Copland Attestation Terms: Semantics and Coq Proofs

John D. Ramsdell Paul D. Rowe

The MITRE Corporation

Abstract

Copland Attestation Terms (CATs) provide a means for specifying layered attestations. The terms are designed to bridge the gap between formal analysis of attestation security guarantees and concrete implementations. We therefore provide two semantic interpretations of terms in our language. The first is a denotational semantics in terms of partially ordered sets of events. This directly connects CATs to prior work on layered attestation. The second is an operational semantics detailing how the data and control flow are executed. This gives explicit implementation guidance for attestation frameworks.

This document is generated from Coq sources that contain the proofs of the connection between the two semantics ensuring that any execution according to the operational semantics is consistent with the denotational event semantics. This ensures that formal guarantees resulting from analyzing the event semantics will hold for executions respecting the operational semantics.

Contents

Chapter 1

Introduction

Copland Attestation Terms (CATs) provide a means for specifying layered attestations. The terms are designed to bridge the gap between formal analysis of attestation security guarantees and concrete implementations. We therefore provide two semantic interpretations of terms in our language. The first is a denotational semantics in terms of partially ordered sets of events. This directly connects CATs to prior work on layered attestation. The second is an operational semantics detailing how the data and control flow are executed. This gives explicit implementation guidance for attestation frameworks.

This document is generated from a set of proof scripts for the Coq proof assistant. Chapter 2 contains a few tactics that are used throughout the proofs that follow. Chapter 3 contains facts about generic lists in Coq. Many of the lemmas are in support of particular proofs, so the motivation for some is obscure. A notable exception is the definition and lemmas about whether an element x is *earlier* than y in list ℓ. This will be used to discuss event orderings in traces. A trace is a list of events.

Chapter 4 precisely specifies CATs and the events that are generated when an CAT is executed. The script also defines annotated terms. To properly distinguish events, each event is associated with a natural number. Annotated terms are used to produce unique natural numbers for events.

Chapter 5 shows our representation of strict partially ordered sets of abstract events. The events are only required to have a function that produces its natural number. Proofs include a demonstration that the relation used to order events is, in fact, a strict partial order. Chapter 6 specializes the event system to the case of CATs and their events.

Chapter 7 defines a big-step semantics for CATs. The semantics associates a term, a place, and some initial evidence with a trace. The semantics is specified inductively, mirroring the structure of CATs. The chapter concludes by showing that the order of events in a trace specified by the big-step semantics is compatible with the partial order given by the CAT event system.

Chapter 8 defines a small-step semantics for CATs using a labeled transition system (LTS). The proofs in this chapter demonstrate that the LTS (1) computes the correct evidence associated with a term; (2) can always proceed unless it is in a halt state; and (3) always terminates.

Chapter 9 contains a proof of the main theorem of this work: a trace generated by the small-step semantics is compatible with the partial order given by the associated CAT event system. The main lemma is that every trace generated by the small-step semantics is a trace of the big-step semantics.

Chapter 2

Preamble

Tactics that provide useful automation.

```
Ltac inv H := inversion H; clear H; subst.
```

Expand let expressions in both the antecedent and the conclusion.

```
Ltac expand_let_pairs :=
  match goal with
  | ⊢ context [let (_,_) := ?e in _] ⇒
    rewrite (surjective_pairing e)
  | [ H: context [let (_,_) := ?e in _] ⊢ _ ] ⇒
    rewrite (surjective_pairing e) in H
  end.
```

Destruct disjuncts in the antecedent without naming them.

```
Ltac destruct_disjunct :=
  match goal with
  | [ H: _ ∨ _ ⊢ _ ] ⇒ destruct H as [H|H]
  end.
```

Chapter 3

More_lists

More facts about lists.

```
Require Import List Omega.
Import List.ListNotations.
Open Scope list_scope.

Set Implicit Arguments.

Section More_lists.

  Variable A: Type.
```

This is the analog of firstn_app from the List library.

```
  Lemma skipn_app n:
    ∀ l1 l2: list A,
      skipn n (l1 ++ l2) = (skipn n l1) ++ (skipn (n - length l1) l2).

  Lemma firstn_append:
    ∀ l l': list A,
      firstn (length l) (l ++ l') = l.

  Lemma skipn_append:
    ∀ l l': list A,
      skipn (length l) (l ++ l') = l'.

  Lemma skipn_all:
    ∀ l: list A,
      skipn (length l) l = [].

  Lemma skipn_nil:
```

```
  ∀ i,
    @skipn A i [] = [].

Lemma firstn_all_n:
  ∀ (l: list A) n,
    length l ≤ n →
    firstn n l = l.

Lemma skipn_all_n:
  ∀ (l: list A) n,
    length l ≤ n →
    skipn n l = [].

Lemma firstn_in:
  ∀ x i (l: list A),
    In x (firstn i l) →
    In x l.

Lemma skipn_in:
  ∀ x i (l: list A),
    In x (skipn i l) →
    In x l.

Lemma skipn_zero:
  ∀ l: list A,
    skipn 0 l = l.

Lemma in_skipn_cons:
  ∀ i x y (l: list A),
    In x (skipn i l) →
    In x (skipn i (y :: l)).
```

Do *l* and *l'* share no elements?

```
Definition disjoint_lists (l l': list A): Prop :=
  ∀ x, In x l → In x l' → False.

Lemma nodup_append:
  ∀ l l': list A,
    NoDup l → NoDup l' →
    disjoint_lists l l' →
    NoDup (l ++ l').

Lemma in_cons_app_cons:
  ∀ x y z (l: list A),
```

$In\ x\ (y :: l ++ [z]) \leftrightarrow$
$x = y \lor In\ x\ l \lor x = z.$

3.1 Earlier

Is x earlier than y in list l? This definition is used in contexts in which l has no duplicates.

`Definition` *earlier* $(l\!: list\ A)\ (x\ y\!: A) :=$
$\exists\ n,$
$In\ x\ (firstn\ n\ l) \land$
$In\ y\ (skipn\ n\ l).$

`Lemma` *earlier_in_left*:
$\forall\ l\ x\ y,$
$earlier\ l\ x\ y \rightarrow In\ x\ l.$

`Lemma` *earlier_in_right*:
$\forall\ l\ x\ y,$
$earlier\ l\ x\ y \rightarrow In\ y\ l.$

x is earlier than y in $p ++ q$ if x is earlier than y in p.

`Lemma` *earlier_left*:
$\forall\ p\ q\ x\ y,$
$earlier\ p\ x\ y \rightarrow earlier\ (p ++ q)\ x\ y.$

x is earlier than y in $p ++ q$ if x is earlier than y in q.

`Lemma` *earlier_right*:
$\forall\ p\ q\ x\ y,$
$earlier\ q\ x\ y \rightarrow earlier\ (p ++ q)\ x\ y.$

`Lemma` *earlier_append*:
$\forall\ p\ q\ x\ y,$
$In\ x\ p \rightarrow In\ y\ q \rightarrow$
$earlier\ (p ++ q)\ x\ y.$

`Lemma` *earlier_append_iff*:
$\forall\ x\ y\ (l\ l'\!: list\ A),$
$earlier\ (l ++ l')\ x\ y \leftrightarrow$
$earlier\ l\ x\ y \lor In\ x\ l \land In\ y\ l' \lor earlier\ l'\ x\ y.$

`Lemma` *earlier_cons*:

$\forall$ *p x y*,
In y p $\rightarrow$
earlier (*x* :: *p*) *x y*.

`Lemma` *earlier_cons_shift*:
$\forall$ *p x y z*,
earlier p x y $\rightarrow$
earlier (*z* :: *p*) *x y*.

`End` *More_lists*.

`Unset Implicit Arguments`.

Chapter 4

Term

This module contains the basic definitions for Copland terms, events, and annotated terms.

```
Require Import Omega Preamble.
```

4.1 Terms and Evidence

A term is either an atomic ASP, a remote call, a sequence of terms with data a dependency, a sequence of terms with no data dependency, or parallel terms.

Plc represents a place.

```
Notation Plc := nat (only parsing).
```

An argument to a userspace or kernel measurement.

```
Inductive Arg: Set :=
| arg: nat → Arg
| pl: Plc → Arg.

Definition eq_arg_dec:
  ∀ x y: Arg, {x = y} + {x ≠ y}.
Hint Resolve eq_arg_dec.

Inductive ASP: Set :=
| CPY: ASP
| KIM: (list Arg) → ASP
| USM: (list Arg) → ASP
```

| *SIG*: *ASP*
| *HSH*: *ASP*.

The method by which data is split is specified by a natural number.

```
Definition Split: Set := (nat × nat).
```

```
Inductive Term: Set :=
| asp: ASP → Term
| att: Plc → Term → Term
| lseq: Term → Term → Term
| bseq: Split → Term → Term → Term
| bpar: Split → Term → Term → Term.
```

The structure of evidence.

```
Inductive Evidence: Set :=
| mt: Evidence
| sp: nat → Evidence → Evidence
| kk: Plc → (list Arg) → Evidence → Evidence
| uu: Plc → (list Arg) → Evidence → Evidence
| gg: Plc → Evidence → Evidence
| hh: Plc → Evidence → Evidence
| ss: Evidence → Evidence → Evidence
| pp: Evidence → Evidence → Evidence.
```

```
Fixpoint eval_asp t p e :=
  match t with
  | CPY ⇒ e
  | KIM A ⇒ kk p A e
  | USM A ⇒ uu p A e
  | SIG ⇒ gg p e
  | HSH ⇒ hh p e
  end.
```

The evidence associated with a term, a place, and some initial evidence.

```
Fixpoint eval t p e :=
  match t with
  | asp a ⇒ eval_asp a p e
  | att q t1 ⇒ eval t1 q e
  | lseq t1 t2 ⇒ eval t2 p (eval t1 p e)
  | bseq s t1 t2 ⇒ ss (eval t1 p (sp (fst s) e))
```

```
                             (eval t2 p (sp (snd s) e))
  | bpar s t1 t2 ⇒ pp (eval t1 p (sp (fst s) e))
                             (eval t2 p (sp (snd s) e))
  end.
```

4.2 Events

There are events for each kind of action. This includes ASP actions such as measurement or data processing. It also includes control flow actions: a `split` occurs when a thread of control splits, and a `join` occurs when two threads join. Each event is distinguished using a unique natural number.

```
Inductive Ev: Set :=
| copy: nat → Plc → Evidence → Ev
| kmeas: nat → Plc → (list Arg) → Evidence → Evidence → Ev
| umeas: nat → Plc → (list Arg) → Evidence → Evidence → Ev
| sign: nat → Plc → Evidence → Evidence → Ev
| hash: nat → Plc → Evidence → Evidence → Ev
| req: nat → Plc → Plc → Evidence → Ev
| rpy: nat → Plc → Plc → Evidence → Ev
| split: nat → Plc → Evidence → Evidence → Evidence → Ev
| join: nat → Plc → Evidence → Evidence → Evidence → Ev.
```

```
Definition eq_ev_dec:
  ∀ x y: Ev, {x = y} + {x ≠ y}.
Hint Resolve eq_ev_dec.
```

The natural number used to distinguish evidence.

```
Definition ev x :=
  match x with
  | copy i _ _ ⇒ i
  | kmeas i _ _ _ _ ⇒ i
  | umeas i _ _ _ _ ⇒ i
  | sign i _ _ _ ⇒ i
  | hash i _ _ _ ⇒ i
  | req i _ _ _ ⇒ i
  | rpy i _ _ _ ⇒ i
  | split i _ _ _ _ ⇒ i
  | join i _ _ _ _ ⇒ i
```

```
end.
```

Events are used in a manner that ensures that

$\forall\ e0\ e1,\ \mathsf{ev}\ e0 = \mathsf{ev}\ e1 \rightarrow e0 = e1.$

See Lemma events_injective.

```
Definition asp_event i x p e :=
  match x with
  | CPY ⇒ copy i p e
  | KIM A ⇒ kmeas i p A e (eval_asp (KIM A) p e)
  | USM A ⇒ umeas i p A e (eval_asp (USM A) p e)
  | SIG ⇒ sign i p e (eval_asp SIG p e)
  | HSH ⇒ hash i p e (eval_asp HSH p e)
  end.
```

4.3 Annotated Terms

Annotated terms are used to ensure that each distinct event has a distinct natural number. To do so, each term is annotated by a pair of numbers called a range. Let (i, k) be the label for term t. The labels will be chosen to have the property such that for each event in the set of events associated with term t, its number j will be in the range $i \leq j < k$.

```
Definition Range: Set := nat × nat.
```

```
Inductive AnnoTerm: Set :=
| aasp: Range → ASP → AnnoTerm
| aatt: Range → Plc → AnnoTerm → AnnoTerm
| alseq: Range → AnnoTerm → AnnoTerm → AnnoTerm
| abseq: Range → Split → AnnoTerm → AnnoTerm → AnnoTerm
| abpar: Range → Split → AnnoTerm → AnnoTerm → AnnoTerm.
```

The number of events associated with a term. The branching terms add a split and a join to the events of their subterms. Similarly, the remote calls add a request and receive to the events of their subterm.

```
Fixpoint esize t :=
  match t with
  | aasp _ _ ⇒ 1
  | aatt _ _ t1 ⇒ 2 + esize t1
```

```
  | alseq _ t1 t2 ⇒ esize t1 + esize t2
  | abseq _ _ t1 t2 ⇒ 2 + esize t1 + esize t2
  | abpar _ _ t1 t2 ⇒ 2 + esize t1 + esize t2
  end.

Definition range x :=
  match x with
  | aasp r _ ⇒ r
  | aatt r _ _ ⇒ r
  | alseq r _ _ ⇒ r
  | abseq r _ _ _ ⇒ r
  | abpar r _ _ _ ⇒ r
  end.
```

This function annotates a term. It feeds a natural number throughout the computation so as to ensure each event has a unique natural number.

```
Fixpoint anno (t: Term) i: nat × AnnoTerm :=
  match t with
  | asp x ⇒ (S i, aasp (i, S i) x)
  | att p x ⇒
    let (j, a) := anno x (S i) in
    (S j, aatt (i, S j) p a)
  | lseq x y ⇒
    let (j, a) := anno x i in
    let (k, b) := anno y j in
    (k, alseq (i, k) a b)
  | bseq s x y ⇒
    let (j, a) := anno x (S i) in
    let (k, b) := anno y j in
    (S k, abseq (i, S k) s a b)
  | bpar s x y ⇒
    let (j, a) := anno x (S i) in
    let (k, b) := anno y j in
    (S k, abpar (i, S k) s a b)
  end.

Lemma anno_range:
  ∀ x i,
    range (snd (anno x i)) = (i, fst (anno x i)).
```

```
Definition annotated x :=
  snd (anno x 0).
```

This predicate determines if an annotated term is well formed, that is if its ranges correctly capture the relations between a term and its associated events.

```
Inductive well_formed: AnnoTerm → Prop :=
| wf_asp: ∀ r x,
    snd r = S (fst r) →
    well_formed (aasp r x)
| wf_att: ∀ r p x,
    well_formed x →
    S (fst r) = fst (range x) →
    snd r = S (snd (range x)) →
    well_formed (aatt r p x)
| wf_lseq: ∀ r x y,
    well_formed x → well_formed y →
    fst r = fst (range x) →
    snd (range x) = fst (range y) →
    snd r = snd (range y) →
    well_formed (alseq r x y)
| wf_bseq: ∀ r s x y,
    well_formed x → well_formed y →
    S (fst r) = fst (range x) →
    snd (range x) = fst (range y) →
    snd r = S (snd (range y)) →
    well_formed (abseq r s x y)
| wf_bpar: ∀ r s x y,
    well_formed x → well_formed y →
    S (fst r) = fst (range x) →
    snd (range x) = fst (range y) →
    snd r = S (snd (range y)) →
    well_formed (abpar r s x y).
Hint Constructors well_formed.
```

```
Lemma well_formed_range:
  ∀ t,
    well_formed t →
    snd (range t) = fst (range t) + esize t.
```

```
Lemma anno_well_formed:
  ∀ t i,
    well_formed (snd (anno t i)).
```

Eval for annotated terms.

```
Fixpoint aeval t p e :=
  match t with
  | aasp _ x ⇒ eval (asp x) p e
  | aatt _ q x ⇒ aeval x q e
  | alseq _ t1 t2 ⇒ aeval t2 p (aeval t1 p e)
  | abseq _ s t1 t2 ⇒ ss (aeval t1 p ((sp (fst s)) e))
                          (aeval t2 p ((sp (snd s)) e))
  | abpar _ s t1 t2 ⇒ pp (aeval t1 p ((sp (fst s)) e))
                          (aeval t2 p ((sp (snd s)) e))
  end.

Lemma eval_aeval:
  ∀ t p e i,
    eval t p e = aeval (snd (anno t i)) p e.
```

This predicate specifies when a term, a place, and some initial evidence is related to an event. In other words, it specifies the set of events associated with a term, a place, and some initial evidence.

```
Inductive events: AnnoTerm → Plc → Evidence → Ev → Prop :=
| evtscpy:
    ∀ r i p e,
      fst r = i →
      events (aasp r CPY) p e (copy i p e)
| evtskim:
    ∀ i r a p e e',
      fst r = i →
      kk p a e = e' →
      events (aasp r (KIM a)) p e (kmeas i p a e e')
| evtsusm:
    ∀ i r a p e e',
      fst r = i →
      uu p a e = e' →
      events (aasp r (USM a)) p e (umeas i p a e e')
| evtssig:
```

```
    ∀ r i p e e',
      fst r = i →
      gg p e = e' →
      events (aasp r SIG) p e (sign i p e e')
| evtshsh:
    ∀ r i p e e',
      fst r = i →
      hh p e = e' →
      events (aasp r HSH) p e (hash i p e e')

| evtsattreq:
    ∀ r p q t e i,
      fst r = i →
      events (aatt r q t) p e (req i p q e)
| evtsatt:
    ∀ r p q t e ev,
      events t q e ev →
      events (aatt r q t) p e ev
| evtsattrpy:
    ∀ r p q t e e' i,
      snd r = S i →
      aeval t q e = e' →
      events (aatt r q t) p e (rpy i p q e')

| evtslseql:
    ∀ r t1 t2 p e ev,
      events t1 p e ev →
      events (alseq r t1 t2) p e ev
| evtslseqr:
    ∀ r t1 t2 p e ev,
      events t2 p (aeval t1 p e) ev →
      events (alseq r t1 t2) p e ev

| evtsbseqsplit:
    ∀ r i s t1 t2 p e,
      fst r = i →
      events (abseq r s t1 t2) p e
             (split i p e (sp (fst s) e) (sp (snd s) e))
```

```
| evtsbseql:
    ∀ r s t1 t2 p e ev,
      events t1 p (sp (fst s) e) ev →
      events (abseq r s t1 t2) p e ev
| evtsbseqr:
    ∀ r s t1 t2 p e ev,
      events t2 p (sp (snd s) e) ev →
      events (abseq r s t1 t2) p e ev
| evtsbseqjoin:
    ∀ r i s t1 t2 p e e1 e2,
      snd r = S i →
      aeval t1 p (sp (fst s) e) = e1 →
      aeval t2 p (sp (snd s) e) = e2 →
      events (abseq r s t1 t2) p e
             (join i p e1 e2 (ss e1 e2))

| evtsbparsplit:
    ∀ r i s t1 t2 p e,
      fst r = i →
      events (abpar r s t1 t2) p e
             (split i p e (sp (fst s) e) (sp (snd s) e))
| evtsbparl:
    ∀ r s t1 t2 p e ev,
      events t1 p (sp (fst s) e) ev →
      events (abpar r s t1 t2) p e ev
| evtsbparr:
    ∀ r s t1 t2 p e ev,
      events t2 p (sp (snd s) e) ev →
      events (abpar r s t1 t2) p e ev
| evtsbparjoin:
    ∀ r i s t1 t2 p e e1 e2,
      snd r = S i →
      aeval t1 p (sp (fst s) e) = e1 →
      aeval t2 p (sp (snd s) e) = e2 →
      events (abpar r s t1 t2) p e
             (join i p e1 e2 (pp e1 e2)).
Hint Constructors events.
```

```
Lemma events_range:
  ∀ t p e v,
    well_formed t →
    events t p e v →
    fst (range t) ≤ ev v < snd (range t).

Lemma at_range:
  ∀ x r i,
    S (fst r) = fst x →
    snd r = S (snd x) →
    fst r ≤ i < snd r →
    i = fst r ∨
    fst x ≤ i < snd x ∨
    i = snd x.

Lemma lin_range:
  ∀ x y i,
    snd x = fst y →
    fst x ≤ i < snd y →
    fst x ≤ i < snd x ∨
    fst y ≤ i < snd y.

Lemma bra_range:
  ∀ x y r i,
    S (fst r) = fst x →
    snd x = fst y →
    snd r = S (snd y) →
    fst r ≤ i < snd r →
    i = fst r ∨
    fst x ≤ i < snd x ∨
    fst y ≤ i < snd y ∨
    i = snd y.
```

Properties of events.

```
Lemma events_range_event:
  ∀ t p e i,
    well_formed t →
    fst (range t) ≤ i < snd (range t) →
    ∃ v, events t p e v ∧ ev v = i.

Ltac events_event_range :=
```

```
  repeat match goal with
         | [ H: events _ _ _ _ ⊢ _ ] ⇒
           apply events_range in H; auto
         end; omega.

Lemma events_injective:
  ∀ t p e v1 v2,
    well_formed t →
    events t p e v1 →
    events t p e v2 →
    ev v1 = ev v2 →
    v1 = v2.
```

Chapter 5

Event_system

Abstract event systems.

```
Require Import Omega Preamble.

Set Implicit Arguments.
```

An event system is a set of events and a strict partial order on the events. An event system is represented by a well-structured **EvSys**.

```
Section Event_system.
```

The sort of an event.

```
  Variable A: Set.
```

The number associated with an event.

```
  Variable ev: A → nat.

  Definition ES_Range: Set := nat × nat.
```

An event system.

```
  Inductive EvSys: Set :=
  | leaf: ES_Range → A → EvSys
  | before: ES_Range → EvSys → EvSys → EvSys
  | merge: ES_Range → EvSys → EvSys → EvSys.

  Definition es_range es :=
    match es with
    | leaf r _ ⇒ r
    | before r _ _ ⇒ r
    | merge r _ _ ⇒ r
```

```
  end.

Fixpoint es_size es :=
  match es with
  | leaf _ _⇒ 1
  | before _ x y ⇒ es_size x + es_size y
  | merge _ x y ⇒ es_size x + es_size y
  end.
```

Definition of a well-structured event system.

```
Inductive well_structured: EvSys → Prop :=
| ws_leaf_event:
    ∀ r e,
      snd r = S (fst r) →
      ev e = fst r →
      well_structured (leaf r e)
| ws_before:
    ∀ r x y,
      well_structured x →
      well_structured y →
      r = (fst (es_range x), snd (es_range y)) →
      snd (es_range x) = fst (es_range y) →
      well_structured (before r x y)
| ws_merge:
    ∀ r x y,
      well_structured x →
      well_structured y →
      r = (fst (es_range x), snd (es_range y)) →
      snd (es_range x) = fst (es_range y) →
      well_structured (merge r x y).
Hint Constructors well_structured.

Lemma well_structured_range:
  ∀ es,
    well_structured es →
    snd (es_range es) = fst (es_range es) + es_size es.
```

Is an event in an event system?

```
Inductive ev_in: A → EvSys → Prop :=
| ein_leaf: ∀ r ev,
```

```
    ev_in ev (leaf r ev)
| ein_beforel: ∀ r ev es1 es2,
    ev_in ev es1 → ev_in ev (before r es1 es2)
| ein_beforer: ∀ r ev es1 es2,
    ev_in ev es2 → ev_in ev (before r es1 es2)
| ein_mergel: ∀ r ev es1 es2,
    ev_in ev es1 → ev_in ev (merge r es1 es2)
| ein_merger: ∀ r ev es1 es2,
    ev_in ev es2 → ev_in ev (merge r es1 es2).
Hint Constructors ev_in.
```

Is one event before another?

```
Inductive prec: EvSys → A → A → Prop :=
| prseq: ∀ r x y e f,
    ev_in e x → ev_in f y →
    prec (before r x y) e f
| prseql: ∀ r x y e f,
    prec x e f →
    prec (before r x y) e f
| prseqr: ∀ r x y e f,
    prec y e f →
    prec (before r x y) e f
| prparl: ∀ r x y e f,
    prec x e f →
    prec (merge r x y) e f
| prparr: ∀ r x y e f,
    prec y e f →
    prec (merge r x y) e f.
Hint Constructors prec.

Lemma prec_in_left:
  ∀ es ev1 ev2,
    prec es ev1 ev2 → ev_in ev1 es.

Lemma prec_in_right:
  ∀ es ev1 ev2,
    prec es ev1 ev2 → ev_in ev2 es.

Lemma ws_evsys_range:
  ∀ es e,
```

$well\_structured\ es \rightarrow$
$ev\_in\ e\ es \rightarrow$
$fst\ (es\_range\ es) \leq ev\ e < snd\ (es\_range\ es).$

`Lemma` *es_injective_events*:
$\forall\ es\ ev0\ ev1,$
$well\_structured\ es \rightarrow$
$ev\_in\ ev0\ es \rightarrow ev\_in\ ev1\ es \rightarrow$
$ev\ ev0 = ev\ ev1 \rightarrow$
$ev0 = ev1.$

A relation is a strict partial order iff it is irreflexive and transitive.

`Lemma` *evsys_irreflexive*:
$\forall\ es\ ev,$
$well\_structured\ es \rightarrow$
$\neg prec\ es\ ev\ ev.$

`Lemma` *evsys_transitive*:
$\forall\ es\ ev0\ ev1\ ev2,$
$well\_structured\ es \rightarrow$
$prec\ es\ ev0\ ev1 \rightarrow$
$prec\ es\ ev1\ ev2 \rightarrow$
$prec\ es\ ev0\ ev2.$

Merge is associative.

`Definition` *same_rel es0 es1* :=
$\forall\ ev0\ ev1,$
$prec\ es0\ ev0\ ev1 \leftrightarrow prec\ es1\ ev0\ ev1.$

`Lemma` *ws_merge1*:
$\forall\ r\ s\ x\ y\ z,$
$well\_structured\ (merge\ r\ x\ y) \rightarrow$
$well\_structured\ (merge\ s\ y\ z) \rightarrow$
$well\_structured\ (merge\ (fst\ r,\ snd\ s)\ x\ (merge\ s\ y\ z)).$

`Lemma` *ws_merge2*:
$\forall\ r\ s\ x\ y\ z,$
$well\_structured\ (merge\ r\ x\ y) \rightarrow$
$well\_structured\ (merge\ s\ y\ z) \rightarrow$
$well\_structured\ (merge\ (fst\ r,\ snd\ s)\ (merge\ r\ x\ y)\ z).$

`Lemma` *merge_associative*:

$\forall$ *r s x y z*,
  same_rel (*merge* (*fst r*, *snd s*) *x* (*merge s y z*))
    (*merge* (*fst r*, *snd s*) (*merge r x y*) *z*).

A more useful form of merge associative

Lemma *merge_associative_pairs*:
  $\forall$ *r0 r1 s0 s1 x y z*,
  same_rel (*merge* (*r0*, *s1*) *x* (*merge* (*s0*, *s1*) *y z*))
    (*merge* (*r0*, *s1*) (*merge* (*r0*, *r1*) *x y*) *z*).

Before is associative.

Lemma *ws_before1*:
  $\forall$ *r s x y z*,
  well_structured (before *r x y*) $\rightarrow$
  well_structured (before *s y z*) $\rightarrow$
  well_structured (before (*fst r*, *snd s*) *x* (before *s y z*)).

Lemma *ws_before2*:
  $\forall$ *r s x y z*,
  well_structured (before *r x y*) $\rightarrow$
  well_structured (before *s y z*) $\rightarrow$
  well_structured (before (*fst r*, *snd s*) (before *r x y*) *z*).

Lemma *before_associative*:
  $\forall$ *r s x y z*,
  same_rel (before (*fst r*, *snd s*) *x* (before *s y z*))
    (before (*fst r*, *snd s*) (before *r x y*) *z*).

Lemma *before_associative_pairs*:
  $\forall$ *r0 r1 s0 s1 x y z*,
  same_rel (before (*r0*, *s1*) *x* (before (*s0*, *s1*) *y z*))
    (before (*r0*, *s1*) (before (*r0*, *r1*) *x y*) *z*).

Lemma *ws_exists*:
  $\forall$ *es*,
  well_structured es $\rightarrow$
  $\exists$ *e*, *ev_in e es*.

Maximal events.

Definition *supreme es e* :=
  ev_in e es $\wedge$ $\forall$ *e0*, *ev_in e0 es* $\rightarrow$ $\neg$*prec es e e0*.

Inductive version of a maximal event.

```
Inductive sup: EvSys → A → Prop :=
| sup_leaf: ∀ r e, sup (leaf r e) e
| sup_before:
    ∀ r es0 es1 e,
      sup es1 e → sup (before r es0 es1) e
| sup_mergel:
    ∀ r es0 es1 e,
      sup es0 e → sup (merge r es0 es1) e
| sup_merger:
    ∀ r es0 es1 e,
      sup es1 e → sup (merge r es0 es1) e.
Hint Constructors sup.

Lemma before_sup:
  ∀ r x y e,
    sup (before r x y) e → sup y e.

Lemma sup_supreme:
  ∀ es e,
    well_structured es →
    sup es e ↔ supreme es e.
```

Return one of possibly many maximal events.

```
Fixpoint max es :=
  match es with
  | leaf _ e ⇒ e
  | before _ x y ⇒ max y
  | merge _ x y ⇒ max y
  end.

Lemma supreme_max:
  ∀ es,
    well_structured es →
    supreme es (max es).

End Event_system.

Hint Constructors well_structured ev_in prec sup.

Unset Implicit Arguments.
```

Chapter 6

Term_system

Copland specific event systems.

```
Require Import Omega Preamble More_lists Term Event_system.
```

Construct an event system from an annotated term, place, and evidence.

```
Fixpoint ev_sys (t: AnnoTerm) p e: EvSys Ev :=
  match t with
  | aasp (i, j) x ⇒ leaf (i, j) (asp_event i x p e)
  | aatt (i, j) q x ⇒
    before (i, j)
      (leaf (i, S i) (req i p q e))
      (before (S i, j)
              (ev_sys x q e)
              (leaf (pred j, j) (rpy (pred j) p q (aeval x q e))))
  | alseq r x y ⇒ before r (ev_sys x p e)
                            (ev_sys y p (aeval x p e))
  | abseq (i, j) s x y ⇒
    before (i, j)
           (leaf (i, S i)
                 (split i p e (sp (fst s) e)
                              (sp (snd s) e)))
           (before (S i, j)
                   (before (S i, (pred j))
                           (ev_sys x p (sp (fst s) e))
                           (ev_sys y p (sp (snd s) e)))
                   (leaf ((pred j), j)
```

```
                    (join (pred j) p
                          (aeval x p (sp (fst s) e))
                          (aeval y p (sp (snd s) e))
            (ss (aeval x p (sp (fst s) e))
                (aeval y p (sp (snd s) e))))))
  | abpar (i, j) s x y ⇒
    before (i, j)
            (leaf (i, S i)
                  (split i p e (sp (fst s) e)
                               (sp (snd s) e)))
            (before (S i, j)
                    (merge (S i, (pred j))
                           (ev_sys x p (sp (fst s) e))
                           (ev_sys y p (sp (snd s) e)))
                    (leaf ((pred j), j)
                    (join (pred j) p
                          (aeval x p (sp (fst s) e))
                          (aeval y p (sp (snd s) e))
            (pp (aeval x p (sp (fst s) e))
                (aeval y p (sp (snd s) e))))))
  end.

Lemma evsys_range:
  ∀ t p e,
    es_range (ev_sys t p e) = range t.

Lemma well_structured_evsys:
  ∀ t p e,
    well_formed t →
    well_structured ev (ev_sys t p e).
```

The events in the event system correspond to the events associated with a term, a place, and some evidence.

```
Lemma evsys_events:
  ∀ t p e ev,
    well_formed t →
    ev_in ev (ev_sys t p e) ↔ events t p e ev.
```

Maximal events are unique.

```
Lemma supreme_unique:
```

```
  ∀ t p e,
    well_formed t →
    ∃ ! v, supreme (ev_sys t p e) v.
Lemma evsys_max_unique:
  ∀ t p e,
    well_formed t →
    unique (supreme (ev_sys t p e)) (max (ev_sys t p e)).
```

Maximal event evidence output matches aeval.

```
Definition out_ev v :=
  match v with
  | copy _ _ e ⇒ e
  | kmeas _ _ _ _ e ⇒ e
  | umeas _ _ _ _ e ⇒ e
  | sign _ _ _ e ⇒ e
  | hash _ _ _ e ⇒ e
  | req _ _ _ e ⇒ e
  | rpy _ _ _ e ⇒ e
  | split _ _ _ _ e ⇒ e
  | join _ _ _ _ e ⇒ e
  end.
Lemma max_eval:
  ∀ t p e,
    well_formed t →
    out_ev (max (ev_sys t p e)) = aeval t p e.
```

lseq is associative relative to the event semantics

```
Lemma lseq_assoc:
  ∀ t1 t2 t3 i p e,
    same_rel
      (ev_sys (snd (anno (lseq t1 (lseq t2 t3)) i)) p e)
      (ev_sys (snd (anno (lseq (lseq t1 t2) t3) i)) p e).
```

Chapter 7

Trace

Traces and their relation to event systems.

```
Require Import List.
Import List.ListNotations.
Open Scope list_scope.
Require Import Omega Preamble More_lists Term Event_system Term_system.
```

7.1 Shuffles

A trace is a list of events. **shuffle** merges two traces as is done by parallel execution. The order of events in the traces to be merged does not change, but all other interleavings are allowed.

```
Inductive shuffle: list Ev → list Ev → list Ev → Prop :=
| shuffle_nil_left: ∀ es, shuffle [] es es
| shuffle_nil_right: ∀ es, shuffle es [] es
| shuffle_left:
    ∀ e es0 es1 es2,
      shuffle es0 es1 es2 →
      shuffle (e :: es0) es1 (e :: es2)
| shuffle_right:
    ∀ e es0 es1 es2,
      shuffle es0 es1 es2 →
      shuffle es0 (e :: es1) (e :: es2).
Hint Constructors shuffle.
```

```
Lemma shuffle_length:
  ∀ es0 es1 es2,
    shuffle es0 es1 es2 →
    length es0 + length es1 = length es2.
Lemma shuffle_in_left:
  ∀ e es0 es1 es2,
    shuffle es0 es1 es2 →
    In e es0 →
    In e es2.
Lemma shuffle_in_right:
  ∀ e es0 es1 es2,
    shuffle es0 es1 es2 →
    In e es1 →
    In e es2.
Lemma shuffle_in:
  ∀ e es0 es1 es2,
    shuffle es0 es1 es2 →
    In e es2 ↔ In e es0 ∨ In e es1.
Lemma shuffle_in_skipn_left:
  ∀ i e es0 es1 es2,
    shuffle es0 es1 es2 →
    In e (skipn i es0) →
    In e (skipn i es2).
Lemma shuffle_in_skipn_right:
  ∀ i e es0 es1 es2,
    shuffle es0 es1 es2 →
    In e (skipn i es1) →
    In e (skipn i es2).
Lemma shuffle_earlier_left:
  ∀ es0 es1 es2 e0 e1,
    earlier es0 e0 e1 →
    shuffle es0 es1 es2 →
    earlier es2 e0 e1.
Lemma shuffle_earlier_right:
  ∀ es0 es1 es2 e0 e1,
    earlier es1 e0 e1 →
```

```
    shuffle es0 es1 es2 →
    earlier es2 e0 e1.
```

```
Lemma shuffle_nodup_append:
  ∀ tr0 tr1 tr2,
    NoDup tr0 → NoDup tr1 →
    disjoint_lists tr0 tr1 →
    shuffle tr0 tr1 tr2 →
    NoDup tr2.
```

7.2 Big-Step Semantics

The traces associated with an annotated term are defined inductively.

```
Inductive trace: AnnoTerm → Plc → Evidence →
                 list Ev → Prop :=
| tasp: ∀ r x p e,
    trace (aasp r x) p e [(asp_event (fst r) x p e)]
| tatt: ∀ r x p e q tr1,
    trace x q e tr1 →
    trace (aatt r q x) p e
          ((req (fst r) p q e)
             :: tr1 ++
             [(rpy (pred (snd r)) p q (aeval x q e))])
| tlseq: ∀ r x y p e tr0 tr1,
    trace x p e tr0 →
    trace y p (aeval x p e) tr1 →
    trace (alseq r x y) p e (tr0 ++ tr1)
| tbseq: ∀ r s x y p e tr0 tr1,
    trace x p (sp (fst s) e) tr0 →
    trace y p (sp (snd s) e) tr1 →
    trace (abseq r s x y) p e
          ((split (fst r) p e
                  (sp (fst s) e)
                  (sp (snd s) e))
             :: tr0 ++ tr1 ++
             [(join (pred (snd r)) p (aeval x p (sp (fst s) e)) (aeval y p (sp
(snd s) e)) (ss (aeval x p (sp (fst s) e)) (aeval y p (sp (snd s) e))))])
| tbpar: ∀ r s x y p e tr0 tr1 tr2,
```

```
    trace x p (sp (fst s) e) tr0 →
    trace y p (sp (snd s) e) tr1 →
    shuffle tr0 tr1 tr2 →
    trace (abpar r s x y) p e
          ((split (fst r) p e
                  (sp (fst s) e)
                  (sp (snd s) e))
             :: tr2 ++
             [(join (pred (snd r)) p (aeval x p (sp (fst s) e)) (aeval y p (sp
(snd s) e)) (pp (aeval x p (sp (fst s) e)) (aeval y p (sp (snd s) e))))]).
Hint Resolve tasp.
```

```
Lemma trace_length:
  ∀ t p e tr,
    trace t p e tr → esize t = length tr.
```

The events in a trace correspond to the events associated with an annotated term, a place, and some evidence.

```
Lemma trace_events:
  ∀ t p e tr v,
    well_formed t →
    trace t p e tr →
    In v tr ↔ events t p e v.
```

```
Lemma trace_range:
  ∀ t p e tr v,
    well_formed t →
    trace t p e tr →
    In v tr →
    fst (range t) ≤ ev v < snd (range t).
```

```
Lemma trace_range_event:
  ∀ t p e tr i,
    well_formed t →
    trace t p e tr →
    fst (range t) ≤ i < snd (range t) →
    ∃ v, In v tr ∧ ev v = i.
```

```
Lemma trace_injective_events:
  ∀ t p e tr v0 v1,
    well_formed t →
```

trace t p e tr →
In v0 tr → *In v1 tr* →
ev v0 = *ev v1* →
v0 = *v1*.

`Lemma` *nodup_trace*:
∀ *t p e tr*,
well_formed t →
trace t p e tr →
NoDup tr.

7.3 Event Systems and Traces

`Lemma` *evsys_tr_in*:
∀ *t p e tr ev0*,
well_formed t →
trace t p e tr →
ev_in ev0 (*ev_sys t p e*) →
In ev0 tr.

The traces associated with an annotated term are compatible with its event system.

`Theorem` *trace_order*:
∀ *t p e tr ev0 ev1*,
well_formed t →
trace t p e tr →
prec (*ev_sys t p e*) *ev0 ev1* →
earlier tr ev0 ev1.

Chapter 8

LTS

A small-step semantics for annotated terms.

```
Require Import List.
Import List.ListNotations.
Open Scope list_scope.
Require Import Omega Preamble Term.
```

8.1 States

```
Inductive St: Set :=
| stop: Plc → Evidence → St
| conf: AnnoTerm → Plc → Evidence → St
| rem: Plc → Plc → St → St
| ls: St → AnnoTerm → St
| bsl: nat → St → AnnoTerm → Plc → Evidence → St
| bsr: nat → Evidence → St → St
| bp: nat → St → St → St.
```

```
Fixpoint pl (s:St) :=
  match s with
  | stop p _ ⇒ p
  | conf _ p _ ⇒ p
  | rem _ p _ ⇒ p
  | ls st _ ⇒ pl st
  | bsl _ _ _ p _ ⇒ p
```

```
  | bsr _ _ st ⇒ pl st
  | bp _ _ st ⇒ pl st
  end.
```

The evidence associated with a state.

```
Fixpoint seval st :=
  match st with
  | stop _ e ⇒ e
  | conf t p e ⇒ aeval t p e
  | rem _ _ st ⇒ seval st
  | ls st t ⇒ aeval t (pl st) (seval st)
  | bsl _ st t p e ⇒ ss (seval st) (aeval t p e)
  | bsr _ e st ⇒ ss e (seval st)
  | bp _ st0 st1 ⇒ pp (seval st0) (seval st1)
end.
```

8.2 Labeled Transition System

The label in a transition is either an event or None when the transition is silent. Notice the use of annotations to provide the correct number for each event.

```
Inductive step: St → option Ev → St → Prop :=
 Measurement
| stasp:
    ∀ r x p e,
      step (conf (aasp r x) p e)
           (Some (asp_event (fst r) x p e))
           (stop p (aeval (aasp r x) p e))
 Remote call
| statt:
    ∀ r x p q e,
      step (conf (aatt r q x) p e)
           (Some (req (fst r) p q e))
           (rem (snd r) p (conf x q e))
| stattstep:
    ∀ st0 ev st1 p j,
      step st0 ev st1 →
```

```
        step (rem j p st0) ev (rem j p st1)
| stattstop:
    ∀ j p q e,
        step (rem j p (stop q e))
             (Some (rpy (pred j) p q e))
             (stop p e)
```

Linear Sequential Composition

```
| stlseq:
    ∀ r x y p e,
        step (conf (alseq r x y) p e)
             None
             (ls (conf x p e) y)
| stlseqstep:
    ∀ st0 ev st1 t,
        step st0 ev st1 →
        step (ls st0 t) ev (ls st1 t)
| stlseqstop:
    ∀ t p e,
        step (ls (stop p e) t) None (conf t p e)
```

Branching Sequential Composition

```
| stbseq:
    ∀ r s x y p e,
        step (conf (abseq r s x y) p e)
             (Some (split (fst r) p e (sp (fst s) e)
                          (sp (snd s) e)))
             (bsl (snd r) (conf x p (sp (fst s) e))
                  y p (sp (snd s) e))
| stbslstep:
    ∀ st0 ev st1 j t p e,
        step st0 ev st1 →
        step (bsl j st0 t p e) ev (bsl j st1 t p e)
| stbslstop:
    ∀ j e e' t p p',
        step (bsl j (stop p e) t p' e')
             None
             (bsr j e (conf t p' e'))
| stbsrstep:
    ∀ st0 ev st1 j e,
```

```
      step st0 ev st1 →
      step (bsr j e st0) ev (bsr j e st1)
| stbsrstop:
    ∀ j e p e',
      step (bsr j e (stop p e'))
           (Some (join (pred j) p e e' (ss e e')))
           (stop p (ss e e'))
```

Branching Parallel composition

```
| stbpar:
    ∀ r s x y p e,
      step (conf (abpar r s x y) p e)
           (Some (split (fst r) p e
                        (sp (fst s) e)
                        (sp (snd s) e)))
           (bp (snd r)
               (conf x p (sp (fst s) e))
               (conf y p (sp (snd s) e)))
| stbpstepleft:
    ∀ st0 st1 st2 ev j,
      step st0 ev st2 →
      step (bp j st0 st1) ev (bp j st2 st1)
| stbpstepright:
    ∀ st0 st1 st2 ev j,
      step st1 ev st2 →
      step (bp j st0 st1) ev (bp j st0 st2)
| stbpstop:
    ∀ j p e p' e',
      step (bp j (stop p e) (stop p' e'))
           (Some (join (pred j) p' e e' (pp e e')))
           (stop p' (pp e e')).
Hint Constructors step.
```

A step preserves place.

```
Lemma step_pl_eq:
  ∀ st0 ev st1,
    step st0 ev st1 → pl st0 = pl st1.
```

A step preserves evidence.

```
Lemma step_seval:
```

```
  ∀ st0 ev st1,
    step st0 ev st1 →
    seval st0 = seval st1.
```

8.3 Transitive Closures

```
Inductive lstar: St → list Ev → St → Prop :=
| lstar_refl: ∀ st, lstar st [] st
| lstar_tran: ∀ st0 e st1 tr st2,
    step st0 (Some e) st1 → lstar st1 tr st2 → lstar st0 (e :: tr) st2
| lstar_silent_tran: ∀ st0 st1 tr st2,
    step st0 None st1 → lstar st1 tr st2 → lstar st0 tr st2.
Hint Resolve lstar_refl.

Lemma lstar_transitive:
  ∀ st0 tr0 st1 tr1 st2,
    lstar st0 tr0 st1 →
    lstar st1 tr1 st2 →
    lstar st0 (tr0 ++ tr1) st2.
```

Transitive closure without labels.

```
Inductive star: St → St → Prop :=
| star_refl: ∀ st, star st st
| star_tran: ∀ st0 e st1 st2,
    step st0 e st1 → star st1 st2 → star st0 st2.
Hint Resolve star_refl.

Lemma star_transitive:
  ∀ st0 st1 st2,
    star st0 st1 →
    star st1 st2 →
    star st0 st2.

Lemma lstar_star:
  ∀ st0 tr st1,
    lstar st0 tr st1 → star st0 st1.

Lemma star_lstar:
  ∀ st0 st1,
    star st0 st1 → ∃ tr, lstar st0 tr st1.
```

```
Lemma star_seval:
  ∀ st0 st1,
    star st0 st1 → seval st0 = seval st1.

Lemma steps_preserves_eval:
  ∀ t p p' e0 e1,
    star (conf t p e0) (stop p' e1) →
    aeval t p e0 = e1.
```

8.4 Correct Path Exists

```
Lemma star_strem:
  ∀ st0 st1 j p,
    star st0 st1 → star (rem j p st0) (rem j p st1).

Lemma star_stls:
  ∀ st0 st1 t,
    star st0 st1 → star (ls st0 t) (ls st1 t).

Lemma star_stbsl:
  ∀ st0 st1 j t p e,
    star st0 st1 →
    star (bsl j st0 t p e) (bsl j st1 t p e).

Lemma star_stbsr:
  ∀ st0 st1 j e,
    star st0 st1 →
    star (bsr j e st0) (bsr j e st1).

Lemma star_stbp:
  ∀ st0 st1 st2 st3 j,
    star st0 st1 →
    star st2 st3 →
    star (bp j st0 st2) (bp j st1 st3).

Theorem correct_path_exists:
  ∀ t p e,
    star (conf t p e) (stop p (aeval t p e)).
```

8.5 Progress

```
Definition halt st :=
  match st with
  | stop _ _ ⇒ True
  | _ ⇒ False
  end.
```

The step relation nevers gets stuck.

```
Theorem never_stuck:
  ∀ st0,
    halt st0 ∨ ∃ e st1, step st0 e st1.
```

8.6 Termination

```
Inductive nstar: nat → St → St → Prop :=
| nstar_refl: ∀ st, nstar 0 st st
| nstar_tran: ∀ st0 st1 st2 e n,
    nstar n st0 st1 → step st1 e st2 → nstar (S n) st0 st2.
Hint Resolve nstar_refl.
```

```
Lemma nstar_transitive:
  ∀ m n st0 st1 st2,
    nstar m st0 st1 →
    nstar n st1 st2 →
    nstar (m + n) st0 st2.
```

```
Lemma nstar_star:
  ∀ n st0 st1,
    nstar n st0 st1 → star st0 st1.
```

```
Lemma star_nstar:
  ∀ st0 st1,
    star st0 st1 →
    ∃ n, nstar n st0 st1.
```

Size of a term (number of steps to reduce).

```
Fixpoint tsize t: nat :=
  match t with
  | aasp _ _ ⇒ 1
```

```
  | aatt _ _ x ⇒ 2 + tsize x
  | alseq _ x y ⇒ 2 + tsize x + tsize y
  | abseq _ _ x y ⇒ 3 + tsize x + tsize y
  | abpar _ _ x y ⇒ 2 + tsize x + tsize y
  end.
```

Size of a state (number of steps to reduce).

```
Fixpoint ssize s: nat :=
  match s with
  | stop _ _ ⇒ 0
  | conf t _ _ ⇒ tsize t
  | rem _ _ x ⇒ 1 + ssize x
  | ls x t ⇒ 1 + ssize x + tsize t
  | bsl _ x t _ _ ⇒ 2 + ssize x + tsize t
  | bsr _ _ x ⇒ 1 + ssize x
  | bp _ x y ⇒ 1 + ssize x + ssize y
  end.
```

Halt state has size 0.

```
Lemma halt_size:
  ∀ st,
    halt st ↔ ssize st = 0.
```

A state decreases its size by one.

```
Lemma step_size:
  ∀ st0 e st1,
    step st0 e st1 →
    S (ssize st1) = ssize st0.

Lemma step_count:
  ∀ n t p e st,
    nstar n (conf t p e) st →
    tsize t = n + ssize st.
```

Every run terminates.

```
Theorem steps_to_stop:
  ∀ t p e st,
    nstar (tsize t) (conf t p e) st →
    halt st.
```

8.7 Numbered Labeled Transitions

`Inductive` *nlstar*: *nat* → *St* → *list Ev* → *St* → `Prop` :=
| *nlstar_refl*: ∀ *st*, *nlstar* 0 *st* [] *st*
| *nlstar_tran*: ∀ *n st0 e st1 tr st2*,
    step st0 (*Some e*) *st1* → *nlstar n st1 tr st2* → *nlstar* (*S n*) *st0* (*e* :: *tr*) *st2*
| *nlstar_silent_tran*: ∀ *n st0 st1 tr st2*,
    step st0 None st1 → *nlstar n st1 tr st2* → *nlstar* (*S n*) *st0 tr st2*.
`Hint Resolve` *nlstar_refl*.

`Lemma` *nlstar_transitive*:
  ∀ *m n st0 tr0 st1 tr1 st2*,
    nlstar m st0 tr0 st1 →
    nlstar n st1 tr1 st2 →
    nlstar (*m* + *n*) *st0* (*tr0* ++ *tr1*) *st2*.

`Lemma` *nlstar_lstar*:
  ∀ *n st0 tr st1*,
    nlstar n st0 tr st1 → *lstar st0 tr st1*.

`Lemma` *lstar_nlstar*:
  ∀ *st0 tr st1*,
    lstar st0 tr st1 →
    ∃ *n*, *nlstar n st0 tr st1*.

`Lemma` *nlstar_step_size*:
  ∀ *n st0 tr st1*,
    nlstar n st0 tr st1 →
    ssize st1 ≤ *ssize st0*.

`Lemma` *lstar_nlstar_size*:
  ∀ *st0 tr st1*,
    lstar st0 tr st1 →
    nlstar (*ssize st0* - *ssize st1*) *st0 tr st1*.

The reverse version of **nlstar**.

`Inductive` *rlstar*: *nat* → *St* → *list Ev* → *St* → `Prop` :=
| *rlstar_refl*: ∀ *st*, *rlstar* 0 *st* [] *st*
| *rlstar_tran*: ∀ *n st0 e st1 tr st2*,
    rlstar n st0 tr st1 → *step st1* (*Some e*) *st2* →
    rlstar (*S n*) *st0* (*tr* ++ [*e*]) *st2*

```
| rlstar_silent_tran: ∀ n st0 st1 tr st2,
    rlstar n st0 tr st1 → step st1 None st2 →
    rlstar (S n) st0 tr st2.
Hint Resolve rlstar_refl.

Lemma rlstar_transitive:
  ∀ m n st0 tr0 st1 tr1 st2,
    rlstar m st0 tr0 st1 →
    rlstar n st1 tr1 st2 →
    rlstar (m + n) st0 (tr0 ++ tr1) st2.

Lemma rlstar_lstar:
  ∀ n st0 tr st1,
    rlstar n st0 tr st1 → lstar st0 tr st1.

Lemma lstar_rlstar:
  ∀ st0 tr st1,
    lstar st0 tr st1 →
    ∃ n, rlstar n st0 tr st1.

Lemma rlstar_nlstar:
  ∀ n st0 tr st1,
    rlstar n st0 tr st1 ↔ nlstar n st0 tr st1.
```

Chapter 9

Main

This chapter contains the proof that traces generated from the small-step semantics are compatible with the related event system.

```
Require Import List.
Import List.ListNotations.
Open Scope list_scope.
Require Import Omega.
Require Import Preamble More_lists Term LTS Event_system Term_system
Trace.
```

The traces associated with a state.

```
Inductive traceS: St → list Ev → Prop :=
| tstop: ∀ p e,
    traceS (stop p e) []
| tconf: ∀ t tr p e,
    trace t p e tr →
    traceS (conf t p e) tr
| trem: ∀ st tr j p,
    traceS st tr →
    traceS (rem j p st)
           (tr ++
              [(rpy (pred j) p (pl st) (seval st))])
| tls: ∀ st tr1 t tr2,
    traceS st tr1 →
    trace t (pl st) (seval st) tr2 →
    traceS (ls st t) (tr1 ++ tr2)
```

```
| tbsl: ∀ st tr1 t p e tr2 j,
    traceS st tr1 →
    trace t p e tr2 →
    traceS (bsl j st t p e)
           (tr1 ++ tr2 ++
                [(join (pred j) p (seval st)
                       (aeval t p e)
                       (ss (seval st) (aeval t p e)))])
| tbsr: ∀ st tr j e,
    traceS st tr →
    traceS (bsr j e st)
           (tr ++ [(join (pred j) (pl st) e
                         (seval st)
                         (ss e (seval st)))])
| tbp: ∀ st1 tr1 st2 tr2 tr3 j,
    traceS st1 tr1 → traceS st2 tr2 →
    shuffle tr1 tr2 tr3 →
    traceS (bp j st1 st2)
           (tr3 ++ [(join (pred j) (pl st2)
                          (seval st1)
                          (seval st2)
                          (pp (seval st1) (seval st2)))]).
Hint Constructors traceS.

Fixpoint esizeS s:=
  match s with
  | stop _ _ ⇒ 0
  | conf t _ _ ⇒ esize t
  | rem _ _ st ⇒ 1 + esizeS st
  | ls st t ⇒ esizeS st + esize t
  | bsl _ st t _ _ ⇒ 1 + esizeS st + esize t
  | bsr _ _ st ⇒ 1 + esizeS st
  | bp _ st1 st2 ⇒ 1 + esizeS st1 + esizeS st2
  end.

Lemma esize_tr:
  ∀ t p e tr,
    trace t p e tr → length tr = esize t.

Lemma esizeS_tr:
```

$\forall$ *st tr*,
traceS st tr $\rightarrow$ *length tr* = *esizeS st*.

`Lemma` *step_silent_tr*:
$\forall$ *st st' tr*,
step st None st' $\rightarrow$
traceS st' tr $\rightarrow$
traceS st tr.

`Lemma` *step_evt_tr*:
$\forall$ *st st' ev tr*,
step st (*Some ev*) *st'* $\rightarrow$
traceS st' tr $\rightarrow$
traceS st (*ev*::*tr*).

`Lemma` *nlstar_trace_helper*:
$\forall$ *e p n st0 tr st1*,
step st0 e st1 $\rightarrow$
nlstar n st1 tr (*stop p* (*seval st0*)) $\rightarrow$
nlstar n st1 tr (*stop p* (*seval st1*)).

`Lemma` *nlstar_trace*:
$\forall$ *n p st tr*,
nlstar n st tr (*stop p* (*seval st*)) $\rightarrow$
traceS st tr.

A trace of the LTS is a trace of the big-step semantics.

`Lemma` *lstar_trace*:
$\forall$ *t p e tr*,
well_formed t $\rightarrow$
lstar (*conf t p e*) *tr* (*stop p* (*aeval t p e*)) $\rightarrow$
trace t p e tr.

The key theorem.

`Theorem` *ordered*:
$\forall$ *t p e tr ev0 ev1*,
well_formed t $\rightarrow$
lstar (*conf t p e*) *tr* (*stop p* (*aeval t p e*)) $\rightarrow$
prec (*ev_sys t p e*) *ev0 ev1* $\rightarrow$
earlier tr ev0 ev1.