

# Applying the Isabelle Insider Framework to Airplane Security

Florian Kammüller and Manfred Kerber

March 31, 2020

## Abstract

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

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

## Contents

<b>1</b>	<b>Kripke structures and CTL</b>	<b>2</b>
1.1	Lemmas to support least and greatest fixpoints . . . . .	2
1.2	Generic type of state with state transition and CTL Operators	9
1.3	Kripke structure and Modelchecking . . . . .	10
1.4	Lemmas for CTL operators . . . . .	10
1.4.1	EF lemmas . . . . .	10
1.4.2	AG lemmas . . . . .	15

## 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  $\tau \implies p \subseteq q \implies \tau p \subseteq \tau q$ 
by (simp add: monotone-def)
```

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

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

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

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

qed

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

**lemma** *less-not-le*:  $\llbracket (x :: \text{nat}) < y; y \leq x \rrbracket \implies \text{False}$   
**by** *arith*

**lemma** *infchain-outruns-all*:  
**assumes** *finite* (*UNIV* :: 'a set)  
**and**  $\forall i :: \text{nat. } (\tau \hat{\ } i) \{\} :: \text{'a set} \subset (\tau \hat{\ } i + (1 :: \text{nat})) \{\}$   
**shows**  $\forall j :: \text{nat. } \exists i :: \text{nat. } j < \text{card } ((\tau \hat{\ } i) \{\})$   
**proof** (*rule allI, induct-tac j*)  
**show**  $\exists i :: \text{nat. } (0 :: \text{nat}) < \text{card } ((\tau \hat{\ } i) \{\})$  **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**  $\bigwedge (j :: \text{nat}) n :: \text{nat. } \exists i :: \text{nat. } n < \text{card } ((\tau \hat{\ } i) \{\})$   
 $\implies \exists i :: \text{nat. } \text{Suc } n < \text{card } ((\tau \hat{\ } i) \{\})$   
**proof** –  
**fix** *j n*  
**assume** *a*:  $\exists i :: \text{nat. } n < \text{card } ((\tau \hat{\ } i) \{\})$   
**obtain** *i* **where**  $n < \text{card } ((\tau \hat{\ } (i :: \text{nat})) \{\})$   
**apply** (*rule exE*)  
**apply** (*rule a*)  
**by** *simp*  
**thus**  $\exists i. \text{Suc } n < \text{card } ((\tau \hat{\ } i) \{\})$  **using** *assms*  
**apply** (*rule-tac x = i + 1 in exI*)  
**apply** (*subgoal-tac card ((\tau \hat{\ } i) \{\}) < card ((\tau \hat{\ } i + (1 :: \text{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* ( $\tau :: \text{'a set} \Rightarrow \text{'a set}$ )  
**and**  $\forall i :: \text{nat. } ((\tau :: \text{'a set} \Rightarrow \text{'a set}) \hat{\ } i) \{\} \subset (\tau \hat{\ } i + (1 :: \text{nat})) (\{\} :: \text{'a$

```

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  $\exists i. card\ UNIV < card\ ((\tau \wedge i)\ \{\})$ . Since all sets are subsets of  $UNIV$ , we also have  $card\ ((\tau \wedge i)\ \{\}) \leq card\ UNIV$ : Contradiction!, i.e. proof of False

**proof** –

```

have a:  $\forall (j :: nat). (\exists (i :: nat). (j :: nat) < card((\tau \wedge i)(\{\} :: 'a\ set)))$  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:  $\exists i :: nat. card\ UNIV < card\ ((\tau \wedge i)\ \{\})$  using a
  apply (drule-tac  $x = card\ UNIV$  in spec)
  by assumption
from this obtain i where e:  $card\ (UNIV :: 'a\ set) < card\ ((\tau \wedge i)\ \{\})$ 
  by (erule exE)
hence f:  $(card((\tau \wedge i)(\{\}))) \leq (card\ (UNIV :: 'a\ set))$  using assms
  thm Finite-Set.card-mono
  apply (rule-tac  $A = ((\tau \wedge i)(\{\}))$  in Finite-Set.card-mono)
  apply assumption
  by (rule subset-UNIV)
thus False using e
  thm less-not-le
  apply (erule-tac  $y = card((\tau \wedge i)(\{\}))$  in less-not-le)
  by assumption
qed

```

**lemma** *finite-fxp*:

```

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

```

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

**proof** –

```

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

```

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

**lemma** *predtrans-UNIV*:

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

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

**lemma** *card-univ-subtract*:

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

```

set))
  apply (rule psubset-card-mono)
  apply (rule finite-subset)
  apply (rule subset-UNIV)
  apply (rule assms(1))
  by (rule b)
  thus card (( $\tau \wedge \text{Suc } n$ ) (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 ( $\tau :: 'a \text{ set} \Rightarrow 'a \text{ set}$ )
  and ( $\forall i :: \text{nat}. ((\tau :: 'a \text{ set} \Rightarrow 'a \text{ set}) \wedge i + (1 :: \text{nat})) (UNIV :: 'a \text{ set}) \subset (\tau \wedge i) \text{ UNIV}$ )
  shows card(( $\tau \wedge (\text{card (UNIV :: 'a set)})$ ) (UNIV :: 'a set))  $\leq 0$ 
proof -
  have ( $\forall i :: \text{nat}. \text{card}((\tau \wedge i) \text{ UNIV :: 'a set})) \leq (\text{card (UNIV :: 'a set)}) - i$ )
using assms
  by (rule card-univ-subtract)
  thus card(( $\tau \wedge (\text{card (UNIV :: 'a set)})$ ) (UNIV :: 'a set))  $\leq 0$ 
  apply (drule-tac  $x = \text{card (UNIV :: 'a set)}$  in spec)
  by simp
qed

```

```

lemma finite-card-zero-empty:  $\llbracket \text{finite } S; \text{card } S \leq 0 \rrbracket \Rightarrow S = \{\}$ 
by simp

```

```

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

```

```

proof –
  have  $(\tau \wedge ((\text{card } (UNIV :: 'a \text{ set})))) UNIV = \{\}$  using assms
  apply (rule UNIV-tau-i-is-empty)
  .
  thus  $\exists (j :: \text{nat}). (\tau \wedge j) UNIV = \{\}$ 
    by (rule exI)
qed

lemma no-infinite-subset-chain2:
  assumes finite  $(UNIV :: 'a \text{ set})$  and monotone  $(\tau :: ('a \text{ set} \Rightarrow 'a \text{ set}))$ 
    and  $\forall i :: \text{nat}. (\tau \wedge i) UNIV \supset (\tau \wedge i + (1 :: \text{nat})) UNIV$ 
  shows False
proof –
  have  $\exists j :: \text{nat}. (\tau \wedge j) UNIV = \{\}$  using assms
  apply (rule down-chain-reaches-empty)
  .
  from this obtain j where a:  $(\tau \wedge j) UNIV = \{\}$  by (erule exE)
  have  $(\tau \wedge j + (1 :: \text{nat})) UNIV \subset (\tau \wedge j) UNIV$  using assms
    by (erule-tac x = j in spec)
  thus False using a by simp
qed

lemma finite-fix2:
  assumes finite  $(UNIV :: 'a \text{ set})$  and monotone  $(\tau :: ('a \text{ set} \Rightarrow 'a \text{ set}))$ 
  shows  $\exists i. (\tau \wedge i) UNIV = (\tau \wedge (i + 1)) UNIV$ 
proof –
  have  $\forall i :: \text{nat}. (\tau \wedge i + (1 :: \text{nat})) UNIV \subseteq (\tau \wedge i) UNIV$ 
  apply (rule predtrans-UNIV) using assms
  by (simp add: assms(2))
  moreover have  $\exists i :: \text{nat}. \neg (\tau \wedge i + (1 :: \text{nat})) UNIV \subset (\tau \wedge i) UNIV$  using
assms
proof –
  have  $\neg (\forall i :: \text{nat}. (\tau \wedge i) UNIV \supset (\tau \wedge (i + 1)) UNIV)$ 
  apply (rule notI)
  apply (rule no-infinite-subset-chain2) using assms
  .
  thus  $\exists i :: \text{nat}. \neg (\tau \wedge i + (1 :: \text{nat})) UNIV \subset (\tau \wedge i) UNIV$  by blast
qed
  ultimately show  $\exists i. (\tau \wedge i) UNIV = (\tau \wedge (i + 1)) UNIV$ 
    by blast
qed

lemma mono-monotone: mono  $(\tau :: ('a \text{ set} \Rightarrow 'a \text{ set})) \Longrightarrow \text{monotone } \tau$ 
by (simp add: monotone-def mono-def)

lemma monotone-mono: monotone  $(\tau :: ('a \text{ set} \Rightarrow 'a \text{ set})) \Longrightarrow \text{mono } \tau$ 
by (simp add: monotone-def mono-def)

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

```

$n$ )  
**proof** (*induct-tac*  $n$ )  
 show  $\tau \hat{\cdot} (0::nat) = (\tau \hat{\cdot} 0::nat)$  **by** (*simp add: id-def*)  
**next show**  $\bigwedge n::nat. \tau \hat{\cdot} n = (\tau \hat{\cdot} n) \implies \tau \hat{\cdot} Suc\ n = (\tau \hat{\cdot} Suc\ n)$   
 by *simp*  
**qed**

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

**lemma** *lfp-loop*:  
 assumes *finite* ( $UNIV :: 'b\ set$ ) **and** *monotone* ( $\tau :: ('b\ set \Rightarrow 'b\ set)$ )  
 shows  $\exists n. lfp\ \tau = (\tau \hat{\cdot} n)\ \{\}$   
**proof** –  
 have  $\exists i::nat. (\tau \hat{\cdot} i)\ \{\} = (\tau \hat{\cdot} i + (1::nat))\ \{\}$  **using** *assms*  
 by (*rule finite-fixp*)  
**from this obtain**  $i$  **where**  $(\tau \hat{\cdot} i)\ \{\} = (\tau \hat{\cdot} i + (1::nat))\ \{\}$   
 by (*erule exE*)  
**hence**  $(\tau \hat{\cdot} i)\ \{\} = (\tau \hat{\cdot} Suc\ i)\ \{\}$   
 by *simp*  
**hence**  $(\tau \hat{\cdot} Suc\ i)\ \{\} = (\tau \hat{\cdot} i)\ \{\}$   
 by (*rule sym*)  
**hence**  $lfp\ \tau = (\tau \hat{\cdot} i)\ \{\}$   
 by (*simp add: assms(2) lfp-Kleene-iter-set*)  
**thus**  $\exists n. lfp\ \tau = (\tau \hat{\cdot} n)\ \{\}$   
 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-gfp*:  
 assumes *mono*  $f$  **and**  $p \leq f\ p$  **shows**  $p \leq (f \hat{\cdot} k)\ (top::'a::order-top)$   
**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**  $(f \hat{\cdot} Suc\ k)\ top = (f \hat{\cdot} k)\ top$   
**shows**  $gfp\ f = (f \hat{\cdot} k)\ top$   
**proof**(*rule antisym*)  
 show  $(f \hat{\cdot} k)\ top \leq gfp\ f$   
**proof**(*rule gfp-upperbound*)  
 show  $(f \hat{\cdot} k)\ top \leq f\ ((f \hat{\cdot} k)\ top)$  **using** *assms*(2) **by** *simp*  
**qed**  
**next**



```

show  $\text{gfp } f \leq (f^{\wedge k}) \text{ top}$ 
  using Kleene-iter-gppf[OF assms(1)] gfp-unfold[OF assms(1)] by simp
qed

```

```

lemma gfp-Kleene-iter-set:
  assumes monotone ( $f :: ('a \text{ set} \Rightarrow 'a \text{ set})$ )
    and  $(f \wedge \text{Suc}(n)) \text{ UNIV} = (f \wedge n) \text{ UNIV}$ 
    shows  $\text{gfp } f = (f \wedge n) \text{ UNIV}$ 
proof -
  have  $a :: \text{mono } f$  using assms
    by (erule-tac  $\tau = f$  in monotone-mono)
  hence  $b :: (f \wedge \text{Suc } (n)) \text{ UNIV} = (f \wedge n) \text{ UNIV}$  using assms
    by (simp add: power-power)
  hence  $c :: \text{gfp } f = (f \wedge (n))(\text{UNIV} :: 'a \text{ set})$  using assms a
    thm gfp-Kleene-iter
    apply (erule-tac  $f = f$  and  $k = n$  in gfp-Kleene-iter)
  .
  thus  $\text{gfp } f = (f \wedge (n))(\text{UNIV} :: 'a \text{ set})$  using assms a
    by (simp add: power-power)
qed

```

```

lemma gfp-loop:
  assumes finite ( $\text{UNIV} :: 'b \text{ set}$ )
    and monotone ( $\tau :: ('b \text{ set} \Rightarrow 'b \text{ set})$ )
    shows  $\exists n . \text{gfp } \tau = (\tau \wedge n)(\text{UNIV} :: 'b \text{ set})$ 
proof -
  have  $\exists i :: \text{nat}. (\tau \wedge i)(\text{UNIV} :: 'b \text{ set}) = (\tau \wedge i + (1 :: \text{nat})) \text{ UNIV}$  using assms
    by (rule finite-fixp2)
  from this obtain  $i$  where  $(\tau \wedge i)(\text{UNIV} :: 'b \text{ set}) = (\tau \wedge i + (1 :: \text{nat})) \text{ UNIV}$ 
by (erule exE)
  thus  $\exists n . \text{gfp } \tau = (\tau \wedge n)(\text{UNIV} :: 'b \text{ set})$  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 :: [ $'a :: \text{type}, 'a$ ]  $\Rightarrow \text{bool}$   $((- \rightarrow_i -) 50)$ 

```

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

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

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

**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-transition-refl*  $((- \rightarrow_i^* -)\ 50)$

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

### 1.4 Lemmas for CTL operators

#### 1.4.1 EF lemmas

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

```

```

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

```

```

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

```

```

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

```

```

lemma EF-lem2b:
  assumes x ∈ (EX' (EF f))
  shows x ∈ EF f
proof (rule EF-lem1)
  show x ∈ f ∨ x ∈ EX' (EF f)

```

```

    apply (rule disjI2)
    by (rule assms)
qed

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

by *blast*  
qed

**lemma** *EX-step*: assumes  $x \rightarrow_i y$  and  $y \in f$  shows  $x \in EX' f$   
**proof** (*unfold EX'-def*)  
 show  $x \in \{s :: 'a. \exists f0 :: 'a \in f. s \rightarrow_i f0\}$   
 apply *simp*  
 apply (*rule-tac*  $x = y$  in *beexI*)  
 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  $a: \bigwedge f :: 'a \text{ set}. EF (f :: ('a :: state) set) = f \cup EX' (EF f)$   
 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  $x \rightarrow_i y$  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  $x \rightarrow_i y$  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*:  $\llbracket x \rightarrow_i^* y; y \in f \rrbracket \Longrightarrow x \in EF f$   
**proof** (*simp add: state-transition-refl-def*)  
 show  $(x, y) \in \{(x :: 'a, y :: 'a). x \rightarrow_i y\}^* \Longrightarrow y \in f \Longrightarrow x \in EF f$   
**proof** (*erule converse-rtrancl-induct*)  
 show  $y \in f \Longrightarrow y \in EF f$   
 by (*erule EF-lem2a*)

```

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

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

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

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

```

qed

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

**proof** (erule *EF-induct*)

show *mono* ( $\lambda Z::'a\ set. s \cup EX'\ Z$ )

apply (*simp add: mono-def EX'-def*)

by *force*

**next show**  $\bigwedge x::'a. x \in s \cup EX'\ (EF\ s \cap \{x::'a. \exists y::'a \in s. x \rightarrow_i^* y\}) \implies \exists y::'a \in s. x \rightarrow_i^* y$

**apply** (erule *UnE*)

apply (*rule-tac x = x in bexI*)

apply (*simp add: state-transition-refl-def*)

apply *assumption*

apply (*simp add: EX'-def*)

apply (erule *bexE*)

apply (erule *IntE*)

apply (*drule CollectD*)

apply (erule *bexE*)

apply (*rule-tac x = xb in bexI*)

apply (*simp add: state-transition-refl-def*)

apply (*rule rtrancl-trans*)

apply (*rule r-into-rtrancl*)

apply (*rule CollectI*)

apply *simp*

by *assumption+*

qed

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

**proof** (*clarify*)

show  $\bigwedge x::'s. I \subseteq \{sa::'s. (\exists i::'s \in I. i \rightarrow_i^* sa) \wedge sa \in EF\ s\} \implies x \in I \implies$

$\exists y::'s \in s. x \rightarrow_i^* y$

apply (*drule subsetD*)

apply *assumption*

apply (*drule CollectD*)

apply (erule *conjE*)

by (erule *EF-step-star-rev*)

qed

### 1.4.2 AG lemmas

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

**proof** (*simp add: AG-def gfp-def*)

show  $\exists xa \subseteq s. xa \subseteq AX\ xa \wedge x \in xa \implies x \in s$

apply (erule *exE*)

apply (erule *conjE*)

by (erule *subsetD, assumption*)

qed

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

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

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

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

**lemma** *AG-all-s*:  $x \rightarrow_i^* y \implies x \in AG s \implies y \in AG s$   
**proof** (*simp add: state-transition-refl-def*)



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

**lemma** *AG-imp-notnotEF*:  
 $I \neq \{\} \implies ((Kripke\ \{s :: ('s :: state). \exists i \in I. (i \rightarrow_i^* s)\} (I :: ('s :: state) set)$   
 $\vdash AG\ s)) \implies$   
 $(\neg(Kripke\ \{s :: ('s :: state). \exists i \in I. (i \rightarrow_i^* s)\} (I :: ('s :: state) set) \vdash EF\ (-$   
 $s)))$   
**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**  $(? x. x \in \{sa::'s. (\exists i::'s \in I. i \rightarrow_i^* sa) \wedge sa \in AG\ s\} \wedge$   
 $x \in \{sa::'s. (\exists i::'s \in I. i \rightarrow_i^* sa) \wedge sa \in EF\ (-\ s)\}) \implies$   
*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  $x \in -s$  using  $a5$ 
proof –
  have  $x \in EF (-s)$  using  $a5$ 
  by simp
  moreover from this obtain  $y$  where  $a7: y \in -s \wedge x \rightarrow_i^* y$ 
  apply (rotate-tac -1)
  apply (drule EF-step-star-rev)
  by blast
  moreover have  $y \in AG s$  using  $a7 a5$ 
  apply (subgoal-tac x ∈ AG s)
  apply (erule conjE)
  apply (drule AG-all-s)
  apply assumption+
  by simp
  ultimately show  $x \in -s$  using  $a5$ 
  apply (rotate-tac -1)
  apply (drule AG-in-lem)
  by blast
qed
qed
ultimately show  $x \in s \wedge x \in -s$ 
by (rule conjI)
qed
thus False
by blast
qed
thus  $\{sa::'s. (\exists i::'s \in I. i \rightarrow_i^* sa) \wedge sa \in AG s\} \cap$ 
 $\{sa::'s. (\exists i::'s \in I. i \rightarrow_i^* sa) \wedge sa \in EF (-s)\} = \{\}$ 
by blast
qed
moreover have  $b: ?x. x : I$  using  $a0$ 
by blast
moreover obtain  $x$  where  $x \in I$ 
apply (rule exE)
apply (rule b)
by simp
ultimately show False using  $a0 a1 a2$ 
by blast
qed
qed

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

end

```

## 2 Insider Framework

```

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

```

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

```

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

```

Alternatives to the type declaration do not work.

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

```

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

```

```

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

```

```

datatype location = Location nat

```

```

datatype igraph = Lgraph (location * location)set location  $\Rightarrow$  identity list
               actor  $\Rightarrow$  (string list * string list) location  $\Rightarrow$  string list

```

```

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

```

```

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

```

```

definition nodes :: igraph  $\Rightarrow$  location set

```

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

**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**  $psy-states = happy \mid depressed \mid disgruntled \mid angry \mid stressed$

**datatype**  $motivations = financial \mid political \mid revenge \mid curious \mid competitive-advantage \mid power \mid peer-recognition$

**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**  $UasI' :: [actor \Rightarrow bool, identity, identity] \Rightarrow bool$

**where**  $UasI'\ P\ a\ b \equiv P\ (Actor\ b) \longrightarrow P\ (Actor\ a)$

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

**definition** *Insider* :: [identity, identity set, identity  $\Rightarrow$  actor-state]  $\Rightarrow$  bool  
**where** *Insider* a C as  $\equiv$  (tipping-point (as a)  $\longrightarrow$  ( $\forall$  b  $\in$  C. UasI a b))

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

**definition** *atI* :: [identity, igrph, location]  $\Rightarrow$  bool (- @(-) - 50)  
**where** a @<sub>G</sub> l  $\equiv$  a  $\in$  set(agra G l)

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

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

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

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

misbehaviour is the complement of behaviour

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

basic lemmas for enable

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

**lemma** *not-enableI2*: [ $\bigwedge$  p e. (p,e)  $\in$  delta I (graphI I) l  $\implies$   
( $\sim$ (t : e) | ( $\sim$ (p(a))))]  $\implies \sim$ (*enables* I l a t)  
**by** (rule not-enableI, rule ballI, auto)

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

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

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

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

**where**

$del\text{-}nil: del\ a\ [] = []$

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

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

**where**

$jonce\text{-}nil: jonce\ a\ [] = False$

$jonce\text{-}cons: jonce\ a\ (x\#\!ls) = (if\ x = a\ then\ (a\ \notin\ (set\ ls))\ else\ jonce\ a\ ls)$

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

**where**

$nodup\text{-}nil: nodup\ a\ [] = True$

$nodup\text{-}step: nodup\ a\ (x\ \#\!ls) = (if\ x = a\ then\ (a\ \notin\ (set\ ls))\ else\ nodup\ a\ ls)$

**definition**  $move\text{-}graph\text{-}a :: [identity, location, location, igraph] \Rightarrow igraph$

**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**  $state\text{-}transition\text{-}in :: [infrastructure, infrastructure] \Rightarrow bool\ ((- \rightarrow_n -)$   
50)

**where**

$move: \llbracket G = graphI\ I; a\ @_G\ l; l \in nodes\ G; l' \in nodes\ G;$

$(a) \in actors\text{-}graph(graphI\ I); enables\ I\ l'\ (Actor\ a)\ move;$

$I' = Infrastructure\ (move\text{-}graph\text{-}a\ a\ l\ l'\ (graphI\ I))(delta\ I) \rrbracket \implies I \rightarrow_n I'$

$| get : \llbracket G = graphI\ I; a\ @_G\ l; a' @_G\ l; has\ G\ (Actor\ a, z);$

$enables\ I\ l\ (Actor\ a)\ get;$

$I' = Infrastructure$

$(Lgraph\ (gra\ G)(agra\ G)$

$((cgra\ G)(Actor\ a' :=$

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

$(lgra\ G))$

$(delta\ I)$

$\rrbracket \implies I \rightarrow_n I'$

$| put : \llbracket G = graphI\ I; a\ @_G\ l; enables\ I\ l\ (Actor\ a)\ put;$

$I' = Infrastructure$

$(Lgraph\ (gra\ G)(agra\ G)(cgra\ G)$

$((lgra\ G)(l := [z])))$

$(delta\ I) \rrbracket$

$\implies I \rightarrow_n I'$

$| put\text{-}remote : \llbracket G = graphI\ I; enables\ I\ l\ (Actor\ a)\ put;$

$I' = Infrastructure$

$(Lgraph\ (gra\ G)(agra\ G)(cgra\ G)$

$((lgra\ G)(l := [z])))$

$(delta\ I) \rrbracket$

$$\implies I \rightarrow_n I'$$

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

**instantiation** *infrastructure* :: *state*  
**begin**

**definition**

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

**instance**

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

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

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

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

**by** (*induct-tac S, auto*)

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

**by** (*induct-tac S, auto*)

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

**by** (*induct-tac l, simp, clarify, case-tac n, simp, simp*)

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

**by** (*induct-tac l, auto*)

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

**by** (*induct-tac l, auto*)

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

**by** (*induct-tac l, auto*)

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

**by** (*induct-tac l, auto*)

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

**by** (*induct-tac list, auto*)

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

**by** (*induct-tac l, simp+, clarify, erule nodup-notin*)

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

**by** (*induct-tac list, auto*)

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

**by** (*induct-tac l, auto*)

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

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

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

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

**lemma** *init-state-policy0*:  
**assumes**  $\forall z \ z'. z \rightarrow_n z' \longrightarrow \text{delta}(z) = \text{delta}(z')$   
**and**  $(x, y) \in \{(x::\text{infrastructure}, y::\text{infrastructure}). x \rightarrow_n y\}^*$   
**shows**  $\text{delta}(x) = \text{delta}(y)$   
**proof** –  
**have** *ind*:  $(x, y) \in \{(x::\text{infrastructure}, y::\text{infrastructure}). x \rightarrow_n y\}^*$   
 $\longrightarrow \text{delta}(x) = \text{delta}(y)$   
**proof** (*insert assms*, *erule rtrancl.induct*)  
**show**  $(\bigwedge a::\text{infrastructure}.$   
 $(\forall (z::\text{infrastructure})(z'::\text{infrastructure}). (z \rightarrow_n z') \longrightarrow (\text{delta } z = \text{delta } z'))$   
 $\implies$   
 $((a, a) \in \{(x::\text{infrastructure}, y::\text{infrastructure}). x \rightarrow_n y\}^*) \longrightarrow$   
 $(\text{delta } a = \text{delta } a))$   
**by** (*rule impI*, *rule refl*)  
**next fix** *a b c*  
**assume** *a0*:  $\forall (z::\text{infrastructure}) \ z'::\text{infrastructure}. z \rightarrow_n z' \longrightarrow \text{delta } z = \text{delta } z'$   
**and** *a1*:  $(a, b) \in \{(x::\text{infrastructure}, y::\text{infrastructure}). x \rightarrow_n y\}^*$   
**and** *a2*:  $(a, b) \in \{(x::\text{infrastructure}, y::\text{infrastructure}). x \rightarrow_n y\}^* \longrightarrow$   
 $\text{delta } a = \text{delta } b$   
**and** *a3*:  $(b, c) \in \{(x::\text{infrastructure}, y::\text{infrastructure}). x \rightarrow_n y\}$   
**show**  $(a, c) \in \{(x::\text{infrastructure}, y::\text{infrastructure}). x \rightarrow_n y\}^* \longrightarrow$   
 $\text{delta } a = \text{delta } c$   
**proof** –  
**have** *a4*:  $\text{delta } b = \text{delta } c$  **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*:  $\llbracket (x, y) \in \{(x::\text{infrastructure}, y::\text{infrastructure}). x \rightarrow_n y\}^* \rrbracket \implies$   
 $\text{delta}(x) = \text{delta}(y)$   
**by** (*rule init-state-policy0*, *rule delta-invariant*)



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

**by** (*clarify*, *erule state-transition-in.cases*,  
(*simp add: move-graph-a-def atI-def actors-graph-def nodes-def*)+)

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

**by** (*erule rtrancl-induct*, *rule refl*, *drule CollectD*, *simp*, *drule same-nodes0*, *simp*)

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

**proof** (*clarify*, *erule state-transition-in.cases*)

**show**  $\bigwedge(z::\text{infrastructure}) (z'::\text{infrastructure}) (G::\text{igraph}) (I::\text{infrastructure}) (a::\text{char list})$

$(l::\text{location}) (a'::\text{char list}) (za::\text{char list}) I'::\text{infrastructure}.$

$z = I \implies$

$z' = I' \implies$

$G = \text{graphI } I \implies$

$a @_G l \implies$

$a' @_G l \implies$

$\text{has } G (\text{Actor } a, za) \implies$

$\text{enables } I l (\text{Actor } a) \text{ get} \implies$

$I' =$

$\text{Infrastructure}$

$(\text{Lgraph } (\text{gra } G) (\text{agra } G))$

$((\text{cgra } G)(\text{Actor } a' := (za \# \text{fst } (\text{cgra } G (\text{Actor } a'))), \text{snd } (\text{cgra } G (\text{Actor } a')))) (\text{lgra } G))$

$(\text{delta } I) \implies$

$\text{actors-graph } (\text{graphI } z) = \text{actors-graph } (\text{graphI } z')$

**by** (*simp add: actors-graph-def nodes-def*)

**next show**  $\bigwedge(z::\text{infrastructure}) (z'::\text{infrastructure}) (G::\text{igraph}) (I::\text{infrastructure}) (a::\text{char list})$

$(l::\text{location}) (I'::\text{infrastructure}) za::\text{char list}.$

$z = I \implies$

$z' = I' \implies$

$G = \text{graphI } I \implies$

$a @_G l \implies$

$\text{enables } I l (\text{Actor } a) \text{ put} \implies$

$I' = \text{Infrastructure } (\text{Lgraph } (\text{gra } G) (\text{agra } G) (\text{cgra } G) ((\text{lgra } G)(l := [za])))$

$(\text{delta } I) \implies$

$\text{actors-graph } (\text{graphI } z) = \text{actors-graph } (\text{graphI } z')$

**by** (*simp add: actors-graph-def nodes-def*)

**next show**  $\bigwedge(z::\text{infrastructure}) (z'::\text{infrastructure}) (G::\text{igraph}) (I::\text{infrastructure}) (l::\text{location})$

$(a::\text{char list}) (I'::\text{infrastructure}) za::\text{char list}.$

$z = I \implies$

$z' = I' \implies$

$G = \text{graphI } I \implies$

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

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

**proof**  $(\text{erule rtrancl-induct})$

**show**  $\text{actors-graph } (\text{graphI } I) = \text{actors-graph } (\text{graphI } I)$

**by**  $(\text{rule refl})$

**next show**  $\bigwedge (y::\text{infrastructure}) \ z::\text{infrastructure}.$

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

```

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

end
end

```

### 3 Airplane case study

```

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  $x = \text{cockpit}$  then ["air"] else []))

**fixes**  $ex\text{-}locs' :: \text{location} \Rightarrow \text{string list}$

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

**fixes**  $ex\text{-}graph :: \text{igraph}$

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

**fixes**  $aid\text{-}graph :: \text{igraph}$

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

**fixes**  $aid\text{-}graph0 :: \text{igraph}$

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

**fixes**  $agid\text{-}graph :: \text{igraph}$

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

**fixes**  $local\text{-}policies :: [\text{igraph}, \text{location}] \Rightarrow \text{policy set}$

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

```

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

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

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

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

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

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

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

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

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

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

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

fixes astate :: identity  $\Rightarrow$  actor-state
defines astate-def: astate x  $\equiv (\text{case } x \text{ of}$ 
  "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:  $\neg \text{global-policy Airplane-scenario "Eve"}$ 
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:  $\llbracket \text{inj } f; x \neq y \rrbracket \implies f\ x \neq f\ y$ 
by (simp add: inj-eq)

lemma inj-on-lem:  $\llbracket \text{inj-on } f\ A; x \neq y; x \in A; y \in A \rrbracket \implies f\ x \neq f\ y$ 
by (simp add: inj-on-def, blast)

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

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

lemma locl-lemma2: isin ex-graph door "norm"  $\neq$  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" =  $(\neg \text{isin ex-graph door "locked"})$ 
by (insert locl-lemma2, blast)

lemma locl-lemma2a: isin aid-graph door "norm"  $\neq$  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" =  $(\neg \text{isin aid-graph door "locked"})$ 
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:  $\neg$ global-policy Airplane-in-danger ("Eve")*

**proof** (*simp add: Airplane-in-danger-def global-policy-def, rule conjI*)

**show** *"Eve"  $\notin$  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  $\in$  airplane-actors*

**shows**  *$\neg$ (Safety Airplane-in-danger s)*

**proof** (*simp add: Airplane-in-danger-def Safety-def enables-def assms*)

**show**  *$\forall x::(\text{actor} \Rightarrow \text{bool}) \times \text{action set} \in \text{local-policies aid-graph cockpit}.$*

*$\neg (\text{case } x \text{ of } (p::\text{actor} \Rightarrow \text{bool}, e::\text{action set}) \Rightarrow \text{move} \in e \wedge p (\text{Actor } s))$*

**by** (*simp add: local-policies-def aid-graph-def ex-locs'-def isin-def*)

**qed**

**lemma** *Security-problem':  $\neg$ (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** *( $\forall n::\text{char list}.$*



$Actor\ n = Actor\ "Bob" \longrightarrow n @_{aid-graph\ cabin} \longrightarrow isin\ aid-graph\ door$   
 $"locked")$   
**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*:  $a \in airplane-actors \longrightarrow global-policy\ Airplane-not-in-danger\ a$   
**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**  $a \notin airplane-actors$   
**show**  $\neg enables\ (Infrastructure\ aid-graph\ local-policies-four-eyes)\ cockpit\ (Actor\ a)\ put$   
**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**  $"Bob" @_{graphI\ Airplane-scenario}\ cockpit$   
**by** (simp add: Airplane-scenario-def atI-def ex-graph-def)  
**next show**  $cockpit \in nodes\ (graphI\ Airplane-scenario)$   
**by** (simp add: ex-graph-def Airplane-scenario-def nodes-def, blast)+  
**next show**  $door \in nodes\ (graphI\ Airplane-scenario)$   
**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**  $"Bob" \in actors-graph\ (graphI\ Airplane-scenario)$   
**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-getting-in-danger0 door
  by (simp add: Airplane-getting-in-danger0-def atI-def aid-graph0-def door-def
cockpit-def)
next show door ∈ nodes (graphI Airplane-getting-in-danger0)
  by (simp add: aid-graph0-def Airplane-getting-in-danger0-def nodes-def, blast)+
next show cabin ∈ nodes (graphI Airplane-getting-in-danger0)
  by (simp add: actors-graph-def door-def cockpit-def nodes-def cabin-def,
rule-tac x = Location 1 in exI,
simp add: Airplane-getting-in-danger0-def aid-graph0-def cockpit-def door-def
cabin-def)
next show "Bob" ∈ actors-graph (graphI Airplane-getting-in-danger0)
  by (simp add: actors-graph-def door-def cockpit-def nodes-def cabin-def
Airplane-getting-in-danger0-def aid-graph0-def, blast)
next show enables Airplane-getting-in-danger0 cabin (Actor "Bob") move
  by (simp add: Airplane-getting-in-danger0-def enables-def local-policies-def ex-creds-def
door-def
cockpit-def cabin-def)
next show Airplane-getting-in-danger =
Infrastructure (move-graph-a "Bob" door cabin (graphI Airplane-getting-in-danger0))
(delta Airplane-getting-in-danger0)
  by (unfold Airplane-getting-in-danger-def,
simp add: Airplane-getting-in-danger0-def agid-graph-def aid-graph0-def
move-graph-a-def door-def cockpit-def cabin-def, rule ext,
simp add: cabin-def door-def)
qed

```

```

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

```

```

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

```

**lemma** *step1r*: *Airplane-getting-in-danger*0  $\rightarrow_n^*$  *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*  $\vdash EF \{x. \neg \text{global-policy } x \text{ "Eve"}\}$   
**proof** (*simp add*: *check-def Air-Kripke-def*, *rule conjI*)  
**show** *Airplane-scenario*  $\in \text{Air-states}$   
**by** (*simp add*: *Air-states-def state-transition-in-refl-def*)  
**next show** *Airplane-scenario*  $\in EF \{x::\text{infrastructure}. \neg \text{global-policy } x \text{ "Eve"}\}$   
**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**  $(\forall l l'. a @_{\text{graphI } I} l \wedge a @_{\text{graphI } I} l' \longrightarrow l = l') \wedge$   
 $(\forall l. \text{nodup } a (\text{agra } (\text{graphI } I) l))$   
**shows**  $(\forall l l'. a @_{\text{graphI } I'} l \wedge a @_{\text{graphI } I'} l' \longrightarrow l = l') \wedge$   
 $(\forall l. \text{nodup } a (\text{agra } (\text{graphI } I') l))$   
**proof** (*rule state-transition-in.cases*, *rule assms(1)*)  
**show**  $\bigwedge (G::\text{igraph}) (Ia::\text{infrastructure}) (aa::\text{char list}) (l::\text{location}) (a'::\text{char list})$   
 $(z::\text{char list})$   
 $I'a::\text{infrastructure}.$   
 $I = Ia \implies$   
 $I' = I'a \implies$   
 $G = \text{graphI } Ia \implies$   
 $aa @_G l \implies$   
 $a' @_G l \implies$   
 $\text{has } G (\text{Actor } aa, z) \implies$   
 $\text{enables } Ia l (\text{Actor } aa) \text{ get} \implies$   
 $I'a =$   
 $\text{Infrastructure}$   
 $(\text{Lgraph } (\text{gra } G) (\text{agra } G)$   
 $((\text{cgra } G) (\text{Actor } a' := (z \# \text{fst } (\text{cgra } G (\text{Actor } a'))), \text{snd } (\text{cgra } G (\text{Actor } a')))))$   
 $(\text{lgra } G))$   
 $(\text{delta } Ia) \implies$   
 $(\forall (l::\text{location}) l'::\text{location}. a @_{\text{graphI } I'} l \wedge a @_{\text{graphI } I'} l' \longrightarrow l = l') \wedge$   
 $(\forall l::\text{location}. \text{nodup } a (\text{agra } (\text{graphI } I') l))$  **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 (∀ (l::location) l'::location. a @graphI I' l ∧ a @graphI I' l' → l = l') ∧
    (∀ l::location. nodup a (agra (graphI I') l)) using assms
  by (simp add: a0 a1 a2 a3 a4 a5 atI-def)
next show ∧(G::igraph) (Ia::infrastructure) (l::location) (aa::char list) (I'a::infrastructure)
  z::char list.
  I = Ia ⇒
  I' = I'a ⇒
  G = graphI Ia ⇒
  enables Ia l (Actor aa) put ⇒
  I'a = Infrastructure (Lgraph (gra G) (agra G) (cgra G) ((lgra G)(l := [z])))
(delta Ia) ⇒
  (∀ (l::location) l'::location. a @graphI I' l ∧ a @graphI I' l' → l = l') ∧
  (∀ l::location. nodup a (agra (graphI I') l))
  by (clarify, simp add: assms atI-def)
next show ∧(G::igraph) (Ia::infrastructure) (aa::char list) (l::location) (l'::location)
  I'a::infrastructure.
  I = Ia ⇒
  I' = I'a ⇒
  G = graphI Ia ⇒
  aa @G l ⇒
  l ∈ nodes G ⇒
  l' ∈ nodes G ⇒
  aa ∈ actors-graph (graphI Ia) ⇒
  enables Ia l' (Actor aa) move ⇒
  I'a = Infrastructure (move-graph-a aa l l' (graphI Ia)) (delta Ia) ⇒
  (∀ (l::location) l'::location. a @graphI I' l ∧ a @graphI I' l' → l = l') ∧
  (∀ 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 ∧(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 ∈ set (agra (graphI I) l) ∧ a ∉ set (agra (graphI I) l')
          then (agra (graphI I))(l := del a (agra (graphI I) l), l' := a # agra
(graphI I) l')
          else agra (graphI I))
        (cgra (graphI I)) (lgra (graphI I))))
      (delta I) ⇒
      a @graphI I l ⇒
      l ∈ nodes (graphI I) ⇒
      l' ∈ nodes (graphI I) ⇒

```

```

    a ∈ actors-graph (graphI I) ⇒
    enables I l' (Actor a) move ⇒
    a ∈ set (agra (graphI I) l) ⇒
    a ∉ set (agra (graphI I) l') ⇒
    a @Lgraph (gra (graphI I)) ((agra (graphI I))(l := del a (agra (graphI I) l), l' := a # agra (graphI I) l))
la ⇒
    a @Lgraph (gra (graphI I)) ((agra (graphI I))(l := del a (agra (graphI I) l), l' := a # agra (graphI I) l))
l'a ⇒
    la = l'a
apply (case-tac la ≠ l' ∧ la ≠ l ∧ l'a ≠ l' ∧ l'a ≠ l)
apply (simp add: atI-def)
apply (subgoal-tac la = l' ∨ la = l ∨ l'a = l' ∨ l'a = l)
prefer 2
using assms(2) atI-def apply blast
apply blast
by (metis agra.simps assms(2) atI-def del-nodup fun-upd-apply)
next show ∧(G::igraph) (Ia::infrastructure) (aa::char list) (l::location) (l'::location)
    I'a::infrastructure.
    I' =
    Infrastructure
    (Lgraph (gra (graphI I))
    (if a ∈ set (agra (graphI I) l) ∧ a ∉ set (agra (graphI I) l')
    then (agra (graphI I))(l := del a (agra (graphI I) l), l' := a # agra
(graphI I) l')
    else agra (graphI I))
    (cgra (graphI I)) (lgra (graphI I)))
    (delta I) ⇒
    a @graphI I l ⇒
    l ∈ nodes (graphI I) ⇒
    l' ∈ nodes (graphI I) ⇒
    a ∈ actors-graph (graphI I) ⇒
    enables I l' (Actor a) move ⇒
    a ∈ set (agra (graphI I) l) ⇒
    a ∉ set (agra (graphI I) l') ⇒
    ∀ la::location.
    (la = l → l ≠ l' → nodup a (del a (agra (graphI I) l))) ∧
    (la ≠ l → la ≠ l' → nodup a (agra (graphI I) la))
by (simp add: assms(2) nodup-down)
next show ∧(G::igraph) (Ia::infrastructure) (aa::char list) (l::location) (l'::location)
    I'a::infrastructure.
    I' =
    Infrastructure
    (Lgraph (gra (graphI I))
    (if a ∈ set (agra (graphI I) l) ∧ a ∉ set (agra (graphI I) l')
    then (agra (graphI I))(l := del a (agra (graphI I) l), l' := a # agra
(graphI I) l')
    else agra (graphI I))
    (cgra (graphI I)) (lgra (graphI I)))
    (delta I) ⇒

```

```

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

```

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

```

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

```



```

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

```

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

**assumes**  $(I, I') \in \{(x::\text{infrastructure}, y::\text{infrastructure}). x \rightarrow_n y\}^*$   
**and**  $\forall a. (\forall l l'. a @_{\text{graphI } I} l \wedge a @_{\text{graphI } I} l' \longrightarrow l = l') \wedge$   
 $(\forall l. \text{nodup } a (\text{agra } (\text{graphI } I) l))$   
**shows**  $\forall a. (\forall l l'. a @_{\text{graphI } I'} l \wedge a @_{\text{graphI } I'} l' \longrightarrow l = l') \wedge$   
 $(\forall l. \text{nodup } a (\text{agra } (\text{graphI } I') l))$   
**proof** –  
**have** *ind*:  $(\forall a. (\forall l l'. a @_{\text{graphI } I} l \wedge a @_{\text{graphI } I} l' \longrightarrow l = l') \wedge$   
 $(\forall l. \text{nodup } a (\text{agra } (\text{graphI } I) l))) \longrightarrow$   
 $(\forall a. (\forall l l'. a @_{\text{graphI } I'} l \wedge a @_{\text{graphI } I'} l' \longrightarrow l = l') \wedge$   
 $(\forall l. \text{nodup } a (\text{agra } (\text{graphI } I') l)))$   
**proof** (*insert assms(1), erule rtrancl.induct*)  
**show**  $\bigwedge a::\text{infrastructure}.$   
 $(\forall aa::\text{char list}.$   
 $(\forall (l::\text{location}) l'::\text{location}. aa @_{\text{graphI } a} l \wedge aa @_{\text{graphI } a} l' \longrightarrow l = l') \wedge$   
 $(\forall l::\text{location}. \text{nodup } aa (\text{agra } (\text{graphI } a) l))) \longrightarrow$   
 $(\forall aa::\text{char list}.$   
 $(\forall (l::\text{location}) l'::\text{location}. aa @_{\text{graphI } a} l \wedge aa @_{\text{graphI } a} l' \longrightarrow l = l') \wedge$   
 $(\forall l::\text{location}. \text{nodup } aa (\text{agra } (\text{graphI } a) l)))$  **by** *simp*  
**next show**  $\bigwedge (a::\text{infrastructure}) (b::\text{infrastructure}) (c::\text{infrastructure}).$   
 $(a, b) \in \{(x::\text{infrastructure}, y::\text{infrastructure}). x \rightarrow_n y\}^* \implies$   
 $(\forall aa::\text{char list}.$   
 $(\forall (l::\text{location}) (l'::\text{location}). (aa @_{\text{graphI } a} l \wedge aa @_{\text{graphI } a} l') \longrightarrow l =$   
 $l') \wedge$   
 $(\forall l::\text{location}. \text{nodup } aa (\text{agra } (\text{graphI } a) l))) \longrightarrow$   
 $(\forall a::\text{char list}.$   
 $(\forall (l::\text{location}) (l'::\text{location}). (a @_{\text{graphI } b} l \wedge a @_{\text{graphI } b} l') \longrightarrow l = l') \wedge$   
 $(\forall l::\text{location}. \text{nodup } a (\text{agra } (\text{graphI } b) l))) \implies$   
 $(b, c) \in \{(x::\text{infrastructure}, y::\text{infrastructure}). x \rightarrow_n y\} \implies$   
 $(\forall aa::\text{char list}.$   
 $(\forall (l::\text{location}) l'::\text{location}. (aa @_{\text{graphI } a} l \wedge aa @_{\text{graphI } a} l') \longrightarrow l = l')$   
 $\wedge$   
 $(\forall l::\text{location}. \text{nodup } aa (\text{agra } (\text{graphI } a) l))) \longrightarrow$   
 $(\forall a::\text{char list}.$   
 $(\forall (l::\text{location}) l'::\text{location}. (a @_{\text{graphI } c} l \wedge a @_{\text{graphI } c} l') \longrightarrow l = l') \wedge$   
 $(\forall l::\text{location}. \text{nodup } a (\text{agra } (\text{graphI } c) l)))$   
**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*:

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

$(\forall l. \text{nodup } a \text{ (agra (graphI Airplane-not-in-danger-init) } l))$   
**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**  $\bigwedge(l::\text{location}) \ l'::\text{location}.$   
"Charly"  
 $\in \text{set (if } l = \text{cockpit then ["Bob", "Charly"]$   
 $\text{else if } l = \text{door then [] else if } l = \text{cabin then ["Alice"]} \text{ else [])} \wedge$   
"Charly"  
 $\in \text{set (if } l' = \text{cockpit then ["Bob", "Charly"]$   
 $\text{else if } l' = \text{door then [] else if } l' = \text{cabin then ["Alice"]} \text{ else []}) \implies$   
 $l = l'$   
**by** (case-tac  $l = l'$ , assumption, rule FalseE, case-tac  $l = \text{cockpit} \vee l = \text{door} \vee$   
 $l = \text{cabin}$ ,  
erule disjE, simp, case-tac  $l' = \text{door} \vee l' = \text{cabin}$ , 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**  $\bigwedge a::\text{char list}.$   
"Charly"  $\neq a \implies$   
 $(\forall (l::\text{location}) \ l'::\text{location}.$   
 $a @_{\text{Lgraph}} \{(cockpit, door), (door, cabin)\} \quad (\lambda x::\text{location}. \quad \text{if } x = \text{cockpit then ["Bob"]$   
 $l \wedge$   
 $a @_{\text{Lgraph}} \{(cockpit, door), (door, cabin)\} \quad (\lambda x::\text{location}. \quad \text{if } x = \text{cockpit then ["Bob"]$   
 $l' \longrightarrow$   
 $l = l')$   
**by** (clarify, simp add: atI-def, case-tac  $l = l'$ , assumption, rule FalseE,  
case-tac  $l = \text{cockpit} \vee l = \text{door} \vee l = \text{cabin}$ , erule disjE, simp,  
case-tac  $l' = \text{door} \vee l' = \text{cabin}$ , erule disjE, simp, simp add: cabin-def door-def,  
simp, erule disjE, simp add: door-def cockpit-def, case-tac  $l = \text{cockpit}$ ,  
simp add: cabin-def cockpit-def, simp add: cabin-def door-def, case-tac  $l' =$   
 $\text{cockpit}$ ,  
simp, simp add: cabin-def, case-tac  $l' = \text{door}$ , simp, simp add: cabin-def,  
simp)  
**qed**

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

$(\text{Airplane-not-in-danger-init}, I) \in \{(x::\text{infrastructure}, y::\text{infrastructure}). x \rightarrow_n y\}^*$   
 $\implies \forall a. (\forall l \ l'. a @_{\text{graphI } I} l \wedge a @_{\text{graphI } I} l' \longrightarrow l = l') \wedge$   
 $(\forall l. \text{nodup } a \text{ (agra (graphI } I) \ l))$   
**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:  $\text{actors-graph (graphI Airplane-not-in-danger-init) = airplane-actors}$

**proof** (simp add: Airplane-not-in-danger-init-def ex-graph-def actors-graph-def nodes-def

*airplane-actors-def, rule equalityI)*

**show**  $\{x::\text{char list}.$   
 $\exists y::\text{location}.$   
 $(y = \text{door} \longrightarrow$   
 $(\text{door} = \text{cockpit} \longrightarrow$   
 $(\exists y::\text{location}. y = \text{cockpit} \vee y = \text{cabin} \vee y = \text{cockpit} \vee y = \text{cockpit} \wedge$   
 $\text{cockpit} = \text{cabin}) \wedge$   
 $(x = \text{"Bob"} \vee x = \text{"Charly"}) \wedge$   
 $\text{door} = \text{cockpit}) \wedge$   
 $(y \neq \text{door} \longrightarrow$   
 $(y = \text{cockpit} \longrightarrow$   
 $(\exists y::\text{location}.$   
 $y = \text{door} \vee$   
 $\text{cockpit} = \text{door} \wedge y = \text{cabin} \vee$   
 $y = \text{cockpit} \wedge \text{cockpit} = \text{door} \vee y = \text{door} \wedge \text{cockpit} = \text{cabin}) \wedge$   
 $(x = \text{"Bob"} \vee x = \text{"Charly"}) \wedge$   
 $(y \neq \text{cockpit} \longrightarrow y = \text{cabin} \wedge x = \text{"Alice"} \wedge y = \text{cabin}))\}$   
 $\subseteq \{\text{"Bob"}, \text{"Charly"}, \text{"Alice"}\}$   
**by** (*rule subsetI, drule CollectD, erule exE, (erule conjE)+,*  
*simp add: door-def cockpit-def cabin-def, (erule conjE)+, force*)  
**next show**  $\{\text{"Bob"}, \text{"Charly"}, \text{"Alice"}\}$   
 $\subseteq \{x::\text{char list}.$   
 $\exists y::\text{location}.$   
 $(y = \text{door} \longrightarrow$   
 $(\text{door} = \text{cockpit} \longrightarrow$   
 $(\exists y::\text{location}.$   
 $y = \text{cockpit} \vee y = \text{cabin} \vee y = \text{cockpit} \vee y = \text{cockpit} \wedge \text{cockpit} =$   
 $\text{cabin}) \wedge$   
 $(x = \text{"Bob"} \vee x = \text{"Charly"}) \wedge$   
 $\text{door} = \text{cockpit}) \wedge$   
 $(y \neq \text{door} \longrightarrow$   
 $(y = \text{cockpit} \longrightarrow$   
 $(\exists y::\text{location}.$   
 $y = \text{door} \vee$   
 $\text{cockpit} = \text{door} \wedge y = \text{cabin} \vee$   
 $y = \text{cockpit} \wedge \text{cockpit} = \text{door} \vee y = \text{door} \wedge \text{cockpit} = \text{cabin}) \wedge$   
 $(x = \text{"Bob"} \vee x = \text{"Charly"}) \wedge$   
 $(y \neq \text{cockpit} \longrightarrow y = \text{cabin} \wedge x = \text{"Alice"} \wedge y = \text{cabin}))\}$   
**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*:  $(\text{Airplane-not-in-danger-init}, y) \in \{(x::\text{infrastructure}, y::\text{infrastructure}). x \rightarrow_n y\}^*$

$\implies \text{actors-graph}(\text{graphI } y) = \text{airplane-actors}$

**by** (*insert Anid-airplane-actors, erule subst, rule sym, erule same-actors*)

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

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

**by** (*simp add: atI-def actors-graph-def, rule exI, rule conjI*)

**lemma not-en-get-Apnid:**  
**assumes** (*Airplane-not-in-danger-init,y*)  $\in \{(x::\text{infrastructure}, y::\text{infrastructure})\}$ .  
 $x \rightarrow_n y\}^*$   
**shows**  $\sim(\text{enables } y \text{ l } (\text{Actor } a) \text{ get})$   
**proof** –  
**have**  $\text{delta } y = \text{delta}(\text{Airplane-not-in-danger-init})$   
**by** (*insert assms, rule sym, erule-tac init-state-policy*)  
**with** *assms* **show** ?thesis  
**by** (*simp add: Airplane-not-in-danger-init-def enables-def local-policies-four-eyes-def*)

**qed**

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

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

**lemma test-graph-atI:** *"Bob"* @*graphI* *Airplane-not-in-danger-init cockpit*  
**by** (*simp add: Airplane-not-in-danger-init-def ex-graph-def atI-def*)

Invariant: number of staff in cockpit never below 2

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

```

has G (Actor a, za) ==>
enables I l (Actor a) get ==>
I' =
Infrastructure
(Lgraph (gra G) (agra G)
((cgra G)(Actor a' := (za # fst (cgra G (Actor a')), snd (cgra G (Actor
a'))))) (lgra G))
(delta I) ==>
(2::nat) ≤ length (agra (graphI z') cockpit) using assms by simp
next show ∧(G::igraph) (I::infrastructure) (a::char list) (l::location) (I'::infrastructure)
za::char list.
z = I ==>
z' = I' ==>
G = graphI I ==>
a @G l ==>
enables I l (Actor a) put ==>
I' = Infrastructure (Lgraph (gra G) (agra G) (cgra G) ((lgra G)(l := [za])))
(delta I) ==>
(2::nat) ≤ length (agra (graphI z') cockpit) using assms by simp
next show ∧(G::igraph) (I::infrastructure) (l::location) (a::char list) (I'::infrastructure)
za::char list.
z = I ==>
z' = I' ==>
G = graphI I ==>
enables I l (Actor a) put ==>
I' = Infrastructure (Lgraph (gra G) (agra G) (cgra G) ((lgra G)(l := [za])))
(delta I) ==>
(2::nat) ≤ length (agra (graphI z') cockpit) using assms by simp
next show ∧(G::igraph) (I::infrastructure) (a::char list) (l::location) (l'::location)
I'::infrastructure.
z = I ==>
z' = I' ==>
G = graphI I ==>
a @G l ==>
l ∈ nodes G ==>
l' ∈ nodes G ==>
a ∈ actors-graph (graphI I) ==>
enables I l' (Actor a) move ==>
I' = Infrastructure (move-graph-a a l l' (graphI I)) (delta I) ==>
(2::nat) ≤ length (agra (graphI z') cockpit)
proof -
fix G :: igraph and I :: infrastructure and a :: char list and l :: location and l'
:: location and I' :: infrastructure
have f1: UasI "Eve" "Charly"
using Eve-precipitating-event Insider-Eve Insider-def by force
obtain ccs :: char list ⇒ char list and ccsa :: char list ⇒ char list where
f2: ∀ cs csa. (¬ UasI cs csa ∨ Actor cs = Actor csa ∧ (∀ csb. (csa = cs ∨
csb = cs ∨ Actor csa ≠ Actor csb) ∨ csa = csb)) ∧ (UasI cs csa ∨ Actor cs ≠
Actor csa ∨ (ccs cs ≠ cs ∧ ccsa cs ≠ cs ∧ Actor (ccs cs) = Actor (ccsa cs)) ∧

```

```

ccs cs ≠ ccsa cs)
  using UasI-def by moura
  have "Bob" @graphI (Infrastructure ex-graph local-policies) Location 2
  using Airplane-not-in-danger-init-def cockpit-def test-graph-atI by force
  then have Actor "Bob" = Actor "Eve"
  using Airplane-scenario-def airplane.cockpit-foe-control airplane-axioms cockpit-def
ex-inv3 global-policy-def by blast
  then show 2 ≤ length (agra (graphI z') cockpit)
  using f2 f1 by auto
qed
qed

```

**lemma** two-person-inv1:

```

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

```

The version of two\_person\_inv above 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  $\text{card}(\text{set}(\text{agra}(\text{graphI } z))) = \text{length}(\text{agra}(\text{graphI } z))$

**lemma** nodup-card-insert:

```

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

```

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

```

  (∀ a. nodup a l) → card (set l) = length l

```

```

by (induct-tac l, simp, clarify, simp, erule impE, rule allI,
drule-tac x = aa in spec, case-tac a = aa, simp, erule nodup-notin, simp)

```

**lemma** two-person-set-inv:

```

  assumes (Airplane-not-in-danger-init,z) ∈ {(x::infrastructure, y::infrastructure).
x →n y}*
  shows (2::nat) ≤ card (set (agra (graphI z) cockpit))
proof –

```

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

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

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

**lemma**  $\text{airplane-actors-inv0}$ [ $\text{rule-format}$ ]:

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

**proof** ( $\text{clarify}$ ,  $\text{erule state-transition-in.cases}$ )

**show**  $\bigwedge (z::\text{infrastructure}) (z':\text{infrastructure}) (h::\text{char list}) (G::\text{igraph}) (I::\text{infrastructure})$   
 $(a::\text{char list}) (l::\text{location}) (a':\text{char list}) (za::\text{char list}) I'::\text{infrastructure}.$   
 $h \in \text{set } (\text{agra } (\text{graphI } z') \text{ cockpit}) \Longrightarrow$   
 $\forall h::\text{char list} \in \text{set } (\text{agra } (\text{graphI } z) \text{ cockpit}). h \in \text{airplane-actors} \Longrightarrow$   
 $(\text{Airplane-not-in-danger-init}, z) \in \{(x::\text{infrastructure}, y::\text{infrastructure}). x$   
 $\rightarrow_n y\}^* \Longrightarrow$

$z = I \Longrightarrow$   
 $z' = I' \Longrightarrow$   
 $G = \text{graphI } I \Longrightarrow$   
 $a @_G l \Longrightarrow$   
 $a' @_G l \Longrightarrow$   
 $\text{has } G (\text{Actor } a, za) \Longrightarrow$   
 $\text{enables } I l (\text{Actor } a) \text{ get} \Longrightarrow$   
 $I' =$   
 $\text{Infrastructure}$   
 $(\text{Lgraph } (\text{gra } G) (\text{agra } G))$   
 $((\text{cgra } G)(\text{Actor } a' := (za \# \text{fst } (\text{cgra } G (\text{Actor } a))), \text{snd } (\text{cgra } G (\text{Actor } a'))))$   
 $(\text{lgra } G))$   
 $(\text{delta } I) \Longrightarrow$   
 $h \in \text{airplane-actors}$

**by**  $\text{simp}$

**next show**  $\bigwedge (z::\text{infrastructure}) (z':\text{infrastructure}) (h::\text{char list}) (G::\text{igraph}) (I::\text{infrastructure})$   
 $(a::\text{char list}) (l::\text{location}) (I':\text{infrastructure}) za::\text{char list}.$   
 $h \in \text{set } (\text{agra } (\text{graphI } z') \text{ cockpit}) \Longrightarrow$   
 $\forall h::\text{char list} \in \text{set } (\text{agra } (\text{graphI } z) \text{ cockpit}). h \in \text{airplane-actors} \Longrightarrow$   
 $(\text{Airplane-not-in-danger-init}, z) \in \{(x::\text{infrastructure}, y::\text{infrastructure}). x$   
 $\rightarrow_n y\}^* \Longrightarrow$   
 $z = I \Longrightarrow$



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

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

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

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

```

(a::char list) (l::location) (l'::location) I'::infrastructure.
h ∈ set ((if a ∈ set (agra (graphI I) l) ∧ a ∉ set (agra (graphI I) l')
  then (agra (graphI I))
    (l := del a (agra (graphI I) l), l' := a # agra (graphI I) l')
  else agra (graphI I))
  cockpit) ⇒
∀ h::char list ∈ set (agra (graphI I) cockpit). h ∈ airplane-actors ⇒
(Airplane-not-in-danger-init, I) ∈ {(x::infrastructure, y::infrastructure). x
→n y}* ⇒
z = I ⇒
z' =
Infrastructure
(Lgraph (gra (graphI I))
  (if a ∈ set (agra (graphI I) l) ∧ a ∉ set (agra (graphI I) l')
    then (agra (graphI I))(l := del a (agra (graphI I) l), l' := a # agra
(graphI I) l')
  else agra (graphI I))
  (cgra (graphI I)) (lgra (graphI I)))
(delta I) ⇒
G = graphI I ⇒
a @graphI I l ⇒
l ∈ nodes (graphI I) ⇒
l' ∈ nodes (graphI I) ⇒
a ∈ actors-graph (graphI I) ⇒
enables I l' (Actor a) move ⇒
I' =
Infrastructure
(Lgraph (gra (graphI I))
  (if a ∈ set (agra (graphI I) l) ∧ a ∉ set (agra (graphI I) l')
    then (agra (graphI I))(l := del a (agra (graphI I) l), l' := a # agra
(graphI I) l')
  else agra (graphI I))
  (cgra (graphI I)) (lgra (graphI I)))
(delta I) ⇒
a ∈ set (agra (graphI I) l) ∧ a ∉ set (agra (graphI I) l') ⇒
l' ≠ cockpit ⇒ cockpit = l ⇒ h ∈ airplane-actors
  by (simp, erule bspec, erule del-up)
qed
next show ∧(z::infrastructure) (z'::infrastructure) (h::char list) (G::igraph)
(I::infrastructure)
(a::char list) (l::location) (l'::location) I'::infrastructure.
h ∈ set ((if a ∈ set (agra (graphI I) l) ∧ a ∉ set (agra (graphI I) l')
  then (agra (graphI I))
    (l := del a (agra (graphI I) l), l' := a # agra (graphI I) l')
  else agra (graphI I))
  cockpit) ⇒
∀ h::char list ∈ set (agra (graphI I) cockpit). h ∈ airplane-actors ⇒
(Airplane-not-in-danger-init, I) ∈ {(x::infrastructure, y::infrastructure). x
→n y}* ⇒

```

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

```

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

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

```

qed  
 qed  
 qed  
 qed  
 qed

**lemma** *airplane-actors-inv*:

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

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

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

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

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



$x \rightarrow_n y\}^*$   
**shows** ( $? x :: \text{identity}. x @_{\text{graphI } I} \text{cockpit} \wedge \text{Actor } x \neq \text{Actor } \text{"Eve"}$ )  
**proof** –  
**have**  $a0: (2 :: \text{nat}) \leq \text{card } (\text{set } (\text{agra } (\text{graphI } I) \text{cockpit}))$   
**by** (*insert assms, erule two-person-set-inv*)  
**have**  $a1: \text{is-singleton}(\{\text{"Charly"}\})$   
**by** (*rule is-singletonI*)  
**have**  $a6: \neg(\forall x \in \text{set}(\text{agra } (\text{graphI } I) \text{cockpit}). (\text{Actor } x = \text{Actor } \text{"Eve"}))$   
**proof** (*rule notI*)  
**assume**  $a7: \forall x :: \text{char list} \in \text{set } (\text{agra } (\text{graphI } I) \text{cockpit}). \text{Actor } x = \text{Actor } \text{"Eve"}$   
**have**  $a5: \forall x :: \text{char list} \in \text{set } (\text{agra } (\text{graphI } I) \text{cockpit}). x = \text{"Charly"}$   
**by** (*insert assms a0 a7, rule ballI, drule-tac x = x in bspec, assumption, subgoal-tac x  $\neq$  "Eve", insert Insider-Eve, unfold Insider-def, (drule mp),*  
*rule Eve-precipitating-event, simp add: UasI-def, erule Eve-not-in-cockpit)*  
**have**  $a4: \text{set } (\text{agra } (\text{graphI } I) \text{cockpit}) = \{\text{"Charly"}\}$   
**by** (*rule equalityI, rule subsetI, insert a5, simp,*  
*rule subsetI, simp, rule Set-all-unique, insert a0, force, rule a5)*  
**have**  $a2: (\text{card}((\text{set } (\text{agra } (\text{graphI } I) \text{cockpit})) :: \text{char list set})) = (1 :: \text{nat})$   
**by** (*insert a1, unfold is-singleton-altdef, erule ssubst, insert a4, erule ssubst,*  
*fold is-singleton-altdef, rule a1)*  
**have**  $a3: (2 :: \text{nat}) \leq (1 :: \text{nat})$   
**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:*  $(\text{Airplane-not-in-danger-init}, I) \in \{(x :: \text{infrastructure}, y :: \text{infrastructure}).$   
 $x \rightarrow_n y\}^* \implies$   
 $\neg \text{enables } I \text{ cockpit } (\text{Actor } \text{"Eve"}) \text{ put}$   
**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:*  $\text{Air-tp-Kripke} \vdash \text{AG } (\{x. \text{global-policy } x \text{"Eve"}\})$   
**proof** (*simp add: Air-tp-Kripke-def check-def, rule conjI*)  
**show**  $\text{Airplane-not-in-danger-init} \in \text{Air-tp-states}$   
**by** (*simp add: Airplane-not-in-danger-init-def Air-tp-states-def*  
*state-transition-in-refl-def*)  
**next show**  $\text{Airplane-not-in-danger-init} \in \text{AG } \{x :: \text{infrastructure}. \text{global-policy } x \text{"Eve"}\}$   
**proof** (*unfold AG-def, simp add: gfp-def,*  
*rule-tac x =  $\{(x :: \text{infrastructure}) \in \text{states Air-tp-Kripke}. \sim(\text{"Eve"} @_{\text{graphI } x} \text{cockpit})\}$  in exI,*  
*rule conjI*)  
**show**  $\{x :: \text{infrastructure} \in \text{states Air-tp-Kripke}. \neg \text{"Eve"} @_{\text{graphI } x} \text{cockpit}\}$   
 $\subseteq \{x :: \text{infrastructure}. \text{global-policy } x \text{"Eve"}\}$

```

    by (unfold global-policy-def, simp add: airplane-actors-def, rule subsetI,
        drule CollectD, rule CollectI, erule conjE,
        simp add: Air-tp-Kripke-def Air-tp-states-def state-transition-in-refl-def,
        erule Fend-2)
  next show {x::infrastructure ∈ states Air-tp-Kripke. ¬ "Eve" @graphI x cockpit}
    ⊆ AX {x::infrastructure ∈ states Air-tp-Kripke. ¬ "Eve" @graphI x cockpit} ∧
    Airplane-not-in-danger-init
    ∈ {x::infrastructure ∈ states Air-tp-Kripke. ¬ "Eve" @graphI x cockpit}
  proof
    show Airplane-not-in-danger-init
      ∈ {x::infrastructure ∈ states Air-tp-Kripke. ¬ "Eve" @graphI x cockpit}
    by (simp add: Airplane-not-in-danger-init-def Air-tp-Kripke-def Air-tp-states-def
        state-transition-refl-def ex-graph-def atI-def Air-tp-Kripke-def
        state-transition-in-refl-def)
  next show {x::infrastructure ∈ states Air-tp-Kripke. ¬ "Eve" @graphI x cockpit}
    ⊆ AX {x::infrastructure ∈ states Air-tp-Kripke. ¬ "Eve" @graphI x cockpit}
  proof (rule subsetI, simp add: AX-def, rule subsetI, rule CollectI, rule conjI)
    show ∧(x::infrastructure) xa::infrastructure.
      x ∈ states Air-tp-Kripke ∧ ¬ "Eve" @graphI x cockpit ⇒
      xa ∈ Collect (state-transition x) ⇒ xa ∈ states Air-tp-Kripke
    by (simp add: Air-tp-Kripke-def Air-tp-states-def state-transition-in-refl-def,
        simp add: atI-def, erule conjE,
        unfold state-transition-infra-def state-transition-in-refl-def,
        erule rtrancl-into-rtrancl, rule CollectI, simp)
  next fix x xa
    assume a0: x ∈ states Air-tp-Kripke ∧ ¬ "Eve" @graphI x cockpit
    and a1: xa ∈ Collect (state-transition x)
    show ¬ "Eve" @graphI xa cockpit
  proof -
    have b: (Airplane-not-in-danger-init, xa)
      ∈ {(x::infrastructure, y::infrastructure). x →n y}*
    proof (insert a0 a1, rule rtrancl-trans)
      show x ∈ states Air-tp-Kripke ∧ ¬ "Eve" @graphI x cockpit ⇒
        xa ∈ Collect (state-transition x) ⇒
        (x, xa) ∈ {(x::infrastructure, y::infrastructure). x →n y}*
      by (unfold state-transition-infra-def, force)
    next show x ∈ states Air-tp-Kripke ∧ ¬ "Eve" @graphI x cockpit ⇒
      xa ∈ Collect (state-transition x) ⇒
      (Airplane-not-in-danger-init, x) ∈ {(x::infrastructure, y::infrastructure).
x →n y}*
    by (erule conjE, simp add: Air-tp-Kripke-def Air-tp-states-def state-transition-in-refl-def)+
  qed
  show ?thesis
    by (insert a0 a1 b, rule-tac P = "Eve" @graphI xa cockpit in notI,
        simp add: atI-def, drule Eve-not-in-cockpit, assumption, simp)
  qed
qed
qed

```

qed  
qed

end

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

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

**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*  $\equiv a \notin airplane-actors$   
 $\longrightarrow \neg(enable\ I\ cockpit\ (Actor\ a)\ put)$

**definition** *ex-creds-def'*: *ex-creds*  $\equiv$

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

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

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

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

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

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

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

**definition** *aid-graph0-def'*: *aid-graph0*  $\equiv Lgraph$

$\{(cockpit, door), (door, cabin)\}$   
 $(\lambda\ x. if\ x = cockpit\ then\ [ "Charly" ]$   
 $\quad else\ (if\ x = door\ then\ [ "Bob" ]$   
 $\quad \quad else\ (if\ x = cabin\ then\ [ "Alice" ]\ else\ []))$

*ex-creds ex-locs*

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

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

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

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

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

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

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

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

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

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

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

$\text{Airplane-not-in-danger} \equiv \text{Infrastructure aid-graph local-policies-four-eyes}$

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

$\text{Airplane-not-in-danger-init} \equiv \text{Infrastructure ex-graph local-policies-four-eyes}$

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

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

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

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

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

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

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

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

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

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

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

**definition** *astate-def'*:  $\text{astate } x \equiv$

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

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

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

**print-interps** *airplane*

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

**axiomatization where**

*cockpit-foe-control'*:  $\text{foe-control cockpit put}$

(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  $\text{consts} :: \text{identity} =_l \text{actor}$  as an abstract function without a definition, we now also now add its semantics mimicking some of the concepts of the conservative type definition of HOL. The alternative method of using a Locale to replace the abstract type\_decl actor in the AirInsider is a more elegant method for representing and abstract type actor but it is

not working properly for our framework since it necessitates introducing a type parameter 'actor into infrastructures which then makes it impossible to instantiate them to the typeclass state in order to use CTL and Kripke and the generic state transition. Therefore, we go the former way of a post-hoc axiomatic redefinition of the abstract type actor by using axiomatization of the existing Locale "type\_definition". This is done in the following. It allows to abstractedly assume as an axiom that there is a type definition for the abstract type actor. Adding a suitable definition of a representation for this type then additionally enables to introduce a definition for the function Actor (again using axiomatization to enforce the new definition).

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

**where**

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

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

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

**by** (*simp add: Actor-Abs-def*)

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

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

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

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

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

**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*:

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

(*Rep* :: *actor*  $\Rightarrow$  *identity option*)(*Abs* :: *identity option*  $\Rightarrow$  *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  $\Rightarrow$  actor)o Actor-Abs*

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

**lemma** *UasI-Actor-redef:*

$\wedge Abs Rep :: actor \Rightarrow char \text{ list option.}$

$((Abs :: identity \text{ option } \Rightarrow actor)o Actor-Abs) \text{ "Eve" } = ((Abs :: identity \text{ option } \Rightarrow actor)o Actor-Abs) \text{ "Charly" } \wedge$

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

$((Abs :: identity \text{ option } \Rightarrow actor)o Actor-Abs) x = ((Abs :: identity \text{ option } \Rightarrow actor)o Actor-Abs) y$

$\longrightarrow x = y)$

**by** (*insert UasI-ActorAbs, simp, clarify, drule-tac x = x in spec, drule-tac x = y in spec,*

*subgoal-tac Actor-Abs x = Actor-Abs y, simp, rule Abs-inj-on, rule Actor-Abs-ran, rule Actor-Abs-ran*)

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

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

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

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

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

*Air-Kripke*

*Airplane-not-in-danger Airplane-not-in-danger-init Air-tp-states*

*Air-tp-Kripke Safety Security foe-control astate*

**by** (*rule airplane.intro, simp add: tipping-point-def,*

*simp add: Insider-def UasI-def tipping-point-def atI-def,*

*insert UasI-Actor, simp add: UasI-def,*

*insert cockpit-foe-control', simp add: foe-control-def' cockpit-def',*

*rule airplane-actors-def',*

*(simp add: airplane-locations-def' cockpit-def' door-def' cabin-def' global-policy-def'*

*ex-creds-def' ex-locs-def' ex-locs'-def' ex-graph-def' aid-graph-def'*

*aid-graph0-def'*

*agid-graph-def' local-policies-def' local-policies-four-eyes-def' Airplane-scenario-def'*)

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

end

## References

- [1] F. Kammüller and M. Kerber. Investigating airplane safety and security against insider threats using logical modeling. In *IEEE Security and Privacy Workshops, Workshop on Research in Insider Threats, WRIT'16*. IEEE, 2016.
- [2] F. Kammüller and M. Kerber. Applying the isabelle insider framework to airplane security, 2020. arXiv preprint 2003.11838.
- [3] F. Kammüller and C. W. Probst. Modeling and verification of insider threats using logical analysis. *IEEE Systems Journal, Special issue on Insider Threats to Information Security, Digital Espionage, and Counter Intelligence*, 11(2):534–545, 2017.
- [4] M. Wenzel. Re: [isabelle] typedecl versus explicit type parameters, 2009. Isabelle users mailing list.