Applying the Isabelle Insider Framework to Airplane Security

Florian Kammüller and Manfred Kerber

March 31, 2020

Abstract

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

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

Contents

1 Kripke structures and CTL

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

theory *MC*
imports *Main*
begin

1.1 Lemmas to support least and greatest fixpoints

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

lemma *monotoneE*: $monotone\ \tau \Longrightarrow p \subseteq q \Longrightarrow \tau\ p \subseteq \tau\ q$
by (*simp add*: *monotone-def*)

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

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

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

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

qed

lemma *ex-card*: *finite S* $\Longrightarrow \exists\ n{::}\ nat.\ card\ S = n$
by *simp*

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

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

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

set)
shows *False*

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

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

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

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

proof –
have *a*: $\forall\, i{::}nat.\ (\tau\ \hat{}\ i)\ (\{\}{::}\ 'a\ set) \subseteq (\tau\ \hat{}\ i + (1{::}nat))\ \{\}$
thm *predtrans-empty*
apply(*rule predtrans-empty*)
by (*rule assms*(*2*))

hence b: $(\exists\ i :: nat.\ \neg((\tau\ \hat{}\ i)\ \{\} \subset (\tau\ \hat{}(i + 1))\ \{\}))$ **using** *assms*
apply (*subgoal-tac* $\neg\ (\forall\ i :: nat.\ (\tau\ \hat{}\ i)\ \{\} \subset (\tau\ \hat{}(i + 1))\ \{\})$)
apply *blast*
apply (*rule notI*)
apply (*rule no-infinite-subset-chain*)
by *assumption*
thus $\exists\ i.\ (\tau\ \hat{}\ i)\ (\{\}) = (\tau\ \hat{}(i + 1))(\{\})$ **using** *a*
by *blast*
qed

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

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

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

set))
  apply (*rule psubset-card-mono*)
  apply (*rule finite-subset*)
  apply (*rule subset-UNIV*)
  apply (*rule assms*(*1*))
  by (*rule b*)
 thus *card* ((τ ^ *Suc n*) (*UNIV* :: *'a set*)) $\leq$ *card* (*UNIV* :: *'a set*) $-$ *Suc n*
using *a*
  by *simp*
 qed
qed

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

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

lemma *UNIV-tau-i-is-empty*:
 assumes *finite* (*UNIV* :: *'a set*) **and** *monotone* (τ :: *'a set* $\Rightarrow$ *'a set*)
 and ($\forall$ *i* :: *nat*. ((τ :: *'a set* $\Rightarrow$ *'a set*) ^ *i* + (*1* :: *nat*)) (*UNIV* :: *'a set*) $\subset$ (τ ^ *i*) *UNIV*)
 shows (τ ^ (*card* (*UNIV* ::*'a set*))) (*UNIV* ::*'a set*) = {}
proof $-$
 have *card* ((τ ^ *card* (*UNIV* ::*'a set*)) *UNIV*) $\leq$ (*0*::*nat*) **using** *assms*
 apply (*rule card-UNIV-tau-i-below-zero*)
 .
 thus (τ ^ (*card* (*UNIV* ::*'a set*))) (*UNIV* ::*'a set*) = {} **using** *assms*
 apply (*rule-tac S* = (τ ^ (*card* (*UNIV* ::*'a set*))) (*UNIV* ::*'a set*) **in** *finite-card-zero-empty*)
 apply (*rule finite-subset*)
 apply (*rule subset-UNIV*)
 .
qed

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

```
proof −
  have (τ ^ ((card (UNIV :: 'a set)))) UNIV = {} using assms
    apply (rule UNIV-tau-i-is-empty)
    .
  thus ∃ (j :: nat). (τ ^ j) UNIV = {}
    by (rule exI)
qed

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

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

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

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

lemma power-power: ((τ :: ('a set ⇒ 'a set)) ^^ n) = ((τ :: ('a set ⇒ 'a set)) ^
```

n)
proof (*induct-tac n*)
  show τ ^^ (*0*::*nat*) = (τ ^ *0*::*nat*) **by** (*simp add*: *id-def*)
next show $\bigwedge$*n*::*nat*. τ ^^ *n* = (τ ^ *n*) $\Longrightarrow$ τ ^^ *Suc n* = (τ ^ *Suc n*)
    by *simp*
qed

lemma *lfp-Kleene-iter-set*: *monotone* (*f* :: ('*a set* $\Rightarrow$ '*a set*)) $\Longrightarrow$
   (*f* ^ *Suc*(*n*)) {} = (*f* ^ *n*) {} $\Longrightarrow$ *lfp f* = (*f* ^ *n*){}
by (*simp add*: *monotone-mono lfp-Kleene-iter power-power*)

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

These next two are produced as duals from the corresponding theorems in HOL/ZF/Nat.thy. Would make sense to have them in the HOL/Library.

lemma *Kleene-iter-gpfp*:
assumes *mono f* **and** *p* $\leq$ *f p* **shows** *p* $\leq$ (*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* $\leq$ *gfp f*
  proof(*rule gfp-upperbound*)
    show (*f*^^*k*) *top* $\leq$ *f* ((*f*^^*k*) *top*) **using** *assms*(*2*) **by** *simp*
  qed
next

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

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

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

1.2 Generic type of state with state transition and CTL Operators

The system states and their transition relation are defined as a class called state containing an abstract constant state transition. It introduces the syntactic infix notation $I \rightarrow_i I'$ 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*] ⇒ *bool* ((- $\rightarrow_i$ -) *50*)

The above class definition lifts Kripke structures and CTL to a general level.

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

definition AX **where** $AX\,f \equiv \{s.\ \{f0.\ s \rightarrow_i f0\} \subseteq f\}$
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** $AF\,f \equiv lfp\ (\lambda\ Z.\ f \cup AX\ Z)$
definition EF **where** $EF\,f \equiv lfp\ (\lambda\ Z.\ f \cup EX'\ Z)$
definition AG **where** $AG\,f \equiv gfp\ (\lambda\ Z.\ f \cap AX\ Z)$
definition EG **where** $EG\,f \equiv gfp\ (\lambda\ Z.\ f \cap EX'\ Z)$
definition AU **where** $AU\,f1\ f2 \equiv lfp(\lambda\ Z.\ f2 \cup (f1 \cap AX\ Z))$
definition EU **where** $EU\,f1\ f2 \equiv lfp(\lambda\ Z.\ f2 \cup (f1 \cap EX'\ Z))$
definition AR **where** $AR\,f1\ f2 \equiv gfp(\lambda\ Z.\ f2 \cap (f1 \cup AX\ Z))$
definition ER **where** $ER\,f1\ f2 \equiv gfp(\lambda\ Z.\ f2 \cap (f1 \cup EX'\ Z))$

1.3 Kripke structure and Modelchecking

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

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

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

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

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

1.4 Lemmas for CTL operators

1.4.1 EF lemmas

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

$f \cup (EX' \ (lfp \ (\lambda Z :: 'a \ set. \ f \cup EX' \ Z)))$
apply (*rule def-lfp-unfold*)
apply (*rule reflexive*)
apply (*unfold mono-def EX'-def*)
by *auto*
thus $(x \in EF \ (f :: ('a :: state) \ set)) = (x \in f \cup EX' \ (lfp \ (\lambda Z :: ('a :: state) \ set. \ f \cup EX' \ Z)))$
by (*simp add*: *EF-def*)
qed

lemma *EF-lem00*: $(EF \ f) = (f \cup EX' \ (lfp \ (\lambda \ Z :: ('a :: state) \ set. \ f \cup EX' \ Z)))$
proof (*rule equalityI*)
show $EF \ f \subseteq f \cup EX' \ (lfp \ (\lambda Z::'a \ set. \ f \cup EX' \ Z))$
apply (*rule subsetI*)
by (*simp add*: *EF-lem0*)
next show $f \cup EX' \ (lfp \ (\lambda Z::'a \ set. \ f \cup EX' \ Z)) \subseteq EF \ f$
apply (*rule subsetI*)
by (*simp add*: *EF-lem0*)
qed

lemma *EF-lem000*: $(EF \ f) = (f \cup EX' \ (EF \ f))$
proof (*subst EF-lem00*)
show $f \cup EX' \ (lfp \ (\lambda Z::'a \ set. \ f \cup EX' \ Z)) = f \cup EX' \ (EF \ f)$
apply (*fold EF-def*)
by (*rule refl*)
qed

lemma *EF-lem1*: $x \in f \vee x \in (EX' \ (EF \ f)) \Longrightarrow x \in EF \ f$
proof (*simp add*: *EF-def*)
assume *a*: $x \in f \vee x \in EX' \ (lfp \ (\lambda Z::'a \ set. \ f \cup EX' \ Z))$
show $x \in lfp \ (\lambda Z::'a \ set. \ f \cup EX' \ Z)$
proof −
have *b*: $lfp \ (\lambda Z :: ('a :: state) \ set. \ f \cup EX' \ Z) =$
$f \cup (EX' \ (lfp \ (\lambda Z :: ('a :: state) \ set. \ f \cup EX' \ Z)))$
apply (*rule def-lfp-unfold*)
apply (*rule reflexive*)
apply (*unfold mono-def EX'-def*)
by *auto*
thus $x \in lfp \ (\lambda Z::'a \ set. \ f \cup EX' \ Z)$ **using** *a*
apply (*subst b*)
by *blast*
qed
qed

lemma *EF-lem2b*:
assumes $x \in (EX' \ (EF \ f))$
shows $x \in EF \ f$
proof (*rule EF-lem1*)
show $x \in f \vee x \in EX' \ (EF \ f)$

apply $(rule\ disjI2)$
by $(rule\ assms)$
qed

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

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

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

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

lemma $EX\text{-}lem0l$: $x \in (EX'\, f) \Longrightarrow x \in (EX'\,(f \cup g))$
proof $(unfold\ EX'\text{-}def)$
show $x \in \{s{::}'a.\ \exists f0{::}'a{\in}f.\ s \rightarrow_i f0\} \Longrightarrow x \in \{s{::}'a.\ \exists f0{::}'a{\in}f \cup g.\ s \rightarrow_i f0\}$
by $blast$
qed

lemma $EX\text{-}lem0r$: $x \in (EX'\ g) \Longrightarrow x \in (EX'\,(f \cup g))$
proof $(unfold\ EX'\text{-}def)$
show $x \in \{s{::}'a.\ \exists f0{::}'a{\in}g.\ s \rightarrow_i f0\} \Longrightarrow x \in \{s{::}'a.\ \exists f0{::}'a{\in}f \cup g.\ s \rightarrow_i f0\}$

by *blast*
qed

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

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

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

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

lemma *EF-step-star*: $[\![\ x \to_i{*}\ y;\ y \in f\ ]\!] \Longrightarrow x \in EF\ f$
proof (*simp add*: *state-transition-refl-def*)
  show $(x, y) \in \{(x::'a, y::'a).\ x \to_i y\}^* \Longrightarrow y \in f \Longrightarrow x \in EF\ f$
  proof (*erule converse-rtrancl-induct*)
    show $y \in f \Longrightarrow y \in EF\ f$
      by (*erule EF-lem2a*)

next show $\bigwedge(ya::'a)\ z::'a.\ y \in f \Longrightarrow$
$(ya,\ z) \in \{(x::'a,\ y::'a).\ x \rightarrow_i y\} \Longrightarrow$
$(z,\ y) \in \{(x::'a,\ y::'a).\ x \rightarrow_i y\}^* \Longrightarrow z \in \mathit{EF}\ f \Longrightarrow ya \in \mathit{EF}\ f$
apply (*clarify*)
apply (*erule EF-step-step*)
by *assumption*
qed
qed

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

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

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

qed

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

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

1.4.2 AG lemmas

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

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

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

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

lemma $AG\text{-}step$: $y \rightarrow_i z \Longrightarrow y \in AG\ s \Longrightarrow z \in AG\ s$
proof $(drule\ AG\text{-}lem2)$
  show $y \rightarrow_i z \Longrightarrow y \in s \cap AX\ (AG\ s) \Longrightarrow z \in AG\ s$
    apply $(erule\ IntE)$
    apply $(unfold\ AX\text{-}def)$
    apply $simp$
    apply $(erule\ subsetD)$
    by $simp$
qed

lemma $AG\text{-}all\text{-}s$: $x \rightarrow_i * \ y \Longrightarrow x \in AG\ s \Longrightarrow y \in AG\ s$
proof $(simp\ add{:}\ state\text{-}transition\text{-}refl\text{-}def)$

show $(x, y) \in \{(x::'a, y::'a).\ x \rightarrow_i y\}^* \Longrightarrow x \in AG\ s \Longrightarrow y \in AG\ s$
  apply (*erule rtrancl-induct*)
proof −
  show $x \in AG\ s \Longrightarrow x \in AG\ s$ **by** *assumption*
next show $\bigwedge(y::'a)\ z::'a.$
  $x \in AG\ s \Longrightarrow$
  $(x, y) \in \{(x::'a, y::'a).\ x \rightarrow_i y\}^* \Longrightarrow$
  $(y, z) \in \{(x::'a, y::'a).\ x \rightarrow_i y\} \Longrightarrow y \in AG\ s \Longrightarrow z \in AG\ s$
  apply *clarify*
  by (*erule AG-step*, *assumption*)
qed
qed

lemma *AG-imp-notnotEF*:
$I \neq \{\} \Longrightarrow ((Kripke\ \{s :: ('s :: state).\ \exists\ i \in I.\ (i \rightarrow_i\!*\ s)\}\ (I :: ('s :: state)set) \vdash AG\ s)) \Longrightarrow$
$(\neg(Kripke\ \{s :: ('s :: state).\ \exists\ i \in I.\ (i \rightarrow_i\!*\ s)\}\ (I :: ('s :: state)set)\ \vdash EF\ (-s)))$
proof (*rule notI*, *simp add*: *check-def*)
  assume *a0*: $I \neq \{\}$ **and**
    a1: $I \subseteq \{sa::'s.\ (\exists i::'s{\in}I.\ i \rightarrow_i\!*\ sa) \wedge sa \in AG\ s\}$ **and**
    a2: $I \subseteq \{sa::'s.\ (\exists i::'s{\in}I.\ i \rightarrow_i\!*\ sa) \wedge sa \in EF\ (-\ s)\}$
  show *False*
  proof −
    have *a3*: $\{sa::'s.\ (\exists i::'s{\in}I.\ i \rightarrow_i\!*\ sa) \wedge sa \in AG\ s\} \cap$
      $\{sa::'s.\ (\exists i::'s{\in}I.\ i \rightarrow_i\!*\ sa) \wedge sa \in EF\ (-\ s)\} = \{\}$
    proof −
      have $(?\ x.\ x \in \{sa::'s.\ (\exists i::'s{\in}I.\ i \rightarrow_i\!*\ sa) \wedge sa \in AG\ s\} \wedge$
        $x \in \{sa::'s.\ (\exists i::'s{\in}I.\ i \rightarrow_i\!*\ sa) \wedge sa \in EF\ (-\ s)\}) \Longrightarrow$ *False*
      proof −
        assume *a4*: $(?\ x.\ x \in \{sa::'s.\ (\exists i::'s{\in}I.\ i \rightarrow_i\!*\ sa) \wedge sa \in AG\ s\} \wedge$
          $x \in \{sa::'s.\ (\exists i::'s{\in}I.\ i \rightarrow_i\!*\ sa) \wedge sa \in EF\ (-\ s)\})$
        from *a4* **obtain** x **where** *a5*: $x \in \{sa::'s.\ (\exists i::'s{\in}I.\ i \rightarrow_i\!*\ sa) \wedge sa \in AG\ s\} \wedge$
          $x \in \{sa::'s.\ (\exists i::'s{\in}I.\ i \rightarrow_i\!*\ sa) \wedge sa \in EF\ (-\ s)\}$
        by (*erule exE*)
        hence $x \in s \wedge x \in -s$
        proof −
          have *a6*: $x \in s$ **using** *a5*
            apply (*subgoal-tac* $x \in AG\ s$)
            apply (*erule AG-in-lem*)
            by *simp*
          moreover have $x \in -s$ **using** *a5*
          proof −
            have $x \in EF\ s$
              apply (*rule-tac* $y = x$ **in** *EF-step-star*)
              apply (*simp add*: *state-transition-refl-def*)
              by (*rule a6*)

```
              thus x ∈ −s using a5
              proof −
                have x ∈ EF (− s) using a5
                  by simp
                moreover from this obtain y where a7: y ∈ − s ∧ x →ᵢ* y
                  apply (rotate-tac −1)
                   apply (drule EF-step-star-rev)
                  by blast
                moreover have y ∈ AG s using a7 a5
                  apply (subgoal-tac x ∈ AG s)
                  apply (erule conjE)
                   apply (drule AG-all-s)
                    apply assumption+
                  by simp
                ultimately show x ∈ −s using a5
                   apply (rotate-tac −1)
                   apply (drule AG-in-lem)
                   by blast
              qed
            qed
            ultimately show x ∈ s ∧ x ∈ −s
              by (rule conjI)
          qed
          thus False
            by blast
        qed
      thus {sa::'s. (∃ i::'s∈I. i →ᵢ* sa) ∧ sa ∈ AG s} ∩
                {sa::'s. (∃ i::'s∈I. i →ᵢ* sa) ∧ sa ∈ EF (− s)} = {}
      by blast
    qed
  moreover have b: ? x. x : I using a0
    by blast
  moreover obtain x where x ∈ I
      apply (rule exE)
       apply (rule b)
    by simp
  ultimately show False using a0 a1 a2
    by blast
  qed
qed

lemma check2-def: (Kripke S I ⊢ f) = (I ⊆ S ∩ f)
proof (simp add: check-def)
  show (I ⊆ {s::'a ∈ S. s ∈ f}) = (I ⊆ S ∧ I ⊆ f) by blast
qed

end
```

2 Insider Framework

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

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

typedecl *actor*
consts *Actor* :: *string* ⇒ *actor*

Alternatives to the type declaration do not work.

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

type-synonym *identity* = *string*
type-synonym *policy* = ((*actor* ⇒ *bool*) ∗ *action set*)

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

datatype *location* = *Location nat*

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

primrec *loc* :: *location* ⇒ *nat*
where *loc*(*Location n*) = *n*
primrec *gra* :: *igraph* ⇒ (*location* ∗ *location*)*set*
where *gra*(*Lgraph g a c l*) = *g*
primrec *agra* :: *igraph* ⇒ (*location* ⇒ *identity list*)
where *agra*(*Lgraph g a c l*) = *a*
primrec *cgra* :: *igraph* ⇒ (*actor* ⇒ *string list* ∗ *string list*)
where *cgra*(*Lgraph g a c l*) = *c*
primrec *lgra* :: *igraph* ⇒ (*location* ⇒ *string list*)
where *lgra*(*Lgraph g a c l*) = *l*

definition *nodes* :: *igraph* ⇒ *location set*

where *nodes g* == { *x*. (*? y*. ((*x*,*y*): *gra g*) | ((*y*,*x*): *gra g*))}

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

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

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

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

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

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

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

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

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

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

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

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

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

definition *atI* :: [*identity*, *igraph*, *location*] ⇒ *bool* (- $@_{(-)}$ - *50*)
where *a* $@_G$ *l* ≡ *a* ∈ *set*(*agra G l*)

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

definition *enables* :: [*infrastructure*, *location*, *actor*, *action*] ⇒ *bool*
where
enables I l a a' ≡ (∃ (*p*,*e*) ∈ *delta I* (*graphI I*) *l*. *a'* ∈ *e* ∧ *p a*)

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

definition *behaviour* :: *infrastructure* ⇒ (*location* ∗ *actor* ∗ *action*)*set*
where *behaviour I* ≡ {(*t*,*a*,*a'*). *enables I t a a'*}

misbehaviour is the complement of behaviour

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

basic lemmas for enable

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

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

lemma *not-enableE*: ⟦ ~(*enables I l a t*); (*p*,*e*) ∈ *delta I* (*graphI I*) *l* ⟧
⟹ (~(*t* : *e*) | (~(*p*(*a*))))
by (*simp add*: *enables-def*, *rule impI*, *force*)

lemma *not-enableE2*: ⟦ ~(*enables I l a t*); (*p*,*e*) ∈ *delta I* (*graphI I*) *l*;
t : *e* ⟧ ⟹ (~(*p*(*a*)))
by (*simp add*: *enables-def*, *force*)

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

primrec *del* :: $['a,\ 'a\ list] \Rightarrow 'a\ list$
where
del-nil: $del\ a\ []\ =\ []\ |$
del-cons: $del\ a\ (x\#ls) = (if\ x = a\ then\ ls\ else\ x\ \#\ (del\ a\ ls))$

primrec *jonce* :: $['a,\ 'a\ list] \Rightarrow bool$
where
jonce-nil: $jonce\ a\ [] = False\ |$
jonce-cons: $jonce\ a\ (x\#ls) = (if\ x = a\ then\ (a \notin (set\ ls))\ else\ jonce\ a\ ls)$

primrec *nodup* :: $['a,\ 'a\ list] \Rightarrow bool$
where
nodup-nil: $nodup\ a\ [] = True\ |$
nodup-step: $nodup\ a\ (x\ \#\ ls) = (if\ x = a\ then\ (a \notin (set\ ls))\ else\ nodup\ a\ ls)$

definition *move-graph-a* :: $[identity,\ location,\ location,\ igraph] \Rightarrow igraph$
where $move\text{-}graph\text{-}a\ n\ l\ l'\ g \equiv Lgraph\ (gra\ g)$
$(if\ n \in set\ ((agra\ g)\ l)\ \&\ \ n \notin set\ ((agra\ g)\ l')\ then$
$((agra\ g)(l := del\ n\ (agra\ g\ l)))(l' := (n\ \#\ (agra\ g\ l')))$
$else\ (agra\ g))(cgra\ g)(lgra\ g)$

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

inductive *state-transition-in* :: $[infrastructure,\ infrastructure] \Rightarrow bool\ ((\text{-} \rightarrow_n \text{-})\ 50)$
where
move: $[\![\ G = graphI\ I;\ a\ @_G\ l;\ l \in nodes\ G;\ l' \in nodes\ G;$
$(a) \in actors\text{-}graph(graphI\ I);\ enables\ I\ l'\ (Actor\ a)\ move;$
$I' = Infrastructure\ (move\text{-}graph\text{-}a\ a\ l\ l'\ (graphI\ I))(delta\ I)\ ]\!] \Longrightarrow I \rightarrow_n I'$
| *get* : $[\![\ G = graphI\ I;\ a\ @_G\ l;\ a'\ @_G\ l;\ has\ G\ (Actor\ a,\ z);$
$enables\ I\ l\ (Actor\ a)\ get;$
$I' = Infrastructure$
$(Lgraph\ (gra\ G)(agra\ G)$
$((cgra\ G)(Actor\ a' :=$
$(z\ \#\ (fst(cgra\ G\ (Actor\ a'))),\ snd(cgra\ G\ (Actor\ a')))))$
$(lgra\ G))$
$(delta\ I)$
$]\!] \Longrightarrow I \rightarrow_n I'$
| *put* : $[\![\ G = graphI\ I;\ a\ @_G\ l;\ enables\ I\ l\ (Actor\ a)\ put;$
$I' = Infrastructure$
$(Lgraph\ (gra\ G)(agra\ G)(cgra\ G)$
$((lgra\ G)(l := [z])))$
$(delta\ I)\ ]\!]$
$\Longrightarrow I \rightarrow_n I'$
| *put-remote* : $[\![\ G = graphI\ I;\ enables\ I\ l\ (Actor\ a)\ put;$
$I' = Infrastructure$
$(Lgraph\ (gra\ G)(agra\ G)(cgra\ G)$
$((lgra\ G)(l := [z])))$
$(delta\ I)\ ]\!]$

$\Longrightarrow I \rightarrow_n I'$

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

instantiation *infrastructure* :: *state*
begin

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

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*]: $a \in \mathit{set}\ (\mathit{del}\ \mathit{aa}\ l) \longrightarrow a \in \mathit{set}\ l$
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*]: $a \notin \mathit{set}\ \mathit{list} \longrightarrow a \notin \mathit{set}\ (\mathit{del}\ \mathit{aa}\ \mathit{list})$
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

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*]: $\forall$ z z'. $z \rightarrow_n z' \longrightarrow$ *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) $\in$ $\{(x::infrastructure,\ y::infrastructure).\ x \rightarrow_n y\}^*$
$\Longrightarrow$ *nodes*(*graphI* y) = *nodes*(*graphI* I)
by (*erule rtrancl-induct*, *rule refl*, *drule CollectD*, *simp*, *drule same-nodes0*, *simp*)

lemma *same-actors0* [*rule-format*]: $\forall$ z z'. $z \rightarrow_n z' \longrightarrow$ *actors-graph*(*graphI* z) = *actors-graph*(*graphI* z')
proof (*clarify*, *erule state-transition-in.cases*)
show $\bigwedge$(z::*infrastructure*) (z'::*infrastructure*) (G::*igraph*) (I::*infrastructure*) (a::*char list*)
(l::*location*) (a'::*char list*) (za::*char list*) I'::*infrastructure*.
$z = I \Longrightarrow$
$z' = I' \Longrightarrow$
G = *graphI* I $\Longrightarrow$
a @$_G$ l $\Longrightarrow$
a' @$_G$ l $\Longrightarrow$
has G (*Actor* a, za) $\Longrightarrow$
enables I l (*Actor* a) *get* $\Longrightarrow$
I' =
Infrastructure
(*Lgraph* (*gra* G) (*agra* G)
((*cgra* G)(*Actor* a' := (za # *fst* (*cgra* G (*Actor* a')), *snd* (*cgra* G (*Actor* a'))))) (*lgra* G))
(*delta* I) $\Longrightarrow$
actors-graph (*graphI* z) = *actors-graph* (*graphI* z')
by (*simp add*: *actors-graph-def nodes-def*)
next show $\bigwedge$(z::*infrastructure*) (z'::*infrastructure*) (G::*igraph*) (I::*infrastructure*) (a::*char list*)
(l::*location*) (I'::*infrastructure*) za::*char list*.
$z = I \Longrightarrow$
$z' = I' \Longrightarrow$
G = *graphI* I $\Longrightarrow$
a @$_G$ l $\Longrightarrow$
enables I l (*Actor* a) *put* $\Longrightarrow$
I' = *Infrastructure* (*Lgraph* (*gra* G) (*agra* G) (*cgra* G) ((*lgra* G)(l := [za]))) (*delta* I) $\Longrightarrow$
actors-graph (*graphI* z) = *actors-graph* (*graphI* z')
by (*simp add*: *actors-graph-def nodes-def*)
next show $\bigwedge$(z::*infrastructure*) (z'::*infrastructure*) (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*) $\Longrightarrow$
actors-graph (*graphI z*) = *actors-graph* (*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* = *graphI I* $\Longrightarrow$ *a* $@_G$ *l* $\Longrightarrow$
l ∈ *nodes G* $\Longrightarrow$ *l′* ∈ *nodes G* $\Longrightarrow$ *a* ∈ *actors-graph* (*graphI I*) $\Longrightarrow$
enables I l′ (*Actor a*) *move* $\Longrightarrow$
I′ = *Infrastructure* (*move-graph-a a l l′* (*graphI I*)) (*delta I*) $\Longrightarrow$
actors-graph (*graphI z*) = *actors-graph* (*graphI z′*)
proof (*rule equalityI*)
show *z* = *I* $\Longrightarrow$ *z′* = *I′* $\Longrightarrow$ *G* = *graphI I* $\Longrightarrow$ *a* $@_G$ *l* $\Longrightarrow$
l ∈ *nodes G* $\Longrightarrow$ *l′* ∈ *nodes G* $\Longrightarrow$ *a* ∈ *actors-graph* (*graphI I*) $\Longrightarrow$
enables I l′ (*Actor a*) *move* $\Longrightarrow$
I′ = *Infrastructure* (*move-graph-a a l l′* (*graphI I*)) (*delta I*) $\Longrightarrow$
actors-graph (*graphI z*) ⊆ *actors-graph* (*graphI z′*)
by (*rule subsetI*, *simp add*: *actors-graph-def* ,(*erule exE*)+, *case-tac x* = *a*,
rule-tac x = *l′* **in** *exI*, *simp add*: *move-graph-a-def nodes-def atI-def*,
rule-tac x = *ya* **in** *exI*, *rule conjI*, *simp add*: *move-graph-a-def nodes-def*
atI-def,
(*erule conjE*)+, *simp add*: *move-graph-a-def*, *rule conjI*, *clarify*,
simp add: *move-graph-a-def nodes-def atI-def*, *rule del-not-a*, *assumption*+,
clarify)
next show *z* = *I* $\Longrightarrow$ *z′* = *I′* $\Longrightarrow$ *G* = *graphI I* $\Longrightarrow$ *a* $@_G$ *l* $\Longrightarrow$
l ∈ *nodes G* $\Longrightarrow$ *l′* ∈ *nodes G* $\Longrightarrow$ *a* ∈ *actors-graph* (*graphI I*) $\Longrightarrow$
enables I l′ (*Actor a*) *move* $\Longrightarrow$
I′ = *Infrastructure* (*move-graph-a a l l′* (*graphI I*)) (*delta I*) $\Longrightarrow$
actors-graph (*graphI z′*) ⊆ *actors-graph* (*graphI z*)
by (*rule subsetI*, *simp add*: *actors-graph-def*, (*erule exE*)+,
case-tac x = *a*, *rule-tac x* = *l* **in** *exI*, *simp add*: *move-graph-a-def nodes-def*
atI-def,
rule-tac x = *ya* **in** *exI*, *rule conjI*, *simp add*: *move-graph-a-def nodes-def*
atI-def,
(*erule conjE*)+, *simp add*: *move-graph-a-def*, *case-tac ya* = *l*, *simp*,
case-tac a ∈ *set* (*agra* (*graphI I*) *l*) ∧ *a* ∉ *set* (*agra* (*graphI I*) *l′*), *simp*,
case-tac l = *l′*, *simp*+, *erule del-up*, *simp*,
case-tac a ∈ *set* (*agra* (*graphI I*) *l*) ∧ *a* ∉ *set* (*agra* (*graphI I*) *l′*), *simp*,
case-tac ya = *l′*, *simp*+)
qed
qed

lemma *same-actors*: (*I*, *y*) ∈ {(*x*::*infrastructure*, *y*::*infrastructure*). $x \to_n y$}*
$\Longrightarrow$ *actors-graph*(*graphI I*) = *actors-graph*(*graphI y*)
proof (*erule rtrancl-induct*)
show *actors-graph* (*graphI I*) = *actors-graph* (*graphI I*)
by (*rule refl*)
next show ⋀(*y*::*infrastructure*) *z*::*infrastructure*.
(*I*, *y*) ∈ {(*x*::*infrastructure*, *y*::*infrastructure*). $x \to_n y$}* $\Longrightarrow$

$(y, z) \in \{(x::infrastructure, y::infrastructure).\ x \rightarrow_n y\} \Longrightarrow$
$actors\text{-}graph\ (graphI\ I) = actors\text{-}graph\ (graphI\ y) \Longrightarrow$
$actors\text{-}graph\ (graphI\ I) = actors\text{-}graph\ (graphI\ z)$
by (*drule CollectD*, *simp*, *drule same-actors0*, *simp*)
qed

end
end

3 Airplane case study

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

locale *airplane* =

fixes *airplane-actors* :: *identity set*
defines *airplane-actors-def*: *airplane-actors* ≡ {''*Bob*'', ''*Charly*'', ''*Alice*''}

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

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

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

fixes *ex-creds* :: *actor* ⇒ (*string list* ∗ *string list*)
defines *ex-creds-def*: *ex-creds* ≡
(λ *x*.(*if x* = *Actor* ''*Bob*''
then ([''*PIN*''], [''*pilot*''])
else (*if x* = *Actor* ''*Charly*''
then ([''*PIN*''],[''*copilot*''])
else (*if x* = *Actor* ''*Alice*''
then ([''*PIN*''],[''*flightattendant*''])
else ([],[])))))

fixes *ex-locs* :: *location* ⇒ *string list*
defines *ex-locs-def*: *ex-locs* ≡ (λ *x*. *if x* = *door then* [''*norm*''] *else*

$(if\ x = cockpit\ then\ [''air''] \ else\ []))$

fixes *ex-locs'* :: *location* $\Rightarrow$ *string list*
defines *ex-locs'-def*: *ex-locs'* $\equiv$ $(\lambda\ x.\ if\ x = door\ then\ [''locked''] \ else$
$(if\ x = cockpit\ then\ [''air''] \ else\ []))$
fixes *ex-graph* :: *igraph*
defines *ex-graph-def*: *ex-graph* $\equiv$ *Lgraph*
$\{(cockpit,\ door),(door,cabin)\}$
$(\lambda\ x.\ if\ x = cockpit\ then\ [''Bob'',\ ''Charly'']$
$else\ (if\ x = door\ then\ []$
$else\ (if\ x = cabin\ then\ [''Alice''] \ else\ [])))$
ex-creds ex-locs

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

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

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

fixes *local-policies* :: [*igraph*, *location*] $\Rightarrow$ *policy set*
defines *local-policies-def*: *local-policies G* $\equiv$
$(\lambda\ y.\ if\ y = cockpit\ then$
$\{(\lambda\ x.\ (?\ n.\ (n\ @_G\ cockpit) \wedge Actor\ n = x),\ \{put\}),$
$(\lambda\ x.\ (?\ n.\ (n\ @_G\ cabin) \wedge Actor\ n = x \wedge has\ G\ (x,\ ''PIN'')$
$\wedge\ isin\ G\ door\ ''norm''),\{move\})$
$\}$
$else\ (if\ y = door\ then\ \{(\lambda\ x.\ True,\ \{move\}),$
$(\lambda\ x.\ (?\ n.\ (n\ @_G\ cockpit) \wedge Actor\ n = x),\ \{put\})\}$
$else\ (if\ y = cabin\ then\ \{(\lambda\ x.\ True,\ \{move\})\}$
$else\ \{\})))$

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

fixes *Airplane-scenario* :: *infrastructure* (**structure**)
defines *Airplane-scenario-def*:
Airplane-scenario $\equiv$ *Infrastructure ex-graph local-policies*

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

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

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

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

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

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

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

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

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

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

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

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

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

assumes *Eve-precipitating-event*: *tipping-point* (*astate* ''*Eve*'')
assumes *Insider-Eve*: *Insider* ''*Eve*'' {''*Charly*''} *astate*
assumes *cockpit-foe-control*: *foe-control cockpit put*

begin

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

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

lemma *ex-inv3*: $\neg$*global-policy Airplane-scenario* ''*Eve*''
proof (*simp add*: *Airplane-scenario-def global-policy-def*, *rule conjI*)
show ''*Eve*'' $\notin$ *airplane-actors* **by** (*simp add*: *airplane-actors-def*)
next show
enables (*Infrastructure ex-graph local-policies*) *cockpit* (*Actor* ''*Eve*'') *put*
proof –
have *a*: *Actor* ''*Charly*'' = *Actor* ''*Eve*''
by (*insert Insider-Eve*, *unfold Insider-def*, (*drule mp*),

```
        rule Eve-precipitating-event, simp add: UasI-def)
    show ?thesis
     by (insert a, simp add: Airplane-scenario-def enables-def ex-creds-def local-policies-def
ex-graph-def,
       insert Insider-Eve, unfold Insider-def, (drule mp), rule Eve-precipitating-event,

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

show Safety for Airplane_scenario

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

show Security for Airplane_scenario

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

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

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

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

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

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

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

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

lemma Security: Security Airplane-scenario s
```

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

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

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

show that pilot can get out of cockpit

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

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

lemma *ex-inv4*: ¬*global-policy Airplane-in-danger* (*''Eve''*)
proof (*simp add*: *Airplane-in-danger-def global-policy-def*, *rule conjI*)
show *''Eve''* ∉ *airplane-actors* **by** (*simp add*: *airplane-actors-def*)
next show *enables* (*Infrastructure aid-graph local-policies*) *cockpit* (*Actor ''Eve''*) *put*
proof −
have *a*: *Actor ''Charly''* = *Actor ''Eve''*
by (*insert Insider-Eve*, *unfold Insider-def*, (*drule mp*), *rule Eve-precipitating-event*, *simp add*: *UasI-def*)
show *?thesis*
apply (*insert a*, *erule subst*)
by (*simp add*: *enables-def local-policies-def cockpit-def aid-graph-def atI-def*)
qed
qed

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

lemma *Security-problem'*: ¬(*enables Airplane-in-danger cockpit* (*Actor ''Bob''*) *move*)
proof (*simp add*: *Airplane-in-danger-def Security-def enables-def local-policies-def*
ex-locs-def locl-lemma3a, *rule impI*)
assume *has aid-graph* (*Actor ''Bob''*, *''PIN''*)
show (∀ *n*::*char list*.

Actor n = Actor ''Bob'' ⟶ n $@_{\mathit{aid\text{-}graph}}$ *cabin ⟶ isin aid-graph door ''locked'')*
by (*simp add*: *aid-graph-def isin-def ex-locs'-def*)
qed

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

lemma *ex-inv5*: *a* ∈ *airplane-actors* ⟶ *global-policy Airplane-not-in-danger a*
by (*simp add*: *Airplane-not-in-danger-def global-policy-def*)

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

lemma *step0*: *Airplane-scenario* $\rightarrow_n$ *Airplane-getting-in-danger0*
proof (*rule-tac l* = *cockpit* **and** *l'* = *door* **and** *a* = *''Bob''* **in** *move*, *rule refl*)
  show *''Bob''* $@_{\mathit{graphI\ Airplane\text{-}scenario}}$ *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* $\rightarrow_n$ *Airplane-getting-in-danger*

proof (*rule-tac* l = *door* **and** l' = *cabin* **and** a = *''Bob''* **in** *move*, *rule refl*)
show *''Bob''* $@_{graphI\ Airplane\text{-}getting\text{-}in\text{-}danger0}$ *door*
by (*simp add*: *Airplane-getting-in-danger0-def atI-def aid-graph0-def door-def cockpit-def*)
next show *door* $\in$ *nodes* (*graphI Airplane-getting-in-danger0*)
by (*simp add*: *aid-graph0-def Airplane-getting-in-danger0-def nodes-def*, *blast*)+
next show *cabin* $\in$ *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''* $\in$ *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* $\rightarrow_n$ *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*)

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

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

lemma *actors-unique-loc-base*:
  assumes $I \rightarrow_n I'$
    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 $\Longrightarrow$
$I'a$ = *Infrastructure* (*Lgraph* (*gra G*) (*agra G*) (*cgra G*) ((*lgra G*)(l := [z]))) (*delta Ia*) $\Longrightarrow$
($\forall$ (l::*location*) l'::*location*. $a\ @_{graphI\ I'}\ l \wedge a\ @_{graphI\ I'}\ l' \longrightarrow l = l'$) $\wedge$
($\forall$ l::*location*. *nodup a* (*agra* (*graphI I′*) l))
by (*clarify*, *simp add*: *assms atI-def*)
next show $\bigwedge$(G::*igraph*) (Ia::*infrastructure*) (aa::*char list*) (l::*location*) (l'::*location*)
$I'a$::*infrastructure*.
$I = Ia \Longrightarrow$
$I' = I'a \Longrightarrow$
$G = graphI\ Ia \Longrightarrow$
$aa\ @_G\ l \Longrightarrow$
$l \in nodes\ G \Longrightarrow$
$l' \in nodes\ G \Longrightarrow$
$aa \in$ *actors-graph* (*graphI Ia*) $\Longrightarrow$
enables Ia l′ (Actor aa) move $\Longrightarrow$
$I'a$ = *Infrastructure* (*move-graph-a aa l l′* (*graphI Ia*)) (*delta Ia*) $\Longrightarrow$
($\forall$ (l::*location*) l'::*location*. $a\ @_{graphI\ I'}\ l \wedge a\ @_{graphI\ I'}\ l' \longrightarrow l = l'$) $\wedge$
($\forall$ l::*location*. *nodup a* (*agra* (*graphI I′*) l))
proof (*simp add*: *move-graph-a-def*,*rule conjI*, *clarify*, *rule conjI*, *clarify*, *rule conjI*, *clarify*)
show $\bigwedge$(G::*igraph*) (Ia::*infrastructure*) (aa::*char list*) (l::*location*) (l'::*location*)
($I'a$::*infrastructure*) (la::*location*) $l'a$::*location*.
$I' =$
Infrastructure
(*Lgraph* (*gra* (*graphI I*))
(*if* $a \in$ *set* (*agra* (*graphI I*) l) $\wedge$ $a \notin$ *set* (*agra* (*graphI I*) l')
then (*agra* (*graphI I*))(l := *del a* (*agra* (*graphI I*) l), l' := a # *agra* (*graphI I*) l')
else agra (*graphI I*))
(*cgra* (*graphI I*)) (*lgra* (*graphI I*)))
(*delta I*) $\Longrightarrow$
$a\ @_{graphI\ I}\ l \Longrightarrow$
$l \in nodes$ (*graphI I*) $\Longrightarrow$
$l' \in nodes$ (*graphI I*) $\Longrightarrow$

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

$la = l'a) \wedge$
$(\forall\, la{::}location.$
$(la = l \longrightarrow$
$(l = l' \longrightarrow nodup\ a\ (agra\ (graphI\ Ia)\ l')) \wedge$
$(l \neq l' \longrightarrow nodup\ a\ (del\ aa\ (agra\ (graphI\ Ia)\ l)))) \wedge$
$(la \neq l \longrightarrow$
$(la = l' \longrightarrow nodup\ a\ (agra\ (graphI\ Ia)\ l')) \wedge$
$(la \neq l' \longrightarrow nodup\ a\ (agra\ (graphI\ Ia)\ la))))) \wedge$
$((aa \in set\ (agra\ (graphI\ Ia)\ l) \longrightarrow aa \in set\ (agra\ (graphI\ Ia)\ l')) \longrightarrow$
$(\forall\, (l{::}location)\ l'{::}location.$
$a\ @_{Lgraph\ (gra\ (graphI\ Ia))\ (agra\ (graphI\ Ia))\ (cgra\ (graphI\ Ia))\ \ \ \ \ \ \ \ \ \ (lgra\ (graphI\ Ia))}$
$l \wedge$
$a\ @_{Lgraph\ (gra\ (graphI\ Ia))\ (agra\ (graphI\ Ia))\ (cgra\ (graphI\ Ia))\ \ \ \ \ \ \ \ \ \ (lgra\ (graphI\ Ia))}$
$l' \longrightarrow$
$l = l') \wedge$
$(\forall\, l{::}location.\ nodup\ a\ (agra\ (graphI\ Ia)\ l)))$

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 $\bigwedge(aa{::}char\ list)\ (l{::}location)\ (l'{::}location)\ l'a{::}location.$
$I' =$
$Infrastructure$
$(Lgraph\ (gra\ (graphI\ I))$
$((agra\ (graphI\ I))(l := del\ aa\ (agra\ (graphI\ I)\ l),\ l' := aa\ \#\ agra\ (graphI$
$I)\ l'))$
$(cgra\ (graphI\ I))\ (lgra\ (graphI\ I)))$
$(delta\ I) \Longrightarrow$
$aa \in set\ (agra\ (graphI\ I)\ l) \Longrightarrow$
$l \in nodes\ (graphI\ I) \Longrightarrow$
$l' \in nodes\ (graphI\ I) \Longrightarrow$
$aa \in actors\text{-}graph\ (graphI\ I) \Longrightarrow$
$enables\ I\ l'\ (Actor\ aa)\ move \Longrightarrow$
$aa \neq a \Longrightarrow$
$aa \notin set\ (agra\ (graphI\ I)\ l') \Longrightarrow$
$l \neq l' \Longrightarrow$
$l'a \neq l \Longrightarrow$
$l'a = l' \Longrightarrow a \in set\ (del\ aa\ (agra\ (graphI\ I)\ l)) \Longrightarrow a \notin set\ (agra\ (graphI$
$I)\ l')$

by (*meson assms(2) atI-def del-notin-down*)

next show $\bigwedge(aa{::}char\ list)\ (l{::}location)\ (l'{::}location)\ l'a{::}location.$
$I' =$
$Infrastructure$
$(Lgraph\ (gra\ (graphI\ I))$
$((agra\ (graphI\ I))(l := del\ aa\ (agra\ (graphI\ I)\ l),\ l' := aa\ \#\ agra\ (graphI$
$I)\ l'))$
$(cgra\ (graphI\ I))\ (lgra\ (graphI\ I)))$
$(delta\ I) \Longrightarrow$
$aa \in set\ (agra\ (graphI\ I)\ l) \Longrightarrow$
$l \in nodes\ (graphI\ I) \Longrightarrow$
$l' \in nodes\ (graphI\ I) \Longrightarrow$

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

next show $\bigwedge(aa::char\ list)\ (l::location)\ l'::location.$
$I' =$
$Infrastructure$
$(Lgraph\ (gra\ (graphI\ I))$
$(if\ aa \notin set\ (agra\ (graphI\ I)\ l')$
$then\ (agra\ (graphI\ I))(l := del\ aa\ (agra\ (graphI\ I)\ l),\ l' := aa\ \#\ agra\ (graphI\ I)\ l')$
$else\ agra\ (graphI\ I))$
$(cgra\ (graphI\ I))\ (lgra\ (graphI\ I)))$
$(delta\ I) \Longrightarrow$
$aa \in set\ (agra\ (graphI\ I)\ l) \Longrightarrow$
$l \in nodes\ (graphI\ I) \Longrightarrow$
$l' \in nodes\ (graphI\ I) \Longrightarrow$
$aa \in actors\text{-}graph\ (graphI\ I) \Longrightarrow$
$enables\ I\ l'\ (Actor\ aa)\ move \Longrightarrow$
$aa \neq a \Longrightarrow$
$aa \notin set\ (agra\ (graphI\ I)\ l') \Longrightarrow$
$\forall\ la::location.$
$(la = l \longrightarrow$
$(l = l' \longrightarrow nodup\ a\ (agra\ (graphI\ I)\ l')) \wedge$
$(l \neq l' \longrightarrow nodup\ a\ (del\ aa\ (agra\ (graphI\ I)\ l)))) \wedge$
$(la \neq l \longrightarrow$
$(la = l' \longrightarrow nodup\ a\ (agra\ (graphI\ I)\ l')) \wedge (la \neq l' \longrightarrow nodup\ a\ (agra\ (graphI\ I)\ la)))$
by $(simp\ add{:}\ assms(2)\ nodup\text{-}down\text{-}notin)$
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-unique-loc-step*:
  assumes $(I, I') \in \{(x{::}\mathit{infrastructure}, y{::}\mathit{infrastructure}).\ x \to_n y\}^*$
    and $\forall\ a.\ (\forall\ l\ l'.\ a\ @_{\mathit{graphI}\ I}\ l \wedge a\ @_{\mathit{graphI}\ I}\ l' \longrightarrow l = l') \wedge$
      $(\forall\ l.\ \mathit{nodup}\ a\ (\mathit{agra}\ (\mathit{graphI}\ I)\ l))$
   shows $\forall\ a.\ (\forall\ l\ l'.\ a\ @_{\mathit{graphI}\ I'}\ l \wedge a\ @_{\mathit{graphI}\ I'}\ l' \longrightarrow l = l') \wedge$
      $(\forall\ l.\ \mathit{nodup}\ a\ (\mathit{agra}\ (\mathit{graphI}\ I')\ l))$
proof −
  have *ind*: $(\forall\ a.\ (\forall\ l\ l'.\ a\ @_{\mathit{graphI}\ I}\ l \wedge a\ @_{\mathit{graphI}\ I}\ l' \longrightarrow l = l') \wedge$
      $(\forall\ l.\ \mathit{nodup}\ a\ (\mathit{agra}\ (\mathit{graphI}\ I)\ l))) \longrightarrow$
    $(\forall\ a.\ (\forall\ l\ l'.\ a\ @_{\mathit{graphI}\ I'}\ l \wedge a\ @_{\mathit{graphI}\ I'}\ l' \longrightarrow l = l') \wedge$
      $(\forall\ l.\ \mathit{nodup}\ a\ (\mathit{agra}\ (\mathit{graphI}\ I')\ l)))$
  proof (*insert assms*(*1*), *erule rtrancl.induct*)
   show $\bigwedge a{::}\mathit{infrastructure}.$
    $(\forall\, aa{::}\mathit{char\ list}.$
      $(\forall\,(l{::}\mathit{location})\ l'{::}\mathit{location}.\ aa\ @_{\mathit{graphI}\ a}\ l \wedge aa\ @_{\mathit{graphI}\ a}\ l' \longrightarrow l = l') \wedge$
      $(\forall\, l{::}\mathit{location}.\ \mathit{nodup}\ aa\ (\mathit{agra}\ (\mathit{graphI}\ a)\ l))) \longrightarrow$
    $(\forall\, aa{::}\mathit{char\ list}.$
      $(\forall\,(l{::}\mathit{location})\ l'{::}\mathit{location}.\ aa\ @_{\mathit{graphI}\ a}\ l \wedge aa\ @_{\mathit{graphI}\ a}\ l' \longrightarrow l = l') \wedge$
      $(\forall\, l{::}\mathit{location}.\ \mathit{nodup}\ aa\ (\mathit{agra}\ (\mathit{graphI}\ a)\ l)))$ **by** *simp*
  next show $\bigwedge(a{::}\mathit{infrastructure})\ (b{::}\mathit{infrastructure})\ (c{::}\mathit{infrastructure}).$
    $(a, b) \in \{(x{::}\mathit{infrastructure}, y{::}\mathit{infrastructure}).\ x \to_n y\}^* \Longrightarrow$
    $(\forall\, aa{::}\mathit{char\ list}.$
       $(\forall\,(l{::}\mathit{location})\ (l'{::}\mathit{location}).\ (aa\ @_{\mathit{graphI}\ a}\ l \wedge aa\ @_{\mathit{graphI}\ a}\ l') \longrightarrow l =$
$l') \wedge$
      $(\forall\, l{::}\mathit{location}.\ \mathit{nodup}\ aa\ (\mathit{agra}\ (\mathit{graphI}\ a)\ l))) \longrightarrow$
    $(\forall\, a{::}\mathit{char\ list}.$
      $(\forall\,(l{::}\mathit{location})\ (l'{::}\mathit{location}).\ (a\ @_{\mathit{graphI}\ b}\ l \wedge a\ @_{\mathit{graphI}\ b}\ l') \longrightarrow l = l') \wedge$
      $(\forall\, l{::}\mathit{location}.\ \mathit{nodup}\ a\ (\mathit{agra}\ (\mathit{graphI}\ b)\ l))) \Longrightarrow$
    $(b, c) \in \{(x{::}\mathit{infrastructure}, y{::}\mathit{infrastructure}).\ x \to_n y\} \Longrightarrow$
    $(\forall\, aa{::}\mathit{char\ list}.$
      $(\forall\,(l{::}\mathit{location})\ l'{::}\mathit{location}.\ (aa\ @_{\mathit{graphI}\ a}\ l \wedge aa\ @_{\mathit{graphI}\ a}\ l') \longrightarrow l = l')$
$\wedge$
      $(\forall\, l{::}\mathit{location}.\ \mathit{nodup}\ aa\ (\mathit{agra}\ (\mathit{graphI}\ a)\ l))) \longrightarrow$
    $(\forall\, a{::}\mathit{char\ list}.$
      $(\forall\,(l{::}\mathit{location})\ l'{::}\mathit{location}.\ (a\ @_{\mathit{graphI}\ c}\ l \wedge a\ @_{\mathit{graphI}\ c}\ l') \longrightarrow l = l') \wedge$
      $(\forall\, l{::}\mathit{location}.\ \mathit{nodup}\ a\ (\mathit{agra}\ (\mathit{graphI}\ c)\ l)))$
   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*:
 $\forall\ a.\ (\forall\ l\ l'.\ a\ @_{\mathit{graphI}\ \mathit{Airplane\text{-}not\text{-}in\text{-}danger\text{-}init}}\ l \wedge$
      $a\ @_{\mathit{graphI}\ \mathit{Airplane\text{-}not\text{-}in\text{-}danger\text{-}init}}\ l' \longrightarrow l = l') \wedge$

(∀ *l*. *nodup a* (*agra* (*graphI Airplane-not-in-danger-init*) *l*))
proof (*simp add*: *Airplane-not-in-danger-init-def ex-graph-def*, *clarify*, *rule conjI*, *clarify*,
rule conjI, *clarify*, *rule impI*, (*rule allI*)+, *rule impI*, *simp add*: *atI-def*)
show $\bigwedge$(*l*::*location*) *l*′::*location*.
′′*Charly*′′
∈ *set* (*if l* = *cockpit then* [′′*Bob*′′, ′′*Charly*′′]
else if l = *door then* [] *else if l* = *cabin then* [′′*Alice*′′] *else* []) ∧
′′*Charly*′′
∈ *set* (*if l*′ = *cockpit then* [′′*Bob*′′, ′′*Charly*′′]
else if l′ = *door then* [] *else if l*′ = *cabin then* [′′*Alice*′′] *else* []) ⟹
l = *l*′
by (*case-tac l* = *l*′, *assumption*, *rule FalseE*, *case-tac l* = *cockpit* ∨ *l* = *door* ∨ *l* = *cabin*,
erule disjE, *simp*, *case-tac l*′ = *door* ∨ *l*′ = *cabin*, *erule disjE*, *simp*,
simp add: *cabin-def door-def*, *simp*, *erule disjE*, *simp add*: *door-def cockpit-def*,

simp add: *cabin-def door-def cockpit-def*, *simp*)
next show $\bigwedge$*a*::*char list*.
′′*Charly*′′ ≠ *a* ⟶
(∀ (*l*::*location*) *l*′::*location*.
a $@_{Lgraph}$ {(*cockpit*, *door*), (*door*, *cabin*)} (λ*x*::*location*. *if x* = *cockpit then* [′′*Bob*
l ∧
a $@_{Lgraph}$ {(*cockpit*, *door*), (*door*, *cabin*)} (λ*x*::*location*. *if x* = *cockpit then* [′′*Bob*
l′ ⟶
l = *l*′)
by (*clarify*, *simp add*: *atI-def*, *case-tac l* = *l*′, *assumption*, *rule FalseE*,
case-tac l = *cockpit* ∨ *l* = *door* ∨ *l* = *cabin*, *erule disjE*, *simp*,
case-tac l′ = *door* ∨ *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-not-in-danger-init*, *I*)∈ {(*x*::*infrastructure*, *y*::*infrastructure*). $x \to_n y$}*
⟹ ∀ *a*. (∀ *l l*′. *a* $@_{graphI\ I}$ *l* ∧ *a* $@_{graphI\ I}$ *l*′ ⟶ *l* = *l*′)∧
(∀ *l*. *nodup a* (*agra* (*graphI I*) *l*))
by (*erule actors-unique-loc-step*, *rule actors-unique-loc-aid-base*)

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

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

airplane-actors-def, *rule equalityI*)
show $\{x{::}char\ list.$
$\exists\, y{::}location.$
$(y = door \longrightarrow$
$(door = cockpit \longrightarrow$
$(\exists\, y{::}location.\ y = cockpit \vee y = cabin \vee y = cockpit \vee y = cockpit \wedge cockpit = cabin) \wedge$
$(x = ''Bob'' \vee x = ''Charly'')) \wedge$
$door = cockpit) \wedge$
$(y \neq door \longrightarrow$
$(y = cockpit \longrightarrow$
$(\exists\, y{::}location.$
$y = door \vee$
$cockpit = door \wedge y = cabin \vee$
$y = cockpit \wedge cockpit = door \vee y = door \wedge cockpit = cabin) \wedge$
$(x = ''Bob'' \vee x = ''Charly'')) \wedge$
$(y \neq cockpit \longrightarrow y = cabin \wedge x = ''Alice'' \wedge y = cabin))\}$
$\subseteq \{''Bob'', ''Charly'', ''Alice''\}$
by (*rule subsetI*, *drule CollectD*, *erule exE*, (*erule conjE*)+,
simp add: *door-def cockpit-def cabin-def*, (*erule conjE*)+, *force*)
next show $\{''Bob'', ''Charly'', ''Alice''\}$
$\subseteq \{x{::}char\ list.$
$\exists\, y{::}location.$
$(y = door \longrightarrow$
$(door = cockpit \longrightarrow$
$(\exists\, y{::}location.$
$y = cockpit \vee y = cabin \vee y = cockpit \vee y = cockpit \wedge cockpit = cabin) \wedge$
$(x = ''Bob'' \vee x = ''Charly'')) \wedge$
$door = cockpit) \wedge$
$(y \neq door \longrightarrow$
$(y = cockpit \longrightarrow$
$(\exists\, y{::}location.$
$y = door \vee$
$cockpit = door \wedge y = cabin \vee$
$y = cockpit \wedge cockpit = door \vee y = door \wedge cockpit = cabin) \wedge$
$(x = ''Bob'' \vee x = ''Charly'')) \wedge$
$(y \neq cockpit \longrightarrow y = cabin \wedge x = ''Alice'' \wedge y = cabin))\}$
by (*rule subsetI*, *rule CollectI*, *simp add*: *door-def cockpit-def cabin-def*,
case-tac $x = ''Bob''$, *force*, *case-tac* $x = ''Charly''$, *force*,
subgoal-tac $x = ''Alice''$, *force*, *simp*)
qed

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

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

$\Longrightarrow a \in \textit{actors-graph}\ (\textit{graphI}\ I)$
by (*simp add*: *atI-def actors-graph-def*, *rule exI*, *rule conjI*)

lemma *not-en-get-Apnid*:
assumes $(\textit{Airplane-not-in-danger-init},y) \in \{(x::\textit{infrastructure}, y::\textit{infrastructure}).\ x \rightarrow_n y\}^*$
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*: $''Bob''\ @_{\textit{graphI Airplane-not-in-danger-init}}\ \textit{cockpit}$
by (*simp add*: *Airplane-not-in-danger-init-def ex-graph-def atI-def*)

Invariant: number of staff in cockpit never below 2

lemma *two-person-inv*:
fixes $z\ z'$
assumes $(2::\textit{nat}) \leq \textit{length}\ (\textit{agra}\ (\textit{graphI}\ z)\ \textit{cockpit})$
and $\textit{nodes}(\textit{graphI}\ z) = \textit{nodes}(\textit{graphI Airplane-not-in-danger-init})$
and $\textit{delta}(z) = \textit{delta}(\textit{Airplane-not-in-danger-init})$
and $(\textit{Airplane-not-in-danger-init},z) \in \{(x::\textit{infrastructure}, y::\textit{infrastructure}).\ x \rightarrow_n y\}^*$
and $z \rightarrow_n z'$
shows $(2::\textit{nat}) \leq \textit{length}\ (\textit{agra}\ (\textit{graphI}\ z')\ \textit{cockpit})$
proof (*insert assms*(*5*), *erule state-transition-in.cases*)
show $\bigwedge(G::\textit{igraph})\ (I::\textit{infrastructure})\ (a::\textit{char list})\ (l::\textit{location})\ (a'::\textit{char list})\ (za::\textit{char list})$
$I'::\textit{infrastructure}.$
$z = I \Longrightarrow$
$z' = I' \Longrightarrow$
$G = \textit{graphI}\ I \Longrightarrow$
$a\ @_G\ l \Longrightarrow$
$a'\ @_G\ l \Longrightarrow$

$has\ G\ (Actor\ a,\ za) \Longrightarrow$
$enables\ I\ l\ (Actor\ a)\ get \Longrightarrow$
$I' =$
$Infrastructure$
$(Lgraph\ (gra\ G)\ (agra\ G)$
$((cgra\ G)(Actor\ a' := (za\ \#\ fst\ (cgra\ G\ (Actor\ a')),\ snd\ (cgra\ G\ (Actor\ a'))))) \ (lgra\ G))$
$(delta\ I) \Longrightarrow$
$(2::nat) \leq length\ (agra\ (graphI\ z')\ cockpit)$ **using** *assms* **by** *simp*

next show $\bigwedge(G::igraph)\ (I::infrastructure)\ (a::char\ list)\ (l::location)\ (I'::infrastructure)$
$za::char\ list.$
$z = I \Longrightarrow$
$z' = I' \Longrightarrow$
$G = graphI\ I \Longrightarrow$
$a\ @_G\ l \Longrightarrow$
$enables\ I\ l\ (Actor\ a)\ put \Longrightarrow$
$I' = Infrastructure\ (Lgraph\ (gra\ G)\ (agra\ G)\ (cgra\ G)\ ((lgra\ G)(l := [za])))\ (delta\ I) \Longrightarrow$
$(2::nat) \leq length\ (agra\ (graphI\ z')\ cockpit)$ **using** *assms* **by** *simp*

next show $\bigwedge(G::igraph)\ (I::infrastructure)\ (l::location)\ (a::char\ list)\ (I'::infrastructure)$
$za::char\ list.$
$z = I \Longrightarrow$
$z' = I' \Longrightarrow$
$G = graphI\ I \Longrightarrow$
$enables\ I\ l\ (Actor\ a)\ put \Longrightarrow$
$I' = Infrastructure\ (Lgraph\ (gra\ G)\ (agra\ G)\ (cgra\ G)\ ((lgra\ G)(l := [za])))\ (delta\ I) \Longrightarrow$
$(2::nat) \leq length\ (agra\ (graphI\ z')\ cockpit)$ **using** *assms* **by** *simp*

next show $\bigwedge(G::igraph)\ (I::infrastructure)\ (a::char\ list)\ (l::location)\ (l'::location)$
$I'::infrastructure.$
$z = I \Longrightarrow$
$z' = I' \Longrightarrow$
$G = graphI\ I \Longrightarrow$
$a\ @_G\ l \Longrightarrow$
$l \in nodes\ G \Longrightarrow$
$l' \in nodes\ G \Longrightarrow$
$a \in actors\text{-}graph\ (graphI\ I) \Longrightarrow$
$enables\ I\ l'\ (Actor\ a)\ move \Longrightarrow$
$I' = Infrastructure\ (move\text{-}graph\text{-}a\ a\ l\ l'\ (graphI\ I))\ (delta\ I) \Longrightarrow$
$(2::nat) \leq length\ (agra\ (graphI\ z')\ cockpit)$

proof −
fix G :: *igraph* **and** I :: *infrastructure* **and** a :: *char list* **and** l :: *location* **and** l' :: *location* **and** I' :: *infrastructure*
have *f1*: *UasI ''Eve'' ''Charly''*
using *Eve-precipitating-event Insider-Eve Insider-def* **by** *force*
obtain *ccs* :: *char list* ⇒ *char list* **and** *ccsa* :: *char list* ⇒ *char list* **where**
f2: $\forall\ cs\ csa.\ (\neg\ UasI\ cs\ csa \vee Actor\ cs = Actor\ csa \wedge (\forall\ csa\ csb.\ (csa = cs \vee csb = cs \vee Actor\ csa \neq Actor\ csb) \vee csa = csb)) \wedge (UasI\ cs\ csa \vee Actor\ cs \neq Actor\ csa \vee (ccs\ cs \neq cs \wedge ccsa\ cs \neq cs \wedge Actor\ (ccs\ cs) = Actor\ (ccsa\ cs)) \wedge$

ccs cs $\neq$ *ccsa cs*)
  using *UasI-def* **by** *moura*
 have *''Bob''* $@_{graphI}$ (*Infrastructure ex-graph local-policies*) *Location 2*
  using *Airplane-not-in-danger-init-def cockpit-def test-graph-atI* **by** *force*
 then have *Actor ''Bob''* = *Actor ''Eve''*
  using *Airplane-scenario-def airplane.cockpit-foe-control airplane-axioms cockpit-def ex-inv3 global-policy-def* **by** *blast*
 then show *2* $\leq$ *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*). $x \to_n y\}^*$
 shows (*2*::*nat*) $\leq$ *length* (*agra* (*graphI z*) *cockpit*)
proof (*insert assms*, *erule rtrancl-induct*)
 show (*2*::*nat*) $\leq$ *length* (*agra* (*graphI Airplane-not-in-danger-init*) *cockpit*)
 by (*simp add*: *Airplane-not-in-danger-init-def ex-graph-def*)
next show $\bigwedge$(*y*::*infrastructure*) *z*::*infrastructure*.
  (*Airplane-not-in-danger-init*, *y*) $\in$ {(*x*::*infrastructure*, *y*::*infrastructure*). $x \to_n y\}^* \Longrightarrow$
  (*y*, *z*) $\in$ {(*x*::*infrastructure*, *y*::*infrastructure*). $x \to_n y\} \Longrightarrow$
  (*2*::*nat*) $\leq$ *length* (*agra* (*graphI y*) *cockpit*) $\Longrightarrow$ (*2*::*nat*) $\leq$ *length* (*agra* (*graphI z*) *cockpit*)
  by (*rule two-person-inv*, *assumption*, *rule same-nodes*, *assumption*, *rule sym*, *rule init-state-policy*, *assumption+*, *simp*)
qed

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

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

lemma *nodup-card-insert*:
  $a \notin$ *set l* $\longrightarrow$ *card* (*insert a* (*set l*)) = *Suc* (*card* (*set l*))
by *auto*

lemma *no-dup-set-list-num-eq*[*rule-format*]:
 ($\forall$ *a*. *nodup a l*) $\longrightarrow$ *card* (*set l*) = *length l*
 by (*induct-tac l*, *simp*, *clarify*, *simp*, *erule impE*, *rule allI*,
  drule-tac x = *aa* **in** *spec*, *case-tac a* = *aa*, *simp*, *erule nodup-notin*, *simp*)

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

have *a*: *card* (*set* (*agra* (*graphI z*) *cockpit*)) = *length*(*agra* (*graphI z*) *cockpit*)
by (*rule no-dup-set-list-num-eq*, *insert assms*, *drule actors-unique-loc-aid-step*,
drule-tac x = *a* **in** *spec*, *erule conjE*, *erule-tac x* = *cockpit* **in** *spec*)
show *?thesis*
by (*insert a*, *erule ssubst*, *rule two-person-inv1*, *rule assms*)
qed

lemma *Pred-all-unique*: $[\![$ *? x. P x*; (! *x. P x* $\longrightarrow$ *x* = *c*)$]\!] \Longrightarrow$ *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*: $[\![$ $S \neq \{\}$; ($\forall$ $x \in S$. $x = c$) $]\!] \Longrightarrow c \in S$
by (*rule-tac P* = λ *x*. $x \in S$ **in** *Pred-all-unique*, *force*, *simp*)

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

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

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

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

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

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

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

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*: $l \in$ *nodes* (*graphI I*)
and *a7*: *cockpit* $\in$ *nodes* (*graphI I*)
and *a8*: $a \in$ *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*: $a \in$ *set* (*agra* (*graphI I*) *l*) $\wedge$ $a \notin$ *set* (*agra* (*graphI I*) *cockpit*)
and *a12*: $l' =$ *cockpit*
and *a13*: $h = a$
show $h \in$ *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 $(Airplane\text{-}not\text{-}in\text{-}danger\text{-}init,z) \in \{(x::infrastructure,\ y::infrastructure).\ x \to_n y\}^*$
shows $\forall\, h::char\ list \in set\ (agra\ (graphI\ z)\ cockpit).\ h \in airplane\text{-}actors$
proof −
have *ind*: $(Airplane\text{-}not\text{-}in\text{-}danger\text{-}init,\ z) \in \{(x::infrastructure,\ y::infrastructure).\ x \to_n y\}^* \longrightarrow$
$(\forall\, h::char\ list \in set\ (agra\ (graphI\ z)\ cockpit).\ h \in airplane\text{-}actors)$
proof (*insert assms*, *erule rtrancl-induct*)
show $(Airplane\text{-}not\text{-}in\text{-}danger\text{-}init,\ Airplane\text{-}not\text{-}in\text{-}danger\text{-}init) \in \{(x,y).\ x \to_n y\}^* \longrightarrow$
$(\forall\, h::char\ list \in set\ (agra\ (graphI\ Airplane\text{-}not\text{-}in\text{-}danger\text{-}init)\ cockpit).\ h \in airplane\text{-}actors)$
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 $\bigwedge(y::infrastructure)\ z::infrastructure.$
$(Airplane\text{-}not\text{-}in\text{-}danger\text{-}init,\ y) \in \{(x::infrastructure,\ y::infrastructure).\ x \to_n y\}^* \Longrightarrow$
$(y,\ z) \in \{(x::infrastructure,\ y::infrastructure).\ x \to_n y\} \Longrightarrow$
$(Airplane\text{-}not\text{-}in\text{-}danger\text{-}init,\ y) \in \{(x,y).\ x \to_n y\}^* \longrightarrow$
$(\forall\, h::char\ list \in set\ (agra\ (graphI\ y)\ cockpit).\ h \in airplane\text{-}actors) \Longrightarrow$
$(Airplane\text{-}not\text{-}in\text{-}danger\text{-}init,\ z) \in \{(x,y).\ x \to_n y\}^* \longrightarrow$
$(\forall\, h::char\ list \in set\ (agra\ (graphI\ z)\ cockpit).\ h \in airplane\text{-}actors)$
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*: $(Airplane\text{-}not\text{-}in\text{-}danger\text{-}init,\ I)$
$\in \{(x::infrastructure,\ y::infrastructure).\ x \to_n y\}^* \Longrightarrow$
$x \in set\ (agra\ (graphI\ I)\ cockpit) \Longrightarrow x \neq {''}Eve{''}$
by (*drule airplane-actors-inv*, *simp add*: *airplane-actors-def*,
drule-tac $x = x$ **in** *bspec*, *assumption*, *force*)

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

lemma *tp-imp-control*:
assumes $(Airplane\text{-}not\text{-}in\text{-}danger\text{-}init,I) \in \{(x::infrastructure,\ y::infrastructure).$

$x \rightarrow_n y\}^*$
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*). $x \rightarrow_n y\}^* \Longrightarrow$
  ¬ *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*). $x \to_n y$}*
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*). $x \to_n y$}*
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*).
$x \to_n y$}*
by (*erule conjE*, *simp add*: *Air-tp-Kripke-def Air-tp-states-def state-transition-in-refl-def*)+
qed
show *?thesis*
by (*insert a0 a1 b*, *rule-tac P* = ''*Eve*'' $@_{graphI\ xa}$ *cockpit* **in** *notI*,
simp add: *atI-def*, *drule Eve-not-in-cockpit*, *assumption*, *simp*)
qed
qed
qed

qed
qed

end

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

definition *airplane-actors-def′*: *airplane-actors* $\equiv$ {*″Bob″*, *″Charly″*, *″Alice″*}
definition *airplane-locations-def′*:
airplane-locations $\equiv$ {*Location 0*, *Location 1*, *Location 2*}
definition *cockpit-def′*: *cockpit* $\equiv$ *Location 2*
definition *door-def′*: *door* $\equiv$ *Location 1*
definition *cabin-def′*: *cabin* $\equiv$ *Location 0*
definition *global-policy-def′*: *global-policy I a* $\equiv$ *a* $\notin$ *airplane-actors*
$\longrightarrow \neg$(*enables I cockpit* (*Actor a*) *put*)
definition *ex-creds-def′*: *ex-creds* $\equiv$
(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* $\equiv$ (λ *x*. *if x* = *door then* [*″norm″*] *else*
(*if x* = *cockpit then* [*″air″*] *else* []))

definition *ex-locs′-def′*: *ex-locs′* $\equiv$ (λ *x*. *if x* = *door then* [*″locked″*] *else*
(*if x* = *cockpit then* [*″air″*] *else* []))

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

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

definition *aid-graph0-def′*: *aid-graph0* $\equiv$ *Lgraph*
{(*cockpit*, *door*),(*door*,*cabin*)}
(λ *x*. *if x* = *cockpit then* [*″Charly″*]
else (*if x* = *door then* [*″Bob″*]
else (*if x* = *cabin then* [*″Alice″*] *else* [])))

ex-creds ex-locs

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

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

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

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

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

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

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

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

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

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

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

definition *Airplane-not-in-danger-init-def'*:
Airplane-not-in-danger-init $\equiv$ *Infrastructure ex-graph local-policies-four-eyes*

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

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

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

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

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

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

print-interps *airplane*

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

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

(The following addresses the issue of redefining an abstract type. We experimented with suggestion given here: Makarius Wenzel, Re: [isabelle] typedecl versus explicit type parameters, Isabelle users mailing list, 2009, https://lists.cam.ac.uk/pipermail/cl-isabelle-users/2009-July/msg00111.html.) We furthermore need axiomatization to add the missing semantics to the abstractly declared type actor and thereby be able to redefine consts Actor. Since the function Actor has also been defined as a consts :: identity =¿ actor as an abstract function without a definition, we now also now add its semantics mimicking some of the concepts of the conservative type definition of HOL. The alternative method of using a Locale to replace the abstract type_decl actor in the AirInsider is a more elegant method for representing and abstract type actor but it is not working properly for our framwework since it necessitates introducing a type parameter 'actor into infrastructures which then makes it impossible to instantiate them to the typeclass state in order to use CTL and Kripke and the generic state transition. Therefore, we go the former way of a post-hoc axiomatic redefinition of the abstract type actor by using axiomatization of the existing Locale "type_definition". This is done in the following. It allows to abstractedly assume as an axiom that there is a type definition for the abstract type actor. Adding a suitable definition of a representation for this type then additionally enables to introduce a definition for the function Actor (again using axiomatization to enforce the new definition).

definition *Actor-Abs* :: *identity* ⇒ *identity option*
  where
Actor-Abs x ≡ (*if x* ∈ {*''Eve''*, *''Charly''*} *then None else Some x*)

lemma *UasI-ActorAbs*: *Actor-Abs ''Eve''* = *Actor-Abs ''Charly''* ∧
  (∀ (*x*::*char list*) *y*::*char list*. *x* ≠ *''Eve''* ∧ *y* ≠ *''Eve''* ∧ *Actor-Abs x* = *Actor-Abs y* ⟶ *x* = *y*)
  by (*simp add*: *Actor-Abs-def*)

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

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

axiomatization where *Actor-type-def*:
type-definition (*Rep* :: *actor* ⇒ *identity option*)(*Abs* :: *identity option* ⇒ *actor*) {*y* :: *identity option*. *y* ∈ *Some* ‘ {*x* :: *identity*. *x* ∉ {*''Eve''*, *''Charly''*}}| *y* = *None*}

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

lemma *Actor-td-Abs-inverse*:
(*y*∈ {*y* :: *identity option*. *y* ∈ *Some* ‘ {*x* :: *identity*. *x* ∉ {*''Eve''*, *''Charly''*}}| *y* = *None*}) ⟹
(*Rep* :: *actor* ⇒ *identity option*)((*Abs* :: *identity option* ⇒ *actor*) *y*) = *y*

by (*insert Actor-type-def*, *drule-tac x = Rep* **in** *meta-spec*, *drule-tac x = Abs* **in** *meta-spec*,
erule type-definition.Abs-inverse, *assumption*)

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

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

need to show that *Abs* (*Actor-Abs x*) = *Abs* (*Actor-Abs y*) ⟶ *Actor-Abs x* = *Actor-Abs y*, i.e. *injective Abs*. Generally, Abs is not injective but *injective-on* the type predicate. So, need to show that for any x, *Actor-Abs x* is in the type predicate, then it would follow. What is the type predicate? {*y*. *y* ∈ *Some* ‘ {*x*. *x* ∉ {''*Eve*'', ''*Charly*''}} ∨ *y* = *None*}

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

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

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

interpretation *airplane airplane-actors airplane-locations cockpit door cabin global-policy*
ex-creds ex-locs ex-locs' ex-graph aid-graph aid-graph0 agid-graph
local-policies local-policies-four-eyes Airplane-scenario Airplane-in-danger
Airplane-getting-in-danger0 Airplane-getting-in-danger Air-states
Air-Kripke
Airplane-not-in-danger Airplane-not-in-danger-init Air-tp-states
Air-tp-Kripke Safety Security foe-control astate
by (*rule airplane.intro*, *simp add*: *tipping-point-def*,
simp add: *Insider-def UasI-def tipping-point-def atI-def*,
insert UasI-Actor, *simp add*: *UasI-def*,
insert cockpit-foe-control', *simp add*: *foe-control-def' cockpit-def'*,
rule airplane-actors-def',
(*simp add*: *airplane-locations-def' cockpit-def' door-def' cabin-def' global-policy-def'*
ex-creds-def' ex-locs-def' ex-locs'-def' ex-graph-def' aid-graph-def'
aid-graph0-def'
agid-graph-def' local-policies-def' local-policies-four-eyes-def' Airplane-scenario-def'

Airplane-in-danger-def ' Airplane-getting-in-danger0-def ' Airplane-getting-in-danger-def '
Air-states-def ' Air-Kripke-def ' Airplane-not-in-danger-def ' Airplane-not-in-danger-init-def '
Air-tp-states-def ' Air-tp-Kripke-def ' Safety-def ' Security-def '
foe-control-def ' astate-def ')+)

end

References

[1] F. Kammüller and M. Kerber. Investigating airplane safety and security against insider threats using logical modeling. In *IEEE Security and Privacy Workshops, Workshop on Research in Insider Threats, WRIT'16*. IEEE, 2016.

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

[3] F. Kammüller and C. W. Probst. Modeling and verification of insider threats using logical analysis. *IEEE Systems Journal, Special issue on Insider Threats to Information Security, Digital Espionage, and Counter Intelligence*, 11(2):534–545, 2017.

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