\ tr\text{-}hide\text{-}set\ t1\ (ev\ `\ A) = tr\text{-}hide\text{-}set\ t\ (ev\ `\ A) \wedge$
$t1 \leq t \wedge (t1 \notin D\ P \vee t1 \in min\text{-}elems\ (D\ P))$
sorry

lemma *mono-HSI3*:
$P \leq Q \Longrightarrow min\text{-}elems\ (D\ (P \setminus A)) \subseteq T\ (Q \setminus A)$
sorry

lemma *mono-HSI*: $P \leq Q \Longrightarrow P \setminus A \leq Q \setminus A$
sorry

lemma *mono-HS-rec*: $mono\ (\lambda\ P.\ (P \setminus A))$
sorry

lemma *mono-PaI-refD*:
$P \sqsubseteq Q \Longrightarrow D\ (Q\ [\![A]\!]\ S) \subseteq D\ (P\ [\![A]\!]\ S)$
sorry

lemma *mono-PaI-refF*: $P \sqsubseteq Q \Longrightarrow F\ (Q\ [\![A]\!]\ S) \subseteq F\ (P\ [\![A]\!]\ S)$
sorry

lemma *mono-PaI-ref-L*: $P \sqsubseteq Q \Longrightarrow (P\ [\![A]\!]\ S) \sqsubseteq (Q\ [\![A]\!]\ S)$
sorry

lemma *mono-PaI-ref-R*: $P \sqsubseteq Q \Longrightarrow (S \ [\![A]\!] \ P) \sqsubseteq (S \ [\![A]\!] \ Q)$
sorry

lemma *mono-PaI-ref*: $[\![P \sqsubseteq P'; \ Q \sqsubseteq Q']\!] \Longrightarrow (P \ [\![A]\!] \ Q) \sqsubseteq (P' \ [\![A]\!] \ Q')$
sorry

lemma *mono-Inter-ref*: $[\![P \sqsubseteq P'; \ Q \sqsubseteq Q']\!] \Longrightarrow (P|||Q) \sqsubseteq (P'|||Q')$
sorry

lemma *mono-Par-ref*: $[\![P \sqsubseteq P'; \ Q \sqsubseteq Q']\!] \Longrightarrow (P||Q) \sqsubseteq (P'||Qk')$
sorry

lemma *least-process*: $\bot \leq (P::'a \ process)$
by(*auto intro*: *Process.le-approx-implies-le-ref*)

lemma *subset-F-Bot*: $F \ Q \leq F \ \bot$
by(*auto intro*: *Process.le-approx-implies-le-ref is-processT2-S*)

lemma *subset-D-Bot*: $D \ Q \leq D \ \bot$
by (*auto intro*: *least-process le-ref1 D-ftF*)

lemma *eq-F-Bot*: $F \ P = F \ \bot = (F \ \bot \subseteq F \ P)$
by (*auto*, *subgoal-tac* $F \ P \subseteq F \ \bot$,
auto simp: *subset-F-Bot Legacy.is-processT2-S*)

lemma *eq-D-Bot*: $D \ P = D \ \bot = (D \ \bot \subseteq D \ P)$
by (*auto*, *subgoal-tac* $D \ P \subseteq D \ \bot$,
auto simp: *subset-D-Bot Legacy.D-ftF*)

lemma *ftF-D-Bot*: *front-tickFree* $t = (t \in D \ \bot)$
by (*simp add*: *D-Bot Bot-is-UU*[*symmetric*])

lemma *ftF-T-Bot*: *front-tickFree* $t = (t \in T \ \bot)$
by (*simp add*: *T-Bot Bot-is-UU*[*symmetric*])

lemma *D-Bot-eq-T-Bot*: $D \ \bot = T \ \bot$
by (*simp add*: *T-Bot D-Bot Bot-is-UU*[*symmetric*])

lemma *is-processT6-S3*: $[\![fst \ x = []; \ tick \notin snd \ x; \ [tick] \in T \ P]\!] \Longrightarrow x \in F \ P$

by (*subst surjective-pairing*, *simp add*: *is-processT6-S2*)

lemma *div-lemma*: ([] ∈ *D P*) = (*P* = ⊥)
sorry

lemma *det-commute* :(*P* □ *Q*) =(*Q* □ *P*)
apply (*simp add*: *det-def*)
by(*rule-tac f*=*Abs-Process* **in** *arg-cong*) *auto*

lemma *det-bot* [*simp*]: (*P* □ ⊥) = ⊥
by (*auto simp*: *Process-eq-spec eq-F-Bot eq-D-Bot*,
auto simp: *Bot-is-UU*[*symmetric*] *subset-F-Bot Legacy.is-processT2-S*
subset-D-Bot F-det F-Bot D-det D-Bot Legacy.D-ftF)

lemma *det-bot'*[*simp*]: (⊥ □ *P*) = ⊥
by(*subst det-commute*, *simp*)

lemma *det-id*[*simp*]: (*P* □ *P*) = *P*
apply (*simp add*: *Process-eq-spec*, *rule conjI*)
apply (*auto*)
apply (*simp-all only*: *D-det F-det Un-def Sigma-def*)
apply (*simp-all only*: *mem-Collect-eq*)
apply (*simp add*: *sym*)
apply ((*erule disjE* | *erule conjE*)+,*simp*)+
apply (*subst is-processT8*, *simp-all*)
apply (*erule conjE*)+
apply (*subst is-processT6-S2*, *simp-all*)
done

lemma *det-assoc*: ((*M* □ *P*) □ *Q*) = (*M* □ (*P* □ *Q*))
apply (*simp add*: *Process-eq-spec*, *rule conjI*)
apply (*auto*)
apply (*simp-all add*: *D-det F-det Un-def Sigma-def*)
apply ((*erule disjE* | *erule conjE*)+, *simp-all*)+
apply (*rule disjI2* | *rule conjI*)+
apply (*simp-all add*: *T-det*)
apply ((*erule disjE* | *erule conjE*)+, *simp-all*)+
done

lemma *det-STOP*[*simp*]: (*P* □ *STOP*) = *P*
apply (*simp add*: *Process-eq-spec*, *rule conjI*)
apply (*auto*)
apply (*simp-all add*: *D-det F-det D-STOP F-STOP T-STOP Un-def Sigma-def*)
apply ((*erule disjE* | *erule conjE*)+, *simp-all*)+
apply (*subst is-processT8*, *simp-all*)
apply (*erule conjE*)+

apply (*subst is-processT6-S2*, *simp-all*)
done

lemma *det-STOP'*[*simp*]: $(STOP \;\Box\; P) = P$
by(*simp add*: *det-commute*)

lemma *ndet-id*[*simp*]: $(P \sqcap P) = P$
by (*simp-all add*: *F-ndet D-ndet Process-eq-spec*)

lemma *ndet-commute*: $(P \sqcap Q) = (Q \sqcap P)$
by(*auto simp*: *ndet-def Un-commute*)

lemma *ndet-bot*[*simp*]: $(P \sqcap \bot) = \bot$
apply (*simp add*: *Process-eq-spec*, *rule conjI*)
apply (*rule equalityI*)
prefer *3*
apply (*rule equalityI*)
apply (*auto simp*: *D-ndet F-ndet subset-F-Bot subset-D-Bot*
Un-upper1 Un-upper2 Legacy.is-processT2-S Legacy.D-ftF)
done

lemma *ndet-bot'*[*simp*]: $(\bot \sqcap P) = \bot$
by(*subst ndet-commute*, *simp*)

lemma *non-det-assoc*: $((M \sqcap P) \sqcap Q) = (M \sqcap (P \sqcap Q))$
by (*simp-all add*: *F-ndet D-ndet Process-eq-spec Un-assoc*)

lemma *det-distrib*: $(M \;\Box\; (P \sqcap Q)) = ((M \;\Box\; P) \sqcap (M \;\Box\; Q))$
apply (*simp-all add*: *Process-eq-spec F-det D-det F-ndet D-ndet Un-def* , *rule conjI*)
apply (*rule set-eqI*)
prefer *2*
apply (*rule equalityI*)
apply (*rule subsetI*)
apply (*simp*)
apply ((*erule disjE* | *erule conjE*)+, *simp-all*)+
apply (*rule subsetI*)
apply (*simp*)
apply ((*erule disjE* | *erule conjE*)+, *simp-all*)+
apply (*auto simp*: *T-ndet*)
done

lemma *non-det-distrib*: $(M \sqcap (P \;\Box\; Q)) = ((M \sqcap P) \;\Box\; (M \sqcap Q))$
apply (*simp-all add*: *Process-eq-spec F-det D-det F-ndet D-ndet Un-def* ,
rule conjI)
prefer *2*
apply (*rule equalityI*)

apply (*rule subsetI*)
apply (*simp*)
apply ((*erule disjE* | *erule conjE*)+, *simp-all*)+
apply (*rule subsetI*)
apply (*simp*)
apply ((*erule disjE* | *erule conjE*)+, *simp-all*)+
apply (*rule equalityI*)
apply (*rule subsetI*)
apply (*simp*)
apply (*auto simp*:*T-ndet*)
apply (*simp-all add*: *is-processT8 excluded-middle is-processT6-S2*)
done

lemma *pref-non-det*: $(a \rightarrow (P \sqcap Q)) = ((a \rightarrow P) \sqcap (a \rightarrow Q))$
apply (*simp add*: *Process-eq-spec*, *rule conjI*)
apply (*auto*)
apply (*simp-all add*: *write0-def D-ndet F-ndet F-Mprefix D-Mprefix Un-def*)
apply ((*erule disjE* | *erule conjE*)+, *simp-all*)+
done

lemma *Mprefix-STOP*: $(Mprefix\ \{\}\ P) = STOP$
apply (*simp-all add*: *Process-eq-spec* , *rule conjI*)
apply (*auto*)
apply (*simp-all add*:*F-Mprefix D-Mprefix D-STOP F-STOP*)
done

lemma *mprefix-Un-distrD*:
$D(Mprefix\ (A \cup B)\ P) = D\ (Mprefix\ A\ P \Box Mprefix\ B\ P)$
apply (*auto*)
apply (*simp-all add*: *D-det D-Mprefix Un-def*)
apply (*erule disjE* | *erule conjE*)+
apply (*simp add*: *image-def*)
apply (*auto*)
done

lemma *mprefix-Un-distrF*:
$F(Mprefix\ (A \cup B)\ P) = F((Mprefix\ A\ P) \Box (Mprefix\ B\ P))$
apply (*auto*)
apply ((*erule disjE* | *erule conjE*)+ |
simp-all add: *F-det F-Mprefix Un-def image-def*)+
apply (*auto*)
done

lemma *mprefix-Un-distr*: $(Mprefix\ (A \cup B)\ P) = ((Mprefix\ A\ P) \Box (Mprefix\ B\ P))$
by (*simp-all add*: *Process-eq-spec mprefix-Un-distrD mprefix-Un-distrF*)

lemma *mnon-det-non-det*: $(Mprefix\ (A \cup \{a\})\ P) = ((Mprefix\ A\ P) \Box (a \rightarrow (P\ a)))$

by (*subst mprefix-Un-distr*, *subst mprefix-singl*, *simp*)

lemma *pref-det-non-det*: $((a \rightarrow P) \,\Box\, (a \rightarrow Q)) = ((a \rightarrow P) \sqcap (a \rightarrow Q))$
apply (*simp-all add*: *Process-eq-spec* , *rule conjI*)
apply (*auto*)
apply (*simp-all add*: *write0-def F-ndet D-ndet F-det D-det*
F-Mprefix D-Mprefix Un-def)
apply ((*erule disjE* | *erule conjE*)+, *simp-all*)+
done

lemma *SEQ-SKIPD*: $D(P\ ;\!\!\rangle\ SKIP) = D\ P$
apply (*simp-all add*: *D-seq D-SKIP*)
apply (*auto simp*: *is-processT7*)
apply (*case-tac tickFree x*)
apply (*rule-tac x=x* **in** *exI*)
apply (*simp*)
apply (*subgoal-tac* $\exists\ s.\ x = s\ @\ [tick]$)
apply (*erule exE*)
apply (*rule-tac x=s* **in** *exI*)
apply (*auto simp*: *is-processT9*)
apply (*rule-tac s=*[*tick*] **and** *P=P* **in** *append-T-imp-tickFree*)
apply (*rule D-T*)
prefer *3*
apply (*rule nonTickFree-n-frontTickFree*)
prefer *2*
apply (*rule-tac P=P* **in** *is-processT2-TR*[*rule-format*])
by (*simp-all add*: *D-T*)

lemma *SEQ-SKIPF*: $F(P\ ;\!\!\rangle\ SKIP) = F\ P$
apply (*simp-all add*: *F-seq D-seq F-SKIP D-SKIP*)
apply (*auto*)
apply (*subgoal-tac b* $\subseteq$ *insert tick b*)
apply (*rule-tac Y=insert tick b* **in** *is-processT4*[*rule-format*])
apply (*simp-all*)
apply (*subst insert-def*)
apply (*auto*)
apply (*simp add*: *T-F is-processT6-S1*)
defer
apply (*subst is-processT8-Pair*)
apply (*simp-all add*: *is-processT7*)
apply (*case-tac tick* $\in$ *b*)
apply (*subgoal-tac insert tick b* $\subseteq$ *b*)
apply (*rule-tac Y=b* **in** *is-processT4*[*rule-format*])
apply (*simp-all*)
apply (*subgoal-tac* $\exists\ s.\ a = s@[tick]$)

apply (*erule exE*)
apply (*case-tac tickFree a*)
apply (*simp-all*)
apply (*case-tac* (*a*,*insert tick b*) ∈ *F P*)
apply (*simp-all*)
apply (*erule-tac x*=*a* **in** *allE*)
apply (*erule-tac x*=[] **in** *allE*, *simp*)
prefer *3*
apply (*subgoal-tac* ∃ *s*. *a* = *s*@[*tick*])
apply (*erule exE*)
apply (*simp-all*)
apply (*erule-tac x*=*s* **in** *allE*)
apply (*erule-tac x*=[*tick*] **in** *allE*, *simp*)
sorry

lemma *SEQ-SKIP*: (*P* ‘;‘ *SKIP*) = *P*
by (*simp-all add*: *Process-eq-spec SEQ-SKIPD SEQ-SKIPF*)

lemma *SKIP-SEQD*: *D*(*SKIP* ‘;‘ *P*) = *D P*
by (*simp-all add*: *D-seq D-SKIP T-SKIP*)

lemma *SKIP-SEQF*: *F*(*SKIP* ‘;‘ *P*) = *F P*
apply (*simp-all add*: *F-seq F-SKIP D-SKIP T-SKIP*)
by (*auto simp*: *is-processT8-Pair*)

lemma *SKIP-SEQ*: (*SKIP* ‘;‘ *P*) = *P*
by (*simp-all add*: *Process-eq-spec SKIP-SEQD SKIP-SEQF*)

lemma *ev-Neq-tick1*:*ev a* = *b* ==> *b* ~= *tick*
by (*auto*)

lemma *Nelem-image-ev*:
⟦∀ *c*. *c* ∈ *B* ⟶ *c* ∉ *C*; *hd x* ∈ *ev* ‘ *B*⟧ ⟹ *hd x* ∉ *ev* ‘ *C*
by (*auto*)

lemma *mprefix-seqD*:
D((*Mprefix A P*) ‘;‘ *Q*) = *D*(*Mprefix A* (λ *x*. (*P x*) ‘;‘ *Q*))
apply (*simp-all add*: *D-Mprefix D-seq T-Mprefix*)
apply (*auto*)
apply (*rule-tac x*=*tl t1* **in** *exI*)
apply (*rule-tac x*=*t2* **in** *exI*)
apply (*case-tac t1*)
apply (*simp-all*)

apply (*case-tac t1*)
apply (*simp-all*)
apply (*rule-tac x=a* **in** *exI*)
apply (*rule conjI*)
apply (*case-tac t1*)
apply (*simp-all*)
apply (*rule disjI2*)
apply (*rule-tac x=tl t1* **in** *exI*)
apply (*rule-tac x=t2* **in** *exI*)
apply (*simp*)
apply (*case-tac t1*)
apply (*simp-all*)
apply (*rule-tac x=ev xa # t1* **in** *exI*)
apply (*rule-tac x=t2* **in** *exI*)
apply (*simp*)
apply (*case-tac x*)
apply (*simp-all*)
apply (*rule-tac x=ev xa # t1* **in** *exI*)
apply (*rule-tac x=t2* **in** *exI*)
apply (*simp*)
apply (*case-tac x*)
apply (*simp-all*)
apply (*auto*)
by (*erule-tac x=ev xa # t1* **in** *allE*, *erule-tac x=t2* **in** *allE*, *simp*)+

lemma *mprefix-seqF1*:
$a \notin B \Longrightarrow (A \cup \{a\}) \cap B = A \cap B$
by (*auto*)

lemma *mprefix-seqF*: $F((Mprefix\ A\ P)\ ;'\ Q) = F(Mprefix\ A\ (\lambda x.\ (P\ x)\ ;'\ Q))$
apply (*simp-all add*: *split-def mprefix-seqD F-D-SEQ-spec F-Mprefix T-Mprefix D-Mprefix*)
apply (*rule equalityI*)
apply (*rule subsetI*)
apply (*simp-all*)
apply (*erule disjE* | *erule conjE*)+
apply (*rule disjI1*, *simp*)
apply (*erule disjE* | *erule conjE* | *erule exE*)+
apply (*rule disjI2*)
apply (*rule conjI*, *simp*)+
apply (*rule-tac x=a* **in** *exI*, *simp-all*)
apply (*rule disjI1*)
apply (*case-tac fst x*, *simp-all*)
apply (*erule disjE* | *erule conjE* | *erule exE*)+
apply (*rule disjI2*)
apply (*rule conjI*)
apply (*simp-all add*: *ev-Neq-tick1*)
apply (*rule impI*, *simp*)

apply (*rule conjI*)
apply (*subst hd-append*, *simp*)
apply (*rule conjI*)
apply (*rule impI*, *simp*)
apply (*rule impI*, *simp*)
apply (*rule-tac x=a* **in** *exI*)
apply (*rule conjI*)
apply (*subst hd-append*, *simp*)
apply (*rule conjI*)
apply (*rule impI*, *simp*)
apply (*rule impI*, *simp*)
apply (*rule disjI2*)
apply (*rule disjI1*)
apply (*rule-tac x=tl t1* **in** *exI*)
apply (*rule-tac x=t2* **in** *exI*, *simp*)
apply (*subgoal-tac t1* $\neq$ [])
apply (*rule conjI*)
apply (*subst tl-append2*)
apply (*simp-all*)
apply (*case-tac t1*, *simp-all*)
apply (*erule disjE* | *erule conjE* | *erule exE*)+
apply (*rule-tac x=a* **in** *exI*)
apply (*rule conjI*, *simp*)
apply (*rule disjI2*)+
apply (*simp*)
apply (*rule conjI*)
apply (*rule subsetI*, *simp*)
apply (*subst image-def*, *simp*)
apply (*rule subsetI*, *simp*)
apply (*erule disjE* | *erule conjE* | *erule exE*)+
apply (*rule disjI1*)
apply (*rule conjI*)
apply (*rule-tac x=a* **in** *exI*, *simp*)
apply (*case-tac fst x*, *simp-all add*: *ev-Neq-tick1*)
apply (*erule disjE* | *erule conjE* | *erule exE*)+
apply (*rule disjI2*)
apply (*rule disjI1*)
apply (*rule-tac x=*(*ev a*)#*t1* **in** *exI*, *simp*)
apply (*rule-tac x=t2* **in** *exI*, *simp*)
apply (*rule conjI*)
apply (*case-tac fst x*, *simp-all*)
by (*auto*)

lemma *mprefix-seq*:
(($Mprefix\ A\ P$) ‘;‘ Q) = ($Mprefix\ A\ (\lambda x.\ (P\ x)$ ‘;‘ $Q)$)
by (*simp-all add*: *Process-eq-spec mprefix-seqF mprefix-seqD*)

lemma *pref-seq*: $((a \rightarrow P)\ ;\!\!;\ Q) = (a \rightarrow (P\ ;\!\!;\ Q))$
by (*simp add*: *write0-def mprefix-seq*)

lemma *STOP-SEQ*: $(STOP\ ;\!\!;\ P) = STOP$
by (*auto simp*: *Process-eq-spec F-SEQ-spec D-seq F-STOP D-STOP T-STOP Un-def*)

lemma *prefix-stop-seq*: $((a \rightarrow STOP)\ ;\!\!;\ P) = (a \rightarrow STOP)$
by (*simp add*: *pref-seq STOP-SEQ*)

lemma *prefix-skip-seq*: $((a \rightarrow SKIP)\ ;\!\!;\ P) = (a \rightarrow P)$
by (*simp add*: *pref-seq SKIP-SEQ*)

lemma *Bot-SEQ*: $(\bot\ ;\!\!;\ P) = \bot$
apply(*simp add*: *Bot-is-UU*[*symmetric*])
apply (*simp add*: *Process-eq-spec D-Bot F-Bot D-seq F-seq T-Bot*)
apply (*auto*)
apply (*subgoal-tac front-tickFree t2*)
apply (*rule front-tickFree-append*, *simp-all*)
apply (*rule-tac a=tick* **in** *front-tickFree-implies-tickFree*, *simp*)
apply (*simp-all add*: *is-processT2 front-tickFree-append*)
apply (*tactic distinct-subgoals-tac*)
apply (*subgoal-tac* $\exists\ S.\ (t2,\ S) \in F\ P$)
apply (*erule exE*)
apply (*subgoal-tac front-tickFree t2*)
apply (*rule front-tickFree-append*, *simp-all*)
apply (*rule-tac a=tick* **in** *front-tickFree-implies-tickFree*, *simp*)
apply (*simp add*: *is-processT2*)
apply (*rule exI*)
apply (*simp add*: *is-processT8*)
apply (*erule-tac x=*[] **in** *allE*)+
apply (*simp*)
apply (*rule-tac x=*[] **in** *exI*)
apply (*simp*)
done

lemma *SEQ-Ndet-distrRD*: $D((P \sqcap Q)\ ;\!\!;\ S) = D((P\ ;\!\!;\ S) \sqcap (Q\ ;\!\!;\ S))$
by (*simp-all add*: *D-seq D-ndet T-ndet Un-def*, *auto*)

lemma *SEQ-Ndet-distrRF*: $F((P \sqcap Q)\ ;\!\!;\ S) = F((P\ ;\!\!;\ S) \sqcap (Q\ ;\!\!;\ S))$
by (*simp-all add*: *F-seq D-ndet T-ndet F-ndet SEQ-Ndet-distrRD Un-def*, *auto*)

lemma *SEQ-Ndet-distrR*: $((P \sqcap Q)\ ;\!\!;\ S) = ((P\ ;\!\!;\ S) \sqcap (Q\ ;\!\!;\ S))$
by (*simp add*: *Process-eq-spec SEQ-Ndet-distrRD SEQ-Ndet-distrRF*)

lemma *SEQ-Ndet-distrLD*: $D(P \text{ ‘;‘ } (Q \sqcap S)) = D((P \text{ ‘;‘ } Q) \sqcap (P \text{ ‘;‘ } S))$
by (*simp-all add*: *D-seq D-ndet T-ndet Un-def*, *auto*)

lemma *SEQ-Ndet-distrLF*: $F(P \text{ ‘;‘ } (Q \sqcap S)) = F((P \text{ ‘;‘ } Q) \sqcap (P \text{ ‘;‘ } S))$
by (*simp-all add*: *F-seq D-ndet T-ndet F-ndet SEQ-Ndet-distrRD Un-def*, *auto*)

lemma *SEQ-Ndet-distrL*: $(P \text{ ‘;‘ } (Q \sqcap S)) = ((P \text{ ‘;‘ } Q) \sqcap (P \text{ ‘;‘ } S))$
by (*simp add*: *Process-eq-spec SEQ-Ndet-distrLD SEQ-Ndet-distrLF*)

lemma *SEQ-Det-distrRD*: $D((P \,\Box\, Q) \text{ ‘;‘ } S) = D((P \text{ ‘;‘ } S) \,\Box\, (Q \text{ ‘;‘ } S))$
by (*simp-all add*: *D-seq D-det T-det Un-def*, *auto*)

lemma *SEQ-Det-distrRF*: $F((P \,\Box\, Q) \text{ ‘;‘ } S) \subseteq F((P \text{ ‘;‘ } S) \,\Box\, (Q \text{ ‘;‘ } S))$
apply (*simp-all add*: *F-SEQ-spec D-det T-det F-det split-def D-seq T-seq Un-def*)
apply (*rule subsetI*, *simp*)
apply (*erule conjE* | *erule disjE*)+
apply (*rule disjI1*)
apply (*rule conjI*, *simp-all*)
apply (*erule conjE* | *erule disjE*)+
apply (*rule disjI2*)
apply (*rule disjI1*)
apply (*rule conjI*)
apply (*simp-all*)
apply (*rule disjI2*)
apply (*erule conjE* | *erule disjE*)+
apply (*rule disjI1*, *simp*)
apply (*rule disjI2*)
apply (*rule disjI2*)
apply (*rule disjI1*, *simp*)
apply (*erule conjE* | *erule disjE* | *erule exE*)+
apply (*rule disjI2*)
apply (*case-tac fst x* = [])
apply (*rule disjI2*)
apply (*rule disjI2*)
apply (*simp-all*)
sorry

find-theorems *front-tickFree*
lemma *par-Int-botD*: $D(P \llbracket A \rrbracket \bot) = D\ \bot$
apply (*auto simp*: *eq-D-Bot*)
apply (*simp add*: *Bot-is-UU*[*symmetric*])
apply (*simp add*: *D-sync D-Bot T-Bot*)
apply (*auto simp*: *Process.front-tickFree-append*)

sorry

lemma *par-Int-botF*: $F(P \llbracket A \rrbracket \bot) = F\ \bot$
sorry

lemma *par-Int-bot*[*simp*]: $(P \llbracket A \rrbracket \bot) = \bot$
by (*simp add*: *Process-eq-spec par-Int-botF par-Int-botD*)

lemma *par-Int-bot1*[*simp*]: $(\bot \llbracket A \rrbracket P) = \bot$
sorry

lemma *par-Int-skip-D*: $D(SKIP \llbracket A \rrbracket SKIP) = D\ SKIP$
by (*simp add*: *D-SKIP D-sync*)

lemma *par-Int-skip-F*: $F(SKIP \llbracket A \rrbracket SKIP) = F\ SKIP$
apply (*simp add*: *F-SKIP D-SKIP T-SKIP F-sync Un-def Sigma-def*)
sorry

lemma *par-Int-skip*: $(SKIP \llbracket A \rrbracket SKIP) = SKIP$
by (*simp add*: *Process-eq-spec par-Int-skip-F par-Int-skip-D*)

lemma *sync-commute*: $(P \llbracket A \rrbracket Q) = (Q \llbracket A \rrbracket P)$
sorry

lemmas *Par-Int-commute* = *sync-commute*

lemma *Inter-commute*: $(P \mid\mid\mid Q) = (Q \mid\mid\mid P)$
by(*rule sync-commute*)

lemma *Inter-skip-D*: $D(P \mid\mid\mid SKIP) = D\ P$
apply (*simp add*: *D-SKIP D-sync T-SKIP*)
sorry

lemma *Inter-skip-F1*:
$[\![(t, X) \in F\ P; fst\ x\ setinterleaves\ ((t, [tick]), \{tick\})]\!]$
$\Longrightarrow x \in F\ P$
sorry

lemma *Inter-skip-F*: $F(P \mid\mid\mid SKIP) = F\ P$
sorry

lemma *Inter-skip1*: $(P \mid\mid\mid SKIP) = P$
sorry

lemma *Inter-skip2*: $(SKIP \mid\mid\mid P) = P$
sorry

lemma *skip-Neq-stop*: $SKIP \neq STOP$
by (*auto simp*: *Process-eq-spec F-SKIP F-STOP D-SKIP D-STOP Un-def*)

lemma *stop-Neq-skip*: $STOP \neq SKIP$
by (*subst not-sym*, *simp-all add*: *skip-Neq-stop*)

lemma *Inter-stop-seq-stop-D*: $D(P \mid\mid\mid STOP) = D(P \text{ `;` } STOP)$
apply (*simp add*: *D-sync D-seq D-STOP T-STOP*)

sorry

lemma *setH1*:
$(X \cup Y) \cap A \cup X \cap Y \cup A = X \cap Y \cup A$
by (*auto*)

lemma *Inter-stop-seq-stop-F*: $F(P \mid\mid\mid STOP) = F(P \text{ `;` } STOP)$
sorry

lemma *Inter-stop-seq-stop*:
$(P \mid\mid\mid STOP) = (P \text{ `;` } STOP)$
by (*auto simp*: *Process-eq-spec Inter-stop-seq-stop-D Inter-stop-seq-stop-F*)

lemma *Inter-stop-seq-stop1*:
$(STOP \mid\mid\mid P) = (P \text{ `;` } STOP)$
sorry

lemma *par-int-ndet-distribD*:
$D(M \llbracket A \rrbracket (P \sqcap Q)) = D((M \llbracket A \rrbracket P) \sqcap (M \llbracket A \rrbracket Q))$
sorry

lemma *par-int-ndet-distribF*:
$F(M \llbracket A \rrbracket (P \sqcap Q)) = F((M \llbracket A \rrbracket P) \sqcap (M \llbracket A \rrbracket Q))$
sorry

lemma *par-int-ndet-distrib*:
$(M \llbracket A \rrbracket (P \sqcap Q)) = ((M \llbracket A \rrbracket P) \sqcap (M \llbracket A \rrbracket Q))$
sorry

lemma *par-int-ndet-distrib1*:
$((P \sqcap Q) \llbracket A \rrbracket M) = ((P \llbracket A \rrbracket M) \sqcap (Q \llbracket A \rrbracket M))$
sorry

lemma *par-comm*: $(P \mid\mid Q) = (Q \mid\mid P)$
sorry

lemma *par-ndet-distrib1*:
$(M \mid\mid (P \sqcap Q)) = ((M \mid\mid P) \sqcap (M \mid\mid Q))$
by (*simp add*: *par-int-ndet-distrib*)

lemma *par-ndet-distrib2*:
$((P \sqcap Q) \mid\mid M) = ((P \mid\mid M) \sqcap (Q \mid\mid M))$
by (*simp add*: *par-int-ndet-distrib1*)

lemma *par-stopD*: $P \neq \bot \Longrightarrow D(P \mid\mid STOP) = D\ STOP$
sorry

lemma *par-stopF*: $P \neq \bot \Longrightarrow F(P \mid\mid STOP) = F\ STOP$
sorry

lemma *par-stop*: $P \neq \bot \Longrightarrow (P \mid\mid STOP) = STOP$
sorry

lemma *par-assocD1*: $D((P \mid\mid Q) \mid\mid S) \subseteq D(P \mid\mid (Q \mid\mid S))$
sorry

lemma *par-assocD2*: $D(P \mid\mid (Q \mid\mid S)) \subseteq D((P \mid\mid Q) \mid\mid S)$
sorry

lemma *par-assocD*: $D((P \mid\mid Q) \mid\mid S) = D(P \mid\mid (Q \mid\mid S))$
sorry

lemma *par-assocF1*: $F((P \mid\mid Q) \mid\mid S) \subseteq F(P \mid\mid (Q \mid\mid S))$
sorry

lemma *par-assocF2*: $F(P \mid\mid (Q \mid\mid S)) \subseteq F((P \mid\mid Q) \mid\mid S)$

sorry

lemma *par-assocF*: $F((P \mid\mid Q) \mid\mid S) = F(P \mid\mid (Q \mid\mid S))$
sorry

lemma *par-assoc*: $((P \mid\mid Q) \mid\mid S) = (P \mid\mid (Q \mid\mid S))$
sorry

lemma *hide-set-botD*: $D(\bot \setminus A) = D\ \bot$
sorry

lemma *hide-set-botF*: $F(\bot \setminus A) = F\ \bot$
sorry

lemma *hide-set-bot*[*simp*]: $(\bot \setminus A) = \bot$
sorry

lemma *hide-set-STOPD*: $D(STOP \setminus A) = D\ STOP$
sorry

lemma *hide-set-STOPF*: $F(STOP \setminus A) = F\ STOP$
sorry

lemma *hide-set-STOP*: $(STOP \setminus A) = STOP$
sorry

lemma *hide-set-SKIPD*: $D(SKIP \setminus A) = D\ SKIP$
sorry

lemma *hide-set-SKIPF*: $F(SKIP \setminus A) = F\ SKIP$
sorry

lemma *hide-set-SKIP*: $(SKIP \setminus A) = SKIP$
sorry

lemma *hide-set-emptyD*: $D(P \setminus \{\}) = D\ P$
sorry

lemma *hide-set-emptyF*: $F(P \setminus \{\}) = F\ P$
sorry

lemma *hide-set-empty*: $(P \setminus \{\}) = P$
sorry

lemma $D(P \setminus (A\ Un\ B)) <= D((P \setminus A) \setminus B)$
sorry

lemma $D((P \setminus A) \setminus B) <= D(P \setminus (A\ Un\ B))$
sorry

lemma *f-mono*:
$\forall i.\ f\ i < f\ (Suc\ i) \Longrightarrow i < j \longrightarrow f\ i < f\ j$
sorry

lemma *f-0-less-f-i*:
$[\![f\ 0 = t; \forall i.\ f\ i < f\ (Suc\ i);\ 1 \leq i]\!] \Longrightarrow t < f\ i$
sorry

lemma
$\exists f.\ f\ IsChainOver\ t \wedge$
$f\ Congruent\ t\ ModuloHide\ A \wedge (\forall i.\ f\ i \in T\ P \vee f\ i \in T\ Q) \Longrightarrow$
$\exists f.\ f\ IsChainOver\ t \wedge$
$f\ Congruent\ t\ ModuloHide\ A \wedge ((\forall i.\ f\ i \in T\ P) \vee (\forall i.\ f\ i \in T\ Q))$
sorry

lemma $D((P \sqcap Q) \setminus A) = D((P \setminus A) \sqcap (Q \setminus A))$
sorry

lemma *le-trans*:
$\bigwedge i.\ [\![i \leq j; j \leq k]\!] \Longrightarrow i \leq (k::nat)$
by (*auto*)

lemma *length-f-i1*:
$f\ 0 = t \wedge (\forall i.\ f\ i < f\ (Suc\ i)) \Longrightarrow \forall i.\ length\ t + i \leq length\ (f\ i)$
sorry

lemma *f-i-Neq-Nil1*:
$f\ 0 = t \wedge (\forall i.\ f\ i < f\ (Suc\ i)) \Longrightarrow \forall i.\ i \neq 0 \longrightarrow f\ i \neq []$
sorry

lemma *tl-f-i-less-tl-f-Suc1*:
$f\ 0 = t \wedge (\forall i.\ f\ i < f\ (Suc\ i)) \Longrightarrow$
$\forall i.\ i \neq 0 \longrightarrow tl\ (f\ i) < tl\ (f\ (Suc\ i))$
sorry

lemma *length-f-i2*:
$f\ 0 = t \wedge (\forall i.\ f\ i < f\ (Suc\ i)) \Longrightarrow \forall i.\ length\ t + i < length\ (f\ (Suc\ i))$
sorry

lemma *tl-f-Suc-i-Neq-Nil*:
$f\ 0 = t \wedge (\forall\, i.\ f\ i < f\ (Suc\ i)) \Longrightarrow \forall\, i.\ i \neq 0 \longrightarrow tl\ (f\ (Suc\ i)) \neq []$
sorry

lemma *tl-f-i-less-tl-f-Suc2*:
$f\ 0 = t \wedge (\forall\, i.\ f\ i < f\ (Suc\ i)) \Longrightarrow \forall\, i.\ f\ (Suc\ i) \neq []$
sorry

lemma *tl-f-Suc-i-less-tl-f-Suc2*:
$f\ 0 = t \wedge (\forall\, i.\ f\ i < f\ (Suc\ i)) \Longrightarrow$
$(if\ i = 0\ then\ t\ else\ tl\ (f\ (Suc\ i))) < tl\ (f\ (Suc\ (Suc\ i)))$
sorry

lemma *det-left-commute*: $(M \ \Box\ P \ \Box\ Q) = (P \ \Box\ M \ \Box\ Q)$
by (*simp add*: *det-commute*)

lemma *det-left-id*: $(M \ \Box\ M \ \Box\ Q) = (M \ \Box\ Q)$
by (*simp add*: *det-id*)

lemma *non-det-left-commute*: $(M \sqcap P \sqcap Q) = (P \sqcap M \sqcap Q)$
by (*simp add*: *ndet-commute*)

lemma *non-det-left-id*: $(M \sqcap M \sqcap Q) = (M \sqcap Q)$
by (*simp add*: *ndet-id*)

lemma *par-left-commute*: $(M \ ||\ P \ ||\ Q) = (P \ ||\ M \ ||\ Q)$
by (*simp add*: *par-comm*)

lemma *mprefix-Par-Int-distr3D1*:
$[\![B \cap C = \{\};\ A \subseteq C]\!]$
$\Longrightarrow D\ (Mprefix\ A\ P\ [\![C\ ]\!]\ Mprefix\ B\ Q)$
$\subseteq D\ (\Box\ x \in\ B \rightarrow (Mprefix\ A\ P\ [\![C\ ]\!]\ Q\ x))$
sorry

lemma *mprefix-Par-Int-distr3D2*:
$[\![B \cap C = \{\};\ A \subseteq C]\!]$
$\Longrightarrow D\ (\Box\ x \in\ B \rightarrow (Mprefix\ A\ P\ [\![C\ ]\!]\ Q\ x))$
$\subseteq D\ (Mprefix\ A\ P\ [\![C\ ]\!]\ Mprefix\ B\ Q)$
sorry

lemma *mprefix-Par-Int-distr3D*:
$[\![B \cap C = \{\};\ A \subseteq C]\!]$

$\Longrightarrow D\ (\Box\ x \in\ B \rightarrow (Mprefix\ A\ P\ [\![C\ ]\!]\ Q\ x))$
$\subseteq D\ (Mprefix\ A\ P\ [\![C]\!]\ Mprefix\ B\ Q)$
sorry

lemma *mprefix-Par-Int-distr3F1*:
$[\![B \cap C = \{\};\ A \subseteq C]\!]$
$\Longrightarrow F\ (Mprefix\ A\ P\ [\![C\ ]\!]\ Mprefix\ B\ Q)$
$\subseteq F\ (\Box\ x \in\ B \rightarrow (Mprefix\ A\ P\ [\![C\ ]\!]\ Q\ x))$
sorry

lemma *mprefix-Par-Int-distr3F2*:
$[\![B \cap C = \{\};\ A \subseteq C]\!]$
$\Longrightarrow F\ (\Box\ x \in\ B \rightarrow (Mprefix\ A\ P\ [\![C\ ]\!]\ Q\ x))$
$\subseteq F\ (Mprefix\ A\ P\ [\![C\ ]\!]\ Mprefix\ B\ Q)$
sorry

lemma *mprefix-Par-Int-distr3F*:
$[\![B \cap C = \{\};\ A \subseteq C]\!]$
$\Longrightarrow F\ (Mprefix\ A\ P\ [\![C\ ]\!]\ Mprefix\ B\ Q) =$
$F\ (\Box\ x \in\ B \rightarrow (Mprefix\ A\ P\ [\![C\ ]\!]\ Q\ x))$
sorry

lemma *mprefix-Par-Int-distr3*:
$[\![B \cap C = \{\};\ A \subseteq C]\!]$
$\Longrightarrow (Mprefix\ A\ P\ [\![C\ ]\!]\ Mprefix\ B\ Q) = \Box\ x \in\ B \rightarrow (Mprefix\ A\ P\ [\![C\ ]\!]\ Q\ x)$
sorry

lemma *mprefix-Par-Int-distr2*:
$[\![A \cap C = \{\};\ B \subseteq C]\!]$
$\Longrightarrow (Mprefix\ A\ P\ [\![C\ ]\!]\ Mprefix\ B\ Q) = \Box\ x \in\ A \rightarrow (P\ x\ [\![C\ ]\!]\ Mprefix\ B\ Q)$
sorry

lemma *mprefix-Par2D1a*:
$[\![B \cap C = \{\};\ A \cap C = \{\};$
$\exists\, t\ u\ r\ v.$
front-tickFree $v \wedge$
tickFree $r \wedge$
$x = r\ @\ v \wedge$
$r\ setinterleaves\ ((t,\ u),\ ev\ `\ C \cup \{tick\}) \wedge$
$t \in D\ (Mprefix\ A\ P) \wedge u \in T\ (Mprefix\ B\ Q)]\!]$
$\Longrightarrow x \in D\ (\Box\ x \in\ A \rightarrow (P\ x\ [\![C\ ]\!]\ Mprefix\ B\ Q)\ \Box$
$\Box\ y \in\ B \rightarrow (Mprefix\ A\ P\ [\![C\ ]\!]\ Q\ y))$
sorry

lemma *mprefix-Par2D1b*:
$[\![B \cap C = \{\};\ A \cap C = \{\};$
$\exists\, t\ u\ r\ v.$

$front\text{-}tickFree\ v \wedge$
$tickFree\ r \wedge$
$x = r\ @\ v \wedge$
$r\ setinterleaves\ ((t,\ u),\ ev\ `\ C \cup \{tick\}) \wedge$
$t \in D\ (Mprefix\ B\ Q) \wedge u \in T\ (Mprefix\ A\ P)[\![$
$\Longrightarrow x \in D\ (\Box\ x \in\ A \rightarrow (P\ x\ [\![C\ ]\!]\ Mprefix\ B\ Q) \Box$
$\Box\ y \in\ B \rightarrow (Mprefix\ A\ P\ [\![C\ ]\!]\ Q\ y))$

sorry

lemma *mprefix-Par2D1*:
$[\![B \cap C = \{\};\ A \cap C = \{\}]\!]$
$\Longrightarrow D\ (Mprefix\ A\ P\ [\![C\ ]\!]\ Mprefix\ B\ Q)$
$\subseteq D\ (\Box\ x \in\ A \rightarrow (P\ x\ [\![C\ ]\!]\ Mprefix\ B\ Q) \Box$
$\Box\ y \in\ B \rightarrow (Mprefix\ A\ P\ [\![C\ ]\!]\ Q\ y))$

sorry

lemma *mprefix-Par2D2a*: [| $B\ Int\ C = \{\};\ A\ Int\ C = \{\};\ x{:}D(\ \Box\ x \in A \rightarrow ((P\ x)\ [\![\ C\ ]\!]$ $(Mprefix\ B\ Q)))$ |] ==> $x : D\ ((Mprefix\ A\ P)\ [\![\ C\ ]\!]\ (Mprefix\ B\ Q))$

sorry

lemma *mprefix-Par2D2b*:
$[\![B \cap C = \{\};\ A \cap C = \{\};\ x \in D\ (\Box\ y \in\ B \rightarrow (Mprefix\ A\ P\ [\![C\ ]\!]\ Q\ y))]\!]$
$\Longrightarrow x \in D\ (Mprefix\ A\ P\ [\![C\ ]\!]\ Mprefix\ B\ Q)$

sorry

lemma *mprefix-Par2D2*:
$[\![B \cap C = \{\};\ A \cap C = \{\}]\!]$
$\Longrightarrow D\ (\Box\ x \in\ A \rightarrow (P\ x\ [\![C\ ]\!]\ Mprefix\ B\ Q) \Box$
$\Box\ y \in\ B \rightarrow (Mprefix\ A\ P\ [\![C\ ]\!]\ Q\ y))$
$\subseteq D\ (Mprefix\ A\ P\ [\![C\ ]\!]\ Mprefix\ B\ Q)$

sorry

lemma *mprefix-Par2D*:
$[\![B \cap C = \{\};\ A \cap C = \{\}]\!]$
$\Longrightarrow D\ (Mprefix\ A\ P\ [\![C\ ]\!]\ Mprefix\ B\ Q) =$
$D\ (\Box\ x \in\ A \rightarrow (P\ x\ [\![C\ ]\!]\ Mprefix\ B\ Q) \Box$
$\Box\ y \in\ B \rightarrow (Mprefix\ A\ P\ [\![C\ ]\!]\ Q\ y))$

sorry

lemma *AuxEv1*:
$\forall\, c.\ c \in A \longrightarrow c \notin B \Longrightarrow \forall\, c.\ c \in ev\ `\ B \longrightarrow c \notin ev\ `\ A$
by (*auto*)

lemma *mprefix-Par2F1*:

$[\![B \cap C = \{\};\ A \cap C = \{\}]\!]$
$\Longrightarrow F\ (Mprefix\ A\ P\ [\![C\ ]\!]\ Mprefix\ B\ Q)$
$\subseteq F\ (\Box\ x \in\ A \rightarrow (P\ x\ [\![C\ ]\!]\ Mprefix\ B\ Q)\ \Box$
$\Box\ y \in\ B \rightarrow (Mprefix\ A\ P\ [\![C\ ]\!]\ Q\ y))$
sorry

lemma *mprefix-Par2F2a*:
$[\![B \cap C = \{\};\ A \cap C = \{\};\ fst\ x \neq [];\ hd\ (fst\ x) \in ev\ `\ A;$
$ev\ a = hd\ (fst\ x);\ (t,\ X) \in F\ (P\ a);\ (u,\ Y) \in F\ (Mprefix\ B\ Q);$
$tl\ (fst\ x)\ setinterleaves\ ((t,\ u),\ ev\ `\ C \cup \{tick\});$
$snd\ x = (X \cup Y) \cap (ev\ `\ C \cup \{tick\}) \cup X \cap Y]\!]$
$\Longrightarrow \exists\, t\ u\ Xa\ Ya.$
$(t = [] \wedge Xa \cap ev\ `\ A = \{\} \vee$
$t \neq [] \wedge$
$hd\ t \in ev\ `\ A \wedge (\exists\, a.\ ev\ a = hd\ t \wedge (tl\ t,\ Xa) \in F\ (P\ a))) \wedge$
$(u = [] \wedge Ya \cap ev\ `\ B = \{\} \vee$
$u \neq [] \wedge$
$hd\ u \in ev\ `\ B \wedge (\exists\, a.\ ev\ a = hd\ u \wedge (tl\ u,\ Ya) \in F\ (Q\ a))) \wedge$
$fst\ x\ setinterleaves\ ((t,\ u),\ ev\ `\ C \cup \{tick\}) \wedge$
$(X \cup Y) \cap (ev\ `\ C \cup \{tick\}) \cup X \cap Y =$
$(Xa \cup Ya) \cap (ev\ `\ C \cup \{tick\}) \cup Xa \cap Ya$
sorry

lemma *mprefix-Par2F2b*:
$(\exists\, t\ u\ Xa\ Ya.$
$P\ t\ Xa \wedge$
$Q\ u\ Ya \wedge$
$fst\ x\ setinterleaves\ ((t,\ u),\ ev\ `\ C \cup \{tick\}) \wedge$
$(X \cup Y) \cap (ev\ `\ C \cup \{tick\}) \cup X \cap Y =$
$(Xa \cup Ya) \cap (ev\ `\ C \cup \{tick\}) \cup Xa \cap Ya) =$
$(\exists\, u\ t\ Ya\ Xa.$
$Q\ u\ Ya \wedge$
$P\ t\ Xa \wedge$
$fst\ x\ setinterleaves\ ((u,\ t),\ ev\ `\ C \cup \{tick\}) \wedge$
$(Y \cup X) \cap (ev\ `\ C \cup \{tick\}) \cup Y \cap X =$
$(Ya \cup Xa) \cap (ev\ `\ C \cup \{tick\}) \cup Ya \cap Xa)$
sorry

lemma *mprefix-Par2F2*:
$[\![B \cap C = \{\};\ A \cap C = \{\}]\!]$
$\Longrightarrow F\ (\Box\ x \in\ A \rightarrow (P\ x\ [\![C\ ]\!]\ Mprefix\ B\ Q)\ \Box$
$\Box\ y \in\ B \rightarrow (Mprefix\ A\ P\ [\![C\ ]\!]\ Q\ y))$
$\subseteq F\ (Mprefix\ A\ P\ [\![C\ ]\!]\ Mprefix\ B\ Q)$
sorry

lemma *mprefix-Par2F*:

$[\![B \cap C = \{\};\ A \cap C = \{\}]\!]$
$\Longrightarrow F\ (Mprefix\ A\ P\ [\![C\ ]\!]\ Mprefix\ B\ Q) =$
$F\ (\Box\ x \in\ A \rightarrow (P\ x\ [\![C\ ]\!]\ Mprefix\ B\ Q)\ \Box$
$\Box\ y \in\ B \rightarrow (Mprefix\ A\ P\ [\![C\ ]\!]\ Q\ y))$
sorry

lemma *mprefix-Par-Int2*:
$[\![B \cap C = \{\};\ A \cap C = \{\}]\!]$
$\Longrightarrow (Mprefix\ A\ P\ [\![C\ ]\!]\ Mprefix\ B\ Q) =$
$(\Box\ x \in\ A \rightarrow (P\ x\ [\![C\ ]\!]\ Mprefix\ B\ Q)\ \Box$
$\Box\ y \in\ B \rightarrow (Mprefix\ A\ P\ [\![C\ ]\!]\ Q\ y))$
sorry

lemma *mprefix-Par-Int-distr1D1a*:
$[\![A \subseteq C;\ B \subseteq C;$
$\exists t\ u\ r\ v.$
$front\text{-}tickFree\ v \wedge$
$tickFree\ r \wedge$
$x = r\ @\ v \wedge$
$r\ setinterleaves\ ((t,\ u),\ ev\ `\ C \cup \{tick\}) \wedge$
$t \in D\ (Mprefix\ A\ P) \wedge u \in T\ (Mprefix\ B\ Q)]\!]$
$\Longrightarrow x \in D\ (\Box\ x \in\ A \cap B \cap C \rightarrow (P\ x\ [\![C\ ]\!]\ Q\ x))$
sorry

lemma *mprefix-Par-Int-distr1D1*:
$[\![A \subseteq C;\ B \subseteq C]\!]$
$\Longrightarrow D\ (Mprefix\ A\ P\ [\![C\ ]\!]\ Mprefix\ B\ Q)$
$\subseteq D\ (\Box\ x \in\ A \cap B \cap C \rightarrow (P\ x\ [\![C\ ]\!]\ Q\ x))$
sorry

lemma *mprefix-Par-Int-distr1D2*:
$[\![A \subseteq C;\ B \subseteq C]\!]$
$\Longrightarrow D\ (\Box\ x \in\ A \cap B \cap C \rightarrow (P\ x\ [\![C\ ]\!]\ Q\ x))$
$\subseteq D\ (Mprefix\ A\ P\ [\![C]\!]\ Mprefix\ B\ Q)$
sorry

lemma *mprefix-Par-Int-distr1D*:
$[\![A \subseteq C;\ B \subseteq C]\!]$
$\Longrightarrow D\ (Mprefix\ A\ P\ [\![C]\!]\ Mprefix\ B\ Q) =$
$D\ (\Box\ x \in\ A \cap B \cap C \rightarrow (P\ x\ [\![C\ ]\!]\ Q\ x))$
sorry

lemma *mprefix-Par-Int-distr1F1*:

$[\![A \subseteq C;\ B \subseteq C]\!]$
$\Longrightarrow F\ (Mprefix\ A\ P\ [\![C]\!]\ Mprefix\ B\ Q)$
$\subseteq F\ (\Box\ x \in A \cap B \cap C \rightarrow (P\ x\ [\![C\ ]\!]\ Q\ x))$
sorry

lemma *mprefix-Par-Int-distr1F2*:
$[\![A{\subseteq}C;\ B{\subseteq}C]\!]$
$\Longrightarrow F(\Box\ x \in A \cap B \cap C{\rightarrow}\ (P\ x)\ [\![C]\!]\ (Q\ x))$
$\subseteq F((\Box\ x \in A \rightarrow P\ x)[\![\ C\ ]\!]\ ((\Box\ x \in B \rightarrow Q\ x)))$
sorry

lemma *mprefix-Par-Int-distr1F*:
$[\![A \subseteq C;\ B \subseteq C]\!]$
$\Longrightarrow F\ (\Box\ x \in\ A \cap B \cap C \rightarrow (P\ x\ [\![C\ ]\!]\ Q\ x))$
$\subseteq F\ (Mprefix\ A\ P\ [\![C\ ]\!]\ Mprefix\ B\ Q)$
sorry

lemma *mprefix-Par-Int-distr1*:
$[\![A \subseteq C;\ B \subseteq C]\!]$
$\Longrightarrow (Mprefix\ A\ P\ [\![C\ ]\!]\ Mprefix\ B\ Q) = \Box\ x \in\ A \cap B \cap C \rightarrow (P\ x\ [\![C\ ]\!]\ Q\ x)$
sorry

lemma *mprefix-Par-Int-skipD*:
$D\ (Mprefix\ A\ P\ [\![B\ ]\!]\ SKIP) = D\ (\Box\ x \in\ A - B \rightarrow (P\ x\ [\![B\ ]\!]\ SKIP))$
sorry

lemma *mprefix-Par-Int-skipF1*:
$[\![X \cap A = \{\};\ tick \notin Y]\!]$
$\Longrightarrow ((X \cup Y) \cap (B \cup \{tick\}) \cup X \cap Y) \cap (A - B) = \{\}$
by (*auto*)

lemma *mprefix-Par-Int-skipF2*:
$X \cap (ev\ `\ A - ev\ `\ B) = \{\} \Longrightarrow$
$X =$
$(X - ev\ `\ A \cup (X - \{tick\})) \cap (ev\ `\ B \cup \{tick\}) \cup$
$(X - ev\ `\ A) \cap (X - \{tick\})$
by (*auto*)

lemma *mprefix-Par-Int-skipF3*:
$[\![x\ setinterleaves\ ((t, [tick]),\ ev\ `\ B \cup \{tick\});\ t \neq []]\!]$
$\Longrightarrow hd\ t \notin ev\ `\ B$
sorry

lemma *mprefix-Par-Int-skipF4*:
$[\![x \neq [];\ ev\ a = hd\ x;$
$tl\ x\ setinterleaves\ ((t, [tick]),\ ev\ `\ B \cup \{tick\});\ hd\ x \notin ev\ `\ B]\!]$
$\Longrightarrow x\ setinterleaves\ ((x, [tick]),\ ev\ `\ B \cup \{tick\})$
sorry

lemma *mprefix-Par-Int-skipF*:
$F((Mprefix\ A\ P)\ [\![B]\!]\ SKIP) = F(Mprefix\ (A - B)(\%x.(P\ x)\ [\![\ B\ ]\!]\ SKIP))$
sorry

lemma *mprefix-Par-Int-skip*:
$F\ (Mprefix\ A\ P\ [\![B]\!]\ SKIP) = F\ (\Box\ x \in\ A - B \rightarrow (P\ x\ [\![B\ ]\!]\ SKIP))$
sorry

lemma *mprefix-Par-Int-skip1*:
$F\ (Mprefix\ A\ P\ [\![B]\!]\ SKIP) = F\ (\Box\ x \in\ A - B \rightarrow (P\ x\ [\![B\ ]\!]\ SKIP))$
sorry

lemma *par-Int-skip-stop*: $(SKIP\ [\![A]\!]\ STOP) = STOP$
sorry

lemma *par-Int-skip-stop1*: $(STOP\ [\![A]\!]\ SKIP) = STOP$
sorry

lemma *Inter-skip-stop*: $(SKIP\ |||\ STOP) = STOP$
by (*simp add*: *Inter-skip2*)

lemma *Inter-stop-skip*: $(STOP\ |||\ SKIP) = STOP$
by (*simp add*: *Inter-skip1*)

lemma *prefix-Par-Int-skip1*:
$a \notin A \Longrightarrow (a \rightarrow P\ [\![A]\!]\ SKIP) = (a \rightarrow (P\ [\![A]\!]\ SKIP))$
apply (*simp add*: *write0-def*)
sorry

lemma *prefix-Par-Int-skip1a*:
$a \notin A \Longrightarrow (SKIP\ [\![A]\!]\ a \rightarrow P) = (a \rightarrow (SKIP\ [\![A]\!]\ P))$
sorry

lemma *prefix-Par-Int-skip2*:
$a \in A \Longrightarrow (a \rightarrow P\ [\![A]\!]\ SKIP) = STOP$
sorry

lemma *prefix-Par-Int-skip*:
$(a \rightarrow P\ [\![A]\!]\ SKIP) = (if\ a \in A\ then\ STOP\ else\ (a \rightarrow (P\ [\![A\ ]\!]\ SKIP)))$
by (*auto simp add*: *prefix-Par-Int-skip2 prefix-Par-Int-skip1*)

lemma *prefix-Par-Int-skip2a*:
$a \in A \Longrightarrow (SKIP\ [\![A]\!]\ a \rightarrow P) = STOP$

sorry

lemma *prefix-par-Int1*:
$[\![a \in A;\ b \in A;\ a \neq b]\!] \Longrightarrow (a \rightarrow P\ [\![A]\!]\ b \rightarrow Q) = STOP$
sorry

lemma *prefix-par-Int2*: $a{:}A ==> ((a \rightarrow P)\ [\![A]\!]\ (a \rightarrow Q)) = (a \rightarrow (P\ [\![\ A\ ]\!]\ Q))$
sorry

lemma *prefix-par*: $((a \rightarrow P)\ ||\ (a \rightarrow Q)) = (a \rightarrow (P\ ||\ Q))$
sorry

lemma *prefix-Par-Int3*:
$[\![a \in C;\ b \notin C]\!] \Longrightarrow (a \rightarrow P\ [\![C]\!]\ b \rightarrow Q) = (b \rightarrow (a \rightarrow P\ [\![C]\!]\ Q))$
sorry

lemma *prefix-Par-Int4*:
$[\![a \notin C;\ b \in C]\!] \Longrightarrow (a \rightarrow P\ [\![C]\!]\ b \rightarrow Q) = (a \rightarrow (P\ [\![C]\!]\ b \rightarrow Q))$
sorry

lemma *prefix-Inter*:
$((a \rightarrow P)\ |||\ (b \rightarrow Q)) = ((a \rightarrow (P\ |||\ (b \rightarrow Q))) \ \Box\ (b \rightarrow ((a \rightarrow P)\ |||\ Q)))$
sorry

lemma *IF-SEQ*: $((if\ b\ then\ P\ else\ Q)\ `;`\ S) = (if\ b\ then\ P\ `;`\ S\ else\ Q\ `;`\ S)$
by (*simp add*: *if-distrib*)

lemma
$a \neq b \Longrightarrow$
$(((b \rightarrow SKIP)\ [\![\{b\}]\!]\ (a \rightarrow SKIP))|||b \rightarrow SKIP) \neq$
$((b \rightarrow SKIP)\ [\![\{b\}]\!]\ ((a \rightarrow SKIP)|||(b \rightarrow SKIP)))$
sorry

lemma $[\![directed\ X;\ finite\ A]\!] \Longrightarrow (\bigsqcup P{\in}X.\ P \setminus A) = lub\ X \setminus A$
sorry

lemma *cont-imp-cont-lub*: $cont\ f \Longrightarrow contlub\ (f)$
by (*simp add*:*cont2contlub*)

lemma *dir-image2*: $[\![directed\ X;\ mono\ f]\!] \Longrightarrow directed\ (f\ `\ X)$
sorry

lemma *mono-contlub-imp-cont*: $[\![mono\ f;\ contlub\ f]\!] \Longrightarrow cont\ f$
sorry

lemma *prefix-contlub*: $cont\ f \Longrightarrow contlub\ (\lambda x.\ a \rightarrow f\ x)$
sorry

lemma *prefix-cont*: $cont\ f \Longrightarrow cont\ (\lambda x.\ a \rightarrow f\ x)$
sorry

lemma *elemDIselemHD*: $t \in D\ P \Longrightarrow tr\text{-}hide\text{-}set\ t\ (ev\ `\ A) \in D\ (P \setminus A)$
sorry

lemma $D((P \setminus \{a\}) \setminus \{b\}) \subseteq D(P \setminus (\{a\} \cup \{b\}))$
sorry

lemma $D(P \setminus (\{a\} \cup \{b\})) \subseteq D((P \setminus \{a\}) \setminus \{b\})$
sorry

lemma *hide-set-UnF1*:
$D((P \setminus \{a\}) \setminus \{b\}) = D(P \setminus (\{a\} \cup \{b\})) \Longrightarrow$
$F((P \setminus \{a\}) \setminus \{b\}) \subseteq F(P \setminus (\{a\} \cup \{b\}))$
sorry

lemma *hide-set-UnF*:
$D((P \setminus \{a\}) \setminus \{b\}) = D(P \setminus (\{a\} \cup \{b\})) \Longrightarrow$
$F(P \setminus (\{a\} \cup \{b\})) \subseteq F((P \setminus \{a\}) \setminus \{b\})$
sorry

lemma *alphabet1*:
$P \setminus A = P \Longrightarrow \forall t.\ t \in T\ P - D\ P \longrightarrow tr\text{-}hide\text{-}set\ t\ (ev\ `\ A) = t$
sorry

lemma *alphabet2*:
$P \setminus A = P \Longrightarrow \forall t.\ t \in min\text{-}elems\ (D\ P) \longrightarrow tr\text{-}hide\text{-}set\ t\ (ev\ `\ A) = t$
sorry

lemma *setinterl-Nil-tick*:
$[\![u = [] \vee u = [tick];\ r\ setinterleaves\ ((t,\ u),\ ev\ `\ A \cup \{tick\})]\!]$
$\Longrightarrow r = t \wedge (\forall x.\ x \in ev\ `\ A \longrightarrow \neg\ member\ t\ x)$
sorry

lemma *NelemAlphSpec*:
$P \setminus A = P \Longrightarrow$
$\forall t.\ t \in T\ P - D\ P \cup min\text{-}elems\ (D\ P) \longrightarrow tr\text{-}hide\text{-}set\ t\ (ev\ `\ A) = t$
sorry

lemma *NelemAlphSpec1a*:
$(P \llbracket A \rrbracket SKIP) \sqsubseteq P \Longrightarrow \forall t.\ t \in T\ P - D\ P \longrightarrow tr\text{-}hide\text{-}set\ t\ (ev\ `\ A) = t$
sorry

lemma *NelemAlphSpec1b*:
$(P \llbracket A \rrbracket SKIP) \sqsubseteq P \Longrightarrow$
$\forall t.\ t \in min\text{-}elems\ (D\ P) \longrightarrow tr\text{-}hide\text{-}set\ t\ (ev\ `\ A) = t$
sorry

lemma *NelemAlphSpec1*:
$(P \llbracket A \rrbracket SKIP) = P \Longrightarrow$
$\forall t.\ t \in T\ P - D\ P \cup min\text{-}elems\ (D\ P) \longrightarrow tr\text{-}hide\text{-}set\ t\ (ev\ `\ A) = t$
sorry

lemma *adm-eq1*:
$[\![cont\ f;\ cont\ g]\!] \Longrightarrow adm\ (\lambda x.\ f\ x = g\ x)$
by (*simp*)

lemma *NelemAlphRec*:
$[\![cont\ u;\ \bigwedge x.\ (x \llbracket A \rrbracket SKIP) = x \Longrightarrow (u\ x \llbracket A \rrbracket SKIP) = u\ x]\!]$
$\Longrightarrow (mu\ u \llbracket A \rrbracket SKIP) = mu\ u$
sorry

11 Infra-structure for Communication Primitives

lemma *read-read-sync*:
assumes *contained*: $(\bigwedge y.\ c\ y \in C)$
shows $((c\ `?`\ x \to P\ x) \llbracket C \rrbracket (c\ `?`\ x \to Q\ x)) =$
$(c\ `?`\ x \to ((P\ x) \llbracket C \rrbracket (Q\ x)))$
proof −
have *A*: *range c* ⊆ *C* **by**(*insert contained*, *auto*)
show *?thesis*
by(*auto simp*: *read-def o-def Set.Int-absorb2 mprefix-Par-Int-distr1 A*)
qed

lemmas *read-Par-Int-distr1* = *read-read-sync*

lemma *read-read-nonsync-left*:
$[\![\bigwedge y.\ c\ y \notin C;\ \bigwedge y.\ d\ y \in C]\!] \Longrightarrow$
$((c\ `?`\ x \to (P\ x))\ \llbracket C \rrbracket\ (d\ `?`\ x \to (Q\ x))) =$
$(c\ `?`\ x \to ((P\ x)\ \llbracket C \rrbracket\ (d\ `?`\ x \to (Q\ x))))$
by(*auto simp*: *read-def o-def intro*!: *mprefix-Par-Int-distr2*)

lemmas *read-Par-Int-distr2* = *read-read-nonsync-left*

lemma *read-read-nonsync-right*:

$[\![\bigwedge y.\ c\ y \notin C;\ \bigwedge y.\ d\ y \in C]\!] \Longrightarrow$
$((d\ `?`\ x \to (Q\ x))\ [\![C]\!]\ (c\ `?`\ x \to (P\ x))) =$
$(c\ `?`\ x \to ((d\ `?`\ x \to (Q\ x))\ \ [\![C]\!]\ (P\ x)\ ))$
apply(*subst sync-commute*)
apply(*auto simp*:*read-read-nonsync-left sync-commute*)
done

lemmas *read-Par-Int-distr3 = read-read-nonsync-right*

lemma *write-read-sync*:
assumes *contained*: $\bigwedge y.\ c\ y \in C$
assumes *is-construct*: *inj c*
shows $((c\ `!`\ a \to P)[\![C]\!](c\ `?`\ x \to Q\ x)) =$
$(c\ `!`\ a \to (P[\![C]\!](Q\ a)))$
proof −
have *A* : *range c* ⊆ *C* **by**(*insert contained*, *auto*)
have *B* : $\{c\ a\} \cap$ *range c* $\cap\ C = \{c\ a\}$ **by**(*insert contained*, *auto*)
show *?thesis*
apply(*simp add*: *read-def write-def o-def*)
apply(*subst mprefix-Par-Int-distr1*)
apply(*auto simp*: *A B contained is-construct mprefix-Par-Int-distr1*
Hilbert-Choice.inv-f-f mprefix-singl)
done
qed

lemmas *write-ParInt-read = write-read-sync*

lemma *read-write-sync*:
assumes *contained*: $\bigwedge y.\ c\ y \in C$
assumes *is-construct*: *inj c*
shows $((c\ `?`\ x \to P\ x)[\![C]\!](c\ `!`\ a \to Q)) =$
$(c\ `!`\ a \to ((P\ a)\ [\![C]\!]\ Q))$
apply(*subst sync-commute*)
apply(*auto simp*: *write-read-sync contained is-construct*)
apply(*simp only*: *sync-commute*)
done

lemmas *read-ParInt-write = read-write-sync*

lemma *write-read-nonsync-left*:
$[\![d\ a \notin C;\ \bigwedge y.\ c\ y \in C]\!] \Longrightarrow$
$((d\ `!`\ a \to P)[\![C]\!](c\ `?`\ x \to Q\ x)) =$
$(d\ `!`\ a \to (P[\![C]\!](c\ `?`\ x \to Q\ x)))$
apply(*simp add*: *write-def read-def o-def*)
apply(*subst mprefix-Par-Int-distr2*)
by *auto*

lemmas *write-ParInt-read2 = write-read-nonsync-left*

lemma *write0-read-nonsync-left* :
$[\![d \in C; \bigwedge y.\ c\ y \notin C]\!] \Longrightarrow$
$((d \to P)[\![C]\!](c\ `?`\ x \to Q\ x)) =$
$(c\ `?`\ x \to ((d \to P)[\![C]\!]Q\ x))$
apply(*simp add*: *read-def o-def*)
apply(*subst mprefix-singl*[*symmetric*])
apply(*subst mprefix-singl*[*symmetric*])
apply(*subst mprefix-Par-Int-distr3*)
apply *auto*
done

lemmas *prefix-ParInt-read2 = write0-read-nonsync-left*

lemma *read-write0-nonsync-left*:
$[\![d \in C; \bigwedge y.\ c\ y \notin C]\!] \Longrightarrow$
$((c\ `?`\ x \to Q\ x)[\![C]\!](d \to P)) =$
$(c\ `?`\ x \to (Q\ x[\![C]\!](d \to P)))$
by(*subst sync-commute*, *auto simp*: *prefix-ParInt-read2 sync-commute*)

lemma *write0-write-nonsync-right*:
$[\![d\ a \notin C;\ c \in C]\!] \Longrightarrow$
$((c \to Q)[\![C]\!](d\ `!`\ a \to P)) =$
$(d\ `!`\ a \to ((c \to Q)[\![C]\!]P))$
apply(*simp add*: *write-def*)
apply(*subst mprefix-singl*[*symmetric*])
apply(*subst mprefix-singl*[*symmetric*])
apply(*auto simp*: *mprefix-Par-Int-distr3*)
done

lemmas *prefix-ParInt-write2 = write0-write-nonsync-right*

lemma *write-write0-nonsync-left*:
$[\![d\ a \notin C;\ c \in C]\!] \Longrightarrow$
$((d\ `!`\ a \to P)[\![C]\!](c \to Q)) =$
$(d\ `!`\ a \to (P[\![C]\!](c \to Q)))$
apply(*subst sync-commute*)
apply(*auto simp*: *prefix-ParInt-write2 sync-commute*)
done

lemmas *write-ParInt-prefix2 = write-write0-nonsync-left*

lemma *write0-write0-sync* :
$c \in C \Longrightarrow ((c \to P)[\![C]\!](c \to Q)) = (c \to (P[\![C]\!]Q))$
sorry

lemmas *sync-rules* =
read-read-sync read-read-nonsync-left read-read-nonsync-right
write-read-sync read-write-sync write-read-nonsync-left
write0-read-nonsync-left read-write0-nonsync-left
write0-write-nonsync-right write-write0-nonsync-left
write0-write0-sync

lemma *no-hide-read-1*:
$(\bigwedge y.\ c\ y \notin B) \Longrightarrow$
$((c\ \text{`?`}\ x \rightarrow (P\ x)) \setminus B) = (c\ \text{`?`}\ x \rightarrow ((P\ x) \setminus B))$
apply(*simp add*: *read-def o-def*)
apply(*subst hide-mprefix-distr*)
apply *auto*
done

lemmas *hide-read-distr1* = *no-hide-read-1*

lemma *no-hide-write*:
$(\bigwedge y.\ c\ y \notin B) \Longrightarrow ((c\ \text{`!`}\ a \rightarrow P) \setminus B) = (c\ \text{`!`}\ a \rightarrow (P \setminus B))$
apply(*simp add*: *write-def*)
apply(*subst hide-mprefix-distr*)
apply *auto*
done

lemmas *hide-write-distr1* = *no-hide-write*

lemma *hide-write*:
$(c\ a) \in B \Longrightarrow ((c\ \text{`!`}\ a \rightarrow P) \setminus B) = (P \setminus B)$
apply(*auto simp*: *write-def*)
sorry

lemmas *hide-write-distr2* = *hide-write*

lemma *hide-write0*:
$c \in B \Longrightarrow ((c \rightarrow P) \setminus B) = (P \setminus B)$
sorry

lemmas *hide-rules* = *no-hide-read-1 no-hide-write hide-write hide-write0*

lemma *mono-read-ref*:
$(\bigwedge x.\ P\ x \sqsubseteq Q\ x) \Longrightarrow\ (c\ \text{`?`}\ x \rightarrow (P\ x)) \sqsubseteq (c\ \text{`?`}\ x \rightarrow (Q\ x))$
by(*simp add*: *read-def mono-mprefix-ref*)

lemma *mono-write-ref*:
$(P \sqsubseteq Q) \Longrightarrow\ (c\ \text{`!`}\ x \rightarrow P) \sqsubseteq (c\ \text{`!`}\ x \rightarrow Q)$
by(*simp add*: *write-def mono-mprefix-ref*)

lemma *mono-write0-ref*:
$(P \sqsubseteq Q) \Longrightarrow (c \rightarrow P) \sqsubseteq (c \rightarrow Q)$
by(*simp add*: *write0-def mono-mprefix-ref*)

lemmas *mono-rules* = *mono-read-ref mono-write-ref mono-write0-ref*

lemmas *Det-commute* = *det-commute*
lemmas *non-det-id* = *ndet-id*
lemmas *Ndet-commute* = *ndet-commute*
lemmas *non-det-bot* = *ndet-bot*

12 Operational Semantics

datatype $'\alpha$ *sevent* = *sevent* $'\alpha$ *event* | τ

inductive *op-sem* :: [$'\alpha$ *process*, $'\alpha$ *sevent*, $'\alpha$ *process*] => *bool*
(- $\longrightarrow$ - - [*0*,*0*,*60*] *60*)
where *refl* : $P \longrightarrow \tau\ P$
|*skip* : *SKIP* $\longrightarrow$ (*sevent*(*tick*)) *Bot*
|*mpref* : $y \in A \Longrightarrow (\Box\ x{\in}A \rightarrow P\ x\ ) \longrightarrow (sevent(ev\ y))\ (P\ y)$
|*refine* : $P \sqsubseteq Q \Longrightarrow Q \longrightarrow a\ Q' \Longrightarrow P \longrightarrow a\ Q'$

end

13 Example: Refinement Example with Buffer over infinite Alphabet

theory *CopyBuffer*
imports *../src/CSP*
begin

14 Defining the Copy-Buffer Example

datatype *'a channel = left 'a | right 'a | mid 'a | ack*

definition *SYN* :: (*'a channel*) *set*
where *SYN* ≡ (*range mid*) ∪ {*ack*}

definition *COPY* :: (*'a channel*) *process*
where *COPY* ≡ (*μ COPY. left‘?‘x → right‘!‘x → COPY*)

definition *SEND* :: (*'a channel*) *process*
where *SEND* ≡ (*μ SEND. left‘?‘x → mid‘!‘x → ack → SEND*)

definition *REC* :: (*'a channel*) *process*
where *REC* ≡ (*μ REC. mid‘?‘x → right‘!‘x → ack → REC*)

definition *SYSTEM* :: (*'a channel*) *process*
where *SYSTEM* ≡ ((*SEND* ⟦ *SYN* ⟧ *REC*) \ *SYN*)

15 The Standard Proof

15.1 Channels and Synchronization Sets

First part: abstract properties for these events to SYN. This kind of stuff could be automated easily by some extra-syntax for channels and SYN-sets.

lemma [*simp*]: *left x ∉ SYN*
by(*auto simp*: *SYN-def*)

lemma [*simp*]: *right x ∉ SYN*
by(*auto simp*: *SYN-def*)

lemma [*simp*]: *ack ∈ SYN*
by(*auto simp*: *SYN-def*)

lemma [*simp*]: *mid x ∈ SYN*
by(*auto simp*: *SYN-def*)

lemma [*simp*]: *inj mid*
by(*auto simp*: *inj-on-def*)

15.2 Definitions by Recursors

Second part: Derive recursive process equations, which are easier to handle in proofs. This part IS actually automated if we could reuse the fixrec-syntax below.

lemma *COPY-rec*:
(*COPY*::*'a channel process*) = (*left‘?‘x → right‘!‘x → COPY*)
by(*simp add*: *COPY-def*,*rule trans*, *rule fix-eq*, *simp*)

lemma *SEND-rec*:
SEND = (*left*‘?‘*x* → *mid*‘!‘*x* → *ack* → *SEND*)
by(*simp add*: *SEND-def*,*rule trans*, *rule fix-eq*, *simp*)

lemma *REC-rec*:
REC = (*mid*‘?‘*x* → *right*‘!‘*x* → *ack* → *REC*)
by(*simp add*: *REC-def*,*rule trans*, *rule fix-eq*, *simp*)

15.3 A Refinement Proof

Third part: No comes the proof by fixpoint induction. Not too bad in automation considering what is inferred, but wouldn't scale for large examples.

lemma *impl-refines-spec* : (*COPY*::*'a channel process*) ⊑ *SYSTEM*
apply(*simp add*: *SYSTEM-def COPY-def*)
apply(*rule fix-ind*, *simp-all*)
apply(*subst SEND-rec*, *subst REC-rec*)
apply(*simp add*: *sync-rules hide-rules*)
apply(*simp add*: *mono-rules*)
done

lemma *spec-refines-impl* :
assumes *fin*: *finite* (*SYN*::(*'a channel*)*set*)
shows *SYSTEM* ⊑ (*COPY*::*'a channel process*)
apply(*unfold SYSTEM-def SEND-def*)
apply(*rule fix-ind*, *simp-all add*: *fin*)
apply(*subst COPY-rec*, *subst REC-rec*)
apply(*simp add*: *sync-rules hide-rules*)
apply(*simp add*: *mono-rules*)
done

Note that this was actually proven for the Process ordering, not the refinement ordering. But the former implies the latter. And due o anti-symmetrie, equality follows for the case of finite alphabets ...

lemma *spec-equal-impl* :
assumes *fin*: *finite* (*SYN*::(*'a channel*)*set*)
shows *SYSTEM* = (*COPY*::*'a channel process*)
apply(*rule Porder.po-class.below-antisym*)
apply(*rule spec-refines-impl*[*OF fin*])
apply(*rule impl-refines-spec*)
done

16 An Alternative Approach: Using the fixrec-Package

16.1 Channels and Synchronisation Sets

As before.

16.2 Process Definitions via fixrec-Package

fixrec
  $COPY'$ :: $('a\ channel)\ process$
and
  $SEND'$ :: $('a\ channel)\ process$
and
  REC' :: $('a\ channel)\ process$
where
  $COPY'$-$rec[simp\ del]$: $COPY' = (left`?`x \rightarrow right`!`x \rightarrow COPY')$
| $SEND'$-$rec[simp\ del]$: $SEND' = (left`?`x \rightarrow mid`!`x \rightarrow ack \rightarrow SEND')$
| REC'-$rec[simp\ del]$: $REC' = (mid`?`x \rightarrow right`!`x \rightarrow ack \rightarrow REC')$

find-theorems $name{:}COPY$

definition $SYSTEM'$ :: $('a\ channel)\ process$
where $SYSTEM' \equiv ((SEND' \llbracket\ SYN\ \rrbracket REC') \setminus SYN)$

16.3 Another Refinement Proof on fixrec-infrastructure

Third part: No comes the proof by fixpoint induction. Not too bad in automation considering what is inferred, but wouldn't scale for large examples.

lemma $impl\text{-}refines\text{-}spec'$: $(COPY'{::}'a\ channel\ process) \sqsubseteq SYSTEM'$
  apply$(unfold\ SYSTEM'\text{-}def)$
  apply$(rule\text{-}tac\ P{=}\lambda\ a\ b\ c.\ a \sqsubseteq\ ?P'$ **in** $COPY'\text{-}SEND'\text{-}REC'.induct)$
  apply$(simp\text{-}all\ add{:}\ split\text{-}def)$
  apply$(subst\ SEND'\text{-}rec)$ **apply**$(\ subst\ REC'\text{-}rec)$
  apply$(simp\ add{:}\ sync\text{-}rules\ hide\text{-}rules)$
  apply$(simp\ only{:}\ mono\text{-}rules)$
done

lemma $spec\text{-}refines\text{-}impl'$:
assumes $fin{:}\ \ finite\ (SYN{::}('a\ channel)set)$
shows $SYSTEM' \sqsubseteq (COPY'{::}'a\ channel\ process)$
  apply$(unfold\ SYSTEM'\text{-}def,unfold\ SEND'\text{-}def)$
  apply$(rule\ fix\text{-}ind)$
  apply$(simp\text{-}all\ add{:}\ \ fin\ split\text{-}def)$
  apply$(subst\ COPY'\text{-}rec,subst\ REC'\text{-}rec)$
  apply$(simp\ add{:}\ sync\text{-}rules\ hide\text{-}rules)$
  apply$(simp\ add{:}\ mono\text{-}rules)$
done

lemma $spec\text{-}equal\text{-}impl'$:
assumes $fin{:}\ \ finite\ (SYN{::}('a\ channel)set)$
shows $SYSTEM' = (COPY'{::}'a\ channel\ process)$
apply$(rule\ Porder.po\text{-}class.below\text{-}antisym)$
apply$(rule\ spec\text{-}refines\text{-}impl'[OF\ fin])$
apply$(rule\ impl\text{-}refines\text{-}spec')$
done

end

References

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

[2] H. Tej and B. Wolff. A corrected failure divergence model for CSP in Isabelle/HOL. In J. S. Fitzgerald, C. B. Jones, and P. Lucas, editors, *Formal Methods Europe (FME)*, volume 1313 of *Lecture Notes in Computer Science*, pages 318–337, Heidelberg, 1997. Springer-Verlag.