Coq Proof for "A small-step approach to multitrace checking against interactions"

We use Coq to prove the correctness of an oracle algorithm with regards to a formal semantics.

This proof accompanies the submitted paper and the implementation.

Context

This formal semantics defines which are the behaviors that are specified by an interaction model (akin to Message Sequence Charts or UML Sequence Diagrams). Those behaviors are described by traces which are sequences of atomic actions that can be observed on the interfaces of a distributed system's sub-systems. In the absence of a global clock, the observations of those atomic actions cannot be ordered globally. As such we rather have multi-traces i.e. sets of traces, each describing the behavior of a given sub-system.

We define the semantics of interactions in terms of traces and of multi-traces.

We then introduce an algorithm to solve the membership problem for multi-traces i.e. able to determine whether or not a given multi-trace belongs to the semantics of a given interaction.

Dependencies

Below are listed the libraries required for this Coq proof.

- "List" provides utilities on lists. We use lists - among other things - to represent traces.
- "Coq.Vectors.Fin." provides a means to represent finite sets indexed by {1,...,n}.
- "Coq.Vectors.VectorDef." provides a Type used to represent lists of fixed size (vectors). We use vectors to define multitraces as fixed sized lists of traces.
- "Coq.Bool.Bool." is used to handle booleans.
- "Psatz." is required for using the "lia" tactic to solve simple arithemtic problems.
- "Coq.Program.Equality." is required for using the "dependent induction" tactic with "generalizing", allowing the generalisation of some variables of the problem in the induction hypothesis.

```
Require Import List.
Require Coq.Vectors.Fin.
Require Coq.Vectors.VectorDef.
Require Import Coq.Bool.Bool.
Require Import Psatz.
Require Import Coq.Program.Equality.
```

Preliminaries

In the following, we introduce some basic types and functions that will allow us to manipulate concepts in the domain of the observation of distributed system executions.

Signature & Actions

The set of lifelines L is defined as a finite set of (any integer) cardinal Lcard indexed by {1,...,LCard} using the Vectors.Fin library.

```
Parameter LCard : nat.
Definition L := Fin.t (S LCard).
```

The set M of messages is defined in the same manner.

```
Parameter MCard : nat.
Definition M := Fin.t (S MCard).
```

To distinguish between emissions "a!m" and receptions "b?m" we encode the kind of action ({!,?}) with an inductive type "ActKind".

```
Inductive ActKind : Set :=
  |ak_snd:ActKind
  |ak_rcv:ActKind.
```

Below is defined the equality relation for the type "ActKind".

```
Definition eq_actkind (ak1 : ActKind) (ak2 : ActKind) : bool :=
  match ak1 with
    | ak_snd =>
        match ak2 with
        | ak_snd => true
        | ak_rcv => false
        end
    | ak_rcv =>
        match ak2 with
        | ak_snd => false
        | ak_rcv => true
        end
  end.
```

We can now define actions with the "Action" type. Below are also define the equality relation on this type and a utility function "lifeline" returning, for any action the lifeline on which it occurs.

```
Definition Action :Set:= L*ActKind*M.

Definition eq_action (action1 : Action) (action2 : Action) : bool :=
  match action1 with
    | (_ as l1, _ as ak1, _ as m1) =>
        match action2 with
        | (_ as l2, _ as ak2, _ as m2) =>
          andb (Fin.eqb l1 l2)
               (andb (eq_actkind ak1 ak2) (Fin.eqb m1 m2) )
  end
  end.

Definition lifeline: Action -> L :=
  fun '(_ as l,_,_) => l.
```

Trace & Multi-Trace Language

The "Trace" type can then be defined as that of lists of actions ("Action" type).

```
Definition Trace : Type := list Action.
```

The "MultiTrace" type is then defined as the type of vectors of traces ("Trace" type) of fixed size "LCard" (cardinal of L).

```
Definition MultiTrace : Type := (VectorDef.t Trace (S LCard)).
```

Below is defined the length "multitrace_sum_len" of a multitrace as the sum of the lengths of its component traces.

```
Definition util_trace_lens (n:nat) (t:Trace): nat :=
  (length t).

Definition multitrace_sum_len (mu:MultiTrace) : nat :=
  VectorDef.fold_left util_trace_lens 0 mu.
```

The empty multitrace "empty_multitrace" (vector of "LCard" empty traces) is defined below. For any integer "n" we define a list of "n" times the empty trace. Then "empty_multitrace" simply is the conversion to a vector of this list with "n=LCard".

```
Fixpoint empty_trace_list (n:nat) : list Trace :=
match n with
| 0 => nil
| S n' => (nil) :: (empty_trace_list n')
end.

Program Definition empty_multitrace : MultiTrace :=
  (VectorDef.of_list (empty_trace_list (S LCard)) ).
Next Obligation.
induction LCard.
- simpl. reflexivity.
- simpl. rewrite IHn. reflexivity.
Qed.
```

Below is defined the concatenation operator "left_append_action_on_multitrace", which, for a given multitrace and action, appends the action to the correct component trace of the multitrace i.e. the one corresponding to the lifeline on which the action occurs.

```
Definition left_append_action_on_multitrace (mu : MultiTrace)
                                            (a:Action)
                                            : MultiTrace :=
  VectorDef.replace mu (lifeline a) ( a:: (VectorDef.nth mu (lifeline a) ) ).
```

"left_append_mu_increase_sum_len" is a Lemma stating that when we append an action to a multitrace using our concatenation operator "left_append_action_on_multitrace", it increases its length "multitrace_sum_len" by 1.

```
Lemma left_append_mu_increase_sum_len :
  forall (mu:MultiTrace) (a:Action),
    (multitrace_sum_len (left_append_action_on_multitrace mu a))
    = (1 + (multitrace_sum_len mu)).
Proof.
intros mu a.
unfold left_append_action_on_multitrace.
```

```
remember (VectorDef.nth mu (lifeline a)) as t0.
pose proof (replace_on_mu_length mu (lifeline a) (a :: t0) t0).
symmetry in Heqt0.
apply H in Heqt0.
rewrite Heqt0.
simpl. lia.
Qed.
```

"multitrace_cases" is a Lemma stating that a multitrace is either empty or can be obtained by appending an action to the left of another multitrace using the "left_append_action_on_multitrace" concatenation operator.

```
Lemma multitrace_cases :
  forall mu:MultiTrace,
    (mu=empty_multitrace)
    \/ ( exists (mu':MultiTrace)
                (a:Action),
           mu = (left_append_action_on_multitrace mu' a)
       ).
```

"empty_mu_length" and "empty_mu_length_equiv" are Lemmas respectively stating that:

- the length if the empty multitrace "empty_multitrace" is 0
- for any multitrace, if its length is 0 then it is equal to the empty multitrace "empty_multitrace"

```
Lemma empty_mu_length :
  (multitrace_sum_len empty_multitrace = 0).

Lemma empty_mu_length_equiv :
  forall (mu:MultiTrace),
  (multitrace_sum_len mu = 0) <-> (mu = empty_multitrace).
```

"left_append_on_mu_cannot_be_nil" is a Lemma stating that a multitrace obtained using the concatenation operator "left_append_action_on_multitrace" cannot be the empty multitrace "empty_multitrace".

```
Lemma left_append_on_mu_cannot_be_nil :
  forall (mu:MultiTrace) (a:Action),
  ~ (left_append_action_on_multitrace mu a = empty_multitrace).
Proof.
intros mu a.
assert ( multitrace_sum_len (left_append_action_on_multitrace mu a)
         <> multitrace_sum_len empty_multitrace
       ).
{ rewrite (left_append_mu_increase_sum_len mu a).
  rewrite empty_mu_length.
  lia.
}
intro. rewrite H0 in H. contradiction.
Qed.
```

"left_append_action_on_multitrace_injective" states the injectivity of the "left_append_action_on_multitrace" concatenation operator.

```
Lemma left_append_action_on_multitrace_injective :
  forall (mu mu':MultiTrace) (a:Action),
    ( (left_append_action_on_multitrace mu a)
       = (left_append_action_on_multitrace mu' a)
    )
```

```
    -> (mu=mu').
```

Projecting traces into multitraces

We define the "project_as_multitrace" function, which projects a globally collected trace into a corresponding locally collected multitrace.

```
Fixpoint project_as_multitrace (t : Trace) : MultiTrace :=
match t with
| nil => empty_multitrace
| a :: t' => left_append_action_on_multitrace (project_as_multitrace t') a
end.
```

"only_nil_project_to_empty" is a Lemma stating that the only multitrace that can be projected to obtain the empty trace is the empty multitrace.

```
Lemma only_nil_project_to_empty :
  forall t:Trace,
     (project_as_multitrace t = empty_multitrace) -> (t=nil).
Proof.
intros t H.
dependent induction t.
- reflexivity.
- simpl in H.
  pose proof (left_append_on_mu_cannot_be_nil (project_as_multitrace t) a) as Hnil.
  rewrite H in Hnil.
  contradiction.
Qed.
```

"always_exist_projection" is a Lemma stating that any multitrace can be obtained by the projection of a trace.

```
Lemma always_exist_projection :
   forall (mu:MultiTrace),
      exists (t:Trace), ((project_as_multitrace t)=mu).
Proof.
intros mu.
remember (multitrace_sum_len mu) as n.
dependent induction n generalizing mu.
- symmetry in Heqn.
  apply empty_mu_length_equiv in Heqn.
  symmetry in Heqn. destruct Heqn.
  exists nil. simpl. reflexivity.
- pose proof (multitrace_cases mu) as H.
  destruct H.
  + apply empty_mu_length_equiv in H.
    rewrite H in Heqn.
    discriminate.
  + destruct H as (mu',H).
    destruct H as (a,H).
    specialize IHn with (mu:=mu').
    assert (n = multitrace_sum_len mu') as Hlen.
    { pose proof (left_append_mu_increase_sum_len mu' a) as Hinc.
      rewrite <- H in Hinc.
      rewrite <- Heqn in Hinc.
      lia.
    }
    apply IHn in Hlen.
    destruct Hlen as (t,Hproj).
    exists (a::t).
```

```
    simpl.
    rewrite Hproj.
    symmetry.
    assumption.
Qed.
```

"length_1_projection" is a Lemma stating that if the projection of a trace "t" is of the form "act.empty_multitrace" where "." is the concatenation operator "left_append_action_on_multitrace", then "t" must be the trace "act::nil" i.e. "act".

```
Lemma length_1_projection :
 forall (t:Trace) (a:Action),
 (left_append_action_on_multitrace empty_multitrace a = project_as_multitrace t)
  -> (t = a :: nil).
```

"projection_injective" is a Lemma stating the injectivity of the "project_as_multitrace" function.

```
Lemma projection_injective :
  forall (t1 t2:Trace),
      (project_as_multitrace t1 = project_as_multitrace t2)
      -> (t1=t2).
```

Interaction Language

We now introduce formally aspects of our interaction modelling language.

Syntax

We first introduce an enumerated (inductive) type "LoopKind" to enumerate the different kinds of loops our formalism specify.

```
Inductive LoopKind : Set :=
  |lstrict:LoopKind
  |lseq:LoopKind
  |lpar:LoopKind.
```

We then define our "Interaction" type, that inductively define interaction terms. From basic building blocks which can either be the empty interaction "interaction_empty" or actions (of type "Action"), we then use different operators to construct more complex interaction terms.

```
Inductive Interaction : Set :=
  interaction_empty:Interaction
  |interaction_act:Action->Interaction
  |interaction_strict:Interaction->Interaction->Interaction
  |interaction_seq:Interaction->Interaction->Interaction
  |interaction_par:Interaction->Interaction->Interaction
  |interaction_alt:Interaction->Interaction->Interaction
  |interaction_loop:LoopKind->Interaction->Interaction.
```

Static analysis of interaction terms

The "express_empty" function can statically determine (based on a static analyse of the interaction term) whether or not a given interaction can accept/express the empty trace i.e. if it can allow the empty execution (nothing happens). As can be seen, we used a "Fixpoint" to define "express_empty", and it returns a computational-logic "bool" value (as opposed to an intuitionistic-logic "Prop"). This therefore guarantees the deterministic character of "express_empty" and its termination in finite time (static function).

```
Fixpoint express_empty (i : Interaction) : bool :=
  match i with
    | interaction_empty => true
    | (interaction_act a) => false
    | (interaction_loop lk i1) => true
    | (interaction_alt i1 i2) => orb (express_empty i1) (express_empty i2)
    | (interaction_par i1 i2) => andb (express_empty i1) (express_empty i2)
    | (interaction_strict i1 i2) => andb (express_empty i1) (express_empty i2)
    | (interaction_seq i1 i2) => andb (express_empty i1) (express_empty i2)
  end.
```

Likewise, we define in the same manner the "avoids" function, which can statically determine whether or not a given interaction accepts execution that do not involve any action of a given lifeline (i.e. it "avoids" the lifeline).

```
Fixpoint avoids
        (i : Interaction)
        (l : L)
        : bool :=
  match i with
    | interaction_empty => true
    | (interaction_act a) => negb (Fin.eqb (lifeline a) l)
    | (interaction_loop lk i1) => true
    | (interaction_alt i1 i2) => orb (avoids i1 l) (avoids i2 l)
    | (interaction_par i1 i2) => andb (avoids i1 l) (avoids i2 l)
    | (interaction_strict i1 i2) => andb (avoids i1 l) (avoids i2 l)
    | (interaction_seq i1 i2) => andb (avoids i1 l) (avoids i2 l)
  end.
```

From a given interaction "i" and lifeline "l" such that "avoids(i,l)=true", the "prune" function computes statically (the return type "Interaction" is a deterministic inductive type) an interaction which accepts exactly all executions of "i" that do not involve the lifeline "l".

```
Fixpoint prune
        (i : Interaction)
        (l : L)
        : Interaction :=
  match i with
    | interaction_empty => interaction_empty
    | (interaction_act a) => (interaction_act a)
    | (interaction_loop lk i1) => match (avoids i1 l) with
                                  | true => interaction_loop lk (prune i1 l)
                                  | false => interaction_empty
                                  end
    | (interaction_alt i1 i2) => match (avoids i1 l) with
                                  | true => match (avoids i2 l) with
                                            | true => interaction_alt
                                                      (prune i1 l)
                                                      (prune i2 l)
                                            | false => prune i1 l
                                            end
                                  | false => prune i2 l
                                  end
    | (interaction_par i1 i2) => interaction_par (prune i1 l) (prune i2 l)
```

```
    | (interaction_strict i1 i2) => interaction_strict (prune i1 l) (prune i2 l)
    | (interaction_seq i1 i2) => interaction_seq (prune i1 l) (prune i2 l)
  end.
```

Managing positions with the Dewey Decimal Notation

Interaction terms have a structure which is that of a binary tree. Indeed, operators used to construct interactions have an arity of at most 2. As a result, we can navigate within interaction terms using a position system in {1,2}^* akin to that of the Dewey Decimal Notation. We define the "Position" inductive type which form words on {1,2}^*, with "epsilon" being the empty position.

```
Inductive Position : Set :=
  |epsilon:Position
  |left:Position->Position
  |right:Position->Position.
```

For practical reasons, that will be made clear in the following, we define a "left_front" operator which, for any list of "Position*Action" tuples, returns a similar list where every position from the original list have been shifted to the left. It can simply be explained by the notation `"left.[(p1,a1),...,(p2,a2)] = [(left p1,a1),...,(left p2,a2)]"`.

```
Fixpoint left_front (e1 : list (Position*Action)) : list (Position*Action) :=
  match e1 with
    | nil => nil
    | (p,a) :: tail => (left p,a) :: left_front tail
  end.
```

"left_front_ex" is a Lemma stating a simple characterization of "left_front" which is that, for a given list "fr" of tuples "Position*Action", for any couple "(p,a)" in "left_front(fr)", there exists a corresponding couple "(p1,a)" in "fr" such that "p=left p1".

```
Lemma left_front_ex :
  forall (fr:list (Position*Action)) (p:Position) (a:Action),
  ( In (p,a) (left_front fr) )
    -> ( exists (p1:Position), (p = left p1) /\ (In (p1,a) fr ) ).
Proof.
intros fr p a H.
induction fr.
- simpl in H. contradiction.
- destruct a0 as (p',a').
  simpl in H.
  destruct H.
  + exists p'.
    simpl.
    split.
    * inversion H. reflexivity.
    * left. inversion H. reflexivity.
  + apply IHfr in H. destruct H as (p1,H).
    exists p1.
    destruct H as (HA,HB).
    split.
    * assumption.
    * simpl. right. assumption.
Qed.
```

"left_front_ex2" is a Lemma stating another simple characterization of "left_front".

```
Lemma left_front_ex2 :
  forall (fr:list (Position*Action)) (p1:Position) (a:Action),
  ( In (left p1,a) (left_front fr) )
    <-> ( In (p1,a) fr ).
Proof.
intros fr p1 a.
split ; intros H.
- induction fr.
  + simpl in H. contradiction.
  + destruct a0 as (p0,a0). simpl in H.
    destruct H.
    * inversion H. destruct H1. destruct H2.
      simpl. left. reflexivity.
    * simpl. right. apply IHfr. assumption.
- induction fr.
  + simpl in H. contradiction.
  + destruct a0 as (p0,a0). simpl in H.
    destruct H.
    * inversion H. destruct H1. destruct H2.
      simpl. left. reflexivity.
    * simpl. right. apply IHfr. assumption.
Qed.
```

After "left_front", we define the symmetric "right_front" operator and the corresponding lemmas "right_front_ex" and "right_front_ex2". As their proofs are identical to that of "left_front_ex" and "left_front_ex2", we hide them in this document.

```
Fixpoint right_front (e2 : list (Position*Action)) : list (Position*Action) :=
  match e2 with
    | nil => nil
    | (p,a) :: tail => (right p,a) :: right_front tail
  end.

Lemma right_front_ex :
  forall (fr:list (Position*Action)) (p:Position) (a:Action),
  ( In (p,a) (right_front fr) )
    -> ( exists (p2:Position), (p = right p2) /\ (In (p2,a) fr ) ).

Lemma right_front_ex2 :
  forall (fr:list (Position*Action)) (p2:Position) (a:Action),
  ( In (right p2,a) (right_front fr) )
    <-> ( In (p2,a) fr ).
```

We now define a filter function "filterfront_on_avoid", which, for any interaction "i" and list "fr" of "Position*Action" tuples returns the sublist of "fr" such that each element "(p,a)" verifies "avoids(i,lifeline(a))=true". "filterfront_on_avoid" therefore filters elements of "fr", keeping only those of which the action "a" is s.t. "avoids(i,lifeline(a))=true".

```
Fixpoint filterfront_on_avoid (i : Interaction) (fr : list (Position*Action))
  : list (Position*Action)
  :=
  match fr with
    | nil => nil
    | (p,a) :: fr' => if (avoids i (lifeline a))
                      then (p,a) :: (filterfront_on_avoid i fr')
                      else (filterfront_on_avoid i fr')
  end.
```

"filterfront_on_avoid_ex" and "filterfront_on_avoid_ex2" are lemmas stating two characterization of "filterfront_on_avoid" in combination with "right_front".

```
Lemma filterfront_on_avoid_ex :
  forall (i:Interaction)
         (fr:list (Position*Action))
         (p:Position)
         (a:Action),
  ( In (p, a) (filterfront_on_avoid i (right_front fr)) )
  -> ( exists (p2:Position),
          (is_true (avoids i (lifeline a)))
          /\ (p = right p2) /\ (In (p2,a) fr )
     ).
Proof.
intros i fr p a H.
induction fr.
- simpl in H. contradiction.
- destruct a0 as (p',a').
  simpl in H.
  remember (avoids i (lifeline a')) as Hav.
  destruct Hav.
  + simpl in H.
    destruct H.
    * inversion H.
      symmetry in H2. destruct H2.
      destruct H1.
      exists p'.
      simpl.
      split. { rewrite <- HeqHav. reflexivity. }
             { split. { reflexivity. } {left. reflexivity. } }
    * apply IHfr in H.
      destruct H as (p2,H).
      destruct H as (HA,HB).
      destruct HB as (HB,HC).
      exists p2.
      split.
      { assumption. }
      { split. { assumption. } { simpl. right. assumption. } }
  + apply IHfr in H.
    destruct H as (p2,H).
    destruct H as (HA,HB).
    destruct HB as (HB,HC).
    exists p2.
    split.
    { assumption. }
    { split. { assumption. } { simpl. right. assumption. } }
Qed.

Lemma filterfront_on_avoid_ex2 :
  forall (i:Interaction)
         (fr:list (Position*Action))
         (p2:Position)
         (a:Action),
  ( In (right p2, a) (filterfront_on_avoid i (right_front fr)) )
  <-> ( (In (p2,a) fr) /\ (is_true (avoids i (lifeline a))) ).
Proof.
intros i fr p2 a.
split ; intros H.
- induction fr.
  + simpl in H. contradiction.
  + destruct a0 as (p0,a0). simpl in H.
    remember (avoids i (lifeline a0)) as Hav.
    destruct Hav.
    { simpl in H.
      destruct H.
      - inversion H. destruct H1. destruct H2.
        simpl. split.
        + left. reflexivity.
        + unfold is_true. symmetry. assumption.
```

```
      - simpl. split.
        + right. apply IHfr. assumption.
        + apply IHfr. assumption.
    }
    { simpl. split.
      - right. apply IHfr. assumption.
      - apply IHfr. assumption.
    }
- induction fr.
  + simpl in H. destruct H as (HA,HB). contradiction.
  + destruct a0 as (p0,a0).
    destruct H as (HA,HB).
    simpl in HA.
    destruct HA.
    { inversion H.
      destruct H1. destruct H2.
      simpl.
      remember (avoids i (lifeline a0)) as Hav.
      destruct Hav.
      - simpl. left. reflexivity.
      - unfold is_true in HB. discriminate.
    }
    { simpl.
      remember (avoids i (lifeline a0)) as Hav.
      destruct Hav.
      - simpl. right. apply IHfr. split; assumption.
      - apply IHfr. split ; assumption.
    }
Qed.
```

Identification & Execution of immediately executable actions

The manipulation of lists of "Position*Action" tuples in the previous section now allows us to define the notion of "frontier". For any interaction "i", "frontier i" is the list of "Position*Action" tuples "(p,a)" s.t. "a" is an immediately executable action of "i" and is to be found at position "p" within "i".

```
Fixpoint frontier (i : Interaction) : list (Position*Action) :=
  match i with
    | interaction_empty => nil
    | (interaction_act a) => (epsilon,a) :: nil
    | (interaction_loop lk i1) => left_front (frontier i1)
    | (interaction_alt i1 i2) => (left_front (frontier i1)) ++
                                 (right_front (frontier i2))
    | (interaction_par i1 i2) => (left_front (frontier i1)) ++
                                 (right_front (frontier i2))
    | (interaction_strict i1 i2) => match (express_empty i1) with
                                    | true => (left_front (frontier i1)) ++
                                              (right_front (frontier i2))
                                    | false => left_front (frontier i1)
                                    end
    | (interaction_seq i1 i2) => (left_front (frontier i1)) ++
                                 (filterfront_on_avoid i1 (right_front (frontier
i2)))
  end.
```

Any frontier element "(p,a)" of an interaction "i" can then be used to compute a next interaction "i'" s.t. "i'" exactly accepts all the continuations of executions of "i" that start with the execution of action "a" at position "p". In the paper version, we defined a constructive execution function. Here, we define an inductive proposition "is_next_of" s.t. "is_next_of i p a i'" states that "i'" is obtained by the execution of "a" at position "p" in "i". The fact that

we have defined "is_next_of" as an inductive proposition instead of as a Fixpoint do not imply properties of:

- existence (i.e. s.t. there exists such an interaction "i'" for frontier elements "(p,a)")
- and unicity (i.e. s.t. if "is_next_of i p a ia" and "is_next_of i p a ib" then we must have "ia=ib")

To counter that, we will introduce and prove existence and unicity theorems.

```
Inductive is_next_of : Interaction -> Position -> Action -> Interaction -> Prop :=
|execute_at_leaf : forall (a:Action), (is_next_of (interaction_act a) epsilon a
interaction_empty)
|execute_left_alt : forall (p1:Position) (a:Action) (i1 i2 i1' : Interaction),
                          (is_next_of i1 p1 a i1')
                             -> (is_next_of (interaction_alt i1 i2) (left p1) a i1')
|execute_right_alt : forall (p2:Position) (a:Action) (i1 i2 i2' : Interaction),
                          (is_next_of i2 p2 a i2')
                             -> (is_next_of (interaction_alt i1 i2) (right p2) a i2')
|execute_left_par : forall (p1:Position) (a:Action) (i1 i2 i1' : Interaction),
                          (is_next_of i1 p1 a i1')
                             -> (is_next_of
                                       (interaction_par i1 i2)
                                       (left p1)
                                       a
                                       (interaction_par i1' i2)
                                   )
|execute_right_par : forall (p2:Position) (a:Action) (i1 i2 i2' : Interaction),
                          (is_next_of i2 p2 a i2')
                             -> (is_next_of
                                       (interaction_par i1 i2)
                                       (right p2)
                                       a
                                       (interaction_par i1 i2')
                                   )
|execute_loop_strict : forall (p1:Position) (a:Action) (i1 i1':Interaction),
                          (is_next_of i1 p1 a i1')
                             -> (is_next_of
                                       (interaction_loop lstrict i1)
                                       (left p1)
                                       a
                                       (interaction_strict i1' (interaction_loop
lstrict i1))
                                   )
|execute_loop_seq : forall (p1:Position) (a:Action) (i1 i1':Interaction),
                          (is_next_of i1 p1 a i1')
                             -> (is_next_of
                                       (interaction_loop lseq i1)
                                       (left p1)
                                       a
                                       (interaction_seq i1' (interaction_loop lseq
i1))
                                   )
|execute_loop_par : forall (p1:Position) (a:Action) (i1 i1':Interaction),
                          (is_next_of i1 p1 a i1')
                             -> (is_next_of
                                       (interaction_loop lpar i1)
                                       (left p1)
                                       a
                                       (interaction_par i1' (interaction_loop lpar
i1))
                                   )
|execute_left_strict : forall (p1:Position) (a:Action) (i1 i2 i1' : Interaction),
                         (is_next_of i1 p1 a i1')
                            -> (is_next_of
                                      (interaction_strict i1 i2)
                                      (left p1)
                                      a
```

```
                                    (interaction_strict i1' i2)
                                )
|execute_right_strict : forall (p2:Position) (a:Action) (i1 i2 i2' : Interaction),
                          ((is_next_of i2 p2 a i2')/\(is_true (express_empty i1)))
                             -> (is_next_of
                                    (interaction_strict i1 i2)
                                    (right p2)
                                    a
                                    i2'
                                )
|execute_left_seq : forall (p1:Position) (a:Action) (i1 i2 i1' : Interaction),
                          (is_next_of i1 p1 a i1')
                             -> (is_next_of
                                    (interaction_seq i1 i2)
                                    (left p1)
                                    a
                                    (interaction_seq i1' i2)
                                )
|execute_right_seq : forall (p2:Position) (a:Action) (i1 i2 i2' : Interaction),
                          ( (is_next_of i2 p2 a i2')
                             /\(is_true (avoids i1 (lifeline a)))
                          )
                             -> (is_next_of
                                    (interaction_seq i1 i2)
                                    (right p2)
                                    a
                                    (interaction_seq (prune i1 (lifeline a)) i2')
                                ).
```

"next_of_existence" is a Lemma which states the existence of a follow-up interaction "i'" for any frontier element "(p,a)". Its proof relies on an induction on the structure of interaction terms.

```
Lemma next_of_existence :
  forall (i:Interaction) (p:Position) (a:Action),
    (In (p,a) (frontier i)) <-> (exists (i':Interaction), (is_next_of i p a i')).
Proof.
intros i p a.
dependent induction i.
- split ; intros H.
  + simpl in H. contradiction.
  + destruct H as (i',H).
    inversion H.
- split ; intros H.
  + simpl in H.
    destruct H.
    * inversion H.
      destruct H1. destruct H2.
      exists interaction_empty.
      apply execute_at_leaf.
    * contradiction.
  + destruct H as (i',H).
    simpl.
    left.
    inversion H.
    split.
- split ; intros H.
  + simpl in H.
    remember (express_empty i1) as Hexp.
    destruct Hexp.
    * apply in_app_or in H.
      destruct H.
      { apply left_front_ex in H.
        destruct H as (p1,H).
        destruct H as (HA,HB).
```

```
        apply IHi1 in HB.
        destruct HB as (i1', HB).
        exists (interaction_strict i1' i2).
        rewrite HA.
        apply execute_left_strict.
        assumption.
      }
      { apply right_front_ex in H.
        destruct H as (p2,H).
        destruct H as (HA,HB).
        apply IHi2 in HB.
        destruct HB as (i2', HB).
        exists i2'.
        rewrite HA.
        apply execute_right_strict.
        split.
        - assumption.
        - unfold is_true. symmetry in HeqHexp.
          assumption.
      }
    * apply left_front_ex in H.
      destruct H as (p1,H).
      destruct H as (HA,HB).
      apply IHi1 in HB.
      destruct HB as (i1', HB).
      exists (interaction_strict i1' i2).
      rewrite HA.
      apply execute_left_strict.
      assumption.
  + destruct H as (i',H).
    inversion H.
    * destruct H0. destruct H2. destruct H3.
      simpl.
      remember (express_empty i0) as Hexp.
      destruct Hexp.
      { apply in_or_app.
        left.
        apply left_front_ex2.
        apply IHi1. exists i1'.
        assumption.
      }
      { apply left_front_ex2.
        apply IHi1.
        exists i1'. assumption.
      }
    * destruct H0. destruct H2. destruct H3. destruct H4.
      simpl.
      remember (express_empty i0) as Hexp.
      destruct Hexp.
      { apply in_or_app.
        right.
        apply right_front_ex2.
        apply IHi2. exists i2'.
        destruct H5.
        assumption.
      }
      { destruct H5. inversion H2. }
- split ; intros H.
  + simpl in H.
    remember (avoids i1 (lifeline a)) as Hexp.
    destruct Hexp.
    * apply in_app_or in H.
      destruct H.
      { apply left_front_ex in H.
        destruct H as (p1,H).
        destruct H as (HA,HB).
        apply IHi1 in HB.
```

```
        destruct HB as (i1', HB).
        exists (interaction_seq i1' i2).
        rewrite HA.
        apply execute_left_seq.
        assumption.
      }
      { apply filterfront_on_avoid_ex in H.
        destruct H as (p2,H).
        destruct H as (HA,HB).
        destruct HB as (HB,HC).
        apply IHi2 in HC.
        destruct HC as (i2', HC).
        exists (interaction_seq (prune i1 (lifeline a)) i2').
        rewrite HB.
        apply execute_right_seq.
        split.
        - assumption.
        - unfold is_true. symmetry in HeqHexp.
          assumption.
      }
    * apply in_app_or in H.
      destruct H.
      { apply left_front_ex in H.
        destruct H as (p1,H).
        destruct H as (HA,HB).
        apply IHi1 in HB.
        destruct HB as (i1', HB).
        exists (interaction_seq i1' i2).
        rewrite HA.
        apply execute_left_seq.
        assumption.
      }
      { simpl in H. apply filterfront_on_avoid_ex in H.
        destruct H as (p2,H).
        destruct H as (HA,HB).
        destruct HB as (HB,HC).
        unfold is_true in HA.
        rewrite <- HeqHexp in HA.
        discriminate.
      }
  + destruct H as (i',H).
    inversion H.
    * destruct H0. destruct H2. destruct H3.
      simpl.
      apply in_or_app.
      left.
      apply left_front_ex2.
      apply IHi1.
      exists i1'.
      assumption.
    * destruct H0. destruct H2. destruct H3.
      simpl.
      apply in_or_app.
      right.
      apply filterfront_on_avoid_ex2.
      destruct H5 as (HA,HB).
      split.
      { apply IHi2. exists i2'. assumption. }
      { assumption. }
- split ; intros H.
  + simpl in H.
    apply in_app_or in H.
    destruct H.
    { apply left_front_ex in H.
      destruct H as (p1,H).
      destruct H as (HA,HB).
      apply IHi1 in HB.
```

```
      destruct HB as (i1', HB).
      exists (interaction_par i1' i2).
      rewrite HA.
      apply execute_left_par.
      assumption.
    }
    { apply right_front_ex in H.
      destruct H as (p2,H).
      destruct H as (HA,HB).
      apply IHi2 in HB.
      destruct HB as (i2', HB).
      exists (interaction_par i1 i2').
      rewrite HA.
      apply execute_right_par.
      assumption.
    }
  + destruct H as (i',H).
    inversion H.
    * destruct H0. destruct H2. destruct H3.
      simpl.
      apply in_or_app.
      left.
      apply left_front_ex2.
      apply IHi1. exists i1'.
      assumption.
   * destruct H0. destruct H2. destruct H3.
      simpl.
      apply in_or_app.
      right.
      apply right_front_ex2.
      apply IHi2. exists i2'.
      assumption.
- split ; intros H.
  + simpl in H.
    apply in_app_or in H.
    destruct H.
    { apply left_front_ex in H.
      destruct H as (p1,H).
      destruct H as (HA,HB).
      apply IHi1 in HB.
      destruct HB as (i1', HB).
      exists i1'.
      rewrite HA.
      apply execute_left_alt.
      assumption.
    }
    { apply right_front_ex in H.
      destruct H as (p2,H).
      destruct H as (HA,HB).
      apply IHi2 in HB.
      destruct HB as (i2', HB).
      exists i2'.
      rewrite HA.
      apply execute_right_alt.
      assumption.
    }
  + destruct H as (i',H).
    inversion H.
    * destruct H0. destruct H2. destruct H3. destruct H4.
      simpl.
      apply in_or_app.
      left.
      apply left_front_ex2.
      apply IHi1.
      exists i1'.
      assumption.
   * destruct H0. destruct H2. destruct H3. destruct H4.
```

```
      simpl.
      apply in_or_app.
      right.
      apply right_front_ex2.
      apply IHi2. exists i2'.
      assumption.
- split ; intros H.
  + simpl in H.
    apply left_front_ex in H.
    destruct H as (p1,H).
    destruct H as (HA,HB).
    apply IHi in HB.
    destruct HB as (i1', HB).
    induction l.
    * exists (interaction_strict i1' (interaction_loop lstrict i)).
      rewrite HA.
      apply execute_loop_strict.
      assumption.
    * exists (interaction_seq i1' (interaction_loop lseq i)).
      rewrite HA.
      apply execute_loop_seq.
      assumption.
    * exists (interaction_par i1' (interaction_loop lpar i)).
      rewrite HA.
      apply execute_loop_par.
      assumption.
  + destruct H as (i',H).
    inversion H.
    * destruct H0. destruct H2. destruct H3.
      simpl.
      apply left_front_ex2.
      apply IHi.
      exists i1'.
      assumption.
   * destruct H0. destruct H2. destruct H3.
      simpl.
      apply left_front_ex2.
      apply IHi.
      exists i1'.
      assumption.
   * destruct H0. destruct H2. destruct H3.
      simpl.
      apply left_front_ex2.
      apply IHi.
      exists i1'.
      assumption.
Qed.
```

"next_of_unicity" is a Lemma which states the unicity of follow-up interactions for any frontier element "(p,a)". Its proof relies on an induction on the structure of interaction terms.

```
Lemma next_of_unicity :
  forall (p:Position) (a1 a2:Action) (i0 i1 i2:Interaction),
     ( (is_next_of i0 p a1 i1) /\ (is_next_of i0 p a2 i2) )
     -> ((a1=a2) /\ (i1=i2)).
Proof.
intros p a1 a2 i0 i1 i2 H.
destruct H as (H1,H2).
dependent induction i0 generalizing p.
- inversion H1.
- inversion H1.
  inversion H2.
  destruct H0. destruct H4. destruct H6. destruct H8.
  split ; reflexivity.
```

```
- inversion H1.
  + destruct H. destruct H3. destruct H4.
    inversion H2.
    * destruct H. destruct H4. destruct H7.
      rewrite <- H3 in H0. inversion H0. destruct H4.
      specialize IHi0_1 with (i1:=i1') (i2:=i1'0) (p:=p1).
      split.
      { apply IHi0_1 ; assumption. }
      { f_equal. apply IHi0_1 ; assumption. }
    * destruct H. destruct H4. destruct H7. destruct H8.
      rewrite <- H3 in H0. inversion H0.
  + destruct H. destruct H3. destruct H4. destruct H5.
    inversion H2.
    * destruct H. destruct H4. destruct H5.
      rewrite <- H3 in H0. inversion H0.
    * destruct H. destruct H4. destruct H5. destruct H7.
      rewrite <- H3 in H0. inversion H0. destruct H4.
      specialize IHi0_2 with (i1:=i2') (i2:=i2'0) (p:=p2).
      destruct H6 as (Hr1,Hr2).
      destruct H8 as (Hr3,Hr4).
      split.
      { apply IHi0_2 ; assumption. }
      { apply IHi0_2 ; assumption. }
- inversion H1.
  + destruct H. destruct H3. destruct H4.
    inversion H2.
    * destruct H. destruct H4. destruct H7.
      rewrite <- H3 in H0. inversion H0. destruct H4.
      specialize IHi0_1 with (i1:=i1') (i2:=i1'0) (p:=p1).
      split.
      { apply IHi0_1 ; assumption. }
      { f_equal. apply IHi0_1 ; assumption. }
    * destruct H. destruct H4. destruct H7.
      rewrite <- H3 in H0. inversion H0.
  + destruct H. destruct H3. destruct H4.
    inversion H2.
    * destruct H. destruct H4. destruct H7.
      rewrite <- H3 in H0. inversion H0.
    * destruct H. destruct H4. destruct H7.
      rewrite <- H3 in H0. inversion H0. destruct H4.
      destruct H6 as (H_1_1,H_1_2).
      destruct H9 as (H_2_1,H_2_2).
      specialize IHi0_2 with (i1:=i2') (i2:=i2'0) (p:=p2).
      apply IHi0_2 in H_1_1.
      { destruct H_1_1 as (H4,H_right). destruct H4.
        split.
        - reflexivity.
        - f_equal.
          destruct H_right.
          reflexivity.
      }
      { assumption. }
- inversion H1.
  + destruct H. destruct H3. destruct H4.
    inversion H2.
    * destruct H. destruct H4. destruct H7.
      rewrite <- H3 in H0. inversion H0. destruct H4.
      specialize IHi0_1 with (i1:=i1') (i2:=i1'0) (p:=p1).
      split.
      { apply IHi0_1 ; assumption. }
      { f_equal. apply IHi0_1 ; assumption. }
    * destruct H. destruct H4. destruct H7.
      rewrite <- H3 in H0. inversion H0.
  + destruct H. destruct H3. destruct H4.
    inversion H2.
    * destruct H. destruct H4. destruct H7.
      rewrite <- H3 in H0. inversion H0.
```

```
    * destruct H. destruct H4. destruct H7.
      rewrite <- H3 in H0. inversion H0. destruct H4.
      specialize IHi0_2 with (i1:=i2') (i2:=i2'0) (p:=p2).
      split.
      { apply IHi0_2 ; assumption. }
      { f_equal. apply IHi0_2 ; assumption. }
- inversion H1.
  + destruct H. destruct H3. destruct H4. destruct H5.
    inversion H2.
    * destruct H. destruct H4. destruct H5. destruct H7.
      rewrite <- H3 in H0. inversion H0. destruct H4.
      specialize IHi0_1 with (i2:=i1') (i3:=i1'0) (p:=p1).
      split.
      { apply IHi0_1 ; assumption. }
      { apply IHi0_1 ; assumption. }
    * destruct H. destruct H4. destruct H5. destruct H7.
      rewrite <- H3 in H0. inversion H0.
  + destruct H. destruct H3. destruct H4. destruct H5.
    inversion H2.
    * destruct H. destruct H4. destruct H5. destruct H7.
      rewrite <- H3 in H0. inversion H0.
    * destruct H. destruct H4. destruct H5. destruct H7.
      rewrite <- H3 in H0. inversion H0. destruct H4.
      specialize IHi0_2 with (i1:=i2') (i2:=i2'0) (p:=p2).
      split ; apply IHi0_2 ; assumption.
- inversion H1.
  + destruct H. destruct H3. destruct H4.
    inversion H2.
    * destruct H. destruct H7.
      rewrite <- H4 in H0. inversion H0. destruct H7.
      specialize IHi0 with (p:=p1) (i1:=i1') (i2:=i1'0).
      split.
      { apply IHi0 ; assumption. }
      { f_equal. apply IHi0 ; assumption. }
  + destruct H. destruct H3. destruct H4.
    inversion H2.
    * destruct H. destruct H7.
      rewrite <- H4 in H0. inversion H0. destruct H7.
      specialize IHi0 with (p:=p1) (i1:=i1') (i2:=i1'0).
      split.
      { apply IHi0 ; assumption. }
      { f_equal. apply IHi0 ; assumption. }
  + destruct H. destruct H3. destruct H4.
    inversion H2.
    * destruct H. destruct H7.
      rewrite <- H4 in H0. inversion H0. destruct H7.
      specialize IHi0 with (p:=p1) (i1:=i1') (i2:=i1'0).
      split.
      { apply IHi0 ; assumption. }
      { f_equal. apply IHi0 ; assumption. }
Qed.
```

Operational Semantics

We can now introduce our small-step operational semantics based on subsequent executions of frontier actions within interactions.

Definitions

"is_accept" is an inductive proposition s.t. for any interaction "i" and trace "t", "is_accept i t" states the membership of "t" to the trace semantics of "i".

```
Inductive is_accept : Interaction -> Trace -> Prop :=
 | accept_empty : forall (i :Interaction),
                     (is_true (express_empty i))
                     -> (is_accept i nil)
 | accept_action : forall (i i':Interaction) (p:Position) (a:Action) (t:Trace),
                     (is_next_of i p a i')/\(is_accept i' t)
                     -> (is_accept i (cons a t) ).
```

Likewise, "is_accept_mult" is an inductive proposition s.t. for any interaction "i" and multitrace "mu", "is_accept_mult i mu" states the membership of "mu" to the multitrace semantics of "i".

```
Inductive is_accept_mult : Interaction -> MultiTrace -> Prop :=
 | mult_accept_empty : forall (i :Interaction),
                          (is_true (express_empty i))
                              -> (is_accept_mult i empty_multitrace)
 | mult_accept_action : forall (i i':Interaction)
                               (p:Position)
                               (a:Action)
                               (mu:MultiTrace),
                          ( (is_next_of i p a i')
                            /\ (is_accept_mult i' mu)
                          )
                          ->
                          ( is_accept_mult
                             i
                             (left_append_action_on_multitrace mu a)
                          ).
```

Characterization with projections

In the following, we prove that the semantics defined by "is_accept_mult" is in facts a projection of the semantics defined by "is_accept". The first theorem "accept_implies_mult_accept" states that for any interaction "i" and trace "t", if "is_accept i t" then "is_accept_mult i (project_as_multitrace t)".

```
Theorem accept_implies_mult_accept :
  forall (i:Interaction) (t:Trace),
     (is_accept i t) -> (is_accept_mult i (project_as_multitrace t)).
Proof.
intros i t H.
dependent induction t generalizing i.
- simpl.
  apply mult_accept_empty.
  inversion H.
  assumption.
- simpl.
  inversion H.
  destruct H1. destruct H0. destruct H3.
  apply (mult_accept_action i0 i' p a0 (project_as_multitrace t0)).
  destruct H2 as (HA,HB).
  split.
  + assumption.
  + apply IHt. assumption.
Qed.
```

Then, the second theorem, "mult_accept_implies_accept" proves the other direction i.e. that for any interaction "i" and multitrace "mu", if "is_accept_mult i mu" then there must

exists a trace "t" such that "mu" is the projection of "t" and "is_accept i t".

```
Theorem mult_accept_implies_accept :
  forall (i:Interaction) (mu:MultiTrace),
     (is_accept_mult i mu)
     ->
     ( exists (t:Trace),
       ( (project_as_multitrace t) = mu )
       /\ ( is_accept i t )
     ).
Proof.
intros i mu.
remember (multitrace_sum_len mu) as n.
dependent induction n generalizing i mu.
- intros H.
  symmetry in Heqn.
  apply empty_mu_length_equiv in Heqn.
  rewrite Heqn.
  symmetry in Heqn. destruct Heqn.
  exists nil.
  split.
  + simpl. reflexivity.
  + apply accept_empty.
    inversion H.
    * assumption.
    * pose proof (left_append_on_mu_cannot_be_nil mu a) as Hnil.
      contradiction.
- intros Hyp.
  inversion Hyp.
  + destruct H1.
    symmetry in H0. destruct H0.
    exists nil.
    split.
    * simpl. reflexivity.
    * apply accept_empty. assumption.
  + symmetry in H0. destruct H0.
    destruct H as (HA,HB).
    specialize IHn with (i:=i') (mu:=mu0).
    pose proof (left_append_mu_increase_sum_len mu0 a) as Hlen.
    rewrite H1 in Hlen.
    rewrite <- Heqn in Hlen.
    simpl in Hlen.
    assert (n = multitrace_sum_len mu0) as Hlen2.
    { lia. }
    apply IHn in Hlen2.
    * destruct Hlen2 as (t,Hacc).
      destruct Hacc as (HC,HD).
      exists (a::t).
      split.
      { simpl. rewrite HC. reflexivity. }
      { apply (accept_action i i' p a).
        split ; assumption.
      }
    * assumption.
Qed.
```

Finally, we can conclude on the equivalence of both semantics via the projection operator with the "accept_mult_is_proj_accept" theorem.

```
Theorem accept_mult_is_proj_accept :
  forall (i:Interaction) (t:Trace),
     (is_accept i t) <-> (is_accept_mult i (project_as_multitrace t)).
Proof.
intros i t.
```

```
split.
- apply accept_implies_mult_accept.
- remember (project_as_multitrace t) as mu.
  intros H.
  apply mult_accept_implies_accept in H.
  destruct H as (t',H).
  destruct H as (H1,H2).
  rewrite Heqmu in H1.
  apply projection_injective in H1.
  destruct H1.
  assumption.
Qed.
```

MultiTrace membership analysis algorithm

In this section, we will define our multitrace analysis algorithm, which is able to solve the membership problem (w.r.t. the multitrace semantics "is_accept_mult") for any interaction "i" and multitrace "mu".

Matches

Firstly, we define function "match_on_multi_trace" which is able to tell if there is a match between a given action and the heads of the component traces of a given multitrace. "match_on_multi_trace" relies on folding the tool function "match_on_trace" on its structure which is that of a vector of traces.

For "match_on_multi_trace act mu" to return "true", it suffices that one of its component trace "t" is s.t. "t=act.t'".

```
Fixpoint match_on_trace (a:Action) (m:bool) (t:Trace): bool :=
 match t with
 | nil => (orb m false)
 | h :: t' => (orb (eq_action a h) (match_on_trace a m t') )
end.

Definition match_on_multi_trace (a:Action) (mu:MultiTrace) : bool :=
  VectorDef.fold_left (match_on_trace a) false mu.
```

We then define "match_in_front" which tells if there is a match between any frontier action of a given interaction and the component heads of a given multitrace.

```
Fixpoint match_in_front_inner (fr:list(Position*Action)) (mu:MultiTrace) : bool :=
match fr with
| nil => false
| (p,a) :: fr' => orb (match_on_multi_trace a mu) (match_in_front_inner fr' mu)
end.

Definition match_in_front (i:Interaction) (mu:MultiTrace) :=
  match_in_front_inner (frontier i) mu.
```

"match_in_front_char" is a Lemma characterizes "match_in_front" s.t. if "match_in_front i mu" is true, then there exists an action "a" which is at the same time a frontier action of "i" and the head of a component of "mu" (and therefore "mu" can be obtained by left concatenation of this action "a" on a shorter "mu'" multitrace).

```
Lemma match_in_front_char :
```

```
  forall (i:Interaction) (mu:MultiTrace),
    (
        (is_true (match_in_front i mu))
        <->
        ( exists (mu':MultiTrace) (a:Action) (p:Position),
          (mu=(left_append_action_on_multitrace mu' a)) /\ (In (p,a) (frontier i) )
        )
    ).
```

The algorithm proper

Below is defined the analysis algorithm itself, as a set of rules defining a graph.

```
Inductive Verdict : Set :=
  |verd_Paccept:Verdict
  |verd_Fail:Verdict.

Inductive mu_analysis :
  Verdict -> Interaction -> MultiTrace -> Prop :=
| ana_rule_r1 : forall (i:Interaction),
                  ( is_true (express_empty i) )
                     -> (mu_analysis verd_Paccept i empty_multitrace)
| ana_rule_r2 : forall (i:Interaction),
                  (~ (is_true (express_empty i)) )
                     -> (mu_analysis verd_Fail i empty_multitrace)
| ana_rule_r3 : forall (i i':Interaction) (p:Position) (a:Action) (mu:MultiTrace)
(v:Verdict),
                  ( (is_next_of i p a i')
                     /\ (mu_analysis v i' mu)
                  )
                    -> (mu_analysis
                          v
                          i
                          (left_append_action_on_multitrace mu a)
                        )
| ana_rule_r4 : forall (i:Interaction) (mu:MultiTrace),
                  ( ~(is_true (match_in_front i mu)) )
                  -> (mu_analysis verd_Fail i mu).
```

Proof of conformity w.r.t. trace semantics

"accept_implies_verd_Paccept" is a theorem which states that for any interaction "i" and global trace "t",

- if "is_accept i t" i.e. "t" is in the trace semantics of "i"
- then "mu_analysis verd_Paccept i (project_as_multitrace t)" i.e. when we apply the multitrace analysis algorithm on the projection of "t" as a multitrace w.r.t. the interaction model "i", it returns the "verd_Paccept" verdict.

```
Theorem accept_implies_verd_Paccept :
   forall (i : Interaction) (t:Trace),
      (is_accept i t) -> (mu_analysis verd_Paccept i (project_as_multitrace t)).
Proof.
intros i t H.
dependent induction t generalizing i.
- simpl.
  apply ana_rule_r1.
  inversion H.
```

```
  assumption.
- simpl.
  inversion H.
  destruct H1. destruct H0. destruct H3.
  apply (ana_rule_r3 i0 i' p a0 (project_as_multitrace t0)).
  destruct H2 as (HA,HB).
  split.
  + assumption.
  + apply IHt. assumption.
Qed.
```

"verd_Paccept_implies_accept" is a theorem which states that for any interaction "i" and multitrace "mu",

- if "mu_analysis verd_Paccept i mu" i.e. when we apply the multitrace analysis algorithm on "mu" w.r.t. the interaction model "i", it returns the "verd_Paccept" verdict
- then there exists a trace "t" which projects to "mu" and which is in the trace semantics of "i" (i.e. "is_accept i t").

```
Theorem verd_Paccept_implies_accept :
  forall (i:Interaction) (mu:MultiTrace),
     (mu_analysis verd_Paccept i mu)
     ->
     ( exists (t:Trace),
       ( (project_as_multitrace t) = mu )
       /\ ( is_accept i t )
     ).
Proof.
intros i mu.
remember (multitrace_sum_len mu) as n.
dependent induction n generalizing i mu.
- intros H.
  symmetry in Heqn.
  apply empty_mu_length_equiv in Heqn.
  rewrite Heqn.
  symmetry in Heqn. destruct Heqn.
  exists nil.
  split.
  + simpl. reflexivity.
  + apply accept_empty.
    inversion H.
    * assumption.
    * pose proof (left_append_on_mu_cannot_be_nil mu a) as Hnil.
      contradiction.
- intros Hyp.
  inversion Hyp.
  + destruct H1.
    destruct H2.
    exists nil.
    split.
    * simpl. reflexivity.
    * apply accept_empty. assumption.
  + symmetry in H0. destruct H0.
    symmetry in H1. destruct H1.
    destruct H as (HA,HB).
    specialize IHn with (i:=i') (mu:=mu0).
    pose proof (left_append_mu_increase_sum_len mu0 a) as Hlen.
    rewrite H2 in Hlen.
    rewrite <- Heqn in Hlen.
    simpl in Hlen.
    assert (n = multitrace_sum_len mu0) as Hlen2.
    { lia. }
    apply IHn in Hlen2.
    * destruct Hlen2 as (t,Hacc).
      destruct Hacc as (HC,HD).
```

```
      exists (a::t).
      split.
      { simpl. rewrite HC. reflexivity. }
      { apply (accept_action i i' p a).
        split ; assumption.
      }
    * assumption.
Qed.
```

Then, the conformity of "mu_analysis" w.r.t. the trace semantics "is_accept" is stated by the theorem "verd_Paccept_is_proj_accept" below, which is implied by the 2 previous.

```
Theorem verd_Paccept_is_proj_accept :
  forall (i:Interaction) (t:Trace),
     (is_accept i t) <-> (mu_analysis verd_Paccept i (project_as_multitrace t)).
Proof.
intros i t.
split.
- apply accept_implies_verd_Paccept.
- remember (project_as_multitrace t) as mu.
  intros H.
  apply verd_Paccept_implies_accept in H.
  destruct H as (t',H).
  destruct H as (H1,H2).
  rewrite Heqmu in H1.
  apply projection_injective in H1.
  destruct H1.
  assumption.
Qed.
```

Proof of conformity w.r.t. multitrace semantics

Finally, the conformity of "mu_analysis" w.r.t. the multitrace semantics "is_accept_mult" is stated by the theorem "verd_Paccept_is_mult_accept" below. It is implied by the "verd_Paccept_is_proj_accept" theorem and the characterization of "is_accept_mult" as the projection of "is_accept" (Lemma "accept_mult_is_proj_accept").

```
Theorem verd_Paccept_is_mult_accept :
  forall (i:Interaction) (mu:MultiTrace),
     (is_accept_mult i mu) <-> (mu_analysis verd_Paccept i mu).
Proof.
intros i mu.
split ; intros H.
- pose proof (always_exist_projection mu) as Ht.
  destruct Ht as (t,Ht).
  destruct Ht.
  apply accept_mult_is_proj_accept in H.
  apply verd_Paccept_is_proj_accept.
  assumption.
- pose proof (always_exist_projection mu) as Ht.
  destruct Ht as (t,Ht).
  destruct Ht.
  apply verd_Paccept_is_proj_accept in H.
  apply accept_mult_is_proj_accept.
  assumption.
Qed.
```