latex

florian

November 17, 2019

Contents

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

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

lemma *monotoneE*: *monotone* $\tau \Longrightarrow p \subseteq q \Longrightarrow \tau\ p \subseteq \tau\ q$
⟨*proof*⟩

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

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

primrec *power* :: ['*a* ⇒ '*a*, *nat*] ⇒ ('*a* ⇒ '*a*) ((- ^ -) *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 $\forall\ i.\ (\tau\ \hat{}\ i)\ (\{\}) \subseteq (\tau\ \hat{}(i + 1))(\{\})$
⟨*proof*⟩

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

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

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

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

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*

⟨*proof*⟩

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

⟨*proof*⟩

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

lemma *Suc-less-le*: $x < (y - n) \Longrightarrow x \leq (y - (Suc\ n))$
⟨*proof*⟩

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 $(\forall\ i :: nat.\ card((\tau\ \hat{}\ i)\ (UNIV ::'a\ set)) \leq (card\ (UNIV :: 'a\ set)) - i)$
⟨*proof*⟩

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 $card((\tau\ \hat{}\ (card\ (UNIV ::'a\ set)))\ (UNIV ::'a\ set)) \leq 0$
⟨*proof*⟩

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

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** $p \leq f\ p$ **shows** $p \leq (f\hat{}\hat{}k)\ (top::'a::order\text{-}top)$
⟨*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 $gfp\ f\ = (f \hat{}\ n)\ UNIV$
⟨*proof*⟩

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

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

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

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

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

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

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

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

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

lemma *EF-lem2a*: **assumes** $x \in f$ **shows** $x \in EF\ f$
⟨*proof*⟩

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

lemma *EF-lem2d*: **assumes** $x \notin EF\ f$ **shows** $x \notin f$
⟨*proof*⟩

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

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

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

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

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

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

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

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

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*]: $x \in EF\ s \Longrightarrow\ (\exists\ y \in s.\ \ x\ \rightarrow_i *\ y)$
⟨*proof*⟩

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

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*: $y \rightarrow_i z \Longrightarrow y \in AG\ s \Longrightarrow z \in AG\ s$
⟨*proof*⟩

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

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

⟨*proof*⟩

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

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

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

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

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

datatype *location* = *Location nat*

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

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

definition *nodes* :: *igraph* ⇒ *location set*
where *nodes g* == { *x*. (*? y*. ((*x*,*y*): *gra g*) | ((*y*,*x*): *gra g*))}

definition *actors-graph* :: *igraph* ⇒ *identity set*
where *actors-graph g* == {*x*. *? y*. *y* : *nodes g* ∧ *x* ∈ *set*(*agra g y*)}

primrec *graphI* :: *infrastructure* ⇒ *igraph*
where *graphI* (*Infrastructure g d*) = *g*

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

definition *credentials* :: *string list* ∗ *string list* ⇒ *string set*
where *credentials lxl* ≡ *set* (*fst lxl*)
definition *has* :: [*igraph*, *actor* ∗ *string*] ⇒ *bool*
where *has G ac* ≡ *snd ac* ∈ *credentials*(*cgra G* (*fst ac*))
definition *roles* :: *string list* ∗ *string list* ⇒ *string set*
where *roles lxl* ≡ *set* (*snd lxl*)
definition *role* :: [*igraph*, *actor* ∗ *string*] ⇒ *bool*
where *role G ac* ≡ *snd ac* ∈ *roles*(*cgra G* (*fst ac*))

definition *isin* :: [*igraph*,*location*, *string*] ⇒ *bool*
where *isin G l s* ≡ *s* ∈ *set*(*lgra G l*)

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

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

definition *tipping-point* :: *actor-state* ⇒ *bool* **where**
tipping-point a ≡ ((*motivation a* ≠ {}) ∧ (*happy* ≠ *psy-state a*))

consts *Isolation* :: [*actor-state*, (*identity* ∗ *identity*) *set*] ⇒ *bool*

definition *lay-off* :: [*infrastructure*,*actor set*] ⇒ *infrastructure*
where *lay-off G A* ≡ *G*

consts *social-graph* :: (*identity* ∗ *identity*) *set*

definition *UasI* :: [*identity*, *identity*] ⇒ *bool*
where *UasI a b* ≡ (*Actor a* = *Actor b*) ∧ (∀ *x y*. *x* ≠ *a* ∧ *y* ≠ *a* ∧ *Actor x* =

$Actor\ y \longrightarrow x = y)$

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

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

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

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

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

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

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

lemma *not-enableI*: $(\forall\ (p,e) \in delta\ I\ (graphI\ I)\ l.\ (\sim(h : e) \mid (\sim(p(a)))))$
$\Longrightarrow \sim(enables\ I\ l\ a\ h)$
⟨*proof*⟩

lemma *not-enableI2*: $[\![\bigwedge p\ e.\ (p,e) \in delta\ I\ (graphI\ I)\ l \Longrightarrow (\sim(t : e) \mid\ (\sim(p(a))))]\!] \Longrightarrow \sim(enables\ I\ l\ a\ t)$
⟨*proof*⟩

lemma *not-enableE*: $[\![\sim(enables\ I\ l\ a\ t); (p,e) \in delta\ I\ (graphI\ I)\ l]\!]$
$\Longrightarrow (\sim(t : e) \mid\ (\sim(p(a))))$
⟨*proof*⟩

lemma *not-enableE2*: $[\![\ {}^{\sim}(enables\ I\ l\ a\ t);\ (p,e) \in delta\ I\ (graphI\ I)\ l;$
$t : e\ ]\!] \Longrightarrow ({}^{\sim}(p(a)))$
⟨*proof*⟩

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

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

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

definition *move-graph-a* :: [*identity*, *location*, *location*, *igraph*] ⇒ *igraph*
where *move-graph-a n l l' g* ≡ *Lgraph* (*gra g*)
(*if n* ∈ *set* ((*agra g*) *l*) & *n* ∉ *set* ((*agra g*) *l'*) *then*
((*agra g*)(*l* := *del n* (*agra g l*)))(*l'* := (*n* # (*agra g l'*)))
else (*agra g*))(*cgra g*)(*lgra g*)

inductive *state-transition-in* :: [*infrastructure*, *infrastructure*] ⇒ *bool* ((- →$_n$ -) *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* : $[\![\ G = graphI\ I;\ a\ @_G\ l;\ enables\ I\ l\ (Actor\ a)\ put;$
$I' = Infrastructure$
$(Lgraph\ (gra\ G)(agra\ G)(cgra\ G)$
$((lgra\ G)(l := [z])))$
$(delta\ I)\ ]\!]$

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

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*)
⟨*proof*⟩

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

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

lemma *move-graph-eq*: *move-graph-a a l l g* = *g*
⟨*proof*⟩

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

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

lemma *init-state-policy*: ⟦ (*x*,*y*) $\in$ {(*x*::*infrastructure*, *y*::*infrastructure*). *x* $\rightarrow_n$ *y*}*
⟧ $\Longrightarrow$
delta(*x*) = *delta*(*y*)
⟨*proof*⟩

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

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

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

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

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

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

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

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

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

fixes *local-policies-four-eyes* :: [*igraph*, *location*] ⇒ *policy set*
defines *local-policies-four-eyes-def*: *local-policies-four-eyes G* ≡
(λ *y*. *if y* = *cockpit then*
{(λ *x*. (*?* *n*. (*n* @$_G$ *cockpit*) ∧ *Actor n* = *x*) ∧
2 ≤ *length*(*agra G y*) ∧ (∀ *h* ∈ *set*(*agra G y*). *h* ∈ *airplane-actors*),

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

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

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

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

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

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

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

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

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

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

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

fixes *Safety* :: [*infrastructure*, *identity*] ⇒ *bool*
defines *Safety-def*: *Safety I a* ≡ *a* ∈ *airplane-actors*
⟶ (*enables I cockpit* (*Actor a*) *move*)

fixes *Security* :: [*infrastructure*, *identity*] ⇒ *bool*
defines *Security-def*: *Security I a* ≡ (*isin* (*graphI I*) *door* ''*locked*'')
⟶ ¬(*enables I cockpit* (*Actor a*) *move*)

fixes *foe-control* :: [*location*, *action*] ⇒ *bool*
defines *foe-control-def*: *foe-control l c* ≡
(! *I*:: *infrastructure*. (*?* x :: *identity*.
x @$_{graphI\ I}$ *l* ∧ *Actor x* ≠ *Actor* ''*Eve*'')
⟶ ¬(*enables I l* (*Actor* ''*Eve*'') *c*))

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

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

begin

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

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

lemma *ex-inv3*: ¬*global-policy Airplane-scenario* ''*Eve*''
⟨*proof*⟩

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

lemma *inj-lem*: ⟦ *inj f*; *x* ≠ *y* ⟧ ⟹ *f x* ≠ *f y*
⟨*proof*⟩

lemma *inj-on-lem*: ⟦ *inj-on f A*; *x* ≠ *y*; *x*∈ *A*; *y* ∈ *A* ⟧ ⟹ *f x* ≠ *f y*
⟨*proof*⟩

lemma *inj-lemma'*: *inj-on* (*isin ex-graph door*) {''*locked*'',''*norm*''}
⟨*proof*⟩

lemma *inj-lemma''*: *inj-on* (*isin aid-graph door*) {''*locked*'',''*norm*''}
⟨*proof*⟩

lemma *locl-lemma2*: *isin ex-graph door* ''*norm*'' ≠ *isin ex-graph door* ''*locked*''
⟨*proof*⟩

lemma *locl-lemma3*: *isin ex-graph door* ''*norm*'' = (¬ *isin ex-graph door* ''*locked*'')
⟨*proof*⟩

lemma *locl-lemma2a*: *isin aid-graph door* ''*norm*'' ≠ *isin aid-graph door* ''*locked*''
⟨*proof*⟩

lemma *locl-lemma3a*: *isin aid-graph door* ''*norm*'' = (¬ *isin aid-graph 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*: ¬*global-policy Airplane-in-danger* (''*Eve*'')
⟨*proof*⟩

lemma *Safety-in-danger*:
fixes *s*
assumes *s* ∈ *airplane-actors*
shows ¬(*Safety Airplane-in-danger s*)
⟨*proof*⟩

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

lemma *ex-inv5*: *a* ∈ *airplane-actors* ⟶ *global-policy Airplane-not-in-danger a*
⟨*proof*⟩

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

⟨*proof*⟩

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

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

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

lemma *step0r*: *Airplane-scenario* $\rightarrow_n*$ *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*:
$(Airplane\text{-}not\text{-}in\text{-}danger\text{-}init,\ I) \in \{(x::infrastructure,\ y::infrastructure).\ x \rightarrow_n y\}^*$
$\Longrightarrow \quad \forall\ a.\ (\forall\ l\ l'.\ a\ @_{graphI\ I}\ l \wedge a\ @_{graphI\ I}\ l' \longrightarrow l = l') \wedge$
$(\forall\ l.\ nodup\ a\ (agra\ (graphI\ I)\ l))$
⟨*proof*⟩

lemma *Anid-airplane-actors*: $actors\text{-}graph\ (graphI\ Airplane\text{-}not\text{-}in\text{-}danger\text{-}init) = airplane\text{-}actors$
⟨*proof*⟩

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

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

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

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

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

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

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 $(2{::}nat) \leq length\ (agra\ (graphI\ z')\ cockpit)$
⟨*proof*⟩

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 $(2{::}nat) \leq length\ (agra\ (graphI\ z)\ cockpit)$
⟨*proof*⟩

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*: $[\![\ ?\ x.\ P\ x;\ (!\ x.\ P\ x \longrightarrow x = c)]\!] \Longrightarrow P\ c$
⟨*proof*⟩

lemma *Set-all-unique*: $[\![\ S \neq \{\};\ (\forall\ x \in S.\ x = c)\ ]\!] \Longrightarrow c \in S$
⟨*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-not-in-danger-init*,*I*) ∈ {(*x*::*infrastructure*, *y*::*infrastructure*). *x* →$_n$ *y*}*
shows (*? x* :: *identity*. *x* @$_{graphI\ I}$ *cockpit* ∧ *Actor x* ≠ *Actor ''Eve''*)
⟨*proof*⟩

lemma *Fend-2*: (*Airplane-not-in-danger-init*,*I*) ∈ {(*x*::*infrastructure*, *y*::*infrastructure*). *x* →$_n$ *y*}* ⟹
¬ *enables I cockpit* (*Actor ''Eve''*) *put*
⟨*proof*⟩

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

end

definition *airplane-actors-def'*: *airplane-actors* ≡ {*''Bob''*, *''Charly''*, *''Alice''*}
definition *airplane-locations-def'*:
airplane-locations ≡ {*Location 0*, *Location 1*, *Location 2*}
definition *cockpit-def'*: *cockpit* ≡ *Location 2*
definition *door-def'*: *door* ≡ *Location 1*
definition *cabin-def'*: *cabin* ≡ *Location 0*
definition *global-policy-def'*: *global-policy I a* ≡ *a* ∉ *airplane-actors*
⟶ ¬(*enables I cockpit* (*Actor a*) *put*)
definition *ex-creds-def'*: *ex-creds* ≡
(λ *x*.(*if x* = *Actor ''Bob''*
then ([*''PIN''*], [*''pilot''*])
else (*if x* = *Actor ''Charly''*
then ([*''PIN''*],[*''copilot''*])
else (*if x* = *Actor ''Alice''*
then ([*''PIN''*],[*''flightattendant''*])
else ([],[])))))

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

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

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

```
            else (if x = cabin then [''Alice''] else []))
    ex-creds ex-locs

definition aid-graph-def': aid-graph ≡ Lgraph
    {(cockpit, door),(door,cabin)}
    (λ x. if x = cockpit then [''Charly'']
        else (if x = door then []
            else (if x = cabin then [''Bob'', ''Alice''] else [])))
    ex-creds ex-locs'

definition aid-graph0-def': aid-graph0 ≡ Lgraph
    {(cockpit, door),(door,cabin)}
    (λ x. if x = cockpit then [''Charly'']
        else (if x = door then [''Bob'']
            else (if x = cabin then [''Alice''] else [])))
     ex-creds ex-locs

definition agid-graph-def': agid-graph ≡ Lgraph
    {(cockpit, door),(door,cabin)}
    (λ x. if x = cockpit then [''Charly'']
        else (if x = door then []
            else (if x = cabin then [''Bob'', ''Alice''] else [])))
    ex-creds ex-locs

definition local-policies-def': local-policies G ≡
  (λ y. if y = cockpit then
          {(λ x. (? n. (n @_G cockpit) ∧ Actor n = x), {put}),
           (λ x. (? n. (n @_G cabin) ∧ Actor n = x ∧ has G (x, ''PIN'')
                ∧ isin G door ''norm''),{move})
          }
       else (if y = door then {(λ x. True, {move}),
                  (λ x. (? n. (n @_G cockpit) ∧ Actor n = x), {put})}
             else (if y = cabin then {(λ x. True, {move})}
                   else {})))
definition local-policies-four-eyes-def': local-policies-four-eyes G ≡
  (λ y. if y = cockpit then
          {(λ x.  (? n. (n @_G cockpit) ∧ Actor n = x) ∧
              2 ≤ length(agra G y) ∧ (∀ h ∈ set(agra G y). h ∈ airplane-actors),
{put}),
           (λ x. (? n. (n @_G cabin) ∧ Actor n = x ∧ has G (x, ''PIN'') ∧
                    isin G door ''norm'' ),{move})
          }
       else (if y = door then
              {(λ x.  ((? n. (n @_G cockpit) ∧ Actor n = x) ∧ 3 ≤ length(agra G
cockpit)), {move})}
             else (if y = cabin then
                    {(λ x. ((? n. (n @_G door) ∧ Actor n = x)), {move})}
                   else {})))
```

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

definition *Airplane-in-danger-def'*:
Airplane-in-danger ≡ *Infrastructure aid-graph local-policies*

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

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

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

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

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

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

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

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

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

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

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

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

print-interps *airplane*

axiomatization where

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

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

lemma *UasI-ActorAbs*: *Actor-Abs ''Eve''* = *Actor-Abs ''Charly''* ∧
(∀ (*x*::*char list*) *y*::*char list*. *x* ≠ *''Eve''* ∧ *y* ≠ *''Eve''* ∧ *Actor-Abs x* = *Actor-Abs y* ⟶ *x* = *y*)
⟨*proof*⟩

lemma *Actor-Abs-ran*: *Actor-Abs x* ∈ {*y* :: *identity option*. *y* ∈ *Some* ' {*x* :: *identity*. *x* ∉ {*''Eve''*, *''Charly''*}}| *y* = *None*}
⟨*proof*⟩

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

lemma *Abs-inj-on*: ⋀ *Abs Rep*:: *actor* ⇒ *char list option*. *x* ∈ {*y* :: *identity option*. *y* ∈ *Some* ' {*x* :: *identity*. *x* ∉ {*''Eve''*, *''Charly''*}}| *y* = *None*}
⟹ *y* ∈ {*y* :: *identity option*. *y* ∈ *Some* ' {*x* :: *identity*. *x* ∉ {*''Eve''*, *''Charly''*}}| *y* = *None*}
⟹ (*Abs* :: *char list option* ⇒ *actor*) *x* = *Abs y* ⟹ *x* = *y*
⟨*proof*⟩

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

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

lemma *UasI-Actor-redef*:
⋀ *Abs Rep*:: *actor* ⇒ *char list option*.
((*Abs* :: *identity option* ⇒ *actor*)*o Actor-Abs*) *''Eve''* = ((*Abs* :: *identity option* ⇒ *actor*)*o Actor-Abs*) *''Charly''* ∧
(∀ (*x*::*char list*) *y*::*char list*. *x* ≠ *''Eve''* ∧ *y* ≠ *''Eve''* ∧
((*Abs* :: *identity option* ⇒ *actor*)*o Actor-Abs*) *x* = ((*Abs* :: *identity option* ⇒ *actor*)*o Actor-Abs*) *y*
⟶ *x* = *y*)

⟨*proof*⟩

lemma *UasI-Actor*: *UasI ''Eve'' ''Charly''*
⟨*proof*⟩

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
⟨*proof*⟩

end