Formal investigation of refinement steps and security properties in distributed systems

Christoph Feller

April 7, 2012

1 Introduction

In this report we summarize some of the formal development (in Isabelle/HOL) we did for the project MoVeSPAcI. There are two directions our research was focused on.

First are results about the security predicate RA. We implemented two examples [1] and noticed that in both we needed security predicates that were data-aware. By data-aware we mean that the predicates do not prevent an attacker to learn *if* some event occured but prevent him from learning which event (from some set) happended. After discussions with members of the mais group from Heiko Mantel we were alerted to the bachelor thesis [2] in which the security predicate RA is introduced that accomplishes exactly that. But there are two important formal tools that this thesis does not show. One is an *unwinding theorem* that would allow us two show the (global) predicate by examining the underlying state transition system (in a somehow local way). The other is a *composition theorem* that allows us to state the predicate for two composed components that each fulfill it themselves.

In in the MoVeSPAcI project we have two formal layers. The model level where abstract modelling happens and the so-called actor level which is much nearer towards the implementation. We call the actor level an refinement of the model level but it is important to note that this is no classical refinement. We also reduce indeterminism but additionally we allow the introduction of new events. This is where our second result comes in. We proved that if a set of traces satisfies RA then a similiar set of traces that just adds certain events (N-events) also fulfills the security predicate RA.

2 The security predicate RA

We first want to show our two results regarding the security predicate RA. We will start with some auxiliary notions then define RA itself. We will see that it will be necessary to extend our definition of a view to a so-called *extended view*. This will be enough for

the unwinding theorem. Then we will introduce the notion of a composition of two state event systems by defining a *well behaved composition*. We also have to specify how our extended views are allowed to behave. Then, with the help of more auxiliary lemmas, we will show the composition theorem.

theory *XView* **imports** *"I-MAKS"*
begin

2.1 Properties of the projection

When working with security properties we again and again need to restrict traces to a certain set; i.e. remove all elements from a trace that are not in the set. This is called projection and already defined in the I-MAKS project but we still need some auxiliary lemmas about this function.

lemma *dbl_proj:*
"xs ↾ X ↾ Y = xs ↾ (X ∩ Y)"

lemma *proj_inter:*
"set (a ↾ X) = set a ∩ X"

lemma *proj_cons:*
"⟦x # xs = y ↾ X⟧ ⟹ ∃zs1 zs2. y = zs1 @ x # zs2 ∧ xs = zs2 ↾ X ∧ zs1 ↾ X = []"

lemma *proj_app:*
"⟦xs @ [x] = y ↾ X⟧ ⟹ ∃zs1 zs2. y = zs1 @ x # zs2 ∧ xs = zs1 ↾ X ∧ zs2 ↾ X = []"

lemma *projection_app1a[simp]:*
"x ∈ X ⟹ (xs @ [x]) ↾ X = xs ↾ X @ [x]"

lemma *projection_app1b[simp]:*
"x ∉ X ⟹ (xs @ [x]) ↾ X = xs ↾ X"

lemma *projection_cons:*
"(x # xs) ↾ X = (if x ∈ X then x # (xs ↾ X) else (xs ↾ X))"

lemma *projection_nil[simp]:*
"[] ↾ X = []"

lemma *singleton_in_proj[simp]:*
"a ∈ X ⟹ [a] ↾ X = [a]"

lemma *projection_same_subset[intro]:*
"⟦xs ↾ X = ys ↾ X; Y ⊆ X⟧ ⟹ xs ↾ Y = ys ↾ Y"

lemma *projection_same_subset_box[intro]:*
"⟦xs ↾ X = ys ↾ Z; Y ⊆ X; Y ⊆ Z⟧ ⟹ xs ↾ Y = ys ↾ Y"

lemmas *projection_app = projection_concatenation_commute*

lemma *subs_set_proj_same:*
"set a ⊆ X ⟹ a ↾ X = a"

lemma *proj_lastcons_empty:*
assumes *"c ↾ X = []"*
shows *"(a @ b # c) ↾ X = (a @ [b]) ↾ X"*

lemma *proj_two_first[dest]:*
assumes *"set a ⊆ X ∪ Y"* **and** *"a ↾ X = c # a'"* **and** *"a ↾ Y = c # a''"*
shows *"∃b. a = c # b"*

lemma *three_part_lists_same[dest]:*
assumes *"as @ a # as' = bs @ a # bs'"*
and *"a ∉ set as"* **and** *"a ∉ set bs"*
shows *"as = bs ∧ as' = bs'"*

lemma *proj_empty_imp_notin[intro]:*
assumes *ainx: "a ∈ X"* **and** *pxe: "as ↾ X = []"*
shows *"a ∉ set as"*

2.2 Extended view and definition of RA

In this section we define an extended view and the security predicate RA itself. We do this mostly as described in [2]. The main difference is the treatment of the added relation (c.f. *xValid_View* below).

record *'e XView = "'e View" +*
ER :: "('e × 'e) set"

abbreviation *xview_rel ("_ ≈_≈ _" [1000,1000] 1000)* **where**
"x ≈xV≈ y ≡ (x,y) ∈ ER xV "

type_synonym *'e XBSP = "('e list ⇒ bool) ⇒ 'e XView ⇒ bool"*

fun *Evs* **where**
"Evs View = V View ∪ N View ∪ C View"

fun *E_star ("E*")* **where**
"E View = {tr. set tr ⊆ Evs View}"*

fun *RA :: "'e XBSP"*
where
"RA Tr xV = (∀α β. ∀c (c'::'e). α @ c # β ∈ Tr ∧ (c ≈xV≈ c')
⟶ (∃α' β'. α' ∈ E xV ∧ β' ∈ E* xV ∧ α' @ c' # β' ∈ Tr ∧*
α ↾ (V xV ∪ C xV) = α' ↾ (V xV ∪ C xV) ∧
β ↾ (V xV ∪ C xV) = β' ↾ (V xV ∪ C xV)))"
definition *xValid_View :: "'a XView ⇒ 'a set ⇒ bool"* **where**
"xValid_View view es ≡
V view ∪ N view ∪ C view = es ∧

```
V view ∩ N view = {} ∧ V view ∩ C view = {} ∧ N view ∩ C view = {} ∧
(∀ e ∈ es. e ≈view≈ e) ∧
(∀ e1 e2. e1 ≈view≈ e2 ⟶ (e1 ∈ es ∧ e2 ∈ es)) ∧
(∀ e1 e2. e1 ≈view≈ e2 ⟶ (e1 = e2) ∨ (e1 ∉ N view ∧ e2 ∉ N view))"
```

lemma *xValid_xldom2[intro]:*
"⟦*xValid_View view es; e ∈ C view*⟧ ⟹ ∃ *e'. e ≈view≈ e'*"

2.3 Unwinding theorem for RA

An unwinding theorem tells us how to derive a global property like RA from local properties. We call RA a global in the sense that it makes a statement about the whole set of traces that represents a state event system's semantics.

In contrast we call the following two properties *lrrf* and *osc'* local because they regard the concrete states and event transitions of the state event system. So one can argue that these are easier to prove than the property RA itself.

We start with some auxiliary lemmas.

context *StateEventSystem* **begin**

lemma *reach_path_reach[intro]:*
assumes *"reachable s1"* **and** *"path s1 a = Some s2"*
shows *"reachable s2"*

lemma *path_in_e[intro]:*
assumes *"enabled s a"* **shows** *"set a ⊆ E_ES induced_ES"*

The first local property, *lrrf*, captures the main idea of RA, namely that we have to be able to replace a state transition emitting an event with a state transition that emits an equivalent event.

definition *lrrf :: "'event XView ⇒ 'state unwindingrelation ⇒ bool"*
where
```
"lrrf view ur ≡  ∀ s1 ∈ (S_SES SES). ∀ s1' ∈ (S_SES SES). ∀ s2 ∈ (S_SES SES).
                 ∀ c c'.
                (reachable s1) ∧ ((T_SES SES) s1 c) = Some s2 ∧ (s1,s1') ∈ ur
                  ∧ (c ≈view≈ c')
                    ⟶ (∃ s2' ∈ (S_SES SES).  ((T_SES SES) s1' c') = Some s2' ∧
((s2, s2') ∈ ur))"
```

For the local property *osc* we conceptually just take this from the I-MAKS framework. For technical reasons we still have to make small change to the original definition so we introduce *osc'*. This allows us to also take an extended view as a parameter of *osc'* (even though we do not need the additional relation).

definition

```
osc' :: "('event,'a) View_scheme  ⇒ 'state unwindingrelation ⇒ bool"
where
"osc' view ur ≡
(∀s1 ∈ (S_SES SES). ∀s1' ∈ (S_SES SES). ∀s2' ∈ (S_SES SES).
∀e ∈ ((E_SES SES) - (C view)).
( ((reachable s1) ∧ (reachable s1') ∧ (T_SES SES) s1' e = Some s2' ∧
((s1', s1) ∈ ur))
⟶ (∃s2 ∈ (S_SES SES). ∃δ. (δ ↾ (C view) = [] ∧
(δ ↾ (V view)) = ([e] ↾ (V view)) ∧ (path s1 δ) = Some s2 ∧
((s2', s2) ∈ ur)))) )"
```

We simply copied the following lemma from the I-MAKS framework and changed references from *osc* to *osc'*.

```
lemma osc'_property:
"⋀s1 s1'. ⟦ osc' view ur;
    s1 ∈ S_SES SES; s1' ∈ S_SES SES;
    α ↾ (C view) = [];
    reachable s1; reachable s1'; enabled s1' α; (s1', s1) ∈ ur ⟧
  ⟹ (∃α'. α' ↾ (C view) = [] ∧
    α' ↾ V view = α ↾ V view ∧ enabled s1 α')"
```

```
lemma lrrf_osc'_property:
assumes lrrf: "lrrf view ur" and osc': "osc' view ur"
    and validV: "xValid_View view (E_ES induced_ES)"
    and s1in: "s1 ∈ S_SES SES" and s1'in: "s1' ∈ S_SES SES"
    and alphapath: "path s1 α = Some s2"
    and startur: "(s1,s1') ∈ ur"
    and reachs1: "reachable s1" and reachs1': "reachable s1'"
  shows "∃α' s2'. α' ↾ (V view ∪ C view) = α ↾ (V view ∪ C view)
             ∧ path s1' α' = Some s2' ∧ (s2,s2') ∈ ur"
```

With the auxiliary lemma above we get the unwinding theorem for RA.

```
theorem unwindRA:
assumes lrrf: "lrrf view ur" and osc': "osc' view ur"
    and urrefl: "∀s. (s,s) ∈ ur"
    and validV: "xValid_View view (E_ES induced_ES)"
shows "RA (Tr_ES induced_ES) view"

end
```

2.4 Compositionality

In this section we will show that there are reasonable conditions under which RA is compositional, i.e. we can show that the composition of two systems that satisfy RA also satisfies RA.

This locale corresponds to the Compositionality locale in the I-MAKS framework. It uses extended views and also adds two axioms (*relV1_subset*, *relV2_subset*) related to the event relation in the extended view. As it does not have to work for as many security predicates it has a much simpler notion of a well-behaved composition.

```
locale XCompositionality = ComposedEventSystem +
fixes xV :: "'a XView"
 and  xV1 :: "'a XView"
 and  xV2 :: "'a XView"


  assumes validViewV[intro]: "xValid_View xV (E_ES (ES1 ∥ ES2))"
  and validViewV1[intro]: "xValid_View xV1 (E_ES ES1)"
  and validViewV2[intro]: "xValid_View xV2 (E_ES ES2)"


  and Vv_inter_E1_is_Vv1: "(V xV) ∩ (E_ES ES1) = (V xV1)"
  and Vv_inter_E2_is_Vv2: "(V xV) ∩ (E_ES ES2) = (V xV2)"
  and Cv_inter_E1_subsetof_Cv1: "(C xV) ∩ (E_ES ES1) ⊆ (C xV1)"
  and Cv_inter_E2_subsetof_Cv2: "(C xV) ∩ (E_ES ES2) ⊆ (C xV2)"
  and disjoint_Nv1_Nv2[simp]: "(N xV1) ∩ (N xV2) = {}"

  and relV1_subset[intro]:
      "⟦a ≈xV≈ b; a ∈ (E_ES ES1) ∨ b ∈ (E_ES ES1)⟧ ⟹ a ≈xV1≈ b"
  and relV2_subset[intro]:
      "⟦a ≈xV≈ b; a ∈ (E_ES ES2) ∨ b ∈ (E_ES ES2)⟧ ⟹ a ≈xV2≈ b"


  and well_behaved_composition[intro]:
      "(N xV1) ∩ (E_ES ES2) = {} ∧ (N xV2) ∩ (E_ES ES1) = {}"

begin

end
```

We add some useful lemmas about event systems.

```
context EventSystem begin

lemma set_trace_in_events[intro]:
"x ∈ (Tr_ES ES) ⟹ set x ⊆ (E_ES ES)"

lemma app2inEvs[intro]:
assumes "a @ b ∈ Tr_ES ES"
shows "set b ⊆ E_ES ES"

lemma proj_app_in[intro]:
assumes "a @ b ∈ Tr_ES ES"
shows "a ∈ Tr_ES ES"

lemma proj_cons_in[intro]:
```

assumes *"a # as ∈ Tr_ES ES"*
shows *"[a] ∈ Tr_ES ES"*

lemma *proj_appcons_in[intro]:*
assumes *"a @ b # c ∈ Tr_ES ES"*
shows *"a @ [b] ∈ Tr_ES ES"*

end

For the compositionality theorem we need a function that can compose two traces in a certain way. This function is *tzip2* and its properties will be shown below. Even if its definition might seem rather unwieldy the properties should be able to convey a sense of *tzip2*'s purpose.

fun *ocons :: "'e ⇒ 'e list option ⇒ 'e list option" ("_ #? _" [1000,1000] 1000)*
where
"ocons e None = None"
| "ocons e (Some l) = Some (e # l)"

lemma *ocons_some[elim!]:*
"⟦a #? as = Some zs; ⋀zs'. ⟦as = Some zs'; a # zs' = zs⟧ ⟹ P a as zs⟧ ⟹ P a as zs"

fun *tzip2 :: "'e set ⇒ 'e set ⇒ 'e set ⇒ 'e list ⇒ 'e list ⇒ 'e list ⇀ 'e list"*
where
"tzip2 X Y Z [] [] [] = Some []"
| "tzip2 X Y Z (z#zs) [] [] =
(if z ∈ Z then None else (tzip2 X Y Z zs [] []))"
| "tzip2 X Y Z [] (a#as) [] =
(if a ∈ Z ∨ a ∈ (X ∩ Y) then None else a #? (tzip2 X Y Z [] as []))"
| "tzip2 X Y Z (z#zs) (a#as) [] =
(if a ∈ (X ∩ Y) then None else
(if a = z then a #? (tzip2 X Y Z zs as []) else
(if z ∉ Z then (tzip2 X Y Z zs (a#as) []) else
(if a ∉ Z then a #? (tzip2 X Y Z (z#zs) as []) else None))))"
| "tzip2 X Y Z [] [] (b#bs) =
(if b ∈ Z ∨ b ∈ (X ∩ Y) then None else b #? (tzip2 X Y Z [] [] bs))"
| "tzip2 X Y Z (z#zs) [] (b#bs) =
(if b ∈ (X ∩ Y) then None else
(if b = z then b #? (tzip2 X Y Z zs [] bs) else
(if z ∉ Z then tzip2 X Y Z zs [] (b#bs) else
(if b ∉ Z then b #? (tzip2 X Y Z (z#zs) [] bs) else None))))"
| "tzip2 X Y Z [] (a#as) (b#bs) =
(if a ∈ Z ∨ b ∈ Z then None else
(if a = b then a #? (tzip2 X Y Z [] as bs) else
(if a ∉ (X ∩ Y) then a #? (tzip2 X Y Z [] as (b#bs)) else

```
                        (if b ∉ (X ∩ Y) then b #? (tzip2 X Y Z [] (a#as) bs) else
                            None))))"
| "tzip2 X Y Z (z#zs) (a#as) (b#bs) =
     (if z ∉ Z then (tzip2 X Y Z zs (a#as) (b#bs)) else
       (if a = b ∧ a ∈ Z ∧ a ≠ z then None else
         (if a = b ∧ a ∈ Z then a #? (tzip2 X Y Z zs as bs) else
           (if a = b then a #? (tzip2 X Y Z (z#zs) as bs) else
             (if a ∉ (X ∩ Y) ∧ a ∉ Z then a #? (tzip2 X Y Z (z#zs) as (b#bs)) else
               (if a ∉ (X ∩ Y) ∧ a = z then a #? (tzip2 X Y Z zs as (b#bs)) else
                 (if b ∉ (X ∩ Y) ∧ b ∉ Z then b #? (tzip2 X Y Z (z#zs) (a#as) bs)

                  else
                   (if b ∉ (X ∩ Y) ∧ b = z then b #? (tzip2 X Y Z zs (a#as) bs)

                    else None ))))))))"
```

lemma *tzip2_keeps_XY[intro]:*
assumes *asX: "set as ⊆ X"* **and** *asY: "set bs ⊆ Y"* **and** *tzips: "tzip2 X Y Z zs as bs = Some rs"*
shows *"set rs ⊆ X ∪ Y"*

lemma *ifeqE:*
"⟦(if Q then m else n) = v; ⟦Q; m = v⟧ ⟹ P; ⟦¬Q; n = v⟧ ⟹ P⟧ ⟹ P"

lemma *tzip2_proj1:*
assumes *"tzip2 X Y Z zs as bs = Some rs"* **and** *"set as ⊆ X"* **and** *"set bs ⊆ Y"*
shows *"rs ↾ X = as"*

lemma *tzip2_proj2':*
assumes *"tzip2 X Y Z zs as bs = Some rs"* **and** *"set as ⊆ X"* **and** *"set bs ⊆ Y"*
shows *"rs ↾ Y = bs"*

lemma *tzip2_zproj1:*
assumes *"tzip2 X Y Z zs as bs = Some rs"* **and** *"set as ⊆ X"* **and** *"set bs ⊆ Y"*
shows *"rs ↾ Z = zs ↾ Z"*

lemma *proj_eq_cons[simp]:*
"((a # as) ↾ X = (a # bs) ↾ X) = (as ↾ X = bs ↾ X)"

lemma *subsetinter_aux:*
"⟦a ∈ X; a ∈ Y; X ∩ Y ⊆ Z⟧ ⟹ a ∈ Z"

lemma *tzip2_not_None[intro]:*
assumes *"set as ⊆ X"* **and** *"set bs ⊆ Y"* **and** *"as ↾ (X ∩ Y) = bs ↾ (X ∩ Y)"*
  and *"zs ↾ Z1 = as ↾ Z1"* **and** *"zs ↾ Z2 = bs ↾ Z2"*
  and *"Z = (Z1 ∪ Z2)"* **and** *"Z1 ⊆ X"* **and** *"Z2 ⊆ Y"* **and** *"(X ∩ Y) ⊆ (Z1 ∪ Z2)"*
  and *"Z1 ∩ Y ⊆ Z2"* **and** *"Z2 ∩ X ⊆ Z1"*
shows *"tzip2 X Y Z zs as bs ≠ None"*

context *XCompositionality* **begin**

lemma *E_ES_comp[intro]:*
"E_ES (ES1 ‖ ES2) = E_ES ES1 ∪ E_ES ES2"

lemma *View_eq_ES_comp[intro]:*
"V xV ∪ N xV ∪ C xV = E_ES (ES1 ‖ ES2)"

lemma *View1_eq_ES1[intro]:*
"V xV1 ∪ N xV1 ∪ C xV1 = E_ES ES1"

lemma *View1_eq_ES1'[intro]:*
"V xV1 ∪ C xV1 ∪ N xV1 = E_ES ES1"

lemma *View2_eq_ES2[intro]:*
"V xV2 ∪ N xV2 ∪ C xV2 = E_ES ES2"

lemma *View2_eq_ES2'[intro]:*
"V xV2 ∪ C xV2 ∪ N xV2 = E_ES ES2"

lemmas *View_simps[intro] = View_eq_ES_comp View1_eq_ES1 View2_eq_ES2*

lemma *view_proj_aux[simp]:*
"xValid_View view es ⟹
 a ↾ es ↾ (V view ∪ C view) = a ↾ (V view ∪ C view)"

lemma *xcomp_es1_es2_inter_view:*
"E_ES ES1 ∩ E_ES ES2 = (V xV1 ∪ C xV1) ∩ (V xV2 ∪ C xV2)"

lemma *inter_union_del:*
"M ∩ Y = {} ⟹ (X ∪ Y) ∩ M = X ∩ M"

lemma *inter_empty_union_add:*
"(X ∪ Y ∪ Z) ∩ M = {} ⟹ (X ∪ Y) ∩ M = {}"

lemma *xcomp_view_inter_woN1:*
"(V xV1 ∪ C xV1 ∪ N xV1) ∩ (V xV2 ∪ C xV2) = (V xV1 ∪ C xV1) ∩ (V xV2 ∪ C xV2)"

lemma *xcomp_view_inter_woN2:*

```
"(V xV2 ∪ C xV2 ∪ N xV2) ∩ (V xV1 ∪ C xV1) = (V xV2 ∪ C xV2) ∩ (V xV1 ∪ C xV1)"
```

lemma *VC_smallerV1C1V2C2:*
```
"(V xV ∪ C xV) ⊆ (V xV1 ∪ C xV1 ∪ V xV2 ∪ C xV2)"
```

lemma *rezip_traces_3part:*
assumes *tr1in:* `"a1 @ c # b1 ∈ Tr_ES ES1"` **and** *tr2in:* `"a2 @ c # b2 ∈ Tr_ES ES2"`
and *cin:* `"c ∈ E_ES ES1 ∩ E_ES ES2"`
and *a1a2same:* `"a1 ↾ (E_ES ES1 ∩ E_ES ES2) = a2 ↾ (E_ES ES1 ∩ E_ES ES2)"`
and *b1b2same:* `"b1 ↾ (E_ES ES1 ∩ E_ES ES2) = b2 ↾ (E_ES ES1 ∩ E_ES ES2)"`
and *ain:* `"a @ cc # b ∈ Tr_ES (ES1 ∥ ES2)"`
and *a1proj:* `"a1 ↾ (V xV1 ∪ C xV1) = a ↾ (V xV1 ∪ C xV1)"`
and *a2proj:* `"a2 ↾ (V xV2 ∪ C xV2) = a ↾ (V xV2 ∪ C xV2)"`
and *b1proj:* `"b1 ↾ (V xV1 ∪ C xV1) = b ↾ (V xV1 ∪ C xV1)"`
and *b2proj:* `"b2 ↾ (V xV2 ∪ C xV2) = b ↾ (V xV2 ∪ C xV2)"`
shows `"∃a3 b3. a3 @ c # b3 ∈ Tr_ES (ES1 ∥ ES2) ∧`
`a3 ↾ E_ES ES1 = a1 ∧ b3 ↾ E_ES ES1 = b1 ∧`
`a3 ↾ E_ES ES2 = a2 ∧ b3 ↾ E_ES ES2 = b2 ∧`
`a3 ↾ (V xV ∪ C xV) = a ↾ (V xV ∪ C xV) ∧`
`b3 ↾ (V xV ∪ C xV) = b ↾ (V xV ∪ C xV)"`

lemma *rezip_traces_3part_noc2:*
assumes *tr1in:* `"a1 @ c # b1 ∈ Tr_ES ES1"` **and** *tr2in:* `"a2 @ b2 ∈ Tr_ES ES2"`
and *cin1:* `"c ∈ E_ES ES1"` **and** *cnin2:* `"c ∉ E_ES ES2"`
and *a1a2same:* `"a1 ↾ (E_ES ES1 ∩ E_ES ES2) = a2 ↾ (E_ES ES1 ∩ E_ES ES2)"`
and *b1b2same:* `"b1 ↾ (E_ES ES1 ∩ E_ES ES2) = b2 ↾ (E_ES ES1 ∩ E_ES ES2)"`
and *ain:* `"a @ cc # b ∈ Tr_ES (ES1 ∥ ES2)"`
and *a1proj:* `"a1 ↾ (V xV1 ∪ C xV1) = a ↾ (V xV1 ∪ C xV1)"`
and *a2proj:* `"a2 ↾ (V xV2 ∪ C xV2) = a ↾ (V xV2 ∪ C xV2)"`
and *b1proj:* `"b1 ↾ (V xV1 ∪ C xV1) = b ↾ (V xV1 ∪ C xV1)"`
and *b2proj:* `"b2 ↾ (V xV2 ∪ C xV2) = b ↾ (V xV2 ∪ C xV2)"`
shows `"∃a3 b3. a3 @ c # b3 ∈ Tr_ES (ES1 ∥ ES2) ∧`
`a3 ↾ E_ES ES1 = a1 ∧ b3 ↾ E_ES ES1 = b1 ∧`
`a3 ↾ E_ES ES2 = a2 ∧ b3 ↾ E_ES ES2 = b2 ∧`
`a3 ↾ (V xV ∪ C xV) = a ↾ (V xV ∪ C xV) ∧`
`b3 ↾ (V xV ∪ C xV) = b ↾ (V xV ∪ C xV)"`

lemma *rezip_traces_3part_noc1:*
assumes *tr1in:* `"a1 @ b1 ∈ Tr_ES ES1"` **and** *tr2in:* `"a2 @ c # b2 ∈ Tr_ES ES2"`
and *cnin1:* `"c ∉ E_ES ES1"` **and** *cin2:* `"c ∈ E_ES ES2"`
and *a1a2same:* `"a1 ↾ (E_ES ES1 ∩ E_ES ES2) = a2 ↾ (E_ES ES1 ∩ E_ES ES2)"`
and *b1b2same:* `"b1 ↾ (E_ES ES1 ∩ E_ES ES2) = b2 ↾ (E_ES ES1 ∩ E_ES ES2)"`
and *ain:* `"a @ cc # b ∈ Tr_ES (ES1 ∥ ES2)"`
and *a1proj:* `"a1 ↾ (V xV1 ∪ C xV1) = a ↾ (V xV1 ∪ C xV1)"`

```
    and a2proj: "a2 ↾ (V xV2 ∪ C xV2) = a ↾ (V xV2 ∪ C xV2)"
    and b1proj: "b1 ↾ (V xV1 ∪ C xV1) = b ↾ (V xV1 ∪ C xV1)"
    and b2proj: "b2 ↾ (V xV2 ∪ C xV2) = b ↾ (V xV2 ∪ C xV2)"
shows "∃a3 b3. a3 @ c # b3 ∈ Tr_ES (ES1 ∥ ES2) ∧
               a3 ↾ E_ES ES1 = a1 ∧ b3 ↾ E_ES ES1 = b1 ∧
               a3 ↾ E_ES ES2 = a2 ∧ b3 ↾ E_ES ES2 = b2 ∧
               a3 ↾ (V xV ∪ C xV) = a ↾ (V xV ∪ C xV) ∧
               b3 ↾ (V xV ∪ C xV) = b ↾ (V xV ∪ C xV)"
```

This theorem effectively proves the compositionality of RA although a more concise formulation is found below in the corrolary *compo_RA*.

```
theorem compo_RA':
fixes a b c c'
assumes ra1: "RA (Tr_ES ES1) xV1" and ra2: "RA (Tr_ES ES2) xV2"
assumes acb: "a @ c # b ∈ Tr_ES (ES1 ∥ ES2)"
assumes cxVc': "c ≈xV≈ c'"
shows " ∃α'. set α' ⊆ V xV ∪ N xV ∪ C xV ∧
               (∃β'. set β' ⊆ V xV ∪ N xV ∪ C xV ∧
                     α' @ c' # β' ∈ Tr_ES (ES1 ∥ ES2) ∧
                     a ↾ (V xV ∪ C xV) = α' ↾ (V xV ∪ C xV) ∧
                     b ↾ (V xV ∪ C xV) = β' ↾ (V xV ∪ C xV))"
      (is "?RA_smt a b xV (Tr_ES (ES1 ∥ ES2))")
```

Finally we get the compositionality of RA.

```
corollary compo_RA[intro]:
"⟦RA (Tr_ES ES1) xV1; RA (Tr_ES ES2) xV2⟧
 ⟹ RA (Tr_ES (ES1 ∥ ES2)) xV"

end


end
```

3 Refinement predicates

We considered too kinds of refinement which we will explain in the next sections.

3.1 Adding additional events

On the model level we often have an abstract view of the system. That means that we cut out unimportant technical details/methods. If we assume that these parts do not affect the our system's security it is indeed apropriate not to consider these in our security analysis on the model level.

Technically we have events on the concrete level that do not appear on the model level an can be classified as N-events in the MAKS framework. That means that these events

can not be seen by the attacker but he is allowed to gain information about them. In the following we show that the addition of these events preserves the security predicate BSD.

```
theory RefTest imports BSPs XView
begin

primrec filterFrom :: "'a ⇒ 'a set ⇒ 'a list ⇒ 'a list"
where
  "filterFrom a P [] = []"
| "filterFrom a P (x # xs) =
            (if a = x then x # (filter P xs) else x # (filterFrom a P xs))"

primrec filterUntil :: "'a ⇒ 'a set ⇒ 'a list ⇒ 'a list"
where
  "filterUntil a P [] = []"
| "filterUntil a P (x # xs) =
           (if a = x then (x # xs) else
              (if P x then (x # (filterUntil a P xs)) else (filterUntil a P xs)))"

primrec filterBetween :: "'a ⇒ 'a ⇒ 'a set ⇒ 'a list ⇒ 'a list"
where
  "filterBetween a1 a2 P [] = []"
| "filterBetween a1 a2 P (x # xs) =
           (if a1 = x then x # (filterUntil a2 P xs)
                      else x # (filterBetween a1 a2 P xs))"

lemma filterUntilApp1[simp]:
"⟦a ∈ set p⟧ ⟹ filterUntil a P (p @ q) = (filterUntil a P p) @ q"

lemma filterUntilApp2[simp]:
"⟦a ∉ set p⟧ ⟹ filterUntil a P (p @ q) = (filter P p) @ (filterUntil a P q)"

lemma filterUntilNonexist[simp]:
"⟦a ∉ set p⟧ ⟹ filterUntil a P p = filter P p"

lemma filterUntilApp1b[rule_format]:
"a ∈ set p ⟶
  (∃p' p''. p = p' @ a # p'' ∧ a ∉ set p' ∧
            filterUntil a P (p' @ a # p'') =
                           (filter P p') @ a # p'')"

lemma filterBetweenNoStart[simp]:
"e1 ∉ set p ⟹ filterBetween e1 e2 E p = p"

lemma filterBetweenApp2[simp]:
"⟦a1 ∉ set p⟧ ⟹ filterBetween a1 a2 P (p @ q) = p @ (filterBetween a1 a2 P q)"
```

lemma *filterBetweenApp1':*
```
"a1 ∈ set p
 ⟶ (∃p' p''. p = p' @ a1 # p'' ∧ a1 ∉ set p' ∧
      filterBetween a1 a2 P (p @ q) = p' @ [a1] @ filterUntil a2 P (p'' @ q))"
```

lemma *filterBetweenApp1[simp]:*
```
"a1 ∈ set p
 ⟹ ∃p' p''. p = p' @ a1 # p'' ∧ a1 ∉ set p' ∧
     filterBetween a1 a2 P (p @ q) = p' @ [a1] @ filterUntil a2 P (p'' @ q)"
```

lemma *filterfilterUntil[simp]:*
```
"filter E (filterUntil e E l) = filter E l"
```

lemma *filterfilterBetween[simp]:*
```
"filter E (filterBetween e1 e2 E l) = filter E l"
```

lemma *filterUntilfilterUntil[simp]:*
```
"filterUntil e E (filterUntil e E l) = filterUntil e E l"
```

lemma *filterBetweenfilterBetween[simp]:*
```
"filterBetween e1 e2 E (filterBetween e1 e2 E l) =
 filterBetween e1 e2 E l"
```

lemma *filterfilterBetweenImage:*
```
"filter E ‘ filterBetween e1 e2 E ‘ X = filter E ‘ X"
```

lemma *filterBetweenfilterBetweenImage[simp]:*
```
"filterBetween e1 e2 E ‘ filterBetween e1 e2 E ‘ X =
 filterBetween e1 e2 E ‘ X"
```

lemma *filterfilter_subset[intro]:*
```
"E ⊆ E' ⟹ filter E (filter E' l) = filter E l"
```

lemma *filterfilter_subset_app[intro]:*
```
"E ⊆ E' ⟹ filter E (filter E' l @ m) = filter E (l @ m)"
```

lemma *filterfilterUntil_subset[intro]:*
```
"E ⊆ E' ⟹ filter E (filterUntil e E' l) = filter E l"
```

lemma *filterfilterBetween_subset[intro]:*
```
"E ⊆ E' ⟹ filter E (filterBetween e1 e2 E' l) = filter E l"
```

lemma *filterUntilprefix[intro]:*
```
"a ≼ b ⟹ filterUntil e E a ≼ filterUntil e E b"
```

lemma *filterBetweenprefix[intro]:*
```
"a ≼ b ⟹ filterBetween e1 e2 E a ≼ filterBetween e1 e2 E b"
```

Here we define what a refinement is: A predicate that relates to sets of events (events

are given by the type parameter e).

type_synonym *'e ref = "('e list set) ⇒ ('e list set) ⇒ bool"*

We now define a refinement that adds events into a part of the trace. The events *e1* and *e2* determine the part where new events can be added. The events to be added come from $E' - E$.

definition *RefAddPart :: "'e set ⇒ 'e set ⇒ 'e ⇒ 'e ⇒ 'e ref"*
where
"RefAddPart E E' e1 e2 Tr Tr' ≡
filter E ‘ Tr = Tr ∧ filter E' ‘ Tr' = Tr' ∧
Tr' = {tr. filterBetween e1 e2 E tr ∈ Tr}"

definition *Evs :: "('e,'a) View_scheme ⇒ 'e set"*
where
"Evs v = V v ∪ N v ∪ C v"

lemma *filterEvsV[simp]:*
"filter (λx. Evs v x ∧ x ∈ V v) l = filter (λx. x ∈ V v) l"

lemma *Evs_V_c[simp]:*
"c ∈ C v ⟹ Evs v c"

lemma *prefixclosed_RefAddPart:*
"⟦prefixclosed Tr; RefAddPart (Evs v) (Evs v') e1 e2 Tr Tr'⟧
⟹ prefixclosed Tr'"

lemma *filterImageAll:*
"(filter E ‘ Tr = Tr) ⟹ (∀x ∈ Tr. filter E x ∈ Tr)"

lemma *filterEqContr[dest]:*
assumes *fs: "filter E ‘ Tr = Tr"* **and** *xinTr: "x ∈ Tr"*
and *fneq: "filter E x ≠ x"*
shows *"False"*

lemma *image_same[intro]:*
"∀x ∈ M. f x = x ⟹ f ‘ M = M"

lemma *filterImageEq:*
"(filter E ‘ Tr = Tr) = (∀x ∈ Tr. filter E x = x)"

lemma *filterBetweenEqImpImage[intro]:*
"∀x ∈ Tr. filterBetween e1 e2 E x = x
⟹ filterBetween e1 e2 E ‘ Tr = Tr"

lemma *filterNotCons[simp]:*
"¬ filter E xs = x # xs"

lemma *filterAppSplit:*
"(filter E (a @ b) = a @ b) = (filter E a = a ∧ filter E b = b)"

lemma *filterSameImpFilterBetweenSame[intro]:*
"filter E x = x ⟹ filterBetween e1 e2 E x = x"

lemma *RefAddPart_subset[intro]:*
"RefAddPart E E' e1 e2 Tr Tr' ⟹ Tr ⊆ Tr'"

lemma *RefAddPart_filter[intro]:*
"⟦RefAddPart E E' e1 e2 Tr Tr'; tr ∈ Tr'⟧
⟹ filterBetween e1 e2 E tr ∈ Tr"

lemma *RefAddPart_filter2[intro]:*
"⟦RefAddPart E E' e1 e2 Tr Tr'; filterBetween e1 e2 E tr ∈ Tr⟧
⟹ tr ∈ Tr'"

lemma *RefAddPart_Tr_filter[simp]:*
"⟦RefAddPart E E' e1 e2 Tr Tr'; tr ∈ Tr⟧
⟹ filter E tr = tr"

lemma *empty_filter_notin_list[intro]:*
"⟦x ∈ M; filter M xs = []⟧ ⟹ x ∉ set xs"

lemma *empty_proj_notin_list[intro]:*
"⟦x ∈ M; xs ↑ M = []⟧ ⟹ x ∉ set xs"

lemma *RefAddPart_inTr_inE[intro]:*
"⟦RefAddPart E E' e1 e2 Tr Tr'; x ∈ set tr; tr ∈ Tr⟧
⟹ E x"

lemma *prefixclosed_App[intro]:*
"⟦prefixclosed Tr; a @ b ∈ Tr⟧ ⟹ a ∈ Tr"

lemma *projection_filter:*
"xs ↑ M = filter M xs"

lemma *RefAddPart_App_filter[intro]:*
"⟦RefAddPart E E' e1 e2 Tr Tr'; a @ b ∈ Tr⟧
⟹ filter E a = a ∧ filter E b = b"

lemma *filterSame_imp_filterUntilSame[intro]:*
"filter E l = l ⟹ filterUntil e E l = l"

lemma *RefAddPart_preserves_BSD:*
"⟦RefAddPart (Evs v) (Evs v') e1 e2 Tr Tr'; BSD Tr v;

```
  v' = v(|N:=N v∪N'|)⟧
⟹ BSD Tr' v'"
```

end

3.2 Reduction of Non-Determinism

We also consider a simple reduction of non-determinism: we assume a system that does something for an arbitrary number of times and then stops. We show that if we restrict the number of times said system can perform the action this reduction in the number of traces preserves the security predicate RA.

theory *NDRef* **imports** *XView*
begin

In this theory we prove that a certain kind of refinement still preserves the security predicate RA. This refinement is given by the predicate *NDRef* below. Here we assume that because of non-determinism we have an arbitrary amount of certain events in a trace. We now remove that non-determinism by limiting the number of those events.

```
definition NDRef :: "nat ⇒ ('e set) ⇒ ('e list) set ⇒ ('e list) set ⇒ bool"
where
"NDRef n Ev Tr Tr' ≡ Tr' = {t ∈ Tr. length (t ↾ Ev) ≤ n}"

lemma length_subset_proj[intro]:
"X ⊆ Y ⟹ length (a ↾ X) ≤ length (a ↾ Y)"

lemma proj_same_subset[intro]:
"⟦X ⊆ Y; a ↾ Y = b ↾ Y⟧ ⟹ a ↾ X = b ↾ X"


theorem NDRef_preserves_RA:
assumes ra: "RA Tr xV" and ndref: "NDRef n Ev Tr Tr'" and evsubs: "Ev ⊆ (V xV
∪ C xV)"
    and cdomev: "∀c c'. (c ≈xV≈ c') ⟶ ((c ∈ Ev) ∧ (c' ∈ Ev))"
shows "RA Tr' xV"

end
```

4 Conclusion / Future Work

The results about the security predicate RA extends the variety of models we can examine with the methods of the MAKS framework. The main goal is to be able to include more realistic examples.

The two refinement predicates give a glimpse at the techniques used to prove security properties on the actor level. But this is also the point where more work has to be done. We need to check if it is feasible to proof that two sets of traces fulfill one of these

refinements. Depending on that we have to adjust our techniques to either support these proof in a generic way or change

References

[1] Christoph Feller. Towards relating security properties on the model layer to implementations - the virtual mall and e-voting case studies. Technical report, University of Kaiserslautern, March 2012.

[2] Tobias Plötz. Vertraulichkeit von Werten im "Modular Assembly Kit for Security" (Bachelor Thesis). Bachelor's thesis, TU Darmstadt, 2011.