latex

florian

November 9, 2019

Contents

theory *MC*
imports *Main*
begin
declare [[*show-types*]]

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

lemma *monotoneE*: *monotone* $\tau \Longrightarrow p \subseteq q \Longrightarrow \tau\ p \subseteq \tau\ q$
$\langle proof \rangle$

lemma *lfp1*: *monotone* $\tau \longrightarrow (lfp\ \tau = \bigcap\ \{Z.\ \tau\ Z \subseteq Z\})$
$\langle proof \rangle$

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

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

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

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

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

lemma *infchain-outruns-all*:
  assumes *finite* $(UNIV\ ::\ 'a\ set)$

and $\forall\, i :: nat.\ (\tau\ \hat{}\ i)\ (\{\}:: 'a\ set) \subset (\tau\ \hat{}\ i + (1 :: nat))\ \{\}$
shows $\forall\, j :: nat.\ \exists\, i :: nat.\ j < card\ ((\tau\ \hat{}\ i)\ \{\})$
$\langle proof \rangle$

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

$\langle proof \rangle$

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

$\langle proof \rangle$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

lemma *EF-lem2a*: **assumes** $x \in f$ **shows** $x \in EF\ f$
$\langle proof \rangle$

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

lemma *EF-lem2d*: **assumes** $x \notin EF\ f$ **shows** $x \notin f$
$\langle proof \rangle$

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

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

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

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

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

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

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

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

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

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

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

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

lemma *EF-step-star-rev*[*rule-format*]: $x \in EF\ s \Longrightarrow\ (\exists\ y \in s.\ \ x\ \to_i\!*\ y)$
⟨proof⟩

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

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

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

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

lemma *AG-lem3*: $AG\ s = (s \cap (AX\ \ (AG\ s)))$
⟨proof⟩

lemma *AG-step*: $y \to_i z \Longrightarrow y \in AG\ s \Longrightarrow z \in AG\ s$
⟨proof⟩

lemma *AG-all-s*: $\ x \to_i\!*\ y \Longrightarrow x \in AG\ s \Longrightarrow y \in AG\ s$
⟨proof⟩

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

⟨*proof*⟩

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

end
theory *AirInsider*
imports *MC*
begin
datatype $action = get \mid move \mid eval \mid put$
typedecl *actor*
type-synonym $identity = string$
consts $Actor :: string => actor$
type-synonym $policy = ((actor => bool) * action\ set)$

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

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 $psy\text{-}states = happy \mid depressed \mid disgruntled \mid angry \mid stressed$
datatype $motivations = financial \mid political \mid revenge \mid curious \mid competitive\text{-}advantage \mid power \mid peer\text{-}recognition$

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

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

consts $Isolation :: [actor\text{-}state,\ (identity * identity)\ set\ ] \Rightarrow bool$

definition $lay\text{-}off :: [infrastructure, actor\ set] \Rightarrow infrastructure$
where $lay\text{-}off\ G\ A \equiv G$

consts $social\text{-}graph :: (identity * identity)\ set$

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

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

consts *astate* :: *identity* ⇒ *actor-state*

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

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

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

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

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

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

lemma *not-enableI*: (∀ (*p*,*e*) ∈ *delta I* (*graphI I*) *l*. (~(*h* : *e*) | (~(*p*(*a*)))))
⟹ ~(*enables I l a h*)
⟨*proof*⟩

lemma *not-enableI2*: ⟦⋀ *p e*. (*p*,*e*) ∈ *delta I* (*graphI I*) *l* ⟹
(~(*t* : *e*) | (~(*p*(*a*)))) ⟧ ⟹ ~(*enables I l a t*)
⟨*proof*⟩

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

lemma *not-enableE2*: ⟦ ~(*enables I l a t*); (*p*,*e*) ∈ *delta I* (*graphI I*) *l*;
t : *e* ⟧ ⟹ (~(*p*(*a*)))
⟨*proof*⟩

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

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

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

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

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

$(delta\ I)$ ⟧
$\Longrightarrow I \rightarrow_n I'$

instantiation *infrastructure* :: *state*
begin

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

instance
⟨*proof*⟩

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

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

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

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

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

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

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

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

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

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

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

$\langle proof \rangle$

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

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

lemma *move-graph-eq*: $move\text{-}graph\text{-}a\ a\ l\ l\ g = g$
$\langle proof \rangle$

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

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

lemma *init-state-policy*: $\llbracket\ (x,y) \in \{(x::infrastructure,\ y::infrastructure).\ x \rightarrow_n y\}^*$
$\rrbracket \Longrightarrow$
$delta(x) = delta(y)$
$\langle proof \rangle$

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

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

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

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

end
end
theory *Airplane*
imports *AirInsider*
begin

declare [[*show-types*]]

datatype *doorstate* = *locked* | *norm* | *unlocked*
datatype *position* = *air* | *airport* | *ground*

locale *airplane* =

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

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

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

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

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

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

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

fixes *ex-graph* :: *igraph*
defines *ex-graph-def*: *ex-graph* ≡ *Lgraph*
{(*cockpit*, *door*),(*door*,*cabin*)}
(λ *x*. *if x* = *cockpit then* [''*Bob*'', ''*Charly*'']

$\quad\quad$ *else* (*if* x = *door then* []
$\quad\quad\quad$ *else* (*if* x = *cabin then* [''*Alice*''] *else* [])))
$\quad$ *ex-creds ex-locs*

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

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

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

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

fixes *local-policies-four-eyes* :: [*igraph*, *location*] $\Rightarrow$ *policy set*
defines *local-policies-four-eyes-def*: *local-policies-four-eyes G* $\equiv$
$\quad$ (λ *y*. *if* y = *cockpit then*
$\quad\quad\quad$ {(λ *x*. (*?* *n*. (*n* $@_G$ *cockpit*) $\wedge$ *Actor n* = *x*) $\wedge$
$\quad\quad\quad\quad$ *2* $\le$ *length*(*agra G y*) $\wedge$ ($\forall$ *h* $\in$ *set*(*agra G y*). *h* $\in$ *airplane-actors*),
{*put*}),

$(\lambda\ x.\ (?\ n.\ (n\ @_G\ cabin) \wedge Actor\ n = x \wedge has\ G\ (x,\ ''PIN'') \wedge$
$isin\ G\ door\ ''norm''\ ),\{move\})$
$\}$
$else\ (if\ y = door\ then$
$\{(\lambda\ x.\ ((?\ n.\ (n\ @_G\ cockpit) \wedge Actor\ n = x) \wedge 3 \leq length(agra\ G$
$cockpit)),\ \{move\})\}$
$else\ (if\ y = cabin\ then$
$\{(\lambda\ x.\ ((?\ n.\ (n\ @_G\ door) \wedge Actor\ n = x)),\ \{move\})\}$
$else\ \{\}))$

fixes *Airplane-scenario* :: *infrastructure*
defines *Airplane-scenario-def*:
Airplane-scenario ≡ *Infrastructure ex-graph local-policies*

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

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

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

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

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

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

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

fixes *Air-tp-states*

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

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

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

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

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

assumes *Eve-precipitating-event*: *tipping-point* (*astate* ''*Eve*'')
assumes *Insider-Eve*: *Insider* ''*Eve*'' {''*Charly*''}
assumes *isin-inj*: $\forall$ *G*. *inj* (*isin G door*)
assumes *cockpit-foe-control*: *foe-control cockpit put*

begin

lemma *ex-inv*: *global-policy Airplane-scenario* ''*Bob*''
⟨*proof*⟩

lemma *ex-inv2*: *global-policy Airplane-scenario* ''*Charly*''
⟨*proof*⟩

lemma *ex-inv3*: $\neg$*global-policy Airplane-scenario* ''*Eve*''
⟨*proof*⟩

lemma *Safety*: *Safety Airplane-scenario* (''*Alice*'')
⟨*proof*⟩

lemma *inj-lem*: $[\![$ *inj f*; $x \neq y$ $]\!] \Longrightarrow f\,x \neq f\,y$
⟨*proof*⟩

lemma *locl-lemma0*: *isin G door* ''*norm*'' $\neq$ *isin G door* ''*locked*''
⟨*proof*⟩

lemma *locl-lemma*: *isin G door ''norm''* = ($\neg$ *isin G door ''locked''*)
⟨*proof*⟩

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

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

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

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

lemma *Safety-in-danger*:
fixes *s*
assumes *s* $\in$ *airplane-actors*
shows $\neg$(*Safety Airplane-in-danger s*)
⟨*proof*⟩

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

lemma *ex-inv5*: *a* $\in$ *airplane-actors* $\longrightarrow$ *global-policy Airplane-not-in-danger a*
⟨*proof*⟩

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

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

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

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

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

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

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

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

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

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

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

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

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

lemma *Anid-airplane-actors*: *actors-graph* (*graphI Airplane-not-in-danger-init*) = *airplane-actors*

$\langle proof \rangle$

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

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

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

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

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

$\langle proof \rangle$

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

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

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

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

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

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

lemma *Pred-all-unique*: $\bigwedge\ P.\ (\llbracket\ \forall\ x.\ (P\ x \longrightarrow (x = c))\ \rrbracket \Longrightarrow P\ c)$
⟨*proof*⟩

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

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

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

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

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

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

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

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

end

end