Network Security Policy Verification

Cornelius Diekmann

March 17, 2025

Abstract. We present a unified theory for verifying network security policies. A security policy is represented as directed graph. To check high-level security goals, security invariants over the policy are expressed. We cover monotonic security invariants, i.e. prohibiting more does not harm security. We provide the following contributions for the security invariant theory. (*i*) Secure auto-completion of scenario-specific knowledge, which eases usability. (*ii*) Security violations can be repaired by tightening the policy iff the security invariants hold for the deny-all policy. (*iii*) An algorithm to compute a security policy. (*iv*) A formalization of stateful connection semantics in network security mechanisms. (*v*) An algorithm to compute a secure stateful implementation of a policy. (*vi*) An executable implementation of all the theory. (*vii*) Examples, ranging from an aircraft cabin data network to the analysis of a large real-world firewall.

For a detailed description, see [2, 3, 1].

Acknowledgements. This entry contains contributions by Lars Hupel and would not have made it into the AFP without him. I want to thank the Isabelle group Munich for always providing valuable help. I would like to express my deep gratitude to my supervisor, Georg Carle, for supporting this topic and facilitating further research possibilities in this field.

Contents

theory *TopoS-Vertices*
imports *Main*
HOL−Library.Char-ord
HOL−Library.List-Lexorder
begin

1 A type for vertices

This theory makes extensive use of graphs. We define a typeclass *vertex* for the vertices we will use in our theory. The vertices will correspond to network or policy entities.

Later, we will conduct some proves by providing counterexamples. Therefore, we say that the type of a vertex has at least three pairwise distinct members.

For example, the types *string*, *nat*, *bool* × *bool* and many other fulfill this assumption. The type *bool* alone does not fulfill this assumption, because it only has two elements.

This is only a constraint over the type, of course, a policy with less than three entities can also be verified.

TL;DR: We define *′a vertex*, which is as good as *′a*.

```
class vertex =
  fixes vertex-1 :: 'a
  fixes vertex-2 :: 'a
  fixes vertex-3 :: 'a
  assumes distinct-vertices: distinct [vertex-1, vertex-2, vertex-3]
begin
  lemma distinct-vertices12[simp]: vertex-1 ≠ vertex-2 using distinct-vertices by(simp)
  lemma distinct-vertices13[simp]: vertex-1 ≠ vertex-3 using distinct-vertices by(simp)
  lemma distinct-vertices23[simp]: vertex-2 ≠ vertex-3 using distinct-vertices by(simp)

  lemmas distinct-vertices-sym = distinct-vertices12[symmetric] distinct-vertices13[symmetric]
        distinct-vertices23[symmetric]
  declare distinct-vertices-sym[simp]
end
```

Numbers, chars and strings are good candidates for vertices.

```
instantiation nat::vertex
begin
  definition vertex-1-nat ::nat where vertex-1 ≡ (1::nat)
  definition vertex-2-nat ::nat where vertex-2 ≡ (2::nat)
  definition vertex-3-nat ::nat where vertex-3 ≡ (3::nat)
instance proof qed(simp add: vertex-1-nat-def vertex-2-nat-def vertex-3-nat-def)
end
value vertex-1::nat

instantiation int::vertex
begin
  definition vertex-1-int ::int where vertex-1 ≡ (1::int)
  definition vertex-2-int ::int where vertex-2 ≡ (2::int)
  definition vertex-3-int ::int where vertex-3 ≡ (3::int)
instance proof qed(simp add: vertex-1-int-def vertex-2-int-def vertex-3-int-def)
end

instantiation char::vertex
```

begin
  definition *vertex-1-char* ::*char* **where** *vertex-1* ≡ *CHR ''A''*
  definition *vertex-2-char* ::*char* **where** *vertex-2* ≡ *CHR ''B''*
  definition *vertex-3-char* ::*char* **where** *vertex-3* ≡ *CHR ''C''*
instance proof(*intro-classes*) **qed**(*simp add*: *vertex-1-char-def vertex-2-char-def vertex-3-char-def*)
end
value *vertex-1*::*char*

instantiation *list* :: (*vertex*) *vertex*
begin
  definition *vertex-1-list* **where** *vertex-1* ≡ []
  definition *vertex-2-list* **where** *vertex-2* ≡ [*vertex-1*]
  definition *vertex-3-list* **where** *vertex-3* ≡ [*vertex-1*, *vertex-1*]
instance proof qed(*simp add*: *vertex-1-list-def vertex-2-list-def vertex-3-list-def*)
end

— for the ML graphviz visualizer
ML ‹

```
fun tune-string-vertex-format (t: term) (s: string) : string =
  if fastype-of t = @{typ string} then
    if String.isPrefix '' s then
      String.substring (s, (size ''), (size s − (size '''')))
    else let val - = writeln (no tune-string-vertex-format for \^s^\) in s end
  else s
  handle Subscript => let val - = writeln (tune-string-vertex-format Subscript excpetion) in s end;
```

›

end
theory *TopoS-Interface*
imports *Main Lib/FiniteGraph TopoS-Vertices Lib/TopoS-Util*
begin

2 Security Invariants

A good documentation of this formalization is available in [3].

We define security invariants over a graph. The graph corresponds to the network's access control structure.

  record (*'v*::*vertex*, *'a*) *TopoS-Params* =
    node-properties :: *'v*::*vertex* ⇒ *'a option*

A Security Invariant is defined as locale.

We successively define more and more locales with more and more assumptions. This clearly depicts which assumptions are necessary to use certain features of a Security Invariant. In addition, it makes instance proofs of Security Invariants easier, since the lemmas obtained by an (easy, few assumptions) instance proof can be used for the complicated (more assumptions) instance proofs.

A security Invariant consists of one function: *sinvar*. Essentially, it is a predicate over the policy (depicted as graph G and a host attribute mapping (nP)).

A Security Invariant where the offending flows (flows that invalidate the policy) can be defined and calculated. No assumptions are necessary for this step.

```
locale SecurityInvariant-withOffendingFlows =
  fixes sinvar::('v::vertex) graph ⇒ ('v::vertex ⇒ 'a) ⇒ bool — policy ⇒ host attribute mapping ⇒
bool
 begin
  — Offending Flows definitions:
  definition is-offending-flows::('v × 'v) set ⇒ 'v graph ⇒ ('v ⇒ 'a) ⇒ bool where
   is-offending-flows f G nP ≡ ¬ sinvar G nP ∧ sinvar (delete-edges G f) nP

  — Above definition is not minimal:
  definition is-offending-flows-min-set::('v × 'v) set ⇒ 'v graph ⇒ ('v ⇒ 'a) ⇒ bool where
   is-offending-flows-min-set f G nP ≡ is-offending-flows f G nP ∧
    (∀ (e1, e2) ∈ f. ¬ sinvar (add-edge e1 e2 (delete-edges G f)) nP)

  — The set of all offending flows.
  definition set-offending-flows::'v graph ⇒ ('v ⇒ 'a) ⇒ ('v × 'v) set set where
   set-offending-flows G  nP = {F. F ⊆ (edges G) ∧ is-offending-flows-min-set F G nP}
```

Some of the *set-offending-flows* definition

```
  lemma offending-not-empty: ⟦ F ∈ set-offending-flows G nP ⟧ ⟹ F ≠ {}
   by(auto simp add: set-offending-flows-def is-offending-flows-def is-offending-flows-min-set-def)
  lemma empty-offending-contra:
     ⟦ F ∈ set-offending-flows G nP; F = {}⟧ ⟹ False
   by(simp add: set-offending-flows-def is-offending-flows-def is-offending-flows-min-set-def)
  lemma offending-notevalD: F ∈ set-offending-flows G nP ⟹ ¬ sinvar G nP
   by(simp add: set-offending-flows-def is-offending-flows-def is-offending-flows-min-set-def)
  lemma sinvar-no-offending: sinvar G nP ⟹ set-offending-flows G nP = {}
    by(simp add: set-offending-flows-def is-offending-flows-def is-offending-flows-min-set-def)
  theorem removing-offending-flows-makes-invariant-hold:
    ∀ F ∈ set-offending-flows G nP. sinvar (delete-edges G F) nP
    proof(cases sinvar G nP)
     case True
      hence no-offending: set-offending-flows G nP = {} using sinvar-no-offending by simp
      thus ∀ F∈set-offending-flows G nP. sinvar (delete-edges G F) nP using empty-iff by simp
     next
     case False thus ∀ F∈set-offending-flows G nP. sinvar (delete-edges G F) nP
    by(simp add: set-offending-flows-def is-offending-flows-def is-offending-flows-min-set-def graph-ops)
    qed
 corollary valid-without-offending-flows:
 ⟦ F ∈ set-offending-flows G nP ⟧ ⟹ sinvar (delete-edges G F) nP
  by(simp add: removing-offending-flows-makes-invariant-hold)

 lemma set-offending-flows-simp:
  ⟦ wf-graph G ⟧ ⟹
   set-offending-flows G nP = {F. F ⊆ edges G ∧
    (¬ sinvar G nP ∧ sinvar (|nodes = nodes G, edges = edges G − F|) nP) ∧
    (∀ (e1, e2)∈F. ¬ sinvar (|nodes = nodes G, edges = {(e1, e2)} ∪ (edges G − F)|) nP)}
  apply(simp only: set-offending-flows-def is-offending-flows-min-set-def
   is-offending-flows-def delete-edges-simp2 add-edge-def graph.select-convs)
  apply(subgoal-tac ⋀F e1 e2. F ⊆ edges G ⟹ (e1, e2) ∈ F ⟹ nodes G ∪ {e1, e2} = nodes G)
   apply fastforce
  apply(simp add: wf-graph-def)
  by (metis fst-conv imageI in-mono insert-absorb snd-conv)
```

end

print-locale! *SecurityInvariant-withOffendingFlows*

The locale *SecurityInvariant-withOffendingFlows* has no assumptions about the security invariant *sinvar*. Undesirable things may happen: The offending flows can be empty, even for a violated invariant.

We provide an example, the security invariant λ- -. *False*. As host attributes, we simply use the identity function *id*.

lemma *SecurityInvariant-withOffendingFlows.set-offending-flows* (λ- -. *False*) ⦇ *nodes* = {''*v1*''}, *edges*={} ⦈ *id* = {}
lemma *SecurityInvariant-withOffendingFlows.set-offending-flows* (λ- -. *False*)
⦇ *nodes* = {''*v1*'', ''*v2*''}, *edges* = {(''*v1*'', ''*v2*'')} ⦈ *id* = {}

In general, there exists a *sinvar* such that the invariant does not hold and no offending flows exits.

lemma ∃ *sinvar*. ¬ *sinvar G nP* ∧ *SecurityInvariant-withOffendingFlows.set-offending-flows sinvar G nP* = {}
apply(*simp add*: *SecurityInvariant-withOffendingFlows.set-offending-flows-def*
SecurityInvariant-withOffendingFlows.is-offending-flows-min-set-def SecurityInvariant-withOffendingFlows.is-offendin
apply(*rule-tac x*=(λ- -. *False*) **in** *exI*)
apply(*simp*)
done

Thus, we introduce usefulness properties that prohibits such useless invariants.

We summarize them in an invariant. It requires the following:

1. The offending flows are always defined.
2. The invariant is monotonic, i.e. prohibiting more is more secure.
3. And, the (non-minimal) offending flows are monotonic, i.e. prohibiting more solves more security issues.

Later, we will show that is suffices to show that the invariant is monotonic. The other two properties can be derived.

locale *SecurityInvariant-preliminaries* = *SecurityInvariant-withOffendingFlows sinvar*
for *sinvar*
\+
assumes
defined-offending:
⟦ *wf-graph G*; ¬ *sinvar G nP* ⟧ ⟹ *set-offending-flows G nP* ≠ {}
and
mono-sinvar:
⟦ *wf-graph* ⦇ *nodes* = *N*, *edges* = *E* ⦈; *E'* ⊆ *E*; *sinvar* ⦇ *nodes* = *N*, *edges* = *E* ⦈ *nP* ⟧ ⟹
sinvar ⦇ *nodes* = *N*, *edges* = *E'* ⦈ *nP*
and *mono-offending*:
⟦ *wf-graph G*; *is-offending-flows ff G nP* ⟧ ⟹ *is-offending-flows* (*ff* ∪ *f'*) *G nP*

begin

To instantiate a *SecurityInvariant-preliminaries*, here are some hints: Have a look at the *TopoS-withOffendingFlows.thy* file. There is a definition of *sinvar-mono*. It impplies *mono-sinvar* and *mono-offending* *apply*(*fact SecurityInvariant-withOffendingFlows.sinvar-mono-imp-sinvar-mono*[*OF sinvar-mono*]) *apply*(*fact SecurityInvariant-withOffendingFlows.sinvar-mono-imp-is-offending-flows-mono*[*OF sinvar-mono*])

In addition, *SecurityInvariant-withOffendingFlows.mono-imp-set-offending-flows-not-empty*[*OF sinvar-mono*] gives a nice proof rule for *defined-offending*

Basically, *sinvar-mono*. implies almost all assumptions here and is equal to *mono-sinvar*.

end

2.1 Security Invariants with secure auto-completion of host attribute mappings

We will now add a new artifact to the Security Invariant. It is a secure default host attribute, we will use the symbol ⊥.

The newly introduced Boolean *receiver-violation* tells whether a security violation happens at the sender's or the receiver's side.

The details can be looked up in [3].

locale *SecurityInvariant* = *SecurityInvariant-preliminaries sinvar*
for *sinvar*::('*v*::*vertex*) *graph* ⇒ ('*v*::*vertex* ⇒ '*a*) ⇒ *bool*
\+
fixes *default-node-properties* :: '*a* (‹⊥›)
and *receiver-violation* :: *bool*
assumes
— default value can never fix a security violation.
— Idea: Assume there is a violation, then there is some offending flow. *receiver-violation* defines whether the violation happens at the sender's or the receiver's side. We call the place of the violation the *offending host*. We replace the host attribute of the offending host with the default attribute. Giving an offending host, a *secure* default attribute does not change whether the invariant holds. I.e. this reconfiguration does not remove information, thus preserves all security critical information. Thought experiment preliminaries: Can a default configuration ever solve an existing security violation? NO! Thought experiment 1: admin forgot to configure host, hence it is handled by default configuration value ... Thought experiment 2: new node (attacker) is added to the network. What is its default configuration value ...

default-secure:
⟦ *wf-graph G*; ¬ *sinvar G nP*; *F* ∈ *set-offending-flows G nP* ⟧ ⟹
(¬ *receiver-violation* ⟶ *i* ∈ *fst* ' *F* ⟶ ¬ *sinvar G* (*nP*(*i* := ⊥))) ∧
(*receiver-violation* ⟶ *i* ∈ *snd* ' *F* ⟶ ¬ *sinvar G* (*nP*(*i* := ⊥)))
and
default-unique:
otherbot ≠ ⊥ ⟹
∃ (*G*::('*v*::*vertex*) *graph*) *nP i F*. *wf-graph G* ∧ ¬ *sinvar G nP* ∧ *F* ∈ *set-offending-flows G nP* ∧
sinvar (*delete-edges G F*) *nP* ∧
(¬ *receiver-violation* ⟶ *i* ∈ *fst* ' *F* ∧ *sinvar G* (*nP*(*i* := *otherbot*))) ∧
(*receiver-violation* ⟶ *i* ∈ *snd* ' *F* ∧ *sinvar G* (*nP*(*i* := *otherbot*)))
begin
— Removes option type, replaces with default host attribute
fun *node-props* :: ('*v*, '*a*) *TopoS-Params* ⇒ ('*v* ⇒ '*a*) **where**
node-props P = (λ *i*. (*case* (*node-properties P*) *i of Some property* ⇒ *property* | *None* ⇒ ⊥))

definition *node-props-formaldef* :: ($'v$, $'a$) *TopoS-Params* ⇒ ($'v$ ⇒ $'a$) **where**
node-props-formaldef P ≡
(λ *i*. (*if i* ∈ *dom* (*node-properties P*) *then the* (*node-properties P i*) *else* ⊥))

lemma *node-props-eq-node-props-formaldef*: *node-props-formaldef* = *node-props*
by(*simp add*: *fun-eq-iff node-props-formaldef-def option.case-eq-if domIff*)

Checking whether a security invariant holds.

1. check that the policy *G* is syntactically valid
2. check the security invariant *sinvar*

definition *eval*::$'v$ *graph* ⇒ ($'v$, $'a$) *TopoS-Params* ⇒ *bool* **where**
eval G P ≡ *wf-graph G* ∧ *sinvar G* (*node-props P*)

lemma *unique-common-math-notation*:
assumes ∀ *G nP i F*. *wf-graph* (*G*::($'v$::*vertex*) *graph*) ∧ ¬ *sinvar G nP* ∧ *F* ∈ *set-offending-flows G nP* ∧
sinvar (*delete-edges G F*) *nP* ∧
(¬ *receiver-violation* ⟶ *i* ∈ *fst* ‘ *F* ⟶ ¬ *sinvar G* (*nP*(*i* := *otherbot*))) ∧
(*receiver-violation* ⟶ *i* ∈ *snd* ‘ *F* ⟶ ¬ *sinvar G* (*nP*(*i* := *otherbot*)))
shows *otherbot* = ⊥
apply(*rule ccontr*)
apply(*drule default-unique*)
using *assms* **by** *blast*
end

print-locale! *SecurityInvariant*

2.2 Information Flow Security and Access Control

receiver-violation defines the offending host. Thus, it defines when the violation happens. We found that this coincides with the invariant's security strategy.

ACS If the violation happens at the sender, we have an access control strategy (*ACS*). I.e. the sender does not have the appropriate rights to initiate the connection.

IFS If the violation happens at the receiver, we have an information flow security strategy (*IFS*) I.e. the receiver lacks the appropriate security level to retrieve the (confidential) information. The violations happens only when the receiver reads the data.

We refine our *SecurityInvariant* locale.

2.3 Information Flow Security Strategy (IFS)

locale *SecurityInvariant-IFS* = *SecurityInvariant-preliminaries sinvar*
for *sinvar*::($'v$::*vertex*) *graph* ⇒ ($'v$::*vertex* ⇒ $'a$) ⇒ *bool*
\+
fixes *default-node-properties* :: $'a$ (‹⊥›)
assumes *default-secure-IFS*:
⟦ *wf-graph G*; *f* ∈ *set-offending-flows G nP* ⟧ ⟹

```
      ∀ i ∈ snd` f. ¬ sinvar G (nP(i := ⊥))
    and
    — If some otherbot fulfills default-secure, it must be ⊥ Hence, ⊥ is uniquely defined
    default-unique-IFS:
    (∀ G f nP i. wf-graph G ∧ f ∈ set-offending-flows G nP ∧ i ∈ snd` f
            ⟶ ¬ sinvar G (nP(i := otherbot))) ⟹ otherbot = ⊥
    begin
      lemma default-unique-EX-notation: otherbot ≠ ⊥ ⟹
        ∃ G nP i f. wf-graph G ∧ ¬ sinvar G nP ∧ f ∈ set-offending-flows G nP ∧
         sinvar (delete-edges G f) nP ∧
         (i ∈ snd` f ∧ sinvar G (nP(i := otherbot)))
        apply(erule contrapos-pp)
        apply(simp)
         using default-unique-IFS SecurityInvariant-withOffendingFlows.valid-without-offending-flows
offending-notevalD
        by metis
    end

  sublocale SecurityInvariant-IFS ⊆ SecurityInvariant where receiver-violation=True
  apply(unfold-locales)
   apply(simp add: default-secure-IFS)
  apply(simp only: HOL.simp-thms)
  apply(drule default-unique-EX-notation)
  apply(assumption)
  done


  locale SecurityInvariant-IFS-otherDirectrion = SecurityInvariant where receiver-violation=True
  sublocale SecurityInvariant-IFS-otherDirectrion ⊆ SecurityInvariant-IFS
  apply(unfold-locales)
   apply (metis default-secure offending-notevalD)
  apply(erule contrapos-pp)
  apply(simp)
  apply(drule default-unique)
  apply(simp)
  apply(blast)
  done


lemma default-uniqueness-by-counterexample-IFS:
  assumes (∀ G F nP i. wf-graph G ∧ F ∈ SecurityInvariant-withOffendingFlows.set-offending-flows
sinvar G nP ∧ i ∈ snd` F
            ⟶ ¬ sinvar G (nP(i := otherbot)))
  and otherbot ≠ default-value ⟹
   ∃ G nP i F. wf-graph G ∧ ¬ sinvar G nP ∧ F ∈ (SecurityInvariant-withOffendingFlows.set-offending-flows
sinvar G nP) ∧
       sinvar (delete-edges G F) nP ∧
        i ∈ snd ` F ∧ sinvar G (nP(i := otherbot))
   shows otherbot = default-value
   using assms by blast
```

2.4 Access Control Strategy (ACS)

```
  locale SecurityInvariant-ACS = SecurityInvariant-preliminaries sinvar
```

for *sinvar*::('*v*::*vertex*) *graph* ⇒ ('*v*::*vertex* ⇒ '*a*) ⇒ *bool*
+
fixes *default-node-properties* :: '*a* (‹⊥›)
assumes *default-secure-ACS*:
⟦ *wf-graph G*; *f* ∈ *set-offending-flows G nP* ⟧ ⟹
∀ *i* ∈ *fst*‘ *f*. ¬ *sinvar G* (*nP*(*i* := ⊥))
and
default-unique-ACS:
(∀ *G f nP i*. *wf-graph G* ∧ *f* ∈ *set-offending-flows G nP* ∧ *i* ∈ *fst*‘ *f*
⟶ ¬ *sinvar G* (*nP*(*i* := *otherbot*))) ⟹ *otherbot* = ⊥
begin
lemma *default-unique-EX-notation*: *otherbot* ≠ ⊥ ⟹
∃ *G nP i f*. *wf-graph G* ∧ ¬ *sinvar G nP* ∧ *f* ∈ *set-offending-flows G nP* ∧
sinvar (*delete-edges G f*) *nP* ∧
(*i* ∈ *fst*‘ *f* ∧ *sinvar G* (*nP*(*i* := *otherbot*)))
apply(*erule contrapos-pp*)
apply(*simp*)
using *default-unique-ACS SecurityInvariant-withOffendingFlows.valid-without-offending-flows offending-notevalD*
by *metis*
end

sublocale *SecurityInvariant-ACS* ⊆ *SecurityInvariant* **where** *receiver-violation*=*False*
apply(*unfold-locales*)
apply(*simp add*: *default-secure-ACS*)
apply(*simp only*: *HOL.simp-thms*)
apply(*drule default-unique-EX-notation*)
apply(*assumption*)
done

locale *SecurityInvariant-ACS-otherDirectrion* = *SecurityInvariant* **where** *receiver-violation*=*False*
sublocale *SecurityInvariant-ACS-otherDirectrion* ⊆ *SecurityInvariant-ACS*
apply(*unfold-locales*)
apply (*metis default-secure offending-notevalD*)
apply(*erule contrapos-pp*)
apply(*simp*)
apply(*drule default-unique*)
apply(*simp*)
apply(*blast*)
done

lemma *default-uniqueness-by-counterexample-ACS*:
assumes (∀ *G F nP i*. *wf-graph G* ∧ *F* ∈ *SecurityInvariant-withOffendingFlows.set-offending-flows sinvar G nP* ∧ *i* ∈ *fst* ‘ *F*
⟶ ¬ *sinvar G* (*nP*(*i* := *otherbot*)))
and *otherbot* ≠ *default-value* ⟹
∃ *G nP i F*. *wf-graph G* ∧ ¬ *sinvar G nP* ∧ *F* ∈ (*SecurityInvariant-withOffendingFlows.set-offending-flows sinvar G nP*) ∧
sinvar (*delete-edges G F*) *nP* ∧
i ∈ *fst* ‘ *F* ∧ *sinvar G* (*nP*(*i* := *otherbot*))
shows *otherbot* = *default-value*

using *assms* **by** *blast*

The sublocale relationships tell that the simplified *SecurityInvariant-ACS* and *SecurityInvariant-IFS* assumptions suffice to do tho generic SecurityInvariant assumptions.

end
theory *TopoS-withOffendingFlows*
imports *TopoS-Interface*
begin

3 *SecurityInvariant* Instantiation Helpers

The security invariant locales are set up hierarchically to ease instantiation proofs. The first locale, *SecurityInvariant-withOffendingFlows* has no assumptions, thus instantiations is for free. The first step focuses on monotonicity,

context *SecurityInvariant-withOffendingFlows*
begin

We define the monotonicity of *sinvar*:

$\bigwedge nP\ N\ E'\ E.\ [\![wf\text{-}graph\ (\!|nodes = N,\ edges = E|\!);\ E' \subseteq E;\ sinvar\ (\!|nodes = N,\ edges = E|\!)\ nP]\!] \Longrightarrow sinvar\ (\!|nodes = N,\ edges = E'|\!)\ nP$

Having a valid invariant, removing edges retains the validity. I.e. prohibiting more, is more or equally secure.

```
definition sinvar-mono :: bool where
  sinvar-mono ⟷ (∀ nP N E' E.
    wf-graph (| nodes = N, edges = E |) ∧
    E' ⊆ E ∧
    sinvar (| nodes = N, edges = E |) nP ⟶ sinvar (| nodes = N, edges = E' |) nP )
```

If one can show *sinvar-mono*, then the instantiation of the *SecurityInvariant-preliminaries* locale is tremendously simplified.

```
lemma sinvar-mono-I-proofrule-simple:
⟦ (∀ G nP. sinvar G nP = (∀ (e1, e2) ∈ edges G. P e1 e2 nP) ) ⟧ ⟹ sinvar-mono
apply(simp add: sinvar-mono-def)
apply(clarify)
apply(fast)
done

lemma sinvar-mono-I-proofrule:
⟦ (∀ nP (G:: 'v graph). sinvar G nP = (∀ (e1, e2) ∈ edges G. P e1 e2 nP G) );
  (∀ nP e1 e2 N E' E.
    wf-graph (| nodes = N, edges = E |) ∧
    (e1,e2) ∈ E ∧
    E' ⊆ E ∧
   P e1 e2 nP (|nodes = N, edges = E|) ⟶ P e1 e2 nP (|nodes = N, edges = E'|)) ⟧ ⟹ sinvar-mono
unfolding sinvar-mono-def
proof(clarify)
  fix nP N E' E
  assume AllForm: (∀ nP (G:: 'v graph). sinvar G nP = (∀ (e1, e2) ∈ edges G. P e1 e2 nP G) )
  and Pmono: ∀ nP e1 e2 N E' E. wf-graph (| nodes = N, edges = E |) ∧ (e1,e2) ∈ E ∧ E' ⊆ E ∧
P e1 e2 nP (|nodes = N, edges = E|) ⟶ P e1 e2 nP (|nodes = N, edges = E'|)
  and wfG: wf-graph (|nodes = N, edges = E|)
```

and *E'subset*: $E' \subseteq E$
and *evalE*: *sinvar* (|*nodes* = *N*, *edges* = *E*|) *nP*

from *Pmono* **have** *Pmono1*:
$\bigwedge$*nP N E' E*. *wf-graph* (| *nodes* = *N*, *edges* = *E* |) $\Longrightarrow$ $E' \subseteq E$ $\Longrightarrow$ ($\forall$(*e1*,*e2*) $\in$ *E*. *P e1 e2 nP* (|*nodes* = *N*, *edges* = *E*|) $\longrightarrow$ *P e1 e2 nP* (|*nodes* = *N*, *edges* = *E'*|))
by *blast*

from *AllForm* **have** *sinvar* (|*nodes* = *N*, *edges* = *E*|) *nP* = ($\forall$ (*e1*, *e2*) $\in$ *E*. *P e1 e2 nP* (|*nodes* = *N*, *edges* = *E*|)) **by** *force*
from *this evalE* **have** ($\forall$ (*e1*, *e2*) $\in$ *E*. *P e1 e2 nP* (|*nodes* = *N*, *edges* = *E*|)) **by** *simp*
from *Pmono1*[*OF wfG E'subset*, *of nP*] *this* **have** $\forall$(*e1*, *e2*) $\in$ *E*. *P e1 e2 nP* (|*nodes* = *N*, *edges* = *E'*|) **by** *fast*
from *this E'subset* **have** $\forall$(*e1*, *e2*) $\in$ *E'*. *P e1 e2 nP* (|*nodes* = *N*, *edges* = *E'*|) **by** *fast*
from *this* **have** $\forall$(*e1*, *e2*) $\in$ (*edges* (|*nodes* = *N*, *edges* = *E'*|)). *P e1 e2 nP* (|*nodes* = *N*, *edges* = *E'*|) **by** *simp*
from *this AllForm* **show** *sinvar* (|*nodes* = *N*, *edges* = *E'*|) *nP* **by** *presburger*
qed

Invariant violations do not disappear if we add more flows.

lemma *sinvar-mono-imp-negative-mono*:
sinvar-mono $\Longrightarrow$ *wf-graph* (| *nodes* = *N*, *edges* = *E* |) $\Longrightarrow$ $E' \subseteq E$ $\Longrightarrow$
$\neg$ *sinvar* (| *nodes* = *N*, *edges* = *E'* |) *nP* $\Longrightarrow$ $\neg$ *sinvar* (| *nodes* = *N*, *edges* = *E* |) *nP*
unfolding *sinvar-mono-def* **by**(*blast*)

corollary *sinvar-mono-imp-negative-delete-edge-mono*:
sinvar-mono $\Longrightarrow$ *wf-graph G* $\Longrightarrow$ $X \subseteq Y$ $\Longrightarrow$ $\neg$ *sinvar* (*delete-edges G Y*) *nP* $\Longrightarrow$ $\neg$ *sinvar* (*delete-edges G X*) *nP*
proof –
assume *sinvar-mono*
and *wf-graph G* **and** $X \subseteq Y$ **and** $\neg$ *sinvar* (*delete-edges G Y*) *nP*
from *delete-edges-wf*[*OF* ‹*wf-graph G*›] **have** *valid-G-delete*: *wf-graph* (|*nodes* = *nodes G*, *edges* = *edges G* − *X*|) **by**(*simp add*: *delete-edges-simp2*)
from ‹$X \subseteq Y$› **have** *edges G* − *Y* $\subseteq$ *edges G* − *X* **by** *blast*
with ‹*sinvar-mono*› *sinvar-mono-def valid-G-delete* **have**
sinvar (|*nodes* = *nodes G*, *edges* = *edges G* − *X*|) *nP* $\Longrightarrow$ *sinvar* (|*nodes* = *nodes G*, *edges* = *edges G* − *Y*|) *nP* **by** *blast*
hence *sinvar* (*delete-edges G X*) *nP* $\Longrightarrow$ *sinvar* (*delete-edges G Y*) *nP* **by**(*simp add*: *delete-edges-simp2*)
with ‹$\neg$ *sinvar* (*delete-edges G Y*) *nP*› **show** *?thesis* **by** *blast*
qed

lemma *sinvar-mono-imp-is-offending-flows-mono*:
assumes *mono*: *sinvar-mono*
and *wfG*: *wf-graph G*
shows *is-offending-flows FF G nP* $\Longrightarrow$ *is-offending-flows* (*FF* $\cup$ *F*) *G nP*
proof –
from *wfG* **have** *wfG'*: *wf-graph* (|*nodes* = *nodes G*, *edges* = {(*e1*, *e2*). (*e1*, *e2*) $\in$ *edges G* $\wedge$ (*e1*, *e2*) $\notin$ *FF*}|)
by (*metis delete-edges-def delete-edges-wf*)

from *mono* **have** *sinvarE*: ($\bigwedge$ *nP N E' E*. *wf-graph* ⦇ *nodes* = *N*, *edges* = *E* ⦈ $\Longrightarrow$ *E'* $\subseteq$ *E* $\Longrightarrow$ *sinvar* ⦇ *nodes* = *N*, *edges* = *E* ⦈ *nP* $\Longrightarrow$ *sinvar* ⦇ *nodes* = *N*, *edges* = *E'* ⦈ *nP*)
unfolding *sinvar-mono-def*
by *metis*
have $\bigwedge$ *G FF F*. {(*e1*, *e2*). (*e1*, *e2*) $\in$ *edges G* $\wedge$ (*e1*, *e2*) $\notin$ *FF* $\wedge$ (*e1*, *e2*) $\notin$ *F*} $\subseteq$ {(*e1*, *e2*). (*e1*, *e2*) $\in$ *edges G* $\wedge$ (*e1*, *e2*) $\notin$ *FF*}
by(*rule Collect-mono*) (*simp*)
from *sinvarE*[*OF wfG' this*]
show *is-offending-flows FF G nP* $\Longrightarrow$ *is-offending-flows* (*FF* $\cup$ *F*) *G nP*
by(*simp add*: *is-offending-flows-def delete-edges-def*)
qed

lemma *sinvar-mono-imp-sinvar-mono*:
sinvar-mono $\Longrightarrow$ *wf-graph* ⦇ *nodes* = *N*, *edges* = *E* ⦈ $\Longrightarrow$ *E'* $\subseteq$ *E* $\Longrightarrow$ *sinvar* ⦇ *nodes* = *N*, *edges* = *E* ⦈ *nP* $\Longrightarrow$
sinvar ⦇ *nodes* = *N*, *edges* = *E'* ⦈ *nP*
apply(*simp add*: *sinvar-mono-def*)
by *blast*

end

3.1 Offending Flows Not Empty Helper Lemmata

context *SecurityInvariant-withOffendingFlows*
begin

Give an over-approximation of offending flows (e.g. all edges) and get back a minimal set

fun *minimalize-offending-overapprox* :: ('*v* $\times$ '*v*) *list* $\Rightarrow$ ('*v* $\times$ '*v*) *list* $\Rightarrow$
'*v graph* $\Rightarrow$ ('*v* $\Rightarrow$ '*a*) $\Rightarrow$ ('*v* $\times$ '*v*) *list* **where**
minimalize-offending-overapprox [] *keep* - - = *keep* |
minimalize-offending-overapprox (*f*#*fs*) *keep G nP* = (*if sinvar* (*delete-edges-list G* (*fs*@*keep*)) *nP then*
minimalize-offending-overapprox fs keep G nP
else
minimalize-offending-overapprox fs (*f*#*keep*) *G nP*
)

The graph we check in *minimalize-offending-overapprox*, *G* (−) (*fs* $\cup$ *keep*) is the graph from the *offending-flows-min-set* condition. We add *f* and remove it.

lemma *minimalize-offending-overapprox-subset*:
set (*minimalize-offending-overapprox ff keeps G nP*) $\subseteq$ *set ff* $\cup$ *set keeps*
proof(*induction ff arbitrary*: *keeps*)
case *Nil*
thus *?case* **by** *simp*
next
case (*Cons a ff*)
from *Cons* **have** *case1*: (*sinvar* (*delete-edges-list G* (*ff* @ *keeps*)) *nP* $\Longrightarrow$
set (*minimalize-offending-overapprox ff keeps G nP*) $\subseteq$ *insert a* (*set ff* $\cup$ *set keeps*))
by *blast*
from *Cons* **have** *case2*: ($\neg$ *sinvar* (*delete-edges-list G* (*ff* @ *keeps*)) *nP* $\Longrightarrow$
set (*minimalize-offending-overapprox ff* (*a* # *keeps*) *G nP*) $\subseteq$ *insert a* (*set ff* $\cup$ *set keeps*))
by *fastforce*
from *case1 case2* **show** *?case* **by** *simp*

qed

lemma *not-model-mono-imp-addedge-mono*:
assumes *mono*: *sinvar-mono*
and *vG*: *wf-graph G* **and** *ain*: (*a1*,*a2*) ∈ *edges G* **and** *xy*: *X* ⊆ *Y* **and** *ns*: ¬ *sinvar* (*add-edge a1 a2* (*delete-edges G* (*Y*))) *nP*
shows ¬ *sinvar* (*add-edge a1 a2* (*delete-edges G X*)) *nP*
proof −
have *wf-graph-add-delete-edge-simp*:
⋀*Y*. *add-edge a1 a2* (*delete-edges G Y*) = (*delete-edges G* (*Y* − {(*a1*,*a2*)}))
apply(*simp add*: *delete-edges-simp2 add-edge-def*)
apply(*rule conjI*)
using *ain* **apply** (*metis insert-absorb vG wf-graph.E-wfD*(*1*) *wf-graph.E-wfD*(*2*))
apply(*auto simp add*: *ain*)
done
from *this ns* **have** *1*: ¬ *sinvar* (*delete-edges G* (*Y* − {(*a1*, *a2*)})) *nP* **by** *simp*
have *2*: *X* − {(*a1*, *a2*)} ⊆ *Y* − {(*a1*, *a2*)} **by** (*metis Diff-mono subset-refl xy*)
from *sinvar-mono-imp-negative-delete-edge-mono*[*OF mono*] *vG* **have**
⋀*X Y*. *X* ⊆ *Y* ⟹ ¬ *sinvar* (*delete-edges G Y*) *nP* ⟹ ¬ *sinvar* (*delete-edges G X*) *nP* **by** *blast*
from *this*[*OF 2 1*] **have** ¬ *sinvar* (*delete-edges G* (*X* − {(*a1*, *a2*)})) *nP* **by** *simp*
from *this wf-graph-add-delete-edge-simp*[*symmetric*] **show** *?thesis* **by** *simp*
qed

theorem *is-offending-flows-min-set-minimalize-offending-overapprox*:
assumes *mono*: *sinvar-mono*
and *vG*: *wf-graph G* **and** *iO*: *is-offending-flows* (*set ff*) *G nP* **and** *sF*: *set ff* ⊆ *edges G* **and** *dF*: *distinct ff*
shows *is-offending-flows-min-set* (*set* (*minimalize-offending-overapprox ff* [] *G nP*)) *G nP*
(**is** *is-offending-flows-min-set ?minset G nP*)
proof −
from *iO* **have** *sinvar* (*delete-edges G* (*set ff*)) *nP* **by** (*metis is-offending-flows-def*)

— *sinvar* holds if we delete *ff*. With the following generalized statement, we show that it also holds if we delete *minimalize-offending-overapprox ff* []
{
fix *keeps*
— Generalized for arbitrary *keeps*
have *sinvar* (*delete-edges G* (*set ff* ∪ *set keeps*)) *nP* ⟹
sinvar (*delete-edges G* (*set* (*minimalize-offending-overapprox ff keeps G nP*))) *nP*
apply(*induction ff arbitrary*: *keeps*)
apply(*simp*)
apply(*simp*)
apply(*rule impI*)
apply(*simp add*:*delete-edges-list-union*)
done
}
— *keeps* = []
note *minimalize-offending-overapprox-maintains-evalmodel*=*this*[*of* []]

from ‹*sinvar* (*delete-edges G* (*set ff*)) *nP*› *minimalize-offending-overapprox-maintains-evalmodel*
have
sinvar (*delete-edges G ?minset*) *nP* **by** *simp*
hence *1*: *is-offending-flows ?minset G nP* **by** (*metis iO is-offending-flows-def*)

We need to show minimality of *minimalize-offending-overapprox ff* []. Minimality means ∀(*e1*, *e2*)∈*set* (*minimalize-offending-overapprox ff* [] *G nP*). ¬ *sinvar* (*add-edge e1 e2* (*delete-edges G* (*set* (*minimalize-offending-overapprox ff* [] *G nP*)))) *nP*. We show the following generalized fact.

{
fix *ff keeps*
have ∀ *x* ∈ *set ff*. *x* ∉ *set keeps* ⟹
∀ *x* ∈ *set ff*. *x* ∈ *edges G* ⟹
distinct ff ⟹
∀(*e1*, *e2*)∈ *set keeps*.
¬ *sinvar* (*add-edge e1 e2* (*delete-edges G* (*set* (*minimalize-offending-overapprox ff keeps G nP*)))) *nP* ⟹
∀(*e1*, *e2*)∈*set* (*minimalize-offending-overapprox ff keeps G nP*).
¬ *sinvar* (*add-edge e1 e2* (*delete-edges G* (*set* (*minimalize-offending-overapprox ff keeps G nP*)))) *nP*
proof(*induction ff arbitrary*: *keeps*)
case *Nil*
from *Nil* **show** *?case* **by**(*simp*)
next
case (*Cons a ff*)
assume *not-in-keeps*: ∀*x*∈*set* (*a* # *ff*). *x* ∉ *set keeps*
hence *a-not-in-keeps*: *a* ∉ *set keeps* **by** *simp*
assume *in-edges*: ∀*x*∈*set* (*a* # *ff*). *x* ∈ *edges G*
hence *ff-in-edges*: ∀*x*∈*set ff*. *x* ∈ *edges G* **and** *a-in-edges*: *a* ∈ *edges G* **by** *simp-all*
assume *distinct*: *distinct* (*a* # *ff*)
hence *ff-distinct*: *distinct ff* **and** *a-not-in-ff*: *a* ∉ *set ff* **by** *simp-all*
assume *minimal*: ∀(*e1*, *e2*)∈*set keeps*.
¬ *sinvar* (*add-edge e1 e2* (*delete-edges G* (*set* (*minimalize-offending-overapprox* (*a* # *ff*) *keeps G nP*)))) *nP*

have *delete-edges-list-union-insert*: ⋀ *f fs keep*. *delete-edges-list G* (*f*#*fs*@*keep*) = *delete-edges G* ({*f*} ∪ *set fs* ∪ *set keep*)
by(*simp add*: *graph-ops delete-edges-list-set*)

let *?goal*=*?case* — we show this by case distinction
show *?case*
proof(*cases sinvar* (*delete-edges-list G* (*ff*@*keeps*)) *nP*)
case *True*
from *True* **have** *sinvar* (*delete-edges-list G* (*ff*@*keeps*)) *nP* **.**
from *this Cons* **show** *?goal* **using** *delete-edges-list-union* **by** *simp*
next
case *False*

{ — a lemma we only need once here
fix *a ff keeps*
assume *mono*: *sinvar-mono* **and** *ankeeps*: *a* ∉ *set keeps*
and *anff*: *a* ∉ *set ff* **and** *aE*: *a* ∈ *edges G*
and *nsinvar*: ¬ *sinvar* (*delete-edges-list G* (*ff* @ *keeps*)) *nP*
have ¬ *sinvar* (*add-edge* (*fst a*) (*snd a*) (*delete-edges G* (*set* (*minimalize-offending-overapprox*

(a # ff) keeps G nP)))) nP
proof −
{ **fix** F Fs keep
from vG **have** F ∈ edges G ⟹ F ∉ set Fs ⟹ F ∉ set keep ⟹
(add-edge (fst F) (snd F) (delete-edges-list G (F#Fs@keep))) = (delete-edges-list G (Fs@keep))
apply(simp add:delete-edges-list-union delete-edges-list-union-insert)
apply(simp add: graph-ops)
apply(rule conjI)
apply(simp add: wf-graph-def)
apply blast
apply(simp add: wf-graph-def)
by fastforce
} **note** delete-edges-list-add-add-iff=this
from aE **have** (fst a, snd a) ∈ edges G **by** simp
from delete-edges-list-add-add-iff[of a ff keeps] **have**
delete-edges-list G (ff @ keeps) = add-edge (fst a) (snd a) (delete-edges-list G (a # ff @ keeps))
by (metis aE anff ankeeps)
from this nsinvar **have** ¬ sinvar (add-edge (fst a) (snd a) (delete-edges-list G (a # ff @ keeps))) nP **by** simp
from this delete-edges-list-union-insert **have** 1:
¬ sinvar (add-edge (fst a) (snd a) (delete-edges G (insert a (set ff ∪ set keeps)))) nP **by** (metis insert-is-Un sup-assoc)

from minimalize-offending-overapprox-subset[of ff a#keeps G nP] **have**
set (minimalize-offending-overapprox ff (a # keeps) G nP) ⊆ insert a (set ff ∪ set keeps) **by** simp

from not-model-mono-imp-addedge-mono[OF mono vG ‹(fst a, snd a) ∈ edges G› this 1] **show** ?thesis
by (metis minimalize-offending-overapprox.simps(2) nsinvar)
qed
} **note** not-model-mono-imp-addedge-mono-minimalize-offending-overapprox=this

from not-model-mono-imp-addedge-mono-minimalize-offending-overapprox[OF mono a-not-in-keeps a-not-in-ff a-in-edges False] **have** a-minimal:
¬ sinvar (add-edge (fst a) (snd a) (delete-edges G (set (minimalize-offending-overapprox (a # ff) keeps G nP)))) nP
by simp
from minimal a-minimal
have a-keeps-minimal: ∀ (e1, e2)∈set (a # keeps).
¬ sinvar (add-edge e1 e2 (delete-edges G (set (minimalize-offending-overapprox ff (a # keeps) G nP)))) nP
using False **by** fastforce
from Cons.prems **have** a-not-in-keeps: ∀ x∈set ff. x ∉ set (a#keeps) **by** auto
from Cons.IH[OF a-not-in-keeps ff-in-edges ff-distinct a-keeps-minimal] **have** IH:
∀ (e1, e2)∈set (minimalize-offending-overapprox ff (a # keeps) G nP).
¬ sinvar (add-edge e1 e2 (delete-edges G (set (minimalize-offending-overapprox ff (a # keeps) G nP)))) nP .

from False **have** ¬ sinvar (delete-edges G (set ff ∪ set keeps)) nP **using** delete-edges-list-union **by** metis
from this **have** set (minimalize-offending-overapprox (a # ff) keeps G nP) =

```
        set (minimalize-offending-overapprox ff (a#keeps) G nP)
         by(simp add: delete-edges-list-union)
       from this IH have ?goal by presburger
       thus ?goal .
     qed
   qed
 } note mono-imp-minimalize-offending-overapprox-minimal=this[of ff []]

 from mono-imp-minimalize-offending-overapprox-minimal[OF - - dF] sF have 2:
   ∀(e1, e2)∈?minset. ¬ sinvar (add-edge e1 e2 (delete-edges G ?minset)) nP
 by auto
 from 1 2 show ?thesis
   by(simp add: is-offending-flows-def is-offending-flows-min-set-def)
qed

corollary mono-imp-set-offending-flows-not-empty:
assumes mono-sinvar: sinvar-mono
and vG: wf-graph G and iO: is-offending-flows (set ff) G nP and sS: set ff ⊆ edges G and dF:
distinct ff
shows
  set-offending-flows G nP ≠ {}
proof −
  from iO SecurityInvariant-withOffendingFlows.is-offending-flows-def have nS: ¬ sinvar G nP by
metis
  from sinvar-mono-imp-negative-delete-edge-mono[OF mono-sinvar] have negative-delete-edge-mono:

   ∀ G nP X Y. wf-graph G ∧ X ⊆ Y ∧ ¬ sinvar (delete-edges G (Y)) nP ⟶ ¬ sinvar (delete-edges
G X) nP by blast

  from is-offending-flows-min-set-minimalize-offending-overapprox[OF mono-sinvar vG iO sS dF]
   have is-offending-flows-min-set (set (minimalize-offending-overapprox ff [] G nP)) G nP by simp
  from this set-offending-flows-def sS have
    (set (minimalize-offending-overapprox ff [] G nP)) ∈ set-offending-flows G nP
    using minimalize-offending-overapprox-subset[where keeps=[]] by fastforce
  thus ?thesis by blast
 qed
```

To show that *set-offending-flows* is not empty, the previous corollary ⟦*sinvar-mono*; *wf-graph ?G*; *is-offending-flows* (*set ?ff*) *?G ?nP*; *set ?ff* ⊆ *edges ?G*; *distinct ?ff*⟧ ⟹ *set-offending-flows ?G ?nP* ≠ {} is very useful. Just select *set ff* = *edges G*.

If there exists a security violations, there a means to fix it if and only if the network in which nobody communicates with anyone fulfills the security requirement

```
 theorem valid-empty-edges-iff-exists-offending-flows:
  assumes mono: sinvar-mono and wfG: wf-graph G and noteval: ¬ sinvar G nP
  shows sinvar (| nodes = nodes G, edges = {} |) nP ⟷ set-offending-flows G nP ≠ {}
 proof
   assume a: sinvar (| nodes = nodes G, edges = {} |) nP

   from finite-distinct-list[OF wf-graph.finiteE] wfG
   obtain list-edges where list-edges-props: set list-edges = edges G ∧ distinct list-edges by blast
   hence listedges-subseteq-edges: set list-edges ⊆ edges G by blast

    have empty-edge-graph-simp: (delete-edges G (edges G)) = (| nodes = nodes G, edges = {} |)
```

by(*auto simp add*: *graph-ops*)
from *a is-offending-flows-def noteval list-edges-props empty-edge-graph-simp*
have *overapprox*: *is-offending-flows* (*set list-edges*) *G nP* **by** *auto*

from *mono-imp-set-offending-flows-not-empty*[*OF mono wfG overapprox listedges-subseteq-edges*] *list-edges-props*
show *set-offending-flows G nP* ≠ {} **by** *simp*
next
assume *a*: *set-offending-flows G nP* ≠ {}

from *a* **obtain** *f* **where** *f-props*: *f* ⊆ *edges G* ∧ *is-offending-flows-min-set f G nP* **using** *set-offending-flows-def* **by** *fastforce*

from *f-props* **have** *sinvar* (*delete-edges G f*) *nP* **using** *is-offending-flows-min-set-def is-offending-flows-def* **by** *simp*
hence *evalGf*: *sinvar* (|*nodes* = *nodes G*, *edges* = {(*e1*, *e2*). (*e1*, *e2*) ∈ *edges G* ∧ (*e1*, *e2*) ∉ *f*}|) *nP* **by**(*simp add*: *delete-edges-def*)
from *delete-edges-wf*[*OF wfG*, *unfolded delete-edges-def*]
have *wfGf*: *wf-graph* (|*nodes* = *nodes G*, *edges* = {(*e1*, *e2*). (*e1*, *e2*) ∈ *edges G* ∧ (*e1*, *e2*) ∉ *f*}|) **by** *simp*
have *emptyseqGf*: {} ⊆ {(*e1*, *e2*). (*e1*, *e2*) ∈ *edges G* ∧ (*e1*, *e2*) ∉ *f*} **by** *simp*

from *mono*[*unfolded sinvar-mono-def*] *evalGf wfGf emptyseqGf* **have** *sinvar* (|*nodes* = *nodes G*, *edges* = {}|) *nP* **by** *blast*
thus *sinvar* (|*nodes* = *nodes G*, *edges* = {}|) *nP* **.**
qed

minimalize-offending-overapprox not only computes a set where *is-offending-flows-min-set* holds, but it also returns a subset of the input.

lemma *minimalize-offending-overapprox-keeps-keeps*: (*set keeps*) ⊆ *set* (*minimalize-offending-overapprox ff keeps G nP*)
proof(*induction ff keeps G nP rule*: *minimalize-offending-overapprox.induct*)
qed(*simp-all*)

lemma *minimalize-offending-overapprox-subseteq-input*: *set* (*minimalize-offending-overapprox ff keeps G nP*) ⊆ (*set ff*) ∪ (*set keeps*)
proof(*induction ff keeps G nP rule*: *minimalize-offending-overapprox.induct*)
case *1* **thus** *?case* **by** *simp*
next
case *2* **thus** *?case* **by**(*simp add*: *delete-edges-list-set delete-edges-simp2*) *blast*
qed

end

context *SecurityInvariant-preliminaries*
begin

sinvar-mono naturally holds in *SecurityInvariant-preliminaries*

lemma *sinvar-monoI*: *sinvar-mono*
unfolding *sinvar-mono-def* **using** *mono-sinvar* **by** *blast*

Note: due to monotonicity, the minimality also holds for arbitrary subsets

lemma assumes *wf-graph G* **and** *is-offending-flows-min-set F G nP* **and** $F \subseteq$ *edges G* **and** $E \subseteq F$ **and** $E \neq \{\}$
shows ¬ *sinvar* (| *nodes* = *nodes G*, *edges* = ((*edges G*) − *F*) ∪ *E* |) *nP*
proof −
from *sinvar-mono-imp-negative-delete-edge-mono*[*OF sinvar-monoI* ‹*wf-graph G*›] **have** *negative-delete-edge-mono*:
⋀*X Y*. *X* ⊆ *Y* ⟹ ¬ *sinvar* (| *nodes* = *nodes G*, *edges* = (*edges G*) − *Y* |) *nP* ⟹ ¬ *sinvar* (| *nodes* = *nodes G*, *edges* = *edges G* − *X* |) *nP*
using *delete-edges-simp2* **by** *metis*
from *assms*(*2*) **have** (∀ (*e1*, *e2*)∈*F*. ¬ *sinvar* (*add-edge e1 e2* (*delete-edges G F*)) *nP*)
unfolding *is-offending-flows-min-set-def* **by** *simp*
with ‹*wf-graph G*› **have** *min*: (∀ (*e1*, *e2*)∈*F*. ¬ *sinvar* (| *nodes* = *nodes G*, *edges* = ((*edges G*) − *F*) ∪ {(*e1*,*e2*)} |) *nP*)
apply(*simp add*: *delete-edges-simp2 add-edge-def*)
apply(*rule*, *rename-tac x*, *case-tac x*, *rename-tac e1 e2*, *simp*)
apply(*erule-tac x*=(*e1*, *e2*) **in** *ballE*)
apply(*simp-all*)
apply(*subgoal-tac insert e1* (*insert e2* (*nodes G*)) = *nodes G*)
apply(*simp*)
by (*metis assms*(*3*) *insert-absorb rev-subsetD wf-graph.E-wfD*(*1*) *wf-graph.E-wfD*(*2*))
from ‹*E* ≠ {}› **obtain** *e* **where** *e* ∈ *E* **by** *blast*
with *min* ‹*E* ⊆ *F*› **have** *mine*: ¬ *sinvar* (| *nodes* = *nodes G*, *edges* = ((*edges G*) − *F*) ∪ {*e*} |) *nP* **by** *fast*
have *e1*: *edges G* − (*F* − {*e*}) = *insert e* (*edges G* − *F*) **using** *DiffD2* ‹*e* ∈ *E*› *assms*(*3*) *assms*(*4*) **by** *auto*
have *e2*: *edges G* − (*F* − *E*) = ((*edges G*) − *F*) ∪ *E* **using** *assms*(*3*) *assms*(*4*) **by** *auto*
from *negative-delete-edge-mono*[**where** *Y*=*F* − {*e*} **and** *X*=*F* − *E*] ‹*e* ∈ *E*› **have**
¬ *sinvar* (|*nodes* = *nodes G*, *edges* = *edges G* − (*F* − {*e*})|) *nP* ⟹ ¬ *sinvar* (|*nodes* = *nodes G*, *edges* = *edges G* − (*F* − *E*)|) *nP* **by** *blast*
with *mine e1 e2* **show** *?thesis* **by** *simp*
qed

The algorithm *minimalize-offending-overapprox* is correct

lemma *minimalize-offending-overapprox-sound*:
⟦ *wf-graph G*; *is-offending-flows* (*set ff*) *G nP*; *set ff* ⊆ *edges G*; *distinct ff* ⟧
⟹ *is-offending-flows-min-set* (*set* (*minimalize-offending-overapprox ff* [] *G nP*)) *G nP*
using *is-offending-flows-min-set-minimalize-offending-overapprox sinvar-monoI* **by** *blast*

If ¬ *sinvar G nP* Given a list ff, (ff is distinct and a subset of G's edges) such that *sinvar* (*V*, *E* − *ff*) *nP* *minimalize-offending-overapprox* minimizes ff such that we get an offending flows Note: choosing ff = edges G is a good choice!

theorem *minimalize-offending-overapprox-gives-back-an-offending-flow*:
⟦ *wf-graph G*; *is-offending-flows* (*set ff*) *G nP*; *set ff* ⊆ *edges G*; *distinct ff* ⟧
⟹
(*set* (*minimalize-offending-overapprox ff* [] *G nP*)) ∈ *set-offending-flows G nP*
apply(*frule*(*3*) *minimalize-offending-overapprox-sound*)
apply(*simp add*: *set-offending-flows-def*)
using *minimalize-offending-overapprox-subseteq-input*[**where** *keeps*=[], *simplified*] **by** *blast*

end

A version which acts on configured security invariants. I.e. there is no type *'a* for the host attributes in it.

```
fun minimalize-offending-overapprox :: ('v graph ⇒ bool) ⇒ ('v × 'v) list ⇒ ('v × 'v) list ⇒
'v graph ⇒('v × 'v) list where
minimalize-offending-overapprox - [] keep - = keep |
minimalize-offending-overapprox m (f#fs) keep G = (if m (delete-edges-list G (fs@keep)) then
    minimalize-offending-overapprox m fs keep G
  else
    minimalize-offending-overapprox m fs (f#keep) G
  )

lemma minimalize-offending-overapprox-boundnP:
shows minimalize-offending-overapprox (λG. m G nP) fs keeps G =
         SecurityInvariant-withOffendingFlows.minimalize-offending-overapprox m fs keeps G nP
  apply(induction fs arbitrary: keeps)
  apply(simp add: SecurityInvariant-withOffendingFlows.minimalize-offending-overapprox.simps; fail)
  apply(simp add: SecurityInvariant-withOffendingFlows.minimalize-offending-overapprox.simps)
  done

context SecurityInvariant-withOffendingFlows
begin
```

If there is a violation and there are no offending flows, there does not exist a possibility to fix the violation by tightening the policy. ⟦*sinvar-mono*; *wf-graph ?G*; ¬ *sinvar ?G ?nP*⟧ ⟹ *sinvar* (|*nodes* = *nodes ?G*, *edges* = {}|) *?nP* = (*set-offending-flows ?G ?nP* ≠ {}) already hints this.

```
    lemma mono-imp-emptyoffending-eq-nevervalid:
      ⟦ sinvar-mono; wf-graph G; ¬ sinvar G nP; set-offending-flows G nP = {}⟧ ⟹
       ¬ (∃ F ⊆ edges G. sinvar (delete-edges G F) nP)
    proof −
      assume mono: sinvar-mono
      and wfG: wf-graph G
      and a1: ¬ sinvar G nP
      and a2: set-offending-flows G nP = {}

      from wfG have wfG': wf-graph (|nodes = nodes G, edges = edges G|) by(simp add:wf-graph-def)

      from a2 set-offending-flows-def have ∀f ⊆ edges G. ¬ is-offending-flows-min-set f G nP by simp
      from this is-offending-flows-min-set-def is-offending-flows-def a1 have notdeleteconj:
        ∀f ⊆ edges G.
          ¬ sinvar (delete-edges G f) nP ∨
          ¬ ((∀(e1, e2)∈f. ¬ sinvar (add-edge e1 e2 (delete-edges G f)) nP))
      by simp
      have ∀f⊆edges G. ¬ sinvar (delete-edges G f) nP
      proof (rule allI, rule impI)
        fix f
        assume f ⊆ edges G
        from this notdeleteconj have
         ¬ sinvar (delete-edges G f) nP ∨
```

¬ ((∀ (*e1*, *e2*)∈*f*. ¬ *sinvar* (*add-edge e1 e2* (*delete-edges G f*)) *nP*)) **by** *simp*
from *this* **show** ¬ *sinvar* (*delete-edges G f*) *nP*
proof
assume ¬ *sinvar* (*delete-edges G f*) *nP* **thus** ¬ *sinvar* (*delete-edges G f*) *nP* **.**
next
assume ¬ (∀ (*e1*, *e2*)∈*f*. ¬ *sinvar* (*add-edge e1 e2* (*delete-edges G f*)) *nP*)
hence ∃ (*e1*,*e2*)∈*f*. *sinvar* (*add-edge e1 e2* (*delete-edges G f*)) *nP* **by**(*auto*)
from *this* **obtain** *e1 e2* **where** *e1e2cond*: (*e1*,*e2*)∈*f* ∧ *sinvar* (*add-edge e1 e2* (*delete-edges G f*)) *nP* **by** *blast*

from ‹*f* ⊆ *edges G*› *wfG* **have** *finite f* **apply**(*simp add*: *wf-graph-def*) **by** (*metis rev-finite-subset*)
from *this* **obtain** *listf* **where** *listf*: *set listf* = *f* ∧ *distinct listf* **by** (*metis finite-distinct-list*)

from *e1e2cond* ‹*f* ⊆ *edges G*› **have** *Geq*:
(*add-edge e1 e2* (*delete-edges G f*)) = (| *nodes* = *nodes G*, *edges* = *edges G* − *f* ∪ {(*e1*,*e2*)}|)
apply(*simp add*: *graph-ops wfG′*)
apply(*clarify*)
using *wfG*[*unfolded wf-graph-def*] **by** *force*

from *this*[*symmetric*] *add-edge-wf*[*OF delete-edges-wf*[*OF wfG*]] **have**
wf-graph (|*nodes* = *nodes G*, *edges* = *edges G* − *f* ∪ {(*e1*, *e2*)}|) **by** *simp*
from *mono this* **have** *mono″*:
⋀ *E′*. *E′* ⊆ *edges G* − *f* ∪ {(*e1*, *e2*)} ⟹
sinvar (|*nodes* = *nodes G*, *edges* = *edges G* − *f* ∪ {(*e1*, *e2*)}|) *nP* ⟹
sinvar (|*nodes* = *nodes G*, *edges* = *E′*|) *nP* **unfolding** *sinvar-mono-def* **by** *blast*

from *e1e2cond Geq* **have** *sinvar* (| *nodes* = *nodes G*, *edges* = *edges G* − *f* ∪ {(*e1*,*e2*)}|) *nP* **by** *simp*
from *this mono″* **have** *sinvar* (| *nodes* = *nodes G*, *edges* = *edges G* − *f*|) *nP* **by** *auto*
hence *overapprox*: *sinvar* (*delete-edges G f*) *nP* **by** (*simp add*: *delete-edges-simp2*)

from *a1 overapprox* **have** *is-offending-flows f G nP* **by**(*simp add*: *is-offending-flows-def*)
from *this listf* **have** *c1*: *is-offending-flows* (*set listf*) *G nP* **by**(*simp add*: *is-offending-flows-def*)
from *listf* ‹*f* ⊆ *edges G*› **have** *c2*: *set listf* ⊆ *edges G* **by** *simp*

from *mono-imp-set-offending-flows-not-empty*[*OF mono wfG c1 c2 conjunct2*[*OF listf*]] **have**

set-offending-flows G nP ≠ {} **.**
from *this a2* **have** *False* **by** *simp*

thus ¬ *sinvar* (*delete-edges G f*) *nP* **by** *simp*
qed
qed
thus *?thesis* **by** *simp*
qed
end

3.2 Monotonicity of offending flows

context *SecurityInvariant-preliminaries*

begin

If there is some F' in the offending flows of a small graph and you have a bigger graph, you can extend F' by some *Fadd* and minimality in *F* is preserved

lemma *minimality-offending-flows-mono-edges-graph-extend*:
⟦ *wf-graph* (| *nodes* = *V*, *edges* = *E* |); $E' \subseteq E$; *Fadd* ∩ E' = {}; F' ∈ *set-offending-flows* (|*nodes* = *V*, *edges* = E'|) *nP* ⟧ ⟹
(∀(*e1*, *e2*)∈F'. ¬ *sinvar* (*add-edge e1 e2* (*delete-edges* (|*nodes* = *V*, *edges* = *E* |) (F' ∪ *Fadd*))) *nP*)
proof −
assume *a1*: *wf-graph* (| *nodes* = *V*, *edges* = *E* |)
and *a2*: $E' \subseteq E$
and *a3*: *Fadd* ∩ E' = {}
and *a4*: F' ∈ *set-offending-flows* (|*nodes* = *V*, *edges* = E'|) *nP*

from *a4* **have** $F' \subseteq E'$ **by**(*simp add*: *set-offending-flows-def*)

obtain *Eadd* **where** *Eadd-prop*: E' ∪ *Eadd* = *E* **and** E' ∩ *Eadd* = {} **using** *a2* **by** *blast*

have *Fadd-notinE'*: ⋀*Fadd*. *Fadd* ∩ E' = {} ⟹ E' − (F' ∪ *Fadd*) = E' − F' **by** *blast*
from ‹$F' \subseteq E'$› *a1*[*simplified wf-graph-def*] *a2* **have** *FinV1*: *fst* ‘ $F' \subseteq V$ **and** *FinV2*: *snd* ‘ F' ⊆ *V*
proof −
from *a1* **have** *fst* ‘ *E* ⊆ *V* **by**(*simp add*: *wf-graph-def*)
with ‹$F' \subseteq E'$› *a2* **show** *fst* ‘ $F' \subseteq V$ **by** *fast*
from *a1* **have** *snd* ‘ *E* ⊆ *V* **by**(*simp add*: *wf-graph-def*)
with ‹$F' \subseteq E'$› *a2* **show** *snd* ‘ $F' \subseteq V$ **by** *fast*
qed
hence *insert-e1-e2-V*: ∀ (*e1*, *e2*) ∈ F'. *insert e1* (*insert e2 V*) = *V* **by** *auto*
hence *add-edge-F*: ∀ (*e1*, *e2*) ∈ F'. *add-edge e1 e2* (|*nodes* = *V*, *edges* = E' − F' |) = (|*nodes* = *V*, *edges* = (E' − F') ∪ {(*e1*, *e2*)}|)
by(*simp add*: *add-edge-def*)

have *Fadd-notinE'*: ⋀*Fadd*. *Fadd* ∩ E' = {} ⟹ E' − (F' ∪ *Fadd*) = E' − F' **by** *blast*
from ‹$F' \subseteq E'$› *this* **have** *Fadd-notinF*: ⋀*Fadd*. *Fadd* ∩ E' = {} ⟹ F' ∩ *Fadd* = {} **by** *blast*

have *Fadd-subseteq-Eadd*: ⋀*Fadd*. (*Fadd* ∩ E' = {} ∧ *Fadd* ⊆ *E*) = (*Fadd* ⊆ *Eadd*)
proof(*rule iffI*, *goal-cases*)
case *1* **thus** *?case* **using** *Eadd-prop a2* **by** *blast*
next
case *2* **thus** *?case* **using** *Eadd-prop a2* ‹E' ∩ *Eadd* = {}› **by** *blast*
qed

from *a4* **have** (∀(*e1*, *e2*)∈F'. ¬ *sinvar* (*add-edge e1 e2* (|*nodes* = *V*, *edges* = E' − F'|)) *nP*)
by(*simp add*: *set-offending-flows-def is-offending-flows-min-set-def delete-edges-simp2*)
with *add-edge-F* **have** *noteval-F*: ∀(*e1*, *e2*)∈F'. ¬ *sinvar* (|*nodes* = *V*, *edges* = (E' − F') ∪ {(*e1*, *e2*)}|) *nP*
by *fastforce*

have *tupleBallI*: ⋀*A P*. (⋀*e1 e2*. (*e1*, *e2*)∈*A* ⟹ *P* (*e1*, *e2*)) ⟹ *ALL* (*e1*, *e2*):*A*. *P* (*e1*, *e2*) **by** *force*
have ∀(*e1*, *e2*)∈F'. ¬ *sinvar* (|*nodes* = *V*, *edges* = (*E* − (F' ∪ *Fadd*)) ∪ {(*e1*, *e2*)}|) *nP*
proof(*rule tupleBallI*)

fix *e1 e2*
assume *f2*: $(e1, e2) \in F'$
with *a3* **have** *gFadd1*: $\neg$ *sinvar* $(\!|$*nodes* $= V$, *edges* $= (E' - (F' \cup Fadd)) \cup \{(e1, e2)\}|\!)$ *nP*
using *Fadd-notinE′ noteval-F* **by** *fastforce*

from *a1 FinV1 FinV2 a3 f2* **have** *gFadd2*:
wf-graph $(\!|$*nodes* $= V$, *edges* $= (E - (F' \cup Fadd)) \cup \{(e1, e2)\}|\!)$
by(*auto simp add*: *wf-graph-def*)
from *a2 a3 f2* **have** *gFadd3*:
$(E' - (F' \cup Fadd)) \cup \{(e1, e2)\} \subseteq (E - (F' \cup Fadd)) \cup \{(e1, e2)\}$ **by** *blast*

from *mono-sinvar*[*OF gFadd2 gFadd3*] *gFadd1*
show $\neg$ *sinvar* $(\!|$*nodes* $= V$, *edges* $= (E - (F' \cup Fadd)) \cup \{(e1, e2)\}|\!)$ *nP* **by** *blast*
qed
thus *?thesis*
apply(*simp add*: *delete-edges-simp2 Fadd-notinE′ add-edge-def*)
apply(*clarify*)
using *insert-e1-e2-V* **by** *fastforce*
qed

The minimality condition of the offending flows also holds if we increase the graph.

corollary *minimality-offending-flows-mono-edges-graph*:
$[\![$ *wf-graph* $(\!|$ *nodes* $= V$, *edges* $= E$ $|\!)$;
$E' \subseteq E$;
$F \in$ *set-offending-flows* $(\!|$*nodes* $= V$, *edges* $= E'|\!)$ *nP* $]\!] \Longrightarrow$
$\forall (e1, e2) \in F$. $\neg$ *sinvar* (*add-edge e1 e2* (*delete-edges* $(\!|$*nodes* $= V$, *edges* $= E$ $|\!)$ *F*)) *nP*
using *minimality-offending-flows-mono-edges-graph-extend*[**where** *Fadd*=$\{\}$, *simplified*] **by** *presburger*

all sets in the set of offending flows are monotonic, hence, for a larger graph, they can be extended to match the smaller graph. I.e. everything is monotonic.

theorem *mono-extend-set-offending-flows*: $[\![$ *wf-graph* $(\!|$ *nodes* $= V$, *edges* $= E$ $|\!)$; $E' \subseteq E$; $F' \in$ *set-offending-flows* $(\!|$ *nodes* $= V$, *edges* $= E'$ $|\!)$ *nP* $]\!] \Longrightarrow$
$\exists$ $F \in$ *set-offending-flows* $(\!|$ *nodes* $= V$, *edges* $= E$ $|\!)$ *nP*. $F' \subseteq F$
proof −
fix F' *V E* E'
assume *a1*: *wf-graph* $(\!|$ *nodes* $= V$, *edges* $= E$ $|\!)$
and *a2*: $E' \subseteq E$
and *a4*: $F' \in$ *set-offending-flows* $(\!|$*nodes* $= V$, *edges* $= E'|\!)$ *nP*

— Idea: $F = F' \cup$ *minimize* $(E - E')$

have $\bigwedge f$. *wf-graph* (*delete-edges* $(\!|$*nodes* $= V$, *edges* $= E|\!)$ *f*)
using *delete-edges-wf*[*OF a1*] **by** *fast*
hence *wf1*: $\bigwedge f$. *wf-graph* $(\!|$*nodes* $= V$, *edges* $= E - f|\!)$
by(*simp add*: *delete-edges-simp2*)

obtain *Eadd* **where** *Eadd-prop*: $E' \cup Eadd = E$ **and** $E' \cap Eadd = \{\}$ **using** *a2* **by** *blast*

from *a4* **have** $F' \subseteq E'$ **by**(*simp add*: *set-offending-flows-def*)

from *wf1* **have** *wf2*: *wf-graph* $(\!|$*nodes* $= V$, *edges* $= E' - F' \cup Eadd|\!)$
apply(*subgoal-tac* $E' - F' \cup Eadd = E - F'$)
apply *fastforce*

using *Eadd-prop* ‹$E' \cap Eadd = \{\}$› ‹$F' \subseteq E'$› **by** *fast*

from *a4* **have** *offending-F*: ¬ *sinvar* (|*nodes* = *V*, *edges* = *E'*|) *nP*
by(*simp add*: *set-offending-flows-def is-offending-flows-min-set-def is-offending-flows-def*)
from *this mono-sinvar*[*OF a1 a2*] **have**
goal-noteval: ¬ *sinvar* (|*nodes* = *V*, *edges* = *E*|) *nP* **by** *blast*

from *a4* **have** *eval-E-minus-FEadd-simp*: *sinvar* (|*nodes* = *V*, *edges* = *E'* − *F'*|) *nP*
by(*simp add*: *set-offending-flows-def is-offending-flows-min-set-def is-offending-flows-def delete-edges-simp2*)

show ∃ *F* ∈ *set-offending-flows* (| *nodes* = *V*, *edges* = *E* |) *nP*. *F'* ⊆ *F*
proof(*cases* ¬ *sinvar* (|*nodes* = *V*, *edges* = *E'* − *F'* ∪ *Eadd*|) *nP*)
assume *assumption-new-violation*: ¬ *sinvar* (|*nodes* = *V*, *edges* = *E'* − *F'* ∪ *Eadd*|) *nP*
from *a1* **have** *finite Eadd*
apply(*simp add*: *wf-graph-def*)
using *Eadd-prop wf-graph.finiteE* **by** *blast*
from *this* **obtain** *Eadd-list* **where** *Eadd-list-prop*: *set Eadd-list* = *Eadd* **and** *distinct Eadd-list* **by** (*metis finite-distinct-list*)
from *a1* **have** *finite E'*
apply(*simp add*: *wf-graph-def*)
using *Eadd-prop* **by** *blast*
from *this* **obtain** *E'-list* **where** *E'-list-prop*: *set E'-list* = *E'* **and** *distinct E'-list* **by** (*metis finite-distinct-list*)
from ‹*finite E'*› ‹$F' \subseteq E'$› **obtain** *F'-list* **where** *set F'-list* = *F'* **and** *distinct F'-list* **by** (*metis finite-distinct-list rev-finite-subset*)

have *E'* − *F'* ∪ *Eadd* − *Eadd* = *E'* − *F'* **using** *Eadd-prop* ‹$E' \cap Eadd = \{\}$› ‹$F' \subseteq E'$› **by** *blast*
with *assumption-new-violation eval-E-minus-FEadd-simp* **have**
is-offending-flows (*set* (*Eadd-list*)) (|*nodes* = *V*, *edges* = (*E'* − *F'*) ∪ *Eadd*|) *nP*
by (*simp add*: *Eadd-list-prop delete-edges-simp2 is-offending-flows-def*)
from *minimalize-offending-overapprox-sound*[*OF wf2 this* - ‹*distinct Eadd-list*›] **have**
is-offending-flows-min-set
(*set* (*minimalize-offending-overapprox Eadd-list* []
(|*nodes* = *V*, *edges* = *E'* − *F'* ∪ *Eadd*|) *nP*)) (|*nodes* = *V*, *edges* = *E'* − *F'* ∪ *Eadd*|) *nP*
by(*simp add*: *Eadd-list-prop*)
with *minimalize-offending-overapprox-subseteq-input*[*of Eadd-list* [] (|*nodes* = *V*, *edges* = *E'* − *F'* ∪ *Eadd*|) *nP*, *simplified Eadd-list-prop*]
obtain *Fadd* **where** *Fadd-prop*: *is-offending-flows-min-set Fadd* (|*nodes* = *V*, *edges* = *E'* − *F'* ∪ *Eadd*|) *nP* **and** *Fadd* ⊆ *Eadd* **by** *auto*

have *graph-edges-simp-helper*: *E'* − *F'* ∪ *Eadd* − *Fadd* = *E* − (*F'* ∪ *Fadd*)
using ‹$E' \cap Eadd = \{\}$› *Eadd-prop* ‹$F' \subseteq E'$› **by** *blast*

from *Fadd-prop graph-edges-simp-helper* **have**
goal-eval-Fadd: *sinvar* (*delete-edges* (|*nodes* = *V*, *edges* = *E*|) (*F'* ∪ *Fadd*)) *nP* **and**
pre-goal-minimal-Fadd: (∀ (*e1*, *e2*)∈*Fadd*. ¬ *sinvar* (*add-edge e1 e2* (*delete-edges* (|*nodes* = *V*, *edges* = *E* |) (*F'* ∪ *Fadd*))) *nP*)
by(*simp add*: *is-offending-flows-min-set-def is-offending-flows-def delete-edges-simp2*)+

from ‹$E' \cap Eadd = \{\}$› ‹$Fadd \subseteq Eadd$› **have** *Fadd* ∩ *E'* = {} **by** *blast*
from *minimality-offending-flows-mono-edges-graph-extend*[*OF a1* ‹$E' \subseteq E$› ‹$Fadd \cap E' = \{\}$›

a4]
have *mono-delete-edges-minimal*: (∀ (*e1*, *e2*)∈*F′*. ¬ *sinvar* (*add-edge e1 e2* (*delete-edges* (|*nodes* = *V*, *edges* = *E* |) (*F′* ∪ *Fadd*))) *nP*) **.**

from *mono-delete-edges-minimal pre-goal-minimal-Fadd* **have** *goal-minimal*:
∀ (*e1*, *e2*)∈*F′* ∪ *Fadd*. ¬ *sinvar* (*add-edge e1 e2* (*delete-edges* (|*nodes* = *V*, *edges* = *E*|) (*F′* ∪ *Fadd*))) *nP* **by** *fastforce*

from *Eadd-prop* ‹*Fadd* ⊆ *Eadd*› ‹*F′* ⊆ *E′*› **have** *goal-subset*: *F′* ⊆ *E* ∧ *Fadd* ⊆ *E* **by** *blast*

show ∃ *F* ∈ *set-offending-flows* (| *nodes* = *V*, *edges* = *E* |) *nP*. *F′* ⊆ *F*
apply(*simp add*: *set-offending-flows-def is-offending-flows-min-set-def is-offending-flows-def*)
apply(*rule-tac x*=*F′* ∪ *Fadd* **in** *exI*)
apply(*simp add*: *goal-noteval goal-eval-Fadd goal-minimal goal-subset*)
done
next
assume ¬ ¬ *sinvar* (|*nodes* = *V*, *edges* = *E′* − *F′* ∪ *Eadd*|) *nP*
hence *assumption-no-new-violation*: *sinvar* (|*nodes* = *V*, *edges* = *E′* − *F′* ∪ *Eadd*|) *nP* **by** *simp*
from *this* ‹*F′* ⊆ *E′*› ‹*E′* ∩ *Eadd* = {}› **have** *sinvar* (|*nodes* = *V*, *edges* = *E* − *F′*|) *nP*
proof(*subst Eadd-prop*[*symmetric*])
assume *a1*: *F′* ⊆ *E′*
assume *a2*: *E′* ∩ *Eadd* = {}
assume *a3*: *sinvar* (|*nodes* = *V*, *edges* = *E′* − *F′* ∪ *Eadd*|) *nP*
have $\bigwedge x_1.\ x_1 \cap E' - Eadd = x_1 \cap E'$
using *a2 Un-Diff-Int* **by** *auto*
hence *F′* − *Eadd* = *F′*
using *a1* **by** *auto*
hence {} ∪ (*Eadd* − *F′*) = *Eadd*
using *Int-Diff Un-Diff-Int sup-commute* **by** *auto*
thus *sinvar* (|*nodes* = *V*, *edges* = *E′* ∪ *Eadd* − *F′*|) *nP*
using *a3* **by** (*metis Un-Diff sup-bot.left-neutral*)
qed
from *this* **have** *goal-eval*: *sinvar* (*delete-edges* (|*nodes* = *V*, *edges* = *E*|) *F′*) *nP*
by(*simp add*: *delete-edges-simp2*)

from *Eadd-prop* ‹*F′* ⊆ *E′*› **have** *goal-subset*: *F′* ⊆ *E* **by**(*blast*)

from *minimality-offending-flows-mono-edges-graph*[*OF a1 a2 a4*]
have *goal-minimal*: (∀ (*e1*, *e2*)∈*F′*. ¬ *sinvar* (*add-edge e1 e2* (*delete-edges* (|*nodes* = *V*, *edges* = *E* |) *F′*)) *nP*) **.**

show ∃ *F* ∈ *set-offending-flows* (| *nodes* = *V*, *edges* = *E* |) *nP*. *F′* ⊆ *F*
apply(*simp add*: *set-offending-flows-def is-offending-flows-min-set-def is-offending-flows-def*)
apply(*rule-tac x*=*F′* **in** *exI*)
apply(*simp add*: *goal-noteval goal-subset goal-minimal goal-eval*)
done
qed
qed

The offending flows are monotonic.

corollary *offending-flows-union-mono*: ⟦ *wf-graph* (| *nodes* = *V*, *edges* = *E* |); *E′* ⊆ *E* ⟧ ⟹
⋃ (*set-offending-flows* (| *nodes* = *V*, *edges* = *E′* |) *nP*) ⊆ ⋃ (*set-offending-flows* (| *nodes* = *V*, *edges* = *E* |) *nP*)

apply(*clarify*)
apply(*drule*(*2*) *mono-extend-set-offending-flows*)
by *blast*

lemma *set-offending-flows-insert-contains-new*:
⟦ *wf-graph* ⦇ *nodes* = *V*, *edges* = *insert e E* ⦈; *set-offending-flows* ⦇*nodes* = *V*, *edges* = *E*⦈ *nP* = {}; *set-offending-flows* ⦇*nodes* = *V*, *edges* = *insert e E*⦈ *nP* ≠ {} ⟧ ⟹
{*e*} ∈ *set-offending-flows* ⦇*nodes* = *V*, *edges* = *insert e E*⦈ *nP*
proof −
assume *wfG*: *wf-graph* ⦇ *nodes* = *V*, *edges* = *insert e E* ⦈
and *a1*: *set-offending-flows* ⦇*nodes* = *V*, *edges* = *E*⦈ *nP* = {}
and *a2*: *set-offending-flows* ⦇*nodes* = *V*, *edges* = *insert e E*⦈ *nP* ≠ {}

from *a1 a2* **have** *e* ∉ *E* **by** (*metis insert-absorb*)

from *a1* **have** *a1'*: ∀ *F* ⊆ *E*. ¬ *is-offending-flows-min-set F* ⦇*nodes* = *V*, *edges* = *E*⦈ *nP*
by(*simp add*: *set-offending-flows-def*)
from *a2* **have** *a2'*: ∃ *F* ⊆ *insert e E*. *is-offending-flows-min-set F* ⦇*nodes* = *V*, *edges* = *insert e E*⦈ *nP*
by(*simp add*: *set-offending-flows-def*)

from *wfG* **have** *wfG'*: *wf-graph* ⦇ *nodes* = *V*, *edges* = *E* ⦈ **by**(*simp add*:*wf-graph-def*)

from *a1 defined-offending*[*OF wfG'*] **have** *evalG*: *sinvar* ⦇*nodes* = *V*, *edges* = *E* ⦈ *nP* **by** *blast*
from *sinvar-monoI*[*unfolded sinvar-mono-def*] *wfG' this*
have *goal-eval*: *sinvar* ⦇*nodes* = *V*, *edges* = *E* − {*e*}⦈ *nP* **by** (*metis Diff-subset*)

from *sinvar-no-offending a2* **have** *goal-not-eval*: ¬ *sinvar* ⦇*nodes* = *V*, *edges* = *insert e E*⦈ *nP* **by** *blast*

obtain *a b* **where** *e*: *e* = (*a*,*b*) **by** (*cases e*) *blast*
with *wfG* **have** *insert-e-V*: *insert a* (*insert b V*) = *V* **by**(*auto simp add*: *wf-graph-def*)

from *a1' a2'* **have** *min-set-e*: *is-offending-flows-min-set* {*e*} ⦇*nodes* = *V*, *edges* = *insert e E*⦈ *nP*
apply(*simp add*: *is-offending-flows-min-set-def is-offending-flows-def add-edge-def delete-edges-simp2 goal-not-eval goal-eval*)
using *goal-not-eval* **by**(*simp add*: *e insert-e-V*)

thus {*e*} ∈ *set-offending-flows* ⦇*nodes* = *V*, *edges* = *insert e E*⦈ *nP*
by(*simp add*: *set-offending-flows-def*)
qed

end

value *Pow* {*1*::*int*, *2*, *3*} ∪ {{*8*}, {*9*}}
value ⋃ *x*∈*Pow* {*1*::*int*, *2*, *3*}. ⋃ *y* ∈ {{*8*::*int*}, {*9*}}. {*x* ∪ *y*}

— combines powerset of A with B
definition *pow-combine* :: *'x set* ⇒ *'x set set* ⇒ *'x set set* **where**

pow-combine A B ≡ ($\bigcup$ *X* ∈ *Pow A*. $\bigcup$ *Y* ∈ *B*. {*X* ∪ *Y*}) ∪ *Pow A*

value *pow-combine* {*1*::*int*,*2*} {{*5*::*int*, *6*}, {*8*}}
value *pow-combine* {*1*::*int*,*2*} {}

lemma *pow-combine-mono*:
fixes *S* :: *'a set set*
and *X* :: *'a set*
and *Y* :: *'a set*
assumes *a1*: ∀ *F* ∈ *S*. *F* ⊆ *X*
shows ∀ *F* ∈ *pow-combine Y S*. *F* ⊆ *Y* ∪ *X*
apply(*simp add*: *pow-combine-def*)
apply(*rule*)
apply(*simp*)
by (*metis Pow-iff assms sup.coboundedI1 sup.orderE sup.orderI sup-assoc*)

lemma *S* ⊆ *pow-combine X S* **by**(*auto simp add*: *pow-combine-def*)
lemma *Pow X* ⊆ *pow-combine X S* **by**(*auto simp add*: *pow-combine-def*)

lemma *rule-pow-combine-fixfst*: *B* ⊆ *C* ⟹ *pow-combine A B* ⊆ *pow-combine A C*
by(*auto simp add*: *pow-combine-def*)

value *pow-combine* {*1*::*int*,*2*} {{*5*::*int*, *6*}, {*1*}} ⊆ *pow-combine* {*1*::*int*,*2*} {{*5*::*int*, *6*}, {*8*}}

lemma *rule-pow-combine-fixfst-Union*: $\bigcup$ *B* ⊆ $\bigcup$ *C* ⟹ $\bigcup$ (*pow-combine A B*) ⊆ $\bigcup$ (*pow-combine A C*)
apply(*rule*)
apply(*fastforce simp*: *pow-combine-def*)
done

context *SecurityInvariant-preliminaries*
begin

lemma *offending-partition-subset-empty*:
assumes *a1*:∀ *F* ∈ (*set-offending-flows* (|*nodes* = *V*, *edges* = *E* ∪ *X*|) *nP*). *F* ⊆ *X*
and *wfGEX*: *wf-graph* (|*nodes* = *V*, *edges* = *E* ∪ *X*|)
and *disj*: *E* ∩ *X* = {}
shows (*set-offending-flows* (|*nodes* = *V*, *edges* = *E*|) *nP*) = {}
proof(*rule ccontr*)
assume *c*: *set-offending-flows* (|*nodes* = *V*, *edges* = *E*|) *nP* ≠ {}
from *this* **obtain** *F'* **where** *F'-prop*: *F'* ∈ *set-offending-flows* (|*nodes* = *V*, *edges* = *E*|) *nP* **by** *blast*
from *F'-prop* **have** *F'* ⊆ *E* **using** *set-offending-flows-def* **by** *simp*
from *mono-extend-set-offending-flows*[*OF wfGEX - F'-prop*] **have**
∃ *F*∈*set-offending-flows* (|*nodes* = *V*, *edges* = *E* ∪ *X*|) *nP*. *F'* ⊆ *F* **by** *blast*
from *this a1* **have** *F'* ⊆ *X* **by** *fast*
from *F'-prop* **have** {} ≠ *F'* **by** (*metis empty-offending-contra*)
from ‹*F'* ⊆ *X*› ‹*F'* ⊆ *E*› *disj* ‹{} ≠ *F'*›
show *False* **by** *blast*
qed

corollary *partitioned-offending-subseteq-pow-combine*:
assumes *wfGEX*: *wf-graph* (|*nodes* = *V*, *edges* = *E* ∪ *X*|)
and *disj*: *E* ∩ *X* = {}
and *partitioned-offending*: ∀ *F* ∈ (*set-offending-flows* (|*nodes* = *V*, *edges* = *E* ∪ *X*|) *nP*). *F* ⊆ *X*
shows (*set-offending-flows* (|*nodes* = *V*, *edges* = *E* ∪ *X*|) *nP*) ⊆ *pow-combine X* (*set-offending-flows* (|*nodes* = *V*, *edges* = *E*|) *nP*)
apply(*subst offending-partition-subset-empty*[*OF partitioned-offending wfGEX disj*])
apply(*simp add*: *pow-combine-def*)
apply(*rule*)
apply(*simp*)
using *partitioned-offending* **by** *simp*
end

context *SecurityInvariant-preliminaries*
begin

Knowing that the ⋃ *offending is* ⊆ *X*, removing something from the graphs's edges, it also disappears from the offending flows.

lemma *Un-set-offending-flows-bound-minus*:
assumes *wfG*: *wf-graph* (| *nodes* = *V*, *edges* = *E* |)
and *Foffending*: ⋃(*set-offending-flows* (|*nodes* = *V*, *edges* = *E*|) *nP*) ⊆ *X*
shows ⋃(*set-offending-flows* (|*nodes* = *V*, *edges* = *E* − {*f*}|) *nP*) ⊆ *X* − {*f*}
proof −
from *wfG* **have** *wfG′*: *wf-graph* (| *nodes* = *V*, *edges* = *E* − {*f*} |)
by(*auto simp add*: *wf-graph-def finite-subset*)

from *offending-flows-union-mono*[*OF wfG*, **where** *E′*=*E* − {*f*}] **have**
⋃(*set-offending-flows* (|*nodes* = *V*, *edges* = *E* − {*f*}|) *nP*) − {*f*} ⊆ ⋃(*set-offending-flows* (|*nodes* = *V*, *edges* = *E*|) *nP*) − {*f*} **by** *blast*
also have
⋃(*set-offending-flows* (|*nodes* = *V*, *edges* = *E* − {*f*}|) *nP*) ⊆ ⋃(*set-offending-flows* (|*nodes* = *V*, *edges* = *E* − {*f*}|) *nP*) − {*f*}
apply(*simp add*: *set-offending-flows-simp*[*OF wfG′*]) **by** *blast*
ultimately have *Un-set-offending-flows-minus*:
⋃(*set-offending-flows* (|*nodes* = *V*, *edges* = *E* − {*f*}|) *nP*) ⊆ ⋃(*set-offending-flows* (|*nodes* = *V*, *edges* = *E* |) *nP*) − {*f*}
by *blast*

from *Foffending Un-set-offending-flows-minus*
show *?thesis* **by** *blast*
qed

If the offending flows are bound by some *X*, the we can remove all finite *E′*from the graph's edges and the offending flows from the smaller graph are bound by *X* − *E′*.

lemma *Un-set-offending-flows-bound-minus-subseteq*:
assumes *wfG*: *wf-graph* (| *nodes* = *V*, *edges* = *E* |)
and *Foffending*: ⋃ (*set-offending-flows* (|*nodes* = *V*, *edges* = *E*|) *nP*) ⊆ *X*
shows ⋃ (*set-offending-flows* (|*nodes* = *V*, *edges* = *E* − *E′*|) *nP*) ⊆ *X* − *E′*
proof −
from *wfG* **have** *wfG′*: *wf-graph* (| *nodes* = *V*, *edges* = *E* − *E′* |)
by(*auto simp add*: *wf-graph-def finite-subset*)

from *offending-flows-union-mono*[*OF wfG*, **where** $E'=E - E'$] **have**
$\bigcup$(*set-offending-flows* (|*nodes* = *V*, *edges* = $E - E'$|) *nP*) $- E' \subseteq \bigcup$(*set-offending-flows* (|*nodes* = *V*, *edges* = *E*|) *nP*) $- E'$ **by** *blast*
also have
$\bigcup$(*set-offending-flows* (|*nodes* = *V*, *edges* = $E - E'$|) *nP*) $\subseteq \bigcup$(*set-offending-flows* (|*nodes* = *V*, *edges* = $E - E'$|) *nP*) $- E'$
apply(*simp add*: *set-offending-flows-simp*[*OF wfG'*]) **by** *blast*
ultimately have *Un-set-offending-flows-minus*:
$\bigcup$ (*set-offending-flows* (|*nodes* = *V*, *edges* = $E - E'$|) *nP*) $\subseteq \bigcup$ (*set-offending-flows* (|*nodes* = *V*, *edges* = *E* |) *nP*) $- E'$
by *blast*

from *Foffending Un-set-offending-flows-minus*
show *?thesis* **by** *blast*
qed

corollary *Un-set-offending-flows-bound-minus-subseteq'*:
⟦ *wf-graph* (| *nodes* = *V*, *edges* = *E* |);
$\bigcup$ (*set-offending-flows* (| *nodes* = *V*, *edges* = *E* |) *nP*) $\subseteq$ *X* ⟧ $\Longrightarrow$
$\bigcup$ (*set-offending-flows* (| *nodes* = *V*, *edges* = $E - E'$ |) *nP*) $\subseteq X - E'$
apply(*drule*(*1*) *Un-set-offending-flows-bound-minus-subseteq*) **by** *blast*

end

end
theory *TopoS-ENF*
imports *Main TopoS-Interface Lib/TopoS-Util TopoS-withOffendingFlows*
begin

4 Special Structures of Security Invariants

Security Invariants may have a common structure: If the function *sinvar* is predicate which starts with $\forall\ (v_1, v_2) \in$ *edges G*. ..., we call this the all edges normal form (ENF). We found that this form has some nice properties. Also, locale instantiation is easier in ENF with the help of the following lemmata.

4.1 Simple Edges Normal Form (ENF)

context *SecurityInvariant-withOffendingFlows*
begin

definition *sinvar-all-edges-normal-form* :: ($'a \Rightarrow 'a \Rightarrow$ *bool*) $\Rightarrow$ *bool* **where**
sinvar-all-edges-normal-form P $\equiv \forall$ *G nP*. *sinvar G nP* = ($\forall$ (*e1*, *e2*)$\in$ *edges G*. *P* (*nP e1*) (*nP e2*))

reflexivity is needed for convenience. If a security invariant is not reflexive, that means that all nodes with the default parameter $\bot$ are not allowed to communicate with each other. Non-reflexivity is possible, but requires more work.

definition *ENF-refl* :: ($'a \Rightarrow 'a \Rightarrow$ *bool*) $\Rightarrow$ *bool* **where**
ENF-refl P $\equiv$ *sinvar-all-edges-normal-form P* $\wedge$ ($\forall$ *p1*. *P p1 p1*)

lemma *monotonicity-sinvar-mono*: *sinvar-all-edges-normal-form P* ⟹ *sinvar-mono*
unfolding *sinvar-all-edges-normal-form-def sinvar-mono-def*
by *auto*

end

4.1.1 Offending Flows

context *SecurityInvariant-withOffendingFlows*
begin

The insight: for all edges in the members of the offending flows, ¬ *P* holds.

lemma *ENF-offending-imp-not-P*:
assumes *sinvar-all-edges-normal-form P F* ∈ *set-offending-flows G nP* (*e1*, *e2*) ∈ *F*
shows ¬ *P* (*nP e1*) (*nP e2*)
using *assms*
unfolding *sinvar-all-edges-normal-form-def set-offending-flows-def is-offending-flows-min-set-def is-offending-flows-def*
by (*fastforce simp*: *graph-ops*)

Hence, the members of *set-offending-flows* must look as follows.

lemma *ENF-offending-set-P-representation*:
assumes *sinvar-all-edges-normal-form P F* ∈ *set-offending-flows G nP*
shows *F* = {(*e1*,*e2*). (*e1*, *e2*) ∈ *edges G* ∧ ¬ *P* (*nP e1*) (*nP e2*)} (**is** *F* = *?E*)
proof −
{ **fix** *a b*
assume (*a*, *b*) ∈ *F*
hence (*a*, *b*) ∈ *?E*
using *assms*
by (*auto simp*: *set-offending-flows-def ENF-offending-imp-not-P*)
}
moreover
{ **fix** *x*
assume *x* ∈ *?E*
hence *x* ∈ *F*
using *assms*
unfolding *sinvar-all-edges-normal-form-def set-offending-flows-def is-offending-flows-min-set-def*
by (*fastforce simp*: *is-offending-flows-def graph-ops*)
}
ultimately show *?thesis*
by *blast*
qed

We can show left to right of the desired representation of *set-offending-flows*

lemma *ENF-offending-subseteq-lhs*:
assumes *sinvar-all-edges-normal-form P*
shows *set-offending-flows G nP* ⊆ { {(*e1*,*e2*). (*e1*, *e2*) ∈ *edges G* ∧ ¬ *P* (*nP e1*) (*nP e2*)} }
using *assms*
by (*force simp*: *ENF-offending-set-P-representation*)

if *set-offending-flows* is not empty, we have the other direction.

lemma *ENF-offending-not-empty-imp-ENF-offending-subseteq-rhs*:
assumes *sinvar-all-edges-normal-form P set-offending-flows G nP* ≠ {}
shows { {(*e1*,*e2*) ∈ *edges G*. ¬ *P* (*nP e1*) (*nP e2*)} } ⊆ *set-offending-flows G nP*

using *assms ENF-offending-set-P-representation*
by *blast*

lemma *ENF-notevalmodel-imp-offending-not-empty*:
sinvar-all-edges-normal-form P $\Longrightarrow$ $\neg$ *sinvar G nP* $\Longrightarrow$ *set-offending-flows G nP* $\neq$ {}

proof −
assume *enf*: *sinvar-all-edges-normal-form P*
and *ns*: $\neg$ *sinvar G nP*

{
let *?F'*={(*e1*,*e2*) ∈ (*edges G*). ¬*P* (*nP e1*) (*nP e2*)}
— select {(*e1*, *e2*). (*e1*, *e2*) ∈ *edges G* ∧ ¬ *P* (*nP e1*) (*nP e2*)} as the list of all edges which violate *P*

from *enf* **have** *ENF-notevalmodel-offending-imp-ex-offending-min*:
⋀*F*. *is-offending-flows F G nP* ⟹ *F* ⊆ *edges G* ⟹
∃ *F'*. *F'* ⊆ *edges G* ∧ *is-offending-flows-min-set F' G nP*
unfolding *sinvar-all-edges-normal-form-def is-offending-flows-min-set-def is-offending-flows-def*
by (−) (*rule exI*[**where** *x*=*?F'*], *fastforce simp*: *graph-ops*)

from *enf ns* **have** ∃ *F*. *F* ⊆ (*edges G*) ∧ *is-offending-flows F G nP*
unfolding *sinvar-all-edges-normal-form-def is-offending-flows-def*
by (−) (*rule exI*[**where** *x*=*?F'*], *fastforce simp*: *graph-ops*)

from *enf ns this ENF-notevalmodel-offending-imp-ex-offending-min* **have** *ENF-notevalmodel-imp-ex-offending-min*:
∃ *F*. *F* ⊆ *edges G* ∧ *is-offending-flows-min-set F G nP* **by** *blast*
} **note** *ENF-notevalmodel-imp-ex-offending-min*=*this*

from *ENF-notevalmodel-imp-ex-offending-min* **show** *set-offending-flows G nP* ≠ {} **using** *set-offending-flows-def* **by** *simp*
qed

lemma *ENF-offending-case1*:
⟦ *sinvar-all-edges-normal-form P*; ¬ *sinvar G nP* ⟧ ⟹
{ {(*e1*,*e2*). (*e1*, *e2*) ∈ (*edges G*) ∧ ¬ *P* (*nP e1*) (*nP e2*)} } = *set-offending-flows G nP*
apply(*rule*)
apply(*frule ENF-notevalmodel-imp-offending-not-empty*, *simp*)
apply(*rule ENF-offending-not-empty-imp-ENF-offending-subseteq-rhs*, *simp*)
apply *simp*
apply(*rule ENF-offending-subseteq-lhs*)
apply *simp*
done

lemma *ENF-offending-case2*:
⟦ *sinvar-all-edges-normal-form P*; *sinvar G nP* ⟧ ⟹
{} = *set-offending-flows G nP*
apply(*drule sinvar-no-offending*[*of G nP*])
apply *simp*
done

theorem *ENF-offending-set*:
⟦ *sinvar-all-edges-normal-form P* ⟧ ⟹

set-offending-flows G nP = (*if sinvar G nP then*
{}
else
{ {(*e1*,*e2*). (*e1*, *e2*) ∈ *edges G* ∧ ¬ *P* (*nP e1*) (*nP e2*)} })
by(*simp add*: *ENF-offending-case1 ENF-offending-case2*)
end

4.1.2 Lemmata

lemma (**in** *SecurityInvariant-withOffendingFlows*) *ENF-offending-members*:
⟦ ¬ *sinvar G nP*; *sinvar-all-edges-normal-form P*; *f* ∈ *set-offending-flows G nP*⟧ ⟹
f ⊆ (*edges G*) ∧ (∀ (*e1*, *e2*)∈ *f*. ¬ *P* (*nP e1*) (*nP e2*))
by(*auto simp add*: *ENF-offending-set*)

4.1.3 Instance Helper

lemma (**in** *SecurityInvariant-withOffendingFlows*) *ENF-refl-not-offedning*:
⟦ ¬ *sinvar G nP*; *f* ∈ *set-offending-flows G nP*;
ENF-refl P⟧ ⟹
∀(*e1*,*e2*) ∈ *f*. *e1* ≠ *e2*
proof −
assume *a-not-eval*: ¬ *sinvar G nP*
and *a-enf-refl*: *ENF-refl P*
and *a-offedning*: *f* ∈ *set-offending-flows G nP*

from *a-enf-refl* **have** *a-enf*: *sinvar-all-edges-normal-form P* **using** *ENF-refl-def* **by** *simp*
hence *a-ENF*: ⋀ *G nP*. *sinvar G nP* = (∀ (*e1*, *e2*) ∈ *edges G*. *P* (*nP e1*) (*nP e2*)) **using** *sinvar-all-edges-normal-form-def* **by** *simp*

from *a-enf-refl ENF-refl-def* **have** *a-refl*: ∀ (*e1*,*e1*) ∈ *f*. *P* (*nP e1*) (*nP e1*) **by** *simp*
from *ENF-offending-members*[*OF a-not-eval a-enf a-offedning*] **have** ∀ (*e1*, *e2*) ∈ *f*. ¬ *P* (*nP e1*) (*nP e2*) **by** *fast*
from *this a-refl* **show** ∀(*e1*,*e2*) ∈ *f*. *e1* ≠ *e2* **by** *fast*
qed

lemma (**in** *SecurityInvariant-withOffendingFlows*) *ENF-default-update-fst*:
fixes *default-node-properties* :: *'a* (‹⊥›)
assumes *modelInv*: ¬ *sinvar G nP*
and *ENFdef*: *sinvar-all-edges-normal-form P*
and *secdef*: ∀ (*nP*::*'v* ⇒ *'a*) *e1 e2*. ¬ (*P* (*nP e1*) (*nP e2*)) ⟶ ¬ (*P* ⊥ (*nP e2*))
shows
¬ (∀ (*e1*, *e2*) ∈ *edges G*. *P* ((*nP*(*i* := ⊥)) *e1*) (*nP e2*))
proof −
from *ENFdef* **have** *ENF*: ⋀ *G nP*. *sinvar G nP* = (∀ (*e1*, *e2*)∈ *edges G*. *P* (*nP e1*) (*nP e2*))
using *sinvar-all-edges-normal-form-def* **by** *simp*
from *modelInv ENF* **have** *modelInv'*: ¬ (∀ (*e1*, *e2*)∈ *edges G*. *P* (*nP e1*) (*nP e2*)) **by** *simp*
from *this secdef* **have** *modelInv''*: ¬ (∀ (*e1*, *e2*)∈ *edges G*. *P* ⊥ (*nP e2*)) **by** *blast*
have *simpUpdateI*: ⋀ *e1 e2*. ¬ *P* (*nP e1*) (*nP e2*) ⟹ ¬ *P* ⊥ (*nP e2*) ⟹ ¬ *P* ((*nP*(*i* := ⊥)) *e1*) (*nP e2*) **by** *simp*
hence ⋀ *X*. ∃(*e1*,*e2*) ∈ *X*. ¬ *P* (*nP e1*) (*nP e2*) ⟹ ∃(*e1*,*e2*) ∈ *X*.¬ *P* ⊥ (*nP e2*) ⟹ ∃(*e1*,*e2*) ∈ *X*.¬ *P* ((*nP*(*i* := ⊥)) *e1*) (*nP e2*)
using *secdef* **by** *blast*
from *this modelInv' modelInv''* **show** ¬ (∀ (*e1*, *e2*) ∈ *edges G*. *P* ((*nP*(*i* := ⊥)) *e1*) (*nP e2*)) **by** *blast*

qed

lemma (**in** *SecurityInvariant-withOffendingFlows*)
fixes *default-node-properties* :: *'a* (‹⊥›)
shows ¬ *sinvar G nP* ⟹ *sinvar-all-edges-normal-form P* ⟹
(∀ (*nP*::*'v* ⇒ *'a*) *e1 e2*. ¬ (*P* (*nP e1*) (*nP e2*)) ⟶ ¬ (*P* ⊥ (*nP e2*))) ⟹
(∀ (*nP*::*'v* ⇒ *'a*) *e1 e2*. ¬ (*P* (*nP e1*) (*nP e2*)) ⟶ ¬ (*P* (*nP e1*) ⊥)) ⟹
(∀ (*nP*::*'v* ⇒ *'a*) *e1 e2*. ¬ *P* ⊥ ⊥)
⟹ ¬ *sinvar G* (*nP*(*i* := ⊥))
proof −
assume *a1*: ¬ *sinvar G nP*
and *a2d*: *sinvar-all-edges-normal-form P*
and *a3*: ∀ (*nP*::*'v* ⇒ *'a*) *e1 e2*. ¬ (*P* (*nP e1*) (*nP e2*)) ⟶ ¬ (*P* ⊥ (*nP e2*))
and *a4*: ∀ (*nP*::*'v* ⇒ *'a*) *e1 e2*. ¬ (*P* (*nP e1*) (*nP e2*)) ⟶ ¬ (*P* (*nP e1*) ⊥)
and *a5*: ∀ (*nP*::*'v* ⇒ *'a*) *e1 e2*. ¬ *P* ⊥ ⊥

from *a2d* **have** *a2*: ⋀ *G nP*. *sinvar G nP* = (∀ (*e1*, *e2*) ∈ *edges G*. *P* (*nP e1*) (*nP e2*))
using *sinvar-all-edges-normal-form-def* **by** *simp*

from *ENF-default-update-fst*[*OF a1 a2d*] *a3* **have** *subgoal1*: ¬ (∀ (*e1*, *e2*) ∈ *edges G*. *P* ((*nP*(*i* := ⊥)) *e1*) (*nP e2*)) **by** *blast*

let *?nP'* = (*nP*(*i* := ⊥))

from *subgoal1* **have** ∃ (*e1*, *e2*) ∈ *edges G*. ¬ *P* (*?nP' e1*) (*nP e2*) **by** *blast*
from *this* **obtain** *e11 e21* **where** *s1cond*: (*e11*, *e21*) ∈ *edges G* ∧ ¬ *P* (*?nP' e11*) (*nP e21*) **by** *blast*

from *s1cond* **have** *i* ≠ *e11* ⟹ ¬ *P* (*nP e11*) (*nP e21*) **by** *simp*
from *s1cond* **have** *e11* ≠ *e21* ⟹ ¬ *P* (*?nP' e11*) (*?nP' e21*)
apply *simp*
apply(*rule conjI*)
apply *blast*
apply(*insert a4*)
by *force*
from *s1cond a4 fun-upd-apply* **have** *ex1*: *e11* ≠ *e21* ⟹ ¬ *P* (*?nP' e11*) (*?nP' e21*) **by** *metis*
from *s1cond a5* **have** *ex2*: *e11* = *e21* ⟹ ¬ *P* (*?nP' e11*) (*?nP' e21*) **by** *auto*

from *ex1 ex2 s1cond* **have** ∃ (*e1*, *e2*) ∈ *edges G*. ¬ *P* (*?nP' e1*) (*?nP' e2*) **by** *blast*
hence ¬ (∀ (*e1*, *e2*) ∈ *edges G*. *P* ((*nP*(*i* := ⊥)) *e1*) ((*nP*(*i* := ⊥)) *e2*)) **by** *fast*
from *this a2* **show** ¬ *sinvar G* (*nP*(*i* := ⊥)) **by** *presburger*
qed

lemma (**in** *SecurityInvariant-withOffendingFlows*) *ENF-fsts-refl-instance*:
fixes *default-node-properties* :: *'a* (‹⊥›)
assumes *a-enf-refl*: *ENF-refl P*
and *a3*: ∀ (*nP*::*'v* ⇒ *'a*) *e1 e2*. ¬ (*P* (*nP e1*) (*nP e2*)) ⟶ ¬ (*P* ⊥ (*nP e2*))
and *a-offending*: *f* ∈ *set-offending-flows G nP*
and *a-i-fsts*: *i* ∈ *fst* ‘ *f*
shows
¬ *sinvar G* (*nP*(*i* := ⊥))
proof −

from *a-offending* **have** *a-not-eval*: ¬ *sinvar G nP* **by** (*metis equals0D sinvar-no-offending*)
from *valid-without-offending-flows*[*OF a-offending*] **have** *a-offending-rm*: *sinvar* (*delete-edges G f*) *nP* **.**

from *a-enf-refl* **have** *a-enf*: *sinvar-all-edges-normal-form P* **using** *ENF-refl-def* **by** *simp*
hence *a2*: ⋀ *G nP*. *sinvar G nP* = (∀ (*e1*, *e2*) ∈ *edges G*. *P* (*nP e1*) (*nP e2*)) **using** *sinvar-all-edges-normal-form-def* **by** *simp*

from *ENF-offending-members*[*OF a-not-eval a-enf a-offending*] **have** *a-f-3-in-f*: ⋀ *e1 e2*. (*e1*, *e2*) ∈ *f* ⟹ ¬ *P* (*nP e1*) (*nP e2*) **by** *fast*

let *?nP′* = (*nP*(*i* := ⊥))

from *offending-not-empty*[*OF a-offending*] *ENF-offending-members*[*OF a-not-eval a-enf a-offending*] *a-i-fsts hd-in-set*
obtain *e1 e2* **where** *e1e2cond*: (*e1*, *e2*) ∈ *f* ∧ *e1* = *i* **by** *force*

from *conjunct1*[*OF e1e2cond*] *a-f-3-in-f* **have** *e1e2notP*: ¬ *P* (*nP e1*) (*nP e2*) **by** *simp*
from *this a3* **have** ¬ *P* ⊥ (*nP e2*) **by** *simp*
from *this e1e2notP* **have** *e1e2subgoal1*: ¬ *P* (*?nP′ e1*) (*nP e2*) **by** *simp*

from *ENF-refl-not-offedning*[*OF a-not-eval a-offending a-enf-refl*] *conjunct1*[*OF e1e2cond*] **have** *ENF-refl*: *e1* ≠ *e2* **by** *fast*

from *e1e2subgoal1* **have** *e1* ≠ *e2* ⟹ ¬ *P* (*?nP′ e1*) (*?nP′ e2*)
apply *simp*
apply(*rule conjI*)
apply *blast*
apply(*insert conjunct2*[*OF e1e2cond*])
by *simp*

from *this ENF-refl ENF-offending-members*[*OF a-not-eval a-enf a-offending*] *conjunct1*[*OF e1e2cond*] **have**
∃ (*e1*, *e2*) ∈ *edges G*. ¬ *P* (*?nP′ e1*) (*?nP′ e2*) **by** *blast*
hence ¬ (∀ (*e1*, *e2*) ∈ *edges G*. *P* ((*nP*(*i* := ⊥)) *e1*) ((*nP*(*i* := ⊥)) *e2*)) **by** *fast*
from *this a2* **show** ¬ *sinvar G* (*nP*(*i* := ⊥)) **by** *presburger*
qed

lemma (**in** *SecurityInvariant-withOffendingFlows*) *ENF-snds-refl-instance*:
fixes *default-node-properties* :: ′*a* (‹⊥›)
assumes *a-enf-refl*: *ENF-refl P*
and *a3*: ∀ (*nP*::′*v* ⇒ ′*a*) *e1 e2*. ¬ (*P* (*nP e1*) (*nP e2*)) ⟶ ¬ (*P* (*nP e1*) ⊥)
and *a-offending*: *f* ∈ *set-offending-flows G nP*
and *a-i-snds*: *i* ∈ *snd* ‘ *f*
shows
¬ *sinvar G* (*nP*(*i* := ⊥))
proof −
from *a-offending* **have** *a-not-eval*: ¬ *sinvar G nP* **by** (*metis equals0D sinvar-no-offending*)
from *valid-without-offending-flows*[*OF a-offending*] **have** *a-offending-rm*: *sinvar* (*delete-edges G f*) *nP* **.**
from *a-enf-refl* **have** *a-enf*: *sinvar-all-edges-normal-form P* **using** *ENF-refl-def* **by** *simp*
hence *a2*: ⋀ *G nP*. *sinvar G nP* = (∀ (*e1*, *e2*) ∈ *edges G*. *P* (*nP e1*) (*nP e2*)) **using** *sin-*

var-all-edges-normal-form-def **by** *simp*

from *ENF-offending-members*[*OF a-not-eval a-enf a-offending*] **have** *a-f-3-in-f*: $\bigwedge$ *e1 e2*. (*e1*, *e2*) ∈ *f* ⟹ ¬ *P* (*nP e1*) (*nP e2*) **by** *fast*

let *?nP′* = (*nP*(*i* := ⊥))

from *offending-not-empty*[*OF a-offending*] *ENF-offending-members*[*OF a-not-eval a-enf a-offending*] *a-i-snds hd-in-set*
obtain *e1 e2* **where** *e1e2cond*: (*e1*, *e2*) ∈ *f* ∧ *e2* = *i* **by** *force*

from *conjunct1*[*OF e1e2cond*] *a-f-3-in-f* **have** *e1e2notP*: ¬ *P* (*nP e1*) (*nP e2*) **by** *simp*
from *this a3* **have** ¬ *P* (*nP e1*) ⊥ **by** *auto*
from *this e1e2notP* **have** *e1e2subgoal1*: ¬ *P* (*nP e1*) (*?nP′ e2*) **by** *simp*

from *ENF-refl-not-offedning*[*OF a-not-eval a-offending a-enf-refl*] *e1e2cond* **have** *ENF-refl*: *e1* ≠ *e2* **by** *fast*

from *e1e2subgoal1* **have** *e1* ≠ *e2* ⟹ ¬ *P* (*?nP′ e1*) (*?nP′ e2*)
apply *simp*
apply(*rule conjI*)
apply(*insert conjunct2*[*OF e1e2cond*])
by *simp-all*

from *this ENF-refl e1e2cond ENF-offending-members*[*OF a-not-eval a-enf a-offending*] *conjunct1*[*OF e1e2cond*] **have**
∃ (*e1*, *e2*) ∈ *edges G*. ¬ *P* (*?nP′ e1*) (*?nP′ e2*) **by** *blast*
hence ¬ (∀ (*e1*, *e2*) ∈ *edges G*. *P* ((*nP*(*i* := ⊥)) *e1*) ((*nP*(*i* := ⊥)) *e2*)) **by** *fast*
from *this a2* **show** ¬ *sinvar G* (*nP*(*i* := ⊥)) **by** *presburger*
qed

4.2 edges normal form ENF with sender and receiver names

definition (**in** *SecurityInvariant-withOffendingFlows*) *sinvar-all-edges-normal-form-sr* :: (′*a* ⇒ ′*v* ⇒ ′*a* ⇒ ′*v* ⇒ *bool*) ⇒ *bool* **where**
sinvar-all-edges-normal-form-sr P ≡ ∀ *G nP*. *sinvar G nP* = (∀ (*s*, *r*)∈ *edges G*. *P* (*nP s*) *s* (*nP r*) *r*)

lemma (**in** *SecurityInvariant-withOffendingFlows*) *ENFsr-monotonicity-sinvar-mono*: ⟦ *sinvar-all-edges-normal-form-sr P* ⟧ ⟹
sinvar-mono
apply(*simp add*: *sinvar-all-edges-normal-form-sr-def sinvar-mono-def*)
by *blast*

4.2.1 Offending Flows:

theorem (**in** *SecurityInvariant-withOffendingFlows*) *ENFsr-offending-set*:
assumes *ENFsr*: *sinvar-all-edges-normal-form-sr P*
shows *set-offending-flows G nP* = (*if sinvar G nP then*
{}
else
{ {(*s*,*r*). (*s*, *r*) ∈ *edges G* ∧ ¬ *P* (*nP s*) *s* (*nP r*) *r*} }) (**is** *?A* = *?B*)

proof(*cases sinvar G nP*)
case *True* **thus** *?A = ?B*
by(*simp add*: *set-offending-flows-def is-offending-flows-min-set-def is-offending-flows-def*)
next
case *False*
from *ENFsr* **have** *ENFsr-offending-imp-not-P*: $\bigwedge$ *F s r*. *F* ∈ *set-offending-flows G nP* ⟹ (*s*, *r*) ∈ *F* ⟹ ¬ *P* (*nP s*) *s* (*nP r*) *r*
unfolding *sinvar-all-edges-normal-form-sr-def*
apply(*simp add*: *set-offending-flows-def is-offending-flows-def is-offending-flows-min-set-def graph-ops*)
apply *clarify*
by *fastforce*
from *ENFsr* **have** *ENFsr-offending-set-P-representation*:
$\bigwedge$ *F*. *F* ∈ *set-offending-flows G nP* ⟹ *F* = {(*s*,*r*). (*s*, *r*) ∈ *edges G* ∧ ¬ *P* (*nP s*) *s* (*nP r*) *r*}
apply –
apply *rule*
apply *rule*
apply *clarify*
apply(*rename-tac a b*)
apply *rule*
apply(*auto simp add*:*set-offending-flows-def*)[*1*]
apply(*simp add*: *ENFsr-offending-imp-not-P*)
unfolding *sinvar-all-edges-normal-form-sr-def*
apply(*simp add*:*set-offending-flows-def is-offending-flows-def is-offending-flows-min-set-def graph-ops*)
apply *clarify*
apply(*rename-tac a b a1 b1*)
apply(*blast*)
done

from *ENFsr False* **have** *ENFsr-offending-flows-exist*: *set-offending-flows G nP* ≠ {}
apply(*simp add*: *set-offending-flows-def is-offending-flows-min-set-def is-offending-flows-def sinvar-all-edges-normal-form-sr-def*
delete-edges-def add-edge-def)
apply(*clarify*)
apply(*rename-tac s r*)
apply(*rule-tac x*={(*s*,*r*). (*s*,*r*) ∈ (*edges G*) ∧ ¬*P* (*nP s*) *s* (*nP r*) *r*} **in** *exI*)
apply(*simp*)
by *blast*

from *ENFsr* **have** *ENFsr-offenindg-not-empty-imp-ENF-offending-subseteq-rhs*:
set-offending-flows G nP ≠ {} ⟹
{ {(*s*,*r*). (*s*, *r*) ∈ *edges G* ∧ ¬ *P* (*nP s*) *s* (*nP r*) *r*} } ⊆ *set-offending-flows G nP*
apply –
apply *rule*
using *ENFsr-offending-set-P-representation*
by *blast*

from *ENFsr* **have** *ENFsr-offending-subseteq-lhs*:
(*set-offending-flows G nP*) ⊆ { {(*s*,*r*). (*s*, *r*) ∈ *edges G* ∧ ¬ *P* (*nP s*) *s* (*nP r*) *r*} }
apply –
apply *rule*
by(*simp add*: *ENFsr-offending-set-P-representation*)

from *False ENFsr-offenindg-not-empty-imp-ENF-offending-subseteq-rhs*[*OF ENFsr-offending-flows-exist*]

ENFsr-offending-subseteq-lhs **show** *?A = ?B*
  by *force*
qed

4.3 edges normal form not refl ENFnrSR

definition (**in** *SecurityInvariant-withOffendingFlows*) *sinvar-all-edges-normal-form-not-refl-SR* :: $('a \Rightarrow 'v \Rightarrow 'a \Rightarrow 'v \Rightarrow bool) \Rightarrow bool$ **where**
sinvar-all-edges-normal-form-not-refl-SR P ≡
$\forall\ G\ nP.\ sinvar\ G\ nP = (\forall\ (s, r) \in edges\ G.\ s \neq r \longrightarrow P\ (nP\ s)\ s\ (nP\ r)\ r)$

we derive everything from the ENFnrSR form

lemma (**in** *SecurityInvariant-withOffendingFlows*) *ENFnrSR-to-ENFsr*:
sinvar-all-edges-normal-form-not-refl-SR P ⟹ *sinvar-all-edges-normal-form-sr* $(\lambda\ p1\ v1\ p2\ v2.\ v1 \neq v2 \longrightarrow P\ p1\ v1\ p2\ v2)$
by(*simp add*: *sinvar-all-edges-normal-form-sr-def sinvar-all-edges-normal-form-not-refl-SR-def*)

4.3.1 Offending Flows

theorem (**in** *SecurityInvariant-withOffendingFlows*) *ENFnrSR-offending-set*:
⟦ *sinvar-all-edges-normal-form-not-refl-SR P* ⟧ ⟹
set-offending-flows G nP = (*if sinvar G nP then*
{}
else
$\{ \{(e1,e2).\ (e1, e2) \in edges\ G \wedge e1 \neq e2 \wedge \neg\ P\ (nP\ e1)\ e1\ (nP\ e2)\ e2\} \})$
by(*auto dest*: *ENFnrSR-to-ENFsr simp*: *ENFsr-offending-set*)

4.3.2 Instance helper

lemma (**in** *SecurityInvariant-withOffendingFlows*) *ENFnrSR-fsts-weakrefl-instance*:
fixes *default-node-properties* :: *'a* (‹⊥›)
assumes *a-enf*: *sinvar-all-edges-normal-form-not-refl-SR P*
and *a-weakrefl*: $\forall\ s\ r.\ P \perp s \perp r$
and *a-botdefault*: $\forall\ s\ r.\ (nP\ r) \neq \perp \longrightarrow \neg\ P\ (nP\ s)\ s\ (nP\ r)\ r \longrightarrow \neg\ P \perp s\ (nP\ r)\ r$
and *a-alltobot*: $\forall\ s\ r.\ P\ (nP\ s)\ s \perp r$
and *a-offending*: $f \in$ *set-offending-flows G nP*
and *a-i-fsts*: $i \in fst`\ f$
shows
$\neg\ sinvar\ G\ (nP(i := \perp))$
proof −
from *a-offending* **have** *a-not-eval*: ¬ *sinvar G nP* **by** (*metis ex-in-conv sinvar-no-offending*)
from *valid-without-offending-flows*[*OF a-offending*] **have** *a-offending-rm*: *sinvar* (*delete-edges G f*) *nP* **.**
from *a-enf* **have** *a-enf'*: $\bigwedge\ G\ nP.\ sinvar\ G\ nP\ = (\forall\ (e1, e2)\in (edges\ G).\ e1 \neq e2 \longrightarrow P\ (nP\ e1)\ e1\ (nP\ e2)\ e2)$
using *sinvar-all-edges-normal-form-not-refl-SR-def* **by** *simp*

from *ENFnrSR-offending-set*[*OF a-enf*] *a-not-eval a-offending* **have** *a-f-3-in-f*: $\bigwedge\ e1\ e2.\ (e1, e2)\in f \Longrightarrow \neg\ P\ (nP\ e1)\ e1\ (nP\ e2)\ e2$ **by**(*simp*)
from *ENFnrSR-offending-set*[*OF a-enf*] *a-not-eval a-offending* **have** *a-f-3-neq*: $\bigwedge\ e1\ e2.\ (e1, e2)\in f \Longrightarrow e1 \neq e2$ **by** *simp*

let *?nP'* = $(nP(i := \perp))$

from *ENFnrSR-offending-set*[*OF a-enf*] *a-not-eval a-offending a-i-fsts*
obtain *e1 e2* **where** *e1e2cond*: (*e1*, *e2*) ∈ *f* ∧ *e1* = *i* **by** *fastforce*

from *conjunct1*[*OF e1e2cond*] *a-offending* **have** (*e1*, *e2*) ∈ *edges G*
by (*metis* (*lifting*, *no-types*) *SecurityInvariant-withOffendingFlows.set-offending-flows-def mem-Collect-eq rev-subsetD*)

from *conjunct1*[*OF e1e2cond*] *a-f-3-in-f* **have** *e1e2notP*: ¬ *P* (*nP e1*) *e1* (*nP e2*) *e2* **by** *simp*
from *e1e2notP a-weakrefl* **have** *e1ore2neqbot*: (*nP e1*) ≠ ⊥ ∨ (*nP e2*) ≠ ⊥ **by** *fastforce*
from *e1e2notP a-alltobot* **have** (*nP e2*) ≠ ⊥ **by** *fastforce*
from *this e1e2notP a-botdefault* **have** ¬ *P* ⊥ *e1* (*nP e2*) *e2* **by** *simp*
from *this e1e2notP* **have** *e1e2subgoal1*: ¬ *P* (*?nP′ e1*) *e1* (*nP e2*) *e2* **by** *auto*

from *a-f-3-neq e1e2cond* **have** *e2* ≠ *e1* **by** *blast*

from *e1e2subgoal1* **have** *e1* ≠ *e2* ⟹ ¬ *P* (*?nP′ e1*) *e1* (*?nP′ e2*) *e2*
apply *simp*
apply(*rule conjI*)
apply *blast*
apply(*insert e1e2cond*)
by *simp*
from *this* ‹*e2* ≠ *e1*› **have** ¬ *P* (*?nP′ e1*) *e1* (*?nP′ e2*) *e2* **by** *simp*

from *this* ‹*e2* ≠ *e1*› *ENFnrSR-offending-set*[*OF a-enf*] *a-offending* ‹(*e1*, *e2*) ∈ *edges G*› **have**
∃ (*e1*, *e2*)∈(*edges G*). *e2* ≠ *e1* ∧ ¬ *P* (*?nP′ e1*) *e1* (*?nP′ e2*) *e2* **by** *blast*
hence ¬ (∀ (*e1*, *e2*)∈(*edges G*). *e2* ≠ *e1* ⟶ *P* ((*nP*(*i* := ⊥)) *e1*) *e1* ((*nP*(*i* := ⊥)) *e2*) *e2*) **by** *fast*
from *this a-enf′* **show** ¬ *sinvar G* (*nP*(*i* := ⊥)) **by** *fast*
qed

lemma (**in** *SecurityInvariant-withOffendingFlows*) *ENFnrSR-snds-weakrefl-instance*:
fixes *default-node-properties* :: *′a* (‹⊥›)
assumes *a-enf*: *sinvar-all-edges-normal-form-not-refl-SR P*
and *a-weakrefl*: ∀ *s r*. *P* ⊥ *s* ⊥ *r*
and *a-botdefault*: ∀ *s r*. (*nP s*) ≠ ⊥ ⟶ ¬ *P* (*nP s*) *s* (*nP r*) *r* ⟶ ¬ *P* (*nP s*) *s* ⊥ *r*
and *a-bottoall*: ∀ *s r*. *P* ⊥ *s* (*nP r*) *r*
and *a-offending*: *f* ∈ *set-offending-flows G nP*
and *a-i-snds*: *i* ∈ *snd‘ f*
shows
¬ *sinvar G* (*nP*(*i* := ⊥))
proof −
from *a-offending* **have** *a-not-eval*: ¬ *sinvar G nP* **by** (*metis equals0D sinvar-no-offending*)
from *valid-without-offending-flows*[*OF a-offending*] **have** *a-offending-rm*: *sinvar* (*delete-edges G f*) *nP* **.**
from *a-enf* **have** *a-enf′*: ⋀ *G nP*. *sinvar G nP* = (∀ (*e1*, *e2*)∈(*edges G*). *e1* ≠ *e2* ⟶ *P* (*nP e1*) *e1* (*nP e2*) *e2*)
using *sinvar-all-edges-normal-form-not-refl-SR-def* **by** *simp*

from *ENFnrSR-offending-set*[*OF a-enf*] *a-not-eval a-offending* **have** *a-f-3-in-f*: ⋀ *s r*. (*s*, *r*)∈*f* ⟹ ¬ *P* (*nP s*) *s* (*nP r*) *r* **by** *simp*
from *ENFnrSR-offending-set*[*OF a-enf*] *a-not-eval a-offending* **have** *a-f-3-neq*: ⋀ *s r*. (*s*, *r*)∈*f* ⟹

s ≠ *r* **by** *simp*

let *?nP′* = (*nP*(*i* := ⊥))

from *ENFnrSR-offending-set*[*OF a-enf*] *a-not-eval a-offending a-i-snds*
obtain *e1 e2* **where** *e1e2cond*: (*e1*, *e2*)∈*f* ∧ *e2* = *i* **by** *fastforce*

from *conjunct1*[*OF e1e2cond*] *a-offending* **have** (*e1*, *e2*) ∈ *edges G*
by (*metis* (*lifting*, *no-types*) *SecurityInvariant-withOffendingFlows.set-offending-flows-def mem-Collect-eq rev-subsetD*)

from *conjunct1*[*OF e1e2cond*] *a-f-3-in-f* **have** *e1e2notP*: ¬ *P* (*nP e1*) *e1* (*nP e2*) *e2* **by** *simp*
from *e1e2notP a-weakrefl* **have** *e1ore2neqbot*: (*nP e1*) ≠ ⊥ ∨ (*nP e2*) ≠ ⊥ **by** *fastforce*
from *e1e2notP a-bottoall* **have** *x1*: (*nP e1*) ≠ ⊥ **by** *fastforce*
from *this e1e2notP a-botdefault* **have** *x2*: ¬ *P* (*nP e1*) *e1* ⊥ *e2* **by** *fast*
from *this e1e2notP* **have** *e1e2subgoal1*: ¬ *P* (*nP e1*) *e1* (*?nP′ e2*) *e2* **by** *auto*

from *a-f-3-neq e1e2cond* **have** *e2* ≠ *e1* **by** *blast*

from *e1e2subgoal1* **have** *e1* ≠ *e2* ⟹ ¬ *P* (*?nP′ e1*) *e1* (*?nP′ e2*) *e2* **by**(*simp add*: *e1e2cond*)

from *this* ‹*e2* ≠ *e1*› *ENFnrSR-offending-set*[*OF a-enf*] *a-offending* ‹(*e1*, *e2*) ∈ *edges G*› **have**
∃ (*e1*, *e2*)∈(*edges G*). *e2* ≠ *e1* ∧ ¬ *P* (*?nP′ e1*) *e1* (*?nP′ e2*) *e2* **by** *fastforce*
hence ¬ (∀ (*e1*, *e2*)∈(*edges G*). *e2* ≠ *e1* ⟶ *P* ((*nP*(*i* := ⊥)) *e1*) *e1* ((*nP*(*i* := ⊥)) *e2*) *e2*) **by** *fast*
from *this a-enf′* **show** ¬ *sinvar G* (*nP*(*i* := ⊥)) **by** *fast*
qed

4.4 edges normal form not refl ENFnr

definition (**in** *SecurityInvariant-withOffendingFlows*) *sinvar-all-edges-normal-form-not-refl* :: (′*a* ⇒ ′*a* ⇒ *bool*) ⇒ *bool* **where**
sinvar-all-edges-normal-form-not-refl P ≡ ∀ *G nP*. *sinvar G nP* = (∀ (*e1*, *e2*) ∈ *edges G*. *e1* ≠ *e2* ⟶ *P* (*nP e1*) (*nP e2*))

we derive everything from the ENFnrSR form

lemma (**in** *SecurityInvariant-withOffendingFlows*) *ENFnr-to-ENFnrSR*:
sinvar-all-edges-normal-form-not-refl P ⟹ *sinvar-all-edges-normal-form-not-refl-SR* (λ *v1* - *v2* -. *P v1 v2*)
by(*simp add*: *sinvar-all-edges-normal-form-not-refl-def sinvar-all-edges-normal-form-not-refl-SR-def*)

4.4.1 Offending Flows

theorem (**in** *SecurityInvariant-withOffendingFlows*) *ENFnr-offending-set*:
⟦ *sinvar-all-edges-normal-form-not-refl P* ⟧ ⟹
set-offending-flows G nP = (*if sinvar G nP then*
{}
else
{ {(*e1*,*e2*). (*e1*, *e2*) ∈ *edges G* ∧ *e1* ≠ *e2* ∧ ¬ *P* (*nP e1*) (*nP e2*)} })
apply(*drule ENFnr-to-ENFnrSR*)
by(*drule*(*1*) *ENFnrSR-offending-set*)

4.4.2 Instance helper

lemma (**in** *SecurityInvariant-withOffendingFlows*) *ENFnr-fsts-weakrefl-instance*:

fixes *default-node-properties* :: *'a* (‹⊥›)
assumes *a-enf*: *sinvar-all-edges-normal-form-not-refl P*
and *a-botdefault*: $\forall$ *e1 e2*. *e2* $\neq \bot \longrightarrow \neg$ *P e1 e2* $\longrightarrow \neg$ *P* $\bot$ *e2*
and *a-alltobot*: $\forall$ *e1*. *P e1* $\bot$
and *a-offending*: *f* $\in$ *set-offending-flows G nP*
and *a-i-fsts*: *i* $\in$ *fst' f*
shows
$\neg$ *sinvar G* (*nP*(*i* := $\bot$))
proof −
from *assms* **show** *?thesis*
apply −
apply(*drule ENFnr-to-ENFnrSR*)
apply(*drule ENFnrSR-fsts-weakrefl-instance*)
by *auto*
qed

lemma (**in** *SecurityInvariant-withOffendingFlows*) *ENFnr-snds-weakrefl-instance*:
fixes *default-node-properties* :: *'a* (‹⊥›)
assumes *a-enf*: *sinvar-all-edges-normal-form-not-refl P*
and *a-botdefault*: $\forall$ *e1 e2*. $\neg$ *P e1 e2* $\longrightarrow \neg$ *P e1* $\bot$
and *a-bottoall*: $\forall$ *e2*. *P* $\bot$ *e2*
and *a-offending*: *f* $\in$ *set-offending-flows G nP*
and *a-i-snds*: *i* $\in$ *snd' f*
shows
$\neg$ *sinvar G* (*nP*(*i* := $\bot$))
proof −
from *assms* **show** *?thesis*
apply −
apply(*drule ENFnr-to-ENFnrSR*)
apply(*drule ENFnrSR-snds-weakrefl-instance*)
by *auto*
qed

lemma (**in** *SecurityInvariant-withOffendingFlows*) *ENF-weakrefl-instance-FALSE*:
fixes *default-node-properties* :: *'a* (‹⊥›)
assumes *a-wfG*: *wf-graph G*
and *a-not-eval*: $\neg$ *sinvar G nP*
and *a-enf*: *sinvar-all-edges-normal-form P*
and *a-weakrefl*: *P* $\bot$ $\bot$
and *a-botisolated*: $\bigwedge$ *e2*. *e2* $\neq \bot \Longrightarrow \neg$ *P* $\bot$ *e2*
and *a-botdefault*: $\bigwedge$ *e1 e2*. *e1* $\neq \bot \Longrightarrow \neg$ *P e1 e2* $\Longrightarrow \neg$ *P e1* $\bot$
and *a-offending*: *f* $\in$ *set-offending-flows G nP*
and *a-offending-rm*: *sinvar* (*delete-edges G f*) *nP*
and *a-i-fsts*: *i* $\in$ *snd' f*
and *a-not-eval-upd*: $\neg$ *sinvar G* (*nP*(*i* := $\bot$))
shows *False*
oops

end
theory *vertex-example-simps*
imports *Lib/FiniteGraph TopoS-Vertices*
beginend
theory *TopoS-Helper*
imports *Main TopoS-Interface*
TopoS-ENF
vertex-example-simps
begin

lemma (**in** *SecurityInvariant-preliminaries*) *sinvar-valid-remove-flattened-offending-flows*:
assumes *wf-graph* (|*nodes* = *nodesG*, *edges* = *edgesG*|)
shows *sinvar* (|*nodes* = *nodesG*, *edges* = *edgesG* − ⋃ (*set-offending-flows* (|*nodes* = *nodesG*, *edges* = *edgesG*|) *nP*) |) *nP*
proof −
{ **fix** *f*
assume ∗: *f*∈*set-offending-flows* (|*nodes* = *nodesG*, *edges* = *edgesG*|) *nP*

from ∗ **have** *1*: *sinvar* (|*nodes* = *nodesG*, *edges* = *edgesG* − *f* |) *nP*
by (*metis* (*opaque-lifting*, *mono-tags*) *SecurityInvariant-withOffendingFlows.valid-without-offending-flows delete-edges-simp2 graph.select-convs*(*1*) *graph.select-convs*(*2*))
from ∗ **have** *2*: *edgesG* − ⋃ (*set-offending-flows* (|*nodes* = *nodesG*, *edges* = *edgesG*|) *nP*) ⊆ *edgesG* − *f*
by *blast*
note *1 2*
}
with *assms* **show** *?thesis*
by (*metis* (*opaque-lifting*, *no-types*) *Diff-empty Union-empty defined-offending equals0I mono-sinvar wf-graph-remove-edges*)
qed

lemma (**in** *SecurityInvariant-preliminaries*) *sinvar-valid-remove-SOME-offending-flows*:
assumes *set-offending-flows* (|*nodes* = *nodesG*, *edges* = *edgesG*|) *nP* ≠ {}
shows *sinvar* (|*nodes* = *nodesG*, *edges* = *edgesG* − (*SOME F*. *F* ∈ *set-offending-flows* (|*nodes* = *nodesG*, *edges* = *edgesG*|) *nP*) |) *nP*
proof −
{ **fix** *f*
assume ∗: *f*∈*set-offending-flows* (|*nodes* = *nodesG*, *edges* = *edgesG*|) *nP*

from ∗ **have** *1*: *sinvar* (|*nodes* = *nodesG*, *edges* = *edgesG* − *f* |) *nP*
by (*metis* (*opaque-lifting*, *mono-tags*) *SecurityInvariant-withOffendingFlows.valid-without-offending-flows delete-edges-simp2 graph.select-convs*(*1*) *graph.select-convs*(*2*))
from ∗ **have** *2*: *edgesG* − ⋃ (*set-offending-flows* (|*nodes* = *nodesG*, *edges* = *edgesG*|) *nP*) ⊆ *edgesG* − *f*
by *blast*
note *1 2*
}
with *assms* **show** *?thesis* **by** (*simp add*: *some-in-eq*)
qed

lemma (**in** *SecurityInvariant-preliminaries*) *sinvar-valid-remove-minimalize-offending-overapprox*:

assumes *wf-graph* ⦇*nodes* = *nodesG*, *edges* = *edgesG*⦈
and ¬ *sinvar* ⦇*nodes* = *nodesG*, *edges* = *edgesG*⦈ *nP*
and *set Es* = *edgesG* **and** *distinct Es*
shows *sinvar* ⦇*nodes* = *nodesG*, *edges* = *edgesG* −
set (*minimalize-offending-overapprox Es* [] ⦇*nodes* = *nodesG*, *edges* = *edgesG*⦈ *nP*) ⦈ *nP*
proof −
from *assms* **have** *off-Es*: *is-offending-flows* (*set Es*) ⦇*nodes* = *nodesG*, *edges* = *edgesG*⦈ *nP*
by (*metis* (*no-types*, *lifting*) *Diff-cancel*
SecurityInvariant-withOffendingFlows.valid-empty-edges-iff-exists-offending-flows defined-offending
delete-edges-simp2 graph.select-convs(*2*) *is-offending-flows-def sinvar-monoI*)
from *minimalize-offending-overapprox-gives-back-an-offending-flow*[*OF assms*(*1*) *off-Es* - *assms*(*4*)]
have
in-offending: *set* (*minimalize-offending-overapprox Es* [] ⦇*nodes* = *nodesG*, *edges* = *edgesG*⦈ *nP*)
∈ *set-offending-flows* ⦇*nodes* = *nodesG*, *edges* = *edgesG*⦈ *nP*
using *assms*(*3*) **by** *simp*

{ fix *f*
assume ∗: *f*∈*set-offending-flows* ⦇*nodes* = *nodesG*, *edges* = *edgesG*⦈ *nP*
from ∗ **have** *1*: *sinvar* ⦇*nodes* = *nodesG*, *edges* = *edgesG* − *f* ⦈ *nP*
by (*metis* (*opaque-lifting*, *mono-tags*) *SecurityInvariant-withOffendingFlows.valid-without-offending-flows*
delete-edges-simp2 graph.select-convs(*1*) *graph.select-convs*(*2*))
note *1*
}
with *in-offending* **show** *?thesis* **by** (*simp add*: *some-in-eq*)
qed

end
theory *SINVAR-Subnets2*
imports../*TopoS-Helper*
begin

4.5 SecurityInvariant Subnets2

Warning, This is just a test. Please look at `SINVAR_Subnets.thy`. This security invariant has the following changes, compared to `SINVAR_Subnets.thy`: A new BorderRouter' is introduced which can send to the members of its subnet. A new InboundRouter is accessible by anyone. It can access all other routers and the outside.

datatype *subnets* = *Subnet nat* | *BorderRouter nat* | *BorderRouter′ nat* | *InboundRouter* | *Unassigned*

definition *default-node-properties* :: *subnets*
where *default-node-properties* ≡ *Unassigned*

fun *allowed-subnet-flow* :: *subnets* ⇒ *subnets* ⇒ *bool* **where**
allowed-subnet-flow (*Subnet s1*) (*Subnet s2*) = (*s1* = *s2*) |
allowed-subnet-flow (*Subnet s1*) (*BorderRouter s2*) = (*s1* = *s2*) |
allowed-subnet-flow (*Subnet s1*) (*BorderRouter′ s2*) = (*s1* = *s2*) |
allowed-subnet-flow (*Subnet* -) - = *True* |
allowed-subnet-flow (*BorderRouter* -) (*Subnet* -) = *False* |
allowed-subnet-flow (*BorderRouter* -) - = *True* |
allowed-subnet-flow (*BorderRouter′ s1*) (*Subnet s2*) = (*s1* = *s2*) |
allowed-subnet-flow (*BorderRouter′* -) - = *True* |
allowed-subnet-flow InboundRouter (*Subnet* -) = *False* |
allowed-subnet-flow InboundRouter - = *True* |

allowed-subnet-flow Unassigned Unassigned = *True* |
allowed-subnet-flow Unassigned InboundRouter = *True* |
allowed-subnet-flow Unassigned - = *False*

fun *sinvar* :: *'v graph* ⇒ (*'v* ⇒ *subnets*) ⇒ *bool* **where**
sinvar G nP = (∀ (*e1*,*e2*) ∈ *edges G*. *allowed-subnet-flow* (*nP e1*) (*nP e2*))

definition *receiver-violation* :: *bool* **where** *receiver-violation* = *False*

Only members of the same subnet or their *BorderRouter'* can access them.

lemma *allowed-subnet-flow a* (*Subnet s1*) ⟹ *a* = (*BorderRouter' s1*) ∨ *a* = (*Subnet s1*)
apply(*cases a*)
apply(*simp-all*)
done

4.5.1 Preliminaries

lemma *sinvar-mono*: *SecurityInvariant-withOffendingFlows.sinvar-mono sinvar*
apply(*simp only*: *SecurityInvariant-withOffendingFlows.sinvar-mono-def*)
apply(*clarify*)
by *auto*

interpretation *SecurityInvariant-preliminaries*
where *sinvar* = *sinvar*
apply *unfold-locales*
apply(*frule-tac finite-distinct-list*[*OF wf-graph.finiteE*])
apply(*erule-tac exE*)
apply(*rename-tac list-edges*)
apply(*rule-tac ff=list-edges* **in** *SecurityInvariant-withOffendingFlows.mono-imp-set-offending-flows-not-empty*[*OF sinvar-mono*])
apply(*auto*)[*6*]
apply(*auto simp add*: *SecurityInvariant-withOffendingFlows.is-offending-flows-def graph-ops*)[*1*]
apply(*fact SecurityInvariant-withOffendingFlows.sinvar-mono-imp-is-offending-flows-mono*[*OF sinvar-mono*])
done

4.5.2 ENF

lemma *All-to-Unassigned*: ∀ *e1*. *allowed-subnet-flow e1 Unassigned*
by (*rule allI*, *case-tac e1*, *simp-all*)
lemma *Unassigned-default-candidate*: ∀ *nP e1 e2*. ¬ *allowed-subnet-flow* (*nP e1*) (*nP e2*) ⟶ ¬ *allowed-subnet-flow Unassigned* (*nP e2*)
apply(*intro allI*)
apply(*case-tac nP e2*)
apply *simp-all*
apply(*case-tac nP e1*)
apply *simp-all*
by(*simp add*: *All-to-Unassigned*)
lemma *allowed-subnet-flow-refl*: ∀ *e*. *allowed-subnet-flow e e*
by(*rule allI*, *case-tac e*, *simp-all*)
lemma *Subnets-ENF*: *SecurityInvariant-withOffendingFlows.sinvar-all-edges-normal-form sinvar allowed-subnet-flow*
unfolding *SecurityInvariant-withOffendingFlows.sinvar-all-edges-normal-form-def*

by *simp*
lemma *Subnets-ENF-refl*: *SecurityInvariant-withOffendingFlows.ENF-refl sinvar allowed-subnet-flow*
unfolding *SecurityInvariant-withOffendingFlows.ENF-refl-def*
apply(*rule conjI*)
apply(*simp add*: *Subnets-ENF*)
apply(*simp add*: *allowed-subnet-flow-refl*)
done

definition *Subnets-offending-set*:: *'v graph* ⇒ (*'v* ⇒ *subnets*) ⇒ (*'v* × *'v*) *set set* **where**
Subnets-offending-set G nP = (*if sinvar G nP then*
{}
else
{ {*e* ∈ *edges G*. *case e of* (*e1*,*e2*) ⇒ ¬ *allowed-subnet-flow* (*nP e1*) (*nP e2*)} })
lemma *Subnets-offending-set*:
SecurityInvariant-withOffendingFlows.set-offending-flows sinvar = *Subnets-offending-set*
apply(*simp only*: *fun-eq-iff ENF-offending-set*[*OF Subnets-ENF*] *Subnets-offending-set-def*)
apply(*rule allI*)+
apply(*rename-tac G nP*)
apply(*auto*)
done

interpretation *Subnets*: *SecurityInvariant-ACS*
where *default-node-properties* = *SINVAR-Subnets2.default-node-properties*
and *sinvar* = *SINVAR-Subnets2.sinvar*
rewrites *SecurityInvariant-withOffendingFlows.set-offending-flows sinvar* = *Subnets-offending-set*
unfolding *SINVAR-Subnets2.default-node-properties-def*
apply *unfold-locales*
apply(*rule ballI*)
apply (*rule SecurityInvariant-withOffendingFlows.ENF-fsts-refl-instance*[*OF Subnets-ENF-refl Unassigned-default-candidate*])[*1*]
apply(*simp-all*)[*2*]
apply(*erule default-uniqueness-by-counterexample-ACS*)
apply (*simp add*: *SecurityInvariant-withOffendingFlows.set-offending-flows-def*
SecurityInvariant-withOffendingFlows.is-offending-flows-min-set-def
SecurityInvariant-withOffendingFlows.is-offending-flows-def)
apply (*simp add*:*graph-ops*)
apply (*simp split*: *prod.split-asm prod.split*)
apply(*rule-tac x*=⦇ *nodes*={*vertex-1*,*vertex-2*}, *edges* = {(*vertex-1*,*vertex-2*)} ⦈ **in** *exI*, *simp*)
apply(*rule conjI*)
apply(*simp add*: *wf-graph-def*)
apply(*case-tac otherbot*, *simp-all*)
apply(*rename-tac mysubnetcase*)
apply(*rule-tac x*=(λ *x*. *Unassigned*)(*vertex-1* := *Unassigned*, *vertex-2* := *BorderRouter mysubnetcase*) **in** *exI*, *simp*)
apply(*rule-tac x*=*vertex-1* **in** *exI*, *simp*)
apply(*rule-tac x*={(*vertex-1*,*vertex-2*)} **in** *exI*, *simp*)
apply(*rule-tac x*=(λ *x*. *Unassigned*)(*vertex-1* := *Unassigned*, *vertex-2* := *BorderRouter whatever*) **in** *exI*, *simp*)
apply(*rule-tac x*=*vertex-1* **in** *exI*, *simp*)
apply(*rule-tac x*={(*vertex-1*,*vertex-2*)} **in** *exI*, *simp*)
apply(*rule-tac x*=(λ *x*. *Unassigned*)(*vertex-1* := *Unassigned*, *vertex-2* := *BorderRouter whatever*) **in** *exI*, *simp*)

apply(*rule-tac x=vertex-1* **in** *exI*, *simp*)
apply(*rule-tac x={(vertex-1,vertex-2)}* **in** *exI*, *simp*)
apply(*rule-tac x=(λ x. Unassigned)(vertex-1 := Unassigned, vertex-2 := BorderRouter whatever)* **in** *exI*, *simp*)
apply(*rule-tac x=vertex-1* **in** *exI*, *simp*)
apply(*rule-tac x={(vertex-1,vertex-2)}* **in** *exI*, *simp*)
apply(*fact Subnets-offending-set*)
done

lemma *TopoS-Subnets2*: *SecurityInvariant sinvar default-node-properties receiver-violation*
unfolding *receiver-violation-def* **by** *unfold-locales*

hide-fact (**open**) *sinvar-mono*
hide-const (**open**) *sinvar receiver-violation default-node-properties*

end
theory *SINVAR-BLPstrict*
imports ../*TopoS-Helper*
begin

4.6 Stricter Bell LaPadula SecurityInvariant

All unclassified data sources must be labeled, default assumption: all is secret.

Warning: This is considered here an access control strategy. By default, everything is secret and one explicitly prohibits sending to non-secret hosts.

datatype *security-level* = *Unclassified* | *Confidential* | *Secret*

instantiation *security-level* :: *linorder*
begin
fun *less-eq-security-level* :: *security-level* ⇒ *security-level* ⇒ *bool* **where**
(*Unclassified* ≤ *Unclassified*) = *True* |
(*Confidential* ≤ *Confidential*) = *True* |
(*Secret* ≤ *Secret*) = *True* |
(*Unclassified* ≤ *Confidential*) = *True* |
(*Confidential* ≤ *Secret*) = *True* |
(*Unclassified* ≤ *Secret*) = *True* |
(*Secret* ≤ *Confidential*) = *False* |
(*Confidential* ≤ *Unclassified*) = *False* |
(*Secret* ≤ *Unclassified*) = *False*

fun *less-security-level* :: *security-level* ⇒ *security-level* ⇒ *bool* **where**
(*Unclassified* < *Unclassified*) = *False* |
(*Confidential* < *Confidential*) = *False* |
(*Secret* < *Secret*) = *False* |
(*Unclassified* < *Confidential*) = *True* |
(*Confidential* < *Secret*) = *True* |
(*Unclassified* < *Secret*) = *True* |
(*Secret* < *Confidential*) = *False* |
(*Confidential* < *Unclassified*) = *False* |
(*Secret* < *Unclassified*) = *False*

instance
apply(*intro-classes*)
apply(*case-tac* [!] *x*)
apply(*simp-all*)
apply(*case-tac* [!] *y*)
apply(*simp-all*)
apply(*case-tac* [!] *z*)
apply(*simp-all*)
done
end

definition *default-node-properties* :: *security-level*
where *default-node-properties* ≡ *Secret*

fun *sinvar* :: *'v graph* ⇒ (*'v* ⇒ *security-level*) ⇒ *bool* **where**
sinvar G nP = (∀ (*e1*,*e2*) ∈ *edges G*. (*nP e1*) ≤ (*nP e2*))

definition *receiver-violation* :: *bool* **where** *receiver-violation* ≡ *False*

lemma *sinvar-mono*: *SecurityInvariant-withOffendingFlows.sinvar-mono sinvar*
apply(*simp only*: *SecurityInvariant-withOffendingFlows.sinvar-mono-def*)
apply(*clarify*)
by *auto*

interpretation *SecurityInvariant-preliminaries*
where *sinvar* = *sinvar*
apply *unfold-locales*
apply(*frule-tac finite-distinct-list*[*OF wf-graph.finiteE*])
apply(*erule-tac exE*)
apply(*rename-tac list-edges*)
apply(*rule-tac ff*=*list-edges* **in** *SecurityInvariant-withOffendingFlows.mono-imp-set-offending-flows-not-empty*[*OF sinvar-mono*])
apply(*auto*)[*6*]
apply(*auto simp add*: *SecurityInvariant-withOffendingFlows.is-offending-flows-def graph-ops*)[*1*]
apply(*fact SecurityInvariant-withOffendingFlows.sinvar-mono-imp-is-offending-flows-mono*[*OF sinvar-mono*])
done

4.7 ENF

lemma *secret-default-candidate*: ⋀ (*nP*::(*'v* ⇒ *security-level*)) *e1 e2*. ¬ (*nP e1*) ≤ (*nP e2*) ⟹ ¬ *Secret* ≤ (*nP e2*)
apply(*case-tac nP e1*)
apply(*simp-all*)
apply(*case-tac* [!] *nP e2*)
apply(*simp-all*)
done
lemma *BLP-ENF*: *SecurityInvariant-withOffendingFlows.sinvar-all-edges-normal-form sinvar* (≤)

```
    unfolding SecurityInvariant-withOffendingFlows.sinvar-all-edges-normal-form-def
    by simp
  lemma BLP-ENF-refl: SecurityInvariant-withOffendingFlows.ENF-refl sinvar (≤)
    unfolding SecurityInvariant-withOffendingFlows.ENF-refl-def
    apply(rule conjI)
     apply(simp add: BLP-ENF)
    apply(simp)
  done

  definition BLP-offending-set:: 'v graph ⇒ ('v ⇒ security-level) ⇒ ('v × 'v) set set where
  BLP-offending-set G nP = (if sinvar G nP then
      {}
     else
      { {e ∈ edges G. case e of (e1,e2) ⇒ (nP e1) > (nP e2)} })
 lemma BLP-offending-set: SecurityInvariant-withOffendingFlows.set-offending-flows sinvar = BLP-offending-set
   apply(simp only: fun-eq-iff SecurityInvariant-withOffendingFlows.ENF-offending-set[OF BLP-ENF]
BLP-offending-set-def)
    apply(rule allI)+
    apply(rename-tac G nP)
    apply(auto)
  done

  interpretation BLPstrict: SecurityInvariant-ACS sinvar default-node-properties

  rewrites SecurityInvariant-withOffendingFlows.set-offending-flows sinvar = BLP-offending-set
    unfolding default-node-properties-def
    apply(unfold-locales)
      apply(rule ballI)
      apply(rule SecurityInvariant-withOffendingFlows.ENF-fsts-refl-instance[OF BLP-ENF-refl])
         apply(simp-all add: BLP-ENF BLP-ENF-refl)[3]
      apply(simp add: secret-default-candidate; fail)
     apply(erule default-uniqueness-by-counterexample-ACS)
    apply(rule-tac x=(| nodes=set [vertex-1,vertex-2], edges = set [(vertex-1,vertex-2)] |) in exI, simp)
     apply(simp add: BLP-offending-set graph-ops wf-graph-def)
     apply(rule-tac x=(λ x. Secret)(vertex-1 := Secret, vertex-2 := Confidential) in exI, simp)
     apply(rule-tac x=vertex-1 in exI, simp)
     apply(rule-tac x=set [(vertex-1,vertex-2)] in exI, simp)
     apply(simp add: BLP-offending-set-def)
     apply(rule conjI)
      apply fastforce
     apply (case-tac otherbot, simp-all)
    apply(fact BLP-offending-set)
   done


  lemma TopoS-BLPstrict: SecurityInvariant sinvar default-node-properties receiver-violation
  unfolding receiver-violation-def by unfold-locales

hide-fact (open) sinvar-mono

hide-const (open) sinvar receiver-violation default-node-properties

end
theory SINVAR-Tainting
```

imports ../*TopoS-Helper*
begin

4.8 SecurityInvariant Tainting for IFS

context
begin

```
qualified type-synonym taints = string set
```

Warning: an infinite set has cardinality 0

```
lemma card (UNIV::taints) = 0 by (simp add: infinite-UNIV-listI)

qualified definition default-node-properties :: taints
  where default-node-properties ≡ {}
```

For all nodes n in the graph, for all nodes r which are reachable from n, node n needs the appropriate tainting fields which are set by r

```
definition sinvar-tainting :: 'v graph ⇒ ('v ⇒ taints) ⇒ bool where
  sinvar-tainting G nP ≡ ∀ n ∈ (nodes G). ∀ r ∈ (succ-tran G n). nP n ⊆ nP r


private lemma sinvar-tainting-edges-def: wf-graph G ⟹
  sinvar-tainting G nP ⟷ (∀ (v1,v2) ∈ edges G. ∀ r ∈ (succ-tran G v1). nP v1 ⊆ nP r)
  unfolding sinvar-tainting-def
  proof
    assume a1: wf-graph G
      and a2: ∀n∈nodes G. ∀r∈succ-tran G n. nP n ⊆ nP r
      from a1[simplified wf-graph-def] have f1: fst ‘ edges G ⊆ nodes G by simp
      from f1 a2 have ∀v ∈ (fst ‘ edges G). ∀r∈succ-tran G v. nP v ⊆ nP r by auto
      thus ∀(v1, -)∈edges G. ∀r∈succ-tran G v1. nP v1 ⊆ nP r by fastforce
    next
    assume a1: wf-graph G
      and a2: ∀(v1, v2)∈edges G. ∀r∈succ-tran G v1. nP v1 ⊆ nP r
      from a2 have g1: ∀ v ∈ (fst ‘ edges G). ∀r∈succ-tran G v. nP v ⊆ nP r by fastforce
      from FiniteGraph.succ-tran-empty[OF a1]
      have g2: ∀ v. v ∉ (fst ‘ edges G) ⟶ (∀r∈succ-tran G v. nP v ⊆ nP r) by blast
      from g1 g2 show ∀n∈nodes G. ∀r∈succ-tran G n. nP n ⊆ nP r by metis
  qed
```

Alternative definition of the *sinvar-tainting*

```
qualified definition sinvar :: 'v graph ⇒ ('v ⇒ taints) ⇒ bool where
  sinvar G nP ≡ ∀ (v1,v2) ∈ edges G. nP v1 ⊆ nP v2


qualified lemma sinvar-preferred-def:
  wf-graph G ⟹ sinvar-tainting G nP = sinvar G nP
  proof(unfold sinvar-tainting-edges-def sinvar-def, rule iffI, goal-cases)
  case 2
    have (v, v') ∈ (edges G)⁺ ⟹ nP v ⊆ nP v' for v v'
      proof(induction rule: trancl-induct)
      case base thus ?case using 2(2) by fastforce
      next
```

```
      case step thus ?case using 2(2) by fastforce
      qed
    thus ?case
    by(simp add: succ-tran-def)
  next
  case 1
    from 1(1)[simplified wf-graph-def] have f1: fst ` edges G ⊆ nodes G by simp
    from f1 1(2) have ∀ v ∈ (fst ` edges G). ∀ v'∈succ-tran G v. nP v ⊆ nP v' by fastforce
    thus ?case unfolding succ-tran-def by fastforce
  qed
```

Information Flow Security

```
qualified definition receiver-violation :: bool where receiver-violation ≡ True


private lemma sinvar-mono: SecurityInvariant-withOffendingFlows.sinvar-mono sinvar
  apply(simp add: SecurityInvariant-withOffendingFlows.sinvar-mono-def sinvar-def)
  apply(clarify)
  by blast
interpretation SecurityInvariant-preliminaries
where sinvar = sinvar
proof(unfold-locales, goal-cases)
  case (1 G nP)
    from 1 show ?case
    apply(frule-tac finite-distinct-list[OF wf-graph.finiteE])
    apply(erule-tac exE)
    apply(rename-tac list-edges)
   apply(rule-tac ff=list-edges in SecurityInvariant-withOffendingFlows.mono-imp-set-offending-flows-not-empty[OF
sinvar-mono])
      apply(auto simp add: sinvar-def)
     apply(auto simp add: sinvar-def SecurityInvariant-withOffendingFlows.is-offending-flows-def
graph-ops)[1]
    done
  next
  case (2 N E E' nP) thus ?case by(simp add: sinvar-def) blast
  next
 case 3 thus ?case by(fact SecurityInvariant-withOffendingFlows.sinvar-mono-imp-is-offending-flows-mono[OF
sinvar-mono])
qed



private  lemma Taints-def-unique: otherbot ≠ {} ⟹
  ∃ G p i f. wf-graph G ∧ ¬ sinvar G p ∧ f ∈ (SecurityInvariant-withOffendingFlows.set-offending-flows
sinvar G p) ∧
      sinvar (delete-edges G f) p ∧
       i ∈ snd ` f ∧ sinvar G (p(i := otherbot))
  apply(simp)
  apply (simp add: SecurityInvariant-withOffendingFlows.set-offending-flows-def
    SecurityInvariant-withOffendingFlows.is-offending-flows-min-set-def
    SecurityInvariant-withOffendingFlows.is-offending-flows-def)
  apply (simp add:graph-ops)
  apply (simp split: prod.split-asm prod.split)
 apply(rule-tac x=⦇ nodes=set [vertex-1,vertex-2], edges = set [(vertex-1,vertex-2)] ⦈ in exI, simp)
```

```
apply(rule conjI)
 apply(simp add: wf-graph-def; fail)
apply(subgoal-tac ∃foo. foo ∈ otherbot)
 prefer 2
 subgoal by fastforce
apply(erule exE, rename-tac foo)
apply(rule-tac x=(λ x. {})(vertex-1 := {foo}, vertex-2 := {}) in exI)
apply(rule conjI)
 apply(simp add: sinvar-def; fail)
apply(rule-tac x=vertex-2 in exI)
apply(rule-tac x=set [(vertex-1,vertex-2)] in exI, simp)
apply(simp add: sinvar-def)
done
```

4.8.1 ENF

```
private lemma Taints-ENF: SecurityInvariant-withOffendingFlows.sinvar-all-edges-normal-form sinvar (⊆)
  unfolding SecurityInvariant-withOffendingFlows.sinvar-all-edges-normal-form-def sinvar-def
  by simp
private lemma Taints-ENF-refl: SecurityInvariant-withOffendingFlows.ENF-refl sinvar (⊆)
  unfolding SecurityInvariant-withOffendingFlows.ENF-refl-def
   by(auto simp add: Taints-ENF)

qualified definition Taints-offending-set:: 'v graph ⇒ ('v ⇒ taints) ⇒ ('v × 'v) set set where
Taints-offending-set G nP = (if sinvar G nP then
    {}
   else
    { {e ∈ edges G. case e of (e1,e2) ⇒ ¬ (nP e1) ⊆ (nP e2)} })
 lemma Taints-offending-set: SecurityInvariant-withOffendingFlows.set-offending-flows sinvar = Taints-offending-set
   by(auto simp add: fun-eq-iff
                SecurityInvariant-withOffendingFlows.ENF-offending-set[OF Taints-ENF]
                Taints-offending-set-def)


interpretation Taints: SecurityInvariant-IFS sinvar default-node-properties
rewrites SecurityInvariant-withOffendingFlows.set-offending-flows sinvar = Taints-offending-set
  unfolding receiver-violation-def
  unfolding default-node-properties-def
  proof(unfold-locales, goal-cases)
  case 1
    from 1(2) show ?case
    apply(intro ballI)
    apply(rule SecurityInvariant-withOffendingFlows.ENF-snds-refl-instance[OF Taints-ENF-refl])
      apply(simp-all add: Taints-ENF Taints-ENF-refl)
    by blast
  next
  case 2 thus ?case
    proof(elim default-uniqueness-by-counterexample-IFS)
    qed(fact Taints-def-unique)
  next
  case 3 show set-offending-flows = Taints-offending-set by(fact Taints-offending-set)
 qed
```

lemma *TopoS-Tainting*: *SecurityInvariant sinvar default-node-properties receiver-violation*
unfolding *receiver-violation-def* **by** *unfold-locales*

end

end
theory *SINVAR-BLPbasic*
imports *../TopoS-Helper*
begin

4.9 SecurityInvariant Basic Bell LaPadula

type-synonym *security-level* = *nat*

definition *default-node-properties* :: *security-level*
where *default-node-properties* ≡ *0*

fun *sinvar* :: *'v graph* ⇒ (*'v* ⇒ *security-level*) ⇒ *bool* **where**
sinvar G nP = (∀ (*e1*,*e2*) ∈ *edges G*. (*nP e1*) ≤ (*nP e2*))

What we call a *security-level* is also referred to as security label (or security clearance of subjects and classification of objects) in the literature. The lowest security level is *0*, which can be understood as unclassified. Consequently, 1 = confidential, 2 = secret, 3 = topSecret, ... The total order of the security levels corresponds to the total order of the natural numbers ≤. It is important that there is smallest security level (i.e. *default-node-properties*), otherwise, a unique and secure default parameter could not exist. Hence, it is not possible to extend the security levels to *int* to model unlimited "un-confidentialness".

definition *receiver-violation* :: *bool* **where** *receiver-violation* ≡ *True*

lemma *sinvar-mono*: *SecurityInvariant-withOffendingFlows.sinvar-mono sinvar*
apply(*simp only*: *SecurityInvariant-withOffendingFlows.sinvar-mono-def*)
apply(*clarify*)
by *auto*

interpretation *SecurityInvariant-preliminaries*
where *sinvar* = *sinvar*
apply *unfold-locales*
apply(*frule-tac finite-distinct-list*[*OF wf-graph.finiteE*])
apply(*erule-tac exE*)
apply(*rename-tac list-edges*)
apply(*rule-tac ff=list-edges* **in** *SecurityInvariant-withOffendingFlows.mono-imp-set-offending-flows-not-empty*[*OF sinvar-mono*])
apply(*auto*)[*6*]
apply(*auto simp add*: *SecurityInvariant-withOffendingFlows.is-offending-flows-def graph-ops*)[*1*]
apply(*fact SecurityInvariant-withOffendingFlows.sinvar-mono-imp-is-offending-flows-mono*[*OF sinvar-mono*])
done

lemma *BLP-def-unique*: *otherbot* ≠ *0* ⟹
∃ *G p i f*. *wf-graph G* ∧ ¬ *sinvar G p* ∧ *f* ∈ (*SecurityInvariant-withOffendingFlows.set-offending-flows sinvar G p*) ∧
sinvar (*delete-edges G f*) *p* ∧
i ∈ *snd* ' *f* ∧ *sinvar G* (*p*(*i* := *otherbot*))
apply(*simp*)
apply (*simp add*: *SecurityInvariant-withOffendingFlows.set-offending-flows-def*
SecurityInvariant-withOffendingFlows.is-offending-flows-min-set-def
SecurityInvariant-withOffendingFlows.is-offending-flows-def)
apply (*simp add*:*graph-ops*)
apply (*simp split*: *prod.split-asm prod.split*)
apply(*rule-tac x*=(| *nodes*=*set* [*vertex-1*,*vertex-2*], *edges* = *set* [(*vertex-1*,*vertex-2*)] |) **in** *exI*, *simp*)
apply(*rule conjI*)
apply(*simp add*: *wf-graph-def*)
apply(*rule-tac x*=(λ *x*. *0*)(*vertex-1* := *1*, *vertex-2* := *0*) **in** *exI*, *simp*)
apply(*rule-tac x*=*vertex-2* **in** *exI*, *simp*)
apply(*rule-tac x*=*set* [(*vertex-1*,*vertex-2*)] **in** *exI*, *simp*)
done

4.9.1 ENF

lemma *zero-default-candidate*: ⋀ *nP e1 e2*. ¬ ((≤)::*security-level* ⇒ *security-level* ⇒ *bool*) (*nP e1*) (*nP e2*) ⟹ ¬ (≤) (*nP e1*) *0*
by *simp-all*
lemma *zero-default-candidate-rule*: ⋀ (*nP*::('*v* ⇒ *security-level*)) *e1 e2*. ¬ (*nP e1*) ≤ (*nP e2*) ⟹ ¬ (*nP e1*) ≤ *0*
by *simp-all*
lemma *privacylevel-refl*: ((≤)::*security-level* ⇒ *security-level* ⇒ *bool*) *e e*
by(*simp-all*)
lemma *BLP-ENF*: *SecurityInvariant-withOffendingFlows.sinvar-all-edges-normal-form sinvar* (≤)
unfolding *SecurityInvariant-withOffendingFlows.sinvar-all-edges-normal-form-def*
by *simp*
lemma *BLP-ENF-refl*: *SecurityInvariant-withOffendingFlows.ENF-refl sinvar* (≤)
unfolding *SecurityInvariant-withOffendingFlows.ENF-refl-def*
apply(*rule conjI*)
apply(*simp add*: *BLP-ENF*)
apply(*simp add*: *privacylevel-refl*)
done

definition *BLP-offending-set*:: '*v graph* ⇒ ('*v* ⇒ *security-level*) ⇒ ('*v* × '*v*) *set set* **where**
BLP-offending-set G nP = (*if sinvar G nP then*
{}
else
{ {*e* ∈ *edges G*. *case e of* (*e1*,*e2*) ⇒ (*nP e1*) > (*nP e2*)} })
lemma *BLP-offending-set*: *SecurityInvariant-withOffendingFlows.set-offending-flows sinvar* = *BLP-offending-set*
apply(*simp only*: *fun-eq-iff SecurityInvariant-withOffendingFlows.ENF-offending-set*[*OF BLP-ENF*] *BLP-offending-set-def*)
apply(*rule allI*)+
apply(*rename-tac G nP*)
apply(*auto*)
done

interpretation *BLPbasic*: *SecurityInvariant-IFS sinvar default-node-properties*

rewrites *SecurityInvariant-withOffendingFlows.set-offending-flows sinvar* = *BLP-offending-set*
unfolding *receiver-violation-def*
unfolding *default-node-properties-def*
apply(*unfold-locales*)
apply(*rule ballI*)
apply(*rule SecurityInvariant-withOffendingFlows.ENF-snds-refl-instance*[*OF BLP-ENF-refl*])
apply(*simp-all add*: *BLP-ENF BLP-ENF-refl*)[*3*]
apply(*erule default-uniqueness-by-counterexample-IFS*)
apply(*fact BLP-def-unique*)
apply(*fact BLP-offending-set*)
done

lemma *TopoS-BLPBasic*: *SecurityInvariant sinvar default-node-properties receiver-violation*
unfolding *receiver-violation-def* **by** *unfold-locales*

Alternate definition of the *sinvar*: For all reachable nodes, the security level is higher

lemma *sinvar-BLPbasic-tancl*:
$wf\text{-}graph\ G \Longrightarrow sinvar\ G\ nP = (\forall\ v \in nodes\ G.\ \forall\, v' \in succ\text{-}tran\ G\ v.\ (nP\ v) \leq (nP\ v'))$
proof(*unfold sinvar.simps*, *rule iffI*, *goal-cases*)
case *1*
have $(v, v') \in (edges\ G)^+ \Longrightarrow nP\ v \leq nP\ v'$ **for** $v\ v'$
proof(*induction rule*: *trancl-induct*)
case *base* **thus** *?case* **using** *1*(*2*) **by** *fastforce*
next
case *step* **thus** *?case* **using** *1*(*2*) **by** *fastforce*
qed
thus *?case*
by(*simp add*: *succ-tran-def*)
next
case *2*
from *2*(*1*)[*simplified wf-graph-def*] **have** *f1*: $fst\ `\ edges\ G \subseteq nodes\ G$ **by** *simp*
from *f1 2*(*2*) **have** $\forall\, v \in (fst\ `\ edges\ G).\ \forall\, v' \in succ\text{-}tran\ G\ v.\ nP\ v \leq nP\ v'$ **by** *auto*
thus *?case* **unfolding** *succ-tran-def* **by** *fastforce*
qed

hide-fact (**open**) *sinvar-mono*
hide-fact *BLP-def-unique zero-default-candidate zero-default-candidate-rule privacylevel-refl BLP-ENF BLP-ENF-refl*

hide-const (**open**) *sinvar receiver-violation default-node-properties*

end
theory *SINVAR-TaintingTrusted*
imports ../*TopoS-Helper*
begin

4.10 SecurityInvariant Tainting with Untainting-Feature for IFS

context
begin
qualified datatype *taints-raw* = *TaintsUntaints-Raw* (*taints-raw*: *string set*) (*untaints-raw*: *string set*)

The *untaints-raw* set must be a subset of *taints-raw*. Otherwise, there can be entries in the untaints set, which do not affect anything. This is certainly undesirable. In addition, a unique default parameter cannot exist if we allow such dead entries.

qualified typedef *taints* = {*ts*::*taints-raw*. *untaints-raw ts* ⊆ *taints-raw ts*}
morphisms *raw-of-taints Abs-taints*
proof
show *TaintsUntaints-Raw* {} {} ∈ {*ts*. *untaints-raw ts* ⊆ *taints-raw ts*} **by** *simp*
qed

setup-lifting *type-definition-taints*

lemma *taints-eq-iff*:
tsx = *tsy* ⟷ *raw-of-taints tsx* = *raw-of-taints tsy*
by (*simp add*: *raw-of-taints-inject*)

definition *taints* :: *taints* ⇒ *string set* **where**
taints ts ≡ *taints-raw* (*raw-of-taints ts*)
definition *untaints* :: *taints* ⇒ *string set* **where**
untaints ts ≡ *untaints-raw* (*raw-of-taints ts*)

lemma *taints-wellformedness*: *untaints ts* ⊆ *taints ts*
using *raw-of-taints taints-def untaints-def* **by** *auto*

Constructor for *taints*:

definition *TaintsUntaints* :: *string set* ⇒ *string set* ⇒ *taints* **where**
TaintsUntaints ts uts = *Abs-taints* (*TaintsUntaints-Raw* (*ts* ∪ *uts*) *uts*)

lemma *raw-of-taints-TaintsUntaints*:
raw-of-taints (*TaintsUntaints ts uts*) = (*TaintsUntaints-Raw* (*ts* ∪ *uts*) *uts*)
by (*simp add*: *TaintsUntaints-def Abs-taints-inverse*)

lemma *taints-TaintsUntaints*[*code*]: *taints* (*TaintsUntaints ts uts*) = *ts* ∪ *uts*
by(*simp add*: *taints-def raw-of-taints-TaintsUntaints*)
lemma *untaints-TaintsUntaints*[*code*]: *untaints* (*TaintsUntaints ts uts*) = *uts*
by(*simp add*: *untaints-def raw-of-taints-TaintsUntaints*)

The things in the first set are tainted, those in the second set are untainted. For example, a machine produces *''foo''*: *TaintsUntaints* {*''foo''*} {}

For example, a machine consumes *''foo''* and *''bar''*, combines them in a way that they are no longer critical and outputs *''baz''*: *TaintsUntaints* {*''foo''*, *''bar''*, *''baz''*} {*''foo''*, *''bar''*} abbreviated: *TaintsUntaints* {*''baz''*} {*''foo''*, *''bar''*}

lemma *TaintsUntaints* {*''foo''*, *''bar''*, *''baz''*} {*''foo''*, *''bar''*} =
TaintsUntaints {*''baz''*} {*''foo''*, *''bar''*}
apply(*simp add*: *taints-eq-iff raw-of-taints-TaintsUntaints*)
by *blast*

qualified definition *default-node-properties* :: *taints*
where *default-node-properties* ≡ *TaintsUntaints* {} {}

qualified definition *sinvar* :: *'v graph* ⇒ (*'v* ⇒ *taints*) ⇒ *bool* **where**
sinvar G nP ≡ ∀ (*v1*,*v2*) ∈ *edges G*.

taints (*nP v1*) − *untaints* (*nP v1*) ⊆ *taints* (*nP v2*)

Information Flow Security

qualified definition *receiver-violation* :: *bool* **where** *receiver-violation* ≡ *True*

private lemma *sinvar-mono*: *SecurityInvariant-withOffendingFlows.sinvar-mono sinvar*
apply(*simp add*: *SecurityInvariant-withOffendingFlows.sinvar-mono-def sinvar-def*)
apply(*clarify*)
by *blast*
interpretation *SecurityInvariant-preliminaries*
where *sinvar* = *sinvar*
proof(*unfold-locales*, *goal-cases*)
case (*1 G nP*)
from *1* **show** *?case*
apply(*frule-tac finite-distinct-list*[*OF wf-graph.finiteE*])
apply(*erule-tac exE*)
apply(*rename-tac list-edges*)
apply(*rule-tac ff=list-edges* **in** *SecurityInvariant-withOffendingFlows.mono-imp-set-offending-flows-not-empty*[*OF sinvar-mono*])
apply(*auto simp add*: *sinvar-def*)
apply(*auto simp add*: *sinvar-def SecurityInvariant-withOffendingFlows.is-offending-flows-def graph-ops*)
done
next
case (*2 N E E′ nP*) **thus** *?case* **by**(*simp add*: *sinvar-def*) *blast*
next
case *3* **thus** *?case* **by**(*fact SecurityInvariant-withOffendingFlows.sinvar-mono-imp-is-offending-flows-mono*[*OF sinvar-mono*])
qed

Needs the well-formedness condition that *untaints otherbot* ⊆ *taints otherbot*

private lemma *Taints-def-unique*: *otherbot* ≠ *default-node-properties* ⟹
∃ *G p i f*. *wf-graph G* ∧ ¬ *sinvar G p* ∧ *f* ∈ (*SecurityInvariant-withOffendingFlows.set-offending-flows sinvar G p*) ∧
sinvar (*delete-edges G f*) *p* ∧
i ∈ *snd* ‘ *f* ∧ *sinvar G* (*p*(*i* := *otherbot*))
apply(*subgoal-tac untaints otherbot* ⊆ *taints otherbot*)
prefer *2*
subgoal using *taints-wellformedness* **by** *simp*
apply(*simp add*: *default-node-properties-def*)
apply (*simp add*: *SecurityInvariant-withOffendingFlows.set-offending-flows-def*
SecurityInvariant-withOffendingFlows.is-offending-flows-min-set-def
SecurityInvariant-withOffendingFlows.is-offending-flows-def)
apply (*simp add*:*graph-ops*)
apply (*simp split*: *prod.split-asm prod.split*)
apply(*rule-tac x*=⦇ *nodes*=*set* [*vertex-1*,*vertex-2*], *edges* = *set* [(*vertex-1*,*vertex-2*)] ⦈ **in** *exI*, *simp*)
apply(*rule conjI*)
apply(*simp add*: *wf-graph-def*; *fail*)
apply(*subgoal-tac* ∃*foo*. *foo* ∈ *taints otherbot*)
prefer *2*
subgoal
apply(*case-tac otherbot*, *rename-tac tsraw*)
apply(*simp*)

apply(*subgoal-tac taints-raw tsraw* ≠ {})
prefer *2* **subgoal for** *tsraw*
apply(*case-tac tsraw*)
apply(*simp add*: *TaintsUntaints-def*)
by *fastforce*
by (*simp add*: *Abs-taints-inverse ex-in-conv taints-def*)
apply(*elim exE*, *rename-tac foo*)
apply(*rule-tac x*=(λ *x*. *default-node-properties*)
(*vertex-1* := *TaintsUntaints* {*foo*} {}, *vertex-2* := *default-node-properties*) **in** *exI*)
apply(*simp add*: *default-node-properties-def*)
apply(*rule conjI*)
apply(*simp add*: *sinvar-def taints-TaintsUntaints untaints-TaintsUntaints*; *fail*)
apply(*rule-tac x*=*vertex-2* **in** *exI*)
apply(*rule-tac x*=*set* [(*vertex-1*,*vertex-2*)] **in** *exI*, *simp*)
apply(*simp add*: *sinvar-def taints-TaintsUntaints untaints-TaintsUntaints*; *fail*)
done

4.10.1 ENF

private lemma *Taints-ENF*: *SecurityInvariant-withOffendingFlows.sinvar-all-edges-normal-form*
sinvar (λ*c1 c2*. *taints c1* − *untaints c1* ⊆ *taints c2*)
unfolding *SecurityInvariant-withOffendingFlows.sinvar-all-edges-normal-form-def sinvar-def*
by *blast*
private lemma *Taints-ENF-refl*: *SecurityInvariant-withOffendingFlows.ENF-refl*
sinvar (λ*c1 c2*. *taints c1* − *untaints c1* ⊆ *taints c2*)
unfolding *SecurityInvariant-withOffendingFlows.ENF-refl-def*
apply(*intro conjI*)
subgoal using *Taints-ENF* **by** *simp*
by *auto*

qualified definition *Taints-offending-set*:: ′*v graph* ⇒ (′*v* ⇒ *taints*) ⇒ (′*v* × ′*v*) *set set* **where**
Taints-offending-set G nP = (*if sinvar G nP then*
{}
else
{ {*e* ∈ *edges G*. *case e of* (*e1*,*e2*) ⇒ ¬ *taints* (*nP e1*) − *untaints* (*nP e1*) ⊆ *taints* (*nP e2*)} })
lemma *Taints-offending-set*: *SecurityInvariant-withOffendingFlows.set-offending-flows sinvar* = *Taints-offending-set*
by(*auto simp add*: *fun-eq-iff*
SecurityInvariant-withOffendingFlows.ENF-offending-set[*OF Taints-ENF*]
Taints-offending-set-def)

interpretation *Taints*: *SecurityInvariant-IFS sinvar default-node-properties*
rewrites *SecurityInvariant-withOffendingFlows.set-offending-flows sinvar* = *Taints-offending-set*
unfolding *receiver-violation-def*
unfolding *default-node-properties-def*
proof(*unfold-locales*, *goal-cases*)
case (*1 G f nP*)
from *1*(*2*) **show** *?case*
apply(*intro ballI*)
apply(*rule SecurityInvariant-withOffendingFlows.ENF-snds-refl-instance*[*OF Taints-ENF-refl*])
apply(*simp add*: *sinvar-def taints-TaintsUntaints untaints-TaintsUntaints*, *blast*)
by(*simp*)+

next
case *2* **thus** *?case*
apply(*elim default-uniqueness-by-counterexample-IFS*)
apply(*rule Taints-def-unique*)
apply(*simp-all add*: *default-node-properties-def*)
done
next
case *3* **show** *set-offending-flows* = *Taints-offending-set* **by**(*fact Taints-offending-set*)
qed

lemma *TopoS-TaintingTrusted*: *SecurityInvariant sinvar default-node-properties receiver-violation*
unfolding *receiver-violation-def* **by** *unfold-locales*

end

code-datatype *TaintsUntaints*

value[*code*] *TaintsUntaints* {*''foo''*} {*''bar''*}

value[*code*] *taints* (*TaintsUntaints* {*''foo''*} {*''bar''*})

end
theory *SINVAR-BLPtrusted*
imports ../*TopoS-Helper*
begin

4.11 SecurityInvariant Basic Bell LaPadula with trusted entities

type-synonym *security-level* = *nat*

record *node-config* =
security-level::*security-level*
trusted::*bool*

definition *default-node-properties* :: *node-config*
where *default-node-properties* ≡ (| *security-level* = *0*, *trusted* = *False* |)

fun *sinvar* :: *'v graph* ⇒ (*'v* ⇒ *node-config*) ⇒ *bool* **where**
sinvar G nP = (∀ (*e1*,*e2*) ∈ *edges G*. (*if trusted* (*nP e2*) *then True else security-level* (*nP e1*) ≤ *security-level* (*nP e2*)))

A simplified version of the Bell LaPadula model was presented in `SINVAR_BLPbasic.thy`. In this theory, we extend this template with a notion of trust by adding a Boolean flag *trusted* to the host attributes. This is a refinement to represent real-world scenarios more accurately and analogously happened to the original Bell LaPadula model (see publication "Looking Back at the Bell-La Padula Model" A trusted host can receive information of any security level and may declassify it, i.e. distribute the information with its own security level. For example, a *trusted sc* = *True* host is allowed to receive any information and with the *0* level, it is allowed to reveal it to anyone.

definition *receiver-violation* :: *bool* **where** *receiver-violation* ≡ *True*

```
lemma sinvar-mono: SecurityInvariant-withOffendingFlows.sinvar-mono sinvar
  apply(simp only: SecurityInvariant-withOffendingFlows.sinvar-mono-def)
  apply(clarify)
  apply(simp split: prod.split prod.split-asm)
  by auto


interpretation SecurityInvariant-preliminaries
where sinvar = sinvar
  apply unfold-locales
    apply(frule-tac finite-distinct-list[OF wf-graph.finiteE])
    apply(erule-tac exE)
    apply(rename-tac list-edges)
   apply(rule-tac ff=list-edges in SecurityInvariant-withOffendingFlows.mono-imp-set-offending-flows-not-empty[OF
sinvar-mono])
       apply(auto split: prod.split prod.split-asm)[6]
  apply(simp add: SecurityInvariant-withOffendingFlows.is-offending-flows-def graph-ops split: prod.split
prod.split-asm)[1]
   apply (metis prod.inject)
  apply(fact SecurityInvariant-withOffendingFlows.sinvar-mono-imp-is-offending-flows-mono[OF sin-
var-mono])
 done


lemma a ≠ b ⟹ ((∃ x. y x)) ⟹ ((∀ x. ¬ y x) ⟹ a = b ) by simp

lemma BLP-def-unique: otherbot ≠ default-node-properties ⟹
   ∃ G p i f. wf-graph G ∧ ¬ sinvar G p ∧ f ∈ (SecurityInvariant-withOffendingFlows.set-offending-flows
sinvar G p) ∧
       sinvar (delete-edges G f) p ∧
         i ∈ snd ‘ f ∧ sinvar G (p(i := otherbot))
  apply(simp add:default-node-properties-def)
  apply (simp add: SecurityInvariant-withOffendingFlows.set-offending-flows-def
      SecurityInvariant-withOffendingFlows.is-offending-flows-min-set-def
      SecurityInvariant-withOffendingFlows.is-offending-flows-def)
  apply (simp add:graph-ops)
  apply (simp split: prod.split-asm prod.split)
  apply(rule-tac x=(| nodes={vertex-1, vertex-2}, edges = {(vertex-1,vertex-2)} |) in exI, simp)
  apply(rule conjI)
   apply(simp add: wf-graph-def)
  apply(rule-tac x=(λ x. default-node-properties)(vertex-1 := (|security-level = 1, trusted = False |),
vertex-2 := (|security-level = 0, trusted = False |)) in exI, simp add:default-node-properties-def)
  apply(rule-tac x=vertex-2 in exI, simp)
  apply(rule-tac x={(vertex-1,vertex-2)} in exI, simp)
  apply(case-tac otherbot)
  apply simp
  apply(erule disjE)
   apply force
  apply fast
  done
```

4.11.1 ENF

definition *BLP-P* **where** *BLP-P* ≡ (*λn1 n2*.(*if trusted n2 then True else security-level n1* ≤ *security-level n2*))
lemma *zero-default-candidate*: ∀ *nP e1 e2*. ¬ *BLP-P* (*nP e1*) (*nP e2*) ⟶ ¬ *BLP-P* (*nP e1*) *default-node-properties*
apply(*rule allI*)+
apply(*case-tac nP e1*)
apply(*case-tac nP e2*)
apply(*rename-tac privacy2 trusted2 more2*)
apply (*simp add*: *BLP-P-def default-node-properties-def*)
done
lemma *privacylevel-refl*: *BLP-P e e*
by(*simp-all add*: *BLP-P-def*)
lemma *BLP-ENF*: *SecurityInvariant-withOffendingFlows.sinvar-all-edges-normal-form sinvar BLP-P*
unfolding *SecurityInvariant-withOffendingFlows.sinvar-all-edges-normal-form-def*
unfolding *BLP-P-def*
by *simp*
lemma *BLP-ENF-refl*: *SecurityInvariant-withOffendingFlows.ENF-refl sinvar BLP-P*
unfolding *SecurityInvariant-withOffendingFlows.ENF-refl-def*
apply(*rule conjI*)
apply(*simp add*: *BLP-ENF*)
apply(*simp add*: *privacylevel-refl*)
done

definition *BLP-offending-set*:: *'v graph* ⇒ (*'v* ⇒ *node-config*) ⇒ (*'v* × *'v*) *set set* **where**
BLP-offending-set G nP = (*if sinvar G nP then*
{}
else
{ {*e* ∈ *edges G*. *case e of* (*e1*,*e2*) ⇒ ¬ *BLP-P* (*nP e1*) (*nP e2*)} })
lemma *BLP-offending-set*: *SecurityInvariant-withOffendingFlows.set-offending-flows sinvar* = *BLP-offending-set*
apply(*simp only*: *fun-eq-iff SecurityInvariant-withOffendingFlows.ENF-offending-set*[*OF BLP-ENF*] *BLP-offending-set-def*)
apply(*rule allI*)+
apply(*rename-tac G nP*)
apply(*auto*)
done

interpretation *BLPtrusted*: *SecurityInvariant-IFS*
where *default-node-properties* = *default-node-properties*
and *sinvar* = *sinvar*
rewrites *SecurityInvariant-withOffendingFlows.set-offending-flows sinvar* = *BLP-offending-set*
apply *unfold-locales*
apply(*rule ballI*)
apply (*rule-tac f=f* **in** *SecurityInvariant-withOffendingFlows.ENF-snds-refl-instance*[*OF BLP-ENF-refl zero-default-candidate*])
apply(*simp*)
apply(*simp*)
apply(*erule default-uniqueness-by-counterexample-IFS*)
apply(*fact BLP-def-unique*)
apply(*fact BLP-offending-set*)
done

lemma *TopoS-BLPtrusted*: *SecurityInvariant sinvar default-node-properties receiver-violation*
unfolding *receiver-violation-def* **by** *unfold-locales*

hide-type (**open**) *node-config*
hide-const (**open**) *sinvar-mono*

hide-const (**open**) *BLP-P*
hide-fact *BLP-def-unique zero-default-candidate privacylevel-refl BLP-ENF BLP-ENF-refl*

hide-const (**open**) *sinvar receiver-violation default-node-properties*

end
theory *Analysis-Tainting*
imports *SINVAR-Tainting SINVAR-BLPbasic*
SINVAR-TaintingTrusted SINVAR-BLPtrusted
begin

term *SINVAR-Tainting.sinvar*
term *SINVAR-BLPbasic.sinvar*

lemma *tainting-imp-blp-cutcard*: $\forall\, ts\ v.\ nP\ v = ts \longrightarrow finite\ ts \Longrightarrow$
SINVAR-Tainting.sinvar $G\ nP \Longrightarrow$ *SINVAR-BLPbasic.sinvar* $G\ ((\lambda ts.\ card\ (ts \cap X)) \circ nP)$
apply(*simp add*: *SINVAR-Tainting.sinvar-def*)
apply(*clarify*, *rename-tac a b*)
apply(*erule-tac x*=(*a*,*b*) **in** *ballE*)
apply(*simp-all*)
apply(*subgoal-tac finite* ($nP\ a \cap X$))
prefer *2* **subgoal using** *finite-Int* **by** *blast*
apply(*subgoal-tac finite* ($nP\ b \cap X$))
prefer *2* **subgoal using** *finite-Int* **by** *blast*
using *card-mono* **by** (*metis Int-subset-iff order-refl subset-antisym*)

lemma *tainting-imp-blp-cutcard2*: *finite* $X \Longrightarrow$
SINVAR-Tainting.sinvar $G\ nP \Longrightarrow$ *SINVAR-BLPbasic.sinvar* $G\ ((\lambda ts.\ card\ (ts \cap X)) \circ nP)$
apply(*simp add*: *SINVAR-Tainting.sinvar-def*)
apply(*clarify*, *rename-tac a b*)
apply(*erule-tac x*=(*a*,*b*) **in** *ballE*)
apply(*simp-all*)
apply(*subgoal-tac finite* ($nP\ a \cap X$))
prefer *2* **subgoal using** *finite-Int* **by** *blast*
apply(*subgoal-tac finite* ($nP\ b \cap X$))
prefer *2* **subgoal using** *finite-Int* **by** *blast*
using *card-mono* **by** (*metis Int-subset-iff order-refl subset-antisym*)

lemma $\forall\, ts\ v.\ nP\ v = ts \longrightarrow finite\ ts \Longrightarrow$
SINVAR-Tainting.sinvar $G\ nP \Longrightarrow$ *SINVAR-BLPbasic.sinvar* $G\ (card \circ nP)$
apply(*drule*(*1*) *tainting-imp-blp-cutcard*[**where** *X*=*UNIV*])
by(*simp*)

```
lemma ∀ b ∈ snd ‘ edges G. finite (nP b) ⟹
  SINVAR-Tainting.sinvar G nP ⟹ SINVAR-BLPbasic.sinvar G (card ∘ nP)
apply(simp add: SINVAR-Tainting.sinvar-def)
apply(clarify, rename-tac a b)
apply(erule-tac x=(a,b) in ballE)
 apply(simp-all)
apply(case-tac finite (nP a))
 apply(case-tac [!] finite (nP b))
   using card-mono apply blast
  apply(simp-all)
done
```

One tainting invariant is equal to many BLP invariants. The BLP invariants are the projection of the tainting mapping for exactly one label

```
lemma tainting-iff-blp:
  defines extract ≡ λa ts. if a ∈ ts then 1::security-level else 0::security-level
  shows SINVAR-Tainting.sinvar G nP ⟷ (∀ a. SINVAR-BLPbasic.sinvar G (extract a ∘ nP))
proof
  showSINVAR-Tainting.sinvar G nP ⟹ ∀ a. SINVAR-BLPbasic.sinvar G (extract a ∘ nP)
    apply(simp add: extract-def)
    apply(safe)
     apply simp
    apply(simp add: SINVAR-Tainting.sinvar-def)
    by fast
  next
    assume blp: ∀ a. SINVAR-BLPbasic.sinvar G (extract a ∘ nP)
    { fix v1 v2
      assume ∗: (v1,v2)∈edges G
      { fix a
        from blp ∗ have (if a ∈ nP v1 then 1::security-level else 0) ≤ (if a ∈ nP v2 then 1 else 0)
          unfolding extract-def
          apply(simp)
          apply(erule-tac x=a in allE)
          apply(erule-tac x=(v1, v2) in ballE)
           apply(simp-all)
          apply(simp split: if-split-asm)
          done
        hence a ∈ nP v1 ⟹ a ∈ nP v2 by(simp split: if-split-asm)
      }
      from this have nP v1 ⊆ nP v2 by auto
    }
    thus SINVAR-Tainting.sinvar G nP unfolding SINVAR-Tainting.sinvar-def by blast
qed
```

If the labels are finite, the above can be generalized to arbitrary subsets of tainting labels.

```
lemma tainting-iff-blp-extended:
  defines extract ≡ λA ts. card (A ∩ ts)
  assumes finite: ∀ ts v. nP v = ts ⟶ finite ts
  shows SINVAR-Tainting.sinvar G nP ⟷ (∀ A. SINVAR-BLPbasic.sinvar G (extract A ∘ nP))
proof
  show SINVAR-Tainting.sinvar G nP ⟹ ∀ A. SINVAR-BLPbasic.sinvar G (extract A ∘ nP)
    apply(simp add: extract-def)
    apply(safe)
```

apply(*simp add*: *SINVAR-Tainting.sinvar-def*)
apply(*rename-tac A a b*)
apply(*subgoal-tac finite* (*A* ∩ *nP a*))
prefer *2* **subgoal using** *finite* **by** *blast*
apply(*rule card-mono*)
apply(*simp add*: *finite*; *fail*)
by *blast*
next
assume *blp*: ∀ *A*. *SINVAR-BLPbasic.sinvar G* (*extract A* ∘ *nP*)
{ fix *v1 v2*
assume ∗: (*v1*,*v2*)∈*edges G*
{ fix *A*
from *blp* ∗ **have** *card* (*A* ∩ *nP v1*) ≤ *card* (*A* ∩ *nP v2*)
unfolding *extract-def*
apply(*clarsimp*)
apply(*erule-tac x*=*A* **in** *allE*)
apply(*erule-tac x*=(*v1*, *v2*) **in** *ballE*)
by(*simp-all*)
}
from *this finite card-seteq* **have** *nP v1* ⊆ *nP v2* **by** (*metis Int-absorb Int-lower1 inf.orderI*)
}
thus *SINVAR-Tainting.sinvar G nP* **unfolding** *SINVAR-Tainting.sinvar-def* **by** *blast*
qed

Translated to the Bell LaPadula model with trust: security level is the number of tainted minus the untainted things We set the Trusted flag if a machine untaints things.

lemma ∀ *ts v*. *nP v* = *ts* ⟶ *finite* (*taints ts*) ⟹
SINVAR-TaintingTrusted.sinvar G nP ⟹
SINVAR-BLPtrusted.sinvar G ((λ *ts*. ⦇*security-level* = *card* (*taints ts* − *untaints ts*), *trusted* = (*untaints ts* ≠ {})⦈) ∘ *nP*)
apply(*simp add*: *SINVAR-TaintingTrusted.sinvar-def*)
apply(*clarify*, *rename-tac a b*)
apply(*erule-tac x*=(*a*,*b*) **in** *ballE*)
apply(*simp-all*)
apply(*subgoal-tac finite* (*taints* (*nP a*) − *untaints* (*nP a*)))
prefer *2* **subgoal by** *blast*
apply(*rule card-mono*)
by *blast*+

lemma *tainting-iff-blp-trusted*:
defines *project* ≡ λ*a ts*. ⦇
security-level =
if
a ∈ (*taints ts* − *untaints ts*)
then
1::*security-level*
else
0::*security-level*
, *trusted* = *a* ∈ *untaints ts*⦈
shows *SINVAR-TaintingTrusted.sinvar G nP* ⟷ (∀ *a*. *SINVAR-BLPtrusted.sinvar G* (*project a* ∘ *nP*))
unfolding *project-def*
apply(*rule iffI*)
subgoal

apply(*simp add*: *SINVAR-TaintingTrusted.sinvar-def*)
apply(*clarify*, *rename-tac a b*)
apply(*erule-tac x*=(*a*,*b*) **in** *ballE*)
apply(*simp-all*)
by *blast*
apply(*simp*)
apply(*simp add*: *SINVAR-TaintingTrusted.sinvar-def*)
apply(*clarify*, *rename-tac a b taintlabel*)
apply(*erule-tac x*=*taintlabel* **in** *allE*)
apply(*erule-tac x*=(*a*,*b*) **in** *ballE*)
apply(*simp-all*)
apply(*simp split*: *if-split-asm*)
using *taints-wellformedness* **by** *blast*

If the labels are finite, the above can be generalized to arbitrary subsets of tainting labels.

lemma *tainting-iff-blp-trusted-extended*:
defines *project* ≡ λ*A ts*.
⦇
security-level = *card* (*A* ∩ (*taints ts* − *untaints ts*))
, *trusted* = (*A* ∩ *untaints ts*) ≠ {}
⦈
assumes *finite*: ∀ *ts v*. *nP v* = *ts* ⟶ *finite* (*taints ts*)
shows *SINVAR-TaintingTrusted.sinvar G nP* ⟷ (∀ *A*. *SINVAR-BLPtrusted.sinvar G* (*project A* ∘ *nP*))
unfolding *project-def*
apply(*rule iffI*)
subgoal
apply(*simp add*: *SINVAR-TaintingTrusted.sinvar-def*)
apply(*clarify*, *rename-tac a b*)
apply(*erule-tac x*=(*a*,*b*) **in** *ballE*)
apply(*simp-all*)
apply(*rule card-mono*)
using *finite* **apply** *blast*
by *blast*
apply(*simp*)
apply(*simp add*: *SINVAR-TaintingTrusted.sinvar-def*)
apply(*clarify*, *rename-tac a b taintlabel*)
apply(*erule-tac x*={*taintlabel*} **in** *allE*)
apply(*erule-tac x*=(*a*,*b*) **in** *ballE*)
apply(*simp-all*)
apply(*simp split*: *if-split-asm*)
using *taints-wellformedness* **apply** *blast*
using *Diff-insert-absorb* **by** *fastforce*

end
theory *TopoS-Interface-impl*
imports *Lib/FiniteGraph Lib/FiniteListGraph TopoS-Interface TopoS-Helper*
begin

5 Executable Implementation with Lists

Correspondence List Implementation and set Specification

5.1 Abstraction from list implementation to set specification

Nomenclature: *-spec* is the specification, *-impl* the corresponding implementation.

-spec and *-impl* only need to comply for *wf-graph*s. We will always require the stricter *wf-list-graph*, which implies *wf-graph*.

lemma *wf-list-graph G* ⟹ *wf-graph* (*list-graph-to-graph G*)

locale *TopoS-List-Impl* =
fixes *default-node-properties* :: *'a* (‹⊥›)
and *sinvar-spec*::(*'v*::*vertex*) *graph* ⇒ (*'v*::*vertex* ⇒ *'a*) ⇒ *bool*
and *sinvar-impl*::(*'v*::*vertex*) *list-graph* ⇒ (*'v*::*vertex* ⇒ *'a*) ⇒ *bool*
and *receiver-violation* :: *bool*
and *offending-flows-impl*::(*'v*::*vertex*) *list-graph* ⇒ (*'v* ⇒ *'a*) ⇒ (*'v* × *'v*) *list list*
and *node-props-impl*::(*'v*::*vertex*, *'a*) *TopoS-Params* ⇒ (*'v* ⇒ *'a*)
and *eval-impl*::(*'v*::*vertex*) *list-graph* ⇒ (*'v*, *'a*) *TopoS-Params* ⇒ *bool*
assumes
spec: *SecurityInvariant sinvar-spec default-node-properties receiver-violation* — specification is valid
and
sinvar-spec-impl: *wf-list-graph G* ⟹
(*sinvar-spec* (*list-graph-to-graph G*) *nP*) = (*sinvar-impl G nP*)
and
offending-flows-spec-impl: *wf-list-graph G* ⟹
(*SecurityInvariant-withOffendingFlows.set-offending-flows sinvar-spec* (*list-graph-to-graph G*) *nP*) =
set'set (*offending-flows-impl G nP*)
and
node-props-spec-impl:
SecurityInvariant.node-props-formaldef default-node-properties P = *node-props-impl P*
and
eval-spec-impl:
(*distinct* (*nodesL G*) ∧ *distinct* (*edgesL G*) ∧
SecurityInvariant.eval sinvar-spec default-node-properties (*list-graph-to-graph G*) *P*) =
(*eval-impl G P*)

5.2 Security Invariants Packed

We pack all necessary functions and properties of a security invariant in a struct-like data structure.

record (*'v*::*vertex*, *'a*) *TopoS-packed* =
nm-name :: *string*
nm-receiver-violation :: *bool*
nm-default :: *'a*
nm-sinvar::(*'v*::*vertex*) *list-graph* ⇒ (*'v* ⇒ *'a*) ⇒ *bool*
nm-offending-flows::(*'v*::*vertex*) *list-graph* ⇒ (*'v* ⇒ *'a*) ⇒ (*'v* × *'v*) *list list*
nm-node-props::(*'v*::*vertex*, *'a*) *TopoS-Params* ⇒ (*'v* ⇒ *'a*)
nm-eval::(*'v*::*vertex*) *list-graph* ⇒ (*'v*, *'a*) *TopoS-Params* ⇒ *bool*

The packed list implementation must comply with the formal definition.

locale *TopoS-modelLibrary* =
fixes *m* :: (*'v*::*vertex*, *'a*) *TopoS-packed* — concrete model implementation
and *sinvar-spec*::(*'v*::*vertex*) *graph* ⇒ (*'v*::*vertex* ⇒ *'a*) ⇒ *bool* — specification

assumes
name-not-empty: *length* (*nm-name m*) > *0*
and
impl-spec: *TopoS-List-Impl*
(*nm-default m*)
sinvar-spec
(*nm-sinvar m*)
(*nm-receiver-violation m*)
(*nm-offending-flows m*)
(*nm-node-props m*)
(*nm-eval m*)

5.3 Helpful Lemmata

show that *sinvar* complies

lemma *TopoS-eval-impl-proofrule*:
assumes *inst*: *SecurityInvariant sinvar-spec default-node-properties receiver-violation*
assumes *ev*: ⋀*nP*. *wf-list-graph G* ⟹ *sinvar-spec* (*list-graph-to-graph G*) *nP* = *sinvar-impl G nP*
shows
(*distinct* (*nodesL G*) ∧ *distinct* (*edgesL G*) ∧
SecurityInvariant.eval sinvar-spec default-node-properties (*list-graph-to-graph G*) *P*) =
(*wf-list-graph G* ∧ *sinvar-impl G* (*SecurityInvariant.node-props default-node-properties P*))
proof (*cases wf-list-graph G*)
case *True*
hence *sinvar-spec* (*list-graph-to-graph G*) (*SecurityInvariant.node-props default-node-properties P*)
=
sinvar-impl G (*SecurityInvariant.node-props default-node-properties P*)
using *ev* **by** *blast*

with *inst* **show** *?thesis*
unfolding *wf-list-graph-def*
by (*simp add*: *wf-list-graph-iff-wf-graph SecurityInvariant.eval-def*)
next
case *False*
hence (*distinct* (*nodesL G*) ∧ *distinct* (*edgesL G*) ∧ *wf-list-graph-axioms G*) = *False*
unfolding *wf-list-graph-def* **by** *blast*
with *False* **show** *?thesis*
unfolding *SecurityInvariant.eval-def*[*OF inst*]
by (*fastforce simp*: *wf-list-graph-iff-wf-graph*)
qed

5.4 Helper lemmata

Provide *sinvar* function and get back a function that computes the list of offending flows
Exponential time!

definition *Generic-offending-list*:: (*'v list-graph* ⇒ (*'v* ⇒ *'a*) ⇒ *bool*)⇒ *'v list-graph* ⇒ (*'v* ⇒ *'a*)
⇒ (*'v* × *'v*) *list list* **where**
Generic-offending-list sinvar G nP = [*f* ← (*subseqs* (*edgesL G*)).
(¬ *sinvar G nP* ∧ *sinvar* (*FiniteListGraph.delete-edges G f*) *nP*) ∧
(∀ (*e1*, *e2*)∈*set f*. ¬ *sinvar* (*add-edge e1 e2* (*FiniteListGraph.delete-edges G f*)) *nP*)]

proof rule: if *sinvar* complies, *Generic-offending-list* complies

lemma *Generic-offending-list-correct*:
assumes *valid*: *wf-list-graph G*
assumes *spec-impl*: $\bigwedge$*G nP*. *wf-list-graph G* $\Longrightarrow$ *sinvar-spec* (*list-graph-to-graph G*) *nP* = *sinvar-impl G nP*
shows *SecurityInvariant-withOffendingFlows.set-offending-flows sinvar-spec* (*list-graph-to-graph G*) *nP* =
set'set(*Generic-offending-list sinvar-impl G nP*)
proof −
have $\bigwedge$ *P G*. *set* ' {*x* ∈ *set* (*subseqs* (*edgesL G*)). *P G* (*set x*)} = {*x* ∈ *set* ' *set* (*subseqs* (*edgesL G*)). *P G* (*x*)}
by *fastforce*
hence *subset-subseqs-filter*: $\bigwedge$ *G P*. {*f*. *f* ⊆ *edges* (*list-graph-to-graph G*) ∧ *P G f*}
= *set* ' *set* [*f*←*subseqs* (*edgesL G*) . *P G* (*set f*)]
unfolding *list-graph-to-graph-def*
by (*auto simp*: *subseqs-powset*)

from *valid delete-edges-wf* **have** ∀*f*. *wf-list-graph*(*FiniteListGraph.delete-edges G f*) **by** *fast*
with *spec-impl*[*symmetric*] *FiniteListGraph.delete-edges-correct*[*of G*] **have** *impl-spec-delete*:
∀*f*. *sinvar-impl* (*FiniteListGraph.delete-edges G f*) *nP* =
sinvar-spec (*FiniteGraph.delete-edges* (*list-graph-to-graph G*) (*set f*)) *nP* **by** *simp*

from *spec-impl*[*OF valid*, *symmetric*] **have** *impl-spec-not*:
(¬ *sinvar-impl G nP*) = (¬ *sinvar-spec* (*list-graph-to-graph G*) *nP*) **by** *auto*

from *spec-impl*[*symmetric*, *OF FiniteListGraph.add-edge-wf*[*OF FiniteListGraph.delete-edges-wf*[*OF valid*]]] **have** *impl-spec-allE*:
∀ *e1 e2 E*. *sinvar-impl* (*FiniteListGraph.add-edge e1 e2* (*FiniteListGraph.delete-edges G E*)) *nP* =
sinvar-spec (*list-graph-to-graph* (*FiniteListGraph.add-edge e1 e2* (*FiniteListGraph.delete-edges G E*))) *nP* **by** *simp*

have *list-graph*: $\bigwedge$ *e1 e2 G f*. (*list-graph-to-graph* (*FiniteListGraph.add-edge e1 e2* (*FiniteListGraph.delete-edges G f*))) =
(*FiniteGraph.add-edge e1 e2* (*FiniteGraph.delete-edges* (*list-graph-to-graph G*) (*set f*)))
by(*simp add*: *FiniteListGraph.add-edge-correct FiniteListGraph.delete-edges-correct*)

show *?thesis*
unfolding *SecurityInvariant-withOffendingFlows.set-offending-flows-def*
SecurityInvariant-withOffendingFlows.is-offending-flows-min-set-def
SecurityInvariant-withOffendingFlows.is-offending-flows-def
Generic-offending-list-def
apply(*subst impl-spec-delete*)
apply(*subst impl-spec-not*)
apply(*subst impl-spec-allE*)
apply(*subst list-graph*)
apply(*rule subset-subseqs-filter*)
done
qed

lemma *all-edges-list-I*: *P* (*list-graph-to-graph G*) = *Pl G* $\Longrightarrow$
(∀(*e1*, *e2*)∈ (*edges* (*list-graph-to-graph G*)). *P* (*list-graph-to-graph G*) *e1 e2*) = (∀(*e1*, *e2*)∈*set* (*edgesL G*). *Pl G e1 e2*)
unfolding *list-graph-to-graph-def*
by *simp*

lemma *all-nodes-list-I*: *P* (*list-graph-to-graph G*) = *Pl G* ⟹
(∀ *n* ∈ (*nodes* (*list-graph-to-graph G*)). *P* (*list-graph-to-graph G*) *n*) = (∀ *n* ∈*set* (*nodesL G*). *Pl G n*)
unfolding *list-graph-to-graph-def*
by *simp*

fun *minimalize-offending-overapprox* :: ('*v list-graph* ⇒ *bool*) ⇒
('*v* × '*v*) *list* ⇒ ('*v* × '*v*) *list* ⇒ '*v list-graph* ⇒ ('*v* × '*v*) *list* **where**
minimalize-offending-overapprox - [] *keep* - = *keep* |
minimalize-offending-overapprox m (*f*#*fs*) *keep G* = (*if m* (*delete-edges G* (*fs*@*keep*)) *then*
minimalize-offending-overapprox m fs keep G
else
minimalize-offending-overapprox m fs (*f*#*keep*) *G*
)

thm *minimalize-offending-overapprox-boundnP*
lemma *minimalize-offending-overapprox-spec-impl*:
assumes *valid*: *wf-list-graph* (*G*::'*v*::*vertex list-graph*)
and *spec-impl*: ⋀*G nP*::('*v* ⇒ '*a*). *wf-list-graph G* ⟹ *sinvar-spec* (*list-graph-to-graph G*) *nP* = *sinvar-impl G nP*
shows *minimalize-offending-overapprox* (λ*G*. *sinvar-impl G nP*) *fs keeps G* =
TopoS-withOffendingFlows.minimalize-offending-overapprox (λ*G*. *sinvar-spec G nP*) *fs keeps* (*list-graph-to-graph G*)
apply(*subst minimalize-offending-overapprox-boundnP*)
using *valid spec-impl* **apply**(*induction fs arbitrary*: *keeps*)
apply(*simp add*: *SecurityInvariant-withOffendingFlows.minimalize-offending-overapprox.simps*; *fail*)
apply(*simp add*: *SecurityInvariant-withOffendingFlows.minimalize-offending-overapprox.simps*)
apply (*metis FiniteListGraph.delete-edges-wf delete-edges-list-set list-graph-correct*(*5*))
done

With *TopoS-Interface-impl.minimalize-offending-overapprox*, we can get one offending flow

lemma *minimalize-offending-overapprox-gives-some-offending-flow*:
assumes *wf*: *wf-list-graph G*
and *NetModelLib*: *TopoS-modelLibrary m sinvar-spec*
and *violation*: ¬ (*nm-sinvar m*) *G nP*
shows *set* (*minimalize-offending-overapprox* (λ*G*. (*nm-sinvar m*) *G nP*) (*edgesL G*) [] *G*) ∈
SecurityInvariant-withOffendingFlows.set-offending-flows sinvar-spec (*list-graph-to-graph G*) *nP*
proof −
from *wf* **have** *wfG*: *wf-graph* (*list-graph-to-graph G*)
by (*simp add*: *wf-list-graph-def wf-list-graph-iff-wf-graph*)
from *wf* **have** *dist-edges*: *distinct* (*edgesL G*) **by** (*simp add*: *wf-list-graph-def*)

let *?spec-algo*=*TopoS-withOffendingFlows.minimalize-offending-overapprox*
(λ*G*. *sinvar-spec G nP*) (*edgesL G*) [] (*list-graph-to-graph G*)

note *spec*=*TopoS-List-Impl.spec*[*OF TopoS-modelLibrary.impl-spec*[*OF NetModelLib*]]

from *spec* **have** *spec-prelim*: *SecurityInvariant-preliminaries sinvar-spec*
by(*simp add*: *SecurityInvariant-def*)
from *spec-prelim SecurityInvariant-preliminaries.sinvar-monoI* **have** *mono*:
SecurityInvariant-withOffendingFlows.sinvar-mono sinvar-spec **by** *blast*

from *spec-prelim* **have** *empty-edges*: *sinvar-spec* (|*nodes* = *set* (*nodesL G*), *edges* = {}|) *nP*
using *SecurityInvariant-preliminaries.defined-offending*
SecurityInvariant-withOffendingFlows.sinvar-mono-imp-sinvar-mono
SecurityInvariant-withOffendingFlows.valid-empty-edges-iff-exists-offending-flows
mono empty-subsetI graph.simps(*1*)
list-graph-to-graph-def local.wf wf-list-graph-def wf-list-graph-iff-wf-graph
by (*metis*)

have *spec-impl*: *wf-list-graph G* ⟹ *sinvar-spec* (*list-graph-to-graph G*) *nP* = (*nm-sinvar m*) *G nP* **for** *G nP*
using *NetModelLib TopoS-List-Impl.sinvar-spec-impl TopoS-modelLibrary.impl-spec* **by** *fastforce*

from *minimalize-offending-overapprox-spec-impl*[*OF wf*] *spec-impl* **have** *alog-spec*:
minimalize-offending-overapprox (λ*G*. (*nm-sinvar m*) *G nP*) *fs keeps G* =
TopoS-withOffendingFlows.minimalize-offending-overapprox (λ*G*. *sinvar-spec G nP*) *fs keeps* (*list-graph-to-graph G*)
for *fs keeps* **by** *blast*

from *spec-impl violation* **have**
SecurityInvariant-withOffendingFlows.is-offending-flows sinvar-spec (*set* (*edgesL G*)) (*list-graph-to-graph G*) *nP*
apply(*simp add*: *SecurityInvariant-withOffendingFlows.is-offending-flows-def*)
apply(*intro conjI*)
apply (*simp add*: *local.wf*; *fail*)
apply(*simp add*: *FiniteGraph.delete-edges-simp2 list-graph-to-graph-def*)
apply(*simp add*: *empty-edges*)
done
hence *goal*: *SecurityInvariant-withOffendingFlows.is-offending-flows-min-set sinvar-spec*
(*set ?spec-algo*) (*list-graph-to-graph G*) *nP*
apply(*subst minimalize-offending-overapprox-boundnP*)
apply(*rule SecurityInvariant-withOffendingFlows.is-offending-flows-min-set-minimalize-offending-overapprox*[*OF mono wfG - - dist-edges*])
apply(*simp add*: *list-graph-to-graph-def*)+
done

from *SecurityInvariant-withOffendingFlows.minimalize-offending-overapprox-subseteq-input*[*of sinvar-spec* (*edgesL G*) []] **have** *subset-edges*:
set ?spec-algo ⊆ *edges* (*list-graph-to-graph G*)
apply(*subst minimalize-offending-overapprox-boundnP*)
by(*simp add*: *list-graph-to-graph-def*)

from *goal* **show** *?thesis*
by(*simp add*: *SecurityInvariant-withOffendingFlows.set-offending-flows-def alog-spec subset-edges*)
qed

6 Security Invariant Library

end
theory *SINVAR-BLPbasic-impl*
imports *SINVAR-BLPbasic ../TopoS-Interface-impl*
begin

6.0.1 SecurityInvariant BLPbasic List Implementation

code-identifier code-module *SINVAR-BLPbasic-impl* => (*Scala*) *SINVAR-BLPbasic*

fun *sinvar* :: *'v list-graph* ⇒ (*'v* ⇒ *security-level*) ⇒ *bool* **where**
sinvar G nP = (∀ (*e1*,*e2*) ∈ *set* (*edgesL G*). (*nP e1*) ≤ (*nP e2*))

definition *BLP-offending-list*:: *'v list-graph* ⇒ (*'v* ⇒ *security-level*) ⇒ (*'v* × *'v*) *list list* **where**
BLP-offending-list G nP = (*if sinvar G nP then*
[]
else
[[*e* ← *edgesL G*. *case e of* (*e1*,*e2*) ⇒ (*nP e1*) > (*nP e2*)]])

definition *NetModel-node-props P* = (λ *i*. (*case* (*node-properties P*) *i of Some property* ⇒ *property* | *None* ⇒ *SINVAR-BLPbasic.default-node-properties*))
lemma[*code-unfold*]: *SecurityInvariant.node-props SINVAR-BLPbasic.default-node-properties P* = *NetModel-node-props P*
apply(*simp add*: *NetModel-node-props-def*)
done

definition *BLP-eval G P* = (*wf-list-graph G* ∧
sinvar G (*SecurityInvariant.node-props SINVAR-BLPbasic.default-node-properties P*))

interpretation *BLPbasic-impl*:*TopoS-List-Impl*
where *default-node-properties*=*SINVAR-BLPbasic.default-node-properties*
and *sinvar-spec*=*SINVAR-BLPbasic.sinvar*
and *sinvar-impl*=*sinvar*
and *receiver-violation*=*SINVAR-BLPbasic.receiver-violation*
and *offending-flows-impl*=*BLP-offending-list*
and *node-props-impl*=*NetModel-node-props*
and *eval-impl*=*BLP-eval*
apply(*unfold TopoS-List-Impl-def*)
apply(*rule conjI*)
apply(*simp add*: *TopoS-BLPBasic*)
apply(*simp add*: *list-graph-to-graph-def*; *fail*)
apply(*rule conjI*)
apply(*simp add*: *list-graph-to-graph-def*)
apply(*simp add*: *list-graph-to-graph-def BLP-offending-set BLP-offending-set-def BLP-offending-list-def*; *fail*)
apply(*rule conjI*)
apply(*simp only*: *NetModel-node-props-def*)
apply(*metis BLPbasic.node-props.simps BLPbasic.node-props-eq-node-props-formaldef*)
apply(*simp only*: *BLP-eval-def*)
apply(*simp add*: *TopoS-eval-impl-proofrule*[*OF TopoS-BLPBasic*])

apply(*simp add*: *list-graph-to-graph-def*)
done

6.0.2 BLPbasic packing

definition *SINVAR-LIB-BLPbasic* :: (*'v*::*vertex*, *security-level*) *TopoS-packed* **where**
SINVAR-LIB-BLPbasic ≡
⦇ *nm-name* = *''BLPbasic''*,
nm-receiver-violation = *SINVAR-BLPbasic.receiver-violation*,
nm-default = *SINVAR-BLPbasic.default-node-properties*,
nm-sinvar = *sinvar*,
nm-offending-flows = *BLP-offending-list*,
nm-node-props = *NetModel-node-props*,
nm-eval = *BLP-eval*
⦈
interpretation *SINVAR-LIB-BLPbasic-interpretation*: *TopoS-modelLibrary SINVAR-LIB-BLPbasic*

SINVAR-BLPbasic.sinvar
apply(*unfold TopoS-modelLibrary-def SINVAR-LIB-BLPbasic-def*)
apply(*rule conjI*)
apply(*simp*)
apply(*simp*)
by(*unfold-locales*)

6.0.3 Example

definition *fabNet* :: *string list-graph* **where**
fabNet ≡ ⦇ *nodesL* = [*''Statistics''*, *''SensorSink''*, *''PresenceSensor''*, *''Webcam''*, *''TempSensor''*, *''FireSesnsor''*,
''MissionControl1'', *''MissionControl2''*, *''Watchdog''*, *''Bot1''*, *''Bot2''*],
edgesL =[(*''PresenceSensor''*, *''SensorSink''*), (*''Webcam''*, *''SensorSink''*),
(*''TempSensor''*, *''SensorSink''*), (*''FireSesnsor''*, *''SensorSink''*),
(*''SensorSink''*, *''Statistics''*),
(*''MissionControl1''*, *''Bot1''*), (*''MissionControl1''*, *''Bot2''*),
(*''MissionControl2''*, *''Bot2''*),
(*''Watchdog''*, *''Bot1''*), (*''Watchdog''*, *''Bot2''*)] ⦈
value *wf-list-graph fabNet*

definition *sensorProps-try1* :: *string* ⇒ *security-level* **where**
sensorProps-try1 ≡ (λ *n*. *SINVAR-BLPbasic.default-node-properties*)(*''PresenceSensor''* := *2*, *''Webcam''* := *3*)
value *BLP-offending-list fabNet sensorProps-try1*
value *sinvar fabNet sensorProps-try1*

definition *sensorProps-try2* :: *string* ⇒ *security-level* **where**
sensorProps-try2 ≡ (λ *n*. *SINVAR-BLPbasic.default-node-properties*)(*''PresenceSensor''* := *2*, *''Webcam''* := *3*,
''SensorSink'' := *3*)
value *BLP-offending-list fabNet sensorProps-try2*
value *sinvar fabNet sensorProps-try2*

definition *sensorProps-try3* :: *string* ⇒ *security-level* **where**
sensorProps-try3 ≡ (λ *n*. *SINVAR-BLPbasic.default-node-properties*)(*''PresenceSensor''* := *2*,

''Webcam'' := 3,
''SensorSink'' := 3, *''Statistics'' := 3*)
value *BLP-offending-list fabNet sensorProps-try3*
value *sinvar fabNet sensorProps-try3*

Another parameter set for confidential controlling information

definition *sensorProps-conf* :: *string* ⇒ *security-level* **where**
sensorProps-conf ≡ (λ *n*. *SINVAR-BLPbasic.default-node-properties*)(*''MissionControl1'' := 1*, *''MissionControl2'' := 2*,
''Bot1'' := 1, *''Bot2'' := 2*)
value *BLP-offending-list fabNet sensorProps-conf*
value *sinvar fabNet sensorProps-conf*

Complete example:

definition *sensorProps-NMParams-try3* :: (*string*, *nat*) *TopoS-Params* **where**
sensorProps-NMParams-try3 ≡ (| *node-properties* = [*''PresenceSensor''* ↦ *2*,
''Webcam'' ↦ *3*,
''SensorSink'' ↦ *3*,
''Statistics'' ↦ *3*] |)
value *BLP-eval fabNet sensorProps-NMParams-try3*

export-code *SINVAR-LIB-BLPbasic* **checking** *Scala*

hide-const (**open**) *NetModel-node-props BLP-offending-list BLP-eval*

hide-const (**open**) *sinvar*

end
theory *SINVAR-Subnets*
imports../*TopoS-Helper*
begin

6.1 SecurityInvariant Subnets

If unsure, maybe you should look at `SINVAR_SubnetsInGW.thy`

datatype *subnets* = *Subnet nat* | *BorderRouter nat* | *Unassigned*

definition *default-node-properties* :: *subnets*
where *default-node-properties* ≡ *Unassigned*

fun *allowed-subnet-flow* :: *subnets* ⇒ *subnets* ⇒ *bool* **where**
allowed-subnet-flow (*Subnet s1*) (*Subnet s2*) = (*s1* = *s2*) |
allowed-subnet-flow (*Subnet s1*) (*BorderRouter s2*) = (*s1* = *s2*) |
allowed-subnet-flow (*Subnet s1*) *Unassigned* = *True* |
allowed-subnet-flow (*BorderRouter s1*) (*Subnet s2*) = *False* |
allowed-subnet-flow (*BorderRouter s1*) *Unassigned* = *True* |
allowed-subnet-flow (*BorderRouter s1*) (*BorderRouter s2*) = *True* |
allowed-subnet-flow Unassigned Unassigned = *True* |
allowed-subnet-flow Unassigned - = *False*

fun *sinvar* :: *'v graph* ⇒ (*'v* ⇒ *subnets*) ⇒ *bool* **where**
sinvar G nP = (∀ (*e1*,*e2*) ∈ *edges G*. *allowed-subnet-flow* (*nP e1*) (*nP e2*))

definition *receiver-violation* :: *bool* **where** *receiver-violation* = *False*

6.1.1 Preliminaries

lemma *sinvar-mono*: *SecurityInvariant-withOffendingFlows.sinvar-mono sinvar*
apply(*simp only*: *SecurityInvariant-withOffendingFlows.sinvar-mono-def*)
apply(*clarify*)
by *auto*

interpretation *SecurityInvariant-preliminaries*
where *sinvar* = *sinvar*
apply *unfold-locales*
apply(*frule-tac finite-distinct-list*[*OF wf-graph.finiteE*])
apply(*erule-tac exE*)
apply(*rename-tac list-edges*)
apply(*rule-tac ff*=*list-edges* **in** *SecurityInvariant-withOffendingFlows.mono-imp-set-offending-flows-not-empty*[*OF sinvar-mono*])
apply(*auto*)[*6*]
apply(*auto simp add*: *SecurityInvariant-withOffendingFlows.is-offending-flows-def graph-ops*)[*1*]
apply(*fact SecurityInvariant-withOffendingFlows.sinvar-mono-imp-is-offending-flows-mono*[*OF sinvar-mono*])
done

6.1.2 ENF

lemma *Unassigned-only-to-Unassigned*: *allowed-subnet-flow Unassigned e2* ⟷ *e2* = *Unassigned*
by(*case-tac e2*, *simp-all*)
lemma *All-to-Unassigned*: ∀ *e1*. *allowed-subnet-flow e1 Unassigned*
by (*rule allI*, *case-tac e1*, *simp-all*)
lemma *Unassigned-default-candidate*: ∀ *nP e1 e2*. ¬ *allowed-subnet-flow* (*nP e1*) (*nP e2*) ⟶ ¬ *allowed-subnet-flow Unassigned* (*nP e2*)
apply(*rule allI*)+
apply(*case-tac nP e2*)
apply *simp*
apply *simp*
by(*simp add*: *All-to-Unassigned*)
lemma *allowed-subnet-flow-refl*: ∀ *e*. *allowed-subnet-flow e e*
by(*rule allI*, *case-tac e*, *simp-all*)
lemma *Subnets-ENF*: *SecurityInvariant-withOffendingFlows.sinvar-all-edges-normal-form sinvar allowed-subnet-flow*
unfolding *SecurityInvariant-withOffendingFlows.sinvar-all-edges-normal-form-def*
by *simp*
lemma *Subnets-ENF-refl*: *SecurityInvariant-withOffendingFlows.ENF-refl sinvar allowed-subnet-flow*
unfolding *SecurityInvariant-withOffendingFlows.ENF-refl-def*
apply(*rule conjI*)
apply(*simp add*: *Subnets-ENF*)
apply(*simp add*: *allowed-subnet-flow-refl*)
done

definition *Subnets-offending-set*:: ′*v graph* ⇒ (′*v* ⇒ *subnets*) ⇒ (′*v* × ′*v*) *set set* **where**
Subnets-offending-set G nP = (*if sinvar G nP then*

{}
else
{ {*e* ∈ *edges G*. *case e of* (*e1*,*e2*) ⇒ ¬ *allowed-subnet-flow* (*nP e1*) (*nP e2*)} })
lemma *Subnets-offending-set*:
SecurityInvariant-withOffendingFlows.set-offending-flows sinvar = *Subnets-offending-set*
apply(*simp only*: *fun-eq-iff ENF-offending-set*[*OF Subnets-ENF*] *Subnets-offending-set-def*)
apply(*rule allI*)+
apply(*rename-tac G nP*)
apply(*auto*)
done

interpretation *Subnets*: *SecurityInvariant-ACS*
where *default-node-properties* = *SINVAR-Subnets.default-node-properties*
and *sinvar* = *SINVAR-Subnets.sinvar*
rewrites *SecurityInvariant-withOffendingFlows.set-offending-flows sinvar* = *Subnets-offending-set*
unfolding *SINVAR-Subnets.default-node-properties-def*
apply *unfold-locales*
apply(*rule ballI*)
apply (*rule SecurityInvariant-withOffendingFlows.ENF-fsts-refl-instance*[*OF Subnets-ENF-refl Unassigned-default-candidate*])[*1*]
apply(*simp-all*)[*2*]
apply(*erule default-uniqueness-by-counterexample-ACS*)
apply (*simp add*: *SecurityInvariant-withOffendingFlows.set-offending-flows-def*
SecurityInvariant-withOffendingFlows.is-offending-flows-min-set-def
SecurityInvariant-withOffendingFlows.is-offending-flows-def)
apply (*simp add*:*graph-ops*)
apply (*simp split*: *prod.split-asm prod.split*)
apply(*rule-tac x*=⦇ *nodes*={*vertex-1*,*vertex-2*}, *edges* = {(*vertex-1*,*vertex-2*)} ⦈ **in** *exI*, *simp*)
apply(*rule conjI*)
apply(*simp add*: *wf-graph-def*)
apply(*case-tac otherbot*, *simp-all*)
apply(*rename-tac mysubnetcase*)
apply(*rule-tac x*=(λ *x*. *Unassigned*)(*vertex-1* := *Unassigned*, *vertex-2* := *BorderRouter mysubnetcase*) **in** *exI*, *simp*)
apply(*rule-tac x*=*vertex-1* **in** *exI*, *simp*)
apply(*rule-tac x*={(*vertex-1*,*vertex-2*)} **in** *exI*, *simp*)
apply(*rule-tac x*=(λ *x*. *Unassigned*)(*vertex-1* := *Unassigned*, *vertex-2* := *BorderRouter whatever*) **in** *exI*, *simp*)
apply(*rule-tac x*=*vertex-1* **in** *exI*, *simp*)
apply(*rule-tac x*={(*vertex-1*,*vertex-2*)} **in** *exI*, *simp*)
apply(*fact Subnets-offending-set*)
done

lemma *TopoS-Subnets*: *SecurityInvariant sinvar default-node-properties receiver-violation*
unfolding *receiver-violation-def* **by** *unfold-locales*

6.1.3 Analysis

lemma *violating-configurations*: ¬ *sinvar G nP* ⟹
∃ (*e1*, *e2*) ∈ *edges G*. *nP e1* = *Unassigned* ∨ (∃ *s1*. *nP e1* = *Subnet s1*) ∨ (∃ *s1*. *nP e1* = *BorderRouter s1*)
apply *simp*

apply *clarify*
apply(*rename-tac a b*)
apply(*case-tac nP b*, *simp-all*)
 apply(*case-tac nP a*, *simp-all*)
  apply *blast*
  apply *blast*
 apply *blast*
 apply(*case-tac nP a*, *simp-all*)
 apply *blast*
 apply *blast*
apply(*simp add*: *All-to-Unassigned*)
done

All cases where the model can become invalid:

theorem *violating-configurations-exhaust*: ¬ *sinvar G nP* ⟷
 (∃ (*e1*, *e2*) ∈ (*edges G*).
  nP e1 = *Unassigned* ∧ *nP e2* ≠ *Unassigned* ∨
  (∃ *s1 s2*. *nP e1* = *Subnet s1* ∧ *s1* ≠ *s2* ∧ (*nP e2* = *Subnet s2* ∨ *nP e2* = *BorderRouter s2*)) ∨
  (∃ *s1 s2*. *nP e1* = *BorderRouter s1* ∧ *nP e2* = *Subnet s2*)
) (**is** *?l* ⟷ *?r*)
proof
 assume *?l*
  have *violating-configurations-exhaust-Unassigned*:
   (*n1*, *n2*) ∈ (*edges G*) ⟹ *nP n1* = *Unassigned* ⟹ ¬ *allowed-subnet-flow* (*nP n1*) (*nP n2*) ⟹
    ∃ (*e1*, *e2*) ∈ (*edges G*). *nP e1* = *Unassigned* ∧ *nP e2* ≠ *Unassigned* **for** *n1 n2*
   by(*cases nP n2*, *simp-all*) *force+*

  have *violating-configurations-exhaust-Subnet*:
   (*n1*, *n2*) ∈ (*edges G*) ⟹ *nP n1* = *Subnet s1′* ⟹ ¬ *allowed-subnet-flow* (*nP n1*) (*nP n2*) ⟹
   ∃ (*e1*, *e2*) ∈ (*edges G*). ∃ *s1 s2*. *nP e1* = *Subnet s1* ∧ *s1* ≠ *s2* ∧ (*nP e2* = *Subnet s2* ∨ *nP e2* = *BorderRouter s2*)
   for *n1 n2 s1′* **by**(*cases nP n2*, *simp-all*) *blast+*

  have *violating-configurations-exhaust-BorderRouter*:
  (*n1*, *n2*) ∈ (*edges G*) ⟹ *nP n1* = *BorderRouter s1′* ⟹ ¬ *allowed-subnet-flow* (*nP n1*) (*nP n2*) ⟹
   ∃ (*e1*, *e2*) ∈ (*edges G*). ∃ *s1 s2*. *nP e1* = *BorderRouter s1* ∧ *nP e2* = *Subnet s2* **for** *n1 n2 s1′*
   by(*cases nP n2*, *simp-all*) *blast+*

  from ‹*?l*› **show** *?r*
  apply *simp*
  apply *clarify*
  apply(*rename-tac n1 n2*)
  apply(*case-tac nP n1*, *simp-all*)
   apply(*rename-tac s1*)
   apply(*drule-tac s1′=s1* **in** *violating-configurations-exhaust-Subnet*, *simp-all*)
   apply *blast*
   apply(*rename-tac s1*)
   apply(*drule-tac s1′=s1* **in** *violating-configurations-exhaust-BorderRouter*, *simp-all*)
   apply *blast*
  apply(*drule-tac violating-configurations-exhaust-Unassigned*, *simp-all*)
  apply *blast*
  done
 next

```
 assume ?r thus ?l
   apply simp
   apply(clarify)
   apply(safe)
      apply(rule-tac x=(a,b) in bexI)
       apply (simp add: Unassigned-only-to-Unassigned; fail)
      apply(simp; fail)
     apply(rule-tac x=(a,b) in bexI)
      apply(simp; fail)
     apply(simp; fail)
    apply(rule-tac x=(a,b) in bexI)
     apply(simp; fail)
    apply(simp; fail)
   apply(rule-tac x=(a,b) in bexI)
    apply(simp; fail)
   apply(simp; fail)
  done
qed


hide-fact (open) sinvar-mono
hide-const (open) sinvar receiver-violation default-node-properties

end
theory SINVAR-Subnets-impl
imports SINVAR-Subnets ../TopoS-Interface-impl
begin
```

6.1.4 SecurityInvariant Subnets List Implementation

```
code-identifier code-module SINVAR-Subnets-impl => (Scala) SINVAR-Subnets

fun sinvar :: 'v list-graph ⇒ ('v ⇒ subnets) ⇒ bool where
  sinvar G nP = (∀ (e1,e2) ∈ set (edgesL G). allowed-subnet-flow (nP e1) (nP e2))


definition Subnets-offending-list:: 'v list-graph ⇒ ('v ⇒ subnets) ⇒ ('v × 'v) list list where
  Subnets-offending-list G nP = (if sinvar G nP then
    []
   else
    [ [e ← edgesL G. case e of (e1,e2) ⇒ ¬ allowed-subnet-flow (nP e1) (nP e2)] ])



definition NetModel-node-props P = (λ i. (case (node-properties P) i of Some property ⇒ property |
None ⇒ SINVAR-Subnets.default-node-properties))
lemma[code-unfold]: SecurityInvariant.node-props SINVAR-Subnets.default-node-properties P = Net-
Model-node-props P
apply(simp add: NetModel-node-props-def)
done

definition Subnets-eval G P = (wf-list-graph G ∧
  sinvar G (SecurityInvariant.node-props SINVAR-Subnets.default-node-properties P))
```

interpretation *Subnets-impl*: *TopoS-List-Impl*
where *default-node-properties=SINVAR-Subnets.default-node-properties*
and *sinvar-spec=SINVAR-Subnets.sinvar*
and *sinvar-impl=sinvar*
and *receiver-violation=SINVAR-Subnets.receiver-violation*
and *offending-flows-impl=Subnets-offending-list*
and *node-props-impl=NetModel-node-props*
and *eval-impl=Subnets-eval*
apply(*unfold TopoS-List-Impl-def*)
apply(*rule conjI*)
apply(*simp add*: *TopoS-Subnets list-graph-to-graph-def*)
apply(*rule conjI*)
apply(*simp add*: *list-graph-to-graph-def Subnets-offending-set Subnets-offending-set-def Subnets-offending-list-def*)
apply(*rule conjI*)
apply(*simp only*: *NetModel-node-props-def*)
apply(*metis Subnets.node-props.simps Subnets.node-props-eq-node-props-formaldef*)
apply(*simp only*: *Subnets-eval-def*)
apply(*simp add*: *TopoS-eval-impl-proofrule*[*OF TopoS-Subnets*])
apply(*simp-all add*: *list-graph-to-graph-def*)
done

6.1.5 Subnets packing

definition *SINVAR-LIB-Subnets* :: (*'v*::*vertex*, *SINVAR-Subnets.subnets*) *TopoS-packed* **where**
SINVAR-LIB-Subnets ≡
⦇ *nm-name* = *''Subnets''*,
nm-receiver-violation = *SINVAR-Subnets.receiver-violation*,
nm-default = *SINVAR-Subnets.default-node-properties*,
nm-sinvar = *sinvar*,
nm-offending-flows = *Subnets-offending-list*,
nm-node-props = *NetModel-node-props*,
nm-eval = *Subnets-eval*
⦈
interpretation *SINVAR-LIB-Subnets-interpretation*: *TopoS-modelLibrary SINVAR-LIB-Subnets SINVAR-Subnets.sinvar*
apply(*unfold TopoS-modelLibrary-def SINVAR-LIB-Subnets-def*)
apply(*rule conjI*)
apply(*simp*)
apply(*simp*)
by(*unfold-locales*)

Examples

definition *example-net-sub* :: *nat list-graph* **where**
example-net-sub ≡ ⦇ *nodesL* = [*1*::*nat*,*2*,*3*,*4*, *8*,*9*, *11*,*12*, *42*],
edgesL = [(*1*,*2*),(*1*,*3*),(*1*,*4*),(*2*,*1*),(*2*,*3*),(*2*,*4*),(*3*,*1*),(*3*,*2*),(*3*,*4*),(*4*,*1*),(*4*,*2*),(*4*,*3*),
(*4*,*11*),(*1*,*11*),
(*8*,*9*),(*9*,*8*),
(*8*,*12*),
(*11*,*12*),
(*11*,*42*), (*12*,*42*), (*3*,*42*)] ⦈
value *wf-list-graph example-net-sub*

definition *example-conf-sub* **where**

example-conf-sub ≡ ((λ*e. SINVAR-Subnets.default-node-properties*)
(*1* := *Subnet 1*, *2*:= *Subnet 1*, *3*:= *Subnet 1*, *4*:=*Subnet 1*,
11:=*BorderRouter 1*,
8:=*Subnet 2*, *9*:=*Subnet 2*,
12:=*BorderRouter 2*,
42 := *Unassigned*))

value *sinvar example-net-sub example-conf-sub*

definition *example-net-sub-invalid* **where**
example-net-sub-invalid ≡ *example-net-sub*(| *edgesL* := (*42*,*4*)#(*3*,*8*)#(*11*,*8*)#(*edgesL example-net-sub*)|)

value *sinvar example-net-sub-invalid example-conf-sub*
value *Subnets-offending-list example-net-sub-invalid example-conf-sub*

value *sinvar*
(| *nodesL* = [*1*::*nat*,*2*,*3*,*4*], *edgesL* = [(*1*,*2*), (*2*,*3*), (*3*,*4*), (*8*,*9*),(*9*,*8*)] |)
(λ*e. SINVAR-Subnets.default-node-properties*)
value *sinvar*
(| *nodesL* = [*1*::*nat*,*2*,*3*,*4*,*8*,*9*,*11*,*12*], *edgesL* = [(*1*,*2*),(*2*,*3*),(*3*,*4*), (*4*,*11*),(*1*,*11*), (*8*,*9*),(*9*,*8*),(*8*,*12*), (*11*,*12*)] |)
((λ*e. SINVAR-Subnets.default-node-properties*)(*1* := *Subnet 1*, *2*:= *Subnet 1*, *3*:= *Subnet 1*, *4*:=*Subnet 1*, *11*:=*BorderRouter 1*,
8:=*Subnet 2*, *9*:=*Subnet 2*, *12*:=*BorderRouter 2*))
value *sinvar*
(| *nodesL* = [*1*::*nat*,*2*,*3*,*4*,*8*,*9*,*11*,*12*], *edgesL* = [(*1*,*2*),(*2*,*3*),(*3*,*4*), (*4*,*11*),(*1*,*11*), (*8*,*9*),(*9*,*8*),(*8*,*12*), (*11*,*12*)] |)
((λ*e. SINVAR-Subnets.default-node-properties*)(*1* := *Subnet 1*, *2*:= *Subnet 1*, *3*:= *Subnet 1*, *4*:=*Subnet 1*, *11*:=*BorderRouter 1*))
value *sinvar*
(| *nodesL* = [*1*::*nat*,*2*,*3*,*4*,*8*,*9*,*10*], *edgesL* = [(*1*,*2*), (*2*,*3*), (*3*,*4*), (*8*,*9*),(*9*,*8*)] |)
((λ*e. SINVAR-Subnets.default-node-properties*)(*8*:=*Subnet 8*, *9*:=*Subnet 8*))

hide-const (**open**) *NetModel-node-props*
hide-const (**open**) *sinvar*

end
theory *SINVAR-DomainHierarchyNG*
imports ../*TopoS-Helper*
HOL−Lattice.CompleteLattice
begin

6.2 SecurityInvariant DomainHierarchyNG

6.2.1 Datatype Domain Hierarchy

A fully qualified domain name for an entity in a tree-like hierarchy

datatype *domainNameDept* = *Dept string domainNameDept* (**infixr** ‹−−› *65*) |
Leaf — leaf of the tree, end of all domainNames

Example: the CoffeeMachine of I8

value *''i8''−−''CoffeeMachine''−−Leaf*

A tree strucuture to represent the general hierarchy, i.e. possible domainNameDepts

datatype *domainTree* = *Department*
string — division
domainTree list — sub divisions

one step in tree to find matching department

fun *hierarchy-next* :: *domainTree list* ⇒ *domainNameDept* ⇒ *domainTree option* **where**
hierarchy-next [] - = *None* |
hierarchy-next (*s*#*ss*) *Leaf* = *None* |
hierarchy-next ((*Department d ds*)#*ss*) (*Dept n ns*) = (*if d*=*n then Some* (*Department d ds*) *else hierarchy-next ss* (*Dept n ns*))

Examples:

lemma *hierarchy-next* [*Department ''i20''* [], *Department ''i8''* [*Department ''CoffeeMachine''* [], *Department ''TeaMachine''* []]]
(*''i8''−−Leaf*)
=
Some (*Department ''i8''* [*Department ''CoffeeMachine''* [], *Department ''TeaMachine''* []]) **by** *eval*
lemma *hierarchy-next* [*Department ''i20''* [], *Department ''i8''* [*Department ''CoffeeMachine''* [], *Department ''TeaMachine''* []]]
(*''i8''−−''whatsoever''−−Leaf*)
=
Some (*Department ''i8''* [*Department ''CoffeeMachine''* [], *Department ''TeaMachine''* []]) **by** *eval*
lemma *hierarchy-next* [*Department ''i20''* [], *Department ''i8''* [*Department ''CoffeeMachine''* [], *Department ''TeaMachine''* []]]
Leaf
= *None* **by** *eval*
lemma *hierarchy-next* [*Department ''i20''* [], *Department ''i8''* [*Department ''CoffeeMachine''* [], *Department ''TeaMachine''* []]]
(*''i0''−−Leaf*)
= *None* **by** *eval*

Does a given *domainNameDept* match the specified tree structure?

fun *valid-hierarchy-pos* :: *domainTree* ⇒ *domainNameDept* ⇒ *bool* **where**
valid-hierarchy-pos (*Department d ds*) *Leaf* = *True* |
valid-hierarchy-pos (*Department d ds*) (*Dept n Leaf*) = (*d*=*n*) |
valid-hierarchy-pos (*Department d ds*) (*Dept n ns*) = (*n*=*d* ∧
(*case hierarchy-next ds ns of*
None ⇒ *False* |
Some t ⇒ *valid-hierarchy-pos t ns*))

Examples:

lemma *valid-hierarchy-pos* (*Department ''TUM''* []) *Leaf* **by** *eval*
lemma *valid-hierarchy-pos* (*Department ''TUM''* []) *Leaf* **by** *eval*
lemma *valid-hierarchy-pos* (*Department ''TUM''* []) (*''TUM''−−Leaf*) **by** *eval*
lemma *valid-hierarchy-pos* (*Department ''TUM''* []) (*''TUM''−−''facilityManagement''−−Leaf*) = *False* **by** *eval*

lemma *valid-hierarchy-pos* (*Department* ''*TUM*'' []) (''*LMU*''−−*Leaf*) = *False* **by** *eval*
lemma *valid-hierarchy-pos* (*Department* ''*TUM*'' [*Department* ''*i8*'' [], (*Department* ''*i20*'' [])]) (''*TUM*''−−*Leaf*) **by** *eval*
lemma *valid-hierarchy-pos* (*Department* ''*TUM*'' [*Department* ''*i8*'' [], *Department* ''*i20*'' []]) (''*TUM*''−−''*i8*''−−*Leaf*) **by** *eval*

```
lemma valid-hierarchy-pos
    (Department ''TUM'' [
       Department ''i8'' [
        Department ''CoffeeMachine'' [],
        Department ''TeaMachine'' []
       ],
       Department ''i20'' []
    ])
    (''TUM''−−''i8''−−''CoffeeMachine''−−Leaf) by eval
```

lemma *valid-hierarchy-pos* (*Department* ''*TUM*'' [*Department* ''*i8*'' [*Department* ''*CoffeeMachine*'' [], *Department* ''*TeaMachine*'' []], *Department* ''*i20*'' []])
(''*TUM*''−−''*i8*''−−''*CleanKitchen*''−−*Leaf*) = *False* **by** *eval*

```
instantiation domainNameDept :: order
begin
  print-context

  fun less-eq-domainNameDept :: domainNameDept ⇒ domainNameDept ⇒ bool where
    Leaf ≤ (Dept - -) = False |
    (Dept - -) ≤ Leaf = True |
    Leaf ≤ Leaf = True |
    (Dept n1 n1s) ≤ (Dept n2 n2s) = (n1=n2 ∧ n1s ≤ n2s)

  fun less-domainNameDept :: domainNameDept ⇒ domainNameDept ⇒ bool where
    Leaf < Leaf = False |
    Leaf < (Dept - -) = False |
    (Dept - -) < Leaf = True |
    (Dept n1 n1s) < (Dept n2 n2s) = (n1=n2 ∧ n1s < n2s)

  lemma Leaf-Top: a ≤ Leaf
    apply(case-tac a)
    by(simp-all)

  lemma Leaf-Top-Unique: Leaf ≤ a = (a = Leaf)
    apply(case-tac a)
    by(simp-all)

  lemma no-Bot: n1 ≠ n2 ⟹ z ≤ n1 −− n1s ⟹ z ≤ n2 −− n2s ⟹ False
    apply(case-tac z)
    by(simp-all)

  lemma uncomparable-sup-is-Top: n1 ≠ n2 ⟹ n1 −− x ≤ z ⟹ n2 −− y ≤ z ⟹ z = Leaf
    apply(case-tac z)
    by(simp-all)

 lemma common-inf-imp-comparable: (z::domainNameDept) ≤ a ⟹ z ≤ b ⟹ a ≤ b ∨ b ≤ a
```

```
    apply(induction z arbitrary: a b)
     apply(rename-tac zn zdpt a b)
     apply(simp-all add: Leaf-Top-Unique)
    apply(case-tac a)
     apply(rename-tac an adpt)
     apply(simp-all add: Leaf-Top)
    apply(case-tac b)
     apply(rename-tac bn bdpt)
     apply(simp-all add: Leaf-Top)
    done

  lemma prepend-domain: a ≤ b ⟹ x−−a ≤ x−−b
    by(simp)
  lemma unfold-dmain-leq: y ≤ zn −− zns ⟹ ∃ yns. y = zn −− yns ∧ yns ≤ zns
    proof −
      assume a1: y ≤ zn −− zns
     obtain sk30 :: domainNameDept ⇒ char list and sk31 :: domainNameDept ⇒ domainNameDept
where ∀ x0. sk30 x0 −− sk31 x0 = x0 ∨ Leaf = x0
        by (metis domainNameDept.exhaust)
      thus ∃ yns. y = zn −− yns ∧ yns ≤ zns
        using a1 by (metis less-eq-domainNameDept.simps(1) less-eq-domainNameDept.simps(4))
    qed

  lemma less-eq-refl:
    fixes x :: domainNameDept
    shows x ≤ y ⟹ y ≤ z ⟹ x ≤ z
    proof −
    have x ≤ y ⟶ y ≤ z ⟶ x ≤ z
      proof(induction z arbitrary:x y)
      case Leaf
        have x ≤ Leaf using Leaf-Top by simp
        thus ?case by simp
      next
      case (Dept zn zns)
        show ?case proof(clarify)
          assume a1: x ≤ y and a2: y ≤ zn−−zns
         from unfold-dmain-leq[OF a2] obtain yns where y1: y = zn−−yns and y2: yns ≤ zns by
auto
          from unfold-dmain-leq this a1 obtain xns where x1: x = zn −− xns and x2: xns ≤ yns
by blast
          from Dept y2 x2 have xns ≤ zns by simp
          from this x1 show x ≤ zn−−zns by simp
        qed
      qed
    thus x ≤ y ⟹ y ≤ z ⟹ x ≤ z by simp
    qed

  instance
    proof
      fix x y ::domainNameDept
      show (x < y) = (x ≤ y ∧ ¬ y ≤ x)
        apply(induction rule: less-domainNameDept.induct)
         apply(simp-all)
        by blast
```

next
fix *x*::*domainNameDept*
show $x \leq x$
using[[*show-types*]] **apply**(*induction x*)
by *simp-all*
next
fix *x y z* :: *domainNameDept*
show $x \leq y \Longrightarrow y \leq z \Longrightarrow x \leq z$ **apply** (*rule less-eq-refl*) **by** *simp-all*
next
fix *x y* ::*domainNameDept*
show $x \leq y \Longrightarrow y \leq x \Longrightarrow x = y$
apply(*induction rule*: *less-domainNameDept.induct*)
by(*simp-all*)
qed
end

instantiation *domainNameDept* :: *Orderings.top*
begin
definition *top-domainNameDept* **where** *Orderings.top* ≡ *Leaf*
instance
by *intro-classes*
end

lemma (*''TUM''*−−*''BLUBB''*−−*Leaf*) ≤ (*''TUM''*−−*Leaf*) **by** *eval*

lemma (*''TUM''*−−*''i8''*−−*Leaf*) ≤ (*''TUM''*−−*Leaf*) **by** *eval*
lemma ¬ (*''TUM''*−−*Leaf*) ≤ (*''TUM''*−−*''i8''*−−*Leaf*) **by** *eval*
lemma *valid-hierarchy-pos* (*Department ''TUM''* [*Department ''i8''* [], *Department ''i20''* []]) (*''TUM''*−−*''i8''*−−*Leaf*) **by** *eval*

lemma (*''TUM''*−−*Leaf*) ≤ *Leaf* **by** *eval*
lemma *valid-hierarchy-pos* (*Department ''TUM''* [*Department ''i8''* [], *Department ''i20''* []]) (*Leaf*) **by** *eval*

lemma ¬ *Leaf* ≤ (*''TUM''*−−*Leaf*) **by** *eval*
lemma *valid-hierarchy-pos* (*Department ''TUM''* [*Department ''i8''* [], *Department ''i20''* []]) (*''TUM''*−−*Leaf*) **by** *eval*

lemma ¬ (*''TUM''*−−*''BLUBB''*−−*Leaf*) ≤ (*''X''*−−*''TUM''*−−*''BLUBB''*−−*Leaf*) **by** *eval*

lemma (*''TUM''*−−*''i8''*−−*''CoffeeMachine''*−−*Leaf*) ≤ (*''TUM''*−−*''i8''*−−*Leaf*) **by** *eval*
lemma (*''TUM''*−−*''i8''*−−*Leaf*) ≤ (*''TUM''*−−*''i8''*−−*Leaf*) **by** *eval*
lemma (*''TUM''*−−*''i8''*−−*''CoffeeMachine''*−−*Leaf*) ≤ (*''TUM''*−−*Leaf*) **by** *eval*
lemma (*''TUM''*−−*''i8''*−−*''CoffeeMachine''*−−*Leaf*) ≤ (*Leaf*) **by** *eval*
lemma ¬ (*''TUM''*−−*''i8''*−−*Leaf*) ≤ (*''TUM''*−−*''i20''*−−*Leaf*) **by** *eval*
lemma ¬ (*''TUM''*−−*''i20''*−−*Leaf*) ≤ (*''TUM''*−−*''i8''*−−*Leaf*) **by** *eval*

6.2.2 Adding Chop

by putting entities higher in the hierarchy.

fun *domainNameDeptChopOne* :: *domainNameDept* ⇒ *domainNameDept* **where**
domainNameDeptChopOne Leaf = *Leaf* |
domainNameDeptChopOne (*name*−−*Leaf*) = *Leaf* |
domainNameDeptChopOne (*name*−−*dpt*) = *name*−−(*domainNameDeptChopOne dpt*)

lemma *domainNameDeptChopOne* (*''i8''−−''CoffeeMachine''−−Leaf*) = *''i8''* −− *Leaf* **by** *eval*
lemma *domainNameDeptChopOne* (*''i8''−−''CoffeeMachine''−−''CoffeeSlave''−−Leaf*) = *''i8''* −− *''CoffeeMachine''* −− *Leaf* **by** *eval*
lemma *domainNameDeptChopOne Leaf* = *Leaf* **by**(*fact domainNameDeptChopOne.simps*(*1*))

theorem *chopOne-not-decrease*: $dn \leq$ *domainNameDeptChopOne dn*
  apply(*induction dn*)
   apply(*rename-tac name dpt*)
   apply(*drule-tac x=name* **in** *prepend-domain*)
   apply(*case-tac dpt*)
    apply *simp-all*
done

lemma *chopOneContinue*: *dpt* ≠ *Leaf* ⟹ *domainNameDeptChopOne* (*name* −− *dpt*) = *name* −− *domainNameDeptChopOne* (*dpt*)
apply(*case-tac dpt*)
 by *simp-all*

fun *domainNameChop* :: *domainNameDept* ⇒ *nat* ⇒ *domainNameDept* **where**
  domainNameChop Leaf - = *Leaf* |
  domainNameChop namedpt 0 = *namedpt* |
  domainNameChop namedpt (*Suc n*) = *domainNameChop* (*domainNameDeptChopOne namedpt*) *n*

lemma *domainNameChop* (*''i8''−−''CoffeeMachine''−−Leaf*) *2* = *Leaf* **by** *eval*
lemma *domainNameChop* (*''i8''−−''CoffeeMachine''−−''CoffeeSlave''−−Leaf*) *2* = *''i8''−−Leaf* **by** *eval*
lemma *domainNameChop* (*''i8''−−Leaf*) *0* = *''i8''−−Leaf* **by** *eval*
lemma *domainNameChop* (*Leaf*) *8* = *Leaf* **by** *eval*

lemma *chop0*[*simp*]: *domainNameChop dn 0* = *dn*
 apply(*case-tac dn*)
  by *simp-all*

lemma (*domainNameDeptChopOne*^^*2*) (*''d1''−−''d2''−−''d3''−−Leaf*) = *''d1''−−Leaf* **by** *eval*

domainNameChop is equal to applying n times chop one

lemma *domainNameChopFunApply*: *domainNameChop dn n* = (*domainNameDeptChopOne*^^*n*) *dn*
  apply(*induction dn n rule*: *domainNameChop.induct*)
   apply (*simp-all*)
  apply(*rename-tac nat,induct-tac nat*, *simp-all*)
  apply(*rename-tac n*)
  by (*metis funpow-swap1*)

lemma *domainNameChopRotateSuc*: *domainNameChop dn* (*Suc n*) = *domainNameDeptChopOne* (*domainNameChop dn n*)
by(*simp add*: *domainNameChopFunApply*)

lemma *domainNameChopRotate*: *domainNameChop* (*domainNameDeptChopOne dn*) *n* = *domain-*

NameDeptChopOne (*domainNameChop dn n*)
apply(*subgoal-tac domainNameChop* (*domainNameDeptChopOne dn*) *n* = *domainNameChop dn* (*Suc n*))
apply *simp*
apply(*simp add*: *domainNameChopFunApply*)
apply(*case-tac dn*)
by(*simp-all*)

theorem *chop-not-decrease-hierarchy*: *dn* ≤ *domainNameChop dn n*
apply(*induction n*)
apply(*simp*)
apply(*case-tac dn*)
apply(*rename-tac name dpt*)
apply (*simp*)
apply(*simp add*:*domainNameChopRotate*)
apply (*metis chopOne-not-decrease less-eq-refl*)
apply *simp*
done

corollary *dn* ≤ *domainNameDeptChopOne* ((*domainNameDeptChopOne* ^^ *n*) (*dn*))
by (*metis chop-not-decrease-hierarchy domainNameChopFunApply domainNameChopRotateSuc*)

compute maximum common level of both inputs

fun *chop-sup* :: *domainNameDept* ⇒ *domainNameDept* ⇒ *domainNameDept* **where**
chop-sup Leaf - = *Leaf* |
chop-sup - *Leaf* = *Leaf* |
chop-sup (*a*−−*as*) (*b*−−*bs*) = (*if a* ≠ *b then Leaf else a*−−(*chop-sup as bs*))

lemma *chop-sup* (''*a*''−−''*b*''−−''*c*''−−*Leaf*) (''*a*''−−''*b*''−−''*d*''−−*Leaf*) = ''*a*'' −− ''*b*'' −− *Leaf* **by** *eval*
lemma *chop-sup* (''*a*''−−''*b*''−−''*c*''−−*Leaf*) (''*a*''−−''*x*''−−''*d*''−−*Leaf*) = ''*a*'' −− *Leaf* **by** *eval*
lemma *chop-sup* (''*a*''−−''*b*''−−''*c*''−−*Leaf*) (''*x*''−−''*x*''−−''*d*''−−*Leaf*) = *Leaf* **by** *eval*

lemma *chop-sup-commute*: *chop-sup a b* = *chop-sup b a*
apply(*induction a b rule*: *chop-sup.induct*)
apply(*rename-tac a*)
apply(*simp-all*)
apply(*case-tac a*, *simp-all*)
done
lemma *chop-sup-max1*: *a* ≤ *chop-sup a b*
apply(*induction a b rule*: *chop-sup.induct*)
by(*simp-all*)
lemma *chop-sup-max2*: *b* ≤ *chop-sup a b*
apply(*subst chop-sup-commute*)
by(*simp add*: *chop-sup-max1*)

lemma *chop-sup-is-sup*: ∀ *z*. *a* ≤ *z* ∧ *b* ≤ *z* ⟶ *chop-sup a b* ≤ *z*
apply(*clarify*)
apply(*induction a b rule*: *chop-sup.induct*)
apply(*simp-all*)
apply(*rule conjI*)
apply(*clarify*)

```
apply(subgoal-tac z=Leaf)
 apply(simp)
apply(simp add: uncomparable-sup-is-Top)
apply(clarify)
apply(case-tac z)
by(simp-all)
```

```
datatype domainName = DN domainNameDept | Unassigned
```

6.2.3 Makeing it a complete Lattice

```
instantiation domainName :: partial-order
begin

  fun leq-domainName :: domainName ⇒ domainName ⇒ bool  where
    leq-domainName Unassigned - = True |
    leq-domainName - Unassigned = False |
    leq-domainName (DN dnA) (DN dnB) = (dnA ≤ dnB)
instance
  apply(intro-classes)

    apply(case-tac x)
   apply(simp-all)

   apply(case-tac x, rename-tac dnX)
    apply(case-tac y, rename-tac dnY)
     apply(case-tac z, rename-tac dnZ)
      apply(simp-all)

  apply(case-tac x, rename-tac dnX)
   apply(case-tac y, rename-tac dnY)
    apply(simp-all)
  apply(metis domainName.exhaust leq-domainName.simps(2))
  done
end

lemma is-Inf {Unassigned, DN Leaf} Unassigned
  by(simp add: is-Inf-def)
```

The infinum of two elements:

```
fun DN-inf :: domainName ⇒ domainName ⇒ domainName where
  DN-inf Unassigned - = Unassigned |
  DN-inf - Unassigned = Unassigned |
  DN-inf (DN a) (DN b) = (if a ≤ b then DN a else if b ≤ a then DN b else Unassigned)

lemma DN-inf (DN (''TUM''--''i8''--Leaf)) (DN (''TUM''--''i20''--Leaf)) = Unassigned
by eval
 lemma DN-inf (DN (''TUM''--''i8''--Leaf)) (DN (''TUM''--Leaf)) = DN (''TUM'' --
''i8'' -- Leaf) by eval

lemma DN-inf-commute: DN-inf x y = DN-inf y x
```

apply(*induction x y rule*: *DN-inf.induct*)
apply(*rename-tac x*)
apply(*case-tac x*)
by (*simp-all*)

lemma *DN-inf-is-inf*: *is-inf x y* (*DN-inf x y*)
apply(*induction x y rule*: *DN-inf.induct*)
apply(*simp add*: *is-inf-def*)
apply(*simp add*: *is-inf-def*)
apply(*simp add*: *is-inf-def*)
apply(*clarify*)
apply(*rename-tac z*)
apply(*case-tac z*)
apply(*simp*)
apply(*rename-tac zn*)
apply(*simp-all*)
using *common-inf-imp-comparable* **by** *blast*

fun *DN-sup* :: *domainName* ⇒ *domainName* ⇒ *domainName* **where**
DN-sup Unassigned a = *a* |
DN-sup a Unassigned = *a* |
DN-sup (*DN a*) (*DN b*) = *DN* (*chop-sup a b*)

lemma *DN-sup-commute*: *DN-sup x y* = *DN-sup y x*
apply(*induction x y rule*: *DN-sup.induct*)
apply(*rename-tac x*)
apply(*case-tac x*)
by(*simp-all add*: *chop-sup-commute*)

lemma *DN-sup-is-sup*: *is-sup x y* (*DN-sup x y*)
apply(*induction x y rule*: *DN-inf.induct*)
apply(*simp add*: *is-sup-def leq-refl*)
apply(*simp add*: *is-sup-def*)
apply(*simp add*: *is-sup-def chop-sup-max1 chop-sup-max2*)
apply(*clarify*)
apply(*rename-tac z*)
apply(*case-tac z*)
apply(*simp*)
apply(*rename-tac zn*)
apply(*simp-all*)
apply(*clarify*)
apply(*simp add*: *chop-sup-is-sup*)
done

domainName is a Lattice:

instantiation *domainName* :: *lattice*
begin
instance
apply *intro-classes*
apply(*rule-tac x*=*DN-inf x y* **in** *exI*)
apply(*fact DN-inf-is-inf*)
apply(*rule-tac x*=*DN-sup x y* **in** *exI*)
apply(*rule DN-sup-is-sup*)

done
end

datatype *domainNameTrust* = *DN* (*domainNameDept* × *nat*) | *Unassigned*

fun *leq-domainNameTrust* :: *domainNameTrust* ⇒ *domainNameTrust* ⇒ *bool* (**infixr** ‹$\sqsubseteq_{trust}$› *65*)
where
leq-domainNameTrust Unassigned - = *True* |
leq-domainNameTrust - *Unassigned* = *False* |
leq-domainNameTrust (*DN* (*dnA*, *trustA*)) (*DN* (*dnB*, *trustB*)) = (*dnA* ≤ (*domainNameChop dnB trustB*))

lemma *leq-domainNameTrust-refl*: $x \sqsubseteq_{trust} x$
apply(*case-tac x*)
apply(*rename-tac prod*)
apply(*case-tac prod*)
apply(*simp add*: *chop-not-decrease-hierarchy*)
by(*simp*)

lemma *leq-domainNameTrust-NOT-trans*: $\exists x\ y\ z.\ x \sqsubseteq_{trust} y \land y \sqsubseteq_{trust} z \land \neg\ x \sqsubseteq_{trust} z$
apply(*rule-tac x*=*DN* (''*TUM*''−−*Leaf*, *0*) **in** *exI*)
apply(*rule-tac x*=*DN* (''*TUM*''−−''*i8*''−−*Leaf*, *1*) **in** *exI*)
apply(*rule-tac x*=*DN* (''*TUM*''−−''*i8*''−−*Leaf*, *0*) **in** *exI*)
apply(*simp*)
done

lemma *leq-domainNameTrust-NOT-antisym*: $\exists x\ y.\ x \sqsubseteq_{trust} y \land y \sqsubseteq_{trust} x \land x \neq y$
apply(*rule-tac x*=*DN* (*Leaf*, *3*) **in** *exI*)
apply(*rule-tac x*=*DN* (*Leaf*, *4*) **in** *exI*)
apply(*simp*)
done

6.2.4 The network security invariant

definition *default-node-properties* :: *domainNameTrust*
where *default-node-properties* = *Unassigned*

The sender is, noticing its trust level, on the same or higher hierarchy level as the receiver.

fun *sinvar* :: *'v graph* ⇒ (*'v* ⇒ *domainNameTrust*) ⇒ *bool* **where**
sinvar G nP = (∀ (*s*, *r*) ∈ *edges G*. (*nP r*) $\sqsubseteq_{trust}$ (*nP s*))

a domain name must be in the supplied tree

fun *verify-globals* :: *'v graph* ⇒ (*'v* ⇒ *domainNameTrust*) ⇒ *domainTree* ⇒ *bool* **where**
verify-globals G nP tree = (∀ *v* ∈ *nodes G*.
case (*nP v*) *of Unassigned* ⇒ *True* | *DN* (*level*, *trust*) ⇒ *valid-hierarchy-pos tree level*
)

lemma *verify-globals* (| *nodes=set* [*1*,*2*,*3*], *edges=set* [] |) (λ*n. default-node-properties*) (*Department* ''*TUM*'' [])
  by (*simp add*: *default-node-properties-def*)

definition *receiver-violation* :: *bool* **where** *receiver-violation* = *False*

thm *SecurityInvariant-withOffendingFlows.sinvar-mono-def*
lemma *sinvar-mono*: *SecurityInvariant-withOffendingFlows.sinvar-mono sinvar*
  apply(*rule-tac SecurityInvariant-withOffendingFlows.sinvar-mono-I-proofrule*)
   apply(*auto*)
  apply(*rename-tac nP e1 e2 N E' e1' e2' E*)
  apply(*blast*)
 done

interpretation *SecurityInvariant-preliminaries*
where *sinvar* = *sinvar*
  apply *unfold-locales*
    apply(*frule-tac finite-distinct-list*[*OF wf-graph.finiteE*])
    apply(*erule-tac exE*)
    apply(*rename-tac list-edges*)
   apply(*rule-tac ff=list-edges* **in** *SecurityInvariant-withOffendingFlows.mono-imp-set-offending-flows-not-empty*[*OF sinvar-mono*])
        apply(*auto*)[*4*]
   apply(*auto simp add*: *SecurityInvariant-withOffendingFlows.is-offending-flows-def graph-ops*)[*1*]
  apply(*fact SecurityInvariant-withOffendingFlows.sinvar-mono-imp-sinvar-mono*[*OF sinvar-mono*])
 apply(*fact SecurityInvariant-withOffendingFlows.sinvar-mono-imp-is-offending-flows-mono*[*OF sinvar-mono*])
done

6.2.5 ENF

 lemma *DomainHierarchyNG-ENF*: *SecurityInvariant-withOffendingFlows.sinvar-all-edges-normal-form sinvar* (λ *s r. r* $\sqsubseteq_{trust}$ *s*)
    unfolding *SecurityInvariant-withOffendingFlows.sinvar-all-edges-normal-form-def*
    by *simp*
  lemma *DomainHierarchyNG-ENF-refl*: *SecurityInvariant-withOffendingFlows.ENF-refl sinvar* (λ *s r. r* $\sqsubseteq_{trust}$ *s*)
    unfolding *SecurityInvariant-withOffendingFlows.ENF-refl-def*
    apply(*rule conjI*)
     apply(*simp add*: *DomainHierarchyNG-ENF*)
    apply(*simp add*: *leq-domainNameTrust-refl*)
  done
   lemma *unassigned-default-candidate*: ∀ *nP s r*. ¬ (*nP r*) $\sqsubseteq_{trust}$ (*nP s*) ⟶ ¬ (*nP r*) $\sqsubseteq_{trust}$ *default-node-properties*
    apply(*clarify*)
    apply (*simp add*: *default-node-properties-def*)
    by (*metis leq-domainNameTrust.elims*(*3*) *leq-domainNameTrust.simps*(*2*))

```
definition DomainHierarchyNG-offending-set:: 'v graph ⇒ ('v ⇒ domainNameTrust) ⇒ ('v × 'v)
set set where
  DomainHierarchyNG-offending-set G nP = (if sinvar G nP then
      {}
     else
      { {e ∈ edges G. case e of (e1,e2) ⇒ ¬ (nP e2) ⊑trust (nP e1)} })
 lemma DomainHierarchyNG-offending-set: SecurityInvariant-withOffendingFlows.set-offending-flows
sinvar = DomainHierarchyNG-offending-set
    apply(simp only: fun-eq-iff SecurityInvariant-withOffendingFlows.ENF-offending-set[OF Domain-
HierarchyNG-ENF] DomainHierarchyNG-offending-set-def)
    apply(rule allI)+
    apply(rename-tac G nP)
    apply(auto split:prod.split-asm prod.split simp add: Let-def)
  done


  lemma Unassigned-unique-default: otherbot ≠ default-node-properties ⟹
        ∃ G nP gP i f.
           wf-graph G ∧
           ¬ sinvar G nP ∧
           f ∈ SecurityInvariant-withOffendingFlows.set-offending-flows sinvar G nP ∧
           sinvar (delete-edges G f) nP ∧
           (i ∈ fst ‘ f ∧ sinvar G (nP(i := otherbot)))
    apply (simp add: SecurityInvariant-withOffendingFlows.set-offending-flows-def
        SecurityInvariant-withOffendingFlows.is-offending-flows-min-set-def
        SecurityInvariant-withOffendingFlows.is-offending-flows-def)
    apply (simp add:graph-ops)
    apply (simp split: prod.split-asm prod.split domainNameTrust.split)
    apply(rule-tac x=⦇ nodes={vertex-1,vertex-2}, edges = {(vertex-1,vertex-2)} ⦈ in exI, simp)
    apply(rule conjI)
     apply(simp add: wf-graph-def)
    apply(case-tac otherbot)
     apply(rename-tac prod)
     apply(case-tac prod)
     apply(rename-tac dn trustlevel)
     apply(clarify)

     apply(case-tac dn)


      apply(rename-tac name dpt)
      apply(simp)
     apply(rule-tac x=(λ x. default-node-properties)(vertex-1 := Unassigned, vertex-2 := DN (name−−dpt,
0 )) in exI, simp)
      apply(rule-tac x=vertex-1 in exI, simp)
      apply(rule-tac x={(vertex-1,vertex-2)} in exI, simp)
      apply(simp add:chop-not-decrease-hierarchy)



     apply(simp)
     apply(rule-tac x=(λ x. default-node-properties)(vertex-1 := Unassigned, vertex-2 := DN (Leaf,
0)) in exI, simp)
     apply(rule-tac x=vertex-1 in exI, simp)
     apply(rule-tac x={(vertex-1,vertex-2)} in exI, simp)
```

apply(*simp add*: *default-node-properties-def*)
done

interpretation *DomainHierarchyNG*: *SecurityInvariant-ACS*
where *default-node-properties* = *default-node-properties*
and *sinvar* = *sinvar*
rewrites *SecurityInvariant-withOffendingFlows.set-offending-flows sinvar* = *DomainHierarchyNG-offending-set*
apply *unfold-locales*
apply(*rule ballI*)
apply(*drule SecurityInvariant-withOffendingFlows.ENF-fsts-refl-instance*[*OF DomainHierarchyNG-ENF-refl unassigned-default-candidate*], *simp-all*)[*1*]
apply(*erule default-uniqueness-by-counterexample-ACS*)
apply(*drule Unassigned-unique-default*)
apply(*simp*)
apply(*fact DomainHierarchyNG-offending-set*)
done

lemma *TopoS-DomainHierarchyNG*: *SecurityInvariant sinvar default-node-properties receiver-violation*
unfolding *receiver-violation-def* **by**(*unfold-locales*)

hide-const (**open**) *sinvar receiver-violation*

end
theory *SINVAR-DomainHierarchyNG-impl*
imports *SINVAR-DomainHierarchyNG* ../*TopoS-Interface-impl*
begin

6.2.6 SecurityInvariant DomainHierarchy List Implementation

code-identifier code-module *SINVAR-DomainHierarchyNG-impl* => (*Scala*) *SINVAR-DomainHierarchyNG*

fun *sinvar* :: *'v list-graph* ⇒ (*'v* ⇒ *domainNameTrust*) ⇒ *bool* **where**
sinvar G nP = (∀ (*s*, *r*) ∈ *set* (*edgesL G*). (*nP r*) $\sqsubseteq_{trust}$ (*nP s*))

definition *DomainHierarchyNG-sanity-check-config* :: *domainNameTrust list* ⇒ *domainTree* ⇒ *bool*
where
DomainHierarchyNG-sanity-check-config host-attributes tree = (∀ *c* ∈ *set host-attributes*.
case c of Unassigned ⇒ *True*
| *DN* (*level*, *trust*) ⇒ *valid-hierarchy-pos tree level*
)

fun *verify-globals* :: *'v list-graph* ⇒ (*'v* ⇒ *domainNameTrust*) ⇒ *domainTree* ⇒ *bool* **where**
verify-globals G nP tree = (∀ *v* ∈ *set* (*nodesL G*).
case (*nP v*) *of Unassigned* ⇒ *True* | *DN* (*level*, *trust*) ⇒ *valid-hierarchy-pos tree level*
)

lemma *DomainHierarchyNG-sanity-check-config c tree* $\Longrightarrow$
  $\{x.\ \exists\, v.\ nP\ v = x\}$ = *set c* $\Longrightarrow$
  verify-globals G nP tree
 apply(*simp add*: *DomainHierarchyNG-sanity-check-config-def split*: *if-split-asm*)
 apply(*clarify*)
 apply(*case-tac nP v*)
  apply(*simp-all*)
 apply(*clarify*)
 by *force*

definition *DomainHierarchyNG-offending-list*:: *'v list-graph* ⇒ (*'v* ⇒ *domainNameTrust*) ⇒ (*'v* × *'v*) *list list* **where**
 DomainHierarchyNG-offending-list G nP = (*if sinvar G nP then*
  []
 else
  [[*e* ← *edgesL G. case e of* (*s*,*r*) ⇒ ¬ (*nP r*) $\sqsubseteq_{trust}$ (*nP s*)]])

lemma *DomainHierarchyNG.node-props P* =
 (λ*i. case node-properties P i of None* ⇒ *SINVAR-DomainHierarchyNG.default-node-properties* | *Some property* ⇒ *property*)
by(*fact SecurityInvariant.node-props.simps*[*OF TopoS-DomainHierarchyNG*, *of P*])

definition *NetModel-node-props P* = (λ *i.* (*case* (*node-properties P*) *i of Some property* ⇒ *property* | *None* ⇒ *SINVAR-DomainHierarchyNG.default-node-properties*))

lemma[*code-unfold*]: *DomainHierarchyNG.node-props P* = *NetModel-node-props P*
by(*simp add*: *NetModel-node-props-def*)

definition *DomainHierarchyNG-eval G P* = (*wf-list-graph G* ∧
 sinvar G (*SecurityInvariant.node-props SINVAR-DomainHierarchyNG.default-node-properties P*))

interpretation *DomainHierarchyNG-impl*:*TopoS-List-Impl*
 where *default-node-properties*=*SINVAR-DomainHierarchyNG.default-node-properties*
 and *sinvar-spec*=*SINVAR-DomainHierarchyNG.sinvar*
 and *sinvar-impl*=*sinvar*
 and *receiver-violation*=*SINVAR-DomainHierarchyNG.receiver-violation*
 and *offending-flows-impl*=*DomainHierarchyNG-offending-list*
 and *node-props-impl*=*NetModel-node-props*
 and *eval-impl*=*DomainHierarchyNG-eval*
apply(*unfold TopoS-List-Impl-def*)
apply(*rule conjI*)
 apply(*simp add*: *TopoS-DomainHierarchyNG list-graph-to-graph-def*; *fail*)
apply(*rule conjI*)
 apply(*simp add*: *list-graph-to-graph-def DomainHierarchyNG-offending-set*
   DomainHierarchyNG-offending-set-def DomainHierarchyNG-offending-list-def; *fail*)
apply(*rule conjI*)

```
  apply(simp only: NetModel-node-props-def)
 apply(metis DomainHierarchyNG.node-props.simps DomainHierarchyNG.node-props-eq-node-props-formaldef)
 apply(simp only: DomainHierarchyNG-eval-def)
 apply(intro allI)
 apply(rule TopoS-eval-impl-proofrule[OF TopoS-DomainHierarchyNG])
 apply(simp add: list-graph-to-graph-def)
done
```

6.2.7 DomainHierarchyNG packing

```
 definition SINVAR-LIB-DomainHierarchyNG :: ('v::vertex, domainNameTrust) TopoS-packed where
   SINVAR-LIB-DomainHierarchyNG ≡
   (| nm-name = ''DomainHierarchyNG'',
     nm-receiver-violation = SINVAR-DomainHierarchyNG.receiver-violation,
     nm-default = SINVAR-DomainHierarchyNG.default-node-properties,
     nm-sinvar = sinvar,
     nm-offending-flows = DomainHierarchyNG-offending-list,
     nm-node-props = NetModel-node-props,
     nm-eval = DomainHierarchyNG-eval
     |)
 interpretation SINVAR-LIB-DomainHierarchyNG-interpretation: TopoS-modelLibrary SINVAR-LIB-DomainHierarc

     SINVAR-DomainHierarchyNG.sinvar
   apply(unfold TopoS-modelLibrary-def SINVAR-LIB-DomainHierarchyNG-def)
   apply(rule conjI)
    apply(simp)
   apply(simp)
   by(unfold-locales)
```

Examples:

```
definition example-TUM-net :: string list-graph where
  example-TUM-net ≡ (| nodesL=[''Gateway'', ''LowerSVR'', ''UpperSRV''],
        edgesL=[
          (''Gateway'',''LowerSVR''), (''Gateway'',''UpperSRV''),
          (''LowerSVR'', ''Gateway''),
          (''UpperSRV'', ''Gateway'')
        ] |)
value wf-list-graph example-TUM-net

definition example-TUM-config :: string ⇒ domainNameTrust where
  example-TUM-config ≡ ((λ e. default-node-properties)
        (''Gateway'':= DN (''ACD''--''AISD''--Leaf, 1),
         ''LowerSVR'':= DN (''ACD''--''AISD''--Leaf, 0),
         ''UpperSRV'':= DN (''ACD''--Leaf, 0)
       ))

definition example-TUM-hierarchy :: domainTree where
example-TUM-hierarchy ≡ (Department ''ACD'' [
            Department ''AISD'' []
       ])

value verify-globals example-TUM-net example-TUM-config example-TUM-hierarchy
value sinvar    example-TUM-net example-TUM-config
```

definition *example-TUM-net-invalid* **where**
example-TUM-net-invalid ≡ *example-TUM-net*⦇*edgesL* :=
(''*LowerSRV*'', ''*UpperSRV*'')#(*edgesL example-TUM-net*)⦈

value *verify-globals example-TUM-net-invalid example-TUM-config example-TUM-hierarchy*
value *sinvar example-TUM-net-invalid example-TUM-config*
value *DomainHierarchyNG-offending-list example-TUM-net-invalid example-TUM-config*

hide-const (**open**) *NetModel-node-props*

hide-const (**open**) *sinvar*

end
theory *SINVAR-BLPtrusted-impl*
imports *SINVAR-BLPtrusted* ../*TopoS-Interface-impl*
begin

6.2.8 SecurityInvariant List Implementation

code-identifier code-module *SINVAR-BLPtrusted-impl* => (*Scala*) *SINVAR-BLPtrusted*

fun *sinvar* :: *'v list-graph* ⇒ (*'v* ⇒ *SINVAR-BLPtrusted.node-config*) ⇒ *bool* **where**
sinvar G nP = (∀ (*e1*,*e2*) ∈ *set* (*edgesL G*). (*if trusted* (*nP e2*) *then True else security-level* (*nP e1*) ≤ *security-level* (*nP e2*)))

definition *BLP-offending-list*:: *'v list-graph* ⇒ (*'v* ⇒ *SINVAR-BLPtrusted.node-config*) ⇒ (*'v* × *'v*) *list list* **where**
BLP-offending-list G nP = (*if sinvar G nP then*
[]
else
[[*e* ← *edgesL G*. *case e of* (*e1*,*e2*) ⇒ ¬ *SINVAR-BLPtrusted.BLP-P* (*nP e1*) (*nP e2*)]])

definition *NetModel-node-props P* = (λ *i*. (*case* (*node-properties P*) *i of Some property* ⇒ *property* | *None* ⇒ *SINVAR-BLPtrusted.default-node-properties*))
lemma[*code-unfold*]: *SecurityInvariant.node-props SINVAR-BLPtrusted.default-node-properties P* = *NetModel-node-props P*
apply(*simp add*: *NetModel-node-props-def*)
done

definition *BLP-eval G P* = (*wf-list-graph G* ∧
sinvar G (*SecurityInvariant.node-props SINVAR-BLPtrusted.default-node-properties P*))

interpretation *BLPtrusted-impl*:*TopoS-List-Impl*
where *default-node-properties*=*SINVAR-BLPtrusted.default-node-properties*
and *sinvar-spec*=*SINVAR-BLPtrusted.sinvar*
and *sinvar-impl*=*sinvar*
and *receiver-violation*=*SINVAR-BLPtrusted.receiver-violation*
and *offending-flows-impl*=*BLP-offending-list*
and *node-props-impl*=*NetModel-node-props*

and *eval-impl=BLP-eval*
apply(*unfold TopoS-List-Impl-def*)
apply(*rule conjI*)
apply(*simp add*: *TopoS-BLPtrusted list-graph-to-graph-def*; *fail*)
apply(*rule conjI*)
apply(*simp add*: *list-graph-to-graph-def BLP-offending-set BLP-offending-set-def BLP-offending-list-def*)
apply(*rule conjI*)
apply(*simp only*: *NetModel-node-props-def*)
apply(*metis BLPtrusted.node-props.simps BLPtrusted.node-props-eq-node-props-formaldef*)
apply(*simp only*: *BLP-eval-def*)
apply(*intro allI*)
apply(*rule TopoS-eval-impl-proofrule*[*OF TopoS-BLPtrusted*])
apply(*simp-all add*: *list-graph-to-graph-def*)
done

6.2.9 BLPtrusted packing

definition *SINVAR-LIB-BLPtrusted* :: (*'v*::*vertex*, *SINVAR-BLPtrusted.node-config*) *TopoS-packed*
where
SINVAR-LIB-BLPtrusted ≡
(| *nm-name* = *''BLPtrusted''*,
nm-receiver-violation = *SINVAR-BLPtrusted.receiver-violation*,
nm-default = *SINVAR-BLPtrusted.default-node-properties*,
nm-sinvar = *sinvar*,
nm-offending-flows = *BLP-offending-list*,
nm-node-props = *NetModel-node-props*,
nm-eval = *BLP-eval*
|)
interpretation *SINVAR-LIB-BLPtrusted-interpretation*: *TopoS-modelLibrary SINVAR-LIB-BLPtrusted*

SINVAR-BLPtrusted.sinvar
apply(*unfold TopoS-modelLibrary-def SINVAR-LIB-BLPtrusted-def*)
apply(*rule conjI*)
apply(*simp*)
apply(*simp*)
by(*unfold-locales*)

6.2.10 Example

export-code *SINVAR-LIB-BLPtrusted* **checking** *Scala*

hide-const (**open**) *NetModel-node-props BLP-offending-list BLP-eval*

hide-const (**open**) *sinvar*

end
theory *SINVAR-SecGwExt*
imports ../*TopoS-Helper*
begin

6.3 SecurityInvariant PolEnforcePointExtended

A PolEnforcePoint is an application-level central policy enforcement point. Legacy note: The old verions called it a SecurityGateway.

Hosts may belong to a certain domain. Sometimes, a pattern where intra-domain communication between domain members must be approved by a central instance is required.

We call such a central instance PolEnforcePoint and present a template for this architecture. Five host roles are distinguished:. A PolEnforcePoint, aPolEnforcePointIN which accessible from the outside, a DomainMember, a less-restricted AccessibleMember which is accessible from the outside world, and a default value Unassigned that reflects none of these roles.

datatype *secgw-member* = *PolEnforcePoint* | *PolEnforcePointIN* | *DomainMember* | *AccessibleMember* | *Unassigned*

definition *default-node-properties* :: *secgw-member*
where *default-node-properties* ≡ *Unassigned*

fun *allowed-secgw-flow* :: *secgw-member* ⇒ *secgw-member* ⇒ *bool* **where**
allowed-secgw-flow PolEnforcePoint - = True |
allowed-secgw-flow PolEnforcePointIN - = True |
allowed-secgw-flow DomainMember DomainMember = False |
allowed-secgw-flow DomainMember - = True |
allowed-secgw-flow AccessibleMember DomainMember = False |
allowed-secgw-flow AccessibleMember - = True |
allowed-secgw-flow Unassigned Unassigned = True |
allowed-secgw-flow Unassigned PolEnforcePointIN = True |
allowed-secgw-flow Unassigned AccessibleMember = True |
allowed-secgw-flow Unassigned - = False

fun *sinvar* :: *'v graph* ⇒ (*'v* ⇒ *secgw-member*) ⇒ *bool* **where**
sinvar G nP = (∀ (*e1*,*e2*) ∈ *edges G*. *e1* ≠ *e2* ⟶ *allowed-secgw-flow* (*nP e1*) (*nP e2*))

definition *receiver-violation* :: *bool* **where** *receiver-violation* = *False*

6.3.1 Preliminaries

lemma *sinvar-mono*: *SecurityInvariant-withOffendingFlows.sinvar-mono sinvar*
apply(*simp only*: *SecurityInvariant-withOffendingFlows.sinvar-mono-def*)
apply(*clarify*)
by *auto*

interpretation *SecurityInvariant-preliminaries*
where *sinvar* = *sinvar*
apply *unfold-locales*
apply(*frule-tac finite-distinct-list*[*OF wf-graph.finiteE*])
apply(*erule-tac exE*)
apply(*rename-tac list-edges*)
apply(*rule-tac ff=list-edges* **in** *SecurityInvariant-withOffendingFlows.mono-imp-set-offending-flows-not-empty*[*OF sinvar-mono*])
apply(*auto*)[*6*]
apply(*auto simp add*: *SecurityInvariant-withOffendingFlows.is-offending-flows-def graph-ops*)[*1*]

apply(*fact SecurityInvariant-withOffendingFlows.sinvar-mono-imp-is-offending-flows-mono*[*OF sinvar-mono*])
done

6.3.2 ENF

lemma *PolEnforcePoint-ENFnr*: *SecurityInvariant-withOffendingFlows.sinvar-all-edges-normal-form-not-refl sinvar allowed-secgw-flow*
by(*simp add*: *SecurityInvariant-withOffendingFlows.sinvar-all-edges-normal-form-not-refl-def*)
lemma *Unassigned-botdefault*: $\forall$ *e1 e2*. *e2* $\neq$ *Unassigned* $\longrightarrow$ $\neg$ *allowed-secgw-flow e1 e2* $\longrightarrow$ $\neg$ *allowed-secgw-flow Unassigned e2*
apply(*rule allI*)+
apply(*case-tac e2*)
apply(*simp-all*)
apply(*case-tac e1*)
apply(*simp-all*)
apply(*case-tac e1*)
apply(*simp-all*)
done
lemma *Unassigned-not-to-Member*: $\neg$ *allowed-secgw-flow Unassigned DomainMember*
by(*simp*)
lemma *All-to-Unassigned*: $\forall$ *e1*. *allowed-secgw-flow e1 Unassigned*
by (*rule allI*, *case-tac e1*, *simp-all*)

definition *PolEnforcePointExtended-offending-set*:: *'v graph* $\Rightarrow$ (*'v* $\Rightarrow$ *secgw-member*) $\Rightarrow$ (*'v* $\times$ *'v*) *set set* **where**
PolEnforcePointExtended-offending-set G nP = (*if sinvar G nP then*
{}
else
{ {*e* $\in$ *edges G*. *case e of* (*e1*,*e2*) $\Rightarrow$ *e1* $\neq$ *e2* $\wedge$ $\neg$ *allowed-secgw-flow* (*nP e1*) (*nP e2*)} })
lemma *PolEnforcePointExtended-offending-set*: *SecurityInvariant-withOffendingFlows.set-offending-flows sinvar* = *PolEnforcePointExtended-offending-set*
apply(*simp only*: *fun-eq-iff ENFnr-offending-set*[*OF PolEnforcePoint-ENFnr*] *PolEnforcePointExtended-offending-set-def*)
apply(*rule allI*)+
apply(*rename-tac G nP*)
apply(*auto*)
done

interpretation *PolEnforcePointExtended*: *SecurityInvariant-ACS*
where *default-node-properties* = *default-node-properties*
and *sinvar* = *sinvar*
rewrites *SecurityInvariant-withOffendingFlows.set-offending-flows sinvar* = *PolEnforcePointExtended-offending-set*
unfolding *default-node-properties-def*
apply *unfold-locales*
apply(*rule ballI*)
apply (*rule SecurityInvariant-withOffendingFlows.ENFnr-fsts-weakrefl-instance*[*OF PolEnforcePoint-ENFnr Unassigned-botdefault All-to-Unassigned*])[*1*]
apply(*simp*)
apply(*simp*)
apply(*erule default-uniqueness-by-counterexample-ACS*)
apply (*simp add*: *SecurityInvariant-withOffendingFlows.set-offending-flows-def*
SecurityInvariant-withOffendingFlows.is-offending-flows-min-set-def
SecurityInvariant-withOffendingFlows.is-offending-flows-def)

apply (*simp add*:*graph-ops*)
apply (*simp split*: *prod.split-asm prod.split*)
apply(*rule-tac x*=⦇ *nodes*={*vertex-1*,*vertex-2*}, *edges* = {(*vertex-1*,*vertex-2*)} ⦈ **in** *exI*, *simp*)
apply(*rule conjI*)
apply(*simp add*: *wf-graph-def*)
apply(*case-tac otherbot*, *simp-all*)
apply(*rule-tac x*=(λ *x*. *Unassigned*)(*vertex-1* := *Unassigned*, *vertex-2* := *DomainMember*) **in** *exI*, *simp*)
apply(*rule-tac x*={(*vertex-1*,*vertex-2*)} **in** *exI*, *simp*)
apply(*rule-tac x*=(λ *x*. *Unassigned*)(*vertex-1* := *Unassigned*, *vertex-2* := *DomainMember*) **in** *exI*, *simp*)
apply(*rule-tac x*=*vertex-1* **in** *exI*, *simp*)
apply(*rule-tac x*={(*vertex-1*,*vertex-2*)} **in** *exI*, *simp*)
apply(*rule-tac x*=(λ *x*. *Unassigned*)(*vertex-1* := *Unassigned*, *vertex-2* := *PolEnforcePoint*) **in** *exI*, *simp*)
apply(*rule-tac x*=*vertex-1* **in** *exI*, *simp*)
apply(*rule-tac x*={(*vertex-1*,*vertex-2*)} **in** *exI*, *simp*)
apply(*rule-tac x*=(λ *x*. *Unassigned*)(*vertex-1* := *Unassigned*, *vertex-2* := *PolEnforcePoint*) **in** *exI*, *simp*)
apply(*rule-tac x*=*vertex-1* **in** *exI*, *simp*)
apply(*rule-tac x*={(*vertex-1*,*vertex-2*)} **in** *exI*, *simp*)

apply(*fact PolEnforcePointExtended-offending-set*)
done

lemma *TopoS-PolEnforcePointExtended*: *SecurityInvariant sinvar default-node-properties receiver-violation*
unfolding *receiver-violation-def* **by** *unfold-locales*

hide-const (**open**) *sinvar receiver-violation*

end
theory *SINVAR-SecGwExt-impl*
imports *SINVAR-SecGwExt* ../*TopoS-Interface-impl*
begin

code-identifier code-module *SINVAR-SecGwExt-impl* => (*Scala*) *SINVAR-SecGwExt*

6.3.3 SecurityInvariant PolEnforcePointExtended List Implementation

fun *sinvar* :: ′*v list-graph* ⇒ (′*v* ⇒ *SINVAR-SecGwExt.secgw-member*) ⇒ *bool* **where**
sinvar G nP = (∀ (*e1*,*e2*) ∈ *set* (*edgesL G*). *e1* ≠ *e2* ⟶ *SINVAR-SecGwExt.allowed-secgw-flow* (*nP e1*) (*nP e2*))

definition *PolEnforcePointExtended-offending-list*:: ′*v list-graph* ⇒ (′*v* ⇒ *secgw-member*) ⇒ (′*v* × ′*v*) *list list* **where**
PolEnforcePointExtended-offending-list G nP = (*if sinvar G nP then*
[]
else
[[*e* ← *edgesL G*. *case e of* (*e1*,*e2*) ⇒ *e1* ≠ *e2* ∧ ¬ *allowed-secgw-flow* (*nP e1*) (*nP e2*)]])

definition *NetModel-node-props P* = (λ *i*. (*case* (*node-properties P*) *i of Some property* ⇒ *property* | *None* ⇒ *SINVAR-SecGwExt.default-node-properties*))
lemma[*code-unfold*]: *SecurityInvariant.node-props SINVAR-SecGwExt.default-node-properties P* = *NetModel-node-props P*
apply(*simp add*: *NetModel-node-props-def*)
done

definition *PolEnforcePoint-eval G P* = (*wf-list-graph G* ∧
sinvar G (*SecurityInvariant.node-props SINVAR-SecGwExt.default-node-properties P*))

interpretation *PolEnforcePoint-impl*:*TopoS-List-Impl*
where *default-node-properties*=*SINVAR-SecGwExt.default-node-properties*
and *sinvar-spec*=*SINVAR-SecGwExt.sinvar*
and *sinvar-impl*=*sinvar*
and *receiver-violation*=*SINVAR-SecGwExt.receiver-violation*
and *offending-flows-impl*=*PolEnforcePointExtended-offending-list*
and *node-props-impl*=*NetModel-node-props*
and *eval-impl*=*PolEnforcePoint-eval*
apply(*unfold TopoS-List-Impl-def*)
apply(*rule conjI*)
apply(*simp add*: *TopoS-PolEnforcePointExtended list-graph-to-graph-def*)
apply(*rule conjI*)
apply(*simp add*: *list-graph-to-graph-def PolEnforcePointExtended-offending-set PolEnforcePointExtended-offending-set-def PolEnforcePointExtended-offending-list-def*)
apply(*rule conjI*)
apply(*simp only*: *NetModel-node-props-def*)
apply(*metis PolEnforcePointExtended.node-props.simps PolEnforcePointExtended.node-props-eq-node-props-formaldef*
apply(*simp only*: *PolEnforcePoint-eval-def*)
apply(*simp add*: *TopoS-eval-impl-proofrule*[*OF TopoS-PolEnforcePointExtended*])
apply(*simp-all add*: *list-graph-to-graph-def*)
done

6.3.4 PolEnforcePoint packing

definition *SINVAR-LIB-PolEnforcePointExtended* :: ('*v*::*vertex*, *secgw-member*) *TopoS-packed* **where**
SINVAR-LIB-PolEnforcePointExtended ≡
⦇ *nm-name* = ''*PolEnforcePointExtended*'',
nm-receiver-violation = *SINVAR-SecGwExt.receiver-violation*,
nm-default = *SINVAR-SecGwExt.default-node-properties*,
nm-sinvar = *sinvar*,
nm-offending-flows = *PolEnforcePointExtended-offending-list*,
nm-node-props = *NetModel-node-props*,
nm-eval = *PolEnforcePoint-eval*
⦈
interpretation *SINVAR-LIB-PolEnforcePointExtended-interpretation*: *TopoS-modelLibrary SINVAR-LIB-PolEnforce*

SINVAR-SecGwExt.sinvar
apply(*unfold TopoS-modelLibrary-def SINVAR-LIB-PolEnforcePointExtended-def*)
apply(*rule conjI*)
apply(*simp*)
apply(*simp*)
by(*unfold-locales*)

Examples

definition *example-net-secgw* :: *nat list-graph* **where**
example-net-secgw ≡ (| *nodesL* = [*1*::*nat*,*2*, *3*, *8*,*9*, *11*,*12*],
edgesL = [(*3*,*8*),(*8*,*3*),(*2*,*8*),(*8*,*1*),(*1*,*9*),(*9*,*2*),(*2*,*9*),(*9*,*1*), (*1*,*3*), (*8*,*11*),(*8*,*12*), (*11*,*9*), (*11*,*3*), (*11*,*12*)] |)
value *wf-list-graph example-net-secgw*

definition *example-conf-secgw* **where**
example-conf-secgw ≡ ((λ*e*. *SINVAR-SecGwExt.default-node-properties*)
(*1* := *DomainMember*, *2*:= *DomainMember*, *3*:= *AccessibleMember*,
8:= *PolEnforcePoint*, *9*:= *PolEnforcePointIN*))

export-code *sinvar* **checking** *SML*
definition *test* = *sinvar* (| *nodesL*=[*1*::*nat*], *edgesL*=[] |) (λ-. *SINVAR-SecGwExt.default-node-properties*)
export-code *test* **checking** *SML*
value *sinvar* (| *nodesL*=[*1*::*nat*], *edgesL*=[] |) (λ-. *SINVAR-SecGwExt.default-node-properties*)

value *sinvar example-net-secgw example-conf-secgw*
value *PolEnforcePoint-offending-list example-net-secgw example-conf-secgw*

definition *example-net-secgw-invalid* **where**
example-net-secgw-invalid ≡ *example-net-secgw*(|*edgesL* := (*3*,*1*)#(*11*,*1*)#(*11*,*8*)#(*1*,*2*)#(*edgesL example-net-secgw*)|)

value *sinvar example-net-secgw-invalid example-conf-secgw*
value *PolEnforcePoint-offending-list example-net-secgw-invalid example-conf-secgw*

hide-const (**open**) *NetModel-node-props*
hide-const (**open**) *sinvar*

end
theory *SINVAR-Sink*
imports ../*TopoS-Helper*
begin

6.4 SecurityInvariant Sink (IFS)

datatype *node-config* = *Sink* | *SinkPool* | *Unassigned*

definition *default-node-properties* :: *node-config*
where *default-node-properties* = *Unassigned*

fun *allowed-sink-flow* :: *node-config* ⇒ *node-config* ⇒ *bool* **where**
allowed-sink-flow Sink - = *False* |
allowed-sink-flow SinkPool SinkPool = *True* |
allowed-sink-flow SinkPool Sink = *True* |
allowed-sink-flow SinkPool - = *False* |
allowed-sink-flow Unassigned - = *True*

fun *sinvar* :: ′*v graph* ⇒ (′*v* ⇒ *node-config*) ⇒ *bool* **where**
sinvar G nP = (∀ (*e1*,*e2*) ∈ *edges G*. *e1* ≠ *e2* ⟶ *allowed-sink-flow* (*nP e1*) (*nP e2*))

definition *receiver-violation* :: *bool* **where** *receiver-violation* = *True*

6.4.1 Preliminaries

lemma *sinvar-mono*: *SecurityInvariant-withOffendingFlows.sinvar-mono sinvar*
apply(*simp only*: *SecurityInvariant-withOffendingFlows.sinvar-mono-def*)
apply(*clarify*)
by *auto*

interpretation *SecurityInvariant-preliminaries*
where *sinvar* = *sinvar*
apply *unfold-locales*
apply(*frule-tac finite-distinct-list*[*OF wf-graph.finiteE*])
apply(*erule-tac exE*)
apply(*rename-tac list-edges*)
apply(*rule-tac ff=list-edges* **in** *SecurityInvariant-withOffendingFlows.mono-imp-set-offending-flows-not-empty*[*OF sinvar-mono*])
apply(*auto*)[*6*]
apply(*auto simp add*: *SecurityInvariant-withOffendingFlows.is-offending-flows-def graph-ops*)[*1*]
apply(*fact SecurityInvariant-withOffendingFlows.sinvar-mono-imp-is-offending-flows-mono*[*OF sinvar-mono*])
done

6.4.2 ENF

lemma *Sink-ENFnr*: *SecurityInvariant-withOffendingFlows.sinvar-all-edges-normal-form-not-refl sinvar allowed-sink-flow*
by(*simp add*: *SecurityInvariant-withOffendingFlows.sinvar-all-edges-normal-form-not-refl-def*)
lemma *Unassigned-to-All*: ∀ *e2*. *allowed-sink-flow Unassigned e2*
by (*rule allI*, *case-tac e2*, *simp-all*)
lemma *Unassigned-default-candidate*: ∀ *e1 e2*. ¬ *allowed-sink-flow e1 e2* ⟶ ¬ *allowed-sink-flow e1 Unassigned*
apply(*rule allI*)+
apply(*case-tac e2*)
apply *simp-all*
apply(*case-tac e1*)
apply *simp-all*
apply(*case-tac e1*)
apply *simp-all*
done

definition *Sink-offending-set*:: *'v graph* ⇒ (*'v* ⇒ *node-config*) ⇒ (*'v* × *'v*) *set set* **where**
Sink-offending-set G nP = (*if sinvar G nP then*
{}
else
{ {*e* ∈ *edges G*. *case e of* (*e1*,*e2*) ⇒ *e1* ≠ *e2* ∧ ¬ *allowed-sink-flow* (*nP e1*) (*nP e2*)} })
lemma *Sink-offending-set*:
SecurityInvariant-withOffendingFlows.set-offending-flows sinvar = *Sink-offending-set*
apply(*simp only*: *fun-eq-iff ENFnr-offending-set*[*OF Sink-ENFnr*] *Sink-offending-set-def*)
apply(*rule allI*)+
apply(*rename-tac G nP*)
apply(*auto*)
done

interpretation *Sink*: *SecurityInvariant-IFS*
where *default-node-properties* = *default-node-properties*
and *sinvar* = *sinvar*
rewrites *SecurityInvariant-withOffendingFlows.set-offending-flows sinvar* = *Sink-offending-set*
unfolding *default-node-properties-def*
apply *unfold-locales*
apply(*rule ballI*)
apply (*rule SecurityInvariant-withOffendingFlows.ENFnr-snds-weakrefl-instance*[*OF Sink-ENFnr Unassigned-default-candidate Unassigned-to-All*])
apply(*simp-all*)[*2*]

apply(*erule default-uniqueness-by-counterexample-IFS*)
apply (*simp add*: *SecurityInvariant-withOffendingFlows.set-offending-flows-def*
SecurityInvariant-withOffendingFlows.is-offending-flows-min-set-def
SecurityInvariant-withOffendingFlows.is-offending-flows-def)
apply (*simp add*:*graph-ops*)
apply (*simp split*: *prod.split-asm prod.split*)
apply(*rule-tac x*=⦇ *nodes*={*vertex-1*,*vertex-2*}, *edges* = {(*vertex-1*,*vertex-2*)} ⦈ **in** *exI*, *simp*)
apply(*rule conjI*)
apply(*simp add*: *wf-graph-def*)
apply(*case-tac otherbot*, *simp-all*)
apply(*rule-tac x*=(λ *x*. *Unassigned*)(*vertex-1* := *SinkPool*, *vertex-2* := *Unassigned*) **in** *exI*, *simp*)
apply(*rule-tac x*=*vertex-2* **in** *exI*, *simp*)
apply(*rule-tac x*={(*vertex-1*, *vertex-2*)} **in** *exI*, *simp*)
apply(*rule-tac x*=(λ *x*. *Unassigned*)(*vertex-1* := *SinkPool*, *vertex-2* := *Unassigned*) **in** *exI*, *simp*)
apply(*rule-tac x*=*vertex-2* **in** *exI*, *simp*)
apply(*rule-tac x*={(*vertex-1*, *vertex-2*)} **in** *exI*, *simp*)

apply(*fact Sink-offending-set*)
done

lemma *TopoS-Sink*: *SecurityInvariant sinvar default-node-properties receiver-violation*
unfolding *receiver-violation-def* **by** *unfold-locales*

hide-fact (**open**) *sinvar-mono*
hide-const (**open**) *sinvar receiver-violation default-node-properties*

end
theory *SINVAR-Sink-impl*
imports *SINVAR-Sink* ../*TopoS-Interface-impl*
begin

code-identifier code-module *SINVAR-Sink-impl* => (*Scala*) *SINVAR-Sink*

6.4.3 SecurityInvariant Sink (IFS) List Implementation

fun *sinvar* :: ′*v list-graph* ⇒ (′*v* ⇒ *node-config*) ⇒ *bool* **where**
sinvar G nP = (∀ (*e1*,*e2*) ∈ *set* (*edgesL G*). *e1* ≠ *e2* ⟶ *SINVAR-Sink.allowed-sink-flow* (*nP e1*) (*nP e2*))

definition *Sink-offending-list*:: ′*v list-graph* ⇒ (′*v* ⇒ *SINVAR-Sink.node-config*) ⇒ (′*v* × ′*v*) *list list*

```
where
  Sink-offending-list G nP = (if sinvar G nP then
    []
   else
    [ [e ← edgesL G. case e of (e1,e2) ⇒ e1 ≠ e2 ∧ ¬ allowed-sink-flow (nP e1) (nP e2)] ])
```

definition *NetModel-node-props P* = (λ *i*. (*case* (*node-properties P*) *i of Some property* ⇒ *property* | *None* ⇒ *SINVAR-Sink.default-node-properties*))
lemma[*code-unfold*]: *SecurityInvariant.node-props SINVAR-Sink.default-node-properties P* = *NetModel-node-props P*
apply(*simp add*: *NetModel-node-props-def*)
done

definition *Sink-eval G P* = (*wf-list-graph G* ∧
  sinvar G (*SecurityInvariant.node-props SINVAR-Sink.default-node-properties P*))

```
interpretation Sink-impl:TopoS-List-Impl
  where default-node-properties=SINVAR-Sink.default-node-properties
  and sinvar-spec=SINVAR-Sink.sinvar
  and sinvar-impl=sinvar
  and receiver-violation=SINVAR-Sink.receiver-violation
  and offending-flows-impl=Sink-offending-list
  and node-props-impl=NetModel-node-props
  and eval-impl=Sink-eval
 apply(unfold TopoS-List-Impl-def)
 apply(rule conjI)
  apply(simp add: TopoS-Sink list-graph-to-graph-def)
 apply(rule conjI)
  apply(simp add: list-graph-to-graph-def Sink-offending-set Sink-offending-set-def Sink-offending-list-def)
 apply(rule conjI)
  apply(simp only: NetModel-node-props-def)
  apply(metis Sink.node-props.simps Sink.node-props-eq-node-props-formaldef)
 apply(simp only: Sink-eval-def)
 apply(intro allI)
 apply(rule TopoS-eval-impl-proofrule[OF TopoS-Sink])
 apply(simp-all add: list-graph-to-graph-def)
done
```

6.4.4 Sink packing

```
definition SINVAR-LIB-Sink :: ('v::vertex, node-config) TopoS-packed where
  SINVAR-LIB-Sink ≡
  (| nm-name = ''Sink'',
    nm-receiver-violation = SINVAR-Sink.receiver-violation,
    nm-default = SINVAR-Sink.default-node-properties,
    nm-sinvar = sinvar,
    nm-offending-flows = Sink-offending-list,
    nm-node-props = NetModel-node-props,
    nm-eval = Sink-eval
    |)
interpretation SINVAR-LIB-Sink-interpretation: TopoS-modelLibrary SINVAR-LIB-Sink
```

SINVAR-Sink.sinvar
apply(*unfold TopoS-modelLibrary-def SINVAR-LIB-Sink-def*)
apply(*rule conjI*)
apply(*simp*)
apply(*simp*)
by(*unfold-locales*)

Examples

definition *example-net-sink* :: *nat list-graph* **where**
example-net-sink ≡ (| *nodesL* = [*1*::*nat*,*2*,*3*, *8*, *11*,*12*],
edgesL = [(*1*,*8*),(*1*,*2*), (*2*,*8*),(*3*,*8*),(*4*,*8*), (*2*,*3*),(*3*,*2*), (*11*,*8*),(*12*,*8*), (*11*,*12*), (*1*,*12*)] |)
value *wf-list-graph example-net-sink*

definition *example-conf-sink* **where**
example-conf-sink ≡ (λ*e*. *SINVAR-Sink.default-node-properties*)(*8*:= *Sink*, *2*:= *SinkPool*, *3*:= *SinkPool*, *4*:= *SinkPool*)

value *sinvar example-net-sink example-conf-sink*
value *Sink-offending-list example-net-sink example-conf-sink*

definition *example-net-sink-invalid* **where**
example-net-sink-invalid ≡ *example-net-sink*(|*edgesL* := (*2*,*1*)#(*8*,*11*)#(*8*,*2*)#(*edgesL example-net-sink*)|)

value *sinvar example-net-sink-invalid example-conf-sink*
value *Sink-offending-list example-net-sink-invalid example-conf-sink*

hide-const (**open**) *NetModel-node-props*
hide-const (**open**) *sinvar*

end
theory *SINVAR-SubnetsInGW*
imports../*TopoS-Helper*
begin

6.5 SecurityInvariant SubnetsInGW

datatype *subnets* = *Member* | *InboundGateway* | *Unassigned*

definition *default-node-properties* :: *subnets*
where *default-node-properties* ≡ *Unassigned*

fun *allowed-subnet-flow* :: *subnets* ⇒ *subnets* ⇒ *bool* **where**
allowed-subnet-flow Member - = *True* |
allowed-subnet-flow InboundGateway - = *True* |
allowed-subnet-flow Unassigned Unassigned = *True* |
allowed-subnet-flow Unassigned InboundGateway = *True*|
allowed-subnet-flow Unassigned Member = *False*

fun *sinvar* :: *'v graph* ⇒ (*'v* ⇒ *subnets*) ⇒ *bool* **where**
sinvar G nP = (∀ (*e1*,*e2*) ∈ *edges G*. *allowed-subnet-flow* (*nP e1*) (*nP e2*))

definition *receiver-violation* :: *bool* **where** *receiver-violation* = *False*

6.5.1 Preliminaries

lemma *sinvar-mono*: *SecurityInvariant-withOffendingFlows.sinvar-mono sinvar*
apply(*simp only*: *SecurityInvariant-withOffendingFlows.sinvar-mono-def*)
apply(*clarify*)
by *auto*

interpretation *SecurityInvariant-preliminaries*
where *sinvar* = *sinvar*
apply *unfold-locales*
apply(*frule-tac finite-distinct-list*[*OF wf-graph.finiteE*])
apply(*erule-tac exE*)
apply(*rename-tac list-edges*)
apply(*rule-tac ff=list-edges* **in** *SecurityInvariant-withOffendingFlows.mono-imp-set-offending-flows-not-empty*[*OF sinvar-mono*])
apply(*auto*)[*6*]
apply(*auto simp add*: *SecurityInvariant-withOffendingFlows.is-offending-flows-def graph-ops*)[*1*]
apply(*fact SecurityInvariant-withOffendingFlows.sinvar-mono-imp-is-offending-flows-mono*[*OF sinvar-mono*])
done

6.5.2 ENF

lemma *Unassigned-not-to-Member*: ¬ *allowed-subnet-flow Unassigned Member*
by(*simp*)
lemma *All-to-Unassigned*: *allowed-subnet-flow e1 Unassigned*
by (*case-tac e1*, *simp-all*)
lemma *Member-to-All*: *allowed-subnet-flow Member e2*
by (*case-tac e2*, *simp-all*)
lemma *Unassigned-default-candidate*: ∀ *nP e1 e2*. ¬ *allowed-subnet-flow* (*nP e1*) (*nP e2*) ⟶ ¬ *allowed-subnet-flow Unassigned* (*nP e2*)
apply(*rule allI*)+
apply(*case-tac nP e2*)
apply *simp*
apply(*case-tac nP e1*)
apply(*simp-all*)[*3*]
by(*simp add*: *All-to-Unassigned*)
lemma *allowed-subnet-flow-refl*: *allowed-subnet-flow e e*
by(*case-tac e*, *simp-all*)
lemma *SubnetsInGW-ENF*: *SecurityInvariant-withOffendingFlows.sinvar-all-edges-normal-form sinvar allowed-subnet-flow*
unfolding *SecurityInvariant-withOffendingFlows.sinvar-all-edges-normal-form-def*
by *simp*
lemma *SubnetsInGW-ENF-refl*: *SecurityInvariant-withOffendingFlows.ENF-refl sinvar allowed-subnet-flow*
unfolding *SecurityInvariant-withOffendingFlows.ENF-refl-def*
apply(*rule conjI*)
apply(*simp add*: *SubnetsInGW-ENF*)
apply(*simp add*: *allowed-subnet-flow-refl*)
done

definition *SubnetsInGW-offending-set*:: *'v graph* ⇒ (*'v* ⇒ *subnets*) ⇒ (*'v* × *'v*) *set set* **where**
SubnetsInGW-offending-set G nP = (*if sinvar G nP then*
{}
else
{ {*e* ∈ *edges G*. *case e of* (*e1*,*e2*) ⇒ ¬ *allowed-subnet-flow* (*nP e1*) (*nP e2*)} })

```
  lemma SubnetsInGW-offending-set:
  SecurityInvariant-withOffendingFlows.set-offending-flows sinvar = SubnetsInGW-offending-set
   apply(simp only: fun-eq-iff ENF-offending-set[OF SubnetsInGW-ENF] SubnetsInGW-offending-set-def)
    apply(rule allI)+
    apply(rename-tac G nP)
    apply(auto)
  done

interpretation SubnetsInGW: SecurityInvariant-ACS
where default-node-properties = SINVAR-SubnetsInGW.default-node-properties
and sinvar = SINVAR-SubnetsInGW.sinvar
rewrites SecurityInvariant-withOffendingFlows.set-offending-flows sinvar = SubnetsInGW-offending-set
  unfolding SINVAR-SubnetsInGW.default-node-properties-def
  apply unfold-locales

    apply(rule ballI)
   thm SecurityInvariant-withOffendingFlows.ENF-fsts-refl-instance[OF SubnetsInGW-ENF-refl Unas-
signed-default-candidate]
   apply(rule SecurityInvariant-withOffendingFlows.ENF-fsts-refl-instance[OF SubnetsInGW-ENF-refl
Unassigned-default-candidate])
      apply(simp-all)[2]

   apply(erule default-uniqueness-by-counterexample-ACS)
   apply (simp add: SecurityInvariant-withOffendingFlows.set-offending-flows-def
      SecurityInvariant-withOffendingFlows.is-offending-flows-min-set-def
      SecurityInvariant-withOffendingFlows.is-offending-flows-def)
   apply (simp add:graph-ops)
   apply (simp split: prod.split-asm prod.split)
   apply(rule-tac x=(| nodes={vertex-1,vertex-2}, edges = {(vertex-1,vertex-2)} |) in exI, simp)
   apply(rule conjI)
    apply(simp add: wf-graph-def)
   apply(case-tac otherbot, simp-all)
   apply(rule-tac x=(λ x. Unassigned)(vertex-1 := Unassigned, vertex-2 := Member) in exI, simp)
    apply(rule-tac x={(vertex-1,vertex-2)} in exI, simp)
   apply(rule-tac x=(λ x. Unassigned)(vertex-1 := Unassigned, vertex-2 := Member) in exI, simp)
   apply(rule-tac x=vertex-1 in exI, simp)
   apply(rule-tac x={(vertex-1,vertex-2)} in exI, simp)

  apply(fact SubnetsInGW-offending-set)
  done


  lemma TopoS-SubnetsInGW: SecurityInvariant sinvar default-node-properties receiver-violation
  unfolding receiver-violation-def by unfold-locales


hide-fact (open) sinvar-mono
hide-const (open) sinvar receiver-violation default-node-properties

end
theory SINVAR-SubnetsInGW-impl
imports SINVAR-SubnetsInGW ../TopoS-Interface-impl
begin
```

code-identifier code-module *SINVAR-SubnetsInGW-impl* => (*Scala*) *SINVAR-SubnetsInGW*

6.5.3 SecurityInvariant SubnetsInGw List Implementation

fun *sinvar* :: *'v list-graph* ⇒ (*'v* ⇒ *subnets*) ⇒ *bool* **where**
sinvar G nP = (∀ (*e1*,*e2*) ∈ *set* (*edgesL G*). *SINVAR-SubnetsInGW.allowed-subnet-flow* (*nP e1*) (*nP e2*))

definition *SubnetsInGW-offending-list*:: *'v list-graph* ⇒ (*'v* ⇒ *subnets*) ⇒ (*'v* × *'v*) *list list* **where**
SubnetsInGW-offending-list G nP = (*if sinvar G nP then*
[]
else
[[*e* ← *edgesL G*. *case e of* (*e1*,*e2*) ⇒ ¬ *allowed-subnet-flow* (*nP e1*) (*nP e2*)]])

definition *NetModel-node-props P* = (λ *i*. (*case* (*node-properties P*) *i of Some property* ⇒ *property* | *None* ⇒ *SINVAR-SubnetsInGW.default-node-properties*))
lemma[*code-unfold*]: *SecurityInvariant.node-props SINVAR-SubnetsInGW.default-node-properties P* = *NetModel-node-props P*
apply(*simp add*: *NetModel-node-props-def*)
done

definition *SubnetsInGW-eval G P* = (*wf-list-graph G* ∧
sinvar G (*SecurityInvariant.node-props SINVAR-SubnetsInGW.default-node-properties P*))

interpretation *SubnetsInGW-impl*:*TopoS-List-Impl*
where *default-node-properties*=*SINVAR-SubnetsInGW.default-node-properties*
and *sinvar-spec*=*SINVAR-SubnetsInGW.sinvar*
and *sinvar-impl*=*sinvar*
and *receiver-violation*=*SINVAR-SubnetsInGW.receiver-violation*
and *offending-flows-impl*=*SubnetsInGW-offending-list*
and *node-props-impl*=*NetModel-node-props*
and *eval-impl*=*SubnetsInGW-eval*
apply(*unfold TopoS-List-Impl-def*)
apply(*rule conjI*)
apply(*simp add*: *TopoS-SubnetsInGW list-graph-to-graph-def*)
apply(*rule conjI*)
apply(*simp add*: *list-graph-to-graph-def SubnetsInGW-offending-set SubnetsInGW-offending-set-def SubnetsInGW-offending-list-def*)
apply(*rule conjI*)
apply(*simp only*: *NetModel-node-props-def*)
apply(*metis SubnetsInGW.node-props.simps SubnetsInGW.node-props-eq-node-props-formaldef*)
apply(*simp only*: *SubnetsInGW-eval-def*)
apply(*simp add*: *TopoS-eval-impl-proofrule*[*OF TopoS-SubnetsInGW*])
apply(*simp-all add*: *list-graph-to-graph-def*)
done

6.5.4 SubnetsInGW packing

definition *SINVAR-LIB-SubnetsInGW* :: (*'v*::*vertex*, *subnets*) *TopoS-packed* **where**

SINVAR-LIB-SubnetsInGW ≡
⦇ *nm-name* = *''SubnetsInGW''*,
nm-receiver-violation = *SINVAR-SubnetsInGW.receiver-violation*,
nm-default = *SINVAR-SubnetsInGW.default-node-properties*,
nm-sinvar = *sinvar*,
nm-offending-flows = *SubnetsInGW-offending-list*,
nm-node-props = *NetModel-node-props*,
nm-eval = *SubnetsInGW-eval*
⦈
interpretation *SINVAR-LIB-SubnetsInGW-interpretation*: *TopoS-modelLibrary SINVAR-LIB-SubnetsInGW SINVAR-SubnetsInGW.sinvar*
apply(*unfold TopoS-modelLibrary-def SINVAR-LIB-SubnetsInGW-def*)
apply(*rule conjI*)
apply(*simp*)
apply(*simp*)
by(*unfold-locales*)

Examples

definition *example-net-sub* :: *nat list-graph* **where**
example-net-sub ≡ ⦇ *nodesL* = [*1*::*nat*,*2*,*3*,*4*, *8*, *11*,*12*,*42*],
edgesL = [(*1*,*2*),(*1*,*3*),(*1*,*4*),(*2*,*1*),(*2*,*3*),(*2*,*4*),(*3*,*1*),(*3*,*2*),(*3*,*4*),(*4*,*1*),(*4*,*2*),(*4*,*3*),
(*8*,*1*),(*8*,*2*),
(*8*,*11*),
(*11*,*8*), (*12*,*8*),
(*11*,*42*), (*12*,*42*), (*8*,*42*)] ⦈
value *wf-list-graph example-net-sub*

definition *example-conf-sub* **where**
example-conf-sub ≡ ((λ*e*. *SINVAR-SubnetsInGW.default-node-properties*)
(*1* := *Member*, *2*:= *Member*, *3*:= *Member*, *4*:=*Member*,
8:=*InboundGateway*))

value *sinvar example-net-sub example-conf-sub*

definition *example-net-sub-invalid* **where**
example-net-sub-invalid ≡ *example-net-sub*⦇*edgesL* := (*42*,*4*)#(*edgesL example-net-sub*)⦈

value *sinvar example-net-sub-invalid example-conf-sub*
value *SubnetsInGW-offending-list example-net-sub-invalid example-conf-sub*

hide-const (**open**) *NetModel-node-props*
hide-const (**open**) *sinvar*

end
theory *SINVAR-CommunicationPartners*
imports ../*TopoS-Helper*
begin

6.6 SecurityInvariant CommunicationPartners

Idea of this securityinvariant: Only some nodes can communicate with Master nodes. It constrains who may access master nodes, Master nodes can access the world (except other prohibited master nodes). A node configured as Master has a list of nodes that can access it. Also, in order to be able to access a Master node, the sender must be denoted as a node we Care about. By default, all nodes are set to DontCare, thus they cannot access Master nodes. But they can access all other DontCare nodes and Care nodes.

TL;DR: An access control list determines who can access a master node.

```
datatype 'v node-config = DontCare | Care | Master 'v list

definition default-node-properties :: 'v node-config
  where default-node-properties = DontCare
```

Unrestricted accesses among DontCare nodes!

```
fun allowed-flow :: 'v node-config ⇒ 'v ⇒ 'v node-config ⇒ 'v ⇒ bool where
  allowed-flow DontCare - DontCare - = True |
  allowed-flow DontCare - Care - = True |
  allowed-flow DontCare - (Master -) - = False |
  allowed-flow Care - Care - = True |
  allowed-flow Care - DontCare - = True |
  allowed-flow Care s (Master M) r = (s ∈ set M) |
  allowed-flow (Master -) s (Master M) r = (s ∈ set M) |
  allowed-flow (Master -) - Care - = True |
  allowed-flow (Master -) - DontCare - = True


fun sinvar :: 'v graph ⇒ ('v ⇒ 'v node-config) ⇒ bool where
  sinvar G nP = (∀ (s,r) ∈ edges G. s ≠ r ⟶ allowed-flow (nP s) s (nP r) r)

definition receiver-violation :: bool where receiver-violation = False
```

6.6.1 Preliminaries

```
  lemma sinvar-mono: SecurityInvariant-withOffendingFlows.sinvar-mono sinvar
    apply(simp only: SecurityInvariant-withOffendingFlows.sinvar-mono-def)
    apply(clarify)
    by auto

  interpretation SecurityInvariant-preliminaries
  where sinvar = sinvar
    apply unfold-locales
      apply(frule-tac finite-distinct-list[OF wf-graph.finiteE])
      apply(erule-tac exE)
      apply(rename-tac list-edges)
    apply(rule-tac ff=list-edges in SecurityInvariant-withOffendingFlows.mono-imp-set-offending-flows-not-empty[OF
sinvar-mono])
         apply(auto)[6]
     apply(auto simp add: SecurityInvariant-withOffendingFlows.is-offending-flows-def graph-ops)[1]
   apply(fact SecurityInvariant-withOffendingFlows.sinvar-mono-imp-is-offending-flows-mono[OF sin-
var-mono])
   done
```

6.6.2 ENRnr

lemma *CommunicationPartners-ENRnrSR*: *SecurityInvariant-withOffendingFlows.sinvar-all-edges-normal-form-not-re sinvar allowed-flow*
  by(*simp add*: *SecurityInvariant-withOffendingFlows.sinvar-all-edges-normal-form-not-refl-SR-def*)
lemma *Unassigned-weakrefl*: ∀ *s r*. *allowed-flow DontCare s DontCare r*
  by(*simp*)
lemma *Unassigned-botdefault*: ∀ *s r*. (*nP r*) ≠ *DontCare* ⟶ ¬ *allowed-flow* (*nP s*) *s* (*nP r*) *r* ⟶ ¬ *allowed-flow DontCare s* (*nP r*) *r*
  apply(*rule allI*)+
  apply(*case-tac nP r*)
   apply(*simp-all*)
  apply(*case-tac nP s*)
   apply(*simp-all*)
  done
lemma ¬ *allowed-flow DontCare s* (*Master M*) *r* **by**(*simp*)
lemma ¬ *allowed-flow any s* (*Master* []) *r* **by**(*cases any*, *simp-all*)

lemma *All-to-Unassigned*: ∀ *s r*. *allowed-flow* (*nP s*) *s DontCare r*
  by (*rule allI*, *rule allI*, *case-tac nP s*, *simp-all*)
lemma *Unassigned-default-candidate*: ∀ *s r*. ¬ *allowed-flow* (*nP s*) *s* (*nP r*) *r* ⟶ ¬ *allowed-flow DontCare s* (*nP r*) *r*
  apply(*intro allI*, *rename-tac s r*)+
  apply(*case-tac nP s*)
   apply(*simp-all*)
   apply(*case-tac nP r*)
    apply(*simp-all*)
  apply(*case-tac nP r*)
   apply(*simp-all*)
  done

definition *CommunicationPartners-offending-set*:: ′*v graph* ⇒ (′*v* ⇒ ′*v node-config*) ⇒ (′*v* × ′*v*) *set set* **where**
CommunicationPartners-offending-set G nP = (*if sinvar G nP then*
  {}
 else
  { {*e* ∈ *edges G*. *case e of* (*e1*,*e2*) ⇒ *e1* ≠ *e2* ∧ ¬ *allowed-flow* (*nP e1*) *e1* (*nP e2*) *e2*} })
lemma *CommunicationPartners-offending-set*:
SecurityInvariant-withOffendingFlows.set-offending-flows sinvar = *CommunicationPartners-offending-set*
  apply(*simp only*: *fun-eq-iff ENFnrSR-offending-set*[*OF CommunicationPartners-ENRnrSR*] *CommunicationPartners-offending-set-def*)
  apply(*rule allI*)+
  apply(*rename-tac G nP*)
  apply(*auto*)
done

interpretation *CommunicationPartners*: *SecurityInvariant-ACS*
where *default-node-properties* = *default-node-properties*
and *sinvar* = *sinvar*
rewrites *SecurityInvariant-withOffendingFlows.set-offending-flows sinvar* = *CommunicationPartners-offending-set*
  unfolding *receiver-violation-def*
  unfolding *default-node-properties-def*
  apply *unfold-locales*

apply(*rule ballI*)
apply (*rule-tac f=f* **in** *SecurityInvariant-withOffendingFlows.ENFnrSR-fsts-weakrefl-instance*[*OF CommunicationPartners-ENRnrSR Unassigned-weakrefl Unassigned-botdefault All-to-Unassigned*])
apply(*simp*)
apply(*simp*)
apply(*erule default-uniqueness-by-counterexample-ACS*)
apply(*rule-tac x=*⦇ *nodes=*{*vertex-1,vertex-2*}, *edges* = {(*vertex-1,vertex-2*)} ⦈ **in** *exI*, *simp*)
apply(*rule conjI*)
apply(*simp add*: *wf-graph-def*)
apply(*simp add*: *CommunicationPartners-offending-set CommunicationPartners-offending-set-def delete-edges-simp2*)
apply(*case-tac otherbot*, *simp-all*)
apply(*rule-tac x=*(λ *x. DontCare*)(*vertex-1* := *DontCare*, *vertex-2* := *Master* [*vertex-1*]) **in** *exI*, *simp*)
apply(*rule-tac x=vertex-1* **in** *exI*, *simp*)
apply(*simp split*: *prod.split*)
apply *force*
apply(*rename-tac M*)
apply(*rule-tac x=*(λ *x. DontCare*)(*vertex-1* := *DontCare*, *vertex-2* := (*Master* (*vertex-1#M′*))) **in** *exI*, *simp*)
apply(*simp split*: *prod.split*)
apply(*clarify*)
apply *force*
apply(*fact CommunicationPartners-offending-set*)
done

lemma *TopoS-SubnetsInGW*: *SecurityInvariant sinvar default-node-properties receiver-violation*
unfolding *receiver-violation-def* **by** *unfold-locales*

Example:

lemma *sinvar* ⦇*nodes* = {*″db1″*, *″db2″*, *″h1″*, *″h2″*, *″foo″*, *″bar″*},
edges = {(*″h1″*, *″db1″*), (*″h2″*, *″db1″*), (*″h1″*, *″h2″*),
(*″db1″*, *″h1″*), (*″db1″*, *″foo″*), (*″db1″*, *″db2″*), (*″db1″*, *″db1″*),
(*″h1″*, *″foo″*), (*″foo″*, *″h1″*), (*″foo″*, *″bar″*)}⦈
(((((λ*h. default-node-properties*)(*″h1″* := *Care*))(*″h2″* := *Care*))
(*″db1″* := *Master* [*″h1″*, *″h2″*]))(*″db2″* := *Master* [*″db1″*])) **by** *eval*

hide-fact (**open**) *sinvar-mono*
hide-const (**open**) *sinvar receiver-violation default-node-properties*

end
theory *SINVAR-CommunicationPartners-impl*
imports *SINVAR-CommunicationPartners ../TopoS-Interface-impl*
begin

code-identifier code-module *SINVAR-CommunicationPartners-impl* => (*Scala*) *SINVAR-CommunicationPartners*

6.6.3 SecurityInvariant CommunicationPartners List Implementation

fun *sinvar* :: *′v list-graph* ⇒ (*′v* ⇒ *′v node-config*) ⇒ *bool* **where**
sinvar G nP = (∀ (*s,r*) ∈ *set* (*edgesL G*). *s* ≠ *r* ⟶ *SINVAR-CommunicationPartners.allowed-flow* (*nP s*) *s* (*nP r*) *r*)

```
definition CommunicationPartners-offending-list:: 'v list-graph ⇒ ('v ⇒ 'v node-config) ⇒ ('v × 'v)
list list where
  CommunicationPartners-offending-list G nP = (if sinvar G nP then
    []
   else
    [ [e ← edgesL G. case e of (e1,e2) ⇒ e1 ≠ e2 ∧ ¬ allowed-flow (nP e1) e1 (nP e2) e2] ])


thm SINVAR-CommunicationPartners.CommunicationPartners.node-props.simps
definition NetModel-node-props (P::('v::vertex, 'v node-config) TopoS-Params) =
  (λ i. (case (node-properties P) i of Some property ⇒ property | None ⇒ SINVAR-CommunicationPartners.default-node-
lemma[code-unfold]: SecurityInvariant.node-props SINVAR-CommunicationPartners.default-node-properties
P = NetModel-node-props P
apply(simp add: NetModel-node-props-def)
done

definition CommunicationPartners-eval G P = (wf-list-graph G ∧
  sinvar G (SecurityInvariant.node-props SINVAR-CommunicationPartners.default-node-properties P))


interpretation CommunicationPartners-impl:TopoS-List-Impl
  where default-node-properties=SINVAR-CommunicationPartners.default-node-properties
  and sinvar-spec=SINVAR-CommunicationPartners.sinvar
  and sinvar-impl=sinvar
  and receiver-violation=SINVAR-CommunicationPartners.receiver-violation
  and offending-flows-impl=CommunicationPartners-offending-list
  and node-props-impl=NetModel-node-props
  and eval-impl=CommunicationPartners-eval
 apply(unfold TopoS-List-Impl-def)
 apply(rule conjI)
  apply(simp add: TopoS-SubnetsInGW list-graph-to-graph-def; fail)
 apply(rule conjI)
  apply(simp add: list-graph-to-graph-def CommunicationPartners-offending-set CommunicationPart-
ners-offending-set-def CommunicationPartners-offending-list-def)
 apply(rule conjI)
  apply(simp only: NetModel-node-props-def)
  apply(metis CommunicationPartners.node-props.simps CommunicationPartners.node-props-eq-node-props-formaldef)
 apply(simp only: CommunicationPartners-eval-def)
 apply(simp add: TopoS-eval-impl-proofrule[OF TopoS-SubnetsInGW])
 apply(simp-all add: list-graph-to-graph-def)
done
```

6.6.4 CommunicationPartners packing

```
 definition SINVAR-LIB-CommunicationPartners :: ('v::vertex, 'v SINVAR-CommunicationPartners.node-config)
TopoS-packed where
   SINVAR-LIB-CommunicationPartners ≡
   (| nm-name = ''CommunicationPartners'',
     nm-receiver-violation = SINVAR-CommunicationPartners.receiver-violation,
     nm-default = SINVAR-CommunicationPartners.default-node-properties,
     nm-sinvar = sinvar,
```

nm-offending-flows = *CommunicationPartners-offending-list*,
nm-node-props = *NetModel-node-props*,
nm-eval = *CommunicationPartners-eval*
⦈

interpretation *SINVAR-LIB-CommunicationPartners-interpretation*: *TopoS-modelLibrary SINVAR-LIB-Communica*
SINVAR-CommunicationPartners.sinvar
apply(*unfold TopoS-modelLibrary-def SINVAR-LIB-CommunicationPartners-def*)
apply(*rule conjI*)
apply(*simp*)
apply(*simp*)
by(*unfold-locales*)

Examples

hide-const (**open**) *NetModel-node-props*
hide-const (**open**) *sinvar*

end
theory *SINVAR-NoRefl*
imports ../*TopoS-Helper*
begin

6.7 SecurityInvariant NoRefl

Hosts are not allowed to communicate with themselves.

This can be used to effectively lift hosts to roles. Just list all roles that are allowed to communicate with themselves. Otherwise, communication between hosts of the same role (group) is prohibited. Useful in conjunction with the security gateway.

datatype *node-config* = *NoRefl* | *Refl*

definition *default-node-properties* :: *node-config*
where *default-node-properties* = *NoRefl*

fun *sinvar* :: *'v graph* ⇒ (*'v* ⇒ *node-config*) ⇒ *bool* **where**
sinvar G nP = (∀ (*s*, *r*) ∈ *edges G*. *s* = *r* ⟶ *nP s* = *Refl*)

definition *receiver-violation* :: *bool* **where** *receiver-violation* = *False*

6.7.1 Preliminaries

lemma *sinvar-mono*: *SecurityInvariant-withOffendingFlows.sinvar-mono sinvar*
apply(*simp only*: *SecurityInvariant-withOffendingFlows.sinvar-mono-def*)
apply(*clarify*)
by *auto*

interpretation *SecurityInvariant-preliminaries*
where *sinvar* = *sinvar*
apply *unfold-locales*
apply(*frule-tac finite-distinct-list*[*OF wf-graph.finiteE*])
apply(*erule-tac exE*)
apply(*rename-tac list-edges*)

apply(*rule-tac ff=list-edges* **in** *SecurityInvariant-withOffendingFlows.mono-imp-set-offending-flows-not-empty*[*OF sinvar-mono*])
apply(*auto*)[*6*]
apply(*auto simp add*: *SecurityInvariant-withOffendingFlows.is-offending-flows-def graph-ops*)[*1*]
apply(*fact SecurityInvariant-withOffendingFlows.sinvar-mono-imp-is-offending-flows-mono*[*OF sinvar-mono*])
done

lemma *NoRefl-ENRsr*: *SecurityInvariant-withOffendingFlows.sinvar-all-edges-normal-form-sr sinvar* (λ nP_s s nP_r r. $s = r \longrightarrow nP_s$ = *Refl*)
by(*simp add*: *SecurityInvariant-withOffendingFlows.sinvar-all-edges-normal-form-sr-def*)

definition *NoRefl-offending-set*:: *'v graph* ⇒ (*'v* ⇒ *node-config*) ⇒ (*'v* × *'v*) *set set* **where**
NoRefl-offending-set G nP = (*if sinvar G nP then*
{}
else
{ {*e* ∈ *edges G*. *case e of* (*e1*,*e2*) ⇒ *e1* = *e2* ∧ *nP e1* = *NoRefl*} })

thm *SecurityInvariant-withOffendingFlows.ENFsr-offending-set*[*OF NoRefl-ENRsr*]

lemma *NoRefl-offending-set*: *SecurityInvariant-withOffendingFlows.set-offending-flows sinvar* = *NoRefl-offending-set*
apply(*simp only*: *fun-eq-iff NoRefl-offending-set-def*)
apply(*intro allI*, *rename-tac G nP*)
apply(*simp only*: *SecurityInvariant-withOffendingFlows.ENFsr-offending-set*[*OF NoRefl-ENRsr*])
apply(*case-tac sinvar G nP*)
apply(*simp*; *fail*)
apply(*simp*)
apply(*rule*)
apply(*rule*)
apply(*clarsimp*)
using *node-config.exhaust* **apply** *blast*
apply(*rule*)
apply(*rule*)
apply(*clarsimp*)
done

lemma *NoRefl-unique-default*:
∀ *G f nP i*. *wf-graph G* ∧ *f* ∈ *set-offending-flows G nP* ∧ *i* ∈ *fst* ‘ *f* ⟶ ¬ *sinvar G* (*nP*(*i* := *otherbot*)) ⟹
otherbot = *NoRefl*
apply(*erule default-uniqueness-by-counterexample-ACS*)
apply (*simp add*: *SecurityInvariant-withOffendingFlows.set-offending-flows-def*
SecurityInvariant-withOffendingFlows.is-offending-flows-min-set-def
SecurityInvariant-withOffendingFlows.is-offending-flows-def)
apply (*simp add*:*graph-ops*)
apply (*simp split*: *prod.split-asm prod.split*)
apply(*rule-tac x*=⦇ *nodes*={*vertex-1*}, *edges* = {(*vertex-1*,*vertex-1*)} ⦈ **in** *exI*, *simp*)
apply(*rule conjI*)
apply(*simp add*: *wf-graph-def*)
apply(*case-tac otherbot*, *simp-all*)
apply(*rule-tac x*=(λ *x*. *NoRefl*)(*vertex-1* := *NoRefl*, *vertex-2* := *NoRefl*) **in** *exI*, *simp*)
apply(*rule-tac x*={(*vertex-1*,*vertex-1*)} **in** *exI*, *simp*)
done

interpretation *NoRefl*: *SecurityInvariant-ACS*
where *default-node-properties* = *default-node-properties*
and *sinvar* = *sinvar*
rewrites *SecurityInvariant-withOffendingFlows.set-offending-flows sinvar* = *NoRefl-offending-set*
  unfolding *default-node-properties-def*
  apply *unfold-locales*
    apply(*rule ballI*)
    apply(*frule SINVAR-NoRefl.offending-notevalD*)
    apply(*simp only*: *SecurityInvariant-withOffendingFlows.ENFsr-offending-set*[*OF NoRefl-ENRsr*])
    apply *fastforce*
   apply(*fact NoRefl-unique-default*)
  apply(*fact NoRefl-offending-set*)
  done

It can also be interpreted as IFS

lemma *NoRefl-SecurityInvariant-IFS*: *SecurityInvariant-IFS sinvar default-node-properties*
  unfolding *default-node-properties-def*
  apply *unfold-locales*
    apply(*rule ballI*)
    apply(*frule SINVAR-NoRefl.offending-notevalD*)
    apply(*simp only*: *SecurityInvariant-withOffendingFlows.ENFsr-offending-set*[*OF NoRefl-ENRsr*])
    apply *fastforce*
   apply(*erule default-uniqueness-by-counterexample-IFS*)
   apply (*simp add*: *SecurityInvariant-withOffendingFlows.set-offending-flows-def*
      SecurityInvariant-withOffendingFlows.is-offending-flows-min-set-def
      SecurityInvariant-withOffendingFlows.is-offending-flows-def)
   apply (*simp add*:*graph-ops*)
   apply (*simp split*: *prod.split-asm prod.split*)
   apply(*rule-tac x*=⦇ *nodes*={*vertex-1*}, *edges* = {(*vertex-1*,*vertex-1*)} ⦈ **in** *exI*, *simp*)
   apply(*rule conjI*)
    apply(*simp add*: *wf-graph-def*)
   apply(*case-tac otherbot*, *simp-all*)
   apply(*rule-tac x*=(λ *x*. *NoRefl*)(*vertex-1* := *NoRefl*, *vertex-2* := *NoRefl*) **in** *exI*, *simp*)
   apply(*rule-tac x*={(*vertex-1*,*vertex-1*)} **in** *exI*, *simp*)
  done

  lemma *TopoS-NoRefl*: *SecurityInvariant sinvar default-node-properties receiver-violation*
  unfolding *receiver-violation-def* **by** *unfold-locales*

hide-fact (**open**) *sinvar-mono*
hide-const (**open**) *sinvar receiver-violation default-node-properties*

end
theory *SINVAR-NoRefl-impl*
imports *SINVAR-NoRefl* ../*TopoS-Interface-impl*
begin

code-identifier code-module *SINVAR-NoRefl-impl* => (*Scala*) *SINVAR-NoRefl*

6.7.2 SecurityInvariant NoRefl List Implementation

fun *sinvar* :: ′*v list-graph* ⇒ (′*v* ⇒ *node-config*) ⇒ *bool* **where**

sinvar G nP = (∀ (*s*,*r*) ∈ *set* (*edgesL G*). *s* = *r* ⟶ *nP s* = *Refl*)

definition *NoRefl-offending-list*:: *'v list-graph* ⇒ (*'v* ⇒ *node-config*) ⇒ (*'v* × *'v*) *list list* **where**
NoRefl-offending-list G nP = (*if sinvar G nP then*
[]
else
[[*e* ← *edgesL G*. *case e of* (*e1*,*e2*) ⇒ *e1* = *e2* ∧ *nP e1* = *NoRefl*]])

definition *NetModel-node-props P* = (λ *i*. (*case* (*node-properties P*) *i of Some property* ⇒ *property* | *None* ⇒ *SINVAR-NoRefl.default-node-properties*))
lemma[*code-unfold*]: *SecurityInvariant.node-props SINVAR-NoRefl.default-node-properties P* = *NetModel-node-props P*
apply(*simp add*: *NetModel-node-props-def*)
done

definition *NoRefl-eval G P* = (*wf-list-graph G* ∧
sinvar G (*SecurityInvariant.node-props SINVAR-NoRefl.default-node-properties P*))

interpretation *NoRefl-impl*:*TopoS-List-Impl*
where *default-node-properties*=*SINVAR-NoRefl.default-node-properties*
and *sinvar-spec*=*SINVAR-NoRefl.sinvar*
and *sinvar-impl*=*sinvar*
and *receiver-violation*=*SINVAR-NoRefl.receiver-violation*
and *offending-flows-impl*=*NoRefl-offending-list*
and *node-props-impl*=*NetModel-node-props*
and *eval-impl*=*NoRefl-eval*
apply(*unfold TopoS-List-Impl-def*)
apply(*rule conjI*)
apply(*simp add*: *TopoS-NoRefl list-graph-to-graph-def*)
apply(*rule conjI*)
apply(*simp add*: *list-graph-to-graph-def NoRefl-offending-set NoRefl-offending-set-def NoRefl-offending-list-def*)
apply(*rule conjI*)
apply(*simp only*: *NetModel-node-props-def*)
apply(*metis NoRefl.node-props.simps NoRefl.node-props-eq-node-props-formaldef*)
apply(*simp only*: *NoRefl-eval-def*)
apply(*simp add*: *TopoS-eval-impl-proofrule*[*OF TopoS-NoRefl*])
apply(*simp add*: *list-graph-to-graph-def*)
done

6.7.3 PolEnforcePoint packing

definition *SINVAR-LIB-NoRefl* :: (*'v*::*vertex*, *node-config*) *TopoS-packed* **where**
SINVAR-LIB-NoRefl ≡
⦇ *nm-name* = *''NoRefl''*,
nm-receiver-violation = *SINVAR-NoRefl.receiver-violation*,
nm-default = *SINVAR-NoRefl.default-node-properties*,
nm-sinvar = *sinvar*,
nm-offending-flows = *NoRefl-offending-list*,
nm-node-props = *NetModel-node-props*,
nm-eval = *NoRefl-eval*

⦈
interpretation *SINVAR-LIB-NoRefl-interpretation*: *TopoS-modelLibrary SINVAR-LIB-NoRefl SINVAR-NoRefl.sinvar*
apply(*unfold TopoS-modelLibrary-def SINVAR-LIB-NoRefl-def*)
apply(*rule conjI*)
apply(*simp*)
apply(*simp*)
by(*unfold-locales*)

Examples

definition *example-net* :: *nat list-graph* **where**
example-net ≡ ⦇ *nodesL* = [*1*::*nat*,*2*,*3*],
edgesL = [(*1*,*2*),(*2*,*2*),(*2*,*1*),(*1*,*3*)] ⦈
lemma *wf-list-graph example-net* **by** *eval*

definition *example-conf* **where**
example-conf ≡ ((λ*e*. *SINVAR-NoRefl.default-node-properties*)(*2*:= *Refl*))

lemma *sinvar example-net example-conf* **by** *eval*
lemma *NoRefl-offending-list example-net* (λ*e*. *SINVAR-NoRefl.default-node-properties*) = [[(*2*, *2*)]]
by *eval*

hide-const (**open**) *NetModel-node-props*
hide-const (**open**) *sinvar*

end
theory *SINVAR-Tainting-impl*
imports *SINVAR-Tainting* ../*TopoS-Interface-impl*
begin

6.7.4 SecurityInvariant Tainting List Implementation

code-identifier code-module *SINVAR-Tainting-impl* => (*Scala*) *SINVAR-Tainting*

fun *sinvar* :: *'v list-graph* ⇒ (*'v* ⇒ *SINVAR-Tainting.taints*) ⇒ *bool* **where**
sinvar G nP = (∀ (*e1*,*e2*) ∈ *set* (*edgesL G*). (*nP e1*) ⊆ (*nP e2*))

definition *Tainting-offending-list*:: *'v list-graph* ⇒ (*'v* ⇒ *SINVAR-Tainting.taints*) ⇒ (*'v* × *'v*) *list list* **where**
Tainting-offending-list G nP = (*if sinvar G nP then*
[]
else
[[*e* ← *edgesL G*. *case e of* (*e1*,*e2*) ⇒ ¬(*nP e1*) ⊆ (*nP e2*)]])

definition *NetModel-node-props P* =
(λ *i*. (*case* (*node-properties P*) *i of*
Some property ⇒ *property*
| *None* ⇒ *SINVAR-Tainting.default-node-properties*))
lemma[*code-unfold*]: *SecurityInvariant.node-props SINVAR-Tainting.default-node-properties P* = *NetModel-node-props P*
by(*simp add*: *NetModel-node-props-def SecurityInvariant.node-props.simps*[*OF TopoS-Tainting*])

```
definition Tainting-eval G P = (wf-list-graph G ∧
  sinvar G (SecurityInvariant.node-props SINVAR-Tainting.default-node-properties P))


interpretation Tainting-impl:TopoS-List-Impl
  where default-node-properties=SINVAR-Tainting.default-node-properties
  and sinvar-spec=SINVAR-Tainting.sinvar
  and sinvar-impl=sinvar
  and receiver-violation=SINVAR-Tainting.receiver-violation
  and offending-flows-impl=Tainting-offending-list
  and node-props-impl=NetModel-node-props
  and eval-impl=Tainting-eval
  apply(unfold TopoS-List-Impl-def)
  apply(rule conjI)
   apply(simp add: TopoS-Tainting)
   apply(simp add: list-graph-to-graph-def SINVAR-Tainting.sinvar-def; fail)
  apply(rule conjI)
   apply(simp add: list-graph-to-graph-def)
   apply(simp add: list-graph-to-graph-def SINVAR-Tainting.sinvar-def Taints-offending-set
                SINVAR-Tainting.Taints-offending-set-def Tainting-offending-list-def; fail)
  apply(rule conjI)
   apply(simp only: NetModel-node-props-def)

  apply (metis SecurityInvariant.node-props.simps SecurityInvariant.node-props-eq-node-props-formaldef
TopoS-Tainting)
  apply(simp only: Tainting-eval-def)
  apply(simp add: TopoS-eval-impl-proofrule[OF TopoS-Tainting])
  apply(simp add: list-graph-to-graph-def SINVAR-Tainting.sinvar-def)
 done
```

6.7.5 Tainting packing

```
  definition SINVAR-LIB-Tainting :: ('v::vertex, SINVAR-Tainting.taints) TopoS-packed where
    SINVAR-LIB-Tainting ≡
    (| nm-name = ''Tainting'',
      nm-receiver-violation = SINVAR-Tainting.receiver-violation,
      nm-default = SINVAR-Tainting.default-node-properties,
      nm-sinvar = sinvar,
      nm-offending-flows = Tainting-offending-list,
      nm-node-props = NetModel-node-props,
      nm-eval = Tainting-eval
      |)
  interpretation SINVAR-LIB-BLPbasic-interpretation: TopoS-modelLibrary SINVAR-LIB-Tainting
      SINVAR-Tainting.sinvar
    apply(unfold TopoS-modelLibrary-def SINVAR-LIB-Tainting-def)
    apply(rule conjI)
     apply(simp)
    apply(simp)
    by(unfold-locales)
```

6.7.6 Example

context
begin
private definition *tainting-example* :: *string list-graph* **where**
tainting-example ≡ (| *nodesL* = [''*produce 1*'',
''*produce 2*'',
''*produce 3*'',
''*read 1 2*'',
''*read 3*'',
''*consume 1 2 3*'',
''*consume 3*''],
edgesL =[(''*produce 1*'', ''*read 1 2*''),
(''*produce 2*'', ''*read 1 2*''),
(''*produce 3*'', ''*read 3*''),
(''*read 3*'', ''*read 1 2*''),
(''*read 1 2*'', ''*consume 1 2 3*''),
(''*read 3*'', ''*consume 3*'')] |)
lemma *wf-list-graph tainting-example* **by** *eval*

private definition *tainting-example-props* :: *string* ⇒ *SINVAR-Tainting.taints* **where**
tainting-example-props ≡ (λ *n. SINVAR-Tainting.default-node-properties*)
(''*produce 1*'' := {''*1*''},
''*produce 2*'' := {''*2*''},
''*produce 3*'' := {''*3*''},
''*read 1 2*'' := {''*1*'',''*2*'', ''*3*''},
''*read 3*'' := {''*3*''},
''*consume 1 2 3*'' := {''*1*'',''*2*'',''*3*''},
''*consume 3*'' := {''*3*''})
private lemma *sinvar tainting-example tainting-example-props* **by** *eval*
end

export-code *SINVAR-LIB-Tainting* **checking** *Scala*

hide-const (**open**) *NetModel-node-props Tainting-offending-list Tainting-eval*

hide-const (**open**) *sinvar*

end
theory *SINVAR-TaintingTrusted-impl*
imports *SINVAR-TaintingTrusted ../TopoS-Interface-impl*
begin

6.7.7 SecurityInvariant Tainting with Trust List Implementation

code-identifier code-module *SINVAR-Tainting-impl* => (*Scala*) *SINVAR-Tainting*

lemma $A - B \subseteq C \longleftrightarrow (\forall a{\in}A.\ a \in C \lor a \in B)$ **by** *blast*
lemma $\neg(A - B \subseteq C) \longleftrightarrow (\exists\, a \in A.\ a \notin C \land a \notin B)$ **by** *blast*

fun *sinvar* :: *'v list-graph* ⇒ (*'v* ⇒ *SINVAR-TaintingTrusted.taints*) ⇒ *bool* **where**
sinvar G nP = (∀ (*v1*,*v2*) ∈ *set* (*edgesL G*). *taints* (*nP v1*) − *untaints* (*nP v1*) ⊆ *taints* (*nP v2*))

export-code *sinvar* **checking** *SML*
value[*code*] *sinvar* (| *nodesL* = [], *edgesL* =[] |) (λ-. *SINVAR-TaintingTrusted.default-node-properties*)
lemma *sinvar* (| *nodesL* = [], *edgesL* =[] |) (λ-. *SINVAR-TaintingTrusted.default-node-properties*) **by** *eval*

definition *TaintingTrusted-offending-list*
:: *'v list-graph* ⇒ (*'v* ⇒ *SINVAR-TaintingTrusted.taints*) ⇒ (*'v* × *'v*) *list list* **where**
TaintingTrusted-offending-list G nP = (*if sinvar G nP then*
[]
else
[[*e* ← *edgesL G. case e of* (*v1*,*v2*) ⇒ ¬(*taints* (*nP v1*) − *untaints* (*nP v1*) ⊆ *taints* (*nP v2*))]])

export-code *TaintingTrusted-offending-list* **checking** *SML*

definition *NetModel-node-props P* =
(λ *i*. (*case* (*node-properties P*) *i of*
Some property ⇒ *property*
| *None* ⇒ *SINVAR-TaintingTrusted.default-node-properties*))
lemma[*code-unfold*]: *SecurityInvariant.node-props SINVAR-TaintingTrusted.default-node-properties P* = *NetModel-node-props P*
by(*simp add*: *NetModel-node-props-def SecurityInvariant.node-props.simps*[*OF TopoS-TaintingTrusted*])

definition *TaintingTrusted-eval G P* = (*wf-list-graph G* ∧
sinvar G (*SecurityInvariant.node-props SINVAR-TaintingTrusted.default-node-properties P*))

interpretation *TaintingTrusted-impl*:*TopoS-List-Impl*
where *default-node-properties*=*SINVAR-TaintingTrusted.default-node-properties*
and *sinvar-spec*=*SINVAR-TaintingTrusted.sinvar*
and *sinvar-impl*=*sinvar*
and *receiver-violation*=*SINVAR-TaintingTrusted.receiver-violation*
and *offending-flows-impl*=*TaintingTrusted-offending-list*
and *node-props-impl*=*NetModel-node-props*
and *eval-impl*=*TaintingTrusted-eval*
apply(*unfold TopoS-List-Impl-def*)
apply(*rule conjI*)
apply(*simp add*: *TopoS-TaintingTrusted*)
apply(*simp add*: *list-graph-to-graph-def SINVAR-TaintingTrusted.sinvar-def*; *fail*)
apply(*rule conjI*)
apply(*simp add*: *list-graph-to-graph-def*)
apply(*simp add*: *list-graph-to-graph-def SINVAR-TaintingTrusted.sinvar-def Taints-offending-set SINVAR-TaintingTrusted.Taints-offending-set-def TaintingTrusted-offending-list-def*; *fail*)
apply(*rule conjI*)
apply(*simp only*: *NetModel-node-props-def*)

apply (*metis SecurityInvariant.node-props.simps SecurityInvariant.node-props-eq-node-props-formaldef TopoS-TaintingTrusted*)

apply(*simp only*: *TaintingTrusted-eval-def*)
apply(*simp add*: *TopoS-eval-impl-proofrule*[*OF TopoS-TaintingTrusted*])
apply(*simp add*: *list-graph-to-graph-def SINVAR-TaintingTrusted.sinvar-def*; *fail*)
done

6.7.8 TaintingTrusted packing

definition *SINVAR-LIB-TaintingTrusted* :: (*'v*::*vertex*, *SINVAR-TaintingTrusted.taints*) *TopoS-packed*
where
SINVAR-LIB-TaintingTrusted ≡
(| *nm-name* = *''TaintingTrusted''*,
nm-receiver-violation = *SINVAR-TaintingTrusted.receiver-violation*,
nm-default = *SINVAR-TaintingTrusted.default-node-properties*,
nm-sinvar = *sinvar*,
nm-offending-flows = *TaintingTrusted-offending-list*,
nm-node-props = *NetModel-node-props*,
nm-eval = *TaintingTrusted-eval*
|)
interpretation *SINVAR-LIB-BLPbasic-interpretation*: *TopoS-modelLibrary SINVAR-LIB-TaintingTrusted SINVAR-TaintingTrusted.sinvar*
apply(*unfold TopoS-modelLibrary-def SINVAR-LIB-TaintingTrusted-def*)
apply(*rule conjI*)
apply(*simp*)
apply(*simp*)
by(*unfold-locales*)

6.7.9 Example

context
begin
private definition *tainting-example* :: *string list-graph* **where**
tainting-example ≡ (| *nodesL* = [*''produce 1''*,
''produce 2'',
''produce 3'',
''read 1 2'',
''read 3'',
''consume 1 2 3'',
''consume 3''],
edgesL =[(*''produce 1''*, *''read 1 2''*),
(*''produce 2''*, *''read 1 2''*),
(*''produce 3''*, *''read 3''*),
(*''read 3''*, *''read 1 2''*),
(*''read 1 2''*, *''consume 1 2 3''*),
(*''read 3''*, *''consume 3''*)] |)
lemma *wf-list-graph tainting-example* **by** *eval*

private definition *tainting-example-props* :: *string* ⇒ *SINVAR-TaintingTrusted.taints* **where**
tainting-example-props ≡ (λ *n*. *SINVAR-TaintingTrusted.default-node-properties*)
(*''produce 1''* := *TaintsUntaints* {*''1''*} {},
''produce 2'' := *TaintsUntaints* {*''2''*} {},
''produce 3'' := *TaintsUntaints* {*''3''*} {},
''read 1 2'' := *TaintsUntaints* {*''3''*,*''foo''*} {*''1''*,*''2''*},
''read 3'' := *TaintsUntaints* {*''3''*} {},

''consume 1 2 3'' := TaintsUntaints {''foo'',''3''} {},
''consume 3'' := TaintsUntaints {''3''} {})

value *tainting-example-props* (''consume 1 2 3'')
value[*code*] *TaintingTrusted-offending-list tainting-example tainting-example-props*
private lemma *sinvar tainting-example tainting-example-props* **by** *eval*
end

export-code *SINVAR-LIB-TaintingTrusted* **checking** *Scala*
export-code *SINVAR-LIB-TaintingTrusted* **checking** *SML*

hide-const (**open**) *NetModel-node-props TaintingTrusted-offending-list TaintingTrusted-eval*

hide-const (**open**) *sinvar*

end
theory *SINVAR-Dependability*
imports ../*TopoS-Helper*
begin

6.8 SecurityInvariant Dependability

type-synonym *dependability-level* = *nat*

definition *default-node-properties* :: *dependability-level*
where *default-node-properties* ≡ *0*

Less-equal other nodes depend on the output of a node than its dependability level.

fun *sinvar* :: 'v graph ⇒ ('v ⇒ dependability-level) ⇒ bool **where**
sinvar G nP = (∀ (e1,e2) ∈ edges G. (num-reachable G e1) ≤ (nP e1))

definition *receiver-violation* :: *bool* **where**
receiver-violation ≡ *False*

It does not matter whether we iterate over all edges or all nodes. We chose all edges because it is in line with the other models.

fun *sinvar-nodes* :: 'v graph ⇒ ('v ⇒ dependability-level) ⇒ bool **where**
sinvar-nodes G nP = (∀ v ∈ nodes G. (num-reachable G v) ≤ (nP v))

theorem *sinvar-edges-nodes-iff*: *wf-graph G* ⟹
sinvar-nodes G nP = *sinvar G nP*
proof(*unfold sinvar-nodes.simps sinvar.simps*, *rule iffI*)
assume *a1*: *wf-graph G*
and *a2*: ∀ v∈nodes G. num-reachable G v ≤ nP v

from *a1*[*simplified wf-graph-def*] **have** *f1*: *fst ‘ edges G* ⊆ *nodes G* **by** *simp*
from *f1 a2* **have** ∀ v ∈ (fst ‘ edges G). num-reachable G v ≤ nP v **by** *auto*

thus ∀ (e1, -) ∈ edges G. num-reachable G e1 ≤ nP e1 **by** *auto*
next
assume *a1*: *wf-graph G*
and *a2*: ∀ (e1, -)∈edges G. num-reachable G e1 ≤ nP e1

from *a2* **have** *g1*: ∀ *v* ∈ (*fst* ' *edges G*). *num-reachable G v* ≤ *nP v* **by** *auto*

from *FiniteGraph.succ-tran-empty*[*OF a1*] *num-reachable-zero-iff*[*OF a1*, *symmetric*]
have *g2*: ∀ *v*. *v* ∉ (*fst* ' *edges G*) ⟶ *num-reachable G v* ≤ *nP v* **by** (*metis le0*)

from *g1 g2* **show** ∀ *v*∈*nodes G*. *num-reachable G v* ≤ *nP v* **by** *metis*
qed

lemma *num-reachable-le-nodes*: ⟦ *wf-graph G* ⟧ ⟹ *num-reachable G v* ≤ *card* (*nodes G*)
unfolding *num-reachable-def*
using *succ-tran-subseteq-nodes card-seteq nat-le-linear wf-graph.finiteV* **by** *metis*

nP is valid if all dependability level are greater equal the total number of nodes in the graph

lemma ⟦ *wf-graph G*; ∀ *v* ∈ *nodes G*. *nP v* ≥ *card* (*nodes G*) ⟧ ⟹ *sinvar G nP*
apply(*subst sinvar-edges-nodes-iff*[*symmetric*], *simp*)
apply(*simp add*:)
using *num-reachable-le-nodes* **by** (*metis le-trans*)

Generate a valid configuration to start from:

Takes arbitrary configuration, returns a valid one

fun *dependability-fix-nP* :: *'v graph* ⇒ (*'v* ⇒ *dependability-level*) ⇒ (*'v* ⇒ *dependability-level*) **where**
dependability-fix-nP G nP = (*λv*. *if num-reachable G v* ≤ (*nP v*) *then* (*nP v*) *else num-reachable G v*)

dependability-fix-nP always gives you a valid solution

lemma *dependability-fix-nP-valid*: ⟦ *wf-graph G* ⟧ ⟹ *sinvar G* (*dependability-fix-nP G nP*)
by(*subst sinvar-edges-nodes-iff*[*symmetric*], *simp-all*)

furthermore, it gives you a minimal solution, i.e. if someone supplies a configuration with a value lower than calculated by *dependability-fix-nP*, this is invalid!

lemma *dependability-fix-nP-minimal-solution*: ⟦ *wf-graph G*; ∃ *v* ∈ *nodes G*. (*nP v*) < (*dependability-fix-nP G* (*λ-*. *0*)) *v* ⟧ ⟹ ¬ *sinvar G nP*
apply(*subst sinvar-edges-nodes-iff*[*symmetric*], *simp*)
apply(*simp*)
apply(*clarify*)
apply(*rule-tac x*=*v* **in** *bexI*)
apply(*simp-all*)
done

lemma *sinvar-mono*: *SecurityInvariant-withOffendingFlows.sinvar-mono sinvar*
apply(*rule-tac SecurityInvariant-withOffendingFlows.sinvar-mono-I-proofrule*)
apply(*auto*)
apply(*rename-tac nP e1 e2 N E' e1' e2' E*)
apply(*drule-tac E'*=*E'* **and** *v*=*e1'* **in** *num-reachable-mono*)
apply *simp*
apply(*subgoal-tac* (*e1'*, *e2'*) ∈ *E*)

apply(*force*)
apply(*blast*)
done

interpretation *SecurityInvariant-preliminaries*
where *sinvar = sinvar*
apply *unfold-locales*
apply(*frule-tac finite-distinct-list*[*OF wf-graph.finiteE*])
apply(*erule-tac exE*)
apply(*rename-tac list-edges*)
apply(*rule-tac ff=list-edges* **in** *SecurityInvariant-withOffendingFlows.mono-imp-set-offending-flows-not-empty*[*OF sinvar-mono*])
apply(*auto*)[*4*]
apply(*auto simp add*: *SecurityInvariant-withOffendingFlows.is-offending-flows-def graph-ops*)[*1*]
apply(*fact SecurityInvariant-withOffendingFlows.sinvar-mono-imp-sinvar-mono*[*OF sinvar-mono*])
apply(*fact SecurityInvariant-withOffendingFlows.sinvar-mono-imp-is-offending-flows-mono*[*OF sinvar-mono*])
done

interpretation *Dependability*: *SecurityInvariant-ACS*
where *default-node-properties = SINVAR-Dependability.default-node-properties*
and *sinvar = SINVAR-Dependability.sinvar*
unfolding *SINVAR-Dependability.default-node-properties-def*
proof
fix *G*::*'a graph* **and** *f nP*
assume *wf-graph G* **and** *f ∈ set-offending-flows G nP*
thus ∀ *i*∈*fst* ‘ *f*. ¬ *sinvar G* (*nP*(*i* := *0*))
apply (*simp add*: *SecurityInvariant-withOffendingFlows.set-offending-flows-def*
SecurityInvariant-withOffendingFlows.is-offending-flows-min-set-def
SecurityInvariant-withOffendingFlows.is-offending-flows-def)
apply (*simp split*: *prod.split-asm prod.split*)
apply (*simp add*:*graph-ops*)
apply(*clarify*)
apply (*metis gr0I le0*)
done
next
fix *otherbot*
assume *assm*: ∀ *G f nP i*. *wf-graph G* ∧ *f* ∈ *set-offending-flows G nP* ∧ *i* ∈ *fst* ‘ *f* ⟶ ¬ *sinvar G* (*nP*(*i* := *otherbot*))
have *unique-default-example-succ-tran*:
succ-tran (|*nodes* = {*vertex-1*, *vertex-2*}, *edges* = {(*vertex-1*, *vertex-2*)}|) *vertex-1* = {*vertex-2*}
using *unique-default-example1* **by** *blast*
from *assm* **show** *otherbot* = *0*
apply −
apply(*elim default-uniqueness-by-counterexample-ACS*)
apply(*simp*)
apply (*simp add*: *SecurityInvariant-withOffendingFlows.set-offending-flows-def*
SecurityInvariant-withOffendingFlows.is-offending-flows-min-set-def
SecurityInvariant-withOffendingFlows.is-offending-flows-def)
apply (*simp add*:*graph-ops*)
apply (*simp split*: *prod.split-asm prod.split*)
apply(*rule-tac x*=(| *nodes*={*vertex-1*,*vertex-2*}, *edges* = {(*vertex-1*,*vertex-2*)} |) **in** *exI*, *simp*)

apply(*rule conjI*)
 apply(*simp add*: *wf-graph-def*)
apply(*rule-tac x*=(λ *x*. *0*)(*vertex-1* := *0*, *vertex-2* := *0*) **in** *exI*, *simp*)
apply(*rule conjI*)
 apply(*simp add*: *unique-default-example-succ-tran num-reachable-def*)
apply(*rule-tac x*=*vertex-1* **in** *exI*, *simp*)
apply(*rule-tac x*={(*vertex-1*,*vertex-2*)} **in** *exI*, *simp*)
apply(*simp add*: *unique-default-example-succ-tran num-reachable-def*)
apply(*simp add*: *succ-tran-def unique-default-example-simp1 unique-default-example-simp2*)
done
qed

lemma *TopoS-Dependability*: *SecurityInvariant sinvar default-node-properties receiver-violation*
unfolding *receiver-violation-def* **by** *unfold-locales*

hide-const (**open**) *sinvar receiver-violation default-node-properties*

end
theory *SINVAR-Dependability-impl*
imports *SINVAR-Dependability* ../*TopoS-Interface-impl*
begin

code-identifier code-module *SINVAR-Dependability-impl* => (*Scala*) *SINVAR-Dependability*

6.8.1 SecurityInvariant Dependability List Implementation

Less-equal other nodes depend on the output of a node than its dependability level.

fun *sinvar* :: *'v list-graph* $\Rightarrow$ (*'v* $\Rightarrow$ *dependability-level*) $\Rightarrow$ *bool* **where**
sinvar G nP = ($\forall$ (*e1*,*e2*) $\in$ *set* (*edgesL G*). (*num-reachable G e1*) $\leq$ (*nP e1*))

value *sinvar*
(| *nodesL* = [*1*::*nat*,*2*,*3*,*4*], *edgesL* = [(*1*,*2*), (*2*,*3*), (*3*,*4*), (*8*,*9*),(*9*,*8*)] |)
(λ*e*. *3*)
value *sinvar*
(| *nodesL* = [*1*::*nat*,*2*,*3*,*4*,*8*,*9*,*10*], *edgesL* = [(*1*,*2*), (*2*,*3*), (*3*,*4*), (*8*,*9*),(*9*,*8*)] |)
(λ*e*. *2*)

Generate a valid configuration to start from:

fun *dependability-fix-nP* :: *'v list-graph* $\Rightarrow$ (*'v* $\Rightarrow$ *dependability-level*) $\Rightarrow$ (*'v* $\Rightarrow$ *dependability-level*) **where**
dependability-fix-nP G nP = (λ*v*. *let nr* = *num-reachable G v in* (*if nr* $\leq$ (*nP v*) *then* (*nP v*) *else nr*))

theorem *dependability-fix-nP-impl-correct*: *wf-list-graph G* $\Longrightarrow$ *dependability-fix-nP G nP* = *SINVAR-Dependability.dependability-fix-nP* (*list-graph-to-graph G*) *nP*
by(*simp add*: *num-reachable-correct fun-eq-iff*)

value *let G* = (| *nodesL* = [*1*::*nat*,*2*,*3*,*4*], *edgesL* = [(*1*,*1*), (*2*,*1*), (*3*,*1*), (*4*,*1*), (*1*,*2*), (*1*,*3*)] |) *in* (*let nP* = *dependability-fix-nP G* (λ*e*. *0*) *in map* (λ*v*. *nP v*) (*nodesL G*))

value *let G = (| nodesL = [1::nat,2,3,4], edgesL = [(1,1)] |) in (let nP = dependability-fix-nP G (λe. 0) in map (λv. nP v) (nodesL G))*

definition *Dependability-offending-list*:: *'v list-graph ⇒ ('v ⇒ dependability-level) ⇒ ('v × 'v) list list*
where
Dependability-offending-list = Generic-offending-list sinvar

definition *NetModel-node-props P = (λ i. (case (node-properties P) i of Some property ⇒ property | None ⇒ SINVAR-Dependability.default-node-properties))*
lemma[*code-unfold*]: *SecurityInvariant.node-props SINVAR-Dependability.default-node-properties P = NetModel-node-props P*
apply(*simp add*: *NetModel-node-props-def*)
done

definition *Dependability-eval G P = (wf-list-graph G ∧*
sinvar G (SecurityInvariant.node-props SINVAR-Dependability.default-node-properties P))

lemma *sinvar-correct*: *wf-list-graph G ⟹ SINVAR-Dependability.sinvar (list-graph-to-graph G) nP = sinvar G nP*

```
   apply(simp)
   apply(rule all-edges-list-I)
   apply(simp add: fun-eq-iff)
   apply(clarify)
   apply(rename-tac x)
   apply(drule-tac v=x in  num-reachable-correct)
  apply presburger
done
```

interpretation *Dependability-impl*:*TopoS-List-Impl*

```
  where default-node-properties=SINVAR-Dependability.default-node-properties
  and sinvar-spec=SINVAR-Dependability.sinvar
  and sinvar-impl=sinvar
  and receiver-violation=SINVAR-Dependability.receiver-violation
  and offending-flows-impl=Dependability-offending-list
  and node-props-impl=NetModel-node-props
  and eval-impl=Dependability-eval
 apply(unfold TopoS-List-Impl-def)
 apply(rule conjI)
  apply(rule conjI)
   apply(simp add: TopoS-Dependability; fail)
  apply(intro allI impI)
  apply(fact sinvar-correct)
 apply(rule conjI)
  apply(unfold Dependability-offending-list-def)
  apply(intro allI impI)
  apply(rule Generic-offending-list-correct)
```

apply(*assumption*)
apply(*simp only*: *sinvar-correct*)
apply(*rule conjI*)
apply(*intro allI*)
apply(*simp only*: *NetModel-node-props-def*)
apply(*metis Dependability.node-props.simps Dependability.node-props-eq-node-props-formaldef*)
apply(*simp only*: *Dependability-eval-def*)
apply(*intro allI impI*)
apply(*rule TopoS-eval-impl-proofrule*[*OF TopoS-Dependability*])
apply(*simp only*: *sinvar-correct*)
done

6.8.2 Dependability packing

definition *SINVAR-LIB-Dependability* :: ('*v*::*vertex*, *SINVAR-Dependability.dependability-level*) *TopoS-packed*
where
SINVAR-LIB-Dependability ≡
(| *nm-name* = ''*Dependability*'',
nm-receiver-violation = *SINVAR-Dependability.receiver-violation*,
nm-default = *SINVAR-Dependability.default-node-properties*,
nm-sinvar = *sinvar*,
nm-offending-flows = *Dependability-offending-list*,
nm-node-props = *NetModel-node-props*,
nm-eval = *Dependability-eval*
|)
interpretation *SINVAR-LIB-Dependability-interpretation*: *TopoS-modelLibrary SINVAR-LIB-Dependability SINVAR-Dependability.sinvar*
apply(*unfold TopoS-modelLibrary-def SINVAR-LIB-Dependability-def*)
apply(*rule conjI*)
apply(*simp*)
apply(*simp*)
by(*unfold-locales*)

Example:

value *let G* = (| *nodesL* = [*1*::*nat*,*2*,*3*,*4*,*8*,*9*,*10*], *edgesL* = [(*1*,*2*), (*2*,*3*), (*3*,*4*), (*8*,*9*),(*9*,*8*)] |)
in sinvar G ((λ *n. SINVAR-Dependability.default-node-properties*)(*1*:=*3*, *2*:=*2*, *3*:=*1*, *4*:=*0*, *8*:=*2*, *9*:=*2*, *10*:=*0*))

value *let G* = (| *nodesL* = [*1*::*nat*,*2*,*3*,*4*,*8*,*9*,*10*], *edgesL* = [(*1*,*2*), (*2*,*3*), (*3*,*4*), (*8*,*9*),(*9*,*8*)] |)
in sinvar G ((λ *n. SINVAR-Dependability.default-node-properties*)(*1*:=*10*, *2*:=*10*, *3*:=*10*, *4*:=*10*, *8*:=*10*, *9*:=*10*, *10*:=*10*))

value *let G* = (| *nodesL* = [*1*::*nat*,*2*,*3*,*4*,*8*,*9*,*10*], *edgesL* = [(*1*,*2*), (*2*,*3*), (*3*,*4*), (*8*,*9*),(*9*,*8*)] |)
in sinvar G ((λ *n. 2*))

value *let G* = (| *nodesL* = [*1*::*nat*,*2*,*3*,*4*,*8*,*9*,*10*], *edgesL* = [(*1*,*2*), (*2*,*3*), (*3*,*4*), (*8*,*9*),(*9*,*8*)] |)
in Dependability-eval G (|*node-properties*=[*1*↦*3*, *2*↦*2*, *3*↦*1*, *4*↦*0*, *8*↦*2*, *9*↦*2*, *10*↦*0*] |)

value *Dependability-offending-list* (| *nodesL* = [*1*::*nat*,*2*,*3*,*4*,*8*,*9*,*10*], *edgesL* = [(*1*,*2*), (*2*,*3*), (*3*,*4*), (*8*,*9*),(*9*,*8*)] |) (λ *n. 2*)

hide-fact (**open**) *sinvar-correct*
hide-const (**open**) *sinvar NetModel-node-props*

end
theory *SINVAR-NonInterference*
imports ../*TopoS-Helper*
begin

6.9 SecurityInvariant NonInterference

datatype *node-config* = *Interfering* | *Unrelated*

definition *default-node-properties* :: *node-config*
 where *default-node-properties* = *Interfering*

definition *undirected-reachable* :: *'v graph* ⇒ *'v* => *'v set* **where**
 undirected-reachable G v = (*succ-tran* (*undirected G*) *v*) − {*v*}

lemma *undirected-reachable-mono*:
 E' ⊆ *E* ⟹ *undirected-reachable* (|*nodes* = *N*, *edges* = *E'*|) *n* ⊆ *undirected-reachable* (|*nodes* = *N*, *edges* = *E*|) *n*
unfolding *undirected-reachable-def undirected-def succ-tran-def*
by (*fastforce intro*: *trancl-mono*)

fun *sinvar* :: *'v graph* ⇒ (*'v* ⇒ *node-config*) ⇒ *bool* **where**
 sinvar G nP = (∀ *n* ∈ (*nodes G*). (*nP n*) = *Interfering* ⟶ (*nP* ‘ (*undirected-reachable G n*)) ⊆ {*Unrelated*})

lemma *sinvar G nP* ⟷
 (∀ *n* ∈ {*v'* ∈ (*nodes G*). (*nP v'*) = *Interfering*}. {*nP v'* | *v'*. *v'* ∈ *undirected-reachable G n*} ⊆ {*Unrelated*})
by *auto*

definition *receiver-violation* :: *bool* **where**
 receiver-violation = *True*

simplifications for sets we need in the uniqueness proof

 lemma *tmp1*: {(*b*, *a*). *a* = *vertex-1* ∧ *b* = *vertex-2*} = {(*vertex-2*, *vertex-1*)} **by** *auto*
 lemma *tmp6*: {(*vertex-1*, *vertex-2*), (*vertex-2*, *vertex-1*)}$^+$ =
  {(*vertex-1*, *vertex-1*), (*vertex-2*, *vertex-2*), (*vertex-1*, *vertex-2*), (*vertex-2*, *vertex-1*)}
  apply(*rule*)
   apply(*rule*)
   apply(*case-tac x*, *simp*)
   apply(*erule-tac r*={(*vertex-1*, *vertex-2*), (*vertex-2*, *vertex-1*)} **in** *trancl-induct*)
    apply(*auto*)
   apply (*metis* (*mono-tags*) *insertCI r-r-into-trancl*)+
 done
 lemma *tmp2*: (*insert* (*vertex-1*, *vertex-2*) {(*b*, *a*). *a* = *vertex-1* ∧ *b* = *vertex-2*})$^+$ =
  {(*vertex-1*, *vertex-1*), (*vertex-2*, *vertex-2*), (*vertex-1*, *vertex-2*), (*vertex-2*, *vertex-1*)}
  apply(*subst tmp1*)
  apply(*fact tmp6*)
  done
 lemma *tmp4*: {(*e1*, *e2*). *e1* = *vertex-1* ∧ *e2* = *vertex-2* ∧ (*e1* = *vertex-1* ⟶ *e2* ≠ *vertex-2*)} = {} **by** *blast*
 lemma *tmp5*: {(*b*, *a*). *a* = *vertex-1* ∧ *b* = *vertex-2* ∨ *a* = *vertex-1* ∧ *b* = *vertex-2* ∧ (*a* = *vertex-1* ⟶ *b* ≠ *vertex-2*)} =

{(*vertex-2*, *vertex-1*)} **by** *fastforce*
lemma *unique-default-example*: *undirected-reachable* (|*nodes* = {*vertex-1*, *vertex-2*}, *edges* = {(*vertex-1*, *vertex-2*)}|) *vertex-1* = {*vertex-2*}
by(*auto simp add*: *undirected-def undirected-reachable-def succ-tran-def tmp2*)
lemma *unique-default-example-hlp1*: *delete-edges* (|*nodes* = {*vertex-1*, *vertex-2*}, *edges* = {(*vertex-1*, *vertex-2*)}|) {(*vertex-1*, *vertex-2*)} =
(|*nodes* = {*vertex-1*, *vertex-2*}, *edges* = {}|)
by(*simp add*: *delete-edges-def*)
lemma *unique-default-example-2*:
undirected-reachable (*delete-edges* (|*nodes* = {*vertex-1*, *vertex-2*}, *edges* = {(*vertex-1*, *vertex-2*)}|) {(*vertex-1*,*vertex-2*)}) *vertex-1* = {}
by(*simp add*: *undirected-def undirected-reachable-def succ-tran-def unique-default-example-hlp1*)
lemma *unique-default-example-3*:
undirected-reachable (*delete-edges* (|*nodes* = {*vertex-1*, *vertex-2*}, *edges* = {(*vertex-1*, *vertex-2*)}|) {(*vertex-1*,*vertex-2*)}) *vertex-2* = {}
by(*simp add*: *undirected-def undirected-reachable-def succ-tran-def unique-default-example-hlp1*)
lemma *unique-default-example-4*:
(*undirected-reachable* (*add-edge vertex-1 vertex-2* (*delete-edges* (|*nodes* = {*vertex-1*, *vertex-2*}, *edges* = {(*vertex-1*, *vertex-2*)}|) {(*vertex-1*, *vertex-2*)})) *vertex-1*) = {*vertex-2*}
apply(*simp add*: *delete-edges-def add-edge-def undirected-def undirected-reachable-def succ-tran-def*)
apply(*subst tmp4*)
apply(*subst tmp5*)
apply(*simp*)
apply(*subst tmp6*)
by *force*
lemma *unique-default-example-5*:
(*undirected-reachable* (*add-edge vertex-1 vertex-2* (*delete-edges* (|*nodes* = {*vertex-1*, *vertex-2*}, *edges* = {(*vertex-1*, *vertex-2*)}|) {(*vertex-1*, *vertex-2*)})) *vertex-2*) = {*vertex-1*}
apply(*simp add*: *delete-edges-def add-edge-def undirected-def undirected-reachable-def succ-tran-def*)
apply(*subst tmp4*)
apply(*subst tmp5*)
apply(*simp*)
apply(*subst tmp6*)
by *force*

lemma *empty-undirected-reachable-false*: *xb* ∈ *undirected-reachable* (*delete-edges G* (*edges G*)) *na* ⟷ *False*
by(*simp add*: *undirected-reachable-def succ-tran-def undirected-def delete-edges-edges-empty*)

6.9.1 monotonic and preliminaries

lemma *sinvar-mono*: *SecurityInvariant-withOffendingFlows.sinvar-mono sinvar*
unfolding *SecurityInvariant-withOffendingFlows.sinvar-mono-def*
apply(*clarsimp*)
apply(*rename-tac nP N E′ n E xa*)
apply(*erule-tac x=n* **in** *ballE*)
prefer *2*
apply *simp*
apply(*simp*)
apply(*drule-tac N=N* **and** *n=n* **in** *undirected-reachable-mono*)
apply(*blast*)

done

interpretation *SecurityInvariant-preliminaries*
where *sinvar = sinvar*
apply *unfold-locales*
apply(*frule-tac finite-distinct-list*[*OF wf-graph.finiteE*])
apply(*erule-tac exE*)
apply(*rename-tac list-edges*)
apply(*rule-tac ff=list-edges* **in** *SecurityInvariant-withOffendingFlows.mono-imp-set-offending-flows-not-empty*[*OF sinvar-mono*])
apply(*auto*)[*4*]
apply(*auto simp add*: *SecurityInvariant-withOffendingFlows.is-offending-flows-def empty-undirected-reachable-false*)[
apply(*fact SecurityInvariant-withOffendingFlows.sinvar-mono-imp-sinvar-mono*[*OF sinvar-mono*])
apply(*fact SecurityInvariant-withOffendingFlows.sinvar-mono-imp-is-offending-flows-mono*[*OF sinvar-mono*])
done

interpretation *NonInterference*: *SecurityInvariant-IFS*
where *default-node-properties = SINVAR-NonInterference.default-node-properties*
and *sinvar = SINVAR-NonInterference.sinvar*
unfolding *SINVAR-NonInterference.default-node-properties-def*
apply *unfold-locales*
apply(*rule ballI*)
apply(*drule SINVAR-NonInterference.offending-notevalD*)
apply(*simp*)
apply *clarify*
apply(*rename-tac xa*)
apply(*case-tac nP xa*)

apply *simp*
apply(*erule-tac x=n* **and** *A=nodes G* **in** *ballE*)
prefer *2*
apply *fast*
apply(*simp*)
apply(*thin-tac wf-graph G*)
apply(*thin-tac* (*a*,*b*) ∈ *f*)
apply(*thin-tac n* ∈ *nodes G*)
apply(*thin-tac nP n = Interfering*)
apply(*erule disjE*)
apply *fastforce*
apply *fastforce*

apply *simp*

apply(*erule default-uniqueness-by-counterexample-IFS*)
apply (*simp add*: *SecurityInvariant-withOffendingFlows.set-offending-flows-def*
SecurityInvariant-withOffendingFlows.is-offending-flows-min-set-def
SecurityInvariant-withOffendingFlows.is-offending-flows-def)
apply (*simp add*:*delete-edges-set-nodes*)
apply (*simp split*: *prod.split-asm prod.split*)
apply(*rule-tac x*=⦇ *nodes*={*vertex-1*,*vertex-2*}, *edges* = {(*vertex-1*,*vertex-2*)} ⦈ **in** *exI*, *simp*)
apply(*rule conjI*)

apply(*simp add*: *wf-graph-def*)
apply(*rule-tac x*=(λ *x*. *default-node-properties*)(*vertex-1* := *Interfering*, *vertex-2* := *Interfering*) **in** *exI*, *simp*)
apply(*rule conjI*)
apply(*simp add*: *unique-default-example*)
apply(*rule-tac x*=*vertex-2* **in** *exI*, *simp*)
apply(*rule-tac x*={(*vertex-1*,*vertex-2*)} **in** *exI*, *simp*)
apply(*simp add*: *unique-default-example*)
apply(*simp add*: *unique-default-example-2*)
apply(*simp add*: *unique-default-example-3*)
apply(*simp add*: *unique-default-example-4*)
apply(*simp add*: *unique-default-example-5*)
apply(*case-tac otherbot*)
apply *simp*
apply(*simp add*:*graph-ops*)
done

lemma *TopoS-NonInterference*: *SecurityInvariant sinvar default-node-properties receiver-violation*
unfolding *receiver-violation-def* **by** *unfold-locales*

hide-const (**open**) *sinvar receiver-violation default-node-properties*

— Hide all the helper lemmas.
hide-fact *tmp1 tmp2 tmp4 tmp5 tmp6 unique-default-example*
unique-default-example-2 unique-default-example-3 unique-default-example-4
unique-default-example-5 empty-undirected-reachable-false

end
theory *SINVAR-NonInterference-impl*
imports *SINVAR-NonInterference* ../*TopoS-Interface-impl*
begin

code-identifier code-module *SINVAR-NonInterference-impl* => (*Scala*) *SINVAR-NonInterference*

6.9.2 SecurityInvariant NonInterference List Implementation

definition *undirected-reachable* :: *'v list-graph* ⇒ *'v* => *'v list* **where**
undirected-reachable G v = *removeAll v* (*succ-tran* (*undirected G*) *v*)

lemma *undirected-reachable-set*: *set* (*undirected-reachable G v*) = {*e2*. (*v*,*e2*) ∈ (*set* (*edgesL* (*undirected G*)))$^+$} − {*v*}
by(*simp add*: *undirected-succ-tran-set undirected-nodes-set undirected-reachable-def*)

fun *sinvar-set* :: *'v list-graph* ⇒ (*'v* ⇒ *node-config*) ⇒ *bool* **where**
sinvar-set G nP = (∀ *n* ∈ *set* (*nodesL G*). (*nP n*) = *Interfering* ⟶ *set* (*map nP* (*undirected-reachable G n*)) ⊆ {*Unrelated*})

fun *sinvar* :: *'v list-graph* ⇒ (*'v* ⇒ *node-config*) ⇒ *bool* **where**
sinvar G nP = (∀ *n* ∈ *set* (*nodesL G*). (*nP n*) = *Interfering* ⟶ (*let result* = *remdups* (*map nP* (*undirected-reachable G n*)) *in result* = [] ∨ *result* = [*Unrelated*]))

lemma *P* = *Q* ⟹ (∀ *x*. *P x*) = (∀ *x*. *Q x*)
  by(*erule arg-cong*)

lemma *sinvar-eq-help1*: *nP ‘ set* (*undirected-reachable G n*) = *set* (*map nP* (*undirected-reachable G n*))
  by *auto*
lemma *sinvar-eq-help2*: *set l* = {*Unrelated*} ⟹ *remdups l* = [*Unrelated*]
 apply(*induction l*)
  apply *simp*
 apply(*simp*)
  apply (*metis empty-iff insertI1 set-empty2 subset-singletonD*)
 done
lemma *sinvar-eq-help3*: (*let result* = *remdups* (*map nP* (*undirected-reachable G n*)) *in result* = [] ∨ *result* = [*Unrelated*]) = (*set* (*map nP* (*undirected-reachable G n*)) ⊆ {*Unrelated*})
  apply *simp*
  apply(*rule iffI*)
   apply(*erule disjE*)
    apply *simp*
   apply(*simp only*: *set-map*[*symmetric*])
   apply(*subst set-remdups*[*symmetric*])
   apply *simp*
  apply(*case-tac nP ‘ set* (*undirected-reachable G n*) = {})
   apply *fast*
  apply(*case-tac nP ‘ set* (*undirected-reachable G n*) = {*Unrelated*})
   defer
   apply(*subgoal-tac nP ‘ set* (*undirected-reachable G n*) ⊆ {*Unrelated*} ⟹
    nP ‘ set (*undirected-reachable G n*) ≠ {} ⟹
    nP ‘ set (*undirected-reachable G n*) ≠ {*Unrelated*} ⟹ *False*)
    apply *fast*
   apply (*metis subset-singletonD*)
  apply *simp*
  apply(*rule disjI2*)
  apply(*simp only*: *sinvar-eq-help1*)
  apply(*simp add*:*sinvar-eq-help2*)
  done

lemma *sinvar-list-eq-set*: *sinvar* = *sinvar-set*
  apply(*insert sinvar-eq-help3*)
  apply(*simp add*: *fun-eq-iff*)
  apply(*rule allI*)+
  apply *fastforce*
  done

value *sinvar*
    (| *nodesL* = [*1*::*nat*,*2*,*3*,*4*], *edgesL* = [(*1*,*2*), (*2*,*3*), (*3*,*4*), (*8*,*9*),(*9*,*8*)] |)
    (λ*e*. *SINVAR-NonInterference.default-node-properties*)
value *sinvar*
    (| *nodesL* = [*1*::*nat*,*2*,*3*,*4*,*8*,*9*,*10*], *edgesL* = [(*1*,*2*), (*2*,*3*), (*3*,*4*)] |)
     ((λ*e*. *SINVAR-NonInterference.default-node-properties*)(*1*:= *Interfering*, *2*:= *Unrelated*, *3*:= *Unrelated*, *4*:= *Unrelated*))

value *sinvar*
⦇ *nodesL* = [*1*::*nat*,*2*,*3*,*4*,*5*, *8*,*9*,*10*], *edgesL* = [(*1*,*2*), (*2*,*3*), (*3*,*4*), (*5*,*4*), (*8*,*9*),(*9*,*8*)] ⦈
((λ*e*. *SINVAR-NonInterference.default-node-properties*)(*1*:= *Interfering*, *2*:= *Unrelated*, *3*:= *Unrelated*, *4*:= *Unrelated*))
value *sinvar*
⦇ *nodesL* = [*1*::*nat*], *edgesL* = [(*1*,*1*)] ⦈
((λ*e*. *SINVAR-NonInterference.default-node-properties*)(*1*:= *Interfering*))

value (*undirected-reachable* ⦇ *nodesL* = [*1*::*nat*], *edgesL* = [(*1*,*1*)] ⦈ *1*) = []

definition *NonInterference-offending-list*:: ′*v list-graph* ⇒ (′*v* ⇒ *node-config*) ⇒ (′*v* × ′*v*) *list list*
where
NonInterference-offending-list = *Generic-offending-list sinvar*

definition *NetModel-node-props P* = (λ *i*. (*case* (*node-properties P*) *i of Some property* ⇒ *property* | *None* ⇒ *SINVAR-NonInterference.default-node-properties*))
lemma[*code-unfold*]: *SecurityInvariant.node-props SINVAR-NonInterference.default-node-properties P* = *NetModel-node-props P*
apply(*simp add*: *NetModel-node-props-def*)
done

definition *NonInterference-eval G P* = (*wf-list-graph G* ∧
sinvar G (*SecurityInvariant.node-props SINVAR-NonInterference.default-node-properties P*))

lemma *sinvar-correct*: *wf-list-graph G* ⟹ *SINVAR-NonInterference.sinvar* (*list-graph-to-graph G*) *nP* = *sinvar G nP*
apply(*simp add*: *sinvar-list-eq-set*)
apply(*rule all-nodes-list-I*)
by (*simp add*: *SINVAR-NonInterference.undirected-reachable-def succ-tran-correct undirected-correct undirected-reachable-def*)

interpretation *NonInterference-impl*:*TopoS-List-Impl*
where *default-node-properties*=*SINVAR-NonInterference.default-node-properties*
and *sinvar-spec*=*SINVAR-NonInterference.sinvar*
and *sinvar-impl*=*sinvar*
and *receiver-violation*=*SINVAR-NonInterference.receiver-violation*
and *offending-flows-impl*=*NonInterference-offending-list*
and *node-props-impl*=*NetModel-node-props*
and *eval-impl*=*NonInterference-eval*
apply(*unfold TopoS-List-Impl-def*)
apply(*rule conjI*)
apply(*rule conjI*)
apply(*simp add*: *TopoS-NonInterference*; *fail*)

apply(*intro allI impI*)
apply(*fact sinvar-correct*)
apply(*rule conjI*)
apply(*unfold NonInterference-offending-list-def*)
apply(*intro allI impI*)
apply(*rule Generic-offending-list-correct*)
apply(*assumption*)
apply(*simp only*: *sinvar-correct*)
apply(*rule conjI*)
apply(*intro allI*)
apply(*simp only*: *NetModel-node-props-def*)
apply(*metis NonInterference.node-props.simps NonInterference.node-props-eq-node-props-formaldef*)
apply(*simp only*: *NonInterference-eval-def*)
apply(*intro allI impI*)
apply(*rule TopoS-eval-impl-proofrule*[*OF TopoS-NonInterference*])
apply(*simp only*: *sinvar-correct*)
done

6.9.3 NonInterference packing

definition *SINVAR-LIB-NonInterference* :: (*'v*::*vertex*, *node-config*) *TopoS-packed* **where**
SINVAR-LIB-NonInterference ≡
(| *nm-name* = *''NonInterference''*,
nm-receiver-violation = *SINVAR-NonInterference.receiver-violation*,
nm-default = *SINVAR-NonInterference.default-node-properties*,
nm-sinvar = *sinvar*,
nm-offending-flows = *NonInterference-offending-list*,
nm-node-props = *NetModel-node-props*,
nm-eval = *NonInterference-eval*
|)
interpretation *SINVAR-LIB-NonInterference-interpretation*: *TopoS-modelLibrary SINVAR-LIB-NonInterference*
SINVAR-NonInterference.sinvar
apply(*unfold TopoS-modelLibrary-def SINVAR-LIB-NonInterference-def*)
apply(*rule conjI*)
apply(*simp*)
apply(*simp*)
by(*unfold-locales*)

Example:

context begin
private definition *example-graph* = (| *nodesL* = [*1*::*nat*,*2*,*3*,*4*,*5*, *8*,*9*,*10*], *edgesL* = [(*1*,*2*), (*2*,*3*), (*3*,*4*), (*5*,*4*), (*8*,*9*), (*9*,*8*)] |)
private definition *example-conf* = ((*λe. SINVAR-NonInterference.default-node-properties*)
(*1*:= *Interfering*, *2*:= *Unrelated*, *3*:= *Unrelated*, *4*:= *Unrelated*, *8*:= *Unrelated*, *9*:= *Unrelated*))

private lemma ¬ *sinvar example-graph example-conf* **by** *eval*
private lemma *NonInterference-offending-list example-graph example-conf* =
[[(*1*, *2*)], [(*2*, *3*)], [(*3*, *4*)], [(*5*, *4*)]] **by** *eval*
end

hide-const (**open**) *NetModel-node-props*
hide-const (**open**) *sinvar*

end
theory *SINVAR-ACLcommunicateWith*
imports ../*TopoS-Helper*
begin

6.10 SecurityInvariant ACLcommunicateWith

An access control list strategy that says that hosts must only transitively access each other if allowed

Warning: this transitive model has exponential computational complexity

definition *default-node-properties* :: *'v list*
  where *default-node-properties* ≡ []

fun *sinvar* :: *'v graph* ⇒ (*'v* ⇒ *'v list*) ⇒ *bool* **where**
  sinvar G nP = (∀ *v* ∈ *nodes G*. (∀ *a* ∈ (*succ-tran G v*). *a* ∈ *set* (*nP v*)))

definition *receiver-violation* :: *bool* **where**
  receiver-violation ≡ *False*

lemma *ACLcommunicateWith-sinvar-alternative*:
  wf-graph G ⟹ *sinvar G nP* = (∀ (*e1*,*e2*) ∈ (*edges G*)$^+$. *e2* ∈ *set* (*nP e1*))
  proof(*unfold sinvar.simps*, *rule iffI*, *goal-cases*)
  case *1*
    from *1*(*1*) **have** *e1-nodes*: (*e1*, *e2*) ∈ *edges G* ⟹ *e1* ∈ *nodes G* **for** *e1 e2*
      by (*simp add*: *wf-graph.E-wfD*(*1*))
    from *1*(*2*) **have** ∀ *v*∈*nodes G*. ∀ *a*. (*v*, *a*) ∈ (*edges G*)$^+$ ⟶ *a* ∈ *set* (*nP v*)
      by(*simp add*: *succ-tran-def*)
    with *e1-nodes* **have** (*e1*, *e2*)∈(*edges G*)$^+$ ⟹ *e2* ∈ *set* (*nP e1*) **for** *e1 e2*
      by (*meson tranclD*)
    thus *?case* **by** *blast*
  next
  case *2*
    from *2*(*1*) **have** *e1-nodes*: (*v*, *a*) ∈ *edges G* ⟹ *v* ∈ *nodes G* **for** *v a*
      by (*simp add*: *wf-graph.E-wfD*(*1*))
    with *2*(*2*) **show** *?case* **by**(*auto simp add*: *succ-tran-def*)
  qed

lemma *sinvar-mono*: *SecurityInvariant-withOffendingFlows.sinvar-mono sinvar*
  unfolding *SecurityInvariant-withOffendingFlows.sinvar-mono-def*
  proof(*clarify*)
  fix *nP*::(*'v* ⇒ *'v list*) **and** *N E' E*
  assume *a1*: *wf-graph* (|*nodes* = *N*, *edges* = *E*|)
  and *a2*: *E'* ⊆ *E*
  and *a3*: *sinvar* (|*nodes* = *N*, *edges* = *E*|) *nP*

  from *a3* **have** *v* ∈ *N* ⟹ ∀ *a*∈(*succ-tran* (|*nodes* = *N*, *edges* = *E*|) *v*). *a* ∈ *set* (*nP v*) **for** *v* **by** *fastforce*
  with *a2* **have** *g2*: *v* ∈ *N* ⟹ (∀ *a* ∈ (*succ-tran* (|*nodes* = *N*, *edges* = *E'*|) *v*). *a* ∈ *set* (*nP v*)) **for** *v*
    using *succ-tran-mono*[*OF a1*] **by** *blast*

thus *sinvar* (|*nodes* = *N*, *edges* = *E'*|) *nP* **by** *simp*
qed

interpretation *SecurityInvariant-preliminaries*
where *sinvar* = *sinvar*
apply *unfold-locales*
apply(*frule-tac finite-distinct-list*[*OF wf-graph.finiteE*])
apply(*erule-tac exE*)
apply(*rename-tac list-edges*)
apply(*rule-tac ff=list-edges* **in** *SecurityInvariant-withOffendingFlows.mono-imp-set-offending-flows-not-empty*[*OF sinvar-mono*])
apply(*auto*)[*4*]
apply(*auto simp add*: *SecurityInvariant-withOffendingFlows.is-offending-flows-def graph-ops False-set succ-tran-empty*)[*1*]
apply(*fact SecurityInvariant-withOffendingFlows.sinvar-mono-imp-sinvar-mono*[*OF sinvar-mono*])
apply(*fact SecurityInvariant-withOffendingFlows.sinvar-mono-imp-is-offending-flows-mono*[*OF sinvar-mono*])
done

lemma *unique-default-example*: *succ-tran* (|*nodes* = {*vertex-1*, *vertex-2*}, *edges* = {(*vertex-1*, *vertex-2*)}|) *vertex-2* = {}
apply (*simp add*: *succ-tran-def*)
by (*metis Domain.DomainI Domain-empty Domain-insert distinct-vertices12 singleton-iff trancl-domain*)

interpretation *ACLcommunicateWith*: *SecurityInvariant-ACS*
where *default-node-properties* = *SINVAR-ACLcommunicateWith.default-node-properties*
and *sinvar* = *SINVAR-ACLcommunicateWith.sinvar*
unfolding *SINVAR-ACLcommunicateWith.default-node-properties-def*
apply *unfold-locales*

apply *simp*
apply(*subst*(*asm*) *SecurityInvariant-withOffendingFlows.set-offending-flows-simp*, *simp*)
apply(*clarsimp*)
apply (*metis*)

apply(*erule default-uniqueness-by-counterexample-ACS*)
apply(*simp*)
apply (*simp add*: *SecurityInvariant-withOffendingFlows.set-offending-flows-def*
SecurityInvariant-withOffendingFlows.is-offending-flows-min-set-def
SecurityInvariant-withOffendingFlows.is-offending-flows-def)
apply (*simp add*:*graph-ops*)
apply (*simp split*: *prod.split-asm prod.split*)
apply(*simp add*: *List.neq-Nil-conv*)
apply(*erule exE*)
apply(*rename-tac canAccessThis*)
apply(*case-tac canAccessThis* = *vertex-1*)
apply(*rule-tac x*=(| *nodes*={*canAccessThis*,*vertex-2*}, *edges* = {(*vertex-2*,*canAccessThis*)} |) **in** *exI*, *simp*)
apply(*rule conjI*)
apply(*simp add*: *wf-graph-def*)
apply(*rule-tac x*=(λ *x*. [])(*vertex-1* := [], *vertex-2* := []) **in** *exI*, *simp*)

apply(*simp add*: *example-simps*)
apply(*rule-tac x*={(*vertex-2*,*vertex-1*)} **in** *exI*, *simp*)
apply(*simp add*: *example-simps*)
apply(*fastforce*)

apply(*rule-tac x*=(| *nodes*={*vertex-1*,*canAccessThis*}, *edges* = {(*vertex-1*,*canAccessThis*)} |) **in** *exI*, *simp*)
apply(*rule conjI*)
apply(*simp add*: *wf-graph-def*)
apply(*rule-tac x*=(λ *x*. [])(*vertex-1* := [], *canAccessThis* := []) **in** *exI*, *simp*)
apply(*simp add*: *example-simps*)
apply(*rule-tac x*={(*vertex-1*,*canAccessThis*)} **in** *exI*, *simp*)
apply(*simp add*: *example-simps*)
apply(*fastforce*)
done

lemma *TopoS-ACLcommunicateWith*: *SecurityInvariant sinvar default-node-properties receiver-violation*
unfolding *receiver-violation-def* **by** *unfold-locales*

hide-const (**open**) *sinvar receiver-violation default-node-properties*

end
theory *SINVAR-ACLnotCommunicateWith*
imports ../*TopoS-Helper SINVAR-ACLcommunicateWith*
begin

6.11 SecurityInvariant ACLnotCommunicateWith

An access control list strategy that says that hosts must not transitively access each other.

node properties: a set of hosts this host must not access

definition *default-node-properties* :: *'v set*
where *default-node-properties* ≡ *UNIV*

fun *sinvar* :: *'v graph* ⇒ (*'v* ⇒ *'v set*) ⇒ *bool* **where**
sinvar G nP = (∀ *v* ∈ *nodes G*. ∀ *a* ∈ (*succ-tran G v*). *a* ∉ (*nP v*))

definition *receiver-violation* :: *bool* **where**
receiver-violation ≡ *False*

It is the inverse of *SINVAR-ACLcommunicateWith.sinvar*

lemma *ACLcommunicateNotWith-inverse-ACLcommunicateWith*:
∀ *v*. *UNIV* − *nP' v* = *set* (*nP v*) ⟹ *SINVAR-ACLcommunicateWith.sinvar G nP* ⟷ *sinvar G nP'*
by *auto*

lemma *sinvar-mono*: *SecurityInvariant-withOffendingFlows.sinvar-mono sinvar*
unfolding *SecurityInvariant-withOffendingFlows.sinvar-mono-def*
proof(*clarify*)
fix *nP*::(*'v* ⇒ *'v set*) **and** *N E' E*
assume *a1*: *wf-graph* (|*nodes* = *N*, *edges* = *E*|)

and *a2*: $E' \subseteq E$
and *a3*: *sinvar* ⦇*nodes* = *N*, *edges* = *E*⦈ *nP*

from *a3* **have** $\bigwedge v\ a.\ v \in N \Longrightarrow\ a \in$ (*succ-tran* ⦇*nodes* = *N*, *edges* = *E*⦈ *v*) $\Longrightarrow a \notin$ (*nP v*) **by** *fastforce*
from *this a2* **have** *g1*: $\bigwedge v\ a.\ v \in N \Longrightarrow a \in$ (*succ-tran* ⦇*nodes* = *N*, *edges* = *E'*⦈ *v*) $\Longrightarrow a \notin$ (*nP v*)
using *succ-tran-mono*[*OF a1*] **by** *blast*

thus *sinvar* ⦇*nodes* = *N*, *edges* = *E'*⦈ *nP*
by(*clarsimp*)
qed

lemma *succ-tran-empty*: (*succ-tran* ⦇*nodes* = *nodes G*, *edges* = {}⦈ *v*) = {}
by(*simp add*: *succ-tran-def*)

interpretation *SecurityInvariant-preliminaries*
where *sinvar* = *sinvar*
apply *unfold-locales*
apply(*frule-tac finite-distinct-list*[*OF wf-graph.finiteE*])
apply(*erule-tac exE*)
apply(*rename-tac list-edges*)
apply(*rule-tac ff=list-edges* **in** *SecurityInvariant-withOffendingFlows.mono-imp-set-offending-flows-not-empty*[*OF sinvar-mono*])
apply(*auto*)[*4*]
apply(*auto simp add*: *SecurityInvariant-withOffendingFlows.is-offending-flows-def graph-ops succ-tran-empty*)[*1*]
apply(*fact SecurityInvariant-withOffendingFlows.sinvar-mono-imp-sinvar-mono*[*OF sinvar-mono*])
apply(*fact SecurityInvariant-withOffendingFlows.sinvar-mono-imp-is-offending-flows-mono*[*OF sinvar-mono*])
done

lemma *unique-default-example*: *succ-tran* ⦇*nodes* = {*vertex-1*, *vertex-2*}, *edges* = {(*vertex-1*, *vertex-2*)}⦈ *vertex-2* = {}
apply (*simp add*: *succ-tran-def*)
by (*metis Domain.DomainI Domain-empty Domain-insert distinct-vertices12 singleton-iff trancl-domain*)

interpretation *ACLnotCommunicateWith*: *SecurityInvariant-ACS*
where *default-node-properties* = *SINVAR-ACLnotCommunicateWith.default-node-properties*
and *sinvar* = *SINVAR-ACLnotCommunicateWith.sinvar*
unfolding *SINVAR-ACLnotCommunicateWith.default-node-properties-def*
apply *unfold-locales*

apply *simp*
apply(*subst*(*asm*) *SecurityInvariant-withOffendingFlows.set-offending-flows-simp*, *simp*)
apply(*clarsimp*)
apply (*metis*)

apply(*erule default-uniqueness-by-counterexample-ACS*)
apply(*simp*)
apply (*simp add*: *SecurityInvariant-withOffendingFlows.set-offending-flows-def*
SecurityInvariant-withOffendingFlows.is-offending-flows-min-set-def
SecurityInvariant-withOffendingFlows.is-offending-flows-def)

```
  apply (simp add:graph-ops)
  apply (simp split: prod.split-asm prod.split)
  apply(case-tac otherbot = {})

   apply(rule-tac x=(| nodes={vertex-1,vertex-2}, edges = {(vertex-1,vertex-2)} |) in exI, simp)
   apply(rule conjI)
    apply(simp add: wf-graph-def)
   apply(rule-tac x=(λ x. UNIV)(vertex-1 := {vertex-2}, vertex-2 := {}) in exI, simp)
   apply(simp add: example-simps)
   apply(rule-tac x={(vertex-1,vertex-2)} in exI, simp)
   apply(simp add: example-simps)

  apply(subgoal-tac ∃ canAccess. canAccess ∈ UNIV ∧ canAccess ∉ otherbot)
   prefer 2
   apply blast
  apply(erule exE)
  apply(rename-tac canAccessThis)
  apply(case-tac vertex-1 ≠ canAccessThis)

   apply(rule-tac x=(| nodes={vertex-1,canAccessThis}, edges = {(vertex-1,canAccessThis)} |) in exI,
simp)
   apply(rule conjI)
    apply(simp add: wf-graph-def)
   apply(rule-tac x=(λ x. UNIV)(vertex-1 := UNIV, canAccessThis := {}) in exI, simp)
   apply(simp add: example-simps)
   apply(rule-tac x={(vertex-1,canAccessThis)} in exI, simp)
   apply(simp add: example-simps)

  apply(rule-tac x=(| nodes={canAccessThis,vertex-2}, edges = {(vertex-2,canAccessThis)} |) in exI,
simp)
  apply(rule conjI)
   apply(simp add: wf-graph-def)
  apply(rule-tac x=(λ x. UNIV)(vertex-2 := UNIV, canAccessThis := {}) in exI, simp)
  apply(simp add: example-simps)
  apply(rule-tac x={(vertex-2,canAccessThis)} in exI, simp)
  apply(simp add: example-simps)
 done

 lemma TopoS-ACLnotCommunicateWith: SecurityInvariant sinvar default-node-properties receiver-violation
  unfolding receiver-violation-def by unfold-locales

hide-const (open) sinvar receiver-violation default-node-properties

end
theory SINVAR-ACLnotCommunicateWith-impl
imports SINVAR-ACLnotCommunicateWith ../TopoS-Interface-impl
begin

code-identifier code-module SINVAR-ACLnotCommunicateWith-impl => (Scala) SINVAR-ACLnotCommunicateW
```

6.11.1 SecurityInvariant ACLnotCommunicateWith List Implementation

```
fun sinvar :: 'v list-graph ⇒ ('v ⇒ 'v set) ⇒ bool where
  sinvar G nP = (∀ v ∈ set (nodesL G). ∀ a ∈ set (succ-tran G v). a ∉ (nP v))
```

definition *NetModel-node-props* (*P*::(*'v*::*vertex*, *'v set*) *TopoS-Params*) =
(λ *i*. (*case* (*node-properties P*) *i of Some property* ⇒ *property* | *None* ⇒ *SINVAR-ACLnotCommunicateWith.default-nod*
lemma[*code-unfold*]: *SecurityInvariant.node-props SINVAR-ACLnotCommunicateWith.default-node-properties*
P = *NetModel-node-props P*
apply(*simp add*: *NetModel-node-props-def*)
done

definition *ACLnotCommunicateWith-offending-list* = *Generic-offending-list sinvar*

definition *ACLnotCommunicateWith-eval G P* = (*wf-list-graph G* ∧
sinvar G (*SecurityInvariant.node-props SINVAR-ACLnotCommunicateWith.default-node-properties*
P))

lemma *sinvar-correct*: *wf-list-graph G* ⟹ *SINVAR-ACLnotCommunicateWith.sinvar* (*list-graph-to-graph*
G) *nP* = *sinvar G nP*
by (*metis SINVAR-ACLnotCommunicateWith.sinvar.simps SINVAR-ACLnotCommunicateWith-impl.sinvar.simps*
graph.select-convs(*1*) *list-graph-to-graph-def succ-tran-correct*)

interpretation *ACLnotCommunicateWith-impl*:*TopoS-List-Impl*
where *default-node-properties*=*SINVAR-ACLnotCommunicateWith.default-node-properties*
and *sinvar-spec*=*SINVAR-ACLnotCommunicateWith.sinvar*
and *sinvar-impl*=*sinvar*
and *receiver-violation*=*SINVAR-ACLnotCommunicateWith.receiver-violation*
and *offending-flows-impl*=*ACLnotCommunicateWith-offending-list*
and *node-props-impl*=*NetModel-node-props*
and *eval-impl*=*ACLnotCommunicateWith-eval*
apply(*unfold TopoS-List-Impl-def*)
apply(*rule conjI*)
apply(*rule conjI*)
apply(*simp add*: *TopoS-ACLnotCommunicateWith*; *fail*)
apply(*intro allI impI*)
apply(*fact sinvar-correct*)
apply(*rule conjI*)
apply(*unfold ACLnotCommunicateWith-offending-list-def*)
apply(*intro allI impI*)
apply(*rule Generic-offending-list-correct*)
apply(*assumption*)
apply(*simp only*: *sinvar-correct*; *fail*)
apply(*rule conjI*)
apply(*intro allI*)
apply(*simp only*: *NetModel-node-props-def*)
apply(*metis ACLnotCommunicateWith.node-props.simps ACLnotCommunicateWith.node-props-eq-node-props-formalo*
apply(*simp only*: *ACLnotCommunicateWith-eval-def*)
apply(*intro allI impI*)
apply(*rule TopoS-eval-impl-proofrule*[*OF TopoS-ACLnotCommunicateWith*])
apply(*simp only*: *sinvar-correct*)
done

6.11.2 packing

definition *SINVAR-LIB-ACLnotCommunicateWith*:: ('*v*::*vertex*, '*v set*) *TopoS-packed* **where**
SINVAR-LIB-ACLnotCommunicateWith ≡
(| *nm-name* = ''*ACLnotCommunicateWith*'',
nm-receiver-violation = *SINVAR-ACLnotCommunicateWith.receiver-violation*,
nm-default = *SINVAR-ACLnotCommunicateWith.default-node-properties*,
nm-sinvar = *sinvar*,
nm-offending-flows = *ACLnotCommunicateWith-offending-list*,
nm-node-props = *NetModel-node-props*,
nm-eval = *ACLnotCommunicateWith-eval*
|)
interpretation *SINVAR-LIB-ACLnotCommunicateWith-interpretation*: *TopoS-modelLibrary SINVAR-LIB-ACLnotCommunicateWith*
SINVAR-ACLnotCommunicateWith.sinvar
apply(*unfold TopoS-modelLibrary-def SINVAR-LIB-ACLnotCommunicateWith-def*)
apply(*rule conjI*)
apply(*simp*)
apply(*simp*)
by(*unfold-locales*)

Examples

hide-const (**open**) *NetModel-node-props*
hide-const (**open**) *sinvar*

end
theory *SINVAR-ACLcommunicateWith-impl*
imports *SINVAR-ACLcommunicateWith ../TopoS-Interface-impl*
begin

code-identifier code-module *SINVAR-ACLcommunicateWith-impl* => (*Scala*) *SINVAR-ACLcommunicateWith*

6.11.3 List Implementation

fun *sinvar* :: '*v list-graph* ⇒ ('*v* ⇒ '*v list*) ⇒ *bool* **where**
sinvar G nP = (∀ *v* ∈ *set* (*nodesL G*). ∀ *a* ∈ (*set* (*succ-tran G v*)). *a* ∈ *set* (*nP v*))

definition *NetModel-node-props* (*P*::('*v*::*vertex*, '*v list*) *TopoS-Params*) =
(λ *i*. (*case* (*node-properties P*) *i of Some property* ⇒ *property* | *None* ⇒ *SINVAR-ACLcommunicateWith.default-node-pr*
lemma[*code-unfold*]: *SecurityInvariant.node-props SINVAR-ACLcommunicateWith.default-node-properties P* = *NetModel-node-props P*
by(*simp add*: *NetModel-node-props-def*)

definition *ACLcommunicateWith-offending-list* = *Generic-offending-list sinvar*

definition *ACLcommunicateWith-eval G P* = (*wf-list-graph G* ∧
sinvar G (*SecurityInvariant.node-props SINVAR-ACLcommunicateWith.default-node-properties P*))

lemma *sinvar-correct*: *wf-list-graph G* ⟹ *SINVAR-ACLcommunicateWith.sinvar* (*list-graph-to-graph G*) *nP* = *sinvar G nP*
by (*metis SINVAR-ACLcommunicateWith.sinvar.simps SINVAR-ACLcommunicateWith-impl.sinvar.simps graph.select-convs*(*1*) *list-graph-to-graph-def succ-tran-correct*)

interpretation *SINVAR-ACLcommunicateWith-impl*: *TopoS-List-Impl*
where *default-node-properties*=*SINVAR-ACLcommunicateWith.default-node-properties*
and *sinvar-spec*=*SINVAR-ACLcommunicateWith.sinvar*
and *sinvar-impl*=*sinvar*
and *receiver-violation*=*SINVAR-ACLcommunicateWith.receiver-violation*
and *offending-flows-impl*=*ACLcommunicateWith-offending-list*
and *node-props-impl*=*NetModel-node-props*
and *eval-impl*=*ACLcommunicateWith-eval*
apply(*unfold TopoS-List-Impl-def*)
apply(*rule conjI*)
apply(*rule conjI*)
apply(*simp add*: *TopoS-ACLcommunicateWith*; *fail*)
apply(*intro allI impI*)
apply(*fact sinvar-correct*)
apply(*rule conjI*)
apply(*unfold ACLcommunicateWith-offending-list-def*)
apply(*intro allI impI*)
apply(*rule Generic-offending-list-correct*)
apply(*assumption*)
apply(*simp only*: *sinvar-correct*; *fail*)
apply(*rule conjI*)
apply(*intro allI*)
apply(*simp only*: *NetModel-node-props-def*)
apply(*metis ACLcommunicateWith.node-props.simps ACLcommunicateWith.node-props-eq-node-props-formaldef*)
apply(*simp only*: *ACLcommunicateWith-eval-def*)
apply(*intro allI impI*)
apply(*rule TopoS-eval-impl-proofrule*[*OF TopoS-ACLcommunicateWith*])
apply(*simp only*: *sinvar-correct*; *fail*)
done

6.11.4 packing

definition *SINVAR-LIB-ACLcommunicateWith*:: (*'v*::*vertex*, *'v list*) *TopoS-packed* **where**
SINVAR-LIB-ACLcommunicateWith ≡
(| *nm-name* = *''ACLcommunicateWith''*,
nm-receiver-violation = *SINVAR-ACLcommunicateWith.receiver-violation*,
nm-default = *SINVAR-ACLcommunicateWith.default-node-properties*,
nm-sinvar = *sinvar*,
nm-offending-flows = *ACLcommunicateWith-offending-list*,
nm-node-props = *NetModel-node-props*,
nm-eval = *ACLcommunicateWith-eval*
|)
interpretation *SINVAR-LIB-ACLcommunicateWith-interpretation*: *TopoS-modelLibrary SINVAR-LIB-ACLcommunic*
SINVAR-ACLcommunicateWith.sinvar
apply(*unfold TopoS-modelLibrary-def SINVAR-LIB-ACLcommunicateWith-def*)
apply(*rule conjI*)
apply(*simp*)
apply(*simp*)
by(*unfold-locales*)

Examples

context begin

1 can access 2 and 3 2 can access 3

private lemma *sinvar*
  (| *nodesL* = [*1*::*nat*, *2*, *3*],
   edgesL = [(*1*,*2*), (*2*,*3*)]|)
  (((λ*v*. *SINVAR-ACLcommunicateWith.default-node-properties*)
   (*1* := [*2*,*3*]))
   (*2* := [*3*])) **by** *eval*

Everyone can access everyone, except for 1: 1 must not access 4. The offending flows may be any edge on the path from 1 to 4

lemma *ACLcommunicateWith-offending-list*
  (| *nodesL* = [*1*::*nat*, *2*, *3*, *4*],
   edgesL = [(*1*,*2*), (*2*,*3*), (*3*, *4*)]|)
  (((((λ*v*. *SINVAR-ACLcommunicateWith.default-node-properties*)
  (*1* := [*1*,*2*,*3*]))
  (*2* := [*1*,*2*,*3*,*4*]))
  (*3* := [*1*,*2*,*3*,*4*]))
  (*4* := [*1*,*2*,*3*,*4*])) =
 [[(*1*, *2*)], [(*2*, *3*)], [(*3*, *4*)]] **by** *eval*

If we add the additional edge from 1 to 3, then the offending flows are either

(3.4) , because this disconnects 4 from the graph completely

- any pair of edges which disconnects 1 from 3

lemma *ACLcommunicateWith-offending-list*
  (| *nodesL* = [*1*::*nat*, *2*, *3*, *4*],
   edgesL = [(*1*,*2*), (*1*,*3*), (*2*,*3*), (*3*, *4*)]|)
  (((((λ*v*. *SINVAR-ACLcommunicateWith.default-node-properties*)
  (*1* := [*1*,*2*,*3*]))
  (*2* := [*1*,*2*,*3*,*4*]))
  (*3* := [*1*,*2*,*3*,*4*]))
  (*4* := [*1*,*2*,*3*,*4*])) =
 [[(*1*, *2*), (*1*, *3*)], [(*1*, *3*), (*2*, *3*)], [(*3*, *4*)]] **by** *eval*
end

hide-const (**open**) *NetModel-node-props*
hide-const (**open**) *sinvar*

end
theory *SINVAR-Dependability-norefl*
imports ../*TopoS-Helper*
begin

6.12 SecurityInvariant *Dependability-norefl*

A version of the Dependability model but if a node reaches itself, it is ignored

type-synonym *dependability-level* = *nat*

definition *default-node-properties* :: *dependability-level*
 where *default-node-properties* ≡ *0*

Less-equal other nodes depend on the output of a node than its dependability level.

fun *sinvar* :: *'v graph* ⇒ (*'v* ⇒ *dependability-level*) ⇒ *bool* **where**
sinvar G nP = (∀ (*e1*,*e2*) ∈ *edges G*. (*num-reachable-norefl G e1*) ≤ (*nP e1*))

definition *receiver-violation* :: *bool* **where**
receiver-violation ≡ *False*

lemma *sinvar-mono*: *SecurityInvariant-withOffendingFlows.sinvar-mono sinvar*
apply(*rule-tac SecurityInvariant-withOffendingFlows.sinvar-mono-I-proofrule*)
apply(*auto*)
apply(*rename-tac nP e1 e2 N E' e1' e2' E*)
apply(*drule-tac E'=E'* **and** *v=e1'* **in** *num-reachable-norefl-mono*)
apply *simp*
apply(*subgoal-tac* (*e1'*, *e2'*) ∈ *E*)
apply(*force*)
apply(*blast*)
done

interpretation *SecurityInvariant-preliminaries*
where *sinvar* = *sinvar*
apply *unfold-locales*
apply(*frule-tac finite-distinct-list*[*OF wf-graph.finiteE*])
apply(*erule-tac exE*)
apply(*rename-tac list-edges*)
apply(*rule-tac ff=list-edges* **in** *SecurityInvariant-withOffendingFlows.mono-imp-set-offending-flows-not-empty*[*OF sinvar-mono*])
apply(*auto*)[*4*]
apply(*auto simp add*: *SecurityInvariant-withOffendingFlows.is-offending-flows-def graph-ops*)[*1*]
apply(*fact SecurityInvariant-withOffendingFlows.sinvar-mono-imp-sinvar-mono*[*OF sinvar-mono*])
apply(*fact SecurityInvariant-withOffendingFlows.sinvar-mono-imp-is-offending-flows-mono*[*OF sinvar-mono*])
done

interpretation *Dependability*: *SecurityInvariant-ACS*
where *default-node-properties* = *SINVAR-Dependability-norefl.default-node-properties*
and *sinvar* = *SINVAR-Dependability-norefl.sinvar*
unfolding *SINVAR-Dependability-norefl.default-node-properties-def*
proof
fix *G*::*'a graph* **and** *f nP*
assume *wf-graph G* **and** *f* ∈ *set-offending-flows G nP*
thus ∀ *i*∈*fst* ' *f*. ¬ *sinvar G* (*nP*(*i* := *0*))
apply (*simp add*: *SecurityInvariant-withOffendingFlows.set-offending-flows-def*
SecurityInvariant-withOffendingFlows.is-offending-flows-min-set-def
SecurityInvariant-withOffendingFlows.is-offending-flows-def)
apply (*simp split*: *prod.split-asm prod.split*)
apply (*simp add*:*graph-ops*)

apply(*clarify*)
apply (*metis gr0I le0*)
done
next
fix *otherbot*
assume *assm*: ∀ *G f nP i. wf-graph G* ∧ *f* ∈ *set-offending-flows G nP* ∧ *i* ∈ *fst ‘ f* ⟶ ¬ *sinvar G* (*nP*(*i* := *otherbot*))
have *unique-default-example-succ-tran*:
succ-tran (|*nodes* = {*vertex-1*, *vertex-2*}, *edges* = {(*vertex-1*, *vertex-2*)}|) *vertex-1* = {*vertex-2*}
using *unique-default-example1* **by** *blast*
from *assm* **show** *otherbot* = *0*
apply −
apply(*elim default-uniqueness-by-counterexample-ACS*)
apply(*simp*)
apply (*simp add*: *SecurityInvariant-withOffendingFlows.set-offending-flows-def*
SecurityInvariant-withOffendingFlows.is-offending-flows-min-set-def
SecurityInvariant-withOffendingFlows.is-offending-flows-def)
apply (*simp add*:*graph-ops*)
apply (*simp split*: *prod.split-asm prod.split*)
apply(*rule-tac x*=(| *nodes*={*vertex-1*,*vertex-2*}, *edges* = {(*vertex-1*,*vertex-2*)} |) **in** *exI*, *simp*)
apply(*rule conjI*)
apply(*simp add*: *wf-graph-def*)
apply(*rule-tac x*=(λ *x*. *0*)(*vertex-1* := *0*, *vertex-2* := *0*) **in** *exI*, *simp*)
apply(*rule conjI*)
apply(*simp add*: *unique-default-example-succ-tran num-reachable-norefl-def*; *fail*)
apply(*rule-tac x*=*vertex-1* **in** *exI*, *simp*)
apply(*rule-tac x*={(*vertex-1*,*vertex-2*)} **in** *exI*, *simp*)
apply(*simp add*: *unique-default-example-succ-tran num-reachable-norefl-def*)
apply(*simp add*: *succ-tran-def unique-default-example-simp1 unique-default-example-simp2*)
done
qed

lemma *TopoS-Dependability-norefl*: *SecurityInvariant sinvar default-node-properties receiver-violation*
unfolding *receiver-violation-def* **by** *unfold-locales*

hide-const (**open**) *sinvar receiver-violation default-node-properties*

end
theory *SINVAR-Dependability-norefl-impl*
imports *SINVAR-Dependability-norefl ../TopoS-Interface-impl*
begin

code-identifier code-module *SINVAR-Dependability-norefl-impl* => (*Scala*) *SINVAR-Dependability-norefl*

6.12.1 SecurityInvariant Dependability norefl List Implementation

fun *sinvar* :: *'v list-graph* ⇒ (*'v* ⇒ *dependability-level*) ⇒ *bool* **where**
sinvar G nP = (∀ (*e1*,*e2*) ∈ *set* (*edgesL G*). (*num-reachable-norefl G e1*) ≤ (*nP e1*))

value *sinvar*
(| *nodesL* = [*1*::*nat*,*2*,*3*,*4*], *edgesL* = [(*1*,*2*), (*2*,*3*), (*3*,*4*), (*8*,*9*),(*9*,*8*)] |)

(λe. 3)
value *sinvar*
(| *nodesL* = [1::nat,2,3,4,8,9,10], *edgesL* = [(1,2), (2,3), (3,4), (8,9),(9,8)] |)
(λe. 2)

definition *Dependability-norefl-offending-list*:: *'v list-graph* ⇒ (*'v* ⇒ *dependability-level*) ⇒ (*'v* × *'v*) *list list* **where**
Dependability-norefl-offending-list = *Generic-offending-list sinvar*

definition *NetModel-node-props P* = (λ *i*. (*case* (*node-properties P*) *i of Some property* ⇒ *property* | *None* ⇒ *SINVAR-Dependability-norefl.default-node-properties*))
lemma[*code-unfold*]: *SecurityInvariant.node-props SINVAR-Dependability-norefl.default-node-properties P* = *NetModel-node-props P*
apply(*simp add*: *NetModel-node-props-def*)
done

definition *Dependability-norefl-eval G P* = (*wf-list-graph G* ∧
sinvar G (*SecurityInvariant.node-props SINVAR-Dependability-norefl.default-node-properties P*))

lemma *sinvar-correct*: *wf-list-graph G* ⟹ *SINVAR-Dependability-norefl.sinvar* (*list-graph-to-graph G*) *nP* = *sinvar G nP*
apply(*simp*)
apply(*rule all-edges-list-I*)
apply(*simp add*: *fun-eq-iff*)
apply(*clarify*)
apply(*rename-tac x*)
apply(*drule-tac v=x* **in** *num-reachable-norefl-correct*)
apply *presburger*
done

interpretation *Dependability-norefl-impl*:*TopoS-List-Impl*
where *default-node-properties=SINVAR-Dependability-norefl.default-node-properties*
and *sinvar-spec=SINVAR-Dependability-norefl.sinvar*
and *sinvar-impl=sinvar*
and *receiver-violation=SINVAR-Dependability-norefl.receiver-violation*
and *offending-flows-impl=Dependability-norefl-offending-list*
and *node-props-impl=NetModel-node-props*
and *eval-impl=Dependability-norefl-eval*
apply(*unfold TopoS-List-Impl-def*)
apply(*rule conjI*)
apply(*rule conjI*)
apply(*simp add*: *TopoS-Dependability-norefl*; *fail*)
apply(*intro allI impI*)
apply(*fact sinvar-correct*)
apply(*rule conjI*)
apply(*unfold Dependability-norefl-offending-list-def*)

apply(*intro allI impI*)
apply(*rule Generic-offending-list-correct*)
apply(*assumption*)
apply(*simp only*: *sinvar-correct*)
apply(*rule conjI*)
apply(*intro allI*)
apply(*simp only*: *NetModel-node-props-def*)
apply(*metis Dependability.node-props.simps Dependability.node-props-eq-node-props-formaldef*)
apply(*simp only*: *Dependability-norefl-eval-def*)
apply(*intro allI impI*)
apply(*rule TopoS-eval-impl-proofrule*[*OF TopoS-Dependability-norefl*])
apply(*simp only*: *sinvar-correct*)
done

6.12.2 packing

definition *SINVAR-LIB-Dependability-norefl* :: (*'v*::*vertex*, *SINVAR-Dependability-norefl.dependability-level*) *TopoS-packed* **where**
SINVAR-LIB-Dependability-norefl ≡
(| *nm-name* = *''Dependability-norefl''*,
nm-receiver-violation = *SINVAR-Dependability-norefl.receiver-violation*,
nm-default = *SINVAR-Dependability-norefl.default-node-properties*,
nm-sinvar = *sinvar*,
nm-offending-flows = *Dependability-norefl-offending-list*,
nm-node-props = *NetModel-node-props*,
nm-eval = *Dependability-norefl-eval*
|)
interpretation *SINVAR-LIB-Dependability-norefl-interpretation*: *TopoS-modelLibrary SINVAR-LIB-Dependability-no*
SINVAR-Dependability-norefl.sinvar
apply(*unfold TopoS-modelLibrary-def SINVAR-LIB-Dependability-norefl-def*)
apply(*rule conjI*)
apply(*simp*)
apply(*simp*)
by(*unfold-locales*)

hide-fact (**open**) *sinvar-correct*
hide-const (**open**) *sinvar NetModel-node-props*

end
theory *TopoS-Library*
imports
Lib/FiniteListGraph-Impl
Security-Invariants/SINVAR-BLPbasic-impl
Security-Invariants/SINVAR-Subnets-impl
Security-Invariants/SINVAR-DomainHierarchyNG-impl
Security-Invariants/SINVAR-BLPtrusted-impl
Security-Invariants/SINVAR-SecGwExt-impl
Security-Invariants/SINVAR-Sink-impl
Security-Invariants/SINVAR-SubnetsInGW-impl
Security-Invariants/SINVAR-CommunicationPartners-impl
Security-Invariants/SINVAR-NoRefl-impl
Security-Invariants/SINVAR-Tainting-impl
Security-Invariants/SINVAR-TaintingTrusted-impl

Security-Invariants/SINVAR-Dependability-impl
Security-Invariants/SINVAR-NonInterference-impl
Security-Invariants/SINVAR-ACLnotCommunicateWith-impl
Security-Invariants/SINVAR-ACLcommunicateWith-impl
Security-Invariants/SINVAR-Dependability-norefl-impl
Lib/Efficient-Distinct
HOL−Library.Code-Target-Nat
begin

end
theory *TopoS-Composition-Theory*
imports *TopoS-Interface TopoS-Helper*
begin

7 Composition Theory

Several invariants may apply to one policy.

The security invariants are all collected in a list. The list corresponds to the security requirements. The list should have the type (*'v graph* ⇒ *bool*) *list*, i.e. a list of predicates over the policy. We need in instantiated security invariant, i.e. get rid of *'a* and *'b*

record (*'v*) *SecurityInvariant-configured* =
c-sinvar::(*'v*) *graph* ⇒ *bool*
c-offending-flows::(*'v*) *graph* ⇒ (*'v* × *'v*) *set set*
c-isIFS::*bool*

— parameters 1-3 are the *SecurityInvariant*: *sinvar* ⊥ *receiver-violation*
Fourth parameter is the host attribute mapping *nP*
TODO: probably check *wf-graph* here and optionally some host-attribute sanity checker as in DomainHierachy.
fun *new-configured-SecurityInvariant* ::
(((*'v*::*vertex*) *graph* ⇒ (*'v* ⇒ *'a*) ⇒ *bool*) × *'a* × *bool* × (*'v* ⇒ *'a*)) ⇒ (*'v SecurityInvariant-configured*) *option* **where**
new-configured-SecurityInvariant (*sinvar*, *defbot*, *receiver-violation*, *nP*) =
(
if SecurityInvariant sinvar defbot receiver-violation then
Some (|
c-sinvar = (λ*G*. *sinvar G nP*),
c-offending-flows = (λ*G*. *SecurityInvariant-withOffendingFlows.set-offending-flows sinvar G nP*),
c-isIFS = *receiver-violation*
|)
else None
)

declare *new-configured-SecurityInvariant.simps*[*simp del*]

lemma *new-configured-TopoS-sinvar-correct*:
SecurityInvariant sinvar defbot receiver-violation ⟹
c-sinvar (*the* (*new-configured-SecurityInvariant* (*sinvar*, *defbot*, *receiver-violation*, *nP*))) = (λ*G*. *sinvar G nP*)

by(*simp add*: *Let-def new-configured-SecurityInvariant.simps*)

lemma *new-configured-TopoS-offending-flows-correct*:
SecurityInvariant sinvar defbot receiver-violation $\Longrightarrow$
c-offending-flows (*the* (*new-configured-SecurityInvariant* (*sinvar*, *defbot*, *receiver-violation*, *nP*))) =

(λG. *SecurityInvariant-withOffendingFlows.set-offending-flows sinvar G nP*)
by(*simp add*: *Let-def new-configured-SecurityInvariant.simps*)

We now collect all the core properties of a security invariant, but without the $'a$ $'b$ types, so it is instantiated with a concrete configuration.

locale *configured-SecurityInvariant* =
fixes *m* :: (*'v*::*vertex*) *SecurityInvariant-configured*
assumes
— As in SecurityInvariant definition
valid-c-offending-flows:
c-offending-flows m G = {*F*. *F* ⊆ (*edges G*) ∧ ¬ *c-sinvar m G* ∧ *c-sinvar m* (*delete-edges G F*) ∧
(∀ (*e1*, *e2*) ∈ *F*. ¬ *c-sinvar m* (*add-edge e1 e2* (*delete-edges G F*)))}
and
— A empty network can have no security violations
defined-offending:
⟦ *wf-graph* ⦇ *nodes* = *N*, *edges* = {} ⦈ ⟧ ⟹ *c-sinvar m* ⦇ *nodes* = *N*, *edges* = {}⦈
and
— prohibiting more does not decrease security
mono-sinvar:
⟦ *wf-graph* ⦇ *nodes* = *N*, *edges* = *E* ⦈; *E'* ⊆ *E*; *c-sinvar m* ⦇ *nodes* = *N*, *edges* = *E* ⦈ ⟧ ⟹
c-sinvar m ⦇ *nodes* = *N*, *edges* = *E'* ⦈
begin

lemma *sinvar-monoI*:
SecurityInvariant-withOffendingFlows.sinvar-mono (λ (*G*::(*'v*::*vertex*) *graph*) (*nP*::*'v* ⇒ *'a*). *c-sinvar m G*)
apply(*simp add*: *SecurityInvariant-withOffendingFlows.sinvar-mono-def*, *clarify*)
by(*fact mono-sinvar*)

if the network where nobody communicates with anyone fulfilles its security requirement, the offending flows are always defined.

lemma *defined-offending'*:
⟦ *wf-graph G*; ¬ *c-sinvar m G* ⟧ ⟹ *c-offending-flows m G* ≠ {}
proof –
assume *a1*: *wf-graph G*
and *a2*: ¬ *c-sinvar m G*
have *subst-set-offending-flows*:
⋀*nP*. *SecurityInvariant-withOffendingFlows.set-offending-flows* (λ*G nP*. *c-sinvar m G*) *G nP* = *c-offending-flows m G*
by(*simp add*: *valid-c-offending-flows fun-eq-iff*
SecurityInvariant-withOffendingFlows.set-offending-flows-def
SecurityInvariant-withOffendingFlows.is-offending-flows-min-set-def
SecurityInvariant-withOffendingFlows.is-offending-flows-def)

from *a1* **have** *wfG-empty*: *wf-graph* ⦇*nodes* = *nodes G*, *edges* = {}⦈ **by**(*simp add*:*wf-graph-def*)

from *a1* **have** ⋀*nP*. ¬ *c-sinvar m G* ⟹ *SecurityInvariant-withOffendingFlows.set-offending-flows* (λ*G nP*. *c-sinvar m G*) *G nP* ≠ {}

apply(*frule-tac finite-distinct-list*[*OF wf-graph.finiteE*])
apply(*erule-tac exE*)
apply(*rename-tac list-edges*)
apply(*rule-tac ff=list-edges* **in** *SecurityInvariant-withOffendingFlows.mono-imp-set-offending-flows-not-empty*[*OF sinvar-monoI*])
by(*auto simp add*: *SecurityInvariant-withOffendingFlows.is-offending-flows-def delete-edges-simp2 defined-offending*[*OF wfG-empty*])

thus *?thesis* **by**(*simp add*: *a2 subst-set-offending-flows*)
qed

lemma *subst-offending-flows*: $\bigwedge$ *nP*. *SecurityInvariant-withOffendingFlows.set-offending-flows* (λ*G nP*. *c-sinvar m G*) *G nP* = *c-offending-flows m G*
apply (*unfold SecurityInvariant-withOffendingFlows.set-offending-flows-def SecurityInvariant-withOffendingFlows.is-offending-flows-min-set-def SecurityInvariant-withOffendingFlows.is-offending-flows-def*)
by(*simp add*: *valid-c-offending-flows*)

all the *SecurityInvariant-preliminaries* stuff must hold, for an arbitrary *nP*

lemma *SecurityInvariant-preliminariesD*:
SecurityInvariant-preliminaries (λ (*G*::('*v*::*vertex*) *graph*) (*nP*::'*v* ⇒ '*a*). *c-sinvar m G*)
proof(*unfold-locales*, *goal-cases*)
case *1* **thus** *?case* **using** *defined-offending'* **by**(*simp add*: *subst-offending-flows*)
next case *2* **thus** *?case* **by**(*fact mono-sinvar*)
next case *3* **thus** *?case* **by**(*fact SecurityInvariant-withOffendingFlows.sinvar-mono-imp-is-offending-flows-mono*[*OF sinvar-monoI*])
qed

lemma *negative-mono*:
$\bigwedge$ *N E' E*. *wf-graph* (| *nodes* = *N*, *edges* = *E* |) ⟹
E' ⊆ *E* ⟹ ¬ *c-sinvar m* (| *nodes* = *N*, *edges* = *E'* |) ⟹ ¬ *c-sinvar m* (| *nodes* = *N*, *edges* = *E* |)
by(*blast dest*: *mono-sinvar*)

7.1 Reusing Lemmata

lemmas *mono-extend-set-offending-flows* =
SecurityInvariant-preliminaries.mono-extend-set-offending-flows[*OF SecurityInvariant-preliminariesD*, *simplified subst-offending-flows*]

⟦*wf-graph* (|*nodes* = *V*, *edges* = *E*|); *E'* ⊆ *E*; *F'* ∈ *c-offending-flows m* (|*nodes* = *V*, *edges* = *E'*|)⟧ ⟹ ∃ *F*∈*c-offending-flows m* (|*nodes* = *V*, *edges* = *E*|). *F'* ⊆ *F*

lemmas *offending-flows-union-mono* =
SecurityInvariant-preliminaries.offending-flows-union-mono[*OF SecurityInvariant-preliminariesD*, *simplified subst-offending-flows*]

⟦*wf-graph* (|*nodes* = *V*, *edges* = *E*|); *E'* ⊆ *E*⟧ ⟹ ⋃ (*c-offending-flows m* (|*nodes* = *V*, *edges* = *E'*|)) ⊆ ⋃ (*c-offending-flows m* (|*nodes* = *V*, *edges* = *E*|))

lemmas *sinvar-valid-remove-flattened-offending-flows* =
SecurityInvariant-preliminaries.sinvar-valid-remove-flattened-offending-flows[*OF SecurityInvariant-preliminariesD*, *simplified subst-offending-flows*]

wf-graph (|*nodes* = *nodesG*, *edges* = *edgesG*|) ⟹ *c-sinvar m* (|*nodes* = *nodesG*, *edges* = *edgesG* − ⋃ (*c-offending-flows m* (|*nodes* = *nodesG*, *edges* = *edgesG*|))|)

lemmas *sinvar-valid-remove-SOME-offending-flows* =
SecurityInvariant-preliminaries.sinvar-valid-remove-SOME-offending-flows[*OF SecurityInvariant-preliminariesD*, *simplified subst-offending-flows*]

c-offending-flows m (|*nodes* = *nodesG*, *edges* = *edgesG*|) ≠ {} ⟹ *c-sinvar m* (|*nodes* = *nodesG*, *edges* = *edgesG* − (*SOME F*. *F* ∈ *c-offending-flows m* (|*nodes* = *nodesG*, *edges* = *edgesG*|))|)

lemmas *sinvar-valid-remove-minimalize-offending-overapprox* =
SecurityInvariant-preliminaries.sinvar-valid-remove-minimalize-offending-overapprox[*OF SecurityInvariant-preliminariesD*, *simplified subst-offending-flows*]

⟦*wf-graph* (|*nodes* = *nodesG*, *edges* = *edgesG*|); ¬ *c-sinvar m* (|*nodes* = *nodesG*, *edges* = *edgesG*|); *set Es* = *edgesG*; *distinct Es*⟧ ⟹ *c-sinvar m* (|*nodes* = *nodesG*, *edges* = *edgesG* − *set* (*SecurityInvariant-withOffendingFlows.minimalize-offending-overapprox* (λ*G nP*. *c-sinvar m G*) *Es* [] (|*nodes* = *nodesG*, *edges* = *edgesG*|) *nP*)|)

lemmas *empty-offending-contra* =
SecurityInvariant-withOffendingFlows.empty-offending-contra[**where** *sinvar*=(λ*G nP*. *c-sinvar m G*), *simplified subst-offending-flows*]

⟦*F* ∈ *c-offending-flows m G*; *F* = {}⟧ ⟹ *False*

lemmas *Un-set-offending-flows-bound-minus-subseteq* =
SecurityInvariant-preliminaries.Un-set-offending-flows-bound-minus-subseteq[*OF SecurityInvariant-preliminariesD*, *simplified subst-offending-flows*]

⟦*wf-graph* (|*nodes* = *V*, *edges* = *E*|); ⋃ (*c-offending-flows m* (|*nodes* = *V*, *edges* = *E*|)) ⊆ *X*⟧ ⟹ ⋃ (*c-offending-flows m* (|*nodes* = *V*, *edges* = *E* − *E′*|)) ⊆ *X* − *E′*

lemmas *Un-set-offending-flows-bound-minus-subseteq′* =
SecurityInvariant-preliminaries.Un-set-offending-flows-bound-minus-subseteq′[*OF SecurityInvariant-preliminariesD*, *simplified subst-offending-flows*]

⟦*wf-graph* (|*nodes* = *V*, *edges* = *E*|); ⋃ (*c-offending-flows m* (|*nodes* = *V*, *edges* = *E*|)) ⊆ *X*⟧ ⟹ ⋃ (*c-offending-flows m* (|*nodes* = *V*, *edges* = *E* − *E′*|)) ⊆ *X* − *E′*

end

thm *configured-SecurityInvariant-def*

configured-SecurityInvariant m ≡ (∀ *G*. *c-offending-flows m G* = {*F*. *F* ⊆ *edges G* ∧ ¬ *c-sinvar m G* ∧ *c-sinvar m* (*delete-edges G F*) ∧ (∀(*e1*, *e2*)∈*F*. ¬ *c-sinvar m* (*add-edge e1 e2* (*delete-edges G F*)))}) ∧ (∀ *N*. *wf-graph* (|*nodes* = *N*, *edges* = {}|) ⟶ *c-sinvar m* (|*nodes* = *N*, *edges* = {}|)) ∧ (∀ *N E E′*. *wf-graph* (|*nodes* = *N*, *edges* = *E*|) ⟶ *E′* ⊆ *E* ⟶ *c-sinvar m* (|*nodes* = *N*, *edges* = *E*|) ⟶ *c-sinvar m* (|*nodes* = *N*, *edges* = *E′*|))

thm *configured-SecurityInvariant.mono-sinvar*

⟦*configured-SecurityInvariant m*; *wf-graph* (|*nodes* = *N*, *edges* = *E*|); *E′* ⊆ *E*; *c-sinvar m* (|*nodes* = *N*, *edges* = *E*|)⟧ ⟹ *c-sinvar m* (|*nodes* = *N*, *edges* = *E′*|)

Naming convention: m :: network security requirement M :: network security requirement list

The function *new-configured-SecurityInvariant* takes some tuple and if it returns a result, the locale assumptions are automatically fulfilled.

theorem *new-configured-SecurityInvariant-sound*:
⟦ *new-configured-SecurityInvariant* (*sinvar*, *defbot*, *receiver-violation*, *nP*) = *Some m* ⟧ ⟹
configured-SecurityInvariant m
proof −
assume *a*: *new-configured-SecurityInvariant* (*sinvar*, *defbot*, *receiver-violation*, *nP*) = *Some m*
hence *NetModel*: *SecurityInvariant sinvar defbot receiver-violation*
by(*simp add*: *new-configured-SecurityInvariant.simps split*: *if-split-asm*)
hence *NetModel-p*: *SecurityInvariant-preliminaries sinvar* **by**(*simp add*: *SecurityInvariant-def*)

from *a* **have** *c-eval*: *c-sinvar m* = (λ*G*. *sinvar G nP*)
and *c-offending*: *c-offending-flows m* = (λ*G*. *SecurityInvariant-withOffendingFlows.set-offending-flows sinvar G nP*)
and *c-isIFS m* = *receiver-violation*
by(*auto simp add*: *new-configured-SecurityInvariant.simps NetModel split*: *if-split-asm*)

have *monoI*: *SecurityInvariant-withOffendingFlows.sinvar-mono sinvar*
apply(*simp add*: *SecurityInvariant-withOffendingFlows.sinvar-mono-def*, *clarify*)
by(*fact SecurityInvariant-preliminaries.mono-sinvar*[*OF NetModel-p*])
from *SecurityInvariant-withOffendingFlows.valid-empty-edges-iff-exists-offending-flows*[*OF monoI*, *symmetric*]
SecurityInvariant-preliminaries.defined-offending[*OF NetModel-p*]
have *eval-empty-graph*: ⋀ *N nP*. *wf-graph* (|*nodes* = *N*, *edges* = {}|) ⟹ *sinvar* (|*nodes* = *N*, *edges* = {}|) *nP*
by *fastforce*

show *?thesis*
apply(*unfold-locales*)
apply(*simp add*: *c-eval c-offending SecurityInvariant-withOffendingFlows.set-offending-flows-def SecurityInvariant-withOffendingFlows.is-offending-flows-min-set-def SecurityInvariant-withOffendingFlows.is-offending-*
apply(*simp add*: *c-eval eval-empty-graph*)
apply(*simp add*: *c-eval*,*drule*(*3*) *SecurityInvariant-preliminaries.mono-sinvar*[*OF NetModel-p*])
done
qed

All security invariants are valid according to the definition

definition *valid-reqs* :: (′*v*::*vertex*) *SecurityInvariant-configured list* ⇒ *bool* **where**
valid-reqs M ≡ ∀ *m* ∈ *set M*. *configured-SecurityInvariant m*

7.2 Algorithms

A (generic) security invariant corresponds to a type of security requirements (type: ′*v graph* ⇒ (′*v* ⇒ ′*a*) ⇒ *bool*). A configured security invariant is a security requirement in a scenario specific setting (type: ′*v graph* ⇒ *bool*). I.e., it is a security requirement as listed in the requirements document. All security requirements are fulfilled for a fixed policy *G* if all security requirements are fulfilled for *G*.

Get all possible offending flows from all security requirements

definition *get-offending-flows* :: ′*v SecurityInvariant-configured list* ⇒ ′*v graph* ⇒ ((′*v* × ′*v*) *set set*) **where**
get-offending-flows M G = (⋃ *m*∈*set M*. *c-offending-flows m G*)

definition *all-security-requirements-fulfilled* :: (*'v*::*vertex*) *SecurityInvariant-configured list* ⇒ *'v graph* ⇒ *bool* **where**
all-security-requirements-fulfilled M G ≡ ∀ *m* ∈ *set M*. (*c-sinvar m*) *G*

Generate a valid topology from the security requirements

fun *generate-valid-topology* :: *'v SecurityInvariant-configured list* ⇒ *'v graph* ⇒ *'v graph* **where**
generate-valid-topology [] *G* = *G* |
generate-valid-topology (*m#Ms*) *G* = *delete-edges* (*generate-valid-topology Ms G*) (⋃ (*c-offending-flows m G*))

— return all Access Control Strategy models from a list of models
definition *get-ACS* :: (*'v*::*vertex*) *SecurityInvariant-configured list* ⇒ *'v SecurityInvariant-configured list* **where**
get-ACS M ≡ [*m* ← *M*. ¬ *c-isIFS m*]
— return all Information Flows Strategy models from a list of models
definition *get-IFS* :: (*'v*::*vertex*) *SecurityInvariant-configured list* ⇒ *'v SecurityInvariant-configured list* **where**
get-IFS M ≡ [*m* ← *M*. *c-isIFS m*]
lemma *get-ACS-union-get-IFS*: *set* (*get-ACS M*) ∪ *set* (*get-IFS M*) = *set M*
by(*auto simp add*: *get-ACS-def get-IFS-def*)

7.3 Lemmata

lemma *valid-reqs1*: *valid-reqs* (*m* # *M*) ⟹ *configured-SecurityInvariant m*
by(*simp add*: *valid-reqs-def*)
lemma *valid-reqs2*: *valid-reqs* (*m* # *M*) ⟹ *valid-reqs M*
by(*simp add*: *valid-reqs-def*)
lemma *get-offending-flows-alt1*: *get-offending-flows M G* = ⋃ {*c-offending-flows m G* | *m*. *m* ∈ *set M*}
apply(*simp add*: *get-offending-flows-def*)
by *fastforce*
lemma *get-offending-flows-un*: ⋃(*get-offending-flows M G*) = (⋃*m*∈*set M*. ⋃(*c-offending-flows m G*))
apply(*simp add*: *get-offending-flows-def*)
by *blast*

lemma *all-security-requirements-fulfilled-mono*:
⟦ *valid-reqs M*; *E'* ⊆ *E*; *wf-graph* (| *nodes* = *V*, *edges* = *E* |) ⟧ ⟹
all-security-requirements-fulfilled M (| *nodes* = *V*, *edges* = *E* |) ⟹
all-security-requirements-fulfilled M (| *nodes* = *V*, *edges* = *E'* |)
apply(*induction M arbitrary*: *E' E*)
apply(*simp-all add*: *all-security-requirements-fulfilled-def*)
apply(*rename-tac m M E' E*)
apply(*rule conjI*)
apply(*erule*(*2*) *configured-SecurityInvariant.mono-sinvar*[*OF valid-reqs1*])
apply(*simp-all*)
apply(*drule valid-reqs2*)
apply *blast*
done

7.4 generate valid topology

lemma *generate-valid-topology-nodes*:

nodes (*generate-valid-topology M G*) = (*nodes G*)
apply(*induction M arbitrary*: *G*)
by(*simp-all add*: *graph-ops*)

lemma *generate-valid-topology-def-alt*:
generate-valid-topology M G = *delete-edges G* ($\bigcup$ (*get-offending-flows M G*))
proof(*induction M arbitrary*: *G*)
case *Nil*
thus *?case* **by**(*simp add*: *get-offending-flows-def*)
next
case (*Cons m M*)
from *Cons*[*simplified delete-edges-simp2 get-offending-flows-def*]
have *edges* (*generate-valid-topology M G*) = *edges G* − $\bigcup$($\bigcup$ *m*∈*set M*. *c-offending-flows m G*)
by (*metis graph.select-convs*(*2*))
thus *?case*
apply(*simp add*: *get-offending-flows-def delete-edges-simp2*)
apply(*rule*)
apply(*simp add*: *generate-valid-topology-nodes*)
by *blast*
qed

lemma *wf-graph-generate-valid-topology*: *wf-graph G* ⟹ *wf-graph* (*generate-valid-topology M G*)
proof(*induction M arbitrary*: *G*)
qed(*simp-all*)

lemma *generate-valid-topology-mono-models*:
edges (*generate-valid-topology* (*m*#*M*) ⦇ *nodes* = *V*, *edges* = *E* ⦈) ⊆ *edges* (*generate-valid-topology M* ⦇ *nodes* = *V*, *edges* = *E* ⦈)
proof(*induction M arbitrary*: *E m*)
case *Nil* **thus** *?case* **by**(*simp add*: *delete-edges-simp2*)
case *Cons* **thus** *?case* **by**(*simp add*: *delete-edges-simp2*)
qed

lemma *generate-valid-topology-subseteq-edges*:
edges (*generate-valid-topology M G*) ⊆ (*edges G*)
proof(*induction M arbitrary*: *G*)
case *Cons* **thus** *?case* **by** (*simp add*: *delete-edges-simp2*) *blast*
qed(*simp*)

generate-valid-topology generates a valid topology (Policy)!

theorem *generate-valid-topology-sound*:
⟦ *valid-reqs M*; *wf-graph* ⦇*nodes* = *V*, *edges* = *E*⦈ ⟧ ⟹
all-security-requirements-fulfilled M (*generate-valid-topology M* ⦇*nodes* = *V*, *edges* = *E*⦈)
proof(*induction M arbitrary*: *V E*)
case *Nil*
thus *?case* **by**(*simp add*: *all-security-requirements-fulfilled-def*)
next
case (*Cons m M*)
from *valid-reqs1*[*OF Cons*(*2*)] **have** *validReq*: *configured-SecurityInvariant m* .

from *Cons*(*3*) **have** *valid-rmUnOff*: *wf-graph* ⦇*nodes* = *V*, *edges* = *E* − $\bigcup$(*c-offending-flows m* ⦇*nodes* = *V*, *edges* = *E*⦈) ⦈
by(*simp add*: *wf-graph-remove-edges*)

from *configured-SecurityInvariant.sinvar-valid-remove-flattened-offending-flows*[*OF validReq Cons*(*3*)]
have *valid-eval-rmUnOff*: *c-sinvar m* (|*nodes* = *V*, *edges* = *E* − ⋃(*c-offending-flows m* (|*nodes* = *V*, *edges* = *E*|)) |) **.**

from *generate-valid-topology-subseteq-edges* **have** *edges-gentopo-subseteq*:
edges (*generate-valid-topology M* (|*nodes* = *V*, *edges* = *E*|)) − ⋃(*c-offending-flows m* (|*nodes* = *V*, *edges* = *E*|))
⊆
E − ⋃(*c-offending-flows m* (|*nodes* = *V*, *edges* = *E*|)) **by** *fastforce*

from *configured-SecurityInvariant.mono-sinvar*[*OF validReq valid-rmUnOff edges-gentopo-subseteq valid-eval-rmUnOff*]
have *c-sinvar m* (|*nodes* = *V*, *edges* = (*edges* (*generate-valid-topology M* (|*nodes* = *V*, *edges* = *E*|))) − ⋃(*c-offending-flows m* (|*nodes* = *V*, *edges* = *E*|)) |) **.**
from *this* **have** *goal1*:
c-sinvar m (*delete-edges* (*generate-valid-topology M* (|*nodes* = *V*, *edges* = *E*|)) (⋃(*c-offending-flows m* (|*nodes* = *V*, *edges* = *E*|))))
by(*simp add*: *delete-edges-simp2 generate-valid-topology-nodes*)

from *valid-reqs2*[*OF Cons*(*2*)] **have** *valid-reqs M* **.**
from *Cons.IH*[*OF* ‹*valid-reqs M*› *Cons*(*3*)] **have** *IH*:
all-security-requirements-fulfilled M (*generate-valid-topology M* (|*nodes* = *V*, *edges* = *E*|)) **.**

have *generate-valid-topology-EX-graph-record*:
∃ *hypE*. (*generate-valid-topology M* (|*nodes* = *V*, *edges* = *E*|)) = (|*nodes* = *V*, *edges* = *hypE*|)

apply(*induction M arbitrary*: *V E*)
by(*simp-all add*: *delete-edges-simp2 generate-valid-topology-nodes*)

from *generate-valid-topology-EX-graph-record* **obtain** *E-IH* **where** *E-IH-prop*:
(*generate-valid-topology M* (|*nodes* = *V*, *edges* = *E*|)) = (|*nodes* = *V*, *edges* = *E-IH*|) **by** *blast*

from *wf-graph-generate-valid-topology*[*OF Cons*(*3*)] *E-IH-prop*
have *valid-G-E-IH*: *wf-graph* (|*nodes* = *V*, *edges* = *E-IH*|) **by** *metis*

— *all-security-requirements-fulfilled M* (|*nodes* = *V*, *edges* = *E-IH*|)
— *?E′* ⊆ *E-IH* ⟹ *all-security-requirements-fulfilled M* (|*nodes* = *V*, *edges* = *?E′*|)

from *all-security-requirements-fulfilled-mono*[*OF* ‹*valid-reqs M*› - *valid-G-E-IH IH*[*simplified E-IH-prop*]] **have** *mono-rule*:
⋀ *E′*. *E′* ⊆ *E-IH* ⟹ *all-security-requirements-fulfilled M* (|*nodes* = *V*, *edges* = *E′*|) **.**

have *all-security-requirements-fulfilled M*
(*delete-edges* (*generate-valid-topology M* (|*nodes* = *V*, *edges* = *E*|)) (⋃(*c-offending-flows m* (|*nodes* = *V*, *edges* = *E*|))))
apply(*subst E-IH-prop*)
apply(*simp add*: *delete-edges-simp2*)
apply(*rule mono-rule*)
by *fast*

from *this* **have** *goal2*:
(∀ *ma*∈*set M*.

c-sinvar ma (*delete-edges* (*generate-valid-topology M* (|*nodes* = *V*, *edges* = *E*|)) (⋃(*c-offending-flows m* (|*nodes* = *V*, *edges* = *E*|)))))
by(*simp add*: *all-security-requirements-fulfilled-def*)

from *goal1 goal2*
show *all-security-requirements-fulfilled* (*m* # *M*) (*generate-valid-topology* (*m* # *M*) (|*nodes* = *V*, *edges* = *E*|))
by (*simp add*: *all-security-requirements-fulfilled-def*)
qed

lemma *generate-valid-topology-as-set*:
generate-valid-topology M G = *delete-edges G* (⋃*m* ∈ *set M*. (⋃ (*c-offending-flows m G*)))
apply(*induction M arbitrary*: *G*)
apply(*simp-all add*: *delete-edges-simp2 generate-valid-topology-nodes*) **by** *fastforce*

lemma *c-offending-flows-subseteq-edges*: *configured-SecurityInvariant m* ⟹ ⋃(*c-offending-flows m G*) ⊆ *edges G*
apply(*clarify*)
apply(*simp only*: *configured-SecurityInvariant.valid-c-offending-flows*)
apply(*thin-tac configured-SecurityInvariant x* **for** *x*)
by *auto*

Does it also generate a maximum topology? It does, if the security invariants are in ENF-form. That means, if all security invariants can be expressed as a predicate over the edges, ∃*P*. ∀*G*. *c-sinvar m G* = (∀(*v1*, *v2*)∈*edges G*. *P* (*v1*, *v2*))

definition *max-topo* :: ('*v*::*vertex*) *SecurityInvariant-configured list* ⇒ '*v graph* ⇒ *bool* **where**
max-topo M G ≡ *all-security-requirements-fulfilled M G* ∧ (
∀ (*v1*, *v2*) ∈ (*nodes G* × *nodes G*) − (*edges G*). ¬ *all-security-requirements-fulfilled M* (*add-edge v1 v2 G*))

lemma *unique-offending-obtain*:
assumes *m*: *configured-SecurityInvariant m* **and** *unique*: *c-offending-flows m G* = {*F*}
obtains *P* **where** *F* = {(*v1*, *v2*) ∈ *edges G*. ¬ *P* (*v1*, *v2*)} **and** *c-sinvar m G* = (∀(*v1*,*v2*) ∈ *edges G*. *P* (*v1*, *v2*)) **and**
(∀(*v1*,*v2*) ∈ *edges G* − *F*. *P* (*v1*, *v2*))
proof −
assume *EX*: (⋀*P*. *F* = {(*v1*, *v2*). (*v1*, *v2*) ∈ *edges G* ∧ ¬ *P* (*v1*, *v2*)} ⟹ *c-sinvar m G* = (∀(*v1*, *v2*)∈*edges G*. *P* (*v1*, *v2*)) ⟹ ∀(*v1*, *v2*)∈*edges G* − *F*. *P* (*v1*, *v2*) ⟹ *thesis*)

from *unique c-offending-flows-subseteq-edges*[*OF m*] **have** *F* ⊆ *edges G* **by** *force*
from *this* **obtain** *P* **where** *F* = {*e* ∈ *edges G*. ¬ *P e*} **by** (*metis double-diff set-diff-eq subset-refl*)
hence *1*: *F* = {(*v1*, *v2*) ∈ *edges G*. ¬ *P* (*v1*, *v2*)} **by** *auto*

from *configured-SecurityInvariant.valid-c-offending-flows*[*OF m*] **have** *c-offending-flows m G* =
{*F*. *F* ⊆ *edges G* ∧ ¬ *c-sinvar m G* ∧ *c-sinvar m* (*delete-edges G F*) ∧
(∀(*e1*, *e2*)∈*F*. ¬ *c-sinvar m* (*add-edge e1 e2* (*delete-edges G F*)))} **.**

from *this unique* **have** ¬ *c-sinvar m G* **and** *2*: *c-sinvar m* (*delete-edges G F*) **and**
3: (∀(*e1*, *e2*)∈*F*. ¬ *c-sinvar m* (*add-edge e1 e2* (*delete-edges G F*))) **by** *auto*

from *this* ‹*F* = {*e* ∈ *edges G*. ¬ *P e*}› **have** *x3*: ∀ *e* ∈ *edges G* − *F*. *P e* **by** (*metis* (*lifting*) *mem-Collect-eq set-diff-eq*)
hence *4*: ∀(*v1*,*v2*) ∈ *edges G* − *F*. *P* (*v1*, *v2*) **by** *blast*

have *F* ≠ {} **by** (*metis assms*(*1*) *assms*(*2*) *configured-SecurityInvariant.empty-offending-contra insertCI*)
from *this* ‹*F* = {*e* ∈ *edges G*. ¬ *P e*}› ‹¬ *c-sinvar m G*› **have** *5*: *c-sinvar m G* = (∀ (*v1*,*v2*) ∈ *edges G*. *P* (*v1*, *v2*))
apply(*simp add*: *graph-ops*)
by(*blast*)

from *EX*[*of P*] *unique 1 x3 5* **show** *?thesis* **by** *fast*
qed

lemma *enf-offending-flows*:
assumes *vm*: *configured-SecurityInvariant m* **and** *enf*: ∀ *G*. *c-sinvar m G* = (∀ *e* ∈ *edges G*. *P e*)
shows ∀ *G*. *c-offending-flows m G* = (*if c-sinvar m G then* {} *else* {{*e* ∈ *edges G*. ¬ *P e*}})
proof −
{ **fix** *G*
from *vm configured-SecurityInvariant.valid-c-offending-flows* **have** *offending-formaldef*:
c-offending-flows m G =
{*F*. *F* ⊆ *edges G* ∧ ¬ *c-sinvar m G* ∧ *c-sinvar m* (*delete-edges G F*) ∧
(∀ (*e1*, *e2*)∈*F*. ¬ *c-sinvar m* (*add-edge e1 e2* (*delete-edges G F*)))} **by** *auto*
have *c-offending-flows m G* = (*if c-sinvar m G then* {} *else* {{*e* ∈ *edges G*. ¬ *P e*}})
proof(*cases c-sinvar m G*)
case *True* **thus** *?thesis* — {}
by(*simp add*: *offending-formaldef*)
next
case *False* **thus** *?thesis* **by**(*auto simp add*: *offending-formaldef graph-ops enf*)
qed
} **thus** *?thesis* **by** *simp*
qed

lemma *enf-not-fulfilled-if-in-offending*:
assumes *validRs*: *valid-reqs M*
and *wfG*: *wf-graph G*
and *enf*: ∀ *m* ∈ *set M*. ∃ *P*. ∀ *G*. *c-sinvar m G* = (∀ *e* ∈ *edges G*. *P e*)
shows ∀ *x* ∈ (⋃ *m*∈*set M*. ⋃ (*c-offending-flows m* (*fully-connected G*))).
¬ *all-security-requirements-fulfilled M* ⦇ *nodes* = *V*, *edges* = *insert x E*⦈
unfolding *all-security-requirements-fulfilled-def*
proof(*simp*, *clarify*, *rename-tac m F a b*)
let *?G*=(*fully-connected G*)
fix *m F v1 v2*
assume *m* ∈ *set M* **and** *F* ∈ *c-offending-flows m ?G* **and** (*v1*, *v2*) ∈ *F*

from *validRs* **have** *valid-mD*:⋀*m*. *m* ∈ *set M* ⟹ *configured-SecurityInvariant m*
by(*simp add*: *valid-reqs-def*)

from ‹*m* ∈ *set M*› *valid-mD* **have** *configured-SecurityInvariant m* **by** *simp*

from *enf* ‹*m* ∈ *set M*› **obtain** *P* **where** *enf-m*: ∀ *G*. *c-sinvar m G* = (∀ *e*∈*edges G*. *P e*) **by** *blast*

from ‹(*v1*, *v2*) ∈ *F*› **have** *F* ≠ {} **by** *auto*

from *enf-offending-flows*[*OF* ‹*configured-SecurityInvariant m*› ‹∀ *G*. *c-sinvar m G* = (∀ *e*∈*edges G*. *P e*)›] **have**

offending: $\bigwedge G$. *c-offending-flows m G* = (*if c-sinvar m G then* {} *else* {{*e* ∈ *edges G*. ¬ *P e*}})
by *simp*
from ‹*F* ∈ *c-offending-flows m ?G*› ‹*F* ≠ {}› **have** *F* = {*e* ∈ *edges ?G*. ¬ *P e*}
by(*simp split*: *if-split-asm add*: *offending*)
from *this* ‹(*v1*, *v2*) ∈ *F*› **have** ¬ *P* (*v1*, *v2*) **by** *simp*

from *this enf-m* **have** ¬ *c-sinvar m* (|*nodes* = *V*, *edges* = *insert* (*v1*, *v2*) *E*|) **by**(*simp*)
thus ∃ *m*∈*set M*. ¬ *c-sinvar m* (|*nodes* = *V*, *edges* = *insert* (*v1*, *v2*) *E*|) **using** ‹*m* ∈ *set M*›
apply(*rule-tac x*=*m* **in** *bexI*)
by *simp-all*
qed

theorem *generate-valid-topology-max-topo*: ⟦ *valid-reqs M*; *wf-graph G*;
∀ *m* ∈ *set M*. ∃ *P*. ∀ *G*. *c-sinvar m G* = (∀ *e* ∈ *edges G*. *P e*)⟧ ⟹
max-topo M (*generate-valid-topology M* (*fully-connected G*))
proof −
let *?G*=(*fully-connected G*)
assume *validRs*: *valid-reqs M*
and *wfG*: *wf-graph G*
and *enf*: ∀ *m* ∈ *set M*. ∃ *P*. ∀ *G*. *c-sinvar m G* = (∀ *e* ∈ *edges G*. *P e*)

obtain *V E* **where** *VE-prop*: (| *nodes* = *V*, *edges* = *E* |) = *generate-valid-topology M ?G* **by** (*metis graph.cases*)
hence *VE-prop-asset*:
(| *nodes* = *V*, *edges* = *E* |) = (| *nodes* = *V*, *edges* = *V* × *V* − (⋃ *m*∈*set M*. ⋃ (*c-offending-flows m ?G*))|)
by(*simp add*: *fully-connected-def generate-valid-topology-as-set delete-edges-simp2*)

from *VE-prop-asset* **have** *E-prop*: *E* = *V* × *V* − (⋃ *m*∈*set M*. ⋃ (*c-offending-flows m ?G*)) **by** *fast*
from *VE-prop* **have** *V-prop*: *nodes G* = *V*
by (*simp add*: *fully-connected-def delete-edges-simp2 generate-valid-topology-def-alt*)
from *VE-prop* **have** *V-full-prop*: *nodes* (*generate-valid-topology M ?G*) = *V* **by** (*metis graph.select-convs*(*1*))
from *VE-prop* **have** *E-full-prop*: *edges* (*generate-valid-topology M ?G*) = *E* **by** (*metis graph.select-convs*(*2*))

from *VE-prop wf-graph-generate-valid-topology*[*OF fully-connected-wf*[*OF wfG*]]
have *wfG-VE*: *wf-graph* (| *nodes* = *V*, *edges* = *E* |) **by** *force*

from *generate-valid-topology-sound*[*OF validRs wfG-VE*] *fully-connected-wf*[*OF wfG*] **have** *VE-all-valid*:

all-security-requirements-fulfilled M (| *nodes* = *V*, *edges* = *V* × *V* − (⋃ *m*∈*set M*. ⋃ (*c-offending-flows m ?G*))|)
by (*metis VE-prop VE-prop-asset fully-connected-def generate-valid-topology-sound validRs*)
hence *goal1*: *all-security-requirements-fulfilled M* (*generate-valid-topology M* (*fully-connected G*)) **by** (*metis VE-prop VE-prop-asset*)

from *validRs* **have** *valid-mD*: $\bigwedge m$. *m* ∈ *set M* ⟹ *configured-SecurityInvariant m*
by(*simp add*: *valid-reqs-def*)

from *c-offending-flows-subseteq-edges*[**where** *G*=*?G*] *validRs* **have** *hlp1*: (⋃ *m*∈*set M*. ⋃ (*c-offending-flows m ?G*)) ⊆ *V* × *V*
apply(*simp add*: *fully-connected-def V-prop*)
using *valid-reqs-def* **by** *blast*

have $\bigwedge A\ B.\ A - (A - B) = B \cap A$ **by** *fast*
from *E-prop hlp1* **have** $V \times V - E = (\bigcup m{\in}set\ M.\ \bigcup(c\text{-}offending\text{-}flows\ m\ ?G))$ **by** *force*

from *enf-not-fulfilled-if-in-offending*[*OF validRs wfG enf*]
have $\forall(v1, v2) \in (\bigcup m{\in}set\ M.\ \bigcup(c\text{-}offending\text{-}flows\ m\ ?G))$.
¬ *all-security-requirements-fulfilled M* (| *nodes* = *V*, *edges* = *E* ∪ {(*v1*, *v2*)}|) **by** *simp*

from *this* ‹$V \times V - E = (\bigcup m{\in}set\ M.\ \bigcup(c\text{-}offending\text{-}flows\ m\ ?G))$› **have** $\forall(v1, v2) \in V \times V - E$.
¬ *all-security-requirements-fulfilled M* (| *nodes* = *V*, *edges* = *E* ∪ {(*v1*, *v2*)}|) **by** *simp*
hence *goal2*: (∀(*v1*, *v2*)∈*nodes* (*generate-valid-topology M ?G*) × *nodes* (*generate-valid-topology M ?G*) −
edges (*generate-valid-topology M ?G*).
¬ *all-security-requirements-fulfilled M* (*add-edge v1 v2* (*generate-valid-topology M ?G*)))
proof(*unfold V-full-prop E-full-prop graph-ops*)
assume *a*: ∀(*v1*, *v2*)∈*V* × *V* − *E*. ¬ *all-security-requirements-fulfilled M* (|*nodes* = *V*, *edges* = *E* ∪ {(*v1*, *v2*)}|)
have ∀(*v1*, *v2*)∈*V* × *V* − *E*. *V* ∪ {*v1*, *v2*} = *V* **by** *blast*
hence ∀(*v1*, *v2*)∈*V* × *V* − *E*. (|*nodes* = *V* ∪ {*v1*, *v2*}, *edges* = {(*v1*, *v2*)} ∪ *E*|) = (|*nodes* = *V*, *edges* = *E* ∪ {(*v1*, *v2*)}|) **by** *blast*
from *this a* **show** ∀(*v1*, *v2*)∈*V* × *V* − *E*. ¬ *all-security-requirements-fulfilled M* (|*nodes* = *V* ∪ {*v1*, *v2*}, *edges* = {(*v1*, *v2*)} ∪ *E*|)
— TODO: this should be trivial ...
apply(*simp*)
apply(*rule ballI*)
apply(*erule-tac x=x* **and** *A=V × V − E* **in** *ballE*)
prefer *2* **apply**(*simp*; *fail*)
apply(*erule-tac x=x* **and** *A=V × V − E* **in** *ballE*)
prefer *2* **apply**(*simp*; *fail*)
apply(*clarify*)
by *presburger*
qed

from *goal1 goal2* **show** *?thesis*
unfolding *max-topo-def* **by** *presburger*
qed

lemma *enf-all-valid-policy-subset-of-max*:
assumes *validRs*: *valid-reqs M*
and *wfG*: *wf-graph G*
and *enf*: ∀*m* ∈ *set M*. ∃*P*. ∀*G*. *c-sinvar m G* = (∀*e* ∈ *edges G*. *P e*)
and *nodesG′*: *nodes G* = *nodes G′*
shows ⟦ *wf-graph G′*;
all-security-requirements-fulfilled M G′⟧ ⟹
edges G′ ⊆ *edges* (*generate-valid-topology M* (*fully-connected G*))
using *nodesG′* **apply**(*cases generate-valid-topology M* (*fully-connected G*), *rename-tac V E*, *simp*)
apply(*cases G′*, *rename-tac V′ E′*, *simp*)
apply(*subgoal-tac nodes G* = *V*)
prefer *2*
apply (*metis fully-connected-def generate-valid-topology-nodes graph.select-convs*(*1*))
apply(*simp*)
proof(*rule ccontr*)
fix *V E V′ E′*

assume *a5*: *all-security-requirements-fulfilled M* ⦇*nodes* = *V*, *edges* = *E′*⦈ **and**
a6: *generate-valid-topology M* (*fully-connected G*) = ⦇*nodes* = *V*, *edges* = *E*⦈ **and**
a10: *wf-graph* ⦇*nodes* = *V*, *edges* = *E′*⦈ **and**
contr: ¬ *E′* ⊆ *E*

from *wfG a6* **have** *wf-graph* ⦇*nodes* = *V*, *edges* = *E*⦈
by (*metis fully-connected-wf wf-graph-generate-valid-topology*)
with *a10* **have** *EE′subsets*: *fst* ‘ *E* ⊆ *V* ∧ *snd* ‘ *E* ⊆ *V* ∧ *fst* ‘ *E′* ⊆ *V* ∧ *snd* ‘ *E′* ⊆ *V*
by(*simp add*: *wf-graph-def*)
hence *EE′subsets′*: *E* ⊆ *V* × *V* ∧ *E′* ⊆ *V* × *V* **by** *auto*

from *generate-valid-topology-max-topo*[*OF validRs wfG enf*]
have *m1*: *all-security-requirements-fulfilled M* ⦇*nodes* = *V*, *edges* = *E*⦈ **and**
m2: (∀ *x*∈*V* × *V* − *E*. *case x of* (*v1*, *v2*) ⇒ ¬ *all-security-requirements-fulfilled M* (*add-edge v1 v2* ⦇*nodes* = *V*, *edges* = *E*⦈))
by(*simp add*: *max-topo-def a6*)+

from *m2* **have** *m2′*: ∀ *x*∈*V* × *V* − *E*. ¬ *all-security-requirements-fulfilled M* ⦇*nodes* = *V*, *edges* = *insert x E*⦈
apply(*simp add*: *add-edge-def*)
apply(*rule ballI*, *rename-tac x*)
apply(*erule-tac x*=*x* **in** *ballE*, *simp-all*)
apply(*case-tac x*, *simp*)
by (*simp add*: *insert-absorb*)

show *False*
proof(*cases V* = {})
case *True*
with *EE′subsets a10* **have** *E* = {} **and** *E′* = {}
by(*simp add*: *wf-graph-def*)+
with *True contr* **show** *?thesis* **by** *simp*
next
case *False*
with *EE′subsets′ contr* **obtain** *x* **where** *x*: *x* ∈ *E′* ∧ *x* ∉ *E* ∧ *x* ∈ *V* × *V*
by *blast*
from *m2′ x* **have** ¬ *all-security-requirements-fulfilled M* ⦇*nodes* = *V*, *edges* = *insert x E*⦈
by (*simp*)

from *a6 x* **have** *x-offedning*: *x* ∈ (⋃ *m*∈*set M*. ⋃(*c-offending-flows m* (*fully-connected G*)))
apply(*simp add*: *generate-valid-topology-as-set delete-edges-simp2 fully-connected-def*)
by *blast*

from *enf-not-fulfilled-if-in-offending*[*OF validRs wfG enf*] *x-offedning* **have**
1: ¬ *all-security-requirements-fulfilled M* ⦇*nodes* = *V*, *edges* = *insert x myE*⦈ **for** *myE* **by** *blast*

from *x* **have** *insertxE′*: *insert x E′* = *E′* **by** *blast*
with *a5* **have**
all-security-requirements-fulfilled M ⦇*nodes* = *V*, *edges* = *insert x E′*⦈ **by** *simp*
with *insertxE′ all-security-requirements-fulfilled-mono*[*OF validRs - a10 a5*] **have**
2: *all-security-requirements-fulfilled M* ⦇*nodes* = *V*, *edges* = *insert x* {}⦈ **by** *blast*
from *1 2* **show** *?thesis* **by** *blast*
qed
qed

7.5 More Lemmata

lemma (**in** *configured-SecurityInvariant*) *c-sinvar-valid-imp-no-offending-flows*:
c-sinvar m G $\Longrightarrow$ *c-offending-flows m G* = {}
by(*simp add*: *valid-c-offending-flows*)

lemma *all-security-requirements-fulfilled-imp-no-offending-flows*:
valid-reqs M $\Longrightarrow$ *all-security-requirements-fulfilled M G* $\Longrightarrow$ ($\bigcup$ *m*∈*set M*. $\bigcup$(*c-offending-flows m G*)) = {}
proof(*induction M*)
case *Cons* **thus** *?case*
unfolding *all-security-requirements-fulfilled-def*
apply(*simp*)
by(*blast dest*: *valid-reqs2 valid-reqs1 configured-SecurityInvariant.c-sinvar-valid-imp-no-offending-flows*)
qed(*simp*)

corollary *all-security-requirements-fulfilled-imp-get-offending-empty*:
valid-reqs M $\Longrightarrow$ *all-security-requirements-fulfilled M G* $\Longrightarrow$ *get-offending-flows M G* = {}
apply(*frule*(*1*) *all-security-requirements-fulfilled-imp-no-offending-flows*)
apply(*simp add*: *get-offending-flows-def*)
apply(*thin-tac all-security-requirements-fulfilled M G*)
apply(*simp add*: *valid-reqs-def*)
apply(*clarify*)
using *configured-SecurityInvariant.empty-offending-contra* **by** *fastforce*

corollary *generate-valid-topology-does-nothing-if-valid*:
⟦ *valid-reqs M*; *all-security-requirements-fulfilled M G*⟧ $\Longrightarrow$
generate-valid-topology M G = *G*
by(*simp add*: *generate-valid-topology-as-set graph-ops all-security-requirements-fulfilled-imp-no-offending-flows*)

lemma *mono-extend-get-offending-flows*: ⟦ *valid-reqs M*;
wf-graph (|*nodes* = *V*, *edges* = *E*|);
$E' \subseteq E$;
$F' \in$ *get-offending-flows M* (|*nodes* = *V*, *edges* = *E'*|) ⟧ $\Longrightarrow$
∃ *F*∈*get-offending-flows M* (|*nodes* = *V*, *edges* = *E*|). $F' \subseteq F$
proof(*induction M*)
case *Nil* **thus** *?case* **by**(*simp add*: *get-offending-flows-def*)
next
case (*Cons m M*)
from *Cons.prems* **have** *configured-SecurityInvariant m*
and *valid-reqs M* **using** *valid-reqs2 valid-reqs1* **by** *blast+*
from *Cons.prems*(*4*) **have**
$F' \in$ *c-offending-flows m* (|*nodes* = *V*, *edges* = *E'*|) ∨
($F' \in$ *get-offending-flows M* (|*nodes* = *V*, *edges* = *E'*|))
by(*simp add*: *get-offending-flows-def*)
from *this* **show** *?case*
proof(*elim disjE*, *goal-cases*)
case *1*
with ‹*configured-SecurityInvariant m*› *Cons.prems*(*2*,*3*,*4*) **obtain** *F* **where**
F∈*c-offending-flows m* (|*nodes* = *V*, *edges* = *E*|) **and** $F' \subseteq F$
by(*blast dest*: *configured-SecurityInvariant.mono-extend-set-offending-flows*)
hence *F*∈*get-offending-flows* (*m* # *M*) (|*nodes* = *V*, *edges* = *E*|)
by (*simp add*: *get-offending-flows-def*)

with ‹$F' \subseteq F$› **show** *?case* **by** *blast*
next
case *2* **with** *Cons* ‹*valid-reqs M*› **show** *?case* **by**(*simp add*: *get-offending-flows-def*) *blast*
qed
qed

lemma *get-offending-flows-subseteq-edges*: *valid-reqs M* $\Longrightarrow$ $F \in$ *get-offending-flows M* (|*nodes* = *V*, *edges* = *E*|) $\Longrightarrow$ $F \subseteq E$
apply(*induction M*)
apply(*simp add*: *get-offending-flows-def*)
apply(*simp add*: *get-offending-flows-def*)
apply(*frule valid-reqs2*, *drule valid-reqs1*)
apply(*simp add*: *configured-SecurityInvariant.valid-c-offending-flows*)
by *blast*

thm *configured-SecurityInvariant.offending-flows-union-mono*
lemma *get-offending-flows-union-mono*: ⟦*valid-reqs M*;
wf-graph (|*nodes* = *V*, *edges* = *E*|); $E' \subseteq E$ ⟧ $\Longrightarrow$
$\bigcup$(*get-offending-flows M* (|*nodes* = *V*, *edges* = *E'*|)) $\subseteq$ $\bigcup$(*get-offending-flows M* (|*nodes* = *V*, *edges* = *E*|))
apply(*induction M*)
apply(*simp add*: *get-offending-flows-def*)
apply(*frule valid-reqs2*, *drule valid-reqs1*)
apply(*drule*(*2*) *configured-SecurityInvariant.offending-flows-union-mono*)
apply(*simp add*: *get-offending-flows-def*)
by *auto*

thm *configured-SecurityInvariant.Un-set-offending-flows-bound-minus-subseteq'*
lemma *Un-set-offending-flows-bound-minus-subseteq'*:⟦*valid-reqs M*;
wf-graph (|*nodes* = *V*, *edges* = *E*|); $E' \subseteq E$;
$\bigcup$(*get-offending-flows M* (|*nodes* = *V*, *edges* = *E*|)) $\subseteq X$ ⟧ $\Longrightarrow$ $\bigcup$(*get-offending-flows M* (|*nodes* = *V*, *edges* = *E* − *E'*|)) $\subseteq X - E'$
proof(*induction M*)
case *Nil* **thus** *?case* **by** (*simp add*: *get-offending-flows-def*)
next
case (*Cons m M*)
from *Cons.prems*(*1*) *valid-reqs2* **have** *valid-reqs M* **by** *force*
from *Cons.prems*(*1*) *valid-reqs1* **have** *configured-SecurityInvariant m* **by** *force*
from *Cons.prems*(*4*) **have** $\bigcup$(*get-offending-flows M* (|*nodes* = *V*, *edges* = *E*|)) $\subseteq X$ **by**(*simp add*: *get-offending-flows-def*)
from *Cons.IH*[*OF* ‹*valid-reqs M*› *Cons.prems*(*2*) *Cons.prems*(*3*) ‹$\bigcup$(*get-offending-flows M* (|*nodes* = *V*, *edges* = *E*|)) $\subseteq X$›] **have** *IH*: $\bigcup$(*get-offending-flows M* (|*nodes* = *V*, *edges* = *E* − *E'*|)) $\subseteq X - E'$ **.**
from *Cons.prems*(*4*) **have** $\bigcup$(*c-offending-flows m* (|*nodes* = *V*, *edges* = *E*|)) $\subseteq X$ **by**(*simp add*: *get-offending-flows-def*)
from *configured-SecurityInvariant.Un-set-offending-flows-bound-minus-subseteq'*[*OF* ‹*configured-SecurityInvariant m*› *Cons.prems*(*2*) ‹$\bigcup$(*c-offending-flows m* (|*nodes* = *V*, *edges* = *E*|)) $\subseteq X$›] **have** $\bigcup$(*c-offending-flows m* (|*nodes* = *V*, *edges* = *E* − *E'*|)) $\subseteq X - E'$ **.**
from *this IH* **show** *?case* **by**(*simp add*: *get-offending-flows-def*)
qed

```
lemma ENF-uniquely-defined-offedning: valid-reqs M ⟹ wf-graph G ⟹
    ∀ m ∈ set M. ∃ P. ∀ G. c-sinvar m G = (∀ e ∈ edges G. P e) ⟹
    ∀ m ∈ set M. ∀ G. ¬ c-sinvar m G ⟶ (∃ OFF. c-offending-flows m G = {OFF})
apply −
apply(induction M)
 apply(simp; fail)
apply(rename-tac m M)
apply(frule valid-reqs1)
apply(drule valid-reqs2)
apply(simp)
apply(elim conjE)
apply(erule-tac x=m in ballE)
 apply(simp-all; fail)
apply(erule exE, rename-tac P)
apply(drule-tac P=P in enf-offending-flows)
 apply(simp; fail)
apply(simp; fail)
done

lemma assumes configured-SecurityInvariant m
     and ∀ G. ¬ c-sinvar m G ⟶ (∃ OFF. c-offending-flows m G = {OFF})
     shows ∃ OFF-P. ∀ G. c-offending-flows m G = (if c-sinvar m G then {} else {OFF-P G})
proof −
  from assms have ∃ OFF-P.
        c-offending-flows m G = (if c-sinvar m G then {} else {OFF-P G}) for G
    apply(erule-tac x=G in allE)
    apply(cases c-sinvar m G)
     apply(drule configured-SecurityInvariant.c-sinvar-valid-imp-no-offending-flows, simp)
     apply(simp; fail)
    apply(simp)
    by meson
  with assms show ?thesis by metis
qed
```

Hilber's eps operator example

```
  lemma (SOME x. x : {1::nat, 2, 3}) = x ⟹ x = 1 ∨ x = 2 ∨ x = 3
    proof −
     have (SOME x. x ∈ {1::nat, 2, 3}) ∈ {1::nat, 2, 3} unfolding some-in-eq by simp
     thus (SOME x. x : {1::nat, 2, 3}) = x ⟹ x = 1 ∨ x = 2 ∨ x = 3 by fast
    qed
```

Only removing one offending flow should be enough

```
  fun generate-valid-topology-SOME :: 'v SecurityInvariant-configured list ⇒ 'v graph ⇒ 'v graph
where
    generate-valid-topology-SOME [] G = G |
    generate-valid-topology-SOME (m#Ms) G = (if c-sinvar m G
      then generate-valid-topology-SOME Ms G
      else delete-edges (generate-valid-topology-SOME Ms G) (SOME F. F ∈ c-offending-flows m G)
      )

  lemma generate-valid-topology-SOME-nodes: nodes (generate-valid-topology-SOME M (|nodes = V,
edges = E|)) = V
    proof(induction M)
```

qed(*simp-all add*: *delete-edges-simp2*)

theorem *generate-valid-topology-SOME-sound*:
⟦ *valid-reqs M*; *wf-graph* (|*nodes* = *V*, *edges* = *E*|) ⟧ ⟹
all-security-requirements-fulfilled M (*generate-valid-topology-SOME M* (|*nodes* = *V*, *edges* = *E*|))
proof(*induction M*)
case *Nil*
thus *?case* **by**(*simp add*: *all-security-requirements-fulfilled-def*)
next
case (*Cons m M*)
from *valid-reqs1*[*OF Cons*(*2*)] **have** *validReq*: *configured-SecurityInvariant m* **.**

from *configured-SecurityInvariant.sinvar-valid-remove-SOME-offending-flows*[*OF validReq*] **have**
c-offending-flows m (|*nodes* = *V*, *edges* = *E*|) ≠ {} ⟹
c-sinvar m (|*nodes* = *V*, *edges* = *E* − (*SOME F*. *F* ∈ *c-offending-flows m* (|*nodes* = *V*, *edges* = *E*|))|) **.**

have *generate-valid-topology-SOME-edges*: *edges* (*generate-valid-topology-SOME M* (|*nodes* = *V*, *edges* = *E*|)) ⊆ *E*
for *M*::′*a SecurityInvariant-configured list* **and** *V E*
proof(*induction M*)
qed(*auto simp add*: *delete-edges-simp2*)

from *configured-SecurityInvariant.mono-sinvar*[*OF validReq Cons.prems*(*2*),
of edges (*generate-valid-topology-SOME M* (|*nodes* = *V*, *edges* = *E*|))]
generate-valid-topology-SOME-edges
have *c-sinvar m* (|*nodes* = *V*, *edges* = *E*|) ⟹
c-sinvar m (|*nodes* = *V*, *edges* = *edges* (*generate-valid-topology-SOME M* (|*nodes* = *V*, *edges* = *E*|))|)
by *simp*
moreover from *configured-SecurityInvariant.defined-offending*′[*OF validReq Cons.prems*(*2*)] **have** *not-sinvar-off*:
¬ *c-sinvar m* (|*nodes* = *V*, *edges* = *E*|) ⟹ *c-offending-flows m* (|*nodes* = *V*, *edges* = *E*|) ≠ {} **by** *blast*
ultimately have *goal-sinvar-m*:
c-offending-flows m (|*nodes* = *V*, *edges* = *E*|) = {} ⟹
c-sinvar m (*generate-valid-topology-SOME M* (|*nodes* = *V*, *edges* = *E*|))
using *generate-valid-topology-SOME-nodes*
by (*metis graph.select-convs*(*1*) *graph.select-convs*(*2*) *graph-eq-intro*)

from *valid-reqs2*[*OF Cons*(*2*)] **have** *valid-reqs M* **.**
from *Cons.IH*[*OF* ‹*valid-reqs M*› *Cons*(*3*)] **have** *IH*:
all-security-requirements-fulfilled M (*generate-valid-topology-SOME M* (|*nodes* = *V*, *edges* = *E*|)) **.**

have *goal-rm-SOME-m*: *c-offending-flows m* (|*nodes* = *V*, *edges* = *E*|) ≠ {} ⟹
c-sinvar m (*delete-edges* (*generate-valid-topology-SOME M* (|*nodes* = *V*, *edges* = *E*|))
(*SOME F*. *F* ∈ *c-offending-flows m* (|*nodes* = *V*, *edges* = *E*|)))
proof −
assume *a1*: *c-offending-flows m* (|*nodes* = *V*, *edges* = *E*|) ≠ {}
have *f2*: (∀ *r ra p*. ¬ *r* ⊆ *ra* ∨ (*p*::′*a* × ′*a*) ∉ *r* ∨ *p* ∈ *ra*) = (∀ *r ra p*. ¬ *r* ⊆ *ra* ∨ (*p*::′*a* × ′*a*) ∉ *r* ∨ *p* ∈ *ra*)
by *meson*
have *f3*: *wf-graph* (|*nodes* = *V*, *edges* = *E* − (*SOME r*. *r* ∈ *c-offending-flows m* (|*nodes* = *V*,

edges = *E*⦈)⦈
by (*simp add*: *Cons.prems*(*2*) *wf-graph-remove-edges*)
have *edges* (*generate-valid-topology-SOME M* ⦇*nodes* = *V*, *edges* = *E*⦈) − (*SOME r*. *r* ∈ *c-offending-flows m* ⦇*nodes* = *V*, *edges* = *E*⦈) ⊆ *E* − (*SOME r*. *r* ∈ *c-offending-flows m* ⦇*nodes* = *V*, *edges* = *E*⦈)
using *f2 generate-valid-topology-SOME-edges*[*of M V E*] **by** *blast*
then have *c-sinvar m* ⦇*nodes* = *V*, *edges* = *edges* (*generate-valid-topology-SOME M* ⦇*nodes* = *V*, *edges* = *E*⦈) − (*SOME r*. *r* ∈ *c-offending-flows m* ⦇*nodes* = *V*, *edges* = *E*⦈)⦈
using *f3 a1* ‹*c-offending-flows m* ⦇*nodes* = *V*, *edges* = *E*⦈ ≠ {} ⟹ *c-sinvar m* ⦇*nodes* = *V*, *edges* = *E* − (*SOME F*. *F* ∈ *c-offending-flows m* ⦇*nodes* = *V*, *edges* = *E*⦈)⦈› *configured-SecurityInvariant.negative-mono validReq* **by** *blast*
then show *c-sinvar m* (*delete-edges* (*generate-valid-topology-SOME M* ⦇*nodes* = *V*, *edges* = *E*⦈) (*SOME r*. *r* ∈ *c-offending-flows m* ⦇*nodes* = *V*, *edges* = *E*⦈))
by (*simp add*: *generate-valid-topology-SOME-nodes graph-ops*(*5*))
qed

have *wf-graph-generate-valid-topology-SOME*: *wf-graph G* ⟹ *wf-graph* (*generate-valid-topology-SOME M G*)
for *G*
apply(*cases G*)
apply(*simp add*: *wf-graph-def generate-valid-topology-SOME-nodes*)
using *generate-valid-topology-SOME-edges* **by** (*meson dual-order.trans image-mono rev-finite-subset*)

{ assume *notempty*: *c-offending-flows m* ⦇*nodes* = *V*, *edges* = *E*⦈ ≠ {}
hence ∃ *hypE*. (*generate-valid-topology-SOME M* ⦇*nodes* = *V*, *edges* = *E*⦈) = ⦇*nodes* = *V*, *edges* = *hypE*⦈
proof(*induction M arbitrary*: *V E*)
qed(*simp-all add*: *delete-edges-simp2 generate-valid-topology-SOME-nodes*)
from *this* **obtain** *E-IH* **where** *E-IH-prop*:
(*generate-valid-topology-SOME M* ⦇*nodes* = *V*, *edges* = *E*⦈) = ⦇*nodes* = *V*, *edges* = *E-IH*⦈ **by** *blast*

from *wf-graph-generate-valid-topology-SOME*[*OF Cons*(*3*)] *E-IH-prop*
have *valid-G-E-IH*: *wf-graph* ⦇*nodes* = *V*, *edges* = *E-IH*⦈ **by** *simp*

from *all-security-requirements-fulfilled-mono*[*OF* ‹*valid-reqs M*› - *valid-G-E-IH*] *IH E-IH-prop*
have *mono-rule*: *E′* ⊆ *E-IH* ⟹ *all-security-requirements-fulfilled M* ⦇*nodes* = *V*, *edges* = *E′*⦈ **for** *E′* **by** *simp*

have *all-security-requirements-fulfilled M*
(*delete-edges* (*generate-valid-topology-SOME M* ⦇*nodes* = *V*, *edges* = *E*⦈)
(*SOME F*. *F* ∈ *c-offending-flows m* ⦇*nodes* = *V*, *edges* = *E*⦈))
unfolding *E-IH-prop* **by**(*auto simp add*: *delete-edges-simp2 intro*:*mono-rule*)
} note *goal-fulfilled-M*=*this*

have *no-offending*: *c-sinvar m* ⦇*nodes* = *V*, *edges* = *E*⦈ ⟹ *c-offending-flows m* ⦇*nodes* = *V*, *edges* = *E*⦈ = {}
by (*simp add*: *configured-SecurityInvariant.c-sinvar-valid-imp-no-offending-flows validReq*)

show *all-security-requirements-fulfilled* (*m* # *M*) (*generate-valid-topology-SOME* (*m* # *M*) ⦇*nodes* = *V*, *edges* = *E*⦈)
apply(*simp add*: *all-security-requirements-fulfilled-def*)
apply(*intro conjI impI*)

subgoal using *goal-sinvar-m no-offending* **by** *blast*
subgoal using *IH* **by**(*simp add*: *all-security-requirements-fulfilled-def*; *fail*)
subgoal using *goal-rm-SOME-m not-sinvar-off* **by** *blast*
subgoal using *goal-fulfilled-M not-sinvar-off* **by**(*simp add*: *all-security-requirements-fulfilled-def*)
done
qed

lemma *generate-valid-topology-SOME-def-alt*:
generate-valid-topology-SOME M G = *delete-edges G* ($\bigcup$ *m* ∈ *set M*. *if c-sinvar m G then* {} *else* (*SOME F*. *F* ∈ *c-offending-flows m G*))
proof(*induction M arbitrary*: *G*)
case *Nil*
thus *?case* **by**(*simp add*: *get-offending-flows-def*)
next
case (*Cons m M*)
from *Cons*[*simplified delete-edges-simp2 get-offending-flows-def*]
have *IH* :*edges* (*generate-valid-topology-SOME M G*) =
edges G − ($\bigcup$ *m*∈*set M*. *if c-sinvar m G then* {} *else SOME F*. *F* ∈ *c-offending-flows m G*)
by *simp*
hence ¬ *c-sinvar m G* ⟹
edges (*generate-valid-topology-SOME* (*m* # *M*) *G*) =
(*edges G*) − ($\bigcup$ *m*∈*set* (*m*#*M*). *if c-sinvar m G then* {} *else SOME F*. *F* ∈ *c-offending-flows m G*)
apply(*simp add*: *get-offending-flows-def delete-edges-simp2*)
by *blast*
with *Cons.IH* **show** *?case* **by**(*simp add*: *get-offending-flows-def delete-edges-simp2*)
qed

lemma *generate-valid-topology-SOME-superset*:
⟦ *valid-reqs M*; *wf-graph G* ⟧ ⟹
edges (*generate-valid-topology M G*) ⊆ *edges* (*generate-valid-topology-SOME M G*)
proof −
have *isabelle2016-1-helper*:
x ∈ ($\bigcup$ *m*∈*set M*. *if c-sinvar m G then* {} *else SOME F*. *F* ∈ *c-offending-flows m G*) ⟷
(∃ *m*∈*set M*. ¬ *c-sinvar m G* ∧ (*c-sinvar m G* ∨ *x* ∈ (*SOME F*. *F* ∈ *c-offending-flows m G*)))
for *x* **by** *auto*
have *1*: *m*∈*set M* ⟹ ¬ *c-sinvar m G* ∧ (*c-sinvar m G* ∨ *x* ∈ (*SOME F*. *F* ∈ *c-offending-flows m G*)) ⟹
c-offending-flows m G ≠ {} ⟹
x ∈ $\bigcup$ ($\bigcup$ *m*∈*set M*. *c-offending-flows m G*)
for *x m*
apply(*simp*)
apply(*rule-tac x*=*m* **in** *bexI*)
apply(*simp-all*)
using *some-in-eq* **by** *blast*

show *valid-reqs M* ⟹ *wf-graph G* ⟹ *?thesis*
unfolding *generate-valid-topology-SOME-def-alt generate-valid-topology-def-alt*
apply (*rule delete-edges-edges-mono*)
apply (*auto simp add*: *delete-edges-simp2 get-offending-flows-def valid-reqs-def*)
apply (*metis* (*full-types*) *configured-SecurityInvariant.defined-offending′ some-in-eq*)
done

qed

Notation: *generate-valid-topology-SOME*: non-deterministic choice *generate-valid-topology-some*: executable which selects always the same

fun *generate-valid-topology-some* :: *'v SecurityInvariant-configured list* ⇒ (*'v*×*'v*) *list* ⇒ *'v graph* ⇒ *'v graph* **where**
generate-valid-topology-some [] - *G* = *G* |
generate-valid-topology-some (*m*#*Ms*) *Es G* = (*if c-sinvar m G*
then generate-valid-topology-some Ms Es G
else delete-edges (*generate-valid-topology-some Ms Es G*) (*set* (*minimalize-offending-overapprox* (*c-sinvar m*) *Es* [] *G*))
)

theorem *generate-valid-topology-some-sound*:
⟦ *valid-reqs M*; *wf-graph* (|*nodes* = *V*, *edges* = *E*|); *set Es* = *E*; *distinct Es* ⟧ ⟹
all-security-requirements-fulfilled M (*generate-valid-topology-some M Es* (|*nodes* = *V*, *edges* = *E*|))
proof(*induction M*)
case *Nil*
thus *?case* **by**(*simp add*: *all-security-requirements-fulfilled-def*)
next
case (*Cons m M*)
from *valid-reqs1*[*OF Cons*(*2*)] **have** *validReq*: *configured-SecurityInvariant m* **.**

from *configured-SecurityInvariant.sinvar-valid-remove-minimalize-offending-overapprox*[*OF validReq Cons.prems*(*2*) - *Cons.prems*(*3*) *Cons.prems*(*4*)] **have** *rm-off-valid*:
¬ *c-sinvar m* (|*nodes* = *V*, *edges* = *E*|)⟹
c-sinvar m (|*nodes* = *V*, *edges* = *E* − (*set* (*minimalize-offending-overapprox* (*c-sinvar m*) *Es* [] (|*nodes* = *V*, *edges* = *E*|)))|)
apply(*subst*(*asm*) *minimalize-offending-overapprox-boundnP*[*symmetric*])
by *blast*

have *generate-valid-topology-some-nodes*: *nodes* (*generate-valid-topology-some M Es* (|*nodes* = *V*, *edges* = *E*|)) = *V*
for *M*::*'a SecurityInvariant-configured list* **and** *V E*
proof(*induction M*)
qed(*simp-all add*: *delete-edges-simp2*)

have *generate-valid-topology-some-edges*: *edges* (*generate-valid-topology-some M Es* (|*nodes* = *V*, *edges* = *E*|)) ⊆ *E*
for *M*::*'a SecurityInvariant-configured list* **and** *V E*
proof(*induction M*)
qed(*auto simp add*: *delete-edges-simp2*)

from *configured-SecurityInvariant.mono-sinvar*[*OF validReq Cons.prems*(*2*),
of edges (*generate-valid-topology-some M Es* (|*nodes* = *V*, *edges* = *E*|))]
generate-valid-topology-some-edges
have *c-sinvar m* (|*nodes* = *V*, *edges* = *E*|) ⟹
c-sinvar m (|*nodes* = *V*, *edges* = *edges* (*generate-valid-topology-some M Es* (|*nodes* = *V*, *edges* = *E*|))|)
by *simp*
moreover from *configured-SecurityInvariant.defined-offending'*[*OF validReq Cons.prems*(*2*)] **have** *not-sinvar-off*:
¬ *c-sinvar m* (|*nodes* = *V*, *edges* = *E*|) ⟹ *c-offending-flows m* (|*nodes* = *V*, *edges* = *E*|) ≠ {} **by** *blast*

ultimately have *goal-sinvar-m*:
c-offending-flows m (|*nodes* = *V*, *edges* = *E*|) = {} ⟹
c-sinvar m (*generate-valid-topology-some M Es* (|*nodes* = *V*, *edges* = *E*|))
using *generate-valid-topology-some-nodes*
by (*metis graph.select-convs*(*1*) *graph.select-convs*(*2*) *graph-eq-intro*)

from *valid-reqs2*[*OF Cons*(*2*)] **have** *valid-reqs M* **.**
from *Cons.IH*[*OF* ‹*valid-reqs M*› *Cons*(*3*)] *Cons.prems* **have** *IH*:
all-security-requirements-fulfilled M (*generate-valid-topology-some M Es* (|*nodes* = *V*, *edges* = *E*|)) **by** *simp*

have *wf-graph-generate-valid-topology-some*: *wf-graph G* ⟹ *wf-graph* (*generate-valid-topology-some M Es G*)
for *G*
apply(*cases G*)
apply(*simp add*: *wf-graph-def generate-valid-topology-some-nodes*)
using *generate-valid-topology-some-edges* **by** (*meson dual-order.trans image-mono rev-finite-subset*)

{ assume *notempty*: *c-offending-flows m* (|*nodes* = *V*, *edges* = *E*|) ≠ {}
hence ∃ *hypE*. (*generate-valid-topology-some M Es* (|*nodes* = *V*, *edges* = *E*|)) = (|*nodes* = *V*, *edges* = *hypE*|)
proof(*induction M arbitrary*: *V E*)
qed(*simp-all add*: *delete-edges-simp2 generate-valid-topology-some-nodes*)
from *this* **obtain** *E-IH* **where** *E-IH-prop*:
(*generate-valid-topology-some M Es* (|*nodes* = *V*, *edges* = *E*|)) = (|*nodes* = *V*, *edges* = *E-IH*|) **by** *blast*

from *wf-graph-generate-valid-topology-some*[*OF Cons*(*3*)] *E-IH-prop*
have *valid-G-E-IH*: *wf-graph* (|*nodes* = *V*, *edges* = *E-IH*|) **by** *simp*

from *all-security-requirements-fulfilled-mono*[*OF* ‹*valid-reqs M*› - *valid-G-E-IH*] *IH E-IH-prop*
have *mono-rule*: *E*′ ⊆ *E-IH* ⟹ *all-security-requirements-fulfilled M* (|*nodes* = *V*, *edges* = *E*′|) **for** *E*′ **by** *simp*

have *all-security-requirements-fulfilled M*
(*delete-edges* (*generate-valid-topology-some M Es* (|*nodes* = *V*, *edges* = *E*|))
(*set* (*minimalize-offending-overapprox* (*c-sinvar m*) *Es* [] (|*nodes* = *V*, *edges* = *E*|))))
unfolding *E-IH-prop* **by**(*auto simp add*: *delete-edges-simp2 intro*:*mono-rule*)
} note *goal-fulfilled-M*=*this*

have *no-offending*: *c-sinvar m* (|*nodes* = *V*, *edges* = *E*|) ⟹ *c-offending-flows m* (|*nodes* = *V*, *edges* = *E*|) = {}
by (*simp add*: *configured-SecurityInvariant.c-sinvar-valid-imp-no-offending-flows validReq*)

show *all-security-requirements-fulfilled* (*m* # *M*) (*generate-valid-topology-some* (*m* # *M*) *Es* (|*nodes* = *V*, *edges* = *E*|))
apply(*simp add*: *all-security-requirements-fulfilled-def*)

apply(*intro conjI impI*)
subgoal using *goal-sinvar-m no-offending* **by** *blast*
subgoal using *IH* **by**(*simp add*: *all-security-requirements-fulfilled-def*; *fail*)
subgoal using *rm-off-valid* **by** (*metis* (*no-types*, *lifting*) *Cons.prems*(*2*) *Diff-mono configured-SecurityInvariant.mono-sinvar delete-edges-simp2 generate-valid-topology-some-edges generate-valid-topology-some-nodes order-refl validReq wf-graph-remove-edges*)
subgoal using *goal-fulfilled-M not-sinvar-off* **by**(*simp add*: *all-security-requirements-fulfilled-def*)
done
qed

end
theory *TopoS-Stateful-Policy*
imports *TopoS-Composition-Theory*
begin

8 Stateful Policy

Details described in [1].

Algorithm

term *TopoS-Composition-Theory.generate-valid-topology*

generates a valid high-level topology. Now we discuss how to turn this into a stateful policy.

Example: SensorNode produces data and has no security level. SensorSink has high security level SensorNode -> SensorSink, but not the other way round. Implementation: UDP in one direction

Alice is in internal protected subnet. Google can not arbitrarily access Alice. Alice sends requests to google. It is desirable that Alice gets the response back Implementation: TCP and stateful packet filter that allows, once Alice establishes a connection, to get a response back via this connection.

Result: IFS violations undesirable. ACS violations may be okay under certain conditions.

term *all-security-requirements-fulfilled*

$G = (V, E_{fix}, E_{state})$

record *′v stateful-policy* =
hosts :: *′v set* — nodes, vertices
flows-fix :: (*′v* × *′v*) *set* — edges in high-level policy
flows-state :: (*′v* × *′v*) *set* — edges that can have stateful flows, i.e. backflows

All the possible ways packets can travel in a *′v stateful-policy*. They can either choose the fixed links; Or use a stateful link, i.e. establish state. Once state is established, packets can flow back via the established link.

definition *all-flows* :: *′v stateful-policy* ⇒ (*′v* × *′v*) *set* **where**
all-flows 𝒯 ≡ *flows-fix* 𝒯 ∪ *flows-state* 𝒯 ∪ *backflows* (*flows-state* 𝒯)

definition *stateful-policy-to-network-graph* :: *′v stateful-policy* ⇒ *′v graph* **where**
stateful-policy-to-network-graph 𝒯 = (| *nodes* = *hosts* 𝒯, *edges* = *all-flows* 𝒯 |)

'v *stateful-policy* syntactically well-formed

```
locale wf-stateful-policy =
  fixes 𝒯 :: 'v stateful-policy
  assumes E-wf: fst ‘ (flows-fix 𝒯) ⊆ (hosts 𝒯)
                snd ‘ (flows-fix 𝒯) ⊆ (hosts 𝒯)
  and E-state-fix: flows-state 𝒯 ⊆ flows-fix 𝒯
  and finite-Hosts: finite (hosts 𝒯)
begin

  lemma E-wfD: assumes (v,v') ∈ flows-fix 𝒯
    shows v ∈ hosts 𝒯 v' ∈ hosts 𝒯
    apply −
     apply (rule subsetD[OF E-wf(1)])
     using assms apply force
    apply (rule subsetD[OF E-wf(2)])
    using assms apply force
    done

  lemma E-state-valid: fst ‘ (flows-state 𝒯) ⊆ (hosts 𝒯)
                       snd ‘ (flows-state 𝒯) ⊆ (hosts 𝒯)
   apply −
   using E-wf(1) E-state-fix apply(blast)
   using E-wf(2) E-state-fix apply(blast)
   done

  lemma E-state-validD: assumes (v,v') ∈ flows-state 𝒯
    shows v ∈ hosts 𝒯 v' ∈ hosts 𝒯
    apply −
     apply (rule subsetD[OF E-state-valid(1)])
     using assms apply force
    apply (rule subsetD[OF E-state-valid(2)])
    using assms apply force
    done

  lemma finite-fix: finite (flows-fix 𝒯)
  proof −
    from finite-subset[OF E-wf(1) finite-Hosts] have 1: finite (fst ‘ flows-fix 𝒯) .
    from finite-subset[OF E-wf(2) finite-Hosts] have 2: finite (snd ‘ flows-fix 𝒯) .
    have s: flows-fix 𝒯 ⊆ (fst ‘ flows-fix 𝒯 × snd ‘ flows-fix 𝒯) by force
    from finite-cartesian-product[OF 1 2] have finite (fst ‘ flows-fix 𝒯 × snd ‘ flows-fix 𝒯) .
    from finite-subset[OF s this] show ?thesis .
  qed

  lemma finite-state: finite (flows-state 𝒯)
    using finite-subset[OF E-state-fix finite-fix] by assumption


  lemma finite-backflows-state: finite (backflows (flows-state 𝒯))
    using [[simproc add: finite-Collect]] by(simp add: backflows-def finite-state)

  lemma E-state-backflows-wf: fst ‘ backflows (flows-state 𝒯) ⊆ (hosts 𝒯)
                              snd ‘ backflows (flows-state 𝒯) ⊆ (hosts 𝒯)
    by(auto simp add: backflows-def E-state-valid E-state-validD)
```

end

Minimizing stateful flows such that only newly added backflows remain

definition *filternew-flows-state* :: *'v stateful-policy* ⇒ (*'v* × *'v*) *set* **where**
filternew-flows-state $\mathcal{T}$ ≡ {(*s*, *r*) ∈ *flows-state* $\mathcal{T}$. (*r*, *s*) ∉ *flows-fix* $\mathcal{T}$}

lemma *filternew-subseteq-flows-state*: *filternew-flows-state* $\mathcal{T}$ ⊆ *flows-state* $\mathcal{T}$
by(*auto simp add*: *filternew-flows-state-def*)

— alternative definitions, all are equal
lemma *filternew-flows-state-alt*: *filternew-flows-state* $\mathcal{T}$ = *flows-state* $\mathcal{T}$ − (*backflows* (*flows-fix* $\mathcal{T}$))
apply(*simp add*: *backflows-def filternew-flows-state-def*)
apply(*rule*)
apply *blast*+
done
lemma *filternew-flows-state-alt2*: *filternew-flows-state* $\mathcal{T}$ = {*e* ∈ *flows-state* $\mathcal{T}$. *e* ∉ *backflows* (*flows-fix* $\mathcal{T}$)}
apply(*simp add*: *backflows-def filternew-flows-state-def*)
apply(*rule*)
apply *blast*+
done
lemma *backflows-filternew-flows-state*: *backflows* (*filternew-flows-state* $\mathcal{T}$) = (*backflows* (*flows-state* $\mathcal{T}$)) − (*flows-fix* $\mathcal{T}$)
by(*simp add*: *filternew-flows-state-alt backflows-minus-backflows*)

lemma *stateful-policy-to-network-graph-filternew*: ⟦ *wf-stateful-policy* $\mathcal{T}$ ⟧ ⟹
stateful-policy-to-network-graph $\mathcal{T}$ =
stateful-policy-to-network-graph (|*hosts* = *hosts* $\mathcal{T}$, *flows-fix* = *flows-fix* $\mathcal{T}$, *flows-state* = *filternew-flows-state* $\mathcal{T}$ |)
apply(*drule wf-stateful-policy.E-state-fix*)
apply(*simp add*: *stateful-policy-to-network-graph-def all-flows-def*)
apply(*rule Set.equalityI*)
apply(*simp add*: *filternew-flows-state-def backflows-def*)
apply(*rule*, *blast*)+
apply(*simp add*: *filternew-flows-state-def backflows-def*)
apply *fastforce*
done

lemma *backflows-filternew-disjunct-flows-fix*:
∀ *b* ∈ (*backflows* (*filternew-flows-state* $\mathcal{T}$)). *b* ∉ *flows-fix* $\mathcal{T}$
by(*simp add*: *filternew-flows-state-def backflows-def*)

Given a high-level policy, we can construct a pretty large syntactically valid low level policy. However, the stateful policy will almost certainly violate security requirements!

lemma *wf-graph G* ⟹ *wf-stateful-policy* (| *hosts* = *nodes G*, *flows-fix* = *nodes G* × *nodes G*, *flows-state* = *nodes G* × *nodes G* |)
by(*simp add*: *wf-stateful-policy-def wf-graph-def*)

wf-stateful-policy implies *wf-graph*

lemma *wf-stateful-policy-is-wf-graph*: *wf-stateful-policy* $\mathcal{T}$ ⟹ *wf-graph* (|*nodes* = *hosts* $\mathcal{T}$, *edges* = *all-flows* $\mathcal{T}$|)
apply(*frule wf-stateful-policy.E-state-backflows-wf*)
apply(*frule wf-stateful-policy.E-state-backflows-wf*(*2*))

apply(*frule wf-stateful-policy.E-state-valid*)
apply(*frule wf-stateful-policy.E-state-valid*(*2*))
apply(*frule wf-stateful-policy.E-wf*)
apply(*frule wf-stateful-policy.E-wf*(*2*))
apply(*simp add*: *all-flows-def wf-graph-def wf-stateful-policy-def wf-stateful-policy.finite-fix wf-stateful-policy.finite-state wf-stateful-policy.finite-backflows-state*)
apply(*rule conjI*)
apply (*metis image-Un sup.bounded-iff*)+
done

lemma ($\forall F \in$ *get-offending-flows* (*get-ACS M*) (*stateful-policy-to-network-graph* $\mathcal{T}$). $F \subseteq$ *backflows* (*filternew-flows-state* $\mathcal{T}$)) $\longleftrightarrow$
$\bigcup$(*get-offending-flows* (*get-ACS M*) (*stateful-policy-to-network-graph* $\mathcal{T}$)) $\subseteq$ (*backflows* (*flows-state* $\mathcal{T}$)) $-$ (*flows-fix* $\mathcal{T}$)
by(*simp add*: *filternew-flows-state-alt backflows-minus-backflows*, *blast*)

When is a stateful policy $\mathcal{T}$ compliant with a high-level policy G and the security requirements M?

locale *stateful-policy-compliance* =
fixes $\mathcal{T}$ :: (*'v*::*vertex*) *stateful-policy*
fixes G :: *'v graph*
fixes M :: (*'v*) *SecurityInvariant-configured list*
assumes
— the graph must be syntactically valid
wfG: *wf-graph G*
and
— security requirements must be valid
validReqs: *valid-reqs M*
and
— the high-level policy must be valid
high-level-policy-valid: *all-security-requirements-fulfilled M G*
and
— the stateful policy must be syntactically valid
stateful-policy-wf:
wf-stateful-policy $\mathcal{T}$
and
— the stateful policy must talk about the same nodes as the high-level policy
hosts-nodes:
hosts $\mathcal{T}$ = *nodes G*
and
— only flows that are allowed in the high-level policy are allowed in the stateful policy
flows-edges:
flows-fix $\mathcal{T} \subseteq$ *edges G*
and
— the low level policy must comply with the high-level policy
— all information flow strategy requirements must be fulfilled, i.e. no leaks!
compliant-stateful-IFS:
all-security-requirements-fulfilled (*get-IFS M*) (*stateful-policy-to-network-graph* $\mathcal{T}$)
and
— No Access Control side effects must occur
compliant-stateful-ACS:
$\forall F \in$ *get-offending-flows* (*get-ACS M*) (*stateful-policy-to-network-graph* $\mathcal{T}$). $F \subseteq$ *backflows*

(*filternew-flows-state* $\mathcal{T}$)

begin

lemma *compliant-stateful-ACS-no-side-effects-filternew-helper*:
∀ E ⊆ *backflows* (*filternew-flows-state* $\mathcal{T}$). ∀ F ∈ *get-offending-flows* (*get-ACS M*) (| *nodes* = *hosts* $\mathcal{T}$, *edges* = *flows-fix* $\mathcal{T}$ ∪ E |). F ⊆ E

proof(*rule*, *rule*)

fix *E*

assume *a1*: E ⊆ *backflows* (*filternew-flows-state* $\mathcal{T}$)

from *validReqs* **have** *valid-ReqsACS*: *valid-reqs* (*get-ACS M*) **by**(*simp add*: *get-ACS-def valid-reqs-def*)

from *compliant-stateful-ACS stateful-policy-to-network-graph-filternew*[*OF stateful-policy-wf*] **have** *compliant-stateful-ACS-only-state-violations-filternew*:
∀ F ∈ *get-offending-flows* (*get-ACS M*) (*stateful-policy-to-network-graph* (|*hosts* = *hosts* $\mathcal{T}$, *flows-fix* = *flows-fix* $\mathcal{T}$, *flows-state* = *filternew-flows-state* $\mathcal{T}$ |)). F ⊆ *backflows* (*filternew-flows-state* $\mathcal{T}$) **by** *simp*

from *wf-stateful-policy-is-wf-graph*[*OF stateful-policy-wf*] **have** *wfGfilternew*:
wf-graph (|*nodes* = *hosts* $\mathcal{T}$, *edges* = *flows-fix* $\mathcal{T}$ ∪ *filternew-flows-state* $\mathcal{T}$ ∪ *backflows* (*filternew-flows-state* $\mathcal{T}$)|)

apply(*simp add*: *all-flows-def filternew-flows-state-alt backflows-minus-backflows*)

by(*auto simp add*: *wf-graph-def*)

from *wf-stateful-policy.E-state-fix*[*OF stateful-policy-wf*] *filternew-subseteq-flows-state* **have** *flows-fix-un-filternew-sim*
flows-fix $\mathcal{T}$ ∪ *filternew-flows-state* $\mathcal{T}$ = *flows-fix* $\mathcal{T}$ **by** *blast*

from *compliant-stateful-ACS-only-state-violations-filternew* **have**
⋀m. m ∈ *set* (*get-ACS M*) ⟹
⋃(*c-offending-flows* m (|*nodes* = *hosts* $\mathcal{T}$, *edges* = *flows-fix* $\mathcal{T}$ ∪ *filternew-flows-state* $\mathcal{T}$ ∪ *backflows* (*filternew-flows-state* $\mathcal{T}$)|)) ⊆ *backflows* (*filternew-flows-state* $\mathcal{T}$)

by(*simp add*: *stateful-policy-to-network-graph-def all-flows-def get-offending-flows-def*, *blast*)

— idea: use ∀ F∈*get-offending-flows* (*get-ACS M*) (*stateful-policy-to-network-graph* $\mathcal{T}$). F ⊆ *backflows* (*filternew-flows-state* $\mathcal{T}$) with the ⟦*configured-SecurityInvariant ?m*; *wf-graph* (|*nodes* = *?V*, *edges* = *?E*|); ⋃ (*c-offending-flows ?m* (|*nodes* = *?V*, *edges* = *?E*|)) ⊆ *?X*⟧ ⟹ ⋃ (*c-offending-flows ?m* (|*nodes* = *?V*, *edges* = *?E* − *?E'*|)) ⊆ *?X* − *?E'* lemma and substract *backflows* (*filternew-flows-state* $\mathcal{T}$) − E, on the right hand side E remains, as Graph's edges *flows-fix* $\mathcal{T}$ ∪ E remains

from *configured-SecurityInvariant.Un-set-offending-flows-bound-minus-subseteq*[**where** X=*backflows* (*filternew-flows-state* $\mathcal{T}$), *OF* - *wfGfilternew this*]
‹*valid-reqs* (*get-ACS M*)›

have
⋀ m E. m ∈ *set* (*get-ACS M*) ⟹
∀ F∈*c-offending-flows* m (|*nodes* = *hosts* $\mathcal{T}$, *edges* = *flows-fix* $\mathcal{T}$ ∪ *filternew-flows-state* $\mathcal{T}$ ∪ *backflows* (*filternew-flows-state* $\mathcal{T}$) − E|). F ⊆ *backflows* (*filternew-flows-state* $\mathcal{T}$) − E

by(*auto simp add*: *all-flows-def valid-reqs-def*)

from *this flows-fix-un-filternew-simp* **have** *rule*:
⋀ m E. m ∈ *set* (*get-ACS M*) ⟹
∀ F∈*c-offending-flows* m (|*nodes* = *hosts* $\mathcal{T}$, *edges* = *flows-fix* $\mathcal{T}$ ∪ *backflows* (*filternew-flows-state* $\mathcal{T}$) − E|). F ⊆ *backflows* (*filternew-flows-state* $\mathcal{T}$) − E

by *simp*

from *backflows-finite rev-finite-subset*[*OF wf-stateful-policy.finite-state*[*OF stateful-policy-wf*] *filternew-subseteq-flows-state*] **have**
finite (*backflows* (*filternew-flows-state* $\mathcal{T}$)) **by** *blast*

from *a1 this* **have** *finite E* **by** (*metis rev-finite-subset*)

from *a1* **obtain** E' **where** E'*-prop1*: *backflows* (*filternew-flows-state* $\mathcal{T}$) $-$ $E' = E$ **and** E'*-prop2*: E' = *backflows* (*filternew-flows-state* $\mathcal{T}$) $-$ E **by** *blast*

from E'*-prop2* ‹*finite* (*backflows* (*filternew-flows-state* $\mathcal{T}$))› ‹*finite E*› **have** *finite* E' **by** *blast*

from *Set.double-diff*[**where** *B*=*backflows* (*filternew-flows-state* $\mathcal{T}$) **and** *C*=*backflows* (*filternew-flows-state* $\mathcal{T}$) **and** *A*=*E*, *OF a1*, *simplified*] **have** *Ebackflowssimp*:
backflows (*filternew-flows-state* $\mathcal{T}$) $-$ (*backflows* (*filternew-flows-state* $\mathcal{T}$) $-$ E) = E **.**

have *flows-fix* $\mathcal{T}$ $\cup$ *backflows* (*filternew-flows-state* $\mathcal{T}$) $-$ (*backflows* (*filternew-flows-state* $\mathcal{T}$) $-$ E) =
(*flows-fix* $\mathcal{T}$ $-$ (*backflows* (*filternew-flows-state* $\mathcal{T}$))) $\cup$ E
apply(*simp add*: *Set.Un-Diff*)
apply(*simp add*: *Ebackflowssimp*)
by *blast*
also have (*flows-fix* $\mathcal{T}$ $-$ (*backflows* (*filternew-flows-state* $\mathcal{T}$))) $\cup$ E = *flows-fix* $\mathcal{T}$ $\cup$ E **using** *backflows-filternew-disjunct-flows-fix* **by** *blast*
finally have *flows-E-simp*: *flows-fix* $\mathcal{T}$ $\cup$ *backflows* (*filternew-flows-state* $\mathcal{T}$) $-$ (*backflows* (*filternew-flows-state* $\mathcal{T}$) $-$ E) = *flows-fix* $\mathcal{T}$ $\cup$ E **.**

from *rule*[*simplified* E'*-prop1* E'*-prop2*] **have**
$\bigwedge$*m*. *m* $\in$ *set* (*get-ACS M*) $\Longrightarrow$
$\forall$ *F*$\in$*c-offending-flows m* (|*nodes* = *hosts* $\mathcal{T}$, *edges* = *flows-fix* $\mathcal{T}$ $\cup$ *backflows* (*filternew-flows-state* $\mathcal{T}$) $-$ (*backflows* (*filternew-flows-state* $\mathcal{T}$) $-$ E)|).
$F \subseteq$ *backflows* (*filternew-flows-state* $\mathcal{T}$) $-$ (*backflows* (*filternew-flows-state* $\mathcal{T}$) $-$ E)
by(*simp*)
from *this Ebackflowssimp flows-E-simp* **have**
$\bigwedge$*m*. *m* $\in$ *set* (*get-ACS M*) $\Longrightarrow$
$\forall$ *F*$\in$*c-offending-flows m* (|*nodes* = *hosts* $\mathcal{T}$, *edges* = *flows-fix* $\mathcal{T}$ $\cup$ E|). $F \subseteq E$
by *simp*
thus $\forall$ *F*$\in$*get-offending-flows* (*get-ACS M*) (|*nodes* = *hosts* $\mathcal{T}$, *edges* = *flows-fix* $\mathcal{T}$ $\cup$ E|). $F \subseteq E$
by(*simp add*: *get-offending-flows-def*)
qed

theorem *compliant-stateful-ACS-no-side-effects*:
$\forall$ $E \subseteq$ *backflows* (*flows-state* $\mathcal{T}$). $\forall$ $F \in$ *get-offending-flows*(*get-ACS M*) (| *nodes* = *hosts* $\mathcal{T}$, *edges* = *flows-fix* $\mathcal{T}$ $\cup$ E |). $F \subseteq E$
proof –
from *compliant-stateful-ACS stateful-policy-to-network-graph-filternew*[*OF stateful-policy-wf*] **have** *a1*:
$\forall$ $F \in$ *get-offending-flows* (*get-ACS M*) (*stateful-policy-to-network-graph* (|*hosts* = *hosts* $\mathcal{T}$, *flows-fix* = *flows-fix* $\mathcal{T}$, *flows-state* = *filternew-flows-state* $\mathcal{T}$ |)). $F \subseteq$ *backflows* (*filternew-flows-state* $\mathcal{T}$) **by** *simp*

have *backflows-split*: *backflows* (*filternew-flows-state* $\mathcal{T}$) $\cup$ (*backflows* (*flows-state* $\mathcal{T}$) $-$ *backflows* (*filternew-flows-state* $\mathcal{T}$)) = *backflows* (*flows-state* $\mathcal{T}$)
by (*metis Diff-subset Un-Diff-cancel Un-absorb1 backflows-minus-backflows filternew-flows-state-alt*)

have
$\forall$ *E*$\subseteq$*backflows* (*filternew-flows-state* $\mathcal{T}$) $\cup$ (*backflows* (*flows-state* $\mathcal{T}$) $-$ *backflows* (*filternew-flows-state* $\mathcal{T}$)).
$\forall$ *F*$\in$*get-offending-flows* (*get-ACS M*) (|*nodes* = *hosts* $\mathcal{T}$, *edges* = *flows-fix* $\mathcal{T}$ $\cup$ E|). $F \subseteq E$
proof(*rule allI*, *rule impI*)

fix *E*
assume *h1*: *E* ⊆ *backflows* (*filternew-flows-state* 𝒯) ∪ (*backflows* (*flows-state* 𝒯) − *backflows* (*filternew-flows-state* 𝒯))

have ∃ *E1 E2*. *E1* ⊆ *backflows* (*filternew-flows-state* 𝒯) ∧ *E2* ⊆ (*backflows* (*flows-state* 𝒯) − *backflows* (*filternew-flows-state* 𝒯)) ∧ *E1* ∪ *E2* = *E* ∧ *E1* ∩ *E2* = {}
apply(*rule-tac x*={*e* ∈ *E*. *e* ∈ *backflows* (*filternew-flows-state* 𝒯)} **in** *exI*)
apply(*rule-tac x*={*e* ∈ *E*. *e* ∈(*backflows* (*flows-state* 𝒯) − *backflows* (*filternew-flows-state* 𝒯))} **in** *exI*)
apply(*simp*)
apply(*rule*)
apply *blast*
apply(*rule*)
apply *blast*
apply(*rule*)
using *h1* **apply** *blast*
using *backflows-filternew-disjunct-flows-fix* **by** *blast*

from *this* **obtain** *E1 E2* **where** *E1-prop*: *E1* ⊆ *backflows* (*filternew-flows-state* 𝒯) **and** *E2-prop*: *E2* ⊆ (*backflows* (*flows-state* 𝒯) − *backflows* (*filternew-flows-state* 𝒯)) **and** *E* = *E1* ∪ *E2* **and** *E1* ∩ *E2* = {} **by** *blast*

— the stateful flows are ⊆ fix flows. If substracting the new stateful flows, onyly the existing fix flows remain
from *E2-prop filternew-flows-state-alt* **have** *E2* ⊆ *flows-fix* 𝒯 **by** (*metis* (*opaque-lifting*, *no-types*) *Diff-subset-conv Un-Diff-cancel2 backflows-minus-backflows inf-sup-ord*(*3*) *order.trans*)
— hence, E2 disappears
from *Set.Un-absorb1*[*OF this*] **have** *E2-absorb*: *flows-fix* 𝒯 ∪ *E2* = *flows-fix* 𝒯 **by** *blast*

from ‹*E* = *E1* ∪ *E2*› **have** *E2E1eq*: *E2* ∪ *E1* = *E* **by** *blast*

from ‹*E* = *E1* ∪ *E2*› ‹*E1* ∩ *E2* = {}› **have** *E1* ⊆ *E* **by** *simp*

from *compliant-stateful-ACS-no-side-effects-filternew-helper E1-prop* **have** ∀ *F*∈*get-offending-flows* (*get-ACS M*) (|*nodes* = *hosts* 𝒯, *edges* = *flows-fix* 𝒯 ∪ *E1* |). *F* ⊆ *E1* **by** *simp*
hence ∀ *F*∈*get-offending-flows* (*get-ACS M*) (|*nodes* = *hosts* 𝒯, *edges* = *flows-fix* 𝒯 ∪ *E2* ∪ *E1* |). *F* ⊆ *E1* **using** *E2-absorb*[*symmetric*] **by** *simp*
hence ∀ *F*∈*get-offending-flows* (*get-ACS M*) (|*nodes* = *hosts* 𝒯, *edges* = *flows-fix* 𝒯 ∪ *E* |). *F* ⊆ *E1* **using** *E2E1eq* **by** (*metis Un-assoc*)

from *this* ‹*E1* ⊆ *E*› **show** ∀ *F*∈*get-offending-flows* (*get-ACS M*) (|*nodes* = *hosts* 𝒯, *edges* = *flows-fix* 𝒯 ∪ *E*|). *F* ⊆ *E* **by** *blast*
qed

from *this backflows-split* **show** *?thesis* **by** *presburger*
qed

corollary *compliant-stateful-ACS-no-side-effects'*: ∀ *E* ⊆ *backflows* (*flows-state* 𝒯). ∀ *F* ∈ *get-offending-flows*(*get-ACS M*) (| *nodes* = *hosts* 𝒯, *edges* = *flows-fix* 𝒯 ∪ *flows-state* 𝒯 ∪ *E* |). *F* ⊆ *E*
using *compliant-stateful-ACS-no-side-effects wf-stateful-policy.E-state-fix*[*OF stateful-policy-wf*] **by** (*metis Un-absorb2*)

The high level graph generated from the low level policy is a valid graph

lemma *valid-stateful-policy*: *wf-graph* (|*nodes* = *hosts* $\mathcal{T}$, *edges* = *all-flows* $\mathcal{T}$|)
by(*rule wf-stateful-policy-is-wf-graph*,*fact stateful-policy-wf*)

The security requirements are definitely fulfilled if we consider only the fixed flows and the normal direction of the stateful flows (i.e. no backflows). I.e. considering no states, everything must be fulfilled

lemma *compliant-stateful-ACS-static-valid*: *all-security-requirements-fulfilled* (*get-ACS M*) (| *nodes* = *hosts* $\mathcal{T}$, *edges* = *flows-fix* $\mathcal{T}$ |)
proof −
from *validReqs* **have** *valid-ReqsACS*: *valid-reqs* (*get-ACS M*) **by**(*simp add*: *get-ACS-def valid-reqs-def*)
from *wfG hosts-nodes*[*symmetric*] **have** *wfG′*: *wf-graph* (| *nodes* = *hosts* $\mathcal{T}$, *edges* = *edges G* |) **by**(*case-tac G*, *simp*)
from *high-level-policy-valid* **have** *all-security-requirements-fulfilled* (*get-ACS M*) *G*
by(*simp add*: *get-ACS-def all-security-requirements-fulfilled-def*)
from *this hosts-nodes*[*symmetric*] **have** *all-security-requirements-fulfilled* (*get-ACS M*) (| *nodes* = *hosts* $\mathcal{T}$, *edges* = *edges G* |)
by(*case-tac G*, *simp*)
from *all-security-requirements-fulfilled-mono*[*OF valid-ReqsACS flows-edges wfG′ this*] **show** *?thesis* **.**
qed
theorem *compliant-stateful-ACS-static-valid′*:
all-security-requirements-fulfilled M (| *nodes* = *hosts* $\mathcal{T}$, *edges* = *flows-fix* $\mathcal{T}$ ∪ *flows-state* $\mathcal{T}$ |)
proof −
from *validReqs* **have** *valid-ReqsIFS*: *valid-reqs* (*get-IFS M*) **by**(*simp add*: *get-IFS-def valid-reqs-def*)

— show that it holds for IFS, by monotonicity as it holds for more in IFS
from *all-security-requirements-fulfilled-mono*[*OF valid-ReqsIFS - valid-stateful-policy compliant-stateful-IFS*[*unfolded stateful-policy-to-network-graph-def*]] **have**
goalIFS: *all-security-requirements-fulfilled* (*get-IFS M*) (| *nodes* = *hosts* $\mathcal{T}$, *edges* = *flows-fix* $\mathcal{T}$ ∪ *flows-state* $\mathcal{T}$ |) **by**(*simp add*: *all-flows-def*)

from *wf-stateful-policy.E-state-fix*[*OF stateful-policy-wf*] **have** *flows-fix* $\mathcal{T}$ ∪ *flows-state* $\mathcal{T}$ = *flows-fix* $\mathcal{T}$ **by** *blast*
from *this compliant-stateful-ACS-static-valid* **have** *goalACS*:
all-security-requirements-fulfilled (*get-ACS M*) (| *nodes* = *hosts* $\mathcal{T}$, *edges* = *flows-fix* $\mathcal{T}$ ∪ *flows-state* $\mathcal{T}$ |) **by** *simp*

— ACS and IFS together form M, we know it holds for ACS
from *goalACS goalIFS* **show** *?thesis*
apply(*simp add*: *all-security-requirements-fulfilled-def get-IFS-def get-ACS-def*)
by *fastforce*
qed

The flows with state are a subset of the flows allowed by the policy

theorem *flows-state-edges*: *flows-state* $\mathcal{T}$ ⊆ *edges G*
using *wf-stateful-policy.E-state-fix*[*OF stateful-policy-wf*] *flows-edges* **by** *simp*

All offending flows are subsets of the reveres stateful flows

lemma *compliant-stateful-ACS-only-state-violations*:
∀ *F* ∈ *get-offending-flows* (*get-ACS M*) (*stateful-policy-to-network-graph* $\mathcal{T}$). *F* ⊆ *backflows* (*flows-state* $\mathcal{T}$)
proof −
have *backflows* (*filternew-flows-state* $\mathcal{T}$) ⊆ *backflows* (*flows-state* $\mathcal{T}$) **by** (*metis Diff-subset backflows-minus-backflows filternew-flows-state-alt*)

from *compliant-stateful-ACS this* **have**
$\forall\ F \in$ *get-offending-flows* (*get-ACS M*) (*stateful-policy-to-network-graph* $\mathcal{T}$). $F \subseteq$ *backflows* (*flows-state* $\mathcal{T}$)
by (*metis subset-trans*)
thus *?thesis* **.**
qed
theorem *compliant-stateful-ACS-only-state-violations'*: $\forall\, F \in$ *get-offending-flows M* (*stateful-policy-to-network-graph* $\mathcal{T}$). $F \subseteq$ *backflows* (*flows-state* $\mathcal{T}$)
proof −
from *validReqs* **have** *valid-ReqsIFS*: *valid-reqs* (*get-IFS M*) **by**(*simp add*: *get-IFS-def valid-reqs-def*)
have *offending-split*: $\bigwedge G$. *get-offending-flows M G* = (*get-offending-flows* (*get-IFS M*) $G \cup$ *get-offending-flows* (*get-ACS M*) G)
apply(*simp add*: *get-offending-flows-def get-IFS-def get-ACS-def*) **by** *blast*
show *?thesis*
apply(*subst offending-split*)
using *compliant-stateful-ACS-only-state-violations*
all-security-requirements-fulfilled-imp-get-offending-empty[*OF valid-ReqsIFS compliant-stateful-IFS*]
by *auto*
qed

All violations are backflows of valid flows

corollary *compliant-stateful-ACS-only-state-violations-union*: $\bigcup$(*get-offending-flows* (*get-ACS M*) (*stateful-policy-to-network-graph* $\mathcal{T}$)) $\subseteq$ *backflows* (*flows-state* $\mathcal{T}$)
using *compliant-stateful-ACS-only-state-violations* **by** *fastforce*

corollary *compliant-stateful-ACS-only-state-violations-union'*: $\bigcup$(*get-offending-flows M* (*stateful-policy-to-network-gr* $\mathcal{T}$)) $\subseteq$ *backflows* (*flows-state* $\mathcal{T}$)
using *compliant-stateful-ACS-only-state-violations'* **by** *fastforce*

All individual flows cause no side effects, i.e. each backflow causes at most itself as violation, no other side-effect violations are induced.

lemma *compliant-stateful-ACS-no-state-singleflow-side-effect*:
$\forall\ (v_1,\ v_2) \in$ *backflows* (*flows-state* $\mathcal{T}$).
$\bigcup$(*get-offending-flows*(*get-ACS M*) (| *nodes* = *hosts* $\mathcal{T}$, *edges* = *flows-fix* $\mathcal{T} \cup$ *flows-state* $\mathcal{T} \cup \{(v_1,\ v_2)\}$ |)) $\subseteq \{(v_1,\ v_2)\}$
using *compliant-stateful-ACS-no-side-effects'* **by** *blast*
end

8.1 Summarizing the important theorems

No information flow security requirements are violated (including all added stateful flows)

thm *stateful-policy-compliance.compliant-stateful-IFS*

There are not access control side effects when allowing stateful backflows. I.e. for all possible subsets of the to-allow backflows, the violations they cause are only these backflows themselves

thm *stateful-policy-compliance.compliant-stateful-ACS-no-side-effects'*

Also, considering all backflows individually, they cause no side effect, i.e. the only violation added is the backflow itself

thm *stateful-policy-compliance.compliant-stateful-ACS-no-state-singleflow-side-effect*

In particular, all introduced offending flows for access control strategies are at most the stateful backflows

thm *stateful-policy-compliance.compliant-stateful-ACS-only-state-violations-union*

Which implies: all introduced offending flows are at most the stateful backflows

thm *stateful-policy-compliance.compliant-stateful-ACS-only-state-violations-union'*

Disregarding the backflows of stateful flows, all security requirements are fulfilled.

thm *stateful-policy-compliance.compliant-stateful-ACS-static-valid'*

end
theory *TopoS-Composition-Theory-impl*
imports *TopoS-Interface-impl TopoS-Composition-Theory*
begin

9 Composition Theory – List Implementation

Several invariants may apply to one policy.

term *X*::('*v*::*vertex*, '*a*) *TopoS-packed*

9.1 Generating instantiated (configured) network security invariants

```
record ('v) SecurityInvariant =
  implc-type :: string
  implc-description :: string
  implc-sinvar ::('v) list-graph ⇒ bool
  implc-offending-flows ::('v) list-graph ⇒ ('v × 'v) list list
  implc-isIFS :: bool
```

Test if this definition is compliant with the formal definition on sets.

```
definition SecurityInvariant-complies-formal-def ::
 ('v) SecurityInvariant ⇒ 'v TopoS-Composition-Theory.SecurityInvariant-configured ⇒ bool where
  SecurityInvariant-complies-formal-def impl spec ≡
   (∀ G. wf-list-graph G ⟶ implc-sinvar impl G = c-sinvar spec (list-graph-to-graph G)) ∧
     (∀ G. wf-list-graph G ⟶ set`set (implc-offending-flows impl G) = c-offending-flows spec
(list-graph-to-graph G)) ∧
   (implc-isIFS impl = c-isIFS spec)
```

```
fun new-configured-list-SecurityInvariant ::
  ('v::vertex, 'a) TopoS-packed ⇒ ('v::vertex, 'a) TopoS-Params ⇒ string ⇒
    ('v SecurityInvariant) where
   new-configured-list-SecurityInvariant m C description =
    (let nP = nm-node-props m C in
     ⦇
       implc-type = nm-name m,
       implc-description = description,
       implc-sinvar = (λG. (nm-sinvar m) G nP),
       implc-offending-flows = (λG. (nm-offending-flows m) G nP),
       implc-isIFS = nm-receiver-violation m
```

|))

the *new-configured-SecurityInvariant* must give a result if we have the SecurityInvariant modelLibrary

lemma *TopoS-modelLibrary-yields-new-configured-SecurityInvariant*:
assumes *NetModelLib*: *TopoS-modelLibrary m sinvar-spec*
and *nPdef*: *nP = nm-node-props m C*
and *formalSpec*: *Spec* = (|
c-sinvar = (λ*G*. *sinvar-spec G nP*),
c-offending-flows = (λ*G*. *SecurityInvariant-withOffendingFlows.set-offending-flows sinvar-spec G nP*),
c-isIFS = *nm-receiver-violation m*
|)
shows *new-configured-SecurityInvariant* (*sinvar-spec*, *nm-default m*, *nm-receiver-violation m*, *nP*) = *Some Spec*
proof –
from *NetModelLib* **have** *NetModel*: *SecurityInvariant sinvar-spec* (*nm-default m*) (*nm-receiver-violation m*)
by(*simp add*: *TopoS-modelLibrary-def TopoS-List-Impl-def*)

have *Spec*: (|*c-sinvar* = λ*G*. *sinvar-spec G nP*,
c-offending-flows = λ*G*. *SecurityInvariant-withOffendingFlows.set-offending-flows sinvar-spec G nP*,
c-isIFS = *nm-receiver-violation m*|) = *Spec*
by(*simp add*: *formalSpec*)
show *?thesis*
unfolding *new-configured-SecurityInvariant.simps*
by(*simp add*: *NetModel Spec*)
qed
thm *TopoS-modelLibrary-yields-new-configured-SecurityInvariant*[*simplified*]

lemma *new-configured-list-SecurityInvariant-complies*:
assumes *NetModelLib*: *TopoS-modelLibrary m sinvar-spec*
and *nPdef*: *nP = nm-node-props m C*
and *formalSpec*: *Spec = new-configured-SecurityInvariant* (*sinvar-spec*, *nm-default m*, *nm-receiver-violation m*, *nP*)
and *implSpec*: *Impl = new-configured-list-SecurityInvariant m C description*
shows *SecurityInvariant-complies-formal-def Impl* (*the Spec*)
proof –
from *TopoS-modelLibrary-yields-new-configured-SecurityInvariant*[*OF NetModelLib nPdef*]
have *SpecUnfolded*: *new-configured-SecurityInvariant* (*sinvar-spec*, *nm-default m*, *nm-receiver-violation m*, *nP*) =
Some (|*c-sinvar* = λ*G*. *sinvar-spec G nP*,
c-offending-flows = λ*G*. *SecurityInvariant-withOffendingFlows.set-offending-flows sinvar-spec G nP*,
c-isIFS = *nm-receiver-violation m*|) **by** *simp*

from *NetModelLib* **show** *?thesis*
apply(*simp add*: *SpecUnfolded formalSpec implSpec Let-def*)
apply(*simp add*: *SecurityInvariant-complies-formal-def-def*)
apply(*simp add*: *TopoS-modelLibrary-def TopoS-List-Impl-def*)
apply(*simp add*: *nPdef*)

done
qed

corollary *new-configured-list-SecurityInvariant-complies'*:
⟦ *TopoS-modelLibrary m sinvar-spec* ⟧ ⟹
SecurityInvariant-complies-formal-def (*new-configured-list-SecurityInvariant m C description*)
(*the* (*new-configured-SecurityInvariant* (*sinvar-spec*, *nm-default m*, *nm-receiver-violation m*, *nm-node-props m C*)))
by(*blast dest*: *new-configured-list-SecurityInvariant-complies*)

— From
thm *new-configured-SecurityInvariant-sound*
— we get that *new-configured-list-SecurityInvariant* has all the necessary properties (modulo *SecurityInvariant-complies-formal-def*)

9.2 About security invariants

specification and implementation comply.

type-synonym *'v security-models-spec-impl*=(*'v SecurityInvariant* × *'v TopoS-Composition-Theory.SecurityInvariant-* *list*

definition *get-spec* :: *'v security-models-spec-impl* ⇒ (*'v TopoS-Composition-Theory.SecurityInvariant-configured*) *list* **where**
get-spec M ≡ [*snd m*. *m* ← *M*]
definition *get-impl* :: *'v security-models-spec-impl* ⇒ (*'v SecurityInvariant*) *list* **where**
get-impl M ≡ [*fst m*. *m* ← *M*]

9.3 Calculating offending flows

fun *implc-get-offending-flows* :: (*'v*) *SecurityInvariant list* ⇒ *'v list-graph* ⇒ ((*'v* × *'v*) *list list*) **where**
implc-get-offending-flows [] *G* = [] |
implc-get-offending-flows (*m#Ms*) *G* = (*implc-offending-flows m G*)@(*implc-get-offending-flows Ms G*)

lemma *implc-get-offending-flows-fold*:
implc-get-offending-flows M G = *fold* (*λm accu*. *accu*@(*implc-offending-flows m G*)) *M* []
proof−
{ fix *accu*
have *accu*@(*implc-get-offending-flows M G*) = *fold* (*λm accu*. *accu*@(*implc-offending-flows m G*)) *M accu*
apply(*induction M arbitrary*: *accu*)
apply(*simp-all*)
by(*metis append-eq-appendI*) **}**
from *this*[**where** *accu2*=[]] **show** *?thesis* **by** *simp*
qed

lemma *implc-get-offending-flows-Un*: *set`set* (*implc-get-offending-flows M G*) = (⋃*m*∈*set M*. *set`set* (*implc-offending-flows m G*))
apply(*induction M*)
apply(*simp-all*)
by (*metis image-Un*)

lemma *implc-get-offending-flows-map-concat*: (*implc-get-offending-flows M G*) = *concat* [*implc-offending-flows m G*. *m* ← *M*]
apply(*induction M*)
by(*simp-all*)

theorem *implc-get-offending-flows-complies*:
assumes *a1*: ∀ (*m-impl*, *m-spec*) ∈ *set M*. *SecurityInvariant-complies-formal-def m-impl m-spec*
and *a2*: *wf-list-graph G*
shows *set'set* (*implc-get-offending-flows* (*get-impl M*) *G*) = (*get-offending-flows* (*get-spec M*) (*list-graph-to-graph G*))
proof –
from *a1* **have** ∀ (*m-impl*, *m-spec*) ∈ *set M*. *set* ' *set* (*implc-offending-flows m-impl G*) = *c-offending-flows m-spec* (*list-graph-to-graph G*)
apply(*simp add*: *SecurityInvariant-complies-formal-def-def*)
using *a2* **by** *blast*
hence ∀ *m* ∈ *set M*. *set* ' *set* (*implc-offending-flows* (*fst m*) *G*) = *c-offending-flows* (*snd m*) (*list-graph-to-graph G*) **by** *fastforce*
thus *?thesis*
by(*simp add*: *get-impl-def get-spec-def implc-get-offending-flows-Un get-offending-flows-def*)
qed

9.4 Accessors

definition *get-IFS* :: *'v SecurityInvariant list* ⇒ *'v SecurityInvariant list* **where**
get-IFS M ≡ [*m* ← *M*. *implc-isIFS m*]
definition *get-ACS* :: *'v SecurityInvariant list* ⇒ *'v SecurityInvariant list* **where**
get-ACS M ≡ [*m* ← *M*. ¬ *implc-isIFS m*]

lemma *get-IFS-get-ACS-complies*:
assumes *a*: ∀ (*m-impl*, *m-spec*) ∈ *set M*. *SecurityInvariant-complies-formal-def m-impl m-spec*
shows ∀ (*m-impl*, *m-spec*) ∈ *set* (*zip* (*get-IFS* (*get-impl M*)) (*TopoS-Composition-Theory.get-IFS* (*get-spec M*))).
SecurityInvariant-complies-formal-def m-impl m-spec
and ∀ (*m-impl*, *m-spec*) ∈ *set* (*zip* (*get-ACS* (*get-impl M*)) (*TopoS-Composition-Theory.get-ACS* (*get-spec M*))).
SecurityInvariant-complies-formal-def m-impl m-spec
proof –
from *a* **have** ∀ (*m-impl*, *m-spec*) ∈ *set M*. *implc-isIFS m-impl* = *c-isIFS m-spec*
apply(*simp add*: *SecurityInvariant-complies-formal-def-def*) **by** *fastforce*
hence *set-zip-IFS*: *set* (*zip* (*filter implc-isIFS* (*get-impl M*)) (*filter c-isIFS* (*get-spec M*))) ⊆ *set M*
apply(*simp add*: *get-impl-def get-spec-def*)
apply(*induction M*)
apply(*simp-all*)
by *force*
from *set-zip-IFS a* **show** ∀ (*m-impl*, *m-spec*) ∈ *set* (*zip* (*get-IFS* (*get-impl M*)) (*TopoS-Composition-Theory.get-IFS* (*get-spec M*))).
SecurityInvariant-complies-formal-def m-impl m-spec
apply(*simp add*: *get-IFS-def get-ACS-def*
TopoS-Composition-Theory.get-IFS-def TopoS-Composition-Theory.get-ACS-def) **by** *blast*
next

from *a* **have** ∀ (*m-impl*, *m-spec*) ∈ *set M*. *implc-isIFS m-impl* = *c-isIFS m-spec*
apply(*simp add*: *SecurityInvariant-complies-formal-def-def*) **by** *fastforce*
hence *set-zip-ACS*: *set* (*zip* [*m*←*get-impl M* . ¬ *implc-isIFS m*] [*m*←*get-spec M* . ¬ *c-isIFS m*]) ⊆ *set M*
apply(*simp add*: *get-impl-def get-spec-def*)
apply(*induction M*)
apply(*simp-all*)
by *force*
from *this a* **show** ∀ (*m-impl*, *m-spec*) ∈ *set* (*zip* (*get-ACS* (*get-impl M*)) (*TopoS-Composition-Theory.get-ACS* (*get-spec M*))).
SecurityInvariant-complies-formal-def m-impl m-spec
apply(*simp add*: *get-IFS-def get-ACS-def*
TopoS-Composition-Theory.get-IFS-def TopoS-Composition-Theory.get-ACS-def) **by** *fast*
qed

lemma *get-IFS-get-ACS-select-simps*:
assumes *a1*: ∀ (*m-impl*, *m-spec*) ∈ *set M*. *SecurityInvariant-complies-formal-def m-impl m-spec*
shows ∀ (*m-impl*, *m-spec*) ∈ *set* (*zip* (*get-IFS* (*get-impl M*)) (*TopoS-Composition-Theory.get-IFS* (*get-spec M*))). *SecurityInvariant-complies-formal-def m-impl m-spec* (**is** ∀ (*m-impl*, *m-spec*) ∈ *set ?zippedIFS*. *SecurityInvariant-complies-formal-def m-impl m-spec*)
and (*get-impl* (*zip* (*TopoS-Composition-Theory-impl.get-IFS* (*get-impl M*)) (*TopoS-Composition-Theory.get-IFS* (*get-spec M*)))) = *TopoS-Composition-Theory-impl.get-IFS* (*get-impl M*)
and (*get-spec* (*zip* (*TopoS-Composition-Theory-impl.get-IFS* (*get-impl M*)) (*TopoS-Composition-Theory.get-IFS* (*get-spec M*)))) = *TopoS-Composition-Theory.get-IFS* (*get-spec M*)
and ∀ (*m-impl*, *m-spec*) ∈ *set* (*zip* (*get-ACS* (*get-impl M*)) (*TopoS-Composition-Theory.get-ACS* (*get-spec M*))). *SecurityInvariant-complies-formal-def m-impl m-spec* (**is** ∀ (*m-impl*, *m-spec*) ∈ *set ?zippedACS*. *SecurityInvariant-complies-formal-def m-impl m-spec*)
and (*get-impl* (*zip* (*TopoS-Composition-Theory-impl.get-ACS* (*get-impl M*)) (*TopoS-Composition-Theory.get-ACS* (*get-spec M*)))) = *TopoS-Composition-Theory-impl.get-ACS* (*get-impl M*)
and (*get-spec* (*zip* (*TopoS-Composition-Theory-impl.get-ACS* (*get-impl M*)) (*TopoS-Composition-Theory.get-ACS* (*get-spec M*)))) = *TopoS-Composition-Theory.get-ACS* (*get-spec M*)
proof −
from *get-IFS-get-ACS-complies*(*1*)[*OF a1*]
show ∀ (*m-impl*, *m-spec*) ∈ *set* (*?zippedIFS*). *SecurityInvariant-complies-formal-def m-impl m-spec* **by** *simp*
next
from *a1* **show** (*get-impl ?zippedIFS*) = *TopoS-Composition-Theory-impl.get-IFS* (*get-impl M*)
apply(*simp add*: *TopoS-Composition-Theory-impl.get-IFS-def get-spec-def get-impl-def TopoS-Composition-Theory
apply(*induction M*)
apply(*simp*)
apply(*simp*)
apply(*rule conjI*)
apply(*clarify*)
using *SecurityInvariant-complies-formal-def-def* **apply** (*auto*)[*1*]
apply(*clarify*)
using *SecurityInvariant-complies-formal-def-def* **apply** (*auto*)[*1*]
done
next
from *a1* **show** (*get-spec ?zippedIFS*) = *TopoS-Composition-Theory.get-IFS* (*get-spec M*)
apply(*simp add*: *TopoS-Composition-Theory-impl.get-IFS-def get-spec-def get-impl-def TopoS-Composition-Theory
apply(*induction M*)
apply(*simp*)

apply(*simp*)
apply(*rule conjI*)
apply(*clarify*)
using *SecurityInvariant-complies-formal-def-def* **apply** (*auto*)[*1*]
apply(*clarify*)
using *SecurityInvariant-complies-formal-def-def* **apply** (*auto*)[*1*]
done
next
from *get-IFS-get-ACS-complies*(*2*)[*OF a1*]
show ∀ (*m-impl*, *m-spec*) ∈ *set* (*?zippedACS*). *SecurityInvariant-complies-formal-def m-impl m-spec* **by** *simp*
next
from *a1* **show** (*get-impl ?zippedACS*) = *TopoS-Composition-Theory-impl.get-ACS* (*get-impl M*)
apply(*simp add*: *TopoS-Composition-Theory-impl.get-ACS-def get-spec-def get-impl-def TopoS-Composition-Theory.get-ACS-def*)
apply(*induction M*)
apply(*simp*)
apply(*simp*)
apply(*rule conjI*)
apply(*clarify*)
using *SecurityInvariant-complies-formal-def-def* **apply** (*auto*)[*1*]
apply(*clarify*)
using *SecurityInvariant-complies-formal-def-def* **apply** (*auto*)[*1*]
done
next
from *a1* **show** (*get-spec ?zippedACS*) = *TopoS-Composition-Theory.get-ACS* (*get-spec M*)
apply(*simp add*: *TopoS-Composition-Theory-impl.get-ACS-def get-spec-def get-impl-def TopoS-Composition-Theory.get-ACS-def*)
apply(*induction M*)
apply(*simp*)
apply(*simp*)
apply(*rule conjI*)
apply(*clarify*)
using *SecurityInvariant-complies-formal-def-def* **apply** (*auto*)[*1*]
apply(*clarify*)
using *SecurityInvariant-complies-formal-def-def* **apply** (*auto*)[*1*]
done
qed

thm *get-IFS-get-ACS-select-simps*

9.5 All security requirements fulfilled

definition *all-security-requirements-fulfilled* :: *'v SecurityInvariant list* ⇒ *'v list-graph* ⇒ *bool* **where**
all-security-requirements-fulfilled M G ≡ ∀ *m* ∈ *set M*. (*implc-sinvar m*) *G*

lemma *all-security-requirements-fulfilled-complies*:
⟦ ∀ (*m-impl*, *m-spec*) ∈ *set M*. *SecurityInvariant-complies-formal-def m-impl m-spec*;
wf-list-graph (*G*::(*'v*::*vertex*) *list-graph*) ⟧ ⟹
all-security-requirements-fulfilled (*get-impl M*) *G* ⟷ *TopoS-Composition-Theory.all-security-requirements-fulfilled* (*get-spec M*) (*list-graph-to-graph G*)
apply(*simp add*: *all-security-requirements-fulfilled-def TopoS-Composition-Theory.all-security-requirements-fulfilled-de*
apply(*simp add*: *get-impl-def get-spec-def*)

using *SecurityInvariant-complies-formal-def-def* **by** *fastforce*

9.6 generate valid topology

value *concat* [[*1*::*int*,*2*,*3*], [*4*,*6*,*5*]]

fun *generate-valid-topology* :: *'v SecurityInvariant list* ⇒ *'v list-graph* ⇒ (*'v list-graph*) **where**
generate-valid-topology M G = *delete-edges G* (*concat* (*implc-get-offending-flows M G*))

lemma *generate-valid-topology-complies*:
⟦ ∀ (*m-impl*, *m-spec*) ∈ *set M*. *SecurityInvariant-complies-formal-def m-impl m-spec*;
wf-list-graph (*G*::(*'v list-graph*)) ⟧ ⟹
list-graph-to-graph (*generate-valid-topology* (*get-impl M*) *G*) =
TopoS-Composition-Theory.generate-valid-topology (*get-spec M*) (*list-graph-to-graph G*)
apply (*subst generate-valid-topology-def-alt*)
apply (*drule*(*1*) *implc-get-offending-flows-complies*)
apply (*simp add*: *delete-edges-correct* [*symmetric*])
done

9.7 generate valid topology

tuned for invariants where we don't want to calculate all offending flows

Theoretic foundations: The algorithm *generate-valid-topology-SOME* picks ONE offending flow non-deterministically. This is sound: ⟦*valid-reqs ?M*; *wf-graph* (|*nodes* = *?V*, *edges* = *?E*|)⟧ ⟹ *TopoS-Composition-Theory.all-security-requirements-fulfilled ?M* (*generate-valid-topology-SOME ?M* (|*nodes* = *?V*, *edges* = *?E*|)). However, this non-deterministic choice is hard to implement. To pick one offending flow deterministically, we have implemented *TopoS-Interface-impl.minimalize-offending-o* It gives back one offending flow: ⟦*SecurityInvariant-preliminaries ?sinvar*; *wf-graph ?G*; *SecurityInvariant-withOffendingFlows.is-offending-flows ?sinvar* (*set ?ff*) *?G ?nP*; *set ?ff* ⊆ *edges ?G*; *distinct ?ff*⟧ ⟹ *set* (*SecurityInvariant-withOffendingFlows.minimalize-offending-overapprox ?sinvar ?ff* [] *?G ?nP*) ∈ *SecurityInvariant-withOffendingFlows.set-offending-flows ?sinvar ?G ?nP* The good thing about this function is, that it does not need to construct the complete *SecurityInvariant-withOffendingFlows.set-offending-flows*. Therefore, it can be used for security invariants which may have an exponential number of offending flows. The corresponding algorithm that uses this function is *generate-valid-topology-some*. It is also sound: ⟦*valid-reqs ?M*; *wf-graph* (|*nodes* = *?V*, *edges* = *?E*|); *set ?Es* = *?E*; *distinct ?Es*⟧ ⟹ *TopoS-Composition-Theory.all-security-requirements-fulfilled ?M* (*generate-valid-topology-some ?M ?Es* (|*nodes* = *?V*, *edges* = *?E*|)).

fun *generate-valid-topology-some* :: *'v SecurityInvariant list* ⇒ *'v list-graph* ⇒ (*'v list-graph*) **where**
generate-valid-topology-some [] *G* = *G* |
generate-valid-topology-some (*m#Ms*) *G* = (*if implc-sinvar m G*
then generate-valid-topology-some Ms G
else delete-edges (*generate-valid-topology-some Ms G*) (*minimalize-offending-overapprox* (*implc-sinvar m*) (*edgesL G*) [] *G*)
)

thm *TopoS-Composition-Theory.generate-valid-topology-some-sound*

lemma *generate-valid-topology-some-complies*:
⟦ ∀ (*m-impl*, *m-spec*) ∈ *set M*. *SecurityInvariant-complies-formal-def m-impl m-spec*;

wf-list-graph (*G*::('*v*::*vertex list-graph*)) ⟧ ⟹
list-graph-to-graph (*generate-valid-topology-some* (*get-impl M*) *G*) =
TopoS-Composition-Theory.generate-valid-topology-some (*get-spec M*) (*edgesL G*) (*list-graph-to-graph G*)
proof(*induction M*)
case *Nil* **thus** *?case* **by**(*simp add*: *get-spec-def get-impl-def*)
next
case (*Cons m M*)
obtain *m-impl m-spec* **where** *m*: *m* = (*m-impl*, *m-spec*) **by**(*cases m*) *blast*
from *m* **have** *m-impl*: *get-impl* ((*m-impl*, *m-spec*) # *M*) = *m-impl* # (*get-impl M*) **by** (*simp add*: *get-impl-def*)
from *m* **have** *m-spec*: *get-spec* ((*m-impl*, *m-spec*) # *M*) = *m-spec* # (*get-spec M*) **by** (*simp add*: *get-spec-def*)

from *Cons.prems*(*1*) *m* **have** *complies-formal-def*: *SecurityInvariant-complies-formal-def m-impl m-spec* **by** *simp*
with *Cons.prems*(*2*) **have** *impl-spec*: *implc-sinvar m-impl G* ⟷ *c-sinvar m-spec* (*list-graph-to-graph G*)
by (*simp add*: *SecurityInvariant-complies-formal-def-def*)

from *complies-formal-def*
have ⋀*G nP*. *wf-list-graph G* ⟹
(λ*G nP*. (*c-sinvar m-spec*) *G*) (*list-graph-to-graph G*) *nP* = (λ*G nP*. (*implc-sinvar m-impl*) *G*) *G nP*
by (*simp add*: *SecurityInvariant-complies-formal-def-def*)

from *minimalize-offending-overapprox-spec-impl*[*OF Cons.prems*(*2*),
of (λ*G nP*. (*c-sinvar m-spec*) *G*) (λ*G nP*. (*implc-sinvar m-impl*) *G*), *OF this*]

have *TopoS-Interface-impl.minimalize-offending-overapprox* (*implc-sinvar m-impl*) *fs keeps G* =
TopoS-withOffendingFlows.minimalize-offending-overapprox (*c-sinvar m-spec*) *fs keeps* (*list-graph-to-graph G*)
for *fs keeps* **by** *simp*
from *this*[*of* (*edgesL G*) []] **have** *minimalize-offending-overapprox-spec*:
TopoS-Interface-impl.minimalize-offending-overapprox (*implc-sinvar m-impl*) (*edgesL G*) [] *G* =
TopoS-withOffendingFlows.minimalize-offending-overapprox (*c-sinvar m-spec*) (*edgesL G*) [] (*list-graph-to-graph G*) **.**

from *Cons* **show** *?case*
apply(*simp*)
apply(*simp add*: *m m-impl m-spec*)
apply(*intro conjI impI*)
apply (*simp add*: *impl-spec*; *fail*)
apply (*simp add*: *impl-spec*; *fail*)
apply(*simp add*: *delete-edges-correct*[*symmetric*])
apply(*simp add*: *list-graph-to-graph-def FiniteGraph.delete-edges-simp2*)
apply(*simp add*: *minimalize-offending-overapprox-spec*)
by (*simp add*: *list-graph-to-graph-def*)
qed

end
theory *TopoS-Stateful-Policy-Algorithm*
imports *TopoS-Stateful-Policy TopoS-Composition-Theory*
begin

10 Stateful Policy – Algorithm

10.1 Some unimportant lemmata

lemma *False-set*: {(*r*, *s*). *False*} = {} **by** *simp*
lemma *valid-reqs-ACS-D*: *valid-reqs M* ⟹ *valid-reqs* (*get-ACS M*)
by(*simp add*: *valid-reqs-def get-ACS-def*)
lemma *valid-reqs-IFS-D*: *valid-reqs M* ⟹ *valid-reqs* (*get-IFS M*)
by(*simp add*: *valid-reqs-def get-IFS-def*)
lemma *all-security-requirements-fulfilled-ACS-D*: *all-security-requirements-fulfilled M G* ⟹
all-security-requirements-fulfilled (*get-ACS M*) *G*
by(*simp add*: *all-security-requirements-fulfilled-def get-ACS-def*)
lemma *all-security-requirements-fulfilled-IFS-D*: *all-security-requirements-fulfilled M G* ⟹
all-security-requirements-fulfilled (*get-IFS M*) *G*
by(*simp add*: *all-security-requirements-fulfilled-def get-IFS-def*)
lemma *all-security-requirements-fulfilled-mono-stateful-policy-to-network-graph*:
⟦ *valid-reqs M*; *E′* ⊆ *E*; *wf-graph* ⦇ *nodes* = *V*, *edges* = *Efix* ∪ *E* ⦈ ⟧ ⟹
all-security-requirements-fulfilled M
(*stateful-policy-to-network-graph* ⦇ *hosts* = *V*, *flows-fix* = *Efix*, *flows-state* = *E* ⦈)⟹
all-security-requirements-fulfilled M
(*stateful-policy-to-network-graph* ⦇ *hosts* = *V*, *flows-fix* = *Efix*, *flows-state* = *E′* ⦈)
apply(*simp add*: *stateful-policy-to-network-graph-def all-flows-def*)
apply(*drule all-security-requirements-fulfilled-mono*[**where** *E*=*Efix* ∪ *E* ∪ *backflows E* **and** *E′*=*Efix* ∪ *E′* ∪ *backflows E′* **and** *V*=*V*])
apply(*thin-tac wf-graph G* **for** *G*)
apply(*thin-tac all-security-requirements-fulfilled M G* **for** *M G*)
apply(*simp add*: *backflows-def*, *blast*)
apply(*thin-tac all-security-requirements-fulfilled M G* **for** *M G*)
apply(*simp add*: *wf-graph-def*)
apply(*simp add*: *backflows-def*)
using [[*simproc add*: *finite-Collect*]] **apply**(*auto*)[*1*]
apply(*simp-all*)
done

10.2 Sketch for generating a stateful policy from a simple directed policy

Having no stateful flows, we trivially get a valid stateful policy.

lemma *trivial-stateful-policy-compliance*:
⟦ *wf-graph* ⦇ *nodes* = *V*, *edges* = *E* ⦈; *valid-reqs M*; *all-security-requirements-fulfilled M* ⦇ *nodes* = *V*, *edges* = *E* ⦈ ⟧ ⟹
stateful-policy-compliance ⦇ *hosts* = *V*, *flows-fix* = *E*, *flows-state* = {} ⦈ ⦇ *nodes* = *V*, *edges* = *E* ⦈ *M*
apply(*unfold-locales*)
apply(*simp-all add*: *wf-graph-def stateful-policy-to-network-graph-def all-flows-def backflows-def False-set*)
apply(*simp add*: *get-IFS-def get-ACS-def all-security-requirements-fulfilled-def*)
apply(*clarify*)
apply(*drule valid-reqs-ACS-D*)

apply(*drule all-security-requirements-fulfilled-ACS-D*)
apply(*drule*(*1*) *all-security-requirements-fulfilled-imp-get-offending-empty*)
by *force*

trying better

First, filtering flows that cause no IFS violations

fun *filter-IFS-no-violations-accu* :: *'v::vertex graph* ⇒ *'v SecurityInvariant-configured list* ⇒ (*'v* × *'v*) *list* ⇒ (*'v* × *'v*) *list* ⇒ (*'v* × *'v*) *list* **where**
filter-IFS-no-violations-accu G M accu [] = *accu* |
filter-IFS-no-violations-accu G M accu (*e#Es*) = (*if*
all-security-requirements-fulfilled (*get-IFS M*) (*stateful-policy-to-network-graph* (| *hosts* = *nodes G*, *flows-fix* = *edges G*, *flows-state* = *set* (*e#accu*) |))
then filter-IFS-no-violations-accu G M (*e#accu*) *Es*
else filter-IFS-no-violations-accu G M accu Es)
definition *filter-IFS-no-violations* :: *'v::vertex graph* ⇒ *'v SecurityInvariant-configured list* ⇒ (*'v* × *'v*) *list* ⇒ (*'v* × *'v*) *list* **where**
filter-IFS-no-violations G M Es = *filter-IFS-no-violations-accu G M* [] *Es*

lemma *filter-IFS-no-violations-subseteq-input*: *set* (*filter-IFS-no-violations G M Es*) ⊆ *set Es*
apply(*subgoal-tac* ∀ *accu*. *set* (*filter-IFS-no-violations-accu G M accu Es*) ⊆ *set Es* ∪ *set accu*)
apply(*erule-tac x*=[] **in** *allE*)
apply(*simp add*: *filter-IFS-no-violations-def*)
unfolding *filter-IFS-no-violations-def*
apply(*induct-tac Es*)
apply(*simp-all*)
apply *force*
done
lemma *filter-IFS-no-violations-accu-correct-induction*: *valid-reqs* (*get-IFS M*) ⟹ *wf-graph* (| *nodes* = *V*, *edges* = *E* |) ⟹
all-security-requirements-fulfilled (*get-IFS M*) (*stateful-policy-to-network-graph* (| *hosts* = *V*, *flows-fix* = *E*, *flows-state* = *set* (*accu*) |)) ⟹
(*set accu*) ∪ (*set edgesList*) ⊆ *E* ⟹
all-security-requirements-fulfilled (*get-IFS M*) (*stateful-policy-to-network-graph* (| *hosts* = *V*, *flows-fix* = *E*, *flows-state* = *set* (*filter-IFS-no-violations-accu* (| *nodes* = *V*, *edges* = *E* |) *M accu edgesList*) |))
apply(*induction edgesList arbitrary*: *accu*)
by(*simp-all*)
lemma *filter-IFS-no-violations-correct*: ⟦*valid-reqs* (*get-IFS M*); *wf-graph G*;
all-security-requirements-fulfilled (*get-IFS M*) *G*;
(*set edgesList*) ⊆ *edges G* ⟧ ⟹
all-security-requirements-fulfilled (*get-IFS M*) (*stateful-policy-to-network-graph* (| *hosts* = *nodes G*, *flows-fix* = *edges G*, *flows-state* = *set* (*filter-IFS-no-violations G M edgesList*) |))
unfolding *filter-IFS-no-violations-def*
apply(*case-tac G*, *simp*)
apply(*drule*(*1*) *filter-IFS-no-violations-accu-correct-induction*[**where** *accu*=[], *simplified*])
apply(*simp-all*)
by(*simp add*: *stateful-policy-to-network-graph-def all-flows-def backflows-def False-set*)
lemma *filter-IFS-no-violations-accu-no-IFS*: *valid-reqs* (*get-IFS M*) ⟹ *wf-graph G* ⟹ *get-IFS M* = [] ⟹
(*set accu*) ∪ (*set edgesList*) ⊆ *edges G* ⟹
filter-IFS-no-violations-accu G M accu edgesList = *rev*(*edgesList*)@*accu*
apply(*induction edgesList arbitrary*: *accu*)

by(*simp-all add*: *all-security-requirements-fulfilled-def*)

lemma *filter-IFS-no-violations-accu-maximal-induction*: *valid-reqs* (*get-IFS M*) ⟹ *wf-graph* (| *nodes* = *V*, *edges* = *E* |) ⟹
set accu ⊆ *E* ⟹ *set edgesList* ⊆ *E* ⟹
∀ *e* ∈ *E* − (*set accu* ∪ *set edgesList*).
¬ *all-security-requirements-fulfilled* (*get-IFS M*) (*stateful-policy-to-network-graph* (| *hosts* = *V*, *flows-fix* = *E*, *flows-state* = {*e*} ∪ (*set accu*) |))
⟹
let stateful = *set* (*filter-IFS-no-violations-accu* (| *nodes* = *V*, *edges* = *E* |) *M accu edgesList*) *in*
(∀ *e* ∈ *E* − *stateful*.
¬ *all-security-requirements-fulfilled* (*get-IFS M*) (*stateful-policy-to-network-graph* (| *hosts* = *V*, *flows-fix* = *E*, *flows-state* = {*e*} ∪ *stateful* |)))
proof(*induction edgesList arbitrary*: *accu*)
case *Nil* **thus** *?case* **by**(*simp add*: *Let-def*)
next
case(*Cons e Es*)
from *Cons.prems*(*3*) *Cons.prems*(*2*) **have** *fst* ‘ *set accu* ⊆ *V* **and** *snd* ‘ *set accu* ⊆ *V*
by(*auto simp add*: *wf-graph-def*)
— *wf-graph* for some complicated structures
from *Cons.prems*(*2*) *this Cons.prems*(*4*) **have** ⋀*ea*. *ea*∈*E* ⟹ *wf-graph* (| *nodes* = *V*, *edges* = *insert e* (*insert ea* (*set accu*)) |)
by(*auto simp add*: *wf-graph-def*)
from *backflows-wf*[*OF this*] *wf-graph-union-edges*[*OF Cons.prems*(*2*)]
have ⋀*ea*. *ea*∈*E* ⟹ *wf-graph* (| *nodes* = *V*, *edges* = *E* ∪ *backflows* (*insert e* (*insert ea* (*set accu*))) |) **by** (*simp*)
hence ⋀*ea*. *ea*∈*E* ⟹ *wf-graph* (| *nodes* = *V*, *edges* = *E* ∪ *set accu* ∪ *backflows* (*insert e* (*insert ea* (*set accu*))) |)
by (*metis Cons.prems*(*3*) *sup.order-iff*)
from *this Cons.prems*(*4*)
have ⋀*ea*. *ea*∈*E* ⟹ *wf-graph* (| *nodes* = *V*, *edges* = *insert e* (*insert ea* (*E* ∪ *set accu* ∪ *backflows* (*insert e* (*insert ea* (*set accu*))))) |)
by(*simp add*: *insert-absorb*)
hence *validgraph1*: ⋀*ea*. *ea*∈*E* − (*set* (*e* # *accu*) ∪ *set Es*) ⟹
wf-graph (| *nodes* = *V*, *edges* = *insert e* (*insert ea* (*E* ∪ *set accu* ∪ *backflows* (*insert e* (*insert ea* (*set accu*))))) |) **by**(*simp*)

have *validgraph2*: ⋀ *ea*.
insert ea (*E* ∪ *set accu* ∪ *backflows* (*insert ea* (*set accu*))) ⊆ *insert e* (*insert ea* (*E* ∪ *set accu* ∪ *backflows* (*insert e* (*insert ea* (*set accu*)))))
apply(*simp add*: *backflows-def*)
by *blast*

from *all-security-requirements-fulfilled-mono*[*OF Cons.prems*(*1*) *validgraph2 validgraph1*] **have** *neg-mono*:
⋀*ea*. *ea* ∈ *E* − (*set* (*e* # *accu*) ∪ *set Es*) ⟹
¬ *all-security-requirements-fulfilled* (*get-IFS M*)
(|*nodes* = *V*, *edges* = *insert ea* (*E* ∪ *set accu* ∪ *backflows* (*insert ea* (*set accu*)))|)
⟹
¬ *all-security-requirements-fulfilled* (*get-IFS M*)
(|*nodes* = *V*, *edges* = *insert e* (*insert ea* (*E* ∪ *set accu* ∪ *backflows* (*insert e* (*insert ea* (*set accu*)))))|)

apply(*simp*)
by *blast*

from *Cons.prems*(*5*) **have** $\bigwedge$*ea. ea*∈*E* − (*set* (*e* # *accu*) ∪ *set Es*) ⟹
¬ *all-security-requirements-fulfilled* (*get-IFS M*) (*stateful-policy-to-network-graph*
(|*hosts* = *V*, *flows-fix* = *E*, *flows-state* = {*ea*} ∪ *set* (*e* # *accu*)|))
apply(*erule-tac x*=*ea* **in** *ballE*)
prefer *2*
apply *simp*
apply(*simp only*: *stateful-policy-to-network-graph-def all-flows-def stateful-policy.select-convs*)
apply(*simp*)
apply(*frule*(*1*) *neg-mono*[*simplified*])
by(*simp*)
hence *goalTrue*:
∀ *ea*∈*E* − (*set* (*e* # *accu*) ∪ *set Es*).
¬ *all-security-requirements-fulfilled* (*get-IFS M*)
(*stateful-policy-to-network-graph* (|*hosts* = *V*, *flows-fix* = *E*, *flows-state* = {*ea*} ∪ *set* (*e* # *accu*)|))
by *simp*

show *?case*
apply(*simp add*: *Let-def*)
apply(*rule conjI*)

apply(*rule impI*)
apply(*thin-tac* -)
using *Cons.IH*[**where** *accu*=*e* # *accu*, *OF Cons.prems*(*1*) *Cons.prems*(*2*) - - *goalTrue*, *simplified Let-def*] *Cons.prems*(*3*) *Cons.prems*(*4*)
apply(*auto*) [*1*]

apply(*rule impI*)
using *Cons.IH*[**where** *accu*=*accu*, *OF Cons.prems*(*1*) *Cons.prems*(*2*), *simplified Let-def*] *Cons.prems*(*5*) *Cons.prems*(*3*) *Cons.prems*(*4*)
apply(*auto*)
done
qed
lemma *filter-IFS-no-violations-maximal*: ⟦*valid-reqs* (*get-IFS M*); *wf-graph G*;
(*set edgesList*) = *edges G* ⟧ ⟹
let stateful = *set* (*filter-IFS-no-violations G M edgesList*) *in*
∀ *e* ∈ *edges G* − *stateful*.
¬ *all-security-requirements-fulfilled* (*get-IFS M*) (*stateful-policy-to-network-graph* (| *hosts* = *nodes G*, *flows-fix* = *edges G*, *flows-state* = {*e*} ∪ *stateful* |))
unfolding *filter-IFS-no-violations-def*
apply(*case-tac G*, *simp*)
apply(*drule*(*1*) *filter-IFS-no-violations-accu-maximal-induction*[**where** *accu*=[] **and** *edgesList*=*edgesList*])
by(*simp-all*)

— It is not only maximal for single flows but all non-empty subsets
corollary *filter-IFS-no-violations-maximal-allsubsets*:
assumes *a1*: *valid-reqs* (*get-IFS M*)
and *a2*: *wf-graph G*
and *a4*: (*set edgesList*) = *edges G*
shows *let stateful* = *set* (*filter-IFS-no-violations G M edgesList*) *in*
∀ *E* ⊆ *edges G* − *stateful*. *E* ≠ {} ⟶

¬ all-security-requirements-fulfilled (get-IFS M) (stateful-policy-to-network-graph (| hosts = nodes G, flows-fix = edges G, flows-state = E ∪ stateful |))
proof −
let ?stateful = set (filter-IFS-no-violations G M edgesList)
from filter-IFS-no-violations-maximal[OF a1 a2 a4] **have** not-fulfilled-single:
∀ e∈edges G − ?stateful. ¬ all-security-requirements-fulfilled (get-IFS M)
(stateful-policy-to-network-graph (|hosts = nodes G, flows-fix = edges G, flows-state = {e} ∪ ?stateful|))
by(simp add: Let-def)
have neg-mono:
⋀ e E. e ∈ E ⟹ E ⊆ edges G − ?stateful ⟹ E ≠ {} ⟹
¬ all-security-requirements-fulfilled (get-IFS M)
(stateful-policy-to-network-graph (|hosts = nodes G, flows-fix = edges G, flows-state = {e} ∪ ?stateful|)) ⟹
¬ all-security-requirements-fulfilled (get-IFS M)
(stateful-policy-to-network-graph (|hosts = nodes G, flows-fix = edges G, flows-state = E ∪ ?stateful|))
proof −
fix e E
assume h1: e ∈ E
and h2: E ⊆ edges G − ?stateful
and h3: E ≠ {}
and h4: ¬ all-security-requirements-fulfilled (get-IFS M)
(stateful-policy-to-network-graph (|hosts = nodes G, flows-fix = edges G, flows-state = {e} ∪ ?stateful|))

from filter-IFS-no-violations-subseteq-input a4 **have** ?stateful ⊆ edges G **by** blast
hence edges G ∪ (E ∪ ?stateful) = edges G **using** h2 **by** blast
from a2 this **have** validgraph1: wf-graph (|nodes = nodes G, edges = edges G ∪ (E ∪ ?stateful)|)
by(case-tac G, simp)

from h1 h2 h3 **have** subseteq: ({e} ∪ ?stateful) ⊆ (E ∪ ?stateful) **by** blast

have revimp: ⋀A B. (A ⟹ B) ⟹ (¬ B ⟹ ¬ A) **by** fast

from all-security-requirements-fulfilled-mono-stateful-policy-to-network-graph[OF a1 subseteq validgraph1] h4
show ¬ all-security-requirements-fulfilled (get-IFS M)
(stateful-policy-to-network-graph (|hosts = nodes G, flows-fix = edges G, flows-state = E ∪ ?stateful|))
apply(rule revimp)
by assumption
qed

show ?thesis
proof(simp add: Let-def,rule allI,rule impI, rule impI)
fix E
assume h1: E ⊆ edges G − ?stateful
and h2: E ≠ {}

from h1 h2 **obtain** e **where** e-prop1: e ∈ E **by** blast
from this h1 **have** e ∈ edges G − ?stateful **by** blast
from this not-fulfilled-single **have** e-prop2: ¬ all-security-requirements-fulfilled (get-IFS M)
(stateful-policy-to-network-graph (|hosts = nodes G, flows-fix = edges G, flows-state = {e} ∪

?stateful⦈)
by *simp*

from *neg-mono*[*OF e-prop1 h1 h2 e-prop2*]
show ¬ *all-security-requirements-fulfilled* (*get-IFS M*)
(*stateful-policy-to-network-graph* ⦇*hosts* = *nodes G*, *flows-fix* = *edges G*, *flows-state* = *E* ∪ *set* (*filter-IFS-no-violations G M edgesList*)⦈)
.
qed
qed

— soundness and completeness
thm *filter-IFS-no-violations-correct filter-IFS-no-violations-maximal*

Next

fun *filter-compliant-stateful-ACS-accu* :: ′*v*::*vertex graph* ⇒ ′*v SecurityInvariant-configured list* ⇒ (′*v* × ′*v*) *list* ⇒ (′*v* × ′*v*) *list* ⇒ (′*v* × ′*v*) *list* **where**
filter-compliant-stateful-ACS-accu G M accu [] = *accu* |
filter-compliant-stateful-ACS-accu G M accu (*e*#*Es*) = (*if*
e ∉ *backflows* (*edges G*) ∧ (∀ *F* ∈ *get-offending-flows* (*get-ACS M*) (*stateful-policy-to-network-graph* ⦇ *hosts* = *nodes G*, *flows-fix* = *edges G*, *flows-state* = *set* (*e*#*accu*) ⦈). *F* ⊆ *backflows* (*set* (*e*#*accu*)))
then filter-compliant-stateful-ACS-accu G M (*e*#*accu*) *Es*
else filter-compliant-stateful-ACS-accu G M accu Es)
definition *filter-compliant-stateful-ACS* :: ′*v*::*vertex graph* ⇒ ′*v SecurityInvariant-configured list* ⇒ (′*v* × ′*v*) *list* ⇒ (′*v* × ′*v*) *list* **where**
filter-compliant-stateful-ACS G M Es = *filter-compliant-stateful-ACS-accu G M* [] *Es*

lemma *filter-compliant-stateful-ACS-subseteq-input*: *set* (*filter-compliant-stateful-ACS G M Es*) ⊆ *set Es*
apply(*subgoal-tac* ∀ *accu*. *set* (*filter-compliant-stateful-ACS-accu G M accu Es*) ⊆ *set Es* ∪ *set accu*)
apply(*erule-tac x*=[] **in** *allE*)
apply(*simp add*: *filter-compliant-stateful-ACS-def*)
apply(*induct-tac Es*)
apply(*simp-all*)
apply (*metis Un-insert-right set-simps*(*2*) *set-subset-Cons set-union subset-trans*)
done
lemma *filter-compliant-stateful-ACS-accu-correct-induction*: *valid-reqs* (*get-ACS M*) ⟹ *wf-graph* ⦇ *nodes* = *V*, *edges* = *E* ⦈ ⟹
(*set accu*) ∪ (*set edgesList*) ⊆ *E* ⟹
∀ *F* ∈ *get-offending-flows* (*get-ACS M*) (*stateful-policy-to-network-graph* ⦇ *hosts* = *V*, *flows-fix* = *E*, *flows-state* = *set* (*accu*) ⦈). *F* ⊆ *backflows* (*set accu*) ⟹
(∀ *a* ∈ *set accu*. *a* ∉ (*backflows E*)) ⟹
$\mathcal{T}$ = ⦇ *hosts* = *V*, *flows-fix* = *E*, *flows-state* = *set* (*filter-compliant-stateful-ACS-accu* ⦇ *nodes* = *V*, *edges* = *E* ⦈ *M accu edgesList*) ⦈ ⟹
∀ *F* ∈ *get-offending-flows* (*get-ACS M*) (*stateful-policy-to-network-graph* $\mathcal{T}$). *F* ⊆ *backflows* (*filternew-flows-state* $\mathcal{T}$)
proof(*induction edgesList arbitrary*: *accu*)
case *Nil*
from *Nil*(*5*) **have** *backflows* (*set accu*) = *backflows* {*e* ∈ *set accu*. *e* ∉ *backflows E*} **by** (*metis* (*lifting*) *Collect-cong Collect-mem-eq*)
from *this Nil*(*4*) **have** ∀ *F*∈*get-offending-flows* (*get-ACS M*) (*stateful-policy-to-network-graph* ⦇*hosts* = *V*, *flows-fix* = *E*, *flows-state* = *set accu*⦈). *F* ⊆ *backflows* {*e* ∈ *set accu*. *e* ∉ *backflows E*}

by *simp*
from *this Nil*(*6*) **show** *?case* **by**(*simp add*: *filternew-flows-state-alt2*)
next
case (*Cons e Es*)
from *Cons.IH*[*OF Cons.prems*(*1*) *Cons.prems*(*2*)] *Cons.prems*(*3*) *Cons.prems*(*4*) *Cons.prems*(*5*) *Cons.prems*(*6*)
show *?case* **by**(*simp add*: *filternew-flows-state-alt2 split*: *if-split-asm*)
qed

lemma *filter-compliant-stateful-ACS-accu-no-side-effects*: *valid-reqs* (*get-ACS M*) $\Longrightarrow$ *wf-graph G* $\Longrightarrow$
$\forall$ *F* $\in$ *get-offending-flows* (*get-ACS M*) (|*nodes* = *nodes G*, *edges* = *edges G* $\cup$ *backflows* (*edges G*)|). *F* $\subseteq$ (*backflows* (*edges G*)) $-$ (*edges G*) $\Longrightarrow$
(*set accu*) $\cup$ (*set edgesList*) $\subseteq$ *edges G* $\Longrightarrow$
($\forall$ *a* $\in$ *set accu*. *a* $\notin$ (*backflows* (*edges G*))) $\Longrightarrow$
filter-compliant-stateful-ACS-accu G M accu edgesList = *rev*([*e* $\leftarrow$ *edgesList*. *e* $\notin$ *backflows* (*edges G*)])@*accu*
apply(*simp add*: *backflows-minus-backflows*)
apply(*induction edgesList arbitrary*: *accu*)
apply(*simp*)
apply(*simp add*: *stateful-policy-to-network-graph-def all-flows-def*)
apply(*rule impI*)
apply(*case-tac G*, *simp*, *rename-tac V E*)
thm *Un-set-offending-flows-bound-minus-subseteq'*[**where** *X*=*backflows E* $-$ *E* **and** *E*=*E* $\cup$ *backflows E*]
apply(*drule-tac X*=*backflows E* $-$ *E* **and** *E*=*E* $\cup$ *backflows E* **and** *E'*=(*E* $\cup$ *backflows E*) $-$ (*insert a* (*E* $\cup$ *set accu* $\cup$ *backflows* (*insert a* (*set accu*)))) **in** *Un-set-offending-flows-bound-minus-subseteq'*)
defer
prefer *2*
apply *blast*
apply *auto*[*1*]
apply(*subgoal-tac E* $\cup$ *backflows E* $-$ (*E* $\cup$ *backflows E* $-$ *insert a* (*E* $\cup$ *set accu* $\cup$ *backflows* (*insert a* (*set accu*)))) = *insert a* (*E* $\cup$ *set accu* $\cup$ *backflows* (*insert a* (*set accu*))))
apply(*simp*)
prefer *2*
apply (*metis Un-assoc Un-least Un-mono backflows-subseteq double-diff insert-def insert-subset subset-refl*)
apply(*subgoal-tac backflows* (*insert a* (*set accu*)) $\subseteq$ *backflows E* $-$ *E* $-$ (*E* $\cup$ *backflows E* $-$ *insert a* (*E* $\cup$ *set accu* $\cup$ *backflows* (*insert a* (*set accu*)))))
apply(*blast*)
apply(*simp add*: *backflows-def*)
apply *fast*
using *FiniteGraph.backflows-wf FiniteGraph.wf-graph-union-edges* **by** *metis*

lemma *filter-compliant-stateful-ACS-correct*:
assumes *a1*: *valid-reqs* (*get-ACS M*)
and *a2*: *wf-graph G*
and *a3*: *set edgesList* $\subseteq$ *edges G*
and *a4*: *all-security-requirements-fulfilled* (*get-ACS M*) *G*
and *a5*: $\mathcal{T}$ = (| *hosts* = *nodes G*, *flows-fix* = *edges G*, *flows-state* = *set* (*filter-compliant-stateful-ACS G M edgesList*) |)

shows ∀ *F* ∈ *get-offending-flows* (*get-ACS M*) (*stateful-policy-to-network-graph* 𝒯). *F* ⊆ *backflows* (*filternew-flows-state* 𝒯)

proof −

obtain *V E* **where** *VE*: *G* = (| *nodes* = *V*, *edges* = *E* |) **by**(*case-tac G*, *blast*)

from *VE a2* **have** *wfVE*: *wf-graph* (| *nodes* = *V*, *edges* = *E* |) **by** *simp*

from *VE a3* **have** *set edgesList* ⊆ *E* **by** *simp*

from *a5 VE* **have** *a5'*: 𝒯 = (|*hosts* = *V*, *flows-fix* = *E*, *flows-state* = *set* (*filter-compliant-stateful-ACS-accu* (|*nodes* = *V*, *edges* = *E*|) *M* [] *edgesList*)|)

unfolding *filter-compliant-stateful-ACS-def*

by(*simp*)

from *all-security-requirements-fulfilled-imp-get-offending-empty*[*OF a1 a4*] *VE*

have ∀ *F*∈*get-offending-flows* (*get-ACS M*) (*stateful-policy-to-network-graph* (|*hosts* = *V*, *flows-fix* = *E*, *flows-state* = {}|)). *F* ⊆ *backflows* {}

by(*simp add*: *stateful-policy-to-network-graph-def all-flows-def backflows-def False-set*)

from *filter-compliant-stateful-ACS-accu-correct-induction*[**where** *accu*=[] **and** *edgesList*=*edgesList*, *simplified*, *OF a1 wfVE* ‹*set edgesList* ⊆ *E*› *this a5'*]

show *?thesis* **.**

qed

lemma *filter-compliant-stateful-ACS-accu-induction-maximal*:⟦ *valid-reqs* (*get-ACS M*); *wf-graph* (| *nodes* = *V*, *edges* = *E* |);

(*set edgesList*) ⊆ *E*;

(*set accu*) ⊆ *E*;

stateful = *set* (*filter-compliant-stateful-ACS-accu* (| *nodes* = *V*, *edges* = *E* |) *M accu edgesList*);

∀ *e* ∈ *E* − (*set edgesList* ∪ *set accu* ∪ {*e* ∈ *E*. *e* ∈ *backflows E*}).

¬ ⋃(*get-offending-flows* (*get-ACS M*) (*stateful-policy-to-network-graph* (| *hosts* = *V*, *flows-fix* = *E*, *flows-state* = *set accu* ∪ {*e*} |)))

⊆ *backflows* (*filternew-flows-state* (| *hosts* = *V*, *flows-fix* = *E*, *flows-state* = *set accu* ∪ {*e*} |))

⟧ ⟹

∀ *e* ∈ *E* − (*stateful* ∪ {*e* ∈ *E*. *e* ∈ *backflows E*}). ~~*F* /// (computed stateful flows plus trivial stateful flows)~~

¬ ⋃(*get-offending-flows* (*get-ACS M*) (*stateful-policy-to-network-graph* (| *hosts* = *V*, *flows-fix* = *E*, *flows-state* = *stateful* ∪ {*e*} |)))

⊆ *backflows* (*filternew-flows-state* (| *hosts* = *V*, *flows-fix* = *E*, *flows-state* = *stateful* ∪ {*e*} |))

proof(*induction edgesList arbitrary*: *accu E*)

case *Nil* **from** *Nil*(*5*)[*simplified*] *Nil*(*6*) **show** *?case* **by**(*simp*)

next

case (*Cons a Es*)

— case distinction

let *?caseDistinction*=*a* ∉ *backflows* (*E*) ∧ (∀ *F*∈*get-offending-flows* (*get-ACS M*)

(*stateful-policy-to-network-graph* (|*hosts* = *V*, *flows-fix* = *E*, *flows-state* = *set* (*a* # *accu*)|)).

F ⊆ *backflows* (*set* (*a* # *accu*)))

from *Cons.prems*(*3*) **have** *set Es* ⊆ *E* **by** *simp*

show *?case*

proof(*cases ?caseDistinction*)

assume *CaseTrue*: *?caseDistinction*

from *CaseTrue* **have**
set (*filter-compliant-stateful-ACS-accu* (|*nodes* = *V*, *edges* = *E*|) *M accu* (*a* # *Es*)) =
set (*filter-compliant-stateful-ACS-accu* (|*nodes* = *V*, *edges* = *E*|) *M* (*a* # *accu*) *Es*)
by(*simp*)
from *this Cons.prems*(*5*) **have** *statefulsimp*:
stateful = *set* (*filter-compliant-stateful-ACS-accu* (|*nodes* = *V*, *edges* = *E*|) *M* (*a* # *accu*) *Es*)
by *simp*
from *Cons.prems*(*3*) *Cons.prems*(*4*) **have** *set* (*a* # *accu*) ⊆ *E* **by** *simp*

have ∀ *e*∈*E* − (*set Es* ∪ *set* (*a* # *accu*) ∪ {*e* ∈ *E*. *e* ∈ *backflows E*}).
¬ ⋃(*get-offending-flows* (*get-ACS M*) (*stateful-policy-to-network-graph* (|*hosts* = *V*, *flows-fix* = *E*, *flows-state* = *set* (*a* # *accu*) ∪ {*e*}|)))
⊆ *backflows* (*filternew-flows-state* (|*hosts* = *V*, *flows-fix* = *E*, *flows-state* = *set* (*a* # *accu*) ∪ {*e*}|))
proof(*rule ballI*)
fix *e*
assume *h1*: *e* ∈ *E* − (*set Es* ∪ *set* (*a* # *accu*) ∪ {*e* ∈ *E*. *e* ∈ *backflows E*})

from *conjunct1*[*OF CaseTrue*] **have** *filternew-flows-state-moveout-a*:
filternew-flows-state (|*hosts* = *V*, *flows-fix* = *E*, *flows-state* = *set* (*a* # *accu*) ∪ {*e*}|) =
{*a*} ∪ *filternew-flows-state* (|*hosts* = *V*, *flows-fix* = *E*, *flows-state* = *set accu* ∪ {*e*}|)
apply(*simp add*: *filternew-flows-state-alt*) **by** *blast*

have *backflowssubseta*: ⋀*X*. *backflows X* ⊆ *backflows* ({*a*} ∪ *X*) **by**(*simp add*: *backflows-def*, *blast*)

from *Cons.prems*(*6*) *h1* **have**
¬ ⋃(*get-offending-flows* (*get-ACS M*) (*stateful-policy-to-network-graph* (|*hosts* = *V*, *flows-fix* = *E*, *flows-state* = *set accu* ∪ {*e*}|)))
⊆ *backflows* (*filternew-flows-state* (|*hosts* = *V*, *flows-fix* = *E*, *flows-state* = *set accu* ∪ {*e*}|)) **by** *simp*
from *this* **obtain** *dat-offender* **where**
dat-in: *dat-offender* ∈ ⋃(*get-offending-flows* (*get-ACS M*) (*stateful-policy-to-network-graph* (|*hosts* = *V*, *flows-fix* = *E*, *flows-state* = *set accu* ∪ {*e*}|)))
and *dat-offends*: *dat-offender* ∉ *backflows* (*filternew-flows-state* (|*hosts* = *V*, *flows-fix* = *E*, *flows-state* = *set accu* ∪ {*e*}|)) **by** *blast*

have *wfGraphA*: *wf-graph* (*stateful-policy-to-network-graph* (|*hosts* = *V*, *flows-fix* = *E*, *flows-state* = *set* (*a* # *accu*) ∪ {*e*}|))
proof(*simp add*: *stateful-policy-to-network-graph-def all-flows-def*)
from *Cons.prems*(*2*) *h1 Cons.prems*(*3*) *Cons.prems*(*4*)
have *wf-graph* (|*nodes*=*V*, *edges* = *insert e* (*insert a* (*set accu*)) |)
apply(*auto simp add*: *wf-graph-def*) **by** *force*
from *this backflows-wf*
have *vgh1*: *wf-graph* (|*nodes* = *V*, *edges* = *backflows* (*insert e* (*insert a* (*set accu*)))|) **by** *auto*
from *Cons.prems*(*2*) *wf-graph-add-subset-edges h1* *Cons.prems*(*3*) *Cons.prems*(*4*)
have *vgh2*: *wf-graph* (|*nodes* = *V*, *edges* = *insert e* ((*insert a E*) ∪ *set accu*)|)
proof −
have *f1*: *e* ∈ *E* − (*set Es* ∪ *insert a* (*set accu*) ∪ {*R* ∈ *E*. *R* ∈ *backflows E*})
using *h1* **by** *simp*
have *f2*: *insert a* (*set accu*) ⊆ *E*
using ‹*set* (*a* # *accu*) ⊆ *E*› **by** *simp*
have *f3*: *e* ∈ *E*

using *f1* **by** *fastforce*
have $E \cup$ *insert a* (*set accu*) = E
using *f2* **by** *fastforce*
thus *wf-graph* (|*nodes* = V, *edges* = *insert e* (*insert a* $E \cup$ *set accu*)|)
using *f3 Cons.prems*(*2*) *Un-insert-right insert-absorb sup-commute* **by** *fastforce*
qed
from *vgh1 vgh2 wf-graph-union-edges*
show *wf-graph* (|*nodes* = V, *edges* = *insert e* (*insert a* ($E \cup$ *set accu* $\cup$ *backflows* (*insert e* (*insert a* (*set accu*)))))|) **by** *fastforce*
qed

from *dat-in* **have** *dat-in-simplified*:
dat-offender $\in \bigcup$(*get-offending-flows* (*get-ACS M*) (|*nodes* = V, *edges* = *insert e* ($E \cup$ *set accu* $\cup$ *backflows* (*insert e* (*set accu*)))|))
by(*simp add*: *stateful-policy-to-network-graph-def all-flows-def*)

have *subsethlp*: *insert e* ($E \cup$ *set accu* $\cup$ *backflows* (*insert e* (*set accu*))) $\subseteq$ $E \cup$ (*set* (*a* # *accu*) $\cup$ {*e*}) $\cup$ *backflows* (*set* (*a* # *accu*) $\cup$ {*e*})
apply(*simp*)
apply(*rule*, *blast*)
apply(*rule*, *blast*)
apply(*rule*)
apply(*simp add*: *backflows-def*, *fast*)
done

from *get-offending-flows-union-mono*[*OF*
Cons.prems(*1*)
wfGraphA[*simplified stateful-policy-to-network-graph-def all-flows-def graph.select-convs stateful-policy.select-convs*],
OF subsethlp]
dat-in-simplified **have** *dat-in-a*: *dat-offender* $\in \bigcup$(*get-offending-flows* (*get-ACS M*)
(*stateful-policy-to-network-graph* (|*hosts* = V, *flows-fix* = E, *flows-state* = *set* (*a* # *accu*) $\cup$ {*e*}|)))
by(*simp add*: *stateful-policy-to-network-graph-def all-flows-def*, *fast*)

have *dat-offender* $\neq$ (*snd a*, *fst a*)
proof(*rule ccontr*)
assume $\neg$ *dat-offender* $\neq$ (*snd a*, *fst a*)
hence *hlpassm*: *dat-offender* = (*snd a*, *fst a*) **by** *simp*
from *this* **obtain** *a1 a2* **where** *dat-offender* = (*a2*, *a1*) **by** *blast*

have $\bigcup$(*get-offending-flows* (*get-ACS M*) (|*nodes* = V, *edges* = *insert e* ($E \cup$ *set accu* $\cup$ *backflows* (*insert e* (*set accu*)))|)) $\subseteq$
insert e ($E \cup$ *set accu* $\cup$ *backflows* (*insert e* (*set accu*)))
by (*metis Cons.prems*(*1*) *Sup-le-iff get-offending-flows-subseteq-edges*)
from *this h1* **have** *UN-get-subset*:
$\bigcup$(*get-offending-flows* (*get-ACS M*) (|*nodes* = V, *edges* = *insert e* ($E \cup$ *set accu* $\cup$ *backflows* (*insert e* (*set accu*)))|)) $\subseteq$
($E \cup$ *set accu* $\cup$ *backflows* (*insert e* (*set accu*)))
by *blast*

from *dat-offends* **have** *dat-offends-simplified*:
dat-offender $\notin$ *backflows* (*insert e* (*set accu*)) $-$ E

by(*simp only*: *filternew-flows-state-alt stateful-policy.select-convs backflows-minus-backflows*, *simp*)

from *conjunct1*[*OF CaseTrue*] *hlpassm* **have** *dat-offender* ∉ *E*
by(*simp add*: *backflows-def*, *fastforce*)
from *dat-in-simplified UN-get-subset this* **have** *dat-offender* ∈ *set accu* ∪ *backflows* (*insert e* (*set accu*)) **by** *blast*
from *this Cons.prems*(*4*) ‹*dat-offender* ∉ *E*› **have** *dat-offender* ∈ *backflows* (*insert e* (*set accu*)) **by** *blast*
from *dat-offends-simplified*[*simplified*] *this* **have** *dat-offender* ∈ *E* **by** *simp*
from ‹*dat-offender* ∉ *E*› ‹*dat-offender* ∈ *E*› **show** *False* **by** *simp*
qed

from *this dat-offends* **have**
dat-offender ∉ *backflows* ({*a*} ∪ *filternew-flows-state* (|*hosts* = *V*, *flows-fix* = *E*, *flows-state* = *set accu* ∪ {*e*}|))
apply(*simp add*: *backflows-def*) **by** *force*

from *dat-in-a this*
show ¬ ⋃(*get-offending-flows* (*get-ACS M*) (*stateful-policy-to-network-graph* (|*hosts* = *V*, *flows-fix* = *E*, *flows-state* = *set* (*a* # *accu*) ∪ {*e*}|)))
⊆ *backflows* (*filternew-flows-state* (|*hosts* = *V*, *flows-fix* = *E*, *flows-state* = *set* (*a* # *accu*) ∪ {*e*}|))
apply(*subst filternew-flows-state-moveout-a*) **by** *blast*
qed

from *Cons.IH*[*OF Cons.prems*(*1*) *Cons.prems*(*2*) ‹*set Es* ⊆ *E*› ‹*set* (*a* # *accu*) ⊆ *E*› *statefulsimp this*] **show** *?case*
by(*simp*)
next
assume *CaseFalse*: ¬ *?caseDistinction*

from *CaseFalse* **have** *funapplysimp*:
set (*filter-compliant-stateful-ACS-accu* (|*nodes* = *V*, *edges* = *E*|) *M accu* (*a* # *Es*)) =
set (*filter-compliant-stateful-ACS-accu* (|*nodes* = *V*, *edges* = *E*|) *M accu Es*)
by *auto*
from *this Cons.prems*(*5*) **have** *statefulsimp*:
stateful = *set* (*filter-compliant-stateful-ACS-accu* (|*nodes* = *V*, *edges* = *E*|) *M accu Es*) **by** *simp*
from *Cons.prems*(*4*) **have** *set accu* ⊆ *E* **.**

have *a* ∈ *E* − (*set Es* ∪ *set accu* ∪ {*e* ∈ *E*. *e* ∈ *backflows E*})⟹ ¬ ⋃(*get-offending-flows* (*get-ACS M*) (*stateful-policy-to-network-graph* (|*hosts* = *V*, *flows-fix* = *E*, *flows-state* = *set accu* ∪ {*a*}|)))
⊆ *backflows* (*filternew-flows-state* (|*hosts* = *V*, *flows-fix* = *E*, *flows-state* = *set accu* ∪ {*a*}|))
proof(*rule ccontr*)
assume *h1*: *a* ∈ *E* − (*set Es* ∪ *set accu* ∪ {*e* ∈ *E*. *e* ∈ *backflows E*})
and ¬ ¬⋃(*get-offending-flows* (*get-ACS M*) (*stateful-policy-to-network-graph* (|*hosts* = *V*, *flows-fix* = *E*, *flows-state* = *set accu* ∪ {*a*}|))) ⊆ *backflows* (*filternew-flows-state* (|*hosts* = *V*, *flows-fix* = *E*, *flows-state* = *set accu* ∪ {*a*}|))
hence *hccontr*: ⋃(*get-offending-flows* (*get-ACS M*) (*stateful-policy-to-network-graph* (|*hosts* = *V*, *flows-fix* = *E*, *flows-state* = *set accu* ∪ {*a*}|))) ⊆ *backflows* (*filternew-flows-state* (|*hosts* = *V*, *flows-fix* = *E*, *flows-state* = *set accu* ∪ {*a*}|)) **by** *simp*

moreover from *h1* **have** *stateful-to-graph*: *stateful-policy-to-network-graph* (|*hosts* = *V*, *flows-fix* = *E*, *flows-state* = *set accu* ∪ {*a*}|) = (|*nodes* = *V*, *edges* = *E* ∪ *set accu* ∪ *backflows* (*insert a* (*set accu*))|)
by(*simp add*: *stateful-policy-to-network-graph-def all-flows-def*, *blast*)
moreover have *backflows* (*filternew-flows-state* (|*hosts* = *V*, *flows-fix* = *E*, *flows-state* = *set accu* ∪ {*a*}|)) = *backflows* (*insert a* (*set accu*)) − *E*
by(*simp add*: *filternew-flows-state-alt backflows-minus-backflows*)
ultimately have *hccontr-simp*:
⋃(*get-offending-flows* (*get-ACS M*) (|*nodes* = *V*, *edges* = *E* ∪ *set accu* ∪ *backflows* (*insert a* (*set accu*))|)) ⊆ *backflows* (*insert a* (*set accu*)) − *E* **by** *simp*

from *Cons.prems*(*3*) *Cons.prems*(*4*) **have** *backaaccusubE*: *backflows* (*set* (*a* # *accu*)) ⊆ *backflows E* **by**(*simp add*: *backflows-def*, *fastforce*)
from *h1* **have** *a* ∉ *backflows E* **by** *fastforce*
from *backaaccusubE* ‹*a* ∉ *backflows E*› **have** *a* ∉ *backflows* (*insert a* (*set accu*)) **by** *auto*

from ‹*a* ∉ *backflows E*› *CaseFalse* **have** ¬ (∀ *F*∈*get-offending-flows* (*get-ACS M*) (*stateful-policy-to-network-graph* (|*hosts* = *V*, *flows-fix* = *E*, *flows-state* = *set* (*a* # *accu*)|)). *F* ⊆ *backflows* (*set* (*a* # *accu*))) **by**(*simp*)
from *this stateful-to-graph* **have** ¬ (∀ *F*∈*get-offending-flows* (*get-ACS M*) (|*nodes* = *V*, *edges* = *E* ∪ *set accu* ∪ *backflows* (*insert a* (*set accu*))|). *F* ⊆ *backflows* (*insert a* (*set accu*))) **by**(*simp*)
from *this hccontr-simp* **show** *False* **by** *blast*
qed
from *Cons.prems*(*6*)[*simplified funapplysimp statefulsimp*] *this*
have ∀ *e*∈*E* − (*set Es* ∪ *set accu* ∪ {*e* ∈ *E*. *e* ∈ *backflows E*}).
¬ ⋃(*get-offending-flows* (*get-ACS M*) (*stateful-policy-to-network-graph* (|*hosts* = *V*, *flows-fix* = *E*, *flows-state* = *set accu* ∪ {*e*}|)))
⊆ *backflows* (*filternew-flows-state* (|*hosts* = *V*, *flows-fix* = *E*, *flows-state* = *set accu* ∪ {*e*}|))
by *auto*

from *Cons.IH*[*OF Cons.prems*(*1*) *Cons.prems*(*2*) ‹*set Es* ⊆ *E*› ‹*set accu* ⊆ *E*› *statefulsimp this*]
show *?case* **by** *simp*
qed
qed

lemma *filter-compliant-stateful-ACS-maximal*: ⟦ *valid-reqs* (*get-ACS M*); *wf-graph* (| *nodes* = *V*, *edges* = *E* |);
(*set edgesList*) = *E*;
stateful = *set* (*filter-compliant-stateful-ACS* (| *nodes* = *V*, *edges* = *E* |) *M edgesList*)
⟧ ⟹
∀ *e* ∈ *E* − (*stateful* ∪ {*e* ∈ *E*. *e* ∈ *backflows E*}). ~~*E* — (*computed stateful flows plus trivial stateful flows*)~~
¬ ⋃ (*get-offending-flows* (*get-ACS M*) (*stateful-policy-to-network-graph* (| *hosts* = *V*, *flows-fix* = *E*, *flows-state* = *stateful* ∪ {*e*} |)))
⊆ *backflows* (*filternew-flows-state* (| *hosts* = *V*, *flows-fix* = *E*, *flows-state* = *stateful* ∪

{e} |))
  apply(*drule*(*1*) *filter-compliant-stateful-ACS-accu-induction-maximal*[**where** *accu*=[], *simplified*])
    apply(*blast*)
   apply(*simp add*: *filter-compliant-stateful-ACS-def*)
  apply(*simp*)
  apply *fastforce*
 apply(*simp add*: *filter-compliant-stateful-ACS-def*)
done

lemma *filter-compliant-stateful-ACS-maximal-allsubsets*:
  assumes *a1*: *valid-reqs* (*get-ACS M*) **and** *a2*: *wf-graph* (| *nodes* = *V*, *edges* = *E* |)
  and *a3*: (*set edgesList*) = *E*
  and *a4*: *stateful* = *set* (*filter-compliant-stateful-ACS* (| *nodes* = *V*, *edges* = *E* |) *M edgesList*)
  and *a5*: *X* ⊆ *E* − (*stateful* ∪ *backflows E*) **and** *a6*: *X* ≠ {}
  shows
  ¬ ⋃(*get-offending-flows* (*get-ACS M*) (*stateful-policy-to-network-graph* (| *hosts* = *V*, *flows-fix* = *E*, *flows-state* = *stateful* ∪ *X* |)))
      ⊆ *backflows* (*filternew-flows-state* (| *hosts* = *V*, *flows-fix* = *E*, *flows-state* = *stateful* ∪ *X* |))
 proof(*rule ccontr*, *simp*)
  from *a5* **have** *X* ⊆ *E* **by** *blast*
  assume *accontr*: ⋃(*get-offending-flows* (*get-ACS M*) (*stateful-policy-to-network-graph* (|*hosts* = *V*, *flows-fix* = *E*, *flows-state* = *stateful* ∪ *X*|))) ⊆ *backflows* (*filternew-flows-state* (|*hosts* = *V*, *flows-fix* = *E*, *flows-state* = *stateful* ∪ *X*|))
  hence ⋃(*get-offending-flows* (*get-ACS M*) (|*nodes* = *V*, *edges* = *E* ∪ (*stateful* ∪ *X*) ∪ *backflows* (*stateful* ∪ *X*)|)) ⊆ *backflows* (*stateful* ∪ *X*) − *E*
 by(*simp add*: *stateful-policy-to-network-graph-def all-flows-def filternew-flows-state-alt backflows-minus-backflows*)
  hence ⋃(*get-offending-flows* (*get-ACS M*) (|*nodes* = *V*, *edges* = *E* ∪ *X* ∪ *backflows* (*stateful* ∪ *X*)|)) ⊆ *backflows* (*stateful* ∪ *X*) − *E*
 using *a4 a3 filter-compliant-stateful-ACS-subseteq-input* **by** (*metis Diff-subset-conv Un-Diff-cancel Un-assoc a3 bot.extremum-unique sup-bot-right*)
  hence *accontr-simp*: ⋃(*get-offending-flows* (*get-ACS M*) (|*nodes* = *V*, *edges* = *E* ∪ (*backflows stateful*) ∪ (*backflows X*)|)) ⊆ *backflows* (*stateful* ∪ *X*) − *E*
  using *Set.Un-absorb2*[*OF* ‹*X* ⊆ *E*›] *backflows-un*[*of stateful X*] **by** (*metis Un-assoc*)

  from *a2 a5* **have** *finite X* **apply**(*simp add*: *wf-graph-def*) **by** (*metis* (*full-types*) *finite-Diff finite-subset*)
  from *a6* **obtain** *x* **where** *x* ∈ *X* **by** *blast*

  from ‹*x* ∈ *X*› *a5* **have** *xX-simp1*: (*backflows X*) − (*backflows* (*X* − {*x*}) − *E*) = *backflows* {*x*}
   apply(*simp add*: *backflows-def*) **by** *fast*
  from *a5* **have** *X* ∩ *stateful* = {} **by** *auto*
  from ‹*x* ∈ *X*› *this* **have** *xX-simp2*: (*backflows stateful*) − (*backflows* (*X* − {*x*}) − *E*) = *backflows stateful*
   apply(*simp add*: *backflows-def*) **by** *fast*
  have *xX-simp3*:*backflows* (*stateful* ∪ *X*) − (*backflows* (*X* − {*x*}) − *E*) = *backflows* (*stateful* ∪ {*x*})
   apply(*simp only*: *backflows-un*)
   using *xX-simp1 xX-simp2* **by** *blast*

  have *xX-simp4*: *backflows* (*stateful* ∪ *X*) − *E* − (*backflows* (*X* − {*x*}) − *E*) = *backflows* (*filternew-flows-state* (|*hosts* = *V*, *flows-fix* = *E*, *flows-state* = *stateful* ∪ {*x*}|))

apply(*simp add*: *filternew-flows-state-alt backflows-minus-backflows*)
using *xX-simp3* **by** *auto*

have *xX-simp5*: $(E \cup \mathit{backflows\ stateful} \cup \mathit{backflows}\ X) - (\mathit{backflows}\ (X - \{x\}) - E) = E \cup \mathit{backflows\ stateful} \cup \mathit{backflows}\ \{x\}$
using *xX-simp3*[*simplified backflows-un*] **by** *blast*

have *Eexpand*: $E \cup \mathit{stateful} \cup \{x\} = E$
using *a4 a3 filter-compliant-stateful-ACS-subseteq-input a5* ‹$x{\in}X$› **by** *blast*

have $\mathit{backflows}\ (\mathit{stateful} \cup X) - E - \mathit{backflows}\ (X - \{x\}) = (\mathit{backflows}\ (\mathit{stateful} \cup X) - E) - \mathit{backflows}\ (X - \{x\})$ **by** *simp*
from ‹*finite X*› *backflows-finite* **have** *finite*: *finite* $(\mathit{backflows}\ (X - \{x\}) - E)$ **by** *auto*
from *a2 a4 a3 filter-compliant-stateful-ACS-subseteq-input* **have** *wf-graph* (|*nodes* = *V*, *edges* = *stateful*|) **by** (*metis Diff-partition wf-graph-remove-edges-union*)
from *backflows-wf*[*OF this*] **have** *wf-graph* (|*nodes* = *V*, *edges* = *backflows stateful*|) **.**
from *a2* ‹$X \subseteq E$› **have** *wf-graph* (|*nodes* = *V*, *edges* = *X*|) **by** (*metis double-diff dual-order.refl wf-graph-remove-edges*)
from *backflows-wf*[*OF this*] **have** *wf-graph* (|*nodes* = *V*, *edges* = *backflows X*|) **.**
from *this wf-graph-union-edges* ‹*wf-graph* (|*nodes* = *V*, *edges* = *backflows stateful*|)› *a2* **have** *wfG*:
wf-graph (|*nodes* = *V*, *edges* = $E \cup \mathit{backflows\ stateful} \cup \mathit{backflows}\ X$|) **by** *metis*

from ‹$x{\in}X$› **have** *subset*: $\mathit{backflows}\ (X - \{x\}) - E \subseteq E \cup \mathit{backflows\ stateful} \cup \mathit{backflows}\ X$
apply(*simp add*: *backflows-def*) **by** *fast*

from *Un-set-offending-flows-bound-minus-subseteq'*[*OF a1 wfG subset accontr-simp*] **have**
$\bigcup$(*get-offending-flows* (*get-ACS M*) (|*nodes* = *V*, *edges* = $(E \cup \mathit{backflows\ stateful} \cup \mathit{backflows}\ X) - (\mathit{backflows}\ (X - \{x\}) - E)$|)) $\subseteq (\mathit{backflows}\ (\mathit{stateful} \cup X) - E) - (\mathit{backflows}\ (X - \{x\}) - E)$ **by** *simp*
from *this xX-simp4 xX-simp5* **have** *trans1*:
$\bigcup$(*get-offending-flows* (*get-ACS M*) (|*nodes* = *V*, *edges* = $E \cup \mathit{backflows\ stateful} \cup \mathit{backflows}\ \{x\}$|)) $\subseteq$ *backflows* (*filternew-flows-state* (|*hosts* = *V*, *flows-fix* = *E*, *flows-state* = $\mathit{stateful} \cup \{x\}$|)) **by** *simp*

hence $\bigcup$(*get-offending-flows* (*get-ACS M*) (|*nodes* = *V*, *edges* = $E \cup \mathit{backflows}\ (\mathit{stateful} \cup \{x\})$|)) $\subseteq$ *backflows* (*filternew-flows-state* (|*hosts* = *V*, *flows-fix* = *E*, *flows-state* = $\mathit{stateful} \cup \{x\}$|))
apply(*simp only*: *backflows-un*) **by** (*metis Un-assoc*)
hence *contr1*: $\bigcup$(*get-offending-flows* (*get-ACS M*) (*stateful-policy-to-network-graph* (|*hosts* = *V*, *flows-fix* = *E*, *flows-state* = $\mathit{stateful} \cup \{x\}$|))) $\subseteq$ *backflows* (*filternew-flows-state* (|*hosts* = *V*, *flows-fix* = *E*, *flows-state* = $\mathit{stateful} \cup \{x\}$|))
apply(*simp only*: *stateful-policy-to-network-graph-def all-flows-def stateful-policy.select-convs*)
using *Eexpand* **by** (*metis Un-assoc*)

from *filter-compliant-stateful-ACS-maximal*[*OF a1 a2 a3 a4*] **have**
$\forall e{\in}E - (\mathit{stateful} \cup \{e \in E.\ e \in \mathit{backflows}\ E\}).\ \neg \bigcup$(*get-offending-flows* (*get-ACS M*) (*stateful-policy-to-network-graph* (|*hosts* = *V*, *flows-fix* = *E*, *flows-state* = $\mathit{stateful} \cup \{e\}$|))) $\subseteq$ *backflows* (*filternew-flows-state* (|*hosts* = *V*, *flows-fix* = *E*, *flows-state* = $\mathit{stateful} \cup \{e\}$|)) **.**
from *this a5* ‹$x \in X$› **have** *contr2*: $\neg \bigcup$(*get-offending-flows* (*get-ACS M*) (*stateful-policy-to-network-graph* (|*hosts* = *V*, *flows-fix* = *E*, *flows-state* = $\mathit{stateful} \cup \{x\}$|))) $\subseteq$ *backflows* (*filternew-flows-state* (|*hosts* = *V*, *flows-fix* = *E*, *flows-state* = $\mathit{stateful} \cup \{x\}$|)) **by** *blast*

from *contr1 contr2*
show *False* **by** *simp*
qed

filter-compliant-stateful-ACS is correct and maximal

thm *filter-compliant-stateful-ACS-correct filter-compliant-stateful-ACS-maximal*

Getting those together. We cannot say *edgesList* = *E* here because one filters first. I guess filtering ACS first is easier, ...

definition *generate-valid-stateful-policy-IFSACS* :: *'v::vertex graph* ⇒ *'v SecurityInvariant-configured list* ⇒ (*'v* × *'v*) *list* ⇒ *'v stateful-policy* **where**
generate-valid-stateful-policy-IFSACS G M edgesList ≡ (*let filterIFS* = *filter-IFS-no-violations G M edgesList in*
(*let filterACS* = *filter-compliant-stateful-ACS G M filterIFS in* (| *hosts* = *nodes G*, *flows-fix* = *edges G*, *flows-state* = *set filterACS* |)))

lemma *generate-valid-stateful-policy-IFSACS-wf-stateful-policy*: **assumes** *wfG*: *wf-graph G*
and *edgesList*: (*set edgesList*) = *edges G*
shows *wf-stateful-policy* (*generate-valid-stateful-policy-IFSACS G M edgesList*)
proof −
from *wfG* **show** *?thesis*
apply(*simp add*: *generate-valid-stateful-policy-IFSACS-def wf-stateful-policy-def*)
apply(*auto simp add*: *wf-graph-def*)
using *edgesList filter-IFS-no-violations-subseteq-input filter-compliant-stateful-ACS-subseteq-input* **by** (*metis rev-subsetD*)
qed

lemma *generate-valid-stateful-policy-IFSACS-select-simps*:
shows *hosts* (*generate-valid-stateful-policy-IFSACS G M edgesList*) = *nodes G*
and *flows-fix* (*generate-valid-stateful-policy-IFSACS G M edgesList*) = *edges G*
and *flows-state* (*generate-valid-stateful-policy-IFSACS G M edgesList*) ⊆ *set edgesList*
proof −
show *hosts* (*generate-valid-stateful-policy-IFSACS G M edgesList*) = *nodes G*
by(*simp add*: *generate-valid-stateful-policy-IFSACS-def*)
show *flows-fix* (*generate-valid-stateful-policy-IFSACS G M edgesList*) = *edges G*
by(*simp add*: *generate-valid-stateful-policy-IFSACS-def*)
show *flows-state* (*generate-valid-stateful-policy-IFSACS G M edgesList*) ⊆ *set edgesList*
apply(*simp add*: *generate-valid-stateful-policy-IFSACS-def*)
using *filter-IFS-no-violations-subseteq-input filter-compliant-stateful-ACS-subseteq-input* **by** (*metis subset-trans*)
qed

lemma *generate-valid-stateful-policy-IFSACS-all-security-requirements-fulfilled-IFS*: **assumes** *validReqs*: *valid-reqs M*
and *wfG*: *wf-graph G*
and *high-level-policy-valid*: *all-security-requirements-fulfilled M G*
and *edgesList*: (*set edgesList*) ⊆ *edges G*
shows *all-security-requirements-fulfilled* (*get-IFS M*) (*stateful-policy-to-network-graph* (*generate-valid-stateful-polic*
G M edgesList))
proof −
have *simp3*: *flows-state* (*generate-valid-stateful-policy-IFSACS G M edgesList*) ⊆ *edges G* **using** *generate-valid-stateful-policy-IFSACS-select-simps*(*3*) *edgesList* **by** *fast*

have *set* (*filter-compliant-stateful-ACS G M* (*filter-IFS-no-violations G M edgesList*)) ⊆ *set* (*filter-IFS-no-violations G M edgesList*)
using *filter-compliant-stateful-ACS-subseteq-input edgesList* **by** (*metis*)
from *backflows-subseteq this* **have**
backflows (*set* (*filter-compliant-stateful-ACS G M* (*filter-IFS-no-violations G M edgesList*))) ⊆ *backflows* (*set* (*filter-IFS-no-violations G M edgesList*)) **by** *metis*
hence *subseteqhlp1*:
edges G ∪ *backflows* (*set* (*filter-compliant-stateful-ACS G M* (*filter-IFS-no-violations G M edgesList*))) ⊆ *edges G* ∪ *backflows* (*set* (*filter-IFS-no-violations G M edgesList*)) **by** *blast*

from *high-level-policy-valid* **have** *all-security-requirements-fulfilled* (*get-IFS M*) *G* **by**(*simp add*: *all-security-requirements-fulfilled-def get-IFS-def*)
from *filter-IFS-no-violations-correct*[*OF valid-reqs-IFS-D*[*OF validReqs*] *wfG this edgesList*] **have**
all-security-requirements-fulfilled (*get-IFS M*) (*stateful-policy-to-network-graph* (|*hosts* = *nodes G*, *flows-fix* = *edges G*, *flows-state* = *set* (*filter-IFS-no-violations G M edgesList*)|)) **.**
from *this edgesList* **have** *goalIFS*:
all-security-requirements-fulfilled (*get-IFS M*) (|*nodes* = *nodes G*, *edges* = *edges G* ∪ *backflows* (*set* (*filter-IFS-no-violations G M edgesList*))|)
apply(*simp add*: *stateful-policy-to-network-graph-def all-flows-def*)
by (*metis Un-absorb2 filter-IFS-no-violations-subseteq-input order-trans*)

from *wfG filter-IFS-no-violations-subseteq-input*[**where** *Es*=*edgesList* **and** *G*=*G* **and** *M*=*M*] *edgesList* **have**
wf-graph (|*nodes* = *nodes G*, *edges* = *set* (*filter-IFS-no-violations G M edgesList*)|)
apply(*case-tac G*, *simp*)
by (*metis le-iff-sup wf-graph-remove-edges-union*)
from *backflows-wf*[*OF this*] **have**
wf-graph (|*nodes* = *nodes G*, *edges* = *backflows* (*set* (*filter-IFS-no-violations G M edgesList*))|) **by**(*simp*)
from *this wf-graph-union-edges wfG* **have**
wf-graph (|*nodes* = *nodes G*, *edges* = *edges G* ∪ *backflows* (*set* (*filter-IFS-no-violations G M edgesList*))|)
by (*metis graph.cases graph.select-convs*(*1*) *graph.select-convs*(*2*))

from *all-security-requirements-fulfilled-mono*[*OF valid-reqs-IFS-D*[*OF validReqs*] *subseteqhlp1 this goalIFS*]
have *all-security-requirements-fulfilled* (*get-IFS M*) (|*nodes* = *nodes G*, *edges* = *edges G* ∪ *backflows* (*set* (*filter-compliant-stateful-ACS G M* (*filter-IFS-no-violations G M edgesList*)))|)
.
thus *?thesis*
apply(*simp add*: *stateful-policy-to-network-graph-def all-flows-def generate-valid-stateful-policy-IFSACS-select-simp*
simp3 Un-absorb2)
by(*simp add*: *generate-valid-stateful-policy-IFSACS-def*)
qed

theorem *generate-valid-stateful-policy-IFSACS-stateful-policy-compliance*:
assumes *validReqs*: *valid-reqs M*
and *wfG*: *wf-graph G*
and *high-level-policy-valid*: *all-security-requirements-fulfilled M G*
and *edgesList*: (*set edgesList*) = *edges G*
and *Tau*: $\mathcal{T}$ = *generate-valid-stateful-policy-IFSACS G M edgesList*
shows *stateful-policy-compliance* $\mathcal{T}$ *G M*
proof –

have *1*: *wf-stateful-policy* $\mathcal{T}$
apply(*simp add*: *Tau*)
by(*simp add*: *generate-valid-stateful-policy-IFSACS-wf-stateful-policy*[*OF wfG edgesList*])
have *2*: *wf-stateful-policy* (*generate-valid-stateful-policy-IFSACS G M edgesList*)
by(*simp add*: *generate-valid-stateful-policy-IFSACS-wf-stateful-policy*[*OF wfG edgesList*])
have *3*: *hosts* $\mathcal{T}$ = *nodes G*
apply(*simp add*: *Tau*)
by(*simp add*: *generate-valid-stateful-policy-IFSACS-select-simps*(*1*))
have *4*: *flows-fix* $\mathcal{T}$ ⊆ *edges G*
apply(*simp add*: *Tau*)
by(*simp add*: *generate-valid-stateful-policy-IFSACS-select-simps*(*2*))
have *5*: *all-security-requirements-fulfilled* (*get-IFS M*) (*stateful-policy-to-network-graph* $\mathcal{T}$)
apply(*simp add*: *Tau*)
using *generate-valid-stateful-policy-IFSACS-all-security-requirements-fulfilled-IFS*[*OF validReqs wfG high-level-policy-valid*] *edgesList* **by** *blast*
have *6*: ∀ *F*∈*get-offending-flows* (*get-ACS M*) (*stateful-policy-to-network-graph* $\mathcal{T}$). *F* ⊆ *backflows* (*filternew-flows-state* $\mathcal{T}$)
using *filter-compliant-stateful-ACS-correct*[*OF valid-reqs-ACS-D*[*OF validReqs*] *wfG - - Tau*[*simplified generate-valid-stateful-policy-IFSACS-def Let-def*]] *all-security-requirements-fulfilled-ACS-D*[*OF high-level-policy-valid*] *edgesList filter-IFS-no-violations-subseteq-input* **by** *metis*

from *1 2 3 4 5 6 validReqs high-level-policy-valid wfG*
show *?thesis*
unfolding *stateful-policy-compliance-def* **by** *simp*
qed

definition *generate-valid-stateful-policy-IFSACS-2* :: *'v*::*vertex graph* ⇒ *'v SecurityInvariant-configured list* ⇒ (*'v* × *'v*) *list* ⇒ *'v stateful-policy* **where**
generate-valid-stateful-policy-IFSACS-2 G M edgesList ≡
(| *hosts* = *nodes G*, *flows-fix* = *edges G*, *flows-state* = *set* (*filter-IFS-no-violations G M edgesList*) ∩ *set* (*filter-compliant-stateful-ACS G M edgesList*) |)

lemma *generate-valid-stateful-policy-IFSACS-2-wf-stateful-policy*: **assumes** *wfG*: *wf-graph G*
and *edgesList*: (*set edgesList*) = *edges G*
shows *wf-stateful-policy* (*generate-valid-stateful-policy-IFSACS-2 G M edgesList*)
proof –
from *wfG* **show** *?thesis*
apply(*simp add*: *generate-valid-stateful-policy-IFSACS-2-def wf-stateful-policy-def*)
apply(*auto simp add*: *wf-graph-def*)
using *edgesList filter-IFS-no-violations-subseteq-input* **by** (*metis rev-subsetD*)
qed

lemma *generate-valid-stateful-policy-IFSACS-2-select-simps*:
shows *hosts* (*generate-valid-stateful-policy-IFSACS-2 G M edgesList*) = *nodes G*
and *flows-fix* (*generate-valid-stateful-policy-IFSACS-2 G M edgesList*) = *edges G*
and *flows-state* (*generate-valid-stateful-policy-IFSACS-2 G M edgesList*) ⊆ *set edgesList*
proof –
show *hosts* (*generate-valid-stateful-policy-IFSACS-2 G M edgesList*) = *nodes G*
by(*simp add*: *generate-valid-stateful-policy-IFSACS-2-def*)

show *flows-fix* (*generate-valid-stateful-policy-IFSACS-2 G M edgesList*) = *edges G*
by(*simp add*: *generate-valid-stateful-policy-IFSACS-2-def*)
show *flows-state* (*generate-valid-stateful-policy-IFSACS-2 G M edgesList*) ⊆ *set edgesList*
apply(*simp add*: *generate-valid-stateful-policy-IFSACS-2-def*)
using *filter-compliant-stateful-ACS-subseteq-input* **by** (*metis inf.coboundedI2*)
qed

lemma *generate-valid-stateful-policy-IFSACS-2-all-security-requirements-fulfilled-IFS*: **assumes** *validReqs*: *valid-reqs M*
and *wfG*: *wf-graph G*
and *high-level-policy-valid*: *all-security-requirements-fulfilled M G*
and *edgesList*: (*set edgesList*) ⊆ *edges G*
shows *all-security-requirements-fulfilled* (*get-IFS M*) (*stateful-policy-to-network-graph* (*generate-valid-stateful-polic G M edgesList*))
proof −
have *subseteq*: *set* (*filter-IFS-no-violations G M edgesList*) ∩ *set* (*filter-compliant-stateful-ACS G M edgesList*) ⊆ *set* (*filter-IFS-no-violations G M edgesList*) **by** *blast*

from *wfG filter-IFS-no-violations-subseteq-input edgesList*
have *wfG′*: *wf-graph* (|*nodes* = *nodes G*, *edges* = *edges G* ∪ *set* (*filter-IFS-no-violations G M edgesList*)|)
by (*metis graph-eq-intro Un-absorb2 graph.select-convs*(*1*) *graph.select-convs*(*2*) *order.trans*)

from *high-level-policy-valid* **have** *all-security-requirements-fulfilled* (*get-IFS M*) *G* **by**(*simp add*: *all-security-requirements-fulfilled-def get-IFS-def*)
from *filter-IFS-no-violations-correct*[*OF valid-reqs-IFS-D*[*OF validReqs*] *wfG this edgesList*] **have**
all-security-requirements-fulfilled (*get-IFS M*) (*stateful-policy-to-network-graph* (|*hosts* = *nodes G*, *flows-fix* = *edges G*, *flows-state* = *set* (*filter-IFS-no-violations G M edgesList*)|)) **.**

from *all-security-requirements-fulfilled-mono-stateful-policy-to-network-graph*[*OF valid-reqs-IFS-D*[*OF validReqs*] *subseteq wfG′ this*]
have *all-security-requirements-fulfilled* (*get-IFS M*) (*stateful-policy-to-network-graph* (*generate-valid-stateful-policy-IF G M edgesList*))
by(*simp add*: *generate-valid-stateful-policy-IFSACS-2-def*)
thus *?thesis* **.**
qed

lemma *generate-valid-stateful-policy-IFSACS-2-filter-compliant-stateful-ACS*:
assumes *validReqs*: *valid-reqs M*
and *wfG*: *wf-graph G*
and *high-level-policy-valid*: *all-security-requirements-fulfilled M G*
and *edgesList*: (*set edgesList*) ⊆ *edges G*
and *Tau*: $\mathcal{T}$ = *generate-valid-stateful-policy-IFSACS-2 G M edgesList*
shows ∀ *F*∈*get-offending-flows* (*get-ACS M*) (*stateful-policy-to-network-graph* $\mathcal{T}$). *F* ⊆ *backflows* (*filternew-flows-state* $\mathcal{T}$)
proof−
let *?filterACS* = *set* (*filter-compliant-stateful-ACS G M edgesList*)
let *?filterIFS* = *set* (*filter-IFS-no-violations G M edgesList*)
from *all-security-requirements-fulfilled-ACS-D*[*OF high-level-policy-valid*] **have** *all-security-requirements-fulfilled* (*get-ACS M*) *G* **.**

from *filter-compliant-stateful-ACS-correct*[*OF valid-reqs-ACS-D*[*OF validReqs*] *wfG edgesList this*]

have
∀ *F*∈*get-offending-flows* (*get-ACS M*) (*stateful-policy-to-network-graph* (|*hosts* = *nodes G*, *flows-fix* = *edges G*, *flows-state* = *?filterACS*|)).
F ⊆ *backflows* (*?filterACS*) − *edges G*
apply(*simp*)
apply(*simp add*: *backflows-minus-backflows*[*symmetric*])
by(*simp add*: *filternew-flows-state-alt*)
hence ∀ *F*∈*get-offending-flows* (*get-ACS M*) (|*nodes* = *nodes G*, *edges* = *edges G* ∪ *backflows* (*?filterACS*)|). *F* ⊆ *backflows* (*?filterACS*) − *edges G*
apply(*simp add*: *stateful-policy-to-network-graph-def all-flows-def*)
using *filter-compliant-stateful-ACS-subseteq-input* **by** (*metis* (*lifting*, *no-types*) *Un-absorb2 edgesList order-trans*)
from *this validReqs* **have** *offending-filterACS-upperbound*:
⋀*m*. *m* ∈ *set* (*get-ACS M*) ⟹
⋃(*c-offending-flows m* (|*nodes* = *nodes G*, *edges* = *edges G* ∪ *backflows* (*?filterACS*)|)) ⊆
backflows (*?filterACS*) − *edges G*
by(*simp add*: *valid-reqs-def get-offending-flows-def*, *blast*)

from *wfG filter-compliant-stateful-ACS-subseteq-input edgesList* **have** *wf-graph* (|*nodes* = *nodes G*, *edges* = *?filterACS*|)
by (*metis graph.cases graph.select-convs*(*1*) *graph.select-convs*(*2*) *le-iff-sup wf-graph-remove-edges-union*)
from *this backflows-wf* **have** *wf-graph* (|*nodes* = *nodes G*, *edges* = *backflows* (*?filterACS*)|) **by** *blast*
moreover have *wf-graph* (|*nodes* = *nodes G*, *edges* = *edges G*|) **using** *wfG* **by**(*case-tac G*, *simp*)
ultimately have *wfG1*: *wf-graph* (|*nodes* = *nodes G*, *edges* = *edges G* ∪ *backflows* (*?filterACS*)|)
using *wf-graph-union-edges* **by** *blast*

from *edgesList* **have** *edgesUnsimp*: *edges G* ∪ (*?filterACS* ∩ *?filterIFS*) = *edges G*
using *filter-IFS-no-violations-subseteq-input filter-compliant-stateful-ACS-subseteq-input* **by** *blast*

— We set up a ?REM that we use in the ⟦*configured-SecurityInvariant ?m*; *wf-graph* (|*nodes* = *?V*, *edges* = *?E*|); ⋃ (*c-offending-flows ?m* (|*nodes* = *?V*, *edges* = *?E*|)) ⊆ *?X*⟧ ⟹ ⋃ (*c-offending-flows ?m* (|*nodes* = *?V*, *edges* = *?E* − *?E′*|)) ⊆ *?X* − *?E′* lemma
let *?REM* = (*backflows* (*?filterACS*) − *backflows* (*?filterIFS*)) − *edges G*

have *REM-gives-desired-upper-bound*: (*backflows* (*?filterACS*) − *edges G*) − *?REM* = *backflows* (*?filterACS* ∩ *?filterIFS*) − *edges G*
by(*simp add*: *backflows-def*, *blast*)

have *REM-gives-desired-edges*: (*edges G* ∪ *backflows* (*?filterACS*)) − *?REM* = *edges G* ∪ (*backflows* (*?filterACS* ∩ *?filterIFS*))
by(*simp add*: *backflows-def*, *blast*)

from *wfG* **have** *finite* (*edges G*) **using** *wf-graph-def* **by** *blast*
hence *finite* (*backflows ?filterACS*) **using** *backflows-finite* **by** (*metis List.finite-set*)
hence *finite1*: *finite* (*backflows* (*?filterACS*) − *backflows* (*?filterIFS*) − *edges G*) **by** *fast*

from *configured-SecurityInvariant.Un-set-offending-flows-bound-minus-subseteq*[**where** *E′*=*?REM* **and** *X*=(*backflows* (*?filterACS*) − *edges G*),
OF - wfG1 offending-filterACS-upperbound, *simplified REM-gives-desired-upper-bound REM-gives-desired-edges*] *valid-reqs-ACS-D*[*OF validReqs*, *unfolded valid-reqs-def*]
have ⋀*m*. *m* ∈ *set* (*get-ACS M*) ⟹
∀ *F*∈*c-offending-flows m* (|*nodes* = *nodes G*, *edges* = *edges G* ∪ *backflows* (*?filterACS* ∩ *?filterIFS*)|).

F ⊆ *backflows* (*?filterACS* ∩ *?filterIFS*) − *edges G* **by** *blast*
hence ∀ *F*∈*get-offending-flows* (*get-ACS M*)
(|*nodes* = *nodes G*, *edges* = *edges G* ∪ (*backflows* (*?filterACS* ∩ *?filterIFS*))|). *F* ⊆ *backflows* (*?filterACS* ∩ *?filterIFS*) − *edges G*
using *get-offending-flows-def* **by** *fast*
hence ∀ *F*∈*get-offending-flows* (*get-ACS M*)
(|*nodes* = *nodes G*, *edges* = *edges G* ∪ (*?filterACS* ∩ *?filterIFS*) ∪ (*backflows* (*?filterACS* ∩ *?filterIFS*))|).
F ⊆ *backflows* (*?filterACS* ∩ *?filterIFS*) − *edges G*
by(*simp add*: *edgesUnsimp*)
hence ∀ *F*∈*get-offending-flows* (*get-ACS M*) (*stateful-policy-to-network-graph* (|*hosts* = *nodes G*, *flows-fix* = *edges G*, *flows-state* = *?filterACS* ∩ *?filterIFS*|)).
F ⊆ *backflows* (*?filterACS* ∩ *?filterIFS*) − *edges G*
by(*simp add*: *stateful-policy-to-network-graph-def all-flows-def*)

thus *?thesis*
apply(*simp add*: *Tau generate-valid-stateful-policy-IFSACS-2-def*)
apply(*simp add*: *filternew-flows-state-alt backflows-minus-backflows*)
by (*metis inf-commute*)
qed

theorem *generate-valid-stateful-policy-IFSACS-2-stateful-policy-compliance*:
assumes *validReqs*: *valid-reqs M*
and *wfG*: *wf-graph G*
and *high-level-policy-valid*: *all-security-requirements-fulfilled M G*
and *edgesList*: (*set edgesList*) = *edges G*
and *Tau*: 𝒯 = *generate-valid-stateful-policy-IFSACS-2 G M edgesList*
shows *stateful-policy-compliance* 𝒯 *G M*
proof −
have *1*: *wf-stateful-policy* 𝒯
apply(*simp add*: *Tau*)
by(*simp add*: *generate-valid-stateful-policy-IFSACS-2-wf-stateful-policy*[*OF wfG edgesList*])
have *2*: *wf-stateful-policy* (*generate-valid-stateful-policy-IFSACS G M edgesList*)
by(*simp add*: *generate-valid-stateful-policy-IFSACS-wf-stateful-policy*[*OF wfG edgesList*])
have *3*: *hosts* 𝒯 = *nodes G*
apply(*simp add*: *Tau*)
by(*simp add*: *generate-valid-stateful-policy-IFSACS-2-select-simps*(*1*))
have *4*: *flows-fix* 𝒯 ⊆ *edges G*
apply(*simp add*: *Tau*)
by(*simp add*: *generate-valid-stateful-policy-IFSACS-2-select-simps*(*2*))
have *5*: *all-security-requirements-fulfilled* (*get-IFS M*) (*stateful-policy-to-network-graph* 𝒯)
apply(*simp add*: *Tau*)
using *generate-valid-stateful-policy-IFSACS-2-all-security-requirements-fulfilled-IFS*[*OF validReqs wfG high-level-policy-valid*] *edgesList* **by** *blast*
have *6*: ∀ *F*∈*get-offending-flows* (*get-ACS M*) (*stateful-policy-to-network-graph* 𝒯). *F* ⊆ *backflows* (*filternew-flows-state* 𝒯)
using *generate-valid-stateful-policy-IFSACS-2-filter-compliant-stateful-ACS*[*OF validReqs wfG high-level-policy-valid*]
Tau edgesList **by** *auto*

from *1 2 3 4 5 6 validReqs high-level-policy-valid wfG*
show *?thesis*
unfolding *stateful-policy-compliance-def* **by** *simp*
qed

If there are no IFS requirements and the ACS requirements cause no side effects, effectively, the graph can be considered as undirected graph!

lemma *generate-valid-stateful-policy-IFSACS-2-noIFS-noACSsideeffects-imp-fullgraph*:
assumes *validReqs*: *valid-reqs M*
and *wfG*: *wf-graph G*
and *high-level-policy-valid*: *all-security-requirements-fulfilled M G*
and *edgesList*: (*set edgesList*) = *edges G*
and *no-ACS-sideeffects*: ∀ *F* ∈ *get-offending-flows* (*get-ACS M*) (|*nodes* = *nodes G*, *edges* = *edges G* ∪ *backflows* (*edges G*)|). *F* ⊆ (*backflows* (*edges G*)) − (*edges G*)
and *no-IFS*: *get-IFS M* = []
shows *stateful-policy-to-network-graph* (*generate-valid-stateful-policy-IFSACS-2 G M edgesList*) = *undirected G*
proof −
from *filter-IFS-no-violations-accu-no-IFS*[*OF valid-reqs-IFS-D*[*OF validReqs*] *wfG no-IFS*] *edgesList*
have *filter-IFS-no-violations G M edgesList* = *rev edgesList*
by(*simp add*: *filter-IFS-no-violations-def*)
from *this filter-compliant-stateful-ACS-subseteq-input* **have** *flows-state-IFS*: *flows-state* (*generate-valid-stateful-policy-G M edgesList*) = *set* (*filter-compliant-stateful-ACS G M edgesList*)
by(*auto simp add*: *generate-valid-stateful-policy-IFSACS-2-def*)

have *flowsfix*: *flows-fix* (*generate-valid-stateful-policy-IFSACS-2 G M edgesList*) = *edges G* **by**(*simp add*: *generate-valid-stateful-policy-IFSACS-2-def*)

have *hosts*: *hosts* (*generate-valid-stateful-policy-IFSACS-2 G M edgesList*) = *nodes G* **by**(*simp add*: *generate-valid-stateful-policy-IFSACS-2-def*)

from *filter-compliant-stateful-ACS-accu-no-side-effects*[*OF valid-reqs-ACS-D*[*OF validReqs*] *wfG no-ACS-sideeffects*] **have**
filter-compliant-stateful-ACS G M edgesList = *rev* [*e*←*edgesList* . *e* ∉ *backflows* (*edges G*)]
by(*simp add*: *filter-compliant-stateful-ACS-def edgesList*)
hence *filterACS*: *set* (*filter-compliant-stateful-ACS G M edgesList*) = *edges G* − (*backflows* (*edges G*)) **using** *edgesList* **by** *force*

show *?thesis*
apply(*simp add*: *undirected-backflows stateful-policy-to-network-graph-def all-flows-def*)
apply(*simp add*: *hosts filterACS flows-state-IFS flowsfix*)
apply(*simp add*: *backflows-minus-backflows*)
by *fast*
qed

lemma *generate-valid-stateful-policy-IFSACS-noIFS-noACSsideeffects-imp-fullgraph*:
assumes *validReqs*: *valid-reqs M*
and *wfG*: *wf-graph G*
and *high-level-policy-valid*: *all-security-requirements-fulfilled M G*
and *edgesList*: (*set edgesList*) = *edges G*
and *no-ACS-sideeffects*: ∀ *F* ∈ *get-offending-flows* (*get-ACS M*) (|*nodes* = *nodes G*, *edges* = *edges G* ∪ *backflows* (*edges G*)|). *F* ⊆ (*backflows* (*edges G*)) − (*edges G*)
and *no-IFS*: *get-IFS M* = []
shows *stateful-policy-to-network-graph* (*generate-valid-stateful-policy-IFSACS G M edgesList*) =

undirected G
proof −
from *filter-IFS-no-violations-accu-no-IFS*[*OF valid-reqs-IFS-D*[*OF validReqs*] *wfG no-IFS*] *edgesList*
have *filter-IFS-no-violations G M edgesList* = *rev edgesList*
by(*simp add*: *filter-IFS-no-violations-def*)
from *this filter-compliant-stateful-ACS-subseteq-input* **have** *flows-state-IFS*: *flows-state* (*generate-valid-stateful-policy-IFSACS G M edgesList*) = *set* (*filter-compliant-stateful-ACS G M* (*rev edgesList*))
by(*simp add*: *generate-valid-stateful-policy-IFSACS-def*)

have *flowsfix*: *flows-fix* (*generate-valid-stateful-policy-IFSACS G M edgesList*) = *edges G* **by**(*simp add*: *generate-valid-stateful-policy-IFSACS-def*)

have *hosts*: *hosts* (*generate-valid-stateful-policy-IFSACS G M edgesList*) = *nodes G* **by**(*simp add*: *generate-valid-stateful-policy-IFSACS-def*)

from *filter-compliant-stateful-ACS-accu-no-side-effects*[*OF valid-reqs-ACS-D*[*OF validReqs*] *wfG no-ACS-sideeffects*] **have**
filter-compliant-stateful-ACS G M (*rev edgesList*) = [*e*←*edgesList* . *e* ∉ *backflows* (*edges G*)]
apply(*simp add*: *filter-compliant-stateful-ACS-def edgesList*) **by** (*metis rev-filter rev-swap*)
hence *filterACS*: *set* (*filter-compliant-stateful-ACS G M* (*rev edgesList*)) = *edges G* − (*backflows* (*edges G*)) **using** *edgesList* **by** *force*

show *?thesis*
apply(*simp add*: *undirected-backflows stateful-policy-to-network-graph-def all-flows-def*)
apply(*simp add*: *hosts filterACS flows-state-IFS flowsfix*)
apply(*simp add*: *backflows-minus-backflows*)
by *fast*
qed

end
theory *TopoS-Stateful-Policy-impl*
imports *TopoS-Composition-Theory-impl TopoS-Stateful-Policy-Algorithm*
begin

11 Stateful Policy – List Implementaion

record *′v stateful-list-policy* =
hostsL :: *′v list*
flows-fixL :: (*′v* ×*′v*) *list*
flows-stateL :: (*′v* ×*′v*) *list*

definition *stateful-list-policy-to-list-graph* :: *′v stateful-list-policy* ⇒ *′v list-graph* **where**
stateful-list-policy-to-list-graph $\mathcal{T}$ = (| *nodesL* = *hostsL* $\mathcal{T}$, *edgesL* = (*flows-fixL* $\mathcal{T}$) @ [*e* ← *flows-stateL* $\mathcal{T}$. *e* ∉ *set* (*flows-fixL* $\mathcal{T}$)] @ [*e* ← *backlinks* (*flows-stateL* $\mathcal{T}$). *e* ∉ *set* (*flows-fixL* $\mathcal{T}$)] |)

lemma *stateful-list-policy-to-list-graph-complies*:
list-graph-to-graph (*stateful-list-policy-to-list-graph* (| *hostsL* = *V*, *flows-fixL* = E_f, *flows-stateL* = E_σ |)) =
stateful-policy-to-network-graph (| *hosts* = *set V*, *flows-fix* = *set* E_f, *flows-state* = *set* E_σ |)

by(*simp add*: *stateful-list-policy-to-list-graph-def stateful-policy-to-network-graph-def all-flows-def list-graph-to-graph-def backlinks-correct*, *blast*)

lemma *wf-list-graph-stateful-list-policy-to-list-graph*:
wf-list-graph G ⟹ *distinct E* ⟹ *set E* ⊆ *set* (*edgesL G*) ⟹ *wf-list-graph* (*stateful-list-policy-to-list-graph* (|*hostsL* = *nodesL G*, *flows-fixL* = *edgesL G*, *flows-stateL* = *E*|))
apply(*simp add*: *wf-list-graph-def stateful-list-policy-to-list-graph-def*)
apply(*rule conjI*)
apply(*simp add*: *backlinks-distinct*)
apply(*rule conjI*)
apply(*simp add*: *backlinks-set*)
apply(*blast*)
apply(*rule conjI*)
apply(*simp add*: *backlinks-set*)
apply(*blast*)
apply(*simp add*: *wf-list-graph-axioms-def*)
apply(*rule conjI*)
apply(*simp add*: *backlinks-set*)
apply(*force*)
apply(*simp add*: *backlinks-set*)
apply(*clarsimp*)
apply(*erule disjE*)
apply(*auto*)[*1*]
apply(*erule disjE*)
apply(*auto*)[*1*]
by *force*

11.1 Algorithms

fun *filter-IFS-no-violations-accu* :: *'v list-graph* ⇒ *'v SecurityInvariant list* ⇒ (*'v* × *'v*) *list* ⇒ (*'v* × *'v*) *list* ⇒ (*'v* × *'v*) *list* **where**
filter-IFS-no-violations-accu G M accu [] = *accu* |
filter-IFS-no-violations-accu G M accu (*e#Es*) = (*if all-security-requirements-fulfilled* (*TopoS-Composition-Theory-impl.get-IFS M*) (*stateful-list-policy-to-list-graph* (| *hostsL* = *nodesL G*, *flows-fixL* = *edgesL G*, *flows-stateL* = (*e#accu*) |))
then filter-IFS-no-violations-accu G M (*e#accu*) *Es*
else filter-IFS-no-violations-accu G M accu Es)
definition *filter-IFS-no-violations* :: *'v list-graph* ⇒ *'v SecurityInvariant list* ⇒ (*'v* × *'v*) *list* **where**
filter-IFS-no-violations G M = *filter-IFS-no-violations-accu G M* [] (*edgesL G*)

lemma *filter-IFS-no-violations-accu-distinct*: ⟦ *distinct* (*Es@accu*) ⟧ ⟹ *distinct* (*filter-IFS-no-violations-accu G M accu Es*)
apply(*induction Es arbitrary*: *accu*)
by(*simp-all*)

lemma *filter-IFS-no-violations-accu-complies*:
⟦∀ (*m-impl*, *m-spec*) ∈ *set M*. *SecurityInvariant-complies-formal-def m-impl m-spec*;
wf-list-graph G; *set Es* ⊆ *set* (*edgesL G*); *set accu* ⊆ *set* (*edgesL G*); *distinct* (*Es@accu*) ⟧ ⟹
filter-IFS-no-violations-accu G (*get-impl M*) *accu Es* = *TopoS-Stateful-Policy-Algorithm.filter-IFS-no-violations-accu* (*list-graph-to-graph G*) (*get-spec M*) *accu Es*
proof(*induction Es arbitrary*: *accu*)
case *Nil*
thus *?case* **by**(*simp add*: *get-impl-def get-spec-def*)
next

case (*Cons e Es*)
— ⟦*set Es* ⊆ *set* (*edgesL G*); *set ?accu* ⊆ *set* (*edgesL G*); *distinct* (*Es* @ *?accu*)⟧ ⟹ *TopoS-Stateful-Policy-impl.filter-IFS-no-violations-accu G* (*get-impl M*) *?accu Es* = *TopoS-Stateful-Policy-Algorithm.filt* (*list-graph-to-graph G*) (*get-spec M*) *?accu Es*
let *?caseDistinction* = *all-security-requirements-fulfilled* (*TopoS-Composition-Theory-impl.get-IFS* (*get-impl M*)) (*stateful-list-policy-to-list-graph* (| *hostsL* = *nodesL G*, *flows-fixL* = *edgesL G*, *flows-stateL* = (*e#accu*) |))

from *get-IFS-get-ACS-select-simps*(*2*)[*OF Cons.prems*(*1*)] **have** *get-impl-zip-simp*: (*get-impl* (*zip* (*TopoS-Composition-Theory-impl.get-IFS* (*get-impl M*)) (*TopoS-Composition-Theory.get-IFS* (*get-spec M*)))) = *TopoS-Composition-Theory-impl.get-IFS* (*get-impl M*) **by** *simp*

from *get-IFS-get-ACS-select-simps*(*3*)[*OF Cons.prems*(*1*)] **have** *get-spec-zip-simp*: (*get-spec* (*zip* (*TopoS-Composition-Theory-impl.get-IFS* (*get-impl M*)) (*TopoS-Composition-Theory.get-IFS* (*get-spec M*)))) = *TopoS-Composition-Theory.get-IFS* (*get-spec M*) **by** *simp*

from *Cons.prems*(*3*) *Cons.prems*(*4*) **have** *set* (*e* # *accu*) ⊆ *set* (*edgesL G*) **by** *simp*
from *Cons.prems*(*4*) **have** *set* (*accu*) ⊆ *set* (*edgesL G*) **by** *simp*
from *Cons.prems*(*5*) **have** *distinct* (*e* # *accu*) **by** *simp*
from *Cons.prems*(*3*) **have** *set Es* ⊆ *set* (*edgesL G*) **by** *simp*
from *Cons.prems*(*5*) **have** *distinct* (*Es* @ *accu*) **by** *simp*
from *Cons.prems*(*5*) **have** *distinct* (*Es* @ (*e* # *accu*)) **by** *simp*

from *Cons.prems*(*2*) **have** *validLG*: *wf-list-graph* (*stateful-list-policy-to-list-graph* (|*hostsL* = *nodesL G*, *flows-fixL* = *edgesL G*, *flows-stateL* = *e* # *accu*|))
apply(*rule wf-list-graph-stateful-list-policy-to-list-graph*)
apply(*fact* ‹*distinct* (*e* # *accu*)›)
apply(*fact* ‹*set* (*e* # *accu*) ⊆ *set* (*edgesL G*)›)
done

from *get-IFS-get-ACS-select-simps*(*1*)[*OF Cons.prems*(*1*)]
have ∀ (*m-impl*, *m-spec*) ∈ *set* (*zip* (*get-IFS* (*get-impl M*)) (*TopoS-Composition-Theory.get-IFS* (*get-spec M*))). *SecurityInvariant-complies-formal-def m-impl m-spec* **.**
from *all-security-requirements-fulfilled-complies*[*OF this*] **have** *all-security-requirements-fulfilled-eq-rule*:

⋀*G*. *wf-list-graph G* ⟹
TopoS-Composition-Theory-impl.all-security-requirements-fulfilled (*TopoS-Composition-Theory-impl.get-IFS* (*get-impl M*)) *G* =
TopoS-Composition-Theory.all-security-requirements-fulfilled (*TopoS-Composition-Theory.get-IFS* (*get-spec M*)) (*list-graph-to-graph G*)
by(*simp add*: *get-impl-zip-simp get-spec-zip-simp*)

have *case-impl-spec*: *?caseDistinction* ⟷ *TopoS-Composition-Theory.all-security-requirements-fulfilled* (*TopoS-Composition-Theory.get-IFS* (*get-spec M*)) (*stateful-policy-to-network-graph* (| *hosts* = *set* (*nodesL G*), *flows-fix* = *set* (*edgesL G*), *flows-state* = *set* (*e#accu*) |))
apply(*subst all-security-requirements-fulfilled-eq-rule*[*OF validLG*])
by(*simp add*: *stateful-list-policy-to-list-graph-complies*)

show *?case*
proof(*case-tac ?caseDistinction*)
assume *cTrue*: *?caseDistinction*

from *cTrue* **have** *g1*: *TopoS-Stateful-Policy-impl.filter-IFS-no-violations-accu G* (*get-impl M*) *accu* (*e* # *Es*) = *TopoS-Stateful-Policy-impl.filter-IFS-no-violations-accu G* (*get-impl M*) (*e* # *accu*) *Es* **by** *simp*

from *cTrue*[*simplified case-impl-spec*] **have** *g2*: *TopoS-Stateful-Policy-Algorithm.filter-IFS-no-violations-accu* (*list-graph-to-graph G*) (*get-spec M*) *accu* (*e* # *Es*) =
TopoS-Stateful-Policy-Algorithm.filter-IFS-no-violations-accu (*list-graph-to-graph G*) (*get-spec M*) (*e*#*accu*)*Es*
by(*simp add*: *list-graph-to-graph-def*)

show *?case*
apply(*simp only*: *g1 g2*)
using *Cons.IH*[*OF Cons.prems*(*1*) *Cons.prems*(*2*) ‹*set Es* ⊆ *set* (*edgesL G*)› ‹*set* (*e* # *accu*) ⊆ *set* (*edgesL G*)› ‹*distinct* (*Es* @ (*e* # *accu*))›] **by** *simp*
next
assume *cFalse*: ¬ *?caseDistinction*

from *cFalse* **have** *g1*: *TopoS-Stateful-Policy-impl.filter-IFS-no-violations-accu G* (*get-impl M*) *accu* (*e* # *Es*) = *TopoS-Stateful-Policy-impl.filter-IFS-no-violations-accu G* (*get-impl M*) *accu Es* **by** *simp*

from *cFalse*[*simplified case-impl-spec*] **have** *g2*: *TopoS-Stateful-Policy-Algorithm.filter-IFS-no-violations-accu* (*list-graph-to-graph G*) (*get-spec M*) *accu* (*e* # *Es*) =
TopoS-Stateful-Policy-Algorithm.filter-IFS-no-violations-accu (*list-graph-to-graph G*) (*get-spec M*) *accu Es*
by(*simp add*: *list-graph-to-graph-def*)

show *?case*
apply(*simp only*: *g1 g2*)
using *Cons.IH*[*OF Cons.prems*(*1*) *Cons.prems*(*2*) ‹*set Es* ⊆ *set* (*edgesL G*)› ‹*set accu* ⊆ *set* (*edgesL G*)› ‹*distinct* (*Es* @ *accu*)›] **by** *simp*
qed
qed

lemma *filter-IFS-no-violations-complies*:
⟦ ∀ (*m-impl*, *m-spec*) ∈ *set M*. *SecurityInvariant-complies-formal-def m-impl m-spec*; *wf-list-graph G* ⟧ ⟹
filter-IFS-no-violations G (*get-impl M*) = *TopoS-Stateful-Policy-Algorithm.filter-IFS-no-violations* (*list-graph-to-graph G*) (*get-spec M*) (*edgesL G*)
apply(*unfold filter-IFS-no-violations-def TopoS-Stateful-Policy-Algorithm.filter-IFS-no-violations-def*)

apply(*rule filter-IFS-no-violations-accu-complies*)
apply(*simp-all*)
apply(*simp add*: *wf-list-graph-def*)
done

fun *filter-compliant-stateful-ACS-accu* :: *'v list-graph* ⇒ *'v SecurityInvariant list* ⇒ (*'v* × *'v*) *list* ⇒ (*'v* × *'v*) *list* ⇒ (*'v* × *'v*) *list* **where**
filter-compliant-stateful-ACS-accu G M accu [] = *accu* |

filter-compliant-stateful-ACS-accu G M accu (*e#Es*) = (*if*
e ∉ *set* (*backlinks* (*edgesL G*)) ∧ (∀ *F* ∈ *set* (*implc-get-offending-flows* (*get-ACS M*) (*stateful-list-policy-to-list-graph* (| *hostsL* = *nodesL G*, *flows-fixL* = *edgesL G*, *flows-stateL* = (*e#accu*) |))). *set F* ⊆ *set* (*backlinks* (*e#accu*)))
then filter-compliant-stateful-ACS-accu G M (*e#accu*) *Es*
else filter-compliant-stateful-ACS-accu G M accu Es)

definition *filter-compliant-stateful-ACS* :: *'v list-graph* ⇒ *'v SecurityInvariant list* ⇒ (*'v* × *'v*) *list*
where
filter-compliant-stateful-ACS G M = *filter-compliant-stateful-ACS-accu G M* [] (*edgesL G*)

lemma *filter-compliant-stateful-ACS-accu-complies*:
⟦∀ (*m-impl*, *m-spec*) ∈ *set M*. *SecurityInvariant-complies-formal-def m-impl m-spec*;
wf-list-graph G; *set Es* ⊆ *set* (*edgesL G*); *set accu* ⊆ *set* (*edgesL G*); *distinct* (*Es@accu*) ⟧ ⟹
filter-compliant-stateful-ACS-accu G (*get-impl M*) *accu Es* = *TopoS-Stateful-Policy-Algorithm.filter-compliant-statefu*
(*list-graph-to-graph G*) (*get-spec M*) *accu Es*
proof(*induction Es arbitrary*: *accu*)
case *Nil*
thus *?case* **by**(*simp add*: *get-impl-def get-spec-def*)
next
case (*Cons e Es*)
— ⟦*set Es* ⊆ *set* (*edgesL G*); *set ?accu* ⊆ *set* (*edgesL G*); *distinct* (*Es* @ *?accu*)⟧ ⟹
TopoS-Stateful-Policy-impl.filter-compliant-stateful-ACS-accu G (*get-impl M*) *?accu Es* = *TopoS-Stateful-Policy-Algorith*
(*list-graph-to-graph G*) (*get-spec M*) *?accu Es*
let *?caseDistinction* = *e* ∉ *set* (*backlinks* (*edgesL G*)) ∧ (∀ *F* ∈ *set* (*implc-get-offending-flows* (*get-ACS* (*get-impl M*)) (*stateful-list-policy-to-list-graph* (| *hostsL* = *nodesL G*, *flows-fixL* = *edgesL G*, *flows-stateL* = (*e#accu*) |))). *set F* ⊆ *set* (*backlinks* (*e#accu*)))

have *backlinks-simp*: (*e* ∉ *set* (*backlinks* (*edgesL G*))) ⟷ (*e* ∉ *backflows* (*set* (*edgesL G*)))
by(*simp add*: *backlinks-correct*)

have ⋀ *G X*. (∀ *F*∈*set* (*implc-get-offending-flows* (*TopoS-Composition-Theory-impl.get-ACS* (*get-impl M*)) *G*). *set F* ⊆ *X*) =
(∀ *F*∈*set* ' *set* (*implc-get-offending-flows* (*TopoS-Composition-Theory-impl.get-ACS* (*get-impl M*)) *G*). *F* ⊆ *X*) **by** *blast*
also have ⋀ *G X*. *wf-list-graph G* ⟹ (∀ *F*∈*set* ' *set* (*implc-get-offending-flows* (*TopoS-Composition-Theory-impl.ge*
(*get-impl M*)) *G*). *F* ⊆ *X*) =
(∀ *F*∈*get-offending-flows* (*TopoS-Composition-Theory.get-ACS* (*get-spec M*)) (*list-graph-to-graph G*). *F* ⊆ *X*)
using *implc-get-offending-flows-complies*[*OF get-IFS-get-ACS-select-simps*(*4*)[*OF Cons.prems*(*1*)], *simplified get-IFS-get-ACS-select-simps*[*OF Cons.prems*(*1*)]] **by** *simp*
finally have *implc-get-offending-flows-simp-rule*: ⋀*G X*. *wf-list-graph G* ⟹
(∀ *F*∈*set* (*implc-get-offending-flows* (*TopoS-Composition-Theory-impl.get-ACS* (*get-impl M*)) *G*). *set F* ⊆ *X*) = (∀ *F*∈*get-offending-flows* (*TopoS-Composition-Theory.get-ACS* (*get-spec M*)) (*list-graph-to-graph G*). *F* ⊆ *X*) **.**

from *Cons.prems*(*3*) *Cons.prems*(*4*) **have** *set* (*e* # *accu*) ⊆ *set* (*edgesL G*) **by** *simp*
from *Cons.prems*(*4*) **have** *set* (*accu*) ⊆ *set* (*edgesL G*) **by** *simp*
from *Cons.prems*(*5*) **have** *distinct* (*e* # *accu*) **by** *simp*
from *Cons.prems*(*3*) **have** *set Es* ⊆ *set* (*edgesL G*) **by** *simp*
from *Cons.prems*(*5*) **have** *distinct* (*Es* @ *accu*) **by** *simp*
from *Cons.prems*(*5*) **have** *distinct* (*Es* @ (*e* # *accu*)) **by** *simp*
from *Cons.prems*(*2*) **have** *validLG*: *wf-list-graph* (*stateful-list-policy-to-list-graph* (|*hostsL* =

nodesL G, *flows-fixL* = *edgesL G*, *flows-stateL* = *e* # *accu*⦈)
apply(*rule wf-list-graph-stateful-list-policy-to-list-graph*)
apply(*fact* ‹*distinct* (*e* # *accu*)›)
apply(*fact* ‹*set* (*e* # *accu*) ⊆ *set* (*edgesL G*)›)
done

have *set* (*backlinks* (*e* # *accu*)) = *backflows* (*insert e* (*set accu*))
by(*simp add*: *backlinks-set backflows-def*)

— (∀ *F*∈*set* (*implc-get-offending-flows* (*TopoS-Composition-Theory-impl.get-ACS* (*get-impl M*)) (*stateful-list-policy-to-list-graph* (|*hostsL* = *nodesL G*, *flows-fixL* = *edgesL G*, *flows-stateL* = *e* # *accu*|))). *set F* ⊆ *?X*) = (∀ *F*∈*get-offending-flows* (*TopoS-Composition-Theory.get-ACS* (*get-spec M*)) (*list-graph-to-graph* (*stateful-list-policy-to-list-graph* (|*hostsL* = *nodesL G*, *flows-fixL* = *edgesL G*, *flows-stateL* = *e* # *accu*|))). *F* ⊆ *?X*)
have *case-impl-spec*: *?caseDistinction* ⟷ (
e ∉ *backflows* (*set* (*edgesL G*)) ∧ (∀ *F* ∈ *get-offending-flows* (*TopoS-Composition-Theory.get-ACS* (*get-spec M*)) (*stateful-policy-to-network-graph* (| *hosts* = *set* (*nodesL G*), *flows-fix* = *set* (*edgesL G*), *flows-state* = *set* (*e*#*accu*) |)). *F* ⊆ (*backflows* (*set* (*e*#*accu*)))))
apply(*simp add*: *backlinks-simp*)
apply(*simp add*: *implc-get-offending-flows-simp-rule*[*OF validLG*])
apply(*simp add*: *stateful-list-policy-to-list-graph-complies*)
by(*simp add*: ‹*set* (*backlinks* (*e* # *accu*)) = *backflows* (*insert e* (*set accu*))›)

show *?case*
proof(*case-tac ?caseDistinction*)
assume *cTrue*: *?caseDistinction*

from *cTrue* **have** *g1*: *TopoS-Stateful-Policy-impl.filter-compliant-stateful-ACS-accu G* (*get-impl M*) *accu* (*e* # *Es*) = *TopoS-Stateful-Policy-impl.filter-compliant-stateful-ACS-accu G* (*get-impl M*) (*e*#*accu*) *Es* **by** *simp*

from *cTrue*[*simplified case-impl-spec*] **have** *g2*: *TopoS-Stateful-Policy-Algorithm.filter-compliant-stateful-ACS-accu* (*list-graph-to-graph G*) (*get-spec M*) *accu* (*e* # *Es*) =
TopoS-Stateful-Policy-Algorithm.filter-compliant-stateful-ACS-accu (*list-graph-to-graph G*) (*get-spec M*) (*e*#*accu*) *Es*
by(*simp add*: *list-graph-to-graph-def*)

show *?case*
apply(*simp only*: *g1 g2*)
using *Cons.IH*[*OF Cons.prems*(*1*) *Cons.prems*(*2*) ‹*set Es* ⊆ *set* (*edgesL G*)› ‹*set* (*e* # *accu*) ⊆ *set* (*edgesL G*)› ‹*distinct* (*Es* @ (*e* # *accu*))›] **by** *simp*
next
assume *cFalse*: ¬ (*?caseDistinction*)

from *cFalse* **have** *g1*: *TopoS-Stateful-Policy-impl.filter-compliant-stateful-ACS-accu G* (*get-impl M*) *accu* (*e* # *Es*) = *TopoS-Stateful-Policy-impl.filter-compliant-stateful-ACS-accu G* (*get-impl M*) *accu Es* **by** *force*

from *cFalse*[*simplified case-impl-spec*] **have** *g2*: *TopoS-Stateful-Policy-Algorithm.filter-compliant-stateful-ACS-accu* (*list-graph-to-graph G*) (*get-spec M*) *accu* (*e* # *Es*) =
TopoS-Stateful-Policy-Algorithm.filter-compliant-stateful-ACS-accu (*list-graph-to-graph G*) (*get-spec M*) *accu Es*
apply(*simp add*: *list-graph-to-graph-def*) **by** *fast*

```
        show ?case
          apply(simp only: g1 g2)
           using Cons.IH[OF Cons.prems(1) Cons.prems(2) ‹set Es ⊆ set (edgesL G)› ‹set accu ⊆
set (edgesL G)› ‹distinct (Es @ accu)›] by simp
        qed
      qed


  lemma filter-compliant-stateful-ACS-cont-complies:
   ⟦ ∀ (m-impl, m-spec) ∈ set M. SecurityInvariant-complies-formal-def m-impl m-spec; wf-list-graph
G; set Es ⊆ set (edgesL G); distinct Es ⟧ ⟹
     filter-compliant-stateful-ACS-accu G (get-impl M) [] Es = TopoS-Stateful-Policy-Algorithm.filter-compliant-stateful-A
(list-graph-to-graph G) (get-spec M) Es
  apply(unfold filter-compliant-stateful-ACS-def TopoS-Stateful-Policy-Algorithm.filter-compliant-stateful-ACS-def)

   apply(rule filter-compliant-stateful-ACS-accu-complies)
      apply(simp-all)
   done

  lemma filter-compliant-stateful-ACS-complies:
   ⟦ ∀ (m-impl, m-spec) ∈ set M. SecurityInvariant-complies-formal-def m-impl m-spec; wf-list-graph
G ⟧ ⟹
     filter-compliant-stateful-ACS G (get-impl M) = TopoS-Stateful-Policy-Algorithm.filter-compliant-stateful-ACS
(list-graph-to-graph G) (get-spec M) (edgesL G)
  apply(unfold filter-compliant-stateful-ACS-def TopoS-Stateful-Policy-Algorithm.filter-compliant-stateful-ACS-def)

   apply(rule filter-compliant-stateful-ACS-accu-complies)
      apply(simp-all)
   apply(simp add: wf-list-graph-def)
   done


  definition generate-valid-stateful-policy-IFSACS :: 'v list-graph ⇒ 'v SecurityInvariant list ⇒ 'v
stateful-list-policy where
   generate-valid-stateful-policy-IFSACS G M = (let filterIFS = filter-IFS-no-violations G M in
       (let filterACS = filter-compliant-stateful-ACS-accu G M [] filterIFS in (| hostsL = nodesL G,
flows-fixL = edgesL G, flows-stateL = filterACS |)))


 fun inefficient-list-intersect :: 'a list ⇒ 'a list ⇒ 'a list where
   inefficient-list-intersect [] bs = [] |
    inefficient-list-intersect (a#as) bs = (if a ∈ set bs then a#(inefficient-list-intersect as bs) else
inefficient-list-intersect as bs)
 lemma inefficient-list-intersect-correct: set (inefficient-list-intersect a b) = (set a) ∩ (set b)
   apply(induction a)
    by(simp-all)

 definition generate-valid-stateful-policy-IFSACS-2 :: 'v list-graph ⇒ 'v SecurityInvariant list ⇒ 'v
stateful-list-policy where
```

generate-valid-stateful-policy-IFSACS-2 G M =
(| *hostsL* = *nodesL G*, *flows-fixL* = *edgesL G*, *flows-stateL* = *inefficient-list-intersect* (*filter-IFS-no-violations G M*) (*filter-compliant-stateful-ACS G M*) |)

lemma *generate-valid-stateful-policy-IFSACS-2-complies*: ⟦∀ (*m-impl*, *m-spec*) ∈ *set M*. *Security-Invariant-complies-formal-def m-impl m-spec*;
wf-list-graph G;
valid-reqs (*get-spec M*);
TopoS-Composition-Theory.all-security-requirements-fulfilled (*get-spec M*) (*list-graph-to-graph G*);
$\mathcal{T}$ = (*generate-valid-stateful-policy-IFSACS-2 G* (*get-impl M*))⟧ ⟹
stateful-policy-compliance (|*hosts* = *set* (*hostsL* $\mathcal{T}$), *flows-fix* = *set* (*flows-fixL* $\mathcal{T}$), *flows-state* = *set* (*flows-stateL* $\mathcal{T}$) |) (*list-graph-to-graph G*) (*get-spec M*)
apply(*rule-tac edgesList*=*edgesL G* **in** *generate-valid-stateful-policy-IFSACS-2-stateful-policy-compliance*)
apply(*simp*)
apply (*metis wf-list-graph-def wf-list-graph-iff-wf-graph*)
apply(*simp*)
apply(*simp add*: *list-graph-to-graph-def*)
apply(*simp add*: *TopoS-Stateful-Policy-Algorithm.generate-valid-stateful-policy-IFSACS-2-def TopoS-Stateful-Policy-i*
apply(*simp add*: *list-graph-to-graph-def inefficient-list-intersect-correct*)
apply(*thin-tac* $\mathcal{T}$ = -)
apply(*frule*(*1*) *filter-compliant-stateful-ACS-complies*)
apply(*frule*(*1*) *filter-IFS-no-violations-complies*)
apply(*thin-tac* -)
apply(*thin-tac* -)
apply(*thin-tac* -)
apply(*thin-tac* -)
apply(*simp*)
by (*metis list-graph-to-graph-def*)

end
theory *METASINVAR-SystemBoundary*
imports *SINVAR-BLPtrusted-impl*
SINVAR-SubnetsInGW-impl
../TopoS-Composition-Theory-impl
begin

11.1.1 Meta SecurityInvariant: System Boundaries

datatype *system-components* = *SystemComponent*
| *SystemBoundaryInput*
| *SystemBoundaryOutput*
| *SystemBoundaryInputOutput*

fun *system-components-to-subnets* :: *system-components* ⇒ *subnets* **where**
system-components-to-subnets SystemComponent = *Member* |
system-components-to-subnets SystemBoundaryInput = *InboundGateway* |
system-components-to-subnets SystemBoundaryOutput = *Member* |
system-components-to-subnets SystemBoundaryInputOutput = *InboundGateway*

fun *system-components-to-blp* :: *system-components* ⇒ *SINVAR-BLPtrusted.node-config* **where**
system-components-to-blp SystemComponent = (| *security-level* = *1*, *trusted* = *False* |) |
system-components-to-blp SystemBoundaryInput = (| *security-level* = *1*, *trusted* = *False* |) |
system-components-to-blp SystemBoundaryOutput = (| *security-level* = *0*, *trusted* = *True* |) |
system-components-to-blp SystemBoundaryInputOutput = (| *security-level* = *0*, *trusted* = *True* |)

definition *new-meta-system-boundary* :: (*'v::vertex* × *system-components*) *list* ⇒ *string* ⇒ (*'v SecurityInvariant*) *list* **where**
new-meta-system-boundary C description = [
new-configured-list-SecurityInvariant SINVAR-LIB-SubnetsInGW (|
node-properties = *map-of* (*map* (λ(*v*,*c*). (*v*, *system-components-to-subnets c*)) *C*)
|) (*description* @ '' (*ACS*)'')
,
new-configured-list-SecurityInvariant SINVAR-LIB-BLPtrusted (|
node-properties = *map-of* (*map* (λ(*v*,*c*). (*v*, *system-components-to-blp c*)) *C*)
|) (*description* @ '' (*IFS*)'')
]

lemma *system-components-to-subnets*:
SINVAR-SubnetsInGW.allowed-subnet-flow
SINVAR-SubnetsInGW.default-node-properties
(*system-components-to-subnets c*) ⟷
c = *SystemBoundaryInput* ∨ *c* = *SystemBoundaryInputOutput*
by(*cases c*)(*simp-all add*: *SINVAR-SubnetsInGW.default-node-properties-def*)

lemma *system-components-to-blp*:
(¬ *trusted SINVAR-BLPtrusted.default-node-properties* ⟶
security-level (*system-components-to-blp c*) ≤ *security-level SINVAR-BLPtrusted.default-node-properties*)
⟷
c = *SystemBoundaryOutput* ∨ *c* = *SystemBoundaryInputOutput*
by(*cases c*)(*simp-all add*: *SINVAR-BLPtrusted.default-node-properties-def*)

lemma *all-security-requirements-fulfilled* (*new-meta-system-boundary C description*) *G* ⟷
(∀ (v_1, v_2) ∈ *set* (*edgesL G*). *case* ((*map-of C*) v_1, (*map-of C*) v_2)
of
— No restrictions outside of the component
(*None*, *None*) ⇒ *True*

— no restrictions inside the component
| (*Some c1*, *Some c2*) ⇒ *True*

— System Boundaries Input
| (*None*, *Some SystemBoundaryInputOutput*) ⇒ *True*
| (*None*, *Some SystemBoundaryInput*) ⇒ *True*

— System Boundaries Output
| (*Some SystemBoundaryOutput*, *None*) ⇒ *True*
| (*Some SystemBoundaryInputOutput*, *None*) ⇒ *True*

— everything else is prohibited
| - ⇒ *False*

```
      )
apply(simp)
apply(simp add: new-meta-system-boundary-def)
apply(simp add: all-security-requirements-fulfilled-def)
apply(simp add: Let-def)
apply(simp add: SINVAR-LIB-SubnetsInGW-def SINVAR-LIB-BLPtrusted-def)
apply(simp add: SINVAR-SubnetsInGW-impl.NetModel-node-props-def SINVAR-BLPtrusted-impl.NetModel-node-props
apply(rule iffI)
 apply(clarsimp)
 subgoal for a b
 apply(erule-tac x=(a,b) in ballE)+
  apply(simp-all)
 apply(case-tac map-of C a)
  apply(case-tac map-of C b)
   apply(simp-all)
  apply(simp add: map-of-map)
  apply(simp split: system-components.split)
  apply(simp add: system-components-to-subnets)
  apply blast
 apply(case-tac map-of C b)
  apply(simp add: map-of-map)
  apply(simp split: system-components.split)
  apply(simp add: system-components-to-blp)
  apply blast
 apply(simp add: map-of-map)
 apply(simp split: system-components.split; fail)
 done
apply(intro conjI)
 apply(simp add: map-of-map)
 apply(clarsimp)
 subgoal for a b
 apply(erule-tac x=(a,b) in ballE)+
  apply(simp-all)
 apply(simp split: option.split-asm system-components.split-asm)
    by(simp-all add: SINVAR-SubnetsInGW.default-node-properties-def)
apply(clarsimp)
subgoal for a b
apply(erule-tac x=(a,b) in ballE)+
 apply(simp-all)
apply(simp add: map-of-map)
apply(simp split: option.split-asm system-components.split-asm)
  apply(simp add: SINVAR-BLPtrusted.default-node-properties-def; fail)
 apply(rename-tac x, case-tac x, simp-all)+
done
done


value[code] let nodes = [1,2,3,4,8,9,10];
           sinvars = new-meta-system-boundary
              [(1::int, SystemBoundaryInput),
               (2, SystemComponent),
               (3, SystemBoundaryOutput),
               (4, SystemBoundaryInputOutput)
               ] ''foobar''
```

in generate-valid-topology sinvars (| *nodesL* = *nodes*, *edgesL* = *List.product nodes nodes* |)

end
theory *TopoS-Impl*
imports *TopoS-Library TopoS-Composition-Theory-impl*

Security-Invariants/METASINVAR-SystemBoundary

Lib/ML-GraphViz
TopoS-Stateful-Policy-impl
begin

12 ML Visualization Interface

definition *print-offending-flows-debug* ::
'*v SecurityInvariant list* ⇒ '*v list-graph* ⇒ (*string* × ('*v* × '*v*) *list list*) *list* **where**
print-offending-flows-debug M G = *map*
(λ*m*.
(*implc-description m* @ '' ('' @ *implc-type m* @ '')''
, *implc-offending-flows m G*)
) *M*

ML‹
fun pretty-assoclist ctxt header t = *let*
val ls : (*term* ∗ *term*) *list* = *t* |> *HOLogic.dest-list* |> *map HOLogic.dest-prod*;
val pretty = *fn t* => *Pretty.string-of* (*Syntax.pretty-term ctxt t*);
in ls
|> *map* (*fn* (*x*, *y*) => ^*pretty x*^: ^*pretty y*)
|> *space-implode* \n
|> (*fn s* => *header*^*s*)
|> *writeln end*
›

12.1 Utility Functions

fun *rembiflowdups* :: ('*a* × '*a*) *list* ⇒ ('*a* × '*a*) *list* **where**
rembiflowdups [] = [] |
rembiflowdups ((*s*,*r*)#*as*) = (*if* (*s*,*r*) ∈ *set as* ∨ (*r*,*s*) ∈ *set as then rembiflowdups as else* (*s*,*r*)#*rembiflowdups as*)

lemma *rembiflowdups-complete*: ⟦ ∀ (*s*,*r*) ∈ *set x*. (*r*,*s*) ∈ *set x* ⟧ ⟹ *set* (*rembiflowdups x*) ∪ *set* (*backlinks* (*rembiflowdups x*)) = *set x*
proof
assume *a*: ∀ (*s*,*r*) ∈ *set x*. (*r*,*s*) ∈ *set x*
have *subset1*: *set* (*rembiflowdups x*) ⊆ *set x*
apply(*induction x*)
apply(*simp*)
apply(*clarsimp*)
apply(*simp split*: *if-split-asm*)
by(*blast*)+
have *set-backlinks-simp*: ⋀ *x*. ∀ (*s*,*r*) ∈ *set x*. (*r*,*s*) ∈ *set x* ⟹ *set* (*backlinks x*) = *set x*

apply(*simp add*: *backlinks-set*)
apply(*rule*)
by *fast+*
have *subset2*: *set* (*backlinks* (*rembiflowdups x*)) ⊆ *set x*
apply(*subst set-backlinks-simp*[*OF a*, *symmetric*])
by(*simp add*: *backlinks-subset subset1*)

from *subset1 subset2*
show *set* (*rembiflowdups x*) ∪ *set* (*backlinks* (*rembiflowdups x*)) ⊆ *set x* **by** *blast*
next
show *set x* ⊆ *set* (*rembiflowdups x*) ∪ *set* (*backlinks* (*rembiflowdups x*))
apply(*rule*)
apply(*induction x*)
apply(*simp*)
apply(*rename-tac a as e*)
apply(*simp*)
apply(*erule disjE*)
apply(*simp*)
defer
apply *fastforce*
apply(*case-tac a*)
apply(*rename-tac s r*)
apply(*case-tac* (*s*,*r*) ∉ *set as* ∧ (*r*,*s*) ∉ *set as*)
apply(*simp*)
apply(*simp add*: *backlinks-set*)
by *blast*
qed

only for prettyprinting

definition *filter-for-biflows*:: (′*a* × ′*a*) *list* ⇒ (′*a* × ′*a*) *list* **where**
filter-for-biflows E ≡ [*e* ← *E*. (*snd e*, *fst e*) ∈ *set E*]

definition *filter-for-uniflows*:: (′*a* × ′*a*) *list* ⇒ (′*a* × ′*a*) *list* **where**
filter-for-uniflows E ≡ [*e* ← *E*. (*snd e*, *fst e*) ∉ *set E*]

lemma *filter-for-biflows-correct*: ∀ (*s*,*r*) ∈ *set* (*filter-for-biflows E*). (*r*,*s*) ∈ *set* (*filter-for-biflows E*)
unfolding *filter-for-biflows-def*
by(*induction E*, *auto*)

lemma *filter-for-biflows-un-filter-for-uniflows*: *set* (*filter-for-biflows E*) ∪ *set* (*filter-for-uniflows E*) = *set E*
apply(*simp add*: *filter-for-biflows-def filter-for-uniflows-def*) **by** *blast*

definition *partition-by-biflows* :: (′*a* × ′*a*) *list* ⇒ ((′*a* × ′*a*) *list* × (′*a* × ′*a*) *list*) **where**
partition-by-biflows E ≡ (*rembiflowdups* (*filter-for-biflows E*), *remdups* (*filter-for-uniflows E*))

lemma *partition-by-biflows-correct*: *case partition-by-biflows E of* (*biflows*, *uniflows*) ⇒ *set biflows* ∪ *set* (*backlinks* (*biflows*)) ∪ *set uniflows* = *set E*
apply(*simp add*: *partition-by-biflows-def*)
by(*simp add*: *filter-for-biflows-un-filter-for-uniflows filter-for-biflows-correct rembiflowdups-complete*)

lemma *partition-by-biflows* [(*1*::*int*, *1*::*int*), (*1*,*2*), (*2*, *1*), (*1*,*3*)] = ([(*1*, *1*), (*2*, *1*)], [(*1*, *3*)]) **by**

eval

ML‹

```
(*apply args to f. f ist best supplied using @{const-name name-of-function} *)
fun apply-function (ctxt: Proof.context) (f: string) (args: term list) : term =
  let
   val - = writeln (applying ^f^ to  ^(fold (fn t => fn acc => acc^(Pretty.string-of (Syntax.pretty-term
(Config.put show-types true ctxt) t))^ ) args ));
    (*val t-eval = Code-Evaluation.dynamic-value-strict thy t;*)
    (* $ associates to the left, give f its arguments*)
    val applied-untyped-uneval: term = list-comb (Const (f, dummyT), args);
    val applied-uneval: term = Syntax.check-term ctxt applied-untyped-uneval;
  in
    applied-uneval |> Code-Evaluation.dynamic-value-strict ctxt
  end;


(*ctxt -> edges -> (biflows, uniflows)*)
fun partition-by-biflows ctxt (t: term) : (term * term) =
  apply-function ctxt @{const-name partition-by-biflows} [t] |> HOLogic.dest-prod


local
  fun get-tune-node-format (edges: term) : term -> string -> string =
    if (fastype-of edges) = @{typ (string × string) list}
    then
      tune-string-vertex-format
    else
      Graphviz.default-tune-node-format;

  fun evalutae-term ctxt (edges: term) : term =
    case Code-Evaluation.dynamic-value ctxt edges
      of SOME x => x
      | NONE => raise TERM (could not evaluate, []);
in
  fun visualize-edges ctxt (edges: term) (coloredges: (string * term) list) (graphviz-header: string) =
    let
      val - = writeln(visualize-edges);
      val (biflows, uniflows) = partition-by-biflows ctxt edges;
    in
      Graphviz.visualize-graph-pretty ctxt (get-tune-node-format edges) ([
      (, uniflows),
      (edge [dir=\none\, color=\#000000\], biflows)] @ coloredges) (*dir=none, dir=both*)
       graphviz-header
    end

  (*iterate over the edges in ML, useful for printing them in certain formats*)
  fun iterate-edges-ML ctxt (edges: term) (all: (string*string) -> unit) (bi: (string*string) -> unit)
(uni: (string*string) -> unit): unit =
    let
      val - = writeln(iterate-edges-ML);
      val tune-node-format = (get-tune-node-format edges);
```

```
    val node-to-string = Graphviz.node-to-string ctxt tune-node-format;
    val evaluated-edges : term = evalutae-term ctxt edges;
    val (biflows, uniflows) = partition-by-biflows ctxt evaluated-edges;
  in
    let
        fun edge-to-list (es: term) : (term * term) list = es |> HOLogic.dest-list |> map HO-
Logic.dest-prod;
      fun edge-to-string (es: (term * term) list) : (string * string) list =
        map (fn (v1, v2) => (node-to-string v1, node-to-string v2)) es
    in
      edge-to-list evaluated-edges |> edge-to-string |> map all;
      edge-to-list biflows |> edge-to-string |> map bi;
      edge-to-list uniflows |> edge-to-string |> map uni;
      ()
    end
    handle Subscript => writeln (Subscript Exception in iterate-edges-ML)
  end;

end
›
```

ML-val‹
```
local
  val (biflows, uniflows) = partition-by-biflows @{context} @{term [(1::int, 1::int), (1,2), (2, 1),
(1,3)]};
in
  val - = Pretty.writeln (Syntax.pretty-term (Config.put show-types true @{context}) biflows);
  val - = Pretty.writeln (Syntax.pretty-term (Config.put show-types true @{context}) uniflows);
end;

val t = fastype-of @{term [(''x'', 2::nat)]};

›
```
ML-val‹(*
```
visualize-edges @{context}  @{term [(1::int, 1::int), (1,2), (2, 1), (1,3)]} []; *)
›
```

definition *internal-get-invariant-types-list*:: *'a SecurityInvariant list* ⇒ *string list* **where**
internal-get-invariant-types-list M ≡ *map implc-type M*

definition *internal-node-configs* :: *'a list-graph* ⇒ (*'a* ⇒ *'b*) ⇒ (*'a* × *'b*) *list* **where**
internal-node-configs G config ≡ *zip* (*nodesL G*) (*map config* (*nodesL G*))

ML ‹
```
local
 fun get-graphivz-node-desc ctxt (node-config: term): string =
  let
    val (node, config) = HOLogic.dest-prod node-config;
    (*TODO: tune node format? There must be a better way ...*)
    val tune-node-format = if (fastype-of node) = @{typ string}
     then
```

```
        tune-string-vertex-format
      else
        Graphviz.default-tune-node-format;
     val node-str = Graphviz.node-to-string ctxt tune-node-format node;
     val config-str = Graphviz.term-to-string-html ctxt config;
   in
    node-str^[label=<<TABLE BORDER=\0\ CELLSPACING=\0\><TR><TD><FONT face=\Verdana
Bold\>^node-str^</FONT></TD></TR><TR><TD>^config-str^</TD></TR></TABLE>>]\n
   end;
in
  fun generate-graphviz-header ctxt (G: term) (configs: term): string =
    let
      val configlist: term list = apply-function ctxt @{const-name internal-node-configs} [G, configs] |>
HOLogic.dest-list;
    in
      fold (fn c => fn acc => acc^get-graphivz-node-desc ctxt c) configlist
    end;
end;

(* Convenience function. Use whenever possible!
  M: security requirements, list
  G: list-graph*)
fun visualize-graph-header ctxt (M: term) (G: term) (Config: term): unit =
  let
    val wf-list-graph = apply-function ctxt @{const-name wf-list-graph} [G];
    val all-fulfilled = apply-function ctxt @{const-name all-security-requirements-fulfilled} [M, G];
    val edges = apply-function ctxt @{const-name edgesL} [G];
    val invariants = apply-function ctxt @{const-name internal-get-invariant-types-list} [M];
    val - = writeln(Invariants: ^ Pretty.string-of (Syntax.pretty-term ctxt invariants));
    val header = if Config = @{term []} then #header else generate-graphviz-header ctxt G Config;
  in
    if wf-list-graph = @{term False} then
      error (The supplied graph is syntactically invalid. Check wf-list-graph.)
    else if all-fulfilled = @{term False} then
      (let
        val offending = apply-function ctxt @{const-name implc-get-offending-flows} [M, G];
            val offending-flat = apply-function ctxt @{const-name List.remdups} [apply-function ctxt
@{const-name List.concat} [offending]];
        val offending-debug = apply-function ctxt @{const-name print-offending-flows-debug} [M, G];
      in
       writeln(offending flows:);
       Pretty.writeln (Syntax.pretty-term ctxt offending);
       pretty-assoclist ctxt Offending flows per invariant:\n offending-debug;
     visualize-edges ctxt edges [(edge [dir=\arrow\, style=dashed, color=\#FF0000\, constraint=false],
offending-flat)] header;
      () end)
    else if all-fulfilled <> @{term True} then raise ERROR all-fulfilled neither False nor True else (
       writeln(All valid:);
       visualize-edges ctxt edges [] header;
      ())
  end;


fun visualize-graph ctxt (M: term) (G: term): unit = visualize-graph-header ctxt M G @{term []};
```

›

end

13 Network Security Policy Verification

theory *Network-Security-Policy-Verification*
imports
  TopoS-Interface
  TopoS-Interface-impl
  TopoS-Library
  TopoS-Composition-Theory
  TopoS-Stateful-Policy
  TopoS-Composition-Theory-impl
  TopoS-Stateful-Policy-Algorithm
  TopoS-Stateful-Policy-impl
  TopoS-Impl
begin

14 A small Tutorial

We demonstrate usage of the executable theory.

Everything that is indented and starts with 'Interlude:' summarizes the main correctness proofs and can be skipped if only the implementation is concerned

14.1 Policy

The secuity policy is a directed graph.

definition *policy* :: *nat list-graph* **where**
  policy ≡ (| *nodesL* = [*1*,*2*,*3*],
      edgesL = [(*1*,*2*), (*2*,*2*), (*2*,*3*)] |)

It is syntactically well-formed

lemma *wf-list-graph-policy*: *wf-list-graph policy* **by** *eval*

In contrast, this is not a syntactically well-formed graph.

lemma ¬ *wf-list-graph* (| *nodesL* = [*1*,*2*]::*nat list*, *edgesL* = [(*1*,*2*), (*2*,*2*), (*2*,*3*)] |) **by** *eval*

Our *policy* has three rules.

lemma *length* (*edgesL policy*) = *3* **by** *eval*

14.2 Security Invariants

We construct a security invariant. Node *2* has confidential data

definition *BLP-security-levels* :: *nat* ⇀ *SINVAR-BLPtrusted.node-config***where**
  BLP-security-levels ≡ [*2* ↦ (| *security-level* = *1*, *trusted* = *False* |)]

definition *BLP-m*::(*nat SecurityInvariant*) **where**
  BLP-m ≡ *new-configured-list-SecurityInvariant SINVAR-LIB-BLPtrusted* (|

node-properties = *BLP-security-levels*
⦈ *''Two has confidential information''*

Interlude: *BLP-m* is a valid implementation of a SecurityInvariant

definition *BLP-m-spec* :: *nat SecurityInvariant-configured option***where**
BLP-m-spec ≡ *new-configured-SecurityInvariant* (
SINVAR-BLPtrusted.sinvar,
SINVAR-BLPtrusted.default-node-properties,
SINVAR-BLPtrusted.receiver-violation,
SecurityInvariant.node-props SINVAR-BLPtrusted.default-node-properties ⦇
node-properties = *BLP-security-levels*
⦈)

Fist, we need to show that the formal definition obeys all requirements, *new-configured-SecurityInvariant* verifies this. To double check, we manually give the configuration.

lemma *BLP-m-spec*: **assumes** *nP* = (λ *v*. (*case BLP-security-levels v of Some c* ⇒ *c* | *None* ⇒ *SINVAR-BLPtrusted.default-node-properties*))
shows *BLP-m-spec* = *Some* ⦇
c-sinvar = (λ*G*. *SINVAR-BLPtrusted.sinvar G nP*),
c-offending-flows = (λ*G*. *SecurityInvariant-withOffendingFlows.set-offending-flows SINVAR-BLPtrusted.sinvar G nP*),
c-isIFS = *SINVAR-BLPtrusted.receiver-violation*
⦈ (**is** *BLP-m-spec* = *Some ?Spec*)
proof −
have *NetModelLib*: *TopoS-modelLibrary SINVAR-LIB-BLPtrusted SINVAR-BLPtrusted.sinvar*
by(*unfold-locales*)
from *assms* **have** *nP*: *nP* = *nm-node-props SINVAR-LIB-BLPtrusted* ⦇
node-properties = *BLP-security-levels*
⦈ **by**(*simp add*: *fun-eq-iff SINVAR-LIB-BLPtrusted-def SINVAR-BLPtrusted-impl.NetModel-node-props-def*)

have *BLP-m-spec* = *new-configured-SecurityInvariant* (*SINVAR-BLPtrusted.sinvar*, *SINVAR-BLPtrusted.default-nod*
SINVAR-BLPtrusted.receiver-violation, *nP*)
unfolding *BLP-m-spec-def nP* **by**(*simp add*: *SINVAR-BLPtrusted-impl.NetModel-node-props-def SINVAR-LIB-BLPtrusted-def*)
also with *TopoS-modelLibrary-yields-new-configured-SecurityInvariant*[*OF NetModelLib nP*]
have . . . = *Some ?Spec* **by** (*simp add*: *SINVAR-LIB-BLPtrusted-def*)
finally show *?thesis* **by** *blast*
qed
lemma *valid-reqs-BLP*: *valid-reqs* [*the BLP-m-spec*]
by(*simp add*: *valid-reqs-def*)(*metis BLP-m-spec-def BLPtrusted-impl.spec new-configured-SecurityInvariant.simps new-configured-SecurityInvariant-sound option.distinct*(*1*) *option.exhaust-sel*)

Interlude: While *BLP-m* is executable code, we will now show that this executable code complies with its formal definition.

lemma *complies-BLP*: *SecurityInvariant-complies-formal-def BLP-m* (*the BLP-m-spec*)
unfolding *BLP-m-def*
apply(*rule new-configured-list-SecurityInvariant-complies*)
apply(*simp-all add*: *BLP-m-spec-def*)
apply(*unfold-locales*)
by(*simp add*: *fun-eq-iff SINVAR-LIB-BLPtrusted-def SINVAR-BLPtrusted-impl.NetModel-node-props-def*)

We define the list of all security invariants of type *nat SecurityInvariant list*. The type *nat* is because the policy's nodes are of type *nat*.

definition *security-invariants* = [*BLP-m*]

We can see that the policy does not fulfill the security invariants.

lemma ¬ *all-security-requirements-fulfilled security-invariants policy* **by** *eval*

We ask why. Obviously, node 2 leaks confidential data to node 3.

value *implc-get-offending-flows security-invariants policy*
lemma *implc-get-offending-flows security-invariants policy* = [[(*2*, *3*)]] **by** *eval*

Interlude: the implementation *implc-get-offending-flows* corresponds to the formal definition *get-offending-flows*

lemma *set ' set* (*implc-get-offending-flows* (*get-impl* [(*BLP-m*, *the BLP-m-spec*)]) *policy*) = *get-offending-flows* (*get-spec* [(*BLP-m*, *the BLP-m-spec*)]) (*list-graph-to-graph policy*)
apply(*rule implc-get-offending-flows-complies*)
by(*simp-all add*: *complies-BLP wf-list-graph-policy*)

Visualization of the violation (only in interactive mode)

ML-val‹
visualize-graph @{*context*} @{*term security-invariants*} @{*term policy*};
›

Experimental: the config (only one) can be added to the end.

ML-val‹
visualize-graph-header @{*context*} @{*term security-invariants*} @{*term policy*} @{*term BLP-security-levels*};
›

The policy has a flaw. We throw it away and generate a new one which fulfills the invariants.

definition *max-policy* = *generate-valid-topology security-invariants* (|*nodesL* = *nodesL policy*, *edgesL* = *List.product* (*nodesL policy*) (*nodesL policy*) |)

Interlude: the implementation *implc-get-offending-flows* corresponds to the formal definition *get-offending-flows*

thm *generate-valid-topology-complies*

Interlude: the formal definition is sound

thm *generate-valid-topology-sound*

Here, it is also complete

lemma *wf-graph G* ⟹ *max-topo* [*the BLP-m-spec*] (*TopoS-Composition-Theory.generate-valid-topology* [*the BLP-m-spec*] (*fully-connected G*))
apply(*rule generate-valid-topology-max-topo*[*OF valid-reqs-BLP*])
apply(*assumption*)
apply(*simp add*: *BLP-m-spec*)
by *blast*

Calculating the maximum policy

value *max-policy*
lemma *max-policy* = (|*nodesL* = [*1*, *2*, *3*], *edgesL* = [(*1*, *1*), (*1*, *2*), (*1*, *3*), (*2*, *2*), (*3*, *1*), (*3*, *2*), (*3*, *3*)]|) **by** *eval*

Visualizing the maximum policy (only in interactive mode)

ML‹
visualize-graph @{*context*} @{*term security-invariants*} @{*term max-policy*};
›

Of course, all security invariants hold for the maximum policy.

lemma *all-security-requirements-fulfilled security-invariants max-policy* **by** *eval*

14.3 A stateful implementation

We generate a stateful policy

definition *stateful-policy* = *generate-valid-stateful-policy-IFSACS-2 policy security-invariants*

When thinking about it carefully, no flow can be stateful without introducing an information leakage here!

value *stateful-policy*
lemma *stateful-policy* = (|*hostsL* = [*1*, *2*, *3*], *flows-fixL* = [(*1*, *2*), (*2*, *2*), (*2*, *3*)], *flows-stateL* = []|)
by *eval*

Interlude: the stateful policy we are computing fulfills all the necessary properties

thm *generate-valid-stateful-policy-IFSACS-2-complies*

thm *filter-compliant-stateful-ACS-correct filter-compliant-stateful-ACS-maximal*
thm *filter-IFS-no-violations-correct filter-IFS-no-violations-maximal*

Visualizing the stateful policy (only in interactive mode)

ML-val‹
visualize-edges @{*context*} @{*term flows-fixL stateful-policy*}
[(*edge* [*dir*=*arrow*\, *style*=*dashed*, *color*=\#*FF8822*\, *constraint*=*false*], @{*term flows-stateL stateful-policy*})] ;
›

This is how it would look like if $(3::'a,\ 1)$ were a stateful flow

ML-val‹
visualize-edges @{*context*} @{*term flows-fixL stateful-policy*}
[(*edge* [*dir*=*arrow*\, *style*=*dashed*, *color*=\#*FF8822*\, *constraint*=*false*], @{*term* [(*3*::*nat*,*1*::*nat*)]})]
;
›

hide-const *policy security-invariants max-policy stateful-policy*

end
theory *Example-BLP*
imports *TopoS-Library*
begin

definition *BLPexample1*::*bool* **where**
BLPexample1 ≡ (*nm-eval SINVAR-LIB-BLPbasic*) *fabNet* (| *node-properties* = [''*PresenceSensor*'' ↦ *2*,
''*Webcam*'' ↦ *3*,

''SensorSink'' ↦ 3,
''Statistics'' ↦ 3] ⦈

definition *BLPexample3*::(*string* × *string*) *list list* **where**
BLPexample3 ≡ (*nm-offending-flows SINVAR-LIB-BLPbasic*) *fabNet* ((*nm-node-props SINVAR-LIB-BLPbasic*) *sensorProps-NMParams-try3*)

value *BLPexample1*
value *BLPexample3*

end
theory *TopoS-generateCode*
imports
TopoS-Library
Example-BLP
begin

```
setup ‹fn thy =>
  let
    val package = package tum.in.net.psn.log-topo.SecurityInvariants.GENERATED;
    val date = Date.toString (Date.fromTimeLocal (Time.now ()));
    val export-file = Context.theory-base-name thy ^ .thy;
    val header = package ^ \n ^ // Generated by  ^ Isabelle-System.identification () ^  on  ^ date ^
\n ^ // src:  ^ export-file ^ \n;
  in
    Code-Target.set-printings (Code-Symbol.Module (, [(Scala, SOME (header, []))])) thy
  end
›
```

export-code
— generic network security invariants
SINVAR-LIB-BLPbasic
SINVAR-LIB-Dependability
SINVAR-LIB-DomainHierarchyNG
SINVAR-LIB-Subnets
SINVAR-LIB-BLPtrusted
SINVAR-LIB-PolEnforcePointExtended
SINVAR-LIB-Sink
SINVAR-LIB-NonInterference
SINVAR-LIB-SubnetsInGW
SINVAR-LIB-CommunicationPartners
— accessors to the packed invariants
nm-eval
nm-node-props
nm-offending-flows
nm-sinvar
nm-default
nm-receiver-violation nm-name
— TopoS Params
node-properties
— Finite Graph functions
FiniteListGraph.wf-list-graph
FiniteListGraph.add-node

FiniteListGraph.delete-node
FiniteListGraph.add-edge
FiniteListGraph.delete-edge
FiniteListGraph.delete-edges
— Examples
BLPexample1 BLPexample3
in *Scala*

end
theory *SINVAR-Examples*
imports
TopoS-Interface
TopoS-Interface-impl
TopoS-Library
TopoS-Composition-Theory
TopoS-Stateful-Policy
TopoS-Composition-Theory-impl
TopoS-Stateful-Policy-Algorithm
TopoS-Stateful-Policy-impl
TopoS-Impl
begin

ML‹
case !*Graphviz.open-viewer of*
OpenImmediately => *Graphviz.open-viewer* := *AskTimeouted 3.0*
| *AskTimeouted* - => ()
| *DoNothing* => ()
›

definition *make-policy* :: ('*a SecurityInvariant*) *list* ⇒ '*a list* ⇒ '*a list-graph* **where**
make-policy sinvars V ≡ *generate-valid-topology sinvars* (|*nodesL* = *V*, *edgesL* = *List.product V V* |)

context begin
private definition *SINK-m* ≡ *new-configured-list-SecurityInvariant SINVAR-LIB-Sink* (|
node-properties = [''*Bot1*'' ↦ *Sink*,
''*Bot2*'' ↦ *Sink*,
''*MissionControl1*'' ↦ *SinkPool*,
''*MissionControl2*'' ↦ *SinkPool*
]
|) ''*bots and control are infromation sink*''
value[*code*] *make-policy* [*SINK-m*] [''*INET*'', ''*Supervisor*'', ''*Bot1*'', ''*Bot2*'', ''*MissionControl1*'', ''*MissionControl2*'']
ML-val‹
visualize-graph-header @{*context*} @{*term* [*SINK-m*]}
@{*term make-policy* [*SINK-m*] [''*INET*'', ''*Supervisor*'', ''*Bot1*'', ''*Bot2*'', ''*MissionControl1*'', ''*MissionControl2*'']}
@{*term* [''*Bot1*'' ↦ *Sink*,
''*Bot2*'' ↦ *Sink*,
''*MissionControl1*'' ↦ *SinkPool*,

''MissionControl2'' ↦ SinkPool
]};
›
end

context begin
private definition *ACL-m* ≡ *new-configured-list-SecurityInvariant SINVAR-LIB-CommunicationPartners*
(|
node-properties = [''db1'' ↦ *Master* [''h1'', ''h2''],
''db2'' ↦ *Master* [''db1''],
''h1'' ↦ *Care*,
''h2'' ↦ *Care*
]
|) ''ACL for databases''
value[*code*] *make-policy* [*ACL-m*] [''db1'', ''db2'', ''h1'', ''h2'', ''h3'']
ML-val‹
visualize-graph-header @{*context*} @{*term* [*ACL-m*]}
@{*term make-policy* [*ACL-m*] [''db1'', ''db2'', ''h1'', ''h2'', ''h3'']}
@{*term* [''db1'' ↦ *Master* [''h1'', ''h2''],
''db2'' ↦ *Master* [''db1''],
''h1'' ↦ *Care*,
''h2'' ↦ *Care*
]};
›
end

definition *CommWith-m*::(*nat SecurityInvariant*) **where**
CommWith-m ≡ *new-configured-list-SecurityInvariant SINVAR-LIB-ACLcommunicateWith* (|
node-properties = [
1 ↦ [*2*,*3*],
2 ↦ [*3*]]
|) ''One can only talk to 2,3''

Experimental: the config (only one) can be added to the end.

ML-val‹
visualize-graph-header @{*context*} @{*term* [*CommWith-m*]}
@{*term* (| *nodesL* = [*1*::*nat*, *2*, *3*],
edgesL = [(*1*,*2*), (*2*,*3*)]|)} @{*term* [
(*1*::*nat*) ↦ [*2*::*nat*,*3*],
2 ↦ [*3*]]};
›

value[*code*] *make-policy* [*CommWith-m*] [*1*,*2*,*3*]
value[*code*] *implc-offending-flows CommWith-m* (|*nodesL* = [*1*,*2*,*3*,*4*], *edgesL* = *List.product* [*1*,*2*,*3*,*4*] [*1*,*2*,*3*,*4*] |)

```
value[code] make-policy [CommWith-m] [1,2,3,4]
```

```
ML-val‹
visualize-graph @{context} @{term [ new-configured-list-SecurityInvariant SINVAR-LIB-ACLcommunicateWith
⦇
          node-properties = [
                  1::nat ↦ [1,2,3],
                  2 ↦ [1,2,3,4],
                  3 ↦ [1,2,3,4],
                  4 ↦ [1,2,3,4]]
        ⦈ ''usefull description here'']} @{term ⦇nodesL = [1::nat,2,3,4], edgesL = [(1,2), (1,3), (2,3),
(3, 4)] ⦈};
›
```

```
lemma implc-offending-flows (new-configured-list-SecurityInvariant SINVAR-LIB-ACLcommunicateWith
⦇
          node-properties = [
                  1::nat ↦ [1,2,3],
                  2 ↦ [1,2,3,4],
                  3 ↦ [1,2,3,4],
                  4 ↦ [1,2,3,4]]
        ⦈ ''usefull description here'') ⦇nodesL = [1::nat,2,3,4], edgesL = [(1,2), (1,3), (2,3), (3, 4)]
⦈ =
     [[(1, 2), (1, 3)], [(1, 3), (2, 3)], [(3, 4)]] by eval
```

```
context begin
  private definition G-dep :: nat list-graph where
    G-dep ≡ ⦇nodesL = [1::nat,2,3,4,5,6,7], edgesL = [(1,2), (2,1), (2,3),
                                                    (4,5), (5,6), (6,7)] ⦈
  private lemma wf-list-graph G-dep by eval

  private definition DEP-m ≡ new-configured-list-SecurityInvariant SINVAR-LIB-Dependability ⦇
          node-properties = Some ∘ dependability-fix-nP G-dep (λ-. 0)
          ⦈ ''automatically computed dependability invariant''
  ML-val‹
  visualize-graph-header @{context} @{term [DEP-m]}
    @{term G-dep}
    @{term Some ∘ dependability-fix-nP G-dep (λ-. 0)};
›
```

Connecting (3::'a, 4::'b). This causes only one offedning flow at (3::'a, 4::'b).

```
  ML-val‹
  visualize-graph-header @{context} @{term [DEP-m]}
    @{term G-dep⦇edgesL := (3,4)#edgesL G-dep⦈}
    @{term Some ∘ dependability-fix-nP G-dep (λ-. 0)};
›
```

We try to increase the dependability level at 3::'a. Suddenly, offending flows everywhere.

```
  ML-val‹
```

```
 visualize-graph-header @{context} @{term [new-configured-list-SecurityInvariant SINVAR-LIB-Dependability
(|
         node-properties = Some ∘ ((dependability-fix-nP G-dep (λ-. 0))(3 := 2))
         |) ''changed deps'']}
   @{term G-dep(|edgesL := (3,4)#edgesL G-dep|)}
   @{term Some ∘ ((dependability-fix-nP G-dep (λ-. 0))(3 := 2))};
›
 lemma implc-offending-flows (new-configured-list-SecurityInvariant SINVAR-LIB-Dependability (|
                       node-properties = Some ∘ ((dependability-fix-nP G-dep (λ-. 0))(3 := 2))
                       |) ''changed deps'')
        (G-dep(|edgesL := (3,4)#edgesL G-dep|)) =
      [[(3, 4)], [(1, 2), (2, 1), (5, 6)], [(1, 2), (4, 5)], [(2, 1), (4, 5)], [(2, 3), (4, 5)], [(2, 3), (5,
6)]]
          by eval
```

If we recompute the dependability levels for the changed graph, we see that suddenly, The level at *1* and *2::'a* increased, though we only added the edge (*3::'a*, *4::'b*). This hints that we connected the graph. If an attacker can now compromise *1*, she may be able to peek much deeper into the network.

```
 ML-val‹
 visualize-graph-header @{context} @{term [new-configured-list-SecurityInvariant SINVAR-LIB-Dependability
(|
         node-properties = Some ∘ dependability-fix-nP (G-dep(|edgesL := (3,4)#edgesL G-dep|)) (λ-.
0)
         |) ''changed deps'']}
   @{term G-dep(|edgesL := (3,4)#edgesL G-dep|)}
   @{term Some ∘ dependability-fix-nP (G-dep(|edgesL := (3,4)#edgesL G-dep|)) (λ-. 0)};
›
```

Dependability is reflexive, a host can depend on itself.

```
 ML-val‹
 visualize-graph-header @{context} @{term [new-configured-list-SecurityInvariant SINVAR-LIB-Dependability
(|
         node-properties = Some ∘ dependability-fix-nP (|nodesL = [1::nat], edgesL = [(1,1)] |) (λ-. 0)
         |) ''changed deps'']}
   @{term (|nodesL = [1::nat], edgesL = [(1,1)] |)}
   @{term Some ∘ dependability-fix-nP (|nodesL = [1::nat], edgesL = [(1,1)] |) (λ-. 0)};
›
 ML-val‹
 visualize-graph-header @{context} @{term [new-configured-list-SecurityInvariant SINVAR-LIB-Dependability-norefl
(|
         node-properties = (λ-::nat. Some 0)
         |) ''changed deps'']}
   @{term (|nodesL = [1::nat], edgesL = [(1,1)] |)}
   @{term (λ-::nat. Some (0::nat))};
›

end


context begin
 private definition G-noninter :: nat list-graph where
```

G-noninter ≡ (|*nodesL* = [*1*::*nat*,*2*,*3*,*4*], *edgesL* = [(*1*,*2*), (*1*,*3*), (*2*,*3*), (*3*, *4*)] |)
private lemma *wf-list-graph G-noninter* **by** *eval*

private definition *NonI-m* ≡ *new-configured-list-SecurityInvariant SINVAR-LIB-NonInterference*
(|
node-properties = [
1::*nat* ↦ *Interfering*,
2 ↦ *Unrelated*,
3 ↦ *Unrelated*,
4 ↦ *Interfering*]
|) *''One and Four interfere''*
ML-val‹
visualize-graph @{*context*} @{*term* [*NonI-m*]} @{*term G-noninter*};
›

lemma *implc-offending-flows NonI-m G-noninter* = [[(*1*, *2*), (*1*, *3*)], [(*1*, *3*), (*2*, *3*)], [(*3*, *4*)]]
by *eval*

ML-val‹
visualize-graph @{*context*} @{*term* [*NonI-m*]} @{*term* (|*nodesL* = [*1*::*nat*,*2*,*3*,*4*], *edgesL* = [(*1*,*2*), (*1*,*3*), (*2*,*3*), (*4*, *3*)] |)};
›

lemma *implc-offending-flows NonI-m* (|*nodesL* = [*1*::*nat*,*2*,*3*,*4*], *edgesL* = [(*1*,*2*), (*1*,*3*), (*2*,*3*), (*4*, *3*)] |) =
[[(*1*, *2*), (*1*, *3*)], [(*1*, *3*), (*2*, *3*)], [(*4*, *3*)]]
by *eval*

In comparison, *SINVAR-LIB-ACLcommunicateWith* is less strict. Changing the direction of the edge (*3*::*'a*, *4*::*'b*) removes the access from *1* to *4*::*'a* and the invariant holds.

lemma *implc-offending-flows* (*new-configured-list-SecurityInvariant SINVAR-LIB-ACLcommunicateWith*
(|
node-properties = [
1::*nat* ↦ [*1*,*2*,*3*],
2 ↦ [*1*,*2*,*3*,*4*],
3 ↦ [*1*,*2*,*3*,*4*],
4 ↦ [*1*,*2*,*3*,*4*]]
|) *''One must not access Four''*) (|*nodesL* = [*1*::*nat*,*2*,*3*,*4*], *edgesL* = [(*1*,*2*), (*1*,*3*), (*2*,*3*), (*4*, *3*)] |) = [] **by** *eval*
end

context begin
private definition *subnets-host-attributes* ≡ [
''v11'' ↦ *Subnet 1*,
''v12'' ↦ *Subnet 1*,
''v13'' ↦ *Subnet 1*,
''v1b'' ↦ *BorderRouter 1*,
''v21'' ↦ *Subnet 2*,
''v22'' ↦ *Subnet 2*,
''v23'' ↦ *Subnet 2*,

```
                    ''v2b'' ↦ BorderRouter 2,
                    ''v3b'' ↦ BorderRouter 3
                    ]
  private definition Subnets-m ≡ new-configured-list-SecurityInvariant SINVAR-LIB-Subnets (|
        node-properties = subnets-host-attributes
        |) ''Collaborating hosts''
  private definition subnet-hosts ≡ [''v11'', ''v12'', ''v13'', ''v1b'',
                              ''v21'', ''v22'', ''v23'', ''v2b'',
                              ''v3b'', ''vo'']

  private lemma dom (subnets-host-attributes) ⊆ set (subnet-hosts)
    by(simp add: subnet-hosts-def subnets-host-attributes-def)
  value[code] make-policy [Subnets-m] subnet-hosts
  ML-val‹
  visualize-graph-header @{context} @{term [Subnets-m]}
    @{term make-policy [Subnets-m] subnet-hosts}
    @{term subnets-host-attributes};
›
```

Emulating the same but with accessible members with SubnetsInGW and ACLs

```
 private definition SubnetsInGW-ACL-ms ≡ [new-configured-list-SecurityInvariant SINVAR-LIB-SubnetsInGW
(|
          node-properties = [''v11'' ↦ Member, ''v12'' ↦ Member, ''v13'' ↦ Member, ''v1b'' ↦
InboundGateway]
        |) ''v1 subnet'',
        new-configured-list-SecurityInvariant SINVAR-LIB-CommunicationPartners (|
        node-properties = [''v1b'' ↦ Master [''v11'', ''v12'', ''v13'', ''v2b'', ''v3b''],
                    ''v11'' ↦ Care,
                    ''v12'' ↦ Care,
                    ''v13'' ↦ Care,
                    ''v2b'' ↦ Care,
                    ''v3b'' ↦ Care
                    ]
        |) ''v1b ACL'',
        new-configured-list-SecurityInvariant SINVAR-LIB-SubnetsInGW (|
          node-properties = [''v21'' ↦ Member, ''v22'' ↦ Member, ''v23'' ↦ Member, ''v2b'' ↦
InboundGateway]
        |) ''v2 subnet'',
        new-configured-list-SecurityInvariant SINVAR-LIB-CommunicationPartners (|
        node-properties = [''v2b'' ↦ Master [''v21'', ''v22'', ''v23'', ''v1b'', ''v3b''],
                    ''v21'' ↦ Care,
                    ''v22'' ↦ Care,
                    ''v23'' ↦ Care,
                    ''v1b'' ↦ Care,
                    ''v3b'' ↦ Care
                    ]
        |) ''v2b ACL'',
        new-configured-list-SecurityInvariant SINVAR-LIB-SubnetsInGW (|          node-properties =
[''v3b'' ↦ InboundGateway]          |) ''v3b'',
        new-configured-list-SecurityInvariant SINVAR-LIB-CommunicationPartners (|
        node-properties = [''v3b'' ↦ Master [''v1b'', ''v2b''],
                    ''v1b'' ↦ Care,
                    ''v2b'' ↦ Care
                    ]
```

⦈ *''v3b ACL''*]
value[*code*] *make-policy SubnetsInGW-ACL-ms subnet-hosts*
lemma *set* (*edgesL* (*make-policy* [*Subnets-m*] *subnet-hosts*)) ⊆ *set* (*edgesL* (*make-policy SubnetsInGW-ACL-ms subnet-hosts*)) **by** *eval*
lemma [*e* <− *edgesL* (*make-policy SubnetsInGW-ACL-ms subnet-hosts*). *e* ∉ *set* (*edgesL* (*make-policy* [*Subnets-m*] *subnet-hosts*))] =
[(*''v1b''*, *''v11''*), (*''v1b''*, *''v12''*), (*''v1b''*, *''v13''*), (*''v2b''*, *''v21''*), (*''v2b''*, *''v22''*), (*''v2b''*, *''v23''*)]
by *eval*
ML-val‹
visualize-graph @{*context*} @{*term SubnetsInGW-ACL-ms*}
@{*term make-policy SubnetsInGW-ACL-ms subnet-hosts*};
›
end

context begin
private definition *secgwext-host-attributes* ≡ [
''hypervisor'' ↦ *PolEnforcePoint*,
''securevm1'' ↦ *DomainMember*,
''securevm2'' ↦ *DomainMember*,
''publicvm1'' ↦ *AccessibleMember*,
''publicvm2'' ↦ *AccessibleMember*
]
private definition *SecGwExt-m* ≡ *new-configured-list-SecurityInvariant SINVAR-LIB-PolEnforcePointExtended* ⦇
node-properties = *secgwext-host-attributes*
⦈ *''secure hypervisor mediates accesses between secure VMs''*
private definition *secgwext-hosts* ≡ [*''hypervisor''*, *''securevm1''*, *''securevm2''*,
''publicvm1'', *''publicvm2''*,
''INET'']

private lemma *dom* (*secgwext-host-attributes*) ⊆ *set* (*secgwext-hosts*)
by(*simp add*: *secgwext-hosts-def secgwext-host-attributes-def*)
value[*code*] *make-policy* [*SecGwExt-m*] *secgwext-hosts*
ML-val‹
visualize-graph-header @{*context*} @{*term* [*SecGwExt-m*]}
@{*term make-policy* [*SecGwExt-m*] *secgwext-hosts*}
@{*term secgwext-host-attributes*};
›

ML-val‹
visualize-graph-header @{*context*} @{*term* [*SecGwExt-m*, *new-configured-list-SecurityInvariant SINVAR-LIB-BLPtrusted* ⦇
node-properties = [*''hypervisor''* ↦ ⦇ *security-level* = *0*, *trusted* = *True* ⦈,
''securevm1'' ↦ ⦇ *security-level* = *1*, *trusted* = *False* ⦈,
''securevm2'' ↦ ⦇ *security-level* = *1*, *trusted* = *False* ⦈
] ⦈ *''secure vms are confidential''*]}
@{*term make-policy* [*SecGwExt-m*, *new-configured-list-SecurityInvariant SINVAR-LIB-BLPtrusted* ⦇
node-properties = [*''hypervisor''* ↦ ⦇ *security-level* = *0*, *trusted* = *True* ⦈,
''securevm1'' ↦ ⦇ *security-level* = *1*, *trusted* = *False* ⦈,
''securevm2'' ↦ ⦇ *security-level* = *1*, *trusted* = *False* ⦈

] ⦈ ''*secure vms are confidential*''] *secgwext-hosts*}
@{*term secgwext-host-attributes*};
›
end

end

15 Example: Imaginary Factory Network

theory *Imaginary-Factory-Network*
imports ../*TopoS-Impl*
begin

In this theory, we give an an example of an imaginary factory network. The example was chosen to show the interplay of several security invariants and to demonstrate their configuration effort.

The specified security invariants deliberately include some minor specification problems. These problems will be used to demonstrate the inner workings of the algorithms and to visualize why some computed results will deviate from the expected results.

The described scenario is an imaginary factory network. It consists of sensors and actuators in a cyber-physical system. The on-site production units of the factory are completely automated and there are no humans in the production area. Sensors are monitoring the building. The production units are two robots (abbreviated bots) which manufacture the actual goods. The robots are controlled by two control systems.

The network consists of the following hosts which are responsible for monitoring the building.

- Statistics: A server which collects, processes, and stores all data from the sensors.
- SensorSink: A device which receives the data from the PresenceSensor, Webcam, TempSensor, and FireSensor. It sends the data to the Statistics server.
- PresenceSensor: A sensor which detects whether a human is in the building.
- Webcam: A camera which monitors the building indoors.
- TempSensor: A sensor which measures the temperature in the building.
- FireSensor: A sensor which detects fire and smoke.

The following hosts are responsible for the production line.

- MissionControl1: An automation device which drives and controls the robots.
- MissionControl2: An automation device which drives and controls the robots. It contains the logic for a secret production step, carried out only by Robot2.
- Watchdog: Regularly checks the health and technical readings of the robots.
- Robot1: Production robot unit 1.
- Robot2: Production robot unit 2. Does a secret production step.

- AdminPc: A human administrator can log into this machine to supervise or troubleshoot the production.

We model one additional special host.

- INET: A symbolic host which represents all hosts which are not part of this network.

The security policy is defined below.

```
definition policy :: string list-graph where
  policy ≡ (| nodesL = [''Statistics'',
                  ''SensorSink'',
                  ''PresenceSensor'',
                  ''Webcam'',
                  ''TempSensor'',
                  ''FireSensor'',
                  ''MissionControl1'',
                  ''MissionControl2'',
                  ''Watchdog'',
                  ''Robot1'',
                  ''Robot2'',
                  ''AdminPc'',
                  ''INET''],
         edgesL = [(''PresenceSensor'', ''SensorSink''),
                  (''Webcam'', ''SensorSink''),
                  (''TempSensor'', ''SensorSink''),
                  (''FireSensor'', ''SensorSink''),
                  (''SensorSink'', ''Statistics''),
                  (''MissionControl1'', ''Robot1''),
                  (''MissionControl1'', ''Robot2''),
                  (''MissionControl2'', ''Robot2''),
                  (''AdminPc'', ''MissionControl2''),
                  (''AdminPc'', ''MissionControl1''),
                  (''Watchdog'', ''Robot1''),
                  (''Watchdog'', ''Robot2'')
                  ] |)
```

lemma *wf-list-graph policy* **by** *eval*

```
ML-val‹
visualize-graph @{context} @{term []::string SecurityInvariant list} @{term policy};
›
```

The idea behind the policy is the following. The sensors on the left can all send their readings in an unidirectional fashion to the sensor sink, which forwards the data to the statistics server. In the production line, on the right, all devices will set up stateful connections. This means, once a connection is established, packet exchange can be bidirectional. This makes sure that the watchdog will receive the health information from the robots, the mission control machines will receive the current state of the robots, and the administrator can actually log into the mission control machines. The policy should only specify who is allowed to set up the connections. We will elaborate on the stateful implementation in `../TopoS_Stateful_Policy.thy` and `../TopoS_Stateful_Policy_Algorithm.thy`.

15.1 Specification of Security Invariants

Several security invariants are specified.

Privacy for employees. The sensors in the building may record any employee. Due to privacy requirements, the sensor readings, processing, and storage of the data is treated with high security levels. The presence sensor does not allow do identify an individual employee, hence produces less critical data, hence has a lower level.

context begin
  private definition *BLP-privacy-host-attributes* ≡ [*''Statistics''* ↦ *3*,
    ''SensorSink'' ↦ *3*,
    ''PresenceSensor'' ↦ *2*, — less critical data
    ''Webcam'' ↦ *3*
    ]
  private lemma *dom* (*BLP-privacy-host-attributes*) ⊆ *set* (*nodesL policy*)
    by(*simp add*: *BLP-privacy-host-attributes-def policy-def*)
  definition *BLP-privacy-m* ≡ *new-configured-list-SecurityInvariant SINVAR-LIB-BLPbasic* ⦇
    node-properties = *BLP-privacy-host-attributes* ⦈ *''confidential sensor data''*
end

Secret corporate knowledge and intellectual property: The production process is a corporate trade secret. The mission control devices have the trade secretes in their program. The important and secret step is done by MissionControl2.

context begin
  private definition *BLP-tradesecrets-host-attributes* ≡ [*''MissionControl1''* ↦ *1*,
    ''MissionControl2'' ↦ *2*,
    ''Robot1'' ↦ *1*,
    ''Robot2'' ↦ *2*
    ]
  private lemma *dom* (*BLP-tradesecrets-host-attributes*) ⊆ *set* (*nodesL policy*)
    by(*simp add*: *BLP-tradesecrets-host-attributes-def policy-def*)
  definition *BLP-tradesecrets-m* ≡ *new-configured-list-SecurityInvariant SINVAR-LIB-BLPbasic* ⦇
    node-properties = *BLP-tradesecrets-host-attributes* ⦈ *''trade secrets''*
end

Note that Invariant 1 and Invariant 2 are two distinct specifications. They specify individual security goals independent of each other. For example, in Invariant 1,*''MissionControl2''* has the security level ⊥ and in Invariant 2, *''PresenceSensor''* has security level ⊥. Consequently, both cannot interact.

Privacy for employees, exporting aggregated data: Monitoring the building while both ensuring privacy of the employees is an important goal for the company. While the presence sensor only collects the single-bit information whether a human is present, the webcam allows to identify individual employees. The data collected by the presence sensor is classified as secret while the data produced by the webcam is top secret. The sensor sink only has the secret security level, hence it is not allowed to process the data generated by the webcam. However, the sensor sink aggregates all data and only distributes a statistical average which does not allow to identify individual employees. It does not store the data over long periods. Therefore, it is marked as trusted and may thus receive the webcam's data. The statistics server, which archives all the data, is considered top secret.

context begin
  private definition *BLP-employee-export-host-attributes* ≡

[''Statistics'' ↦ ⦇ *security-level* = *3*, *trusted* = *False* ⦈,
''SensorSink'' ↦ ⦇ *security-level* = *2*, *trusted* = *True* ⦈,
''PresenceSensor'' ↦ ⦇ *security-level* = *2*, *trusted* = *False* ⦈,
''Webcam'' ↦ ⦇ *security-level* = *3*, *trusted* = *False* ⦈
]

private lemma *dom* (*BLP-employee-export-host-attributes*) ⊆ *set* (*nodesL policy*)
by(*simp add*: *BLP-employee-export-host-attributes-def policy-def*)

definition *BLP-employee-export-m* ≡ *new-configured-list-SecurityInvariant SINVAR-LIB-BLPtrusted* ⦇
node-properties = *BLP-employee-export-host-attributes* ⦈ ''*employee data* (*privacy*)''

end

Who can access bot2? Robot2 carries out a mission-critical production step. It must be made sure that Robot2 only receives packets from Robot1, the two mission control devices and the watchdog.

context begin
private definition *ACL-bot2-host-attributues* ≡
[''Robot2'' ↦ *Master* [''Robot1'',
''MissionControl1'',
''MissionControl2'',
''Watchdog''],
''MissionControl1'' ↦ *Care*,
''MissionControl2'' ↦ *Care*,
''Watchdog'' ↦ *Care*
]

private lemma *dom* (*ACL-bot2-host-attributues*) ⊆ *set* (*nodesL policy*)
by(*simp add*: *ACL-bot2-host-attributues-def policy-def*)

definition *ACL-bot2-m* ≡ *new-configured-list-SecurityInvariant SINVAR-LIB-CommunicationPartners*
⦇*node-properties* = *ACL-bot2-host-attributues* ⦈ ''*Robot2 ACL*''

Note that Robot1 is in the access list of Robot2 but it does not have the *Care* attribute. This means, Robot1 can never access Robot2. A tool could automatically detect such inconsistencies and emit a warning. However, a tool should only emit a warning (not an error) because this setting can be desirable.

In our factory, this setting is currently desirable: Three months ago, Robot1 had an irreparable hardware error and needed to be removed from the production line. When removing Robot1 physically, all its host attributes were also deleted. The access list of Robot2 was not changed. It was planned that Robot1 will be replaced and later will have the same access rights again. A few weeks later, a replacement for Robot1 arrived. The replacement is also called Robot1. The new robot arrived neither configured nor tested for the production. After carefully testing Robot1, Robot1 has been given back the host attributes for the other security invariants. Despite the ACL entry of Robot2, when Robot1 was added to the network, because of its missing *Care* attribute, it was not given automatically access to Robot2. This prevented that Robot1 would accidentally impact Robot2 without being fully configured. In our scenario, once Robot1 will be fully configured, tested, and verified, it will be given the *Care* attribute back.

In general, this design choice of the invariant template prevents that a newly added host may inherit access rights due to stale entries in access lists. At the same time, it does not force administrators to clean up their access lists because a host may only be removed temporarily and wants to be given back its access rights later on. Note that managing access lists scales quadratically in the number of hosts. In contrast, the *Care* attribute can be considered as a Boolean flag which allows to temporarily enable or disable the access rights of a host locally without touching the carefully constructed access lists of other hosts. It also prevents that new hosts which have the name of hosts removed long ago (but where stale access rights were not cleaned up) accidentally inherit their access rights.

end

Hierarchy of fab robots: The production line is designed according to a strict command hierarchy. On top of the hierarchy are control terminals which allow a human operator to intervene and supervise the production process. On the level below, one distinguishes between supervision devices and control devices. The watchdog is a typical supervision device whereas the mission control devices are control devices. Directly below the control devices are the robots. This is the structure that is necessary for the example. However, the company defined a few more sub-departments for future use. The full domain hierarchy tree is visualized below.

Apart from the watchdog, only the following linear part of the tree is used: *''Robots''* ⊑ *''ControlDevices''* ⊑ *''ControlTerminal''*. Because the watchdog is in a different domain, it needs a trust level of 1 to access the robots it is monitoring.

```
context begin
  private definition DomainHierarchy-host-attributes ≡
              [(''MissionControl1'',
                  DN (''ControlTerminal''−−''ControlDevices''−−Leaf, 0)),
               (''MissionControl2'',
                  DN (''ControlTerminal''−−''ControlDevices''−−Leaf, 0)),
               (''Watchdog'',
                  DN (''ControlTerminal''−−''Supervision''−−Leaf, 1)),
               (''Robot1'',
                  DN (''ControlTerminal''−−''ControlDevices''−−''Robots''−−Leaf, 0)),
               (''Robot2'',
                  DN (''ControlTerminal''−−''ControlDevices''−−''Robots''−−Leaf, 0)),
               (''AdminPc'',
                  DN (''ControlTerminal''−−Leaf, 0))
               ]
  private lemma dom (map-of DomainHierarchy-host-attributes) ⊆ set (nodesL policy)
    by(simp add: DomainHierarchy-host-attributes-def policy-def)

  lemma DomainHierarchyNG-sanity-check-config
    (map snd DomainHierarchy-host-attributes)
        (
        Department ''ControlTerminal'' [
          Department ''ControlDevices'' [
            Department ''Robots'' [],
            Department ''OtherStuff'' [],
            Department ''ThirdSubDomain'' []
          ],
          Department ''Supervision'' [
            Department ''S1'' [],
            Department ''S2'' []
          ]
        ]) by eval

  definition Control-hierarchy-m ≡ new-configured-list-SecurityInvariant
```

SINVAR-LIB-DomainHierarchyNG
(| *node-properties = map-of DomainHierarchy-host-attributes* |)
''Production device hierarchy''
end

Sensor Gateway: The sensors should not communicate among each other; all accesses must be mediated by the sensor sink.

context begin
private definition *PolEnforcePoint-host-attributes* ≡
[*''SensorSink''* ↦ *PolEnforcePoint*,
''PresenceSensor'' ↦ *DomainMember*,
''Webcam'' ↦ *DomainMember*,
''TempSensor'' ↦ *DomainMember*,
''FireSensor'' ↦ *DomainMember*
]
private lemma *dom PolEnforcePoint-host-attributes* ⊆ *set* (*nodesL policy*)
by(*simp add*: *PolEnforcePoint-host-attributes-def policy-def*)
definition *PolEnforcePoint-m* ≡ *new-configured-list-SecurityInvariant*
SINVAR-LIB-PolEnforcePointExtended
(| *node-properties = PolEnforcePoint-host-attributes* |)
''sensor slaves''
end

Production Robots are an information sink: The actual control program of the robots is a corporate trade secret. The control commands must not leave the robots. Therefore, they are declared information sinks. In addition, the control command must not leave the mission control devices. However, the two devices could possibly interact to synchronize and they must send their commands to the robots. Therefore, they are labeled as sink pools.

context begin
private definition *SinkRobots-host-attributes* ≡
[*''MissionControl1''* ↦ *SinkPool*,
''MissionControl2'' ↦ *SinkPool*,
''Robot1'' ↦ *Sink*,
''Robot2'' ↦ *Sink*
]
private lemma *dom SinkRobots-host-attributes* ⊆ *set* (*nodesL policy*)
by(*simp add*: *SinkRobots-host-attributes-def policy-def*)
definition *SinkRobots-m* ≡ *new-configured-list-SecurityInvariant*
SINVAR-LIB-Sink
(| *node-properties = SinkRobots-host-attributes* |)
''non−leaking production units''
end

Subnet of the fab: The sensors, including their sink and statistics server are located in their own subnet and must not be accessible from elsewhere. Also, the administrator's PC is in its own subnet. The production units (mission control and robots) are already isolated by the DomainHierarchy and are not added to a subnet explicitly.

context begin
private definition *Subnets-host-attributes* ≡
[*''Statistics''* ↦ *Subnet 1*,
''SensorSink'' ↦ *Subnet 1*,
''PresenceSensor'' ↦ *Subnet 1*,

''Webcam'' ↦ Subnet 1,
''TempSensor'' ↦ Subnet 1,
''FireSensor'' ↦ Subnet 1,
''AdminPc'' ↦ Subnet 4
]

private lemma *dom Subnets-host-attributes* ⊆ *set* (*nodesL policy*)
by(*simp add*: *Subnets-host-attributes-def policy-def*)
definition *Subnets-m* ≡ *new-configured-list-SecurityInvariant*
SINVAR-LIB-Subnets
(| *node-properties* = *Subnets-host-attributes* |)
''network segmentation''
end

Access Gateway for the Statistics server: The statistics server is further protected from external accesses. Another, smaller subnet is defined with the only member being the statistics server. The only way it may be accessed is via that sensor sink.

context begin
private definition *SubnetsInGW-host-attributes* ≡
[''Statistics'' ↦ Member,
''SensorSink'' ↦ InboundGateway
]

private lemma *dom SubnetsInGW-host-attributes* ⊆ *set* (*nodesL policy*)
by(*simp add*: *SubnetsInGW-host-attributes-def policy-def*)
definition *SubnetsInGW-m* ≡ *new-configured-list-SecurityInvariant*
SINVAR-LIB-SubnetsInGW
(| *node-properties* = *SubnetsInGW-host-attributes* |)
''Protectting statistics srv''
end

NonInterference (for the sake of example): The fire sensor is managed by an external company and has a built-in GSM module to call the fire fighters in case of an emergency. This additional, out-of-band connectivity is not modeled. However, the contract defines that the company's administrator must not interfere in any way with the fire sensor.

context begin
private definition *NonInterference-host-attributes* ≡
[''Statistics'' ↦ Unrelated,
''SensorSink'' ↦ Unrelated,
''PresenceSensor'' ↦ Unrelated,
''Webcam'' ↦ Unrelated,
''TempSensor'' ↦ Unrelated,
''FireSensor'' ↦ Interfering, — (!)
''MissionControl1'' ↦ Unrelated,
''MissionControl2'' ↦ Unrelated,
''Watchdog'' ↦ Unrelated,
''Robot1'' ↦ Unrelated,
''Robot2'' ↦ Unrelated,
''AdminPc'' ↦ Interfering, — (!)
''INET'' ↦ Unrelated
]

private lemma *dom NonInterference-host-attributes* ⊆ *set* (*nodesL policy*)
by(*simp add*: *NonInterference-host-attributes-def policy-def*)
definition *NonInterference-m* ≡ *new-configured-list-SecurityInvariant SINVAR-LIB-NonInterference*
(| *node-properties* = *NonInterference-host-attributes* |)

''for the sake of an acdemic example!''
end

As discussed, this invariant is very strict and rather theoretical. It is not ENF-structured and may produce an exponential number of offending flows. Therefore, we exclude it by default from our algorithms.

definition *invariants* ≡ [*BLP-privacy-m*, *BLP-tradesecrets-m*, *BLP-employee-export-m*,
ACL-bot2-m, *Control-hierarchy-m*,
PolEnforcePoint-m, *SinkRobots-m*, *Subnets-m*, *SubnetsInGW-m*]

We have excluded *NonInterference-m* because of its infeasible runtime.

lemma *length invariants = 9* **by** *eval*

15.2 Policy Verification

The given policy fulfills all the specified security invariants. Also with *NonInterference-m*, the policy fulfills all security invariants.

lemma *all-security-requirements-fulfilled* (*NonInterference-m#invariants*) *policy* **by** *eval*
ML‹
visualize-graph @{*context*} @{*term invariants*} @{*term policy*};
›

definition *make-policy* :: (*'a SecurityInvariant*) *list* ⇒ *'a list* ⇒ *'a list-graph* **where**
make-policy sinvars Vs ≡ *generate-valid-topology sinvars* ⦇*nodesL* = *Vs*, *edgesL* = *List.product Vs Vs* ⦈

definition *make-policy-efficient* :: (*'a SecurityInvariant*) *list* ⇒ *'a list* ⇒ *'a list-graph* **where**
make-policy-efficient sinvars Vs ≡ *generate-valid-topology-some sinvars* ⦇*nodesL* = *Vs*, *edgesL* = *List.product Vs Vs* ⦈

The question, "how good are the specified security invariants?" remains. Therefore, we use the algorithm from *make-policy* to generate a policy. Then, we will compare our policy with the automatically generated one. If we exclude the NonInterference invariant from the policy construction, we know that the resulting policy must be maximal. Therefore, the computed policy reflects the view of the specified security invariants. By maximality of the computed policy and monotonicity, we know that our manually specified policy must be a subset of the computed policy. This allows to compare the manually-specified policy to the policy implied by the security invariants: If there are too many flows which are allowed according to the computed policy but which are not in our manually-specified policy, we can conclude that our security invariants are not strict enough.

value[*code*] *make-policy invariants* (*nodesL policy*)
lemma *make-policy invariants* (*nodesL policy*) =
⦇*nodesL* =
[*''Statistics''*, *''SensorSink''*, *''PresenceSensor''*, *''Webcam''*, *''TempSensor''*,
''FireSensor'', *''MissionControl1''*, *''MissionControl2''*, *''Watchdog''*, *''Robot1''*,
''Robot2'', *''AdminPc''*, *''INET''*],
edgesL =
[(*''Statistics''*, *''Statistics''*), (*''SensorSink''*, *''Statistics''*),
(*''SensorSink''*, *''SensorSink''*), (*''SensorSink''*, *''Webcam''*),

(''PresenceSensor'', ''SensorSink''), (''PresenceSensor'', ''PresenceSensor''),
(''Webcam'', ''SensorSink''), (''Webcam'', ''Webcam''),
(''TempSensor'', ''SensorSink''), (''TempSensor'', ''TempSensor''),
(''TempSensor'', ''INET''), (''FireSensor'', ''SensorSink''),
(''FireSensor'', ''FireSensor''), (''FireSensor'', ''INET''),
(''MissionControl1'', ''MissionControl1''),
(''MissionControl1'', ''MissionControl2''), (''MissionControl1'', ''Robot1''),
(''MissionControl1'', ''Robot2''), (''MissionControl2'', ''MissionControl2''),
(''MissionControl2'', ''Robot2''), (''Watchdog'', ''MissionControl1''),
(''Watchdog'', ''MissionControl2''), (''Watchdog'', ''Watchdog''),
(''Watchdog'', ''Robot1''), (''Watchdog'', ''Robot2''), (''Watchdog'', ''INET''),
(''Robot1'', ''Robot1''), (''Robot2'', ''Robot2''), (''AdminPc'', ''MissionControl1''),
(''AdminPc'', ''MissionControl2''), (''AdminPc'', ''Watchdog''),
(''AdminPc'', ''Robot1''), (''AdminPc'', ''AdminPc''), (''AdminPc'', ''INET''),
(''INET'', ''INET'')]⦈ **by** *eval*

Additional flows which would be allowed but which are not in the policy

lemma *set* [*e* ← *edgesL* (*make-policy invariants* (*nodesL policy*)). *e* ∉ *set* (*edgesL policy*)] =
set [(*v*,*v*). *v* ← (*nodesL policy*)] ∪
set [(''SensorSink'', ''Webcam''),
(''TempSensor'', ''INET''),
(''FireSensor'', ''INET''),
(''MissionControl1'', ''MissionControl2''),
(''Watchdog'', ''MissionControl1''),
(''Watchdog'', ''MissionControl2''),
(''Watchdog'', ''INET''),
(''AdminPc'', ''Watchdog''),
(''AdminPc'', ''Robot1''),
(''AdminPc'', ''INET'')] **by** *eval*

We visualize this comparison below. The solid edges correspond to the manually-specified policy. The dotted edges correspond to the flow which would be additionally permitted by the computed policy.

ML-val‹
visualize-edges @{*context*} @{*term edgesL policy*}
[(*edge* [*dir*=*arrow*\, *style*=*dashed*, *color*=\#*FF8822*\, *constraint*=*false*],
@{*term* [*e* ← *edgesL* (*make-policy invariants* (*nodesL policy*)).
e ∉ *set* (*edgesL policy*)]})] ;
›

The comparison reveals that the following flows would be additionally permitted. We will discuss whether this is acceptable or if the additional permission indicates that we probably forgot to specify an additional security goal.

- All reflexive flows, i.e. all host can communicate with themselves. Since each host in the policy corresponds to one physical entity, there is no need to explicitly prohibit or allow in-host communication.
- The ''*SensorSink*'' may access the ''*Webcam*''. Both share the same security level, there is no problem with this possible information flow. Technically, a bi-directional connection may even be desirable, since this allows the sensor sink to influence the video stream, e.g. request a lower bit rate if it is overloaded.

- Both the *''TempSensor''* and the *''FireSensor''* may access the Internet. No security levels or other privacy concerns are specified for them. This may raise the question whether this data is indeed public. It is up to the company to decide that this data should also be considered confidential.

- *''MissionControl1''* can send to *''MissionControl2''*. This may be desirable since it was stated anyway that the two may need to cooperate. Note that the opposite direction is definitely prohibited since the critical and secret production step only known to *''MissionControl2''* must not leak.

- The *''Watchdog''* may access *''MissionControl1''*, *''MissionControl2''*, and the *''INET''*. While it may be acceptable that the watchdog which monitors the robots may also access the control devices, it should raise a concern that the watchdog may freely send data to the Internet. Indeed, the watchdog can access devices which have corporate trade secrets stored but it was never specified that the watchdog should be treated confidentially. Note that in the current setting, the trade secrets will never leave the robots. This is because the policy only specifies a unidirectional information flow from the watchdog to the robots; the robots will not leak any information back to the watchdog. This also means that the watchdog cannot actually monitor the robots. Later, when implementing the scenario, we will see that the simple, hand-waving argument "the watchdog connects to the robots and the robots send back their data over the established connection" will not work because of this possible information leak.

- The *''AdminPc''* is allowed to access the *''Watchdog''*, *''Robot1''*, and the *''INET''*. Since this machine is trusted anyway, the company does not see a problem with this.

without *NonInterference-m*

lemma *all-security-requirements-fulfilled invariants* (*make-policy invariants* (*nodesL policy*)) **by** *eval*

Side note: what if we exclude subnets?

ML-val ‹
visualize-edges @{*context*} @{*term edgesL* (*make-policy invariants* (*nodesL policy*))}
[(*edge* [*dir*=*arrow*\, *style*=*dashed*, *color*=\#*FF8822*\, *constraint*=*false*],
@{*term* ‹[*e* ← *edgesL* (*make-policy* [*BLP-privacy-m*, *BLP-tradesecrets-m*, *BLP-employee-export-m*,
ACL-bot2-m, *Control-hierarchy-m*,
PolEnforcePoint-m, *SinkRobots-m*, ~~*Subnets-m*,~~ *SubnetsInGW-m*] (*nodesL policy*)).
e ∉ *set* (*edgesL* (*make-policy invariants* (*nodesL policy*)))]›})] ;
›

15.3 About NonInterference

The NonInterference template was deliberately selected for our scenario as one of the 'problematic' and rather theoretical invariants. Our framework allows to specify almost arbitrary invariant templates. We concluded that all non-ENF-structured invariants which may produce an exponential number of offending flows are problematic for practical use. This includes "Comm. With" (`../Security_Invariants/SINVAR_ACLcommunicateWith.thy`), "Not Comm. With" (`../Security_Invariants/SINVAR_ACLnotCommunicateWith.thy`), Dependability (`../Security_Invariants/SINVAR_Dependability.thy`), and NonInterference (`../Security_Invariants/SINVAR_NonInterference.thy`). In this section, we discuss the consequences of the NonInterference invariant for automated policy construction. We will conclude that, though we can solve all technical challenges, said invariants are —due to their inherent ambiguity— not very well suited for automated policy construction.

The computed maximum policy does not fulfill invariant 10 (NonInterference). This is because the fire sensor and the administrator's PC may be indirectly connected over the Internet.

lemma ¬ *all-security-requirements-fulfilled* (*NonInterference-m#invariants*) (*make-policy invariants* (*nodesL policy*)) **by** *eval*

Since the NonInterference template may produce an exponential number of offending flows, it is infeasible to try our automated policy construction algorithm with it. We have tried to do so on a machine with 128GB of memory but after a few minutes, the computation ran out of memory. On said machine, we were unable to run our policy construction algorithm with the NonInterference invariant for more that five hosts.
Algorithm *make-policy-efficient* improves the policy construction algorithm. The new algorithm instantly returns a solution for this scenario with a very small memory footprint.

The more efficient algorithm does not need to construct the complete set of offending flows

value[*code*] *make-policy-efficient* (*invariants*@[*NonInterference-m*]) (*nodesL policy*)
value[*code*] *make-policy-efficient* (*NonInterference-m#invariants*) (*nodesL policy*)

lemma *make-policy-efficient* (*invariants*@[*NonInterference-m*]) (*nodesL policy*) =
make-policy-efficient (*NonInterference-m#invariants*) (*nodesL policy*) **by** *eval*

But *NonInterference-m* insists on removing something, which would not be necessary.

lemma *make-policy invariants* (*nodesL policy*) ≠ *make-policy-efficient* (*NonInterference-m#invariants*) (*nodesL policy*) **by** *eval*

lemma *set* (*edgesL* (*make-policy-efficient* (*NonInterference-m#invariants*) (*nodesL policy*)))
⊆
set (*edgesL* (*make-policy invariants* (*nodesL policy*))) **by** *eval*

This is what it wants to be gone.

lemma [*e* ← *edgesL* (*make-policy invariants* (*nodesL policy*)).
e ∉ *set* (*edgesL* (*make-policy-efficient* (*NonInterference-m#invariants*) (*nodesL policy*)))]
=
[(''*AdminPc*'', ''*MissionControl1*''), (''*AdminPc*'', ''*MissionControl2*''),
(''*AdminPc*'', ''*Watchdog*''), (''*AdminPc*'', ''*Robot1*''), (''*AdminPc*'', ''*INET*'')]
by *eval*

lemma [*e* ← *edgesL* (*make-policy invariants* (*nodesL policy*)).
e ∉ *set* (*edgesL* (*make-policy-efficient* (*NonInterference-m#invariants*) (*nodesL policy*)))]
=
[*e* ← *edgesL* (*make-policy invariants* (*nodesL policy*)). *fst e* = ''*AdminPc*'' ∧ *snd e* ≠ ''*AdminPc*'']
by *eval*
ML-val‹
visualize-edges @{*context*} @{*term edgesL policy*}
[(*edge* [*dir*=*arrow*\, *style*=*dashed*, *color*=\#*FF8822*\, *constraint*=*false*],
@{*term* [*e* ← *edgesL* (*make-policy invariants* (*nodesL policy*)).
e ∉ *set* (*edgesL* (*make-policy-efficient* (*NonInterference-m#invariants*) (*nodesL policy*)))]})]
;

›

However, it is an inherent property of the NonInterferance template (and similar templates), that the set of offending flows is not uniquely defined. Consequently, since several solutions are possible, even our new algorithm may not be able to compute one maximum solution. It would be possible to construct some maximal solution, however, this would require to enumerate all offending flows, which is infeasible. Therefore, our algorithm can only return some (valid but probably not maximal) solution for non-END-structured invariants.

As a human, we know the scenario and the intention behind the policy. Probably, the best solution for policy construction with the NonInterferance property would be to restrict outgoing edges from the fire sensor. If we consider the policy above which was constructed without NonInterference, if we cut off the fire sensor from the Internet, we get a valid policy for the NonInterference property. Unfortunately, an algorithm does not have the information of which flows we would like to cut first and the algorithm needs to make some choice. In this example, the algorithm decides to isolate the administrator's PC from the rest of the world. This is also a valid solution. We could change the order of the elements to tell the algorithm which edges we would rather sacrifice than others. This may help but requires some additional input. The author personally prefers to construct only maximum policies with Φ-structured invariants and afterwards fix the policy manually for the remaining non-Φ-structured invariants. Though our new algorithm gives better results and returns instantly, the very nature of invariant templates with an exponential number of offending flows tells that these invariants are problematic for automated policy construction.

15.4 Stateful Implementation

In this section, we will implement the policy and deploy it in a network. As the scenario description stated, all devices in the production line should establish stateful connections which allows – once the connection is established – packets to travel in both directions. This is necessary for the watchdog, the mission control devices, and the administrator's PC to actually perform their task.

We compute a stateful implementation. Below, the stateful implementation is visualized. It consists of the policy as visualized above. In addition, dotted edges visualize where answer packets are permitted.

definition *stateful-policy = generate-valid-stateful-policy-IFSACS policy invariants*
lemma *stateful-policy =*
(|*hostsL = nodesL policy,*
flows-fixL = edgesL policy,
flows-stateL =
[(*''Webcam''*, *''SensorSink''*),
(*''SensorSink''*, *''Statistics''*)]|) **by** *eval*

ML-val‹
visualize-edges @{*context*} @{*term flows-fixL stateful-policy*}
[(*edge* [*dir*=*arrow*\, *style*=*dashed*, *color*=\#*FF8822*\, *constraint*=*false*], @{*term flows-stateL stateful-policy*})] ;
›

As can be seen, only the flows (*''Webcam''*, *''SensorSink''*) and (*''SensorSink''*, *''Statistics''*) are allowed to be stateful. This setup cannot be practically deployed because the watchdog, the mission control devices, and the administrator's PC also need to set up stateful connections. Previous section's discussion already hinted at this problem. The reason why the desired stateful connections are not permitted is due to information leakage. In detail: *BLP-tradesecrets-m* and *SinkRobots-m* are responsible. Both invariants prevent that any data leaves the robots and the mission control devices. To verify this suspicion, the two invariants are removed and the stateful flows are computed again. The result visualized is below.

lemma *generate-valid-stateful-policy-IFSACS policy*
  [*BLP-privacy-m*, *BLP-employee-export-m*,
  ACL-bot2-m, *Control-hierarchy-m*,
  PolEnforcePoint-m, *Subnets-m*, *SubnetsInGW-m*] =
(|*hostsL* = *nodesL policy*,
 flows-fixL = *edgesL policy*,
 flows-stateL =
  [(*''Webcam''*, *''SensorSink''*),
  (*''SensorSink''*, *''Statistics''*),
  (*''MissionControl1''*, *''Robot1''*),
  (*''MissionControl1''*, *''Robot2''*),
  (*''MissionControl2''*, *''Robot2''*),
  (*''AdminPc''*, *''MissionControl2''*),
  (*''AdminPc''*, *''MissionControl1''*),
  (*''Watchdog''*, *''Robot1''*),
  (*''Watchdog''*, *''Robot2''*)]|) **by** *eval*

This stateful policy could be transformed into a fully functional implementation. However, there would be no security invariants specified which protect the trade secrets. Without those two invariants, the invariant specification is too permissive. For example, if we recompute the maximum policy, we can see that the robots and mission control can leak any data to the Internet. Even without the maximum policy, in the stateful policy above, it can be seen that MissionControl1 can exfiltrate information from robot 2, once it establishes a stateful connection.

Without the two invariants, the security goals are way too permissive!

lemma *set* [*e* ← *edgesL* (*make-policy* [*BLP-privacy-m*, *BLP-employee-export-m*,
  ACL-bot2-m, *Control-hierarchy-m*,
  PolEnforcePoint-m, *Subnets-m*, *SubnetsInGW-m*] (*nodesL policy*)). *e* ∉ *set* (*edgesL policy*)] =
  set [(*v*,*v*). *v* ← (*nodesL policy*)] ∪
  set [(*''SensorSink''*, *''Webcam''*),
    (*''TempSensor''*, *''INET''*),
    (*''FireSensor''*, *''INET''*),
    (*''MissionControl1''*, *''MissionControl2''*),
    (*''Watchdog''*, *''MissionControl1''*),
    (*''Watchdog''*, *''MissionControl2''*),
    (*''Watchdog''*, *''INET''*),
    (*''AdminPc''*, *''Watchdog''*),
    (*''AdminPc''*, *''Robot1''*),
    (*''AdminPc''*, *''INET''*)] ∪
  set [(*''MissionControl1''*, *''INET''*),
    (*''MissionControl2''*, *''MissionControl1''*),
    (*''MissionControl2''*, *''Robot1''*),
    (*''MissionControl2''*, *''INET''*),
    (*''Robot1''*, *''INET''*),
    (*''Robot2''*, *''Robot1''*),

(''Robot2'', ''INET'')] **by** *eval*

ML-val‹
visualize-edges @{*context*} @{*term flows-fixL* (*generate-valid-stateful-policy-IFSACS policy* [*BLP-privacy-m*, *BLP-employee-export-m*,
ACL-bot2-m, *Control-hierarchy-m*,
PolEnforcePoint-m, *Subnets-m*, *SubnetsInGW-m*])}
[(*edge* [*dir*=*arrow*\, *style*=*dashed*, *color*=\#*FF8822*\, *constraint*=*false*],
@{*term flows-stateL* (*generate-valid-stateful-policy-IFSACS policy* [*BLP-privacy-m*, *BLP-employee-export-m*,
ACL-bot2-m, *Control-hierarchy-m*,
PolEnforcePoint-m, *Subnets-m*, *SubnetsInGW-m*])})] ;
›

Therefore, the two invariants are not removed but repaired. The goal is to allow the watchdog, administrator's pc, and the mission control devices to set up stateful connections without leaking corporate trade secrets to the outside.

First, we repair *BLP-tradesecrets-m*. On the one hand, the watchdog should be able to send packets both *''Robot1''* and *''Robot2''*. *''Robot1''* has a security level of *1* and *''Robot2''* has a security level of *2*. Consequently, in order to be allowed to send packets to both, *''Watchdog''* must have a security level not higher than *1*. On the other hand, the *''Watchdog''* should be able to receive packets from both. By the same argument, it must have a security level of at least *2*. Consequently, it is impossible to express the desired meaning in the BLP basic template. There are only two solutions to the problem: Either the company installs one watchdog for each security level, or the watchdog must be trusted. We decide for the latter option and upgrade the template to the Bell LaPadula model with trust. We define the watchdog as trusted with a security level of *1*. This means, it can receive packets from and send packets to both robots but it cannot leak information to the outside world. We do the same for the *''AdminPc''*.

Then, we repair *SinkRobots-m*. We realize that the following set set of hosts forms one big pool of devices which must all somehow interact but where information must not leave the pool: The administrator's PC, the mission control devices, the robots, and the watchdog. Therefore, all those devices are configured to be in the same *SinkPool*.

definition *invariants-tuned* ≡ [*BLP-privacy-m*, *BLP-employee-export-m*,
ACL-bot2-m, *Control-hierarchy-m*,
PolEnforcePoint-m, *Subnets-m*, *SubnetsInGW-m*,
new-configured-list-SecurityInvariant SINVAR-LIB-Sink
(| *node-properties* = [*''MissionControl1''* ↦ *SinkPool*,
''MissionControl2'' ↦ *SinkPool*,
''Robot1'' ↦ *SinkPool*,
''Robot2'' ↦ *SinkPool*,
''Watchdog'' ↦ *SinkPool*,
''AdminPc'' ↦ *SinkPool*
] |)
''non−leaking production units'',
new-configured-list-SecurityInvariant SINVAR-LIB-BLPtrusted
(| *node-properties* = [*''MissionControl1''* ↦ (| *security-level* = *1*, *trusted* = *False* |),
''MissionControl2'' ↦ (| *security-level* = *2*, *trusted* = *False* |),
''Robot1'' ↦ (| *security-level* = *1*, *trusted* = *False* |),
''Robot2'' ↦ (| *security-level* = *2*, *trusted* = *False* |),
''Watchdog'' ↦ (| *security-level* = *1*, *trusted* = *True* |),

— trust because *bot2* must send to it. *security-level* 1 to interact with *bot* 1

''AdminPc'' ↦ (| *security-level* = *1*, *trusted* = *True* |)
] |)
''trade secrets''
]

lemma *all-security-requirements-fulfilled invariants-tuned policy* **by** *eval*

definition *stateful-policy-tuned* = *generate-valid-stateful-policy-IFSACS policy invariants-tuned*

The computed stateful policy is visualized below.

lemma *stateful-policy-tuned*
=
(|*hostsL* = *nodesL policy*,
flows-fixL = *edgesL policy*,
flows-stateL =
[(''Webcam'', ''SensorSink''),
(''SensorSink'', ''Statistics''),
(''MissionControl1'', ''Robot1''),
(''MissionControl2'', ''Robot2''),
(''AdminPc'', ''MissionControl2''),
(''AdminPc'', ''MissionControl1''),
(''Watchdog'', ''Robot1''),
(''Watchdog'', ''Robot2'')]|) **by** *eval*

We even get a better (i.e. stricter) maximum policy

lemma *set* (*edgesL* (*make-policy invariants-tuned* (*nodesL policy*))) ⊂
set (*edgesL* (*make-policy invariants* (*nodesL policy*))) **by** *eval*
lemma *set* [*e* ← *edgesL* (*make-policy invariants-tuned* (*nodesL policy*)). *e* ∉ *set* (*edgesL policy*)] =
set [(*v*,*v*). *v* ← (*nodesL policy*)] ∪
set [(''SensorSink'', ''Webcam''),
(''TempSensor'', ''INET''),
(''FireSensor'', ''INET''),
(''MissionControl1'', ''MissionControl2''),
(''Watchdog'', ''MissionControl1''),
(''Watchdog'', ''MissionControl2''),
(''AdminPc'', ''Watchdog''),
(''AdminPc'', ''Robot1'')] **by** *eval*

It can be seen that all connections which should be stateful are now indeed stateful. In addition, it can be seen that MissionControl1 cannot set up a stateful connection to Bot2. This is because MissionControl1 was never declared a trusted device and the confidential information in MissionControl2 and Robot2 must not leak.

The improved invariant definition even produces a better (i.e. stricter) maximum policy.

15.5 Iptables Implementation

firewall – classical use case

ML-val‹

```
(*header*)
writeln (*(echo 1 > /proc/sys/net/ipv4/ip-forward^\n^
        # flush all rules^\n^
        iptables −F^\n^
        #default policy for FORWARD chain:^\n^
        iptables −P FORWARD DROP);*)
        (*filter\n^
         :INPUT ACCEPT [0:0]\n^
         :FORWARD ACCEPT [0:0]\n^
         :OUTPUT ACCEPT [0:0]);

iterate-edges-ML @{context}  @{term flows-fixL stateful-policy-tuned}
 (fn (v1,v2) => writeln (−A FORWARD −i $^v1^-iface −s $^v1^-ipv4 −o $^v2^-iface −d $^v2^-ipv4
−j ACCEPT) )
                    (*(iptables −A FORWARD −i $\\$\\mathit{^v1^\\-iface}$ −s $\\$\\mathit{^
                    v1^\\-ipv4}$ −o $\\$\\mathit{^v2^\\-iface}$ −d $\\$\\mathit{^v2^\\-ipv4}$
−j ACCEPT) )*)
  (fn - => () )
  (fn - => () );

iterate-edges-ML @{context} @{term flows-stateL stateful-policy-tuned}
  (fn (v1,v2) => writeln (−I FORWARD −m state −−state ESTABLISHED −i $^v2^-iface −s $^
                    v2^-ipv4 −o $^v1^-iface −d $^v1^-ipv4 −j ACCEPT) )
                   (*(iptables −I FORWARD −m state −−state ESTABLISHED −i $\\$\\mathit{^
                     v2^\\-iface}$ −s $\\$\\mathit{^v2^\\-ipv4}$ −o $\\$\\mathit{^
                       v1^\\-iface}$ −d $\\$\\mathit{^v1^-ipv4}$ −j ACCEPT # ^v2^ −> ^v1^
(answer)) )*)
  (fn - => () )
  (fn - => () );

writeln COMMIT;
›
```

Using, https://github.com/diekmann/Iptables_Semantics, the iptables ruleset is indeed correct.

end

References

[1] C. Diekmann, L. Hupel, and G. Carle. Directed Security Policies: A Stateful Network Implementation. In J. Pang and Y. Liu, editors, *Engineering Safety and Security Systems*, volume 150 of *Electronic Proceedings in Theoretical Computer Science*, pages 20–34, Singapore, May 2014. Open Publishing Association.

[2] C. Diekmann, A. Korsten, and G. Carle. Demonstrating *topoS*: Theorem-Prover-Based Synthesis of Secure Network Configurations. In *2nd International Workshop on Management of SDN and NFV Systems, manSDN/NFV*, Barcelona, Spain, Nov. 2015.

[3] C. Diekmann, S.-A. Posselt, H. Niedermayer, H. Kinkelin, O. Hanka, and G. Carle. Verifying Security Policies using Host Attributes. In *FORTE – 34th IFIP International Conference on Formal Techniques for Distributed Objects, Components and Systems*, Berlin, Germany, June 2014.