Attack Trees in Isabelle for GDPR compliance of IoT healthcare systems

Florian Kammüller

March 17, 2025

Abstract

In this article, we present a proof theory for Attack Trees. Attack Trees are a well established and useful model for the construction of attacks on systems since they allow a stepwise exploration of high level attacks in application scenarios. Using the expressiveness of Higher Order Logic in Isabelle, we succeed in developing a generic theory of Attack Trees with a state-based semantics based on Kripke structures and CTL (see [2] for more details). The resulting framework allows mechanically supported logic analysis of the meta-theory of the proof calculus of Attack Trees and at the same time the developed proof theory enables application to case studies. A central correctness and completeness result proved in Isabelle establishes a connection between the notion of Attack tTree validity and CTL. The application is illustrated on the example of a healthcare IoT system and GDPR compliance verification. A more detailed account of the Attack Tree formalisation is given in [3] and the case study is described in detail in [4].

Contents

1 Kripke structures and CTL

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

theory *MC*
imports *Main*
begin

1.1 Lemmas to support least and greatest fixpoints

lemma *predtrans-empty*:
assumes *mono* (τ :: $'a$ *set* $\Rightarrow$ $'a$ *set*)
shows $\forall$ *i*. (τ $\widehat{\ }\widehat{\ }$ *i*) ({}) $\subseteq$ (τ $\widehat{\ }\widehat{\ }$(*i* + *1*))({})
using *assms funpow-decreasing le-add1* **by** *blast*

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

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

lemma *infchain-outruns-all*:
assumes *finite* (*UNIV* :: *'a set*)
and $\forall i$ :: *nat*. $((\tau :: 'a\ set \Rightarrow 'a\ set) \frown i)\ (\{\} :: 'a\ set) \subset (\tau \frown (i + 1))\ \{\}$
shows $\forall j$ :: *nat*. $\exists i$ :: *nat*. $j <$ *card* $((\tau \frown i)\ \{\})$
proof (*rule allI*, *induct-tac j*)
show $\exists i.\ 0 <$ *card* $((\tau \frown i)\ \{\})$ **using** *assms*
by (*metis bot.not-eq-extremum card-gt-0-iff finite-subset subset-UNIV*)
next show $\bigwedge j\ n.\ \exists i.\ n <$ *card* $((\tau \frown i)\ \{\})$
$\Longrightarrow \exists i.\ Suc\ n <$ *card* $((\tau \frown i)\ \{\})$
proof −
fix *j n*
assume *a*: $\exists i.\ n <$ *card* $((\tau \frown i)\ \{\})$
obtain *i* **where** $n <$ *card* $((\tau \frown i)\ \{\})$
using *a* **by** *blast*
thus $\exists\ i.\ Suc\ n <$ *card* $((\tau \frown i)\ \{\})$ **using** *assms*
by (*meson finite-subset le-less-trans le-simps*(*3*) *psubset-card-mono subset-UNIV*)
qed
qed

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

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

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

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

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

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

lemma *predtrans-UNIV*:
assumes *mono* (*τ* :: (*'a set* ⇒ *'a set*))
shows ∀ *i*. (*τ* ^^ *i*) (*UNIV*) ⊇ (*τ* ^^(*i* + *1*))(*UNIV*)
proof (*rule allI*, *induct-tac i*)
show (*τ* ^^ ((*0*) + (*1*))) *UNIV* ⊆ (*τ* ^^ (*0*)) *UNIV*
by *simp*
next show ⋀(*i*) *n*.
(*τ* ^^ (*n* + (*1*))) *UNIV* ⊆ (*τ* ^^ *n*) *UNIV* ⟹ (*τ* ^^ (*Suc n* + (*1*))) *UNIV* ⊆ (*τ* ^^ *Suc n*) *UNIV*
proof −
fix *i n*
assume *a*: (*τ* ^^ (*n* + (*1*))) *UNIV* ⊆ (*τ* ^^ *n*) *UNIV*
have (*τ* ((*τ* ^^ *n*) *UNIV*)) ⊇ (*τ* ((*τ* ^^ (*n* + (*1* :: *nat*))) *UNIV*)) **using** *assms a*
by (*rule monoE*)
thus (*τ* ^^ (*Suc n* + (*1*))) *UNIV* ⊆ (*τ* ^^ *Suc n*) *UNIV* **by** *simp*
qed
qed

lemma *Suc-less-le*: *x* < (*y* − *n*) ⟹ *x* ≤ (*y* − (*Suc n*))
by *simp*

lemma *card-univ-subtract*:
assumes *finite* (*UNIV* :: *'a set*) **and** *mono τ*

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

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

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

lemma *UNIV-tau-i-is-empty*:
assumes *finite* $(UNIV :: 'a\ set)$ **and** *mono* $(\tau :: ('a\ set \Rightarrow 'a\ set))$
and $(\forall i :: nat.\ ((\tau :: 'a\ set \Rightarrow 'a\ set) \hat{\ }\hat{\ } (i + (1 :: nat)))(UNIV :: 'a\ set) \subset (\tau \hat{\ }\hat{\ } i)\ UNIV)$
shows $(\tau \hat{\ }\hat{\ } (card\ (UNIV ::'a\ set)))\ (UNIV ::'a\ set) = \{\}$
by (*meson assms card-UNIV-tau-i-below-zero finite-card-zero-empty finite-subset subset-UNIV*)

lemma *down-chain-reaches-empty*:
assumes *finite* (*UNIV* :: *'a set*) **and** *mono* (*τ* :: *'a set* ⇒ *'a set*)
and (∀ *i* :: *nat*. ((*τ* :: *'a set* ⇒ *'a set*) ^^ (*i* + (*1* :: *nat*))) *UNIV* ⊂ (*τ* ^^ *i*) *UNIV*)
shows ∃ (*j* :: *nat*). (*τ* ^^ *j*) *UNIV* = {}
using *UNIV-tau-i-is-empty assms* **by** *blast*

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

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

lemma *lfp-loop*:
assumes *finite* (*UNIV* :: *'b set*) **and** *mono* (*τ* :: (*'b set* ⇒ *'b set*))
shows ∃ *n* . *lfp τ* = (*τ* ^^ *n*) {}
proof −
have ∃ *i*. (*τ* ^^ *i*) {} = (*τ* ^^ (*i* + (*1*))) {} **using** *assms*
by (*rule finite-fixp*)
from *this* **obtain** *i* **where** (*τ* ^^ *i*) {} = (*τ* ^^ (*i* + (*1*))) {}
by (*erule exE*)
hence (*τ* ^^ *i*) {} = (*τ* ^^ *Suc i*) {}
by *simp*
hence (*τ* ^^ *Suc i*) {} = (*τ* ^^ *i*) {}
by (*rule sym*)

hence *lfp* $\tau = (\tau \hat{\ } i)\ \{\}$
by (*simp add*: *assms*(*2*) *lfp-Kleene-iter*)
thus $\exists\ n$. *lfp* $\tau = (\tau \hat{\ } n)\ \{\}$
by (*rule exI*)
qed

These next two are repeated from the corresponding theorems in HOL/ZF/Nat.thy for the sake of self-containedness of the exposition.

lemma *Kleene-iter-gpfp*:
assumes *mono f* **and** $p \leq f\ p$ **shows** $p \leq (f\hat{\ }k)\ (top::'a::order\text{-}top)$
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-loop*:
assumes *finite* (*UNIV* :: *'b set*)
and *mono* (τ :: (*'b set* $\Rightarrow$ *'b set*))
shows $\exists\ n$. *gfp* $\tau = (\tau \hat{\ } n)\, UNIV$
proof −
have $\exists i.\ (\tau \hat{\ } i)(UNIV :: 'b\ set) = (\tau \hat{\ } (i + (1)))\ UNIV$ **using** *assms*
by (*rule finite-fixp2*)
from *this* **obtain** *i* **where** $(\tau \hat{\ } i)\, UNIV = (\tau \hat{\ } (i + (1)))\ UNIV$ **by** (*erule exE*)
thus $\exists\ n$. *gfp* $\tau = (\tau \hat{\ } n)\, UNIV$ **using** *assms*
by (*metis Suc-eq-plus1 gfp-Kleene-iter*)
qed

1.2 Generic type of state with state transition and CTL operators

The system states and their transition relation are defined as a class called *state* containing an abstract constant*state-transition*. It introduces the syntactic infix notation $I \rightarrow_i I'$ to denote that system state I and I' are in this relation over an arbitrary (polymorphic) type $'a$.

class *state* =
fixes *state-transition* :: [*'a* :: *type*, *'a*] $\Rightarrow$ *bool* (**infixr** $\langle\rightarrow_i\rangle$ *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 structures and Modelchecking

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

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

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

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

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

1.4 Lemmas for CTL operators

1.4.1 EF lemmas

lemma *EF-lem0*: $(x \in EF\ f) = (x \in f \cup EX'\ (lfp\ (\lambda Z :: ('a :: state)\ set.\ f \cup EX'\ Z)))$
proof −
  have $lfp\ (\lambda Z :: ('a :: state)\ set.\ f \cup EX'\ Z) =$
    $f \cup (EX'\ (lfp\ (\lambda Z :: 'a\ set.\ f \cup EX'\ Z)))$
    by (*rule def-lfp-unfold*, *rule reflexive*, *unfold mono-def EX'-def*, *auto*)
  thus $(x \in EF\ (f :: ('a :: state)\ set)) = (x \in f \cup EX'\ (lfp\ (\lambda Z :: ('a :: state)\ set.\ f \cup EX'\ Z)))$
    by (*simp add*: *EF-def*)

qed

lemma *EF-lem00*: $(EF\ f) = (f \cup EX'\ (lfp\ (\lambda\ Z :: ('a :: state)\ set.\ f \cup EX'\ Z)))$
by (*auto simp*: *EF-lem0*)

lemma *EF-lem000*: $(EF\ f) = (f \cup EX'\ (EF\ f))$
by (*metis EF-def EF-lem00*)

lemma *EF-lem1*: $x \in f \vee x \in (EX'\ (EF\ f)) \Longrightarrow x \in EF\ f$
proof (*simp add*: *EF-def*)
assume *a*: $x \in f \vee x \in EX'\ (lfp\ (\lambda Z::'a\ set.\ f \cup EX'\ Z))$
show $x \in lfp\ (\lambda Z::'a\ set.\ f \cup EX'\ Z)$
proof −
have *b*: $lfp\ (\lambda Z :: ('a :: state)\ set.\ f \cup EX'\ Z) =$
$f \cup (EX'\ (lfp\ (\lambda Z :: ('a :: state)\ set.\ f \cup EX'\ Z)))$
using *EF-def EF-lem00* **by** *blast*
thus $x \in lfp\ (\lambda Z::'a\ set.\ f \cup EX'\ Z)$ **using** *a*
by (*subst b*, *blast*)
qed
qed

lemma *EF-lem2b*:
assumes $x \in (EX'\ (EF\ f))$
shows $x \in EF\ f$
by (*simp add*: *EF-lem1 assms*)

lemma *EF-lem2a*: **assumes** $x \in f$ **shows** $x \in EF\ f$
by (*simp add*: *EF-lem1 assms*)

lemma *EF-lem2c*: **assumes** $x \notin f$ **shows** $x \in EF\ (-\ f)$
by (*simp add*: *EF-lem1 assms*)

lemma *EF-lem2d*: **assumes** $x \notin EF\ f$ **shows** $x \notin f$
using *EF-lem1 assms* **by** *auto*

lemma *EF-lem3b*: **assumes** $x \in EX'\ (f \cup EX'\ (EF\ f))$ **shows** $x \in (EF\ f)$
by (*metis EF-lem000 EF-lem2b assms*)

lemma *EX-lem0l*: $x \in (EX'\ f) \Longrightarrow x \in (EX'\ (f \cup g))$
by (*auto simp*: *EX'-def*)

lemma *EX-lem0r*: $x \in (EX'\ g) \Longrightarrow x \in (EX'\ (f \cup g))$
by (*auto simp*: *EX'-def*)

lemma *EX-step*: **assumes** $x\ \rightarrow_i\ y$ **and** $y \in f$ **shows** $x \in EX'\ f$
using *assms* **by** (*auto simp*: *EX'-def*)

lemma *EF-E*[*rule-format*]: $\forall\ f.\ x \in (EF\ f) \longrightarrow x \in (f \cup EX'\ (EF\ f))$
using *EF-lem000* **by** *blast*

lemma *EF-step*: **assumes** $x \ \rightarrow_i \ y$ **and** $y \in f$ **shows** $x \in EF\ f$
using *EF-lem3b EX-step assms* **by** *blast*

lemma *EF-step-step*: **assumes** $x \ \rightarrow_i \ y$ **and** $y \in EF\ f$ **shows** $x \in EF\ f$
using *EF-lem2b EX-step assms* **by** *blast*

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

lemma *EF-induct*: $(a::'a::state) \in EF\ f \Longrightarrow$
$mono\ (\lambda\ Z.\ f \cup EX'\ Z) \Longrightarrow$
$(\bigwedge x.\ x \in ((\lambda\ Z.\ f \cup EX'\ Z)(EF\ f \cap \{x::'a::state.\ P\ x\})) \Longrightarrow P\ x) \Longrightarrow$
$P\ a$
by (*metis* (*mono-tags*, *lifting*) *EF-def def-lfp-induct-set*)

lemma *valEF-E*: $M \vdash EF\ f \Longrightarrow x \in init\ M \Longrightarrow x \in EF\ f$
by (*auto simp*: *check-def*)

lemma *EF-step-star-rev*[*rule-format*]: $x \in EF\ s \Longrightarrow \ (\exists\ y \in s.\ \ x \ \rightarrow_i* \ y)$
proof (*erule EF-induct*)
show *mono* $(\lambda Z::'a\ set.\ s \cup EX'\ Z)$
by (*force simp add*: *mono-def EX'-def*)
next show $\bigwedge x::'a.\ x \in s \cup EX'\ (EF\ s \cap \{x::'a.\ \exists\ y::'a \in s.\ x \rightarrow_i* \ y\}) \Longrightarrow \exists\ y::'a \in s.$
$x \rightarrow_i* \ y$
apply (*erule UnE*)
using *state-transition-refl-def* **apply** *auto*[*1*]
by (*auto simp add*: *EX'-def state-transition-refl-def intro*: *converse-rtrancl-into-rtrancl*)
qed

lemma *EF-step-inv*: $(I \subseteq \{sa::'s :: state.\ (\exists\ i \in I.\ i \rightarrow_i* \ sa) \wedge sa \in EF\ s\})$
$\Longrightarrow \forall\ x \in I.\ \exists\ y \in s.\ x \rightarrow_i* \ y$
using *EF-step-star-rev* **by** *fastforce*

1.4.2 AG lemmas

lemma *AG-in-lem*: $\ x \in AG\ s \Longrightarrow x \in s$
by (*auto simp add*: *AG-def gfp-def*)

lemma *AG-lem1*: $x \in s \wedge x \in (AX\ (AG\ s)) \Longrightarrow x \in AG\ s$
proof (*simp add*: *AG-def*)
have *gfp* $(\lambda Z::'a\ set.\ s \cap AX\ Z) = s \cap (AX\ (gfp\ (\lambda Z::'a\ set.\ s \cap AX\ Z)))$
by (*rule def-gfp-unfold*) (*auto simp*: *mono-def AX-def*)
then show $x \in s \wedge x \in AX\ (gfp\ (\lambda Z::'a\ set.\ s \cap AX\ Z)) \Longrightarrow x \in gfp\ (\lambda Z::'a\ set.\ s \cap AX\ Z)$
by *blast*
qed

lemma *AG-lem2*: $x \in AG\ s \Longrightarrow x \in (s \cap (AX\ (AG\ s)))$
proof −
have *a*: $AG\ s = s \cap (AX\ (AG\ s))$
unfolding *AG-def*
by (*rule def-gfp-unfold*) (*auto simp*: *mono-def AX-def*)
thus $x \in AG\ s \Longrightarrow x \in (s \cap (AX\ (AG\ s)))$
by (*erule subst*)
qed

lemma *AG-lem3*: $AG\ s = (s \cap (AX\ (AG\ s)))$
using *AG-lem1 AG-lem2* **by** *blast*

lemma *AG-step*: $y \rightarrow_i z \Longrightarrow y \in AG\ s \Longrightarrow z \in AG\ s$
using *AG-lem2 AX-def* **by** *blast*

lemma *AG-all-s*: $x \rightarrow_i* y \Longrightarrow x \in AG\ s \Longrightarrow y \in AG\ s$
proof (*simp add*: *state-transition-refl-def*)
show $(x, y) \in \{(x,y).\ x \rightarrow_i y\}^* \Longrightarrow x \in AG\ s \Longrightarrow y \in AG\ s$
by (*erule rtrancl-induct*) (*auto simp add*: *AG-step*)
qed

lemma *AG-imp-notnotEF*:
$I \neq \{\} \Longrightarrow ((Kripke\ \{s.\ \exists\ i \in I.\ (i \rightarrow_i* s)\}\ I \vdash AG\ s)) \Longrightarrow$
$(\neg(Kripke\ \{s.\ \exists\ i \in I.\ (i \rightarrow_i* s)\}\ (I :: ('s :: state)set) \vdash EF\ (-\ s)))$
proof (*rule notI*, *simp add*: *check-def*)
assume *a0*: $I \neq \{\}$ **and**
a1: $I \subseteq \{sa::'s.\ (\exists\, i{\in}I.\ i \rightarrow_i* sa) \wedge sa \in AG\ s\}$ **and**
a2: $I \subseteq \{sa::'s.\ (\exists\, i{\in}I.\ i \rightarrow_i* sa) \wedge sa \in EF\ (-\ s)\}$
show *False*
proof −
have *a3*: $\{sa::'s.\ (\exists\, i{\in}I.\ i \rightarrow_i* sa) \wedge sa \in AG\ s\} \cap$
$\{sa::'s.\ (\exists\, i{\in}I.\ i \rightarrow_i* sa) \wedge sa \in EF\ (-\ s)\} = \{\}$
proof −
have $(?\ x.\ x \in \{sa::'s.\ (\exists\, i{\in}I.\ i \rightarrow_i* sa) \wedge sa \in AG\ s\} \wedge$
$x \in \{sa::'s.\ (\exists\, i{\in}I.\ i \rightarrow_i* sa) \wedge sa \in EF\ (-\ s)\}) \Longrightarrow False$
proof −
assume *a4*: $(?\ x.\ x \in \{sa::'s.\ (\exists\, i{\in}I.\ i \rightarrow_i* sa) \wedge sa \in AG\ s\} \wedge$
$x \in \{sa::'s.\ (\exists\, i{\in}I.\ i \rightarrow_i* sa) \wedge sa \in EF\ (-\ s)\})$
from *a4* **obtain** *x* **where** *a5*: $x \in \{sa::'s.\ (\exists\, i{\in}I.\ i \rightarrow_i* sa) \wedge sa \in AG\ s\} \wedge$

$x \in \{sa::'s.\ (\exists\, i{\in}I.\ i \rightarrow_i{*}\ sa) \wedge sa \in EF\ (-\ s)\}$
by (*erule exE*)
thus *False*
by (*metis* (*mono-tags*, *lifting*) *AG-all-s AG-in-lem ComplD EF-step-star-rev a5 mem-Collect-eq*)
qed
thus $\{sa::'s.\ (\exists\, i{\in}I.\ i \rightarrow_i{*}\ sa) \wedge sa \in AG\ s\} \cap$
$\{sa::'s.\ (\exists\, i{\in}I.\ i \rightarrow_i{*}\ sa) \wedge sa \in EF\ (-\ s)\} = \{\}$
by *blast*
qed
moreover have *b*: *? x*. $x : I$ **using** *a0*
by *blast*
moreover obtain x **where** $x \in I$
using *b* **by** *blast*
ultimately show *False* **using** *a0 a1 a2*
by *blast*
qed
qed

A simplified way of Modelchecking is given by the following lemma.

lemma *check2-def*: $(Kripke\ S\ I \vdash f) = (I \subseteq S \cap f)$
by (*auto simp add*: *check-def*)

end

2 Attack Trees

theory *AT*
imports *MC*
begin

Attack Trees are an intuitive and practical formal method to analyse and quantify attacks on security and privacy. They are very useful to identify the steps an attacker takes through a system when approaching the attack goal. Here, we provide a proof calculus to analyse concrete attacks using a notion of attack validity. We define a state based semantics with Kripke models and the temporal logic CTL in the proof assistant Isabelle [6] using its Higher Order Logic (HOL). We prove the correctness and completeness (adequacy) of Attack Trees in Isabelle with respect to the model.

2.1 Attack Tree datatype

The following datatype definition *attree* defines attack trees. The simplest case of an attack tree is a base attack. The principal idea is that base attacks are defined by a pair of state sets representing the initial states and the *attack property* – a set of states characterized by the fact that this property holds in them. Attacks can also be combined as the conjunction or disjunction of

other attacks. The operator $\oplus_\vee$ creates or-trees and $\oplus_\wedge$ creates and-trees. And-attack trees $l \oplus_\wedge s$ and or-attack trees $l \oplus_\vee s$ combine lists of attack trees l either conjunctively or disjunctively and consist of a list of sub-attacks – again attack trees.

datatype $('s :: state)\ attree = BaseAttack\ ('s\ set) * ('s\ set)\ (\langle \mathcal{N}_{(\text{-})} \rangle) \mid$
$AndAttack\ ('s\ attree)\ list\ ('s\ set) * ('s\ set)\ (\langle \text{-}\ \oplus_\wedge{}^{(\text{-})} \rangle\ 60) \mid$
$OrAttack\ \ ('s\ attree)\ list\ ('s\ set) * ('s\ set)\ (\langle \text{-}\ \oplus_\vee{}^{(\text{-})} \rangle\ 61)$

primrec $attack :: ('s :: state)\ attree \Rightarrow ('s\ set) * ('s\ set)$
where
$attack\ (BaseAttack\ b) = b|$
$attack\ (AndAttack\ as\ s) = s\ \ |$
$attack\ (OrAttack\ as\ s) = s$

2.2 Attack Tree refinement

When we develop an attack tree, we proceed from an abstract attack, given by an attack goal, by breaking it down into a series of sub-attacks. This proceeding corresponds to a process of *refinement*. Therefore, as part of the attack tree calculus, we provide a notion of attack tree refinement.

The relation *refines-to* "constructs" the attack tree. Here the above defined attack vectors are used to define how nodes in an attack tree can be expanded into more detailed (refined) attack sequences. This process of refinement $\sqsubseteq$ allows to eventually reach a fully detailed attack that can then be proved using $\vdash$.

inductive $refines\text{-}to :: [('s :: state)\ attree, 's\ attree] \Rightarrow bool\ (\langle \text{-} \sqsubseteq \text{-} \rangle\ [40]\ 40)$
where
$refI$: $[\![\ A = ((l\ @\ [\ \mathcal{N}_{(si',si'')}]\ @\ l'') \oplus_\wedge{}^{(si,si''')}\ );$
$A' = (l' \oplus_\wedge{}^{(si',si'')});$
$A'' = (l\ @\ l'\ @\ l'' \oplus_\wedge{}^{(si,si''')})$
$]\!] \Longrightarrow A \sqsubseteq A''|$
$ref\text{-}or$: $[\![\ as \neq []; \forall\ A' \in set(as).\ (A\ \sqsubseteq A') \wedge attack\ A = s\ ]\!] \Longrightarrow A \sqsubseteq (as \oplus_\vee{}^{s}) \mid$
$ref\text{-}trans$: $[\![\ A \sqsubseteq A';\ A' \sqsubseteq A''\ ]\!] \Longrightarrow A \sqsubseteq A''|$
$ref\text{-}refl : A \sqsubseteq A$

2.3 Validity of Attack Trees

A valid attack, intuitively, is one which is fully refined into fine-grained attacks that are feasible in a model. The general model we provide is a Kripke structure, i.e., a set of states and a generic state transition. Thus, feasible steps in the model are single steps of the state transition. We call them valid base attacks. The composition of sequences of valid base attacks into and-attacks yields again valid attacks if the base attacks line up with respect to the states in the state transition. If there are different valid

attacks for the same attack goal starting from the same initial state set, these can be summarized in an or-attack. More precisely, the different cases of the validity predicate are distinguished by pattern matching over the attack tree structure.

- A base attack $\mathcal{N}(s0,s1)$ is valid if from all states in the pre-state set *s0* we can get with a single step of the state transition relation to a state in the post-state set *s1*. Note, that it is sufficient for a post-state to exist for each pre-state. After all, we are aiming to validate attacks, that is, possible attack paths to some state that fulfills the attack property.
- An and-attack $As \oplus_\wedge (s0,s1)$ is a valid attack if either of the following cases holds:
  - empty attack sequence *As*: in this case all pre-states in *s0* must already be attack states in *s1*, i.e., $s0 \subseteq s1$;
  - attack sequence *As* is singleton: in this case, the singleton element attack a in $[a]$, must be a valid attack and it must be an attack with pre-state *s0* and post-state *s1*;
  - otherwise, *As* must be a list matching $a \# l$ for some attack a and tail of attack list l such that a is a valid attack with pre-state identical to the overall pre-state *s0* and the goal of the tail l is *s1* the goal of the overall attack. The pre-state of the attack represented by l is $snd(attack\ a)$ since this is the post-state set of the first step a.
- An or-attack $As \oplus_\vee (s0,s1)$ is a valid attack if either of the following cases holds:
  - the empty attack case is identical to the and-attack above: $s0 \subseteq s1$;
  - attack sequence *As* is singleton: in this case, the singleton element attack a must be a valid attack and its pre-state must include the overall attack pre-state set *s0* (since a is singleton in the or) while the post-state of a needs to be included in the global attack goal *s1*;
  - otherwise, *As* must be a list $a \# l$ for an attack a and a list l of alternative attacks. The pre-states can be just a subset of *s0* (since there are other attacks in l that can cover the rest) and the goal states $snd(attack\ a)$ need to lie all in the overall goal state *set s1*. The other or-attacks in l need to cover only the pre-states $fst\ s - fst(attack\ a)$ (where $-$ is set difference) and have the same goal *snd s*.

The proof calculus is thus completely described by one recursive function.

fun *is-attack-tree* :: $[('s :: state)\ attree] \Rightarrow bool$ $(\langle\vdash\text{-}\rangle\ [40]\ 40)$
where
att-base: $(\vdash \mathcal{N}_s) = (\ (\forall\ x \in (fst\ s).\ (\exists\ y \in (snd\ s).\ x\ \rightarrow_i\ y\ )))\ |$
att-and: $(\vdash(As \oplus_\wedge{}^s)) =$
$(case\ As\ of$
$[] \Rightarrow (fst\ s \subseteq snd\ s)$
$|\ [a] \Rightarrow (\ \vdash a \wedge attack\ a = s)$
$|\ (a\ \#\ l) \Rightarrow ((\ \vdash a) \wedge (fst(attack\ a) = fst\ s) \wedge$
$(\vdash(l \oplus_\wedge{}^{(snd(attack\ a), snd(s))})))\ |$
att-or: $(\vdash(As \oplus_\vee{}^s)) =$
$(case\ As\ of$
$[] \Rightarrow (fst\ s \subseteq snd\ s)$
$|\ [a] \Rightarrow (\ \vdash a \wedge (fst(attack\ a) \supseteq fst\ s) \wedge (snd(attack\ a) \subseteq snd\ s))$
$|\ (a\ \#\ l) \Rightarrow ((\ \vdash a) \wedge fst(attack\ a) \subseteq fst\ s \wedge$
$snd(attack\ a) \subseteq snd\ s \wedge$
$(\ \vdash(l \oplus_\vee{}^{(fst\ s\ -\ fst(attack\ a),\ snd\ s)})))$

Since the definition is constructive, code can be generated directly from it, here into the programming language Scala.

export-code *is-attack-tree* **in** *Scala*

2.4 Lemmas for Attack Tree validity

lemma *att-and-one*: **assumes** $\vdash a$ **and** $attack\ a = s$
shows $\vdash([a] \oplus_\wedge{}^s)$
proof −
show $\vdash([a] \oplus_\wedge{}^s)$ **using** *assms*
by (*subst att-and*, *simp del*: *att-and att-or*)
qed

declare *is-attack-tree.simps*[*simp del*]

lemma *att-and-empty*[*rule-format*] : $\vdash([] \oplus_\wedge{}^{(s',\ s'')}) \longrightarrow s' \subseteq s''$
by (*simp add*: *is-attack-tree.simps*(*2*))

lemma *att-and-empty2*: $\vdash([] \oplus_\wedge{}^{(s,\ s)})$
by (*simp add*: *is-attack-tree.simps*(*2*))

lemma *att-or-empty*[*rule-format*] : $\vdash([] \oplus_\vee{}^{(s',\ s'')}) \longrightarrow s' \subseteq s''$
by (*simp add*: *is-attack-tree.simps*(*3*))

lemma *att-or-empty-back*[*rule-format*]: $s' \subseteq s'' \longrightarrow \vdash([] \oplus_\vee{}^{(s',\ s'')})$
by (*simp add*: *is-attack-tree.simps*(*3*))

lemma *att-or-empty-rev*: **assumes** $\vdash(l \oplus_\vee{}^{(s,\ s')})$ **and** $\neg(s \subseteq s')$ **shows** $l \neq []$
using *assms att-or-empty* **by** *blast*

lemma *att-or-empty2*: $\vdash([] \oplus_\vee{}^{(s,\ s)})$
by (*simp add*: *att-or-empty-back*)

lemma *att-andD1*: $\vdash(x1\ \#\ x2 \oplus_\wedge{}^{s}) \Longrightarrow \vdash x1$
by (*metis* (*no-types*, *lifting*) *is-attack-tree.simps*(2) *list.exhaust list.simps*(4) *list.simps*(5))

lemma *att-and-nonemptyD2*[*rule-format*]:
$(x2 \neq [] \longrightarrow \vdash(x1\ \#\ x2 \oplus_\wedge{}^{s}) \longrightarrow \vdash (x2 \oplus_\wedge{}^{(snd(attack\ x1),snd\ s)}))$
by (*metis* (*no-types*, *lifting*) *is-attack-tree.simps*(2) *list.exhaust list.simps*(5))

lemma *att-andD2* : $\vdash(x1\ \#\ x2 \oplus_\wedge{}^{s}) \Longrightarrow \vdash (x2 \oplus_\wedge{}^{(snd(attack\ x1),snd\ s)})$
by (*metis* (*mono-tags*, *lifting*) *att-and-empty2 att-and-nonemptyD2 is-attack-tree.simps*(2) *list.simps*(4) *list.simps*(5))

lemma *att-and-fst-lem*[*rule-format*]:
$\vdash(x1\ \#\ x2a \oplus_\wedge{}^{x}) \longrightarrow xa \in fst\ (attack\ (x1\ \#\ x2a \oplus_\wedge{}^{x}))$
$\longrightarrow xa \in fst\ (attack\ x1)$
by (*induction x2a*, (*subst att-and*, *simp*)+)

lemma *att-orD1*: $\vdash(x1\ \#\ x2 \oplus_\vee{}^{x}) \Longrightarrow \vdash x1$
by (*case-tac x2*, (*subst* (*asm*) *att-or*, *simp*)+)

lemma *att-or-snd-hd*: $\vdash(a\ \#\ list \oplus_\vee{}^{(aa,\ b)}) \Longrightarrow snd(attack\ a) \subseteq b$
by (*case-tac list*, (*subst* (*asm*) *att-or*, *simp*)+)

lemma *att-or-singleton*[*rule-format*]:
$\vdash([x1] \oplus_\vee{}^{x}) \longrightarrow \vdash([] \oplus_\vee{}^{(fst\ x\ -\ fst\ (attack\ x1),\ snd\ x)})$
by (*subst att-or*, *simp*, *rule impI*, *rule att-or-empty-back*, *blast*)

lemma *att-orD2*[*rule-format*]:
$\vdash(x1\ \#\ x2 \oplus_\vee{}^{x}) \longrightarrow \vdash (x2 \oplus_\vee{}^{(fst\ x\ -\ fst(attack\ x1),\ snd\ x)})$
by (*case-tac x2*, *simp add*: *att-or-singleton*, *simp*, *subst att-or*, *simp*)

lemma *att-or-snd-att*[*rule-format*]: $\forall\ x. \vdash (x2 \oplus_\vee{}^{x}) \longrightarrow (\forall\ a \in (set\ x2).\ snd(attack\ a) \subseteq snd\ x\ )$
proof (*induction x2*)
case *Nil*
then show *?case* **by** (*simp add*: *att-or*)
next
case (*Cons a x2*)
then show *?case* **using** *att-orD2 att-or-snd-hd* **by** *fastforce*
qed

lemma *singleton-or-lem*: $\vdash([x1] \oplus_\vee{}^{x}) \Longrightarrow fst\ x \subseteq fst(attack\ x1)$
by (*subst* (*asm*) *att-or*, *simp*)+

lemma *or-att-fst-sup0*[*rule-format*]: $x2 \neq [] \longrightarrow (\forall\ x.\ (\vdash ((x2 \oplus_\vee{}^{x})::('s :: state)\ attree)) \longrightarrow$
$((\bigcup\ y::'s\ attree \in set\ x2.\ fst\ (attack\ y)) \supseteq fst(x)))$

proof (*induction x2*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons a x2*)
then show *?case* **using** *att-orD2 singleton-or-lem* **by** *fastforce*
qed

lemma *or-att-fst-sup*:
assumes ($\vdash$ ((*x1* # *x2* $\oplus_\vee{}^x$)::('*s* :: *state*) *attree*))
shows ((⋃ *y*::'*s attree*∈ *set* (*x1* # *x2*). *fst* (*attack y*)) ⊇ *fst*(*x*))
by (*rule or-att-fst-sup0*, *simp*, *rule assms*)

The lemma *att-elem-seq* is the main lemma for Correctness. It shows that for a given attack tree x1, for each element in the set of start sets of the first attack, we can reach in zero or more steps a state in the states in which the attack is successful (the final attack state *snd*(*attack x1*)). This proof is a big alternative to an earlier version of the proof with *first-step* etc that mapped first on a sequence of sets of states.

lemma *att-elem-seq*[*rule-format*]: $\vdash$ *x1* ⟶ (∀ *x* ∈ *fst*(*attack x1*).
(∃ *y*. *y* ∈ *snd*(*attack x1*) ∧ *x* $\rightarrow_i*$ *y*))

First attack tree induction

proof (*induction x1*)
case (*BaseAttack x*)
then show *?case*
by (*metis AT.att-base EF-step EF-step-star-rev attack.simps*(*1*))
next
case (*AndAttack x1a x2*)
then show *?case*
apply (*rule-tac x* = *x2* **in** *spec*)
apply (*subgoal-tac* (∀ *x1aa*::'*a attree*.
x1aa ∈ *set x1a* ⟶
$\vdash$*x1aa* ⟶
(∀ *x*::'*a*∈*fst* (*attack x1aa*). ∃ *y*::'*a*. *y* ∈ *snd* (*attack x1aa*) ∧
x $\rightarrow_i*$ *y*)))
apply (*rule mp*)
prefer *2*
apply (*rotate-tac* −*1*)
apply *assumption*

Induction for ∧: the manual instantiation seems tedious but in the ∧ case necessary to get the right induction hypothesis.

proof (*rule-tac list* = *x1a* **in** *list.induct*)

The ∧ induction empty case

show (∀ *x1aa*::'*a attree*.
x1aa ∈ *set* [] ⟶

$\vdash x1aa \longrightarrow (\forall x{::}'a{\in}fst\ (attack\ x1aa).\ \exists y{::}'a.\ y \in snd\ (attack\ x1aa) \wedge x \rightarrow_i{*}\ y)) \longrightarrow$
$(\forall x{::}'a\ set \times\ 'a\ set.$
$\vdash([] \oplus_\wedge{}^x) \longrightarrow$
$(\forall xa{::}'a{\in}fst\ (attack\ ([] \oplus_\wedge{}^x)).\ \exists y{::}'a.\ y \in snd\ (attack\ ([] \oplus_\wedge{}^x)) \wedge xa \rightarrow_i{*}\ y))$
using *att-and-empty state-transition-refl-def* **by** *fastforce*

The $\wedge$ induction case nonempty

next show $\bigwedge(x1a{::}'a\ attree\ list)\ (x2{::}'a\ set \times\ 'a\ set)\ (x1{::}'a\ attree)\ (x2a{::}'a\ attree\ list).$
$(\bigwedge x1aa{::}'a\ attree.$
$(x1aa \in set\ x1a) \Longrightarrow$
$((\vdash x1aa) \longrightarrow (\forall x{::}'a{\in}fst\ (attack\ x1aa).\ \exists y{::}'a.\ y \in snd\ (attack\ x1aa) \wedge x \rightarrow_i{*}\ y))) \Longrightarrow$
$\forall x1aa{::}'a\ attree.$
$(x1aa \in set\ x1a) \longrightarrow$
$(\vdash x1aa) \longrightarrow ((\forall x{::}'a{\in}fst\ (attack\ x1aa).\ \exists y{::}'a.\ y \in snd\ (attack\ x1aa) \wedge x \rightarrow_i{*}\ y)) \Longrightarrow$
$(\forall x1aa{::}'a\ attree.$
$(x1aa \in set\ x2a) \longrightarrow$
$(\vdash x1aa) \longrightarrow (\forall x{::}'a{\in}fst\ (attack\ x1aa).\ \exists y{::}'a.\ y \in snd\ (attack\ x1aa) \wedge x \rightarrow_i{*}\ y)) \longrightarrow$
$(\forall x{::}'a\ set \times\ 'a\ set.$
$(\vdash(x2a \oplus_\wedge{}^x)) \longrightarrow$
$((\forall xa{::}'a{\in}fst\ (attack\ (x2a \oplus_\wedge{}^x)).\ \exists y{::}'a.\ y \in snd\ (attack\ (x2a \oplus_\wedge{}^x)) \wedge xa \rightarrow_i{*}\ y))) \Longrightarrow$
$((\forall x1aa{::}'a\ attree.$
$(x1aa \in set\ (x1\ \#\ x2a)) \longrightarrow$
$(\vdash x1aa) \longrightarrow ((\forall x{::}'a{\in}fst\ (attack\ x1aa).\ \exists y{::}'a.\ y \in snd\ (attack\ x1aa) \wedge x \rightarrow_i{*}\ y))) \longrightarrow$
$(\forall x{::}'a\ set \times\ 'a\ set.$
$(\vdash(x1\ \#\ x2a \oplus_\wedge{}^x)) \longrightarrow$
$(\forall xa{::}'a{\in}fst\ (attack\ (x1\ \#\ x2a \oplus_\wedge{}^x)).$
$(\exists y{::}'a.\ y \in snd\ (attack\ (x1\ \#\ x2a \oplus_\wedge{}^x)) \wedge (xa \rightarrow_i{*}\ y)))))$
apply (*rule impI*, *rule allI*, *rule impI*)

Set free the Induction Hypothesis: this is necessary to provide the grounds for specific instantiations in the step.

apply (*subgoal-tac* $(\forall x{::}'a\ set \times\ 'a\ set.$
$\vdash(x2a \oplus_\wedge{}^x) \longrightarrow$
$(\forall xa{::}'a{\in}fst\ (attack\ (x2a \oplus_\wedge{}^x)).$
$\exists y{::}'a.\ y \in snd\ (attack\ (x2a \oplus_\wedge{}^x)) \wedge xa \rightarrow_i{*}\ y)))$
prefer *2*
apply *simp*

The following induction step for $\wedge$ needs a number of manual instantiations so that the proof is not found automatically. In the subsequent case for $\vee$, sledgehammer finds the proof.

proof −
show $\bigwedge$(*x1a*::'*a attree list*) (*x2*::'*a set* × '*a set*) (*x1*::'*a attree*) (*x2a*::'*a attree list*) *x*::'*a set* × '*a set*.
∀ *x1aa*::'*a attree*.
x1aa ∈ *set* (*x1* # *x2a*) ⟶
⊢*x1aa* ⟶ (∀ *x*::'*a*∈*fst* (*attack x1aa*). ∃ *y*::'*a*. *y* ∈ *snd* (*attack x1aa*) ∧ *x* →$_i$* *y*) ⟹
⊢(*x1* # *x2a* ⊕$_\wedge{}^x$) ⟹
∀ *x*::'*a set* × '*a set*.
⊢(*x2a* ⊕$_\wedge{}^x$) ⟶
(∀ *xa*::'*a*∈*fst* (*attack* (*x2a* ⊕$_\wedge{}^x$)). ∃ *y*::'*a*. *y* ∈ *snd* (*attack* (*x2a* ⊕$_\wedge{}^x$)) ∧ *xa* →$_i$* *y*) ⟹
∀ *xa*::'*a*∈*fst* (*attack* (*x1* # *x2a* ⊕$_\wedge{}^x$)). ∃ *y*::'*a*. *y* ∈ *snd* (*attack* (*x1* # *x2a* ⊕$_\wedge{}^x$)) ∧ *xa* →$_i$* *y*
apply (*rule ballI*)
apply (*rename-tac xa*)

Prepare the steps

apply (*drule-tac x* = (*snd*(*attack x1*), *snd x*) **in** *spec*)
apply (*rotate-tac* −*1*)
apply (*erule impE*)
apply (*erule att-andD2*)

Premise for x1

apply (*drule-tac x* = *x1* **in** *spec*)
apply (*drule mp*)
apply *simp*
apply (*drule mp*)
apply (*erule att-andD1*)

Instantiate first step for xa

apply (*rotate-tac* −*1*)
apply (*drule-tac x* = *xa* **in** *bspec*)
apply (*erule att-and-fst-lem*, *assumption*)
apply (*erule exE*)
apply (*erule conjE*)

Take this y and put it as first into the second part

apply (*drule-tac x* = *y* **in** *bspec*)
apply *simp*
apply (*erule exE*)
apply (*erule conjE*)

Bind the first *xa* →$_i$* *y* and second *y* →$_i$* *ya* together for solution

apply (*rule-tac x* = *ya* **in** *exI*)
apply (*rule conjI*)
apply *simp*
by (*simp add*: *state-transition-refl-def*)

qed
qed *auto*
next
case (*OrAttack x1a x2*)
then show *?case*
proof (*induction x1a arbitrary*: *x2*)
case *Nil*
then show *?case*
by (*metis EF-lem2a EF-step-star-rev att-or-empty attack.simps*(*3*) *subsetD surjective-pairing*)
next
case (*Cons a x1a*)
then show *?case*
by (*smt DiffI att-orD1 att-orD2 att-or-snd-att attack.simps*(*3*) *insert-iff list.set*(*2*) *prod.sel*(*1*) *snd-conv subset-iff*)
qed
qed

lemma *att-elem-seq0*: $\vdash$ *x1* $\Longrightarrow$ ($\forall$ $x \in fst(attack\ x1)$.
($\exists$ y. $y \in snd(attack\ x1) \wedge x \rightarrow_i* y$))
by (*simp add*: *att-elem-seq*)

2.5 Valid refinements

definition *valid-ref* :: [('*s* :: *state*) *attree*, '*s attree*] $\Rightarrow$ *bool* ($\langle$- $\sqsubseteq_V$ -$\rangle$ *50*)
where
$A \sqsubseteq_V A' \equiv$ ($(A \sqsubseteq A') \wedge \vdash A'$)

definition *ref-validity* :: [('*s* :: *state*) *attree*] $\Rightarrow$ *bool* ($\langle\vdash_V$ -$\rangle$ *50*)
where
$\vdash_V A \equiv (\exists A'. (A \sqsubseteq_V A'))$

lemma *ref-valI*: $A \sqsubseteq A' \Longrightarrow \vdash A' \Longrightarrow \vdash_V A$
using *ref-validity-def valid-ref-def* **by** *blast*

3 Correctness and Completeness

This section presents the main theorems of Correctness and Completeness between AT and Kripke, essentially:
$\vdash$ (*init K*, *p*) $\equiv$ $K \vdash EF\ p$.
First, we proof a number of lemmas needed for both directions before we show the Correctness theorem followed by the Completeness theorem.

3.1 Lemma for Correctness and Completeness

lemma *nth-app-eq*[*rule-format*]:

$\forall\ sl\ x.\ sl \neq [] \longrightarrow sl\ !\ (length\ sl - Suc\ (0)) = x$
$\longrightarrow (l\ @\ sl)\ !\ (length\ l + length\ sl - Suc\ (0)) = x$
by (*induction l*) *auto*

lemma *nth-app-eq1*[*rule-format*]: $i < length\ sla \Longrightarrow (sla\ @\ sl)\ !\ i = sla\ !\ i$
by (*simp add*: *nth-append*)

lemma *nth-app-eq1-rev*: $i < length\ sla \Longrightarrow\ sla\ !\ i = (sla\ @\ sl)\ !\ i$
by (*simp add*: *nth-append*)

lemma *nth-app-eq2*[*rule-format*]: $\forall\ sl\ i.\ length\ sla \leq i \wedge i < length\ (sla\ @\ sl)$
$\longrightarrow (sla\ @\ sl)\ !\ i = sl\ !\ (i - (length\ sla))$
by (*simp add*: *nth-append*)

lemma *tl-ne-ex*[*rule-format*]: $l \neq [] \longrightarrow (?\ x\ .\ l = x\ \#\ (tl\ l))$
by (*induction l*, *auto*)

lemma *tl-nempty-lngth*[*rule-format*]: $tl\ sl \neq [] \longrightarrow 2 \leq length(sl)$
using *le-less* **by** *fastforce*

lemma *list-app-one-length*: $length\ l = n \Longrightarrow (l\ @\ [s])\ !\ n = s$
by (*erule subst*, *simp*)

lemma *tl-lem1*[*rule-format*]: $l \neq [] \longrightarrow tl\ l = [] \longrightarrow length\ l = 1$
by (*induction l*, *simp+*)

lemma *nth-tl-length*[*rule-format*]: $tl\ sl \neq [] \longrightarrow$
$tl\ sl\ !\ (length\ (tl\ sl) - Suc\ (0)) = sl\ !\ (length\ sl - Suc\ (0))$
by (*induction sl*, *simp+*)

lemma *nth-tl-length1*[*rule-format*]: $tl\ sl \neq [] \longrightarrow$
$tl\ sl\ !\ n = sl\ !\ (n + 1)$
by (*induction sl*, *simp+*)

lemma *ineq1*: $i < length\ sla - n\ \Longrightarrow$
$(0) \leq n \Longrightarrow i < length\ sla$
by *simp*

lemma *ineq2*[*rule-format*]: $length\ sla \leq i \longrightarrow i + (1) - length\ sla = i - length\ sla + 1$
by *arith*

lemma *ineq3*: $tl\ sl \neq []\ \Longrightarrow length\ sla \leq i \Longrightarrow i < length\ (sla\ @\ tl\ sl) - (1)$
$\Longrightarrow i - length\ sla + (1) < length\ sl - (1)$
by *simp*

lemma *tl-eq1*[*rule-format*]: $sl \neq [] \longrightarrow tl\ sl\ !\ (0) = sl\ !\ Suc\ (0)$

by (*induction sl*, *simp+*)

lemma *tl-eq2*[*rule-format*]: *tl sl* = [] ⟶ *sl* ! (*0*) = *sl* ! (*length sl* − (*1*))
by (*induction sl*, *simp+*)

lemma *tl-eq3*[*rule-format*]: *tl sl* ≠ [] ⟶
tl sl ! (*length sl* − *Suc* (*Suc* (*0*))) = *sl* ! (*length sl* − *Suc* (*0*))
by (*induction sl*, *simp+*)

lemma *nth-app-eq3*: **assumes** *tl sl* ≠ []
shows (*sla* @ *tl sl*) ! (*length* (*sla* @ *tl sl*) − (*1*)) = *sl* ! (*length sl* − (*1*))
using *assms nth-app-eq nth-tl-length* **by** *fastforce*

lemma *not-empty-ex*: *A* ≠ {} ⟹ *?* *x*. *x* ∈ *A*
by *force*

lemma *fst-att-eq*: (*fst x* # *sl*) ! (*0*) = *fst* (*attack* (*al* $\oplus_\wedge{}^{x}$))
by *simp*

lemma *list-eq1*[*rule-format*]: *sl* ≠ [] ⟶
(*fst x* # *sl*) ! (*length* (*fst x* # *sl*) − (*1*)) = *sl* ! (*length sl* − (*1*))
by (*induction sl*, *auto*)

lemma *attack-eq1*: *snd* (*attack* (*x1* # *x2a* $\oplus_\wedge{}^{x}$)) = *snd* (*attack* (*x2a* $\oplus_\wedge{}^{(snd\ (attack\ x1),\ snd\ x)}$))
by *simp*

lemma *fst-lem1*[*rule-format*]: ∀ (*a*:: *'s set*) *b* (*c* :: *'s set*) *d*. (*a*, *c*) = (*b*, *d*) ⟶ *a* = *b*
by *auto*

lemma *fst-eq1*: (*sla* ! (*0*), *y*) = *attack x1* ⟹
sla ! (*0*) = *fst* (*attack x1*)
by (*rule-tac c* = *y* **and** *d* = *snd*(*attack x1*) **in** *fst-lem1*, *simp*)

lemma *base-att-lem1*: *y0* ⊆ *y1* ⟹ ⊢ $\mathcal{N}_{(y1,\ y)}$ ⟹⊢ $\mathcal{N}_{(y0,\ y)}$
by (*simp add*: *att-base*, *blast*)

lemma *ref-pres-att*: *A* ⊑ *A'* ⟹ *attack A* = *attack A'*
proof (*erule refines-to.induct*)
show ⋀(*A*::*'a attree*) (*l*::*'a attree list*) (*si'*::*'a set*) (*si''*::*'a set*) (*l''*::*'a attree list*) (*si*::*'a set*)
(*si'''*::*'a set*) (*A'*::*'a attree*) (*l'*::*'a attree list*) *A''*::*'a attree*.
A = (*l* @ [$\mathcal{N}_{(si',\ si'')}$] @ *l''* $\oplus_\wedge{}^{(si,\ si''')}$) ⟹
A' = (*l'* $\oplus_\wedge{}^{(si',\ si'')}$) ⟹ *A''* = (*l* @ *l'* @ *l''* $\oplus_\wedge{}^{(si,\ si''')}$) ⟹ *attack A* = *attack A''*
by *simp*
next show ⋀(*as*::*'a attree list*) (*A*::*'a attree*) (*s*::*'a set* × *'a set*).
as ≠ [] ⟹

$(\forall A'::'a\ attree\in (set\ as).\ ((A \sqsubseteq A') \wedge (attack\ A = attack\ A')) \wedge attack\ A = s) \Longrightarrow$
$attack\ A = attack\ (as \oplus_\vee {}^s)$
using *last-in-set* **by** *auto*
next show $\bigwedge(A::'a\ attree)\ (A'::'a\ attree)\ A''::'a\ attree.$
$A \sqsubseteq A' \Longrightarrow attack\ A = attack\ A' \Longrightarrow A' \sqsubseteq A'' \Longrightarrow attack\ A' = attack\ A'' \Longrightarrow attack\ A = attack\ A''$
by *simp*
next show $\bigwedge A::'a\ attree.\ attack\ A = attack\ A$ **by** (*rule refl*)
qed

lemma *base-subset*:
assumes $xa \subseteq xc$
shows $\vdash\mathcal{N}_{(x,\ xa)} \Longrightarrow \vdash\mathcal{N}_{(x,\ xc)}$
proof (*simp add*: *att-base*)
show $\forall x::'a\in x.\ \exists xa::'a\in xa.\ x \rightarrow_i xa \Longrightarrow \forall x::'a\in x.\ \exists xa::'a\in xc.\ x \rightarrow_i xa$
by (*meson assms in-mono*)
qed

3.2 Correctness Theorem

Proof with induction over the definition of EF using the main lemma *att-elem-seq0*.
There is also a second version of Correctness for valid refinements.

theorem *AT-EF*: **assumes** $\vdash (A :: ('s :: state)\ attree)$
and $attack\ A = (I,s)$
shows $Kripke\ \{s :: ('s :: state).\ \exists\ i \in I.\ (i \rightarrow_i* s)\}\ (I :: ('s :: state)set)$
$\vdash EF\ s$
proof (*simp add*:*check-def*)
show $I \subseteq \{sa::('s :: state).\ (\exists i::'s\in I.\ i \rightarrow_i* sa) \wedge sa \in EF\ s\}$
proof (*rule subsetI*, *rule CollectI*, *rule conjI*, *simp add*: *state-transition-refl-def*)
show $\bigwedge x::'s.\ x \in I \Longrightarrow \exists i::'s\in I.\ (i, x) \in \{(x::'s, y::'s).\ x \rightarrow_i y\}^*$
by (*rule-tac* $x = x$ **in** *bexI*, *simp*)
next show $\bigwedge x::'s.\ x \in I \Longrightarrow x \in EF\ s$ **using** *assms*
proof −
have *a*: $\forall\ x \in I.\ \exists\ y \in s.\ x \rightarrow_i* y$ **using** *assms*
proof −
have $\forall x::'s\in fst\ (attack\ A).\ \exists y::'s.\ y \in snd\ (attack\ A) \wedge x \rightarrow_i* y$
by (*rule att-elem-seq0*, *rule assms*)
thus $\forall x::'s\in I.\ \exists y::'s\in s.\ x \rightarrow_i* y$ **using** *assms*
by *force*
qed
thus $\bigwedge x::'s.\ x \in I \Longrightarrow x \in EF\ s$
proof −
fix *x*
assume *b*: $x \in I$
have $\exists y::'s\in s::('s :: state)\ set.\ x \rightarrow_i* y$
by (*rule-tac* $x = x$ **and** $A = I$ **in** *bspec*, *rule a*, *rule b*)
from *this* **obtain** *y* **where** $y \in s$ **and** $x \rightarrow_i* y$ **by** (*erule bexE*)

thus $x \in EF\ s$
by (*erule-tac* $f = s$ **in** *EF-step-star*)
qed
qed
qed
qed

theorem *ATV-EF*: $[\![\vdash_V A;\ (I,s) = attack\ A\ ]\!] \Longrightarrow$
(*Kripke* $\{s.\ \exists\ i \in I.\ (i \rightarrow_i* s)\}\ I \vdash EF\ s$)
by (*metis* (*full-types*) *AT-EF ref-pres-att ref-validity-def valid-ref-def*)

3.3 Completeness Theorem

This section contains the completeness direction, informally:
$\vdash EF\ s \Longrightarrow \exists\ A. \vdash A \wedge attack\ A = (I,s)$.
The main theorem is presented last since its proof just summarises a number of main lemmas *Compl-step1*, *Compl-step2*, *Compl-step3*, *Compl-step4* which are presented first together with other auxiliary lemmas.

3.3.1 Lemma *Compl-step1*

lemma *Compl-step1*:
Kripke $\{s :: ('s :: state).\ \exists\ i \in I.\ (i \rightarrow_i* s)\}\ I \vdash EF\ s$
$\Longrightarrow \forall\ x \in I.\ \exists\ y \in s.\ x \rightarrow_i* y$
by (*simp add*: *EF-step-star-rev valEF-E*)

3.3.2 Lemma *Compl-step2*

First, we prove some auxiliary lemmas.

lemma *rtrancl-imp-singleton-seq2*: $x \rightarrow_i* y \Longrightarrow$
$x = y \vee (\exists\ s.\ s \neq [] \wedge (tl\ s \neq []) \wedge s\ !\ 0 = x \wedge s\ !\ (length\ s - 1) = y \wedge$
$(\forall\ i < (length\ s - 1).\ (s\ !\ i) \rightarrow_i (s\ !\ (Suc\ i))))$

unfolding *state-transition-refl-def*
proof (*induction rule*: *rtrancl-induct*)
case *base*
then show *?case*
by *simp*
next
case (*step y z*)
show *?case*
using *step.IH*
proof (*elim disjE exE conjE*)
assume *x=y*
with *step.hyps* **show** *?case*
by (*force intro*!: *exI* [**where** *x*=[*y*,*z*]])
next

show $\bigwedge s.\ [\![s \neq [];\ tl\ s \neq [];\ s\ !\ 0 = x;$
$s\ !\ (length\ s - 1) = y;$
$\forall i{<}length\ s - 1.$
$s\ !\ i \rightarrow_i s\ !\ Suc\ i]\!]$
$\Longrightarrow x = z \vee$
$(\exists s.\ s \neq [] \wedge$
$tl\ s \neq [] \wedge s\ !\ 0 = x \wedge$
$s\ !\ (length\ s - 1) = z \wedge$
$(\forall i{<}length\ s - 1.\ s\ !\ i \rightarrow_i s\ !\ Suc\ i))$
apply (*rule disjI2*)
apply (*rule-tac x*=*s* @ [*z*] **in** *exI*)
apply (*auto simp*: *nth-append*)
by (*metis One-nat-def Suc-lessI diff-Suc-1 mem-Collect-eq old.prod.case step.hyps*(*2*))
qed
qed

lemma *tl-nempty-length*[*rule-format*]: $s \neq [] \longrightarrow tl\ s \neq [] \longrightarrow 0 < length\ s - 1$
by (*induction s*, *simp*+)

lemma *tl-nempty-length2*[*rule-format*]: $s \neq [] \longrightarrow tl\ s \neq [] \longrightarrow Suc\ 0 < length\ s$
by (*induction s*, *simp*+)

lemma *length-last*[*rule-format*]: $(l\ @\ [x])\ !\ (length\ (l\ @\ [x]) - 1) = x$
by (*induction l*, *simp*+)

lemma *Compl-step2*: $\forall\ x \in I.\ \exists\ y \in s.\ x \rightarrow_i{*}\ y \Longrightarrow$
$(\ \forall\ x \in I.\ \ x \in s \vee (\exists\ (sl :: (((\prime s :: state)\ set)list)).$
$(sl \neq []) \wedge (tl\ sl \neq []) \wedge$
$(sl\ !\ 0,\ sl\ !\ (length\ sl - 1)) = (\{x\},s) \wedge$
$(\forall\ i < (length\ sl - 1).\ \vdash \mathcal{N}_{(sl\ !\ i,sl\ !\ (i+1)\ )}$
$)))$
proof (*rule ballI*, *drule-tac x* = *x* **in** *bspec*, *assumption*, *erule bexE*)
fix *x y*
assume *a*: $x \in I$ **and** *b*: $y \in s$ **and** *c*: $x \rightarrow_i{*}\ y$
show $x \in s \vee$
$(\exists sl{::}\prime s\ set\ list.$
$sl \neq [] \wedge$
$tl\ sl \neq [] \wedge$
$(sl\ !\ (0),\ sl\ !\ (length\ sl - (1))) = (\{x\},\ s) \wedge$
$(\forall i{<}length\ sl - (1).\ \vdash \mathcal{N}_{(sl\ !\ i,\ sl\ !\ (i + (1)))}))$
proof −
have $d : x = y \vee$
$(\exists s'.\ s' \neq [] \wedge$
$tl\ s' \neq [] \wedge$
$s'\ !\ (0) = x \wedge$
$s'\ !\ (length\ s' - (1)) = y \wedge (\forall i{<}length\ s' - (1).\ s'\ !\ i \rightarrow_i s'\ !\ Suc\ i))$
using *c rtrancl-imp-singleton-seq2* **by** *blast*
thus $x \in s \vee$

(∃ *sl*::'*s set list*.
sl ≠ [] ∧
tl sl ≠ [] ∧
(*sl* ! (*0*), *sl* ! (*length sl* − (*1*))) = ({*x*}, *s*) ∧
(∀ *i*<*length sl* − (*1*). ⊢$\mathcal{N}_{(sl\ !\ i,\ sl\ !\ (i\ +\ (1)))}$))
apply (*rule disjE*)
using *b* **apply** *blast*
apply (*rule disjI2*, *elim conjE exE*)
apply (*rule-tac x* = [{*s'* ! *j*}. *j* ← [*0*..<(*length s'* − *1*)]] @ [*s*] **in** *exI*)
apply (*auto simp*: *nth-append*)
apply (*metis AT.att-base Suc-lessD fst-conv prod.sel*(*2*) *singletonD singletonI*)
apply (*metis AT.att-base Suc-lessI b fst-conv prod.sel*(*2*) *singletonD*)
using *tl-nempty-length2* **by** *blast*
qed
qed

3.3.3 Lemma *Compl-step3*

First, we need a few lemmas.

lemma *map-hd-lem*[*rule-format*] : *n* > *0* ⟶ (*f 0* # *map* (λ*i*. *f i*) [*1*..<*n*]) = *map* (λ*i*. *f i*) [*0*..<*n*]
by (*simp add* : *hd-map upt-rec*)

lemma *map-Suc-lem*[*rule-format*] : *n* > *0* ⟶ *map* (λ *i*:: *nat*. *f i*)[*1*..<*n*] =
map (λ *i*:: *nat*. *f*(*Suc i*))[*0*..<(*n* − *1*)]
proof −
have (*f 0* # *map* (λ*n*. *f* (*Suc n*)) [*0*..<*n* − *1*] = *f 0* # *map f* [*1*..<*n*]) = (*map* (λ*n*. *f* (*Suc n*)) [*0*..<*n* − *1*] = *map f* [*1*..<*n*])
by *blast*
then show *?thesis*
by (*metis Suc-pred'* *map-hd-lem map-upt-Suc*)
qed

lemma *forall-ex-fun*: *finite S* ⟹ (∀ *x* ∈ *S*. (∃ *y*. *P y x*)) ⟶ (∃ *f*. ∀ *x* ∈ *S*. *P* (*f x*) *x*)
proof (*induction rule*: *finite.induct*)
case *emptyI*
then show *?case*
by *simp*
next
case (*insertI F x*)
then show *?case*
proof (*clarify*)
assume *d*: (∀ *x*::'*a*∈*insert x F*. ∃ *y*::'*b*. *P y x*)
have (∀ *x*::'*a*∈*F*. ∃ *y*::'*b*. *P y x*)
using *d* **by** *blast*
then obtain *f* **where** *f*: ∀ *x*::'*a*∈*F*. *P* (*f x*) *x*
using *insertI.IH* **by** *blast*

from d **obtain** y **where** $P\ y\ x$ **by** *blast*
thus $(\exists f{::}'a \Rightarrow 'b.\ \forall x{::}'a{\in}insert\ x\ F.\ P\ (f\ x)\ x)$ **using** f
by (*rule-tac* $x = \lambda\ z.\ if\ z = x\ then\ y\ else\ f\ z$ **in** *exI*, *simp*)
qed
qed

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 *nodup-all*:: '*a list* ⇒ *bool*
where
nodup-all l ≡ ∀ *x* ∈ *set l*. *nodup x l*

lemma *nodup-all-lem*[*rule-format*]:
nodup-all (*x1* # *a* # *l*) ⟶ (*insert x1* (*insert a* (*set l*)) − {*x1*}) = *insert a* (*set l*)
by (*induction l*, (*simp add*: *nodup-all-def*)+)

lemma *nodup-all-tl*[*rule-format*]: *nodup-all* (*x* # *l*) ⟶ *nodup-all l*
by (*induction l*, (*simp add*: *nodup-all-def*)+)

lemma *finite-nodup*: *finite I* ⟹ ∃ *l*. *set l* = *I* ∧ *nodup-all l*
proof (*induction rule*: *finite.induct*)
case *emptyI*
then show *?case*
by (*simp add*: *nodup-all-def*)
next
case (*insertI A a*)
then show *?case*
by (*metis insertE insert-absorb list.simps*(*15*) *nodup-all-def nodup-step*)
qed

lemma *Compl-step3*: *I* ≠ {} ⟹ *finite I* ⟹
(∀ *x* ∈ *I*. *x* ∈ *s* ∨ (∃ (*sl* :: (((*'s* :: *state*) *set*)*list*)).
(*sl* ≠ []) ∧ (*tl sl* ≠ []) ∧
(*sl* ! *0*, *sl* ! (*length sl* − *1*)) = ({*x*},*s*) ∧
(∀ *i* < (*length sl* − *1*). ⊢ $\mathcal{N}_{(sl\ !\ i, sl\ !\ (i+1))}$
)) ⟹
(∃ *lI*. *set lI* = {*x* :: *'s* :: *state*. *x* ∈ *I* ∧ *x* ∉ *s*} ∧ (∃ *Sj* :: (((*'s* :: *state*) *set*)*list*) *list*.
length Sj = *length lI* ∧ *nodup-all lI* ∧
(∀ *j* < *length Sj*. (((*Sj* ! *j*) ≠ []) ∧ (*tl* (*Sj* ! *j*) ≠ []) ∧
((*Sj* ! *j*) ! *0*, (*Sj* ! *j*) ! (*length* (*Sj* ! *j*) − *1*)) = ({*lI* ! *j*},*s*) ∧
(∀ *i* < (*length* (*Sj* ! *j*) − *1*). ⊢ $\mathcal{N}_{((Sj\ !\ j)\ !\ i, (Sj\ !\ j)\ !\ (i+1))}$
))))))
proof −
assume *i*: *I* ≠ {} **and** *f*: *finite I* **and**

```
    fa: ∀ x::'s∈I.
     x ∈ s ∨
     (∃ sl::'s set list.
         sl ≠ [] ∧
         tl sl ≠ [] ∧
         (sl ! (0), sl ! (length sl − (1))) = ({x}, s) ∧
         (∀ i<length sl − (1). ⊢𝒩(sl ! i, sl ! (i + (1)))))
have a: ∃ lI. set lI = {x::'s ∈ I. x ∉ s} ∧ nodup-all lI
  by (simp add: f finite-nodup)
from this obtain lI where b: set lI = {x::'s ∈ I. x ∉ s} ∧ nodup-all lI
  by (erule exE)
thus ∃ lI::'s list.
     set lI = {x::'s ∈ I. x ∉ s} ∧
     (∃ Sj::'s set list list.
         length Sj = length lI ∧
         nodup-all lI ∧
         (∀ j<length Sj.
             Sj ! j ≠ [] ∧
             tl (Sj ! j) ≠ [] ∧
             (Sj ! j ! (0), Sj ! j ! (length (Sj ! j) − (1))) = ({lI ! j}, s) ∧
             (∀ i<length (Sj ! j) − (1). ⊢𝒩(Sj ! j ! i, Sj ! j ! (i + (1)))))))
  apply (rule-tac x = lI in exI)
  apply (rule conjI)
   apply (erule conjE, assumption)
proof −
  have c: ∀ x ∈ set(lI). (∃ sl::'s set list.
            sl ≠ [] ∧
            tl sl ≠ [] ∧
            (sl ! (0), sl ! (length sl − (1))) = ({x}, s) ∧
            (∀ i<length sl − (1). ⊢𝒩(sl ! i, sl ! (i + (1)))))
    using b fa by fastforce
  thus ∃ Sj::'s set list list.
     length Sj = length lI ∧
     nodup-all lI ∧
     (∀ j<length Sj.
         Sj ! j ≠ [] ∧
         tl (Sj ! j) ≠ [] ∧
         (Sj ! j ! (0), Sj ! j ! (length (Sj ! j) − (1))) = ({lI ! j}, s) ∧
         (∀ i<length (Sj ! j) − (1). ⊢𝒩(Sj ! j ! i, Sj ! j ! (i + (1)))))
    apply (subgoal-tac finite (set lI))
     apply (rotate-tac −1)
     apply (drule forall-ex-fun)
     apply (drule mp)
      apply assumption
     apply (erule exE)
     apply (rule-tac x = [f (lI ! j). j ← [0..<(length lI)]] in exI)
     apply simp
     apply (insert b)
     apply (erule conjE, assumption)
```

apply (*rule-tac A = set lI* **and** *B = I* **in** *finite-subset*)
apply *blast*
by (*rule f*)
qed
qed

3.3.4 Lemma *Compl-step4*

Again, we need some additional lemmas first.

lemma *list-one-tl-empty*[*rule-format*]: *length l = Suc (0 :: nat) ⟶ tl l = []*
by (*induction l*, *simp+*)

lemma *list-two-tl-not-empty*[*rule-format*]: *∀ list. length l = Suc (Suc (length list)) ⟶ tl l ≠ []*
by (*induction l*, *simp+*, *force*)

lemma *or-empty*: *⊢([] ⊕∨({}, s))* **by** (*simp add*: *att-or*)

Note, this is not valid for any l, i.e., *⊢ l ⊕∨({}, s)* is not a theorem.

lemma *list-or-upt*[*rule-format*]:
∀ l . lI ≠ [] ⟶ length l = length lI ⟶ nodup-all lI ⟶
(∀ i < length lI. (⊢ (l ! i)) ∧ (attack (l ! i) = ({lI ! i}, s)))
⟶ (⊢ (l ⊕∨(set lI, s)))
proof (*induction lI*, *simp*, *clarify*)
fix *x1 x2 l*
show *∀ l::'a attree list.*
x2 ≠ [] ⟶
length l = length x2 ⟶
nodup-all x2 ⟶
(∀ i<length x2. ⊢(l ! i) ∧ attack (l ! i) = ({x2 ! i}, s)) ⟶ ⊢(l ⊕∨(set x2, s))
⟹
x1 # x2 ≠ [] ⟹
length l = length (x1 # x2) ⟹
nodup-all (x1 # x2) ⟹
∀ i<length (x1 # x2). ⊢(l ! i) ∧ attack (l ! i) = ({(x1 # x2) ! i}, s) ⟹ ⊢(l ⊕∨(set (x1 # x2), s))
apply (*case-tac x2*, *simp*, *subst att-or*, *case-tac l*, *simp+*)

Case *∀ i<Suc (Suc (length list)). ⊢l ! i ∧ attack (l ! i) = ({(x1 # a # list) ! i}, s) ⟹ x2 = a # list ⟹ ⊢l ⊕∨(insert x1 (insert a (set list)), s)*

apply (*subst att-or*, *case-tac l*, *simp*, *clarify*, *simp*, *rename-tac lista*, *case-tac lista*, *simp+*)

Remaining conjunct of three conditions: *⊢aa ∧ fst (attack aa) ⊆ insert x1 (insert a (set list)) ∧ snd (attack aa) ⊆ s ∧ ⊢ab # listb ⊕∨(insert x1 (insert a (set list)) − fst (attack aa), s)*

apply (*rule conjI*)

First condition *⊢aa*

apply (*drule-tac x = 0* **in** *spec*, *drule mp*, *simp*, (*erule conjE*)+, *simp*, *rule conjI*)

Second condition $fst\ (attack\ aa) \subseteq insert\ x1\ (insert\ a\ (set\ list))$

apply (*drule-tac x = 0* **in** *spec*, *drule mp*, *simp*, *erule conjE*, *simp*)

The remaining conditions
$snd\ (attack\ aa) \subseteq s \wedge \vdash ab\ \#\ listb \oplus_{\vee}^{(insert\ x1\ (insert\ a\ (set\ list))\ -\ fst\ (attack\ aa),\ s)}$
are solved automatically!

by (*metis Suc-mono add.right-neutral add-Suc-right list.size*(*4*) *nodup-all-lem nodup-all-tl nth-Cons-0 nth-Cons-Suc order-refl prod.sel*(*1*) *prod.sel*(*2*) *zero-less-Suc*)
qed

lemma *app-tl-empty-hd*[*rule-format*]: $tl\ (l\ @\ [a]) = [] \longrightarrow hd\ (l\ @\ [a]) = a$
by (*induction l*) *auto*

lemma *tl-hd-empty*[*rule-format*]: $tl\ (l\ @\ [a]) = [] \longrightarrow l = []$
by (*induction l*) *auto*

lemma *tl-hd-not-empty*[*rule-format*]: $tl\ (l\ @\ [a]) \neq [] \longrightarrow l \neq []$
by (*induction l*) *auto*

lemma *app-tl-empty-length*[*rule-format*]: $tl\ (map\ f\ [0..<length\ l]\ @\ [a]) = [] \Longrightarrow l = []$
by (*drule tl-hd-empty*, *simp*)

lemma *not-empty-hd-fst*[*rule-format*]: $l \neq [] \longrightarrow hd(l\ @\ [a]) = l\ !\ 0$
by (*induction l*) *auto*

lemma *app-tl-hd-list*[*rule-format*]: $tl\ (map\ f\ [0..<length\ l]\ @\ [a]) \neq [] \Longrightarrow hd(map\ f\ [0..<length\ l]\ @\ [a]) = (map\ f\ [0..<length\ l])\ !\ 0$
by (*drule tl-hd-not-empty*, *erule not-empty-hd-fst*)

lemma *tl-app-in*[*rule-format*]: $l \neq [] \longrightarrow map\ f\ [0..<(length\ l - (Suc\ 0 :: nat))]\ @\ [f(length\ l - (Suc\ 0 :: nat))] = map\ f\ [0..<length\ l]$
by (*induction l*) *auto*

lemma *map-fst*[*rule-format*]: $n > 0 \longrightarrow map\ f\ [0..<n] = f\ 0\ \#\ (map\ f\ [1..<n])$
by (*induction n*) *auto*

lemma *step-lem*[*rule-format*]: $l \neq [] \Longrightarrow$
$tl\ (map\ (\lambda\ i.\ f((x1\ \#\ a\ \#\ l)\ !\ i)((a\ \#\ l)\ !\ i))\ [0..<length\ l]) =$
$map\ (\lambda i.\ f((a\ \#\ l)\ !\ i)(l\ !\ i))\ [0..<length\ l - (1)]$
proof (*simp*)
assume *l*: $l \neq []$
have *a*: $map\ (\lambda i.\ f\ ((x1\ \#\ a\ \#\ l)\ !\ i)\ ((a\ \#\ l)\ !\ i))\ [0..<length\ l] =$

```
            (f(x1)(a) # (map (λi. f ((a # l) ! i) (l ! i)) [0..<(length l − 1)]))
  proof −
    have b : map (λi. f ((x1 # a # l) ! i) ((a # l) ! i)) [0..<length l] =
                 f ((x1 # a # l) ! 0) ((a # l) ! 0) #
                 (map (λi. f ((x1 # a # l) ! i) ((a # l) ! i)) [1..<length l])
      by (rule map-fst, simp, rule l)
    have c: map (λi. f ((x1 # a # l) ! i) ((a # l) ! i)) [Suc (0)..<length l] =
            map (λi. f ((x1 # a # l) ! Suc i) ((a # l) ! Suc i)) [(0)..<(length l
− 1)]
      by (subgoal-tac [Suc (0)..<length l] = map Suc [0..<(length l − 1)],
          simp, simp add: map-Suc-upt l)
    thus map (λi. f ((x1 # a # l) ! i) ((a # l) ! i)) [0..<length l] =
          f x1 a # map (λi. f ((a # l) ! i) (l ! i)) [0..<length l − (1)]
      by (simp add: b c)
  qed
  thus l ≠ [] ⟹
  tl (map (λi. f ((x1 # a # l) ! i) ((a # l) ! i)) [0..<length l]) =
  map (λi. f ((a # l) ! i) (l ! i)) [0..<length l − Suc (0)]
    by (subst a, simp)
qed

lemma step-lem2a[rule-format]: 0 < length list ⟹ map (λi. N_((x1 # a # list) ! i, (a # list) ! i))
      [0..<length list] @
      [N_((x1 # a # list) ! length list, (a # list) ! length list)] =
      aa # listb ⟶ N_((x1, a)) = aa
 by (subst map-fst, assumption, simp)

lemma step-lem2b[rule-format]: 0 = length list ⟹ map (λi. N_((x1 # a # list) ! i, (a # list) ! i))
      [0..<length list] @
      [N_((x1 # a # list) ! length list, (a # list) ! length list)] =
      aa # listb ⟶ N_((x1, a)) = aa
by simp

lemma step-lem2: map (λi. N_((x1 # a # list) ! i, (a # list) ! i))
      [0..<length list] @
      [N_((x1 # a # list) ! length list, (a # list) ! length list)] =
      aa # listb ⟹ N_((x1, a)) = aa
proof (case-tac length list, rule step-lem2b, erule sym, assumption)
 show ⋀nat.
      map (λi. N_((x1 # a # list) ! i, (a # list) ! i)) [0..<length list] @
      [N_((x1 # a # list) ! length list, (a # list) ! length list)] =
      aa # listb ⟹
      length list = Suc nat ⟹ N_(x1, a) = aa
   by (rule-tac list = list in step-lem2a, simp)
qed

lemma base-list-and[rule-format]: Sji ≠ [] ⟶ tl Sji ≠ [] ⟶
```

$(\forall\ li.\ \ Sji\ !\ (0) = li \longrightarrow$
$Sji!\ (length\ (Sji) - 1) = s \longrightarrow$
$(\forall\, i{<}length\ (Sji) - 1.\ \vdash \mathcal{N}_{(Sji\ !\ i,\ Sji\ !\ Suc\ i)}) \longrightarrow$
$\vdash (map\ (\lambda i.\ \mathcal{N}_{(Sji\ !\ i,\ Sji\ !\ Suc\ i)})$
$[0..{<}length\ (Sji) - Suc\ (0)]\ \oplus_{\wedge}{}^{(li,\ s)}))$

proof (*induction Sji*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons a Sji*)
then show *?case*
apply (*subst att-and*, *case-tac Sji*, *simp*, *simp*)
apply (*rule impI*)+
proof −
fix *aa list*
show *list* ≠ [] ⟶
list ! (*length list* − *Suc 0*) = *s* ⟶
(∀ *i*<*length list*. ⊢$\mathcal{N}_{((aa\ \#\ list)\ !\ i,\ list\ !\ i)}$) ⟶
⊢(*map* (λ*i*. $\mathcal{N}_{((aa\ \#\ list)\ !\ i,\ list\ !\ i)}$) [0..<*length list*] $\oplus_{\wedge}{}^{(aa,\ s)}$) ⟹
Sji = *aa* # *list* ⟹
(*aa* # *list*) ! *length list* = *s* ⟹
∀ *i*<*Suc* (*length list*). ⊢$\mathcal{N}_{((a\ \#\ aa\ \#\ list)\ !\ i,\ (aa\ \#\ list)\ !\ i)}$ ⟹
case map (λ*i*. $\mathcal{N}_{((a\ \#\ aa\ \#\ list)\ !\ i,\ (aa\ \#\ list)\ !\ i)}$) [0..<*length list*] @
[$\mathcal{N}_{((a\ \#\ aa\ \#\ list)\ !\ length\ list,\ s)}$] *of*
[] ⇒ *fst* (*a*, *s*) ⊆ *snd* (*a*, *s*) | [*aa*] ⇒ ⊢*aa* ∧ *attack aa* = (*a*, *s*)
| *aa* # *ab* # *list* ⇒
⊢*aa* ∧ *fst* (*attack aa*) = *fst* (*a*, *s*) ∧ ⊢(*ab* # *list* $\oplus_{\wedge}{}^{(snd\ (attack\ aa),\ snd\ (a,\ s))}$)
proof (*case-tac map* (λ*i*. $\mathcal{N}_{((a\ \#\ aa\ \#\ list)\ !\ i,\ (aa\ \#\ list)\ !\ i)}$) [0..<*length list*]
@
[$\mathcal{N}_{((a\ \#\ aa\ \#\ list)\ !\ length\ list,\ s)}$], *simp*, *clarify*, *simp*)
fix *ab lista*
have ∗: *tl* (*map* (λ*i*. $\mathcal{N}_{((a\ \#\ aa\ \#\ list)\ !\ i,\ (aa\ \#\ list)\ !\ i)}$) [0..<*length list*])
= (*map* (λ*i*. $\mathcal{N}_{((aa\ \#\ list)\ !\ i,\ (list)\ !\ i)}$) [0..<(*length list* − *1*)])
if *list* ≠ []
apply (*subgoal-tac tl* (*map* (λ*i*. $\mathcal{N}_{((a\ \#\ aa\ \#\ list)\ !\ i,\ (aa\ \#\ list)\ !\ i)}$) [0..<*length list*])
= (*map* (λ*i*. $\mathcal{N}_{((aa\ \#\ list)\ !\ i,\ (list)\ !\ i)}$) [0..<(*length list* − *1*)]))
apply *blast*
apply (*subst step-lem* [*OF that*])
apply *simp*
done
show *list* ≠ [] ⟶
(∀ *i*<*length list*. ⊢$\mathcal{N}_{((aa\ \#\ list)\ !\ i,\ list\ !\ i)}$) ⟶
⊢(*map* (λ*i*. $\mathcal{N}_{((aa\ \#\ list)\ !\ i,\ list\ !\ i)}$)
[0..<*length list*] $\oplus_{\wedge}{}^{(aa,\ list\ !\ (length\ list - Suc\ 0))}$) ⟹
Sji = *aa* # *list* ⟹

$\forall\, i{<}Suc\ (length\ list).\ \vdash \mathcal{N}_{((a\ \#\ aa\ \#\ list)\ !\ i,\ (aa\ \#\ list)\ !\ i)} \Longrightarrow$
$map\ (\lambda i.\ \mathcal{N}_{((a\ \#\ aa\ \#\ list)\ !\ i,\ (aa\ \#\ list)\ !\ i)})\ [0..{<}length\ list]\ @$
$[\mathcal{N}_{((a\ \#\ aa\ \#\ list)\ !\ length\ list,\ (aa\ \#\ list)\ !\ length\ list)}] =$
$ab\ \#\ lista \Longrightarrow$
$s = (aa\ \#\ list)\ !\ length\ list \Longrightarrow$
$case\ lista\ of\ [] \Rightarrow \vdash ab \wedge attack\ ab = (a,\ (aa\ \#\ list)\ !\ length\ list)$
$|\ aba\ \#\ lista \Rightarrow$
$\vdash ab \wedge fst\ (attack\ ab) = a \wedge \vdash (aba\ \#\ lista \oplus_{\wedge}{}^{(snd\ (attack\ ab),\ (aa\ \#\ list)\ !\ length\ list)})$
apply (*auto simp*: *split*: *list.split*)
apply (*metis* (*no-types*, *lifting*) *app-tl-hd-list length-greater-0-conv list.sel*(*1*) *list.sel*(*3*) *list.simps*(*3*) *list.simps*(*8*) *list.size*(*3*) *map-fst nth-Cons-0 self-append-conv2 upt-0 zero-less-Suc*)
apply (*metis* (*no-types*, *lifting*) *app-tl-hd-list attack.simps*(*1*) *fst-conv length-greater-0-conv list.sel*(*1*) *list.sel*(*3*) *list.simps*(*3*) *list.simps*(*8*) *list.size*(*3*) *map-fst nth-Cons-0 self-append-conv2 upt-0*)
apply (*metis* (*mono-tags*, *lifting*) *app-tl-hd-list attack.simps*(*1*) *fst-conv length-greater-0-conv list.sel*(*1*) *list.sel*(*3*) *list.simps*(*3*) *list.simps*(*8*) *list.size*(*3*) *map-fst nth-Cons-0 self-append-conv2 upt-0*)
by (*smt* ∗ *One-nat-def app-tl-hd-list attack.simps*(*1*) *length-greater-0-conv list.sel*(*1*) *list.sel*(*3*) *list.simps*(*3*) *list.simps*(*8*) *list.size*(*3*) *map-fst nth-Cons-0 nth-Cons-pos self-append-conv2 snd-conv tl-app-in tl-append2 upt-0*)
qed
qed
qed

lemma *Compl-step4*: $I \neq \{\} \Longrightarrow$ *finite* $I \Longrightarrow \neg\ I \subseteq s \Longrightarrow$
$(\exists\ lI.\ set\ lI = \{x.\ x \in I \wedge x \notin s\} \wedge (\exists\ Sj :: (((' s :: state)\ set)list)\ list.$
$length\ Sj = length\ lI \wedge nodup\text{-}all\ lI \wedge$
$(\forall\ j < length\ Sj.\ (((Sj\ !\ j)\ \neq []) \wedge (tl\ (Sj\ !\ j) \neq []) \wedge$
$((Sj\ !\ j)\ !\ 0,\ (Sj\ !\ j)\ !\ (length\ (Sj\ !\ j) - 1)) = (\{lI\ !\ j\},s) \wedge$
$(\forall\ i < (length\ (Sj\ !\ j) - 1).\ \vdash \mathcal{N}_{((Sj\ !\ j)\ !\ i,\ (Sj\ !\ j)\ !\ (i{+}1))}$
$)))))$
$\Longrightarrow\ \exists\ (A :: (' s :: state)\ attree).\ \vdash A \wedge attack\ A = (I,s)$
proof (*erule exE*, *erule conjE*, *erule exE*, *erule conjE*)
fix *lI Sj*
assume *a*: $I \neq \{\}$ **and** *b*: *finite* I **and** *c*: $\neg\ I \subseteq s$
and *d*: *set lI* $= \{x{::}'s \in I.\ x \notin s\}$ **and** *e*: *length Sj* = *length lI*
and *f*: *nodup-all lI* $\wedge$
$(\forall j{<}length\ Sj.\ Sj\ !\ j \neq [] \wedge$
$tl\ (Sj\ !\ j) \neq [] \wedge$
$(Sj\ !\ j\ !\ (0),\ Sj\ !\ j\ !\ (length\ (Sj\ !\ j) - (1))) = (\{lI\ !\ j\},\ s) \wedge$
$(\forall i{<}length\ (Sj\ !\ j) - (1).\ \vdash \mathcal{N}_{(Sj\ !\ j\ !\ i,\ Sj\ !\ j\ !\ (i + (1)))}))$
show $\exists A{::}'s\ attree.\ \vdash A \wedge attack\ A = (I,\ s)$
apply (*rule-tac* $x =$
$[([] \oplus_{\vee}{}^{(\{x.\ x \in I \wedge x \in s\},\ s)}),$
$([[\mathcal{N}_{((Sj\ !\ j)\ !\ i,\ (Sj\ !\ j)\ !\ (i + (1)))}\cdot$
$i \leftarrow [0..{<}(length\ (Sj\ !\ j){-}(1))]] \oplus_{\wedge}{}^{((\{lI\ !\ j\},s))}.\ j \leftarrow [0..{<}(length\ Sj)]]$
$\oplus_{\vee}{}^{(\{x.\ x \in I \wedge x \notin s\},s)})] \oplus_{\vee}{}^{(I,\ s)}$ **in** *exI*)

proof
show $\vdash([[]\ \oplus_\vee(\{x::'s \in I.\ x \in s\},\ s)$,
map ($\lambda j.$
((*map* ($\lambda i.\ \mathcal{N}_{(Sj\ !\ j\ !\ i,\ Sj\ !\ j\ !\ (i\ +\ (1)))}$)
[0..<*length* (*Sj* ! *j*) − (1)]) $\oplus_\wedge(\{lI\ !\ j\},\ s)$))
[0..<*length Sj*] $\oplus_\vee(\{x::'s \in I.\ x \notin s\},\ s)$] $\oplus_\vee(I,\ s)$)
proof −
have *g*: $I - \{x::'s \in I.\ x \in s\} = \{x::'s \in I.\ x \notin s\}$ **by** *blast*
thus $\vdash([[]\ \oplus_\vee(\{x::'s \in I.\ x \in s\},\ s)$,
(*map* ($\lambda j.$
((*map* ($\lambda i.\ \mathcal{N}_{(Sj\ !\ j\ !\ i,\ Sj\ !\ j\ !\ (i\ +\ (1)))}$)
[0..<*length* (*Sj* ! *j*) − (1)]) $\oplus_\wedge(\{lI\ !\ j\},\ s)$))
[0..<*length Sj*]) $\oplus_\vee(\{x::'s \in I.\ x \notin s\},\ s)$] $\oplus_\vee(I,\ s)$)
apply (*subst att-or*, *simp*)
proof
show $I - \{x \in I.\ x \in s\} = \{x \in I.\ x \notin s\} \Longrightarrow \vdash([]\ \oplus_\vee(\{x \in I.\ x \in s\},\ s)$)
by (*metis* (*no-types*, *lifting*) *CollectD att-or-empty-back subsetI*)
next show $I - \{x \in I.\ x \in s\} = \{x \in I.\ x \notin s\} \Longrightarrow$
$\vdash([map\ (\lambda j.\ ((map\ (\lambda i.\ \mathcal{N}_{(Sj\ !\ j\ !\ i,\ Sj\ !\ j\ !\ Suc\ i)})\ [0..<length\ (Sj\ !\ j) - Suc\ 0])$
$\oplus_\wedge(\{lI\ !\ j\},\ s)))$
[0..<*length Sj*] $\oplus_\vee(\{x \in I.\ x \notin s\},\ s)$] $\oplus_\vee(\{x \in I.\ x \notin s\},\ s)$)

Use lemma *list-or-upt* to distribute attack validity over list lI

proof (*erule ssubst*, *subst att-or*, *simp*, *rule subst*, *rule d*, *rule-tac lI* = *lI* **in** *list-or-upt*)
show $lI \neq []$
using *c d* **by** *auto*
next show $\bigwedge i.$
$i <$ *length lI* $\Longrightarrow$
$\vdash$(*map* ($\lambda j.$
((*map* ($\lambda i.\ \mathcal{N}_{(Sj\ !\ j\ !\ i,\ Sj\ !\ j\ !\ Suc\ i)}$)
[0..<*length* (*Sj* ! *j*) − *Suc* (0)]) $\oplus_\wedge(\{lI\ !\ j\},\ s)$))
[0..<*length Sj*] !
i) ∧
(*attack*
(*map* ($\lambda j.$
((*map* ($\lambda i.\ \mathcal{N}_{(Sj\ !\ j\ !\ i,\ Sj\ !\ j\ !\ Suc\ i)}$)
[0..<*length* (*Sj* ! *j*) − *Suc* (0)]) $\oplus_\wedge(\{lI\ !\ j\},\ s)$))
[0..<*length Sj*] !
i) =
($\{lI\ !\ i\},\ s$))
proof (*simp add*: *a b c d e f*)
show $\bigwedge i.$
$i <$ *length lI* $\Longrightarrow$
$\vdash$(*map* ($\lambda ia.\ \mathcal{N}_{(Sj\ !\ i\ !\ ia,\ Sj\ !\ i\ !\ Suc\ ia)}$)
[0..<*length* (*Sj* ! *i*) − *Suc* (0)] $\oplus_\wedge(\{lI\ !\ i\},\ s)$)

proof −
fix i :: *nat*
assume *a1*: $i <$ *length lI*
have $\forall n. \vdash$*map* (λ*na*. $\mathcal{N}_{(Sj\ !\ n\ !\ na,\ Sj\ !\ n\ !\ Suc\ na)}$) [*0*..< *length* (*Sj* ! *n*) − *1*] $\oplus_\wedge$(*Sj* ! *n* ! *0*, *Sj* ! *n* ! (*length* (*Sj* ! *n*) − *1*)) $\vee \neg$ *n* < *length Sj*
by (*metis* (*no-types*) *One-nat-def add.right-neutral add-Suc-right base-list-and f*)
then show ⊢*map* (λ*n*. $\mathcal{N}_{(Sj\ !\ i\ !\ n,\ Sj\ !\ i\ !\ Suc\ n)}$) [*0*..< *length* (*Sj* ! *i*) − *Suc 0*] $\oplus_\wedge$({*lI* ! *i*}, *s*)
using *a1* **by** (*metis* (*no-types*) *One-nat-def e f*)
qed
qed
qed (*auto simp add*: *e f*)
qed
qed
qed *auto*
qed

3.3.5 Main Theorem Completeness

theorem *Completeness*: $I \neq \{\} \Longrightarrow$ *finite I* $\Longrightarrow$
Kripke {*s* :: ('*s* :: *state*). ∃ *i* ∈ *I*. (*i* →$_i$* *s*)} (*I* :: ('*s* :: *state*)*set*) ⊢ *EF s*
$\Longrightarrow$ ∃ (*A* :: ('*s* :: *state*) *attree*). ⊢ *A* ∧ *attack A* = (*I*,*s*)
proof (*case-tac I* ⊆ *s*)
show $I \neq \{\} \Longrightarrow$ *finite I* $\Longrightarrow$
Kripke {*s*::'*s*. ∃ *i*::'*s*∈*I*. *i* →$_i$* *s*} *I* ⊢ *EF s* $\Longrightarrow$ *I* ⊆ *s* $\Longrightarrow$ ∃ *A*::'*s attree*. ⊢*A* ∧ *attack A* = (*I*, *s*)
using *att-or-empty-back attack.simps*(*3*) **by** *blast*
next
show $I \neq \{\} \Longrightarrow$ *finite I* $\Longrightarrow$
Kripke {*s*::'*s*. ∃ *i*::'*s*∈*I*. *i* →$_i$* *s*} *I* ⊢ *EF s* $\Longrightarrow \neg$ *I* ⊆ *s*
$\Longrightarrow$ ∃ *A*::'*s attree*. ⊢*A* ∧ *attack A* = (*I*, *s*)
by (*iprover intro*: *Compl-step1 Compl-step2 Compl-step3 Compl-step4 elim*:)
qed

3.3.6 Contrapositions of Correctness and Completeness

lemma *contrapos-compl*:
$I \neq \{\} \Longrightarrow$ *finite I* $\Longrightarrow$
($\neg$ (∃ (*A* :: ('*s* :: *state*) *attree*). ⊢ *A* ∧ *attack A* = (*I*, − *s*))) $\Longrightarrow$
$\neg$ (*Kripke* {*s*. ∃ *i*∈*I*. *i* →$_i$* *s*} *I* ⊢ *EF* (− *s*))
using *Completeness* **by** *auto*

lemma *contrapos-corr*:
($\neg$(*Kripke* {*s* :: ('*s* :: *state*). ∃ *i* ∈ *I*. (*i* →$_i$* *s*)} *I* ⊢ *EF s*))
$\Longrightarrow$ *attack A* = (*I*,*s*)
$\Longrightarrow \neg$ (⊢ *A*)
using *AT-EF* **by** *blast*

end

4 Infrastructures

The Isabelle Infrastructure framework supports the representation of infrastructures as graphs with actors and policies attached to nodes. These infrastructures are the *states* of the Kripke structure. The transition between states is triggered by non-parametrized actions *get*, *move*, *eval*, *put* executed by actors. Actors are given by an abstract type *actor* and a function *Actor* that creates elements of that type from identities (of type *string*). Policies are given by pairs of predicates (conditions) and sets of (enabled) actions.

4.1 Actors, actions, and data labels

theory *Infrastructure*
  imports *AT*
begin
datatype *action* = *get* | *move* | *eval* | *put*

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

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

The Decentralised Label Model (DLM) [5] introduced the idea to label data by owners and readers. We pick up this idea and formalize a new type to encode the owner and the set of readers as a pair. The first element is the owner of a data item, the second one is the set of all actors that may access the data item. This enables the unique security labelling of data within the system additionally taking the ownership into account.

type-synonym *data* = *nat*
type-synonym *dlm* = *actor* ∗ *actor set*

4.2 Infrastructure graphs and policies

Actors are contained in an infrastructure graph. An *igraph* contains a set of location pairs representing the topology of the infrastructure as a graph of nodes and a list of actor identities associated to each node (location) in the graph. Also an *igraph* associates actors to a pair of string sets by a pair-valued function whose first range component is a set describing the credentials in the possession of an actor and the second component is a set

defining the roles the actor can take on. More importantly in this context, an *igraph* assigns locations to a pair of a string that defines the state of the component and an element of type $(dlm * data)$ *set*. This set of labelled data may represent a condition on that data. Corresponding projection functions for each of these components of an *igraph* are provided; they are named *gra* for the actual set of pairs of locations, *agra* for the actor map, *cgra* for the credentials, and *lgra* for the state of a location and the data at that location.

```
datatype location = Location nat
  datatype igraph = Lgraph (location * location)set location ⇒ identity set
                           actor ⇒ (string set * string set)
                           location ⇒ string * (dlm * data) set
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 set)
where  agra(Lgraph g a c l) = a
primrec cgra :: igraph ⇒ (actor ⇒ string set * string set)
where  cgra(Lgraph g a c l) = c
primrec lgra :: igraph ⇒ (location ⇒ string * (dlm * data) set)
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 ∈ (agra g y)}
```

There are projection functions text@ *graphI* and text@ *delta* when applied to an infrastructure return the graph and the policy, respectively. Other projections are introduced for the labels, the credential, and roles and to express their meaning.

```
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 set * string set
  where tspace (Infrastructure g d) = cgra g
primrec lspace :: [infrastructure, location ] ⇒ string * (dlm * data)set
where lspace (Infrastructure g d) = lgra g

definition credentials :: string set * string set ⇒ string set
  where  credentials lxl ≡ (fst lxl)
definition has :: [igraph, actor * string] ⇒ bool
```

where *has G ac* ≡ *snd ac* ∈ *credentials*(*cgra G* (*fst ac*))
definition *roles* :: *string set* ∗ *string set* ⇒ *string set*
where *roles lxl* ≡ (*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* = *fst* (*lgra G l*)

Predicates and projections for the labels to encode their meaning.

definition *owner* :: *dlm* ∗ *data* ⇒ *actor* **where** *owner d* ≡ *fst*(*fst d*)
definition *owns* :: [*igraph*, *location*, *actor*, *dlm* ∗ *data*] ⇒ *bool*
where *owns G l a d* ≡ *owner d* = *a*
definition *readers* :: *dlm* ∗ *data* ⇒ *actor set*
where *readers d* ≡ *snd* (*fst d*)

The predicate *has-access* is true for owners or readers.

definition *has-access* :: [*igraph*, *location*, *actor*, *dlm* ∗ *data*] ⇒ *bool*
where *has-access G l a d* ≡ *owns G l a d* ∨ *a* ∈ *readers d*

We define a type of functions that preserves the security labeling and a corresponding function application operator.

typedef *label-fun* = {*f* :: *dlm* ∗ *data* ⇒ *dlm* ∗ *data*.
∀ *x*:: *dlm* ∗ *data*. *fst x* = *fst* (*f x*)}
by (*fastforce*)

definition *secure-process* :: *label-fun* ⇒ *dlm* ∗ *data* ⇒ *dlm* ∗ *data* (**infixr** ‹⇳› *50*)
where *f* ⇳ *d* ≡ (*Rep-label-fun f*) *d*

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

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

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

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

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

definition *behaviour* :: *infrastructure* ⇒ (*location* ∗ *actor* ∗ *action*)*set*
where *behaviour I* ≡ {(*t*,*a*,*a′*). *enables I t a a′*}

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

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

4.3 State transition on infrastructures

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

First we introduce some auxiliary functions dealing with repetitions in lists and actors moving in an igraph.

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*)

definition *move-graph-a* :: [*identity*, *location*, *location*, *igraph*] ⇒ *igraph*
where *move-graph-a n l l′ g* ≡ *Lgraph* (*gra g*)
(*if n* ∈ ((*agra g*) *l*) & *n* ∉ ((*agra g*) *l′*) *then*
((*agra g*)(*l* := (*agra g l*) − {*n*}))(*l′* := (*insert 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* ∈ *nodes G*; *l′* ∈ *nodes G*;
(*a*) ∈ *actors-graph*(*graphI I*); *enables I l′* (*Actor a*) *move*;
I′ = *Infrastructure* (*move-graph-a a l l′* (*graphI I*))(*delta I*) ⟧ ⟹ *I* →$_n$ *I′*
| *get* : ⟦ *G* = *graphI I*; *a* @$_G$ *l*; *a′* @$_G$ *l*; *has G* (*Actor a*, *z*);
enables I l (*Actor a*) *get*;
I′ = *Infrastructure*
(*Lgraph* (*gra G*)(*agra G*)
((*cgra G*)(*Actor a′* :=
(*insert z* (*fst*(*cgra G* (*Actor a′*))), *snd*(*cgra G* (*Actor a′*)))))
(*lgra G*))
(*delta I*)
⟧ ⟹ *I* →$_n$ *I′*
| *get-data* : *G* = *graphI I* ⟹ *a* @$_G$ *l* ⟹
enables I l′ (*Actor a*) *get* ⟹
((*Actor a′*, *as*), *n*) ∈ *snd* (*lgra G l′*) ⟹ *Actor a* ∈ *as* ⟹
I′ = *Infrastructure*
(*Lgraph* (*gra G*)(*agra G*)(*cgra G*)
((*lgra G*)(*l* := (*fst* (*lgra G l*),

$snd\ (lgra\ G\ l)\ \cup \{((Actor\ a', as), n)\})))$
$(delta\ I)$
$\Longrightarrow I \to_n I'$
$|\ process : G = graphI\ I \Longrightarrow a\ @_G\ l \Longrightarrow$
$enables\ I\ l\ (Actor\ a)\ eval \Longrightarrow$
$((Actor\ a', as), n) \in snd\ (lgra\ G\ l) \Longrightarrow Actor\ a \in as \Longrightarrow$
$I' = Infrastructure$
$(Lgraph\ (gra\ G)(agra\ G)(cgra\ G)$
$((lgra\ G)(l := (fst\ (lgra\ G\ l),$
$snd\ (lgra\ G\ l)\ - \{((Actor\ a', as), n)\}$
$\cup \{(f :: label\text{-}fun) \Updownarrow ((Actor\ a', as), n)\})))$
$(delta\ I)$
$\Longrightarrow I \to_n I'$
$|\ del\text{-}data : G = graphI\ I \Longrightarrow a \in actors\ G \Longrightarrow l \in nodes\ G \Longrightarrow$
$((Actor\ a, as), n) \in snd\ (lgra\ G\ l) \Longrightarrow$
$I' = Infrastructure$
$(Lgraph\ (gra\ G)(agra\ G)(cgra\ G)$
$((lgra\ G)(l := (fst\ (lgra\ G\ l), snd\ (lgra\ G\ l) - \{((Actor\ a, as),$
$n)\})))$
$(delta\ I)$
$\Longrightarrow I \to_n I'$
$|\ put : G = graphI\ I \Longrightarrow a\ @_G\ l \Longrightarrow enables\ I\ l\ (Actor\ a)\ put \Longrightarrow$
$I' = Infrastructure$
$(Lgraph\ (gra\ G)(agra\ G)(cgra\ G)$
$((lgra\ G)(l := (s, snd\ (lgra\ G\ l) \cup \{((Actor\ a, as), n)\})))$
$(delta\ I)$
$\Longrightarrow I \to_n I'$

Note that the type infrastructure can now be instantiated to the axiomatic type class *state* which enables the use of the underlying Kripke structures and CTL.

instantiation *infrastructure* :: *state*
begin
definition
state-transition-infra-def: $(i \to_i i') = (i \to_n (i' :: infrastructure))$

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

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

end

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

end

5 Application example from IoT healthcare

The example of an IoT healthcare systems is taken from the context of the CHIST-ERA project SUCCESS [1]. In this system architecture, data is collected by sensors in the home or via a smart phone helping to monitor bio markers of the patient. The data collection is in a cloud based server to enable hospitals (or scientific institutions) to access the data which is controlled via the smart phone. The identities Patient and Doctor represent patients and their doctors; double quotes "s" indicate strings in Isabelle/HOL. The global policy is 'only the patient and the doctor can access the data in the cloud'.

theory *GDPRhealthcare*
imports *Infrastructure*
begin

Local policies are represented as a function over an *igraph* G that additionally assigns each location of a scenario to its local policy given as a pair of requirements to an actor (first element of the pair) in order to grant him actions in the location (second element of the pair). The predicate $@G$ checks whether an actor is at a given location in the *igraph* G.

locale *scenarioGDPR* =
fixes *gdpr-actors* :: *identity set*
defines *gdpr-actors-def*: *gdpr-actors* ≡ {*''Patient''*, *''Doctor''*}
fixes *gdpr-locations* :: *location set*
defines *gdpr-locations-def*: *gdpr-locations* ≡
{*Location 0*, *Location 1*, *Location 2*, *Location 3*}
fixes *sphone* :: *location*
defines *sphone-def*: *sphone* ≡ *Location 0*
fixes *home* :: *location*
defines *home-def*: *home* ≡ *Location 1*
fixes *hospital* :: *location*
defines *hospital-def*: *hospital* ≡ *Location 2*
fixes *cloud* :: *location*
defines *cloud-def*: *cloud* ≡ *Location 3*
fixes *global-policy* :: [*infrastructure*, *identity*] ⇒ *bool*
defines *global-policy-def*: *global-policy I a* ≡ *a* ≠ *''Doctor''*
⟶ ¬(*enables I hospital* (*Actor a*) *eval*)
fixes *global-policy′* :: [*infrastructure*, *identity*] ⇒ *bool*
defines *global-policy′-def*: *global-policy′ I a* ≡ *a* ∉ *gdpr-actors*
⟶ ¬(*enables I cloud* (*Actor a*) *get*)
fixes *ex-creds* :: *actor* ⇒ (*string set* ∗ *string set*)
defines *ex-creds-def*: *ex-creds* ≡ (λ *x*. *if x* = *Actor ''Patient'' then*
({*''PIN''*,*''skey''*}, {}) *else*
(*if x* = *Actor ''Doctor'' then*
({*''PIN''*},{}) *else* ({},{})))
fixes *ex-locs* :: *location* ⇒ *string* ∗ (*dlm* ∗ *data*) *set*
defines *ex-locs* ≡ (λ *x*. *if x* = *cloud then*

```
            (''free'',{((Actor ''Patient'',{Actor ''Doctor''}),42)})
            else (''',{}))
fixes ex-loc-ass :: location ⇒ identity set
defines ex-loc-ass-def: ex-loc-ass ≡
          (λ x.  if x = home then {''Patient''}
                 else (if x = hospital then {''Doctor'', ''Eve''}
                       else {}))

fixes ex-graph :: igraph
defines ex-graph-def: ex-graph ≡ Lgraph
    {(home, cloud), (sphone, cloud), (cloud,hospital)}
    ex-loc-ass
    ex-creds ex-locs
fixes ex-graph' :: igraph
defines ex-graph'-def: ex-graph' ≡ Lgraph
    {(home, cloud), (sphone, cloud), (cloud,hospital)}
      (λ x. if x = cloud then {''Patient''} else
          (if x = hospital then {''Doctor'',''Eve''} else {}))
    ex-creds ex-locs
fixes ex-graph'' :: igraph
defines ex-graph''-def: ex-graph'' ≡ Lgraph
    {(home, cloud), (sphone, cloud), (cloud,hospital)}
      (λ x. if x = cloud then {''Patient'', ''Eve''} else
          (if x = hospital then {''Doctor''} else {}))
    ex-creds ex-locs

fixes local-policies :: [igraph, location] ⇒ policy set
defines local-policies-def: local-policies G ≡
   (λ x. if x = home then
      {(λ y. True, {put,get,move,eval})}
        else (if x = sphone then
          {((λ y. has G (y, ''PIN'')), {put,get,move,eval})}
            else (if x = cloud then
            {(λ y. True, {put,get,move,eval})}
                  else (if x = hospital then
            {((λ y. (∃ n. (n  @_G hospital) ∧ Actor n = y ∧
                      has G (y, ''skey''))), {put,get,move,eval})} else {}))))

fixes gdpr-scenario :: infrastructure
defines gdpr-scenario-def:
gdpr-scenario ≡ Infrastructure ex-graph local-policies
fixes Igdpr :: infrastructure set
defines Igdpr-def:
  Igdpr ≡ {gdpr-scenario}


fixes gdpr-scenario' :: infrastructure
defines gdpr-scenario'-def:
gdpr-scenario' ≡ Infrastructure ex-graph' local-policies
```

fixes *GDPR′* :: *infrastructure set*
defines *GDPR′-def*:
GDPR′ ≡ {*gdpr-scenario′*}

fixes *gdpr-scenario″* :: *infrastructure*
defines *gdpr-scenario″-def*:
gdpr-scenario″ ≡ *Infrastructure ex-graph″ local-policies*
fixes *GDPR″* :: *infrastructure set*
defines *GDPR″-def*:
GDPR″ ≡ {*gdpr-scenario″*}
fixes *gdpr-states*
defines *gdpr-states-def*: *gdpr-states* ≡ { *I*. *gdpr-scenario* →$_i$∗ *I* }
fixes *gdpr-Kripke*
defines *gdpr-Kripke* ≡ *Kripke gdpr-states* {*gdpr-scenario*}
fixes *sgdpr*
defines *sgdpr* ≡ {*x*. ¬ (*global-policy′ x ″Eve″*)}
begin

5.1 Using Attack Tree Calculus

Since we consider a predicate transformer semantics, we use sets of states to represent properties. For example, the attack property is given by the above *set sgdpr*.

The attack we are interested in is to see whether for the scenario

gdpr scenario ≡ *Infrastructure ex-graph local-policies*

from the initial state

Igdpr ≡{*gdpr scenario*},

the critical state *sgdpr* can be reached, i.e., is there a valid attack (*Igdpr*,*sgdpr*)?

We first present a number of lemmas showing single and multi-step state transitions for relevant states reachable from our *gdpr-scenario*.

lemma *step1*: *gdpr-scenario* →$_n$ *gdpr-scenario′*
proof (*rule-tac l* = *home* **and** *a* = *″Patient″* **and** *l′* = *cloud* **in** *move*)
show *graphI gdpr-scenario* = *graphI gdpr-scenario* **by** (*rule refl*)
next show *″Patient″* @$_{graphI\ gdpr\text{-}scenario}$ *home*
by (*simp add*: *gdpr-scenario-def ex-graph-def ex-loc-ass-def atI-def nodes-def*)
next show *home* ∈ *nodes* (*graphI gdpr-scenario*)
by (*simp add*: *gdpr-scenario-def ex-graph-def ex-loc-ass-def atI-def nodes-def*, *blast*)
next show *cloud* ∈ *nodes* (*graphI gdpr-scenario*)
by (*simp add*: *gdpr-scenario-def nodes-def ex-graph-def*, *blast*)
next show *″Patient″* ∈ *actors-graph* (*graphI gdpr-scenario*)
by (*simp add*: *actors-graph-def gdpr-scenario-def ex-graph-def ex-loc-ass-def nodes-def*, *blast*)
next show *enables gdpr-scenario cloud* (*Actor ″Patient″*) *move*
by (*simp add*: *enables-def gdpr-scenario-def ex-graph-def local-policies-def ex-creds-def ex-locs-def has-def credentials-def*)

next show *gdpr-scenario′ =*
Infrastructure (*move-graph-a ″Patient″ home cloud* (*graphI gdpr-scenario*))
(*delta gdpr-scenario*)
apply (*simp add*: *gdpr-scenario′-def ex-graph′-def move-graph-a-def*
gdpr-scenario-def ex-graph-def home-def cloud-def hospital-def
ex-loc-ass-def ex-creds-def)
apply (*rule ext*)
by (*simp add*: *hospital-def*)
qed

lemma *step1r*: *gdpr-scenario* $\rightarrow_n*$ *gdpr-scenario′*
proof (*simp add*: *state-transition-in-refl-def*)
show (*gdpr-scenario*, *gdpr-scenario′*) ∈ {(*x*::*infrastructure*, *y*::*infrastructure*). *x* $\rightarrow_n$ *y*}*
by (*insert step1*, *auto*)
qed

lemma *step2*: *gdpr-scenario′* $\rightarrow_n$ *gdpr-scenario″*
proof (*rule-tac l* = *hospital* **and** *a* = *″Eve″* **and** *l′* = *cloud* **in** *move*, *rule refl*)
show *″Eve″* @$_{graphI\ gdpr\text{-}scenario'}$ *hospital*
by (*simp add*: *gdpr-scenario′-def ex-graph′-def hospital-def cloud-def atI-def*
nodes-def)
next show *hospital* ∈ *nodes* (*graphI gdpr-scenario′*)
by (*simp add*: *gdpr-scenario′-def ex-graph′-def hospital-def cloud-def atI-def*
nodes-def, *blast*)
next show *cloud* ∈ *nodes* (*graphI gdpr-scenario′*)
by (*simp add*: *gdpr-scenario′-def nodes-def ex-graph′-def*, *blast*)
next show *″Eve″* ∈ *actors-graph* (*graphI gdpr-scenario′*)
by (*simp add*: *actors-graph-def gdpr-scenario′-def ex-graph′-def nodes-def*
hospital-def cloud-def, *blast*)
next show *enables gdpr-scenario′ cloud* (*Actor ″Eve″*) *move*
by (*simp add*: *enables-def gdpr-scenario′-def ex-graph-def local-policies-def*
ex-creds-def ex-locs-def has-def credentials-def cloud-def sphone-def)
next show *gdpr-scenario″ =*
Infrastructure (*move-graph-a ″Eve″ hospital cloud* (*graphI gdpr-scenario′*))
(*delta gdpr-scenario′*)
apply (*simp add*: *gdpr-scenario′-def ex-graph″-def move-graph-a-def gdpr-scenario″-def*
ex-graph′-def home-def cloud-def hospital-def ex-creds-def)
apply (*rule ext*)
apply (*simp add*: *hospital-def*)
by *blast*
qed

lemma *step2r*: *gdpr-scenario′* $\rightarrow_n*$ *gdpr-scenario″*
proof (*simp add*: *state-transition-in-refl-def*)
show (*gdpr-scenario′*, *gdpr-scenario″*) ∈ {(*x*::*infrastructure*, *y*::*infrastructure*). *x* $\rightarrow_n$ *y*}*
by (*insert step2*, *auto*)

qed

For the Kripke structure
gdpr-Kripke ≡ *Kripke* { *I*. *gdpr-scenario* $\rightarrow_i*$ *I* } {*gdpr-scenario*}
we first derive a valid and-attack using the attack tree proof calculus.
$\vdash[\mathcal{N}_{(Igdpr,GDPR')}, \mathcal{N}_{(GDPR',sgdpr)}] \oplus_\wedge{}^{(Igdpr,sgdpr)}$
The set *GDPR′* (see above) is an intermediate state where Eve accesses the cloud.

lemma *gdpr-ref*: $[\mathcal{N}_{(Igdpr,sgdpr)}] \oplus_\wedge{}^{(Igdpr,sgdpr)} \sqsubseteq$
$([\mathcal{N}_{(Igdpr,GDPR')}, \mathcal{N}_{(GDPR',sgdpr)}] \oplus_\wedge{}^{(Igdpr,sgdpr)})$
proof (*rule-tac* $l = []$ **and** $l' = [\mathcal{N}_{(Igdpr,GDPR')}, \mathcal{N}_{(GDPR',sgdpr)}]$ **and**
$l'' = []$ **and** *si* = *Igdpr* **and** *si′* = *Igdpr* **and**
si″ = *sgdpr* **and** *si‴* = *sgdpr* **in** *refI*, *simp*, *rule refl*)
show $([\mathcal{N}_{(Igdpr,\ GDPR')}, \mathcal{N}_{(GDPR',\ sgdpr)}] \oplus_\wedge{}^{(Igdpr,\ sgdpr)}) =$
$([] @ [\mathcal{N}_{(Igdpr,\ GDPR')}, \mathcal{N}_{(GDPR',\ sgdpr)}] @ [] \oplus_\wedge{}^{(Igdpr,\ sgdpr)})$
by *simp*
qed

lemma *att-gdpr*: $\vdash([\mathcal{N}_{(Igdpr,GDPR')}, \mathcal{N}_{(GDPR',sgdpr)}] \oplus_\wedge{}^{(Igdpr,sgdpr)})$
proof (*subst att-and*, *simp*, *rule conjI*)
show $\vdash\mathcal{N}_{(Igdpr,\ GDPR')}$
apply (*simp add*: *Igdpr-def GDPR′-def att-base*)
using *state-transition-infra-def step1* **by** *blast*
next
have ¬ *global-policy′ gdpr-scenario″ ″Eve″ gdpr-scenario′* $\rightarrow_n$ *gdpr-scenario″*
using *step2*
by (*auto simp*: *global-policy′-def gdpr-scenario″-def gdpr-actors-def*
enables-def local-policies-def cloud-def sphone-def intro!: *step2*)
then show $\vdash([\mathcal{N}_{(GDPR',\ sgdpr)}] \oplus_\wedge{}^{(GDPR',\ sgdpr)})$
apply (*subst att-and*)
apply (*simp add*: *GDPR′-def sgdpr-def att-base*)
using *state-transition-infra-def* **by** *blast*
qed

lemma *gdpr-abs-att*: $\vdash_V([\mathcal{N}_{(Igdpr,sgdpr)}] \oplus_\wedge{}^{(Igdpr,sgdpr)})$
by (*rule ref-valI*, *rule gdpr-ref*, *rule att-gdpr*)

We can then simply apply the Correctness theorem *AT EF* to immediately prove the following CTL statement.

gdpr-Kripke ⊢ *EF sgdpr*

This application of the meta-theorem of Correctness of attack trees saves us proving the CTL formula tediously by exploring the state space.

lemma *gdpr-att*: *gdpr-Kripke* ⊢ *EF* {*x*. ¬(*global-policy′ x ″Eve″*)}
proof −

have *a*: $\vdash([\mathcal{N}_{(Igdpr,\ GDPR')}, \mathcal{N}_{(GDPR',\ sgdpr)}] \oplus_{\wedge}{}^{(Igdpr,\ sgdpr)})$
by (*rule att-gdpr*)
hence (*Igdpr*,*sgdpr*) = *attack* $([\mathcal{N}_{(Igdpr,\ GDPR')}, \mathcal{N}_{(GDPR',\ sgdpr)}] \oplus_{\wedge}{}^{(Igdpr,\ sgdpr)})$
by *simp*
hence *Kripke* {*s*::*infrastructure*. ∃ *i*::*infrastructure*∈*Igdpr*. *i* $\rightarrow_i*$ *s*} *Igdpr* ⊢ *EF sgdpr*
using *ATV-EF gdpr-abs-att* **by** *fastforce*
thus *gdpr-Kripke* ⊢ *EF* {*x*::*infrastructure*. ¬ *global-policy'* *x* ''*Eve*''}
by (*simp add*: *gdpr-Kripke-def gdpr-states-def Igdpr-def sgdpr-def*)
qed

theorem *gdpr-EF*: *gdpr-Kripke* ⊢ *EF sgdpr*
using *gdpr-att sgdpr-def* **by** *blast*

Similarly, vice-versa, the CTL statement proved in *gdpr-EF* can now be directly translated into Attack Trees using the Completeness Theorem[1].

theorem *gdpr-AT*: ∃ *A*. ⊢ *A* ∧ *attack A* = (*Igdpr*,*sgdpr*)
proof −
have *a*: *gdpr-Kripke* ⊢ *EF sgdpr* **by** (*rule gdpr-EF*)
have *b*: *Igdpr* ≠ {} **by** (*simp add*: *Igdpr-def*)
thus ∃ *A*::*infrastructure attree*. ⊢*A* ∧ *attack A* = (*Igdpr*, *sgdpr*)
proof (*rule Completeness*)
show *Kripke* {*s*. ∃ *i*∈*Igdpr*. *i* $\rightarrow_i*$ *s*} *Igdpr* ⊢ *EF sgdpr*
using *a* **by** (*simp add*: *gdpr-Kripke-def Igdpr-def gdpr-states-def*)
qed (*auto simp*: *Igdpr-def*)
qed

Conversely, since we have an attack given by rule *gdpr-AT*, we can immediately infer *EF s* using Correctness *AT-EF*[2].

theorem *gdpr-EF'*: *gdpr-Kripke* ⊢ *EF sgdpr*
using *gdpr-AT* **by** (*auto simp*: *gdpr-Kripke-def gdpr-states-def Igdpr-def dest*: *AT-EF*)

6 Data Protection by Design for GDPR compliance

6.1 General Data Protection Regulation (GDPR)

Since 26th May 2018, the GDPR has become mandatory within the European Union and hence also for any supplier of IT products. Breaches of the regulation will be fined with penalties of 20 Million EUR. Despite the

[1]This theorem could easily be proved as a direct instance of *att-gdpr* above but we want to illustrate an alternative proof method using Completeness here.

[2]Clearly, this theorem is identical to *gdpr-EF* and could thus be inferred from that one but we want to show here an alternative way of proving it using the Correctness theorem *AT-EF*.

relatively large size of the document of 209 pages, the technically relevant portion for us is only about 30 pages (Pages 81–111, Chapters I to Chapter III, Section 3). In summary, Chapter III specifies that the controller must give the data subject read access (1) to any information, communications, and "meta-data" of the data, e.g., retention time and purpose. In addition, the system must enable deletion of data (2) and restriction of processing. An invariant condition for data processing resulting from these Articles is that the system functions must preserve any of the access rights of personal data (3).

Using labeled data, we can now express the essence of Article 4 Paragraph (1): 'personal data' means any information relating to an identified or identifiable natural person ('data subject').

The labels of data must not be changed by processing: we have identified this as an invariant (3) resulting from the GDPR above. This invariant is formalized in our Isabelle model by the type definition of functions on labeled data *label-fun* (see Section 4.2) that preserve the data labels.

6.2 Policy enforcement and privacy preservation

We can now use the labeled data to encode the privacy constraints of the GDPR in the rules. For example, the get data rule (see Section 4.3) has labelled data $((Actor\ a', as), n)$ and uses the labeling in the precondition to guarantee that only entitled users can get data.

We can prove that processing preserves ownership as defined in the initial state for all paths globally (AG) within the Kripke structure and in all locations of the graph.

```
lemma gdpr-three: h ∈ gdpr-actors ⟹ l ∈ gdpr-locations ⟹
        owns (Igraph gdpr-scenario) l (Actor h) d ⟹
        gdpr-Kripke ⊢ AG {x. ∀ l ∈ gdpr-locations. owns (Igraph x) l (Actor h) d
}
proof (simp add: gdpr-Kripke-def check-def, rule conjI)
 show gdpr-scenario ∈ gdpr-states by (simp add: gdpr-states-def state-transition-refl-def)
next
  show h ∈ gdpr-actors ⟹
    l ∈ gdpr-locations ⟹
    owns (Igraph gdpr-scenario) l (Actor h) d ⟹
     gdpr-scenario ∈ AG {x::infrastructure. ∀ l∈gdpr-locations. owns (Igraph x) l
(Actor h) d}
    apply (simp add: AG-def gfp-def)
     apply (rule-tac x = {x::infrastructure. ∀ l∈gdpr-locations. owns (Igraph x) l
(Actor h) d} in exI)
    by (auto simp: AX-def gdpr-scenario-def owns-def)
qed
```

The final application example of Correctness contraposition shows that there

is no attack to ownership possible. The proved meta-theory for attack trees can be applied to facilitate the proof. The contraposition of the Correctness property grants that if there is no attack on $(I,\neg f)$, then $(EF\ \neg f)$ does not hold in the Kripke structure. This yields the theorem since the $AG\ f$ statement corresponds to $\neg(EF\ \neg f)$.

theorem *no-attack-gdpr-three*:
$h \in$ *gdpr-actors* $\Longrightarrow$ $l \in$ *gdpr-locations* $\Longrightarrow$
owns (*Igraph gdpr-scenario*) *l* (*Actor h*) *d* $\Longrightarrow$
attack A = (*Igdpr*, $-\{x.\ \forall\ l \in$ *gdpr-locations*. *owns* (*Igraph x*) *l* (*Actor h*) *d* $\}$)
$\Longrightarrow \neg\ (\vdash A)$
proof (*rule-tac I* = *Igdpr* **and**
s = $-\{x$::*infrastructure*. $\forall\, l \in$*gdpr-locations*. *owns* (*Igraph x*) *l* (*Actor h*) *d*$\}$
in *contrapos-corr*)
show $h \in$ *gdpr-actors* $\Longrightarrow$
$l \in$ *gdpr-locations* $\Longrightarrow$
owns (*Igraph gdpr-scenario*) *l* (*Actor h*) *d* $\Longrightarrow$
attack A = (*Igdpr*, $-$ $\{x$::*infrastructure*. $\forall\, l \in$*gdpr-locations*. *owns* (*Igraph x*) *l* (*Actor h*) *d*$\}$) $\Longrightarrow$
$\neg$ (*Kripke* $\{s$::*infrastructure*. $\exists\, i$::*infrastructure*$\in$*Igdpr*. $i \rightarrow_i * s\}$
Igdpr $\vdash$ *EF* ($-$ $\{x$::*infrastructure*. $\forall\, l \in$*gdpr-locations*. *owns* (*Igraph x*) *l* (*Actor h*) *d*$\}$))
apply (*rule AG-imp-notnotEF*)
apply (*simp add*: *gdpr-Kripke-def Igdpr-def gdpr-states-def*)
using *Igdpr-def gdpr-Kripke-def gdpr-states-def gdpr-three* **by** *auto*
qed
end
end

References

[1] CHIST-ERA. Success: Secure accessibility for the internet of things, 2016. http://www.chistera.eu/projects/success.

[2] F. Kammüller. Isabelle modelchecking for insider threats. In *Data Privacy Management, DPM'16, 11th Int. Workshop*, volume 9963 of *LNCS*. Springer, 2016. Co-located with ESORICS'16.

[3] F. Kammüller. Attack trees in isabelle. In *20th International Conference on Information and Communications Security, ICICS2018*, volume 11149 of *LNCS*. Springer, 2018.

[4] F. Kammüller. Formal modeling and analysis of data protection for gdpr compliance of iot healthcare systems. In *IEEE Systems, Man and Cybernetics, SMC2018*. IEEE, 2018.

[5] A. C. Myers and B. Liskov. Complete, safe information flow with decentralized labels. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE, 1999.

[6] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002. http://www.in.tum.de/~nipkow/LNCS2283/.