Applying the Isabelle Insider Framework to Airplane Security

Florian Kammüller and Manfred Kerber

March 29, 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 methododology are described in more detail in the preprint [2].

Contents

1 Fixpoint lemmas to support the definition of Kripke structures and CTL

theory *MC*
imports *Main*
begin

thm *monotone-def*
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*: $monotone\ \tau \longrightarrow (lfp\ \tau = \bigcap\ \{Z.\ \tau\ Z \subseteq Z\})$
⟨*proof*⟩

lemma *gfp1*: $monotone\ \tau \longrightarrow (gfp\ \tau = \bigcup\ \{Z.\ Z \subseteq \tau\ Z\})$
⟨*proof*⟩

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

lemma *predtrans-empty*:
  assumes $monotone\ \tau$
  shows $\forall\ i.\ (\tau\ \hat{}\ i)\ (\{\}) \subseteq (\tau\ \hat{}(i + 1))(\{\})$
⟨*proof*⟩

lemma *ex-card*: $finite\ S \Longrightarrow \exists\ n{::}\ nat.\ card\ S = n$
⟨*proof*⟩

lemma *less-not-le*: $[\![(x{::}\ nat) < y;\ y \leq x]\!] \Longrightarrow False$
⟨*proof*⟩

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

lemma *no-infinite-subset-chain*:
   assumes $finite\ (UNIV :: 'a\ set)$
    and $monotone\ (\tau :: ('a\ set \Rightarrow 'a\ set))$
    and $\forall i :: nat.\ ((\tau :: 'a\ set \Rightarrow 'a\ set)\ \hat{}\ i)\ \{\} \subset (\tau\ \hat{}\ i + (1 :: nat))\ (\{\} :: 'a\ set)$
  shows *False*

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

$\langle proof \rangle$

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

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

$\langle proof \rangle$

lemma *predtrans-UNIV*:
 assumes $monotone\ \tau$
 shows $\forall\ i.\ (\tau\ \hat{}\ i)\ (UNIV) \supseteq (\tau\ \hat{}(i + 1))(UNIV)$
$\langle proof \rangle$

lemma *Suc-less-le*: $x < (y - n) \Longrightarrow x \leq (y - (Suc\ n))$
$\langle proof \rangle$

lemma *card-univ-subtract*:
 assumes $finite\ (UNIV :: 'a\ set)$ **and** $monotone\ (\tau :: 'a\ set \Rightarrow 'a\ set)$
  and $(\forall\, i :: nat.\ ((\tau :: 'a\ set \Rightarrow 'a\ set)\ \hat{}\ i + (1 :: nat))\ (UNIV :: 'a\ set) \subset (\tau\ \hat{}\ i)\ UNIV)$
 shows $(\forall\ i :: nat.\ card((\tau\ \hat{}\ i)\ (UNIV :: 'a\ set)) \leq (card\ (UNIV :: 'a\ set)) - i)$
$\langle proof \rangle$

lemma *card-UNIV-tau-i-below-zero*:
 assumes $finite\ (UNIV :: 'a\ set)$ **and** $monotone\ (\tau :: 'a\ set \Rightarrow 'a\ set)$
 and $(\forall\, i :: nat.\ ((\tau :: 'a\ set \Rightarrow 'a\ set)\ \hat{}\ i + (1 :: nat))\ (UNIV :: 'a\ set) \subset (\tau\ \hat{}\ i)\ UNIV)$
 shows $card((\tau\ \hat{}\ (card\ (UNIV :: 'a\ set)))\ (UNIV :: 'a\ set)) \leq 0$
$\langle proof \rangle$

lemma *finite-card-zero-empty*: $[\![\ finite\ S;\ card\ S \leq 0]\!] \Longrightarrow S = \{\}$
$\langle proof \rangle$

lemma *UNIV-tau-i-is-empty*:
 assumes $finite\ (UNIV :: 'a\ set)$ **and** $monotone\ (\tau :: 'a\ set \Rightarrow 'a\ set)$
  and $(\forall\, i :: nat.\ ((\tau :: 'a\ set \Rightarrow 'a\ set)\ \hat{}\ i + (1 :: nat))\ (UNIV :: 'a\ set) \subset (\tau\ \hat{}\ i)\ UNIV)$

shows $(\tau \ \hat{}\ (card\ (UNIV\ ::'a\ set)))\ (UNIV\ ::'a\ set) = \{\}$
⟨*proof*⟩

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

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

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

lemma *mono-monotone*: *mono* $(\tau\ ::\ ('a\ set \Rightarrow 'a\ set)) \Longrightarrow$ *monotone* τ
⟨*proof*⟩

lemma *monotone-mono*: *monotone* $(\tau\ ::\ ('a\ set \Rightarrow 'a\ set)) \Longrightarrow$ *mono* τ
⟨*proof*⟩

lemma *power-power*: $((\tau\ ::\ ('a\ set \Rightarrow 'a\ set))\ \hat{}\hat{}\ n) = ((\tau\ ::\ ('a\ set \Rightarrow 'a\ set))\ \hat{}$
$n)$
⟨*proof*⟩

lemma *lfp-Kleene-iter-set*: *monotone* $(f\ ::\ ('a\ set \Rightarrow 'a\ set)) \Longrightarrow$
$(f\ \hat{}\ Suc(n))\ \{\} = (f\ \hat{}\ n)\ \{\} \Longrightarrow lfp\ f = (f\ \hat{}\ n)\{\}$
⟨*proof*⟩

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

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-gpfp*:
assumes *mono f* **and** $p \le f\ p$ **shows** $p \le (f\hat{}\hat{}k)\ (top::'a::order\text{-}top)$
⟨*proof*⟩

lemma *gfp-Kleene-iter*: **assumes** *mono f* **and** $(f\hat{}\hat{}Suc\ k)\ top = (f\hat{}\hat{}k)\ top$
shows $gfp\ f = (f\hat{}\hat{}k)\ top$
⟨*proof*⟩

lemma *gfp-Kleene-iter-set*:
assumes *monotone* $(f :: ('a\ set \Rightarrow 'a\ set))$
and $(f \mathbin{\hat{}} Suc(n))\ UNIV = (f \mathbin{\hat{}} n)\ UNIV$
shows $gfp\ f = (f \mathbin{\hat{}} n)\ UNIV$
⟨*proof*⟩

lemma *gfp-loop*:
assumes *finite* $(UNIV :: 'b\ set)$
and *monotone* $(\tau :: ('b\ set \Rightarrow 'b\ set))$
shows $\exists\ n\ .\ gfp\ \tau = (\tau \mathbin{\hat{}} n)(UNIV :: 'b\ set)$
⟨*proof*⟩

Definitions of the generic type of state with state transition and CTL Operators

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

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

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** $ER\ f1\ f2 \equiv gfp(\lambda\ Z.\ f2 \cap (f1 \cup EX'\ Z))$

Kripke and Modelchecking

datatype *'a kripke* =
Kripke 'a set 'a set

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

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

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

Support lemmas

lemma *EF-lem0*: $(x \in EF\ f) = (x \in f \cup EX'\ (lfp\ (\lambda Z :: ('a :: state)\ set.\ f \cup EX'\ Z)))$
⟨*proof*⟩

lemma *EF-lem00*: $(EF\ f) = (f \cup EX'\ (lfp\ (\lambda\ Z :: ('a :: state)\ set.\ f \cup EX'\ Z)))$

⟨*proof*⟩

lemma *EF-lem000*: $(EF\ f) = (f \cup EX'\ (EF\ f))$
⟨*proof*⟩

lemma *EF-lem1*: $x \in f \vee x \in (EX'\ (EF\ f)) \Longrightarrow x \in EF\ f$
⟨*proof*⟩

lemma *EF-lem2b*:
  assumes $x \in (EX'\ (EF\ f))$
  shows $x \in EF\ f$
⟨*proof*⟩

lemma *EF-lem2a*: **assumes** $x \in f$ **shows** $x \in EF\ f$
⟨*proof*⟩

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

lemma *EF-lem2d*: **assumes** $x \notin EF\ f$ **shows** $x \notin f$
⟨*proof*⟩

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

lemma *EX-lem0l*: $x \in (EX'\ f) \Longrightarrow x \in (EX'\ (f \cup g))$
⟨*proof*⟩

lemma *EX-lem0r*: $x \in (EX'\ g) \Longrightarrow x \in (EX'\ (f \cup g))$
⟨*proof*⟩

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

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

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

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

lemma *EF-step-star*: $[\![\ x\ \rightarrow_i\!*\ y;\ y \in f\ ]\!] \Longrightarrow x \in EF\ f$
⟨*proof*⟩

lemma *EF-induct-prep*:
 assumes $(a::'a::state) \in lfp\ (\lambda\ Z.\ (f::'a::state\ set) \cup EX'\ Z)$

and *mono* $(\lambda\ Z.\ (f{::}'a{::}state\ set) \cup EX'\ Z)$
shows $(\bigwedge x{::}'a{::}state.$
$x \in ((\lambda\ Z.\ (f{::}'a{::}state\ set) \cup EX'\ Z)(lfp\ (\lambda\ Z.\ (f{::}'a{::}state\ set) \cup EX'\ Z) \cap \{x{::}'a{::}state.\ (P{::}'a{::}state \Rightarrow bool)\ x\})) \Longrightarrow P\ x) \Longrightarrow$
$P\ a$
$\langle proof \rangle$

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

lemma *valEF-E*: $M \vdash EF\ f \Longrightarrow x \in init\ M \Longrightarrow x \in EF\ f$
$\langle proof \rangle$

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

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

AG lemmas

lemma *AG-in-lem*: $x \in AG\ s \Longrightarrow x \in s$
$\langle proof \rangle$

lemma *AG-lem1*: $x \in s \wedge x \in (AX\ (AG\ s)) \Longrightarrow x \in AG\ s$
$\langle proof \rangle$

lemma *AG-lem2*: $x \in AG\ s \Longrightarrow x \in (s \cap (AX\ (AG\ s)))$
$\langle proof \rangle$

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

lemma *AG-step*: $y \rightarrow_i z \Longrightarrow y \in AG\ s \Longrightarrow z \in AG\ s$
$\langle proof \rangle$

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

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

$\langle proof \rangle$

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

end

2 Insider

theory *AirInsider*
imports *MC*
begin
datatype $action = get \mid move \mid eval \mid put$

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 =¿ 'actor" begin ... 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 outside 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 $igraph = Lgraph\ (location * location)set\ location \Rightarrow identity\ list$
$\quad actor \Rightarrow (string\ list * string\ list)\ \ location \Rightarrow string\ list$
datatype $infrastructure =$
$\quad Infrastructure\ igraph$
$\quad\quad [igraph, location] \Rightarrow policy\ set$

primrec $loc :: location \Rightarrow nat$
where $loc(Location\ n) = n$
primrec $gra :: igraph \Rightarrow (location * location)set$
where $gra(Lgraph\ g\ a\ c\ l) = g$

primrec *agra* :: *igraph* ⇒ (*location* ⇒ *identity list*)
where *agra*(*Lgraph g a c l*) = *a*
primrec *cgra* :: *igraph* ⇒ (*actor* ⇒ *string list* ∗ *string list*)
where *cgra*(*Lgraph g a c l*) = *c*
primrec *lgra* :: *igraph* ⇒ (*location* ⇒ *string list*)
where *lgra*(*Lgraph g a c l*) = *l*

definition *nodes* :: *igraph* ⇒ *location set*
where *nodes g* == { *x*. (*? y*. ((*x*,*y*): *gra g*) | ((*y*,*x*): *gra g*))}

definition *actors-graph* :: *igraph* ⇒ *identity set*
where *actors-graph g* == {*x*. *? y*. *y* : *nodes g* ∧ *x* ∈ *set*(*agra g y*)}

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

definition *credentials* :: *string list* ∗ *string list* ⇒ *string set*
where *credentials lxl* ≡ *set* (*fst lxl*)
definition *has* :: [*igraph*, *actor* ∗ *string*] ⇒ *bool*
where *has G ac* ≡ *snd ac* ∈ *credentials*(*cgra G* (*fst ac*))
definition *roles* :: *string list* ∗ *string list* ⇒ *string set*
where *roles lxl* ≡ *set* (*snd lxl*)
definition *role* :: [*igraph*, *actor* ∗ *string*] ⇒ *bool*
where *role G ac* ≡ *snd ac* ∈ *roles*(*cgra G* (*fst ac*))

definition *isin* :: [*igraph*,*location*, *string*] ⇒ *bool*
where *isin G l s* ≡ *s* ∈ *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* ⇒ *motivations set*
where *motivation* (*Actor-state p m*) = *m*
primrec *psy-state* :: *actor-state* ⇒ *psy-states*
where *psy-state* (*Actor-state p m*) = *p*

definition *tipping-point* :: *actor-state* ⇒ *bool* **where**
tipping-point a ≡ ((*motivation a* ≠ {}) ∧ (*happy* ≠ *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 $Insider\ a\ C\ as \equiv (tipping\text{-}point\ (as\ a) \longrightarrow (\forall\ b{\in}C.\ UasI\ a\ b))$

definition *Insider'* :: [*actor* $\Rightarrow$ *bool*, *identity*, *identity set*, *identity* $\Rightarrow$ *actor-state*] $\Rightarrow$ *bool*
where $Insider'\ P\ a\ C\ as \equiv (tipping\text{-}point\ (as\ a) \longrightarrow (\forall\ b{\in}C.\ UasI'\ P\ a\ b \wedge inj\text{-}on\ Actor\ C))$

definition *atI* :: [*identity*, *igraph*, *location*] $\Rightarrow$ *bool* (- $@_{(\text{-})}$ - *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 $misbehaviour\ I \equiv -(behaviour\ I)$

basic lemmas for enable

lemma *not-enableI*: $(\forall\ (p,e) \in delta\ I\ (graphI\ I)\ l.\ ({\sim}(h : e) \mid ({\sim}(p(a)))))$
$\Longrightarrow {\sim}(enables\ I\ l\ a\ h)$
⟨*proof*⟩

lemma *not-enableI2*: $[\![\bigwedge\ p\ e.\ (p,e) \in delta\ I\ (graphI\ I)\ l \Longrightarrow$
$({\sim}(t : e) \mid\ ({\sim}(p(a))))\ ]\!] \Longrightarrow {\sim}(enables\ I\ l\ a\ t)$
⟨*proof*⟩

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

$\Longrightarrow (\sim(t : e) \mid (\sim(p(a))))$
⟨*proof*⟩

lemma *not-enableE2*: $[\![\sim(enables\ I\ l\ a\ t);\ (p,e) \in delta\ I\ (graphI\ I)\ l;$
$t : e\ ]\!] \Longrightarrow (\sim(p(a)))$
⟨*proof*⟩

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 $move\text{-}graph\text{-}a\ n\ l\ l'\ g \equiv Lgraph\ (gra\ g)$
$(if\ n \in set\ ((agra\ g)\ l)\ \&\ \ n \notin set\ ((agra\ g)\ l')\ then$
$((agra\ g)(l := del\ n\ (agra\ g\ l)))(l' := (n\ \#\ (agra\ g\ l')))$
$else\ (agra\ g))(cgra\ g)(lgra\ g)$

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

inductive *state-transition-in* :: $[infrastructure,\ infrastructure] \Rightarrow bool\ ((\text{-} \rightarrow_n \text{-})\ 50)$
where
move: $[\![\ G = graphI\ I;\ a\ @_G\ l;\ l \in nodes\ G;\ l' \in nodes\ G;$
$(a) \in actors\text{-}graph(graphI\ I);\ enables\ I\ l'\ (Actor\ a)\ move;$
$I' = Infrastructure\ (move\text{-}graph\text{-}a\ a\ l\ l'\ (graphI\ I))(delta\ I)\ ]\!] \Longrightarrow I \rightarrow_n I'$
| *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* : $[\![\ G = graphI\ I;\ a\ @_G\ l;\ enables\ I\ l\ (Actor\ a)\ put;$
$I' = Infrastructure$

$(Lgraph\ (gra\ G)(agra\ G)(cgra\ G)$
$((lgra\ G)(l := [z])))$
$(delta\ I)\ \rrbracket$
$\Longrightarrow I \rightarrow_n I'$
$|\ put\text{-}remote : \llbracket\ G = graphI\ I;\ enables\ I\ l\ (Actor\ a)\ put;$
$I' = Infrastructure$
$(Lgraph\ (gra\ G)(agra\ G)(cgra\ G)$
$((lgra\ G)(l := [z])))$
$(delta\ I)\ \rrbracket$
$\Longrightarrow I \rightarrow_n I'$

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

instantiation *infrastructure* :: *state*
begin

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

instance
⟨*proof*⟩

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

lemma *del-del*[*rule-format*]: $n \in set\ (del\ a\ S) \longrightarrow n \in set\ S$
⟨*proof*⟩

lemma *del-dec*[*rule-format*]: $a \in set\ S \longrightarrow length\ (del\ a\ S) < length\ S$
⟨*proof*⟩

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

lemma *del-jonce*: $jonce\ a\ l \longrightarrow a \notin set\ (del\ a\ l)$
⟨*proof*⟩

lemma *del-nodup*[*rule-format*]: $nodup\ a\ l \longrightarrow a \notin set(del\ a\ l)$
⟨*proof*⟩

lemma *nodup-up*[*rule-format*]: $a \in set\ (del\ a\ l) \longrightarrow a \in set\ l$
⟨*proof*⟩

lemma *del-up* [*rule-format*]: $a \in set\ (del\ aa\ l) \longrightarrow a \in set\ l$
⟨*proof*⟩

lemma *nodup-notin*[*rule-format*]: $a \notin set\ list \longrightarrow nodup\ a\ list$
⟨*proof*⟩

lemma *nodup-down*[*rule-format*]: *nodup a l* $\longrightarrow$ *nodup a* (*del a l*)
⟨*proof*⟩

lemma *del-notin-down*[*rule-format*]: $a \notin$ *set list* $\longrightarrow$ $a \notin$ *set* (*del aa list*)
⟨*proof*⟩

lemma *del-not-a*[*rule-format*]: $x \neq a \longrightarrow x \in$ *set l* $\longrightarrow x \in$ *set* (*del a l*)
⟨*proof*⟩

lemma *nodup-down-notin*[*rule-format*]: *nodup a l* $\longrightarrow$ *nodup a* (*del aa l*)
⟨*proof*⟩

lemma *move-graph-eq*: *move-graph-a a l l g* = *g*
⟨*proof*⟩

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

lemma *delta-invariant*: $\forall\ z\ z'.\ z \rightarrow_n z' \longrightarrow$ *delta*(z) = *delta*(z')
⟨*proof*⟩

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

lemma *init-state-policy*: $[\![\ (x,y) \in \{(x::infrastructure,\ y::infrastructure).\ x \rightarrow_n y\}^*\ ]\!] \Longrightarrow$
delta(x) = *delta*(y)
⟨*proof*⟩

lemma *same-nodes0*[*rule-format*]: $\forall\ z\ z'.\ z \rightarrow_n z' \longrightarrow$ *nodes*(*graphI* z) = *nodes*(*graphI* z')
⟨*proof*⟩

lemma *same-nodes*: $(I,\ y) \in \{(x::infrastructure,\ y::infrastructure).\ x \rightarrow_n y\}^*$
$\Longrightarrow$ *nodes*(*graphI* y) = *nodes*(*graphI* I)
⟨*proof*⟩

lemma *same-actors0*[*rule-format*]: $\forall\ z\ z'.\ z \rightarrow_n z' \longrightarrow$ *actors-graph*(*graphI* z) = *actors-graph*(*graphI* z')
⟨*proof*⟩

lemma *same-actors*: $(I,\ y) \in \{(x::infrastructure,\ y::infrastructure).\ x \rightarrow_n y\}^*$
$\Longrightarrow$ *actors-graph*(*graphI* I) = *actors-graph*(*graphI* y)
⟨*proof*⟩

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* ≡ {''*Bob*'', ''*Charly*'', ''*Alice*''}

fixes *airplane-locations* :: *location set*
defines *airplane-locations-def*:
airplane-locations ≡ {*Location 0*, *Location 1*, *Location 2*}

fixes *cockpit* :: *location*
defines *cockpit-def*: *cockpit* ≡ *Location 2*
fixes *door* :: *location*
defines *door-def*: *door* ≡ *Location 1*
fixes *cabin* :: *location*
defines *cabin-def*: *cabin* ≡ *Location 0*

fixes *global-policy* :: [*infrastructure*, *identity*] ⇒ *bool*
defines *global-policy-def*: *global-policy I a* ≡ *a* ∉ *airplane-actors*
⟶ ¬(*enables I cockpit* (*Actor a*) *put*)

fixes *ex-creds* :: *actor* ⇒ (*string list* ∗ *string list*)
defines *ex-creds-def*: *ex-creds* ≡
(λ *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* ⇒ *string list*
defines *ex-locs-def*: *ex-locs* ≡ (λ *x*. *if x* = *door then* [''*norm*''] *else*
(*if x* = *cockpit then* [''*air*''] *else* []))

fixes *ex-locs'* :: *location* ⇒ *string list*
defines *ex-locs'-def*: *ex-locs'* ≡ (λ *x*. *if x* = *door then* [''*locked*''] *else*
(*if x* = *cockpit then* [''*air*''] *else* []))
fixes *ex-graph* :: *igraph*
defines *ex-graph-def*: *ex-graph* ≡ *Lgraph*
{(*cockpit*, *door*),(*door*,*cabin*)}
(λ *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* :: *igraph*
defines *aid-graph-def*: *aid-graph* ≡ *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 *aid-graph0* :: *igraph*
defines *aid-graph0-def*: *aid-graph0* ≡ *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* ≡ *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*] ⇒ *policy set*
defines *local-policies-def*: *local-policies* G ≡
(λ y. *if* y = *cockpit* *then*
{(λ x. (*?* n. (n $@_G$ *cockpit*) $\wedge$ *Actor* n = x), {*put*}),
(λ x. (*?* n. (n $@_G$ *cabin*) $\wedge$ *Actor* n = x $\wedge$ *has* G (x, ''*PIN*'')
$\wedge$ *isin* G *door* ''*norm*''),{*move*})
}
else (*if* y = *door* *then* {(λ x. *True*, {*move*}),
(λ x. (*?* n. (n $@_G$ *cockpit*) $\wedge$ *Actor* n = x), {*put*})}
else (*if* y = *cabin* *then* {(λ x. *True*, {*move*})}
else {})))

fixes *local-policies-four-eyes* :: [*igraph*, *location*] ⇒ *policy set*
defines *local-policies-four-eyes-def*: *local-policies-four-eyes* G ≡
(λ y. *if* y = *cockpit* *then*
{(λ x. (*?* n. (n $@_G$ *cockpit*) $\wedge$ *Actor* n = x) $\wedge$
2 $\leq$ *length*(*agra* G y) $\wedge$ ($\forall$ h $\in$ *set*(*agra* G y). h $\in$ *airplane-actors*),
{*put*}),
(λ x. (*?* n. (n $@_G$ *cabin*) $\wedge$ *Actor* n = x $\wedge$ *has* G (x, ''*PIN*'') $\wedge$
isin G *door* ''*norm*''),{*move*})

}
else (*if* y = *door then*
{(λ x. ((*?* n. (n $@_G$ *cockpit*) $\wedge$ *Actor* n = x) $\wedge$ *3* $\leq$ *length*(*agra G cockpit*)), {*move*})}
else (*if* y = *cabin then*
{(λ x. ((*?* n. (n $@_G$ *door*) $\wedge$ *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* $\to_n *$ *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* $\to_n *$ *I* }

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

fixes *Safety* :: [*infrastructure*, *identity*] ⇒ *bool*
defines *Safety-def*: *Safety I a* ≡ *a* ∈ *airplane-actors*
⟶ (*enables I cockpit* (*Actor a*) *move*)

fixes *Security* :: [*infrastructure*, *identity*] ⇒ *bool*
defines *Security-def*: *Security I a* ≡ (*isin* (*graphI I*) *door* ''*locked*'')
⟶ ¬(*enables I cockpit* (*Actor a*) *move*)

fixes *foe-control* :: [*location*, *action*] ⇒ *bool*
defines *foe-control-def*: *foe-control l c* ≡
(! *I*:: *infrastructure*. (? *x* :: *identity*.
$x\ @_{graphI\ I}\ l \wedge Actor\ x \neq Actor$ ''*Eve*'')
⟶ ¬(*enables I l* (*Actor* ''*Eve*'') *c*))

fixes *astate*:: *identity* ⇒ *actor-state*
defines *astate-def*: *astate x* ≡ (*case x of*
''*Eve*'' ⇒ *Actor-state depressed* {*revenge*, *peer-recognition*}
| - ⇒ *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*: ¬*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*: ⟦ *inj f*; $x \neq y$ ⟧ ⟹ $f\ x \neq f\ y$
⟨*proof*⟩

lemma *inj-on-lem*: ⟦ *inj-on f A*; $x \neq y$; $x \in A$; $y \in A$ ⟧ ⟹ $f\ x \neq f\ y$
⟨*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''* ≠ *isin ex-graph door ''locked''*
⟨*proof*⟩

lemma *locl-lemma3*: *isin ex-graph door ''norm''* = (¬ *isin ex-graph door ''locked''*)
⟨*proof*⟩

lemma *locl-lemma2a*: *isin aid-graph door ''norm''* ≠ *isin aid-graph door ''locked''*
⟨*proof*⟩

lemma *locl-lemma3a*: *isin aid-graph door ''norm''* = (¬ *isin aid-graph door ''locked''*)
⟨*proof*⟩

lemma *Security*: *Security Airplane-scenario s*
⟨*proof*⟩

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

lemma *Security-problem*: *Security Airplane-scenario ''Bob''*
⟨*proof*⟩

show that pilot can get out of cockpit

lemma *pilot-can-leave-cockpit*: (*enables Airplane-scenario cabin* (*Actor ''Bob''*) *move*)
⟨*proof*⟩

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

lemma *ex-inv4*: ¬*global-policy Airplane-in-danger* (*''Eve''*)
⟨*proof*⟩

lemma *Safety-in-danger*:
fixes *s*
assumes *s* ∈ *airplane-actors*
shows ¬(*Safety Airplane-in-danger s*)
⟨*proof*⟩

lemma *Security-problem'*: ¬(*enables Airplane-in-danger cockpit* (*Actor ''Bob''*) *move*)
⟨*proof*⟩

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* ∈ *airplane-actors* ⟶ *global-policy Airplane-not-in-danger a*
⟨*proof*⟩

lemma *ex-inv6*: *global-policy Airplane-not-in-danger a*

⟨*proof*⟩

lemma *step0*: *Airplane-scenario* $\rightarrow_n$ *Airplane-getting-in-danger0*
⟨*proof*⟩

lemma *step1*: *Airplane-getting-in-danger0* $\rightarrow_n$ *Airplane-getting-in-danger*
⟨*proof*⟩

lemma *step2*: *Airplane-getting-in-danger* $\rightarrow_n$ *Airplane-in-danger*
⟨*proof*⟩

lemma *step0r*: *Airplane-scenario* $\rightarrow_n *$ *Airplane-getting-in-danger0*
⟨*proof*⟩

lemma *step1r*: *Airplane-getting-in-danger0* $\rightarrow_n *$ *Airplane-getting-in-danger*
⟨*proof*⟩

lemma *step2r*: *Airplane-getting-in-danger* $\rightarrow_n *$ *Airplane-in-danger*
⟨*proof*⟩

theorem *step-allr*: *Airplane-scenario* $\rightarrow_n *$ *Airplane-in-danger*
⟨*proof*⟩

theorem *aid-attack*: *Air-Kripke* $\vdash$ *EF* $(\{x.\ \neg\ \mathit{global\text{-}policy}\ x\ ''Eve''\})$
⟨*proof*⟩

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

lemma *actors-unique-loc-base*:
assumes $I \rightarrow_n I'$
and $(\forall\ l\ l'.\ a\ @_{graphI\ I}\ l \wedge a\ @_{graphI\ I}\ l' \longrightarrow l = l') \wedge$
$(\forall\ l.\ nodup\ a\ (agra\ (graphI\ I)\ l))$
shows $(\forall\ l\ l'.\ a\ @_{graphI\ I'}\ l \wedge a\ @_{graphI\ I'}\ l' \longrightarrow l = l') \wedge$
$(\forall\ l.\ nodup\ a\ (agra\ (graphI\ I')\ l))$
⟨*proof*⟩

lemma *actors-unique-loc-step*:
assumes $(I, I') \in \{(x::infrastructure,\ y::infrastructure).\ x \rightarrow_n y\}^*$
and $\forall\ a.\ (\forall\ l\ l'.\ a\ @_{graphI\ I}\ l \wedge a\ @_{graphI\ I}\ l' \longrightarrow l = l') \wedge$
$(\forall\ l.\ nodup\ a\ (agra\ (graphI\ I)\ l))$
shows $\forall\ a.\ (\forall\ l\ l'.\ a\ @_{graphI\ I'}\ l \wedge a\ @_{graphI\ I'}\ l' \longrightarrow l = l') \wedge$
$(\forall\ l.\ nodup\ a\ (agra\ (graphI\ I')\ l))$
⟨*proof*⟩

lemma *actors-unique-loc-aid-base*:
$\forall\ a.\ (\forall\ l\ l'.\ a\ @_{graphI\ Airplane\text{-}not\text{-}in\text{-}danger\text{-}init}\ l \wedge$
$a\ @_{graphI\ Airplane\text{-}not\text{-}in\text{-}danger\text{-}init}\ l' \longrightarrow l = l') \wedge$
$(\forall\ l.\ nodup\ a\ (agra\ (graphI\ Airplane\text{-}not\text{-}in\text{-}danger\text{-}init)\ l))$
⟨*proof*⟩

lemma *actors-unique-loc-aid-step*:
$(Airplane\text{-}not\text{-}in\text{-}danger\text{-}init,\ I) \in \{(x::infrastructure,\ y::infrastructure).\ x \rightarrow_n y\}^*$
$\Longrightarrow \quad \forall\ a.\ (\forall\ l\ l'.\ a\ @_{graphI\ I}\ l \wedge a\ @_{graphI\ I}\ l' \longrightarrow l = l') \wedge$
$(\forall\ l.\ nodup\ a\ (agra\ (graphI\ I)\ l))$
⟨*proof*⟩

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*: $actors\text{-}graph\ (graphI\ Airplane\text{-}not\text{-}in\text{-}danger\text{-}init) = airplane\text{-}actors$
⟨*proof*⟩

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

lemma *actors-at-loc-in-graph*: $[\![\ l \in nodes(graphI\ I);\ a\ \ @_{graphI\ I}\ l]\!]$
$\Longrightarrow a \in actors\text{-}graph\ (graphI\ I)$
⟨*proof*⟩

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

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

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

⟨*proof*⟩

lemma *test-graph-atI*: $''Bob''\ @_{graphI\ Airplane\text{-}not\text{-}in\text{-}danger\text{-}init}\ cockpit$
⟨*proof*⟩

Invariant: number of staff in cockpit never below 2

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

shows $(2::nat) \leq length\ (agra\ (graphI\ z')\ cockpit)$
⟨*proof*⟩

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

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 agra (graphI z) cockpit therefore then card(set(agra (graphI z))) = length(agra (graphI z))

lemma *nodup-card-insert*:
$a \notin set\ l \longrightarrow card\ (insert\ a\ (set\ l)) = Suc\ (card\ (set\ l))$
⟨*proof*⟩

lemma *no-dup-set-list-num-eq*[*rule-format*]:
$(\forall\ a.\ nodup\ a\ l) \longrightarrow card\ (set\ l) = length\ l$
⟨*proof*⟩

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

lemma *Pred-all-unique*: $[\![\ ?\ x.\ P\ x;\ (!\ x.\ P\ x \longrightarrow x = c)]\!] \Longrightarrow P\ c$
⟨*proof*⟩

lemma *Set-all-unique*: $[\![\ S \neq \{\};\ (\forall\ x \in S.\ x = c)\ ]\!] \Longrightarrow c \in S$
⟨*proof*⟩

lemma *airplane-actors-inv0*[*rule-format*]:
$\forall\ z\ z'.\ (\forall h::char\ list \in set\ (agra\ (graphI\ z)\ cockpit).\ h \in airplane\text{-}actors) \wedge$
$(Airplane\text{-}not\text{-}in\text{-}danger\text{-}init,z) \in \{(x::infrastructure,\ y::infrastructure).\ x \rightarrow_n y\}^* \wedge$
$z \rightarrow_n z' \longrightarrow (\forall h::char\ list{\in}set\ (agra\ (graphI\ z')\ cockpit).\ h \in airplane\text{-}actors)$
⟨*proof*⟩

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

lemma *Eve-not-in-cockpit*: $(\textit{Airplane-not-in-danger-init}, I)$
$\in \{(x{::}\textit{infrastructure}, y{::}\textit{infrastructure}).\ x \rightarrow_n y\}^* \Longrightarrow$
$x \in \textit{set}\ (\textit{agra}\ (\textit{graphI}\ I)\ \textit{cockpit}) \Longrightarrow x \neq ''\textit{Eve}''$
$\langle \textit{proof} \rangle$

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

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

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

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

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'*: $\textit{airplane-actors} \equiv \{''\textit{Bob}'', ''\textit{Charly}'', ''\textit{Alice}''\}$
definition *airplane-locations-def'*:
$\textit{airplane-locations} \equiv \{\textit{Location}\ 0, \textit{Location}\ 1, \textit{Location}\ 2\}$
definition *cockpit-def'*: $\textit{cockpit} \equiv \textit{Location}\ 2$
definition *door-def'*: $\textit{door} \equiv \textit{Location}\ 1$
definition *cabin-def'*: $\textit{cabin} \equiv \textit{Location}\ 0$
definition *global-policy-def'*: $\textit{global-policy}\ I\ a \equiv a \notin \textit{airplane-actors}$
$\longrightarrow \neg(\textit{enables}\ I\ \textit{cockpit}\ (\textit{Actor}\ a)\ \textit{put})$
definition *ex-creds-def'*: $\textit{ex-creds} \equiv$
$(\lambda\ x.(\textit{if}\ x = \textit{Actor}\ ''\textit{Bob}''$
$\textit{then}\ ([''\textit{PIN}''], [''\textit{pilot}''])$
$\textit{else}\ (\textit{if}\ x = \textit{Actor}\ ''\textit{Charly}''$
$\textit{then}\ ([''\textit{PIN}''],[''\textit{copilot}''])$
$\textit{else}\ (\textit{if}\ x = \textit{Actor}\ ''\textit{Alice}''$
$\textit{then}\ ([''\textit{PIN}''],[''\textit{flightattendant}''])$
$\textit{else}\ ([],[])))))$

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

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

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

definition *aid-graph-def'*: *aid-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'

definition *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

definition *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

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

$\quad\quad$ *else* (*if* y = *cabin then*
$\{(\lambda\ x.\ ((?\ n.\ (n\ @_G\ door) \wedge Actor\ n = x)), \{move\})\}$
else $\{\}$)))

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

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

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

definition *Airplane-getting-in-danger0-def'*:
Airplane-getting-in-danger0 $\equiv$ *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* $\equiv \{\ I.\ Airplane\text{-}scenario \rightarrow_n * I\ \}$

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

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* $\equiv \{\ I.\ Airplane\text{-}not\text{-}in\text{-}danger\text{-}init \rightarrow_n *$
$I\ \}$

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

definition *Safety-def'*: *Safety I a* $\equiv a \in$ *airplane-actors*
$\longrightarrow$ (*enables I cockpit* (*Actor a*) *move*)

definition *Security-def'*: *Security I a* $\equiv$ (*isin* (*graphI I*) *door* ''*locked*'')
$\longrightarrow \neg$(*enables I cockpit* (*Actor a*) *move*)

definition *foe-control-def'*: *foe-control l c* $\equiv$
(! I:: *infrastructure*. (? x :: *identity*.
$x\ @_{graphI\ I}\ l \wedge Actor\ x \neq Actor$ ''*Eve*'')
$\longrightarrow \neg$(*enables I l* (*Actor* ''*Eve*'') *c*))

definition *astate-def'*: *astate x* $\equiv$
(*case x of*
''*Eve*'' $\Rightarrow$ *Actor-state depressed* $\{revenge, peer\text{-}recognition\}$
| - $\Rightarrow$ *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 =¿ 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 framwework 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* ⇒ *identity option*
where
Actor-Abs x ≡ (*if x* ∈ {''*Eve*'', ''*Charly*''} *then None else Some x*)

lemma *UasI-ActorAbs*: *Actor-Abs* ''*Eve*'' = *Actor-Abs* ''*Charly*'' ∧
(∀ (*x*::*char list*) *y*::*char list*. *x* ≠ ''*Eve*'' ∧ *y* ≠ ''*Eve*'' ∧ *Actor-Abs x* = *Actor-Abs y* ⟶ *x* = *y*)
⟨*proof*⟩

lemma *Actor-Abs-ran*: *Actor-Abs x* ∈ {*y* :: *identity option*. *y* ∈ *Some* ‘ {*x* :: *identity*. *x* ∉ {''*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* ⇒ *identity option*)(*Abs* :: *identity option* ⇒ *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*. $x \in \{y :: identity\ option.\ y \in Some\ `\ \{x :: identity.\ x \notin \{''Eve'', ''Charly''\}\}|\ y = None\}$
$\Longrightarrow y \in \{y :: identity\ option.\ y \in Some\ `\ \{x :: identity.\ x \notin \{''Eve'', ''Charly''\}\}|\ y = None\}$
$\Longrightarrow (Abs :: char\ list\ option \Rightarrow actor)\ x = Abs\ y \Longrightarrow x = y$
$\langle proof \rangle$

lemma *Actor-td-Abs-inverse*:
$(y \in \{y :: identity\ option.\ y \in Some\ `\ \{x :: identity.\ x \notin \{''Eve'', ''Charly''\}\}|\ y = None\}) \Longrightarrow$
$(Rep :: actor \Rightarrow identity\ option)((Abs :: identity\ option \Rightarrow actor)\ y) = y$
$\langle proof \rangle$

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

axiomatization where *Actor-redef*: $Actor = (Abs :: identity\ option \Rightarrow actor) o\ Actor\text{-}Abs$

need to show that $Abs\ (Actor\text{-}Abs\ x) = Abs\ (Actor\text{-}Abs\ y) \longrightarrow Actor\text{-}Abs\ x = Actor\text{-}Abs\ y$, 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*.
$((Abs :: identity\ option \Rightarrow actor) o\ Actor\text{-}Abs)\ ''Eve'' = ((Abs :: identity\ option \Rightarrow actor) o\ Actor\text{-}Abs)\ ''Charly'' \wedge$
$(\forall (x{::}char\ list)\ y{::}char\ list.\ x \neq ''Eve'' \wedge y \neq ''Eve'' \wedge$
$((Abs :: identity\ option \Rightarrow actor) o\ Actor\text{-}Abs)\ x = ((Abs :: identity\ option \Rightarrow actor) o\ Actor\text{-}Abs)\ y$
$\longrightarrow x = y)$
$\langle proof \rangle$

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''*
$\langle proof \rangle$

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. arxive 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.