Chapter 1

Library Manifest_NFM

```
Require Import Lia.
Require Import Relations.
Require Import Logic.FunctionalExtensionality.
Require Import Lists.List.
Import ListNotations.
Require Import String.
Require Import Cop.Copland.
Import Copland.Term.
Require Import Utils.Utilities.
```

Manifest defines a single attestation manger and its interconnections. Information includes:

1.1 asps: a list of ASPs (measurement operations the AM can preform)

1.2 M : can measure relation (other AMs the current AM knows of)

1.3 C : context relation (other AMs the current AM depends on)

1.4 Policy : local policy specific to the current AM.

1.5 Minimally includes privacy policy and may possibly include selection policy

1.6 Other information not necessary for reasoning includes:

1.7 *key* simulates public key

1.8 *address* simulates address information and

1.9 *tpm* simulates cruft necessary to initialize its TPM

`Record` *Manifest* := {

asps : *list ASP* ;
M : *list Plc* ;
C : *list Plc* ;
Policy : *ASP* → *Plc* → `Prop` ;

}.

Environment is a set of AM's each defined by a *Manifest*.

1.10 The domain of an *Environment* provides names for each *Manifest*.

1.11 Names should be the hash of their public key, but this restriction

1.12 is not enforced here.

```
Definition Environment : Type := Plc → (option Manifest).

Definition e_empty : Environment := (fun _ ⇒ None).

Definition e_update (m : Environment) (x : Plc) (v : (option Manifest)) :=
    fun x' ⇒ if plc_dec x x' then v else m x'.

Definition System := list Environment.

Definition hasASPe(k:Plc)(e:Environment)(a:ASP):Prop :=
match (e k) with
| None ⇒ False
| Some m ⇒ In a m.(asps)
end.

Fixpoint hasASPs(k:Plc)(s:System)(a:ASP):Prop :=
    match s with
    | [] ⇒ False
    | s1 :: s2 ⇒ (hasASPe k s1 a) ∨ (hasASPs k s2 a)
    end.

Theorem hasASPe_dec: ∀ k e a, {hasASPe k e a}+{~hasASPe k e a}.
Proof.
  intros k e a.
  unfold hasASPe.
  destruct (e k).
  + induction (asps m).
  ++ auto.
  ++ inverts IHl.
  +++ simpl. left. right. apply H.
  +++ simpl. assert (asp_dec : {a = a0} + {a≠a0}).
          { repeat decide equality. }
      inverts asp_dec.
  ++++ left. auto.
  ++++ right. unfold not. intros. inverts H1; auto.
  + auto.
Defined.
```

```
Theorem hasASPs_dec: ∀ k e a, {hasASPs k e a}+{~hasASPs k e a}.
Proof.
  intros k e a.
  induction e.
  + simpl in *. right. unfold not. intros. apply H.
  + simpl in *. pose proof hasASPe_dec k a0 a. inverts H.
  ++ left. left. apply H0.
  ++ inverts IHe.
  +++ left. right. apply H.
  +++ right. unfold not. intros. inverts H1; auto.
Defined.

Definition knowsOfe(k:Plc)(e:Environment)(p:Plc):Prop :=
match (e k) with
| None ⇒ False
| Some m ⇒ In p m.(M)
end.

Print System.
Print Environment.
```

Determine if place k within the system s knows of p

```
Fixpoint knowsOfs(k:Plc)(s:System)(p:Plc):Prop :=
match s with
| [] ⇒ False
| s1 :: ss ⇒ (knowsOfe k s1 p) ∨ (knowsOfs k ss p)
end.

Theorem knowsOfe_dec:∀ k e p, {(knowsOfe k e p)}+{~(knowsOfe k e p)}.
Proof.
  intros k e p.
  unfold knowsOfe.
  destruct (e k); auto.
  + induction (M m).
  ++ auto.
  ++ assert (H: {p = a} + {p ≠ a}). {repeat decide equality. }
     inversion H.
  +++ simpl. left. auto.
  +++ simpl. inverts IHl; auto. right. unfold not. intros. inverts H2; auto.
Defined.

Theorem knowsOfs_dec:∀ k s p, {(knowsOfs k s p)}+{~(knowsOfs k s p)}.
Proof.
    intros k s p.
    induction s; simpl in *.
    + right. unfold not. intros. inversion H.
```

```
    + pose proof knowsOfe_dec k a p. inverts H.
    ++ left. left. apply H0.
    ++ inverts IHs.
    +++ left. right. apply H.
    +++ right. unfold not. intros. inversion H1; auto.
Defined.
```

Determine if place k within the environment e depends on place p (the context relation)

```
Definition dependsOne (k:Plc)(e:Environment)(p:Plc):Prop :=
match (e k) with
| None ⇒ False
| Some m ⇒ In p m.(C)
end.
```

Determine if place k within the system s depends on place p (the context relation)

```
Fixpoint dependsOns (k:Plc)(s:System)(p:Plc):Prop :=
match s with
| [] ⇒ False
| s1 :: ss ⇒ (dependsOne k s1 p) ∨ (dependsOns k ss p)
end.
```

```
Theorem dependsOne_dec : ∀ k e p, {(dependsOne k e p)}+{~(dependsOne k e p)}.
Proof.
  intros k e p.
  unfold dependsOne.
  destruct (e k).
  + induction (C m).
  ++ auto.
  ++ simpl. inversion IHl.
  +++ auto.
  +++ assert (H': {a = p } + { a ≠ p}). {repeat decide equality. } inversion H'.
  ++++ left. left. apply H0.
  ++++ right. unfold not. intros. inversion H1; auto.
  + auto.
Defined.
```

```
Theorem dependsOns_dec : ∀ k s p, {dependsOns k s p} + {~ dependsOns k s p}.
Proof.
  intros. induction s.
  + simpl. auto.
  + simpl. pose proof dependsOne_dec k a p. inversion IHs.
  ++ left. right. apply H0.
  ++ inversion H.
  +++ left. left. apply H1.
  +++ right. unfold not. intros. inversion H2; auto.
```

Defined.

Is term t exectuable on the attestation manager named k in environment e? Are ASPs available at the right attesation managers and are necessary communications allowed?

```
Fixpoint executable(t:Term)(k:Plc)(e:Environment):Prop :=
match t with
| asp a ⇒ hasASPe k e a
| att p t ⇒ knowsOfe k e p → executable t p e
| lseq t1 t2 ⇒ executable t1 k e ∧ executable t2 k e
| bseq _ t1 t2 ⇒ executable t1 k e ∧ executable t2 k e
| bpar _ t1 t2 ⇒ executable t1 k e ∧ executable t2 k e
end.
```

```
Ltac right_dest_contr H := right; unfold not; intros H; destruct H; contradiction.
```

```
Theorem executable_dec:∀ t k e,{(executable t k e)}+{~(executable t k e)}.
intros. generalize k. induction t; intros.
+ unfold executable. apply hasASPe_dec.
+ simpl. pose proof knowsOfe_dec k0 e p. destruct H.
++ destruct (IHt p).
+++ left; auto.
+++ right. unfold not. intros; auto.
++ destruct (IHt p).
+++ left; auto.
+++ left. intros. congruence.
+ simpl. specialize IHt1 with k0. specialize IHt2 with k0.
  destruct IHt1,IHt2; try right_dest_contr H.
++ left. split ; assumption.
+ simpl. specialize IHt1 with k0. specialize IHt2 with k0. destruct IHt1,IHt2; try
right_dest_contr H.
++ left. split ; assumption.
+ simpl. specialize IHt1 with k0. specialize IHt2 with k0. destruct IHt1,IHt2; try
right_dest_contr H.
++ left. split ; assumption.
Defined.
```

Is term t executable on the attestation mnanager named k in

1.13 system *s*? Are ASPs available at the right attestation managers

1.14 and are necessary communications allowed?

```
Fixpoint executables(t:Term)(k:Plc)(s:System):Prop :=
```

```
  match t with
  | asp a ⇒ hasASPs k s a
  | att p t ⇒ knowsOfs k s p → executables t p s
  | lseq t1 t2 ⇒ executables t1 k s ∧ executables t2 k s
  | bseq _ t1 t2 ⇒ executables t1 k s ∧ executables t2 k s
  | bpar _ t1 t2 ⇒ executables t1 k s ∧ executables t2 k s
end.

Ltac prove_exec :=
    match goal with
    | ⊢ {executables (asp _) _ _} + {_} ⇒ unfold executables; apply hasASPs_dec
    | IHt1 : _ , IHt2 : _ ⊢ {executables _ ?k ?s} + {_} ⇒ simpl; specialize IHt1 with
k s; specialize IHt2 with k s; destruct IHt1,IHt2 ; try( left; split ; assumption)
    end.

Theorem executables_dec : ∀ t k s, {executables t k s} + {~executables t k s}.
Proof.
intros. generalize k s. induction t; intros; try prove_exec; try right_dest_contr H.
+ simpl. destruct (IHt p s0).
++ auto.
++ pose proof knowsOfs_dec k0 s0 p. destruct H.
+++ right. unfold not; intros. intuition.
+++ left. intros. congruence.
Defined.

Notation Rely := "Rely"%string.
Notation Target := "Target"%string.
Notation Appraise := "Appraise"%string.

Notation aspc1 :=
  (ASPC ALL EXTD (asp_paramsC "asp1"%string ["x"%string;"y"%string] Target Tar-
get)).
Notation aspc2 :=
  (ASPC ALL EXTD (asp_paramsC "asp2"%string ["x"%string] Target Target)).

Inductive rely_Policy : ASP → Plc → Prop :=
| p_aspc1 : ∀ p, rely_Policy aspc1 p.

Inductive tar_Policy : ASP → Plc → Prop :=
| p_aspc2 : tar_Policy aspc2 Appraise
| p_SIG : ∀ p, tar_Policy SIG p.

Inductive app_Policy : ASP → Plc → Prop :=
| p_HSH : ∀ p, app_Policy HSH p.

Definition e0 := e_empty.
Definition e_Rely :=
```

```
  e_update e_empty Rely (Some {| asps := [aspc1]; M:= [Target] ; C := [] ; Policy :=
rely_Policy |}).
Definition e_Targ :=
  e_update e_empty Target (Some {| asps := [SIG; aspc2]; M:= [Appraise] ; C := [] ;
Policy := tar_Policy|}).
Definition e_App :=
  e_update e_empty Appraise (Some {| asps := [HSH] ; M:= [] ; C := [Target] ; Policy
:= app_Policy |}).

Definition example_sys_1 := [e_Rely; e_Targ; e_App].

Example ex1: hasASPe Rely e_Rely aspc1.
Proof. unfold hasASPe. simpl. left. reflexivity. Qed.

Example ex2: hasASPe Rely e_Rely CPY → False.
Proof. unfold hasASPe. simpl. intros. inverts H. inverts H0. auto. Qed.

Example ex3: hasASPs Rely (example_sys_1) aspc1.
Proof. unfold hasASPs. unfold hasASPe. simpl. left. left. reflexivity. Qed.

Example ex4: knowsOfs Rely example_sys_1 Target.
Proof.
unfold knowsOfs. simpl. left. unfold knowsOfe. simpl. auto.
Qed.

Example ex5: knowsOfe Rely e_App Appraise → False.
Proof.
unfold knowsOfe. simpl. intros. destruct H.
Qed.

Example ex5': knowsOfs Rely example_sys_1 Appraise → False.
Proof.
unfold knowsOfs. simpl. unfold knowsOfe. simpl. intros. inverts H. inverts H0. inverts
H. apply H. inverts H0. apply H. inverts H. apply H0. apply H0.
Qed.

Example ex6: knowsOfs Rely example_sys_1 Target.
Proof.
unfold knowsOfs,knowsOfe. simpl. auto.
Qed.

Example ex7: knowsOfs Rely [e_Rely] Target.
Proof.
unfold knowsOfs,knowsOfe. simpl. auto.
Qed.

Example ex8 : dependsOne Appraise e_App Target.
Proof.
unfold dependsOne. simpl. auto.
```

```
Qed.

Example ex9 : dependsOns Appraise example_sys_1 Target.
Proof.
unfold dependsOns. simpl. unfold dependsOne. simpl. auto.
Qed.

Ltac prove_exec' :=
    simpl; auto; match goal with
                 | ⊢ hasASPe _ _ _ ⇒ cbv; left; reflexivity
                 | ⊢ knowsOfe _ _ _ ⇒ unfold knowsOfe; simpl; left; reflexivity
                 | ⊢ _ ∧ _ ⇒ split; prove_exec'
                 | ⊢ ?A ⇒ idtac A
                 end.

Example ex10: (executable (asp SIG) Target e_Targ).
Proof. prove_exec'. Qed.

Example ex11: (executable (asp CPY) Target e_App) → False.
Proof.
  intros Hcontra; cbv in *; destruct Hcontra.
Qed.

Example ex12: (executable (lseq (asp SIG) (asp SIG)) Target e_Targ).
Proof. prove_exec'. Qed.

Example ex13: (executables (lseq (asp aspc1)
                                 (att Target
                                      (lseq (asp SIG)
                                            (asp SIG))))
                    Rely example_sys_1).
Proof.
  prove_exec'; cbv; auto.
Qed.
```