HOL-CSP Version 2.0

Burkhart Wolff

April 2, 2010

Contents

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

theory *Process*
imports *HOLCF*
begin

ML⟪ *quick-and-dirty*:=*true*⟫

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.

defaultsort *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 $'\alpha\ event = ev\ '\alpha \mid tick$

```
types   'α trace = ('α event) list
```

We chose as standard ordering on traces the prefix ordxering.

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

definition le-list-def : s ≤ t ⟷ (∃ r. s @ r = t)

definition less-list-def: (s::'a list) < t ⟷ s ≤ t ∧ s ≠ t

instance


proof
  fix x y ::'α list
  show (x < y) = (x ≤ y ∧ ¬ y ≤ x)  by(auto simp: le-list-def less-list-def)
 next
  fix x ::'α list
  show x ≤ x by(simp add: le-list-def)
 next
  fix x y z :: 'α list
  assume A:x ≤ y and B:y ≤ z thus x ≤ 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 :: 'α list
  assume A:x ≤ y and B:y ≤ 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]: [] ≤ s
by(induct s, simp-all, auto simp: le-list-def)

lemma nil-le2[simp]: s ≤ [] = (s = [])
by(induct s, auto simp:le-list-def)

lemma nil-less[simp]: ¬ 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)
```

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.

constdefs
tickFree :: *'α trace* ⇒ *bool*
tickFree s ≡ ¬ *tick mem s*
front-tickFree :: *'α trace* ⇒ *bool*
front-tickFree s ≡ (*s* =[] ∨ *tickFree*(*tl*(*rev s*)))

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

lemma *tickFree-Cons* [*simp*]: *tickFree* (*a* # *t*) = (*a* ≠ *tick* ∧ *tickFree t*)
by(*subst HOL.neq-commute*, *simp add*: *tickFree-def*)

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

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

lemma *non-tickFree-implies-nonMt*: ¬ *tickFree s* ⟹ *s* ≠ []
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 mem-iff*)

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 mem-iff*,*safe*)
apply(*erule contrapos-np*, *simp*,(*erule rev-mp*)+)
apply(*rule-tac xs*=*s* **in** *List.rev-induct*,*simp-all*)
done

lemma *list-nonMt-append*:
s ≠ [] ⟹ ∃ *a t*. *s* = *t* @ [*a*]
by(*erule rev-mp*,*induct s*,*simp-all*,*case-tac s* = [],*auto*)

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 mem-iff)

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

```
types
  'α refusal     = ('α event) set
  'α failure     = 'α trace × 'α refusal
  'α divergence  = 'α trace set
  'α process-pre = 'α failure set × 'α divergence

constdefs
  FAILURES       :: 'α process-pre ⇒ ('α failure set)
  FAILURES P    ≡ fst P

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

  DIVERGENCES   :: 'α process-pre ⇒ 'α divergence
  DIVERGENCES P ≡ snd P

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

1.3 The Process Type Invariant

```
constdefs
  is-process    :: 'α process-pre ⇒ bool
  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)
∧
      (∀ s X Y. (s,X) ∈ FAILURES P ∧
      (∀ c.     c ∈ Y ⟶ ((s@[c],{})∉FAILURES P)) ⟶
                                    (s,X ∪ Y)∈FAILURES P) ∧
     (∀ s X.  (s@[tick],{}) : FAILURES P ⟶ (s,X−{tick}) ∈ FAILURES P) ∧
      (∀ s t.  s ∈ DIVERGENCES P ∧ tickFree s ∧ front-tickFree t
                                  ⟶ s@t ∈ DIVERGENCES P)  ∧
      (∀ s X. s ∈ DIVERGENCES P ⟶ (s,X) ∈ FAILURES P) ∧
      (∀ s. s @ [tick] : DIVERGENCES P ⟶ s ∈ DIVERGENCES P)
```

```
lemma is-process-spec:
 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) ∧
      (∀ s X Y.(s,X) ∈ FAILURES P ∧
     (∀ c. c ∈  Y ⟶ ((s@[c],{}) ∉  FAILURES P)) ⟶(s,X ∪ Y) ∈  FAILURES
P) ∧
      (∀ s X. (s@[tick],{}) ∈  FAILURES P ⟶ (s,X − {tick}) ∈  FAILURES
P) ∧
      (∀ s t. s ∉ DIVERGENCES P ∨ ¬tickFree s ∨ ¬front-tickFree t
                              ∨ s @ t ∈ DIVERGENCES P) ∧
      (∀ s X. s ∉ DIVERGENCES P ∨ (s,X) ∈  FAILURES P) ∧
      (∀ s. s @ [tick] ∉  DIVERGENCES P ∨ s ∈  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 ∧ 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*:
$\mathit{is\text{-}process}\ P \Longrightarrow \forall s\ X\ Y.\ (s,\ Y) \notin \mathit{FAILURES}\ P \lor \neg\ X \subseteq\ Y \lor (s,\ X) \in \mathit{FAILURES}\ P$
by(*simp only*: *is-process-spec*, *simp*)

lemma *is-process4-S*:
$[\![\mathit{is\text{-}process}\ P;\ (s,\ Y) \in \mathit{FAILURES}\ P;\ X \subseteq\ Y]\!] \Longrightarrow (s,\ X) \in \mathit{FAILURES}\ P$
by(*drule is-process4*, *auto*)

lemma *is-process4-S1*:
$[\![\mathit{is\text{-}process}\ P;\ x \in \mathit{FAILURES}\ P;\ X \subseteq \mathit{snd}\ x]\!] \Longrightarrow (\mathit{fst}\ x,\ X) \in \mathit{FAILURES}\ P$
by(*drule is-process4-S*, *auto*)

lemma *is-process5*:
$\mathit{is\text{-}process}\ P \Longrightarrow$
$\quad \forall \mathit{sa}\ X\ Y.$
$\qquad (\mathit{sa},\ X) \in \mathit{FAILURES}\ P \land (\forall c.\ c \in Y \longrightarrow (\mathit{sa}\ @\ [c],\ \{\}) \notin \mathit{FAILURES}\ P)$
$\longrightarrow$
$\qquad (\mathit{sa},\ X \cup Y) \in \mathit{FAILURES}\ P$
by(*drule is-process-spec*[*THEN iffD1*],*metis*)

lemma *is-process5-S*:
$[\![\mathit{is\text{-}process}\ P;\ (\mathit{sa},\ X) \in \mathit{FAILURES}\ P;$
$\ \forall c.\ c \in Y \longrightarrow (\mathit{sa}\ @\ [c],\ \{\}) \notin \mathit{FAILURES}\ P]\!]$
$\ \Longrightarrow (\mathit{sa},\ X \cup Y) \in \mathit{FAILURES}\ P$
by(*drule is-process5*, *metis*)

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

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

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

lemma *is-process7*:
$\mathit{is\text{-}process}\ P \Longrightarrow$
$\forall\ s\ t.\ s \notin \mathit{DIVERGENCES}\ P \lor$
$\qquad \neg\ \mathit{tickFree}\ s \lor$
$\qquad \neg\ \mathit{front\text{-}tickFree}\ t \lor$
$\qquad s\ @\ t \in \mathit{DIVERGENCES}\ P$

by(*drule is-process-spec*[*THEN iffD1*], *metis*)

lemma *is-process7-S*:
⟦ *is-process P*;*s* : *DIVERGENCES P*;*tickFree s*;*front-tickFree t*⟧
⟹ *s* @ *t* ∈ *DIVERGENCES P*
by(*drule is-process7*, *metis*)

lemma *is-process8*:
is-process P ⟹ ∀ *s X*. *s* ∉ *DIVERGENCES P* ∨ (*s*,*X*) ∈ *FAILURES P*
by(*drule is-process-spec*[*THEN iffD1*], *metis*)

lemma *is-process8-S*:
⟦ *is-process P*; *s* ∈ *DIVERGENCES P* ⟧ ⟹ (*s*,*X*) ∈ *FAILURES P*
by(*drule is-process8*, *metis*)

lemma *is-process9*:
is-process P ⟹ ∀ *s*. *s*@[*tick*] ∉ *DIVERGENCES P* ∨ *s* ∈ *DIVERGENCES P*
by(*drule is-process-spec*[*THEN iffD1*], *metis*)

lemma *is-process9-S*:
⟦ *is-process P*;*s*@[*tick*] ∈ *DIVERGENCES P* ⟧ ⟹ *s* ∈ *DIVERGENCES P*
by(*drule is-process9*, *metis*)

lemma *Failures-implies-Traces*:
⟦*is-process P*; (*s*, *X*) ∈ *FAILURES P*⟧ ⟹ *s* ∈ *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

constdefs
  F   :: ′α *process* ⇒ (′α *failure set*)
  F P ≡ *FAILURES* (*Rep-Process P*)
  T   :: ′α *process* ⇒ (′α *trace set*)
  T P ≡ *TRACES* (*Rep-Process P*)
  D   :: ′α *process* ⇒ ′α *divergence*
  D P ≡ *DIVERGENCES* (*Rep-Process P*)
  R   :: ′α *process* ⇒ (′α *refusal set*)
  R P ≡ *REFUSALS* (*Rep-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* ∧ *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*:

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

theorem *process-charn*:
$([],\ \{\}) \in F\ P \wedge$
$(\forall s\ X.\ (s,\ X) \in F\ P \longrightarrow \textit{front-tickFree}\ s) \wedge$
$(\forall s\ t.\ (s\ @\ t,\ \{\}) \notin F\ P \vee (s,\ \{\}) \in F\ P) \wedge$
$(\forall s\ X\ Y.\ (s,\ Y) \notin F\ P \vee \neg\ X \subseteq Y \vee (s,\ X) \in F\ P) \wedge$
$(\forall s\ X\ Y.\ (s,\ X) \in F\ P \wedge (\forall c.\ c \in Y \longrightarrow (s\ @\ [c],\ \{\}) \notin F\ P) \longrightarrow$
$(s,\ X \cup Y) \in F\ P) \wedge$
$(\forall s\ X.\ (s\ @\ [\textit{tick}],\ \{\}) \in F\ P \longrightarrow (s,\ X - \{\textit{tick}\}) \in F\ P) \wedge$
$(\forall s\ t.\ s \notin D\ P \vee \neg\ \textit{tickFree}\ s \vee \neg\ \textit{front-tickFree}\ t \vee s\ @\ t \in D\ P) \wedge$
$(\forall s\ X.\ s \notin D\ P \vee (s,\ X) \in F\ P) \wedge (\forall s.\ s\ @\ [\textit{tick}] \notin D\ P \vee s \in D\ P)$
proof −
have A : $!!P.\ (\forall s\ t.\ (s\ @\ t,\ \{\}) \notin F\ P \vee\ (s,\ \{\}) \in F\ P) =$
$(\forall s\ t.\ (s\ @\ t,\ \{\}) \in F\ P \longrightarrow (s,\ \{\}) \in F\ P)$
by *metis*
have B : $!!P.\ (\forall s\ X\ Y.\ (s,\ Y) \notin F\ P \vee \neg\ X \subseteq Y \vee (s,\ X) \in F\ P)\ =$
$(\forall s\ X\ Y.\ (s,\ Y) \in F\ P \wedge X \subseteq Y \longrightarrow (s,\ X) \in F\ P)$
by *metis*
have C : $!!P.\ (\forall s\ t.\ s \notin D\ P \vee \neg\ \textit{tickFree}\ s \vee$
$\neg\ \textit{front-tickFree}\ t \vee s\ @\ t \in D\ P)\ =$
$(\forall s\ t.\ s \in D\ P \wedge \textit{tickFree}\ s \wedge \textit{front-tickFree}\ t \longrightarrow s\ @\ t \in D\ P)$
by *metis*
have D: $!!P.\ (\forall s\ X.\ s \notin D\ P \vee (s,\ X) \in F\ P) = (\forall s\ X.\ s \in D\ P \longrightarrow (s,\ X) \in F\ P)$
by *metis*
have E:$!!P.\ (\forall s.\ s\ @\ [\textit{tick}] \notin D\ P \vee s \in D\ P) =$
$(\forall s.\ s\ @\ [\textit{tick}] \in D\ P \longrightarrow s \in D\ P)$
by *metis*
show *?thesis*
apply(*simp only*: $A\ B\ C\ D\ E$)
apply(*rule is-processT*)
done
qed

split of `is_processT`:

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

lemma *is-processT2*:
∀ *s X*. (*s*, *X*) ∈ *F P* ⟶ *front-tickFree s*
by(*simp add:process-charn*)

lemma *is-processT2-TR* : ∀ *s*. *s* ∈ *T P* ⟶ *front-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*) ∈ *F P*; *s* ≠ []⟧ ⟹ ¬ *tick mem 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* :
∀ *s t*. (*s* @ *t*, {}) ∈ *F P* ⟶ (*s*, {}) ∈ *F P*
by(*simp only*: *process-charn HOL.nnf-simps*(*3*), *simp*)

lemma *is-processT3-S-pref* :
⟦(*t*, {}) ∈ *F P*; *s* ≤ *t*⟧ ⟹ (*s*, {}) ∈ *F P*
apply(*simp only*: *le-list-def*, *safe*)
apply(*erule is-processT3*[*rule-format*])
done

lemma *is-processT4* :
∀ *s X Y*. (*s*, *Y*) ∈ *F P* ∧ *X* ⊆ *Y* ⟶ (*s*, *X*) ∈ *F P*
by(*insert process-charn* [*of P*], *metis*)

lemma *is-processT4-S1* :
⟦*x* ∈ *F P*; *X* ⊆ *snd x*⟧ ⟹ (*fst x*, *X*) ∈ *F P*
apply(*rule-tac Y* = *snd x* **in** *is-processT4*[*rule-format*])
apply(*simp add*: *surjective-pairing*[*symmetric*])
done

lemma *is-processT5*:
∀ *s X Y*.(*s*,*X*) ∈ *F P* ∧ (∀ *c*. *c*∈*Y* ⟶ (*s*@[*c*],{}) ∉ *F P*) ⟶ (*s*,*X*∪*Y*)∈*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 : $\{c\} = (\{\}\ \mathit{Un}\ \{c\})$ **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*:
$\forall\, s\ X.\ (s\ @\ [\mathit{tick}],\ \{\}) \in F\ P \longrightarrow (s,\ X - \{\mathit{tick}\}) \in F\ P$
by(*simp add*: *process-charn*)

lemma *is-processT7*:
$\forall s\ t.\ s \in D\ P \land \mathit{tickFree}\ s \land \mathit{front\text{-}tickFree}\ t \longrightarrow s\ @\ t \in D\ P$
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*: $\mathit{fst}\ s \in D\ P \Longrightarrow s \in F\ P$
apply(*subst surjective-pairing*)
apply(*rule is-processT8-S*, *simp*)
done

lemma *is-processT9*:
$\forall s.\ s\ @\ [\mathit{tick}] \in D\ P \longrightarrow s \in D\ P$
by(*insert process-charn*[*of P*], *metis*)

lemma *is-processT9-S-swap*: $s \notin D\ P \Longrightarrow s\ @\ [\mathit{tick}] \notin D\ P$
by(*erule contrapos-nn*,*simp add*: *is-processT9*[*rule-format*])

1.5 Some Consequences of the Process Characterization

lemma *no-Trace-implies-no-Failure*:
$s \notin T\ P \Longrightarrow (s,\ \{\}) \notin F\ P$
by(*simp add*: *T-def TRACES-def F-def*)

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

lemma *T-def-spec*:
$T\ P = \{\mathit{tr}.\ ?\ a.\ a : F\ P\ \&\ \mathit{tr} = \mathit{fst}\ a\}$
by(*simp add*: *T-def TRACES-def F-def*)

lemma *F-T*:
$(s,\ X) \in F\ P \Longrightarrow s \in T\ P$
by(*simp add*: *T-def-spec split-def*, *metis*)

lemma *F-T1*:
$a \in F\ P \Longrightarrow \mathit{fst}\ a \in T\ P$
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 ∉ X;(s @ [tick], {}) ∈ F P ⟧ ⟹ (s::'a event list, X) ∈ 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 ∈ D P} ⊆ F P
by(auto intro:is-processT8-Pair)

lemma append-T-imp-tickFree:
⟦t @ s ∈ T P; s ≠ []⟧ ⟹ tickFree t
by(frule is-processT2-TR[rule-format],
  simp add: front-tickFree-def tickFree-rev)


lemma F-subset-imp-T-subset:
F P ⊆ F Q ⟹ T P ⊆ 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 ∉ X; [tick] ∈ T P⟧ ⟹ ([], X) ∈ F P
by(erule is-processT6-S1, simp add: T-F-spec)

lemma is-processT9-tick:
⟦[tick] ∈ D P; front-tickFree s⟧ ⟹ s ∈ 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 ∈ T P; ¬ tickFree t⟧ ⟹ ∃ 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* ...

constdefs $min\text{-}elems \; :: ('s::ord)\ set \Rightarrow 's\ set$
$min\text{-}elems\ X \equiv \{s \in X.\ \forall t.\ t \in X \longrightarrow \neg\ (t < s)\}$

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

constdefs $Ra \; :: ['\alpha\ process,\ '\alpha\ trace] \Rightarrow ('\alpha\ refusal\ set)$
$Ra\ P\ s \equiv \{X.\ (s,\ X) \in F\ P\}$

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*) *sq-ord*
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*:
$\llbracket\ P \sqsubseteq Q;\ s \notin D\ P \rrbracket \Longrightarrow (s,X) \in F\ Q = ((s,X) \in F\ P)$
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*:
$\llbracket\ P \sqsubseteq Q;\ s \notin D\ P \rrbracket \Longrightarrow\ s \in T\ Q = (s \in T\ P)$
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

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

lemma *min-elems-le-self*[*simp*] : $(min\text{-}elems\ A) \subseteq A$
by(*auto simp*: *min-elems-def*)

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

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 ⊑ Q and B:Q ⊑ 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 ::'α process
  assume A: P ⊑ Q and B: Q ⊑ R thus P ⊑ R
  proof −
    have C : D R ⊆ D P
          by(insert A[THEN le-approx1] B[THEN le-approx1], auto)
    have D : ∀ s. s ∉ D P ⟶ {X. (s, X) ∈ F P} = {X. (s, X) ∈ F R}
          apply(rule allI, rule impI, rule set-ext, 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) ⊆ 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 ∈ 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` : $[\![chain\ ?Y;\ ?i \le ?j]\!] \Longrightarrow ?Y\ ?i \sqsubseteq ?Y\ ?j$
- `Porder.directed_chain` : $chain\ ?S \Longrightarrow directed\ (range\ ?S)$
- `Porder.directed_def` :
  $directed\ ?S = ((\exists x.\ x \in ?S) \land (\forall x{\in}?S.\ \forall y{\in}?S.\ \exists z{\in}?S.\ x \sqsubseteq z \land y \sqsubseteq z))$

- `Porder.directedD1` : $\mathit{directed}\ ?S \Longrightarrow \exists\, z.\ z \in ?S$

- `Porder.directedD2` :
  $[\![\mathit{directed}\ ?S;\ ?x \in ?S;\ ?y \in ?S]\!] \Longrightarrow \exists\, z{\in}?S.\ ?x \sqsubseteq z \wedge ?y \sqsubseteq z$

- `Porder.directedI` : $[\![\exists\, z.\ z \in ?S;\ \bigwedge x\, y.\ [\![x \in ?S;\ y \in ?S]\!] \Longrightarrow \exists\, z{\in}?S.\ x \sqsubseteq z \wedge y \sqsubseteq z]\!] \Longrightarrow \mathit{directed}\ ?S$

- `Porder.is_ubD` : $[\![?S <|\ ?u;\ ?x \in ?S]\!] \Longrightarrow ?x \sqsubseteq ?u$

- `Porder.ub_rangeI` :
  $(\bigwedge i.\ ?S\ i \sqsubseteq ?x) \Longrightarrow \mathit{range}\ ?S <|\ ?x$

- `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` : $[\![?S <<|\ ?x;\ ?S <|\ ?u]\!] \Longrightarrow ?x \sqsubseteq ?u$

- `Porder.is_lub_range_shift`:
  $\mathit{chain}\ ?S \Longrightarrow \mathit{range}\ (\lambda i.\ ?S\ (i + ?j)) <<|\ ?x = \mathit{range}\ ?S <<|\ ?x$

- `Porder.is_ub_lub`: $\mathit{range}\ ?S <<|\ ?x \Longrightarrow ?S\ ?i \sqsubseteq ?x$

- `Porder.thelubI`: $?M <<|\ ?l \Longrightarrow \mathit{lub}\ ?M = ?l$

- `Porder.unique_lub`:$[\![?S <<|\ ?x;\ ?S <<|\ ?y]\!] \Longrightarrow ?x = ?y$

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

lemma *min-elems2*:
$[|s\ \tilde{}{:}\ D\ P\ ;s\ @\ [c]\ :\ D\ P\ ;\ P << S;\ Q << S|]\ \ ==> (s\ @\ [c],\{\}){:}\ F\ Q$
sorry

lemma *ND-F-dir2*:
$[|s\ \tilde{}{:}\ D\ P\ ;\ (s,\{\})\ :\ F\ P\ ;\ P << S;\ Q << S|]\ ==> (s,\{\})\ :\ F\ Q$
sorry

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 (⋂ a::nat. F (S a), ⋂ a::nat. D (S a))
        by(auto simp: DIVERGENCES-def FAILURES-def is-processT)
next
  fix s::'a trace fix X::'a event set
  assume (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 => bool
  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 Un Y) ∈ FAILURES (⋂ a::nat. F (S a), ⋂ a::nat. D (S a))
        apply(insert Porder.directed-chain[OF chain])
      apply(insert A B, simp add: DIVERGENCES-def FAILURES-def directed-def)
        apply auto
        apply(case-tac ! x. x : (range S) −−> (s, X Un Y) : F x,auto)
        apply(case-tac Y={}, auto)
        apply(erule-tac x=x and P=λ x. x ∈ Y ⟶ ?Q x in allE,auto)
        apply(erule-tac x=a and P = λ a. (s, X) ∈ F (S a) in all-dupE, auto)
        apply(erule-tac x=xa and P = λ a. (s, X) ∈ F (S a) in all-dupE, auto)
        apply(erule-tac x=aa and P = λ a. (s, X) ∈ F (S a) in allE)
        apply(erule-tac x=a in allE)
        apply(erule-tac x=aa in allE)
        apply auto
        apply(erule contrapos-np)back
        apply(frule NF-ND)back

        apply(rule is-processT5[rule-format],auto)
prefer 2
        apply(erule contrapos-np)back
        apply(rule ND-F-dir2) apply assumption
prefer 2 apply assumption apply simp-all
```

```
apply(simp-all add: NF-ND ND-F-dir2)


         apply(case-tac a = aa, simp)




sorry

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 ⟹ 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 ⟹ 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 ⟹ 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

instance
  process :: (type) cpo
proof
  fix S ::nat ⇒ 'α process
  assume C:chain S thus ∃ x. range S <<| x
  proof −
     have lim-proc-is-ub :range S <| lim-proc (range S)
           apply(insert C, simp add: is-ub-def le-approx-def)
           apply(rule allI, rule impI)
           apply(simp add: F-LUB D-LUB T-LUB Ra-def)
           apply(rule conjI, blast)
           apply(rule conjI)
find-theorems chain -

sorry






     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)
           apply(auto simp: is-ub-def C D-LUB F-LUB Ra-def INTER-def)
           apply(erule-tac x=S x in allE, simp add: le-approx2)
            apply(erule-tac x=S x in all-dupE, erule-tac x=S xb in allE,simp
add: le-approx2)
sorry

     have lim-proc-is-lub3:
           ∀ u. range S <| u ⟶  min-elems (D (lim-proc (range S))) ⊆ T u
           apply(auto simp: is-ub-def C D-LUB F-LUB Ra-def INTER-def)
           apply(insert C[THEN Porder.directed-chain])
           apply(auto simp: min-elems-def directed-def)
thm tickFree-implies-front-tickFree
           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 simp: front-tickFree-Nil
                   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)
    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)
    apply(rule D-imp-front-tickFree, simp add: D-def DIVERGENCES-def)
    apply(erule contrapos-np,
          rule is-processT2[rule-format],
          simp add: F-def FAILURES-def)
    apply(simp add: min-elems-def front-tickFree-charn,safe)
    apply(auto simp: Nil-elem-T nil-less2)
    done
  qed
qed
```

1.7 Process Refinement is a Partial Ordering

The following type instantiation declares the refinement order $\_ \leq \_$ 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* : $P \leq Q \equiv D\ Q \subseteq D\ P \wedge F\ Q \subseteq F\ P$

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 $P\ Q$ ::$'\alpha$ *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
fix P ::$'\alpha$ *process*
show $P \leq P$ **by**(*simp add*: *le-ref-def*)
next
fix $P\ Q\ R$ :: $'\alpha$ *process*
assume $A{:}P \leq Q$ **and** $B{:}Q \leq R$ **thus** $P \leq R$
by(*insert A B*, *simp add*: *le-ref-def*, *auto*)
next
fix $P\ Q$ :: $'\alpha$ *process*
assume $A{:}P \leq Q$ **and** $B{:}Q \leq P$ **thus** $P = Q$
by(*insert A B*, *auto simp*: *le-ref-def Process-eq-spec*)
qed

end

theory *Bot*
imports *Process*
begin

definition *Bot* :: $'\alpha$ *process*
where *Bot* $\equiv$ *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*: *front-tickFree-Nil 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*: *F Bot* = $\{(s,X).$ *front-tickFree s*$\}$
by(*simp add*: *Bot-def FAILURES-def F-def Rep-Abs-Bot*)

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

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

axioms
Bot-is-UU : *Bot* = $\bot$

end

theory *Skip*
imports *Process*

```
begin

constdefs
    SKIP :: 'a process
   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
                    tickFree-Nil HOL.nnf-simps(2) 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 FAILURES-def F-def is-process-REP-Skip2)

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

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

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* ≡ *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 ftF-Nil*)

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

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

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

3.1 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)
```

3.2 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(□ *x* ∈ *A* → *P x*) = {*d*. *d* ≠ [] ∧ *hd d* ∈ (*ev* ‘ *A*) ∧ (∃ *a*. *ev a* = *hd d* ∧ *tl d* ∈ *D*(*P a*))}
by(*simp add*:*Mprefix-def D-def Rep-Abs-Mp DIVERGENCES-def*)

lemma *T-Mprefix*:
T(□ *x* ∈ *A* → *P x*)={*s*. *s*=[] ∨ (∃ *a*. *a* ∈ *A* & *s*≠[] ∧ *hd s* = *ev a* ∧ *tl s* ∈ *T*(*P a*))}
by(*auto simp*: *T-F-spec*[*symmetric*] *F-Mprefix*)

3.3 Basic Properties

lemma *tick-T-Mprefix* [*simp*]: [*tick*] ∉ *T*(□ *x* ∈ *A* → *P x*)
by(*simp add*:*T-Mprefix*)

lemma *Nil-Nin-D-Mprefix* [*simp*]: [] ∉ *D*(□ *x* ∈ *A* → *P x*)
by(*simp add*: *D-Mprefix*)

3.4 Proof of Continuity Rule

lemma *proc-ord2a* :
⟦*P* ⊑ *Q*; *s* ∉ *D P*⟧ ⟹ ((*s*, *X*) ∈ *F P*) = ((*s*, *X*) ∈ *F Q*)
by(*auto simp*: *process-ord-def Ra-def*)

lemma *mono-Mprefix1*:
∀ *a*. *P a* ⊑ *Q a* ⟹ *D* (*Mprefix A Q*) ⊆ *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*:
∀ *x*. *P x* ⊑ *Q x* ⟹ ∀ *s*. *s* ∉ *D* (*Mprefix A P*) ⟶ *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* :
∀ *x*. *P x* ⊑ *Q x* ⟹ *min-elems* (*D* (*Mprefix A P*)) ⊆ *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*:
∀ *x*. *P x* ⊑ *Q x* ⟹ *Mprefix A P* ⊑ *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*: *Ffun.less-fun-def monofun-def mono-Mprefix0*)

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

lemma *cont-revert2cont-pointwise*:
⋀*x*. *cont* (*f x*) ⟹ *cont* (λ*x y*. *f y x*)
sorry

lemma *Mprefix-cont* :
⋀ *x*. *cont*((*f*::([′*a*,′*a process*]=>′*a process*)) *x*) ⟹ *cont*(λ *y*. *Mprefix A* (λ *z*. *f z y*))
apply(*rule-tac f* = %*z y*. (*f y z*) **in** *Cont.cont2cont-compose*)
apply(*rule Cont.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* = *Ffun.less-fun-def*
lemmas *cont-compose1* = *Cont.cont2cont-compose*
lemmas *mono-contlub-imp-cont* = *Cont.monocontlub2cont*

3.5 High-level Syntax

constdefs
read :: [′*a*=>′*b*,′*a set*, ′*a* => ′*b process*] => ′*b process*

```
  read c A P  ≡ Mprefix(c ‘ A) (P o (inv c))
  write    :: ['a=>'b, 'a, 'b process] => 'b process
  write c a P ≡ Mprefix {c a} (λ x. P)
  write0   :: ['a, 'a process] => 'a process
  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 (%p. P)
 -write c p P    == CONST write c p P
 -readX c p b P  => CONST read c {p. b} (%p. P)
 -writeS a P     == CONST write0 a P
```

end

4 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(*xsymbol*)
det (**infixl** $\Box$ *18*)

axioms
F-ndet : $F(P\ [+]\ Q) = \{(s,X).\ s = [] \wedge (s,X) \in F\ P \cap F\ Q\}$
$\cup \{(s,X).\ s \neq [] \wedge (s,X) \in F\ P \cup F\ Q\}$
$\cup \{(s,X).\ s = [] \wedge s \in D\ P \cup D\ Q\}$
$\cup \{(s,X).\ s = [] \wedge \mathit{tick} \notin X \wedge [\mathit{tick}] \in T\ P \cup T\ Q\}$
D-ndet : $D(P\ [+]\ Q) = D\ P \cup D\ Q$
T-ndet : $T(P\ [+]\ Q) = T\ P \cup T\ Q$
ndet-cont : $[|\ \mathit{cont}\ f;\ \mathit{cont}\ g\ |] ==> \mathit{cont}\ (\lambda x.\ f\ x\ [+]\ g\ x)$

end

5 Nondeterministic Choice Operator Definition

theory *Ndet*
imports *Process*
begin

definition
ndet :: $['\alpha\ \mathit{process},'\alpha\ \mathit{process}] \Rightarrow {'\alpha}\ \mathit{process}$ (**infixl** $|{-}|$ *16*)
where $P\ |{-}|\ Q \equiv \mathit{Abs\text{-}Process}(F\ P \cup F\ Q\ ,\ D\ P \cup D\ Q)$

notation(*xsymbol*)
ndet (**infixl** $\sqcap$ *16*)

axioms
F-ndet : $F(P \sqcap Q) = F\ P \cup F\ Q$
D-ndet : $D(P \sqcap Q) = D\ P \cup D\ Q$
T-ndet : $T(P \sqcap Q) = T\ P \cup T\ Q$
ndet-cont : $[\![\mathit{cont}\ f;\ \mathit{cont}\ g]\!] \Longrightarrow \mathit{cont}\ (\lambda x.\ f\ x \sqcap g\ x)$

end

6 The Sequence Operator

theory *Seq*
imports *Process*

begin

constdefs *seq* :: $['a\ \mathit{process},'a\ \mathit{process}] => {'a}\ \mathit{process}$ (**infixl** ‘;‘ *24*)

$P\ \text{‘;‘}\ Q \equiv \mathit{Abs\text{-}Process}$
$(\{(t,\ X).\ (t,\ X \cup \{\mathit{tick}\}) \in F\ P \wedge \mathit{tickFree}\ t\} \cup$
$\{(t,\ X).\ \exists\, t1\ t2.\ t = t1\ @\ t2 \wedge t1\ @\ [\mathit{tick}] \in T\ P \wedge (t2,\ X) \in F\ Q\} \cup$
$\{(t,\ X).\ \exists\, t1\ t2.\ t = t1\ @\ t2 \wedge t1 \in D\ P \wedge \mathit{tickFree}\ t1 \wedge \mathit{front\text{-}tickFree}\ t2\} \cup$
$\{(t,\ X).\ \exists\, t1\ t2.\ t = t1\ @\ t2 \wedge t1\ @\ [\mathit{tick}] \in T\ P \wedge t2 \in D\ Q\},$
$\{t1\ @\ t2\ | t1\ t2.\ t1 \in D\ P \wedge \mathit{tickFree}\ t1 \wedge \mathit{front\text{-}tickFree}\ t2\} \cup$
$\{t1\ @\ t2\ | t1\ t2.\ t1\ @\ [\mathit{tick}] \in T\ P \wedge t2 \in D\ Q\})$

axioms
F-seq $: F(P\ \text{‘;‘}\ Q) = \{(t,\ X).\ (t,\ X \cup \{\mathit{tick}\}) \in F\ P \wedge \mathit{tickFree}\ t\} \cup$
$\{(t,\ X).\ \exists\, t1\ t2.\ t = t1\ @\ t2 \wedge t1\ @\ [\mathit{tick}] \in T\ P \wedge (t2,\ X) \in F\ Q\} \cup$
$\{(t,\ X).\ \exists\, t1\ t2.\ t = t1\ @\ t2 \wedge t1 \in D\ P \wedge \mathit{tickFree}\ t1 \wedge \mathit{front\text{-}tickFree}\ t2\} \cup$
$\{(t,\ X).\ \exists\, t1\ t2.\ t = t1\ @\ t2 \wedge t1\ @\ [\mathit{tick}] \in T\ P \wedge t2 \in D\ Q\}$

D-seq $: D(P\ \text{‘;‘}\ Q) = \{t1\ @\ t2\ | t1\ t2.\ t1 \in D\ P \wedge \mathit{tickFree}\ t1 \wedge \mathit{front\text{-}tickFree}\ t2\} \cup$
$\{t1\ @\ t2\ | t1\ t2.\ t1\ @\ [\mathit{tick}] \in T\ P \wedge t2 \in D\ Q\}$

T-seq $: T(P\ \text{‘;‘}\ Q) = \{t.\ \exists\ X.\ (t,\ X \cup \{\mathit{tick}\}) \in F\ P \wedge \mathit{tickFree}\ t\} \cup$ (∗ *REALLY ???* ∗)
$\{t.\ \exists\ t1\ t2.\ t = t1\ @\ t2 \wedge t1\ @\ [\mathit{tick}] \in T\ P \wedge t2 \in T\ Q\} \cup$
$\{t1\ @\ t2\ | t1\ t2.\ t1 \in D\ P \wedge \mathit{tickFree}\ t1 \wedge \mathit{front\text{-}tickFree}\ t2\} \cup$
$\{t1\ @\ t2\ | t1\ t2.\ t1\ @\ [\mathit{tick}] \in T\ P \wedge t2 \in D\ Q\}$

seq-cont: $[\![\mathit{cont}\ f;\ \mathit{cont}\ g]\!] \Longrightarrow \mathit{cont}\ (\lambda\ x.\ f\ x\ \text{‘;‘}\ g\ x)$

end

7 The Hiding Operator

theory *Hide*
imports *Process*
begin

primrec *trace-hide* $:: [{}'\alpha\ \mathit{trace}, ({}'\alpha\ \mathit{event})\ \mathit{set}] => {}'\alpha\ \mathit{trace}$ **where**
$\mathit{trace\text{-}hide}\ []\ A = []$

```
|        trace-hide (x # s) A = (if x ∈ A
                                 then trace-hide s A
                                 else x # (trace-hide s A))


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

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


definition
  Hide  :: ['α process ,'α set] => 'α 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-Hide    : 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-Hide    : D(P \ A) = {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)))}

  T-Hide    : T(P \ A) = {s.    ∃  t. s = trace-hide t (ev‘A) ∧ t ∈  T P}
```

Hide-cont : $[\![cont\ f;\ finite\ A]\!] \Longrightarrow cont\ (\lambda x.\ f\ x \setminus A)$

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

end

theory *Sync*
imports *Process*
begin

consts *setinterleaving* ::$'a\ trace \times ('a\ event)\ set \times 'a\ trace \Rightarrow ('a\ trace)set$

recdef *setinterleaving measure*$(\lambda(l1,\ s,\ l2).\ size\ l1 + size\ l2)$

```
si-empty1: setinterleaving([], X, []) = {[]}
si-empty2: setinterleaving([], X, (y # t)) =
             (if (y ∈ X)
              then {}
              else {z.∃ u. z = (y # u) ∧ u ∈ setinterleaving ([], X, t)})
si-empty3: setinterleaving((x # s), X, []) =
             (if (x ∈ X)
              then {}
              else {z.∃ u. z = (x # u) ∧ u ∈ setinterleaving (s, X, [])})
si-neq   : setinterleaving((x # s), X, (y # t)) =
             (if (x ∈ X)
              then if (y ∈ X)
                   then if (x = y)
                        then {z.∃ u. z = (x#u) ∧ u ∈ setinterleaving(s, X, t)}
                        else {}
```

```
                    else {z.∃ u. z = (y#u) ∧ u ∈ setinterleaving ((x#s), X, t)}
               else if (y ∉ X)
                   then {z.∃ u. z = (x # u) ∧ u ∈ setinterleaving (s, X, (y # t))}
                       ∪ {z.∃ u. z = (y # u) ∧ u ∈ setinterleaving((x # s), X, t)}
                  else {z.∃ u. z = (x # u) ∧ u ∈ setinterleaving (s, X, (y # t))})


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


lemma sym2 [simp]:
      ∀ s. setinterleaving (s, X, t) = setinterleaving (t, X, s)
      ⟶ 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)


consts setinterleaves :: ['a trace, ('a trace×'a trace)×('a event) set] ⇒ bool
                         (infixl setinterleaves 70)
translations
 u setinterleaves ((s, t), X) == (u ∈ setinterleaving(s, X, t))


definition sync :: ['a process,'a set,'a process] => 'a process
          ((3- ⟦- ⟧/ -) [14,0,15] 14)
where
 P ⟦ A ⟧ Q ==
      Abs-Process({(s,R).∃ t u X Y. (t,X) ∈ F P ∧ (u,Y) ∈ F Q ∧
                               s setinterleaves ((t,u),(ev‘A) ∪ {tick}) ∧
                               R = (X ∪ Y) ∩ ((ev‘A) ∪ {tick}) ∪ X ∩ Y} ∪
                 {(s,R).∃ t u r v. front-tickFree v ∧ (tickFree r ∨ v=[]) ∧
                               s = r@v ∧
                               r setinterleaves ((t,u),(ev‘A) ∪ {tick}) ∧
                               (t ∈ D P ∧ u ∈ T Q ∨ t ∈ D Q ∧ u ∈ T P)},
                 {s.   ∃ t u r v. front-tickFree v ∧ (tickFree r ∨ v=[]) ∧
                               s = r@v ∧
                               r setinterleaves ((t,u),(ev‘A) ∪ {tick}) ∧
                               (t ∈ D P ∧ u ∈ T Q ∨ t ∈ D Q ∧ u ∈ T P)})


axioms
```

$$\begin{array}{ll}
\mathit{F\text{-}sync} & : F(P \llbracket A \rrbracket Q) = \\
& \quad \{(s,R). \exists\ t\ u\ X\ Y.\ (t,X) \in F\ P \land \\
& \qquad\qquad\qquad (u,Y) \in F\ Q \land \\
& \qquad\qquad\qquad s\ \mathit{setinterleaves}\ ((t,u),(\mathit{ev}`A) \cup \{\mathit{tick}\}) \land \\
& \qquad\qquad\qquad R = (X \cup Y) \cap ((\mathit{ev}`A) \cup \{\mathit{tick}\}) \cup X \cap Y\} \cup \\
& \quad \{(s,R). \exists\ t\ u\ r\ v.\ \mathit{front\text{-}tickFree}\ v \land \\
& \qquad\qquad\qquad (\mathit{tickFree}\ r \lor v{=}[]) \land \\
& \qquad\qquad\qquad s = r@v \land \\
& \qquad\qquad\qquad r\ \mathit{setinterleaves}\ ((t,u),(\mathit{ev}`A){\cup}\{\mathit{tick}\}) \land \\
& \qquad\qquad\qquad (t \in D\ P \land u \in T\ Q \lor t \in D\ Q \land u \in T\ P)\}
\end{array}$$

$$\begin{array}{ll}
\mathit{D\text{-}sync} & : D(P \llbracket A \rrbracket Q) = \\
& \quad \{s.\quad \exists\ t\ u\ r\ v.\ \mathit{front\text{-}tickFree}\ v \land (\mathit{tickFree}\ r \lor v{=}[]) \land \\
& \qquad\quad s = r@v \land r\ \mathit{setinterleaves}\ ((t,u),(\mathit{ev}`A) \cup \{\mathit{tick}\}) \land \\
& \qquad\quad (t \in D\ P \land u \in T\ Q \lor t \in D\ Q \land u \in T\ P)\}
\end{array}$$

$$\begin{array}{ll}
\mathit{T\text{-}sync} & : T(P \llbracket A \rrbracket Q) = \\
& \quad \{s.\quad \forall\ t\ u.\ t \in T\ P \land u \in T\ Q \land \\
& \qquad\qquad s\ \mathit{setinterleaves}\ ((t,u),(\mathit{ev}`A) \cup \{\mathit{tick}\})\}
\end{array}$$

end

8 Toplevel Theory

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

8.1 Refinement Proof Rules

8.2 The "Laws" of CSP

end

9 Refinement Example with Buffer over infinite Alphabet

theory *CopyBuffer*
imports *CSP*
begin

10 Defining the Copy-Buffer Example

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

constdefs $\mathit{SYN} :: ('a\ \mathit{channel})\ \mathit{set}$
where $\mathit{SYN} \equiv (\mathit{range}\ \mathit{mid}) \cup \{\mathit{ack}\}$

constdefs $\mathit{COPY} :: ('a\ \mathit{channel})\ \mathit{process}$
where $\mathit{COPY} \equiv (\mu\ \mathit{COPY}.\ \mathit{left}`?`x \rightarrow \mathit{right}`!`x \rightarrow \mathit{COPY})$

constdefs $\mathit{SEND} :: ('a\ \mathit{channel})\ \mathit{process}$
where $\mathit{SEND} \equiv (\mu\ \mathit{SEND}.\ \mathit{left}`?`x \rightarrow \mathit{mid}`!`x \rightarrow \mathit{ack} \rightarrow \mathit{SEND})$

constdefs $\mathit{REC} :: ('a\ \mathit{channel})\ \mathit{process}$
where $\mathit{REC} \equiv (\mu\ \mathit{REC}.\ \mathit{mid}`?`x \rightarrow \mathit{right}`!`x \rightarrow \mathit{ack} \rightarrow \mathit{REC})$

constdefs
$\mathit{SYSTEM} :: ('a\ \mathit{channel})\ \mathit{process}$
$\mathit{SYSTEM} \equiv ((\mathit{SEND}\ [\![\ \mathit{SYN}\ ]\!]\ \mathit{REC}) \setminus \mathit{SYN})$

11 The Standard Proof

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.