Applying the Isabelle Insider Framework to Airplane Security

Florian Kammüller and Manfred Kerber

April 9, 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 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 $\mathit{monotone} :: ('a\ \mathit{set} \Rightarrow 'a\ \mathit{set}) \Rightarrow \mathit{bool}$
where $\mathit{monotone}\ \tau \equiv (\forall\ p\ q.\ p \subseteq q \longrightarrow \tau\ p \subseteq \tau\ q\ )$

lemma $\mathit{monotoneE}$: $\mathit{monotone}\ \tau \Longrightarrow p \subseteq q \Longrightarrow \tau\ p \subseteq \tau\ q$
by $(\mathit{simp\ add}: \mathit{monotone\text{-}def})$

lemma $\mathit{lfp1}$: $\mathit{monotone}\ \tau \longrightarrow (\mathit{lfp}\ \tau = \bigcap\ \{Z.\ \tau\ Z \subseteq Z\})$
by $(\mathit{simp\ add}: \mathit{monotone\text{-}def}\ \mathit{lfp\text{-}def})$

lemma $\mathit{gfp1}$: $\mathit{monotone}\ \tau \longrightarrow (\mathit{gfp}\ \tau = \bigcup\ \{Z.\ Z \subseteq \tau\ Z\})$
by $(\mathit{simp\ add}: \mathit{monotone\text{-}def}\ \mathit{gfp\text{-}def})$

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

lemma $\mathit{predtrans\text{-}empty}$:
  assumes $\mathit{monotone}\ \tau$
  shows $\forall\ i.\ (\tau\ \hat{}\ i)\ (\{\}) \subseteq (\tau\ \hat{}(i + 1))(\{\})$
proof $(\mathit{rule\ allI},\ \mathit{induct\text{-}tac}\ i)$
  show $(\tau\ \hat{}\ 0{::}\mathit{nat})\ \{\} \subseteq (\tau\ \hat{}\ (0{::}\mathit{nat}) + (1{::}\mathit{nat}))\ \{\}$ **by** *simp*
next show $\bigwedge(i{::}\mathit{nat})\ n{::}\mathit{nat}.\ (\tau\ \hat{}\ n)\ \{\} \subseteq (\tau\ \hat{}\ n + (1{::}\mathit{nat}))\ \{\}$
    $\Longrightarrow (\tau\ \hat{}\ \mathit{Suc}\ n)\ \{\} \subseteq (\tau\ \hat{}\ \mathit{Suc}\ n + (1{::}\mathit{nat}))\ \{\}$
  proof −
    fix $i\ n$
    assume $a:\ (\tau\ \hat{}\ n)\ \{\} \subseteq (\tau\ \hat{}\ n + (1{::}\mathit{nat}))\ \{\}$
    have $(\tau\ ((\tau\ \hat{}\ n)\ \{\})) \subseteq (\tau\ ((\tau\ \hat{}\ (n + (1\ {::}\ \mathit{nat})))\ \{\}))$ **using** *assms*
      apply $(\mathit{rule\ monotoneE})$
      by $(\mathit{rule}\ a)$
    thus $(\tau\ \hat{}\ \mathit{Suc}\ n)\ \{\} \subseteq (\tau\ \hat{}\ \mathit{Suc}\ n + (1{::}\mathit{nat}))\ \{\}$ **by** *simp*
  qed

```
qed

lemma ex-card: finite S ⟹ ∃ n:: nat. card S = n
by simp

lemma less-not-le: ⟦(x:: nat) < y; y ≤ x⟧ ⟹ False
by arith

lemma infchain-outruns-all:
  assumes finite (UNIV :: 'a set)
    and ∀ i :: nat. (τ ^ i) ({}:: 'a set) ⊂ (τ ^ i + (1 :: nat)) {}
  shows ∀ j :: nat. ∃ i :: nat. j < card ((τ ^ i) {})
proof (rule allI, induct-tac j)
  show ∃ i::nat. (0::nat) < card ((τ ^ i) {}) using assms
    apply (drule-tac x = 0 in spec)
    apply (rule-tac x = 1 in exI)
    apply simp
    apply (subgoal-tac card {} = 0)
    apply (erule subst)
    apply (rule psubset-card-mono)
    apply (rule-tac B = UNIV in finite-subset)
    apply simp
    apply assumption+
      by simp
  next show ⋀(j::nat) n::nat. ∃ i::nat. n < card ((τ ^ i) {})
          ⟹ ∃ i::nat. Suc n < card ((τ ^ i) {})
    proof −
      fix j n
      assume a: ∃ i::nat. n < card ((τ ^ i) {})
      obtain i where n < card ((τ ^ (i :: nat)) {})
        apply (rule exE)
         apply (rule a)
        by simp
      thus ∃ i. Suc n < card ((τ ^ i) {}) using assms
        apply (rule-tac x = i + 1 in exI)
        apply (subgoal-tac card((τ ^ i) {}) < card((τ ^ i + (1 :: nat)) {}))
        apply arith
        apply (rule psubset-card-mono)
        apply (rule-tac B = UNIV in finite-subset)
        apply simp
        apply (rule assms)
        by (erule spec)
    qed
  qed

lemma no-infinite-subset-chain:
   assumes finite (UNIV :: 'a set)
    and     monotone (τ :: ('a set ⇒ 'a set))
    and     ∀ i :: nat. ((τ :: 'a set ⇒ 'a set) ^ i) {} ⊂ (τ ^ i + (1 :: nat)) ({} :: 'a
```

set)
  shows *False*

Proof 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

proof −
  have *a*: $\forall\ (j :: nat).\ (\exists\ (i :: nat).\ (j :: nat) < card((\tau\ \hat{}\ i)(\{\} :: 'a\ set)))$ **using** *assms*
    apply (*erule-tac* $\tau = \tau$ **in** *infchain-outruns-all*)
    by *assumption*
  hence *b*: $\exists\ (n :: nat).\ card(UNIV :: 'a\ set) = n$ **using** *assms*
    by (*erule-tac* $S = UNIV$ **in** *ex-card*)
  from *this* **obtain** *n* **where** *c*: $card(UNIV :: 'a\ set) = n$ **by** (*erule exE*)
  hence *d*: $\exists\, i{::}nat.\ card\ UNIV < card\ ((\tau\ \hat{}\ i)\ \{\})$ **using** *a*
    apply (*drule-tac* $x = card\ UNIV$ **in** *spec*)
    by *assumption*
  from *this* **obtain** *i* **where** *e*: $card\ (UNIV :: 'a\ set) < card\ ((\tau\ \hat{}\ i)\ \{\})$
    by (*erule exE*)
  hence *f*: $(card((\tau\ \hat{}\ i)\{\})) \leq (card\ (UNIV :: 'a\ set))$ **using** *assms*
    thm *Finite-Set.card-mono*
      apply (*rule-tac* $A = ((\tau\ \hat{}\ i)\{\})$ **in** *Finite-Set.card-mono*)
      apply *assumption*
    by (*rule subset-UNIV*)
  thus *False* **using** *e*
    thm *less-not-le*
    apply (*erule-tac* $y = card((\tau\ \hat{}\ i)\{\})$ **in** *less-not-le*)
    by *assumption*
qed

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

Proof 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*.

proof −
  have *a*: $\forall\, i{::}nat.\ (\tau\ \hat{}\ i)\ (\{\}{::}\ 'a\ set) \subseteq (\tau\ \hat{}\ i + (1{::}nat))\ \{\}$

```
    thm predtrans-empty
    apply(rule predtrans-empty)
    by (rule assms(2))
  hence b: (∃ i :: nat. ¬((τ ^ i) {} ⊂ (τ ^(i + 1)) {})) using assms
    apply (subgoal-tac ¬ (∀ i :: nat. (τ ^ i) {} ⊂ (τ ^(i + 1)) {}))
    apply blast
    apply (rule notI)
    apply (rule no-infinite-subset-chain)
    by assumption
  thus ∃ i. (τ ^ i) ({}) = (τ ^(i + 1))({}) using a
    by blast
qed

lemma predtrans-UNIV:
  assumes monotone τ
  shows ∀ i. (τ ^ i) (UNIV) ⊇ (τ ^(i + 1))(UNIV)
proof (rule allI, induct-tac i)
  show (τ ^ (0::nat) + (1::nat)) UNIV ⊆ (τ ^ 0::nat) UNIV by simp
next show ⋀(i::nat) n::nat.
      (τ ^ n + (1::nat)) UNIV ⊆ (τ ^ n) UNIV ⟹ (τ ^ Suc n + (1::nat)) UNIV
⊆ (τ ^ Suc n) UNIV
  proof −
    fix i n
    assume a: (τ ^ n + (1::nat)) UNIV ⊆ (τ ^ n) UNIV
    have (τ ((τ ^ n) UNIV)) ⊇ (τ ((τ ^ (n + (1 :: nat))) UNIV)) using assms
      apply (rule monotoneE)
      by (rule a)
    thus (τ ^ Suc n + (1::nat)) UNIV ⊆ (τ ^ Suc n) UNIV by simp
   qed
 qed

lemma Suc-less-le: x < (y − n) ⟹ x ≤ (y − (Suc n))
 by simp

lemma card-univ-subtract:
  assumes finite (UNIV :: 'a set) and  monotone (τ :: 'a set ⇒ 'a set)
     and  (∀i :: nat. ((τ :: 'a set ⇒ 'a set) ^ i + (1 :: nat)) (UNIV :: 'a set) ⊂
(τ ^ i) UNIV)
   shows (∀ i :: nat. card((τ ^ i) (UNIV ::'a set)) ≤ (card (UNIV :: 'a set)) − i)
proof (rule allI, induct-tac i)
  show card ((τ ^ 0::nat) UNIV) ≤ card (UNIV :: 'a set) − (0::nat) using assms
    by (simp)
next show ⋀(i::nat) n::nat.
       card ((τ ^ n) (UNIV :: 'a set)) ≤ card (UNIV :: 'a set) − n ⟹
       card ((τ ^ Suc n) (UNIV :: 'a set)) ≤ card (UNIV :: 'a set) − Suc n using
assms
  proof −
    fix i n
    assume a: card ((τ ^ n) (UNIV :: 'a set)) ≤ card (UNIV :: 'a set) − n
    have b: (τ ^ n + (1::nat)) (UNIV :: 'a set) ⊂ (τ ^ n) UNIV using assms
      by (erule-tac x = n in spec)
    have card((τ ^ n + (1 :: nat)) (UNIV :: 'a set)) < card((τ ^ n) (UNIV :: 'a
set))
      apply (rule psubset-card-mono)
      apply (rule finite-subset)
      apply (rule subset-UNIV)
       apply (rule assms(1))
      by (rule b)
    thus card ((τ ^ Suc n) (UNIV :: 'a set)) ≤ card (UNIV :: 'a set) − Suc n
using a
      by simp
    qed
  qed

lemma card-UNIV-tau-i-below-zero:
  assumes finite (UNIV :: 'a set) and monotone (τ :: 'a set ⇒ 'a set)
   and (∀ i :: nat. ((τ :: 'a set ⇒ 'a set) ^ i + (1 :: nat)) (UNIV :: 'a set) ⊂ (τ
^ i) UNIV)
 shows card((τ ^ (card (UNIV ::'a set))) (UNIV ::'a set)) ≤ 0
proof −
  have (∀ i :: nat. card((τ ^ i) (UNIV ::'a set)) ≤ (card (UNIV :: 'a set)) − i)
using assms
    by (rule card-univ-subtract)
  thus card((τ ^ (card (UNIV ::'a set))) (UNIV ::'a set)) ≤ 0
   apply (drule-tac x = card (UNIV ::'a set) in spec)
    by simp
qed

lemma finite-card-zero-empty: ⟦ finite S; card S ≤ 0⟧ ⟹ S = {}
by simp

lemma UNIV-tau-i-is-empty:
  assumes finite (UNIV :: 'a set) and monotone (τ :: 'a set ⇒ 'a set)
    and (∀ i :: nat. ((τ :: 'a set ⇒ 'a set) ^ i + (1 :: nat)) (UNIV :: 'a set) ⊂
(τ ^ i) UNIV)
  shows (τ ^ (card (UNIV ::'a set))) (UNIV ::'a set) = {}
proof −
  have card ((τ ^ card (UNIV ::'a set)) UNIV) ≤ (0::nat) using assms
    apply (rule card-UNIV-tau-i-below-zero)
    .
  thus (τ ^ (card (UNIV ::'a set))) (UNIV ::'a set) = {} using assms
   apply (rule-tac S = (τ ^ (card (UNIV ::'a set))) (UNIV ::'a set) in finite-card-zero-empty)
    apply (rule finite-subset)
    apply (rule subset-UNIV)
    .
qed

lemma down-chain-reaches-empty:
  assumes finite (UNIV :: 'a set) and monotone (τ :: 'a set ⇒ 'a set)
   and (∀ i :: nat. ((τ :: 'a set ⇒ 'a set) ^ i + (1 :: nat)) UNIV ⊂ (τ ^ i) UNIV)
 shows ∃ (j :: nat). (τ ^ j) UNIV = {}
proof −
  have (τ ^ ((card (UNIV :: 'a set)))) UNIV = {} using assms
    apply (rule UNIV-tau-i-is-empty)
    .
  thus ∃ (j :: nat). (τ ^ j) UNIV = {}
    by (rule exI)
qed

lemma no-infinite-subset-chain2:
  assumes finite (UNIV :: 'a set) and monotone (τ :: ('a set ⇒ 'a set))
      and ∀ i :: nat. (τ ^ i) UNIV ⊃ (τ ^ i + (1 :: nat)) UNIV
  shows False
proof −
  have ∃ j :: nat. (τ ^ j) UNIV = {} using assms
    apply (rule down-chain-reaches-empty)
    .
  from this obtain j where a: (τ ^ j) UNIV = {} by (erule exE)
  have (τ ^ j + (1::nat)) UNIV ⊂ (τ ^ j) UNIV using assms
    by (erule-tac x = j in spec)
  thus False using a by simp
qed

lemma finite-fixp2:
  assumes finite(UNIV :: 'a set) and monotone (τ :: ('a set ⇒ 'a set))
  shows ∃ i. (τ ^ i) UNIV = (τ ^(i + 1)) UNIV
proof −
  have ∀ i::nat. (τ ^ i + (1::nat)) UNIV ⊆ (τ ^ i) UNIV
    apply (rule predtrans-UNIV) using assms
    by (simp add: assms(2))
   moreover have ∃ i::nat. ¬ (τ ^ i + (1::nat)) UNIV ⊂ (τ ^ i) UNIV using
assms
  proof −
    have ¬ (∀ i :: nat. (τ ^ i) UNIV ⊃ (τ ^(i + 1)) UNIV)
      apply (rule notI)
      apply (rule no-infinite-subset-chain2) using assms
      .
    thus ∃ i::nat. ¬ (τ ^ i + (1::nat)) UNIV ⊂ (τ ^ i) UNIV by blast
  qed
  ultimately show ∃ i. (τ ^ i) UNIV = (τ ^(i + 1)) UNIV
    by blast
qed

lemma mono-monotone: mono (τ :: ('a set ⇒ 'a set)) ⟹ monotone τ
by (simp add: monotone-def mono-def)

lemma monotone-mono: monotone (τ :: ('a set ⇒ 'a set)) ⟹ mono τ
```

```
by (simp add: monotone-def mono-def)

lemma power-power: ((τ :: ('a set ⇒ 'a set)) ^^ n) = ((τ :: ('a set ⇒ 'a set)) ^
n)
proof (induct-tac n)
  show τ ^^ (0::nat) = (τ ^ 0::nat) by (simp add: id-def)
next show ⋀n::nat. τ ^^ n = (τ ^ n) ⟹ τ ^^ Suc n = (τ ^ Suc n)
    by simp
qed

lemma lfp-Kleene-iter-set: monotone (f :: ('a set ⇒ 'a set)) ⟹
   (f ^ Suc(n)) {} = (f ^ n) {} ⟹ lfp f = (f ^ n){}
by (simp add: monotone-mono lfp-Kleene-iter power-power)

lemma lfp-loop:
  assumes finite (UNIV :: 'b set) and monotone (τ :: ('b set ⇒ 'b set))
  shows ∃ n . lfp τ = (τ ^ n) {}
proof −
  have ∃ i::nat. (τ ^ i) {} = (τ ^ i + (1::nat)) {} using assms
    by (rule finite-fixp)
  from this obtain i where (τ ^ i) {} = (τ ^ i + (1::nat)) {}
    by (erule exE)
  hence (τ ^ i) {} = (τ ^ Suc i) {}
    by simp
  hence (τ ^ Suc i) {} = (τ ^ i) {}
    by (rule sym)
  hence lfp τ = (τ ^ i) {}
     by (simp add: assms(2) lfp-Kleene-iter-set)
   thus ∃ n . lfp τ = (τ ^ n) {}
   by (rule exI)
qed
```

These next two are repeated from the corresponding theorems in HOL/ZF/Nat.thy for the sake of self-containedness of the exposition.

```
lemma Kleene-iter-gpfp:
  assumes mono f and p ≤ f p shows p ≤ (f^^k) (top::'a::order-top)
proof(induction k)
  case 0 show ?case by simp
next
  case Suc
  from monoD[OF assms(1) Suc] assms(2)
  show ?case by simp
qed

lemma gfp-Kleene-iter: assumes mono f and (f^^Suc k) top = (f^^k) top
shows gfp f = (f^^k) top
proof(rule antisym)
  show (f^^k) top ≤ gfp f
  proof(rule gfp-upperbound)
```

```
    show (f^^k) top ≤ f ((f^^k) top)  using assms(2) by simp
  qed
next
  show gfp f ≤ (f^^k) top
    using Kleene-iter-gpfp[OF assms(1)] gfp-unfold[OF assms(1)] by simp
qed

lemma gfp-Kleene-iter-set:
  assumes monotone (f :: ('a set ⇒ 'a set))
      and (f ^ Suc(n)) UNIV = (f ^ n) UNIV
    shows gfp f  = (f ^ n) UNIV
proof −
  have a: mono f using assms
    by (erule-tac τ = f in monotone-mono)
  hence b: (f ^^ Suc (n)) UNIV = (f ^^ n) UNIV using assms
    by (simp add: power-power)
  hence c: gfp f = (f ^^ (n))(UNIV :: 'a set) using assms a
    thm gfp-Kleene-iter
    apply (erule-tac f = f and k = n in gfp-Kleene-iter)
    .
  thus gfp f = (f ^ (n))(UNIV :: 'a set) using assms a
    by (simp add: power-power)
qed

lemma gfp-loop:
  assumes finite (UNIV :: 'b set)
   and monotone (τ :: ('b set ⇒ 'b set))
    shows ∃ n . gfp τ  = (τ ^ n)(UNIV :: 'b set)
proof −
  have ∃ i::nat. (τ ^ i)(UNIV :: 'b set) = (τ ^ i + (1::nat)) UNIV using assms
    by (rule finite-fixp2)
  from this obtain i where (τ ^ i)(UNIV :: 'b set) = (τ ^ i + (1::nat)) UNIV
by (erule exE)
  thus ∃ n . gfp τ  = (τ ^ n)(UNIV :: 'b set) using assms
    apply (rule-tac x = i in exI)
    apply (rule gfp-Kleene-iter-set)
    apply assumption
    apply (rule sym)
    by simp
qed
```

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 $I \to_i I'$ to denote that system state I and I' are in this relation over an arbitrary (polymorphic) type $'a$.

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

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

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 structures 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*

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

1.4 Lemmas for CTL operators

1.4.1 EF lemmas

```
lemma EF-lem0: (x ∈ EF f) = (x ∈ f ∪ EX' (lfp (λZ :: ('a :: state) set. f ∪
EX' Z)))
proof −
  have lfp (λZ :: ('a :: state) set. f ∪ EX' Z) =
                    f ∪ (EX' (lfp (λZ :: 'a set. f ∪ EX' Z)))
    apply (rule def-lfp-unfold)
    apply (rule reflexive)
    apply (unfold mono-def EX'-def)
    by auto
  thus (x ∈ EF (f :: ('a :: state) set)) = (x ∈ f ∪ EX' (lfp (λZ :: ('a :: state)
set. f ∪ EX' Z)))
    by (simp add: EF-def)
qed

lemma EF-lem00: (EF f) = (f ∪ EX' (lfp (λ Z :: ('a :: state) set. f ∪ EX' Z)))
proof (rule equalityI)
  show EF f ⊆ f ∪ EX' (lfp (λZ::'a set. f ∪ EX' Z))
   apply (rule subsetI)
   by (simp add: EF-lem0)
  next show f ∪ EX' (lfp (λZ::'a set. f ∪ EX' Z)) ⊆ EF f
   apply (rule subsetI)
   by (simp add: EF-lem0)
 qed

lemma EF-lem000: (EF f) = (f ∪ EX' (EF f))
proof (subst EF-lem00)
  show f ∪ EX' (lfp (λZ::'a set. f ∪ EX' Z)) = f ∪ EX' (EF f)
    apply (fold EF-def)
    by (rule refl)
qed

lemma EF-lem1: x ∈ f ∨ x ∈ (EX' (EF f)) ⟹ x ∈ EF f
proof (simp add: EF-def)
  assume a: x ∈ f ∨ x ∈ EX' (lfp (λZ::'a set. f ∪ EX' Z))
  show x ∈ lfp (λZ::'a set. f ∪ EX' Z)
  proof −
    have b: lfp (λZ :: ('a :: state) set. f ∪ EX' Z) =
                    f ∪ (EX' (lfp (λZ :: ('a :: state) set. f ∪ EX' Z)))
      apply (rule def-lfp-unfold)
      apply (rule reflexive)
      apply (unfold mono-def EX'-def)
      by auto
    thus x ∈ lfp (λZ::'a set. f ∪ EX' Z) using a
     apply (subst b)
     by blast
 qed
```

qed

lemma *EF-lem2b*:
    assumes $x \in (EX'\ (EF\ f))$
   shows $x \in EF\ f$
proof (*rule EF-lem1*)
  show $x \in f \lor x \in EX'\ (EF\ f)$
    apply (*rule disjI2*)
    by (*rule assms*)
qed

lemma *EF-lem2a*: **assumes** $x \in f$ **shows** $x \in EF\ f$
proof (*rule EF-lem1*)
  show $x \in f \lor x \in EX'\ (EF\ f)$
    apply (*rule disjI1*)
    by (*rule assms*)
qed

lemma *EF-lem2c*: **assumes** $x \notin f$ **shows** $x \in EF\ (-\ f)$
proof −
  have $x \in (-\ f)$ **using** *assms*
    by *simp*
  thus $x \in EF\ (-\ f)$
    by (*rule EF-lem2a*)
qed

lemma *EF-lem2d*: **assumes** $x \notin EF\ f$ **shows** $x \notin f$
proof −
  have $x \in f \Longrightarrow x \in EF\ f$
    by (*erule EF-lem2a*)
  thus $x \notin f$ **using** *assms*
    thm *contrapos-nn*
    apply (*erule-tac* $P = x \in f$ **in** *contrapos-nn*)
    apply (*erule meta-mp*)
    .
qed

lemma *EF-lem3b*: **assumes** $x \in EX'\ (f \cup EX'\ (EF\ f))$ **shows** $x \in (EF\ f)$
proof (*simp add*: *EF-lem0*)
  show $x \in f \lor x \in EX'\ (lfp\ (\lambda Z{::}'a\ set.\ f \cup EX'\ Z))$
   apply (*rule disjI2*)
   apply (*fold EF-def*)
   apply (*subst EF-lem00*)
   apply (*fold EF-def*)
   by (*rule assms*)
qed

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

show $x \in \{s{::}'a.\ \exists f0{::}'a{\in}f.\ s \to_i f0\} \Longrightarrow x \in \{s{::}'a.\ \exists f0{::}'a{\in}f \cup g.\ s \to_i f0\}$
by *blast*
qed

lemma *EX-lem0r*: $x \in (EX'\ g) \Longrightarrow x \in (EX'\ (f \cup g))$
proof (*unfold EX'-def*)
show $x \in \{s{::}'a.\ \exists f0{::}'a{\in}g.\ s \to_i f0\} \Longrightarrow x \in \{s{::}'a.\ \exists f0{::}'a{\in}f \cup g.\ s \to_i f0\}$
by *blast*
qed

lemma *EX-step*: **assumes** $x \to_i y$ **and** $y \in f$ **shows** $x \in EX'\ f$
proof (*unfold EX'-def*)
show $x \in \{s{::}'a.\ \exists f0{::}'a{\in}f.\ s \to_i f0\}$
apply *simp*
apply (*rule-tac* $x = y$ **in** *bexI*)
by (*rule assms*)+
qed

lemma *EF-E*[*rule-format*]: $\forall\ f.\ x \in (EF\ (f :: ('a :: state)\ set)) \longrightarrow x \in (f \cup EX'\ (EF\ f))$
proof −
have *a*: $\bigwedge f{::}'a\ set.\ EF\ (f :: ('a :: state)\ set) = f \cup EX'\ (EF\ f)$
by (*rule EF-lem000*)
thus $(\forall\ f.\ x \in EF\ (f :: ('a :: state)\ set) \longrightarrow x \in f \cup EX'\ (EF\ f))$
apply (*rule-tac* $P = (\lambda\ f.\ x \in EF\ (f :: ('a :: state)\ set) \longrightarrow x \in f \cup EX'\ (EF\ f))$ **in** *allI*)
apply (*subst a*)
apply (*rule impI*)
by *assumption*
qed

lemma *EF-step*: **assumes** $x \to_i y$ **and** $y \in f$ **shows** $x \in EF\ f$
proof (*rule EF-lem3b*)
show $x \in EX'\ (f \cup EX'\ (EF\ f))$
apply (*rule EX-step*)
apply (*rule assms(1)*)
by (*simp add*: *assms(2)*)
qed

lemma *EF-step-step*: **assumes** $x \to_i y$ **and** $y \in EF\ f$ **shows** $x \in EF\ f$
proof −
have $y \in f \cup EX'\ (EF\ f)$
apply (*rule EF-E*)
by (*rule assms(2)*)
thus $x \in EF\ f$
apply (*rule-tac* $x = x$ **and** $f = f$ **in** *EF-lem3b*)
apply (*rule EX-step*)
by (*rule assms*)
qed

```
lemma EF-step-star: ⟦ x →ᵢ∗ y; y ∈ f ⟧ ⟹ x ∈ EF f
proof (simp add: state-transition-refl-def)
  show (x, y) ∈ {(x::'a, y::'a). x →ᵢ y}* ⟹ y ∈ f ⟹ x ∈ EF f
  proof (erule converse-rtrancl-induct)
    show y ∈ f ⟹ y ∈ EF f
      by (erule EF-lem2a)
    next show ⋀(ya::'a) z::'a. y ∈ f ⟹
                (ya, z) ∈ {(x::'a, y::'a). x →ᵢ y} ⟹
                (z, y) ∈ {(x::'a, y::'a). x →ᵢ y}* ⟹ z ∈ EF f ⟹ ya ∈ EF f
        apply (clarify)
        apply (erule EF-step-step)
        by assumption
    qed
  qed

lemma EF-induct-prep:
  assumes (a::'a::state) ∈ lfp (λ Z. (f::'a::state set) ∪ EX' Z)
      and mono  (λ Z. (f::'a::state set) ∪ EX' Z)
     shows (⋀x::'a::state.
     x ∈ ((λ Z. (f::'a::state set) ∪ EX' Z)(lfp (λ Z. (f::'a::state set) ∪ EX' Z) ∩
{x::'a::state. (P::'a::state ⇒ bool) x})) ⟹ P x) ⟹
      P a
proof −
  show (⋀x::'a::state.
     x ∈ ((λ Z. (f::'a::state set) ∪ EX' Z)(lfp (λ Z. (f::'a::state set) ∪ EX' Z) ∩
{x::'a::state. (P::'a::state ⇒ bool) x})) ⟹ P x) ⟹
      P a
    apply (rule-tac A = EF f in def-lfp-induct-set)
    apply (rule EF-def)
    apply (rule assms(2))
    by (simp add: EF-def assms)+
qed

lemma EF-induct: (a::'a::state) ∈ EF (f :: 'a :: state set) ⟹
    mono  (λ Z. (f::'a::state set) ∪ EX' Z) ⟹
    (⋀x::'a::state.
       x ∈ ((λ Z. (f::'a::state set) ∪ EX' Z)(EF f ∩ {x::'a::state. (P::'a::state ⇒
bool) x})) ⟹ P x) ⟹
    P a
proof (simp add: EF-def)
  show a ∈ lfp (λZ::'a set. f ∪ EX' Z) ⟹
    mono (λZ::'a set. f ∪ EX' Z) ⟹
    (⋀x::'a. x ∈ f ∨ x ∈ EX' (lfp (λZ::'a set. f ∪ EX' Z) ∩ Collect P) ⟹ P x)
⟹ P a
    apply (erule EF-induct-prep)
    apply assumption
  by simp
qed
```

lemma *valEF-E*: $M \vdash EF\ f \Longrightarrow x \in init\ M \Longrightarrow x \in EF\ f$
proof (*simp add*: *check-def*)
  show $init\ M \subseteq \{s::'a \in states\ M.\ s \in EF\ f\} \Longrightarrow x \in init\ M \Longrightarrow x \in EF\ f$
   apply (*drule subsetD*)
   apply *assumption*
    by *simp*
qed

lemma *EF-step-star-rev*[*rule-format*]: $x \in EF\ s \Longrightarrow (\exists\ y \in s.\ x \rightarrow_i{*}\ y)$
proof (*erule EF-induct*)
  show $mono\ (\lambda Z::'a\ set.\ s \cup EX'\ Z)$
    apply (*simp add*: *mono-def EX'-def*)
    by *force*
next show $\bigwedge x::'a.\ x \in s \cup EX'\ (EF\ s \cap \{x::'a.\ \exists\, y::'a{\in}s.\ x \rightarrow_i{*}\ y\}) \Longrightarrow \exists\, y::'a{\in}s.$
$x \rightarrow_i{*}\ y$
apply (*erule UnE*)
   apply (*rule-tac* $x = x$ **in** *bexI*)
    apply (*simp add*: *state-transition-refl-def*)
   apply *assumption*
  apply (*simp add*: *EX'-def*)
  apply (*erule bexE*)
  apply (*erule IntE*)
  apply (*drule CollectD*)
  apply (*erule bexE*)
  apply (*rule-tac* $x = xb$ **in** *bexI*)
   apply (*simp add*: *state-transition-refl-def*)
   apply (*rule rtrancl-trans*)
    apply (*rule r-into-rtrancl*)
    apply (*rule CollectI*)
    apply *simp*
  by *assumption+*
qed

lemma *EF-step-inv*: $(I \subseteq \{sa::'s :: state.\ (\exists\, i::'s{\in}I.\ i \rightarrow_i{*}\ sa) \wedge sa \in EF\ s\})$
        $\Longrightarrow \forall\ x \in I.\ \exists\ y \in s.\ x \rightarrow_i{*}\ y$
proof (*clarify*)
  show $\bigwedge x::'s.\ I \subseteq \{sa::'s.\ (\exists\, i::'s{\in}I.\ i \rightarrow_i{*}\ sa) \wedge sa \in EF\ s\} \Longrightarrow x \in I \Longrightarrow$
$\exists\, y::'s{\in}s.\ x \rightarrow_i{*}\ y$
    apply (*drule subsetD*)
    apply *assumption*
    apply (*drule CollectD*)
    apply (*erule conjE*)
    by (*erule EF-step-star-rev*)
qed

1.4.2 AG lemmas

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

proof (*simp add*: *AG-def gfp-def*)
  show $\exists\, xa \subseteq s.\ xa \subseteq AX\ xa \wedge x \in xa \Longrightarrow x \in s$
    apply (*erule exE*)
    apply (*erule conjE*)+
    by (*erule subsetD*, *assumption*)
qed

lemma *AG-lem1*: $x \in s \wedge x \in (AX\ (AG\ s)) \Longrightarrow x \in AG\ s$
proof (*simp add*: *AG-def*)
  show $x \in s \wedge x \in AX\ (gfp\ (\lambda Z{::}'a\ set.\ s \cap AX\ Z)) \Longrightarrow x \in gfp\ (\lambda Z{::}'a\ set.\ s \cap AX\ Z)$
  apply (*subgoal-tac* $gfp\ (\lambda Z{::}'a\ set.\ s \cap AX\ Z) =$
                    $s \cap (AX\ (gfp\ (\lambda Z{::}'a\ set.\ s \cap AX\ Z))))$
  apply (*erule ssubst*)
  apply *simp*
  apply (*rule def-gfp-unfold*)
  apply (*rule reflexive*)
  apply (*unfold mono-def AX-def*)
  by *auto*
qed

lemma *AG-lem2*: $x \in AG\ s \Longrightarrow x \in (s \cap (AX\ (AG\ s)))$
proof −
  have *a*: $AG\ s = s \cap (AX\ (AG\ s))$
    apply (*simp add*: *AG-def*)
    apply (*rule def-gfp-unfold*)
    apply (*rule reflexive*)
    apply (*unfold mono-def AX-def*)
    by *auto*
  thus $x \in AG\ s \Longrightarrow x \in (s \cap (AX\ (AG\ s)))$
   by (*erule subst*)
qed

lemma *AG-lem3*: $AG\ s = (s \cap (AX\ (AG\ s)))$
proof (*rule equalityI*)
  show $AG\ s \subseteq s \cap AX\ (AG\ s)$
    apply (*rule subsetI*)
    by (*erule AG-lem2*)
  next show $s \cap AX\ (AG\ s) \subseteq AG\ s$
    apply (*rule subsetI*)
    apply (*rule AG-lem1*)
    by *simp*
qed

lemma *AG-step*: $y \to_i z \Longrightarrow y \in AG\ s \Longrightarrow z \in AG\ s$
proof (*drule AG-lem2*)
  show $y \to_i z \Longrightarrow y \in s \cap AX\ (AG\ s) \Longrightarrow z \in AG\ s$
    apply (*erule IntE*)
    apply (*unfold AX-def*)

```
    apply simp
    apply (erule subsetD)
    by simp
qed

lemma AG-all-s: x →i* y ⟹ x ∈ AG s ⟹ y ∈ AG s
proof (simp add: state-transition-refl-def)
  show (x, y) ∈ {(x::'a, y::'a). x →i y}* ⟹ x ∈ AG s ⟹ y ∈ AG s
    apply (erule rtrancl-induct)
  proof -
    show x ∈ AG s ⟹ x ∈ AG s by assumption
  next show ⋀(y::'a) z::'a.
       x ∈ AG s ⟹
       (x, y) ∈ {(x::'a, y::'a). x →i y}* ⟹
       (y, z) ∈ {(x::'a, y::'a). x →i y} ⟹ y ∈ AG s ⟹ z ∈ AG s
      apply clarify
      by (erule AG-step, assumption)
  qed
qed

lemma AG-imp-notnotEF:
I ≠ {} ⟹ ((Kripke {s :: ('s :: state). ∃ i ∈ I. (i →i* s)} (I :: ('s :: state)set)
⊢ AG s)) ⟹
 (¬(Kripke {s :: ('s :: state). ∃ i ∈ I. (i →i* s)} (I :: ('s :: state)set) ⊢ EF (−
s)))
proof (rule notI, simp add: check-def)
  assume a0: I ≠ {} and
    a1: I ⊆ {sa::'s. (∃ i::'s∈I. i →i* sa) ∧ sa ∈ AG s} and
    a2: I ⊆ {sa::'s. (∃ i::'s∈I. i →i* sa) ∧ sa ∈ EF (− s)}
  show False
  proof -
    have a3: {sa::'s. (∃ i::'s∈I. i →i* sa) ∧ sa ∈ AG s} ∩
                     {sa::'s. (∃ i::'s∈I. i →i* sa) ∧ sa ∈ EF (− s)} = {}
      proof -
        have (? x. x ∈ {sa::'s. (∃ i::'s∈I. i →i* sa) ∧ sa ∈ AG s} ∧
                           x ∈ {sa::'s. (∃ i::'s∈I. i →i* sa) ∧ sa ∈ EF (− s)}) ⟹
False
        proof -
          assume a4: (? x. x ∈ {sa::'s. (∃ i::'s∈I. i →i* sa) ∧ sa ∈ AG s} ∧
                           x ∈ {sa::'s. (∃ i::'s∈I. i →i* sa) ∧ sa ∈ EF (− s)})
            from a4 obtain x where  a5: x ∈ {sa::'s. (∃ i::'s∈I. i →i* sa) ∧ sa ∈
AG s} ∧
                              x ∈ {sa::'s. (∃ i::'s∈I. i →i* sa) ∧ sa ∈ EF (− s)}
            by (erule exE)
            hence x ∈ s ∧ x ∈ −s
            proof -
              have a6: x ∈ s using a5
                apply (subgoal-tac x ∈ AG s)
                apply (erule AG-in-lem)
            by simp
          moreover have $x \in -s$ using a5
          proof -
            have $x \in EF\ s$
              apply (rule-tac $y = x$ in EF-step-star)
              apply (simp add: state-transition-refl-def)
              by (rule a6)
            thus $x \in -s$ using a5
            proof -
              have $x \in EF\ (-\ s)$ using a5
                by simp
              moreover from this obtain y where a7: $y \in -\ s \wedge x \rightarrow_i * y$
                apply (rotate-tac -1)
                 apply (drule EF-step-star-rev)
                by blast
              moreover have $y \in AG\ s$ using a7 a5
                apply (subgoal-tac $x \in AG\ s$)
                apply (erule conjE)
                 apply (drule AG-all-s)
                  apply assumption+
                by simp
              ultimately show $x \in -s$ using a5
                 apply (rotate-tac -1)
                 apply (drule AG-in-lem)
                 by blast
            qed
          qed
          ultimately show $x \in s \wedge x \in -s$
            by (rule conjI)
        qed
        thus False
          by blast
      qed
    thus $\{sa::'s.\ (\exists\, i::'s{\in}I.\ i \rightarrow_i * sa) \wedge sa \in AG\ s\} \cap$
                $\{sa::'s.\ (\exists\, i::'s{\in}I.\ i \rightarrow_i * sa) \wedge sa \in EF\ (-\ s)\} = \{\}$
      by blast
    qed
  moreover have b: ? x. x : I using a0
    by blast
  moreover obtain x where $x \in I$
      apply (rule exE)
       apply (rule b)
    by simp
  ultimately show False using a0 a1 a2
    by blast
  qed
qed
```

A simplified way of Modelchecking is given by the following lemma.

lemma *check2-def*: $(Kripke\ S\ I \vdash f) = (I \subseteq S \cap f)$
proof (*simp add*: *check-def*)
  show $(I \subseteq \{s{::}'a \in S.\ s \in f\}) = (I \subseteq S \wedge I \subseteq f)$ **by** *blast*
qed

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* ⇒ *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* ⇒ *bool*) ∗ *action set*)

definition *ID* :: [*actor*, *string*] ⇒ *bool*
where *ID a s* ≡ (*a* = *Actor s*)

datatype *location* = *Location nat*

datatype *igraph* = *Lgraph* (*location* ∗ *location*)*set location* ⇒ *identity list*
                    actor ⇒ (*string list* ∗ *string list*) *location* ⇒ *string list*
datatype *infrastructure* =
    Infrastructure igraph
            [*igraph*, *location*] ⇒ *policy set*

primrec *loc* :: *location* ⇒ *nat*
where *loc*(*Location n*) = *n*
primrec *gra* :: *igraph* ⇒ (*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*] ⇒ *bool*
where *UasI a b* ≡ (*Actor a* = *Actor b*) ∧ (∀ *x y*. *x* ≠ *a* ∧ *y* ≠ *a* ∧ *Actor x* = *Actor y* ⟶ *x* = *y*)

definition *UasI′* :: [*actor* => *bool*, *identity*, *identity*] ⇒ *bool*
where *UasI′ P a b* ≡ *P* (*Actor b*) ⟶ *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* ⇒ *actor-state*] ⇒ *bool*
where *Insider a C as* ≡ (*tipping-point* (*as a*) ⟶ (∀ *b*∈*C*. *UasI a b*))

definition *Insider′* :: [*actor* ⇒ *bool*, *identity*, *identity set*, *identity* ⇒ *actor-state*] ⇒ *bool*
where *Insider′ P a C as* ≡ (*tipping-point* (*as a*) ⟶ (∀ *b*∈*C*. *UasI′ P a b* ∧ *inj-on Actor C*))

definition *atI* :: [*identity*, *igraph*, *location*] ⇒ *bool* (- $@_{(\text{-})}$ - *50*)
where *a* $@_{G}$ *l* ≡ *a* ∈ *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*] ⇒ *bool*
where
enables I l a a′ ≡ (∃ (*p*,*e*) ∈ *delta I* (*graphI I*) *l*. *a′* ∈ *e* ∧ *p a*)

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

definition *behaviour* :: *infrastructure* ⇒ (*location* ∗ *actor* ∗ *action*)*set*
where *behaviour I* ≡ {(*t*,*a*,*a′*). *enables I t a a′*}

misbehaviour is the complement of behaviour

definition *misbehaviour* :: *infrastructure* ⇒ (*location* ∗ *actor* ∗ *action*)*set*
  where *misbehaviour I* ≡ −(*behaviour I*)

basic lemmas for enable

lemma *not-enableI*: (∀ (*p*,*e*) ∈ *delta I* (*graphI I*) *l*. (~(*h* : *e*) | (~(*p*(*a*)))))
            ⟹ ~(*enables I l a h*)
  by (*simp add*: *enables-def*, *blast*)

lemma *not-enableI2*: ⟦⋀ *p e*. (*p*,*e*) ∈ *delta I* (*graphI I*) *l* ⟹
          (~(*t* : *e*) | (~(*p*(*a*)))) ⟧ ⟹ ~(*enables I l a t*)
 by (*rule not-enableI*, *rule ballI*, *auto*)

lemma *not-enableE*: ⟦ ~(*enables I l a t*); (*p*,*e*) ∈ *delta I* (*graphI I*) *l* ⟧

$\Longrightarrow (\sim(t : e) \mid \ (\sim(p(a))))$
by (*simp add*: *enables-def*, *rule impI*, *force*)

lemma *not-enableE2*: $[\![\sim(enables\ I\ l\ a\ t);\ (p,e) \in delta\ I\ (graphI\ I)\ l;$
$t : e\ ]\!] \Longrightarrow (\sim(p(a)))$
by (*simp add*: *enables-def*, *force*)

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: $del\ a\ []\ =\ []\ \mid$
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: $jonce\ a\ []\ =\ False\ \mid$
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: $nodup\ a\ []\ =\ True\ \mid$
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: $[\![\ G = graphI\ I;\ a\ @_G\ l;\ l \in nodes\ G;\ l' \in nodes\ G;$
$(a) \in actors\text{-}graph(graphI\ I);\ enables\ I\ l'\ (Actor\ a)\ move;$
$I' = Infrastructure\ (move\text{-}graph\text{-}a\ a\ l\ l'\ (graphI\ I))(delta\ I)\ ]\!] \Longrightarrow I \rightarrow_n I'$
$\mid get : [\![\ G = graphI\ I;\ a\ @_G\ l;\ a'\ @_G\ l;\ has\ G\ (Actor\ a,\ z);$
$enables\ I\ l\ (Actor\ a)\ get;$
$I' = Infrastructure$
$(Lgraph\ (gra\ G)(agra\ G)$
$((cgra\ G)(Actor\ a' :=$
$(z\ \#\ (fst(cgra\ G\ (Actor\ a'))),\ snd(cgra\ G\ (Actor\ a')))))$
$(lgra\ G))$
$(delta\ I)$
$]\!] \Longrightarrow I \rightarrow_n I'$
$\mid put : [\![\ G = graphI\ I;\ a\ @_G\ l;\ enables\ I\ l\ (Actor\ a)\ put;$
$I' = Infrastructure$

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

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
by (*rule MC.class.MC.state.of-class.intro*)

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

lemma *del-del*[*rule-format*]: $n \in set\ (del\ a\ S) \longrightarrow n \in set\ S$
by (*induct-tac S*, *auto*)

lemma *del-dec*[*rule-format*]: $a \in set\ S \longrightarrow length\ (del\ a\ S) < length\ S$
by (*induct-tac S*, *auto*)

lemma *del-sort*[*rule-format*]: $\forall\ n.\ (Suc\ n\ ::nat) \leq length\ (l) \longrightarrow n \leq length\ (del\ a\ (l))$
by (*induct-tac l*, *simp*, *clarify*, *case-tac n*, *simp*, *simp*)

lemma *del-jonce*: $jonce\ a\ l \longrightarrow a \notin set\ (del\ a\ l)$
by (*induct-tac l*, *auto*)

lemma *del-nodup*[*rule-format*]: $nodup\ a\ l \longrightarrow a \notin set(del\ a\ l)$
by (*induct-tac l*, *auto*)

lemma *nodup-up*[*rule-format*]: $a \in set\ (del\ a\ l) \longrightarrow a \in set\ l$
by (*induct-tac l*, *auto*)

lemma *del-up* [*rule-format*]: $a \in set\ (del\ aa\ l) \longrightarrow a \in set\ l$
by (*induct-tac l*, *auto*)

lemma *nodup-notin*[*rule-format*]: $a \notin set\ list \longrightarrow nodup\ a\ list$
by (*induct-tac list*, *auto*)

lemma *nodup-down*[*rule-format*]: *nodup a l* $\longrightarrow$ *nodup a* (*del a l*)
  by (*induct-tac l*, *simp+*, *clarify*, *erule nodup-notin*)

lemma *del-notin-down*[*rule-format*]: $a \notin$ *set list* $\longrightarrow$ $a \notin$ *set* (*del aa list*)
  by (*induct-tac list*, *auto*)

lemma *del-not-a*[*rule-format*]: $x \neq a \longrightarrow x \in$ *set l* $\longrightarrow x \in$ *set* (*del a l*)
  by (*induct-tac l*, *auto*)

lemma *nodup-down-notin*[*rule-format*]: *nodup a l* $\longrightarrow$ *nodup a* (*del aa l*)
  by (*induct-tac l*, *simp+*, *rule conjI*, *clarify*, *erule nodup-notin*, (*rule impI*)+,
    erule del-notin-down)

lemma *move-graph-eq*: *move-graph-a a l l g* = *g*
  by (*simp add*: *move-graph-a-def*, *case-tac g*, *force*)

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'*)
  by (*clarify*, *erule state-transition-in.cases*, *simp+*)

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 –
  have *ind*: (*x*,*y*) $\in$ {(*x*::*infrastructure*, *y*::*infrastructure*). $x \rightarrow_n y$}*
          $\longrightarrow$ *delta*(*x*) = *delta*(*y*)
  proof (*insert assms*, *erule rtrancl.induct*)
    show ($\bigwedge$ *a*::*infrastructure*.
       ($\forall$ (*z*::*infrastructure*)(*z'*::*infrastructure*). ($z \rightarrow_n z'$) $\longrightarrow$ (*delta z* = *delta z'*))
$\Longrightarrow$
       (((*a*, *a*) $\in$ {(*x* ::*infrastructure*, *y* :: *infrastructure*). $x \rightarrow_n y$}*) $\longrightarrow$
       (*delta a* = *delta a*)))
    by (*rule impI*, *rule refl*)
next fix *a b c*
  assume *a0*: $\forall$ (*z*::*infrastructure*) *z'*::*infrastructure*. $z \rightarrow_n z' \longrightarrow$ *delta z* = *delta*
z'
     and *a1*: (*a*, *b*) $\in$ {(*x*::*infrastructure*, *y*::*infrastructure*). $x \rightarrow_n y$}*
     and *a2*: (*a*, *b*) $\in$ {(*x*::*infrastructure*, *y*::*infrastructure*). $x \rightarrow_n y$}* $\longrightarrow$
         delta a = *delta b*
     and *a3*: (*b*, *c*) $\in$ {(*x*::*infrastructure*, *y*::*infrastructure*). $x \rightarrow_n y$}
     show (*a*, *c*) $\in$ {(*x*::*infrastructure*, *y*::*infrastructure*). $x \rightarrow_n y$}* $\longrightarrow$
       delta a = *delta c*
  proof –
    have *a4*: *delta b* = *delta c* **using** *a0 a1 a2 a3* **by** *simp*
    show *?thesis* **using** *a0 a1 a2 a3* **by** *simp*
  qed
qed

show *?thesis*
  by (*insert ind*, *insert assms*(*2*), *simp*)
qed

lemma *init-state-policy*: ⟦ (*x*,*y*) ∈ {(*x*::*infrastructure*, *y*::*infrastructure*). $x \to_n y$}*
⟧ ⟹
                    delta(*x*) = *delta*(*y*)
  by (*rule init-state-policy0*, *rule delta-invariant*)

lemma *same-nodes0*[*rule-format*]: ∀ *z z′*. $z \to_n z'$ ⟶ *nodes*(*graphI z*) = *nodes*(*graphI z′*)
  by (*clarify*, *erule state-transition-in.cases*,
      (*simp add*: *move-graph-a-def atI-def actors-graph-def nodes-def*)+)

lemma *same-nodes*: (*I*, *y*) ∈ {(*x*::*infrastructure*, *y*::*infrastructure*). $x \to_n y$}*
            ⟹ *nodes*(*graphI y*) = *nodes*(*graphI I*)
 by (*erule rtrancl-induct*, *rule refl*, *drule CollectD*, *simp*, *drule same-nodes0*, *simp*)

lemma *same-actors0*[*rule-format*]: ∀ *z z′*. $z \to_n z'$ ⟶ *actors-graph*(*graphI z*) = *actors-graph*(*graphI z′*)
proof (*clarify*, *erule state-transition-in.cases*)
 show ⋀(*z*::*infrastructure*) (*z′*::*infrastructure*) (*G*::*igraph*) (*I*::*infrastructure*) (*a*::*char list*)
      (*l*::*location*) (*a′*::*char list*) (*za*::*char list*) *I′*::*infrastructure*.
      z = *I* ⟹
      z′ = *I′* ⟹
      G = *graphI I* ⟹
      $a \ @_G \ l$ ⟹
      $a' \ @_G \ l$ ⟹
      has G (*Actor a*, *za*) ⟹
      enables I l (*Actor a*) *get* ⟹
      I′ =
      Infrastructure
       (*Lgraph* (*gra G*) (*agra G*)
         ((*cgra G*)(*Actor a′* := (*za* # *fst* (*cgra G* (*Actor a′*)), *snd* (*cgra G* (*Actor a′*))))) (*lgra G*))
       (*delta I*) ⟹
      actors-graph (*graphI z*) = *actors-graph* (*graphI z′*)
    by (*simp add*: *actors-graph-def nodes-def*)
 next show ⋀(*z*::*infrastructure*) (*z′*::*infrastructure*) (*G*::*igraph*) (*I*::*infrastructure*) (*a*::*char list*)
      (*l*::*location*) (*I′*::*infrastructure*) *za*::*char list*.
      z = *I* ⟹
      z′ = *I′* ⟹
      G = *graphI I* ⟹
      $a \ @_G \ l$ ⟹
      enables I l (*Actor a*) *put* ⟹
      I′ = *Infrastructure* (*Lgraph* (*gra G*) (*agra G*) (*cgra G*) ((*lgra G*)(*l* := [*za*])))

$(\mathit{delta}\ I) \Longrightarrow$
    $\mathit{actors\text{-}graph}\ (\mathit{graphI}\ z) = \mathit{actors\text{-}graph}\ (\mathit{graphI}\ z')$
  by $(\mathit{simp\ add}{:}\ \mathit{actors\text{-}graph\text{-}def}\ \mathit{nodes\text{-}def})$
next show $\bigwedge(z{::}\mathit{infrastructure})\ (z'{::}\mathit{infrastructure})\ (G{::}\mathit{igraph})\ (I{::}\mathit{infrastructure})$
$(l{::}\mathit{location})$
    $(a{::}\mathit{char\ list})\ (I'{::}\mathit{infrastructure})\ za{::}\mathit{char\ list}.$
    $z = I \Longrightarrow$
    $z' = I' \Longrightarrow$
    $G = \mathit{graphI}\ I \Longrightarrow$
    $\mathit{enables}\ I\ l\ (\mathit{Actor}\ a)\ \mathit{put} \Longrightarrow$
    $I' = \mathit{Infrastructure}\ (\mathit{Lgraph}\ (\mathit{gra}\ G)\ (\mathit{agra}\ G)\ (\mathit{cgra}\ G)\ ((\mathit{lgra}\ G)(l := [za])))$
$(\mathit{delta}\ I) \Longrightarrow$
    $\mathit{actors\text{-}graph}\ (\mathit{graphI}\ z) = \mathit{actors\text{-}graph}\ (\mathit{graphI}\ z')$
  by $(\mathit{simp\ add}{:}\ \mathit{actors\text{-}graph\text{-}def}\ \mathit{nodes\text{-}def})$
next fix $z\ z'\ G\ I\ a\ l\ l'\ I'$
 show $z = I \Longrightarrow z' = I' \Longrightarrow G = \mathit{graphI}\ I \Longrightarrow a\ @_G\ l \Longrightarrow$
    $l \in \mathit{nodes}\ G \Longrightarrow l' \in \mathit{nodes}\ G \Longrightarrow a \in \mathit{actors\text{-}graph}\ (\mathit{graphI}\ I) \Longrightarrow$
    $\mathit{enables}\ I\ l'\ (\mathit{Actor}\ a)\ \mathit{move} \Longrightarrow$
    $I' = \mathit{Infrastructure}\ (\mathit{move\text{-}graph\text{-}a}\ a\ l\ l'\ (\mathit{graphI}\ I))\ (\mathit{delta}\ I) \Longrightarrow$
    $\mathit{actors\text{-}graph}\ (\mathit{graphI}\ z) = \mathit{actors\text{-}graph}\ (\mathit{graphI}\ z')$
 proof $(\mathit{rule\ equalityI})$
  show $z = I \Longrightarrow z' = I' \Longrightarrow G = \mathit{graphI}\ I \Longrightarrow a\ @_G\ l \Longrightarrow$
  $l \in \mathit{nodes}\ G \Longrightarrow l' \in \mathit{nodes}\ G \Longrightarrow a \in \mathit{actors\text{-}graph}\ (\mathit{graphI}\ I) \Longrightarrow$
  $\mathit{enables}\ I\ l'\ (\mathit{Actor}\ a)\ \mathit{move} \Longrightarrow$
  $I' = \mathit{Infrastructure}\ (\mathit{move\text{-}graph\text{-}a}\ a\ l\ l'\ (\mathit{graphI}\ I))\ (\mathit{delta}\ I) \Longrightarrow$
  $\mathit{actors\text{-}graph}\ (\mathit{graphI}\ z) \subseteq \mathit{actors\text{-}graph}\ (\mathit{graphI}\ z')$
 by (*rule subsetI*, *simp add*: *actors-graph-def* ,(*erule exE*)+, *case-tac* $x = a$,
   rule-tac $x = l'$ **in** *exI*, *simp add*: *move-graph-a-def nodes-def atI-def*,
    rule-tac $x = ya$ **in** *exI*, *rule conjI*, *simp add*: *move-graph-a-def nodes-def*
atI-def,
   (*erule conjE*)+, *simp add*: *move-graph-a-def*, *rule conjI*, *clarify*,
    simp add: *move-graph-a-def nodes-def atI-def*, *rule del-not-a*, *assumption*+,
clarify)
next show $z = I \Longrightarrow z' = I' \Longrightarrow G = \mathit{graphI}\ I \Longrightarrow a\ @_G\ l \Longrightarrow$
  $l \in \mathit{nodes}\ G \Longrightarrow l' \in \mathit{nodes}\ G \Longrightarrow a \in \mathit{actors\text{-}graph}\ (\mathit{graphI}\ I) \Longrightarrow$
  $\mathit{enables}\ I\ l'\ (\mathit{Actor}\ a)\ \mathit{move} \Longrightarrow$
  $I' = \mathit{Infrastructure}\ (\mathit{move\text{-}graph\text{-}a}\ a\ l\ l'\ (\mathit{graphI}\ I))\ (\mathit{delta}\ I) \Longrightarrow$
  $\mathit{actors\text{-}graph}\ (\mathit{graphI}\ z') \subseteq \mathit{actors\text{-}graph}\ (\mathit{graphI}\ z)$
 by (*rule subsetI*, *simp add*: *actors-graph-def*, (*erule exE*)+,
   case-tac $x = a$, *rule-tac* $x = l$ **in** *exI*, *simp add*: *move-graph-a-def nodes-def*
atI-def,
    rule-tac $x = ya$ **in** *exI*, *rule conjI*, *simp add*: *move-graph-a-def nodes-def*
atI-def,
   (*erule conjE*)+, *simp add*: *move-graph-a-def*, *case-tac* $ya = l$, *simp*,
   case-tac $a \in \mathit{set}\ (\mathit{agra}\ (\mathit{graphI}\ I)\ l) \wedge a \notin \mathit{set}\ (\mathit{agra}\ (\mathit{graphI}\ I)\ l')$, *simp*,
   case-tac $l = l'$, *simp*+, *erule del-up*, *simp*,
   case-tac $a \in \mathit{set}\ (\mathit{agra}\ (\mathit{graphI}\ I)\ l) \wedge a \notin \mathit{set}\ (\mathit{agra}\ (\mathit{graphI}\ I)\ l')$, *simp*,
   case-tac $ya = l'$, *simp*+)
qed

qed

lemma *same-actors*: $(I,\ y) \in \{(x{::}infrastructure,\ y{::}infrastructure).\ x \rightarrow_n y\}^*$
$\Longrightarrow$ *actors-graph*(*graphI I*) = *actors-graph*(*graphI y*)
proof (*erule rtrancl-induct*)
show *actors-graph* (*graphI I*) = *actors-graph* (*graphI I*)
by (*rule refl*)
next show $\bigwedge$(*y*::*infrastructure*) *z*::*infrastructure*.
$(I,\ y) \in \{(x{::}infrastructure,\ y{::}infrastructure).\ x \rightarrow_n y\}^* \Longrightarrow$
$(y,\ z) \in \{(x{::}infrastructure,\ y{::}infrastructure).\ x \rightarrow_n y\} \Longrightarrow$
actors-graph (*graphI I*) = *actors-graph* (*graphI y*) $\Longrightarrow$
actors-graph (*graphI I*) = *actors-graph* (*graphI z*)
by (*drule CollectD*, *simp*, *drule same-actors0*, *simp*)
qed

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) ∧ 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*] ⇒ *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) ∧ 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 ≡ *Infrastructure ex-graph local-policies*

fixes *Airplane-in-danger* :: *infrastructure*
defines *Airplane-in-danger-def*:
Airplane-in-danger ≡ *Infrastructure aid-graph local-policies*

fixes *Airplane-getting-in-danger0* :: *infrastructure*
defines *Airplane-getting-in-danger0-def*:
Airplane-getting-in-danger0 ≡ *Infrastructure aid-graph0 local-policies*

fixes *Airplane-getting-in-danger* :: *infrastructure*
defines *Airplane-getting-in-danger-def*:
Airplane-getting-in-danger ≡ *Infrastructure agid-graph local-policies*

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

fixes *Air-Kripke*
defines *Air-Kripke* ≡ *Kripke Air-states* {*Airplane-scenario*}

fixes *Airplane-not-in-danger* :: *infrastructure*
defines *Airplane-not-in-danger-def*:
Airplane-not-in-danger ≡ *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 ≡ *Infrastructure ex-graph local-policies-four-eyes*

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

fixes *Air-tp-Kripke*
defines *Air-tp-Kripke* ≡ *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* ∧ *Actor x* ≠ *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*''
by (*simp add*: *Airplane-scenario-def global-policy-def airplane-actors-def*)

lemma *ex-inv2*: *global-policy Airplane-scenario* ''*Charly*''
by (*simp add*: *Airplane-scenario-def global-policy-def airplane-actors-def*)

lemma *ex-inv3*: ¬*global-policy Airplane-scenario ''Eve''*
proof (*simp add*: *Airplane-scenario-def global-policy-def*, *rule conjI*)
  show *''Eve''* ∉ *airplane-actors* **by** (*simp add*: *airplane-actors-def*)
next show
  enables (*Infrastructure ex-graph local-policies*) *cockpit* (*Actor ''Eve''*) *put*
  proof −
    have *a*: *Actor ''Charly''* = *Actor ''Eve''*
      by (*insert Insider-Eve*, *unfold Insider-def*, (*drule mp*),
          rule Eve-precipitating-event, *simp add*: *UasI-def*)
    show *?thesis*
     by (*insert a*, *simp add*: *Airplane-scenario-def enables-def ex-creds-def local-policies-def ex-graph-def*,
       insert Insider-Eve, *unfold Insider-def*, (*drule mp*), *rule Eve-precipitating-event*,

          simp add: *UasI-def*, *rule-tac x* = *''Charly''* **in** *exI*, *simp add*: *cockpit-def atI-def*)
  qed
qed

show Safety for Airplane_scenario

lemma *Safety*: *Safety Airplane-scenario* (*''Alice''*)
proof −
  show *Safety Airplane-scenario ''Alice''*
    by (*simp add*: *Airplane-scenario-def Safety-def enables-def ex-creds-def*
              local-policies-def ex-graph-def cockpit-def, *rule impI*,
        rule-tac x = *''Alice''* **in** *exI*, *simp add*: *atI-def cabin-def ex-locs-def door-def*,
      rule conjI, *simp add*: *has-def credentials-def*, *simp add*: *isin-def credentials-def*)
qed

show Security for Airplane_scenario

lemma *inj-lem*: ⟦ *inj f*; *x* ≠ *y* ⟧ ⟹ *f x* ≠ *f y*
by (*simp add*: *inj-eq*)

lemma *inj-on-lem*: ⟦ *inj-on f A*; *x* ≠ *y*; *x*∈ *A*; *y* ∈ *A* ⟧ ⟹ *f x* ≠ *f y*
by (*simp add*: *inj-on-def*, *blast*)

lemma *inj-lemma'*: *inj-on* (*isin ex-graph door*) {*''locked''*,*''norm''*}
  by (*unfold inj-on-def ex-graph-def isin-def*, *simp*, *unfold ex-locs-def*, *simp*)

lemma *inj-lemma''*: *inj-on* (*isin aid-graph door*) {*''locked''*,*''norm''*}
 by (*unfold inj-on-def aid-graph-def isin-def*, *simp*, *unfold ex-locs'-def*, *simp*)

lemma *locl-lemma2*: *isin ex-graph door ''norm''* ≠ *isin ex-graph door ''locked''*
by (*rule-tac A* = {*''locked''*,*''norm''*} **and** *f* = *isin ex-graph door* **in** *inj-on-lem*,
      rule inj-lemma', *simp+*)

lemma *locl-lemma3*: *isin ex-graph door ''norm''* = (¬ *isin ex-graph door ''locked''*)
by (*insert locl-lemma2*, *blast*)

lemma *locl-lemma2a*: *isin aid-graph door ''norm''* ≠ *isin aid-graph door ''locked''*
by (*rule-tac A* = {*''locked''*,*''norm''*} **and** *f* = *isin aid-graph door* **in** *inj-on-lem*,
    rule inj-lemma'', *simp*+)

lemma *locl-lemma3a*: *isin aid-graph door ''norm''* = (¬ *isin aid-graph door ''locked''*)
by (*insert locl-lemma2a*, *blast*)

lemma *Security*: *Security Airplane-scenario s*
  by (*simp add*: *Airplane-scenario-def Security-def enables-def local-policies-def ex-locs-def locl-lemma3*)

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

lemma *Security-problem*: *Security Airplane-scenario ''Bob''*
by (*rule Security*)

show that pilot can get out of cockpit

lemma *pilot-can-leave-cockpit*: (*enables Airplane-scenario cabin* (*Actor ''Bob''*) *move*)
 by (*simp add*: *Airplane-scenario-def Security-def ex-creds-def ex-graph-def enables-def*

      local-policies-def ex-locs-def, *simp add*: *cockpit-def cabin-def door-def*)

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 (*simp add*: *Airplane-in-danger-def global-policy-def*, *rule conjI*)
  show *''Eve''* ∉ *airplane-actors* **by** (*simp add*: *airplane-actors-def*)
next show *enables* (*Infrastructure aid-graph local-policies*) *cockpit* (*Actor ''Eve''*) *put*
  proof −
    have *a*: *Actor ''Charly''* = *Actor ''Eve''*
      by (*insert Insider-Eve*, *unfold Insider-def*, (*drule mp*),
        rule Eve-precipitating-event, *simp add*: *UasI-def*)
    show *?thesis*
     apply (*insert a*, *erule subst*)
     by (*simp add*: *enables-def local-policies-def cockpit-def aid-graph-def atI-def*)
 qed
qed

lemma *Safety-in-danger*:
  fixes *s*
  assumes *s* ∈ *airplane-actors*
  shows ¬(*Safety Airplane-in-danger s*)
proof (*simp add*: *Airplane-in-danger-def Safety-def enables-def assms*)
  show ∀ *x*::(*actor* ⇒ *bool*) × *action set*∈*local-policies aid-graph cockpit*.
      ¬ (*case x of* (*p*::*actor* ⇒ *bool*, *e*::*action set*) ⇒ *move* ∈ *e* ∧ *p* (*Actor s*))
    by (*simp add*: *local-policies-def aid-graph-def ex-locs'-def isin-def*)
qed

```
lemma Security-problem': ¬(enables Airplane-in-danger cockpit (Actor ''Bob'')
move)
proof (simp add: Airplane-in-danger-def Security-def enables-def local-policies-def

      ex-locs-def locl-lemma3a, rule impI)
  assume has aid-graph (Actor ''Bob'', ''PIN'')
  show (∀ n::char list.
         Actor n = Actor ''Bob'' ⟶ n @aid-graph cabin ⟶ isin aid-graph door
''locked'')
by (simp add: aid-graph-def isin-def ex-locs'-def)
qed
```

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
by (simp add: Airplane-not-in-danger-def global-policy-def)

lemma ex-inv6: global-policy Airplane-not-in-danger a
proof (simp add: Airplane-not-in-danger-def global-policy-def, rule impI)
  assume a ∉ airplane-actors
  show ¬ enables (Infrastructure aid-graph local-policies-four-eyes) cockpit (Actor
a) put
by (simp add: aid-graph-def ex-locs'-def enables-def local-policies-four-eyes-def)
qed

lemma step0: Airplane-scenario →n Airplane-getting-in-danger0
proof (rule-tac l = cockpit and l' = door and a = ''Bob'' in move, rule refl)
  show ''Bob'' @graphI Airplane-scenario cockpit
  by (simp add: Airplane-scenario-def atI-def ex-graph-def)
next show cockpit ∈ nodes (graphI Airplane-scenario)
    by (simp add: ex-graph-def Airplane-scenario-def nodes-def, blast)+
next show door ∈ nodes (graphI Airplane-scenario)
   by (simp add: actors-graph-def door-def cockpit-def nodes-def cabin-def,
       rule-tac x = Location 2 in exI,
       simp add: Airplane-scenario-def ex-graph-def cockpit-def door-def)
next show ''Bob'' ∈ actors-graph (graphI Airplane-scenario)
    by (simp add: actors-graph-def Airplane-scenario-def nodes-def ex-graph-def,
blast)
next show enables Airplane-scenario door (Actor ''Bob'') move
    by (simp add: Airplane-scenario-def enables-def local-policies-def ex-creds-def
door-def cockpit-def)
next show Airplane-getting-in-danger0 =
    Infrastructure (move-graph-a ''Bob'' cockpit door (graphI Airplane-scenario))
     (delta Airplane-scenario)
  proof −
     have a: (move-graph-a ''Bob'' cockpit door (graphI Airplane-scenario)) =
aid-graph0
      by (simp add: move-graph-a-def door-def cockpit-def Airplane-scenario-def
        aid-graph0-def ex-graph-def, rule ext, simp add: cabin-def door-def)
    show ?thesis
      by (unfold Airplane-getting-in-danger0-def, insert a, erule ssubst,
          simp add: Airplane-scenario-def)
  qed
qed

lemma step1: Airplane-getting-in-danger0 →n Airplane-getting-in-danger
proof (rule-tac l = door and l' = cabin and a = ''Bob'' in  move, rule refl)
  show ''Bob'' @graphI Airplane-getting-in-danger0 door
   by (simp add: Airplane-getting-in-danger0-def atI-def aid-graph0-def door-def
cockpit-def)
next show door ∈ nodes (graphI Airplane-getting-in-danger0)
   by (simp add: aid-graph0-def Airplane-getting-in-danger0-def nodes-def, blast)+
next show cabin ∈ nodes (graphI Airplane-getting-in-danger0)
    by (simp add: actors-graph-def door-def cockpit-def nodes-def cabin-def,
    rule-tac x = Location 1 in exI,
     simp add: Airplane-getting-in-danger0-def aid-graph0-def cockpit-def door-def
cabin-def)
next show ''Bob'' ∈ actors-graph (graphI Airplane-getting-in-danger0)
   by (simp add: actors-graph-def door-def cockpit-def nodes-def cabin-def
                 Airplane-getting-in-danger0-def aid-graph0-def, blast)
next show enables Airplane-getting-in-danger0 cabin (Actor ''Bob'') move
  by (simp add: Airplane-getting-in-danger0-def enables-def local-policies-def ex-creds-def
door-def
               cockpit-def cabin-def)
next show Airplane-getting-in-danger =
   Infrastructure (move-graph-a ''Bob'' door cabin (graphI Airplane-getting-in-danger0))
     (delta Airplane-getting-in-danger0)
    by (unfold Airplane-getting-in-danger-def,
        simp add: Airplane-getting-in-danger0-def agid-graph-def aid-graph0-def
                 move-graph-a-def door-def cockpit-def cabin-def, rule ext,
        simp add: cabin-def door-def)
qed

lemma step2: Airplane-getting-in-danger →n Airplane-in-danger
proof (rule-tac l = door and a = ''Charly'' and z = ''locked'' in  put-remote,
rule refl)
  show enables Airplane-getting-in-danger door (Actor ''Charly'') put
   by (simp add: enables-def local-policies-def ex-creds-def door-def cockpit-def,
       unfold Airplane-getting-in-danger-def,
       simp add: local-policies-def cockpit-def cabin-def door-def,
       rule-tac x = ''Charly'' in exI, rule conjI,
       simp add: atI-def agid-graph-def door-def cockpit-def, rule refl)
next show Airplane-in-danger =
    Infrastructure
   (Lgraph (gra (graphI Airplane-getting-in-danger)) (agra (graphI Airplane-getting-in-danger))
       (cgra (graphI Airplane-getting-in-danger))
       ((lgra (graphI Airplane-getting-in-danger))(door := [''locked''])))
    (delta Airplane-getting-in-danger)
   by (unfold Airplane-in-danger-def, simp add: aid-graph-def agid-graph-def
            ex-locs'-def ex-locs-def Airplane-getting-in-danger-def, force)
qed

lemma step0r: Airplane-scenario →n* Airplane-getting-in-danger0
  by (simp add: state-transition-in-refl-def, insert step0, auto)

lemma step1r: Airplane-getting-in-danger0 →n* Airplane-getting-in-danger
  by (simp add: state-transition-in-refl-def, insert step1, auto)

lemma step2r: Airplane-getting-in-danger →n* Airplane-in-danger
  by (simp add: state-transition-in-refl-def, insert step2, auto)

theorem step-allr:  Airplane-scenario →n* Airplane-in-danger
  by (insert step0r step1r step2r, simp add: state-transition-in-refl-def)

theorem aid-attack: Air-Kripke ⊢ EF ({x. ¬ global-policy x ''Eve''})
proof (simp add: check-def Air-Kripke-def, rule conjI)
  show Airplane-scenario ∈ Air-states
    by (simp add: Air-states-def state-transition-in-refl-def)
next show Airplane-scenario ∈ EF {x::infrastructure. ¬ global-policy x ''Eve''}
   by (rule EF-lem2b, subst EF-lem000, rule EX-lem0r, subst EF-lem000, rule
EX-step,
      unfold state-transition-infra-def, rule step0, rule EX-lem0r,
      rule-tac y = Airplane-getting-in-danger in EX-step,
      unfold state-transition-infra-def, rule step1, subst EF-lem000, rule EX-lem0l,
      rule-tac y = Airplane-in-danger in EX-step, unfold state-transition-infra-def,
      rule step2, rule CollectI, rule ex-inv4)
qed
```

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

```
lemma  actors-unique-loc-base:
  assumes I →n I'
      and (∀ l l'. a @graphI I l ∧ a @graphI I l' ⟶ l = l')∧
          (∀ l. nodup a (agra (graphI I) l))
    shows (∀ l l'. a @graphI I' l ∧ a @graphI I' l'  ⟶ l = l') ∧
          (∀ l. nodup a (agra (graphI I') l))
proof (rule state-transition-in.cases, rule assms(1))
  show ⋀(G::igraph) (Ia::infrastructure) (aa::char list) (l::location) (a'::char list)
(z::char list)
       I'a::infrastructure.
       I = Ia ⟹
       I' = I'a ⟹
       G = graphI Ia ⟹
       aa @G l ⟹
       a' @G l ⟹
       has G (Actor aa, z) ⟹
       enables Ia l (Actor aa) get ⟹
```

$I'a =$
Infrastructure
$(Lgraph\ (gra\ G)\ (agra\ G)$
$((cgra\ G)(Actor\ a' := (z\ \#\ fst\ (cgra\ G\ (Actor\ a')),\ snd\ (cgra\ G\ (Actor\ a'))))))\ (lgra\ G))$
$(delta\ Ia) \Longrightarrow$
$(\forall\ (l::location)\ l'::location.\ a\ @_{graphI\ I'}\ l \wedge a\ @_{graphI\ I'}\ l' \longrightarrow l = l') \wedge$
$(\forall\ l::location.\ nodup\ a\ (agra\ (graphI\ I')\ l))$ **using** *assms*
by (*simp add*: *atI-def*)
next fix *G Ia aa l I'a z*
assume *a0*: $I = Ia$ **and** *a1*: $I' = I'a$ **and** *a2*: $G = graphI\ Ia$ **and** *a3*: $aa\ @_G\ l$
and *a4*: *enables Ia l* (*Actor aa*) *put*
and *a5*: $I'a = Infrastructure\ (Lgraph\ (gra\ G)\ (agra\ G)\ (cgra\ G)\ ((lgra\ G)(l := [z])))\ (delta\ Ia)$
show $(\forall\ (l::location)\ l'::location.\ a\ @_{graphI\ I'}\ l \wedge a\ @_{graphI\ I'}\ l' \longrightarrow l = l') \wedge$
$(\forall\ l::location.\ nodup\ a\ (agra\ (graphI\ I')\ l))$**using** *assms*
by (*simp add*: *a0 a1 a2 a3 a4 a5 atI-def*)
next show $\bigwedge(G::igraph)\ (Ia::infrastructure)\ (l::location)\ (aa::char\ list)\ (I'a::infrastructure)$
$z::char\ list.$
$I = Ia \Longrightarrow$
$I' = I'a \Longrightarrow$
$G = graphI\ Ia \Longrightarrow$
$enables\ Ia\ l\ (Actor\ aa)\ put \Longrightarrow$
$I'a = Infrastructure\ (Lgraph\ (gra\ G)\ (agra\ G)\ (cgra\ G)\ ((lgra\ G)(l := [z])))\ (delta\ Ia) \Longrightarrow$
$(\forall\ (l::location)\ l'::location.\ a\ @_{graphI\ I'}\ l \wedge a\ @_{graphI\ I'}\ l' \longrightarrow l = l') \wedge$
$(\forall\ l::location.\ nodup\ a\ (agra\ (graphI\ I')\ l))$
by (*clarify*, *simp add*: *assms atI-def*)
next show $\bigwedge(G::igraph)\ (Ia::infrastructure)\ (aa::char\ list)\ (l::location)\ (l'::location)$
$I'a::infrastructure.$
$I = Ia \Longrightarrow$
$I' = I'a \Longrightarrow$
$G = graphI\ Ia \Longrightarrow$
$aa\ @_G\ l \Longrightarrow$
$l \in nodes\ G \Longrightarrow$
$l' \in nodes\ G \Longrightarrow$
$aa \in actors\text{-}graph\ (graphI\ Ia) \Longrightarrow$
$enables\ Ia\ l'\ (Actor\ aa)\ move \Longrightarrow$
$I'a = Infrastructure\ (move\text{-}graph\text{-}a\ aa\ l\ l'\ (graphI\ Ia))\ (delta\ Ia) \Longrightarrow$
$(\forall\ (l::location)\ l'::location.\ a\ @_{graphI\ I'}\ l \wedge a\ @_{graphI\ I'}\ l' \longrightarrow l = l') \wedge$
$(\forall\ l::location.\ nodup\ a\ (agra\ (graphI\ I')\ l))$
proof (*simp add*: *move-graph-a-def*,*rule conjI*, *clarify*, *rule conjI*, *clarify*, *rule conjI*, *clarify*)
show $\bigwedge(G::igraph)\ (Ia::infrastructure)\ (aa::char\ list)\ (l::location)\ (l'::location)$
$(I'a::infrastructure)\ (la::location)\ l'a::location.$
$I' =$
Infrastructure
$(Lgraph\ (gra\ (graphI\ I))$
$(if\ a \in set\ (agra\ (graphI\ I)\ l) \wedge a \notin set\ (agra\ (graphI\ I)\ l')$

then (*agra* (*graphI I*))(*l* := *del a* (*agra* (*graphI I*) *l*), *l'* := *a* # *agra* (*graphI I*) *l'*)
else agra (*graphI I*))
(*cgra* (*graphI I*)) (*lgra* (*graphI I*)))
(*delta I*) $\Longrightarrow$
a $@_{graphI\ I}$ *l* $\Longrightarrow$
l $\in$ *nodes* (*graphI I*) $\Longrightarrow$
l' $\in$ *nodes* (*graphI I*) $\Longrightarrow$
a $\in$ *actors-graph* (*graphI I*) $\Longrightarrow$
enables I l' (*Actor a*) *move* $\Longrightarrow$
a $\in$ *set* (*agra* (*graphI I*) *l*) $\Longrightarrow$
a $\notin$ *set* (*agra* (*graphI I*) *l'*) $\Longrightarrow$
a $@_{Lgraph\ (gra\ (graphI\ I))}$ ((*agra* (*graphI I*))(*l* := *del a* (*agra* (*graphI I*) *l*), *l'* := *a* # *agra* (*graphI I*)
la $\Longrightarrow$
a $@_{Lgraph\ (gra\ (graphI\ I))}$ ((*agra* (*graphI I*))(*l* := *del a* (*agra* (*graphI I*) *l*), *l'* := *a* # *agra* (*graphI I*)
l'a $\Longrightarrow$
la = *l'a*
apply (*case-tac la* $\neq$ *l'* $\wedge$ *la* $\neq$ *l* $\wedge$ *l'a* $\neq$ *l'* $\wedge$ *l'a* $\neq$ *l*)
apply (*simp add*: *atI-def*)
apply (*subgoal-tac la* = *l'* $\vee$ *la* = *l* $\vee$ *l'a* = *l'* $\vee$ *l'a* = *l*)
prefer *2*
using *assms*(*2*) *atI-def* **apply** *blast*
apply *blast*
by (*metis agra.simps assms*(*2*) *atI-def del-nodup fun-upd-apply*)
next show $\bigwedge$(*G*::*igraph*) (*Ia*::*infrastructure*) (*aa*::*char list*) (*l*::*location*) (*l'*::*location*)
I'a::*infrastructure*.
I' =
Infrastructure
(*Lgraph* (*gra* (*graphI I*))
(*if a* $\in$ *set* (*agra* (*graphI I*) *l*) $\wedge$ *a* $\notin$ *set* (*agra* (*graphI I*) *l'*)
then (*agra* (*graphI I*))(*l* := *del a* (*agra* (*graphI I*) *l*), *l'* := *a* # *agra* (*graphI I*) *l'*)
else agra (*graphI I*))
(*cgra* (*graphI I*)) (*lgra* (*graphI I*)))
(*delta I*) $\Longrightarrow$
a $@_{graphI\ I}$ *l* $\Longrightarrow$
l $\in$ *nodes* (*graphI I*) $\Longrightarrow$
l' $\in$ *nodes* (*graphI I*) $\Longrightarrow$
a $\in$ *actors-graph* (*graphI I*) $\Longrightarrow$
enables I l' (*Actor a*) *move* $\Longrightarrow$
a $\in$ *set* (*agra* (*graphI I*) *l*) $\Longrightarrow$
a $\notin$ *set* (*agra* (*graphI I*) *l'*) $\Longrightarrow$
$\forall$ *la*::*location*.
(*la* = *l* $\longrightarrow$ *l* $\neq$ *l'* $\longrightarrow$ *nodup a* (*del a* (*agra* (*graphI I*) *l*))) $\wedge$
(*la* $\neq$ *l* $\longrightarrow$ *la* $\neq$ *l'* $\longrightarrow$ *nodup a* (*agra* (*graphI I*) *la*))
by (*simp add*: *assms*(*2*) *nodup-down*)
next show $\bigwedge$(*G*::*igraph*) (*Ia*::*infrastructure*) (*aa*::*char list*) (*l*::*location*) (*l'*::*location*)
I'a::*infrastructure*.
I' =

```
    Infrastructure
     (Lgraph (gra (graphI I))
       (if a ∈ set (agra (graphI I) l) ∧ a ∉ set (agra (graphI I) l')
          then (agra (graphI I))(l := del a (agra (graphI I) l), l' := a # agra (graphI I) l')
        else agra (graphI I))
       (cgra (graphI I)) (lgra (graphI I)))
     (delta I) ⟹
    a @graphI I l ⟹
    l ∈ nodes (graphI I) ⟹
    l' ∈ nodes (graphI I) ⟹
    a ∈ actors-graph (graphI I) ⟹
    enables I l' (Actor a) move ⟹
    (a ∈ set (agra (graphI I) l) ⟶ a ∈ set (agra (graphI I) l')) ⟶
    (∀ (l::location) l'::location.
      a @Lgraph (gra (graphI I)) (agra (graphI I)) (cgra (graphI I)) (lgra (graphI I)) l ∧
      a @Lgraph (gra (graphI I)) (agra (graphI I)) (cgra (graphI I)) (lgra (graphI I)) l' ⟶
        l = l') ∧
    (∀ l::location. nodup a (agra (graphI I) l))
   by (simp add: assms(2) atI-def)
 next show ⋀(G::igraph) (Ia::infrastructure) (aa::char list) (l::location) (l'::location)
    I'a::infrastructure.
    I = Ia ⟹
    I' =
    Infrastructure
     (Lgraph (gra (graphI Ia))
       (if aa ∈ set (agra (graphI Ia) l) ∧ aa ∉ set (agra (graphI Ia) l')
        then (agra (graphI Ia))(l := del aa (agra (graphI Ia) l), l' := aa # agra (graphI Ia) l')
        else agra (graphI Ia))
       (cgra (graphI Ia)) (lgra (graphI Ia)))
     (delta Ia) ⟹
    G = graphI Ia ⟹
    aa @graphI Ia l ⟹
    l ∈ nodes (graphI Ia) ⟹
    l' ∈ nodes (graphI Ia) ⟹
    aa ∈ actors-graph (graphI Ia) ⟹
    enables Ia l' (Actor aa) move ⟹
    I'a =
    Infrastructure
     (Lgraph (gra (graphI Ia))
       (if aa ∈ set (agra (graphI Ia) l) ∧ aa ∉ set (agra (graphI Ia) l')
        then (agra (graphI Ia))(l := del aa (agra (graphI Ia) l), l' := aa # agra (graphI Ia) l')
        else agra (graphI Ia))
       (cgra (graphI Ia)) (lgra (graphI Ia)))
     (delta Ia) ⟹
```

aa ≠ *a* ⟶
(*aa* ∈ *set* (*agra* (*graphI Ia*) *l*) ∧ *aa* ∉ *set* (*agra* (*graphI Ia*) *l*′) ⟶
(∀ (*la*::*location*) *l*′*a*::*location*.
a $@_{Lgraph}$ (*gra* (*graphI Ia*)) ((*agra* (*graphI Ia*)) (*l* := *del aa* (*agra* (*graphI Ia*) *l*), *l*
la ∧
a $@_{Lgraph}$ (*gra* (*graphI Ia*)) ((*agra* (*graphI Ia*)) (*l* := *del aa* (*agra* (*graphI Ia*) *l*), *l*
l′*a* ⟶
la = *l*′*a*) ∧
(∀ *la*::*location*.
(*la* = *l* ⟶
(*l* = *l*′ ⟶ *nodup a* (*agra* (*graphI Ia*) *l*′)) ∧
(*l* ≠ *l*′ ⟶ *nodup a* (*del aa* (*agra* (*graphI Ia*) *l*)))) ∧
(*la* ≠ *l* ⟶
(*la* = *l*′ ⟶ *nodup a* (*agra* (*graphI Ia*) *l*′)) ∧
(*la* ≠ *l*′ ⟶ *nodup a* (*agra* (*graphI Ia*) *la*))))) ∧
((*aa* ∈ *set* (*agra* (*graphI Ia*) *l*) ⟶ *aa* ∈ *set* (*agra* (*graphI Ia*) *l*′)) ⟶
(∀ (*l*::*location*) *l*′::*location*.
a $@_{Lgraph}$ (*gra* (*graphI Ia*)) (*agra* (*graphI Ia*)) (*cgra* (*graphI Ia*)) (*lgra* (*graphI Ia*))
l ∧
a $@_{Lgraph}$ (*gra* (*graphI Ia*)) (*agra* (*graphI Ia*)) (*cgra* (*graphI Ia*)) (*lgra* (*graphI Ia*))
l′ ⟶
l = *l*′) ∧
(∀ *l*::*location*. *nodup a* (*agra* (*graphI Ia*) *l*)))

proof (*clarify*, *simp add*: *atI-def*,*rule conjI*,*clarify*,*rule conjI*,*clarify*,*rule conjI*,
clarify,*rule conjI*,*clarify*,*simp*,*clarify*,*rule conjI*,(*rule impI*)+)
show ⋀(*aa*::*char list*) (*l*::*location*) (*l*′::*location*) *l*′*a*::*location*.
I′ =
Infrastructure
(*Lgraph* (*gra* (*graphI I*))
((*agra* (*graphI I*))(*l* := *del aa* (*agra* (*graphI I*) *l*), *l*′ := *aa* # *agra* (*graphI*
I) *l*′))
(*cgra* (*graphI I*)) (*lgra* (*graphI I*)))
(*delta I*) ⟹
aa ∈ *set* (*agra* (*graphI I*) *l*) ⟹
l ∈ *nodes* (*graphI I*) ⟹
l′ ∈ *nodes* (*graphI I*) ⟹
aa ∈ *actors-graph* (*graphI I*) ⟹
enables I l′ (*Actor aa*) *move* ⟹
aa ≠ *a* ⟹
aa ∉ *set* (*agra* (*graphI I*) *l*′) ⟹
l ≠ *l*′ ⟹
l′*a* ≠ *l* ⟹
l′*a* = *l*′ ⟹ *a* ∈ *set* (*del aa* (*agra* (*graphI I*) *l*)) ⟹ *a* ∉ *set* (*agra* (*graphI*
I) *l*′)
by (*meson assms*(*2*) *atI-def del-notin-down*)
next show ⋀(*aa*::*char list*) (*l*::*location*) (*l*′::*location*) *l*′*a*::*location*.
I′ =
Infrastructure
(*Lgraph* (*gra* (*graphI I*))

((*agra* (*graphI I*))(*l* := *del aa* (*agra* (*graphI I*) *l*), *l′* := *aa* # *agra* (*graphI I*) *l′*))
(*cgra* (*graphI I*)) (*lgra* (*graphI I*)))
(*delta I*) ⟹
aa ∈ *set* (*agra* (*graphI I*) *l*) ⟹
l ∈ *nodes* (*graphI I*) ⟹
l′ ∈ *nodes* (*graphI I*) ⟹
aa ∈ *actors-graph* (*graphI I*) ⟹
enables I l′ (*Actor aa*) *move* ⟹
aa ≠ *a* ⟹
aa ∉ *set* (*agra* (*graphI I*) *l′*) ⟹
l ≠ *l′* ⟹
l′a ≠ *l* ⟹
l′a ≠ *l′* ⟶ *a* ∈ *set* (*del aa* (*agra* (*graphI I*) *l*)) ⟶ *a* ∉ *set* (*agra* (*graphI I*) *l′a*)
by (*meson assms*(*2*) *atI-def del-notin-down*)
next show ⋀(*aa*::*char list*) (*l*::*location*) (*l′*::*location*) *la*::*location*.
I′ =
Infrastructure
(*Lgraph* (*gra* (*graphI I*))
(*if aa* ∉ *set* (*agra* (*graphI I*) *l′*)
then (*agra* (*graphI I*))(*l* := *del aa* (*agra* (*graphI I*) *l*), *l′* := *aa* # *agra* (*graphI I*) *l′*)
else agra (*graphI I*))
(*cgra* (*graphI I*)) (*lgra* (*graphI I*)))
(*delta I*) ⟹
aa ∈ *set* (*agra* (*graphI I*) *l*) ⟹
l ∈ *nodes* (*graphI I*) ⟹
l′ ∈ *nodes* (*graphI I*) ⟹
aa ∈ *actors-graph* (*graphI I*) ⟹
enables I l′ (*Actor aa*) *move* ⟹
aa ≠ *a* ⟹
aa ∉ *set* (*agra* (*graphI I*) *l′*) ⟹
la ≠ *l* ⟶
(*la* = *l′* ⟶
(∀ *l′a*::*location*.
(*l′a* = *l* ⟶
l ≠ *l′* ⟶ *a* ∈ *set* (*agra* (*graphI I*) *l′*) ⟶ *a* ∉ *set* (*del aa* (*agra* (*graphI I*) *l*))) ∧
(*l′a* ≠ *l* ⟶
l′a ≠ *l′* ⟶ *a* ∈ *set* (*agra* (*graphI I*) *l′*) ⟶ *a* ∉ *set* (*agra* (*graphI I*) *l′a*)))) ∧
(*la* ≠ *l′* ⟶
(∀ *l′a*::*location*.
(*l′a* = *l* ⟶
(*l* = *l′* ⟶ *a* ∈ *set* (*agra* (*graphI I*) *la*) ⟶ *a* ∉ *set* (*agra* (*graphI I*) *l′*)) ∧
(*l* ≠ *l′* ⟶ *a* ∈ *set* (*agra* (*graphI I*) *la*) ⟶ *a* ∉ *set* (*del aa* (*agra* (*graphI I*) *l*)))) ∧

$(l'a \neq l \longrightarrow$
$(l'a = l' \longrightarrow a \in set\ (agra\ (graphI\ I)\ la) \longrightarrow a \notin set\ (agra\ (graphI\ I)\ l')) \wedge$
$(l'a \neq l' \longrightarrow$
$a \in set\ (agra\ (graphI\ I)\ la) \wedge a \in set\ (agra\ (graphI\ I)\ l'a) \longrightarrow la = l'a))))$
by (*meson assms(2) atI-def del-notin-down*)
next show $\bigwedge(aa::char\ list)\ (l::location)\ l'::location.$
$I' =$
Infrastructure
(*Lgraph* (*gra* (*graphI I*))
(*if aa* $\notin$ *set* (*agra* (*graphI I*) l')
then (*agra* (*graphI I*))(l := *del aa* (*agra* (*graphI I*) l), l' := *aa* # *agra* (*graphI I*) l')
else agra (*graphI I*))
(*cgra* (*graphI I*)) (*lgra* (*graphI I*)))
(*delta I*) $\Longrightarrow$
$aa \in set\ (agra\ (graphI\ I)\ l) \Longrightarrow$
$l \in nodes\ (graphI\ I) \Longrightarrow$
$l' \in nodes\ (graphI\ I) \Longrightarrow$
$aa \in actors\text{-}graph\ (graphI\ I) \Longrightarrow$
$enables\ I\ l'\ (Actor\ aa)\ move \Longrightarrow$
$aa \neq a \Longrightarrow$
$aa \notin set\ (agra\ (graphI\ I)\ l') \Longrightarrow$
$\forall\ la::location.$
$(la = l \longrightarrow$
$(l = l' \longrightarrow nodup\ a\ (agra\ (graphI\ I)\ l')) \wedge$
$(l \neq l' \longrightarrow nodup\ a\ (del\ aa\ (agra\ (graphI\ I)\ l)))) \wedge$
$(la \neq l \longrightarrow$
$(la = l' \longrightarrow nodup\ a\ (agra\ (graphI\ I)\ l')) \wedge (la \neq l' \longrightarrow nodup\ a\ (agra\ (graphI\ I)\ la)))$
by (*simp add*: *assms(2) nodup-down-notin*)
next show $\bigwedge(aa::char\ list)\ (l::location)\ l'::location.$
$I' =$
Infrastructure
(*Lgraph* (*gra* (*graphI I*))
(*if aa* $\notin$ *set* (*agra* (*graphI I*) l')
then (*agra* (*graphI I*))(l := *del aa* (*agra* (*graphI I*) l), l' := *aa* # *agra* (*graphI I*) l')
else agra (*graphI I*))
(*cgra* (*graphI I*)) (*lgra* (*graphI I*)))
(*delta I*) $\Longrightarrow$
$aa \in set\ (agra\ (graphI\ I)\ l) \Longrightarrow$
$l \in nodes\ (graphI\ I) \Longrightarrow$
$l' \in nodes\ (graphI\ I) \Longrightarrow$
$aa \in actors\text{-}graph\ (graphI\ I) \Longrightarrow$
$enables\ I\ l'\ (Actor\ aa)\ move \Longrightarrow$
$aa \neq a \Longrightarrow$
$aa \in set\ (agra\ (graphI\ I)\ l') \longrightarrow$

$(\forall (l::location)\ l'::location.$
$a \in set\ (agra\ (graphI\ I)\ l) \wedge a \in set\ (agra\ (graphI\ I)\ l') \longrightarrow l = l') \wedge$
$(\forall\, l::location.\ nodup\ a\ (agra\ (graphI\ I)\ l))$
using *assms(2) atI-def* **by** *blast*
qed
qed
qed

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 –
have *ind*: $(\forall\ a.\ (\forall\ l\ l'.\ a\ @_{graphI\ I}\ l \wedge a\ @_{graphI\ I}\ l' \longrightarrow l = l') \wedge$
$(\forall\ l.\ nodup\ a\ (agra\ (graphI\ I)\ l))) \longrightarrow$
$(\forall\ a.\ (\forall\ l\ l'.\ a\ @_{graphI\ I'}\ l \wedge a\ @_{graphI\ I'}\ l' \longrightarrow l = l') \wedge$
$(\forall\ l.\ nodup\ a\ (agra\ (graphI\ I')\ l)))$
proof (*insert assms(1)*, *erule rtrancl.induct*)
show $\bigwedge a::infrastructure.$
$(\forall\, aa::char\ list.$
$(\forall\, (l::location)\ l'::location.\ aa\ @_{graphI\ a}\ l \wedge aa\ @_{graphI\ a}\ l' \longrightarrow l = l') \wedge$
$(\forall\, l::location.\ nodup\ aa\ (agra\ (graphI\ a)\ l))) \longrightarrow$
$(\forall\, aa::char\ list.$
$(\forall\, (l::location)\ l'::location.\ aa\ @_{graphI\ a}\ l \wedge aa\ @_{graphI\ a}\ l' \longrightarrow l = l') \wedge$
$(\forall\, l::location.\ nodup\ aa\ (agra\ (graphI\ a)\ l)))$ **by** *simp*
next show $\bigwedge (a::infrastructure)\ (b::infrastructure)\ (c::infrastructure).$
$(a,\ b) \in \{(x::infrastructure,\ y::infrastructure).\ x \rightarrow_n y\}^* \Longrightarrow$
$(\forall\, aa::char\ list.$
$(\forall\, (l::location)\ (l'::location).\ (aa\ @_{graphI\ a}\ l \wedge aa\ @_{graphI\ a}\ l') \longrightarrow l =$
$l') \wedge$
$(\forall\, l::location.\ nodup\ aa\ (agra\ (graphI\ a)\ l))) \longrightarrow$
$(\forall\, a::char\ list.$
$(\forall\, (l::location)\ (l'::location).\ (a\ @_{graphI\ b}\ l \wedge a\ @_{graphI\ b}\ l') \longrightarrow l = l') \wedge$
$(\forall\, l::location.\ nodup\ a\ (agra\ (graphI\ b)\ l))) \Longrightarrow$
$(b,\ c) \in \{(x::infrastructure,\ y::infrastructure).\ x \rightarrow_n y\} \Longrightarrow$
$(\forall\, aa::char\ list.$
$(\forall\, (l::location)\ l'::location.\ (aa\ @_{graphI\ a}\ l \wedge aa\ @_{graphI\ a}\ l') \longrightarrow l = l')$
$\wedge$
$(\forall\, l::location.\ nodup\ aa\ (agra\ (graphI\ a)\ l))) \longrightarrow$
$(\forall\, a::char\ list.$
$(\forall\, (l::location)\ l'::location.\ (a\ @_{graphI\ c}\ l \wedge a\ @_{graphI\ c}\ l') \longrightarrow l = l') \wedge$
$(\forall\, l::location.\ nodup\ a\ (agra\ (graphI\ c)\ l)))$
by (*rule impI*, *rule allI*, *rule actors-unique-loc-base*, *drule CollectD*,
simp,*erule impE*, *assumption*, *erule spec*)
qed

show *?thesis*
by (*insert ind*, *insert assms*(*2*), *simp*)
qed

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 (*simp add*: *Airplane-not-in-danger-init-def ex-graph-def*, *clarify*, *rule conjI*, *clarify*,
rule conjI, *clarify*, *rule impI*, (*rule allI*)+, *rule impI*, *simp add*: *atI-def*)
show $\bigwedge(l{::}location)\ l'{::}location.$
''Charly''
$\in$ *set* (*if* l = *cockpit then* [*''Bob''*, *''Charly''*]
else if l = *door then* [] *else if* l = *cabin then* [*''Alice''*] *else* []) $\wedge$
''Charly''
$\in$ *set* (*if* l' = *cockpit then* [*''Bob''*, *''Charly''*]
else if l' = *door then* [] *else if* l' = *cabin then* [*''Alice''*] *else* []) $\Longrightarrow$
$l = l'$
by (*case-tac* $l = l'$, *assumption*, *rule FalseE*, *case-tac* l = *cockpit* $\vee$ l = *door* $\vee$ l = *cabin*,
erule disjE, *simp*, *case-tac* l' = *door* $\vee$ l' = *cabin*, *erule disjE*, *simp*,
simp add: *cabin-def door-def*, *simp*, *erule disjE*, *simp add*: *door-def cockpit-def*,

simp add: *cabin-def door-def cockpit-def*, *simp*)
next show $\bigwedge a{::}char\ list.$
''Charly'' $\neq a \longrightarrow$
$(\forall\,(l{::}location)\ l'{::}location.$
$a\ @_{Lgraph\ \{(cockpit,\ door),\ (door,\ cabin)\}}$ $(\lambda x{::}location.$ *if* x = *cockpit then* [*''Bob*
$l\ \wedge$
$a\ @_{Lgraph\ \{(cockpit,\ door),\ (door,\ cabin)\}}$ $(\lambda x{::}location.$ *if* x = *cockpit then* [*''Bob*
$l' \longrightarrow$
$l = l')$
by (*clarify*, *simp add*: *atI-def*, *case-tac* $l = l'$, *assumption*, *rule FalseE*,
case-tac l = *cockpit* $\vee$ l = *door* $\vee$ l = *cabin*, *erule disjE*, *simp*,
case-tac l' = *door* $\vee$ l' = *cabin*, *erule disjE*, *simp*, *simp add*: *cabin-def door-def*,
simp, *erule disjE*, *simp add*: *door-def cockpit-def*, *case-tac* l = *cockpit*,
simp add: *cabin-def cockpit-def*, *simp add*: *cabin-def door-def*, *case-tac* l' = *cockpit*,
simp, *simp add*: *cabin-def*, *case-tac* l' = *door*, *simp*, *simp add*: *cabin-def*, *simp*)
qed

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

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 (graphI Airplane-not-in-danger-init) =
airplane-actors
proof (simp add: Airplane-not-in-danger-init-def ex-graph-def actors-graph-def nodes-def

                airplane-actors-def, rule equalityI)
  show {x::char list.
     ∃ y::location.
        (y = door ⟶
         (door = cockpit ⟶
           (∃ y::location. y = cockpit ∨ y = cabin ∨ y = cockpit ∨ y = cockpit ∧
cockpit = cabin) ∧
           (x = ''Bob'' ∨ x = ''Charly'')) ∧
         door = cockpit) ∧
        (y ≠ door ⟶
         (y = cockpit ⟶
          (∃ y::location.
              y = door ∨
              cockpit = door ∧ y = cabin ∨
              y = cockpit ∧ cockpit = door ∨ y = door ∧ cockpit = cabin) ∧
          (x = ''Bob'' ∨ x = ''Charly'')) ∧
         (y ≠ cockpit ⟶ y = cabin ∧ x = ''Alice'' ∧ y = cabin))}
    ⊆ {''Bob'', ''Charly'', ''Alice''}
  by (rule subsetI, drule CollectD, erule exE, (erule conjE)+,
      simp add: door-def cockpit-def cabin-def, (erule conjE)+, force)
next show {''Bob'', ''Charly'', ''Alice''}
    ⊆ {x::char list.
        ∃ y::location.
          (y = door ⟶
           (door = cockpit ⟶
            (∃ y::location.
                 y = cockpit ∨ y = cabin ∨ y = cockpit ∨ y = cockpit ∧ cockpit =
cabin) ∧
            (x = ''Bob'' ∨ x = ''Charly'')) ∧
           door = cockpit) ∧
          (y ≠ door ⟶
           (y = cockpit ⟶
            (∃ y::location.
                y = door ∨
                cockpit = door ∧ y = cabin ∨
                y = cockpit ∧ cockpit = door ∨ y = door ∧ cockpit = cabin) ∧
            (x = ''Bob'' ∨ x = ''Charly'')) ∧
           (y ≠ cockpit ⟶ y = cabin ∧ x = ''Alice'' ∧ y = cabin))}
  by (rule subsetI, rule CollectI, simp add: door-def cockpit-def cabin-def,
      case-tac x = ''Bob'', force, case-tac x = ''Charly'', force,
      subgoal-tac x = ''Alice'', force, simp)
qed
```

lemma *all-airplane-actors*: $(Airplane\text{-}not\text{-}in\text{-}danger\text{-}init,\ y) \in \{(x{::}infrastructure,$
$y{::}infrastructure).\ x \rightarrow_n y\}^*$
$\Longrightarrow actors\text{-}graph(graphI\ y) = airplane\text{-}actors$
by (*insert Anid-airplane-actors*, *erule subst*, *rule sym*, *erule same-actors*)

lemma *actors-at-loc-in-graph*: $[\![\ l \in nodes(graphI\ I);\ a\ \ @_{graphI\ I}\ l]\!]$
$\Longrightarrow a \in actors\text{-}graph\ (graphI\ I)$
by (*simp add*: *atI-def actors-graph-def*, *rule exI*, *rule conjI*)

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 –
have *delta y* = *delta*(*Airplane-not-in-danger-init*)
by (*insert assms*, *rule sym*, *erule-tac init-state-policy*)
with *assms* **show** *?thesis*
by (*simp add*: *Airplane-not-in-danger-init-def enables-def local-policies-four-eyes-def*)

qed

lemma *Apnid-tsp-test*: $\sim$(*enables Airplane-not-in-danger-init cockpit* (*Actor ''Alice''*)
get)
by (*simp add*: *Airplane-not-in-danger-init-def ex-creds-def enables-def*
local-policies-four-eyes-def cabin-def door-def cockpit-def
ex-graph-def ex-locs-def)

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

by (*simp add*: *Airplane-not-in-danger-init-def ex-creds-def enables-def*
local-policies-four-eyes-def cabin-def door-def cockpit-def
ex-graph-def ex-locs-def)

lemma *test-graph-atI*: $''Bob''\ @_{graphI\ Airplane\text{-}not\text{-}in\text{-}danger\text{-}init}\ cockpit$
by (*simp add*: *Airplane-not-in-danger-init-def ex-graph-def atI-def*)

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-not-in-danger-init*)
and *delta*(*z*) = *delta*(*Airplane-not-in-danger-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 (*insert assms*(*5*), *erule state-transition-in.cases*)
show $\bigwedge(G{::}igraph)\ (I{::}infrastructure)\ (a{::}char\ list)\ (l{::}location)\ (a'{::}char\ list)$

(za::*char list*)
I'::*infrastructure*.
$z = I \Longrightarrow$
$z' = I' \Longrightarrow$
$G = graphI\ I \Longrightarrow$
$a\ @_G\ l \Longrightarrow$
$a'\ @_G\ l \Longrightarrow$
has G (*Actor* a, za) $\Longrightarrow$
enables $I\ l$ (*Actor* a) *get* $\Longrightarrow$
$I' =$
Infrastructure
(*Lgraph* (*gra* G) (*agra* G)
((*cgra* G)(*Actor* a' := (za # *fst* (*cgra* G (*Actor* a')), *snd* (*cgra* G (*Actor* a'))))) (*lgra* G))
(*delta* I) $\Longrightarrow$
(*2*::*nat*) $\leq$ *length* (*agra* (*graphI* z') *cockpit*) **using** *assms* **by** *simp*
next show $\bigwedge$(G::*igraph*) (I::*infrastructure*) (a::*char list*) (l::*location*) (I'::*infrastructure*)
za::*char list*.
$z = I \Longrightarrow$
$z' = I' \Longrightarrow$
$G = graphI\ I \Longrightarrow$
$a\ @_G\ l \Longrightarrow$
enables $I\ l$ (*Actor* a) *put* $\Longrightarrow$
I' = *Infrastructure* (*Lgraph* (*gra* G) (*agra* G) (*cgra* G) ((*lgra* G)(l := [za])))
(*delta* I) $\Longrightarrow$
(*2*::*nat*) $\leq$ *length* (*agra* (*graphI* z') *cockpit*) **using** *assms* **by** *simp*
next show $\bigwedge$(G::*igraph*) (I::*infrastructure*) (l::*location*) (a::*char list*) (I'::*infrastructure*)
za::*char list*.
$z = I \Longrightarrow$
$z' = I' \Longrightarrow$
$G = graphI\ I \Longrightarrow$
enables $I\ l$ (*Actor* a) *put* $\Longrightarrow$
I' = *Infrastructure* (*Lgraph* (*gra* G) (*agra* G) (*cgra* G) ((*lgra* G)(l := [za])))
(*delta* I) $\Longrightarrow$
(*2*::*nat*) $\leq$ *length* (*agra* (*graphI* z') *cockpit*) **using** *assms* **by** *simp*
next show $\bigwedge$(G::*igraph*) (I::*infrastructure*) (a::*char list*) (l::*location*) (l'::*location*)
I'::*infrastructure*.
$z = I \Longrightarrow$
$z' = I' \Longrightarrow$
$G = graphI\ I \Longrightarrow$
$a\ @_G\ l \Longrightarrow$
$l \in nodes\ G \Longrightarrow$
$l' \in nodes\ G \Longrightarrow$
$a \in$ *actors-graph* (*graphI* I) $\Longrightarrow$
enables $I\ l'$ (*Actor* a) *move* $\Longrightarrow$
I' = *Infrastructure* (*move-graph-a* $a\ l\ l'$ (*graphI* I)) (*delta* I) $\Longrightarrow$
(*2*::*nat*) $\leq$ *length* (*agra* (*graphI* z') *cockpit*)
proof −
fix G :: *igraph* **and** I :: *infrastructure* **and** a :: *char list* **and** l :: *location* **and** l'

:: *location* **and** I' :: *infrastructure*
  have *f1*: *UasI ''Eve'' ''Charly''*
    using *Eve-precipitating-event Insider-Eve Insider-def* **by** *force*
  obtain *ccs* :: *char list* ⇒ *char list* **and** *ccsa* :: *char list* ⇒ *char list* **where**
    f2: ∀ *cs csa*. (¬ *UasI cs csa* ∨ *Actor cs* = *Actor csa* ∧ (∀ *csa csb*. (*csa* = *cs* ∨ *csb* = *cs* ∨ *Actor csa* ≠ *Actor csb*) ∨ *csa* = *csb*)) ∧ (*UasI cs csa* ∨ *Actor cs* ≠ *Actor csa* ∨ (*ccs cs* ≠ *cs* ∧ *ccsa cs* ≠ *cs* ∧ *Actor* (*ccs cs*) = *Actor* (*ccsa cs*)) ∧ *ccs cs* ≠ *ccsa cs*)
    using *UasI-def* **by** *moura*
  have *''Bob''* $@_{graphI}$ (*Infrastructure ex-graph local-policies*) *Location 2*
    using *Airplane-not-in-danger-init-def cockpit-def test-graph-atI* **by** *force*
  then have *Actor ''Bob''* = *Actor ''Eve''*
   using *Airplane-scenario-def airplane.cockpit-foe-control airplane-axioms cockpit-def ex-inv3 global-policy-def* **by** *blast*
  then show *2* ≤ *length* (*agra* (*graphI* z') *cockpit*)
    using *f2 f1* **by** *auto*
qed
qed

lemma *two-person-inv1*:
 assumes (*Airplane-not-in-danger-init*,*z*) ∈ {(*x*::*infrastructure*, *y*::*infrastructure*). $x \to_n y\}^*$
  shows (*2*::*nat*) ≤ *length* (*agra* (*graphI z*) *cockpit*)
proof (*insert assms*, *erule rtrancl-induct*)
  show (*2*::*nat*) ≤ *length* (*agra* (*graphI Airplane-not-in-danger-init*) *cockpit*)
  by (*simp add*: *Airplane-not-in-danger-init-def ex-graph-def*)
next show ⋀(*y*::*infrastructure*) *z*::*infrastructure*.
        (*Airplane-not-in-danger-init*, *y*) ∈ {(*x*::*infrastructure*, *y*::*infrastructure*). $x \to_n y\}^*$ ⟹
       (*y*, *z*) ∈ {(*x*::*infrastructure*, *y*::*infrastructure*). $x \to_n y$} ⟹
         (*2*::*nat*) ≤ *length* (*agra* (*graphI y*) *cockpit*) ⟹ (*2*::*nat*) ≤ *length* (*agra* (*graphI z*) *cockpit*)
    by (*rule two-person-inv*, *assumption*, *rule same-nodes*, *assumption*, *rule sym*,
        rule init-state-policy, *assumption*+, *simp*)
qed

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 ∉ *set l* ⟶ *card* (*insert a* (*set l*)) = *Suc* (*card* (*set l*))
by *auto*

lemma *no-dup-set-list-num-eq*[*rule-format*]:
    (∀ *a*. *nodup a l*) ⟶ *card* (*set l*) = *length l*
  by (*induct-tac l*, *simp*, *clarify*, *simp*, *erule impE*, *rule allI*,

drule-tac x = aa **in** *spec*, *case-tac a = aa*, *simp*, *erule nodup-notin*, *simp*)

lemma *two-person-set-inv*:
assumes (*Airplane-not-in-danger-init*,*z*) $\in$ {(*x*::*infrastructure*, *y*::*infrastructure*). *x* $\rightarrow_n$ *y*}*
shows (*2*::*nat*) $\leq$ *card* (*set* (*agra* (*graphI z*) *cockpit*))
proof −
have *a*: *card* (*set* (*agra* (*graphI z*) *cockpit*)) = *length*(*agra* (*graphI z*) *cockpit*)
by (*rule no-dup-set-list-num-eq*, *insert assms*, *drule actors-unique-loc-aid-step*, *drule-tac x = a* **in** *spec*, *erule conjE*, *erule-tac x = cockpit* **in** *spec*)
show *?thesis*
by (*insert a*, *erule ssubst*, *rule two-person-inv1*, *rule assms*)
qed

lemma *Pred-all-unique*: $\llbracket$ *? x. P x*; (! *x. P x* $\longrightarrow$ *x = c*)$\rrbracket$ $\Longrightarrow$ *P c*
by (*case-tac P c*, *assumption*, *erule exE*, *drule-tac x = x* **in** *spec*, *drule mp*, *assumption*, *erule subst*)

lemma *Set-all-unique*: $\llbracket$ *S* $\neq$ {}; ($\forall$ *x* $\in$ *S*. *x* = *c*) $\rrbracket$ $\Longrightarrow$ *c* $\in$ *S*
by (*rule-tac P* = λ *x*. *x* $\in$ *S* **in** *Pred-all-unique*, *force*, *simp*)

lemma *airplane-actors-inv0*[*rule-format*]:
$\forall$ *z z'*. ($\forall$ *h*::*char list* $\in$ *set* (*agra* (*graphI z*) *cockpit*). *h* $\in$ *airplane-actors*) $\wedge$
(*Airplane-not-in-danger-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-actors*)
proof (*clarify*, *erule state-transition-in.cases*)
show $\bigwedge$(*z*::*infrastructure*) (*z'*::*infrastructure*) (*h*::*char list*) (*G*::*igraph*) (*I*::*infrastructure*) (*a*::*char list*) (*l*::*location*) (*a'*::*char list*) (*za*::*char list*) *I'*::*infrastructure*.
h $\in$ *set* (*agra* (*graphI z'*) *cockpit*) $\Longrightarrow$
$\forall$ *h*::*char list*$\in$*set* (*agra* (*graphI z*) *cockpit*). *h* $\in$ *airplane-actors* $\Longrightarrow$
(*Airplane-not-in-danger-init*, *z*) $\in$ {(*x*::*infrastructure*, *y*::*infrastructure*). *x* $\rightarrow_n$ *y*}* $\Longrightarrow$
z = *I* $\Longrightarrow$
z' = *I'* $\Longrightarrow$
G = *graphI I* $\Longrightarrow$
a $@_G$ *l* $\Longrightarrow$
a' $@_G$ *l* $\Longrightarrow$
has G (*Actor a*, *za*) $\Longrightarrow$
enables I l (*Actor a*) *get* $\Longrightarrow$
I' =
Infrastructure
(*Lgraph* (*gra G*) (*agra G*)
((*cgra G*)(*Actor a'* := (*za* # *fst* (*cgra G* (*Actor a'*)), *snd* (*cgra G* (*Actor a'*))))) (*lgra G*))
(*delta I*) $\Longrightarrow$
h $\in$ *airplane-actors*
by *simp*

next show $\bigwedge(z::infrastructure)$ $(z'::infrastructure)$ $(h::char\ list)$ $(G::igraph)$ $(I::infrastructure)$
$(a::char\ list)$ $(l::location)$ $(I'::infrastructure)$ $za::char\ list.$
$h \in set\ (agra\ (graphI\ z')\ cockpit) \Longrightarrow$
$\forall h::char\ list{\in}set\ (agra\ (graphI\ z)\ cockpit).\ h \in airplane\text{-}actors \Longrightarrow$
$(Airplane\text{-}not\text{-}in\text{-}danger\text{-}init,\ z) \in \{(x::infrastructure,\ y::infrastructure).\ x$
$\rightarrow_n\ y\}^* \Longrightarrow$
$z = I \Longrightarrow$
$z' = I' \Longrightarrow$
$G = graphI\ I \Longrightarrow$
$a\ @_G\ l \Longrightarrow$
$enables\ I\ l\ (Actor\ a)\ put \Longrightarrow$
$I' = Infrastructure\ (Lgraph\ (gra\ G)\ (agra\ G)\ (cgra\ G)\ ((lgra\ G)(l := [za])))$
$(delta\ I) \Longrightarrow$
$h \in airplane\text{-}actors$
by *simp*
next show $\bigwedge(z::infrastructure)$ $(z'::infrastructure)$ $(h::char\ list)$ $(G::igraph)$ $(I::infrastructure)$
$(l::location)$ $(a::char\ list)$ $(I'::infrastructure)$ $za::char\ list.$
$h \in set\ (agra\ (graphI\ z')\ cockpit) \Longrightarrow$
$\forall h::char\ list{\in}set\ (agra\ (graphI\ z)\ cockpit).\ h \in airplane\text{-}actors \Longrightarrow$
$(Airplane\text{-}not\text{-}in\text{-}danger\text{-}init,\ z) \in \{(x::infrastructure,\ y::infrastructure).\ x$
$\rightarrow_n\ y\}^* \Longrightarrow$
$z = I \Longrightarrow$
$z' = I' \Longrightarrow$
$G = graphI\ I \Longrightarrow$
$enables\ I\ l\ (Actor\ a)\ put \Longrightarrow$
$I' = Infrastructure\ (Lgraph\ (gra\ G)\ (agra\ G)\ (cgra\ G)\ ((lgra\ G)(l := [za])))$
$(delta\ I) \Longrightarrow$
$h \in airplane\text{-}actors$
by *simp*
next show $\bigwedge(z::infrastructure)$ $(z'::infrastructure)$ $(h::char\ list)$ $(G::igraph)$ $(I::infrastructure)$
$(a::char\ list)$ $(l::location)$ $(l'::location)$ $I'::infrastructure.$
$h \in set\ (agra\ (graphI\ z')\ cockpit) \Longrightarrow$
$\forall h::char\ list{\in}set\ (agra\ (graphI\ z)\ cockpit).\ h \in airplane\text{-}actors \Longrightarrow$
$(Airplane\text{-}not\text{-}in\text{-}danger\text{-}init,\ z) \in \{(x::infrastructure,\ y::infrastructure).\ x$
$\rightarrow_n\ y\}^* \Longrightarrow$
$z = I \Longrightarrow$
$z' = I' \Longrightarrow$
$G = graphI\ I \Longrightarrow$
$a\ @_G\ l \Longrightarrow$
$l \in nodes\ G \Longrightarrow$
$l' \in nodes\ G \Longrightarrow$
$a \in actors\text{-}graph\ (graphI\ I) \Longrightarrow$
$enables\ I\ l'\ (Actor\ a)\ move \Longrightarrow$
$I' = Infrastructure\ (move\text{-}graph\text{-}a\ a\ l\ l'\ (graphI\ I))\ (delta\ I) \Longrightarrow h \in$
$airplane\text{-}actors$
proof $(simp\ add:\ move\text{-}graph\text{-}a\text{-}def,$
$case\text{-}tac\ a \in set\ (agra\ (graphI\ I)\ l) \land a \notin set\ (agra\ (graphI\ I)\ l'))$
show $\bigwedge(z::infrastructure)$ $(z'::infrastructure)$ $(h::char\ list)$ $(G::igraph)$ $(I::infrastructure)$
$(a::char\ list)$ $(l::location)$ $(l'::location)$ $I'::infrastructure.$

h ∈ *set* ((*if a* ∈ *set* (*agra* (*graphI I*) *l*) ∧ *a* ∉ *set* (*agra* (*graphI I*) *l′*)
then (*agra* (*graphI I*))
(*l* := *del a* (*agra* (*graphI I*) *l*), *l′* := *a* # *agra* (*graphI I*) *l′*)
else agra (*graphI I*))
cockpit) ⟹
∀ *h*::*char list*∈*set* (*agra* (*graphI I*) *cockpit*). *h* ∈ *airplane-actors* ⟹
(*Airplane-not-in-danger-init*, *I*) ∈ {(*x*::*infrastructure*, *y*::*infrastructure*). *x*
$\rightarrow_n$ *y*$\}^*$ ⟹
z = *I* ⟹
z′ =
Infrastructure
(*Lgraph* (*gra* (*graphI I*))
(*if a* ∈ *set* (*agra* (*graphI I*) *l*) ∧ *a* ∉ *set* (*agra* (*graphI I*) *l′*)
then (*agra* (*graphI I*))(*l* := *del a* (*agra* (*graphI I*) *l*), *l′* := *a* # *agra*
(*graphI I*) *l′*)
else agra (*graphI I*))
(*cgra* (*graphI I*)) (*lgra* (*graphI I*)))
(*delta I*) ⟹
G = *graphI I* ⟹
a $@_{graphI\ I}$ *l* ⟹
l ∈ *nodes* (*graphI I*) ⟹
l′ ∈ *nodes* (*graphI I*) ⟹
a ∈ *actors-graph* (*graphI I*) ⟹
enables I l′ (*Actor a*) *move* ⟹
I′ =
Infrastructure
(*Lgraph* (*gra* (*graphI I*))
(*if a* ∈ *set* (*agra* (*graphI I*) *l*) ∧ *a* ∉ *set* (*agra* (*graphI I*) *l′*)
then (*agra* (*graphI I*))(*l* := *del a* (*agra* (*graphI I*) *l*), *l′* := *a* # *agra*
(*graphI I*) *l′*)
else agra (*graphI I*))
(*cgra* (*graphI I*)) (*lgra* (*graphI I*)))
(*delta I*) ⟹
¬ (*a* ∈ *set* (*agra* (*graphI I*) *l*) ∧ *a* ∉ *set* (*agra* (*graphI I*) *l′*)) ⟹ *h* ∈
airplane-actors
by *simp*
next show ⋀(*z*::*infrastructure*) (*z′*::*infrastructure*) (*h*::*char list*) (*G*::*igraph*)
(*I*::*infrastructure*)
(*a*::*char list*) (*l*::*location*) (*l′*::*location*) *I′*::*infrastructure*.
h ∈ *set* ((*if a* ∈ *set* (*agra* (*graphI I*) *l*) ∧ *a* ∉ *set* (*agra* (*graphI I*) *l′*)
then (*agra* (*graphI I*))
(*l* := *del a* (*agra* (*graphI I*) *l*), *l′* := *a* # *agra* (*graphI I*) *l′*)
else agra (*graphI I*))
cockpit) ⟹
∀ *h*::*char list*∈*set* (*agra* (*graphI I*) *cockpit*). *h* ∈ *airplane-actors* ⟹
(*Airplane-not-in-danger-init*, *I*) ∈ {(*x*::*infrastructure*, *y*::*infrastructure*). *x*
$\rightarrow_n$ *y*$\}^*$ ⟹
z = *I* ⟹
z′ =

Infrastructure
(*Lgraph* (*gra* (*graphI I*))
(*if a* $\in$ *set* (*agra* (*graphI I*) *l*) $\wedge$ *a* $\notin$ *set* (*agra* (*graphI I*) *l'*)
then (*agra* (*graphI I*))(*l* := *del a* (*agra* (*graphI I*) *l*), *l'* := *a* # *agra* (*graphI I*) *l'*)
else agra (*graphI I*))
(*cgra* (*graphI I*)) (*lgra* (*graphI I*)))
(*delta I*) $\Longrightarrow$
G = *graphI I* $\Longrightarrow$
a $@_{graphI\ I}$ *l* $\Longrightarrow$
l $\in$ *nodes* (*graphI I*) $\Longrightarrow$
l' $\in$ *nodes* (*graphI I*) $\Longrightarrow$
a $\in$ *actors-graph* (*graphI I*) $\Longrightarrow$
enables I l' (*Actor a*) *move* $\Longrightarrow$
I' =
Infrastructure
(*Lgraph* (*gra* (*graphI I*))
(*if a* $\in$ *set* (*agra* (*graphI I*) *l*) $\wedge$ *a* $\notin$ *set* (*agra* (*graphI I*) *l'*)
then (*agra* (*graphI I*))(*l* := *del a* (*agra* (*graphI I*) *l*), *l'* := *a* # *agra* (*graphI I*) *l'*)
else agra (*graphI I*))
(*cgra* (*graphI I*)) (*lgra* (*graphI I*)))
(*delta I*) $\Longrightarrow$
a $\in$ *set* (*agra* (*graphI I*) *l*) $\wedge$ *a* $\notin$ *set* (*agra* (*graphI I*) *l'*) $\Longrightarrow$ *h* $\in$ *airplane-actors*
proof (*case-tac l'* = *cockpit*)
show $\bigwedge$(*z*::*infrastructure*) (*z'*::*infrastructure*) (*h*::*char list*) (*G*::*igraph*) (*I*::*infrastructure*) (*a*::*char list*) (*l*::*location*) (*l'*::*location*) *I'*::*infrastructure*.
h $\in$ *set* ((*if a* $\in$ *set* (*agra* (*graphI I*) *l*) $\wedge$ *a* $\notin$ *set* (*agra* (*graphI I*) *l'*)
then (*agra* (*graphI I*))
(*l* := *del a* (*agra* (*graphI I*) *l*), *l'* := *a* # *agra* (*graphI I*) *l'*)
else agra (*graphI I*))
cockpit) $\Longrightarrow$
$\forall$ *h*::*char list*$\in$*set* (*agra* (*graphI I*) *cockpit*). *h* $\in$ *airplane-actors* $\Longrightarrow$
(*Airplane-not-in-danger-init*, *I*) $\in$ {(*x*::*infrastructure*, *y*::*infrastructure*). *x* $\to_n$ *y*}* $\Longrightarrow$
z = *I* $\Longrightarrow$
z' =
Infrastructure
(*Lgraph* (*gra* (*graphI I*))
(*if a* $\in$ *set* (*agra* (*graphI I*) *l*) $\wedge$ *a* $\notin$ *set* (*agra* (*graphI I*) *l'*)
then (*agra* (*graphI I*))(*l* := *del a* (*agra* (*graphI I*) *l*), *l'* := *a* # *agra* (*graphI I*) *l'*)
else agra (*graphI I*))
(*cgra* (*graphI I*)) (*lgra* (*graphI I*)))
(*delta I*) $\Longrightarrow$
G = *graphI I* $\Longrightarrow$
a $@_{graphI\ I}$ *l* $\Longrightarrow$
l $\in$ *nodes* (*graphI I*) $\Longrightarrow$

l′ ∈ nodes (graphI I) ⟹
a ∈ actors-graph (graphI I) ⟹
enables I l′ (Actor a) move ⟹
I′ =
Infrastructure
(Lgraph (gra (graphI I))
(if a ∈ set (agra (graphI I) l) ∧ a ∉ set (agra (graphI I) l′)
then (agra (graphI I))(l := del a (agra (graphI I) l), l′ := a # agra (graphI I) l′)
else agra (graphI I))
(cgra (graphI I)) (lgra (graphI I)))
(delta I) ⟹
a ∈ set (agra (graphI I) l) ∧ a ∉ set (agra (graphI I) l′) ⟹
l′ ≠ cockpit ⟹ h ∈ airplane-actors
proof (*case-tac cockpit = l*)
show ⋀(*z::infrastructure*) (*z′::infrastructure*) (*h::char list*) (*G::igraph*) (*I::infrastructure*)
(*a::char list*) (*l::location*) (*l′::location*) *I′::infrastructure.*
h ∈ set ((if a ∈ set (agra (graphI I) l) ∧ a ∉ set (agra (graphI I) l′)
then (agra (graphI I))
(l := del a (agra (graphI I) l), l′ := a # agra (graphI I) l′)
else agra (graphI I))
cockpit) ⟹
∀ h::char list∈set (agra (graphI I) cockpit). h ∈ airplane-actors ⟹
(Airplane-not-in-danger-init, I) ∈ {(x::infrastructure, y::infrastructure). x $\rightarrow_n$ *y}* ⟹*
z = I ⟹
z′ =
Infrastructure
(Lgraph (gra (graphI I))
(if a ∈ set (agra (graphI I) l) ∧ a ∉ set (agra (graphI I) l′)
then (agra (graphI I))(l := del a (agra (graphI I) l), l′ := a # agra (graphI I) l′)
else agra (graphI I))
(cgra (graphI I)) (lgra (graphI I)))
(delta I) ⟹
G = graphI I ⟹
a $@_{graphI\ I}$ *l ⟹*
l ∈ nodes (graphI I) ⟹
l′ ∈ nodes (graphI I) ⟹
a ∈ actors-graph (graphI I) ⟹
enables I l′ (Actor a) move ⟹
I′ =
Infrastructure
(Lgraph (gra (graphI I))
(if a ∈ set (agra (graphI I) l) ∧ a ∉ set (agra (graphI I) l′)
then (agra (graphI I))(l := del a (agra (graphI I) l), l′ := a # agra (graphI I) l′)
else agra (graphI I))

$(cgra\ (graphI\ I))\ (lgra\ (graphI\ I)))$
$(delta\ I) \Longrightarrow$
$a \in set\ (agra\ (graphI\ I)\ l) \wedge a \notin set\ (agra\ (graphI\ I)\ l') \Longrightarrow$
$l' \neq cockpit \Longrightarrow cockpit \neq l \Longrightarrow h \in airplane\text{-}actors$
by *simp*
next show $\bigwedge(z::infrastructure)\ (z'::infrastructure)\ (h::char\ list)\ (G::igraph)$ $(I::infrastructure)$
$(a::char\ list)\ (l::location)\ (l'::location)\ I'::infrastructure.$
$h \in set\ ((if\ a \in set\ (agra\ (graphI\ I)\ l) \wedge a \notin set\ (agra\ (graphI\ I)\ l')$
$then\ (agra\ (graphI\ I))$
$(l := del\ a\ (agra\ (graphI\ I)\ l),\ l' := a\ \#\ agra\ (graphI\ I)\ l')$
$else\ agra\ (graphI\ I))$
$cockpit) \Longrightarrow$
$\forall h::char\ list{\in}set\ (agra\ (graphI\ I)\ cockpit).\ h \in airplane\text{-}actors \Longrightarrow$
$(Airplane\text{-}not\text{-}in\text{-}danger\text{-}init,\ I) \in \{(x::infrastructure,\ y::infrastructure).\ x \to_n y\}^* \Longrightarrow$
$z = I \Longrightarrow$
$z' =$
$Infrastructure$
$(Lgraph\ (gra\ (graphI\ I))$
$(if\ a \in set\ (agra\ (graphI\ I)\ l) \wedge a \notin set\ (agra\ (graphI\ I)\ l')$
$then\ (agra\ (graphI\ I))(l := del\ a\ (agra\ (graphI\ I)\ l),\ l' := a\ \#\ agra\ (graphI\ I)\ l')$
$else\ agra\ (graphI\ I))$
$(cgra\ (graphI\ I))\ (lgra\ (graphI\ I)))$
$(delta\ I) \Longrightarrow$
$G = graphI\ I \Longrightarrow$
$a\ @_{graphI\ I}\ l \Longrightarrow$
$l \in nodes\ (graphI\ I) \Longrightarrow$
$l' \in nodes\ (graphI\ I) \Longrightarrow$
$a \in actors\text{-}graph\ (graphI\ I) \Longrightarrow$
$enables\ I\ l'\ (Actor\ a)\ move \Longrightarrow$
$I' =$
$Infrastructure$
$(Lgraph\ (gra\ (graphI\ I))$
$(if\ a \in set\ (agra\ (graphI\ I)\ l) \wedge a \notin set\ (agra\ (graphI\ I)\ l')$
$then\ (agra\ (graphI\ I))(l := del\ a\ (agra\ (graphI\ I)\ l),\ l' := a\ \#\ agra\ (graphI\ I)\ l')$
$else\ agra\ (graphI\ I))$
$(cgra\ (graphI\ I))\ (lgra\ (graphI\ I)))$
$(delta\ I) \Longrightarrow$
$a \in set\ (agra\ (graphI\ I)\ l) \wedge a \notin set\ (agra\ (graphI\ I)\ l') \Longrightarrow$
$l' \neq cockpit \Longrightarrow cockpit = l \Longrightarrow h \in airplane\text{-}actors$
by (*simp*, *erule bspec*, *erule del-up*)
qed
next show $\bigwedge(z::infrastructure)\ (z'::infrastructure)\ (h::char\ list)\ (G::igraph)$ $(I::infrastructure)$
$(a::char\ list)\ (l::location)\ (l'::location)\ I'::infrastructure.$
$h \in set\ ((if\ a \in set\ (agra\ (graphI\ I)\ l) \wedge a \notin set\ (agra\ (graphI\ I)\ l')$

then $(agra\ (graphI\ I))$
$(l := del\ a\ (agra\ (graphI\ I)\ l),\ l' := a\ \#\ agra\ (graphI\ I)\ l')$
else $agra\ (graphI\ I))$
$cockpit) \Longrightarrow$
$\forall h{::}char\ list{\in}set\ (agra\ (graphI\ I)\ cockpit).\ h \in airplane\text{-}actors \Longrightarrow$
$(Airplane\text{-}not\text{-}in\text{-}danger\text{-}init,\ I) \in \{(x{::}infrastructure,\ y{::}infrastructure).\ x \rightarrow_n y\}^* \Longrightarrow$
$z = I \Longrightarrow$
$z' =$
Infrastructure
$(Lgraph\ (gra\ (graphI\ I))$
$(if\ a \in set\ (agra\ (graphI\ I)\ l) \wedge a \notin set\ (agra\ (graphI\ I)\ l')$
then $(agra\ (graphI\ I))(l := del\ a\ (agra\ (graphI\ I)\ l),\ l' := a\ \#\ agra\ (graphI\ I)\ l')$
else $agra\ (graphI\ I))$
$(cgra\ (graphI\ I))\ (lgra\ (graphI\ I)))$
$(delta\ I) \Longrightarrow$
$G = graphI\ I \Longrightarrow$
$a\ @_{graphI\ I}\ l \Longrightarrow$
$l \in nodes\ (graphI\ I) \Longrightarrow$
$l' \in nodes\ (graphI\ I) \Longrightarrow$
$a \in actors\text{-}graph\ (graphI\ I) \Longrightarrow$
$enables\ I\ l'\ (Actor\ a)\ move \Longrightarrow$
$I' =$
Infrastructure
$(Lgraph\ (gra\ (graphI\ I))$
$(if\ a \in set\ (agra\ (graphI\ I)\ l) \wedge a \notin set\ (agra\ (graphI\ I)\ l')$
then $(agra\ (graphI\ I))(l := del\ a\ (agra\ (graphI\ I)\ l),\ l' := a\ \#\ agra\ (graphI\ I)\ l')$
else $agra\ (graphI\ I))$
$(cgra\ (graphI\ I))\ (lgra\ (graphI\ I)))$
$(delta\ I) \Longrightarrow$
$a \in set\ (agra\ (graphI\ I)\ l) \wedge a \notin set\ (agra\ (graphI\ I)\ l') \Longrightarrow$
$l' = cockpit \Longrightarrow h \in airplane\text{-}actors$
proof (*simp*, *erule disjE*)
show $\bigwedge(z{::}infrastructure)\ (z'{::}infrastructure)\ (h{::}char\ list)\ (G{::}igraph)\ (I{::}infrastructure)$
$(a{::}char\ list)\ (l{::}location)\ (l'{::}location)\ I'{::}infrastructure.$
$\forall h{::}char\ list{\in}set\ (agra\ (graphI\ I)\ cockpit).\ h \in airplane\text{-}actors \Longrightarrow$
$(Airplane\text{-}not\text{-}in\text{-}danger\text{-}init,\ I) \in \{(x{::}infrastructure,\ y{::}infrastructure).\ x \rightarrow_n y\}^* \Longrightarrow$
$z = I \Longrightarrow$
$z' =$
Infrastructure
$(Lgraph\ (gra\ (graphI\ I))$
$((agra\ (graphI\ I))$
$(l := del\ a\ (agra\ (graphI\ I)\ l),\ cockpit := a\ \#\ agra\ (graphI\ I)\ cockpit))$
$(cgra\ (graphI\ I))\ (lgra\ (graphI\ I)))$
$(delta\ I) \Longrightarrow$

```
    G = graphI I ⟹
    a @graphI I l ⟹
    l ∈ nodes (graphI I) ⟹
    cockpit ∈ nodes (graphI I) ⟹
    a ∈ actors-graph (graphI I) ⟹
    enables I cockpit (Actor a) move ⟹
    I' =
    Infrastructure
     (Lgraph (gra (graphI I))
       ((agra (graphI I))
        (l := del a (agra (graphI I) l), cockpit := a # agra (graphI I) cockpit))
       (cgra (graphI I)) (lgra (graphI I)))
     (delta I) ⟹
    a ∈ set (agra (graphI I) l) ∧ a ∉ set (agra (graphI I) cockpit) ⟹
    l' = cockpit ⟹ h ∈ set (agra (graphI I) cockpit) ⟹ h ∈ airplane-actors
       by (erule bspec)
   next fix z z' h G I a l l' I'
    assume a0: ∀ h::char list∈set (agra (graphI I) cockpit). h ∈ airplane-actors
  and a1: (Airplane-not-in-danger-init, I) ∈ {(x::infrastructure, y::infrastructure).
x →n y}*
   and a2: z = I
   and a3: z' =
    Infrastructure
     (Lgraph (gra (graphI I))
       ((agra (graphI I))
        (l := del a (agra (graphI I) l), cockpit := a # agra (graphI I) cockpit))
       (cgra (graphI I)) (lgra (graphI I)))
     (delta I)
   and a4: G = graphI I
   and a5: a @graphI I l
   and a6: l ∈ nodes (graphI I)
   and a7: cockpit ∈ nodes (graphI I)
   and a8: a ∈ actors-graph (graphI I)
   and a9: enables I cockpit (Actor a) move
   and a10: I' =
    Infrastructure
     (Lgraph (gra (graphI I))
       ((agra (graphI I))
        (l := del a (agra (graphI I) l), cockpit := a # agra (graphI I) cockpit))
       (cgra (graphI I)) (lgra (graphI I)))
     (delta I)
    and a11: a ∈ set (agra (graphI I) l) ∧ a ∉ set (agra (graphI I) cockpit)
    and a12: l' = cockpit
    and a13: h = a
     show  h ∈ airplane-actors
    proof −
    have a: delta(I) = delta(Airplane-not-in-danger-init)
      by (rule sym, rule init-state-policy, rule a1)
    show ?thesis
```

by (*insert a0 a1 a2 a3 a4 a5 a6 a7 a8 a9 a10 a11 a12 a13 a*,
simp add: *enables-def*, *erule bexE*, *simp add*: *Airplane-not-in-danger-init-def*,
unfold local-policies-four-eyes-def, *simp*, *erule disjE*, *simp*+,

erule exE, (*erule conjE*)+,
fold local-policies-four-eyes-def Airplane-not-in-danger-init-def,
drule all-airplane-actors, *erule subst*)
qed
qed
qed
qed
qed

lemma *airplane-actors-inv*:
assumes (*Airplane-not-in-danger-init*,*z*) ∈ {(*x*::*infrastructure*, *y*::*infrastructure*). $x \to_n y\}^*$
shows ∀ *h*::*char list*∈*set* (*agra* (*graphI z*) *cockpit*). *h* ∈ *airplane-actors*
proof −
have *ind*: (*Airplane-not-in-danger-init*, *z*) ∈ {(*x*::*infrastructure*, *y*::*infrastructure*). $x \to_n y\}^* \longrightarrow$
(∀ *h*::*char list*∈*set* (*agra* (*graphI z*) *cockpit*). *h* ∈ *airplane-actors*)
proof (*insert assms*, *erule rtrancl-induct*)
show (*Airplane-not-in-danger-init*, *Airplane-not-in-danger-init*) ∈ {(*x*,*y*). $x \to_n y\}^* \longrightarrow$
(∀ *h*::*char list*∈*set* (*agra* (*graphI Airplane-not-in-danger-init*) *cockpit*). *h* ∈ *airplane-actors*)
by (*rule impI*, *rule ballI*,
simp add: *Airplane-not-in-danger-init-def ex-graph-def airplane-actors-def ex-locs-def*,
blast)
next show ⋀(*y*::*infrastructure*) *z*::*infrastructure*.
(*Airplane-not-in-danger-init*, *y*) ∈ {(*x*::*infrastructure*, *y*::*infrastructure*). $x \to_n y\}^* \Longrightarrow$
(*y*, *z*) ∈ {(*x*::*infrastructure*, *y*::*infrastructure*). $x \to_n y\} \Longrightarrow$
(*Airplane-not-in-danger-init*, *y*) ∈ {(*x*,*y*). $x \to_n y\}^* \longrightarrow$
(∀ *h*::*char list*∈*set* (*agra* (*graphI y*) *cockpit*). *h* ∈ *airplane-actors*) ⟹
(*Airplane-not-in-danger-init*, *z*) ∈ {(*x*,*y*). $x \to_n y\}^* \longrightarrow$
(∀ *h*::*char list*∈*set* (*agra* (*graphI z*) *cockpit*). *h* ∈ *airplane-actors*)
by (*rule impI*, *rule ballI*, *rule-tac z* = *y* **in** *airplane-actors-inv0*,
rule conjI, *erule impE*, *assumption*+, *simp*)
qed
show *?thesis*
by (*insert ind*, *insert assms*, *simp*)
qed

lemma *Eve-not-in-cockpit*: (*Airplane-not-in-danger-init*, *I*)
∈ {(*x*::*infrastructure*, *y*::*infrastructure*). $x \to_n y\}^* \Longrightarrow$
x ∈ *set* (*agra* (*graphI I*) *cockpit*) ⟹ *x* ≠ ''*Eve*''

```
 by (drule airplane-actors-inv, simp add: airplane-actors-def,
     drule-tac x = x in bspec, assumption, force)
```

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

```
lemma tp-imp-control:
 assumes (Airplane-not-in-danger-init,I) ∈ {(x::infrastructure, y::infrastructure).
x →n y}*
  shows (? x :: identity.  x @graphI I cockpit ∧ Actor x ≠ Actor ''Eve'')
proof −
  have a0: (2::nat) ≤ card (set (agra (graphI I) cockpit))
    by (insert assms, erule two-person-set-inv)
  have a1: is-singleton({''Charly''})
    by (rule is-singletonI)
  have a6: ¬(∀ x ∈ set(agra (graphI I) cockpit). (Actor x = Actor ''Eve''))
    proof (rule notI)
       assume a7: ∀x::char list∈set (agra (graphI I) cockpit). Actor x = Actor
''Eve''
      have a5: ∀x::char list∈set (agra (graphI I) cockpit). x = ''Charly''
        by (insert assms a0 a7, rule ballI, drule-tac x = x in bspec, assumption,
           subgoal-tac x ≠ ''Eve'', insert Insider-Eve, unfold Insider-def, (drule mp),

           rule Eve-precipitating-event, simp add: UasI-def, erule Eve-not-in-cockpit)
      have a4: set (agra (graphI I) cockpit) = {''Charly''}
        by (rule equalityI, rule subsetI, insert a5, simp,
            rule subsetI, simp, rule Set-all-unique, insert a0, force, rule a5)
      have a2: (card((set (agra (graphI I) cockpit)) :: char list set)) = (1 :: nat)
       by (insert a1, unfold is-singleton-altdef, erule ssubst, insert a4, erule ssubst,
           fold is-singleton-altdef, rule a1)
      have a3: (2 :: nat) ≤ (1 ::nat)
         by (insert a0, insert a2, erule subst, assumption)
      show False
        by (insert a5 a4 a3 a2, arith)
    qed
  show ?thesis  by (insert assms a0 a6, simp add: atI-def, blast)
qed

lemma Fend-2:   (Airplane-not-in-danger-init,I) ∈ {(x::infrastructure, y::infrastructure).
x →n y}* ⟹
        ¬ enables I cockpit (Actor ''Eve'') put
  by (insert cockpit-foe-control, simp add: foe-control-def, drule-tac x = I in spec,
      erule mp, erule tp-imp-control)

theorem Four-eyes-no-danger: Air-tp-Kripke ⊢ AG ({x. global-policy x ''Eve''})
proof (simp add: Air-tp-Kripke-def check-def, rule conjI)
  show Airplane-not-in-danger-init ∈ Air-tp-states
    by (simp add: Airplane-not-in-danger-init-def Air-tp-states-def
                  state-transition-in-refl-def)
next show Airplane-not-in-danger-init ∈ AG {x::infrastructure. global-policy x
''Eve''}
  proof (unfold AG-def, simp add: gfp-def,
    rule-tac x = {(x :: infrastructure) ∈ states Air-tp-Kripke. ~(''Eve'' @graphI x
cockpit)} in exI,
    rule conjI)
     show {x::infrastructure ∈ states Air-tp-Kripke. ¬ ''Eve'' @graphI x cockpit}
     ⊆ {x::infrastructure. global-policy x ''Eve''}
      by (unfold global-policy-def, simp add: airplane-actors-def, rule subsetI,
          drule CollectD, rule CollectI, erule conjE,
          simp add: Air-tp-Kripke-def Air-tp-states-def state-transition-in-refl-def,
          erule Fend-2)
 next show {x::infrastructure ∈ states Air-tp-Kripke. ¬ ''Eve'' @graphI x cockpit}
     ⊆ AX {x::infrastructure ∈ states Air-tp-Kripke. ¬ ''Eve'' @graphI x cockpit} ∧
     Airplane-not-in-danger-init
     ∈ {x::infrastructure ∈ states Air-tp-Kripke. ¬ ''Eve'' @graphI x cockpit}
   proof
     show Airplane-not-in-danger-init
          ∈ {x::infrastructure ∈ states Air-tp-Kripke. ¬ ''Eve'' @graphI x cockpit}
    by (simp add: Airplane-not-in-danger-init-def Air-tp-Kripke-def Air-tp-states-def
                  state-transition-refl-def ex-graph-def atI-def Air-tp-Kripke-def
                  state-transition-in-refl-def)
  next show {x::infrastructure ∈ states Air-tp-Kripke. ¬ ''Eve'' @graphI x cockpit}
     ⊆ AX {x::infrastructure ∈ states Air-tp-Kripke. ¬ ''Eve'' @graphI x cockpit}
     proof (rule subsetI, simp add: AX-def, rule subsetI, rule CollectI, rule conjI)
       show ⋀(x::infrastructure) xa::infrastructure.
        x ∈ states Air-tp-Kripke ∧ ¬ ''Eve'' @graphI x cockpit ⟹
        xa ∈ Collect (state-transition x) ⟹ xa ∈ states Air-tp-Kripke
        by (simp add:  Air-tp-Kripke-def Air-tp-states-def state-transition-in-refl-def,
            simp add: atI-def, erule conjE,
            unfold state-transition-infra-def state-transition-in-refl-def,
            erule rtrancl-into-rtrancl, rule CollectI, simp)
     next fix x xa
         assume a0: x ∈ states Air-tp-Kripke ∧ ¬ ''Eve'' @graphI x cockpit
          and a1: xa ∈ Collect (state-transition x)
         show ¬ ''Eve'' @graphI xa cockpit
       proof −
         have b: (Airplane-not-in-danger-init, xa)
         ∈ {(x::infrastructure, y::infrastructure). x →n y}*
          proof (insert a0 a1, rule rtrancl-trans)
            show x ∈ states Air-tp-Kripke ∧ ¬ ''Eve'' @graphI x cockpit ⟹
                  xa ∈ Collect (state-transition x) ⟹
                  (x, xa) ∈ {(x::infrastructure, y::infrastructure). x →n y}*
              by (unfold state-transition-infra-def, force)
          next show x ∈ states Air-tp-Kripke ∧ ¬ ''Eve'' @graphI x cockpit ⟹
                  xa ∈ Collect (state-transition x) ⟹
               (Airplane-not-in-danger-init, x) ∈ {(x::infrastructure, y::infrastructure).
x →n y}*
           by (erule conjE, simp add: Air-tp-Kripke-def Air-tp-states-def state-transition-in-refl-def)+
        qed
        show ?thesis
         by (insert a0 a1 b, rule-tac P = ''Eve'' @graphI xa cockpit in notI,
            simp add: atI-def, drule Eve-not-in-cockpit, assumption, simp)
      qed
    qed
  qed
qed
qed

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-actors ≡ {''Bob'', ''Charly'', ''Alice''}
definition airplane-locations-def':
airplane-locations ≡ {Location 0, Location 1, Location 2}
definition cockpit-def': cockpit ≡ Location 2
definition door-def': door ≡ Location 1
definition cabin-def': cabin ≡ Location 0
definition global-policy-def': global-policy I a ≡ a ∉ airplane-actors
              ⟶ ¬(enables I cockpit (Actor a) put)
definition 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 ([],[])))))

definition ex-locs-def': ex-locs ≡  (λ x. if x = door then [''norm''] else
                                     (if x = cockpit then [''air''] else []))

definition ex-locs'-def': ex-locs' ≡  (λ x. if x = door then [''locked''] else
                                       (if x = cockpit then [''air''] else []))

definition 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

definition 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'

definition 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

definition 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

definition local-policies-def': local-policies G ≡
   (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 {})))
definition local-policies-four-eyes-def': local-policies-four-eyes G ≡
   (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 {})))

definition Airplane-scenario-def':
Airplane-scenario ≡ Infrastructure ex-graph local-policies

definition Airplane-in-danger-def':
Airplane-in-danger ≡ 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-scenario* $\rightarrow_n *$ *I* $\}$

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

definition *Air-tp-Kripke-def'*:
Air-tp-Kripke $\equiv$ *Kripke Air-tp-states* $\{$*Airplane-not-in-danger-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-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*)
  by (*simp add*: *Actor-Abs-def*)

lemma *Actor-Abs-ran*: *Actor-Abs x* ∈ {*y* :: *identity option*. *y* ∈ *Some* ‘ {*x* :: *identity*. *x* ∉ {''*Eve*'', ''*Charly*''}}| *y* = *None*}
  by (*simp add*: *Actor-Abs-def*)

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* ∈ *Some* ‘ {*x* :: *identity*. *x* ∉ {''*Eve*'', ''*Charly*''}}| *y* = *None*}

lemma *Abs-inj-on*: ⋀ *Abs Rep*:: *actor* ⇒ *char list option*. *x* ∈ {*y* :: *identity option*. *y* ∈ *Some* ‘ {*x* :: *identity*. *x* ∉ {''*Eve*'', ''*Charly*''}}| *y* = *None*}
  ⟹ *y* ∈ {*y* :: *identity option*. *y* ∈ *Some* ‘ {*x* :: *identity*. *x* ∉ {''*Eve*'', ''*Charly*''}}| *y* = *None*}
  ⟹ (*Abs* :: *char list option* ⇒ *actor*) *x* = *Abs y* ⟹ *x* = *y*
by (*insert Actor-type-def*, *drule-tac x* = *Rep* **in** *meta-spec*, *drule-tac x* = *Abs* **in** *meta-spec*,
  frule-tac x = *Abs x* **and** *y* = *Abs y* **in** *type-definition.Rep-inject*,
  subgoal-tac (*Rep* (*Abs x*) = *Rep* (*Abs y*)), *subgoal-tac Rep* (*Abs x*) = *x*,
  subgoal-tac Rep (*Abs y*) = *y*, *erule subst*, *erule subst*, *assumption*,

(*erule type-definition.Abs-inverse*, *assumption*)+, *simp*)

lemma *Actor-td-Abs-inverse*:
(y∈ {y :: *identity option*. y ∈ *Some* ‘ {x :: *identity*. $x \notin$ {''*Eve*'', ''*Charly*''}}| y = *None*}) ⟹
(*Rep* :: *actor* ⇒ *identity option*)((*Abs* :: *identity option* ⇒ *actor*) y) = y
by (*insert Actor-type-def*, *drule-tac* x = *Rep* **in** *meta-spec*, *drule-tac* x = *Abs* **in** *meta-spec*,
erule type-definition.Abs-inverse, *assumption*)

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

axiomatization where *Actor-redef*: *Actor* = (*Abs* :: *identity option* ⇒ *actor*)*o* *Actor-Abs*

need to show that *Abs* (*Actor-Abs* x) = *Abs* (*Actor-Abs* y) ⟶ *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 ∈ *Some* ‘ {x. $x \notin$ {''*Eve*'', ''*Charly*''}} ∨ y = *None*}

lemma *UasI-Actor-redef*:
⋀ *Abs Rep*:: *actor* ⇒ *char list option*.
((*Abs* :: *identity option* ⇒ *actor*)*o* *Actor-Abs*) ''*Eve*'' = ((*Abs* :: *identity option* ⇒ *actor*)*o* *Actor-Abs*) ''*Charly*'' ∧
  (∀ (x::*char list*) y::*char list*. x ≠ ''*Eve*'' ∧ y ≠ ''*Eve*'' ∧
 ((*Abs* :: *identity option* ⇒ *actor*)*o* *Actor-Abs*) x = ((*Abs* :: *identity option* ⇒ *actor*)*o* *Actor-Abs*) y
  ⟶ x = y)
by (*insert UasI-ActorAbs*, *simp*, *clarify*, *drule-tac* x = x **in** *spec*, *drule-tac* x = y **in** *spec*,
  subgoal-tac Actor-Abs x = *Actor-Abs* y, *simp*, *rule Abs-inj-on*, *rule Actor-Abs-ran*, *rule Actor-Abs-ran*)

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*''
by (*unfold UasI-def*, *insert Actor-redef*, *drule meta-spec*, *erule ssubst*, *rule UasI-Actor-redef*)

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
by (*rule airplane.intro*, *simp add*: *tipping-point-def*,
  simp add: *Insider-def UasI-def tipping-point-def atI-def*,
  insert UasI-Actor, *simp add*: *UasI-def*,

```
      insert cockpit-foe-control', simp add: foe-control-def' cockpit-def',
      rule airplane-actors-def',
    (simp add: airplane-locations-def' cockpit-def' door-def' cabin-def' global-policy-def'
                ex-creds-def' ex-locs-def' ex-locs'-def' ex-graph-def' aid-graph-def'
aid-graph0-def'
           agid-graph-def' local-policies-def' local-policies-four-eyes-def' Airplane-scenario-def'
           Airplane-in-danger-def' Airplane-getting-in-danger0-def' Airplane-getting-in-danger-def'
           Air-states-def'  Air-Kripke-def' Airplane-not-in-danger-def' Airplane-not-in-danger-init-def'
               Air-tp-states-def' Air-tp-Kripke-def' Safety-def' Security-def'
               foe-control-def' astate-def')+)
```

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.