Require Import Arith.

Require Import ZArith.

Require Import Bool.

Require Import List.

Require Import String.

Require Import Logic.

Require Import Relations.

(\* Compensating CSP \*)

Section cCSP.

Set Implicit Arguments.

Unset Strict Implicit.

(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Prcess Section \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)

(\* isomorephisim \*)

Parameter A:Set.

(\* Channel \*)

Axiom Ch :Set -> Set.

(\* Inductive types Decleration \*)

(\* Channel Type \*)

Inductive Channeltype (M:Set) :Set := addM : M -> Channeltype M -> Channeltype M.

(\* Actions \*)

Inductive Action :Type:=

 |SUCC : Action

 |Thr : Action

 |Y : Action

 |TAUI : Action

 |CompI : Action -> Action -> Action

 |TAU : forall (A:Set), Ch A -> Action

 |Atomic : forall (A:Set), A -> Action

 |IN : forall (A:Set), Ch A -> A -> Action

 |OUT : forall (A:Set), Ch A -> A -> Action.

(\* Process \*)

CoInductive Process : Type :=

 | STOP : Process

 | Pref : Action -> Process -> Process

 | Par : Process -> Process -> Process

 | inchoice : Process -> Process -> Process

 | exchoice : Process -> Process -> Process

 | Seq : Process -> Process -> Process

 | interrupt : Process -> Process -> Process

 | Compen : Process -> Process -> Process.

(\* Simulation of CSP notations \*)

Variable (pro1 pro2:Process)(va:A)(cha:Ch A)(ac:Action).

Notation "cha ! va " := (OUT cha va )(at level 100).

Notation "cha ? va " := (IN cha va )(at level 100).

Notation " ac ->> pro1 " := (Pref ac pro1) (at level 80).

Notation " pro1 |P| pro2" := (Par pro1 pro2)(at level 80).

Notation " pro1 [] pro2 " := (exchoice pro1 pro2)(at level 80).

Notation " pro1 |-| pro2 " := (inchoice pro1 pro2)(at level 80).

Notation " pro1 ; pro2 " := (Seq pro1 pro2)(at level 80).

Notation " pro1 (%) pro2 " := (Compen pro1 pro2)(at level 80).

Notation " pro1 |> pro2 " := (interrupt pro1 pro2)(at level 80).

(\* Standerd Processes \*)

Definition Skip :Process := Pref SUCC STOP.

Definition Throw :Process := Pref Thr STOP.

Definition Yield :Process := Pref Y STOP.

Definition Skipp := Skip (%) Skip.

Definition Throww := Throw (%) Skip.

Definition Yieldd := Yield (%) Skip.

CoFixpoint Run (a: list Action) :Process :=

 match a with

 |nil => STOP

 | b :: nil => Pref b (Run a)

 |\_ :: b :: \_ => Pref b (Run a)

 end.

Definition Chaos ( a : list Action ) :Process := inchoice STOP (Run a).

(\* Operational Semantics \*)

Inductive Transition: Process -> Action -> Process -> Prop :=

 | Tr\_prefix : forall l p, Transition ( l ->> p ) l p

 | Tr\_Parl : forall p p' q l, Transition p l p' -> Transition (p |P| q) l (p'|P| q)

 | Tr\_Parr : forall p q q' l, Transition q l q' -> Transition (p |P| q) l (p |P| q')

 | Tr\_ParSen : forall p p' q q' l, (Transition p l p') /\ (Transition q l q') -> Transition (p |P| q) l (p' |P| q')

 | Tr\_inchl : forall A (c:Ch A) p q, Transition ( p |-| q)(TAU c) p

 | Tr\_inchr : forall A (c:Ch A) p q, Transition ( p |-| q)(TAU c) q

 | Tr\_exchl : forall p p' q l, Transition p l p' -> Transition (p [] q)l p'

 | Tr\_exchr : forall p q q' l, Transition q l q' -> Transition (p [] q)l q'

 | Tr\_Seq1 : forall p p' q l, Transition p l p' -> Transition (p ; q) l (p' ; q)

 | Tr\_Seq2 : forall A (c:Ch A) p p' q , Transition p SUCC p' -> Transition (p ; q) (TAU c) q

 | Tr\_Compen1 : forall p p' q l , Transition p l p' -> Transition ( p (%) q) l (p' (%) q)

 | Tr\_Compen2 : forall p q , Transition p SUCC STOP -> Transition (p (%) q) SUCC q

 | Tr\_Compen3 : forall p q , Transition p Thr STOP -> Transition (p (%) q) Thr STOP

 | Tr\_interp1 : forall p p' q l, Transition p l p' -> Transition (p |> q) l (p'|> q)

 | Tr\_interp2 : forall p q' q l, Transition p Thr STOP /\ Transition q l q' -> Transition (p |> q) l q'.

(\* Transitions \*)

Inductive Transitions : Process -> Process -> Prop :=

 | Transition0: forall p q l, Transition p l q -> Transitions p q

 | Transitioni : forall p q r l , Transition p l q -> Transitions q r -> Transitions p r.

(\* Collection of Lemmas (Process Section) \*)

Lemma One: forall (l1 l2:Action) p q , l1=l2 /\ Pref l1 p = Pref l2 q -> p = q .

Proof.

 intros.

 destruct H.

 generalize H0.

 rewrite H.

 intros.

 injection H1.

 intros.

 auto.

Qed.

Lemma one :forall (l:Action) , (Pref l STOP) <> STOP.

Proof.

 intros.

 case Pref.

 intros.

 unfold not.

 intros.

 discriminate.

Qed.

Lemma OOne :forall (p q r :Process) (v:A)(c: Ch A) (l:Action) ,

 p= ((c ! v) ->> r) /\ l= (OUT c v) -> Transition p l q = Transition ((c ! v) ->> r) (OUT c v) q.

Proof.

 intros.

 destruct H.

 rewrite H.

 rewrite H0.

 reflexivity.

Qed.

Lemma tOOne :forall A (c:Ch A)(p:Process) (v:A),Transition ((c ! v) ->> p) (OUT c v) p.

Proof.

 intros.

 apply (Tr\_prefix (OUT c v) p).

Qed.

Lemma two : forall p (c:Ch A) (k v:A) ,exists q,

 (Transition ((c ! v) ->> ( (c ! k) ->> p)) (OUT c v) q) /\ (Transition q (OUT c k) p).

Proof.

 intros.

 exists ((c ! k) ->> p).

 split.

 apply (Tr\_prefix (OUT c v) ((c ! k) ->> p)).

 apply (Tr\_prefix (OUT c k) p).

Qed.

Lemma Trans: forall p q r, Transitions p q -> Transitions q r -> Transitions p r.

Proof.

 intros p q r H H0.

 generalize H.

 induction 1.

 apply (Transitioni H1 H0) .

 apply (Transitioni H1).

 apply IHTransitions; auto.

Qed.

(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* end of Prcess Section \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)

(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Trace Section \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)

(\* Lazy Lists\*)

CoInductive Llist (A:Type) :Type :=

 | LNil : Llist A

 | LCons : A -> Llist A -> Llist A.

CoInductive Llists (A:Type) :Type :=

 | LNils : Llists A

 | LConss : A -> Llists A -> Llists A

 | LlConss : A -> Llists A -> Llists A -> Llists A.

CoFixpoint LAppend (A:Type)(u v: Llist A) :Llist A:=

 match u with

 | LNil => v

 | LCons a u' => LCons a (LAppend u' v )

end.

(\* Trace Predicate \*)

Inductive isTrace : Process -> list Action -> Prop :=

 |empty\_Trace : isTrace STOP nil

 |lcons\_TracePrf : forall (p:Process)(l: list Action)(a:Action), isTrace p l -> isTrace (a ->> p) (a::l)

 |lcons\_TraceSeq : forall (p q:Process)(l k :list Action)(a:Action),

 isTrace p (a::l) -> isTrace q k -> isTrace (p ; q) (a::(l++k))

 |lcons\_Traceinchl : forall (p q:Process)(l k :list Action),

 isTrace p l -> isTrace q k -> isTrace (p |-| q) (TAUI::l)

 |lcons\_Traceinchr : forall (p q:Process)(l k : list Action),

 isTrace p l -> isTrace q k -> isTrace (p |-| q) (TAUI::k)

 |lcons\_Traceexchl : forall (p q:Process)(l k : list Action)(a:Action),

 isTrace p l -> isTrace q k -> isTrace (p [] q) l

 |lcons\_Traceexchr : forall (p q:Process)(l k : list Action)(b:Action),

 isTrace p l -> isTrace q k -> isTrace (p [] q) k

 |lcons\_TraceParl : forall (p q:Process)(l k pk: list Action)(a b:Action),

 isTrace p (a::l) -> isTrace q (b::k) -> isTrace (p |P| q) pk -> isTrace (p |P| q) (a::(b::(pk)))

 |lcons\_TraceParr : forall (p q:Process)(l k pk: list Action)(a b:Action),

 isTrace p (a::l) -> isTrace q (b::k) -> isTrace (p |P| q) pk -> isTrace (p |P| q) (a::(b::(pk)))

 |lcons\_TraceSen : forall (p q:Process)(l k pk: list Action)(a:Action),

 isTrace p (a::l) -> isTrace q (a::k) -> isTrace (p |P| q) pk -> isTrace (p |P| q) (a::pk)

 |lcons\_TraceCompen: forall (p q:Process)(l k pk:list Action)(a b:Action),

 isTrace p (a::l) -> isTrace q (b::k) -> isTrace (p (%) q) pk -> isTrace (p (%) q)((CompI a b )::pk)

 |lcons\_TraceInter1 : forall (p q:Process)(l k : list Action)(a:Action),

 isTrace p (a::l) -> isTrace q k -> isTrace (p |> q) (a::l)

 |lcons\_TraceInter2 : forall (p q:Process)(l k : list Action)(b:Action),

 isTrace p (Thr::nil) -> isTrace q (b::k) -> isTrace (p |> q) (b::k).

Variable trace : forall p:Process, {t:list Action | isTrace p t}.

Definition Enumtrace (p:Process)(T:{t:list Action | isTrace p t}) : list Action :=

 match T with

 | exist t h => t end.

Definition Trace (p:Process) : {T:(list Action) | isTrace p T }.

 intros.

 case p.

 exists nil.

 apply (empty\_Trace).

 intros.

 exists(a:: Enumtrace(trace p0)).

 apply (lcons\_TracePrf a) .

 destruct trace. simpl.

 exact i.

 intros;apply trace.

 intros;apply trace.

 intros;apply trace.

 intros;apply trace.

 intros;apply trace.

 intros;apply trace.

Defined.

Definition intrace (p:Process)(t:list Action): Prop := (isTrace p t).

CoInductive AllTraces :Process -> Llist (list Action) -> Prop :=

 |TracesofSTOP : AllTraces STOP (LCons nil (LNil (list Action)))

 |TracesOfPrf: forall (p:Process)(l: list Action)(k:Llist (list Action))(a:Action),

 isTrace p l -> AllTraces p k -> AllTraces (a ->> p) (LCons (a::l) k)

 |TracesOfSeq: forall (p q:Process)(l: list Action)(k1 k:Llist (list Action))(a:Action),

 isTrace p l -> AllTraces p k -> AllTraces q k1 -> AllTraces (p ; q) (LCons l (LAppend k k1))

 |TracesOfinchl: forall (p q:Process)(l: list Action)(k1 k:Llist (list Action)),

 isTrace p l -> AllTraces p k -> AllTraces q k1 -> AllTraces (p |-| q) (LCons (TAUI::l) k)

 |TracesOfinchr: forall (p q:Process)(l: list Action)(k1 k:Llist (list Action)),

 isTrace q l -> AllTraces p k -> AllTraces q k1 -> AllTraces (p |-| q) (LCons (TAUI::l) k1)

 |TracesOfexchl : forall (p q:Process)(l: list Action)(k1 k:Llist (list Action))(a:Action),

 isTrace p (a::l) -> AllTraces p k -> AllTraces q k1 -> AllTraces (p [] q) (LCons (a::l) k)

 |TracesOfexchr : forall (p q:Process)(l: list Action)(k1 k:Llist (list Action))(a:Action),

 isTrace q (a::l) -> AllTraces p k -> AllTraces q k1 -> AllTraces (p [] q) (LCons (a::l) k1)

 |TracesOfParl : forall (p q:Process)(l k kp: list Action)(a b:Action)(k1:Llist (list Action)),

 isTrace q (a::l) -> isTrace p (b::k) -> isTrace (p |P| q) kp -> AllTraces (p |P| q) k1 ->

 AllTraces (p |P| q) (LCons (a::(b::kp))k1)

 |TracesOfParr : forall (p q:Process)(l k kp: list Action)(a b:Action)(k1:Llist (list Action)),

 isTrace q (a::l) -> isTrace p (b::k) -> isTrace (p |P| q) kp -> AllTraces (p |P| q) k1 ->

 AllTraces (p |P| q) (LCons (b::(a::kp))k1)

 |TracesOfSen : forall (p q:Process)(l k kp: list Action)(a:Action)(k1:Llist (list Action)),

 isTrace q (a::l) -> isTrace p (a::k) -> isTrace (p |P| q) kp -> AllTraces (p |P| q) k1 ->

 AllTraces (p |P| q) (LCons (a::kp) k1)

 |TracesOfComp : forall (p q:Process)(l k kp: list Action)(a b:Action)(k1:Llist (list Action)),

 isTrace q (a::l)-> isTrace p (b::k)-> isTrace (p (%) q) kp -> AllTraces (p (%) q) k1 ->

 AllTraces (p (%) q) (LCons ((CompI a b)::kp) k1)

 |TracesOfInter1: forall (p q:Process)(l k: list Action)(k1:Llist (list Action)),

 isTrace q l -> isTrace p k -> AllTraces (p (%) q) k1 -> AllTraces (p (%) q) (LCons l k1)

 |TracesOfInter2: forall (p q:Process)(l k: list Action)(k1:Llist (list Action)),

 isTrace q (Thr::nil) -> isTrace p k -> AllTraces (p (%) q) k1 -> AllTraces (p (%) q) (LCons k k1).

Variable traces : forall p:Process, {t:Llist(list Action) | AllTraces p t}.

Definition Enumtraces (p:Process)(T:{t:Llist(list Action) | AllTraces p t}) :Llist( list Action) :=

 match T with

 | exist t h => t end.

Definition Traces (p:Process) : {T:Llist (list Action) | AllTraces p T }.

 intros.

 case p.

 exists (LCons nil (LNil (list Action))).

 apply TracesofSTOP.

 intros.

 exists (LCons (a::Enumtrace(trace p0)) (Enumtraces(traces p0))).

 apply (TracesOfPrf a) .

 destruct trace. simpl.

 exact i.

 destruct traces. simpl.

 exact a0.

 intros;apply traces.

 intros;apply traces.

 intros;apply traces.

 intros;apply traces.

 intros;apply traces.

 intros;apply traces.

Defined.

(\* Process Bisimulation \*)

CoInductive Process\_eq :Process -> Process -> Prop :=

 Pro\_eq :

 forall p q:Process,

 (forall (a:Action) (p':Process), Transition p a p' -> exists q' :Process , Transition q a q' /\ Process\_eq p' q' ) ->

 (forall (a:Action) (q':Process), Transition q a q' -> exists p' :Process , Transition p a p' /\ Process\_eq p' q') ->

 Process\_eq p q.

Lemma Process\_refl : forall p:Process, Process\_eq p p.

Proof.

 cofix.

 intros.

 apply (Pro\_eq ).

 intros. exists p'.

 split; auto with v62.

 intros. exists q'.

 split; auto with v62.

Qed.

Lemma Process\_sym : forall p q : Process, Process\_eq p q -> Process\_eq q p.

cofix.

intros p q H. inversion\_clear H.

apply Pro\_eq.

intros a q'.

intros trans.

elim (H1 a q' trans). clear H1.

intros p' H; elim H; exists p'; split; auto with v62.

intros a p' trans.

elim (H0 a p' trans); clear H0.

intros q' H; elim H; clear H.

intros trans1 str.

exists q'; split; auto with v62.

Qed.

Hint Immediate Process\_sym.

Lemma Process\_trans :

 forall p q r : Process, Process\_eq p q -> Process\_eq q r -> Process\_eq p r.

cofix.

intros p q r pq qr.

inversion\_clear pq; inversion\_clear qr.

apply Pro\_eq.

intros a p' tp.

cut (exists q' : Process, Transition q a q' /\ Process\_eq p' q').

intros G. elim G. clear G.

intros q' G. elim G.

intros tq pq'. elim (H1 a q' tq).

clear G.

intros r' G; elim G; clear G.

intros tr qr'; exists r'; split. auto with v62.

apply Process\_trans with q'; auto with v62.

elim (H a p' tp).

intros q' G; elim G; clear G.

intros tq pq'.

exists q'; split; auto with v62.

intros a r' tr.

cut (exists q' : Process, Transition q a q' /\ Process\_eq q' r').

intros G; elim G; clear G.

intros q' G; elim G; clear G.

intros tq qr'.

elim (H0 a q' tq).

intros p' G; elim G; clear G.

intros tp pq'; exists p'; split; auto with v62.

apply Process\_trans with q'; auto with v62.

elim (H2 a r' tr).

intros q' G; elim G; clear G.

intros tq qr'; exists q'; split; auto with v62.

Qed.

(\* Traces Bisimulation \*)

(\*

CoInductive Traces\_eq : Llist (list Action) -> Llist (list Action)-> Prop :=

 |Ts\_eq: forall (p q: Process) (t:list Action),

 Traces\_eq (Enumtraces(Traces p)) (Enumtraces(Traces q)) /\ isTrace p t /\ isTrace q t ->

 Traces\_eq (LCons t (Enumtraces(Traces p))) (LCons t (Enumtraces(Traces q))).

\*)

CoInductive Llist\_eq (A:Type):Llist A-> Llist A -> Prop:=

 | eq0: Llist\_eq (LNil A) (LNil A)

 | eq1: forall (a : A)(u u':Llist A), Llist\_eq u u' -> Llist\_eq (LCons a u) (LCons a u').

Lemma Llist\_refl : forall (A:Type)(l:Llist A), Llist\_eq l l.

Proof.

 cofix H.

 intros.

 case l; [left | right ].

 apply H.

Qed.

Lemma Pro\_Traces\_refl : forall p:Process, Llist\_eq (Enumtraces(Traces p)) (Enumtraces(Traces p)).

cofix H.

intros.

case (Enumtraces(Traces p)).

apply eq0.

intros.

apply eq1.

apply (Llist\_refl l0).

Qed.

(\* Trace Equavilence \*)

Theorem Trace\_Eqv : forall p q:Process , Llist\_eq (Enumtraces(Traces p)) (Enumtraces(Traces q))

 -> Process\_eq p q.

Proof.

 cofix H.

 intros.

 destruct p .

 destruct q .

 apply (Process\_refl STOP).

 apply (Pro\_eq).

 intros; exists p' ; split ; inversion H0 ; apply (Process\_refl p') .

 intros; exists q'; split ; inversion H0 ; apply (Process\_refl q').

 apply (Pro\_eq).

 intros ; exists p'; split ; inversion H1; apply (Process\_refl p').

 intros .

 cut (exists p' : Process, Transition STOP a p' /\ Process\_eq p' q').

 intros G; elim G.

 intros p' m. apply G. elim H1.

 intros.

 intros tq qr'.

 elim m. intros.

 apply G .

 exists q'. split. inversion H0.

 inversion H1.

 elim (H1 a q').

 exists q'. split .

(\*\*\*\*\*\*\*\*\*\*\*\*\*\* End of Trace Section \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)

(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Refusals Section \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)

(\* Refusals \*)

CoInductive Refusal (relem:Set) :Set :=

 | addredf : relem -> Refusal relem -> Refusal relem.

(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)

(\* Pictorial Representation of a Process \*)

CoInductive Tree (A:Type):Type :=

 | leaf : Tree A

 | OneNode : A -> Tree A -> Tree A

 | OneNodeE : Tree A -> Tree A

 | TwoNodes : A -> Tree A -> Tree A -> Tree A

 | TwoNodesE: Tree A -> Tree A -> Tree A.

CoFixpoint Seq2To1 (p q :Process):Process :=

 match p with

 | STOP => q

 | a ->> p'' => a ->> (Seq2To1 p'' q)

 | inchoice p'' q' => inchoice (Seq2To1 p'' q) (Seq2To1 q' q)

 | exchoice p'' q' => exchoice (Seq2To1 p'' q) (Seq2To1 q' q)

 | Seq p'' q' => Seq p'' (Seq2To1 q' q)

 | Par p'' q' => Par (Seq2To1 p'' q) (Seq2To1 q' q)

 | \_ => q

 end.

CoFixpoint PTrace (p:Process): Tree Action :=

 match p with

 | STOP => leaf Action

 | a ->> p' => OneNode a (PTrace p')

 | inchoice p' q => TwoNodes TAUI (PTrace p') (PTrace q)

 | exchoice p' q => TwoNodesE (PTrace p') (PTrace q)

 | Seq p' q => OneNodeE (PTrace (Seq2To1 p' q) )

 | Par p' q =>

 match p', q with

 | STOP , \_ => OneNodeE (PTrace q)

 | \_ , STOP => OneNodeE (PTrace p')

 | a->> p'' , b ->> q' => TwoNodesE (OneNode a (PTrace (p'' |P| q))) (OneNode b (PTrace (p' |P| q')))

 | a->> p'' , q' [] q'' =>

 TwoNodesE (OneNode a (PTrace (p'' |P| q))) (TwoNodesE (PTrace (p' |P| q'))(PTrace (p' |P| q'')))

 | a->> p'' , q' |-| q'' =>

 TwoNodesE (OneNode a (PTrace (p'' |P| q))) (TwoNodes TAUI (PTrace (p' |P| q'))(PTrace (p' |P| q'')))

 | a->> p'' , q' |P| q'' =>

 TwoNodesE (OneNode a (PTrace (p'' |P| q))) (OneNodeE (PTrace (q |P| p')))

 | a->> p'' , q' ; q'' => OneNode a (PTrace (p'' |P| (Seq2To1 q' q'')))

 | a ->> p'' , q' |> q'' => OneNode a (PTrace (p'' |P| q'))

 | a ->> p'' , q' (%) q'' => OneNode a (PTrace (p'' |P| q'))

 | q' [] q'', \_ => TwoNodesE (PTrace (q' |P| q)) (PTrace (q'' |P| q))

 | q' |-| q'', \_ => TwoNodes TAUI (PTrace (q' |P| q)) (PTrace (q'' |P| q))

 | q' ; q'' , \_ => OneNodeE (PTrace ((Seq2To1 q' q'') |P| q))

 | q' |P| q'' , \_ => OneNodeE (PTrace (q' |P| (q'' |P| q)) )

 |\_ , \_ =>leaf Action

 end

 |\_ =>leaf Action

 end.

Fixpoint unfold\_Tree (A:Type)(n:nat)(t1 :Tree A):Tree A := match n with

 | O => t1

 | S n' => match t1 with

 | leaf => (leaf A)

 | OneNodeE t1' => OneNodeE (unfold\_Tree n' t1')

 | OneNode a t1'=> OneNode a (unfold\_Tree n' t1')

 | TwoNodes a t1' t1'' => TwoNodes a (unfold\_Tree n' t1') (unfold\_Tree n' t1'')

 | TwoNodesE t1' t1''=> TwoNodesE (unfold\_Tree n' t1') (unfold\_Tree n' t1'')

 end

 end.

CoFixpoint Ptf (t1:Tree Action)(li:list Action): Tree (list Action) :=

 match t1 with

 |leaf => (leaf (list Action))

 |OneNode a t1'=> OneNode (a::li)(Ptf t1' (a::li) )

 |OneNodeE t1' => OneNodeE (Ptf t1' li )

 |TwoNodes a t1' t2 => TwoNodes (a::li) ( Ptf t1' (a::li) ) ( Ptf t2 (a::li) )

 |TwoNodesE t1' t2=> TwoNodesE (Ptf t1' (li) ) ( Ptf t2 (li) )

 end.

Definition TracesSimulation (t1:Tree Action): Tree (list Action) := Ptf t1 nil .

Variable a b c f d a' b' c':Action.

Definition xx:Process := a ->> (f ->> (d ->> ((b ->> STOP) [] (c ->> STOP) ))).

Definition xx':Process := a' ->> ((b' ->> STOP) [] (c' ->> STOP) ).

Eval compute in (unfold\_Tree 50 ( (PTrace (xx ; (xx' ; xx'))) )).

Eval compute in (unfold\_Tree 100( TracesSimulation (PTrace (xx ; (xx' ; xx') )))).

(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)

(\* Testing Example \*)

Variable a b c a' b' c':Action.

Definition xx:Process := a ->> ((b ->> STOP) [] (c ->> STOP) ).

Definition xx':Process := a' ->> ((b' ->> STOP) [] (c' ->> STOP) ).

Eval compute in (unfold\_n 300( PTrace (( Seq2To1 xx xx') |P| xx)) ).