Attack Trees in Isabelle for GDPR compliance of IoT healthcare systems

Florian Kammüller

September 26, 2023

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*))({})
⟨*proof*⟩

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

lemma *less-not-le*: ⟦(*x*:: *nat*) < *y*; *y* $\leq$ *x*⟧ $\Longrightarrow$ *False*
⟨*proof*⟩

lemma *infchain-outruns-all*:
assumes *finite* (*UNIV* :: *'a set*)
and $\forall i :: nat.\ ((\tau :: 'a\ set \Rightarrow 'a\ set) \frown i)\ (\{\} :: 'a\ set) \subset (\tau \frown (i + 1))\ \{\}$
shows $\forall j :: nat.\ \exists i :: nat.\ j < card\ ((\tau \frown i)\ \{\})$
⟨*proof*⟩

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*⟩

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

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

⟨*proof*⟩

lemma *predtrans-UNIV*:
assumes *mono* $(\tau :: ('a\ set \Rightarrow 'a\ set))$
shows $\forall\ i.\ (\tau \frown i)\ (UNIV) \supseteq (\tau \frown (i + 1))(UNIV)$
⟨*proof*⟩

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

lemma *card-univ-subtract*:
assumes *finite* (*UNIV* :: *'a set*) **and** *mono* τ
and $(\forall i :: nat.\ ((\tau :: 'a\ set \Rightarrow 'a\ set) \frown (i + (1 :: nat)))(UNIV :: 'a\ set) \subset (\tau \frown i)\ UNIV)$

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

lemma *card-UNIV-tau-i-below-zero*:
assumes *finite* $(UNIV :: 'a\ set)$ **and** *mono* τ
and $(\forall i :: nat.\ ((\tau :: ('a\ set \Rightarrow 'a\ set))\ \widehat{}\ (i + (1 :: nat)))(UNIV :: 'a\ set) \subset (\tau\ \widehat{}\ i)\ UNIV)$
shows $card((\tau\ \widehat{}\ (card\ (UNIV ::'a\ set)))\ (UNIV ::'a\ set)) \leq 0$
⟨*proof*⟩

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

lemma *UNIV-tau-i-is-empty*:
assumes *finite* $(UNIV :: 'a\ set)$ **and** *mono* $(\tau :: ('a\ set \Rightarrow 'a\ set))$
and $(\forall i :: nat.\ ((\tau :: 'a\ set \Rightarrow 'a\ set)\ \widehat{}\ (i + (1 :: nat)))(UNIV :: 'a\ set) \subset (\tau\ \widehat{}\ i)\ UNIV)$
shows $(\tau\ \widehat{}\ (card\ (UNIV ::'a\ set)))\ (UNIV ::'a\ set) = \{\}$
⟨*proof*⟩

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

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

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

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

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** $p \leq (f\widehat{}k)\ (top::'a::order\text{-}top)$
⟨*proof*⟩

lemma *gfp-loop*:
assumes *finite* (*UNIV* :: *'b set*)
and *mono* (*τ* :: (*'b set* ⇒ *'b set*))
shows ∃ *n* . *gfp τ* = (*τ* ^^ *n*) *UNIV*
⟨*proof*⟩

1.2 Generic type of state with state transition and CTL operators

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

class *state* =
fixes *state-transition* :: [*'a* :: *type*, *'a*] ⇒ *bool* (**infixr** →*i* *50*)

The above class definition lifts Kripke structures and CTL to a general level. The definition of the inductive relation is given by a set of specific rules which are, however, part of an application like infrastructures. Branching time temporal logic CTL is defined in general over Kripke structures with arbitrary state transitions and can later be applied to suitable theories, like infrastructures. Based on the generic state transition → 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* ≡ {*s*. {*f0*. *s* →*i* *f0*} ⊆ *f*}
definition *EX'* **where** *EX' f* ≡ {*s* . ∃ *f0* ∈ *f*. *s* →*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* ≡ *lfp* (λ *Z*. *f* ∪ *AX Z*)
definition *EF* **where** *EF f* ≡ *lfp* (λ *Z*. *f* ∪ *EX' Z*)
definition *AG* **where** *AG f* ≡ *gfp* (λ *Z*. *f* ∩ *AX Z*)
definition *EG* **where** *EG f* ≡ *gfp* (λ *Z*. *f* ∩ *EX' Z*)
definition *AU* **where** *AU f1 f2* ≡ *lfp*(λ *Z*. *f2* ∪ (*f1* ∩ *AX Z*))
definition *EU* **where** *EU f1 f2* ≡ *lfp*(λ *Z*. *f2* ∪ (*f1* ∩ *EX' Z*))
definition *AR* **where** *AR f1 f2* ≡ *gfp*(λ *Z*. *f2* ∩ (*f1* ∪ *AX Z*))
definition *ER* **where** *ER f1 f2* ≡ *gfp*(λ *Z*. *f2* ∩ (*f1* ∪ *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* (- ⊢ - *50*)
where *M* ⊢ *f* ≡ (*init M*) ⊆ {*s* ∈ (*states M*). *s* ∈ *f* }

definition *state-transition-refl* (**infixr** →$_i$* *50*)
where *s* →$_i$* *s'* ≡ ((*s*,*s'*) ∈ {(*x*,*y*). *state-transition x y*}*)

1.4 Lemmas for CTL operators

1.4.1 EF lemmas

lemma *EF-lem0*: (*x* ∈ *EF f*) = (*x* ∈ *f* ∪ *EX'* (*lfp* (λ*Z* :: ('*a* :: *state*) *set*. *f* ∪ *EX'* *Z*)))
⟨*proof*⟩

lemma *EF-lem00*: (*EF f*) = (*f* ∪ *EX'* (*lfp* (λ *Z* :: ('*a* :: *state*) *set*. *f* ∪ *EX'* *Z*)))
⟨*proof*⟩

lemma *EF-lem000*: (*EF f*) = (*f* ∪ *EX'* (*EF f*))
⟨*proof*⟩

lemma *EF-lem1*: *x* ∈ *f* ∨ *x* ∈ (*EX'* (*EF f*)) ⟹ *x* ∈ *EF f*
⟨*proof*⟩

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

lemma *EF-lem2a*: **assumes** *x* ∈ *f* **shows** *x* ∈ *EF f*
⟨*proof*⟩

lemma *EF-lem2c*: **assumes** *x* ∉ *f* **shows** *x* ∈ *EF* (− *f*)
⟨*proof*⟩

lemma *EF-lem2d*: **assumes** *x* ∉ *EF f* **shows** *x* ∉ *f*
⟨*proof*⟩

lemma *EF-lem3b*: **assumes** *x* ∈ *EX'* (*f* ∪ *EX'* (*EF f*)) **shows** *x* ∈ (*EF f*)
⟨*proof*⟩

lemma *EX-lem0l*: *x* ∈ (*EX' f*) ⟹ *x* ∈ (*EX'* (*f* ∪ *g*))
⟨*proof*⟩

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

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

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

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

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

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

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

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

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

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

1.4.2 AG lemmas

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

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

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

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

lemma *AG-step*: $y \rightarrow_i z \Longrightarrow y \in AG\ s \Longrightarrow z \in AG\ s$

⟨*proof*⟩

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

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

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

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

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

primrec *attack* :: ('*s* :: *state*) *attree* ⇒ ('*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-to* :: [('*s* :: *state*) *attree*, '*s attree*] ⇒ *bool* (- ⊑ - [*40*] *40*)
where
refI: ⟦ $A = ((l \mathbin{@} [\mathcal{N}_{(si',si'')}] \mathbin{@} l'') \oplus_{\wedge}{}^{(si,si''')}$);
$A' = (l' \oplus_{\wedge}{}^{(si',si'')})$;
$A'' = (l \mathbin{@} l' \mathbin{@} l'' \oplus_{\wedge}{}^{(si,si''')})$
⟧ ⟹ $A \sqsubseteq A''$|
ref-or: ⟦ *as* ≠ []; ∀ $A' \in set(as)$. $(A \sqsubseteq A') \wedge attack\ A = s$ ⟧ ⟹ $A \sqsubseteq (as \oplus_{\vee}{}^{s})$ |
ref-trans: ⟦ $A \sqsubseteq A'$; $A' \sqsubseteq A''$ ⟧ ⟹ $A \sqsubseteq A''$|
ref-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* :: $[('s :: state)\ attree] \Rightarrow bool$ $(\vdash\text{-}\ [40]\ 40)$
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*⟩

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

lemma *att-and-empty*[*rule-format*] : $\vdash([] \oplus_\wedge^{(s',\ s'')}) \longrightarrow s' \subseteq s''$
⟨*proof*⟩

lemma *att-and-empty2*: $\vdash([] \oplus_\wedge^{(s,\ s)})$
⟨*proof*⟩

lemma *att-or-empty*[*rule-format*] : $\vdash([] \oplus_\vee^{(s',\ s'')}) \longrightarrow s' \subseteq s''$
⟨*proof*⟩

lemma *att-or-empty-back*[*rule-format*]: $s' \subseteq s'' \longrightarrow \vdash([] \oplus_\vee^{(s',\ s'')})$
⟨*proof*⟩

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

lemma *att-or-empty2*: $\vdash([] \oplus_\vee^{(s,\ s)})$
⟨*proof*⟩

lemma *att-andD1*: $\vdash(x1\ \#\ x2 \oplus_\wedge^{s}) \Longrightarrow \vdash x1$
⟨*proof*⟩

lemma *att-and-nonemptyD2*[*rule-format*]:
$(x2 \neq [] \longrightarrow \vdash(x1\ \#\ x2 \oplus_\wedge^{s}) \longrightarrow \vdash (x2 \oplus_\wedge^{(snd(attack\ x1),snd\ s)}))$
⟨*proof*⟩

lemma *att-andD2* : $\vdash(x1\ \#\ x2 \oplus_\wedge^{s}) \Longrightarrow \vdash (x2 \oplus_\wedge^{(snd(attack\ x1),snd\ s)})$
⟨*proof*⟩

lemma *att-and-fst-lem*[*rule-format*]:
$\vdash(x1\ \#\ x2a\ \oplus_\wedge{}^x) \longrightarrow xa \in fst\ (attack\ (x1\ \#\ x2a\ \oplus_\wedge{}^x))$
$\longrightarrow xa \in fst\ (attack\ x1)$
⟨*proof*⟩

lemma *att-orD1*: $\vdash(x1\ \#\ x2\ \oplus_\vee{}^x) \Longrightarrow \vdash x1$
⟨*proof*⟩

lemma *att-or-snd-hd*: $\vdash(a\ \#\ list\ \oplus_\vee{}^{(aa,\ b)}) \Longrightarrow snd(attack\ a) \subseteq b$
⟨*proof*⟩

lemma *att-or-singleton*[*rule-format*]:
$\vdash([x1]\ \oplus_\vee{}^x) \longrightarrow \vdash([]\ \oplus_\vee{}^{(fst\ x\ -\ fst\ (attack\ x1),\ snd\ x)})$
⟨*proof*⟩

lemma *att-orD2*[*rule-format*]:
$\vdash(x1\ \#\ x2\ \oplus_\vee{}^x) \longrightarrow\ \vdash\ (x2\ \oplus_\vee{}^{(fst\ x\ -\ fst(attack\ x1),\ snd\ x)})$
⟨*proof*⟩

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*⟩

lemma *singleton-or-lem*: $\vdash([x1]\ \oplus_\vee{}^x)\ \Longrightarrow fst\ x \subseteq fst(attack\ x1)$
⟨*proof*⟩

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*⟩

lemma *or-att-fst-sup*:
assumes $(\vdash ((x1\ \#\ x2\ \oplus_\vee{}^x)::('s :: state)\ attree))$
shows $((\bigcup\ y::'s\ attree \in set\ (x1\ \#\ x2).\ fst\ (attack\ y)) \supseteq fst(x))$
⟨*proof*⟩

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*]: $\vdash x1 \longrightarrow (\forall\ x \in fst(attack\ x1).$
$(\exists\ y.\ y \in snd(attack\ x1) \wedge x \rightarrow_i{}* y))$

First attack tree induction

⟨*proof*⟩

lemma *att-elem-seq0*: $\vdash x1 \Longrightarrow (\forall\ x \in fst(attack\ x1).$
$(\exists\ y.\ y \in snd(attack\ x1) \wedge x \rightarrow_i * y))$
⟨*proof*⟩

2.5 Valid refinements

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

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

lemma *ref-valI*: $A \sqsubseteq A' \Longrightarrow\ \vdash A' \Longrightarrow \vdash_V A$
⟨*proof*⟩

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*]:
$\forall\ sl\ x.\ sl \neq [] \longrightarrow sl\ !\ (length\ sl - Suc\ (0)) = x$
$\longrightarrow (l\ @\ sl)\ !\ (length\ l + length\ sl - Suc\ (0)) = x$
⟨*proof*⟩

lemma *nth-app-eq1*[*rule-format*]: $i < length\ sla \Longrightarrow (sla\ @\ sl)\ !\ i = sla\ !\ i$
⟨*proof*⟩

lemma *nth-app-eq1-rev*: $i < length\ sla \Longrightarrow\ sla\ !\ i = (sla\ @\ sl)\ !\ i$
⟨*proof*⟩

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

lemma *tl-ne-ex*[*rule-format*]: $l \neq [] \longrightarrow (?\ x\ .\ l = x\ \#\ (tl\ l))$
⟨*proof*⟩

lemma *tl-nempty-lngth*[*rule-format*]: *tl sl* ≠ [] ⟶ *2* ≤ *length*(*sl*)
⟨*proof*⟩

lemma *list-app-one-length*: *length l* = *n* ⟹ (*l* @ [*s*]) ! *n* = *s*
⟨*proof*⟩

lemma *tl-lem1*[*rule-format*]: *l* ≠ [] ⟶ *tl l* = [] ⟶ *length l* = *1*
⟨*proof*⟩

lemma *nth-tl-length*[*rule-format*]: *tl sl* ≠ [] ⟶
tl sl ! (*length* (*tl sl*) − *Suc* (*0*)) = *sl* ! (*length sl* − *Suc* (*0*))
⟨*proof*⟩

lemma *nth-tl-length1*[*rule-format*]: *tl sl* ≠ [] ⟶
tl sl ! *n* = *sl* ! (*n* + *1*)
⟨*proof*⟩

lemma *ineq1*: *i* < *length sla* − *n* ⟹
(*0*) ≤ *n* ⟹ *i* < *length sla*
⟨*proof*⟩

lemma *ineq2*[*rule-format*]: *length sla* ≤ *i* ⟶ *i* + (*1*) − *length sla* = *i* − *length sla* + *1*
⟨*proof*⟩

lemma *ineq3*: *tl sl* ≠ [] ⟹ *length sla* ≤ *i* ⟹ *i* < *length* (*sla* @ *tl sl*) − (*1*)
⟹ *i* − *length sla* + (*1*) < *length sl* − (*1*)
⟨*proof*⟩

lemma *tl-eq1*[*rule-format*]: *sl* ≠ [] ⟶ *tl sl* ! (*0*) = *sl* ! *Suc* (*0*)
⟨*proof*⟩

lemma *tl-eq2*[*rule-format*]: *tl sl* = [] ⟶ *sl* ! (*0*) = *sl* ! (*length sl* − (*1*))
⟨*proof*⟩

lemma *tl-eq3*[*rule-format*]: *tl sl* ≠ [] ⟶
tl sl ! (*length sl* − *Suc* (*Suc* (*0*))) = *sl* ! (*length sl* − *Suc* (*0*))
⟨*proof*⟩

lemma *nth-app-eq3*: **assumes** *tl sl* ≠ []
shows (*sla* @ *tl sl*) ! (*length* (*sla* @ *tl sl*) − (*1*)) = *sl* ! (*length sl* − (*1*))
⟨*proof*⟩

lemma *not-empty-ex*: *A* ≠ {} ⟹ *?* *x*. *x* ∈ *A*
⟨*proof*⟩

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

⟨*proof*⟩

lemma *list-eq1*[*rule-format*]: $sl \neq [] \longrightarrow$
$(fst\ x\ \#\ sl)\ !\ (length\ (fst\ x\ \#\ sl) - (1)) = sl\ !\ (length\ sl - (1))$
⟨*proof*⟩

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

lemma *fst-lem1*[*rule-format*]: $\forall\ (a::\ 's\ set)\ b\ (c::\ 's\ set)\ d.\ (a,\ c) = (b,\ d) \longrightarrow a = b$
⟨*proof*⟩

lemma *fst-eq1*: $(sla\ !\ (0),\ y) = attack\ x1 \Longrightarrow sla\ !\ (0) = fst\ (attack\ x1)$
⟨*proof*⟩

lemma *base-att-lem1*: $y0 \subseteq y1 \Longrightarrow \vdash \mathcal{N}_{(y1,\ y)} \Longrightarrow \vdash \mathcal{N}_{(y0,\ y)}$
⟨*proof*⟩

lemma *ref-pres-att*: $A \sqsubseteq A' \Longrightarrow attack\ A = attack\ A'$
⟨*proof*⟩

lemma *base-subset*:
assumes $xa \subseteq xc$
shows $\vdash \mathcal{N}_{(x,\ xa)} \Longrightarrow \vdash \mathcal{N}_{(x,\ xc)}$
⟨*proof*⟩

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*⟩

theorem *ATV-EF*: $[\![\vdash_V A;\ (I,s) = attack\ A\ ]\!] \Longrightarrow$
$(Kripke\ \{s.\ \exists\ i \in I.\ (i \rightarrow_i * s)\ \}\ I\ \vdash EF\ s)$
⟨*proof*⟩

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 num-

ber 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*:
$Kripke\ \{s :: ('s :: state).\ \exists\ i \in I.\ (i \rightarrow_i* s)\}\ I \vdash EF\ s$
$\Longrightarrow \forall\ x \in I.\ \exists\ y \in s.\ x \rightarrow_i* y$
⟨*proof*⟩

3.3.2 Lemma *Compl-step2*

First, we prove some auxiliary lemmas.

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

⟨*proof*⟩

lemma *tl-nempty-length*[*rule-format*]: $s \neq [] \longrightarrow tl\ s \neq [] \longrightarrow 0 < length\ s - 1$
⟨*proof*⟩

lemma *tl-nempty-length2*[*rule-format*]: $s \neq [] \longrightarrow tl\ s \neq [] \longrightarrow Suc\ 0 < length\ s$
⟨*proof*⟩

lemma *length-last*[*rule-format*]: $(l\ @\ [x])\ !\ (length\ (l\ @\ [x]) - 1) = x$
⟨*proof*⟩

lemma *Compl-step2*: $\forall\ x \in I.\ \exists\ y \in s.\ x \rightarrow_i* y \Longrightarrow$
$(\forall\ x \in I.\ x \in s \vee (\exists\ (sl :: (((('s :: state)\ set)list)).$
$(sl \neq []) \wedge (tl\ sl \neq []) \wedge$
$(sl\ !\ 0,\ sl\ !\ (length\ sl - 1)) = (\{x\},s) \wedge$
$(\forall\ i < (length\ sl - 1).\ \vdash \mathcal{N}_{(sl\ !\ i, sl\ !\ (i+1))}$
$)))$
⟨*proof*⟩

3.3.3 Lemma *Compl-step3*

First, we need a few lemmas.

lemma *map-hd-lem*[*rule-format*] : $n > 0 \longrightarrow (f\ 0\ \#\ \ map\ (\lambda i.\ f\ i)\ [1..<n]) = map\ \ (\lambda i.\ f\ i)\ [0..<n]$
⟨*proof*⟩

lemma *map-Suc-lem*[*rule-format*] : $n > 0 \longrightarrow map\ (\lambda\ i::\ nat.\ f\ i)[1..<n] =$
$map\ (\lambda\ i::\ nat.\ f(Suc\ i))[0..<(n - 1)]$
⟨*proof*⟩

lemma *forall-ex-fun*: *finite* $S \Longrightarrow (\forall\ x \in S.\ (\exists\ y.\ P\ y\ x)) \longrightarrow (\exists\ f.\ \forall\ x \in S.\ P\ (f\ x)\ x)$
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *nodup-all-tl*[*rule-format*]: *nodup-all* (*x* # *l*) ⟶ *nodup-all l*
⟨*proof*⟩

lemma *finite-nodup*: *finite I* ⟹ ∃ *l*. *set l* = *I* ∧ *nodup-all l*
⟨*proof*⟩

lemma *Compl-step3*: $I \neq \{\} \Longrightarrow$ *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*⟩

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* = []
⟨*proof*⟩

lemma *list-two-tl-not-empty*[*rule-format*]: ∀ *list*. *length l* = *Suc* (*Suc* (*length list*)) ⟶ *tl l* ≠ []
⟨*proof*⟩

lemma *or-empty*: $\vdash([] \oplus_\vee^{(\{\},\ s)})$ ⟨*proof*⟩

Note, this is not valid for any l, i.e., $\vdash l \oplus_\vee^{(\{\},\ s)}$ is not a theorem.

lemma *list-or-upt*[*rule-format*]:
$\forall\ l\ .\ lI \neq [] \longrightarrow length\ l = length\ lI \longrightarrow nodup\text{-}all\ lI \longrightarrow$
$(\forall\ i < length\ lI.\ (\vdash (l\ !\ i)) \wedge (attack\ (l\ !\ i) = (\{lI\ !\ i\},\ s)))$
$\longrightarrow (\vdash (l \oplus_\vee^{(set\ lI,\ s)}))$
⟨*proof*⟩

lemma *app-tl-empty-hd*[*rule-format*]: $tl\ (l\ @\ [a]) = [] \longrightarrow hd\ (l\ @\ [a]) = a$
⟨*proof*⟩

lemma *tl-hd-empty*[*rule-format*]: $tl\ (l\ @\ [a]) = [] \longrightarrow l = []$
⟨*proof*⟩

lemma *tl-hd-not-empty*[*rule-format*]: $tl\ (l\ @\ [a]) \neq [] \longrightarrow l \neq []$
⟨*proof*⟩

lemma *app-tl-empty-length*[*rule-format*]: $tl\ (map\ f\ [0..<length\ l]\ @\ [a]) = []$
$\Longrightarrow l = []$
⟨*proof*⟩

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

lemma *app-tl-hd-list*[*rule-format*]: $tl\ (map\ f\ [0..<length\ l]\ @\ [a]) \neq []$
$\Longrightarrow hd(map\ f\ [0..<length\ l]\ @\ [a]) = (map\ f\ [0..<length\ l])\ !\ 0$
⟨*proof*⟩

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

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

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*⟩

lemma *step-lem2a*[*rule-format*]: $0 < length\ list \Longrightarrow map\ (\lambda i.\ \mathcal{N}_{((x1\ \#\ a\ \#\ list)\ !\ i,\ (a\ \#\ list)\ !\ i)})$
$[0..<length\ list]\ @$
$[\mathcal{N}_{((x1\ \#\ a\ \#\ list)\ !\ length\ list,\ (a\ \#\ list)\ !\ length\ list)}] =$
$aa\ \#\ listb \longrightarrow \mathcal{N}_{((x1,\ a))} = aa$
⟨*proof*⟩

lemma *step-lem2b*[*rule-format*]: $0 = length\ list \Longrightarrow map\ (\lambda i.\ \mathcal{N}_{((x1\ \#\ a\ \#\ list)\ !\ i,\ (a\ \#\ list)\ !\ i)})$
$[0..<length\ list]\ @$
$[\mathcal{N}_{((x1\ \#\ a\ \#\ list)\ !\ length\ list,\ (a\ \#\ list)\ !\ length\ list)}] =$
$aa\ \#\ listb \longrightarrow \mathcal{N}_{((x1,\ a))} = aa$
⟨*proof*⟩

lemma *step-lem2*: $map\ (\lambda i.\ \mathcal{N}_{((x1\ \#\ a\ \#\ list)\ !\ i,\ (a\ \#\ list)\ !\ i)})$
$[0..<length\ list]\ @$
$[\mathcal{N}_{((x1\ \#\ a\ \#\ list)\ !\ length\ list,\ (a\ \#\ list)\ !\ length\ list)}] =$
$aa\ \#\ listb \Longrightarrow \mathcal{N}_{((x1,\ a))} = aa$
⟨*proof*⟩

lemma *base-list-and*[*rule-format*]: $Sji \neq [] \longrightarrow tl\ Sji \neq [] \longrightarrow$
$(\forall\ li.\ \ Sji\ !\ (0) = li \longrightarrow$
$Sji!\ (length\ (Sji) - 1) = s \longrightarrow$
$(\forall\, i{<}length\ (Sji) - 1.\ \vdash \mathcal{N}_{(Sji\ !\ i,\ Sji\ !\ Suc\ i)}) \longrightarrow$
$\vdash (map\ (\lambda i.\ \mathcal{N}_{(Sji\ !\ i,\ Sji\ !\ Suc\ i)})$
$[0..<length\ (Sji) - Suc\ (0)]\ \oplus_{\wedge}{}^{(li,\ s)}))$
⟨*proof*⟩

lemma *Compl-step4*: $I \neq \{\} \Longrightarrow finite\ I \Longrightarrow \neg\ I \subseteq s \Longrightarrow$
$(\exists\ lI.\ set\ lI = \{x.\ x \in I \wedge x \notin s\} \wedge (\exists\ Sj :: (((\prime s :: state)\ set)list)\ list.$
$length\ Sj = length\ lI \wedge nodup\text{-}all\ lI \wedge$
$(\forall\ j < length\ Sj.\ (((Sj\ !\ j)\ \neq []) \wedge (tl\ (Sj\ !\ j) \neq []) \wedge$
$((Sj\ !\ j)\ !\ 0,\ (Sj\ !\ j)\ !\ (length\ (Sj\ !\ j) - 1)) = (\{lI\ !\ j\},s) \wedge$
$(\forall\ i < (length\ (Sj\ !\ j) - 1).\ \vdash \mathcal{N}_{((Sj\ !\ j)\ !\ i,\ (Sj\ !\ j)\ !\ (i+1)\ )}$
$)))))$
$\Longrightarrow\ \exists\ (A :: (\prime s :: state)\ attree).\ \vdash A \wedge attack\ A = (I,s)$
⟨*proof*⟩

3.3.5 Main Theorem Completeness

theorem *Completeness*: $I \neq \{\} \Longrightarrow finite\ I \Longrightarrow$
$Kripke\ \{s :: (\prime s :: state).\ \exists\ i \in I.\ (i \rightarrow_i{*}\ s)\}\ (I :: (\prime s :: state)set)\ \vdash EF\ s$
$\Longrightarrow \exists\ (A :: (\prime s :: state)\ attree).\ \vdash A \wedge attack\ A = (I,s)$
⟨*proof*⟩

3.3.6 Contrapositions of Correctness and Completeness

lemma *contrapos-compl*:
$I \neq \{\} \Longrightarrow finite\ I \Longrightarrow$
$(\neg\ (\exists\ (A :: (\prime s :: state)\ attree).\ \vdash A \wedge attack\ A = (I, -\ s))) \Longrightarrow$
$\neg\ (Kripke\ \{s.\ \exists\, i{\in}I.\ i \rightarrow_i{*}\ s\}\ I \vdash EF\ (-\ s))$
⟨*proof*⟩

lemma *contrapos-corr*:
$(\neg(Kripke\ \{s :: (\prime s :: state).\ \exists\ i \in I.\ (i \rightarrow_i{*}\ s)\}\ I\ \vdash EF\ s))$

$\Longrightarrow$ *attack A* = (*I*,*s*)
$\Longrightarrow$ ¬ ($\vdash$ *A*)
⟨*proof*⟩

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

Predicates and projections for the labels to encode their meaning.

definition *owner* :: *dlm* $*$ *data* $\Rightarrow$ *actor* **where** *owner d* $\equiv$ *fst*(*fst d*)
definition *owns* :: [*igraph*, *location*, *actor*, *dlm* $*$ *data*] $\Rightarrow$ *bool*
where *owns G l a d* $\equiv$ *owner d* = *a*
definition *readers* :: *dlm* $*$ *data* $\Rightarrow$ *actor set*
where *readers d* $\equiv$ *snd* (*fst d*)

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

definition *has-access* :: [*igraph*, *location*, *actor*, *dlm* $*$ *data*] $\Rightarrow$ *bool*
where *has-access G l a d* $\equiv$ *owns G l a d* $\vee$ *a* $\in$ *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* $\Rightarrow$ *dlm* $*$ *data*.
$\forall$ *x*:: *dlm* $*$ *data*. *fst x* = *fst* (*f x*)}
⟨*proof*⟩

definition *secure-process* :: *label-fun* $\Rightarrow$ *dlm* $*$ *data* $\Rightarrow$ *dlm* $*$ *data* (**infixr** $\Updownarrow$ *50*)
where *f* $\Updownarrow$ *d* $\equiv$ (*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*] $\Rightarrow$ *bool* (- $@_{(-)}$ - *50*)
where *a* $@_G$ *l* $\equiv$ *a* $\in$ (*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*] $\Rightarrow$ *bool*
where
enables I l a a′ $\equiv$ ($\exists$ (*p*,*e*) $\in$ *delta I* (*graphI I*) *l*. *a′* $\in$ *e* $\wedge$ *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* ⟹

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

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: $(i \rightarrow_i i') = (i \rightarrow_n (i' :: infrastructure))$

instance
⟨*proof*⟩

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

end

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

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*⟩

lemma *step1r*: *gdpr-scenario* →$_n$∗ *gdpr-scenario′*
⟨*proof*⟩

lemma *step2*: *gdpr-scenario′* →$_n$ *gdpr-scenario″*
⟨*proof*⟩

lemma *step2r*: *gdpr-scenario′* →$_n$∗ *gdpr-scenario″*
⟨*proof*⟩

For the Kripke structure

$gdpr\text{-}Kripke \equiv Kripke\ \{\ I.\ gdpr\text{-}scenario \rightarrow_i * I\ \}\ \{gdpr\text{-}scenario\}$
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*⟩

lemma *att-gdpr*: $\vdash([\mathcal{N}_{(Igdpr,GDPR')}, \mathcal{N}_{(GDPR',sgdpr)}] \oplus_\wedge{}^{(Igdpr,sgdpr)})$
⟨*proof*⟩

lemma *gdpr-abs-att*: $\vdash_V([\mathcal{N}_{(Igdpr,sgdpr)}] \oplus_\wedge{}^{(Igdpr,sgdpr)})$
⟨*proof*⟩

We can then simply apply the Correctness theorem *AT EF* to immediately prove the following CTL statement.
$gdpr\text{-}Kripke \vdash EF\ sgdpr$
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*: $gdpr\text{-}Kripke \vdash EF\ \{x.\ \neg(global\text{-}policy'\ x\ ''Eve'')\}$
⟨*proof*⟩

theorem *gdpr-EF*: $gdpr\text{-}Kripke \vdash EF\ sgdpr$
⟨*proof*⟩

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*: $\exists\ A. \vdash A \wedge attack\ A = (Igdpr,sgdpr)$
⟨*proof*⟩

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'*: $gdpr\text{-}Kripke \vdash EF\ sgdpr$
⟨*proof*⟩

[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*.

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 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*: $h \in gdpr\text{-}actors \Longrightarrow l \in gdpr\text{-}locations \Longrightarrow$
$owns\ (Igraph\ gdpr\text{-}scenario)\ l\ (Actor\ h)\ d \Longrightarrow$
$gdpr\text{-}Kripke \vdash AG\ \{x.\ \forall\ l \in gdpr\text{-}locations.\ owns\ (Igraph\ x)\ l\ (Actor\ h)\ d\}$
⟨*proof*⟩

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*⟩
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/.