

Applying the Isabelle Insider Framework to Airplane Security

Florian Kammüller and Manfred Kerber

March 31, 2020

Abstract

Avionics is one of the fields in which verification methods have been pioneered and brought a new level of reliability to systems used in safety critical environments. Tragedies, like the 2015 insider attack on a German airplane, in which all 150 people on board died, show that safety and security crucially depend not only on the well functioning of systems but also on the way how humans interact with the systems. Policies are a way to describe how humans should behave in their interactions with technical systems, formal reasoning about such policies requires integrating the human factor into the verification process.

We model insider attacks on airplanes using logical modelling and analysis of infrastructure models and policies with actors to scrutinize security policies in the presence of insiders [1]. The Isabelle Insider framework framework has been first presented in [3]. Triggered by case studies, like the present one of airplane security, it has been greatly extended now formalizing Kripke structures and the temporal logic CTL to enable reasoning on dynamic system states. Furthermore, we illustrate that Isabelle modelling and invariant reasoning reveal subtle security assumptions: the formal development uses locales to model the assumptions on insider and their access credentials. Technically interesting is how the locale is interpreted in the presence of an abstract type declaration for actor in the Insider framework redefining this type declaration at a later stage like a “post-hoc type definition” as proposed in [4]. The case study and the application of the methodology are described in more detail in the preprint [2].

Contents

1	Kripke structures and CTL	2
1.1	Lemmas to support least and greatest fixpoints	2
1.2	Generic type of state with state transition and CTL Operators	5
1.3	Kripke structure and Modelchecking	6
1.4	Lemmas for CTL operators	6
1.4.1	EF lemmas	6
1.4.2	AG lemmas	8

1 Kripke structures and CTL

We apply Kripke structures and CTL to model state based systems and analyse properties under dynamic state changes. Snapshots of systems are the states on which we define a state transition. Temporal logic is then employed to express security and privacy properties.

```
theory MC
imports Main
begin
```

1.1 Lemmas to support least and greatest fixpoints

```
definition monotone :: ('a set  $\Rightarrow$  'a set)  $\Rightarrow$  bool
where monotone  $\tau \equiv (\forall p\ q. p \subseteq q \longrightarrow \tau\ p \subseteq \tau\ q)$ 
```

```
lemma monotoneE: monotone  $\tau \Longrightarrow p \subseteq q \Longrightarrow \tau\ p \subseteq \tau\ q$ 
<proof>
```

```
lemma lfp1: monotone  $\tau \longrightarrow (\text{lfp } \tau = \bigcap \{Z. \tau\ Z \subseteq Z\})$ 
<proof>
```

```
lemma gfp1: monotone  $\tau \longrightarrow (\text{gfp } \tau = \bigcup \{Z. Z \subseteq \tau\ Z\})$ 
<proof>
```

```
primrec power :: ['a  $\Rightarrow$  'a, nat]  $\Rightarrow$  ('a  $\Rightarrow$  'a) ((-  $\wedge$  -) 40)
where
power-zero: (f  $\wedge$  0) = ( $\lambda x. x$ ) |
power-suc: (f  $\wedge$  (Suc n)) = (f o (f  $\wedge$  n))
```

```
lemma predtrans-empty:
  assumes monotone  $\tau$ 
  shows  $\forall i. (\tau \wedge i) (\{\}) \subseteq (\tau \wedge (i + 1))(\{\})$ 
<proof>
```

```
lemma ex-card: finite S  $\Longrightarrow \exists n::nat. \text{card } S = n$ 
<proof>
```

```
lemma less-not-le:  $\llbracket (x::nat) < y; y \leq x \rrbracket \Longrightarrow \text{False}$ 
<proof>
```

```
lemma infchain-outruns-all:
  assumes finite (UNIV :: 'a set)
  and  $\forall i::nat. (\tau \wedge i) (\{\})::'a \text{ set} \subset (\tau \wedge i + (1::nat)) (\{\})$ 
  shows  $\forall j::nat. \exists i::nat. j < \text{card } ((\tau \wedge i) (\{\}))$ 
```

<proof>

lemma *no-infinite-subset-chain*:

assumes *finite* (*UNIV* :: 'a set)
and *monotone* ($\tau :: ('a \text{ set} \Rightarrow 'a \text{ set})$)
and $\forall i :: \text{nat. } ((\tau :: 'a \text{ set} \Rightarrow 'a \text{ set}) \wedge i) \{\} \subset (\tau \wedge i + (1 :: \text{nat})) (\{\} :: 'a \text{ set})$
shows *False*

Proof idea: Since *UNIV* is finite, we have from *ex_card* that there is an *n* with $\text{card } UNIV = n$. Now, use *infchain_outruns_all* to show as contradiction point that $\exists i. \text{card } UNIV < \text{card } ((\tau \wedge i) \{\})$. Since all sets are subsets of *UNIV*, we also have $\text{card } ((\tau \wedge i) \{\}) \leq \text{card } UNIV$: Contradiction!, i.e. proof of *False*

<proof>

lemma *finite-fixp*:

assumes *finite*(*UNIV* :: 'a set)
and *monotone* ($\tau :: ('a \text{ set} \Rightarrow 'a \text{ set})$)
shows $\exists i. (\tau \wedge i) (\{\}) = (\tau \wedge (i + 1))(\{\})$

Proof idea: with *predtrans-empty* we know $\forall i. (\tau \wedge i) \{\} \subseteq (\tau \wedge i + 1) \{\}$ (1). If we can additionally show $\exists i. (\tau \wedge i + 1) \{\} \subseteq (\tau \wedge i) \{\}$ (2), we can get the goal together with *equalityI* $\subseteq + \supseteq \longrightarrow =$. To prove (1) we observe that $(\tau \wedge i + 1) \{\} \subseteq (\tau \wedge i) \{\}$ can be inferred from $\neg (\tau \wedge i) \{\} \subseteq (\tau \wedge i + 1) \{\}$ and (1). Finally, the latter is solved directly by *no_infinite_subset_chain*.

<proof>

lemma *predtrans-UNIV*:

assumes *monotone* τ
shows $\forall i. (\tau \wedge i) (UNIV) \supseteq (\tau \wedge (i + 1))(UNIV)$

<proof>

lemma *Suc-less-le*: $x < (y - n) \implies x \leq (y - (\text{Suc } n))$

<proof>

lemma *card-univ-subtract*:

assumes *finite* (*UNIV* :: 'a set) **and** *monotone* ($\tau :: 'a \text{ set} \Rightarrow 'a \text{ set}$)
and $(\forall i :: \text{nat. } ((\tau :: 'a \text{ set} \Rightarrow 'a \text{ set}) \wedge i + (1 :: \text{nat})) (UNIV :: 'a \text{ set}) \subset (\tau \wedge i) UNIV)$
shows $(\forall i :: \text{nat. } \text{card}((\tau \wedge i) (UNIV :: 'a \text{ set})) \leq (\text{card } (UNIV :: 'a \text{ set})) - i)$
<proof>

lemma *card-UNIV-tau-i-below-zero*:

assumes *finite* (*UNIV* :: 'a set) **and** *monotone* ($\tau :: 'a \text{ set} \Rightarrow 'a \text{ set}$)
and $(\forall i :: \text{nat. } ((\tau :: 'a \text{ set} \Rightarrow 'a \text{ set}) \wedge i + (1 :: \text{nat})) (UNIV :: 'a \text{ set}) \subset (\tau \wedge i) UNIV)$

shows $\text{card}((\tau \wedge (\text{card } (UNIV :: 'a \text{ set}))) (UNIV :: 'a \text{ set})) \leq 0$
 $\langle \text{proof} \rangle$

lemma *finite-card-zero-empty*: $\llbracket \text{finite } S; \text{card } S \leq 0 \rrbracket \implies S = \{\}$
 $\langle \text{proof} \rangle$

lemma *UNIV-tau-i-is-empty*:
assumes *finite* $(UNIV :: 'a \text{ set})$ **and** *monotone* $(\tau :: 'a \text{ set} \Rightarrow 'a \text{ set})$
and $(\forall i :: \text{nat}. ((\tau :: 'a \text{ set} \Rightarrow 'a \text{ set}) \wedge i + (1 :: \text{nat})) (UNIV :: 'a \text{ set}) \subset (\tau \wedge i) UNIV)$
shows $(\tau \wedge (\text{card } (UNIV :: 'a \text{ set}))) (UNIV :: 'a \text{ set}) = \{\}$
 $\langle \text{proof} \rangle$

lemma *down-chain-reaches-empty*:
assumes *finite* $(UNIV :: 'a \text{ set})$ **and** *monotone* $(\tau :: 'a \text{ set} \Rightarrow 'a \text{ set})$
and $(\forall i :: \text{nat}. ((\tau :: 'a \text{ set} \Rightarrow 'a \text{ set}) \wedge i + (1 :: \text{nat})) UNIV \subset (\tau \wedge i) UNIV)$
shows $\exists (j :: \text{nat}). (\tau \wedge j) UNIV = \{\}$
 $\langle \text{proof} \rangle$

lemma *no-infinite-subset-chain2*:
assumes *finite* $(UNIV :: 'a \text{ set})$ **and** *monotone* $(\tau :: ('a \text{ set} \Rightarrow 'a \text{ set}))$
and $\forall i :: \text{nat}. (\tau \wedge i) UNIV \supset (\tau \wedge i + (1 :: \text{nat})) UNIV$
shows *False*
 $\langle \text{proof} \rangle$

lemma *finite-fixp2*:
assumes *finite* $(UNIV :: 'a \text{ set})$ **and** *monotone* $(\tau :: ('a \text{ set} \Rightarrow 'a \text{ set}))$
shows $\exists i. (\tau \wedge i) UNIV = (\tau \wedge (i + 1)) UNIV$
 $\langle \text{proof} \rangle$

lemma *mono-monotone*: *mono* $(\tau :: ('a \text{ set} \Rightarrow 'a \text{ set})) \implies \text{monotone } \tau$
 $\langle \text{proof} \rangle$

lemma *monotone-mono*: *monotone* $(\tau :: ('a \text{ set} \Rightarrow 'a \text{ set})) \implies \text{mono } \tau$
 $\langle \text{proof} \rangle$

lemma *power-power*: $((\tau :: ('a \text{ set} \Rightarrow 'a \text{ set})) \wedge \wedge n) = ((\tau :: ('a \text{ set} \Rightarrow 'a \text{ set})) \wedge n)$
 $\langle \text{proof} \rangle$

lemma *lfp-Kleene-iter-set*: *monotone* $(f :: ('a \text{ set} \Rightarrow 'a \text{ set})) \implies$
 $(f \wedge \text{Suc}(n)) \{\} = (f \wedge n) \{\} \implies \text{lfp } f = (f \wedge n) \{\}$
 $\langle \text{proof} \rangle$

lemma *lfp-loop*:
assumes *finite* $(UNIV :: 'b \text{ set})$ **and** *monotone* $(\tau :: ('b \text{ set} \Rightarrow 'b \text{ set}))$
shows $\exists n. \text{lfp } \tau = (\tau \wedge n) \{\}$
 $\langle \text{proof} \rangle$

These next two are produced as duals from the corresponding theorems in

HOL/ZF/Nat.thy. Would make sense to have them in the HOL/Library.

lemma *Kleene-iter-gfp*:

assumes *mono f* **and** $p \leq f p$ **shows** $p \leq (f^\wedge k)$ (*top::'a::order-top*)
 $\langle proof \rangle$

lemma *gfp-Kleene-iter*: **assumes** *mono f* **and** $(f^\wedge \text{Suc } k) \text{ top} = (f^\wedge k) \text{ top}$
shows $\text{gfp } f = (f^\wedge k) \text{ top}$
 $\langle proof \rangle$

lemma *gfp-Kleene-iter-set*:

assumes *monotone* ($f :: ('a \text{ set} \Rightarrow 'a \text{ set})$)
and $(f^\wedge \text{Suc}(n)) \text{ UNIV} = (f^\wedge n) \text{ UNIV}$
shows $\text{gfp } f = (f^\wedge n) \text{ UNIV}$
 $\langle proof \rangle$

lemma *gfp-loop*:

assumes *finite* ($\text{UNIV} :: 'b \text{ set}$)
and *monotone* ($\tau :: ('b \text{ set} \Rightarrow 'b \text{ set})$)
shows $\exists n. \text{gfp } \tau = (\tau^\wedge n)(\text{UNIV} :: 'b \text{ set})$
 $\langle proof \rangle$

1.2 Generic type of state with state transition and CTL Operators

The system states and their transition relation are defined as a class called *state* containing an abstract constant state transition. It introduces the syntactic infix notation $I \rightarrow_i I'$ to denote that system state *I* and *I'* are in this relation over an arbitrary (polymorphic) type 'a.

class *state* =

fixes *state-transition* :: $['a :: \text{type}, 'a] \Rightarrow \text{bool}$ $((- \rightarrow_i -) 50)$

The above class definition lifts Kripke structures and CTL to a general level. The definition of the inductive relation is given by a set of specific rules which are, however, part of an application like infrastructures. Branching time temporal logic CTL is defined in general over Kripke structures with arbitrary state transitions and can later be applied to suitable theories, like infrastructures. Based on the generic state transition \rightarrow of the type class *state*, the CTL-operators EX and AX express that property *f* holds in some or all next states, respectively.

definition *AX* **where** $\text{AX } f \equiv \{s. \{f0. s \rightarrow_i f0\} \subseteq f\}$

definition *EX'* **where** $\text{EX}' f \equiv \{s. \exists f0 \in f. s \rightarrow_i f0\}$

The CTL formula AG *f* means that on all paths branching from a state *s* the formula *f* is always true (*G* stands for 'globally'). It can be defined using the Tarski fixpoint theory by applying the greatest fixpoint operator. In a similar way, the other CTL operators are defined.

definition AF **where** $AF\ f \equiv lfp\ (\lambda\ Z.\ f \cup AX\ Z)$
definition EF **where** $EF\ f \equiv lfp\ (\lambda\ Z.\ f \cup EX'\ Z)$
definition AG **where** $AG\ f \equiv gfp\ (\lambda\ Z.\ f \cap AX\ Z)$
definition EG **where** $EG\ f \equiv gfp\ (\lambda\ Z.\ f \cap EX'\ Z)$
definition AU **where** $AU\ f1\ f2 \equiv lfp(\lambda\ Z.\ f2 \cup (f1 \cap AX\ Z))$
definition EU **where** $EU\ f1\ f2 \equiv lfp(\lambda\ Z.\ f2 \cup (f1 \cap EX'\ Z))$
definition AR **where** $AR\ f1\ f2 \equiv gfp(\lambda\ Z.\ f2 \cap (f1 \cup AX\ Z))$
definition ER **where** $ER\ f1\ f2 \equiv gfp(\lambda\ Z.\ f2 \cap (f1 \cup EX'\ Z))$

1.3 Kripke structure and Modelchecking

datatype $'a\ kripke =$
 $\quad Kripke\ 'a\ set\ 'a\ set$

primrec $states$ **where** $states\ (Kripke\ S\ I) = S$
primrec $init$ **where** $init\ (Kripke\ S\ I) = I$

The formal Isabelle definition of what it means that formula f holds in a Kripke structure M can be stated as: the initial states of the Kripke structure $init\ M$ need to be contained in the set of all states $states\ M$ that imply f .

definition $check\ (- \vdash -\ 50)$
where $M \vdash f \equiv (init\ M) \subseteq \{s \in (states\ M). s \in f\}$

definition $state-transition-refl\ ((- \rightarrow_i^* -)\ 50)$
where $s \rightarrow_i^* s' \equiv ((s, s') \in \{(x, y). state-transition\ x\ y\}^*)$

1.4 Lemmas for CTL operators

1.4.1 EF lemmas

lemma $EF\text{-}lem0$: $(x \in EF\ f) = (x \in f \cup EX'\ (lfp\ (\lambda Z :: ('a :: state)\ set.\ f \cup EX'\ Z)))$
 $\langle proof \rangle$

lemma $EF\text{-}lem00$: $(EF\ f) = (f \cup EX'\ (lfp\ (\lambda Z :: ('a :: state)\ set.\ f \cup EX'\ Z)))$
 $\langle proof \rangle$

lemma $EF\text{-}lem000$: $(EF\ f) = (f \cup EX'\ (EF\ f))$
 $\langle proof \rangle$

lemma $EF\text{-}lem1$: $x \in f \vee x \in (EX'\ (EF\ f)) \implies x \in EF\ f$
 $\langle proof \rangle$

lemma $EF\text{-}lem2b$:
assumes $x \in (EX'\ (EF\ f))$
shows $x \in EF\ f$
 $\langle proof \rangle$

lemma $EF\text{-}lem2a$: **assumes** $x \in f$ **shows** $x \in EF\ f$

$\langle proof \rangle$

lemma *EF-lem2c*: **assumes** $x \notin f$ **shows** $x \in EF \ (-f)$
 $\langle proof \rangle$

lemma *EF-lem2d*: **assumes** $x \notin EF \ f$ **shows** $x \notin f$
 $\langle proof \rangle$

lemma *EF-lem3b*: **assumes** $x \in EX' \ (f \cup EX' \ (EF \ f))$ **shows** $x \in (EF \ f)$
 $\langle proof \rangle$

lemma *EX-lem0l*: $x \in (EX' \ f) \implies x \in (EX' \ (f \cup g))$
 $\langle proof \rangle$

lemma *EX-lem0r*: $x \in (EX' \ g) \implies x \in (EX' \ (f \cup g))$
 $\langle proof \rangle$

lemma *EX-step*: **assumes** $x \rightarrow_i y$ **and** $y \in f$ **shows** $x \in EX' \ f$
 $\langle proof \rangle$

lemma *EF-E[rule-format]*: $\forall f. x \in (EF \ (f :: ('a :: state) \ set)) \longrightarrow x \in (f \cup EX' \ (EF \ f))$
 $\langle proof \rangle$

lemma *EF-step*: **assumes** $x \rightarrow_i y$ **and** $y \in f$ **shows** $x \in EF \ f$
 $\langle proof \rangle$

lemma *EF-step-step*: **assumes** $x \rightarrow_i y$ **and** $y \in EF \ f$ **shows** $x \in EF \ f$
 $\langle proof \rangle$

lemma *EF-step-star*: $\llbracket x \rightarrow_i^* y; y \in f \rrbracket \implies x \in EF \ f$
 $\langle proof \rangle$

lemma *EF-induct-prep*:
 assumes $(a :: 'a :: state) \in \text{lf}p \ (\lambda Z. (f :: 'a :: state \ set) \cup EX' \ Z)$
 and $\text{mono} \ (\lambda Z. (f :: 'a :: state \ set) \cup EX' \ Z)$
 shows $(\bigwedge x :: 'a :: state.$
 $x \in ((\lambda Z. (f :: 'a :: state \ set) \cup EX' \ Z)(\text{lf}p \ (\lambda Z. (f :: 'a :: state \ set) \cup EX' \ Z) \cap$
 $\{x :: 'a :: state. (P :: 'a :: state \Rightarrow \text{bool}) \ x\})) \implies P \ x \implies$
 $P \ a$
 $\langle proof \rangle$

lemma *EF-induct*: $(a :: 'a :: state) \in EF \ (f :: 'a :: state \ set) \implies$
 $\text{mono} \ (\lambda Z. (f :: 'a :: state \ set) \cup EX' \ Z) \implies$
 $(\bigwedge x :: 'a :: state.$
 $x \in ((\lambda Z. (f :: 'a :: state \ set) \cup EX' \ Z)(EF \ f \cap \{x :: 'a :: state. (P :: 'a :: state \Rightarrow$
 $\text{bool}) \ x\})) \implies P \ x) \implies$
 $P \ a$
 $\langle proof \rangle$

lemma *valEF-E*: $M \vdash EF\ f \implies x \in \text{init } M \implies x \in EF\ f$
 $\langle \text{proof} \rangle$

lemma *EF-step-star-rev*[*rule-format*]: $x \in EF\ s \implies (\exists\ y \in s. x \rightarrow_i^* y)$
 $\langle \text{proof} \rangle$

lemma *EF-step-inv*: $(I \subseteq \{sa :: 's :: \text{state}. (\exists\ i :: 's \in I. i \rightarrow_i^* sa) \wedge sa \in EF\ s\})$
 $\implies \forall\ x \in I. \exists\ y \in s. x \rightarrow_i^* y$
 $\langle \text{proof} \rangle$

1.4.2 AG lemmas

lemma *AG-in-lem*: $x \in AG\ s \implies x \in s$
 $\langle \text{proof} \rangle$

lemma *AG-lem1*: $x \in s \wedge x \in (AX\ (AG\ s)) \implies x \in AG\ s$
 $\langle \text{proof} \rangle$

lemma *AG-lem2*: $x \in AG\ s \implies x \in (s \cap (AX\ (AG\ s)))$
 $\langle \text{proof} \rangle$

lemma *AG-lem3*: $AG\ s = (s \cap (AX\ (AG\ s)))$
 $\langle \text{proof} \rangle$

lemma *AG-step*: $y \rightarrow_i z \implies y \in AG\ s \implies z \in AG\ s$
 $\langle \text{proof} \rangle$

lemma *AG-all-s*: $x \rightarrow_i^* y \implies x \in AG\ s \implies y \in AG\ s$
 $\langle \text{proof} \rangle$

lemma *AG-imp-notnotEF*:
 $I \neq \{\} \implies ((\text{Kripke } \{s :: ('s :: \text{state}). \exists\ i \in I. (i \rightarrow_i^* s)\} (I :: ('s :: \text{state})\text{set})$
 $\vdash AG\ s)) \implies$
 $(\neg(\text{Kripke } \{s :: ('s :: \text{state}). \exists\ i \in I. (i \rightarrow_i^* s)\} (I :: ('s :: \text{state})\text{set}) \vdash EF\ (-$
 $s)))$
 $\langle \text{proof} \rangle$

lemma *check2-def*: $(\text{Kripke } S\ I \vdash f) = (I \subseteq S \cap f)$
 $\langle \text{proof} \rangle$

end

2 Insider Framework

theory *AirInsider*
imports *MC*
begin
datatype *action* = *get* | *move* | *eval* | *put*

We use an abstract type declaration actor that can later be instantiated by a more concrete type.

```
typedecl actor
consts Actor :: string  $\Rightarrow$  actor
```

Alternatives to the type declaration do not work.

context fixes Abs Rep actor assumes td: "type_definition Abs Rep actor"
begin definition Actor where "Actor = Abs" ...doesn't work for replacing
the actor typedecl because in "type_definition" above the "actor" is a set
not a type! So can't be used for our purposes. Trying a locale instead for
polymorphic type Actor locale ACT = fixes Actor :: "string =_i 'actor" be-
gin ... That is a nice idea and works quite far but clashes with the generic
state_transition later (it's not possible to instantiate within a locale and out-
side it we cannot instantiate "a infrastructure" to state (clearly an abstract
thing as an instance is strange)

```
type-synonym identity = string
type-synonym policy = ((actor  $\Rightarrow$  bool) * action set)
```

```
definition ID :: [actor, string]  $\Rightarrow$  bool
where ID a s  $\equiv$  (a = Actor s)
```

```
datatype location = Location nat
```

```
datatype igragh = Lgragh (location * location)set location  $\Rightarrow$  identity list
          actor  $\Rightarrow$  (string list * string list) location  $\Rightarrow$  string list
```

```
datatype infrastructure =
  Infrastructure igragh
  [igragh, location]  $\Rightarrow$  policy set
```

```
primrec loc :: location  $\Rightarrow$  nat
where loc(Location n) = n
primrec gra :: igragh  $\Rightarrow$  (location * location)set
where gra(Lgragh g a c l) = g
primrec agra :: igragh  $\Rightarrow$  (location  $\Rightarrow$  identity list)
where agra(Lgragh g a c l) = a
primrec cgra :: igragh  $\Rightarrow$  (actor  $\Rightarrow$  string list * string list)
where cgra(Lgragh g a c l) = c
primrec lgra :: igragh  $\Rightarrow$  (location  $\Rightarrow$  string list)
where lgra(Lgragh g a c l) = l
```

```
definition nodes :: igragh  $\Rightarrow$  location set
where nodes g == { x. (? y. ((x,y): gra g) | ((y,x): gra g)) }
```

```
definition actors-graph :: igragh  $\Rightarrow$  identity set
where actors-graph g == { x. ? y. y : nodes g  $\wedge$  x  $\in$  set(agra g y) }
```

```
primrec graphI :: infrastructure  $\Rightarrow$  igragh
```

where *graphI* (*Infrastructure g d*) = *g*
primrec *delta* :: [*infrastructure, igraph, location*] \Rightarrow *policy set*
where *delta* (*Infrastructure g d*) = *d*
primrec *tspace* :: [*infrastructure, actor*] \Rightarrow *string list * string list*
where *tspace* (*Infrastructure g d*) = *cgra g*
primrec *lspace* :: [*infrastructure, location*] \Rightarrow *string list*
where *lspace* (*Infrastructure g d*) = *lgra g*

definition *credentials* :: *string list * string list* \Rightarrow *string set*
where *credentials lxl* \equiv *set (fst lxl)*
definition *has* :: [*igraph, actor * string*] \Rightarrow *bool*
where *has G ac* \equiv *snd ac* \in *credentials(cgra G (fst ac))*
definition *roles* :: *string list * string list* \Rightarrow *string set*
where *roles lxl* \equiv *set (snd lxl)*
definition *role* :: [*igraph, actor * string*] \Rightarrow *bool*
where *role G ac* \equiv *snd ac* \in *roles(cgra G (fst ac))*

definition *isin* :: [*igraph, location, string*] \Rightarrow *bool*
where *isin G l s* \equiv *s* \in *set(lgra G l)*

datatype *psy-states* = *happy* | *depressed* | *disgruntled* | *angry* | *stressed*
datatype *motivations* = *financial* | *political* | *revenge* | *curious* | *competitive-advantage*
| *power* | *peer-recognition*

datatype *actor-state* = *Actor-state psy-states motivations set*
primrec *motivation* :: *actor-state* \Rightarrow *motivations set*
where *motivation (Actor-state p m)* = *m*
primrec *psy-state* :: *actor-state* \Rightarrow *psy-states*
where *psy-state (Actor-state p m)* = *p*

definition *tipping-point* :: *actor-state* \Rightarrow *bool* **where**
tipping-point a \equiv (*(motivation a* \neq *{})*) \wedge (*happy* \neq *psy-state a*)

UasI and UasI' are the central predicates allowing to specify Insiders. They define which identities can be mapped to the same role by the Actor function. For all other identities, Actor is defined as injective on those identities.

definition *UasI* :: [*identity, identity*] \Rightarrow *bool*
where *UasI a b* \equiv (*Actor a* = *Actor b*) \wedge ($\forall x y. x \neq a \wedge y \neq a \wedge$ *Actor x* = *Actor y* \longrightarrow *x* = *y*)

definition *UasI'* :: [*actor => bool, identity, identity*] \Rightarrow *bool*
where *UasI' P a b* \equiv *P (Actor b)* \longrightarrow *P (Actor a)*

Two versions of Insider predicate corresponding to UasI and UasI'. Under the assumption that the tipping point has been reached for a person a then a can impersonate all b (take all of b's "roles") where the b's are specified by a given set of identities

definition *Insider* :: [*identity, identity set, identity* \Rightarrow *actor-state*] \Rightarrow *bool*

where $Insider\ a\ C\ as \equiv (tipping-point\ (as\ a) \longrightarrow (\forall\ b \in C. UasI\ a\ b))$

definition $Insider' :: [actor \Rightarrow bool, identity, identity\ set, identity \Rightarrow actor\ state] \Rightarrow bool$

where $Insider'\ P\ a\ C\ as \equiv (tipping-point\ (as\ a) \longrightarrow (\forall\ b \in C. UasI'\ P\ a\ b \wedge inj-on\ Actor\ C))$

definition $atI :: [identity, igraph, location] \Rightarrow bool\ (-\ @_{(-)}\ -\ 50)$

where $a\ @_G\ l \equiv a \in set(agra\ G\ l)$

enables is the central definition of the behaviour as given by a policy that specifies what actions are allowed in a certain location for what actors

definition $enables :: [infrastructure, location, actor, action] \Rightarrow bool$

where

$enables\ I\ l\ a\ a' \equiv (\exists\ (p,e) \in delta\ I\ (graphI\ I)\ l. a' \in e \wedge p\ a)$

behaviour is the good behaviour, i.e. everything allowed by policy

definition $behaviour :: infrastructure \Rightarrow (location * actor * action) set$

where $behaviour\ I \equiv \{(t,a,a').\ enables\ I\ t\ a\ a'\}$

misbehaviour is the complement of behaviour

definition $misbehaviour :: infrastructure \Rightarrow (location * actor * action) set$

where $misbehaviour\ I \equiv \neg(behaviour\ I)$

basic lemmas for enable

lemma $not-enableI: (\forall\ (p,e) \in delta\ I\ (graphI\ I)\ l. (\sim(h : e) \mid (\sim(p(a)))) \implies \sim(enables\ I\ l\ a\ h)$

$\langle proof \rangle$

lemma $not-enableI2: [\bigwedge\ p\ e. (p,e) \in delta\ I\ (graphI\ I)\ l \implies (\sim(t : e) \mid (\sim(p(a))))] \implies \sim(enables\ I\ l\ a\ t)$

$\langle proof \rangle$

lemma $not-enableE: [\sim(enables\ I\ l\ a\ t); (p,e) \in delta\ I\ (graphI\ I)\ l] \implies (\sim(t : e) \mid (\sim(p(a))))$

$\langle proof \rangle$

lemma $not-enableE2: [\sim(enables\ I\ l\ a\ t); (p,e) \in delta\ I\ (graphI\ I)\ l; t : e] \implies (\sim(p(a)))$

$\langle proof \rangle$

some constructions to deal with lists of actors in locations for the semantics of action move

primrec $del :: ['a, 'a\ list] \Rightarrow 'a\ list$

where

$del-nil: del\ a\ [] = []$

$del-cons: del\ a\ (x\#\!ls) = (if\ x = a\ then\ ls\ else\ x\ \#\! (del\ a\ ls))$

primrec *jonce* :: [*'a*, *'a list*] \Rightarrow *bool*

where

jonce-nil: *jonce* *a* [] = *False* |

jonce-cons: *jonce* *a* (*x* # *ls*) = (if *x* = *a* then (*a* \notin (set *ls*)) else *jonce* *a* *ls*)

primrec *nodup* :: [*'a*, *'a list*] \Rightarrow *bool*

where

nodup-nil: *nodup* *a* [] = *True* |

nodup-step: *nodup* *a* (*x* # *ls*) = (if *x* = *a* then (*a* \notin (set *ls*)) else *nodup* *a* *ls*)

definition *move-graph-a* :: [*identity*, *location*, *location*, *igraph*] \Rightarrow *igraph*

where *move-graph-a* *n l l' g* \equiv *Lgraph* (*gra g*)

(if *n* \in set ((*agra g*) *l*) & *n* \notin set ((*agra g*) *l'*) then

((*agra g*)(*l* := del *n* (*agra g l*)))(*l'* := (*n* # (*agra g l'*)))

else (*agra g*))(*cgra g*)(*lgra g*)

State transition relation over infrastructures (the states) defining the semantics of actions in systems with humans and potentially insiders *)

inductive *state-transition-in* :: [*infrastructure*, *infrastructure*] \Rightarrow *bool* ((- \rightarrow_n -) 50)

where

move: [*G* = *graphI I*; *a* @_{*G*} *l*; *l* \in nodes *G*; *l'* \in nodes *G*;

(*a*) \in actors-graph(*graphI I*); enables *I l'* (Actor *a*) *move*;

I' = *Infrastructure* (*move-graph-a a l l' (graphI I)*)(*delta I*)] \Longrightarrow *I* \rightarrow_n *I'*

| *get* : [*G* = *graphI I*; *a* @_{*G*} *l*; *a'* @_{*G*} *l*; has *G* (Actor *a*, *z*);

enables *I l* (Actor *a*) *get*;

I' = *Infrastructure*

(*Lgraph* (*gra G*)(*agra G*)

((*cgra G*)(Actor *a'* :=

(*z* # (fst(*cgra G* (Actor *a'*))), snd(*cgra G* (Actor *a'*))))

(*lgra G*))

(*delta I*)]

\Longrightarrow *I* \rightarrow_n *I'*

| *put* : [*G* = *graphI I*; *a* @_{*G*} *l*; enables *I l* (Actor *a*) *put*;

I' = *Infrastructure*

(*Lgraph* (*gra G*)(*agra G*)(*cgra G*)

((*lgra G*)(*l* := [*z*]))

(*delta I*)]

\Longrightarrow *I* \rightarrow_n *I'*

| *put-remote* : [*G* = *graphI I*; enables *I l* (Actor *a*) *put*;

I' = *Infrastructure*

(*Lgraph* (*gra G*)(*agra G*)(*cgra G*)

((*lgra G*)(*l* := [*z*]))

(*delta I*)]

\Longrightarrow *I* \rightarrow_n *I'*

show that this infrastructure is a state as given in MC.thy

instantiation *infrastructure* :: *state*

begin

definition

state-transition-infra-def: $(i \rightarrow_i i') = (i \rightarrow_n (i' :: \text{infrastructure}))$

instance

$\langle \text{proof} \rangle$

definition *state-transition-in-refl* $((- \rightarrow_n^* -) \ 50)$

where $s \rightarrow_n^* s' \equiv ((s, s') \in \{(x, y). \text{state-transition-in } x \ y\}^*)$

lemma *del-del*[*rule-format*]: $n \in \text{set } (\text{del } a \ S) \longrightarrow n \in \text{set } S$

$\langle \text{proof} \rangle$

lemma *del-dec*[*rule-format*]: $a \in \text{set } S \longrightarrow \text{length } (\text{del } a \ S) < \text{length } S$

$\langle \text{proof} \rangle$

lemma *del-sort*[*rule-format*]: $\forall \ n. (\text{Suc } n :: \text{nat}) \leq \text{length } (l) \longrightarrow n \leq \text{length } (\text{del } a \ (l))$

$\langle \text{proof} \rangle$

lemma *del-jonce*: $\text{jonce } a \ l \longrightarrow a \notin \text{set } (\text{del } a \ l)$

$\langle \text{proof} \rangle$

lemma *del-nodup*[*rule-format*]: $\text{nodup } a \ l \longrightarrow a \notin \text{set } (\text{del } a \ l)$

$\langle \text{proof} \rangle$

lemma *nodup-up*[*rule-format*]: $a \in \text{set } (\text{del } a \ l) \longrightarrow a \in \text{set } l$

$\langle \text{proof} \rangle$

lemma *del-up* [*rule-format*]: $a \in \text{set } (\text{del } aa \ l) \longrightarrow a \in \text{set } l$

$\langle \text{proof} \rangle$

lemma *nodup-notin*[*rule-format*]: $a \notin \text{set } \text{list} \longrightarrow \text{nodup } a \ \text{list}$

$\langle \text{proof} \rangle$

lemma *nodup-down*[*rule-format*]: $\text{nodup } a \ l \longrightarrow \text{nodup } a \ (\text{del } a \ l)$

$\langle \text{proof} \rangle$

lemma *del-notin-down*[*rule-format*]: $a \notin \text{set } \text{list} \longrightarrow a \notin \text{set } (\text{del } aa \ \text{list})$

$\langle \text{proof} \rangle$

lemma *del-not-a*[*rule-format*]: $x \neq a \longrightarrow x \in \text{set } l \longrightarrow x \in \text{set } (\text{del } a \ l)$

$\langle \text{proof} \rangle$

lemma *nodup-down-notin*[*rule-format*]: $\text{nodup } a \ l \longrightarrow \text{nodup } a \ (\text{del } aa \ l)$

$\langle \text{proof} \rangle$

lemma *move-graph-eq*: $\text{move-graph-a } a \ l \ l \ g = g$

$\langle \text{proof} \rangle$

Some useful properties about the invariance of the nodes, the actors, and the policy with respect to the state transition

lemma *delta-invariant*: $\forall z z'. z \rightarrow_n z' \longrightarrow \text{delta}(z) = \text{delta}(z')$
 $\langle \text{proof} \rangle$

lemma *init-state-policy0*:
assumes $\forall z z'. z \rightarrow_n z' \longrightarrow \text{delta}(z) = \text{delta}(z')$
and $(x, y) \in \{(x::\text{infrastructure}, y::\text{infrastructure}). x \rightarrow_n y\}^*$
shows $\text{delta}(x) = \text{delta}(y)$
 $\langle \text{proof} \rangle$

lemma *init-state-policy*: $\llbracket (x, y) \in \{(x::\text{infrastructure}, y::\text{infrastructure}). x \rightarrow_n y\}^* \rrbracket \implies$
 $\text{delta}(x) = \text{delta}(y)$
 $\langle \text{proof} \rangle$

lemma *same-nodes0*[*rule-format*]: $\forall z z'. z \rightarrow_n z' \longrightarrow \text{nodes}(\text{graphI } z) = \text{nodes}(\text{graphI } z')$
 $\langle \text{proof} \rangle$

lemma *same-nodes*: $(I, y) \in \{(x::\text{infrastructure}, y::\text{infrastructure}). x \rightarrow_n y\}^*$
 $\implies \text{nodes}(\text{graphI } y) = \text{nodes}(\text{graphI } I)$
 $\langle \text{proof} \rangle$

lemma *same-actors0*[*rule-format*]: $\forall z z'. z \rightarrow_n z' \longrightarrow \text{actors-graph}(\text{graphI } z) = \text{actors-graph}(\text{graphI } z')$
 $\langle \text{proof} \rangle$

lemma *same-actors*: $(I, y) \in \{(x::\text{infrastructure}, y::\text{infrastructure}). x \rightarrow_n y\}^*$
 $\implies \text{actors-graph}(\text{graphI } I) = \text{actors-graph}(\text{graphI } y)$
 $\langle \text{proof} \rangle$

end
end

3 Airplane case study

theory *Airplane*
imports *AirInsider*
begin
datatype *doorstate* = *locked* | *norm* | *unlocked*
datatype *position* = *air* | *airport* | *ground*

locale *airplane* =

fixes *airplane-actors* :: *identity set*
defines *airplane-actors-def*: *airplane-actors* $\equiv \{\text{"Bob"}, \text{"Charly"}, \text{"Alice"}\}$

```

fixes airplane-locations :: location set
defines airplane-locations-def:
  airplane-locations  $\equiv$  {Location 0, Location 1, Location 2}

fixes cockpit :: location
defines cockpit-def: cockpit  $\equiv$  Location 2
fixes door :: location
defines door-def: door  $\equiv$  Location 1
fixes cabin :: location
defines cabin-def: cabin  $\equiv$  Location 0

fixes global-policy :: [infrastructure, identity]  $\Rightarrow$  bool
defines global-policy-def: global-policy I a  $\equiv$  a  $\notin$  airplane-actors
   $\longrightarrow \neg(\text{enables } I \text{ cockpit (Actor } a) \text{ put})$ 

fixes ex-creds :: actor  $\Rightarrow$  (string list * string list)
defines ex-creds-def: ex-creds  $\equiv$ 
  ( $\lambda$  x. (if x = Actor "Bob"
    then (["PIN"], ["pilot"])
    else (if x = Actor "Charly"
      then (["PIN"], ["copilot"])
      else (if x = Actor "Alice"
        then (["PIN"], ["flightattendant"])
        else ([], []))))))

fixes ex-locs :: location  $\Rightarrow$  string list
defines ex-locs-def: ex-locs  $\equiv$  ( $\lambda$  x. if x = door then ["norm"] else
  (if x = cockpit then ["air"] else []))

fixes ex-locs' :: location  $\Rightarrow$  string list
defines ex-locs'-def: ex-locs'  $\equiv$  ( $\lambda$  x. if x = door then ["locked"] else
  (if x = cockpit then ["air"] else []))

fixes ex-graph :: igragh
defines ex-graph-def: ex-graph  $\equiv$  Lgraph
  {(cockpit, door), (door, cabin)}
  ( $\lambda$  x. if x = cockpit then ["Bob", "Charly"]
    else (if x = door then []
      else (if x = cabin then ["Alice"] else [])))
  ex-creds ex-locs

fixes aid-graph :: igragh
defines aid-graph-def: aid-graph  $\equiv$  Lgraph
  {(cockpit, door), (door, cabin)}
  ( $\lambda$  x. if x = cockpit then ["Charly"]
    else (if x = door then []
      else (if x = cabin then ["Bob", "Alice"] else [])))
  ex-creds ex-locs'

fixes aid-graph0 :: igragh

```

defines *aid-graph0-def*: *aid-graph0* \equiv *Lgraph*

```
{(cockpit, door),(door,cabin)}
(λ x. if x = cockpit then ["Charly"]
    else (if x = door then ["Bob"]
        else (if x = cabin then ["Alice"] else [])))
ex-creds ex-locs
```

fixes *agid-graph* :: *igraph*

defines *agid-graph-def*: *agid-graph* \equiv *Lgraph*

```
{(cockpit, door),(door,cabin)}
(λ x. if x = cockpit then ["Charly"]
    else (if x = door then []
        else (if x = cabin then ["Bob", "Alice"] else [])))
ex-creds ex-locs
```

fixes *local-policies* :: [*igraph*, *location*] \Rightarrow *policy set*

defines *local-policies-def*: *local-policies* *G* \equiv

```
(λ y. if y = cockpit then
    {(λ x. (? n. (n @G cockpit) ∧ Actor n = x), {put}),
     (λ x. (? n. (n @G cabin) ∧ Actor n = x ∧ has G (x, "PIN")
        ∧ isin G door "norm"),{move})}
    }
  else (if y = door then {(λ x. True, {move}),
    (λ x. (? n. (n @G cockpit) ∧ Actor n = x), {put})}
    else (if y = cabin then {(λ x. True, {move})}
    else {})))
```

fixes *local-policies-four-eyes* :: [*igraph*, *location*] \Rightarrow *policy set*

defines *local-policies-four-eyes-def*: *local-policies-four-eyes* *G* \equiv

```
(λ y. if y = cockpit then
    {(λ x. (? n. (n @G cockpit) ∧ Actor n = x) ∧
        2 ≤ length(agra G y) ∧ (∀ h ∈ set(agra G y). h ∈ airplane-actors),
    {put}),
    (λ x. (? n. (n @G cabin) ∧ Actor n = x ∧ has G (x, "PIN") ∧
        isin G door "norm"),{move})}
    }
  else (if y = door then
    {(λ x. ((? n. (n @G cockpit) ∧ Actor n = x) ∧ 3 ≤ length(agra G
    cockpit)), {move})}
    else (if y = cabin then
    {(λ x. ((? n. (n @G door) ∧ Actor n = x)), {move})}
    else {})))
```

fixes *Airplane-scenario* :: *infrastructure* (**structure**)

defines *Airplane-scenario-def*:

Airplane-scenario \equiv *Infrastructure ex-graph local-policies*

fixes *Airplane-in-danger* :: *infrastructure*


```

defines Airplane-in-danger-def:
Airplane-in-danger  $\equiv$  Infrastructure aid-graph local-policies

fixes Airplane-getting-in-danger0 :: infrastructure
defines Airplane-getting-in-danger0-def:
Airplane-getting-in-danger0  $\equiv$  Infrastructure aid-graph0 local-policies

fixes Airplane-getting-in-danger :: infrastructure
defines Airplane-getting-in-danger-def:
Airplane-getting-in-danger  $\equiv$  Infrastructure agid-graph local-policies

fixes Air-states
defines Air-states-def: Air-states  $\equiv$  { I. Airplane-scenario  $\rightarrow_n^*$  I }

fixes Air-Kripke
defines Air-Kripke  $\equiv$  Kripke Air-states {Airplane-scenario}

fixes Airplane-not-in-danger :: infrastructure
defines Airplane-not-in-danger-def:
Airplane-not-in-danger  $\equiv$  Infrastructure aid-graph local-policies-four-eyes

fixes Airplane-not-in-danger-init :: infrastructure
defines Airplane-not-in-danger-init-def:
Airplane-not-in-danger-init  $\equiv$  Infrastructure ex-graph local-policies-four-eyes

fixes Air-tp-states
defines Air-tp-states-def: Air-tp-states  $\equiv$  { I. Airplane-not-in-danger-init  $\rightarrow_n^*$  I
}

fixes Air-tp-Kripke
defines Air-tp-Kripke  $\equiv$  Kripke Air-tp-states {Airplane-not-in-danger-init}

fixes Safety :: [infrastructure, identity]  $\Rightarrow$  bool
defines Safety-def: Safety I a  $\equiv$  a  $\in$  airplane-actors
 $\longrightarrow$  (enables I cockpit (Actor a) move)

fixes Security :: [infrastructure, identity]  $\Rightarrow$  bool
defines Security-def: Security I a  $\equiv$  (isin (graphI I) door "locked")
 $\longrightarrow \neg(\text{enables } I \text{ cockpit (Actor } a) \text{ move})$ 

fixes foe-control :: [location, action]  $\Rightarrow$  bool
defines foe-control-def: foe-control l c  $\equiv$ 
(! I :: infrastructure. (? x :: identity.
 $x @_{\text{graphI } I} l \wedge \text{Actor } x \neq \text{Actor "Eve"}$ )

```

$\longrightarrow \neg(\text{enables } I \text{ l } (\text{Actor } \text{"Eve"}) \text{ c}))$

fixes *astate*:: *identity* \Rightarrow *actor-state*

defines *astate-def*: *astate* $x \equiv$ (case x of
 "Eve" \Rightarrow *Actor-state depressed* {*revenge*, *peer-recognition*}
 | - \Rightarrow *Actor-state happy* {})

assumes *Eve-precipitating-event*: *tipping-point* (*astate "Eve"*)

assumes *Insider-Eve*: *Insider "Eve" {"Charly"} astate*

assumes *cockpit-foe-control*: *foe-control cockpit put*

begin

lemma *ex-inv*: *global-policy Airplane-scenario "Bob"*

<proof>

lemma *ex-inv2*: *global-policy Airplane-scenario "Charly"*

<proof>

lemma *ex-inv3*: \neg *global-policy Airplane-scenario "Eve"*

<proof>

show *Safety for Airplane_scenario*

lemma *Safety*: *Safety Airplane-scenario ("Alice")*

<proof>

show *Security for Airplane_scenario*

lemma *inj-lem*: $\llbracket \text{inj } f; x \neq y \rrbracket \Longrightarrow f x \neq f y$

<proof>

lemma *inj-on-lem*: $\llbracket \text{inj-on } f A; x \neq y; x \in A; y \in A \rrbracket \Longrightarrow f x \neq f y$

<proof>

lemma *inj-lemma'*: *inj-on (isin ex-graph door) {"locked","norm"}*

<proof>

lemma *inj-lemma''*: *inj-on (isin aid-graph door) {"locked","norm"}*

<proof>

lemma *locl-lemma2*: *isin ex-graph door "norm" \neq isin ex-graph door "locked"*

<proof>

lemma *locl-lemma3*: *isin ex-graph door "norm" = (\neg isin ex-graph door "locked")*

<proof>

lemma *locl-lemma2a*: *isin aid-graph door "norm" \neq isin aid-graph door "locked"*

<proof>

lemma *locl-lemma3a*: *isin aid-graph door "norm" = (\neg isin aid-graph door "locked")*

$\langle proof \rangle$

lemma *Security: Security Airplane-scenario s*
 $\langle proof \rangle$

show that pilot can't get into cockpit if outside and locked = Airplane_in_danger

lemma *Security-problem: Security Airplane-scenario "Bob"*
 $\langle proof \rangle$

show that pilot can get out of cockpit

lemma *pilot-can-leave-cockpit: (enables Airplane-scenario cabin (Actor "Bob") move)*
 $\langle proof \rangle$

show that in Airplane_in_danger copilot can still do put = put position to ground

lemma *ex-inv4: \neg global-policy Airplane-in-danger ("Eve")*
 $\langle proof \rangle$

lemma *Safety-in-danger:*
 fixes s
 assumes $s \in \text{airplane-actors}$
 shows $\neg(\text{Safety Airplane-in-danger } s)$
 $\langle proof \rangle$

lemma *Security-problem': $\neg(\text{enables Airplane-in-danger cockpit (Actor "Bob") move})$*
 $\langle proof \rangle$

show that with the four eyes rule in Airplane_not_in_danger Eve cannot crash plane, i.e. cannot put position to ground

lemma *ex-inv5: $a \in \text{airplane-actors} \longrightarrow \text{global-policy Airplane-not-in-danger } a$*
 $\langle proof \rangle$

lemma *ex-inv6: global-policy Airplane-not-in-danger a*
 $\langle proof \rangle$

lemma *step0: Airplane-scenario \rightarrow_n Airplane-getting-in-danger0*
 $\langle proof \rangle$

lemma *step1: Airplane-getting-in-danger0 \rightarrow_n Airplane-getting-in-danger*
 $\langle proof \rangle$

lemma *step2: Airplane-getting-in-danger \rightarrow_n Airplane-in-danger*
 $\langle proof \rangle$

lemma *step0r: Airplane-scenario \rightarrow_n^* Airplane-getting-in-danger0*
 $\langle proof \rangle$

lemma *step1r: Airplane-getting-in-danger0* \rightarrow_n^* *Airplane-getting-in-danger*
 $\langle \text{proof} \rangle$

lemma *step2r: Airplane-getting-in-danger* \rightarrow_n^* *Airplane-in-danger*
 $\langle \text{proof} \rangle$

theorem *step-allr: Airplane-scenario* \rightarrow_n^* *Airplane-in-danger*
 $\langle \text{proof} \rangle$

theorem *aid-attack: Air-Kripke* $\vdash EF (\{x. \neg \text{global-policy } x \text{ "Eve"}\})$
 $\langle \text{proof} \rangle$

Invariant: actors cannot be at two places at the same time

lemma *actors-unique-loc-base:*

assumes $I \rightarrow_n I'$

and $(\forall l l'. a @_{\text{graph} I} l \wedge a @_{\text{graph} I} l' \longrightarrow l = l') \wedge$
 $(\forall l. \text{nodup } a (\text{agra } (\text{graph} I) l))$

shows $(\forall l l'. a @_{\text{graph} I'} l \wedge a @_{\text{graph} I'} l' \longrightarrow l = l') \wedge$
 $(\forall l. \text{nodup } a (\text{agra } (\text{graph} I') l))$

$\langle \text{proof} \rangle$

lemma *actors-unique-loc-step:*

assumes $(I, I') \in \{(x::\text{infrastructure}, y::\text{infrastructure}). x \rightarrow_n y\}^*$

and $\forall a. (\forall l l'. a @_{\text{graph} I} l \wedge a @_{\text{graph} I} l' \longrightarrow l = l') \wedge$
 $(\forall l. \text{nodup } a (\text{agra } (\text{graph} I) l))$

shows $\forall a. (\forall l l'. a @_{\text{graph} I'} l \wedge a @_{\text{graph} I'} l' \longrightarrow l = l') \wedge$
 $(\forall l. \text{nodup } a (\text{agra } (\text{graph} I') l))$

$\langle \text{proof} \rangle$

lemma *actors-unique-loc-aid-base:*

$\forall a. (\forall l l'. a @_{\text{graph} I} \text{Airplane-not-in-danger-init } l \wedge$
 $a @_{\text{graph} I} \text{Airplane-not-in-danger-init } l' \longrightarrow l = l') \wedge$
 $(\forall l. \text{nodup } a (\text{agra } (\text{graph} I) \text{Airplane-not-in-danger-init } l))$

$\langle \text{proof} \rangle$

lemma *actors-unique-loc-aid-step:*

$(\text{Airplane-not-in-danger-init}, I) \in \{(x::\text{infrastructure}, y::\text{infrastructure}). x \rightarrow_n y\}^*$

$\implies \forall a. (\forall l l'. a @_{\text{graph} I} l \wedge a @_{\text{graph} I} l' \longrightarrow l = l') \wedge$
 $(\forall l. \text{nodup } a (\text{agra } (\text{graph} I) l))$

$\langle \text{proof} \rangle$

Using the state transition, Kripke structure and CTL, we can now also express (and prove!) unreachability properties which enable to formally verify security properties for specific policies, like two-person rule.

lemma *Anid-airplane-actors: actors-graph* $(\text{graph} I \text{Airplane-not-in-danger-init}) = \text{airplane-actors}$

$\langle \text{proof} \rangle$

lemma *all-airplane-actors*: $(\text{Airplane-not-in-danger-init}, y) \in \{(x::\text{infrastructure}, y::\text{infrastructure}). x \rightarrow_n y\}^*$
 $\implies \text{actors-graph}(\text{graphI } y) = \text{airplane-actors}$
 $\langle \text{proof} \rangle$

lemma *actors-at-loc-in-graph*: $\llbracket l \in \text{nodes}(\text{graphI } I); a \ @_{\text{graphI } I} l \rrbracket$
 $\implies a \in \text{actors-graph } (\text{graphI } I)$
 $\langle \text{proof} \rangle$

lemma *not-en-get-Apnid*:
assumes $(\text{Airplane-not-in-danger-init}, y) \in \{(x::\text{infrastructure}, y::\text{infrastructure}). x \rightarrow_n y\}^*$
shows $\sim(\text{enables } y \ l \ (\text{Actor } a) \ \text{get})$
 $\langle \text{proof} \rangle$

lemma *Apnid-tsp-test*: $\sim(\text{enables } \text{Airplane-not-in-danger-init cockpit } (\text{Actor } "Alice") \ \text{get})$
 $\langle \text{proof} \rangle$

lemma *Apnid-tsp-test-gen*: $\sim(\text{enables } \text{Airplane-not-in-danger-init } l \ (\text{Actor } a) \ \text{get})$
 $\langle \text{proof} \rangle$

lemma *test-graph-atI*: $"Bob" \ @_{\text{graphI } \text{Airplane-not-in-danger-init cockpit}}$
 $\langle \text{proof} \rangle$

Invariant: number of staff in cockpit never below 2

lemma *two-person-inv*:
fixes $z \ z'$
assumes $(2::\text{nat}) \leq \text{length } (\text{agra } (\text{graphI } z) \ \text{cockpit})$
and $\text{nodes}(\text{graphI } z) = \text{nodes}(\text{graphI } \text{Airplane-not-in-danger-init})$
and $\text{delta}(z) = \text{delta}(\text{Airplane-not-in-danger-init})$
and $(\text{Airplane-not-in-danger-init}, z) \in \{(x::\text{infrastructure}, y::\text{infrastructure}). x \rightarrow_n y\}^*$
and $z \rightarrow_n z'$
shows $(2::\text{nat}) \leq \text{length } (\text{agra } (\text{graphI } z') \ \text{cockpit})$
 $\langle \text{proof} \rangle$

lemma *two-person-inv1*:
assumes $(\text{Airplane-not-in-danger-init}, z) \in \{(x::\text{infrastructure}, y::\text{infrastructure}). x \rightarrow_n y\}^*$
shows $(2::\text{nat}) \leq \text{length } (\text{agra } (\text{graphI } z) \ \text{cockpit})$
 $\langle \text{proof} \rangle$

The version of `two_person_inv` above we need, uses cardinality of lists of actors rather than length of lists. Therefore first some equivalences and then a restatement of `two_person_inv` in terms of sets

proof idea: show since there are no duplicates in the list $\text{agra}(\text{graphI } z)$ cockpit therefore then $\text{card}(\text{set}(\text{agra}(\text{graphI } z))) = \text{length}(\text{agra}(\text{graphI } z))$

lemma *nodup-card-insert*:

$a \notin \text{set } l \longrightarrow \text{card}(\text{insert } a (\text{set } l)) = \text{Suc}(\text{card}(\text{set } l))$
 $\langle \text{proof} \rangle$

lemma *no-dup-set-list-num-eq*[*rule-format*]:

$(\forall a. \text{nodup } a \ l) \longrightarrow \text{card}(\text{set } l) = \text{length } l$
 $\langle \text{proof} \rangle$

lemma *two-person-set-inv*:

assumes $(\text{Airplane-not-in-danger-init}, z) \in \{(x::\text{infrastructure}, y::\text{infrastructure}). x \rightarrow_n y\}^*$
shows $(2::\text{nat}) \leq \text{card}(\text{set}(\text{agra}(\text{graphI } z) \text{ cockpit}))$
 $\langle \text{proof} \rangle$

lemma *Pred-all-unique*: $\llbracket ? x. P \ x; (! x. P \ x \longrightarrow x = c) \rrbracket \Longrightarrow P \ c$

$\langle \text{proof} \rangle$

lemma *Set-all-unique*: $\llbracket S \neq \{\}; (\forall x \in S. x = c) \rrbracket \Longrightarrow c \in S$

$\langle \text{proof} \rangle$

lemma *airplane-actors-inv0*[*rule-format*]:

$\forall z \ z'. (\forall h::\text{char list} \in \text{set}(\text{agra}(\text{graphI } z) \text{ cockpit}). h \in \text{airplane-actors}) \wedge$
 $(\text{Airplane-not-in-danger-init}, z) \in \{(x::\text{infrastructure}, y::\text{infrastructure}). x \rightarrow_n y\}^* \wedge$
 $z \rightarrow_n z' \longrightarrow (\forall h::\text{char list} \in \text{set}(\text{agra}(\text{graphI } z') \text{ cockpit}). h \in \text{airplane-actors})$
 $\langle \text{proof} \rangle$

lemma *airplane-actors-inv*:

assumes $(\text{Airplane-not-in-danger-init}, z) \in \{(x::\text{infrastructure}, y::\text{infrastructure}). x \rightarrow_n y\}^*$
shows $\forall h::\text{char list} \in \text{set}(\text{agra}(\text{graphI } z) \text{ cockpit}). h \in \text{airplane-actors}$
 $\langle \text{proof} \rangle$

lemma *Eve-not-in-cockpit*: $(\text{Airplane-not-in-danger-init}, I)$

$\in \{(x::\text{infrastructure}, y::\text{infrastructure}). x \rightarrow_n y\}^* \Longrightarrow$

$x \in \text{set}(\text{agra}(\text{graphI } I) \text{ cockpit}) \Longrightarrow x \neq \text{"Eve"}$

$\langle \text{proof} \rangle$

2 person invariant implies that there is always some x in cockpit x not equal Eve

lemma *tp-imp-control*:

assumes $(\text{Airplane-not-in-danger-init}, I) \in \{(x::\text{infrastructure}, y::\text{infrastructure}). x \rightarrow_n y\}^*$
shows $(? x :: \text{identity}. x @_{\text{graphI } I} \text{ cockpit} \wedge \text{Actor } x \neq \text{Actor "Eve"})$
 $\langle \text{proof} \rangle$

lemma *Fend-2*: $(Airplane\text{-}not\text{-}in\text{-}danger\text{-}init, I) \in \{(x::infrastructure, y::infrastructure)\}.$
 $x \rightarrow_n y\}^* \implies$
 $\neg \text{enables } I \text{ cockpit (Actor "Eve")} \text{ put}$
 $\langle proof \rangle$

theorem *Four-eyes-no-danger*: $Air\text{-}tp\text{-}Kripke \vdash AG \{x. \text{global-policy } x \text{ "Eve"}\}$
 $\langle proof \rangle$

end

In the following we construct an instance of the locale airplane and proof that it is an interpretation. This serves the validation.

definition *airplane-actors-def'*: $airplane\text{-}actors \equiv \{"Bob", "Charly", "Alice"\}$

definition *airplane-locations-def'*:

$airplane\text{-}locations \equiv \{Location\ 0, Location\ 1, Location\ 2\}$

definition *cockpit-def'*: $cockpit \equiv Location\ 2$

definition *door-def'*: $door \equiv Location\ 1$

definition *cabin-def'*: $cabin \equiv Location\ 0$

definition *global-policy-def'*: $global\text{-}policy\ I\ a \equiv a \notin airplane\text{-}actors$
 $\longrightarrow \neg(\text{enables } I \text{ cockpit (Actor } a) \text{ put})$

definition *ex-creds-def'*: $ex\text{-}creds \equiv$

$(\lambda x. (\text{if } x = \text{Actor "Bob"}$
 $\text{then } (["PIN"], ["pilot"])$
 $\text{else } (\text{if } x = \text{Actor "Charly"}$
 $\text{then } (["PIN"], ["copilot"])$
 $\text{else } (\text{if } x = \text{Actor "Alice"}$
 $\text{then } (["PIN"], ["flightattendant"])$
 $\text{else } ([], []))))$

definition *ex-locs-def'*: $ex\text{-}locs \equiv (\lambda x. \text{if } x = \text{door then } ["norm"] \text{ else}$
 $(\text{if } x = \text{cockpit then } ["air"] \text{ else } []))$

definition *ex-locs'-def'*: $ex\text{-}locs' \equiv (\lambda x. \text{if } x = \text{door then } ["locked"] \text{ else}$
 $(\text{if } x = \text{cockpit then } ["air"] \text{ else } []))$

definition *ex-graph-def'*: $ex\text{-}graph \equiv Lgraph$

$\{(cockpit, door), (door, cabin)\}$
 $(\lambda x. \text{if } x = \text{cockpit then } ["Bob", "Charly"]$
 $\text{else } (\text{if } x = \text{door then } []$
 $\text{else } (\text{if } x = \text{cabin then } ["Alice"] \text{ else } []))$
 $ex\text{-}creds\ ex\text{-}locs$

definition *aid-graph-def'*: $aid\text{-}graph \equiv Lgraph$

$\{(cockpit, door), (door, cabin)\}$
 $(\lambda x. \text{if } x = \text{cockpit then } ["Charly"]$
 $\text{else } (\text{if } x = \text{door then } []$
 $\text{else } (\text{if } x = \text{cabin then } ["Bob", "Alice"] \text{ else } []))$
 $ex\text{-}creds\ ex\text{-}locs'$

definition *aid-graph0-def'*: $\text{aid-graph0} \equiv \text{Lgraph}$
 $\{(cockpit, door), (door, cabin)\}$
 $(\lambda x. \text{if } x = cockpit \text{ then } ["Charly"]$
 $\quad \text{else } (\text{if } x = door \text{ then } ["Bob"]$
 $\quad \quad \text{else } (\text{if } x = cabin \text{ then } ["Alice"] \text{ else } []))$
 ex-creds ex-locs

definition *agid-graph-def'*: $\text{agid-graph} \equiv \text{Lgraph}$
 $\{(cockpit, door), (door, cabin)\}$
 $(\lambda x. \text{if } x = cockpit \text{ then } ["Charly"]$
 $\quad \text{else } (\text{if } x = door \text{ then } []$
 $\quad \quad \text{else } (\text{if } x = cabin \text{ then } ["Bob", "Alice"] \text{ else } []))$
 ex-creds ex-locs

definition *local-policies-def'*: $\text{local-policies } G \equiv$
 $(\lambda y. \text{if } y = cockpit \text{ then}$
 $\quad \{(\lambda x. (? n. (n @_G cockpit) \wedge \text{Actor } n = x), \{put\}),$
 $\quad \quad (\lambda x. (? n. (n @_G cabin) \wedge \text{Actor } n = x \wedge \text{has } G(x, "PIN")$
 $\quad \quad \quad \wedge \text{isin } G \text{ door "norm"}), \{move\})$
 $\quad \}$
 $\quad \text{else } (\text{if } y = door \text{ then } \{(\lambda x. \text{True}, \{move\}),$
 $\quad \quad (\lambda x. (? n. (n @_G cockpit) \wedge \text{Actor } n = x), \{put\})\}$
 $\quad \quad \text{else } (\text{if } y = cabin \text{ then } \{(\lambda x. \text{True}, \{move\})\}$
 $\quad \quad \quad \text{else } \{\})\}))$

definition *local-policies-four-eyes-def'*: $\text{local-policies-four-eyes } G \equiv$
 $(\lambda y. \text{if } y = cockpit \text{ then}$
 $\quad \{(\lambda x. (? n. (n @_G cockpit) \wedge \text{Actor } n = x) \wedge$
 $\quad \quad 2 \leq \text{length}(\text{agra } G y) \wedge (\forall h \in \text{set}(\text{agra } G y). h \in \text{airplane-actors}),$
 $\quad \quad \{put\}),$
 $\quad \quad (\lambda x. (? n. (n @_G cabin) \wedge \text{Actor } n = x \wedge \text{has } G(x, "PIN") \wedge$
 $\quad \quad \quad \text{isin } G \text{ door "norm"}), \{move\})$
 $\quad \}$
 $\quad \text{else } (\text{if } y = door \text{ then}$
 $\quad \quad \{(\lambda x. ((? n. (n @_G cockpit) \wedge \text{Actor } n = x) \wedge 3 \leq \text{length}(\text{agra } G$
 $\text{cockpit})), \{move\})\}$
 $\quad \quad \text{else } (\text{if } y = cabin \text{ then}$
 $\quad \quad \quad \{(\lambda x. ((? n. (n @_G door) \wedge \text{Actor } n = x)), \{move\})\}$
 $\quad \quad \quad \text{else } \{\})\}))$

definition *Airplane-scenario-def'*:
 $\text{Airplane-scenario} \equiv \text{Infrastructure ex-graph local-policies}$

definition *Airplane-in-danger-def'*:
 $\text{Airplane-in-danger} \equiv \text{Infrastructure aid-graph local-policies}$

Intermediate step where pilot left cockpit but door still in norm position

definition *Airplane-getting-in-danger0-def'*:
 $\text{Airplane-getting-in-danger0} \equiv \text{Infrastructure aid-graph0 local-policies}$

definition *Airplane-getting-in-danger-def'*:

Airplane-getting-in-danger \equiv *Infrastructure agid-graph local-policies*

definition *Air-states-def'*: *Air-states* $\equiv \{ I. \text{Airplane-scenario} \rightarrow_n^* I \}$

definition *Air-Kripke-def'*: *Air-Kripke* \equiv *Kripke Air-states* $\{ \text{Airplane-scenario} \}$

definition *Airplane-not-in-danger-def'*:

Airplane-not-in-danger \equiv *Infrastructure aid-graph local-policies-four-eyes*

definition *Airplane-not-in-danger-init-def'*:

Airplane-not-in-danger-init \equiv *Infrastructure ex-graph local-policies-four-eyes*

definition *Air-tp-states-def'*: *Air-tp-states* $\equiv \{ I. \text{Airplane-not-in-danger-init} \rightarrow_n^* I \}$

definition *Air-tp-Kripke-def'*:

Air-tp-Kripke \equiv *Kripke Air-tp-states* $\{ \text{Airplane-not-in-danger-init} \}$

definition *Safety-def'*: *Safety* $I a \equiv a \in \text{airplane-actors}$

$\longrightarrow (\text{enables } I \text{ cockpit } (\text{Actor } a) \text{ move})$

definition *Security-def'*: *Security* $I a \equiv (\text{isin } (\text{graphI } I) \text{ door } \text{"locked"})$

$\longrightarrow \neg(\text{enables } I \text{ cockpit } (\text{Actor } a) \text{ move})$

definition *foe-control-def'*: *foe-control* $l c \equiv$

$(! I :: \text{infrastructure}. (? x :: \text{identity}.$

$x @_{\text{graphI } I} l \wedge \text{Actor } x \neq \text{Actor } \text{"Eve"})$

$\longrightarrow \neg(\text{enables } I l (\text{Actor } \text{"Eve"}) c))$

definition *astate-def'*: *astate* $x \equiv$

$(\text{case } x \text{ of}$

$\text{"Eve"} \Rightarrow \text{Actor-state depressed } \{ \text{revenge}, \text{peer-recognition} \}$

$| - \Rightarrow \text{Actor-state happy } \{ \})$

print-interps *airplane*

The additional assumption identified in the case study needs to be given as an axiom

axiomatization where

cockpit-foe-control': *foe-control cockpit put*

(The following addresses the issue of redefining an abstract type. We experimented with suggestion given here: Makarius Wenzel, Re: [isabelle] typedecl versus explicit type parameters, Isabelle users mailing list, 2009, <https://lists.cam.ac.uk/pipermail/cl-isabelle-users/2009-July/msg00111.html>.) We furthermore need axiomatization to add the missing semantics to the abstractly declared type actor and thereby be able to redefine consts Actor. Since the function Actor has also

been defined as a `consts :: identity =i actor` as an abstract function without a definition, we now also now add its semantics mimicking some of the concepts of the conservative type definition of HOL. The alternative method of using a Locale to replace the abstract `type_decl actor` in the `AirInsider` is a more elegant method for representing and abstract type actor but it is not working properly for our framework since it necessitates introducing a type parameter `'actor` into infrastructures which then makes it impossible to instantiate them to the `typeclass state` in order to use CTL and Kripke and the generic state transition. Therefore, we go the former way of a post-hoc axiomatic redefinition of the abstract type actor by using axiomatization of the existing Locale `"type_definition"`. This is done in the following. It allows to abstractedly assume as an axiom that there is a type definition for the abstract type actor. Adding a suitable definition of a representation for this type then additionally enables to introduce a definition for the function `Actor` (again using axiomatization to enforce the new definition).

definition *Actor-Abs* :: *identity* \Rightarrow *identity option*

where

Actor-Abs *x* \equiv (if *x* \in {"Eve", "Charly"} then *None* else *Some* *x*)

lemma *UasI-ActorAbs*: *Actor-Abs* "Eve" = *Actor-Abs* "Charly" \wedge

($\forall (x::char\ list). y::char\ list. x \neq "Eve" \wedge y \neq "Eve" \wedge Actor-Abs\ x = Actor-Abs\ y \longrightarrow x = y$)

<proof>

lemma *Actor-Abs-ran*: *Actor-Abs* *x* \in {*y* :: *identity option*. *y* \in *Some* ' {*x* :: *identity*. *x* \notin {"Eve", "Charly"}} | *y* = *None*}

<proof>

With the following axiomatization, we can simulate the abstract type actor and postulate some unspecified `Abs` and `Rep` functions between it and the simulated `identity option` subtype.

axiomatization **where** *Actor-type-def*:

type-definition (*Rep* :: *actor* \Rightarrow *identity option*)(*Abs* :: *identity option* \Rightarrow *actor*)
{*y* :: *identity option*. *y* \in *Some* ' {*x* :: *identity*. *x* \notin {"Eve", "Charly"}} | *y* = *None*}

lemma *Abs-inj-on*: $\bigwedge Abs\ Rep:: actor \Rightarrow char\ list\ option. x \in \{y :: identity\ option. y \in Some\ ' \{x :: identity. x \notin \{"Eve", "Charly"\}\} | y = None\}$

$\implies y \in \{y :: identity\ option. y \in Some\ ' \{x :: identity. x \notin \{"Eve", "Charly"\}\} | y = None\}$

$\implies (Abs :: char\ list\ option \Rightarrow actor)\ x = Abs\ y \implies x = y$

<proof>

lemma *Actor-td-Abs-inverse*:

($y \in \{y :: identity\ option. y \in Some\ ' \{x :: identity. x \notin \{"Eve", "Charly"\}\} | y = None\}$) \implies

(*Rep* :: *actor* \Rightarrow *identity option*)((*Abs* :: *identity option* \Rightarrow *actor*) *y*) = *y*

⟨proof⟩

Now, we can redefine the function Actor using a second axiomatization

axiomatization where *Actor-redef*: $Actor = (Abs :: identity\ option \Rightarrow actor) \circ Actor-Abs$

need to show that $Abs (Actor-Abs\ x) = Abs (Actor-Abs\ y) \longrightarrow Actor-Abs\ x = Actor-Abs\ y$, i.e. *injective Abs*. Generally, Abs is not injective but *injective-on* the type predicate. So, need to show that for any x, *Actor-Abs* x is in the type predicate, then it would follow. What is the type predicate? $\{y. y \in Some\ ' \{x. x \notin \{"Eve", "Charly"\}\} \vee y = None\}$

lemma *UasI-Actor-redef*:

$\bigwedge Abs\ Rep :: actor \Rightarrow char\ list\ option.$

$((Abs :: identity\ option \Rightarrow actor) \circ Actor-Abs)\ "Eve" = ((Abs :: identity\ option \Rightarrow actor) \circ Actor-Abs)\ "Charly" \wedge$

$(\forall (x :: char\ list)\ y :: char\ list. x \neq "Eve" \wedge y \neq "Eve" \wedge$

$((Abs :: identity\ option \Rightarrow actor) \circ Actor-Abs)\ x = ((Abs :: identity\ option \Rightarrow actor) \circ Actor-Abs)\ y$
 $\longrightarrow x = y)$

⟨proof⟩

Finally all of this allows us to show the last assumption contained in the Insider Locale assumption needed for the interpretation of airplane.

lemma *UasI-Actor*: *UasI "Eve" "Charly"*

⟨proof⟩

interpretation *airplane airplane-actors airplane-locations cockpit door cabin global-policy*

ex-creds ex-locs ex-locs' ex-graph aid-graph aid-graph0 agid-graph
local-policies local-policies-four-eyes Airplane-scenario Airplane-in-danger
Airplane-getting-in-danger0 Airplane-getting-in-danger Air-states

Air-Kripke

Airplane-not-in-danger Airplane-not-in-danger-init Air-tp-states
Air-tp-Kripke Safety Security foe-control astate

⟨proof⟩

end

References

- [1] F. Kammüller and M. Kerber. Investigating airplane safety and security against insider threats using logical modeling. In *IEEE Security and Privacy Workshops, Workshop on Research in Insider Threats, WRIT'16*. IEEE, 2016.
- [2] F. Kammüller and M. Kerber. Applying the isabelle insider framework to airplane security, 2020. arxiv preprint 2003.11838.

- [3] F. Kammüller and C. W. Probst. Modeling and verification of insider threats using logical analysis. *IEEE Systems Journal, Special issue on Insider Threats to Information Security, Digital Espionage, and Counter Intelligence*, 11(2):534–545, 2017.
- [4] M. Wenzel. Re: [isabelle] typedecl versus explicit type parameters, 2009. Isabelle users mailing list.