Applying the Isabelle Insider Framework to Airplane Security

Florian Kammüller and Manfred Kerber

April 22, 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 [2]. The Isabelle Insider framework has been first presented in [4]. 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 [8]. The case study and the application of the methododology are described in more detail in the preprint [3].

Contents

1 Kripke structures and CTL **2**
1.1 Lemmas to support least and greatest fixpoints 2
1.2 Generic type of state with state transition and CTL operators 8
1.3 Kripke structures and Modelchecking 9
1.4 Lemmas for CTL operators 9
1.4.1 EF lemmas 9
1.4.2 AG lemmas 13

2 Insider Framework **15**
2.1 Actors and actions 15
2.2 Infrastructure graphs and policies 17
2.3 Insider predicate 18
2.4 State transition on infrastructures 20

3 Airplane case study **26**
3.1 Formalization of Airplane Infrastructure and Properties . . . 26
3.2 Insider Attack, Safety, and Security 31

4 Analysis of Safety and Security Properties **35**
4.1 Introduce Two-Person Rule 38
4.2 Revealing Necessary Assumption by Proof Failure 51
4.3 Proving Security in Refined Model 53
4.4 Locale interpretation 64

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

```
lemma predtrans-empty:
  assumes mono (τ :: 'a set ⇒ 'a set)
  shows ∀ i. (τ ^^ i) ({}) ⊆ (τ ^^(i + 1))({})
proof (rule allI, induct-tac i)
  show ⋀i::nat. (τ ^^ (0::nat)) {} ⊆ (τ ^^ ((0::nat) + (1::nat))) {}
   by simp
next show ⋀(i::nat) n::nat. (τ ^^ n) {} ⊆ (τ ^^ (n + (1::nat))) {}
      ⟹ (τ ^^ Suc n) {} ⊆ (τ ^^ (Suc n + (1::nat))) {}
  proof −
    fix i n
    assume a : (τ ^^ n) {} ⊆ (τ ^^ (n + (1::nat))) {}
    have (τ ((τ ^^ n) {})) ⊆ (τ ((τ ^^ (n + (1 :: nat))) {})) using assms
      using a monoE by blast
    thus (τ ^^ Suc n) {} ⊆ (τ ^^ (Suc n + (1::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 :: 'a set ⇒ 'a set)^^ 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
    by (metis bot.not-eq-extremum card-gt-0-iff finite-subset subset-UNIV)
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)) {})
        using a by blast
      thus ∃ i. Suc n < card ((τ ^^ i) {}) using assms
      by (meson finite-subset le-less-trans le-simps(3) psubset-card-mono subset-UNIV)
    qed
  qed

lemma no-infinite-subset-chain:
   assumes finite (UNIV :: 'a set)
    and     mono (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 $\exists\, i.\ card\ UNIV < card\ (\tau^i\ \{\})$. Since all sets are subsets of *UNIV*, we also have $card\ (\tau^i\ \{\}) \le card\ UNIV$: Contradiction!, i.e. proof of False

```
proof −
  have a: ∀ (j :: nat). (∃ (i :: nat). (j :: nat) < card((τ ^^ i)({} :: 'a set))) using
assms
    by (erule-tac τ = τ in infchain-outruns-all)
  hence b: ∃ (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: ∃ i::nat. card UNIV < card ((τ ^^ i) {}) using a
    by (drule-tac x = card UNIV in spec)
  from this obtain i where e: card (UNIV :: 'a set) < card ((τ ^^ i) {})
    by (erule exE)
  hence f: (card((τ ^^ i){})) ≤ (card (UNIV :: 'a set)) using assms
    apply (erule-tac A = ((τ ^^ i){}) in Finite-Set.card-mono)
    by (rule subset-UNIV)
  thus False using e
    by (erule-tac y = card((τ ^^ i){}) in less-not-le)
qed
```

```
lemma finite-fixp:
  assumes finite(UNIV :: 'a set)
      and mono (τ :: ('a set ⇒ 'a set))
    shows ∃ i. (τ ^^ i) ({}) = (τ ^^(i + 1))({})
```

Proof idea: with *predtrans-empty* we know
$\forall\, i.\ \tau^{i}\ \{\} \subseteq \tau^{i\,+\,1}\ \{\}$ (1).
If we can additionally show
$\exists\, i.\ \tau^{i\,+\,1}\ \{\} \subseteq \tau^{i}\ \{\}$ (2),
we can get the goal together with equalityI $\subseteq + \supseteq \longrightarrow =$. To prove (1) we observe that $\tau^{i\,+\,1}\ \{\} \subseteq \tau^{i}\ \{\}$ can be inferred from $\neg\ \tau^{i}\ \{\} \subseteq \tau^{i\,+\,1}\ \{\}$ and (1). Finally, the latter is solved directly by *no-infinite-subset-chain*.

```
proof −
  have a: ∀ i::nat. (τ ^^ i) ({}:: 'a set) ⊆ (τ ^^ (i + (1::nat))) {}
    by(rule predtrans-empty, rule assms(2))
  have a3: ¬ (∀ i :: nat. (τ ^^ i) {} ⊂ (τ ^^(i + 1)) {})
    by (rule notI, rule no-infinite-subset-chain, (rule assms)+)
  hence b: (∃ i :: nat. ¬((τ ^^ i) {} ⊂ (τ ^^(i + 1)) {})) using assms a3
    by blast
  thus ∃ i. (τ ^^ i) ({}) = (τ ^^(i + 1))({}) using a
    by blast
qed
```

```
lemma predtrans-UNIV:
  assumes mono (τ :: ('a set ⇒ 'a set))
  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
a
      by (rule monoE)
    thus (τ ^^ (Suc n + (1::nat))) UNIV ⊆ (τ ^^ Suc n) UNIV by simp
```

qed
qed

lemma *Suc-less-le*: $x < (y - n) \Longrightarrow x \leq (y - (\mathit{Suc}\ n))$
by *simp*

lemma *card-univ-subtract*:
assumes $\mathit{finite}\ (\mathit{UNIV} :: {}'a\ \mathit{set})$ **and** $\mathit{mono}\ (\tau :: ({}'a\ \mathit{set} \Rightarrow {}'a\ \mathit{set}))$
and $(\forall i :: \mathit{nat}.\ ((\tau :: {}'a\ \mathit{set} \Rightarrow {}'a\ \mathit{set})\ \hat{}\hat{}\ (i + (1 :: \mathit{nat})))(\mathit{UNIV} :: {}'a\ \mathit{set}) \subset (\tau\ \hat{}\hat{}\ i)\ \mathit{UNIV})$
shows $(\forall\ i :: \mathit{nat}.\ \mathit{card}((\tau\ \hat{}\hat{}\ i)\ (\mathit{UNIV} :: {}'a\ \mathit{set})) \leq (\mathit{card}\ (\mathit{UNIV} :: {}'a\ \mathit{set})) - i)$
proof (*rule allI*, *induct-tac i*)
show $\mathit{card}\ ((\tau\ \hat{}\hat{}\ (0{::}\mathit{nat}))\ \mathit{UNIV}) \leq \mathit{card}\ (\mathit{UNIV} :: {}'a\ \mathit{set}) - (0{::}\mathit{nat})$ **using** *assms*
by (*simp*)
next show $\bigwedge(i{::}\mathit{nat})\ n{::}\mathit{nat}.$
$\mathit{card}\ ((\tau\ \hat{}\hat{}\ n)\ (\mathit{UNIV} :: {}'a\ \mathit{set})) \leq \mathit{card}\ (\mathit{UNIV} :: {}'a\ \mathit{set}) - n \Longrightarrow$
$\mathit{card}\ ((\tau\ \hat{}\hat{}\ \mathit{Suc}\ n)\ (\mathit{UNIV} :: {}'a\ \mathit{set})) \leq \mathit{card}\ (\mathit{UNIV} :: {}'a\ \mathit{set}) - \mathit{Suc}\ n$ **using** *assms*
proof −
fix *i n*
assume *a*: $\mathit{card}\ ((\tau\ \hat{}\hat{}\ n)\ (\mathit{UNIV} :: {}'a\ \mathit{set})) \leq \mathit{card}\ (\mathit{UNIV} :: {}'a\ \mathit{set}) - n$
have *b*: $(\tau\ \hat{}\hat{}\ (n + (1{::}\mathit{nat})))(\mathit{UNIV} :: {}'a\ \mathit{set}) \subset (\tau\ \hat{}\hat{}\ n)\ \mathit{UNIV}$ **using** *assms*
by (*erule-tac* $x = n$ **in** *spec*)
have $\mathit{card}((\tau\ \hat{}\hat{}\ (n + (1 :: \mathit{nat})))(\mathit{UNIV} :: {}'a\ \mathit{set})) < \mathit{card}((\tau\ \hat{}\hat{}\ n)\ (\mathit{UNIV} :: {}'a\ \mathit{set}))$
by (*rule psubset-card-mono*, *rule finite-subset*, *rule subset-UNIV*, *rule assms*(*1*), *rule b*)
thus $\mathit{card}\ ((\tau\ \hat{}\hat{}\ \mathit{Suc}\ n)\ (\mathit{UNIV} :: {}'a\ \mathit{set})) \leq \mathit{card}\ (\mathit{UNIV} :: {}'a\ \mathit{set}) - \mathit{Suc}\ n$ **using** *a*
by *simp*
qed
qed

lemma *card-UNIV-tau-i-below-zero*:
assumes $\mathit{finite}\ (\mathit{UNIV} :: {}'a\ \mathit{set})$ **and** $\mathit{mono}\ (\tau :: ({}'a\ \mathit{set} \Rightarrow {}'a\ \mathit{set}))$
and $(\forall i :: \mathit{nat}.\ ((\tau :: ({}'a\ \mathit{set} \Rightarrow {}'a\ \mathit{set}))\ \hat{}\hat{}\ (i + (1 :: \mathit{nat})))(\mathit{UNIV} :: {}'a\ \mathit{set}) \subset (\tau\ \hat{}\hat{}\ i)\ \mathit{UNIV})$
shows $\mathit{card}((\tau\ \hat{}\hat{}\ (\mathit{card}\ (\mathit{UNIV} :: {}'a\ \mathit{set})))\ (\mathit{UNIV} :: {}'a\ \mathit{set})) \leq 0$
proof −
have $(\forall\ i :: \mathit{nat}.\ \mathit{card}((\tau\ \hat{}\hat{}\ i)\ (\mathit{UNIV} :: {}'a\ \mathit{set})) \leq (\mathit{card}\ (\mathit{UNIV} :: {}'a\ \mathit{set})) - i)$ **using** *assms*
by (*rule card-univ-subtract*)
thus $\mathit{card}((\tau\ \hat{}\hat{}\ (\mathit{card}\ (\mathit{UNIV} :: {}'a\ \mathit{set})))\ (\mathit{UNIV} :: {}'a\ \mathit{set})) \leq 0$
by (*drule-tac* $x = \mathit{card}\ (\mathit{UNIV} :: {}'a\ \mathit{set})$ **in** *spec*, *simp*)
qed

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

```
by simp

lemma UNIV-tau-i-is-empty:
  assumes finite (UNIV :: 'a set) and mono (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) = {}
  by (meson assms card-UNIV-tau-i-below-zero finite-card-zero-empty finite-subset
subset-UNIV)

lemma down-chain-reaches-empty:
  assumes finite (UNIV :: 'a set) and mono (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 = {}
  using UNIV-tau-i-is-empty assms by blast

lemma no-infinite-subset-chain2:
  assumes finite (UNIV :: 'a set) and mono (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
    by (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 mono (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
    by (rule predtrans-UNIV , 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)
      using assms(1) assms(2) no-infinite-subset-chain2 by blast
    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 lfp-loop:
  assumes finite (UNIV :: 'b set) and mono (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)
   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 mono (f :: ('a set ⇒ 'a set))
      and (f ^^ Suc(n)) UNIV = (f ^^ n) UNIV
    shows gfp f  = (f ^^ n) UNIV
  by (rule  Nat.gfp-Kleene-iter, rule assms, rule assms)

lemma gfp-loop:
```

assumes *finite* (*UNIV* :: *'b set*)
and *mono* (τ :: (*'b set* $\Rightarrow$ *'b set*))
shows $\exists$ *n* . *gfp* τ = (τ ^^ *n*)(*UNIV* :: *'b set*)

proof −

have $\exists$ *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 $\exists$ *n* . *gfp* τ = (τ ^^ *n*)(*UNIV* :: *'b set*) **using** *assms*
by (*metis Suc-eq-plus1 gfp-Kleene-iter-set*)

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

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

The above class definition lifts Kripke structures and CTL to a general level. The definition of the inductive relation is given by a set of specific rules which are, however, part of an application like infrastructures. Branching time temporal logic CTL is defined in general over Kripke structures with arbitrary state transitions and can later be applied to suitable theories, like infrastructures. Based on the generic state transition → 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* $\equiv$ $\{s.\ \{f0.\ s \rightarrow_i f0\} \subseteq f\}$
definition *EX'* **where** *EX' f* $\equiv$ $\{s\ .\ \exists\ f0 \in f.\ s \rightarrow_i f0\ \}$

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

definition *AF* **where** *AF f* $\equiv$ *lfp* (λ *Z*. *f* $\cup$ *AX Z*)
definition *EF* **where** *EF f* $\equiv$ *lfp* (λ *Z*. *f* $\cup$ *EX' Z*)
definition *AG* **where** *AG f* $\equiv$ *gfp* (λ *Z*. *f* $\cap$ *AX Z*)
definition *EG* **where** *EG f* $\equiv$ *gfp* (λ *Z*. *f* $\cap$ *EX' Z*)
definition *AU* **where** *AU f1 f2* $\equiv$ *lfp*(λ *Z*. *f2* $\cup$ (*f1* $\cap$ *AX Z*))
definition *EU* **where** *EU f1 f2* $\equiv$ *lfp*(λ *Z*. *f2* $\cup$ (*f1* $\cap$ *EX' Z*))
definition *AR* **where** *AR f1 f2* $\equiv$ *gfp*(λ *Z*. *f2* $\cap$ (*f1* $\cup$ *AX Z*))
definition *ER* **where** *ER f1 f2* $\equiv$ *gfp*(λ *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 (- ⊢ - 50)
 where M ⊢ f ≡ (init M) ⊆ {s ∈ (states M). s ∈ f }

definition state-transition-refl (infixr →ᵢ* 50)
where s →ᵢ* s' ≡ ((s,s') ∈ {(x,y). state-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)))
    by (rule def-lfp-unfold, rule reflexive, unfold mono-def EX'-def, 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))
   by (rule subsetI, simp add: EF-lem0)
  next show f ∪ EX' (lfp (λZ::'a set. f ∪ EX' Z)) ⊆ EF f
   by (rule subsetI, 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)
    by (fold EF-def, 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)))
      by (rule def-lfp-unfold, rule reflexive, unfold mono-def EX'-def, auto)
    thus x ∈ lfp (λZ::'a set. f ∪ EX' Z) using a
      by (subst b, blast)
  qed
qed

lemma EF-lem2b:
    assumes x ∈ (EX' (EF f))
   shows x ∈ EF f
proof (rule EF-lem1)
  show x ∈ f ∨ x ∈ EX' (EF f)
    by (rule disjI2, rule assms)
qed

lemma EF-lem2a: assumes x ∈ f shows x ∈ EF f
proof (rule EF-lem1)
  show x ∈ f ∨ x ∈ EX' (EF f)
    by (rule disjI1, rule assms)
qed

lemma EF-lem2c: assumes x ∉ f shows x ∈ EF (− f)
proof −
  have x ∈ (− f) using assms
    by simp
  thus x ∈ EF (− f)
    by (rule EF-lem2a)
qed

lemma EF-lem2d: assumes x ∉ EF f shows x ∉ f
proof −
  have x ∈ f ⟹ x ∈ EF f
    by (erule EF-lem2a)
  thus x ∉ f using assms
    apply (erule-tac P = x ∈ f in contrapos-nn)
    by (erule meta-mp)
qed

lemma EF-lem3b: assumes x ∈ EX' (f ∪ EX' (EF f)) shows x ∈ (EF f)
proof (simp add: EF-lem0)
  show x ∈ f ∨ x ∈ EX' (lfp (λZ::'a set. f ∪ EX' Z))
    by (metis EF-def EF-lem00 assms)
qed

lemma EX-lem0l: x ∈ (EX' f) ⟹ x ∈ (EX' (f ∪ g))
proof (unfold EX'-def)
  show x ∈ {s::'a. ∃f0::'a∈f. s →ᵢ f0} ⟹ x ∈ {s::'a. ∃f0::'a∈f ∪ g. s →ᵢ f0}
```

```
    by blast
qed

lemma EX-lem0r: x ∈ (EX' g) ⟹ x ∈ (EX' (f ∪ g))
proof (unfold EX'-def)
  show x ∈ {s::'a. ∃ f0::'a∈g. s →ᵢ f0} ⟹ x ∈ {s::'a. ∃ f0::'a∈f ∪ g. s →ᵢ f0}
    by blast
qed

lemma EX-step: assumes x  →ᵢ y and y ∈ f shows x ∈ EX' f
proof (unfold EX'-def)
  show  x ∈ {s::'a. ∃ f0::'a∈f. s →ᵢ f0}
    using assms(1) assms(2) by blast
qed

lemma EF-E[rule-format]: ∀ f. x ∈ (EF (f :: ('a :: state) set)) ⟶ x ∈ (f ∪ EX'
(EF f))
proof −
  have a: ⋀f::'a set. EF (f :: ('a :: state) set) = f ∪ EX' (EF f)
    by (rule EF-lem000)
  thus (∀ f. x ∈ EF (f :: ('a :: state) set) ⟶ x ∈ f ∪ EX' (EF f))
    by auto
qed

lemma EF-step: assumes x  →ᵢ y and y ∈ f shows x ∈ EF f
proof (rule EF-lem3b)
  show x ∈ EX' (f ∪ EX' (EF f))
    using EX-step assms(1) assms(2) by blast
qed

lemma EF-step-step: assumes x  →ᵢ y and y ∈ EF f shows  x ∈ EF f
proof −
  have y ∈ f ∪ EX' (EF f)
    by (rule EF-E, rule assms(2))
  thus x ∈ EF f
    using EF-lem3b EX-step assms(1) by blast
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 z::'a. y ∈ f ⟹
            (ya, z) ∈ {(x,y). x →ᵢ y} ⟹
            (z, y) ∈ {(x,y). x →ᵢ y}* ⟹ z ∈ EF f ⟹ ya ∈ EF f
        by (simp add: EF-step-step)
    qed
```

qed

lemma *EF-induct-prep*:
assumes $(a::'a::\mathit{state}) \in \mathit{lfp}\ (\lambda\ Z.\ (f::'a::\mathit{state\ set}) \cup EX'\ Z)$
and *mono* $(\lambda\ Z.\ (f::'a::\mathit{state\ set}) \cup EX'\ Z)$
shows $(\bigwedge x::'a::\mathit{state}.$
$x \in ((\lambda\ Z.\ (f::'a::\mathit{state\ set}) \cup EX'\ Z)(\mathit{lfp}\ (\lambda\ Z.\ (f::'a::\mathit{state\ set}) \cup EX'\ Z) \cap \{x::'a::\mathit{state}.\ (P::'a::\mathit{state} \Rightarrow \mathit{bool})\ x\})) \Longrightarrow P\ x) \Longrightarrow$
$P\ a$
proof −
show $(\bigwedge x::'a::\mathit{state}.$
$x \in ((\lambda\ Z.\ (f::'a::\mathit{state\ set}) \cup EX'\ Z)(\mathit{lfp}\ (\lambda\ Z.\ (f::'a::\mathit{state\ set}) \cup EX'\ Z) \cap \{x::'a::\mathit{state}.\ (P::'a::\mathit{state} \Rightarrow \mathit{bool})\ x\})) \Longrightarrow P\ x) \Longrightarrow$
$P\ a$
by (*rule-tac A* = *EF f* **in** *def-lfp-induct-set*, *rule EF-def*, *rule assms*(*2*), (*simp add*: *EF-def assms*)+)
qed

lemma *EF-induct*: $(a::'a::\mathit{state}) \in EF\ (f\ ::\ 'a\ ::\ \mathit{state\ set}) \Longrightarrow$
mono $(\lambda\ Z.\ (f::'a::\mathit{state\ set}) \cup EX'\ Z) \Longrightarrow$
$(\bigwedge x::'a::\mathit{state}.$
$x \in ((\lambda\ Z.\ (f::'a::\mathit{state\ set}) \cup EX'\ Z)(EF\ f \cap \{x::'a::\mathit{state}.\ (P::'a::\mathit{state} \Rightarrow \mathit{bool})\ x\})) \Longrightarrow P\ x) \Longrightarrow$
$P\ a$
proof (*simp add*: *EF-def*)
show $a \in \mathit{lfp}\ (\lambda Z::'a\ \mathit{set}.\ f \cup EX'\ Z) \Longrightarrow$
mono $(\lambda Z::'a\ \mathit{set}.\ f \cup EX'\ Z) \Longrightarrow$
$(\bigwedge x::'a.\ x \in f \vee x \in EX'\ (\mathit{lfp}\ (\lambda Z::'a\ \mathit{set}.\ f \cup EX'\ Z) \cap \mathit{Collect}\ P) \Longrightarrow P\ x) \Longrightarrow P\ a$
by (*erule EF-induct-prep*, *assumption*, *simp*)
qed

lemma *valEF-E*: $M \vdash EF\ f \Longrightarrow x \in \mathit{init}\ M \Longrightarrow x \in EF\ f$
proof (*simp add*: *check-def*)
show $\mathit{init}\ M \subseteq \{s::'a \in \mathit{states}\ M.\ s \in EF\ f\} \Longrightarrow x \in \mathit{init}\ M \Longrightarrow x \in EF\ f$
by *blast*
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* $(\lambda Z::'a\ \mathit{set}.\ s \cup EX'\ Z)$
by (*simp add*: *mono-def EX'-def*, *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*)
using *state-transition-refl-def* **apply** *auto*[*1*]
by (*auto simp add*: *EX'-def state-transition-refl-def intro*: *converse-rtrancl-into-rtrancl*)
qed

```
lemma EF-step-inv: (I ⊆ {sa::'s :: state. (∃ i::'s∈I. i →i* sa) ∧ sa ∈ EF s})
        ⟹ ∀ x ∈ I. ∃ y ∈ s. x →i* y
proof (clarify)
  show ⋀x::'s. I ⊆ {sa::'s. (∃ i::'s∈I. i →i* sa) ∧ sa ∈ EF s} ⟹ x ∈ I ⟹
∃ y::'s∈s. x →i* y
    using EF-step-star-rev by fastforce
qed
```

1.4.2 AG lemmas

```
lemma AG-in-lem:   x ∈ AG s ⟹ x ∈ s
proof (simp add: AG-def gfp-def)
  show ∃ xa⊆s. xa ⊆ AX xa ∧ x ∈ xa ⟹ x ∈ s
    by blast
qed

lemma AG-lem1: x ∈ s ∧ x ∈ (AX (AG s)) ⟹ x ∈ AG s
proof (simp add: AG-def)
  show x ∈ s ∧ x ∈ AX (gfp (λZ::'a set. s ∩ AX Z)) ⟹ x ∈ gfp (λZ::'a set. s
∩ AX Z)
  apply (subgoal-tac gfp (λZ::'a set. s ∩ AX Z) =
                    s ∩ (AX (gfp (λZ::'a set. s ∩ 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 ∈ AG s ⟹ x ∈ (s ∩ (AX (AG s)))
proof −
  have a: AG s = s ∩ (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 ∈ AG s ⟹ x ∈ (s ∩ (AX (AG s)))
   by (erule subst)
qed

lemma AG-lem3: AG s = (s ∩ (AX (AG s)))
proof (rule equalityI)
  show AG s ⊆ s ∩ AX (AG s)
    using AG-lem2 by blast
  next show s ∩ AX (AG s) ⊆ AG s
      using AG-lem1 by blast
qed
```

```
lemma AG-step: y →_i z ⟹ y ∈ AG s ⟹ z ∈ AG s
  using AG-lem2 AX-def by blast

lemma AG-all-s:  x →_i* y ⟹ x ∈ AG s ⟹ y ∈ AG s
proof (simp add: state-transition-refl-def)
  show (x, y) ∈ {(x,y). x →_i y}^* ⟹ x ∈ AG s ⟹ y ∈ AG s
  proof (erule rtrancl-induct)
    show x ∈ AG s ⟹ x ∈ AG s by assumption
  next show ⋀(y::'a) z::'a.
       x ∈ AG s ⟹
       (x, y) ∈ {(x,y). x →_i y}^* ⟹
       (y, z) ∈ {(x,y). x →_i y} ⟹ y ∈ AG s ⟹ z ∈ AG s
      by (simp add: AG-step)
  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
                using AG-in-lem by blast
              moreover have x ∈ −s using a5
              proof −
                have x ∈ EF s
                  by (simp add: EF-lem2a calculation)
                thus $x \in -s$ using a5
                proof −
                  have $x \in EF\ (-\ s)$ using a5
                    by simp
                  moreover from this obtain y where a7: $y \in -\ s \wedge x \to_i* y$
                    using EF-step-star-rev by blast
                  moreover have $y \in AG\ s$ using a7 a5
                    using AG-all-s by blast
                  ultimately show $x \in -s$ using a5
                    using 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 $\{sa::'s.\ (\exists\, i::'s\in I.\ i \to_i* sa) \wedge sa \in AG\ s\} \cap$
                     $\{sa::'s.\ (\exists\, i::'s\in I.\ i \to_i* sa) \wedge sa \in EF\ (-\ s)\} = \{\}$
        by blast
      qed
    moreover have b: ? x. x : I using a0
      by blast
    moreover obtain x where $x \in I$
      using b by blast
    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

In the Isabelle/HOL theory for Insiders, one expresses policies over actions *get*, *move*, *eval*, and *put*.

2.1 Actors and actions

The theory Airinsider is an instance of the Insider framework for the case study of airplane insiders. Although the Isabelle Insider framework is a generic framework the actual semantics of the actions is specific to applications. Therefore we use here an "instance" of the framework in the form of a theory "Airinsider" but the main part of definitions and declarations is the same.

theory *AirInsider*
  imports *MC*
begin

An actor may be enabled to

- *get* data or physical items, like keys,
- *move* to a location,
- *eval* a program,
- *put* data at locations or physical items – like airplanes – "to the ground".

The precise semantics of these actions is refined in the state transition rules for the concrete infrastructure. The framework abstracts from concrete data – actions have no parameters:

datatype *action* = *get* | *move* | *eval* | *put*

The human component is the *Actor* which is represented by an abstract type *actor* and a function *Actor* that creates elements of that type from identities (of type *string*):

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

typedecl *actor*
type-synonym *identity* = *string*
consts *Actor* :: *identity* ⇒ *actor*

Note that it would seem more natural and simpler to just define *actor* as a datatype over identities with a constructor *Actor* instead of a simple constant together with a type declaration like, for example, in the Isabelle inductive approach to security protocol verification [6, 7]. This would, however, make the constructor *Actor* an injective function by the underlying foundation of datatypes therefore excluding the fine grained modelling that is at the core of the insider definition: In fact, it defines the function *Actor* to be injective for all except insiders and explicitly enables insiders to have different roles by identifying *Actor* images.

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 as an alternative to 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 is a suggested alternative [8].

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 of it we cannot instantiate *'a infrastructure* to state (clearly an abstract thing as an instance is strange).

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

2.2 Infrastructure graphs and policies

Actors are contained in an infrastructure graph. An *igraph* contains a set of location pairs representing the topology of the infrastructure as a graph of nodes and a list of actor identities associated to each node (location) in the graph. Also an *igraph* associates actors to a pair of string sets by a pair-valued function whose first range component is a set describing the credentials in the possession of an actor and the second component is a set defining the roles the actor can take on. Finally, an *igraph* assigns locations to a pair of a string that defines the state of the component. Corresponding projection functions for each of these components of an *igraph* are provided; they are named *gra* for the actual set of pairs of locations, *agra* for the actor map, *cgra* for the credentials, and *lgra* for the state of a location and the data at that location.

datatype *location* = *Location nat*
datatype *igraph* = *Lgraph* (*location* ∗ *location*)*set location* ⇒ *identity list*
actor ⇒ (*string list* ∗ *string list*) *location* ⇒ *string list*

Atomic policies of type *apolicy* describe prerequisites for actions to be granted to actors given by pairs of predicates (conditions) and sets of (enabled) actions:

type-synonym *apolicy* = ((*actor* ⇒ *bool*) ∗ *action set*)

datatype *infrastructure* =
Infrastructure igraph
[*igraph*, *location*] ⇒ *apolicy 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] ⇒ apolicy 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)
```

2.3 Insider predicate

The human actor's level is modelled in the Isabelle Insider framework by assigning the individual actor's psychological disposition[1] *actor-state* to each actor's identity.

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

The values used for the definition of the types *motivations* and *psy-state* are based on a taxonomy from psychological insider research [5]. The transition to become an insider is represented by a *Catalyst* that tips the insider over the edge so he acts as an insider formalized as a "tipping point" predicate.

```
datatype actor-state = Actor-state psy-states motivations set
primrec motivation :: actor-state ⇒ motivations set
```

[1] Note that the determination of the psychological state of an actor is not done using the formal system. It is up to a psychologist to determine this. However, if for instance, an actor is classified as *disgruntled* then this may have an influence on what they are allowed to do according to a company policy and this can be formally described and reasoned about in Isabelle.

where $\mathit{motivation}\ \ (\mathit{Actor\text{-}state}\ p\ m) =\ \ m$
primrec $\mathit{psy\text{-}state} :: \mathit{actor\text{-}state} \Rightarrow \mathit{psy\text{-}states}$
where $\mathit{psy\text{-}state}\ \ (\mathit{Actor\text{-}state}\ p\ m) = p$

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

To embed the fact that the attacker is an insider, the actor can then impersonate other actors. In the Isabelle Insider framework, the predicate *Insider* must be used as a *locale* assumption to enable impersonation for the insider: this assumption entails that an insider $\mathit{Actor}\ ''\mathit{Eve}''$ can act like their alter ego, say $\mathit{Actor}\ ''\mathit{Charly}''$ within the context of the locale. This is realized by the predicate *UasI*: *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 (an impersonation predicate ”a can act as b”). For all other identities, *Actor* is defined as injective on those identities. The first one is stronger and allows substitution of the insider in any context; the second one is parameterized over a context predicate to describe this.

definition $\mathit{UasI} ::\ [\mathit{identity}, \mathit{identity}] \Rightarrow \mathit{bool}$
where $\mathit{UasI}\ a\ b \equiv (\mathit{Actor}\ a = \mathit{Actor}\ b) \wedge (\forall\ x\ y.\ x \neq a \wedge y \neq a \wedge \mathit{Actor}\ x = \mathit{Actor}\ y \longrightarrow x = y)$
definition $\mathit{UasI}' ::\ [\mathit{actor} \Rightarrow \mathit{bool}, \mathit{identity}, \mathit{identity}] \Rightarrow \mathit{bool}$
where $\mathit{UasI}'\ P\ a\ b \equiv P\ (\mathit{Actor}\ b) \longrightarrow P\ (\mathit{Actor}\ a)$

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

The predicate *atI* – mixfix syntax $@_G$ – expresses that an actor (identity) is at a certain location in an igraph.

definition $\mathit{atI} :: [\mathit{identity}, \mathit{igraph}, \mathit{location}] \Rightarrow \mathit{bool}\ (\text{-}\ @_{(\text{-})}\ \text{-}\ \mathit{50})$
where $a\ @_G\ l \equiv a \in \mathit{set}(\mathit{agra}\ G\ l)$

The enables predicate 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. Policies specify the expected behaviour of actors of an infrastructure. They are defined by the *enables* predicate: an actor h is enabled to perform an action a in infrastructure I, at location l if there exists a pair (p,e) in the local policy of l ($\mathit{delta}\ I\ l$ projects to the local policy) such that the action

a is a member of the action set e and the policy predicate p holds for actor h.

definition $\mathit{enables} :: [\mathit{infrastructure}, \mathit{location}, \mathit{actor}, \mathit{action}] \Rightarrow \mathit{bool}$
where
$\mathit{enables}\ I\ l\ a\ a' \equiv\ (\exists\ (p,e) \in \mathit{delta}\ I\ (\mathit{graphI}\ I)\ l.\ a' \in e \wedge p\ a)$

For example, the *apolicy* pair ($\lambda\ x.\ \mathit{True}$, $\{\mathit{move}\}$) specifies that all actors are enabled to perform action *move*.

The behaviour is the good behaviour, i.e. everything allowed by the policy of Infrastructure I.

definition $\mathit{behaviour} :: \mathit{infrastructure} \Rightarrow (\mathit{location} * \mathit{actor} * \mathit{action})\mathit{set}$
where $\mathit{behaviour}\ I \equiv \{(t,a,a').\ \mathit{enables}\ I\ t\ a\ a'\}$

The misbehaviour is the complement of behaviour of an Infrastructure I.

definition $\mathit{misbehaviour} :: \mathit{infrastructure} \Rightarrow (\mathit{location} * \mathit{actor} * \mathit{action})\mathit{set}$
where $\mathit{misbehaviour}\ I \equiv -(\mathit{behaviour}\ I)$

We prove some basic lemmas for the predicate *enable*.

lemma *not-enableI*: $(\forall\ (p,e) \in \mathit{delta}\ I\ (\mathit{graphI}\ I)\ l.\ (\neg(h : e)\ |\ (\neg(p(a)))))$
$\Longrightarrow \neg(\mathit{enables}\ I\ l\ a\ h)$
by (*simp add*: *enables-def*, *blast*)

lemma *not-enableI2*: $[\![\bigwedge\ p\ e.\ (p,e) \in \mathit{delta}\ I\ (\mathit{graphI}\ I)\ l \Longrightarrow$
$(\neg(t : e)\ |\ (\neg(p(a))))\ ]\!] \Longrightarrow \neg(\mathit{enables}\ I\ l\ a\ t)$
by (*rule not-enableI*, *rule ballI*, *auto*)

lemma *not-enableE*: $[\![\ \neg(\mathit{enables}\ I\ l\ a\ t);\ (p,e) \in \mathit{delta}\ I\ (\mathit{graphI}\ I)\ l\ ]\!]$
$\Longrightarrow (\neg(t : e)\ |\ (\neg(p(a))))$
by (*simp add*: *enables-def*, *rule impI*, *force*)

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

2.4 State transition on infrastructures

The state transition defines how actors may act on infrastructures through actions within the boundaries of the policy. It is given as an inductive definition over the states which are infrastructures. This state transition relation is dependent on actions but also on enabledness and the current state of the infrastructure.

First we introduce some auxiliary functions dealing with repetitions in lists and actors moving in an *igraph* and some constructions to deal with lists of actors in locations for the semantics of action *move*.

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

```
where
del-nil: del a [] = [] |
del-cons: del a (x#ls) = (if x = a then ls else x # (del a ls))

primrec jonce :: ['a, 'a list] ⇒ bool
where
jonce-nil: jonce a [] = False |
jonce-cons: jonce a (x#ls) = (if x = a then (a ∉ (set ls)) else jonce a ls)

primrec nodup :: ['a, 'a list] ⇒ bool
  where
    nodup-nil: nodup a [] = True |
    nodup-step: nodup a (x # ls) = (if x = a then (a ∉ (set ls)) else nodup a ls)

definition move-graph-a :: [identity, location, location, igraph] ⇒ igraph
where move-graph-a n l l' g ≡ Lgraph (gra g)
                    (if n ∈ set ((agra g) l) &  n ∉ 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] ⇒ bool ((- →ₙ -)
50)
where
  move: ⟦ G = graphI I; a @G l; l ∈ nodes G; l' ∈ nodes G;
         (a) ∈ actors-graph(graphI I); enables I l' (Actor a) move;
        I' = Infrastructure (move-graph-a a l l' (graphI I))(delta I) ⟧ ⟹ I →ₙ I'
| get : ⟦ G = graphI I; a @G l; a' @G l; has G (Actor a, z);
        enables I l (Actor a) get;
        I' = Infrastructure
                   (Lgraph (gra G)(agra G)
                           ((cgra G)(Actor a' :=
                                (z # (fst(cgra G (Actor a'))), snd(cgra G (Actor a')))))
                           (lgra G))
                   (delta I)
         ⟧ ⟹ I →ₙ I'
| put : ⟦ G = graphI I; a @G l; enables I l (Actor a) put;
        I' = Infrastructure
                  (Lgraph (gra G)(agra G)(cgra G)
                          ((lgra G)(l := [z])))
                   (delta I) ⟧
         ⟹ I →ₙ I'
| put-remote : ⟦ G = graphI I; enables I l (Actor a) put;
        I' = Infrastructure
                  (Lgraph (gra G)(agra G)(cgra G)
                            ((lgra G)(l := [z])))
                    (delta I) ⟧
         ⟹ I →ₙ I'
```

Note that the type infrastructure can now be instantiated to the axiomatic type class *state* which enables the use of the underlying Kripke structures and CTL. We need to show that this infrastructure is a state as given in MC.thy

instantiation *infrastructure* :: *state*
begin
definition
  state-transition-infra-def: $(i \rightarrow_i i') = (i \rightarrow_n (i' :: \mathit{infrastructure}))$

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

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

Lemmas about the auxiliary functions *del*, *jonce*, *nodup* are provided.

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

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

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

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

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

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

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

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

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

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

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

lemma *delta-invariant*: $\forall$ *z z'*. *z* $\to_n$ *z'* $\longrightarrow$ *delta*(*z*) = *delta*(*z'*)
  by (*clarify*, *erule state-transition-in.cases*, *simp*+)

lemma *init-state-policy0*:
  assumes $\forall$ *z z'*. *z* $\to_n$ *z'* $\longrightarrow$ *delta*(*z*) = *delta*(*z'*)
    and (*x*,*y*) $\in$ {(*x*::*infrastructure*, *y*::*infrastructure*). *x* $\to_n$ *y*}*
   shows *delta*(*x*) = *delta*(*y*)
proof −
  have *ind*: (*x*,*y*) $\in$ {(*x*::*infrastructure*, *y*::*infrastructure*). *x* $\to_n$ *y*}*
      $\longrightarrow$ *delta*(*x*) = *delta*(*y*)
  proof (*insert assms*, *erule rtrancl.induct*)
   show ($\bigwedge$ *a*::*infrastructure*.
     ($\forall$(*z*::*infrastructure*)(*z'*::*infrastructure*). (*z* $\to_n$ *z'*) $\longrightarrow$ (*delta z* = *delta z'*))
$\Longrightarrow$
     (((*a*, *a*) $\in$ {(*x* ::*infrastructure*, *y* :: *infrastructure*). *x* $\to_n$ *y*}*) $\longrightarrow$
     (*delta a* = *delta a*)))
   by (*rule impI*, *rule refl*)
next fix *a b c*
  assume *a0*: $\forall$(*z*::*infrastructure*) *z'*::*infrastructure*. *z* $\to_n$ *z'* $\longrightarrow$ *delta z* = *delta z'*
    and *a1*: (*a*, *b*) $\in$ {(*x*::*infrastructure*, *y*::*infrastructure*). *x* $\to_n$ *y*}*
    and *a2*: (*a*, *b*) $\in$ {(*x*::*infrastructure*, *y*::*infrastructure*). *x* $\to_n$ *y*}* $\longrightarrow$
      delta a = *delta b*
    and *a3*: (*b*, *c*) $\in$ {(*x*::*infrastructure*, *y*::*infrastructure*). *x* $\to_n$ *y*}
    show (*a*, *c*) $\in$ {(*x*::*infrastructure*, *y*::*infrastructure*). *x* $\to_n$ *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*) $\in$ {(*x*::*infrastructure*, *y*::*infrastructure*). *x* $\to_n$ *y*}* ⟧ $\Longrightarrow$
        delta(*x*) = *delta*(*y*)
  by (*rule init-state-policy0*, *rule delta-invariant*)

```
lemma same-nodes0[rule-format]: ∀ z z'. z →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 →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 →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])))
(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)
(l::location)
      (a::char list) (I'::infrastructure) za::char list.
      z = I ⟹
      z' = I' ⟹
```

$G = \mathit{graphI}\ I \Longrightarrow$
$\mathit{enables}\ I\ l\ (\mathit{Actor}\ a)\ \mathit{put} \Longrightarrow$
$I' = \mathit{Infrastructure}\ (\mathit{Lgraph}\ (\mathit{gra}\ G)\ (\mathit{agra}\ G)\ (\mathit{cgra}\ G)\ ((\mathit{lgra}\ G)(l := [\mathit{za}])))$
$(\mathit{delta}\ I) \Longrightarrow$
$\mathit{actors\text{-}graph}\ (\mathit{graphI}\ z) = \mathit{actors\text{-}graph}\ (\mathit{graphI}\ z')$
by (*simp add*: *actors-graph-def nodes-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 (*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 $x = \mathit{ya}$ **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 $x = \mathit{ya}$ **in** *exI*, *rule conjI*, *simp add*: *move-graph-a-def nodes-def atI-def*,
(*erule conjE*)+, *simp add*: *move-graph-a-def*, *case-tac* $\mathit{ya} = l$, *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 $\mathit{ya} = l'$, *simp*+)
qed
qed

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

```
    (I, y) ∈ {(x::infrastructure, y::infrastructure). x →n y}* ⟹
    (y, z) ∈ {(x::infrastructure, y::infrastructure). x →n y} ⟹
    actors-graph (graphI I) = actors-graph (graphI y) ⟹
    actors-graph (graphI I) = actors-graph (graphI z)
  by (drule CollectD, simp, drule same-actors0, simp)
qed

end
end
```

3 Airplane case study

In this section we first provide the necessary infrastructure, then specify global and local policies, and finally formalize insider attacks and safety and security.

theory *Airplane*
imports *AirInsider*
begin

3.1 Formalization of Airplane Infrastructure and Properties

We restrict the Airplane scenario to four identities: Bob, Charly, Alice, and Eve. Bob acts as the pilot, Charly as the copilot, and Alice as the flight attendant. Eve is an identity representing the malicious agent that can act as the copilot although not officially acting as an airplane actor. The identities that act legally inside the airplane infrastructure are listed in the set of airplane actors.

To represent the layout of the airplane, a simple architecture is best suited for the purpose of security policy verification. The locations we consider for the graph are *cockpit*, *door*, and *cabin*. They are defined as locale definitions and assembled in a set *airplane-locations*.

The actual layout and the initial distribution of the actors in the airplane infrastructure is defined by the graph *ex-graph* in which the actors Bob and Charly are in the cockpit and Alice is in the cabin.

The two additional inputs *ex-creds* and *ex-locs* for the constructor *Lgraph* are the credential and role assignment to actors and the state function for locations introduced in Section **??**, respectively. For the airplane scenario, we use the function *ex-creds* to assign the roles and credentials to actors. For example, for *Actor ''Bob''* this function returns the pair of lists ([*''PIN''*], [*''pilot''*]) assigning the credential *PIN* to this actor and designating the role *pilot* to him. Similar to the previous function *ex-creds*, the function *ex-locs* assigns values to the locations of the infrastructure graph. These values are simply of type string allowing to store arbitrary state information about the locations, for example, the door is "locked" or the airplane is on the

”ground”.

In the Isabelle Insider framework, we define a global policy reflecting the global safety and security goal and then break that down into local policies on the infrastructure. The verification will then analyze whether the infrastructure's local policies yield the global policy.

subsection *Initial Global and Local Policies* Globally, we want to exclude attackers to ground the plane. In the formal model, landing the airplane results from an actor performing a *put* action in the cockpit and thereby changing the state from *air* to *ground*.

Therefore, we specify the global policy as “no one except airplane actors can perform *put* actions at location cockpit” by the following predicate over infrastructures I and actor identities a.

We next attempt to define the *local-policies* for each location as a function mapping locations to sets of pairs: the first element of each pair for a location l is a predicate over actors specifying the conditions necessary for an actor to be able to perform the actions specified in the set of actions which is the second element of that pair. Local policy functions are additionally parameterized over an infrastructure graph G since this may dynamically change through the state transition. The policy *local-policies* expresses that any actor can move to door and cabin but places the following restrictions on cockpit.

put: to perform a *put* action, that is, put the plane into a new position or put the lock, an actor must be at position cockpit, i.e., in the cockpit;

move: to perform a move action at location cockpit, that is, move into it, an actor must be at the position cabin, must be in possession of PIN, and door must be in state norm.

Although this policy abstracts from the buzzer, the 30 sec delay, and a few other technical details, it captures the essential features of the cockpit door.

The graph, credentials, and features are plugged together with the policy into the infrastructure *Airplane-scenario*.

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] ⇒ apolicy 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] ⇒ apolicy 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
```

3.2 Insider Attack, Safety, and Security

Above, we first stage the insider attack and introduce basic definitions of safety and security for the airplane scenario. To invoke the insider within an application of the Isabelle Insider framework, we assume in the locale *airplane* as a locale assumption with *assumes* that the tipping point has been reached for *Eve* which manifests itself in her *actor-state* assigned by the locale function *astate*.

In addition, we state that she is an insider being able to impersonate *Charly* by locally assuming the *Insider* predicate. This predicate allows an insider to impersonate a set of other actor identities; in this case the set is singleton.

Next, the process of analysis uses this assumption as well as the definitions of the previous section to prove security properties interactively as theorems in Isabelle. We use the strong insider assumption here up front to provide a first sanity check on the model by validating the infrastructure for the "normal" case. We prove that the global policy holds for the pilot Bob.

```
lemma ex-inv: global-policy Airplane-scenario ''Bob''
by (simp add: Airplane-scenario-def global-policy-def airplane-actors-def)
```

We can prove the same theorem for *Charly* who is the copilot in the scenario (omitting the proof and accompanying Isabelle commands).

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

But *Eve* is an insider and is able to impersonate *Charly*. She will ignore the global policy. This insider threat can now be formalized as an invalidation of the global company policy for *''Eve''* in the following "attack" theorem named *ex-inv3*:

```
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

Safety and security are sometimes introduced in textbooks as complementary properties, see, e.g., [1]. Safety expresses that humans and goods should be protected from negative effects caused by machines while security is the inverse direction: machines (computers) should be protected from malicious humans. Similarly, the following descriptions of safety and security in the airplane scenario also illustrate this complementarity: one says that the door must stay closed to the outside; the other that there must be a possibility to open it from the outside.

Safety: if the actors in the cockpit are out of action, there must be a possibility to get into the cockpit from the cabin, and

Security: if the actors in the cockpit fear an attack from the cabin, they can lock the door.

In the formal translation of these properties into HOL, this complementarity manifests itself even more clearly: the conclusions of the two formalizations of the properties are negations of each other. Safety is quite concisely described by stating that airplane actors can move into the cockpit.

We 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

Security can also be defined in a simple manner as the property that no actor can move into the cockpit if the door is on lock.

We show Security for *Airplane-scenario*. We need some lemmas first that use the injectivity of the *is-in* predicate to infer that the lock and the norm states of the door must be actually different.

lemma *inj-lem*: $[\![$ *inj f*; $x \neq y$ $]\!] \Longrightarrow f\,x \neq f\,y$
by (*simp add*: *inj-eq*)

lemma *inj-on-lem*: $[\![$ *inj-on f A*; $x \neq y$; $x \in A$; $y \in A$ $]\!] \Longrightarrow f\,x \neq f\,y$
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*)

In general, we could prove safety for any airplane actor who is in the cabin for this state of the infrastructure.

In a slightly more complex proof, we can prove security for any other identity which can be simply instantiated to *″Bob″*, for example.

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

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

We 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*)

We show that in *Airplane-in-danger*, the copilot can still do *put* and therefore can *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
```

We show that with the four eyes rule in *Airplane-not-in-danger* Eve cannot crash the 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
```

The simple formalizations of safety and security enable proofs only over a particular state of the airplane infrastructure at a time but this is not enough since the general airplane structure is subject to state changes. For a general verification, we need to prove that the properties of interest are preserved under potential changes. Since the airplane infrastructure permits, for example, that actors move about inside the airplane, we need to verify safety and security properties in a dynamic setting. After all, the insider attack on Germanwings Flight 9525 appeared when the pilot had moved out of the cockpit. Furthermore, we want to redefine the policy into the two-person policy and examine whether safety and security are improved. For these reasons, we next apply the general Kripke structure mechanism introduced initially to the airplane scenario.

4 Analysis of Safety and Security Properties

For the analysis of security, we need to ask whether the infrastructure state *Airplane-in-danger* is reachable via the state transition relation from the initial state. It is. We can prove the theorem *step-all-r* in the locale *airplane.* As the name of this theorem suggests it is the result of lining up a sequence of steps that lead from the initial *Airplane-scenario* to that *Airplane-in-danger* state (for the state definitions see the above defintion section of the locale). In fact there are three steps via two intermediary infrastructure states *Airplane-getting-in-danger0* and *Airplane-getting-in-danger.* The former encodes the state where *Bob* has moved to the cabin and the latter encodes the successor state in which additionally the lock state has changed to *locked.*

```
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* $\rightarrow_n *$ *Airplane-getting-in-danger0*
  by (*simp add*: *state-transition-in-refl-def*, *insert step0*, *auto*)

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

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

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

Using the formalization of CTL over Kripke structures introduced initiall, we can now transform the attack sequence represented implicitly by the above theorem *step-allr* into a temporal logic statement. This attack theorem states that there is a path from the initial state of the Kripke structure *Air-Kripke* on which eventually the global policy is violated by the attacker.

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

The proof uses the underlying formalization of CTL and the lemmas that are provided to evaluate the *EF* statement on the Kripke structure. However, the attack sequence is already provided by the previous theorem. So the proof just consists in supplying the step lemmas for each step and finally proving that for the state at the end of the attack path, i.e., for *Airplane-in-danger*, the global policy is violated. This proof corresponds precisely to the proof of the attack theorem *ex-inv3*. It is not surprising that the security attack is possible in the reachable state *Airplane-in-danger* when it was already possible in the initial state. However, this statement is not satisfactory since the model does not take into account whether the copilot is on his own when he launches the attack. This is the purpose of the two-person rule which we want to investigate in more detail in this paper. Therefore, we next address how to add the two-person role to the model.

4.1 Introduce Two-Person Rule

To express the rule that two authorized personnel must be present at all times in the cockpit, we have define a second set of local policies *local-policies-four-eues* (see above). It realizes the two-person constraint requesting that the number of actors at the location *cockpit* in the graph G given as input must be at least two to enable actors at the location to perform the action *put*. Formally, we can express this here as $2 \leq length(agra\ G\ cockpit)$ since we have all of arithmetic available (remember *agra G y* is the list of actors at location y in G.

Note that the two-person rule requires three people to be at the cockpit before one of them can leave. This is formalized as a condition on the *move* action of location *door*. A move of an actor x in the cockpit to *door* is only allowed if three people are in the cockpit. Practically, it enforces a person, say Alice to first enter the cockpit before the pilot Bob can leave. However, this condition is necessary to guarantee that the two-person requirement for *cockpit* is sustained by the dynamic changes to the infrastructure state caused by actors' moves. A move to location *cabin* is only allowed from *door* so no additional condition is necessary here.

What is stated informally above seems intuitive and quite easy to believe. However, comparing to the earlier formalization of this two-person rule [2], it appears that the earlier version did not have the additional condition on the action *move* to *door*. One may argue that in the earlier version the authors did not consider this because they had neither state transitions, Kripke structures, nor CTL to consider dynamic changes. However, in the current paper this additional side condition only occurred to us when we tried to prove the invariant *two-person-invariant1* which is needed in the subsequent security proof.

The proof of *two-person-invariant1* requires an induction over the state transition relation starting in the infrastructure state *Airplane-not-in-danger-init* (see above) with Charly and Bob in the cockpit and the two-person policy in place.

The corresponding Kripke structure of all states originating in this infrastructure state is defined as *Air-tp-Kripke*. Within the induction for the proof of the above *two-person-inv1*, a preservation lemma is required that proves that if the condition

$2 \leq length\ (agra\ (graphI\ I)\ cockpit)$

holds for I and $I \to I'$ then it also holds for I'. The preservation lemma is actually trickier to prove. It uses a case analysis over all the transition rules for each action. The rules for *put* and *get* are easy to prove for the user as they are solved by the simplification tactic automatically. The case for action *move* is the difficult case. Here we actually need to use the precondition of the policy for location *door* in order to prove that the two-person invariant is preserved by an actor moving out of the cockpit. In this case, we need for example, invariants like the following lemma *actors-unique-loc-aid-step* that shows that in any infrastructure state originating from *Airplane-not-in-danger-init* actors only ever appear in one location and they do not appear more than once in a location – which is expressed in the predicate *nodup* (see above).

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

lemma *actors-unique-loc-base*:
  assumes $I \rightarrow_n I'$
    and $(\forall\ l\ l'.\ a\ @_{graphI\ I}\ l \wedge a\ @_{graphI\ I}\ l' \longrightarrow l = l') \wedge$
      $(\forall\ l.\ nodup\ a\ (agra\ (graphI\ I)\ l))$
   shows $(\forall\ l\ l'.\ a\ @_{graphI\ I'}\ l \wedge a\ @_{graphI\ I'}\ l' \longrightarrow l = l') \wedge$
      $(\forall\ l.\ nodup\ a\ (agra\ (graphI\ I')\ l))$
proof (*rule state-transition-in.cases*, *rule assms*(*1*))
  show $\bigwedge(G{::}igraph)\ (Ia{::}infrastructure)\ (aa{::}char\ list)\ (l{::}location)\ (a'{::}char\ list)$ $(z{::}char\ list)$
    $I'a{::}infrastructure.$
    $I = Ia \Longrightarrow$
    $I' = I'a \Longrightarrow$
    $G = graphI\ Ia \Longrightarrow$
    $aa\ @_G\ l \Longrightarrow$
    $a'\ @_G\ l \Longrightarrow$
    $has\ G\ (Actor\ aa,\ z) \Longrightarrow$
    $enables\ Ia\ l\ (Actor\ aa)\ get \Longrightarrow$
    $I'a =$
    $Infrastructure$
     $(Lgraph\ (gra\ G)\ (agra\ G)$
       $((cgra\ G)(Actor\ a' := (z\ \#\ fst\ (cgra\ G\ (Actor\ a')),\ snd\ (cgra\ G\ (Actor\ a'))))))\ (lgra\ G))$
     $(delta\ Ia) \Longrightarrow$
    $(\forall (l{::}location)\ l'{::}location.\ a\ @_{graphI\ I'}\ l \wedge a\ @_{graphI\ I'}\ l' \longrightarrow l = l') \wedge$
    $(\forall\ l{::}location.\ nodup\ a\ (agra\ (graphI\ I')\ l))$ **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 $\bigwedge(G{::}igraph)\ (Ia{::}infrastructure)\ (l{::}location)\ (aa{::}char\ list)\ (I'a{::}infrastructure)$
    $z{::}char\ list.$
    $I = Ia \Longrightarrow$
    $I' = I'a \Longrightarrow$
    $G = graphI\ Ia \Longrightarrow$
    $enables\ Ia\ l\ (Actor\ aa)\ put \Longrightarrow$

$I'a = \mathit{Infrastructure}\ (\mathit{Lgraph}\ (\mathit{gra}\ G)\ (\mathit{agra}\ G)\ (\mathit{cgra}\ G)\ ((\mathit{lgra}\ G)(l := [z])))$
$(\mathit{delta}\ \mathit{Ia}) \Longrightarrow$

$(\forall\,(l{::}\mathit{location})\ l'{::}\mathit{location}.\ a\ @_{\mathit{graphI}\ I'}\ l \wedge a\ @_{\mathit{graphI}\ I'}\ l' \longrightarrow l = l') \wedge$

$(\forall\,l{::}\mathit{location}.\ \mathit{nodup}\ a\ (\mathit{agra}\ (\mathit{graphI}\ I')\ l))$

by (*clarify*, *simp add*: *assms atI-def*)

next show $\bigwedge(G{::}\mathit{igraph})\ (\mathit{Ia}{::}\mathit{infrastructure})\ (\mathit{aa}{::}\mathit{char\ list})\ (l{::}\mathit{location})\ (l'{::}\mathit{location})$

$I'a{::}\mathit{infrastructure}.$

$I = \mathit{Ia} \Longrightarrow$

$I' = I'a \Longrightarrow$

$G = \mathit{graphI}\ \mathit{Ia} \Longrightarrow$

$\mathit{aa}\ @_{G}\ l \Longrightarrow$

$l \in \mathit{nodes}\ G \Longrightarrow$

$l' \in \mathit{nodes}\ G \Longrightarrow$

$\mathit{aa} \in \mathit{actors\text{-}graph}\ (\mathit{graphI}\ \mathit{Ia}) \Longrightarrow$

$\mathit{enables}\ \mathit{Ia}\ l'\ (\mathit{Actor}\ \mathit{aa})\ \mathit{move} \Longrightarrow$

$I'a = \mathit{Infrastructure}\ (\mathit{move\text{-}graph\text{-}a}\ \mathit{aa}\ l\ l'\ (\mathit{graphI}\ \mathit{Ia}))\ (\mathit{delta}\ \mathit{Ia}) \Longrightarrow$

$(\forall\,(l{::}\mathit{location})\ l'{::}\mathit{location}.\ a\ @_{\mathit{graphI}\ I'}\ l \wedge a\ @_{\mathit{graphI}\ I'}\ l' \longrightarrow l = l') \wedge$

$(\forall\,l{::}\mathit{location}.\ \mathit{nodup}\ a\ (\mathit{agra}\ (\mathit{graphI}\ I')\ l))$

proof (*simp add*: *move-graph-a-def*,*rule conjI*, *clarify*, *rule conjI*, *clarify*, *rule conjI*, *clarify*)

show $\bigwedge(G{::}\mathit{igraph})\ (\mathit{Ia}{::}\mathit{infrastructure})\ (\mathit{aa}{::}\mathit{char\ list})\ (l{::}\mathit{location})\ (l'{::}\mathit{location})$

$(I'a{::}\mathit{infrastructure})\ (\mathit{la}{::}\mathit{location})\ l'a{::}\mathit{location}.$

$I' =$

$\mathit{Infrastructure}$

$(\mathit{Lgraph}\ (\mathit{gra}\ (\mathit{graphI}\ I))$

$(\mathit{if}\ a \in \mathit{set}\ (\mathit{agra}\ (\mathit{graphI}\ I)\ l) \wedge a \notin \mathit{set}\ (\mathit{agra}\ (\mathit{graphI}\ I)\ l')$

$\mathit{then}\ (\mathit{agra}\ (\mathit{graphI}\ I))(l := \mathit{del}\ a\ (\mathit{agra}\ (\mathit{graphI}\ I)\ l),\ l' := a\ \#\ \mathit{agra}$
$(\mathit{graphI}\ I)\ l')$

$\mathit{else}\ \mathit{agra}\ (\mathit{graphI}\ I))$

$(\mathit{cgra}\ (\mathit{graphI}\ I))\ (\mathit{lgra}\ (\mathit{graphI}\ I)))$

$(\mathit{delta}\ I) \Longrightarrow$

$a\ @_{\mathit{graphI}\ I}\ l \Longrightarrow$

$l \in \mathit{nodes}\ (\mathit{graphI}\ I) \Longrightarrow$

$l' \in \mathit{nodes}\ (\mathit{graphI}\ I) \Longrightarrow$

$a \in \mathit{actors\text{-}graph}\ (\mathit{graphI}\ I) \Longrightarrow$

$\mathit{enables}\ I\ l'\ (\mathit{Actor}\ a)\ \mathit{move} \Longrightarrow$

$a \in \mathit{set}\ (\mathit{agra}\ (\mathit{graphI}\ I)\ l) \Longrightarrow$

$a \notin \mathit{set}\ (\mathit{agra}\ (\mathit{graphI}\ I)\ l') \Longrightarrow$

$a\ @_{\mathit{Lgraph}\ (\mathit{gra}\ (\mathit{graphI}\ I))\qquad ((\mathit{agra}\ (\mathit{graphI}\ I))(l := \mathit{del}\ a\ (\mathit{agra}\ (\mathit{graphI}\ I)\ l),\ l' := a\ \#\ \mathit{agra}\ (\mathit{graphI}\ I)}$
$\mathit{la} \Longrightarrow$

$a\ @_{\mathit{Lgraph}\ (\mathit{gra}\ (\mathit{graphI}\ I))\qquad ((\mathit{agra}\ (\mathit{graphI}\ I))(l := \mathit{del}\ a\ (\mathit{agra}\ (\mathit{graphI}\ I)\ l),\ l' := a\ \#\ \mathit{agra}\ (\mathit{graphI}\ I)}$
$l'a \Longrightarrow$

$\mathit{la} = l'a$

apply $(\mathit{case\text{-}tac}\ \mathit{la} \neq l' \wedge \mathit{la} \neq l \wedge l'a \neq l' \wedge l'a \neq l)$

apply (*simp add*: *atI-def*)

apply $(\mathit{subgoal\text{-}tac}\ \mathit{la} = l' \vee \mathit{la} = l \vee l'a = l' \vee l'a = l)$

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

next show $\bigwedge(G{::}igraph)\ (Ia{::}infrastructure)\ (aa{::}char\ list)\ (l{::}location)\ (l'{::}location)$

$I'a{::}infrastructure.$

$I = Ia \Longrightarrow$

$I' =$

$Infrastructure$

$(Lgraph\ (gra\ (graphI\ Ia))$

$(if\ aa \in set\ (agra\ (graphI\ Ia)\ l) \wedge aa \notin set\ (agra\ (graphI\ Ia)\ l')$

$then\ (agra\ (graphI\ Ia))(l := del\ aa\ (agra\ (graphI\ Ia)\ l),\ l' := aa\ \#\ agra$
$(graphI\ Ia)\ l')$

$else\ agra\ (graphI\ Ia))$

$(cgra\ (graphI\ Ia))\ (lgra\ (graphI\ Ia)))$

$(delta\ Ia) \Longrightarrow$

$G = graphI\ Ia \Longrightarrow$

$aa\ @_{graphI\ Ia}\ l \Longrightarrow$

$l \in nodes\ (graphI\ Ia) \Longrightarrow$

$l' \in nodes\ (graphI\ Ia) \Longrightarrow$

$aa \in actors\text{-}graph\ (graphI\ Ia) \Longrightarrow$

$enables\ Ia\ l'\ (Actor\ aa)\ move \Longrightarrow$

$I'a =$

$Infrastructure$

$(Lgraph\ (gra\ (graphI\ Ia))$

$(if\ aa \in set\ (agra\ (graphI\ Ia)\ l) \wedge aa \notin set\ (agra\ (graphI\ Ia)\ l')$

$then\ (agra\ (graphI\ Ia))(l := del\ aa\ (agra\ (graphI\ Ia)\ l),\ l' := aa\ \#\ agra$
$(graphI\ Ia)\ l')$

$else\ agra\ (graphI\ Ia))$

$(cgra\ (graphI\ Ia))\ (lgra\ (graphI\ Ia)))$

$(delta\ Ia) \Longrightarrow$

$aa \neq a \longrightarrow$

$(aa \in set\ (agra\ (graphI\ Ia)\ l) \wedge aa \notin set\ (agra\ (graphI\ Ia)\ l') \longrightarrow$

$(\forall (la{::}location)\ l'a{::}location.$

$a\ @_{Lgraph\ (gra\ (graphI\ Ia))\qquad ((agra\ (graphI\ Ia))\qquad (l := del\ aa\ (agra\ (graphI\ Ia)\ l),\ l}$
$la\ \wedge$

$a\ @_{Lgraph\ (gra\ (graphI\ Ia))\qquad ((agra\ (graphI\ Ia))\qquad (l := del\ aa\ (agra\ (graphI\ Ia)\ l),\ l}$
$l'a \longrightarrow$

$la = l'a) \wedge$

$(\forall\ la{::}location.$

$(la = l \longrightarrow$

$(l = l' \longrightarrow nodup\ a\ (agra\ (graphI\ Ia)\ l')) \wedge$

$(l \neq l' \longrightarrow nodup\ a\ (del\ aa\ (agra\ (graphI\ Ia)\ l)))) \wedge$

$(la \neq l \longrightarrow$

$(la = l' \longrightarrow nodup\ a\ (agra\ (graphI\ Ia)\ l')) \wedge$

$(la \neq l' \longrightarrow nodup\ a\ (agra\ (graphI\ Ia)\ la))))) \wedge$

$((aa \in set\ (agra\ (graphI\ Ia)\ l) \longrightarrow aa \in set\ (agra\ (graphI\ Ia)\ l')) \longrightarrow$

$(\forall (l{::}location)\ l'{::}location.$

$a\ @_{Lgraph\ (gra\ (graphI\ Ia))\ (agra\ (graphI\ Ia))\ (cgra\ (graphI\ Ia))\qquad (lgra\ (graphI\ Ia))}$
$l\ \wedge$

$a\ @_{Lgraph\ (gra\ (graphI\ Ia))\ (agra\ (graphI\ Ia))\ (cgra\ (graphI\ Ia))\qquad (lgra\ (graphI\ Ia))}$
$l' \longrightarrow$

```
          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 ≠ l ⟶
           (l′a = l′ ⟶ a ∈ set (agra (graphI I) la) ⟶ a ∉ set (agra (graphI I) l′)) ∧
           (l′a ≠ l′ ⟶
            a ∈ set (agra (graphI I) la) ∧ a ∈ set (agra (graphI I) l′a) ⟶ la = l′a))))
      by (meson assms(2) atI-def del-notin-down)
  next show ⋀(aa::char list) (l::location) l′::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) ⟹
```

$enables\ I\ l'\ (Actor\ aa)\ move \Longrightarrow$
$aa \neq a \Longrightarrow$
$aa \notin set\ (agra\ (graphI\ I)\ l') \Longrightarrow$
$\forall\ la{::}location.$
$(la = l \longrightarrow$
$(l = l' \longrightarrow nodup\ a\ (agra\ (graphI\ I)\ l')) \wedge$
$(l \neq l' \longrightarrow nodup\ a\ (del\ aa\ (agra\ (graphI\ I)\ l)))) \wedge$
$(la \neq l \longrightarrow$
$(la = l' \longrightarrow nodup\ a\ (agra\ (graphI\ I)\ l')) \wedge (la \neq l' \longrightarrow nodup\ a\ (agra\ (graphI\ I)\ la)))$
by $(simp\ add{:}\ assms(2)\ nodup\text{-}down\text{-}notin)$
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 $assms(2)\ atI\text{-}def$ **by** $blast$
qed
qed
qed

lemma $actors\text{-}unique\text{-}loc\text{-}step$:
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 $\bigwedge a{::}infrastructure.$

$(\forall\ \mathit{aa}::\mathit{char\ list}.$
$(\forall\ (l::\mathit{location})\ l'::\mathit{location}.\ \mathit{aa}\ @_{\mathit{graphI}\ a}\ l \wedge \mathit{aa}\ @_{\mathit{graphI}\ a}\ l' \longrightarrow l = l') \wedge$
$(\forall\ l::\mathit{location}.\ \mathit{nodup}\ \mathit{aa}\ (\mathit{agra}\ (\mathit{graphI}\ a)\ l))) \longrightarrow$
$(\forall\ \mathit{aa}::\mathit{char\ list}.$
$(\forall\ (l::\mathit{location})\ l'::\mathit{location}.\ \mathit{aa}\ @_{\mathit{graphI}\ a}\ l \wedge \mathit{aa}\ @_{\mathit{graphI}\ a}\ l' \longrightarrow l = l') \wedge$
$(\forall\ l::\mathit{location}.\ \mathit{nodup}\ \mathit{aa}\ (\mathit{agra}\ (\mathit{graphI}\ a)\ l)))$ **by** *simp*
next show $\bigwedge(a::\mathit{infrastructure})\ (b::\mathit{infrastructure})\ (c::\mathit{infrastructure}).$
$(a,\ b) \in \{(x::\mathit{infrastructure},\ y::\mathit{infrastructure}).\ x \rightarrow_n y\}^* \Longrightarrow$
$(\forall\ \mathit{aa}::\mathit{char\ list}.$
$(\forall\ (l::\mathit{location})\ (l'::\mathit{location}).\ (\mathit{aa}\ @_{\mathit{graphI}\ a}\ l \wedge \mathit{aa}\ @_{\mathit{graphI}\ a}\ l') \longrightarrow l =$
$l') \wedge$
$(\forall\ l::\mathit{location}.\ \mathit{nodup}\ \mathit{aa}\ (\mathit{agra}\ (\mathit{graphI}\ a)\ l))) \longrightarrow$
$(\forall\ a::\mathit{char\ list}.$
$(\forall\ (l::\mathit{location})\ (l'::\mathit{location}).\ (a\ @_{\mathit{graphI}\ b}\ l \wedge a\ @_{\mathit{graphI}\ b}\ l') \longrightarrow l = l') \wedge$
$(\forall\ l::\mathit{location}.\ \mathit{nodup}\ a\ (\mathit{agra}\ (\mathit{graphI}\ b)\ l))) \Longrightarrow$
$(b,\ c) \in \{(x::\mathit{infrastructure},\ y::\mathit{infrastructure}).\ x \rightarrow_n y\} \Longrightarrow$
$(\forall\ \mathit{aa}::\mathit{char\ list}.$
$(\forall\ (l::\mathit{location})\ l'::\mathit{location}.\ (\mathit{aa}\ @_{\mathit{graphI}\ a}\ l \wedge \mathit{aa}\ @_{\mathit{graphI}\ a}\ l') \longrightarrow l = l')$
$\wedge$
$(\forall\ l::\mathit{location}.\ \mathit{nodup}\ \mathit{aa}\ (\mathit{agra}\ (\mathit{graphI}\ a)\ l))) \longrightarrow$
$(\forall\ a::\mathit{char\ list}.$
$(\forall\ (l::\mathit{location})\ l'::\mathit{location}.\ (a\ @_{\mathit{graphI}\ c}\ l \wedge a\ @_{\mathit{graphI}\ c}\ l') \longrightarrow l = l') \wedge$
$(\forall\ l::\mathit{location}.\ \mathit{nodup}\ a\ (\mathit{agra}\ (\mathit{graphI}\ c)\ l)))$
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*:
$\forall\ \ a.\ (\forall\ \ l\ l'.\ a\ @_{\mathit{graphI\ Airplane\text{-}not\text{-}in\text{-}danger\text{-}init}}\ l \wedge$
$a\ @_{\mathit{graphI\ Airplane\text{-}not\text{-}in\text{-}danger\text{-}init}}\ l' \longrightarrow l = l') \wedge$
$(\forall\ \ l.\ \mathit{nodup}\ a\ (\mathit{agra}\ (\mathit{graphI\ Airplane\text{-}not\text{-}in\text{-}danger\text{-}init})\ l))$
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 $\bigwedge(l::\mathit{location})\ l'::\mathit{location}.$
''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* $l = \mathit{cockpit} \vee l = \mathit{door} \vee$
$l = \mathit{cabin}$,
erule disjE, *simp*, *case-tac* $l' = \mathit{door} \vee l' = \mathit{cabin}$, *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 the 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 $\{x{::}char\ list.$
$\exists\, y{::}location.$
$(y = door \longrightarrow$
$(door = cockpit \longrightarrow$
$(\exists\, y{::}location.\ y = cockpit \vee y = cabin \vee y = cockpit \vee y = cockpit \wedge cockpit = cabin) \wedge$
$(x = ''Bob'' \vee x = ''Charly'')) \wedge$
$door = cockpit) \wedge$
$(y \neq door \longrightarrow$
$(y = cockpit \longrightarrow$
$(\exists\, y{::}location.$
$y = door \vee$
$cockpit = door \wedge y = cabin \vee$
$y = cockpit \wedge cockpit = door \vee y = door \wedge cockpit = cabin) \wedge$
$(x = ''Bob'' \vee x = ''Charly'')) \wedge$

```
         (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-not-in-danger-init, y) ∈ {(x::infrastructure,
y::infrastructure). x →ₙ y}*
             ⟹ actors-graph(graphI y) = airplane-actors
  by (insert Anid-airplane-actors, erule subst, rule sym, erule same-actors)

lemma actors-at-loc-in-graph: ⟦ l ∈ nodes(graphI I); a @graphI I l⟧
                              ⟹ a ∈ actors-graph (graphI I)
  by (simp add: atI-def actors-graph-def, rule exI, rule conjI)

lemma not-en-get-Apnid:
 assumes (Airplane-not-in-danger-init,y) ∈ {(x::infrastructure, y::infrastructure).
x →ₙ y}*
  shows  ~(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: ~(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*: ~(*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*)

The following invariant shows that the number of staff in the cockpit is never below 2.

lemma *two-person-inv*:
fixes *z z'*
assumes (*2*::*nat*) ≤ *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-not-in-danger-init*,*z*) ∈ {(*x*::*infrastructure*, *y*::*infrastructure*). $x \rightarrow_n y\}^*$
and $z \rightarrow_n z'$
shows (*2*::*nat*) ≤ *length* (*agra* (*graphI z'*) *cockpit*)
proof (*insert assms*(*5*), *erule state-transition-in.cases*)
show ⋀(*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*) ⟹
(*2*::*nat*) ≤ *length* (*agra* (*graphI z'*) *cockpit*) **using** *assms* **by** *simp*
next show ⋀(*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* $\Longrightarrow$
$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* $\Rightarrow$ *char list* **and** *ccsa* :: *char list* $\Rightarrow$ *char list* **where**
f2: $\forall$ *cs csa*. ($\neg$ *UasI cs csa* $\vee$ *Actor cs* = *Actor csa* $\wedge$ ($\forall$ *csa csb*. (*csa* = *cs* $\vee$ *csb* = *cs* $\vee$ *Actor csa* $\neq$ *Actor csb*) $\vee$ *csa* = *csb*)) $\wedge$ (*UasI cs csa* $\vee$ *Actor cs* $\neq$ *Actor csa* $\vee$ (*ccs cs* $\neq$ *cs* $\wedge$ *ccsa cs* $\neq$ *cs* $\wedge$ *Actor* (*ccs cs*) = *Actor* (*ccsa cs*)) $\wedge$ *ccs cs* $\neq$ *ccsa cs*)
using *UasI-def* **by** *moura*
have ''*Bob*'' $@_{\mathit{graphI}\ (\mathit{Infrastructure\ ex\text{-}graph\ local\text{-}policies})}$ *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* $\leq$ *length* (*agra* (*graphI z'*) *cockpit*)
using *f2 f1* **by** *auto*
qed
qed

lemma *two-person-inv1*:
assumes (*Airplane-not-in-danger-init*,*z*) $\in$ {(*x*::*infrastructure*, *y*::*infrastructure*). $x \rightarrow_n y\}^*$

```
  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
→n y}* ⟹
       (y, z) ∈ {(x::infrastructure, y::infrastructure). x →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, that we need, uses cardinality of lists of actors rather than length of lists. Therefore, we first need some equivalences to then prove a restatement of *two-person-inv* in terms of sets.

The proof idea is to 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) ∈ {(x::infrastructure, y::infrastructure).
x →n y}*
    shows (2::nat) ≤ 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
```

4.2 Revealing Necessary Assumption by Proof Failure

We would expect – and this has in fact been presented in [2] – that the two-person rule guarantees the absence of the insider attack.

This is indeed a provable fact in the following state *Airplane-not-in-danger* defined similar to *Airplane-in-danger* from Section **??** but using the two-person policy.

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

For this state, it can be proved [2] that for any actor identity a the global policy holds.

global-policy Airplane-not-in-danger a

So, in the state *Airplane-not-in-danger* with the two-person rule, there seems to be no danger. But this is precisely the scenario of the suicide attack! Charly is on his own in the cockpit – why then does the two-person rule imply he cannot act?

The state *Airplane-not-in-danger* defined in the earlier formalization is misnamed: it uses the graph *aid-graph* to define a state in which Bob has left the cockpit and the door is locked. Since there is only one actor present, the precondition of the local policy for *cockpit* is not met and hence the action *put* is not enabled for actor Charly. Thus, the policy rule for cockpit is true because the precondition of this implication – two people in the cockpit – is false, and false implies anything: seemingly a disastrous failure of logic.

Fortunately, the above theorem has been derived in a preliminary model only [2] in which state changes were not integrated yet and which has been precisely for this reason recognized as inadequate. Now, with state changes in the improved model, we have proved the two-person invariant *two-person-inv1*. Thus, we can see that the system – if started in *Airplane-not-in-danger-init* – cannot reach the mis-named state *Airplane-not-in-danger* in which Charly is on his own in the cockpit.

However, so far, no such general theorem has been proved yet. We only used CTL to discover attacks using EF formulas. What we need for general security and what we consider next is to prove a global property with the temporal operator AG that proves that from a given initial state the global policy holds in all (A) states globally (G).

As we have seen in the previous section when looking at the proof of *two-person-inv1*, it is not evident and trivial to prove that all state changes preserve security properties. However, even this invariant does not suffice.

Even if the two-person rule is successfully enforced in a state, it is on its own still not sufficient. When we try to prove

Air-tp-Kripke $\vdash AG$ \{*x. global-policy x ''Eve''*\}

for the Kripke structure *Air-tp-Kripke* of all states originating in *Airplane-not-in-danger-init*, we cannot succeed. In fact, in that Kripke structure there are infrastructure states where the insider attack is possible. Despite the fact that we have stipulated the two-person rule as part of the new policy and despite the fact that we can prove that this policy is preserved by all state changes, the rule has no consequence on the insider. Since Eve can impersonate the copilot Charly, whether two people are in the cockpit or not, the attack can happen.

What we realize through this failed attempt to prove a global property is that the policy formulation does not entail that the presence of two people in itself actually disables an attacker.

This insight reveals a hidden assumption. Formal reasoning systems have the advantage that hidden assumptions must be made explicit. In human reasoning they occur when people assume a common understanding, which may or may not be actually the case. In the case of the rule above, its purpose may lead to an assumption that humans accept but which is not warranted.

We have used above a locale definition to encode this intentional understanding of the two-person rule. The formula *foe-control* encodes for any action c at a location l that if there is an *Actor* x that is not an insider, that is, is not impersonated by Eve, then the insider is disabled for that action c.

4.3 Proving Security in Refined Model

Having identified the missing formulation of the intentional effects of the two-person rule, we can now finally prove the general security property using the above locale definition. We assume in the locale *airplane* an instance of *foe-control* for the cockpit and the action *put*.

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

With this assumption, we are now able to prove

theorem Four-eyes-no-danger: *Air-tp-Kripke* $\vdash$ *AG* $\{x.$ *global-policy* x *''Eve''*$\}$

that is, for all infrastructure states of the system *airplane* originating in state *Airplane-not-in-danger-init* Eve cannot put the airplane to the ground.

The proof uses as the key lemma *tp-imp-control* that within Kripke structure *Air-tp-Kripke* there is always someone in the cockpit who is not the insider. For this lemma, we first need some preparation.

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* $P = \lambda\ x.\ x \in S$ **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-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 $\Longrightarrow$
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-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-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 $\Longrightarrow$
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-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-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* $\Longrightarrow$
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*) $\Longrightarrow$ *h* $\in$
airplane-actors
proof (*simp add*: *move-graph-a-def*,
case-tac a $\in$ *set* (*agra* (*graphI I*) *l*) $\wedge$ *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 $\in$ *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*
$\rightarrow_n$ *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-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*) ⟹
¬ (*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* $\to_n$ *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*
proof (*case-tac l*′ = *cockpit*)
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) $\Longrightarrow$

$\forall$ *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*). $x \rightarrow_n y\}^* \Longrightarrow$

$z = I \Longrightarrow$

$z' =$

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-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* $\Longrightarrow$ $h \in$ *airplane-actors*

proof (*case-tac cockpit* = l)

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*). $x \rightarrow_n y\}^* \Longrightarrow$

$z = I \Longrightarrow$

$z' =$

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 $@_{\mathit{graphI}\ I}$ *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'*)* $\Longrightarrow$
l' $\neq$ *cockpit* $\Longrightarrow$ *cockpit* $\neq$ *l* $\Longrightarrow$ *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) (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 $@_{\mathit{graphI}\ I}$ *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 ⟹ cockpit = l ⟹ h ∈ airplane-actors
          by (simp, erule bspec, erule del-up)
            qed
        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 →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 $a \in$ 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-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-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))
((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 $\Longrightarrow$
$a\ @_{graphI\ I}\ l$ $\Longrightarrow$
$l \in$ nodes (graphI I) $\Longrightarrow$
cockpit $\in$ nodes (graphI I) $\Longrightarrow$
$a \in$ actors-graph (graphI I) $\Longrightarrow$
enables I cockpit (Actor a) move $\Longrightarrow$
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) $\Longrightarrow$
$a \in$ set (agra (graphI I) l) $\wedge$ $a \notin$ set (agra (graphI I) cockpit) $\Longrightarrow$
l′ = cockpit $\Longrightarrow$ $h \in$ set (agra (graphI I) cockpit) $\Longrightarrow$ $h \in$ airplane-actors

by (erule bspec)

next fix z z′ h G I a l l′ I′

assume a0: $\forall$ h::char list$\in$set (agra (graphI I) cockpit). $h \in$ airplane-actors

and a1: (Airplane-not-in-danger-init, I) $\in$ {(x::infrastructure, y::infrastructure). $x \to_n y\}^*$

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 →_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 →_n y}^* ⟶
    (∀ 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
→n y}* ⟶
      (∀ 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
→n y}* ⟹
       (y, z) ∈ {(x::infrastructure, y::infrastructure). x →n y} ⟹
       (Airplane-not-in-danger-init, y) ∈ {(x,y). x →n y}* ⟶
       (∀ h::char list∈set (agra (graphI y) cockpit). h ∈ airplane-actors) ⟹
       (Airplane-not-in-danger-init, z) ∈ {(x,y). x →n y}* ⟶
       (∀ 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 →n y}* ⟹
      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)
```

The 2 person invariant implies that there is always some x in cockpit where 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
```

4.4 Locale interpretation

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
```

This is the intermediate step where pilot left the cockpit but the door is still in norm position.

```
definition Airplane-getting-in-danger0-def':
Airplane-getting-in-danger0 ≡ Infrastructure aid-graph0 local-policies
```

```
definition Airplane-getting-in-danger-def':
Airplane-getting-in-danger ≡ Infrastructure agid-graph local-policies
```

```
definition Air-states-def': Air-states ≡ { I. Airplane-scenario →n* I }
```

```
definition Air-Kripke-def': Air-Kripke ≡ Kripke Air-states {Airplane-scenario}
```

```
definition Airplane-not-in-danger-def':
Airplane-not-in-danger ≡ Infrastructure aid-graph local-policies-four-eyes
```

```
definition Airplane-not-in-danger-init-def':
Airplane-not-in-danger-init ≡ Infrastructure ex-graph local-policies-four-eyes
```

```
definition Air-tp-states-def': Air-tp-states ≡ { I. Airplane-not-in-danger-init →n*
I }
```

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

Air-tp-Kripke ≡ *Kripke Air-tp-states* {*Airplane-not-in-danger-init*}

definition *Safety-def′*: *Safety I a* ≡ *a* ∈ *airplane-actors*
⟶ (*enables I cockpit* (*Actor a*) *move*)

definition *Security-def′*: *Security I a* ≡ (*isin* (*graphI I*) *door ′′locked′′*)
⟶ ¬(*enables I cockpit* (*Actor a*) *move*)

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

definition *astate-def′*: *astate x* ≡
(*case x of*
′′Eve′′ ⇒ *Actor-state depressed* {*revenge*, *peer-recognition*}
| *-* ⇒ *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 in [8]. Following this, we 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 theory *AirInsider* is a more elegant method for representing an abstract type *actor* but it is not working properly for our framework since it necessitates introducing a type parameter *′actor* into infrastructures which then makes it impossible to instantiate them to the *typeclass* state in order to use CTL and Kripke and the generic state transition. Therefore, we go the former way of a post-hoc axiomatic redefinition of the abstract type actor by using axiomatization of the existing locale *type-definition*. This is done in the following. It allows to abstractedly assume as an axiom that there is a type definition for the abstract type actor. Adding a suitable definition of a representation for this type then additionally enables to introduce a definition for the function Actor (again using axiomatization to enforce the new definition).

definition *Actor-Abs* :: *identity* ⇒ *identity option*
where

$\textit{Actor-Abs } x \equiv (\textit{if } x \in \{''\textit{Eve}'', ''\textit{Charly}''\} \textit{ then None else Some } x)$

lemma $\textit{UasI-ActorAbs}$: $\textit{Actor-Abs } ''\textit{Eve}'' = \textit{Actor-Abs } ''\textit{Charly}'' \wedge$
$(\forall (x{::}\textit{char list})\ y{::}\textit{char list}.\ x \neq ''\textit{Eve}'' \wedge y \neq ''\textit{Eve}'' \wedge \textit{Actor-Abs } x = \textit{Actor-Abs } y \longrightarrow x = y)$
by $(\textit{simp add}: \textit{Actor-Abs-def})$

lemma $\textit{Actor-Abs-ran}$: $\textit{Actor-Abs } x \in \{y :: \textit{identity option}.\ y \in \textit{Some } `\ \{x :: \textit{identity}.\ x \notin \{''\textit{Eve}'', ''\textit{Charly}''\}\}|\ y = \textit{None}\}$
by $(\textit{simp add}: \textit{Actor-Abs-def})$

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

lemma $\textit{Abs-inj-on}$: $\bigwedge \textit{Abs Rep}{::}\ \textit{actor} \Rightarrow \textit{char list option}.\ x \in \{y :: \textit{identity option}.\ y \in \textit{Some } `\ \{x :: \textit{identity}.\ x \notin \{''\textit{Eve}'', ''\textit{Charly}''\}\}|\ y = \textit{None}\}$
$\Longrightarrow y \in \{y :: \textit{identity option}.\ y \in \textit{Some } `\ \{x :: \textit{identity}.\ x \notin \{''\textit{Eve}'', ''\textit{Charly}''\}\}|\ y = \textit{None}\}$
$\Longrightarrow (\textit{Abs} :: \textit{char list option} \Rightarrow \textit{actor})\ x = \textit{Abs } y \Longrightarrow x = y$
by $(\textit{insert Actor-type-def}, \textit{drule-tac } x = \textit{Rep}$ **in** $\textit{meta-spec}, \textit{drule-tac } x = \textit{Abs}$ **in** $\textit{meta-spec},$
$\textit{frule-tac } x = \textit{Abs } x$ **and** $y = \textit{Abs } y$ **in** $\textit{type-definition.Rep-inject},$
$\textit{subgoal-tac } (\textit{Rep } (\textit{Abs } x) = \textit{Rep } (\textit{Abs } y)), \textit{subgoal-tac Rep } (\textit{Abs } x) = x,$
$\textit{subgoal-tac Rep } (\textit{Abs } y) = y, \textit{erule subst}, \textit{erule subst}, \textit{assumption},$
$(\textit{erule type-definition.Abs-inverse}, \textit{assumption})+, \textit{simp})$

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

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

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

We need to show that

$\textit{Abs } (\textit{Actor-Abs } x) = \textit{Abs } (\textit{Actor-Abs } y) \longrightarrow \textit{Actor-Abs } x = \textit{Actor-Abs } y$, i.e. *injective Abs*.

Generally, *Abs* is not injective but *injective-on* the type predicate. So, we need to show that for any x, *Actor-Abs x* is in the type predicate, then it

would follow. This is the type predicate:
$\{y.\ y \in \mathit{Some}\ `\ \{x.\ x \notin \{''\mathit{Eve}'', ''\mathit{Charly}''\}\} \lor y = \mathit{None}\}$.

lemma *UasI-Actor-redef*:
$\bigwedge$ *Abs Rep*:: *actor* $\Rightarrow$ *char list option*.
$((\mathit{Abs} :: \mathit{identity\ option} \Rightarrow \mathit{actor}) o\ \mathit{Actor\text{-}Abs})\ ''\mathit{Eve}'' = ((\mathit{Abs} :: \mathit{identity\ option} \Rightarrow$
$\mathit{actor}) o\ \mathit{Actor\text{-}Abs})\ ''\mathit{Charly}'' \land$
$(\forall (x{::}\mathit{char\ list})\ y{::}\mathit{char\ list}.\ x \neq ''\mathit{Eve}'' \land y \neq ''\mathit{Eve}'' \land$
$((\mathit{Abs} :: \mathit{identity\ option} \Rightarrow \mathit{actor}) o\ \mathit{Actor\text{-}Abs})\ x = ((\mathit{Abs} :: \mathit{identity\ option} \Rightarrow$
$\mathit{actor}) o\ \mathit{Actor\text{-}Abs})\ y$
$\longrightarrow x = y)$
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] D. Gollmann. *Computer Security.* Wiley, 2008.

[2] 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.

[3] F. Kammüller and M. Kerber. Applying the isabelle insider framework to airplane security, 2020. arxive preprint 2003.11838.

[4] 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.

[5] J. R. C. Nurse, O. Buckley, P. A. Legg, M. Goldsmith, S. Creese, G. R. T. Wright, and M. Whitty. Understanding Insider Threat: A Framework for Characterising Attacks. In *IEEE Security and Privacy Workshops (SPW)*. IEEE, 2014.

[6] L. C. Paulson. Proving properties of security protocols by induction. In *CSFW*, pages 70–83. IEEE Computer Society, 1997.

[7] L. C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6(1-2):85–128, 1998.

[8] M. Wenzel. Re: [isabelle] typedecl versus explicit type parameters, 2009. Isabelle users mailing list.