HOL-TestGenFW

Achim D. Brucker   Lukas Brügger   Burkhart Wolff

April 2, 2010

Contents

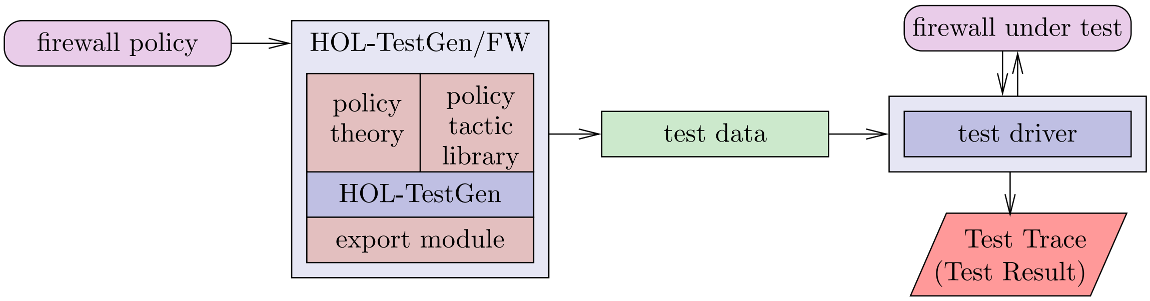

Figure 1: The HOL-TestGen/FW architecture.

1 Introduction

As HOL-TestGen is built on the framework of Isabelle with a general plug-in mechanism, HOL-TestGen can be customized to implement domain-specific, model-based test tools in its own right. As an example for such a domain-specific test-tool, we developed HOL-TestGen/FW which extends HOL-TestGen by:

1. a theory (or library) formalising networks, protocols and firewall policies,
2. domain-specific extensions of the generic test-case procedures (tactics), and
3. support for an export format of test-data for external tools such as [4].

HOL-TestGen/FW is part of the HOL-TestGen distribution. It is located in the directory `add-ons/security`; see [3, 2] for more details.

Figure 1 shows the overall architecture of HOL-TestGen/FW.

In fact, item 1 defines the formal semantics (in HOL) of a specification language for firewall policies; see [2] and the accompanying examples for details. On the technical level, this library also contains simplification rules together with the corresponding setup of the constraint resolution procedures.

With item 2 we refer to domain-specific processing encapsulated into the general HOL-TestGen test-case generation. Since test specifications in our domain have a specific pattern consisting of a limited set of predicates and policy combinators, this can be exploited in specific pre-processing and post-processing of an optimised version of the procedure, now tuned for stateless firewall policies.

With item 3, we refer to an own XML-like format for exchanging test-data for firewalls, i.e. a description of packets to be send together with the expected behavior of the firewall. This data data can be imported in a test-driver for firewalls, for example [4]. This completes our toolchain which, thus, supports the execution of test data on firewall implementations based on test cases derived from formal specifications.

2 Installing and using HOL-TestGen/FW

To install HOL-TestGen/FW you need a working installation of HOL-TestGen as described in the HOL-TestGen User Guide. To build the extension, go into the `add-ons/security/src/firewall/` directory and build the HOL-TestGen/FW heap image for Isabelle by calling

```
isabelle make
```

HOL-TestGen/FW can now be started using the `isabelle` command:

```
isabelle emacs -k HOL-TestGen -l HOL-TestGenFW
```

or, if HOL-TestGen was built on top of HOLCF instead on HOL only:

```
isabelle emacs -k HOLCF-TestGen -l HOLCF-TestGenFW
```

3 Preliminaries

theory
  FWTesting
imports
  PacketFilter / PacketFilter
  FWCompilation / FWCompilationProof
  StatefulFW / StatefulFW
  Testing
begin

This is the formalisation in Isabelle/HOL of firewall policies and corresponding networks and packets. It first contains the formalisation of stateless packet filters as described in [2], followed by a verified policy normalisation technique (described in [1]), and a formalisation of stateful protocols described in [3].

The following statement adjusts the pre-normalization step of the default test case generation algorithm. This turns out to be more efficient for the specific case of firewall policies.

```
setup⟪ map-testgen-params(TestGen.pre-normalizeTNF-tac-update (
            fn ctxt =>
                fn clasimp =>
                        (TestGen.ALLCASES (asm-full-simp-tac (simpset-of
(ThyInfo.get-theory Int))))))
 ⟫
```

Next, the Isar command *prepare-fw-spec* is specified. It can be used to turn test specifications of the form: "$C\ x \Longrightarrow FUT\ x\ =\ policy\ x$" into the desired form for test case generation.

```
ML ⟪
fun prepare-fw-spec-tac ctxt  =
            (TRY((res-inst-tac ctxt [((x,0),x)] spec 1) THEN
             (resolve-tac [allI] 1) THEN
             (split-all-tac 1) THEN
             (TRY (resolve-tac [impI] 1))));
⟫


method-setup prepare-fw-spec =
 ⟪
   Scan.succeed (fn ctxt => SIMPLE-METHOD
      (prepare-fw-spec-tac ctxt))⟫ Prepares the firewall test theorem

end
```

4 Packets and Networks

theory *NetworkCore*
imports *Main*
begin

In networks based e.g. on TCP/IP, a message from A to B is encapsulated in *packets*, which contain the content of the message and routing information. The routing information mainly contains its source and its destination address.

In the case of stateless packet filters, a firewall bases its decision upon this routing information and, in the stateful case, on the content. Thus, we model a packet as a four-tuple of the mentioned elements, together with an id field.

The ID is just an integer:

types $id = int$

To enable different representations of addresses (e.g. IPv4 and IPv6, with or without ports), we model them as an unconstrained type class and directly provide several instances:

axclass $adr < type$
types $'\alpha\ src\ = {'\alpha}::adr$
$'\alpha\ dest = {'\alpha}::adr$

instance $int ::adr$ **..**
instance $nat ::adr$ **..**
instance $fun :: (adr,adr)\ adr$ **..**
instance $* :: (adr,adr)\ adr$ **..**

The content is also specified with an unconstrained generic type:

types $'\beta\ content = {'\beta}$

For applications were the concrete representation of the content field does not matter (usually the case for stateless packet filters), we provide a default type which can be used in those cases:

datatype $DummyContent = data$

A packet is thus:

types $('\alpha,'\beta)\ packet = id \times ({'\alpha}::adr)\ src \times ({'\alpha}::adr)\ dest \times {'\beta}\ content$

Please note that protocols (e.g. http) are not modelled explicitly. In the case of stateless packet filters, they are only visible by the destination port of a packet, which will be modelled as part of the address. Additionally, stateful firewalls will often determine the protocol by the content of a packet which is thus kept as a generic type.

Port numbers (which are part of an address) are also modelled in a generic way. The integers and the naturals are typical representations of port numbers.

axclass $port < type$
instance $int ::port$ **..**
instance $nat :: port$ **..**

A packet therefore has two parameters, the first being the address, the second the content. These should be specified before the test data generation later. For the sake of simplicity, we do not allow to have a different address representation format for the source and the destination of a packet respectively.

In order to access the different parts of a packet directly, we define a couple of projectors:

definition *id* :: $('\alpha,'\beta)$ *packet* $\Rightarrow$ *id*
where *id* $\equiv$ *fst*

definition *src* :: $('\alpha,'\beta)$ *packet* $\Rightarrow$ $('\alpha::adr)$ *src*
where *src* $\equiv$ *fst o snd*

definition *dest* :: $('\alpha,'\beta)$ *packet* $\Rightarrow$ $('\alpha::adr)$ *dest*
where *dest* $\equiv$ *fst o snd o snd*

definition *content* :: $('\alpha,'\beta)$ *packet* $\Rightarrow$ $'\beta$ *content*
where *content* $\equiv$ *snd o snd o snd*

The following two constants give the source and destination port number of a packet. Address representations using port numbers need to provide a definition for these types.

consts *src-port* :: $('\alpha,'\beta)$ *packet* $\Rightarrow$ $'\gamma::port$
consts *dest-port* :: $('\alpha,'\beta)$ *packet* $\Rightarrow$ $'\gamma::port$

A subnetwork (or simply a network) is a set of sets of addresses.

types $'\alpha$ *net* = $'\alpha::adr$ *set set*

The relation in_subnet ($\sqsubset$) checks if an address is in a specific network.

definition
in-subnet :: $'\alpha::adr \Rightarrow {}'\alpha$ *net* $\Rightarrow$ *bool* (**infixl** $\sqsubset$ *100*) **where**
in-subnet a S $\equiv \exists\ s \in S.\ a \in s$

The following lemmas will be useful later.

lemma *in-subnet*:
$(((a),\ e) \sqsubset \{\{((x1),y).\ P\ x1\ y\}\}) = (P\ a\ e)$
by (*simp add*: *in-subnet-def*)

lemma *src-in-subnet*:
$((src(q,((a),e),r,t)) \sqsubset \{\{((x1),y).\ P\ x1\ y\}\}) = (P\ a\ e)$
by (*simp add*: *in-subnet-def in-subnet src-def*)

lemma *dest-in-subnet*:
$((dest\ (q,r,((a),e),t)) \sqsubset \{\{((x1),y).\ P\ x1\ y\}\}) = (P\ a\ e)$
by (*simp add*: *in-subnet-def in-subnet dest-def*)

Address models should provide a definition for the following constant, returning a network consisting of the input address only.

consts *subnet-of* :: $'\alpha::adr \Rightarrow {}'\alpha$ *net*

end

5 Address Representations

theory
NetworkModels
imports

DatatypeAddress
DatatypePort

IntegerAddress
IntegerPort

IPv4

begin

One can think of many different possible address representations. In this distribution, we include 5 different variants:

- DatatypeAddress: Three explicitly named addresses, which build up a network consisting of three disjunct subnetworks. I.e. there are no overlaps and there is no way to distinguish between individual hosts within a network.
- DatatypePort: An address is a pair, with the first element being the same as above, and the second being a port number modelled as an Integer[1].
- IntegerAddress: An address in an Integer.
- IntegerPort: An address is a pair of an Integer and a port (which is again an Integer).
- IPv4: An address is a pair. The first element is a four-tuple of Integers, modelling an IPv4 address, the second element is an Integer denoting the port number.

The respective theories of the networks are relatively small. It suffices to provide the respective types, a couple of lemmas, and - if required - a definition for the source and destination ports of a packet.

end

[1]For technical reasons, we always use Integers instead of Naturals. As a consequence, the test specifications have to be adjusted to eliminate negative numbers.

5.1 Datatype Addresses

theory *DatatypeAddress*
imports *NetworkCore*
begin

A theory describing a network consisting of three subnetworks. Hosts within a network are not distinguished.

datatype *DatatypeAddress* = *dmz-adr* | *intranet-adr* | *internet-adr*

definition
  dmz::*DatatypeAddress net* **where**
  dmz ≡ {{*dmz-adr*}}
definition
  intranet::*DatatypeAddress net* **where**
  intranet ≡ {{*intranet-adr*}}
definition
  internet::*DatatypeAddress net* **where**
  internet ≡ {{*internet-adr*}}

end

5.2 Datatype Addresses with Ports

theory *DatatypePort*
imports *NetworkCore*
begin

A theory describing a network consisting of three subnetworks, including port numbers modelled as Integers. Hosts within a network are not distinguished.

datatype *DatatypeAddress* = *dmz-adr* | *intranet-adr* | *internet-adr*

types
  port = *int*
  DatatypePort = (*DatatypeAddress* × *port*)

instance *DatatypeAddress* :: *adr* **..**

definition
  dmz::*DatatypePort net* **where**
  dmz ≡ {{(*a*,*b*). *a* = *dmz-adr*}}

definition
$intranet$::$DatatypePort$ net **where**
$intranet \equiv \{\{(a,b).\ a = intranet\text{-}adr\}\}$
definition
$internet$::$DatatypePort$ net **where**
$internet \equiv \{\{(a,b).\ a = internet\text{-}adr\}\}$

defs (**overloaded**)
$src\text{-}port\text{-}def$: $src\text{-}port$ $(x::(DatatypePort,'\beta)\ packet) \equiv (snd\ o\ fst\ o\ snd)\ x$
$dest\text{-}port\text{-}def$: $dest\text{-}port$ $(x::(DatatypePort,'\beta)\ packet) \equiv (snd\ o\ fst\ o\ snd\ o\ snd)\ x$
$subnet\text{-}of\text{-}def$: $subnet\text{-}of$ $(x::DatatypePort) \equiv \{\{(a,b).\ a = fst\ x\}\}$

lemma $src\text{-}port$: $src\text{-}port$ $((a,x,d,e)::(DatatypePort,'\beta)\ packet) = snd\ x$
by ($simp$ add: $src\text{-}port\text{-}def$ $in\text{-}subnet$)

lemma $dest\text{-}port$: $dest\text{-}port$ $((a,d,x,e)::(DatatypePort,'\beta)\ packet) = snd\ x$
by ($simp$ add: $dest\text{-}port\text{-}def$ $in\text{-}subnet$)

lemmas *DatatypePortLemmas* = *src-port dest-port src-port-def dest-port-def*
end

5.3 Integer Addresses

theory *IntegerAddress*
imports *NetworkCore*
begin

A theory where addresses are modelled as Integers.

types
IntegerAddress = *int*

end

5.4 Integer Addresses with Ports

theory *IntegerPort*
imports *NetworkCore*
begin

A theory describing addresses which are modelled as a pair of Integers - the first being the host address, the second the port number.

types
  address = *int*
  port = *int*
  IntegerPort = *address* × *port*

defs (**overloaded**)
src-port-def: *src-port* (*x*::(*IntegerPort*,′β) *packet*) ≡ (*snd o fst o snd*) *x*
dest-port-def: *dest-port* (*x*::(*IntegerPort*,′β) *packet*) ≡ (*snd o fst o snd o snd*) *x*
subnet-of-def: *subnet-of* (*x*::(*IntegerPort*)) ≡ {{(*a*,*b*). *a* = *fst x*}}

lemma *src-port*: *src-port* (*a*,*x*::*IntegerPort*,*d*,*e*) = *snd x*
  by (*simp add*: *src-port-def in-subnet*)

lemma *dest-port*: *dest-port* (*a*,*d*,*x*::*IntegerPort*,*e*) = *snd x*
  by (*simp add*: *dest-port-def in-subnet*)

lemmas *IntegerPortLemmas* = *src-port dest-port src-port-def dest-port-def*

end

5.5 IPv4 Addresses

theory *IPv4*
imports *NetworkCore*
begin

A theory describing IPv4 addresses with ports. The host address is a four-tuple of Integers, the port number is a single Integer.

types
  ipv4-ip = (*int* × *int* × *int* × *int*)
  port = *int*
  ipv4 = (*ipv4-ip* × *port*)

defs (**overloaded**)
src-port-def: *src-port* (*x*::(*ipv4*,′β) *packet*) ≡ (*snd o fst o snd*) *x*
defs (**overloaded**)
dest-port-def:*dest-port* (*x*::(*ipv4*,′β) *packet*) ≡ (*snd o fst o snd o snd*) *x*
defs (**overloaded**)
subnet-of-def: *subnet-of* (*x*::*ipv4*) ≡ {{(*a*,*b*). *a* = *fst x*}}

definition *subnet-of-ip* :: *ipv4-ip* ⇒ *ipv4 net*
where *subnet-of-ip ip* ≡ {{(*a*,*b*). (*a* = *ip*)}}

lemma *src-port*: *src-port* $(a,(x::ipv4),d,e) = snd\ x$
by (*simp add*: *src-port-def in-subnet*)

lemma *dest-port*: *dest-port* $(a,d,(x::ipv4),e) = snd\ x$
by (*simp add*: *dest-port-def in-subnet*)

lemmas *IPv4Lemmas* = *src-port dest-port src-port-def dest-port-def*

end

6 Policies

6.1 Policy Core

theory *PolicyCore*
imports *NetworkCore*
begin

Next, we define the concept of a policy. From an abstract point of view, a policy is a partial mapping of packets to decisions. Thus, we model the decision as a datatype.

datatype $'\alpha$ *out* = *accept* $'\alpha$ | *deny* $'\alpha$

A policy is seen as a partial mapping from packet to packet out.

types $('\alpha, '\beta)$ *Policy* = $('\alpha, '\beta)$ *packet* $\rightharpoonup$ $(('\alpha, '\beta)$ *packet*$)$ *out*

When combining several rules, the firewall is supposed to apply the first matching one. In our setting this means the first rule which maps the packet in question to *Some* (*packet out*). This is exactly what happens when using the map-add operator (*rule1* ++ *rule2*). The only difference is that the rules must be given in reverse order.

The constant *p-accept* is *True* iff the policy accepts the packet.

definition
p-accept :: $('\alpha, '\beta)$ *packet* $\Rightarrow$ $('\alpha, '\beta)$ *Policy* $\Rightarrow$ *bool* **where**
p-accept p policy $\equiv$ *policy p* = *Some* (*accept p*)

end

6.2 Policy Combinators

theory *PolicyCombinators*
imports
PolicyCore
begin

In order to ease the specification of a concrete policy, we define some combinators. Using these combinators, the specification of a policy gets very easy, and can be done similarly as in tools like IPTables.

definition
  $allow\text{-}all :: ('\alpha, '\beta)\ Policy$ **where**
  $allow\text{-}all\ p \equiv Some\ (accept\ p)$

definition
  $deny\text{-}all :: ('\alpha,'\beta)\ Policy$ **where**
  $deny\text{-}all\ p \equiv Some\ (deny\ p)$

definition
  $allow\text{-}all\text{-}from :: ('\alpha::adr)\ net \Rightarrow ('\alpha,'\beta)\ Policy$ **where**
  $allow\text{-}all\text{-}from\ src\text{-}net \equiv allow\text{-}all\ |`\ \{pa.\ src\ pa \sqsubset src\text{-}net\}$

definition
  $deny\text{-}all\text{-}from :: ('\alpha::adr)\ net \Rightarrow ('\alpha,'\beta)\ Policy$ **where**
  $deny\text{-}all\text{-}from\ src\text{-}net \equiv deny\text{-}all\ |`\ \{pa.\ src\ pa \sqsubset src\text{-}net\}$

definition
  $allow\text{-}all\text{-}to :: ('\alpha::adr)\ net \Rightarrow ('\alpha,'\beta)\ Policy$ **where**
  $allow\text{-}all\text{-}to\ dest\text{-}net \equiv allow\text{-}all\ |`\ \{pa.\ dest\ pa \sqsubset dest\text{-}net\}$

definition
  $deny\text{-}all\text{-}to :: ('\alpha::adr)\ net \Rightarrow ('\alpha,'\beta)\ Policy$ **where**
  $deny\text{-}all\text{-}to\ dest\text{-}net \equiv deny\text{-}all\ |`\ \{pa.\ dest\ pa \sqsubset dest\text{-}net\}$

definition
  $allow\text{-}all\text{-}from\text{-}to :: ('\alpha::adr)\ net \Rightarrow ('\alpha::adr)\ net \Rightarrow ('\alpha,'\beta)\ Policy$ **where**
  $allow\text{-}all\text{-}from\text{-}to\ src\text{-}net\ dest\text{-}net \equiv allow\text{-}all\ |`\ \{pa.\ src\ pa \sqsubset src\text{-}net \wedge dest\ pa \sqsubset dest\text{-}net\}$

definition
  $deny\text{-}all\text{-}from\text{-}to :: ('\alpha::adr)\ net \Rightarrow ('\alpha::adr)\ net \Rightarrow ('\alpha,'\beta)\ Policy$ **where**
  $deny\text{-}all\text{-}from\text{-}to\ src\text{-}net\ dest\text{-}net \equiv deny\text{-}all\ |`\ \{pa.\ src\ pa \sqsubset src\text{-}net \wedge dest\ pa \sqsubset dest\text{-}net\}$

All these combinators and the default rules are put into one single lemma called *PolicyCombinators* to faciliate proving over policies.

lemmas *PolicyCombinators* =
  allow-all-def deny-all-def allow-all-from-def deny-all-from-def
  allow-all-to-def deny-all-to-def allow-all-from-to-def deny-all-from-to-def

map-add-def restrict-map-def

end

6.3 Policy Combinators with Ports

theory *PortCombinators*
imports *PolicyCombinators*
begin

This theory defines policy combinators for those network models which have ports. They are provided in addition to the the ones defined in the Policy-Combinators theory.

This theory requires from the network models a definition for the two following constants:

- $src\_port :: ('\alpha,'\beta)packet \Rightarrow ('\gamma :: port)$
- $dest\_port :: ('\alpha,'\beta)packet \Rightarrow ('\gamma :: port)$

definition
allow-all-from-port :: $('\alpha::adr)$ *net* $\Rightarrow$ $'\gamma::port$ $\Rightarrow$ $('\alpha, '\beta)$ *Policy* **where**
allow-all-from-port src-net s-port $\equiv$ *allow-all-from src-net* |‘ {*pa. src-port pa* = *s-port*}

definition
deny-all-from-port :: $('\alpha::adr)$ *net* $\Rightarrow$ $'\gamma::port$ $\Rightarrow$ $('\alpha, '\beta)$ *Policy* **where**
deny-all-from-port src-net s-port $\equiv$ *deny-all-from src-net* |‘ {*pa. src-port pa* = *s-port*}

definition
allow-all-to-port :: $('\alpha::adr)$ *net* $\Rightarrow$ $'\gamma::port$ $\Rightarrow$ $('\alpha, '\beta)$ *Policy* **where**
allow-all-to-port dest-net d-port $\equiv$ *allow-all-to dest-net* |‘ {*pa. dest-port pa* = *d-port*}

definition
deny-all-to-port :: $('\alpha::adr)$ *net* $\Rightarrow$ $'\gamma::port$ $\Rightarrow$ $('\alpha, '\beta)$ *Policy* **where**
deny-all-to-port dest-net d-port $\equiv$ *deny-all-to dest-net*|‘ {*pa. dest-port pa* = *d-port*}

definition
allow-all-from-port-to :: $('\alpha::adr)$ *net* $\Rightarrow$ $'\gamma::port$ $\Rightarrow$ $('\alpha::adr)$ *net* $\Rightarrow$ $('\alpha, '\beta)$ *Policy* **where**
allow-all-from-port-to src-net s-port dest-net
$\equiv$ *allow-all-from-to src-net dest-net* |‘ {*pa. src-port pa* = *s-port*}

definition
deny-all-from-port-to :: $('\alpha::adr)$ *net* $\Rightarrow$ $'\gamma::port$ $\Rightarrow$ $('\alpha::adr)$ *net* $\Rightarrow$ $('\alpha,\ '\beta)$ *Policy* **where**
deny-all-from-port-to src-net s-port dest-net
$\equiv$ *deny-all-from-to src-net dest-net*|' $\{pa.\ src\text{-}port\ pa = s\text{-}port\}$

definition
allow-all-from-port-to-port :: $('\alpha::adr)$ *net* $\Rightarrow$ $'\gamma::port$ $\Rightarrow$ $('\alpha::adr)$ *net* $\Rightarrow$ $'\gamma::port$ $\Rightarrow$ $('\alpha,\ '\beta)$ *Policy* **where**
allow-all-from-port-to-port src-net s-port dest-net d-port $\equiv$
allow-all-from-port-to src-net s-port dest-net |' $\{pa.\ dest\text{-}port\ pa = d\text{-}port\}$

definition
deny-all-from-port-to-port :: $('\alpha::adr)$ *net* $\Rightarrow$ $'\gamma::port$ $\Rightarrow$ $('\alpha::adr)$ *net* $\Rightarrow$ $'\gamma::port$ $\Rightarrow$ $('\alpha,\ '\beta)$ *Policy* **where**
deny-all-from-port-to-port src-net s-port dest-net d-port $\equiv$
deny-all-from-port-to src-net s-port dest-net|' $\{pa.\ dest\text{-}port\ pa = d\text{-}port\}$

definition
allow-all-from-to-port :: $('\alpha::adr)$ *net* $\Rightarrow$ $'\gamma::port$ $\Rightarrow$ $('\alpha::adr)$ *net* $\Rightarrow$ $'\gamma::port$ $\Rightarrow$ $('\alpha,\ '\beta)$ *Policy* **where**
allow-all-from-to-port src-net s-port dest-net d-port $\equiv$ *allow-all-from-to src-net dest-net*|'
$\{pa.\ src\text{-}port\ pa = s\text{-}port \wedge dest\text{-}port\ pa = d\text{-}port\}$

definition
deny-all-from-to-port :: $('\alpha::adr)$ *net* $\Rightarrow$ $'\gamma::port$ $\Rightarrow$ $('\alpha::adr)$ *net* $\Rightarrow$ $'\gamma::port$ $\Rightarrow$ $('\alpha,\ '\beta)$ *Policy* **where**
deny-all-from-to-port src-net s-port dest-net d-port $\equiv$ *deny-all-from-to src-net dest-net* |'
$\{pa.\ src\text{-}port\ pa = s\text{-}port \wedge dest\text{-}port\ pa = d\text{-}port\}$

definition
allow-from-port-to :: $'\gamma::port$ $\Rightarrow$ $('\alpha::adr)$ *net* $\Rightarrow$ $('\alpha::adr)$ *net* $\Rightarrow$ $('\alpha,'\beta)$ *Policy* **where**
allow-from-port-to port src-net dest-net $\equiv$ *allow-all* |'
$\{pa.\ src\ pa \sqsubset src\text{-}net \wedge dest\ pa \sqsubset dest\text{-}net \wedge (src\text{-}port\ pa = port)\}$

definition
deny-from-port-to :: $'\gamma::port$ $\Rightarrow$ $('\alpha::adr)$ *net* $\Rightarrow$ $('\alpha::adr)$ *net* $\Rightarrow$ $('\alpha,'\beta)$ *Policy* **where**
deny-from-port-to port src-net dest-net $\equiv$ *deny-all* |'
$\{pa.\ src\ pa \sqsubset src\text{-}net \wedge dest\ pa \sqsubset dest\text{-}net \wedge (src\text{-}port\ pa = port)\}$

definition
allow-from-to-port :: $'\gamma::port$ $\Rightarrow$ $('\alpha::adr)$ *net* $\Rightarrow$ $('\alpha::adr)$ *net* $\Rightarrow$ $('\alpha,'\beta)$ *Policy* **where**
allow-from-to-port port src-net dest-net $\equiv$ *allow-all* |'

$\{pa.\ src\ pa \sqsubset\ src\text{-}net \wedge dest\ pa \sqsubset\ dest\text{-}net \wedge (dest\text{-}port\ pa = port)\}$

definition
deny-from-to-port :: $'\gamma$::*port* ⇒ ($'\alpha$::*adr*) *net* ⇒ ($'\alpha$::*adr*) *net* ⇒ ($'\alpha$,$'\beta$) *Policy*
where
deny-from-to-port port src-net dest-net ≡ *deny-all* |‘
$\{pa.\ src\ pa \sqsubset\ src\text{-}net \wedge dest\ pa \sqsubset\ dest\text{-}net \wedge (dest\text{-}port\ pa = port)\}$

definition
allow-from-ports-to :: $'\gamma$::*port set* ⇒ ($'\alpha$::*adr*) *net* ⇒ ($'\alpha$::*adr*) *net* ⇒ ($'\alpha$,$'\beta$) *Policy* **where**
allow-from-ports-to ports src-net dest-net ≡ *allow-all* |‘
$\{pa.\ src\ pa \sqsubset\ src\text{-}net \wedge dest\ pa \sqsubset\ dest\text{-}net \wedge (src\text{-}port\ pa \in ports)\}$

definition
allow-from-to-ports :: $'\gamma$::*port set* ⇒ ($'\alpha$::*adr*) *net* ⇒ ($'\alpha$::*adr*) *net* ⇒ ($'\alpha$,$'\beta$) *Policy* **where**
allow-from-to-ports ports src-net dest-net ≡ *allow-all* |‘
$\{pa.\ src\ pa \sqsubset\ src\text{-}net \wedge dest\ pa \sqsubset\ dest\text{-}net \wedge (dest\text{-}port\ pa \in ports)\}$

As before, we put all the rules into one lemma called PortCombinators to ease writing later.

lemmas *PortCombinators* =
allow-all-from-port-def deny-all-from-port-def allow-all-to-port-def
deny-all-to-port-def allow-all-from-to-port-def
deny-all-from-to-port-def
allow-from-ports-to-def allow-from-to-ports-def
allow-all-from-port-to-def deny-all-from-port-to-def
allow-from-port-to-def allow-from-to-port-def deny-from-to-port-def
deny-from-port-to-def
end

6.4 Ports

theory *Ports*
imports *Main*
begin

This theory can be used if we want to specify the port numbers by names denoting their default Integer values. If you want to use them, please add *Ports* to the simplifier before test data generation.

definition *http*::*int* **where** *http* ≡ *80*

lemma *http1*: $x \neq 80 \Longrightarrow x \neq http$
by (*simp add*: *http-def*)

lemma *http2*: $x \neq 80 \Longrightarrow http \neq x$
by (*simp add*: *http-def*)

definition *smtp*::*int* **where** $smtp \equiv 25$

lemma *smtp1*: $x \neq 25 \Longrightarrow x \neq smtp$
by (*simp add*: *smtp-def*)

lemma *smtp2*: $x \neq 25 \Longrightarrow smtp \neq x$
by (*simp add*: *smtp-def*)

definition *ftp*::*int* **where** $ftp \equiv 21$

lemma *ftp1*: $x \neq 21 \Longrightarrow x \neq ftp$
by (*simp add*: *ftp-def*)

lemma *ftp2*: $x \neq 21 \Longrightarrow ftp \neq x$
by (*simp add*: *ftp-def*)

And so on for all desired port numbers.

lemmas *Ports* = *http1 http2 ftp1 ftp2 smtp1 smtp2*

end

7 Policy Normalisation

theory
  FWCompilation
imports
  ../PacketFilter/PacketFilter
  Testing
begin

This theory contains all the definitions used for policy normalisation as described in [1].

The normalisation procedure transforms policies into semantically equivalent ones which are "easier" to test. It is organized into nine phases. We impose the following two restrictions on the input policies:

- Each policy must contain a `DenyAll` rule. If this restriction were to be lifted, the `insertDenies` phase would have to be adjusted accordingly.
- For each pair of networks n_1 and n_2, the networks are either disjoint or equal. If this restriction were to be lifted, we would need some additional phases before the start of the normalisation procedure presented below. This rule would split single rules into several by splitting up the networks such that they are all pairwise disjoint or equal. Such a transformation is clearly semantics-preserving and the condition would hold after these phases.

As a result, the procedure generates a list of policies, in which:

- each element of the list contains a policy which completely specifies the blocking behavior between two networks, and
- there are no shadowed rules.

This result is desirable since the test case generation for rules between networks A and B is independent of the rules that specify the behavior for traffic flowing between networks C and D. Thus, the different segments of the policy can be processed individually. The normalization procedure does not aim to minimize the number of rules. While it does remove unnecessary ones, it also adds new ones, enabling a policy to be split into several independent parts.

Policy transformations are functions that map policies to policies. We decided to represent policy transformations as *syntactic rules*; this choice paves the way for expressing the entire normalisation process inside HOL by functions manipulating abstract policy syntax.

7.1 Basics

We define a very simple policy language:

datatype $('\alpha,'\beta)$ *Combinators* =
  DenyAll
| *DenyAllFromTo* $'\alpha$ $'\alpha$
| *AllowPortFromTo* $'\alpha$ $'\alpha$ $'\beta$
| *Conc* $(('\alpha,'\beta)$ *Combinators*$)$ $(('\alpha,'\beta)$ *Combinators*$)$ (**infixr** $\oplus$ *80*)

And define the semantic interpretation of it. For technical reasons, we fix here the type to policies over IntegerPort addresses. However, we could easily provide definitions for other address types as well, using a generic consts for the type definition and a primrec definition for each desired address model.

fun C :: ($IntegerPort\ net$, $port$) $Combinators \Rightarrow$ ($IntegerPort$,$DummyContent$) $Policy$
where
$C\ DenyAll = deny\text{-}all$
$|C\ (DenyAllFromTo\ x\ y) = deny\text{-}all\text{-}from\text{-}to\ x\ y$
$|C\ (AllowPortFromTo\ x\ y\ p) = allow\text{-}from\text{-}to\text{-}port\ p\ x\ y$
$|C\ (x \oplus y) = C\ x\ {+}{+}\ C\ y$

7.2 Auxiliary definitions and functions.

This subsection defines several functions which are useful later for the combinators, invariants, and proofs.

fun $position :: {'}\alpha \Rightarrow {'}\alpha\ list \Rightarrow nat$ **where**
$position\ a\ [] = 0$
$|\ (position\ a\ (x\#xs)) = (if\ a = x\ then\ 1\ else\ (Suc\ (position\ a\ xs)))$

fun $srcNet$ **where**
$srcNet\ (DenyAllFromTo\ x\ y) = x$
$|srcNet\ (AllowPortFromTo\ x\ y\ p) = x$

fun $destNet$ **where**
$destNet\ (DenyAllFromTo\ x\ y) = y$
$|destNet\ (AllowPortFromTo\ x\ y\ p) = y$

fun $srcnets$::($IntegerPort\ net$,$port$) $Combinators \Rightarrow$ ($IntegerPort\ net$) $list$ **where**
$srcnets\ DenyAll = []$
$|srcnets\ (DenyAllFromTo\ x\ y) = [x]$
$|srcnets\ (AllowPortFromTo\ x\ y\ p) = [x]$
$|(srcnets\ (x \oplus y)) = (srcnets\ x)@(srcnets\ y)$

fun $destnets$::($IntegerPort\ net$,$port$) $Combinators \Rightarrow$ ($IntegerPort\ net$) $list$ **where**
$destnets\ DenyAll = []$
$|destnets\ (DenyAllFromTo\ x\ y) = [y]$
$|destnets\ (AllowPortFromTo\ x\ y\ p) = [y]$
$|(destnets\ (x \oplus y)) = (destnets\ x)@(destnets\ y)$

fun ($sequential$) $net\text{-}list\text{-}aux$ **where**
$net\text{-}list\text{-}aux\ [] = []$
$|net\text{-}list\text{-}aux\ (DenyAll\#xs) = net\text{-}list\text{-}aux\ xs$
$|net\text{-}list\text{-}aux\ ((DenyAllFromTo\ x\ y)\#xs) = x\#y\#(net\text{-}list\text{-}aux\ xs)$
$|net\text{-}list\text{-}aux\ ((AllowPortFromTo\ x\ y\ p)\#xs) = x\#y\#(net\text{-}list\text{-}aux\ xs)$
$|net\text{-}list\text{-}aux\ ((x \oplus y)\#xs) = (net\text{-}list\text{-}aux\ [x])@(net\text{-}list\text{-}aux\ [y])@(net\text{-}list\text{-}aux\ xs)$

fun $net\text{-}list$ **where** $net\text{-}list\ p = remdups\ (net\text{-}list\text{-}aux\ p)$

definition $bothNets$ **where** $bothNets\ x = (zip\ (srcnets\ x)\ (destnets\ x))$

fun ($sequential$) $normBothNets$ **where**
$normBothNets\ ((a,b)\#xs) = (if\ ((b,a) \in set\ xs) \vee (a,b) \in set\ (xs)\ then\ (normBothNets$

xs) *else* (*a*,*b*)#(*normBothNets xs*))
|*normBothNets x* = *x*

fun *makeSets* **where**
makeSets ((*a*,*b*)#*xs*) = ({*a*,*b*}#(*makeSets xs*))
|*makeSets* [] = []

fun *bothNet* **where**
bothNet DenyAll = {}
|*bothNet* (*DenyAllFromTo a b*) = {*a*,*b*}
|*bothNet* (*AllowPortFromTo a b p*) = {*a*,*b*}

Nets_List provides from a list of rules a list where the entries are the appearing sets of source and destination network of each rule.

definition *Nets-List* **where** *Nets-List x* = *makeSets* (*normBothNets* (*bothNets x*))

fun (*sequential*) *first-srcNet* **where**
first-srcNet (*x*⊕*y*) = *first-srcNet x*
| *first-srcNet x* = *srcNet x*

fun (*sequential*) *first-destNet* **where**
first-destNet (*x*⊕*y*) = *first-destNet x*
| *first-destNet x* = *destNet x*

fun (*sequential*) *first-bothNet* **where**
first-bothNet (*x*⊕*y*) = *first-bothNet x*
|*first-bothNet x* = *bothNet x*

fun (*sequential*) *in-list* **where**
in-list DenyAll l = *True*
|*in-list x l* = (*bothNet x* ∈ *set l*)

fun *all-in-list* **where**
all-in-list [] *l* = *True*
|*all-in-list* (*x*#*xs*) *l* = (*in-list x l* ∧ *all-in-list xs l*)

fun (*sequential*) *member* **where**
member a (*x*⊕*xs*) = ((*member a x*) ∨ (*member a xs*))
|*member a x* = (*a* = *x*)

fun *noneMT* **where**
noneMT (*x*#*xs*) = (*dom* (*C x*) ≠ {} ∧ (*noneMT xs*))
|*noneMT* [] = *True*

fun *notMTpolicy* **where**
notMTpolicy (*x*#*xs*) = (*if* (*dom* (*C x*) = {}) *then* (*notMTpolicy xs*) *else True*)
|*notMTpolicy* [] = *False*

fun *sdnets* **where**

$sdnets\ DenyAll = \{\}$
$|\ sdnets\ (DenyAllFromTo\ a\ b) = \{(a,b)\}$
$|\ sdnets\ (AllowPortFromTo\ a\ b\ c) = \{(a,b)\}$
$|\ sdnets\ (a \oplus b) = sdnets\ a \cup sdnets\ b$

definition *packet-Nets* **where** $packet\text{-}Nets\ x\ a\ b \equiv (src\ x \sqsubset a \wedge dest\ x \sqsubset b) \vee (src\ x \sqsubset b \wedge dest\ x \sqsubset a)$

fun *matching-rule-rev* **where**
$matching\text{-}rule\text{-}rev\ a\ (x\#xs) = (if\ a \in dom\ (C\,x)\ then\ (Some\,x)\ else\ (matching\text{-}rule\text{-}rev\ a\ xs))$
$|matching\text{-}rule\text{-}rev\ a\ [] = None$

Provides the first matching rule of a policy given as a list of rules.

definition *matching-rule* **where**
$matching\text{-}rule\ a\ x \equiv (matching\text{-}rule\text{-}rev\ a\ (rev\ x))$

definition *subnetsOfAdr* **where** $subnetsOfAdr\ a \equiv \{x.\ a \sqsubset x\}$

definition *fst-set* **where** $fst\text{-}set\ s \equiv \{a.\ \exists\ b.\ (a,b) \in s\}$

definition *snd-set* **where** $snd\text{-}set\ s \equiv \{a.\ \exists\ b.\ (b,a) \in s\}$

fun *memberP* **where**
$memberP\ r\ (x\#xs) = (member\ r\ x \vee memberP\ r\ xs)$
$|memberP\ r\ [] = False$

fun *firstList* **where**
$firstList\ (x\#xs) = (first\text{-}bothNet\ x)$
$|firstList\ [] = \{\}$

7.3 Invariants

If there is a DenyAll, it is at the first position

fun *wellformed-policy1*:: $((IntegerPort\ net,\ port)\ Combinators)\ list \Rightarrow bool$ **where**

$wellformed\text{-}policy1\ [] = True$
$|\ wellformed\text{-}policy1\ (x\#xs) = (DenyAll \notin (set\ xs))$

There is a DenyAll at the first position

fun *wellformed-policy1-strong*:: $((IntegerPort\ net,\ port)\ Combinators)\ list \Rightarrow bool$ **where**
$wellformed\text{-}policy1\text{-}strong\ [] = False$
$|\ wellformed\text{-}policy1\text{-}strong\ (x\#xs) = (x{=}DenyAll \wedge (DenyAll \notin (set\ xs)))$

All rules appearing at the left of a DenyAllFromTo, have disjunct domains from it (except DenyAll)

fun (*sequential*) *wellformed-policy2* **where**

$wellformed\text{-}policy2\ [] = True$
$|\ wellformed\text{-}policy2\ (DenyAll\#xs) = wellformed\text{-}policy2\ xs$
$|\ wellformed\text{-}policy2\ (x\#xs) = ((\forall\ c\ a\ b.\ c = DenyAllFromTo\ a\ b \wedge c \in set\ xs \longrightarrow Map.dom\ (C\ x) \cap Map.dom\ (C\ c) = \{\}) \wedge wellformed\text{-}policy2\ xs)$

An allow rule is disjunct with all rules appearing at the right of it. This invariant is not necessary as it is a consequence from others, but facilitates some proofs.

fun *(sequential) wellformed-policy3* **where**
$wellformed\text{-}policy3\ [] = True$
$|\ wellformed\text{-}policy3\ ((AllowPortFromTo\ a\ b\ p)\#xs) = ((\forall\ r.\ r \in set\ xs \longrightarrow dom\ (C\ r) \cap dom\ (C\ (AllowPortFromTo\ a\ b\ p)) = \{\}) \wedge wellformed\text{-}policy3\ xs)$
$|\ wellformed\text{-}policy3\ (x\#xs) = wellformed\text{-}policy3\ xs$

All two networks are either disjoint or equal.

definition *netsDistinct* **where** $netsDistinct\ a\ b \equiv \neg\ (\exists\ x.\ x \sqsubset a \wedge x \sqsubset b)$

definition *twoNetsDistinct* **where** $twoNetsDistinct\ a\ b\ c\ d \equiv netsDistinct\ a\ c \vee netsDistinct\ b\ d$

definition *allNetsDistinct* **where** $allNetsDistinct\ p \equiv \forall\ a\ b.\ (a \neq b \wedge a \in set\ (net\text{-}list\ p) \wedge b \in set\ (net\text{-}list\ p)) \longrightarrow netsDistinct\ a\ b$

definition *disjSD-2* **where**
$disjSD\text{-}2\ x\ y \equiv \forall\ a\ b\ c\ d.\ ((a,b) \in sdnets\ x \wedge (c,d) \in sdnets\ y \longrightarrow (twoNetsDistinct\ a\ b\ c\ d \wedge twoNetsDistinct\ a\ b\ d\ c))$

The policy is given as a list of single rules.

fun *singleCombinators* **where**
$singleCombinators\ [] = True$
$|singleCombinators\ ((x \oplus y)\#xs) = False$
$|singleCombinators\ (x\#xs) = singleCombinators\ xs$

definition *onlyTwoNets* **where**
$onlyTwoNets\ x \equiv ((\exists\ a\ b.\ (sdnets\ x = \{(a,b)\})) \vee (\exists\ a\ b.\ sdnets\ x = \{(a,b),(b,a)\}))$

Each entry of the list contains rules between two networks only.

fun *OnlyTwoNets* **where**
$OnlyTwoNets\ (DenyAll\#xs) = OnlyTwoNets\ xs$
$|OnlyTwoNets\ (x\#xs) = (onlyTwoNets\ x \wedge OnlyTwoNets\ xs)$
$|OnlyTwoNets\ [] = True$

fun *noDenyAll* **where**
$noDenyAll\ (x\#xs) = ((\neg\ member\ DenyAll\ x) \wedge noDenyAll\ xs)$
$|noDenyAll\ [] = True$

fun *noDenyAll1* **where**
$noDenyAll1\ (DenyAll\#xs) = noDenyAll\ xs$

| *noDenyAll1 xs = noDenyAll xs*

fun *separated* **where**
separated (x#xs) = ((∀ s. s ∈ set xs ⟶ disjSD-2 x s) ∧ separated xs)
| *separated [] = True*

fun *NetsCollected* **where**
NetsCollected (x#xs) = (((first-bothNet x ≠ firstList xs) ⟶ (∀ a∈set xs. first-bothNet x ≠ first-bothNet a)) ∧ NetsCollected (xs))
| *NetsCollected [] = True*

fun *NetsCollected2* **where**
NetsCollected2 (x#xs) = (xs = [] ∨ (first-bothNet x ≠ firstList xs ∧ NetsCollected2 xs))
|*NetsCollected2 [] = True*

7.4 Transformations

The following two functions transform a policy into a list of single rules and vice-versa.

fun *policy2list*::(*IntegerPort net*, *port*) *Combinators* ⇒ ((*IntegerPort net*, *port*) *Combinators*) *list* **where**
policy2list (x ⊕ y) = (concat [(policy2list x),(policy2list y)])
|*policy2list x = [x]*

fun *list2policy*::((*IntegerPort net*, *port*) *Combinators*) *list* ⇒ ((*IntegerPort net*, *port*) *Combinators*) **where**
list2policy (x#[]) = x
| *list2policy (x#y) = x ⊕ (list2policy y)*

Remove all the rules appearing before a DenyAll. There are two alternative versions.

fun *removeShadowRules1* **where**
removeShadowRules1 (x#xs) = (if (DenyAll ∈ set xs) then ((removeShadowRules1 xs)) else x#xs)
| *removeShadowRules1 [] = []*

fun *removeShadowRules1-alternative-rev* **where**
removeShadowRules1-alternative-rev [] = []
| *removeShadowRules1-alternative-rev (DenyAll#xs) = [DenyAll]*
| *removeShadowRules1-alternative-rev [x] = [x]*
| *removeShadowRules1-alternative-rev (x#xs)= x#(removeShadowRules1-alternative-rev xs)*

definition *removeShadowRules1-alternative* **where** *removeShadowRules1-alternative p = rev (removeShadowRules1-alternative-rev (rev p))*

Remove all the rules which allow a port, but are shadowed by a deny between these subnets

fun *removeShadowRules2*::
$((IntegerPort\ net, port)\ Combinators)\ list \Rightarrow ((IntegerPort\ net, port)\ Combinators)\ list$
where
$(removeShadowRules2\ ((AllowPortFromTo\ x\ y\ p)\#z)) =$
$(if\ (((DenyAllFromTo\ x\ y) \in set\ z))\ then\ ((removeShadowRules2\ z))\ else\ (((AllowPortFromTo\ x\ y\ p)\#(removeShadowRules2\ z))))$
$|\ removeShadowRules2\ (x\#y) = x\#(removeShadowRules2\ y)$
$|\ removeShadowRules2\ [] = []$

Sorting a pocliy. We first need to define an ordering on rules. This ordering depends on the $Nets\_List$ of a policy.

fun *smaller* :: $(IntegerPort\ net, port)\ Combinators \Rightarrow$
$(IntegerPort\ net, port)\ Combinators \Rightarrow$
$((IntegerPort\ net)\ set)\ list \Rightarrow bool$
where
$smaller\ DenyAll\ x\ l = True$
$|\ smaller\ x\ DenyAll\ l = False$
$|\ smaller\ x\ y\ l =$
$((x = y) \vee \quad (if\ (bothNet\ x) = (bothNet\ y)\ then$
$(case\ y\ of\ (DenyAllFromTo\ a\ b) \Rightarrow (x = DenyAllFromTo\ b\ a)$
$|\ \_ \Rightarrow\ True)$
$else$
$(position\ (bothNet\ x)\ l <= position\ (bothNet\ y)\ l)))$

We use insertion sort for sorting a policy.

fun *insort* **where**
$insort\ a\ []\ l = [a]$
$|\ insort\ a\ (x\#xs)\ l = (if\ (smaller\ a\ x\ l)\ then\ a\#x\#xs\ else\ x\#(insort\ a\ xs\ l))$

fun *sort* **where**
$sort\ []\ l = []$
$|\ sort\ (x\#xs)\ l = insort\ x\ (sort\ xs\ l)\ l$

fun *sorted* **where**
$sorted\ []\ l \longleftrightarrow True\ |$
$sorted\ [x]\ l \longleftrightarrow True\ |$
$sorted\ (x\#y\#zs)\ l \longleftrightarrow smaller\ x\ y\ l \wedge sorted\ (y\#zs)\ l$

separate works on a sorted policy: it joins the rules which talk about the traffic between the same two networks.

fun *separate* **where**
$separate\ (DenyAll\#x) = DenyAll\#(separate\ x)$
$|\ separate\ (x\#y\#z) = (if\ (first\text{-}bothNet\ x = first\text{-}bothNet\ y)$
$then\ (separate\ ((x \oplus y)\#z))$
$else\ (x\#(separate(y\#z))))$
$|separate\ x = x$

Insert the DenyAllFromTo rules, such that traffic between two networks can be tested individually

fun *insertDenies* **where**
insertDenies $(x\#xs) = (case\ x\ of\ DenyAll \Rightarrow (DenyAll\#(insertDenies\ xs))$
$|\ \text{-} \Rightarrow (DenyAllFromTo\ (first\text{-}srcNet\ x)\ (first\text{-}destNet\ x) \oplus$
$(DenyAllFromTo\ (first\text{-}destNet\ x)\ (first\text{-}srcNet\ x)) \oplus$
$x)\#$
$(insertDenies\ xs))$
$|\ insertDenies\ [] = []$

Remove duplicate rules. This is especially necessary as insertDenies might have inserted duplicate rules.

The second function is supposed to work on a list of policies. Only rules which are duplicated within the same policy are removed.

fun *removeDuplicates* **where**
$removeDuplicates\ (x \oplus xs) = (if\ member\ x\ xs\ then\ (removeDuplicates\ xs)\ else\ x \oplus (removeDuplicates\ xs))$
$|\ removeDuplicates\ x = x$

fun *removeAllDuplicates* **where**
$removeAllDuplicates\ (x\#xs) = ((removeDuplicates\ (x))\#(removeAllDuplicates\ xs))$
$|removeAllDuplicates\ x = x$

Remove rules with an empty domain - they never match any packet.

fun *removeShadowRules3* **where**
$removeShadowRules3\ (x\#xs) = (if\ (dom\ (C\ x) = \{\})\ then\ (removeShadowRules3\ xs)\ else\ (x\#(removeShadowRules3\ xs)))$
$|removeShadowRules3\ [] = []$

Insert a DenyAll at the beginning of a policy.

fun *insertDeny* **where**
$insertDeny\ (DenyAll\#xs) = DenyAll\#xs$
$|insertDeny\ xs = DenyAll\#xs$

Now do everything:

definition $sort'\ p\ l \equiv sort\ l\ p$

definition $normalize'\ p \equiv (removeAllDuplicates\ o\ insertDenies\ o\ separate\ o\ (sort'\ (Nets\text{-}List\ p))\ o\ removeShadowRules2\ o\ remdups\ o\ removeShadowRules3\ o\ insertDeny\ o\ removeShadowRules1\ o\ policy2list)\ p$

definition $normalize\ p \equiv removeAllDuplicates\ (insertDenies\ (separate\ (sort\ (removeShadowRules2\ (remdups\ (removeShadowRules3\ (insertDeny\ (removeShadowRules1\ (policy2list\ p)))))) ((Nets\text{-}List\ p)))))$

definition $normalize\text{-}manual\text{-}order\ p\ l \equiv removeAllDuplicates\ (insertDenies\ (separate\ (sort\ (removeShadowRules2\ (remdups\ (removeShadowRules3\ (insertDeny\ (removeShadowRules1\ (policy2list\ p)))))) ((l)))))$

Of course, normalize is equal to normalize', the latter looks nicer though.

lemma *normalize* = *normalize'*
by (*rule ext*, *simp add*: *normalize-def normalize'-def sort'-def*)

The following definition helps in creating the test specification for the individual parts of a normalized policy.

definition *makeFUT* **where** *makeFUT FUT p x n* =
(*packet-Nets x* (*fst*(((*normBothNets* (*bothNets p*)))!*n*)) (*snd*(((*normBothNets* (*bothNets p*)))!*n*)) ⟶ *FUT x* = *C* ((*normalize p*)!(*n*+*1*)) *x*)

declare *C.simps* [*simp del*]

lemmas *PLemmas* = *C.simps dom-def PolicyCombinators.PolicyCombinators PortCombinators.PortCombinators src-def dest-def in-subnet-def IntegerPort.src-port-def IntegerPort.dest-port-def*

end

8 Stateful Firewalls

8.1 Basic Constructs

theory *Stateful*
imports ../*PacketFilter*/*PacketFilter Testing*
begin

The simple system of a stateless packet filter is not enough to model all common real-world scenarios. Some protocols need further actions in order to be secured. A prominent example is the File Transfer Protocol (FTP), which is a popular means to move files across the Internet. It behaves quite differently from most other application layer protocols as it uses a two-way connection establishment which opens a dynamic port. A stateless packet filter would only have the possibility to either always open all the possible dynamic ports or not to allow that protocol at all. Neither of these options is satisfactory. In the first case, all ports above 1024 would have to be opened which introduces a big security hole in the system, in the second case users wouldn't be very happy. A firewall which tracks the state of the TCP connections on a system doesn't help here either, as the opening and closing of the ports takes place on the application layer. Therefore, a firewall needs to have some knowledge of the application protocols being run and track the states of these protocols. We next model this behaviour.

The key point of our model is the idea that a policy remains the same as before: a mapping from packet to packet out. We still specify for every packet, based on its source and destination address, the expected action. The only thing that changes now is that this mapping is allowed to change over time. This indicates that our test data will not consist of single packets but rather of sequences thereof.

At first we hence need a state. It is a tupel from some memory to be refined later and the current policy.

types $('\alpha,'\beta,'\gamma)$ *FWState* $= '\alpha \times ('\beta,'\gamma)$ *Policy*

Having a state, we need of course some state transitions. Such a transition can happen every time a new packet arrives. State transitions can be modelled using a state-exception monad.

types $('\alpha,'\beta,'\gamma)$ *FWStateTransition* $= ('\beta,'\gamma)$ *packet* $\Rightarrow$ (*unit*, $('\alpha,'\beta,'\gamma)$ *FWState*) *MON-SE*

The memory could be modelled as a list of accepted packets.

types $('\beta,'\gamma)$ *history* $= ('\beta,'\gamma)$ *packet list*

The next two constants will help us later in defining the state transitions. The constant *before* is *True* if for all elements which appear before the first element for which q holds, p must hold.

consts *before* :: $('\alpha \Rightarrow bool) \Rightarrow ('\alpha \Rightarrow bool) \Rightarrow '\alpha\ list \Rightarrow bool$
primrec
  $before\ p\ q\ [] = False$
  $before\ p\ q\ (a\ \#\ S) = (q\ a \vee (p\ a \wedge (before\ p\ q\ S)))$

Analogously there is an operator *not-before* which returns *True* if for all elements which appear before the first element for which q holds, p must not hold.

consts *not-before* :: $('\alpha \Rightarrow bool) \Rightarrow ('\alpha \Rightarrow bool) \Rightarrow '\alpha\ list \Rightarrow bool$
primrec
  $not\text{-}before\ p\ q\ [] = False$
  $not\text{-}before\ p\ q\ (a\ \#\ S) = (q\ a \vee (\neg\ (p\ a) \wedge (not\text{-}before\ p\ q\ S)))$

The next two operators can be used to combine state transitions. It takes the first transition which maps to *Some* $'\alpha$.

definition *orelse*:: $('\alpha,'\beta,'\gamma)$ *FWStateTransition* $\Rightarrow ('\alpha,'\beta,'\gamma)$ *FWStateTransition* $\Rightarrow ('\alpha,'\beta,'\gamma)$ *FWStateTransition* (**infixl** *orelse 100*) **where**
$(f\ orelse\ g)\ x \equiv \lambda\ \sigma.\ (case\ f\ x\ \sigma\ of\ None \Rightarrow g\ x\ \sigma\ |\ Some\ y \Rightarrow Some\ y)$

end

8.2 FTP Protocol

theory *FTP*
imports
  Stateful
begin

The File Transfer Protocol FTP is a well known example of a protocol which uses dynamic ports and is therefore a natural choice to use as an example for our model.

We model only a simplified version of the FTP protocol over IntegerPort addresses, still containing all messages that matter for our purposes. It consists of the following four messages:

1. *ftp-init*: The client contacts the server indicating his wish to get some data.
2. *ftp-port-request p*: The client, usually after having received an acknowledgement of the server, indicates a port number on which he wants to receive the data.
3. *ftp-data*: The server sends the requested data over the new channel. There might be an arbitrary number of such messages, including zero.
4. *ftp-close*: The client closes the connection. The dynamic port gets closed again.

The content field of a packet therefore now consists of either one of those four messages or a default one.

datatype *ftp-msg* = *ftp-init*
  | *ftp-port-request port*
  | *ftp-data*
  | *ftp-close*
  | *other*

We now also make use of the ID field of a packet. It is used as session ID and we make the assumption that they are all unique among different protocol runs.

At first, we need some predicates which check if a packet is a specific FTP message and has the correct session ID.

definition
  is-init :: *id* ⇒ (*IntegerPort*, *ftp-msg*) *packet* ⇒ *bool* **where**
  is-init i p ≡ *id p* = *i* ∧ *content p* = *ftp-init*

definition
is-port-request :: *id* ⇒ *port* ⇒(*IntegerPort*, *ftp-msg*) *packet* ⇒ *bool* **where**
is-port-request i port p ≡ *id p* = *i* ∧ *content p* = *ftp-port-request port*

definition
is-data :: *id* ⇒ (*IntegerPort*, *ftp-msg*) *packet* ⇒ *bool* **where**
is-data i p ≡ *id p* = *i* ∧ *content p* = *ftp-data*

definition
is-close :: *id* ⇒ (*IntegerPort*, *ftp-msg*) *packet* ⇒ *bool* **where**
is-close i p ≡ *id p* = *i* ∧ *content p* = *ftp-close*

definition
port-open :: (*IntegerPort*, *ftp-msg*) *history* ⇒ *id* ⇒ *port* ⇒ *bool* **where**
port-open L a p ≡ *not-before* (*is-close a*) (*is-port-request a p*) *L*

We now have to model the respective state transitions. It is important to note that state transitions themselves allow all packets which are allowed by the policy, not only those which are allowed by the protocol. Their only task is to change the policy. As an alternative, we could have decided that they only allow packets which follow the protocol (e.g. come on the correct ports), but this should in our view rather be reflected in the policy itself.

Of course, not every message changes the policy. In such cases, we do not have to model different cases, one is enough. In our example, only messages 2 and 4 need special transitions. The default says that if the policy accepts the packet, it is added to the history, otherwise it is simply dropped. The policy remains the same in both cases.

fun *FTP-ST* :: ((*IntegerPort*,*ftp-msg*) *history*, *IntegerPort*, *ftp-msg*) *FWStateTransition*
where

FTP-ST (*i*,*s*,*d*,*ftp-port-request pr*) (*InL*, *policy*) = (*if p-accept* (*i*,*s*,*d*,*ftp-port-request pr*) *policy then*
(*if not-before* (*is-close i*) (*is-init i*) *InL* ∧ *dest-port* (*i*,*s*,*d*,*ftp-port-request pr*) = (*21*::*port*) *then*
Some ((),((*i*,*s*,*d*,*ftp-port-request pr*)#*InL*, *policy* ++
(*allow-from-to-port pr* (*subnet-of d*) (*subnet-of s*))))
else Some ((),((*i*,*s*,*d*,*ftp-port-request pr*)#*InL*,*policy*)))
else Some ((),(*InL*,*policy*)))

|*FTP-ST* (*i*,*s*,*d*,*ftp-close*) (*InL*,*policy*) =
(*if* (*p-accept* (*i*,*s*,*d*,*ftp-close*) *policy*) *then*
(*if* (∃ *p*. *port-open InL i p*) ∧ *dest-port* (*i*,*s*,*d*,*ftp-close*) = (*21*::*port*) *then*
Some((),((*i*,*s*,*d*,*ftp-close*)#*InL*, *policy* ++
deny-from-to-port (*Eps* (λ *p*. *port-open InL i p*)) (*subnet-of d*) (*subnet-of s*)))

$\quad\quad\quad\quad$ *else Some* ((),((*i*,*s*,*d*,*ftp-close*)#*InL*, *policy*)))
$\quad\quad$ *else Some* ((),(*InL*,*policy*)))

|*FTP-ST p* (*InL*,*policy*) = (*if p-accept p policy then*
$\quad\quad\quad\quad\quad\quad\quad\quad$ *Some* ((),(*p*#*InL*,*policy*))
$\quad\quad\quad\quad\quad\quad\quad$ *else*
$\quad\quad\quad\quad\quad\quad\quad\quad$ *Some* ((),(*InL*,*policy*)))

The second message of the protocol is the port request. If the packet is allowed by the policy, and iff there is an opened but not yet closed FTP-Session with the same session ID, we change the policy such that the requested port is opened. If the policy allows the packet but there is no open protocol run, we do allow the packet but do not open the requested port.

In the last message, we need to close a port which we do not know directly. It has only been specified in a preceding port_request message. Therefore a predicate is needed which checks if there is an open protocol run with an opened port. This transition is the trickiest one. We need to close the port wich has been opened but not yet closed by a packet with the same session ID. Here we use the assumption that they are supposed to be unique. This transition introduces some kind of inconsistency. If the port that was requested was already open to start with, it gets closed here. The tester should be aware of this fact.

This transition has also some other consequences. The Hilbert epsilon operator *Eps*, also written as *SOME*, returns an arbitrary object for which the following predicate is *True* and is undefined otherwise. We use it to get the number of the port which we want to close. With the if-condition it is assured that such a port exists, but we might have problems if there are several of them. However, due to our assumption that the session IDs are unique, there won't be a problem as long as we do not open several ports in one single protocol run. This should not occur by the definition of the protocol, but if it does, which might happen if we want to test illegal protocol runs, some proof work might be needed.

Now we specify our test scenario in more detail. We could test:

- one correct FTP-Protocol run,
- several runs after another,
- several runs interleaved,
- an illegal protocol run, or
- several illegal protocol runs.

We only do the the simplest case here: one correct protocol run.

There are four different states which are modelled as a datatype.

datatype *ftp-states* = *S0* | *S1* | *S2* | *S3*

The following constant is *True* for all sets which are correct FTP runs for a given source and destination address, ID, and data-port number.

consts
is-ftp :: *ftp-states* ⇒ *IntegerPort* ⇒ *IntegerPort* ⇒ *id* ⇒ *port* ⇒ (*IntegerPort*,*ftp-msg*) *history* ⇒ *bool*
primrec
is-ftp H c s i p [] = (*H*=*S3*)
is-ftp H c s i p (*x*#*InL*) = (λ (*id*,*sr*,*de*,*co*). (((*id* = *i* ∧ (
(*H*=*S2* ∧ *sr* = *c* ∧ *de* = *s* ∧ *co* = *ftp-init* ∧ *is-ftp S3 c s i p InL*) ∨
(*H*=*S1* ∧ *sr* = *c* ∧ *de* = *s* ∧ *co* = *ftp-port-request p* ∧ *is-ftp S2 c s i p InL*) ∨
(*H*=*S1* ∧ *sr* = *s* ∧ *de* = (*fst c*,*p*) ∧ *co*= *ftp-data* ∧ *is-ftp S1 c s i p InL*) ∨
(*H*=*S0* ∧ *sr* = *c* ∧ *de* = *s* ∧ *co* = *ftp-close* ∧ *is-ftp S1 c s i p InL*)))))) *x*

This definition is crucial for specifying what we actually want to test. Extending it produces more test cases but increases the time necessary to create them and vice-versa.

The following constant then returns a set of all the historys which denote such a normal behaviour FTP run, again for a given source and destination address, ID, and data-port.

definition
NB-ftp :: *IntegerPort src* ⇒ *IntegerPort dest* ⇒ *id* ⇒ *port* ⇒ (*IntegerPort*,*ftp-msg*) *history set* **where**
NB-ftp s d i p ≡ {*x*. (*is-ftp S0 s d i p x*)}

Contrary to the case of a stateless packet filter, a lot of the proof work will only be done during the test *data* generation. This means that we need to add the required lemmas to the simplifier set, such that they will be used. The following additional lemmas are neccessary when we use the IntegerPort address representation. They should be added to the simplifier set just before test data generation.

lemma *subnetOf-lemma*: (*a*::*int*) ≠ (*c*::*int*) ⟹ ∀ *x*∈*subnet-of* (*a*, *b*::*port*). (*c*, *d*) ∉ *x*
apply (*rule ballI*)
apply (*simp add*: *IntegerPort.subnet-of-def*)
done

lemma *subnetOf-lemma2*: ∀ *x*∈*subnet-of* (*a*::*int*, *b*::*port*). (*a*, *b*) ∈ *x*
apply (*rule ballI*)
apply (*simp add*: *IntegerPort.subnet-of-def*)
done

lemma *subnetOf-lemma3*: (∃ *x*. *x* ∈ *subnet-of* (*a*::*int*, *b*::*port*))

apply (*rule exI*)
apply (*simp add*: *IntegerPort.subnet-of-def*)
done

lemma *subnetOf-lemma4*: $\exists x \in$ *subnet-of* (*a*::*int*, *b*::*port*). (*a*, *c*::*port*) $\in x$
apply (*rule bexI*)
apply (*simp-all add*: *IntegerPort.subnet-of-def*)
done

lemma *port-open-lemma*: $\neg$ (*Ex* (*port-open* [] (*x*::*port*)))
apply (*simp add*: *port-open-def*)
done

end

9 Examples

9.1 Stateless Example

theory
SimpleDMZIntegerDocument
imports
FWTesting
begin

This is a typical example for a small stateless packet filter. There are three subnetworks, with either none or some protocols allowed between them.

We use IntegerPort as the address model.

constdefs
intranet::*IntegerPort net*
intranet $\equiv$ {{(*a*,*b*) . *a* = *3*}}

dmz :: *IntegerPort net*
dmz $\equiv$ {{(*a*,*b*). *a* = *7*}}

internet :: *IntegerPort net*
internet $\equiv$ {{(*a*,*b*). $\neg$ (*a*=*3* $\vee$ *a* =*7*)}}

constdefs
Intranet-DMZ-Port :: (*IntegerPort*,*DummyContent*) *Policy*
Intranet-DMZ-Port $\equiv$ *allow-from-to-port ftp intranet dmz*

Intranet-Internet-Port :: (*IntegerPort*,*DummyContent*) *Policy*
Intranet-Internet-Port ≡ *allow-from-to-port http intranet internet*

Internet-DMZ-Port :: (*IntegerPort*,*DummyContent*) *Policy*
Internet-DMZ-Port ≡ *allow-from-to-port smtp internet dmz*

The policy:

definition *policy* :: (*IntegerPort*, *DummyContent*) *Policy* **where**
policy ≡ *deny-all* ++
Intranet-Internet-Port ++
Intranet-DMZ-Port ++
Internet-DMZ-Port

lemmas *PolicyLemmas* = *dmz-def internet-def intranet-def*
Intranet-Internet-Port-def Intranet-DMZ-Port-def
Internet-DMZ-Port-def policy-def
src-def dest-def in-subnet-def
IntegerPortLemmas
content-def

Only create test cases crossing network boundaries.

definition *not-in-same-net* :: (*IntegerPort*,*DummyContent*) *packet* ⇒ *bool* **where**
not-in-same-net x ≡ (*src x* ⊏ *internet* ⟶ ¬ *dest x* ⊏ *internet*) ∧
(*src x* ⊏ *intranet* ⟶ ¬ *dest x* ⊏ *intranet*) ∧
(*src x* ⊏ *dmz* ⟶ ¬ *dest x* ⊏ *dmz*)

declare *Ports* [*simp add*]

The test specification:

test-spec *not-in-same-net x* ⟶ *FUT x* = *policy x*
apply (*prepare-fw-spec*)
apply (*simp add*: *not-in-same-net-def PolicyLemmas PortCombinators PolicyCombinators*)
apply (*gen-test-cases FUT*)
apply (*simp-all add*: *PolicyLemmas*)
store-test-thm *PolicyTest*

testgen-params[*iterations=100*]

gen-test-data *PolicyTest*

The set of generated test data is:

$FUT\ (-3, (7, 8), (10, 7), data) = Some\ (deny\ (-3, (7, 8), (10, 7), data))$

$FUT\ (-2, (7, -2), (10, 10), data) = Some\ (deny\ (-2, (7, -2), (10, 10), data))$

$FUT\ (-2,\ (7,\ -10),\ (10,\ 10),\ data) = Some\ (deny\ (-2,\ (7,\ -10),\ (10,\ 10),\ data))$
$FUT\ (-2,\ (7,\ -7),\ (8,\ -6),\ data) = Some\ (deny\ (-2,\ (7,\ -7),\ (8,\ -6),\ data))$
$FUT\ (-3,\ (7,\ -10),\ (4,\ 3),\ data) = Some\ (deny\ (-3,\ (7,\ -10),\ (4,\ 3),\ data))$
$FUT\ (2,\ (7,\ 7),\ (-2,\ -5),\ data) = Some\ (deny\ (2,\ (7,\ 7),\ (-2,\ -5),\ data))$
$FUT\ (-6,\ (7,\ 5),\ (10,\ 7),\ data) = Some\ (deny\ (-6,\ (7,\ 5),\ (10,\ 7),\ data))$
$FUT\ (8,\ (7,\ -2),\ (-4,\ 1),\ data) = Some\ (deny\ (8,\ (7,\ -2),\ (-4,\ 1),\ data))$
$FUT\ (-4,\ (7,\ -1),\ (-2,\ 5),\ data) = Some\ (deny\ (-4,\ (7,\ -1),\ (-2,\ 5),\ data))$
$FUT\ (-8,\ (7,\ -4),\ (5,\ -3),\ data) = Some\ (deny\ (-8,\ (7,\ -4),\ (5,\ -3),\ data))$
$FUT\ (-9,\ (7,\ 9),\ (3,\ 3),\ data) = Some\ (deny\ (-9,\ (7,\ 9),\ (3,\ 3),\ data))$
$FUT\ (6,\ (7,\ 10),\ (3,\ -2),\ data) = Some\ (deny\ (6,\ (7,\ 10),\ (3,\ -2),\ data))$
$FUT\ (-8,\ (3,\ -2),\ (-2,\ http),\ data) = Some\ (accept\ (-8,\ (3,\ -2),\ (-2,\ http),\ data))$
$FUT\ (3,\ (3,\ 3),\ (7,\ ftp),\ data) = Some\ (accept\ (3,\ (3,\ 3),\ (7,\ ftp),\ data))$
$FUT\ (-10,\ (3,\ -3),\ (7,\ -1),\ data) = Some\ (deny\ (-10,\ (3,\ -3),\ (7,\ -1),\ data))$
$FUT\ (2,\ (3,\ -5),\ (7,\ -9),\ data) = Some\ (deny\ (2,\ (3,\ -5),\ (7,\ -9),\ data))$
$FUT\ (4,\ (3,\ -9),\ (7,\ ftp),\ data) = Some\ (accept\ (4,\ (3,\ -9),\ (7,\ ftp),\ data))$
$FUT\ (2,\ (3,\ 2),\ (-1,\ -4),\ data) = Some\ (deny\ (2,\ (3,\ 2),\ (-1,\ -4),\ data))$
$FUT\ (6,\ (3,\ 9),\ (0,\ 8),\ data) = Some\ (deny\ (6,\ (3,\ 9),\ (0,\ 8),\ data))$
$FUT\ (5,\ (3,\ -10),\ (-2,\ 7),\ data) = Some\ (deny\ (5,\ (3,\ -10),\ (-2,\ 7),\ data))$
$FUT\ (1,\ (3,\ -10),\ (1,\ 9),\ data) = Some\ (deny\ (1,\ (3,\ -10),\ (1,\ 9),\ data))$
$FUT\ (5,\ (3,\ -7),\ (-9,\ 7),\ data) = Some\ (deny\ (5,\ (3,\ -7),\ (-9,\ 7),\ data))$
$FUT\ (5,\ (3,\ 0),\ (-2,\ -10),\ data) = Some\ (deny\ (5,\ (3,\ 0),\ (-2,\ -10),\ data))$
$FUT\ (4,\ (3,\ -3),\ (-2,\ -7),\ data) = Some\ (deny\ (4,\ (3,\ -3),\ (-2,\ -7),\ data))$
$FUT\ (2,\ (7,\ -4),\ (8,\ 10),\ data) = Some\ (deny\ (2,\ (7,\ -4),\ (8,\ 10),\ data))$
$FUT\ (-8,\ (7,\ -2),\ (-5,\ 9),\ data) = Some\ (deny\ (-8,\ (7,\ -2),\ (-5,\ 9),$

$data$))
FUT (-10, (7, -5), (6, 0), $data$) = $Some$ ($deny$ (-10, (7, -5), (6, 0), $data$))
FUT (10, (7, -10), (5, 1), $data$) = $Some$ ($deny$ (10, (7, -10), (5, 1), $data$))
FUT (-3, (7, -7), (-2, -7), $data$) = $Some$ ($deny$ (-3, (7, -7), (-2, -7), $data$))
FUT (-8, (7, 8), (2, 4), $data$) = $Some$ ($deny$ (-8, (7, 8), (2, 4), $data$))
FUT (-4, (7, 5), (4, -10), $data$) = $Some$ ($deny$ (-4, (7, 5), (4, -10), $data$))
FUT (8, (7, -7), (9, 3), $data$) = $Some$ ($deny$ (8, (7, -7), (9, 3), $data$))
FUT (-10, (7, -8), (-10, 7), $data$) = $Some$ ($deny$ (-10, (7, -8), (-10, 7), $data$))
FUT (5, (7, 2), (1, 5), $data$) = $Some$ ($deny$ (5, (7, 2), (1, 5), $data$))
FUT (-1, (7, -1), (-1, 8), $data$) = $Some$ ($deny$ (-1, (7, -1), (-1, 8), $data$))
FUT (-1, (7, -6), (8, -6), $data$) = $Some$ ($deny$ (-1, (7, -6), (8, -6), $data$))
FUT (-4, (6, 10), (3, -3), $data$) = $Some$ ($deny$ (-4, (6, 10), (3, -3), $data$))
FUT (-3, (8, -7), (3, 8), $data$) = $Some$ ($deny$ (-3, (8, -7), (3, 8), $data$))
FUT (1, (-6, -5), (3, -4), $data$) = $Some$ ($deny$ (1, (-6, -5), (3, -4), $data$))
FUT (-10, (-10, 4), (3, 9), $data$) = $Some$ ($deny$ (-10, (-10, 4), (3, 9), $data$))
FUT (10, (4, -5), (3, -2), $data$) = $Some$ ($deny$ (10, (4, -5), (3, -2), $data$))
FUT (3, (6, -10), (3, -8), $data$) = $Some$ ($deny$ (3, (6, -10), (3, -8), $data$))
FUT (5, (-6, 8), (3, 9), $data$) = $Some$ ($deny$ (5, (-6, 8), (3, 9), $data$))
FUT (0, (-2, 6), (3, 3), $data$) = $Some$ ($deny$ (0, (-2, 6), (3, 3), $data$))
FUT (3, (0, 2), (3, -6), $data$) = $Some$ ($deny$ (3, (0, 2), (3, -6), $data$))
FUT (2, (-9, -6), (3, 4), $data$) = $Some$ ($deny$ (2, (-9, -6), (3, 4), $data$))
FUT (5, (4, -3), (3, -10), $data$) = $Some$ ($deny$ (5, (4, -3), (3, -10), $data$))
FUT (-4, (7, 8), (3, 0), $data$) = $Some$ ($deny$ (-4, (7, 8), (3, 0), $data$))
FUT (-9, (-3, 1), (3, -2), $data$) = $Some$ ($deny$ (-9, (-3, 1), (3, -2), $data$))

$FUT\ (9,\ (0,\ -5),\ (3,\ 2),\ data) = Some\ (deny\ (9,\ (0,\ -5),\ (3,\ 2),\ data))$
$FUT\ (-2,\ (7,\ -1),\ (3,\ -4),\ data) = Some\ (deny\ (-2,\ (7,\ -1),\ (3,\ -4),\ data))$
$FUT\ (-9,\ (0,\ 5),\ (3,\ 0),\ data) = Some\ (deny\ (-9,\ (0,\ 5),\ (3,\ 0),\ data))$
$FUT\ (9,\ (8,\ 2),\ (3,\ 6),\ data) = Some\ (deny\ (9,\ (8,\ 2),\ (3,\ 6),\ data))$
$FUT\ (-6,\ (7,\ -4),\ (3,\ 0),\ data) = Some\ (deny\ (-6,\ (7,\ -4),\ (3,\ 0),\ data))$
$FUT\ (6,\ (-8,\ 10),\ (3,\ -8),\ data) = Some\ (deny\ (6,\ (-8,\ 10),\ (3,\ -8),\ data))$
$FUT\ (-4,\ (10,\ 2),\ (3,\ 7),\ data) = Some\ (deny\ (-4,\ (10,\ 2),\ (3,\ 7),\ data))$
$FUT\ (1,\ (2,\ -10),\ (3,\ 3),\ data) = Some\ (deny\ (1,\ (2,\ -10),\ (3,\ 3),\ data))$
$FUT\ (10,\ (8,\ 2),\ (3,\ -7),\ data) = Some\ (deny\ (10,\ (8,\ 2),\ (3,\ -7),\ data))$
$FUT\ (-7,\ (7,\ 7),\ (3,\ -2),\ data) = Some\ (deny\ (-7,\ (7,\ 7),\ (3,\ -2),\ data))$
$FUT\ (-3,\ (10,\ -10),\ (3,\ 2),\ data) = Some\ (deny\ (-3,\ (10,\ -10),\ (3,\ 2),\ data))$
$FUT\ (4,\ (9,\ -9),\ (7,\ smtp),\ data) = Some\ (accept\ (4,\ (9,\ -9),\ (7,\ smtp),\ data))$
$FUT\ (-3,\ (-9,\ 0),\ (7,\ -2),\ data) = Some\ (deny\ (-3,\ (-9,\ 0),\ (7,\ -2),\ data))$
$FUT\ (-3,\ (4,\ 9),\ (7,\ smtp),\ data) = Some\ (accept\ (-3,\ (4,\ 9),\ (7,\ smtp),\ data))$
$FUT\ (-1,\ (-5,\ 7),\ (7,\ -8),\ data) = Some\ (deny\ (-1,\ (-5,\ 7),\ (7,\ -8),\ data))$
$FUT\ (6,\ (-8,\ -4),\ (7,\ -10),\ data) = Some\ (deny\ (6,\ (-8,\ -4),\ (7,\ -10),\ data))$
$FUT\ (-3,\ (-10,\ 4),\ (7,\ smtp),\ data) = Some\ (accept\ (-3,\ (-10,\ 4),\ (7,\ smtp),\ data))$
$FUT\ (-9,\ (4,\ -3),\ (7,\ 7),\ data) = Some\ (deny\ (-9,\ (4,\ -3),\ (7,\ 7),\ data))$
$FUT\ (-6,\ (-4,\ 6),\ (7,\ smtp),\ data) = Some\ (accept\ (-6,\ (-4,\ 6),\ (7,\ smtp),\ data))$
$FUT\ (-7,\ (8,\ -9),\ (7,\ 0),\ data) = Some\ (deny\ (-7,\ (8,\ -9),\ (7,\ 0),\ data))$
$FUT\ (-6,\ (10,\ -6),\ (7,\ -10),\ data) = Some\ (deny\ (-6,\ (10,\ -6),\ (7,\ -10),\ data))$
$FUT\ (-8,\ (3,\ -4),\ (7,\ -2),\ data) = Some\ (deny\ (-8,\ (3,\ -4),\ (7,\ -2),\ data))$
$FUT\ (-1,\ (-3,\ 10),\ (7,\ -3),\ data) = Some\ (deny\ (-1,\ (-3,\ 10),\ (7,\ -3),\ data))$

end

9.2 FTP Example

theory *FTPTestDocument*
imports
  FWTesting
begin

In this theory we generate the test data for correct runs of the FTP protocol. As usual, we start with definining the networks and the policy. We use a rather simple policy which allows only FTP connections starting from the intranet going to the internet and denies everything else.

constdefs
  $intranet :: IntegerPort\ net$
  $intranet \equiv \{\{(a,b)\ .\ a = 3\}\}$

  $internet :: IntegerPort\ net$
  $internet \equiv \{\{(a,b).\ a > 3\}\}$

constdefs
  $ftp\text{-}policy :: (IntegerPort,ftp\text{-}msg)\ Policy$
  $ftp\text{-}policy \equiv deny\text{-}all\ {+}{+}\ allow\text{-}from\text{-}to\text{-}port\ ftp\ intranet\ \ internet$

The next two constants check if an address is in the Intranet or in the Internet respectively.

constdefs
  $is\text{-}in\text{-}intranet :: IntegerPort\ \Rightarrow bool$
  $is\text{-}in\text{-}intranet\ a \equiv (fst\ a) = 3$

  $is\text{-}in\text{-}internet :: IntegerPort \Rightarrow bool$
  $is\text{-}in\text{-}internet\ a \equiv\ (fst\ a) > 3$

The next definition is our starting state: an empty trace and the just defined policy.

constdefs
  $\sigma\text{-}0\text{-}ftp ::\ (IntegerPort,\ ftp\text{-}msg)\ history \times$
  $\qquad (IntegerPort,\ ftp\text{-}msg)\ Policy$
  $\sigma\text{-}0\text{-}ftp \equiv ([],ftp\text{-}policy)$

Next we state the conditions we have on our trace: a normal behaviour FTP run from the intranet to some server in the internet starting on port 21.

constdefs $accept\text{-}ftp ::\ (IntegerPort,\ ftp\text{-}msg)\ history \Rightarrow bool$

accept-ftp t $\equiv \exists$ *c s i p*. *t* $\in$ *NB-ftp c s i p* $\wedge$ *is-in-intranet c* $\wedge$ *is-in-internet s* $\wedge$ (*snd s*) = *21*

The depth of the test case generation corresponds to the maximal length of generated traces. 4 is the minimum to get a full FTP protocol run.

testgen-params [*depth=4*]

The test specification:

test-spec *accept-ftp* (*rev t*) $\longrightarrow$
(σ*-0-ftp* $\models$ (*os* $\leftarrow$ *mbind t FTP-ST*; (λ σ. *Some* (*FUT* (*rev t*) = σ, σ))))
apply(*simp add*: *accept-ftp-def* σ*-0-ftp-def*)
apply (*rule impI*)+
apply (*unfold NB-ftp-def is-in-internet-def is-in-intranet-def*)
apply *simp*
apply (*gen-test-cases FUT split*: *HOL.split-if-asm*)
apply (*simp-all*)
store-test-thm *ftp-test*

We need to add all required lemmas to the simplifier set, such that they can be used during test data generation.

lemmas *ST-simps* = *Let-def valid-def unit-SE-def bind-SE-def orelse-def in-subnet-def src-def dest-def IntegerPort.dest-port-def subnet-of-def id-def port-open-def is-init-def is-data-def is-port-request-def is-close-def p-accept-def content-def PolicyCombinators PortCombinators is-in-intranet-def is-in-internet-def intranet-def internet-def exI subnetOf-lemma subnetOf-lemma2 subnetOf-lemma3 subnetOf-lemma4 port-open-lemma ftp-policy-def*

declare *ST-simps* [*simp*]

gen-test-data ftp-test

declare *ST-simps* [*simp del*]

The generated test data look as follows (with the unfolded policy rewritten):

- FUT [(4, (3, 5), (8, 21), *ftp_close*), (4, (3, 5), (8, 21), *ftp_port_request* 4), (4, (3, 5), (8, 21), *ftp_init*)] = ([(4, (3, 5), (8, 21), *ftp_close*), (4, (3, 5), (8, 21), *ftp_port_request* 4), (4, (3, 5), (8, 21), *ftp_init*)],policy)
- FUT [(1, (3, 7), (9, 21), *ftp_close*), (1, (9, 21), (3, 6), *ftp_data*), (1, (3, 7), (9, 21), *ftp_port_request* 6),(1, (3, 7), (9, 21), *ftp_init*)] = ([(1, (3, 7), (9, 21), *ftp_close*), (1, (9, 21), (3, 6), *ftp_data*), (1, (3, 7), (9, 21), *ftp_port_request* 6), (1, (3, 7), (9, 21), *ftp_init*)],policy)

end

9.3 FTP with Observers

theory *FTPObserver2Document*
imports *FWTesting*
begin

In this theory, we formalise an adapted version of an FTP protocol using the observers. The protocol consists of four messages:

- portReq X: the client initiates a session, and specifies a port range, where the data should be sent to (only an upper bound, for the sake of simplicity).
- portAck Y: the server acknowledges the connection, and non-deterministically chooses a port number from the specified range.
- data: the server sends data on the specified port. This message can happen arbitrarily many times.
- close: the client closes the connection.

We will make use of the observer2, and closely follow the corresponding example from the HOL-TestGen distribution.

The test case generation is done on the basis of *abstract traces*. Such abstract traces contain explicit variables, and the functions substitute and rebind are used to replace them with concrete values during the run of the test driver.

datatype $vars = X \mid Y$

datatype $data = Data$

types $chan = port$

types $env = vars \rightharpoonup chan$

definition $lookup :: ['a \rightharpoonup 'b,'a] \Rightarrow 'b$ **where**
$lookup\ env\ v \equiv the\ (env\ v)$

The traces are lists of packets. However, in this case, we will not make use of the usual packet definition *directly*, but use a datatype representation of them. There are abstract and concrete packets:

datatype *ftp-packet-abs* = *port-reqA vars id IntegerPort IntegerPort* |
port-ackA vars id IntegerPort IntegerPort |
dataA vars id IntegerPort address |
closeA id IntegerPort IntegerPort

datatype *ftp-packet-conc* = *port-reqC port id IntegerPort IntegerPort* |
port-ackC port id IntegerPort IntegerPort |
dataC id IntegerPort IntegerPort |
closeC id IntegerPort IntegerPort

types *ftp-packet* = *ftp-packet-abs* + *ftp-packet-conc*

The following two functions then make the connection between the packet representations. Note that in the way this function is defined, a data message will always be allowed. In contrast to the other form of FTP testing, we do not change the policy during protocol execution, rather we take more control of the protocol execution itself:

datatype *ftp-event* = *port-req* | *port-ack* | *data* | *close*

fun *packet-accept* :: *ftp-packet-abs* ⇒ (*IntegerPort*,*ftp-event*) *Policy* ⇒ *bool* **where**
packet-accept (*port-reqA v i s d*) *p* = *p-accept* (*i*,*s*,*d*,*port-req*) *p*
|*packet-accept* (*port-ackA v i s d*) *p* = *p-accept* (*i*,*s*,*d*,*port-ack*) *p*
|*packet-accept* (*closeA i s d*) *p* = *p-accept* (*i*,*s*,*d*,*close*) *p*
|*packet-accept* (*dataA v i s da*) *p* = *True*

fun *packet-accept-conc* :: *ftp-packet-conc* ⇒ (*IntegerPort*,*ftp-event*) *Policy* ⇒ *bool* **where**
packet-accept-conc (*port-reqC v i s d*) *p* = *p-accept* (*i*,*s*,*d*,*port-req*) *p*
|*packet-accept-conc* (*port-ackC v i s d*) *p* = *p-accept* (*i*,*s*,*d*,*port-ack*) *p*
|*packet-accept-conc* (*closeC i s d*) *p* = *p-accept* (*i*,*s*,*d*,*close*) *p*
|*packet-accept-conc* (*dataC i s da*) *p* = *True*

The usual function substitute and rebind:

fun *substitute* :: [*env*, *ftp-packet-abs*] ⇒ *ftp-packet-conc* **where**
substitute env (*port-reqA v i s d*) = (*port-reqC* (*lookup env v*) *i s d*)
|*substitute env* (*port-ackA v i s d*) = (*port-reqC* (*lookup env v*)*i s d*)
|*substitute env* (*dataA v i s da*) = (*dataC i s* (*da*,(*lookup env v*)))
|*substitute env* (*closeA i s d*) = (*closeC i s d*)

fun *rebind* :: [*env*, *ftp-packet-conc*] ⇒ *env* **where**
rebind env (*port-reqC p i s d*) = *env*(*X* ↦ *p*)
|*rebind env* (*port-ackC p i s d*) = *env*(*Y* ↦ *p*)
|*rebind env* (*dataC i s d*) = *env*
|*rebind env* (*closeC i s d*) = *env*

The automaton which describes successful executions of the protocol:

datatype *ftp-states* = *S0* | *S1* | *S2* | *S3*

fun *ftp-automaton* :: *ftp-states* ⇒ *ftp-packet-abs list* ⇒ (*IntegerPort*,*ftp-event*) *Policy* ⇒
id ⇒ *IntegerPort* ⇒ *IntegerPort* ⇒ *bool* **where**
ftp-automaton H [] *p i c s* = (*H* = *S3*)

$|$*ftp-automaton H* $(x\#xs)$ *policy ii c s* $=$ (*case H of*
S0 $=>$ (*case x of* (*port-reqA X i sr de*) $\Rightarrow ii = i \wedge sr = c \wedge de = s \wedge$ *packet-accept x policy* $\wedge$ *ftp-automaton S1 xs policy ii c s*
$|$ - $\Rightarrow$ *False*)
$|$ *S1* $=>$ (*case x of* (*port-ackA Y i sr de*) $\Rightarrow ii = i \wedge sr = s \wedge de =$ (*fst c*,*21*) $\wedge$ *packet-accept x policy* $\wedge$ *ftp-automaton S2 xs policy ii c s*
$|$ - $\Rightarrow$ *False*)
$|$ *S2* $\Rightarrow$ (*case x of* (*dataA Y i sr da*) $\Rightarrow ii = i \wedge sr = s \wedge$ *fst c* $=$ (*da*) $\wedge$ *ftp-automaton S2 xs policy ii c s*
$|$ (*closeA i sr de*) $\Rightarrow ii = i \wedge sr = c \wedge de = s \wedge$ *packet-accept x policy* $\wedge$ *ftp-automaton S3 xs policy ii c s*
$|$ - $\Rightarrow$ *False*)
$|$ *S3* $\Rightarrow$ *False*)

Next, we declare our specific setting and the policy:

constdefs
intranet :: *IntegerPort net*
intranet $\equiv \{\{(a,e)\ .\ a = 3\}\}$

internet :: *IntegerPort net*
internet $\equiv \{\{(a,c).\ a > 3\}\}$

constdefs
ftp-policy :: (*IntegerPort*,*ftp-event*) *Policy*
ftp-policy $\equiv$ *deny-all* ++ *allow-from-to-port* (*21*::*port*) *internet intranet* ++ *allow-all-from-to intranet internet*

The next two constants check if an address is in the Intranet or in the Internet respectively.

constdefs
is-in-intranet :: *IntegerPort* $\Rightarrow$ *bool*
is-in-intranet a $\equiv$ (*fst a*) $=$ *3*

is-in-internet :: *IntegerPort* $\Rightarrow$ *bool*
is-in-internet a $\equiv$ (*fst a*) $>$*3*

definition
NB-ftp **where**
NB-ftp i c s $\equiv \{x.$ (*ftp-automaton S0 x ftp-policy i c s*)$\}$

definition *accept-ftp* :: *ftp-packet-abs list* $\Rightarrow$ *bool* **where**
accept-ftp t $\equiv \exists$ *i c s*. $t \in$ *NB-ftp i c s* $\wedge$ *is-in-intranet c* $\wedge$ *is-in-internet s*

The postcondition:

fun *postcond* :: *env* $\Rightarrow$ $'\sigma$ $\Rightarrow$ *ftp-packet-conc* $\Rightarrow$ *ftp-packet-conc* $\Rightarrow$ *bool* **where**

postcond env x (port-reqC p i c s) y = (case y of (port-ackC pa i s c) => (pa <= p) | - ⇒ False)
| *postcond env x (port-ackC p i s c) y = (case y of (dataC i s c) => (snd c = p ∧ p = lookup env Y) |- ⇒ False)*
| *postcond env x (dataC i s c) y = (case y of (dataC i s c) ⇒ (snd c = lookup env Y)*
|(closeC i c s) ⇒ True)
| *postcond env x y z = False*

declare *NB-ftp-def accept-ftp-def ftp-policy-def accept-ftp-def packet-accept-def p-accept-def intranet-def internet-def is-in-intranet-def is-in-internet-def* [*simp add*]

Next some theorem proving, trying to achieve better test case generation results:

lemma *allowAll*[*simp*]: *packet-accept x allow-all*
apply (*case-tac x*, *simp-all*)
apply (*simp-all add*: *PLemmas p-accept-def*)
done

lemma *start*[*simp*]: *ftp-automaton S0 (x#xs) p i c s = ((x = port-reqA X i c s) ∧ ftp-automaton S1 xs p i c s ∧ packet-accept x p)*
apply *simp*
apply (*case-tac x*,*simp-all*)
apply (*rule vars.exhaust*, *auto*)
done

lemma *step1*[*simp*]: *ftp-automaton S1 (x#xs) p i c s = ((x = port-ackA Y i s (fst c,21)) ∧ ftp-automaton S2 xs p i c s ∧ packet-accept x p)*
apply *simp*
apply (*case-tac x*,*simp-all*)
apply (*case-tac vars*,*simp-all*)
apply (*rule vars.exhaust*, *auto*)
done

lemma *step2*[*simp*]: *ftp-automaton S2 (x#xs) p i c s = ((x = dataA Y i s (fst c)) ∧ ftp-automaton S2 xs p i c s∧ packet-accept x p) ∨*
((x = closeA i c s) ∧ ftp-automaton S3 xs p i c s)
apply *simp*
apply (*case-tac x*,*simp-all*)
apply (*case-tac vars*,*simp-all*)
apply (*rule vars.exhaust*, *auto*)
done

lemma *step3*[*simp*]: *ftp-automaton S2* [*x*] *p i c s* = (*x* = *closeA i c s*∧ *packet-accept x p*)
apply *simp*
apply (*case-tac x*, *simp-all*)
apply (*case-tac vars*)
apply *simp*
apply *simp*
apply *auto*
done

lemma *packet-accept-a*[*simp*]: *packet-accept* (*dataA a b c d*) *p*
apply *simp*
done

lemma *packet-accept-b*[*simp*]: *is-in-intranet c* ∧ *is-in-internet s* ⟹ *packet-accept* (*port-reqA x i c s*) *ftp-policy*
apply *simp*
apply (*simp add*: *ftp-policy-def*)
apply (*simp add*: *p-accept-def*)
apply (*simp add*: *is-in-intranet-def*)
apply (*simp add*: *PLemmas intranet-def internet-def*)
apply *auto*
done

lemma *packet-accept-c*[*simp*]: *is-in-intranet c* ∧ *is-in-internet s* ∧ *snd c* = *21* ⟹ *packet-accept* (*port-ackA y i s c*) *ftp-policy*
apply *simp*
apply (*simp add*: *ftp-policy-def*)
apply (*simp add*: *p-accept-def*)
apply (*simp add*: *is-in-intranet-def*)
apply (*simp add*: *PLemmas intranet-def internet-def*)
apply *auto*
done

lemma *packet-accept-d*[*simp*]: *is-in-intranet c* ∧ *is-in-internet s* ⟹ *packet-accept* (*closeA i c s*) *ftp-policy*
apply *simp*
apply (*simp add*: *ftp-policy-def*)
apply (*simp add*: *p-accept-def*)
apply (*simp add*: *is-in-intranet-def*)
apply (*simp add*: *PLemmas intranet-def internet-def*)
apply *auto*
done

Now the test specification:

test-spec *accept-ftp t* ⟶
(([*X*↦*init-value*],()) ⊨ (*os* ← (*mbind t* (*observer2 rebind substitute postcond ioprog*));

result (*length trace* = *length os*)))
apply (*simp add*: *accept-ftp-def NB-ftp-def accept-ftp-def packet-accept-def p-accept-def intranet-def internet-def is-in-intranet-def is-in-internet-def*)
apply (*gen-test-cases 5 1 ioprog*)
store-test-thm *ftp*

testgen-params[*iterations=100*]

gen-test-data *ftp*

thm *ftp.test-data*

From inspecting the test theorem and the test data, it is obvious that there is still some more theorem proving required to get better results.

end

A Appendix

theory *FWCompilationProof*
imports *FWCompilation*
begin

This theory contains the complete proofs of the normalisation procedure.

lemma *wellformed-policy1-charn*[*rule-format*] : *wellformed-policy1 p* $\longrightarrow$ *DenyAll* $\in$ *set p* $\longrightarrow$ ($\exists$ *p'*. *p* = *DenyAll* # *p'* $\wedge$ *DenyAll* $\notin$ *set p'*)
by(*induct p*,*simp-all*)

lemma *singleCombinatorsConc*: *singleCombinators* (*x*#*xs*) $\Longrightarrow$ *singleCombinators xs*
by (*case-tac x*,*simp-all*)

lemma *aux0-0*: *singleCombinators x* $\Longrightarrow$ $\neg$ ($\exists$ *a b*. (*a*$\oplus$*b*) $\in$ *set x*)
apply (*induct x*, *simp-all*)
apply (*rule allI*)+
by (*case-tac a*,*simp-all*)

lemma *aux0-4*: (*a* $\in$ *set x* $\vee$ *a* $\in$ *set y*) = (*a* $\in$ *set* (*x*@*y*))
by *auto*

lemma *aux0-1*: $[\![$*singleCombinators xs*; *singleCombinators* [*x*]$]\!]$ $\Longrightarrow$ *singleCombinators* (*x*#*xs*)
by (*case-tac x*,*simp-all*)

lemma *aux0-6*: $[\![$*singleCombinators xs*; $\neg$ ($\exists$ *a b*. *x* = *a* $\oplus$ *b*)$]\!]$ $\Longrightarrow$ *singleCombinators*(*x*#*xs*)
apply (*rule aux0-1*,*simp-all*)
apply (*case-tac x*,*simp-all*)
apply *auto*
done

lemma *aux0-5*: $\neg$ ($\exists$ *a b*. (*a*$\oplus$*b*) $\in$ *set x*) $\Longrightarrow$ *singleCombinators x*
apply (*induct x*)
apply *simp-all*
apply (*rule aux0-6*)
apply *auto*
done

lemma *aux0-7*: $[\![$*singleCombinators x*; *singleCombinators y*$]\!]$ $\Longrightarrow$ *singleCombina-*

tors $(x@y)$
apply (*rule aux0-5*)
apply *auto*
apply (*insert aux0-0* [*of x*])
apply (*insert aux0-0* [*of y*])
apply *auto*
done

lemma *ConcAssoc*: $C((A \oplus B) \oplus D) = C(A \oplus (B \oplus D))$
apply (*simp add*: *C.simps*)
done

lemma *Caux*: $x \in dom\ (C\ b) \Longrightarrow (C\ a\ {++}\ C\ b)\ x = C\ b\ x$
by (*auto simp*: *C.simps dom-def*)

lemma *nCauxb*: $x \notin dom\ (b) \Longrightarrow (a\ {++}\ b)\ x = a\ x$
by (*simp-all add*: *C.simps dom-def map-add-def option.simps*(*4*))

lemma *Cauxb*: $x \notin dom\ (C\ b) \Longrightarrow (C\ a\ {++}\ C\ b)\ x = C\ a\ x$
apply (*rule nCauxb*)
by *simp*

lemma *aux0*: *singleCombinators* (*policy2list p*)
apply (*induct-tac p*)
apply *simp-all*
apply (*rule aux0-7*)
apply *simp-all*
done

lemma *ANDConc*[*rule-format*]: *allNetsDistinct* (*a*#*p*) $\longrightarrow$ *allNetsDistinct* (*p*)
apply (*simp add*: *allNetsDistinct-def*)
apply (*case-tac a*)
by *simp-all*

lemma *aux6*: *twoNetsDistinct a1 a2 a b* $\Longrightarrow$ *dom* (*deny-all-from-to a1 a2*) $\cap$ *dom* (*deny-all-from-to a b*) $= \{\}$
by (*auto simp*: *twoNetsDistinct-def netsDistinct-def src-def dest-def in-subnet-def PolicyCombinators.PolicyCombinators dom-def*)

lemma *aux5*[*rule-format*]: (*DenyAllFromTo a b*) ∈ *set p* ⟶ *a* ∈ *set* (*net-list p*)
by (*rule net-list-aux.induct*,*simp-all*)

lemma *aux5a*[*rule-format*]: (*DenyAllFromTo b a*) ∈ *set p* ⟶ *a* ∈ *set* (*net-list p*)
by (*rule net-list-aux.induct*,*simp-all*)

lemma *aux5c*[*rule-format*]: (*AllowPortFromTo a b po*) ∈ *set p* ⟶ *a* ∈ *set* (*net-list p*)
by (*rule net-list-aux.induct*,*simp-all*)

lemma *aux5d*[*rule-format*]: (*AllowPortFromTo b a po*) ∈ *set p* ⟶ *a* ∈ *set* (*net-list p*)
by (*rule net-list-aux.induct*,*simp-all*)

lemma *aux10*[*rule-format*]: *a* ∈ *set* (*net-list p*) ⟶ *a* ∈ *set* (*net-list-aux p*)
by *simp*

lemma *srcInNetListaux*[*simp*]: ⟦*x* ∈ *set p*; *singleCombinators*[*x*]; *x* ≠ *DenyAll*⟧ ⟹ *srcNet x* ∈ *set* (*net-list-aux p*)
apply (*induct p*)
apply *simp-all*
apply (*case-tac x* = *a*, *simp-all*)
apply (*case-tac a*, *simp-all*)+
done

lemma *destInNetListaux*[*simp*]: ⟦*x* ∈ *set p*; *singleCombinators*[*x*]; *x* ≠ *DenyAll*⟧ ⟹ *destNet x* ∈ *set* (*net-list-aux p*)
apply (*induct p*)
apply *simp-all*
apply (*case-tac x* = *a*, *simp-all*)
apply (*case-tac a*, *simp-all*)+
done

lemma *tND1*: ⟦*allNetsDistinct p*; *x* ∈ *set p*; *y* ∈ *set p*; *a* = *srcNet x*; *b* = *destNet x*; *c* = *srcNet y*; *d* = *destNet y*; *a* ≠ *c*;
singleCombinators[*x*]; *x* ≠ *DenyAll*; *singleCombinators*[*y*]; *y* ≠ *DenyAll*⟧ ⟹ *twoNetsDistinct a b c d*
apply (*simp add*: *allNetsDistinct-def twoNetsDistinct-def*)
done

lemma *tND2*: ⟦*allNetsDistinct p*; *x* ∈ *set p*; *y* ∈ *set p*; *a* = *srcNet x*; *b* = *destNet x*; *c* = *srcNet y*; *d* = *destNet y*; *b* ≠ *d*;

$singleCombinators[x]$; $x \neq DenyAll$; $singleCombinators[y]$; $y \neq DenyAll$⟧
⟹ *twoNetsDistinct a b c d*
apply (*simp add*: *allNetsDistinct-def twoNetsDistinct-def*)
done

lemma *tND*: ⟦*allNetsDistinct p*; $x \in set\ p$; $y \in set\ p$; $a = srcNet\ x$; $b = destNet\ x$; $c = srcNet\ y$; $d = destNet\ y$; $a \neq c \vee b \neq d$;
$singleCombinators[x]$; $x \neq DenyAll$; $singleCombinators[y]$; $y \neq DenyAll$⟧
⟹ *twoNetsDistinct a b c d*
apply (*case-tac* $a \neq c$, *simp-all*)
apply (*erule-tac* $x = x$ **and** $y = y$ **in** *tND1*, *simp-all*)
apply (*erule-tac* $x = x$ **and** $y = y$ **in** *tND2*, *simp-all*)
done

lemma *aux7*: ⟦*DenyAllFromTo a b* ∈ *set p*; *allNetsDistinct* ((*DenyAllFromTo c d*)#*p*); $a \neq c \vee b \neq d$⟧ ⟹ *twoNetsDistinct a b c d*
apply (*erule-tac* x = *DenyAllFromTo a b* **and** y = *DenyAllFromTo c d* **in** *tND*)
apply *simp-all*
done

lemma *aux7a*: ⟦*DenyAllFromTo a b* ∈ *set p*; *allNetsDistinct* ((*AllowPortFromTo c d po*)#*p*); $a \neq c \vee b \neq d$⟧ ⟹ *twoNetsDistinct a b c d*
apply (*erule-tac* x = *DenyAllFromTo a b* **and** y = *AllowPortFromTo c d po* **in** *tND*)
apply *simp-all*
done

lemma *nDComm*: **assumes** *ab*: *netsDistinct a b* **shows** *ba*: *netsDistinct b a*
apply (*insert ab*)
by (*auto simp*: *netsDistinct-def in-subnet-def*)

lemma *tNDComm*: **assumes** *abcd*: *twoNetsDistinct a b c d* **shows** *twoNetsDistinct c d a b*
apply (*insert abcd*)
apply (*metis twoNetsDistinct-def nDComm*)
done

lemma *aux*[*rule-format*]: $a \in set$ (*removeShadowRules2 p*) ⟶ $a \in set\ p$
apply (*case-tac a*)
by (*rule removeShadowRules2.induct*, *simp-all*)+

lemma *aux12*: ⟦$a \in x$; $b \notin x$⟧ ⟹ $a \neq b$
by *auto*

lemma *aux26*[*simp*]: *twoNetsDistinct a b c d* ⟹ *dom* (*C* (*AllowPortFromTo a b*

p)) ∩ *dom* (*C* (*DenyAllFromTo c d*)) = {}
by (*auto simp*: *PLemmas twoNetsDistinct-def netsDistinct-def*) *auto*

lemma *ND0aux1*[*rule-format*]: *DenyAllFromTo x y* ∈ *set b* ⟹ *x* ∈ *set* (*net-list-aux b*)
by (*metis aux5 net-list.simps set-remdups*)

lemma *ND0aux2*[*rule-format*]: *DenyAllFromTo x y* ∈ *set b* ⟹ *y* ∈ *set* (*net-list-aux b*)
by (*metis aux5a net-list.simps set-remdups*)

lemma *ND0aux3*[*rule-format*]: *AllowPortFromTo x y p* ∈ *set b* ⟹ *x* ∈ *set* (*net-list-aux b*)
by (*metis aux5c net-list.simps set-remdups*)

lemma *ND0aux4*[*rule-format*]: *AllowPortFromTo x y p* ∈ *set b* ⟹ *y* ∈ *set* (*net-list-aux b*)
by (*metis aux5d net-list.simps set-remdups*)

lemma *aNDSubsetaux*[*rule-format*]: *singleCombinators a* ⟶ *set a* ⊆ *set b* ⟶ *set* (*net-list-aux a*) ⊆ *set* (*net-list-aux b*)
apply (*induct a*)
apply *simp-all*
apply *clarify*
apply (*drule mp*, *erule singleCombinatorsConc*)
apply (*case-tac a1*)
apply (*simp-all add*: *contra-subsetD*)
apply (*metis contra-subsetD*)
apply (*metis ND0aux1 ND0aux2 contra-subsetD mem-def*)
apply (*metis ND0aux3 ND0aux4 contra-subsetD mem-def*)
done

lemma *aNDSetsEqaux*[*rule-format*]: *singleCombinators a* ⟶ *singleCombinators b* ⟶ *set a* = *set b* ⟶ *set* (*net-list-aux a*) = *set* (*net-list-aux b*)
apply (*rule impI*)+
apply (*rule equalityI*)
apply (*rule aNDSubsetaux*, *simp-all*)+
done

lemma *aNDSubset*: ⟦*singleCombinators a*;*set a* ⊆ *set b*; *allNetsDistinct b*⟧ ⟹ *allNetsDistinct a*
apply (*simp add*: *allNetsDistinct-def*)
apply (*rule allI*)+
apply (*rule impI*)+
apply (*drule-tac x* = *aa* **in** *spec*, *drule-tac x* = *ba* **in** *spec*)
apply (*metis subsetD aNDSubsetaux*)
done

lemma *aNDSetsEq*: ⟦*singleCombinators a*; *singleCombinators b*; *set a* = *set b*;

allNetsDistinct b⟧ $\Longrightarrow$ *allNetsDistinct a*
apply (*simp add*: *allNetsDistinct-def*)
apply (*rule allI*)+
apply (*rule impI*)+
apply (*drule-tac x* = *aa* **in** *spec*, *drule-tac x* = *ba* **in** *spec*)
apply (*metis aNDSetsEqaux mem-def*)
done

lemma *SCConca*: ⟦*singleCombinators p*; *singleCombinators* [*a*]⟧ $\Longrightarrow$ *singleCombinators* (*a*#*p*)
by (*case-tac a*,*simp-all*)

lemma *aux3*[*simp*]: ⟦*singleCombinators p*; *singleCombinators* [*a*]; *allNetsDistinct* (*a*#*p*)⟧ $\Longrightarrow$ *allNetsDistinct* (*a*#*a*#*p*)
apply (*insert aNDSubset*[*of* (*a*#*a*#*p*) (*a*#*p*)])
apply (*simp add*: *SCConca*)
done

lemma *wp2-aux*[*rule-format*]: *wellformed-policy2* (*xs* @ [*x*]) $\longrightarrow$ *wellformed-policy2* *xs*
apply (*induct xs*, *simp-all*)
apply (*case-tac a*, *simp-all*)
done

lemma *wp1-aux1a*[*rule-format*]: *xs* $\neq$ [] $\longrightarrow$ *wellformed-policy1-strong* (*xs* @ [*x*]) $\longrightarrow$ *wellformed-policy1-strong xs*
by (*induct xs*,*simp-all*)

lemma *wp1alternative-RS1*[*rule-format*]: *DenyAll* $\in$ *set p* $\longrightarrow$ *wellformed-policy1-strong* (*removeShadowRules1 p*)
by (*induct p*,*simp-all*)

lemma *wellformed-eq*: *DenyAll* $\in$ *set p* $\longrightarrow$ ((*wellformed-policy1 p*) = (*wellformed-policy1-strong p*))
by (*induct p*,*simp-all*)

lemma *set-insort*: *set*(*insort x xs l*) = *insert x* (*set xs*)
by (*induct xs*) *auto*

lemma *set-sort*[*simp*]: *set*(*sort xs l*) = *set xs*
by (*induct xs*) (*simp-all add*:*set-insort*)

lemma *aux79*[*rule-format*]: *y* $\in$ *set* (*insort x a l*) $\longrightarrow$ *y* $\neq$ *x* $\longrightarrow$ *y* $\in$ *set a*
apply (*induct a*)
by *auto*

lemma *aux80*: ⟦*y* $\notin$ *set p*; *y* $\neq$ *x*⟧ $\Longrightarrow$ *y* $\notin$ *set* (*insort x* (*sort p l*) *l*)
apply (*metis aux79 set-sort*)
done

lemma *aux82*: (*insort DenyAll p l*) = *DenyAll#p*
by (*induct p*,*simp-all*)

lemma *WP1Conca*: *DenyAll* $\notin$ *set p* $\Longrightarrow$ *wellformed-policy1* (*a#p*)
by (*case-tac a*,*simp-all*)

lemma *Cdom2*: $x \in dom(C\ b) \Longrightarrow C\ (a \oplus b)\ x = (C\ b)\ x$
by (*auto simp*: *C.simps*)

lemma *wp2Conc*[*rule-format*]: *wellformed-policy2* (*x* # *xs*) $\Longrightarrow$ *wellformed-policy2 xs*
by (*case-tac x*,*simp-all*)

lemma *saux*[*simp*]: (*insort DenyAll p l*) = *DenyAll#p*
by (*induct-tac p*,*simp-all*)

lemma *saux3*[*rule-format*]: *DenyAllFromTo a b* $\in$ *set list* $\longrightarrow$ *DenyAllFromTo c d* $\notin$ *set list* $\longrightarrow$ ($a \neq c$) $\vee$ ($b \neq d$)
by *blast*

lemma *waux2*[*rule-format*]: (*DenyAll* $\notin$ *set xs*) $\longrightarrow$ *wellformed-policy1 xs*
by (*induct-tac xs*,*simp-all*)

lemma *waux3*[*rule-format*]: $[\![x \neq a;\ x \notin set\ p]\!] \Longrightarrow x \notin set\ (insort\ a\ p\ l)$
by (*metis aux79*)

lemma *wellformed1-sorted-aux*[*rule-format*]: *wellformed-policy1* (*x#p*) $\Longrightarrow$ *wellformed-policy1* (*insort x p l*)
apply (*case-tac x*,*simp-all*)
by (*rule waux2*,*rule waux3*, *simp-all*)+

lemma *SR1Subset*: *set* (*removeShadowRules1 p*) $\subseteq$ *set p*
apply (*induct-tac p*, *simp-all*)
apply (*case-tac a*, *simp-all*)
by *auto*

lemma *SCSubset*[*rule-format*]: *singleCombinators b* $\longrightarrow$ *set a* $\subseteq$ *set b* $\longrightarrow$*singleCombinators a*
apply (*induct-tac a*)
apply *auto*
apply (*case-tac a*)
apply *simp-all*
apply (*subgoal-tac Combinators1* $\oplus$ *Combinators2* $\in$ *set b* $\longrightarrow$ $\neg$ *singleCombinators b*, *simp*)
apply (*rule singleCombinators.induct*, *simp-all*)
done

lemma *setInsert*[*simp*]: *set list* $\subseteq$ *insert a* (*set list*)

by *auto*

lemma *SC-RS1*[*rule-format*,*simp*]: *singleCombinators p* $\longrightarrow$ *allNetsDistinct p* $\longrightarrow$ *singleCombinators* (*removeShadowRules1 p*)
apply (*induct-tac p*)
apply *simp-all*
apply (*rule impI*)+
apply (*drule mp*)
apply (*erule SCSubset*,*simp*)
by (*simp add*: *ANDConc*)

lemma *RS2Set*[*rule-format*]: *set* (*removeShadowRules2 p*) $\subseteq$ *set p*
apply (*induct p*, *simp-all*)
apply (*case-tac a*, *simp-all*)
apply *auto*
done

lemma *WP1*: *a* $\notin$ *set list* $\Longrightarrow$ *a* $\notin$ *set* (*removeShadowRules2 list*)
apply (*insert RS2Set* [*of list*])
apply *blast*
done

lemma *denyAllDom*[*simp*]: *x* $\in$ *dom* (*deny-all*)
by (*simp add*: *PLemmas*)

lemma *DAimpliesMR-E*[*rule-format*]: *DenyAll* $\in$ *set p* $\longrightarrow$ ($\exists$ *r*. *matching-rule x p* = *Some r*)
apply (*simp add*: *matching-rule-def*)
apply (*rule-tac xs* = *p* **in** *rev-induct*)
apply *simp-all*
by (*metis C.simps*(*1*) *denyAllDom*)

lemma *DAimplieMR*[*rule-format*]: *DenyAll* $\in$ *set p* $\Longrightarrow$ *matching-rule x p* $\neq$ *None*
by (*auto intro*: *DAimpliesMR-E*)

lemma *MRList1*[*rule-format*]: *x* $\in$ *dom* (*C a*) $\Longrightarrow$ *matching-rule x* (*b*@[*a*]) = *Some a*
by (*simp add*: *matching-rule-def*)

lemma *MRList2*: *x* $\in$ *dom* (*C a*) $\Longrightarrow$ *matching-rule x* (*c*@*b*@[*a*]) = *Some a*
by (*simp add*: *matching-rule-def*)

lemma *MRList3*: *x* $\notin$ *dom* (*C xa*) $\Longrightarrow$ *matching-rule x* (*a* @ *b* # *xs* @ [*xa*]) = *matching-rule x* (*a* @ *b* # *xs*)
by (*simp add*: *matching-rule-def*)

lemma *CConcEnd*[*rule-format*]: *C a x* = *Some y* $\longrightarrow$ *C* (*list2policy* (*xs* @ [*a*])) *x* = *Some y*
(**is** *?P xs*)

apply (*rule-tac P = ?P* **in** *list2policy.induct*)
by (*simp-all add:C.simps*)

lemma *CConcStartaux*: $[\![C\ a\ x = None]\!] \Longrightarrow (C\ aa\ {++}\ C\ a)\ x = C\ aa\ x$
by (*simp add*: *PLemmas*)

lemma *CConcStart*[*rule-format*]: $xs \neq [] \longrightarrow C\ a\ x = None \longrightarrow C$ (*list2policy* (*xs* @ [*a*])) *x* = *C* (*list2policy xs*) *x*
apply (*rule list2policy.induct*)
by (*simp-all add*: *PLemmas*)

lemma *mrNnt*[*simp*]: *matching-rule x p* = *Some a* $\Longrightarrow p \neq []$
apply (*simp add*: *matching-rule-def*)
by *auto*

lemma *mr-is-C*[*rule-format*]: *matching-rule x p* = *Some a* $\longrightarrow$ *C* (*list2policy* (*p*)) *x* = *C a x*
apply (*simp add*: *matching-rule-def*)
apply (*rule rev-induct*)
apply *simp-all*
apply *safe*
apply (*metis CConcEnd rotate-simps*)
apply (*metis CConcEnd*)
apply (*metis CConcStart domD domIff foldl-Nil matching-rule-rev.simps*(*2*) *option.simps*(*1*) *rev-foldl-cons rotate-simps*)
done

lemma *CConcStart2*: $[\![p \neq []; x \notin dom\ (C\ a)]\!] \Longrightarrow$ *C* (*list2policy* (*p*@[*a*])) *x* = *C* (*list2policy p*)*x*
by (*erule CConcStart*,*simp add*: *PLemmas*)

lemma *lCdom2*: (*list2policy* (*a* @ (*b* @ *c*))) = (*list2policy* ((*a*@*b*)@*c*))
by *auto*

lemma *CConcEnd1*: $[\![q@p \neq []; x \notin dom\ (C\ a)]\!] \Longrightarrow$ *C* (*list2policy* (*q*@*p*@[*a*])) *x* = *C* (*list2policy* (*q*@*p*))*x*
apply (*subst lCdom2*)
by (*rule CConcStart2*, *simp-all*)

lemma *CConcEnd2*[*rule-format*]: $x \in dom\ (C\ a) \longrightarrow$ *C* (*list2policy* (*xs* @ [*a*])) *x* = *C a x*
(**is** *?P xs*)
apply (*rule-tac P = ?P* **in** *list2policy.induct*)
by (*auto simp*:*C.simps*)

lemma *SCConcEnd*: *singleCombinators* (*xs* @ [*xa*]) $\Longrightarrow$ *singleCombinators xs*
by (*induct xs*, *simp-all*, *case-tac a*, *simp-all*)

lemma *bar3*: $x \in dom$ (*C* (*list2policy* (*xs* @ [*xa*]))) $\Longrightarrow x \in dom$ (*C* (*list2policy*

xs)) ∨ *x* ∈ *dom* (*C xa*)
by (*metis CConcEnd1 domIff list2policy.simps*(*1*) *rotate-simps self-append-conv2*)

lemma *CeqEnd*[*rule-format*,*simp*]: *a* ≠ [] ⟶ *x* ∈ *dom* (*C* (*list2policy a*)) ⟶
C (*list2policy* (*b*@*a*)) *x* = (*C* (*list2policy a*)) *x*
apply (*rule rev-induct*,*simp-all*)
apply (*case-tac xs* ≠ [], *simp-all*)
apply (*case-tac x* ∈ *dom* (*C xa*))
apply (*metis CConcEnd2 MRList2 mr-is-C rotate-simps*)
apply (*metis CConcEnd1 CConcStart2 Nil-is-append-conv bar3 rotate-simps*)
apply (*metis MRList2 eq-Nil-appendI mr-is-C rotate-simps*)
done

lemma *CConcStartA*[*rule-format*,*simp*]: *x* ∈ *dom* (*C a*) ⟶ *x* ∈ *dom* (*C* (*list2policy* (*a* # *b*)))
(**is** *?P b*)
apply (*rule-tac P* = *?P* **in** *list2policy.induct*)
apply (*simp-all add*: *C.simps*)
done

lemma *list2policyconc*[*rule-format*]: *a* ≠ [] ⟶ (*list2policy* (*xa* # *a*)) = (*xa*) ⊕ (*list2policy a*)
by (*induct a*,*simp-all*)

lemma *domConc*: ⟦*x* ∈ *dom* (*C* (*list2policy b*)); *b* ≠ []⟧ ⟹ *x* ∈ *dom* (*C* (*list2policy* (*a*@*b*)))
by (*auto simp*: *PLemmas*)

lemma *CeqStart*[*rule-format*,*simp*]: *x* ∉ *dom* (*C* (*list2policy a*)) ⟶ *a* ≠ [] ⟶ *b* ≠ [] ⟶
C (*list2policy* (*b*@*a*)) *x* = (*C* (*list2policy b*)) *x*
apply (*rule list2policy.induct*,*simp-all*)
apply (*auto simp*: *list2policyconc PLemmas*)
done

lemma *C-eq-if-mr-eq2*: ⟦*matching-rule x a* = *Some r*; *matching-rule x b* = *Some r*; *a* ≠ []; *b* ≠ []⟧ ⟹
(*C* (*list2policy a*)) *x* = (*C* (*list2policy b*)) *x*
by (*metis mr-is-C*)

lemma *nMRtoNone*[*rule-format*]: *p* ≠ [] ⟶ *matching-rule x p* = *None* ⟶ *C* (*list2policy p*) *x* = *None*
apply (*rule rev-induct*, *simp-all*)
apply (*case-tac xs* = [], *simp-all*)
by (*simp-all add*: *matching-rule-def dom-def*)

lemma *C-eq-if-mr-eq*:
⟦*matching-rule x b* = *matching-rule x a*; *a* ≠ []; *b* ≠ []⟧ ⟹
(*C* (*list2policy a*)) *x* = (*C* (*list2policy b*)) *x*

apply (*cases matching-rule x a* = *None*)
apply *simp-all*
apply (*subst nMRtoNone*)
apply (*simp-all*)
apply (*subst nMRtoNone*)
apply *simp-all*
by (*auto intro*: *C-eq-if-mr-eq2*)

lemma *wp1n-tl* [*rule-format*]: *wellformed-policy1-strong p* $\longrightarrow$ *p* = (*DenyAll*#(*tl p*))
by (*induct p*, *simp-all*)

lemma *foo2*: $[\![$*a* $\notin$ *set ps*; *a* $\notin$ *set ss*; *set p* = *set s*; *p* = (*a*#(*ps*)); *s* = (*a*#*ss*)$]\!]$ $\Longrightarrow$ *set* (*ps*) = *set* (*ss*)
by *auto*

lemma *bar5*: *matching-rule x* (*p*@[*a*]) $\neq$ *Some a* $\Longrightarrow$ *x* $\notin$ *dom* (*C a*)
by (*simp add*: *matching-rule-def split*: *if-splits*)

lemma *foo3a*[*rule-format*]: *matching-rule x* (*a*@[*b*]@*c*) = *Some b* $\longrightarrow$ *r* $\in$ *set c* $\longrightarrow$ *b* $\notin$ *set c* $\longrightarrow$ *x* $\notin$ *dom* (*C r*)
apply (*rule rev-induct*)
apply *simp-all*
apply (*rule impI*|*rule conjI*|*simp*)+
apply (*rule-tac p* = *a* @ *b* # *xs* **in** *bar5*,*simp-all*)
apply (*rule impI*,*simp*)+
apply (*drule sym*,*drule mp*, *simp-all*)
apply (*rule MRList3*[*symmetric*],*drule sym*)
apply (*rule-tac p* = *a* @ *b* # *xs* **in** *bar5*,*simp-all*)
done

lemma *foo3D*: $[\![$*wellformed-policy1 p*; *p* = (*DenyAll*#*ps*); *matching-rule x p* = *Some DenyAll*; *r* $\in$ *set ps*$]\!]$ $\Longrightarrow$ *x* $\notin$ *dom* (*C r*)
by (*rule-tac a* = [] **and** *b* = *DenyAll* **and** *c* = *ps* **in** *foo3a*, *simp-all*)

lemma *foo4*[*rule-format*]: *set p* = *set s* $\wedge$ ($\forall$ *r*. *r* $\in$ *set p* $\longrightarrow$ *x* $\notin$ *dom* (*C r*)) $\longrightarrow$ ($\forall$ *r* .*r* $\in$ *set s* $\longrightarrow$ *x* $\notin$ *dom* (*C r*))
by *simp*

lemma *foo5b*[*rule-format*]: *x* $\in$ *dom* (*C b*) $\longrightarrow$ ($\forall$ *r*. *r* $\in$ *set c* $\longrightarrow$ *x* $\notin$ *dom* (*C r*)) $\longrightarrow$
matching-rule x (*b*#*c*) = *Some b*
apply (*simp add*: *matching-rule-def*)
apply (*rule-tac xs* = *c* **in** *rev-induct*, *simp-all*)
done

lemma *mr-first*: $[\![$*x* $\in$ *dom* (*C b*); ($\forall$ *r*. *r* $\in$ *set c* $\longrightarrow$ *x* $\notin$ *dom* (*C r*)); *s* = *b*#*c*$]\!]$ $\Longrightarrow$
matching-rule x s = *Some b*

by (*simp add*: *foo5b*)

lemma *mr-charn*[*rule-format*]: $a \in set\ p \longrightarrow (x \in dom\ (C\ a)) \longrightarrow (\forall\ r.\ r \in set\ p \wedge x \in dom\ (C\ r) \longrightarrow r = a) \longrightarrow\ matching\text{-}rule\ x\ p = Some\ a$
apply (*rule-tac xs* = *p* **in** *rev-induct*)
by (*simp-all add*: *matching-rule-def*)

lemma *foo8*: $[\![(\forall\ r.\ r \in set\ p \wedge x \in dom\ (C\ r) \longrightarrow r = a);\ set\ p = set\ s]\!] \Longrightarrow (\forall\ r.\ r \in set\ s \wedge x \in dom\ (C\ r) \longrightarrow r = a)$
by *auto*

lemma *mrConcEnd*[*rule-format*]: $matching\text{-}rule\ x\ (b\ \#\ p) = Some\ a \longrightarrow a \neq b \longrightarrow matching\text{-}rule\ x\ p = Some\ a$
apply (*simp add*: *matching-rule-def*)
apply (*rule-tac xs* = *p* **in** *rev-induct*,*simp-all*)
by *auto*

lemma *wp3tl*[*rule-format*]: $wellformed\text{-}policy3\ p \longrightarrow wellformed\text{-}policy3\ (tl\ p)$
by (*induct p*, *simp-all*, *case-tac a*, *simp-all*)

lemma *wp3Conc*[*rule-format*]: $wellformed\text{-}policy3\ (a\#p) \longrightarrow wellformed\text{-}policy3\ p$
by (*induct p*, *simp-all*, *case-tac a*, *simp-all*)

lemma *SCnotConc*[*rule-format*,*simp*]: $a \oplus b \in set\ p \longrightarrow singleCombinators\ p \longrightarrow False$
by (*induct p*, *simp-all*, *case-tac aa*, *simp-all*)

lemma *foo98*[*rule-format*]:$matching\text{-}rule\ x\ (aa\ \#\ p) = Some\ a \longrightarrow x \in dom\ (C\ r) \longrightarrow$
$r \in set\ p$
$\longrightarrow a \in set\ p$
apply (*simp add*: *matching-rule-def*)
apply (*rule rev-induct*)
apply *simp-all*
apply (*case-tac r* = *xa*, *simp-all*)
done

lemma *auxx8*: $removeShadowRules1\text{-}alternative\text{-}rev\ [x] = [x]$
by (*case-tac x*, *simp-all*)

lemma *RS1End*[*rule-format*]: $x \neq DenyAll \longrightarrow removeShadowRules1\ (xs\ @\ [x]) = (removeShadowRules1\ xs)@[x]$
by (*induct-tac xs*, *simp-all*)

lemma *aux114*: $x \neq DenyAll \Longrightarrow removeShadowRules1\text{-}alternative\text{-}rev\ (x\#xs) = x\#(removeShadowRules1\text{-}alternative\text{-}rev\ xs)$
apply (*induct-tac xs*)
apply (*auto simp*: *auxx8*)
by (*case-tac x*, *simp-all*)

lemma *aux115*[*rule-format*]: $x \neq DenyAll \Longrightarrow$ *removeShadowRules1-alternative* $(xs@[x]) = ($*removeShadowRules1-alternative xs*$)@[x]$
apply (*simp add*: *removeShadowRules1-alternative-def aux114*)
done

lemma *RS1-DA*[*simp*]: *removeShadowRules1* (*xs* @ [*DenyAll*]) = [*DenyAll*]
by (*induct-tac xs*, *simp-all*)

lemma *rSR1-eq*: *removeShadowRules1-alternative* = *removeShadowRules1*
apply (*rule ext*)
apply (*simp add*: *removeShadowRules1-alternative-def*)
apply (*rule-tac xs* = *x* **in** *rev-induct*)
apply *simp-all*
apply (*case-tac xa* = *DenyAll*, *simp-all*)
apply (*metis RS1End aux114 rev.simps*)
done

lemma *mrMTNone*[*simp*]: *matching-rule x* [] = *None*
by (*simp add*: *matching-rule-def*)

lemma *DAAux*[*simp*]: $x \in$ *dom* (*C DenyAll*)
by (*simp add*: *dom-def PolicyCombinators.PolicyCombinators C.simps*)

lemma *mrSet*[*rule-format*]: *matching-rule x p* = *Some r* $\longrightarrow r \in$ *set p*
apply (*simp add*: *matching-rule-def*)
apply (*rule-tac xs=p* **in** *rev-induct*)
apply *simp-all*
done

lemma *mr-not-Conc*: *singleCombinators p* $\Longrightarrow$ *matching-rule x p* $\neq$ *Some* $(a \oplus b)$
apply (*auto simp*: *mrSet*)
apply (*drule mrSet*)
apply (*erule SCnotConc*,*simp*)
done

lemma *foo25*[*rule-format*]: *wellformed-policy3* $(p@[x]) \longrightarrow$ *wellformed-policy3 p*
by (*induct p*, *simp-all*, *case-tac a*, *simp-all*)

lemma *mr-in-dom*[*rule-format*]: *matching-rule x p* = *Some a* $\longrightarrow x \in$ *dom* (*C a*)
apply (*rule-tac xs* = *p* **in** *rev-induct*)
by (*auto simp*: *matching-rule-def*)

lemma *domInterMT*[*rule-format*]: $[\![dom\ a \cap dom\ b = \{\}; x \in dom\ a]\!] \Longrightarrow x \notin dom\ b$
by *auto*

lemma *wp3EndMT*[*rule-format*]: *wellformed-policy3* $(p@[xs]) \longrightarrow$ *AllowPortFromTo a b po* $\in$ *set p* $\longrightarrow$

$dom\ (C\ (AllowPortFromTo\ a\ b\ po)) \cap dom\ (C\ xs) = \{\}$
apply (*induct p*,*simp-all*)
apply (*rule impI*)+
apply (*drule mp*)
apply (*erule wp3Conc*)
by *clarify auto*

lemma *foo29*: $[\![dom\ (C\ a) \neq \{\};\ dom\ (C\ a) \cap dom\ (C\ b) = \{\}]\!] \Longrightarrow a \neq b$
by *auto*

lemma *foo28*: $[\![AllowPortFromTo\ a\ b\ po \in set\ p;\ dom\ (C\ (AllowPortFromTo\ a\ b\ po)) \neq \{\};\ (wellformed\text{-}policy3\ (p@[x]))]\!] \Longrightarrow$
$x \neq AllowPortFromTo\ a\ b\ po$
by (*metis foo29 C.simps wp3EndMT*)

lemma *foo28a*[*rule-format*]: $x \in dom\ (C\ a) \Longrightarrow dom\ (C\ a) \neq \{\}$
by *auto*

lemma *allow-deny-dom*[*simp*]: $dom\ (C\ (AllowPortFromTo\ a\ b\ po)) \subseteq dom\ (C\ (DenyAllFromTo\ a\ b))$
by (*simp-all add*: *twoNetsDistinct-def netsDistinct-def PLemmas*) *auto*

lemma *DenyAllowDisj*: $dom\ (C\ (AllowPortFromTo\ a\ b\ p)) \neq \{\} \Longrightarrow$
$dom\ (C\ (DenyAllFromTo\ a\ b)) \cap dom\ (C\ (AllowPortFromTo\ a\ b\ p)) \neq \{\}$
by (*metis Int-absorb1 allow-deny-dom*)

lemma *domComm*: $dom\ a \cap dom\ b = dom\ b \cap dom\ a$
by *auto*

lemma *foo31*: $[\![(\forall\ r.\ r \in set\ p \wedge x \in dom\ (C\ r) \longrightarrow (r = AllowPortFromTo\ a\ b\ po \vee r = DenyAllFromTo\ a\ b \vee r = DenyAll));$
$set\ p = set\ s]\!] \Longrightarrow$
$(\forall\ r.\ r \in set\ s \wedge x \in dom\ (C\ r) \longrightarrow (r = AllowPortFromTo\ a\ b\ po \vee r = DenyAllFromTo\ a\ b \vee r = DenyAll))$
by *auto*

lemma *r-not-DA-in-tl*[*rule-format*]: $wellformed\text{-}policy1\text{-}strong\ p \longrightarrow a \in set\ p \longrightarrow a \neq DenyAll \longrightarrow a \in set\ (tl\ p)$
by (*induct p*,*simp-all*)

lemma *wp1-aux1aa*[*rule-format*]: $wellformed\text{-}policy1\text{-}strong\ p \longrightarrow DenyAll \in set\ p$
by (*induct p*,*simp-all*)

lemma *mauxa*: $(\exists\ r.\ a\ b = Some\ r) = (a\ b \neq None)$
by *auto*

lemma *wp1-auxa*: $wellformed\text{-}policy1\text{-}strong\ p \Longrightarrow (\exists\ r.\ matching\text{-}rule\ x\ p = Some$

r)
apply (*rule DAimpliesMR-E*)
by (*erule wp1-aux1aa*)

lemma *l2p-aux*[*rule-format*]: *list* $\neq$ [] $\longrightarrow$ *list2policy* (*a* # *list*) = *a* $\oplus$(*list2policy* *list*)
by (*induct list*, *simp-all*)

lemma *l2p-aux2*[*rule-format*]: *list* = [] $\Longrightarrow$ *list2policy* (*a* # *list*) = *a*
by *simp*

lemma *deny-dom*[*simp*]: *twoNetsDistinct a b c d* $\Longrightarrow$ *dom* (*C* (*DenyAllFromTo* *a b*)) $\cap$ *dom* (*C* (*DenyAllFromTo c d*)) = {}
apply (*simp add*: *C.simps*)
by (*erule aux6*)

lemma *domTrans*: $[\![$*dom a* $\subseteq$ *dom b*; *dom*(*b*) $\cap$ *dom* (*c*) = {}$]\!]$ $\Longrightarrow$ *dom*(*a*) $\cap$ *dom*(*c*) = {}
by *auto*

lemma *DomInterAllowsMT*: $[\![$ *twoNetsDistinct a b c d*$]\!]$
$\Longrightarrow$ *dom* (*C* (*AllowPortFromTo a b p*)) $\cap$ *dom* (*C* (*AllowPortFromTo c d* *po*)) = {}
apply (*case-tac p* = *po*, *simp-all*)
apply (*rule-tac b* = *C* (*DenyAllFromTo a b*) **in** *domTrans*, *simp-all*)
apply (*metis domComm aux26 tNDComm*)
by (*simp add*: *twoNetsDistinct-def netsDistinct-def PLemmas*) *auto*

lemma *DomInterAllowsMT-Ports*: $[\![$*p* $\neq$ *po*$]\!]$
$\Longrightarrow$ *dom* (*C* (*AllowPortFromTo a b p*)) $\cap$ *dom* (*C* (*AllowPortFromTo c d* *po*)) = {}
by (*simp add*: *twoNetsDistinct-def netsDistinct-def PLemmas*) *auto*

lemma *aux7aa*: $[\![$*AllowPortFromTo a b poo* $\in$ *set p*; *allNetsDistinct* ((*AllowPortFromTo* *c d po*) # *p*); *a* $\neq$ *c* $\vee$ *b* $\neq$ *d*$]\!]$ $\Longrightarrow$ *twoNetsDistinct a b c d*
apply (*simp add*: *allNetsDistinct-def twoNetsDistinct-def*)
apply (*case-tac a* $\neq$ *c*)
apply (*rule disjI1*)
apply (*drule-tac x* = *a* **in** *spec*, *drule-tac x* = *c* **in** *spec*)
apply (*simp split*: *if-splits*)
apply (*simp-all add*: *ND0aux3*,*metis*)
apply (*rule disjI2*)
apply (*drule-tac x* = *b* **in** *spec*, *drule-tac x* = *d* **in** *spec*)
apply (*simp split*: *if-splits*)
apply (*metis ND0aux4 mem-def mem-iff*)+
done

lemma *wellformed-policy3-charn*[*rule-format*]: *singleCombinators p* $\longrightarrow$ *distinct* *p* $\longrightarrow$ *allNetsDistinct p* $\longrightarrow$ *wellformed-policy1 p* $\longrightarrow$ *wellformed-policy2 p* $\longrightarrow$

wellformed-policy3 p
apply (*induct-tac p*)
apply *simp-all*
apply *clarify*
apply *simp-all*
apply (*auto intro*: *singleCombinatorsConc ANDConc waux2 wp2Conc*)
apply (*case-tac a*)
apply *simp-all*
apply *clarify*
apply (*case-tac r*)
apply *simp-all*
apply (*metis Int-commute*)
apply (*metis DomInterAllowsMT aux7aa DomInterAllowsMT-Ports*)
apply (*metis aux0-0 mem-def*)
done

lemma *ANDConcEnd*: $\llbracket$ *allNetsDistinct* (*xs* @ [*xa*]); *singleCombinators xs*$\rrbracket$ $\Longrightarrow$ *allNetsDistinct xs*
by (*rule aNDSubset*) *auto*

lemma *WP1ConcEnd*[*rule-format*]: *wellformed-policy1* (*xs* @ [*xa*]) $\longrightarrow$ *wellformed-policy1 xs*
by (*induct xs*, *simp-all*)

lemma *NDComm*: *netsDistinct a b* = *netsDistinct b a*
by (*auto simp*: *netsDistinct-def in-subnet-def*)

lemma *DistinctNetsDenyAllow*:
$\llbracket$*DenyAllFromTo b c* $\in$ *set p*; *AllowPortFromTo a d po* $\in$ *set p*; *allNetsDistinct p*; *dom* (*C* (*DenyAllFromTo b c*)) $\cap$ *dom* (*C* (*AllowPortFromTo a d po*)) $\neq$ {}$\rrbracket$
$\Longrightarrow$ *b* = *a* $\wedge$ *c* = *d*
apply (*simp add*: *allNetsDistinct-def*)
apply (*frule-tac x* = *b* **in** *spec*)
apply (*drule-tac x* = *d* **in** *spec*)
apply (*drule-tac x* = *a* **in** *spec*)
apply (*drule-tac x* = *c* **in** *spec*)
apply (*metis Int-commute ND0aux1 ND0aux3 NDComm aux26 twoNetsDistinct-def ND0aux2 ND0aux4*)
done

lemma *DistinctNetsAllowAllow*:
$\llbracket$*AllowPortFromTo b c poo* $\in$ *set p*; *AllowPortFromTo a d po* $\in$ *set p*; *allNetsDistinct p*; *dom* (*C* (*AllowPortFromTo b c poo*)) $\cap$ *dom* (*C* (*AllowPortFromTo a d po*)) $\neq$ {}$\rrbracket$
$\Longrightarrow$ *b* = *a* $\wedge$ *c* = *d* $\wedge$ *poo* = *po*
apply (*simp add*: *allNetsDistinct-def*)
apply (*frule-tac x* = *b* **in** *spec*)
apply (*drule-tac x* = *d* **in** *spec*)
apply (*drule-tac x* = *a* **in** *spec*)

apply (*drule-tac x = c* **in** *spec*)
apply (*metis DomInterAllowsMT DomInterAllowsMT-Ports ND0aux3 ND0aux4 NDComm twoNetsDistinct-def*)
done

lemma *WP2RS2*[*simp*]:
⟦*singleCombinators p*;
distinct p;
allNetsDistinct p⟧ ⟹ *wellformed-policy2* (*removeShadowRules2 p*)
proof (*induct p*)
  case *Nil* **thus** *?case* **by** *simp*
next
  case (*Cons x xs*)
    have *wp-xs*: *wellformed-policy2* (*removeShadowRules2 xs*) **using** *prems* **by** (*metis ANDConc distinct.simps singleCombinatorsConc*)
    show *?case*
    proof (*cases x*)
      case *DenyAll* **thus** *?thesis* **using** *wp-xs* **by** *simp*
    next
      case (*DenyAllFromTo a b*) **thus** *?thesis*
        using *prems wp-xs* **by** (*simp*,*metis Cons DenyAllFromTo aux aux7 tNDComm mem-def deny-dom*)
    next
      case (*AllowPortFromTo a b p*) **thus** *?thesis*
        using *prems wp-xs* **by** (*simp*, *metis aux26 AllowPortFromTo Cons*(*4*) *aux aux7a mem-def tNDComm*)
    next
      case (*Conc a b*) **thus** *?thesis*
        using *prems* **by** (*metis Conc Cons*(*2*) *singleCombinators.simps*(*2*))
    qed
qed

lemma *wellformed1-sorted*[*simp*]:
  assumes *wp1*: *wellformed-policy1 p*
  shows *wellformed-policy1* (*sort p l*)
proof (*cases p*)
  case *Nil* **thus** *?thesis* **by** *simp*
next
  case (*Cons x xs*) **thus** *?thesis*
  proof (*cases x = DenyAll*)
    case *True* **thus** *?thesis* **using** *prems* **by** *simp*
    next
    case *False* **thus** *?thesis* **using** *prems*
    by (*metis Cons set-sort False waux2 wellformed-eq wellformed-policy1-strong.simps*(*2*))

  qed
qed

lemma *SC1*[*simp*]: *singleCombinators p* ⟹ *singleCombinators* (*removeShadowRules1*

p)
by (*erule SCSubset*) (*rule SR1Subset*)

lemma *SC2*[*simp*]: *singleCombinators p* $\Longrightarrow$ *singleCombinators* (*removeShadowRules2 p*)
by (*erule SCSubset*) (*rule RS2Set*)

lemma *SC3*[*simp*]: *singleCombinators p* $\Longrightarrow$ *singleCombinators* (*sort p l*)
by (*erule SCSubset*) *simp*

lemma *aND-RS1*[*simp*]: $[\![$*singleCombinators p*; *allNetsDistinct p*$]\!]$ $\Longrightarrow$ *allNetsDistinct* (*removeShadowRules1 p*)
apply (*rule aNDSubset*)
apply (*erule SC-RS1*, *simp-all*)
apply (*rule SR1Subset*)
done

lemma *aND-RS2*[*simp*]: $[\![$*singleCombinators p*; *allNetsDistinct p*$]\!]$ $\Longrightarrow$ *allNetsDistinct* (*removeShadowRules2 p*)
apply (*rule aNDSubset*)
apply (*erule SC2*, *simp-all*)
apply (*rule RS2Set*)
done

lemma *aND-sort*[*simp*]: $[\![$*singleCombinators p*; *allNetsDistinct p*$]\!]$ $\Longrightarrow$ *allNetsDistinct* (*sort p l*)
apply (*rule aNDSubset*)
by (*erule SC3*, *simp-all*)

lemma *inRS2*[*rule-format*,*simp*]: $x \notin$ *set p* $\longrightarrow$ $x \notin$ *set* (*removeShadowRules2 p*)
apply (*insert RS2Set* [*of p*])
by *blast*

lemma *distinct-RS2*[*rule-format*,*simp*]: *distinct p* $\longrightarrow$ *distinct* (*removeShadowRules2 p*)
apply (*induct p*)
apply *simp-all*
apply *clarify*
apply (*case-tac a*)
by *auto*

lemma *setPaireq*: $\{x, y\} = \{a, b\} \Longrightarrow x = a \wedge y = b \vee x = b \wedge y = a$
by (*metis Un-empty-left Un-insert-left doubleton-eq-iff*)

lemma *position-positive*[*rule-format*]: $a \in$ *set l* $\longrightarrow$ *position a l* > 0
by (*induct l*, *simp-all*)

lemma *pos-noteq*[*rule-format*]:

$a \in set\ l \longrightarrow b \in set\ l \longrightarrow c \in set\ l \longrightarrow a \neq b \longrightarrow$
$(position\ a\ l) <= (position\ b\ l) \longrightarrow$
$(position\ b\ l) <= (position\ c\ l) \longrightarrow$
$a \neq c$
apply (*induct l*)
apply *simp-all*
apply (*rule conjI*)
apply (*rule impI*)+
apply (*simp add*: *position-positive*)+
apply (*metis gr-implies-not0 position-positive*)
done

lemma *setPair-noteq*: $\{a,b\} \neq \{c,d\} \Longrightarrow \neg ((a = c) \wedge (b = d))$
by *auto*

lemma *setPair-noteq-allow*: $\{a,b\} \neq \{c,d\} \Longrightarrow \neg ((a = c) \wedge (b = d) \wedge P)$
by *auto*

lemma *order-trans*:
$[\![$*in-list x l*; *in-list y l*; *in-list z l*; *singleCombinators* [*x*]; *singleCombinators* [*y*]; *singleCombinators* [*z*]; *smaller x y l*; *smaller y z l*$]\!] \Longrightarrow$
smaller x z l
apply (*case-tac x*)
apply *simp-all*
apply (*case-tac z*)
apply *simp-all*
apply (*case-tac y*)
apply *simp-all*
apply (*case-tac y*)
apply *simp-all*
apply (*rule conjI*|*rule impI*)+
apply (*rule setPaireq*,*simp*)
apply (*rule conjI*|*rule impI*)+
apply (*simp-all split*: *if-splits*)
apply *metis*
apply *metis*
apply (*simp add*: *setPair-noteq*)
apply (*rule impI*, *simp-all*)
apply (*erule setPaireq*)
apply (*rule impI*)
apply (*case-tac y*, *simp-all*)
apply (*simp-all split*: *if-splits*)
apply *metis*
apply (*simp-all add*: *setPair-noteq setPair-noteq-allow*)
apply (*case-tac z*)
apply *simp-all*
apply (*case-tac y*)
apply *simp-all*
apply (*case-tac y*)

apply *simp-all*
apply (*rule impI*|*rule conjI*)+
apply (*simp-all split*: *if-splits*)
apply (*simp add*: *setPair-noteq*)
apply (*erule pos-noteq*)
apply *simp-all*
apply (*rule impI*)
apply (*simp add*: *setPair-noteq*)
apply (*rule conjI*)
apply (*simp add*: *setPair-noteq-allow*)
apply (*erule pos-noteq*, *simp-all*)
apply (*rule impI*)
apply (*simp add*: *setPair-noteq-allow*)
apply (*rule impI*)
apply (*rule disjI2*)
apply (*case-tac y*, *simp-all*)
apply (*simp-all split*: *if-splits*)
apply *metis*
apply (*simp-all add*: *setPair-noteq-allow*)
done

lemma *sortedConcStart*[*rule-format*]:
sorted (*a* # *aa* # *p*) *l* ⟶ *in-list a l* ⟶ *in-list aa l* ⟶ *all-in-list p l* ⟶ *singleCombinators* [*a*] ⟶ *singleCombinators* [*aa*] ⟶ *singleCombinators p* ⟶ *sorted* (*a*#*p*) *l*
apply (*induct p*)
apply *simp-all*
apply (*rule impI*)+
apply *simp*
apply (*rule-tac y* = *aa* **in** *order-trans*)
apply *simp-all*
apply (*case-tac ab*, *simp-all*)
done

lemma *singleCombinatorsStart*[*simp*]: *singleCombinators* (*x*#*xs*) ⟹ *singleCombinators* [*x*]
by (*case-tac x*, *simp-all*)

lemma *sorted-is-smaller*[*rule-format*]: *sorted* (*a* # *p*) *l* ⟶ *in-list a l* ⟶ *in-list b l* ⟶ *all-in-list p l* ⟶ *singleCombinators* [*a*] ⟶ *singleCombinators p* ⟶ *b* ∈ *set p* ⟶ *smaller a b l*
apply (*induct p*)
apply (*auto intro*: *singleCombinatorsConc sortedConcStart*)
done

lemma *sortedConcEnd*[*rule-format*]: *sorted* (*a* # *p*) *l* ⟶ *in-list a l* ⟶ *all-in-list p l* ⟶ *singleCombinators* [*a*] ⟶ *singleCombinators p* ⟶ *sorted p l*
apply (*induct p*)

apply (*auto intro*: *singleCombinatorsConc sortedConcStart*)
done

lemma *AD-aux*: $[\![$*AllowPortFromTo a b po* $\in$ *set p* ;*DenyAllFromTo c d* $\in$ *set p*;
allNetsDistinct p ; *singleCombinators p*;
$a \neq c \vee b \neq d]\!]$
$\Longrightarrow$ *dom* (*C* (*AllowPortFromTo a b po*)) $\cap$ *dom* (*C* (*DenyAllFromTo c d*)) = {}
apply (*rule aux26*)
apply (*rule-tac x* = *AllowPortFromTo a b po* **and** *y* = *DenyAllFromTo c d* **in** *tND*)
apply *auto*
done

lemma *in-set-in-list*[*rule-format*]: *a* $\in$ *set p* $\longrightarrow$ *all-in-list p l* $\longrightarrow$ *in-list a l*
by (*induct p*) *auto*

lemma *sorted-WP2*[*rule-format*]: *sorted p l* $\longrightarrow$ *all-in-list p l* $\longrightarrow$ *distinct p* $\longrightarrow$ *allNetsDistinct p* $\longrightarrow$ *singleCombinators p* $\longrightarrow$ *wellformed-policy2 p*
proof (*induct p*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons a p*) **thus** *?case*
proof (*cases a*)
case *DenyAll* **thus** *?thesis* **using** *prems* **by** (*auto intro*: *ANDConc singleCombinatorsConc sortedConcEnd*)
next
case (*DenyAllFromTo c d*) **thus** *?thesis* **using** *prems*
apply *simp*
apply (*rule impI*)+
apply (*rule conjI*)
apply (*rule allI*)+
apply (*rule impI*)+
apply (*rule deny-dom*)
apply (*auto intro*: *aux7 tNDComm ANDConc singleCombinatorsConc sortedConcEnd*)
done
next
case (*AllowPortFromTo c d e*) **thus** *?thesis* **using** *prems*
apply *simp*
apply (*rule impI*|*rule conjI*|*rule allI*)+
apply (*rule aux26*)
apply (*rule-tac x* = *AllowPortFromTo c d e* **and** *y* = *DenyAllFromTo aa b* **in** *tND*)
apply (*assumption*,*simp-all*)
apply (*subgoal-tac smaller* (*AllowPortFromTo c d e*) (*DenyAllFromTo aa b*) *l*)
apply (*simp split*: *if-splits*)
apply *metis*

apply (*erule sorted-is-smaller*)
apply *simp-all*
apply (*metis List.set.simps*(*2*) *bothNet.simps*(*2*) *in-list.simps*(*2*) *in-set-in-list mem-def set-empty2*)
apply (*auto intro*: *aux7 tNDComm ANDConc singleCombinatorsConc sortedConcEnd*)
done
next
case (*Conc a b*) **thus** *?thesis* **using** *prems* **by** *simp*
qed
qed

lemma *sorted-Consb*[*rule-format*]: *all-in-list* (*x#xs*) *l* ⟶ *singleCombinators* (*x#xs*) ⟶ (*sorted xs l* & (*ALL y*:*set xs. smaller x y l*)) ⟶ (*sorted* (*x#xs*) *l*)
apply(*induct xs arbitrary*: *x*)
apply *simp*
apply (*auto simp*: *order-trans*)
done

lemma *sorted-Cons*: ⟦*all-in-list* (*x#xs*) *l*; *singleCombinators* (*x#xs*)⟧ ⟹ (*sorted xs l* & (*ALL y*:*set xs. smaller x y l*)) = (*sorted* (*x#xs*) *l*)
apply *auto*
apply (*rule sorted-Consb*, *simp-all*)
apply (*metis singleCombinatorsConc singleCombinatorsStart sortedConcEnd*)
apply (*erule sorted-is-smaller*)
apply (*auto intro*: *singleCombinatorsConc singleCombinatorsStart in-set-in-list*)
done

lemma *smaller-antisym*: ⟦¬ *smaller a b l*; *in-list a l*; *in-list b l*; *singleCombinators*[*a*]; *singleCombinators* [*b*]⟧ ⟹ *smaller b a l*
apply (*case-tac a*)
apply *simp-all*
apply (*case-tac b*)
apply *simp-all*
apply (*simp-all split*: *if-splits*)
apply (*rule setPaireq*)
apply *simp*
apply (*case-tac b*)
apply *simp-all*
apply (*simp-all split*: *if-splits*)
done

lemma *set-insort-insert*: *set* (*insort x xs l*) ⊆ *insert x* (*set xs*)
by (*induct xs*) (*auto simp*: *set-insert*)

lemma *all-in-listSubset*[*rule-format*]: *all-in-list b l* ⟶ *singleCombinators a* ⟶ *set a* ⊆ *set b* ⟶ *all-in-list a l*
by (*induct-tac a*) (*auto intro*: *in-set-in-list singleCombinatorsConc*)

lemma *singleCombinators-insort*: ⟦*singleCombinators* [*x*]; *singleCombinators xs*⟧ ⟹ *singleCombinators* (*insort x xs l*)
by (*metis SCSubset SCConca FWCompilationProof.set-insort set.simps*(*2*) *subset-refl*)

lemma *all-in-list-insort*: ⟦*all-in-list xs l*; *singleCombinators* (*x#xs*); *in-list x l*⟧ ⟹ *all-in-list* (*insort x xs l*) *l*
apply (*rule-tac b* = *x#xs* **in** *all-in-listSubset*)
apply *simp-all*
apply (*metis singleCombinatorsConc singleCombinatorsStart singleCombinators-insort*)
apply (*rule set-insort-insert*)
done

lemma *sorted-ConsA*:⟦*all-in-list* (*x#xs*) *l*; *singleCombinators* (*x#xs*)⟧ ⟹ (*sorted* (*x#xs*) *l*) = (*sorted xs l* & (*ALL y*:*set xs*. *smaller x y l*))
by (*metis sorted-Cons*)

lemma *is-in-insort*: *y* ∈ *set xs* ⟹ *y* ∈ *set* (*insort x xs l*)
by (*metis ListMem-iff insert mem-def set-insort set.simps*(*2*))

lemma *sorted-insorta*[*rule-format*]: *sorted* (*insort x xs l*) *l* ⟶ *all-in-list* (*x#xs*) *l* ⟶ *distinct* (*x#xs*) ⟶ *singleCombinators* [*x*] ⟶ *singleCombinators xs* ⟶ *sorted xs l*
apply (*induct xs*)
apply *simp-all*
apply (*rule impI*)+
apply *simp*
apply (*auto intro*: *is-in-insort sorted-ConsA set-insort singleCombinators-insort singleCombinatorsConc sortedConcEnd all-in-list-insort*)
apply (*metis sort.simps*(*2*) *set-sort SCSubset all-in-list-insort set-subset-Cons singleCombinators.simps*(*3*) *singleCombinatorsConc singleCombinatorsStart singleCombinators-insort sortedConcEnd*)
apply (*rule sorted-Consb*)
apply *simp-all*
apply (*rule ballI*)
apply (*rule-tac p* = *insort x xs l* **in** *sorted-is-smaller*)
apply (*auto intro*: *in-set-in-list all-in-listSubset singleCombinators-insort singleCombinatorsConc set-insort-insert is-in-insort*)
apply (*rule-tac b* = *x#xs* **in** *all-in-listSubset*)
apply *simp-all*
apply (*erule singleCombinators-insort*)
apply (*erule singleCombinatorsConc*)
apply (*rule set-insort-insert*)
done

lemma *sorted-insortb*[*rule-format*]: *sorted xs l* ⟶ *all-in-list* (*x#xs*) *l* ⟶ *distinct* (*x#xs*) ⟶ *singleCombinators* [*x*] ⟶ *singleCombinators xs* ⟶ *sorted* (*insort x xs l*) *l*
apply (*induct xs*)
apply *simp-all*

apply (*rule impI*)+
apply (*subgoal-tac sorted* (*FWCompilation.insort x xs l*) *l*)
apply *simp*
defer *1*
apply (*metis FWCompilationProof.sorted-Cons all-in-list.simps*(*2*) *singleCombinatorsConc*)
apply (*rule sorted-Consb*)
apply *simp-all*
apply *auto*
apply (*rule-tac b* = *x#xs* **in** *all-in-listSubset*)
apply *simp-all*
apply (*rule singleCombinators-insort*, *simp-all*)
apply (*erule singleCombinatorsConc*)
apply (*rule set-insort-insert*)
apply (*metis SCConca singleCombinatorsConc singleCombinatorsStart singleCombinators-insort*)
apply (*case-tac y* = *x*)
apply *simp-all*
apply (*rule smaller-antisym*)
apply *simp-all*
apply (*subgoal-tac y* ∈ *set xs*)
apply (*auto intro*: *in-set-in-list all-in-list-insort aux0-1 singleCombinatorsConc aux79 sorted-is-smaller smaller-antisym*)
done

lemma *sorted-insort*: ⟦*all-in-list* (*x#xs*) *l*; *distinct*(*x#xs*); *singleCombinators* [*x*]; *singleCombinators xs*⟧ ⟹
sorted (*insort x xs l*) *l* = *sorted xs l*
by (*auto intro*: *sorted-insorta sorted-insortb*)

lemma *distinct-insort*: *distinct* (*insort x xs l*) = (*x* ∉ *set xs* ∧ *distinct xs*)
by(*induct xs*)(*auto simp*:*set-insort*)

lemma *distinct-sort*[*simp*]: *distinct* (*sort xs l*) = *distinct xs*
by(*induct xs*)(*simp-all add*:*distinct-insort*)

lemma *sort-is-sorted*[*rule-format*]: *all-in-list p l* ⟶ *distinct p* ⟶ *singleCombinators p* ⟶ *sorted* (*sort p l*) *l*
apply (*induct p*)
apply (*auto intro*: *SC3 all-in-listSubset SC3 singleCombinatorsConc sorted-insort*)
apply (*subst sorted-insort*)
apply (*auto intro*: *singleCombinatorsConc all-in-listSubset SC3*)
apply (*erule all-in-listSubset*, *auto intro*: *SC3 singleCombinatorsConc sorted-insort*)
done

lemma *wellformed2-sorted*[*simp*]: ⟦*all-in-list p l*; *distinct p*; *allNetsDistinct p*; *singleCombinators p*⟧ ⟹ *wellformed-policy2* (*sort p l*)
apply (*rule sorted-WP2*)
apply (*erule sort-is-sorted*, *simp-all*)

apply (*erule all-in-listSubset*, *auto intro*: *SC3 singleCombinatorsConc sorted-insort*)
done

lemma *inSet-not-MT*: $a \in set\ p \Longrightarrow p \neq []$
by *auto*

lemma *C-DenyAll*[*simp*]: $C\ (list2policy\ (xs\ @\ [DenyAll]))\ x = Some\ (deny\ x)$
by (*auto simp*: *PLemmas*)

lemma *RS1n-assoc*: $x \neq DenyAll$
$\Longrightarrow removeShadowRules1\text{-}alternative\ xs\ @\ [x] = removeShadowRules1\text{-}alternative\ (xs\ @\ [x])$
by (*simp add*: *removeShadowRules1-alternative-def aux114*)

lemma *RS1n-nMT*[*rule-format*,*simp*]: $p \neq [] \longrightarrow removeShadowRules1\text{-}alternative\ p \neq []$
apply (*simp add*: *removeShadowRules1-alternative-def*)
apply (*rule-tac xs = p* **in** *rev-induct*, *simp-all*)
apply (*case-tac xs* = [], *simp-all*)
apply (*case-tac x*, *simp-all*)
apply (*rule-tac xs = xs* **in** *rev-induct*, *simp-all*)
apply (*case-tac x*, *simp-all*)+
done

lemma *RS1N-DA*[*simp*]: $removeShadowRules1\text{-}alternative\ (a@[DenyAll]) = [DenyAll]$
by (*simp add*: *removeShadowRules1-alternative-def*)

lemma *C-eq-RS1n*: $C(list2policy\ (removeShadowRules1\text{-}alternative\ p)) = C(list2policy\ p)$
apply (*case-tac p* = [])
apply *simp-all*
apply (*metis rSR1-eq removeShadowRules1.simps*(*2*))
apply (*rule rev-induct*)
apply (*metis rSR1-eq removeShadowRules1.simps*(*2*))
apply (*case-tac xs* = [], *simp-all*)
apply (*simp add*: *removeShadowRules1-alternative-def*)
apply (*case-tac x*, *simp-all*)
apply (*rule ext*)
apply (*case-tac x = DenyAll*)
apply (*simp-all add*: *C-DenyAll PLemmas*)
apply (*rule-tac t* = *removeShadowRules1-alternative* (*xs* @ [*x*]) **and** *s* = (*removeShadowRules1-alternative xs*)@[*x*] **in** *subst*)
apply (*erule RS1n-assoc*)
apply (*case-tac* $xa \in dom\ (C\ x)$)
apply *simp-all*
done

lemma *C-eq-RS1*[*simp*]: $p \neq [] \Longrightarrow C(list2policy\ (removeShadowRules1\ p)) = C(list2policy\ p)$

by (*metis rSR1-eq C-eq-RS1n*)

lemma *EX-MR-aux*[*rule-format*]: *matching-rule* x (*DenyAll* # p) $\neq$ *Some DenyAll* $\longrightarrow$ ($\exists$ y. *matching-rule* x p = *Some* y)
apply (*simp add*: *matching-rule-def*)
apply (*rule-tac xs* = p **in** *rev-induct*, *simp-all*)
done

lemma *EX-MR* : $[\![$*matching-rule* x p $\neq$ (*Some DenyAll*); p = *DenyAll*#*ps*$]\!] \Longrightarrow$
(*matching-rule* x p = *matching-rule* x *ps*)
apply *auto*
apply (*subgoal-tac matching-rule* x (*DenyAll*#*ps*) $\neq$ *None*)
apply *auto*
apply (*metis mrConcEnd the.simps*)
apply (*metis DAimpliesMR-E is-in-insort saux wellformed-policy1-strong.simps*(*2*) *wp1-auxa*)
done

lemma *mr-not-DA*: $[\![$*wellformed-policy1-strong* s; *matching-rule* x p = *Some* (*DenyAllFromTo* a *ab*); *set* p = *set* $s]\!] \Longrightarrow$
matching-rule x s $\neq$ *Some DenyAll*
apply (*subst wp1n-tl*, *simp-all*)
apply (*subgoal-tac* $x \in$ *dom* (*C* (*DenyAllFromTo* a *ab*)))
apply (*subgoal-tac DenyAllFromTo* a *ab* $\in$ *set* (*tl* s))
apply (*metis wp1n-tl foo98 wellformed-policy1-strong.simps*(*2*))
apply (*erule r-not-DA-in-tl*, *simp-all*)
apply (*subgoal-tac DenyAllFromTo* a *ab* $\in$ *set* p, *simp*)
apply (*erule mrSet*)
apply (*erule mr-in-dom*)
done

lemma *domsMT-notND-DD*: $[\![$*dom* (*C* (*DenyAllFromTo* a b)) $\cap$ *dom* (*C* (*DenyAllFromTo* c d)) $\neq$ {}$]\!] \Longrightarrow \neg$ *netsDistinct* a c
apply (*erule contrapos-nn*)
apply (*simp add*: *C.simps*)
apply (*rule aux6*)
apply (*simp add*: *twoNetsDistinct-def*)
done

lemma *WP1n-DA-notinSet*[*rule-format*]: *wellformed-policy1-strong* p $\longrightarrow$ *DenyAll* $\notin$ *set* (*tl* p)
by (*induct* p) (*simp-all*)

lemma *domsMT-notND-DD2*: $[\![$*dom* (*C* (*DenyAllFromTo* a b)) $\cap$ *dom* (*C* (*DenyAllFromTo* c d)) $\neq$ {}$]\!] \Longrightarrow \neg$ *netsDistinct* b d
apply (*erule contrapos-nn*)
apply (*simp add*: *C.simps*)
apply (*rule aux6*)

apply (*simp add*: *twoNetsDistinct-def*)
done

lemma *domsMT-notND-DD3*: $[\![x \in dom\ (C\ (DenyAllFromTo\ a\ b)); x \in dom\ (C\ (DenyAllFromTo\ c\ d))]\!] \Longrightarrow \neg\ netsDistinct\ a\ c$
apply (*rule domsMT-notND-DD*)
apply *auto*
done

lemma *domsMT-notND-DD4*: $[\![x \in dom\ (C\ (DenyAllFromTo\ a\ b)); x \in dom\ (C\ (DenyAllFromTo\ c\ d))]\!] \Longrightarrow \neg\ netsDistinct\ b\ d$
apply (*rule domsMT-notND-DD2*)
apply *auto*
done

lemma *NetsEq-if-sameP-DD*: $[\![allNetsDistinct\ p; u \in set\ p; v \in set\ p; u = (DenyAllFromTo\ a\ b); v = (DenyAllFromTo\ c\ d); x \in dom\ (C\ (u)); x \in dom\ (C\ (v))]\!] \Longrightarrow a = c \wedge b = d$
apply (*simp add*: *allNetsDistinct-def*)
apply (*metis ND0aux1 ND0aux2 domsMT-notND-DD3 domsMT-notND-DD4 mem-def*)
done

lemma *mt-sym*: $dom\ a \cap dom\ b = \{\} \Longrightarrow dom\ b \cap dom\ a = \{\}$
by *auto*

lemma *rule-charn1*:
 assumes *aND*: *allNetsDistinct p*
 and *mr-is-allow*: *matching-rule x p* = *Some* (*AllowPortFromTo a b po*)
 and *SC*: *singleCombinators p*
 and *inp*: $r \in set\ p$
 and *inDom*: $x \in dom\ (C\ r)$
 shows ($r = AllowPortFromTo\ a\ b\ po \vee r = DenyAllFromTo\ a\ b \vee r = DenyAll$)
proof (*cases r*)
  case *DenyAll* **show** *?thesis* **using** *prems* **by** *simp*
next
  case (*DenyAllFromTo x y*) **show** *?thesis* **using** *prems*
    apply (*simp*,*rule-tac p* = *p* **and** *po* =*po* **in** *DistinctNetsDenyAllow*, *simp-all*)
    apply (*metis mrSet*)
    by (*metis Int-iff mr-in-dom inSet-not-MT mem-def set-empty2*)
next
  case (*AllowPortFromTo x y b*) **show** *?thesis* **using** *prems*
    apply *simp*
    apply (*rule DistinctNetsAllowAllow*, *simp-all*)
    apply (*metis mrSet*)
    by (*metis Int-iff mr-in-dom inSet-not-MT mem-def set-empty2*)
next
  case (*Conc x y*) **thus** *?thesis* **using** *prems* **by** (*metis aux0-0*)
qed

lemma *DAnotTL*[*rule-format*]: *xs* ≠ [] ⟶ *wellformed-policy1* (*xs* @ [*DenyAll*]) ⟶ *False*
by (*induct xs*, *simp-all*)

lemma *nMTRS3*[*simp*]: *noneMT* (*removeShadowRules3 p*)
by (*induct p*) *simp-all*

lemma *nMTcharn*: *noneMT p* = (∀ *r* ∈ *set p*. *dom* (*C r*) ≠ {})
by (*induct p*) *simp-all*

lemma *nMTeqSet*: *set p* = *set s* ⟹ *noneMT p* = *noneMT s*
by (*simp add*: *nMTcharn*)

lemma *nMTSort*: *noneMT p* ⟹ *noneMT* (*sort p l*)
by (*metis set-sort nMTeqSet*)

lemma *wp3char*[*rule-format*]: *noneMT xs* ∧ *dom* (*C* (*AllowPortFromTo a b po*)) ≠ {} ∧ *wellformed-policy3* (*xs* @ [*DenyAllFromTo a b*]) ⟶ *AllowPortFromTo a b po* ∉ *set xs*
apply (*induct xs*)
apply *simp-all*
apply (*metis wp3Conc Int-absorb1 Int-commute allow-deny-dom in-set-conv-decomp mem-def not-Cons-self removeShadowRules2.simps*(*1*) *set-empty2 wellformed-policy3.simps*(*2*))
done

lemma *wp3charn*[*rule-format*]:
assumes *domAllow*: *dom* (*C* (*AllowPortFromTo a b po*)) ≠ {}
and *wp3*: *wellformed-policy3* (*xs* @ [*DenyAllFromTo a b*])
shows *allowNotInList*: *AllowPortFromTo a b po* ∉ *set xs*
apply (*insert prems*)
proof (*induct xs*)
 case *Nil* **show** *?case* **by** *simp*
next
 case (*Cons x xs*) **show** *?case* **using** *prems*
 by (*simp*,*auto intro*: *wp3Conc*) (*auto simp*: *DenyAllowDisj domAllow*)
qed

lemma *notMTnMT*: ⟦*a* ∈ *set p*; *noneMT p*⟧ ⟹ *dom* (*C a*) ≠ {}
by (*simp add*: *nMTcharn*)

lemma *noneMTconc*[*rule-format*]: *noneMT* (*a*@[*b*]) ⟶ *noneMT a*
by (*induct a*, *simp-all*)

lemma *rule-charn2*:
 assumes *aND*: *allNetsDistinct p*
 and *wp1*: *wellformed-policy1 p*
 and *SC*: *singleCombinators p*
 and *wp3*: *wellformed-policy3 p*
 and *allow-in-list*: *AllowPortFromTo c d po* ∈ *set p*

and *x-in-dom-allow*: $x \in dom$ (*C* (*AllowPortFromTo c d po*))
shows *matching-rule x p* = *Some* (*AllowPortFromTo c d po*)
proof (*insert prems*, *induct p rule*: *rev-induct*)
  case *Nil* **show** *?case* **using** *prems* **by** *simp*
next
  case (*snoc y ys*) **show** *?case* **using** *prems*
    apply *simp*
    apply (*case-tac y* = (*AllowPortFromTo c d po*))
    apply (*simp add*: *matching-rule-def*)
    apply *simp-all*
    apply (*subgoal-tac ys* $\neq$ [])
    apply (*subgoal-tac matching-rule x ys* = *Some* (*AllowPortFromTo c d po*))
    defer *1*
    apply (*metis ANDConcEnd SCConcEnd WP1ConcEnd foo25 snoc*(*2*) *snoc*(*3*) *snoc*(*4*) *snoc*(*5*))
apply (*metis inSet-not-MT*)
    proof (*cases y*)
     case *DenyAll* **thus** *?thesis* **using** *prems*
        apply *simp*
       by (*metis DAnotTL DenyAll inSet-not-MT mem-def policy2list.simps*(*2*))
     next
     case (*DenyAllFromTo a b*) **thus** *?thesis* **using** *prems* **apply** *simp*
    apply (*simp-all add*: *matching-rule-def*)
    apply (*rule conjI*)
    apply (*metis domInterMT wp3EndMT*)
    apply (*rule impI*)
    by (*metis ANDConcEnd DenyAllFromTo SCConcEnd WP1ConcEnd foo25*)
     next
    case (*AllowPortFromTo a1 a2 b*) **thus** *?thesis* **using** *prems* **apply** *simp*
    apply (*simp-all add*: *matching-rule-def*)
    apply (*rule conjI*)
    apply (*metis domInterMT wp3EndMT*)
    by (*metis ANDConcEnd AllowPortFromTo SCConcEnd WP1ConcEnd foo25 x-in-dom-allow*)
    next
    case (*Conc a b*) **thus** *?thesis* **using** *prems* **apply** *simp*
     by (*metis Conc aux0-0 in-set-conv-decomp*)
    qed
qed

lemma *rule-charn3*:
  [[*wellformed-policy1 p*; *allNetsDistinct p*; *singleCombinators p*; *wellformed-policy3 p*;
    matching-rule x p = *Some* (*DenyAllFromTo c d*); *AllowPortFromTo a b po* $\in$ *set p*]] $\Longrightarrow$
     $x \notin dom$ (*C* (*AllowPortFromTo a b po*))
by (*clarify*, *auto simp*: *rule-charn2 dom-def*)

lemma *rule-charn4*:

assumes *wp1*: *wellformed-policy1 p*
and *aND*: *allNetsDistinct p*
and *SC*: *singleCombinators p*
and *wp3*: *wellformed-policy3 p*
and *DA*: *DenyAll* $\notin$ *set p*
and *mr*: *matching-rule x p* = *Some* (*DenyAllFromTo a b*)
and *rinp*: *r* $\in$ *set p*
and *xindom*: *x* $\in$ *dom* (*C r*)
shows *r* = *DenyAllFromTo a b*
proof (*cases r*)
  case *DenyAll* **thus** *?thesis* **using** *prems* **by** *simp*
next
  case (*DenyAllFromTo c d*) **thus** *?thesis* **using** *prems* **apply** *simp*
    apply (*erule-tac x* = *x* **and** *p* = *p* **and** *v* = (*DenyAllFromTo a b*) **and** *u* = (*DenyAllFromTo c d*) **in** *NetsEq-if-sameP-DD*)
    apply *simp-all*
    apply (*erule mrSet*)
    by (*erule mr-in-dom*)
next
  case (*AllowPortFromTo c d e*) **thus** *?thesis* **using** *prems* **apply** *simp*
    apply (*subgoal-tac x* $\notin$ *dom* (*C* (*AllowPortFromTo c d e*)))
    apply *simp*
    apply (*rule-tac p* = *p* **in** *rule-charn3*)
    by (*auto intro*: *SCnotConc*)
next
  case (*Conc a b*) **thus** *?thesis* **using** *prems* **apply** *simp*
    by (*metis Conc aux0-0 in-set-conv-decomp*)
qed

lemma *AND-tl*[*rule-format*]: *allNetsDistinct* (*p*) $\longrightarrow$ *allNetsDistinct* (*tl p*)
apply (*induct p*, *simp-all*)
by (*auto intro*: *ANDConc*)

lemma *distinct-tl*[*rule-format*]: *distinct p* $\longrightarrow$ *distinct* (*tl p*)
by (*induct p*, *simp-all*)

lemma *SC-tl*[*rule-format*]: *singleCombinators* (*p*) $\longrightarrow$ *singleCombinators* (*tl p*)
apply (*induct p*, *simp-all*)
by (*auto intro*: *singleCombinatorsConc*)

lemma *Conc-not-MT*: *p* = *x#xs* $\Longrightarrow$ *p* $\neq$ []
by *auto*

lemma *wp1-tl*[*rule-format*]: *p* $\neq$ [] $\wedge$ *wellformed-policy1 p* $\longrightarrow$ *wellformed-policy1* (*tl p*)
apply (*induct p*)

apply *simp-all*
apply (*auto intro*: *waux2*)
done

lemma *nMTtail*[*rule-format*]: $noneMT\ p \longrightarrow noneMT\ (tl\ p)$
by (*induct p*, *simp-all*)

lemma *foo31a*: $[\![(\forall\ r.\ r \in set\ p \wedge x \in dom\ (C\ r) \longrightarrow (r = AllowPortFromTo\ a\ b\ po \vee r = DenyAllFromTo\ a\ b \vee r = DenyAll));$
$set\ p = set\ s;\ r \in set\ s\ ;\ x \in dom\ (C\ r)]\!] \Longrightarrow (r = AllowPortFromTo\ a\ b\ po \vee r = DenyAllFromTo\ a\ b \vee r = DenyAll)$
by *auto*

lemma *wp1-eq*[*rule-format*]: $wellformed\text{-}policy1\text{-}strong\ p \Longrightarrow wellformed\text{-}policy1\ p$
apply (*case-tac DenyAll* $\in$ *set p*)
apply (*subst wellformed-eq*)
apply *simp-all*
apply (*erule waux2*)
done

lemma *aux4*[*rule-format*]:
$matching\text{-}rule\ x\ (a\#p) = Some\ a \longrightarrow a \notin set\ (p) \longrightarrow matching\text{-}rule\ x\ p = None$
apply (*rule rev-induct*)
apply *simp-all*
apply (*rule impI*)+
apply *simp*
apply (*simp add*: *matching-rule-def*)
apply (*simp split*: *if-splits*)
done

lemma *mrDA-tl*:
assumes *mr-DA*: $matching\text{-}rule\ x\ p = Some\ DenyAll$
and *wp1n*: *wellformed-policy1-strong p*
shows $matching\text{-}rule\ x\ (tl\ p) = None$
apply (*rule aux4* [**where** $a = DenyAll$])
apply (*metis wp1n-tl mr-DA wp1n*)
by (*metis WP1n-DA-notinSet wp1n*)

lemma *rule-charnDAFT*:
$[\![wellformed\text{-}policy1\text{-}strong\ p;\ allNetsDistinct\ p;\ singleCombinators\ p;\ wellformed\text{-}policy3\ p;$
$matching\text{-}rule\ x\ p = Some\ (DenyAllFromTo\ a\ b);\ r \in set\ (tl\ p);\ x \in dom\ (C\ r)]\!]$
$\Longrightarrow r = DenyAllFromTo\ a\ b$
apply (*subgoal-tac* $p = DenyAll\#(tl\ p)$)
apply (*rule-tac* $p = tl\ p$ **in** *rule-charn4*)
apply *simp-all*
apply (*metis wellformed-policy1-strong.simps*(*1*) *wp1-eq wp1-tl*)
apply (*erule AND-tl*)
apply (*erule SC-tl*)

apply (*erule wp3tl*)
apply (*erule WP1n-DA-notinSet*)
apply (*metis Combinators.simps*(*1*) *DAAux EX-MR matching-rule-def matching-rule-rev.simps*(*1*) *mem-def mrSet option.inject rev-rev-ident set-rev tl.simps*(*2*) *wellformed-policy1-charn wp1-eq*)
apply (*metis wp1n-tl*)
done

lemma *mrDenyAll-is-unique*: $[\![$*wellformed-policy1-strong p*; *matching-rule x p* = *Some DenyAll*; $r \in$ *set* (*tl p*)$]\!] \Longrightarrow x \notin$ *dom* (*C r*)
apply (*rule-tac a* = [] **and** *b* = *DenyAll* **and** *c* = *tl p* **in** *foo3a*, *simp-all*)
apply (*metis wp1n-tl*)
by (*metis WP1n-DA-notinSet*)

theorem *C-eq-Sets-mr*:
assumes *sets-eq*: *set p* = *set s*
and *SC*: *singleCombinators p*
and *wp1-p*: *wellformed-policy1-strong p*
and *wp1-s*: *wellformed-policy1-strong s*
and *wp3-p*: *wellformed-policy3 p*
and *wp3-s*: *wellformed-policy3 s*
and *aND*: *allNetsDistinct p*

shows *matching-rule x p* = *matching-rule x s*
proof (*cases matching-rule x p*)

case *None*
have *DA*: *DenyAll* $\in$ *set p* **using** *wp1-p* **by** (*auto simp*: *wp1-aux1aa*)
have *notDA*: *DenyAll* $\notin$ *set p* **using** *None* **by** (*auto simp*: *DAimplieMR*)
thus *?thesis* **using** *DA* **by** (*contradiction*)
next

case (*Some y*) **thus** *?thesis*
proof (*cases y*)
have *tl-p*: *p* = *DenyAll*#(*tl p*) **by** (*metis wp1-p wp1n-tl*)
have *tl-s*: *s* = *DenyAll*#(*tl s*) **by** (*metis wp1-s wp1n-tl*)
have *tl-eq*: *set* (*tl p*) = *set* (*tl s*)
by (*metis tl.simps*(*2*) *WP1n-DA-notinSet foo2 mem-def sets-eq wellformed-policy1-charn wp1-aux1aa wp1-eq wp1-p wp1-s*)
{
case *DenyAll*
have *mr-p-is-DenyAll*: *matching-rule x p* = *Some DenyAll* **by** (*simp add*: *DenyAll Some*)
hence *x-notin-tl-p*: $\forall$ *r*. $r \in$ *set* (*tl p*) $\longrightarrow$ $x \notin$ *dom* (*C r*) **using** *wp1-p* **by** (*auto simp*: *mrDenyAll-is-unique*)
hence *x-notin-tl-s*: $\forall$ *r*. $r \in$ *set* (*tl s*) $\longrightarrow$ $x \notin$ *dom* (*C r*) **using** *tl-eq* **by** *auto*
hence *mr-s-is-DenyAll*: *matching-rule x s* = *Some DenyAll* **using** *tl-s* **by** (*auto simp*: *mr-first*)
thus *?thesis* **using** *mr-p-is-DenyAll* **by** *simp*

}
{ **case** (*DenyAllFromTo a b*)
have *mr-p-is-DAFT*: *matching-rule x p* = *Some* (*DenyAllFromTo a b*) **by** (*simp add*: *DenyAllFromTo Some*)
have *DA-notin-tl*: *DenyAll* $\notin$ *set* (*tl p*) **by** (*metis WP1n-DA-notinSet wp1-p*)

have *mr-tl-p*: *matching-rule x p* = *matching-rule x* (*tl p*) **by** (*metis Combinators.simps*(*1*) *DenyAllFromTo Some mrConcEnd tl-p*)
have *dom-tl-p*: $\bigwedge$ *r*. *r* $\in$ *set* (*tl p*) $\wedge$ *x* $\in$ *dom* (*C r*) $\Longrightarrow$ *r* = (*DenyAllFromTo a b*) **using** *wp1-p aND SC wp3-p mr-p-is-DAFT*
by (*auto simp*: *rule-charnDAFT*)
hence *dom-tl-s*: $\bigwedge$ *r*. *r* $\in$ *set* (*tl s*) $\wedge$ *x* $\in$ *dom* (*C r*) $\Longrightarrow$ *r* = (*DenyAllFromTo a b*) **using** *tl-eq* **by** *auto*
have *DAFT-in-tl-s*: *DenyAllFromTo a b* $\in$ *set* (*tl s*) **using** *mr-tl-p* **by** (*metis DenyAllFromTo mrSet mr-p-is-DAFT tl-eq*)
have *x-in-dom-DAFT*: *x* $\in$ *dom* (*C* (*DenyAllFromTo a b*)) **by** (*metis mr-p-is-DAFT DenyAllFromTo mr-in-dom*)
hence *mr-tl-s-is-DAFT*: *matching-rule x* (*tl s*) = *Some* (*DenyAllFromTo a b*) **using** *DAFT-in-tl-s dom-tl-s* **by** (*auto simp*: *mr-charn*)
hence *mr-s-is-DAFT*: *matching-rule x s* = *Some* (*DenyAllFromTo a b*) **using** *tl-s*
by (*metis DA-notin-tl DenyAllFromTo EX-MR mrDA-tl mr-p-is-DAFT not-Some-eq tl-eq wellformed-policy1-strong.simps*(*2*))
thus *?thesis* **using** *mr-p-is-DAFT* **by** *simp*
}
{ **case** (*AllowPortFromTo a b c*)
have *wp1s*: *wellformed-policy1 s* **by** (*metis wp1-eq wp1-s*)
have *mr-p-is-A*: *matching-rule x p* = *Some* (*AllowPortFromTo a b c*) **by** (*simp add*: *AllowPortFromTo Some*)
hence *A-in-s*: *AllowPortFromTo a b c* $\in$ *set s* **using** *sets-eq* **by** (*auto intro*: *mrSet*)
have *x-in-dom-A*: *x* $\in$ *dom* (*C* (*AllowPortFromTo a b c*)) **by** (*metis mr-p-is-A AllowPortFromTo mr-in-dom*)
have *SCs*: *singleCombinators s* **using** *SC sets-eq* **by** (*auto intro*: *SCSubset*)
hence *ANDs*: *allNetsDistinct s* **using** *aND sets-eq SC* **by** (*auto intro*: *aND-SetsEq*)
hence *mr-s-is-A*: *matching-rule x s* = *Some* (*AllowPortFromTo a b c*) **using** *A-in-s wp1s mr-p-is-A aND SCs wp3-s x-in-dom-A*
by (*simp add*: *rule-charn2*)
thus *?thesis* **using** *mr-p-is-A* **by** *simp*
}
case (*Conc a b*) **thus** *?thesis* **by** (*metis Some mr-not-Conc SC*)
qed
qed

lemma *C-eq-Sets*:
$[\![$*singleCombinators p*; *wellformed-policy1-strong p*; *wellformed-policy1-strong s*; *wellformed-policy3 p*; *wellformed-policy3 s*;
allNetsDistinct p; *set p* = *set s*$]\!]$ $\Longrightarrow$

C (*list2policy p*) *x* = *C* (*list2policy s*) *x*
apply (*rule C-eq-if-mr-eq*)
apply (*rule C-eq-Sets-mr* [*symmetric*])
apply *simp-all*
apply (*metis wellformed-policy1-strong.simps*(*1*) *wp1-auxa*)+
done

lemma *wellformed1-alternative-sorted*: *wellformed-policy1-strong p* $\Longrightarrow$ *wellformed-policy1-strong* (*sort p l*)
by (*case-tac p*, *simp-all*)

lemma *C-eq-sorted*: $[\![$*distinct p*; *all-in-list p l*; *singleCombinators p*; *wellformed-policy1-strong p*; *wellformed-policy3 p*; *allNetsDistinct p*$]\!]$ $\Longrightarrow$
C (*list2policy* (*sort p l*))= *C* (*list2policy p*)
apply (*rule ext*)
apply (*rule C-eq-Sets*)
apply (*auto simp*: *nMTSort wellformed1-alternative-sorted wellformed-policy3-charn wellformed1-sorted wp1-eq*)
done

lemma *wp1n-RS2*[*rule-format*]: *wellformed-policy1-strong p* $\longrightarrow$ *wellformed-policy1-strong* (*removeShadowRules2 p*)
by (*induct p*, *simp-all*)

lemma *RS2-NMT*[*rule-format*]: *p* $\neq$ [] $\longrightarrow$ *removeShadowRules2 p* $\neq$ []
apply (*induct p*, *simp-all*)
apply (*case-tac p* $\neq$ [], *simp-all*)
apply (*case-tac a*, *simp-all*)+
done

lemma *mrconc*[*rule-format*]: *matching-rule x p* = *Some a* $\longrightarrow$ *matching-rule x* (*b#p*) = *Some a*
apply (*rule rev-induct*) **back**
apply (*simp*)
apply (*rule impI*)
apply (*case-tac x* $\in$ *dom* (*C xa*))
apply (*simp-all add*: *matching-rule-def*)
done

lemma *mreq-end*: $[\![$*matching-rule x b* = *Some r*; *matching-rule x c* = *Some r*$]\!]$ $\Longrightarrow$

matching-rule x (*a#b*) = *matching-rule x* (*a#c*)
by (*simp add*: *mrconc*)

lemma *mrconcNone*[*rule-format*]: *matching-rule x p* = *None* $\longrightarrow$ *matching-rule x* (*b#p*) = *matching-rule x* [*b*]
apply (*rule-tac xs* = *p* **in** *rev-induct*)
apply *simp-all*
apply (*rule impI*)

apply (*case-tac* $x \in$ *dom* (*C xa*))
apply (*simp-all add*: *matching-rule-def*)
done

lemma *mreq-endNone*: ⟦*matching-rule x b* = *None*; *matching-rule x c* = *None*⟧ $\Longrightarrow$
  matching-rule x (*a*#*b*) = *matching-rule x* (*a*#*c*)
by (*metis mrconcNone*)

lemma *mreq-end2*: *matching-rule x b* = *matching-rule x c* $\Longrightarrow$
  matching-rule x (*a*#*b*) = *matching-rule x* (*a*#*c*)
apply (*case-tac matching-rule x b* = *None*)
apply (*auto intro*: *mreq-end mreq-endNone*)
done

lemma *mreq-end3*: *matching-rule x p* $\neq$ *None* $\Longrightarrow$ *matching-rule x* (*b* # *p*) = *matching-rule x* (*p*)
by (*auto simp*: *mrconc*)

lemma *mrNoneMT*[*rule-format*]: *r* $\in$ *set p* $\longrightarrow$ *matching-rule x p* = *None* $\longrightarrow$ *x* $\notin$ *dom* (*C r*)
apply (*rule rev-induct*, *simp-all*)
apply (*rule conjI*| *rule impI*)+
apply *simp-all*
apply (*case-tac xa* $\in$ *set xs*)
apply (*simp-all add*: *matching-rule-def split*: *if-splits*)
done

lemma *C-eq-RS2-mr*: *matching-rule x* (*removeShadowRules2 p*)= *matching-rule x p*
proof (*induct p*)
 case *Nil* **thus** *?case* **by** *simp* **next**
 case (*Cons y ys*) **thus** *?case*
 proof (*cases ys* = [])
  case *True* **thus** *?thesis* **by** (*cases y*, *simp-all*) **next**
  case *False* **thus** *?thesis*
   proof (*cases y*)
    case *DenyAll* **thus** *?thesis* **by** (*simp*, *metis Cons DenyAll mreq-end2*) **next**
    case (*DenyAllFromTo a b*) **thus** *?thesis* **by** (*simp*, *metis Cons DenyAllFromTo mreq-end2*) **next**
    case (*AllowPortFromTo a b p*) **thus** *?thesis*
      proof (*cases DenyAllFromTo a b* $\in$ *set ys*)
        case *True* **thus** *?thesis* **using** *prems*
          apply (*cases matching-rule x ys* = *None*, *simp-all*)
          apply (*subgoal-tac x* $\notin$ *dom* (*C* (*AllowPortFromTo a b p*)))
          apply (*subst mrconcNone*, *simp-all*)
          apply (*simp add*: *matching-rule-def*)
          apply (*rule contra-subsetD* [*OF allow-deny-dom*])
          apply (*erule mrNoneMT*,*simp*)

apply (*metis AllowPortFromTo mrconc*)
done
next
case *False* **thus** *?thesis* **using** *prems* **by** (*simp*, *metis AllowPortFromTo Cons mreq-end2*) **qed**
next
case (*Conc a b*) **thus** *?thesis* **by** (*metis Cons mreq-end2 removeShadowRules2.simps*(*4*))
qed
qed
qed

lemma *wp1-alternative-not-mt*[*simp*]: *wellformed-policy1-strong p* $\Longrightarrow$ *p* $\neq$ []
by *auto*

lemma *C-eq-None*[*rule-format*]: *p* $\neq$ [] $-->$ *matching-rule x p* = *None* $\longrightarrow$ *C* (*list2policy p*) *x* = *None*
apply (*simp add*: *matching-rule-def*)
apply (*rule rev-induct*, *simp-all*)
apply (*rule impI*)+
apply *simp*
apply (*case-tac xs* $\neq$ [])
apply (*simp-all add*: *dom-def*)
done

lemma *C-eq-None2*: $[\![$*a* $\neq$ []; *b* $\neq$ []; *matching-rule x a* = *None*; *matching-rule x b* = *None*$]\!]$ $\Longrightarrow$
(*C* (*list2policy a*)) *x* = (*C* (*list2policy b*)) *x*
by (*auto simp*: *C-eq-None*)

lemma *C-eq-RS2*: *wellformed-policy1-strong p* $\Longrightarrow$
C (*list2policy* (*removeShadowRules2 p*))= *C* (*list2policy p*)
apply (*rule ext*)
apply (*rule C-eq-if-mr-eq*)
apply (*rule C-eq-RS2-mr* [*symmetric*], *simp-all*)
apply (*metis wp1-alternative-not-mt wp1n-RS2*)
done

lemma *AIL1*[*rule-format*,*simp*]: *all-in-list p l* $\longrightarrow$ *all-in-list* (*removeShadowRules1 p*) *l*
by (*induct-tac p*, *simp-all*)

lemma *noneMTsubset*[*rule-format*]: *noneMT a* $\longrightarrow$ *set b* $\subseteq$ *set a* $\longrightarrow$ *noneMT b*
by (*induct b*,*auto simp*: *notMTnMT*)

lemma *noneMTRS2*: *noneMT p* $\Longrightarrow$ *noneMT* (*removeShadowRules2 p*)
by (*auto simp*: *noneMTsubset RS2Set*)

lemma *CconcNone*: $[\![$*dom* (*C a*) = {}; *p* $\neq$ []$]\!]$ $\Longrightarrow$ *C* (*list2policy* (*a* # *p*)) *x* = *C* (*list2policy p*) *x*

apply (*case-tac p* = [], *simp-all*)
apply (*case-tac x*∈ *dom* (*C* (*list2policy*(*p*))))
apply (*metis Cdom2 list2policyconc mem-def*)
apply (*metis C.simps*(*4*) *Cauxb domIff inSet-not-MT list2policyconc set-empty2*)
done

lemma *notMTpolicyimpnotMT*[*simp*]: *notMTpolicy p* $\Longrightarrow$ *p* ≠ []
by *auto*

lemma *SR3nMT*[*rule-format*]: ¬ *notMTpolicy p* $\longrightarrow$ *removeShadowRules3 p* = []
by (*induct p*, *simp-all*)

lemma *wp1ID*: *wellformed-policy1-strong* (*insertDeny* (*removeShadowRules1 p*))
by (*induct p*, *simp-all*, *case-tac a*, *simp-all*)

lemma *noneMTrd*[*rule-format*]: *noneMT p* $\longrightarrow$ *noneMT* (*remdups p*)
by (*induct p*, *simp-all*)

lemma *DARS3*[*rule-format*]: *DenyAll* ∉ *set p* $\longrightarrow$ *DenyAll* ∉ *set* (*removeShadowRules3 p*)
by (*induct p*, *simp-all*)

lemma *DAnMT*: *dom* (*C DenyAll*) ≠ {}
by (*simp add*: *dom-def C.simps PolicyCombinators.PolicyCombinators*)

lemma *wp1n-RS3*[*rule-format*,*simp*]: *wellformed-policy1-strong p* $\longrightarrow$ *wellformed-policy1-strong* (*removeShadowRules3 p*)
apply (*induct p*, *simp-all*)
apply (*rule conjI*| *rule impI*|*simp*)+
apply (*metis DAAux inSet-not-MT set-empty2*)
apply (*rule conjI*| *rule impI*|*simp*)+
apply (*metis DARS3*)
done

lemma *dRD*[*simp*]: *distinct* (*remdups p*)
by *simp*

lemma *AILrd*[*rule-format*,*simp*]: *all-in-list p l* $\longrightarrow$ *all-in-list* (*remdups p*) *l*
by (*induct p*, *simp-all*)

lemma *AILRS3*[*rule-format*,*simp*]: *all-in-list p l* $\longrightarrow$ *all-in-list* (*removeShadowRules3 p*) *l*
by (*induct p*, *simp-all*)

lemma *AILiD*[*rule-format*,*simp*]: *all-in-list p l* $\longrightarrow$ *all-in-list* (*insertDeny p*) *l*
apply (*induct p*, *simp-all*)
apply (*rule impI*, *simp*)
apply (*case-tac a*, *simp-all*)
done

lemma *SCrd*[*rule-format*,*simp*]: *singleCombinators* *p* $\longrightarrow$ *singleCombinators*(*remdups* *p*)
apply (*induct* *p*, *simp-all*)
apply (*case-tac* *a*, *simp-all*)
done

lemma *SCRiD*[*rule-format*,*simp*]: *singleCombinators* *p* $\longrightarrow$ *singleCombinators*(*insertDeny* *p*)
apply (*induct* *p*, *simp-all*)
apply (*case-tac* *a*, *simp-all*)
done

lemma *SCRS3*[*rule-format*,*simp*]: *singleCombinators* *p* $\longrightarrow$ *singleCombinators*(*removeShadowRules3* *p*)
apply (*induct* *p*, *simp-all*)
apply (*case-tac* *a*, *simp-all*)
done

lemma *WP1rd*[*rule-format*,*simp*]: *wellformed-policy1-strong* *p* $\longrightarrow$ *wellformed-policy1-strong* (*remdups* *p*)
apply (*induct* *p*, *simp-all*)
done

lemma *ANDrd*[*rule-format*,*simp*]: *singleCombinators* *p* $\longrightarrow$ *allNetsDistinct* *p* $\longrightarrow$ *allNetsDistinct* (*remdups* *p*)
apply (*rule* *impI*)+
apply (*rule-tac* *b* = *p* **in** *aNDSubset*)
apply *simp-all*
done

lemma *RS3subset*: *set* (*removeShadowRules3* *p*) $\subseteq$ *set* *p*
by (*induct* *p*, *auto*)

lemma *ANDRS3*[*simp*]: $[\![$*singleCombinators* *p*; *allNetsDistinct* *p*$]\!]$ $\Longrightarrow$ *allNetsDistinct* (*removeShadowRules3* *p*)
apply (*rule-tac* *b* = *p* **in** *aNDSubset*)
apply *simp-all*
apply (*rule* *RS3subset*)
done

lemma *ANDiD*[*rule-format*,*simp*]: *allNetsDistinct* *p* $\longrightarrow$ *allNetsDistinct* (*insertDeny* *p*)
apply (*induct* *p*, *simp-all*)
apply (*simp* *add*: *allNetsDistinct-def*)
apply (*auto* *intro*: *ANDConc*)
apply (*case-tac* *a*)
apply (*simp-all* *add*: *allNetsDistinct-def*)
done

lemma *nlpaux*: $x \notin dom\ (C\ b) \Longrightarrow C\ (a \oplus b)\ x = C\ a\ x$
by (*simp add*: *C.simps Cauxb*)

lemma *notindom*[*rule-format*]: $a \in set\ p \longrightarrow\ x \notin dom\ (C\ (list2policy\ p)) \longrightarrow x \notin dom\ (C\ a)$
apply (*induct p*)
apply *simp-all*
apply (*rule conjI*| *rule impI*)+
apply (*metis CConcStartA*)
apply (*rule impI*)+
apply *simp*
apply (*metis CConcStartA Cdom2 domIff insert-absorb list.simps*(*1*) *list2policyconc set.simps*(*2*) *set-empty set-empty2*)
done

lemma *C-eq-rd*[*rule-format*]: $p \neq [] \Longrightarrow C\ (list2policy\ (remdups\ p)) = C\ (list2policy\ p)$
apply (*rule ext*)
proof (*induct p*)
  case *Nil* **thus** *?case* **by** *simp* **next**
  case (*Cons y ys*) **thus** *?case*
    proof (*cases ys* = [])
      case *True* **thus** *?thesis* **by** *simp* **next**
      case *False* **thus** *?thesis* **using** *prems* **apply** *simp*
       apply (*rule conjI*, *rule impI*)
       apply (*cases* $x \in dom\ (C\ (list2policy\ ys))$)
       apply (*metis Cdom2 False list2policyconc mem-def*)
       apply (*metis False domIff list2policyconc mem-def nlpaux notindom*)
       apply (*rule impI*)
       apply (*cases* $x \in dom\ (C\ (list2policy\ ys))$)
       apply (*subgoal-tac* $x \in dom\ (C\ (list2policy\ (remdups\ ys)))$)
       apply (*metis Cdom2 False list2policyconc mem-def remdups-eq-nil-iff*)
       apply (*metis domIff*)
       apply (*subgoal-tac* $x \notin dom\ (C\ (list2policy\ (remdups\ ys)))$)
       apply (*metis False list2policyconc nlpaux remdups-eq-nil-iff*)
       apply (*metis domIff*)
       done
  qed
qed

lemma *RS3nMT*[*rule-format*]: *notMTpolicy p* $\longrightarrow$ *notMTpolicy* (*removeShadowRules3 p*)
by (*induct p*,*simp-all*)

lemma *nMT-domMT*: $[\![\neg\ notMTpolicy\ p;\ p \neq []]\!] \Longrightarrow r \notin dom\ (C\ (list2policy\ p))$
 proof (*induct p*)
  case *Nil* **thus** *?case* **by** *simp* **next**

  case (*Cons x xs*) **thus** *?case* **apply** *simp*
    apply (*simp split*: *if-splits*)
    apply (*cases xs* = [])
    apply *simp-all*
    apply (*metis CconcNone domIff set-empty2*)
  done
qed

lemma *C-eq-RS3-aux*[*rule-format*]: *notMTpolicy p* $\Longrightarrow$ *C* (*list2policy p*) *x* = *C* (*list2policy* (*removeShadowRules3 p*)) *x*
proof (*induct p*)
 case *Nil* **thus** *?case* **by** *simp* **next**
 case (*Cons y ys*) **thus** *?case*
  proof (*cases notMTpolicy ys*)
   case *True* **thus** *?thesis* **using** *prems* **apply** *simp*
    apply (*rule conjI*, *rule impI*, *simp*)
    apply (*metis CconcNone True notMTpolicyimpnotMT set-empty2*)
    apply (*rule impI*, *simp*)
    apply (*cases x* $\in$ *dom* (*C* (*list2policy ys*)))
    apply (*subgoal-tac x* $\in$ *dom* (*C* (*list2policy* (*removeShadowRules3 ys*))))
     apply (*metis Cdom2 RS3nMT True list2policyconc mem-def notMTpolicy-impnotMT*)
    apply (*simp add*: *domIff*)
    apply (*subgoal-tac x* $\notin$ *dom* (*C* (*list2policy* (*removeShadowRules3 ys*))))
    apply (*metis RS3nMT True list2policyconc nlpaux notMTpolicyimpnotMT*)
    apply (*metis domIff*)
   done
   next
   case *False* **thus** *?thesis* **using** *prems*
     proof (*cases ys* = [])
      case *True* **thus** *?thesis* **using** *prems* **by** (*simp*) (*rule impI*, *simp*) **next**
      case *False* **thus** *?thesis* **using** *prems* **apply** (*simp*)
       apply (*rule conjI*| *rule impI*| *simp*)+
       apply (*subgoal-tac removeShadowRules3 ys* = [])
       apply *simp-all*
       apply (*subgoal-tac x* $\notin$ *dom* (*C* (*list2policy ys*)))
       apply (*metis False list2policyconc nlpaux*)
       apply (*erule nMT-domMT*, *simp-all*)
       by (*metis SR3nMT*)
     qed
   qed
qed

lemma *mr-iD*[*rule-format*]: *wellformed-policy1-strong p* $\longrightarrow$ *matching-rule x p* = *matching-rule x* (*insertDeny p*)
by (*induct p*, *simp-all*)

lemma *WP1iD*[*rule-format*,*simp*]: *wellformed-policy1-strong p* $\longrightarrow$ *wellformed-policy1-strong* (*insertDeny p*)

by (*induct p*, *simp-all*)

lemma *C-eq-id*: *wellformed-policy1-strong p* $\Longrightarrow$ *C*(*list2policy* (*insertDeny p*)) = *C* (*list2policy p*)
apply (*rule ext*)
apply (*rule C-eq-if-mr-eq*)
apply *simp-all*
apply (*erule mr-iD*)
done

lemma *C-eq-RS3*: *notMTpolicy p* $\Longrightarrow$ *C*(*list2policy* (*removeShadowRules3 p*)) = *C* (*list2policy p*)
apply (*rule ext*)
by (*erule C-eq-RS3-aux*[*symmetric*])

lemma *NMPcharn*[*rule-format*]: *a* $\in$ *set p* $\longrightarrow$ *dom* (*C a*) $\neq$ {} $\longrightarrow$ *notMTpolicy p*
by (*induct p*, *simp-all*)

lemma *NMPrd*[*rule-format*]: *notMTpolicy p* $\longrightarrow$ *notMTpolicy* (*remdups p*)
apply (*induct p*, *simp-all*)
by (*auto simp*: *NMPcharn*)

lemma *NMPRS3*[*rule-format*]: *notMTpolicy p* $\longrightarrow$ *notMTpolicy* (*removeShadowRules3 p*)
by (*induct p*, *simp-all*)

lemma *DAiniD*: *DenyAll* $\in$ *set* (*insertDeny p*)
by (*induct p*, *simp-all*, *case-tac a*, *simp-all*)

lemma *NMPDA*[*rule-format*]: *DenyAll* $\in$ *set p* $\longrightarrow$ *notMTpolicy p*
by (*induct p*, *simp-all add*: *DAnMT*)

lemma *NMPiD*[*rule-format*]: *notMTpolicy* (*insertDeny p*)
apply (*insert DAiniD* [*of p*])
apply (*erule NMPDA*)
done

lemma *p2lNmt*: *policy2list p* $\neq$ []
by (*rule policy2list.induct*, *simp-all*)

lemma *list2policy2list*[*rule-format*]: *C* (*list2policy*(*policy2list p*)) = (*C p*)
apply (*rule ext*)
apply (*induct-tac p*, *simp-all*)
apply (*case-tac x* $\in$ *dom* (*C* (*Combinators2*)))
apply (*metis Cdom2 CeqEnd domIff p2lNmt*)
apply (*metis CeqStart domIff p2lNmt nlpaux*)
done

lemma *AIL2*[*rule-format*,*simp*]: *all-in-list p l* $\longrightarrow$ *all-in-list* (*removeShadowRules2* *p*) *l*
by (*induct-tac p*, *simp-all*, *case-tac a*, *simp-all*)

lemmas *C-eq-Lemmas* = *noneMTRS2 noneMTrd dRD SC2 SCrd SCRS3 SCRiD SC1 aux0 wp1n-RS2 WP1rd WP2RS2 wp1n-RS3 wp1ID NMPiD wp1alternative-RS1 p2lNmt list2policy2list wellformed-policy3-charn waux2 wp1-eq*

lemmas *C-eq-subst-Lemmas* = *C-eq-sorted C-eq-RS2 C-eq-rd C-eq-RS3 C-eq-id*

lemma *C-eq-All-untilSorted*:
$[\![$*DenyAll* $\in$ *set* (*policy2list p*); *all-in-list* (*policy2list p*) *l*; *allNetsDistinct* (*policy2list* *p*)$]\!] \Longrightarrow$
C(*list2policy* (*sort* (*removeShadowRules2* (*remdups* (*removeShadowRules3* (*insertDeny* (*removeShadowRules1* (*policy2list p*)))))) *l*)) = *C p*
apply (*subst C-eq-sorted*)
apply (*simp-all add*: *C-eq-Lemmas*)
apply (*subst C-eq-RS2*)
apply (*simp-all add*: *C-eq-Lemmas*)
apply (*subst C-eq-rd*)
apply (*simp-all add*: *C-eq-Lemmas*)
apply (*subst C-eq-RS3*)
apply (*simp-all add*: *C-eq-Lemmas*)
apply (*subst C-eq-id*)
apply (*simp-all add*: *C-eq-Lemmas*)
done

lemma *C-eq-All-untilSorted-withSimps*:
$[\![$*DenyAll* $\in$ *set* (*policy2list p*); *all-in-list* (*policy2list p*) *l*; *allNetsDistinct* (*policy2list* *p*)$]\!] \Longrightarrow$
C(*list2policy* (*sort* (*removeShadowRules2* (*remdups* (*removeShadowRules3* (*insertDeny* (*removeShadowRules1* (*policy2list p*)))))) *l*)) = *C p*
by (*simp-all add*: *C-eq-Lemmas C-eq-subst-Lemmas*)

lemma *InDomConc*[*rule-format*]: $p \neq [] \longrightarrow x \in$ *dom* (*C* (*list2policy* (*p*))) $\longrightarrow x$ $\in$ *dom* (*C* (*list2policy* (*a#p*)))
apply (*induct p*)
apply *simp-all*
apply (*case-tac p* = [])
apply (*simp-all add*: *dom-def C.simps*)
done

lemma *not-in-member*[*rule-format*]: *member a b* $\longrightarrow x \notin$ *dom* (*C b*) $\longrightarrow x \notin$ *dom* (*C a*)
apply (*induct b*)
apply (*simp-all add*: *dom-def C.simps*)

done

lemma *subnetAux*: $D \cap A \neq \{\} \Longrightarrow A \subseteq B \Longrightarrow\ D \cap B \neq \{\}$
apply *auto*
done

lemma *soadisj*: $[\![x \in \mathit{subnetsOfAdr}\ a;\ y \in \mathit{subnetsOfAdr}\ a]\!] \Longrightarrow \neg\ \mathit{netsDistinct}\ x$ y
by (*simp add*: *subnetsOfAdr-def netsDistinct-def*, *auto simp*: *PLemmas*)

lemma *not-member*: $\neg$ *member a* $(x \oplus y) \Longrightarrow \neg$ *member a x*
apply *auto*
done

lemma *src-in-sdnets*[*rule-format*]: $\neg$ *member DenyAll x* $\longrightarrow$ $p \in$ *dom* (*C x*) $\longrightarrow$ *subnetsOfAdr* (*src p*) $\cap$ (*fst-set* (*sdnets x*)) $\neq \{\}$
apply (*induct rule*: *Combinators.induct*)
apply *simp*
apply (*simp add*: *fst-set-def subnetsOfAdr-def PLemmas*)
apply (*simp add*: *fst-set-def subnetsOfAdr-def PLemmas*)
apply (*rule impI*)+
apply (*simp add*: *fst-set-def*)
apply (*case-tac p* $\in$ *dom* (*C Combinators2*))
apply *simp-all*
apply (*rule subnetAux*)
apply *assumption*
apply (*auto simp*: *PLemmas*)
done

lemma *dest-in-sdnets*[*rule-format*]: $\neg$ *member DenyAll x* $\longrightarrow$ $p \in$ *dom* (*C x*) $\longrightarrow$ *subnetsOfAdr* (*dest p*) $\cap$ (*snd-set* (*sdnets x*)) $\neq \{\}$
apply (*induct rule*: *Combinators.induct*)
apply *simp*
apply (*simp add*: *snd-set-def subnetsOfAdr-def PLemmas*)
apply (*simp add*: *snd-set-def subnetsOfAdr-def PLemmas*)
apply (*rule impI*)+
apply (*simp add*: *snd-set-def*)
apply (*case-tac p* $\in$ *dom* (*C Combinators2*))
apply *simp-all*
apply (*rule subnetAux*)
apply *assumption*
apply (*auto simp*: *PLemmas*)
done

lemma *soadisj2*: ($\forall$ *a x y*. $x \in$ *subnetsOfAdr a* $\wedge$ $y \in$ *subnetsOfAdr a* $\longrightarrow \neg$ *netsDistinct x y*)
by (*simp add*: *subnetsOfAdr-def netsDistinct-def*, *auto simp*: *PLemmas*)

lemma *ndFalse1*: $[\![(\forall a\ b\ c\ d.\ (a,b) \in A\ \wedge (c,d) \in B \longrightarrow \mathit{netsDistinct}\ a\ c);$

$\exists (a, b) \in A.\ a \in subnetsOfAdr\ D;$
$\exists (a, b) \in B.\ a \in subnetsOfAdr\ D]\!]$
$\Longrightarrow False$
apply (*auto simp*: *soadisj*)
apply (*insert soadisj2*)
apply (*rotate-tac* -1, *drule-tac* $x = D$ **in** *spec*)
apply (*rotate-tac* -1, *drule-tac* $x = a$ **in** *spec*)
apply (*rotate-tac* -1, *drule-tac* $x = aa$ **in** *spec*)
by *auto*

lemma *ndFalse2*: $[\![(\forall\, a\ b\ c\ d.\ (a,b)\in A\ \wedge (c,d)\in B \longrightarrow netsDistinct\ b\ d);$
$\exists (a, b) \in A.\ b \in subnetsOfAdr\ D;$
$\exists (a, b) \in B.\ b \in subnetsOfAdr\ D]\!]$
$\Longrightarrow False$
apply (*auto simp*: *soadisj*)
apply (*insert soadisj2*)
apply (*rotate-tac* -1, *drule-tac* $x = D$ **in** *spec*)
apply (*rotate-tac* -1, *drule-tac* $x = b$ **in** *spec*)
apply (*rotate-tac* -1, *drule-tac* $x = ba$ **in** *spec*)
apply *simp*
apply *auto*
done

lemma *tndFalse*: $[\![(\forall\, a\ b\ c\ d.\ (a,b)\in A\ \wedge (c,d)\in B \longrightarrow twoNetsDistinct\ a\ b\ c\ d);$
$\exists (a, b) \in A.\ a \in subnetsOfAdr\ (D::('a::adr)) \wedge b \in subnetsOfAdr\ (F::'a);$
$\exists (a, b) \in B.\ a \in subnetsOfAdr\ D \wedge b \in subnetsOfAdr\ F]\!]$
$\Longrightarrow False$
apply (*simp add*: *twoNetsDistinct-def*)
apply (*auto simp*: *ndFalse1 ndFalse2*)
apply (*metis soadisj*)
done

lemma *sdnets-in-subnets*[*rule-format*]: $p\in dom\ (C\ x) \longrightarrow \neg\ member\ DenyAll\ x$
$\longrightarrow (\exists\ (a,b)\in sdnets\ x.\ a \in subnetsOfAdr\ (src\ p) \wedge b \in subnetsOfAdr\ (dest\ p))$
apply (*rule Combinators.induct*)
apply *simp-all*
apply (*simp add*: *PLemmas subnetsOfAdr-def*)
apply (*simp add*: *PLemmas subnetsOfAdr-def*)
apply (*rule impI*)+
apply *simp*
apply (*case-tac* $p \in dom\ (C\ (Combinators2)))$
apply *simp-all*
apply (*auto simp*: *PLemmas subnetsOfAdr-def*)
done

lemma *disjSD-no-p-in-both*[*rule-format*]:
$[\![disjSD\text{-}2\ x\ y;\ \neg\ member\ DenyAll\ x;\ \neg\ member\ DenyAll\ y;$
$p \in dom\ (C\ x);\ p \in dom\ (C\ y)]\!] \Longrightarrow False$
apply (*rule-tac* $A = sdnets\ x$ **and** $B = sdnets\ y$ **and** $D = src\ p$ **and** $F = dest\ p$

in *tndFalse*)
by (*auto simp*: *dest-in-sdnets src-in-sdnets sdnets-in-subnets disjSD-2-def*)

lemma *list2policy-eq*: $zs \neq [] \Longrightarrow C\ (list2policy\ (x \oplus y \ \#\ z))\ p = C\ (x \oplus list2policy\ (y \ \#\ z))\ p$
apply (*metis C.simps*(*4*) *CConcStartaux C-eq-None C-eq-RS3 C-eq-if-mr-eq C-eq-rd Cdom2 ConcAssoc domIff in-set-conv-decomp l2p-aux2 list.simps*(*1*) *list2policy.simps*(*2*) *list2policyconc map-add-None mem-def mrMTNone mrconcNone mreq-end3 mreq-endNone nlpaux not-Cons-self remdups.simps*(*2*) *removeShadowRules3.simps*(*2*) *self-append-conv2*)
done

lemma *sepnMT*[*rule-format*]: $p \neq [] \longrightarrow (separate\ p) \neq []$
apply (*rule separate.induct*) **back back back**
by *simp-all*

lemma *sepDA*[*rule-format*]: $DenyAll \notin set\ p \longrightarrow DenyAll \notin set\ (separate\ p)$
apply (*rule separate.induct*) **back**
apply *simp-all*
done

lemma *dom-sep*[*rule-format*]: $x \in dom\ (C\ (list2policy\ p)) \longrightarrow x \in dom\ (C\ (list2policy(separate\ p)))$
apply (*rule separate.induct*) **back**
apply *simp-all*
apply (*rule conjI*)
apply (*rule impI*)+
apply *simp*
apply (*thin-tac False* $\Longrightarrow$ *?S*)
apply (*drule mp*)
apply (*case-tac* $x \in dom\ (C\ (DenyAllFromTo\ v\ va))$)
apply (*metis CConcStartA domIff eq-Nil-appendI in-set-conv-decomp l2p-aux2 list2policyconc mem-def not-Cons-self notindom*)
apply (*subgoal-tac* $x \in dom\ (C\ (list2policy\ (y\ \#z)))$)
apply (*metis CConcStartA Cdom2 InDomConc domIff l2p-aux2 list2policyconc nlpaux*)
apply (*subgoal-tac* $x \in dom\ (C\ (list2policy\ ((DenyAllFromTo\ v\ va)\#y\#z)))$)
apply (*simp add*: *dom-def C.simps*)
apply *simp*
apply *simp*
apply (*rule impI*)+
apply *simp*
apply (*thin-tac False* $\Longrightarrow$ *?S*)
apply (*case-tac* $x \in dom\ (C\ (DenyAllFromTo\ v\ va))$)
apply *simp-all*
apply (*subgoal-tac* $x \in dom\ (C\ (list2policy\ (y\ \#z)))$)
apply (*metis InDomConc sepnMT list.simps*(*2*))
apply (*subgoal-tac* $x \in dom\ (C\ (list2policy\ ((DenyAllFromTo\ v\ va)\#y\#z)))$)
apply (*simp add*: *dom-def C.simps*)
apply *simp*

apply (*rule impI* | *rule conjI*)+
apply *simp*
apply (*case-tac* $x \in$ *dom* (*C* (*AllowPortFromTo v va vb*)))
apply (*metis CConcStartA domIff eq-Nil-appendI in-set-conv-decomp l2p-aux2 list2policyconc mem-def not-Cons-self notindom*)
apply (*subgoal-tac* $x \in$ *dom* (*C* (*list2policy* (*y* #*z*))))
apply *simp*
apply (*metis CConcStartA Cdom2 InDomConc domIff l2p-aux2 list2policyconc nlpaux*)
apply (*simp add*: *dom-def C.simps*)
apply (*rule impI*)+
apply *simp*
apply (*case-tac* $x \in$ *dom* (*C* (*AllowPortFromTo v va vb*)))
apply (*metis CConcStartA*)
apply (*metis CConcStartA InDomConc domIff list.simps*(*1*) *list2policy.simps*(*2*) *nlpaux sepnMT*)
apply (*rule conjI* | *rule impI*)+
apply *simp*
apply (*thin-tac False* $\Longrightarrow$ *?S*)
apply (*drule mp*)
apply (*case-tac* $x \in$ *dom* (*C* (($v \oplus va$))))
apply (*metis C.simps*(*4*) *CConcStartA ConcAssoc domIff eq-Nil-appendI in-set-conv-decomp list2policy2list list2policyconc mem-def notindom p2lNmt*)
defer *1*
apply *simp-all*
apply (*rule impI*)+
apply *simp*
apply (*thin-tac False* $\Longrightarrow$ *?S*)
apply (*case-tac* $x \in$ *dom* (*C* (($v \oplus va$))))
apply (*metis CConcStartA*)
apply (*drule mp*)
apply (*simp add*: *C.simps dom-def*)
apply (*metis InDomConc list.simps*(*1*) *mem-def sepnMT*)
apply (*subgoal-tac* $x \in$ *dom* (*C* (*list2policy* (*y*#*z*))))
apply (*case-tac* $x \in$ *dom* (*C y*))
apply *simp-all*
apply (*metis CConcStartA Cdom2 ConcAssoc domIff mem-def*)
apply (*metis InDomConc domIff l2p-aux2 list2policyconc nlpaux*)
apply (*case-tac* $x \in$ *dom* (*C y*))
apply *simp-all*
apply (*metis InDomConc domIff l2p-aux2 list2policyconc nlpaux*)
done

lemma *domdConcStart*[*rule-format*]: $x \in$ *dom* (*C* (*list2policy* (*a*#*b*))) $\longrightarrow$
$x \notin$ *dom* (*C* (*list2policy b*))
$\longrightarrow x \in$ *dom* (*C* (*a*))
apply (*induct b*, *simp-all*)
apply (*auto simp*: *PLemmas*)
done

lemma *sep-dom2-aux*: $[\![x \in dom\ (C\ (list2policy\ (a \oplus y\ \#\ z)))]\!]$
$\Longrightarrow x \in dom\ (C\ (a \oplus list2policy\ (y\ \#\ z)))$
by (*metis CConcStartA InDomConc domIff domdConcStart l2p-aux2 list.simps*(*1*) *list2policy.simps*(*2*) *nlpaux*)

lemma *sep-dom2-aux2*:
$[\![(x \in dom\ (C\ (list2policy\ (separate\ (y\ \#\ z)))) \longrightarrow x \in dom\ (C\ (list2policy\ (y\ \#\ z))));$
$x \in dom\ (C\ (list2policy\ (a\ \#\ separate\ (y\ \#\ z))))]\!]$
$\Longrightarrow x \in dom\ (C\ (list2policy\ (a \oplus y\ \#\ z)))$
by (*metis CConcStartA Cdom2 InDomConc domIff l2p-aux2 list2policyconc mem-def nlpaux*)

lemma *sep-dom2*[*rule-format*]:
$x \in dom\ (C\ (list2policy\ (separate\ p))) \longrightarrow x \in dom\ (C\ (list2policy(\ p)))$
apply (*rule separate.induct*)
by (*simp-all add*: *sep-dom2-aux sep-dom2-aux2*)

lemma *sepDom*: $dom\ (C\ (list2policy\ p)) = dom\ (C\ (list2policy\ (separate\ p)))$
apply (*rule equalityI*)
by (*rule subsetI*, (*erule dom-sep*|*erule sep-dom2*))+

lemma *C-eq-s-ext*[*rule-format*]: $p \neq [] \longrightarrow C\ (list2policy\ (separate\ p))\ a\ = C\ (list2policy\ p)\ a$
proof (*induct rule*: *separate.induct*)
case *goal1* **thus** *?case*
apply *simp*
apply (*cases* $x = []$)
apply (*metis l2p-aux2 separate.simps*(*5*))
apply *simp*
apply (*cases* $a \in dom\ (C\ (list2policy\ x))$)
apply (*subgoal-tac* $a \in dom\ (C\ (list2policy\ (separate\ x)))$)
apply (*metis Cdom2 list2policyconc mem-def sepDom sepnMT*)
apply (*metis sepDom*)
apply (*subgoal-tac* $a \notin dom\ (C\ (list2policy\ (separate\ x)))$)
apply (*subst list2policyconc*)
apply (*simp add*: *sepnMT*)
apply (*subst list2policyconc*)
apply (*simp add*: *sepnMT*)
apply (*metis nlpaux sepDom*)
apply (*metis sepDom*)
done
next
case *goal2* **thus** *?case*
apply *simp*
apply (*cases* $z = []$)
apply *simp-all*
apply (*rule conjI*|*rule impI*|*simp*)+

```
    apply (subst list2policyconc)
     apply (metis not-Cons-self sepnMT)
    apply (metis C.simps(4) CConcStartaux Cdom2 domIff)
    apply (rule conjI|rule impI|simp)+
    apply (erule list2policy-eq)
    apply (rule impI, simp)
    apply (subst list2policyconc)
    apply (metis list.simps(1) sepnMT)
     apply (metis C.simps(4) CConcStartaux Cdom2 domIff list2policy.simps(2)
sepDom)
    done
  next
  case goal3 thus ?case
  apply simp
    apply (cases z = [])
    apply simp-all
    apply (rule conjI|rule impI|simp)+
    apply (subst list2policyconc)
     apply (metis not-Cons-self sepnMT)
    apply (metis C.simps(4) CConcStartaux Cdom2 domIff)
    apply (rule conjI|rule impI|simp)+
    apply (erule list2policy-eq)
    apply (rule impI, simp)
    apply (subst list2policyconc)
    apply (metis list.simps(1) sepnMT)
     apply (metis C.simps(4) CConcStartaux Cdom2 domIff list2policy.simps(2)
sepDom)
    done
  next
  case goal4 thus ?case
  apply simp
    apply (cases z = [])
    apply simp-all
    apply (rule conjI|rule impI|simp)+
    apply (subst list2policyconc)
     apply (metis not-Cons-self sepnMT)
    apply (metis C.simps(4) CConcStartaux Cdom2 domIff)
    apply (rule conjI|rule impI|simp)+
    apply (erule list2policy-eq)
    apply (rule impI, simp)
    apply (subst list2policyconc)
    apply (metis list.simps(1) sepnMT)
     apply (metis C.simps(4) CConcStartaux Cdom2 domIff list2policy.simps(2)
sepDom)
    done
   next
  case goal5 thus ?case by simp next
  case goal6 thus ?case by simp next
  case goal7 thus ?case by simp next
```

case *goal8* **thus** *?case* **by** *simp* **next**
qed

lemma *C-eq-s*: $p \neq [] \Longrightarrow C\ (list2policy\ (separate\ p)) = C\ (list2policy\ p)$
apply (*rule ext*)
apply (*rule C-eq-s-ext*)
apply *simp*
done

lemma *setnMT*: $set\ a = set\ b \Longrightarrow a \neq [] \Longrightarrow b \neq []$
by *auto*

lemma *sortnMT*: $p \neq [] \Longrightarrow sort\ p\ l \neq []$
by (*metis set-sort setnMT*)

lemmas *C-eq-Lemmas-sep* = *C-eq-Lemmas sortnMT RS2-NMT notMTpolicyimp-notMT NMPrd NMPRS3 NMPiD*

lemma *C-eq-until-separated*: ⟦*DenyAll* ∈ *set* (*policy2list p*); *all-in-list* (*policy2list p*) *l*; *allNetsDistinct* (*policy2list p*)⟧
⟹ *C* (*list2policy*
(*separate* (*sort* (*removeShadowRules2* (*remdups* (*removeShadowRules3* (*insertDeny* (*removeShadowRules1* (*policy2list p*)))))) *l*))) =
C p
apply (*subst C-eq-s*)
apply (*simp-all add*: *C-eq-Lemmas-sep*)
apply (*rule C-eq-All-untilSorted*)
apply *simp-all*
done

lemma *idNMT*[*rule-format*]: $p \neq [] \longrightarrow insertDenies\ p \neq []$
apply (*induct p*, *simp-all*)
apply (*case-tac a*, *simp-all*)
done

lemma *domID*[*rule-format*]: $p \neq [] \wedge x \in dom(C(list2policy\ p)) \longrightarrow x \in dom\ (C(list2policy(insertDenies\ p)))$
proof(*induct p*)
case *Nil* **then show** *?case* **by** *simp*
next
case (*Cons a p*) **then show** *?case*
proof(*cases p*=[])
case *goal1* **then show** *?case*
apply(*simp*) **apply**(*rule impI*)
apply (*cases a*, *simp-all*)
apply (*simp add*: *C.simps dom-def*)+
apply (*metis domIff mem-def Cdom2 ConcAssoc*)
done

```
next
  case goal2 then show ?case
   proof(cases x ∈ dom(C(list2policy p)))
     case goal1 then show ?case
       apply simp apply (rule impI)
       apply (cases a, simp-all)
       apply (metis InDomConc goal1(2) idNMT)
       apply (rule InDomConc, simp-all add: idNMT)+
       done
   next
     case goal2 then show ?case
       apply simp apply (rule impI)
       proof(cases x ∈ dom (C (list2policy (insertDenies p))))
         case goal1 then show ?case
           proof(induct a)
             case DenyAll then show ?case by simp
           next
             case (DenyAllFromTo src dest) then show ?case
               apply simp by( rule InDomConc, simp add: idNMT)
           next
            case (AllowPortFromTo src dest port) then show ?case
               apply simp by(rule InDomConc, simp  add: idNMT)
           next
            case (Conc - -) then show ?case
               apply simp by(rule InDomConc, simp add: idNMT)
           qed
       next
         case goal2 then show ?case
           proof (induct a)
            case DenyAll then show ?case by simp
           next
            case (DenyAllFromTo src dest) then show ?case
                 by(simp,metis domIff CConcStartA list2policyconc nlpaux
Cdom2)
           next
            case (AllowPortFromTo src dest port) then show ?case
                 by(simp,metis domIff CConcStartA list2policyconc nlpaux
Cdom2)
           next
            case (Conc - -) then show ?case
                 by(simp,metis domIff CConcStartA list2policyconc nlpaux
Cdom2)
           qed
       qed
   qed
 qed
qed
```

lemma *DA-is-deny*: $x \in dom$ (*C* (*DenyAllFromTo a b* $\oplus$ *DenyAllFromTo b a* $\oplus$ *DenyAllFromTo a b*))
$\Longrightarrow$ *C* (*DenyAllFromTo a b* $\oplus$ *DenyAllFromTo b a* $\oplus$ *DenyAllFromTo a b*) *x* = *Some* (*deny x*)
apply (*case-tac x* $\in$ *dom* (*C* (*DenyAllFromTo a b*)))
apply (*simp-all add*: *PLemmas*)
apply (*simp-all split*: *if-splits*)
done

lemma *iDdomAux*[*rule-format*]: $p \neq []$ $\longrightarrow$ $x \notin dom$ (*C* (*list2policy p*)) $\longrightarrow$ $x \in$ *dom* (*C* (*list2policy* (*insertDenies p*))) $\longrightarrow$
C (*list2policy* (*insertDenies p*)) *x* = *Some* (*deny x*)
proof (*induct p*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons y ys*) **thus** *?case*
proof (*cases y*)
case *DenyAll* **then show** *?thesis* **by** *simp* **next**
case (*DenyAllFromTo a b*) **then show** *?thesis* **using** *prems*
apply *simp*
apply (*rule impI*)+
proof (*cases ys* = [])
case *goal1* **then show** *?case* **by** (*simp add*: *DA-is-deny*) **next**
case *goal2* **then show** *?case*
apply *simp*
apply (*drule mp*)
apply (*metis DenyAllFromTo InDomConc goal2*(*3*) *goal2*(*5*))
apply (*cases x* $\in$ *dom* (*C* (*list2policy* (*insertDenies ys*))))
apply *simp-all*
apply (*metis Cdom2 DenyAllFromTo goal2*(*5*) *idNMT list2policyconc*)
apply (*subgoal-tac C* (*list2policy* (*DenyAllFromTo a b* $\oplus$ *DenyAllFromTo b a* $\oplus$ *DenyAllFromTo a b*#*insertDenies ys*)) *x* =
C ((*DenyAllFromTo a b* $\oplus$ *DenyAllFromTo b a* $\oplus$ *DenyAllFromTo a b*)) *x*)
apply *simp*
apply (*rule DA-is-deny*)
apply (*metis DenyAllFromTo domdConcStart goal2*(*4*))
apply (*metis DenyAllFromTo l2p-aux2 list2policyconc nlpaux*)
done
qed
next
case (*AllowPortFromTo a b c*) **then show** *?thesis* **using** *prems*
proof (*cases ys* = [])
case *goal1* **then show** *?case*
apply *simp*
apply (*rule impI*)+
apply (*subgoal-tac x* $\in$ *dom* (*C* (*DenyAllFromTo a b* $\oplus$ *DenyAllFromTo b a*)))
apply (*simp-all add*: *PLemmas*)

apply (*simp split*: *if-splits*) **apply** *auto*
done next
case *goal2* **then show** *?case*
apply *simp*
apply (*rule impI*)+
apply (*drule mp*)
apply (*metis AllowPortFromTo InDomConc goal2*(*4*))
apply (*cases* $x \in$ *dom* (*C* (*list2policy* (*insertDenies ys*))))
apply *simp-all*
apply (*metis AllowPortFromTo Cdom2 goal2*(*4*) *idNMT list2policyconc*)
apply (*subgoal-tac C* (*list2policy* (*DenyAllFromTo a b* $\oplus$ *DenyAllFromTo b a* $\oplus$ *AllowPortFromTo a b c#insertDenies ys*)) x =
C ((*DenyAllFromTo a b* $\oplus$ *DenyAllFromTo b a*)) x)
apply *simp*
defer *1*
apply (*metis AllowPortFromTo CConcStartA ConcAssoc goal2*(*4*) *idNMT list2policyconc nlpaux*)
apply (*simp add*: *PLemmas*, *simp split*: *if-splits*) **apply** *auto*
done
qed
next
case (*Conc a b*) **then show** *?thesis*
proof (*cases ys* = [])
case *goal1* **then show** *?case*
apply *simp*
apply (*rule impI*)+
apply (*subgoal-tac* $x \in$ *dom* (*C* (*DenyAllFromTo* (*first-srcNet a*) (*first-destNet a*) $\oplus$ *DenyAllFromTo* (*first-destNet a*) (*first-srcNet a*))))
apply (*simp-all add*: *PLemmas*)
apply (*simp split*: *if-splits*) **apply** *auto*
done next
case *goal2* **then show** *?case*
apply *simp*
apply (*rule impI*)+
apply (*cases* $x \in$ *dom* (*C* (*list2policy* (*insertDenies ys*))))
apply (*metis Cdom2 Conc Cons InDomConc goal2*(*2*) *idNMT list2policyconc*)
apply (*subgoal-tac C* (*list2policy* (*DenyAllFromTo* (*first-srcNet a*) (*first-destNet a*) $\oplus$ *DenyAllFromTo* (*first-destNet a*) (*first-srcNet a*) $\oplus$ *a*$\oplus$ *b#insertDenies ys*)) x =
C ((*DenyAllFromTo* (*first-srcNet a*) (*first-destNet a*) $\oplus$ *DenyAllFromTo* (*first-destNet a*) (*first-srcNet a*) $\oplus$ *a*$\oplus$ *b*)) x)
apply *simp*
defer *1*
apply (*metis Conc l2p-aux2 list2policyconc nlpaux*)
apply (*subgoal-tac C* ((*DenyAllFromTo* (*first-srcNet a*) (*first-destNet a*) $\oplus$ *DenyAllFromTo* (*first-destNet a*) (*first-srcNet a*) $\oplus$ *a*$\oplus$ *b*)) x =
C ((*DenyAllFromTo* (*first-srcNet a*) (*first-destNet a*) $\oplus$ *DenyAllFromTo* (*first-destNet a*) (*first-srcNet a*))) x)
apply *simp*

defer *1*
apply (*metis CConcStartA Conc ConcAssoc nlpaux*)
apply (*simp add*: *PLemmas*, *simp split*: *if-splits*) **apply** *auto*
done
qed
qed
qed

lemma *iD-isD*[*rule-format*]: $p \neq [] \longrightarrow x \notin dom\ (C\ (list2policy\ p))$
$\longrightarrow C\ (DenyAll \oplus list2policy\ (insertDenies\ p))\ x = C\ DenyAll\ x$
apply (*case-tac* $x \in dom\ (C\ (list2policy\ (insertDenies\ p)))$)
apply (*rule impI*)+
apply (*metis C.simps*(*1*) *deny-all-def iDdomAux mem-def Cdom2*)
apply (*rule impI*)+
apply (*subst nlpaux*)
apply *simp-all*
done

lemma *OTNoTN*[*rule-format*]: *OnlyTwoNets* $p \longrightarrow x \neq DenyAll \longrightarrow x \in set\ p$
$\longrightarrow$ *onlyTwoNets x*
apply (*induct p*, *simp-all*)
apply (*rule impI*)+
apply (*rule conjI*)
apply (*rule impI*)
apply *simp*
apply (*case-tac a*, *simp-all*)
apply (*rule impI*)
apply (*drule mp*, *simp-all*)
apply (*case-tac a*, *simp-all*)
done

lemma *first-isIn*[*rule-format*]: $\neg$ *member DenyAll x* $\longrightarrow$ (*first-srcNet x*,*first-destNet x*) $\in$ *sdnets x*
by (*induct x*,*case-tac x*, *simp-all*)

lemma *sdnets2*: $[\![\exists\, a\ b.\ sdnets\ x = \{(a,\ b),\ (b,\ a)\};\ \neg\ member\ DenyAll\ x]\!] \Longrightarrow$
sdnets x = {(*first-srcNet x*, *first-destNet x*), (*first-destNet x*, *first-srcNet x*)}
apply (*subgoal-tac* (*first-srcNet x*, *first-destNet x*) $\in$ *sdnets x*)
apply (*drule exE*)
prefer *2*
apply *assumption*
apply (*drule exE*)
prefer *2*
apply *assumption*
apply *simp*
apply (*case-tac first-srcNet x* = *a* $\wedge$ *first-destNet x* = *b*)
apply *simp-all*
apply (*metis insert-commute*)
apply (*erule first-isIn*)

done

lemma *alternativelistconc1*[*rule-format*]: $a \in set$ (*net-list-aux* [x]) $\longrightarrow a \in set$ (*net-list-aux* [x,y])
by (*induct* x,*simp-all*)

lemma *alternativelistconc2*[*rule-format*]: $a \in set$ (*net-list-aux* [x]) $\longrightarrow a \in set$ (*net-list-aux* [y,x])
by (*induct* y, *simp-all*)

lemma *noDA*[*rule-format*]:*noDenyAll xs* $\longrightarrow s \in set\ xs \longrightarrow \neg$ *member DenyAll s*
by (*induct xs*, *simp-all*)

lemma *isInAlternativeList*: ($aa \in set$ (*net-list-aux* [a]) $\vee$ $aa \in set$ (*net-list-aux* p))
$\Longrightarrow aa \in set$ (*net-list-aux* (a # p))
apply (*case-tac* a,*simp-all*)
done

lemma *netlistaux*: $x \in set$ (*net-list-aux* (a # p))$\Longrightarrow$$x \in set$ (*net-list-aux* ([a])) $\vee$ $x \in set$ (*net-list-aux* (p))
apply (*case-tac* $x \in set$ (*net-list-aux* [a]))
apply *simp-all*
apply (*case-tac* a, *simp-all*)
done

lemma *firstInNet*[*rule-format*]: $\neg$ *member DenyAll a* $\longrightarrow$ *first-destNet a* $\in set$ (*net-list-aux* (a # p))
apply (*rule Combinators.induct*)
apply *simp-all*
apply (*metis netlistaux*)
done

lemma *firstInNeta*[*rule-format*]: $\neg$ *member DenyAll a* $\longrightarrow$ *first-srcNet a* $\in set$ (*net-list-aux* (a # p))
apply (*rule Combinators.induct*)
apply *simp-all*
apply (*metis netlistaux*)
done

lemma *disjComm*: *disjSD-2 a b* $\Longrightarrow$ *disjSD-2 b a*
apply (*simp add*: *disjSD-2-def*)
apply (*rule allI*)+
apply (*rule impI*)
apply (*rule conjI*)
apply (*drule-tac* $x = c$ **in** *spec*)
apply (*drule-tac* $x = d$ **in** *spec*)
apply (*drule-tac* $x = aa$ **in** *spec*)

apply (*drule-tac x* = *ba* **in** *spec*)
apply (*metis tNDComm*)
apply (*drule-tac x* = *c* **in** *spec*)
apply (*drule-tac x* = *d* **in** *spec*)
apply (*drule-tac x* = *aa* **in** *spec*)
apply (*drule-tac x* = *ba* **in** *spec*)
apply *simp*
apply (*simp add*: *twoNetsDistinct-def*)
apply (*metis nDComm*)+
done

lemma *disjSD2aux*: ⟦*disjSD-2 a b*; ¬ *member DenyAll a*; ¬ *member DenyAll b*⟧ ⟹
disjSD-2 (*DenyAllFromTo* (*first-srcNet a*) (*first-destNet a*) ⊕ *DenyAllFromTo* (*first-destNet a*) (*first-srcNet a*) ⊕ *a*) *b*
apply (*drule disjComm*)
apply (*rule disjComm*)
apply (*simp add*: *disjSD-2-def*)
apply (*rule allI*)+
apply (*rule impI*)+
apply *safe*
apply (*drule-tac x* = *aa* **in** *spec*, *drule-tac x* = *ba* **in** *spec*, *drule-tac x* = *first-srcNet a* **in** *spec*, *drule-tac x* = *first-destNet a* **in** *spec*, *auto intro*: *first-isIn*)+
done

lemma *inDomConc*:⟦ *x*∉*dom* (*C a*); *x*∉*dom* (*C* (*list2policy p*))⟧ ⟹ *x* ∉ *dom* (*C* (*list2policy*(*a*#*p*)))
by (*metis domdConcStart*)

lemma *domsdisj*[*rule-format*]: *p* ≠ [] ⟶ (∀ *x s*. *s* ∈ *set p* ∧ *x* ∈ *dom* (*C A*) ⟶ *x* ∉ *dom* (*C s*)) ⟶ *y* ∈ *dom* (*C A*) ⟶ *y* ∉ *dom* (*C* (*list2policy p*))
apply (*induct p*)
apply *simp*
apply (*case-tac p* = [])
apply *simp*
apply (*rule-tac x* = *y* **in** *spec*)
apply (*simp add*: *split-tupled-all*)
apply (*rule impI*)+
apply (*rule inDomConc*)
apply (*drule-tac x* = *y* **in** *spec*, *drule-tac x* = *a* **in** *spec*)
apply *auto*
done

lemma *isSepaux*: ⟦*p* ≠ []; *noDenyAll* (*a*#*p*); *separated* (*a* # *p*);
x ∈ *dom* (*C* (*DenyAllFromTo* (*first-srcNet a*) (*first-destNet a*) ⊕ *DenyAllFromTo* (*first-destNet a*) (*first-srcNet a*) ⊕ *a*))⟧ ⟹
x ∉ *dom* (*C* (*list2policy p*))
apply (*rule-tac A* = (*DenyAllFromTo* (*first-srcNet a*) (*first-destNet a*) ⊕ *DenyAllFromTo* (*first-destNet a*) (*first-srcNet a*) ⊕ *a*) **in** *domsdisj*)

apply *simp-all*
apply (*rule notI*)
apply (*rule-tac p* = *xa* **and** *x* = (*DenyAllFromTo* (*first-srcNet a*) (*first-destNet a*) $\oplus$ *DenyAllFromTo* (*first-destNet a*) (*first-srcNet a*) $\oplus$ *a*) **and** *y* = *s* **in** *disjSD-no-p-in-both*)
apply *simp-all*
apply (*simp add*: *disjSD-2-def*)
apply (*rule allI*)+
apply (*metis first-isIn tNDComm twoNetsDistinct-def*)
apply (*metis noDA*)
done

lemma *noDA1eq*[*rule-format*]: *noDenyAll p* $\longrightarrow$ *noDenyAll1 p*
apply (*induct p*)
apply *simp*
apply (*case-tac a*, *simp-all*)
done

lemma *noDA1C*[*rule-format*]: *noDenyAll1* (*a*#*p*) $\longrightarrow$ *noDenyAll1 p*
apply (*case-tac a*, *simp-all*)
apply (*rule impI*, *rule noDA1eq*, *simp*)+
done

lemma *disjSD-2IDa*: $[\![$*disjSD-2 x y*; $\neg$ *member DenyAll x*; $\neg$ *member DenyAll y*; *a* = (*first-srcNet x*); *b* = (*first-destNet x*)$]\!]$ $\Longrightarrow$
disjSD-2 ((*DenyAllFromTo a b*) $\oplus$ (*DenyAllFromTo b a*) $\oplus$ *x*) *y*
apply *simp*
apply (*rule disjSD2aux*)
apply *simp-all*
done

lemma *noDAID*[*rule-format*]: *noDenyAll p* $\longrightarrow$ *noDenyAll* (*insertDenies p*)
apply (*induct p*)
apply *simp-all*
apply (*case-tac a*, *simp-all*)
done

lemma *isInIDo*[*rule-format*]: *noDenyAll p* $\longrightarrow$ *s* $\in$ *set* (*insertDenies p*) $\longrightarrow$
($\exists$! *a*. *s* = (*DenyAllFromTo* (*first-srcNet a*) (*first-destNet a*)) $\oplus$ (*DenyAllFromTo* (*first-destNet a*) (*first-srcNet a*)) $\oplus$ *a* $\wedge$ *a* $\in$ *set p*)
apply (*induct p*)
apply *simp-all*
apply (*case-tac a* = *DenyAll*)
apply *simp*
apply (*case-tac a*, *simp-all*)
apply *auto*
done

lemma *id-aux1*[*rule-format*]: *DenyAllFromTo* (*first-srcNet s*) (*first-destNet s*) $\oplus$ *DenyAllFromTo* (*first-destNet s*) (*first-srcNet s*) $\oplus$ *s*$\in$ *set* (*insertDenies p*)

$\longrightarrow s \in set\ p$
apply (*induct p*)
apply *simp-all*
apply (*case-tac a*, *simp-all*)
done

lemma *id-aux2*: $[\![noDenyAll\ p;\ (\forall s.\ s \in set\ p \longrightarrow disjSD\text{-}2\ a\ s);\ \neg\ member\ DenyAll\ a;$
$((DenyAllFromTo\ (first\text{-}srcNet\ s)\ (first\text{-}destNet\ s)) \oplus (DenyAllFromTo\ (first\text{-}destNet\ s)\ (first\text{-}srcNet\ s)) \oplus s) \in set\ (insertDenies\ p)]\!] \Longrightarrow$
$disjSD\text{-}2\ a\ ((DenyAllFromTo\ (first\text{-}srcNet\ s)\ (first\text{-}destNet\ s)) \oplus (DenyAllFromTo\ (first\text{-}destNet\ s)\ (first\text{-}srcNet\ s)) \oplus s)$
apply (*rule disjComm*)
apply (*rule disjSD-2IDa*)
apply *simp-all*
apply (*metis disjComm id-aux1*)
apply (*metis id-aux1 noDA*)
done

lemma *id-aux4*[*rule-format*]: $[\![noDenyAll\ p;\ (\forall s.\ s \in set\ p \longrightarrow disjSD\text{-}2\ a\ s);s \in set\ (insertDenies\ p);\ \neg\ member\ DenyAll\ a]\!] \Longrightarrow\ disjSD\text{-}2\ a\ s$
apply (*subgoal-tac* $\exists a.\ s =$
$DenyAllFromTo\ (first\text{-}srcNet\ a)\ (first\text{-}destNet\ a) \oplus$
$DenyAllFromTo\ (first\text{-}destNet\ a)\ (first\text{-}srcNet\ a) \oplus a \wedge$
$a \in set\ p$)
apply (*drule-tac Q = disjSD-2 a s* **in** *exE*)
apply *simp-all*
apply (*rule id-aux2*, *simp-all*)
apply (*rule ex1-implies-ex*)
apply (*rule isInIDo*)
apply *simp-all*
done

lemma *sepNetsID*[*rule-format*]: $noDenyAll1\ p \longrightarrow separated\ p \longrightarrow separated\ (insertDenies\ p)$
apply (*induct p*)
apply *simp-all*
apply (*rule impI*)
apply (*drule mp*)
apply (*erule noDA1C*)
apply (*rule impI*)
apply (*case-tac a = DenyAll*)
apply *simp-all*
apply (*simp add*: *disjSD-2-def*)
apply (*case-tac a*,*simp-all*)
apply *auto*
apply (*rule disjSD-2IDa*, *simp-all*, *rule id-aux4*, *simp-all*, *metis noDA noDAID*)+
done

lemma *noneMTsep*[*rule-format*]: *noneMT p* $\longrightarrow$ *noneMT* (*separate p*)
apply (*rule separate.induct*) **back**
apply *simp-all*
apply (*rule impI*, *simp*)
apply (*rule impI*)
apply *simp*
apply (*drule mp*)
apply (*simp add*: *C.simps*)
apply *simp*
apply (*rule impI*)+
apply *simp*
apply (*drule mp*)
apply (*simp add*: *C.simps*)
apply *simp*
apply (*rule impI*)+
apply (*simp*)
apply (*drule mp*)
apply (*simp add*: *C.simps*)
apply (*simp*)
done

lemma *aNDDA*[*rule-format*]: *allNetsDistinct p* $\longrightarrow$ *allNetsDistinct*(*DenyAll#p*)
apply (*case-tac p*)
apply *simp*
apply (*rule impI*)
apply (*simp add*: *allNetsDistinct-def*)
apply (*rule impI*)
apply (*auto*)
apply (*simp add*: *allNetsDistinct-def*)
done

lemma *OTNConc*[*rule-format*]: *OnlyTwoNets* (*y* # *z*) $\longrightarrow$ *OnlyTwoNets z*
apply (*case-tac y*, *simp-all*)
done

lemma *first-bothNetsd*: $\neg$ *member DenyAll x* $\Longrightarrow$ *first-bothNet x* = {*first-srcNet x*, *first-destNet x*}
apply (*induct x*)
apply *simp-all*
done

lemma *bNaux*: $[\![\neg$ *member DenyAll x*; $\neg$ *member DenyAll y*;*first-bothNet x* = *first-bothNet y*$]\!]$ $\Longrightarrow$ {*first-srcNet x*, *first-destNet x*} = {*first-srcNet y*, *first-destNet y*}
apply (*simp add*: *first-bothNetsd*)
done

lemma *setPair*: {*a*,*b*} = {*a*,*d*} $\Longrightarrow$ *b* = *d*
apply (*metis Un-empty-right Un-insert-right insert-absorb2 setPaireq*)

done

lemma *setPair1*: $\{a,b\} = \{d,a\} \Longrightarrow b = d$
apply (*metis Un-empty-right Un-insert-right insert-absorb2 setPaireq*)
done

lemma *setPair4*: $\{a,b\} = \{c,d\} \Longrightarrow a \neq c \Longrightarrow a = d$
by *auto*

lemma *otnaux1*: $\{x, y, x, y\} = \{x,y\}$
by *auto*

lemma *OTNIDaux4*: $\{x,y,x\} = \{y,x\}$
by *auto*

lemma *setPair5*: $\{a,b\} = \{c,d\} \Longrightarrow a \neq c \Longrightarrow a = d$
by *auto*

lemma *otnaux*:
 $[\![$*first-bothNet* x = *first-bothNet* y; $\neg$ *member DenyAll* x; $\neg$ *member DenyAll* y;
 onlyTwoNets y; *onlyTwoNets* $x]\!] \Longrightarrow$
 onlyTwoNets $(x \oplus y)$
apply (*simp add*: *onlyTwoNets-def*)
apply (*subgoal-tac* {*first-srcNet* x, *first-destNet* x} = {*first-srcNet* y, *first-destNet* y})
apply (*case-tac* ($\exists$ *a b*. *sdnets* y = {(*a*, *b*)}))
apply *simp-all*
apply (*case-tac* ($\exists$ *a b*. *sdnets* x = {(*a*, *b*)}))
apply *simp-all*
apply (*subgoal-tac sdnets* x = {(*first-srcNet* x, *first-destNet* x)})
apply (*subgoal-tac sdnets* y = {(*first-srcNet* y, *first-destNet* y)})
apply *simp*
apply (*case-tac first-srcNet* x = *first-srcNet* y)
apply *simp-all*
apply (*rule disjI1*)
apply (*rule setPair*)
apply *simp*
apply (*subgoal-tac first-srcNet* x = *first-destNet* y)
apply *simp*
apply (*subgoal-tac first-destNet* x = *first-srcNet* y)
apply *simp*
apply (*rule-tac* x =*first-srcNet* y **in** *exI*, *rule-tac* x = *first-destNet* y **in** *exI*,*simp*)
apply (*rule setPair1*)
apply *simp*
apply (*rule setPair4*)
apply *simp-all*
apply (*metis first-isIn singletonE*)
apply (*metis first-isIn singletonE*)

apply (*subgoal-tac sdnets x* = {(*first-srcNet x*, *first-destNet x*),(*first-destNet x*, *first-srcNet x*)})
apply (*subgoal-tac sdnets y* = {(*first-srcNet y*, *first-destNet y*)})
apply *simp*
apply (*case-tac first-srcNet x* = *first-srcNet y*)
apply *simp-all*
apply (*subgoal-tac first-destNet x* = *first-destNet y*)
apply *simp*
apply (*rule setPair*)
apply *simp*
apply (*subgoal-tac first-srcNet x* = *first-destNet y*)
apply *simp*
apply (*subgoal-tac first-destNet x* = *first-srcNet y*)
apply *simp*
apply (*rule-tac x* =*first-srcNet y* **in** *exI*, *rule-tac x* = *first-destNet y* **in** *exI*)
apply (*metis DomainI Domain-empty Domain-insert OTNIDaux4 RangeI Range-empty Range-insert insertE insert-absorb insert-commute insert-iff mem-def singletonE*)
apply (*rule setPair1*)
apply *simp*
apply (*rule setPair5*)
apply *assumption*
apply *simp*
apply (*metis first-isIn singletonE*)
apply (*rule sdnets2*)
apply *simp-all*
apply (*case-tac* (∃ *a b*. *sdnets x* = {(*a*, *b*)}))
apply *simp-all*
apply (*subgoal-tac sdnets x* = {(*first-srcNet x*, *first-destNet x*)})
apply (*subgoal-tac sdnets y* = {(*first-srcNet y*, *first-destNet y*),(*first-destNet y*, *first-srcNet y*)})
apply *simp*
apply (*case-tac first-srcNet x* = *first-srcNet y*)
apply *simp-all*
apply (*subgoal-tac first-destNet x* = *first-destNet y*)
apply *simp*
apply (*rule-tac x* =*first-srcNet y* **in** *exI*, *rule-tac x* = *first-destNet y* **in** *exI*)
apply (*metis DomainI Domain-empty Domain-insert OTNIDaux4 RangeI Range-empty Range-insert insertE insert-absorb insert-commute insert-iff mem-def singletonE*)
apply (*rule setPair*)
apply *simp*
apply (*subgoal-tac first-srcNet x* = *first-destNet y*)
apply *simp*
apply (*subgoal-tac first-destNet x* = *first-srcNet y*)
apply *simp*
apply (*rule setPair1*)
apply *simp*
apply (*rule setPair4*)
apply *assumption*
apply *simp*

apply (*rule sdnets2*)
apply *simp*
apply *simp*
apply (*metis singletonE first-isIn*)
apply (*subgoal-tac sdnets x* = {(*first-srcNet x*, *first-destNet x*),(*first-destNet x*, *first-srcNet x*)})
apply (*subgoal-tac sdnets y* = {(*first-srcNet y*, *first-destNet y*),(*first-destNet y*, *first-srcNet y*)})
apply *simp*
apply (*case-tac first-srcNet x* = *first-srcNet y*)
apply *simp-all*
apply (*subgoal-tac first-destNet x* = *first-destNet y*)
apply *simp*
apply (*rule-tac x* =*first-srcNet y* **in** *exI*, *rule-tac x* = *first-destNet y* **in** *exI*)
apply (*rule otnaux1*)
apply (*rule setPair*)
apply *simp*
apply (*subgoal-tac first-srcNet x* = *first-destNet y*)
apply *simp*
apply (*subgoal-tac first-destNet x* = *first-srcNet y*)
apply *simp*
apply (*rule-tac x* =*first-srcNet y* **in** *exI*, *rule-tac x* = *first-destNet y* **in** *exI*)
apply (*metis DomainI Domain-empty Domain-insert OTNIDaux4 RangeI Range-empty Range-insert first-isIn insertE insert-absorb insert-commute insert-iff mem-def singletonE*)
apply (*rule setPair1*)
apply *simp*
apply (*rule setPair4*)
apply *assumption*
apply *simp*
apply (*rule sdnets2*,*simp-all*)+
apply (*rule bNaux*, *simp-all*)
done

lemma *OTNSepaux*: ⟦*onlyTwoNets* ($a \oplus y$) ∧ *OnlyTwoNets z* ⟶
OnlyTwoNets (*separate* ($a \oplus y$ # z));
¬ *FWCompilation.member DenyAll a*;
¬ *FWCompilation.member DenyAll y*; *noDenyAll z*;
onlyTwoNets a; *OnlyTwoNets* (y # z);*first-bothNet* (a) = *first-bothNet y*⟧
⟹ *OnlyTwoNets* (*separate* ($a \oplus y$ # z))
apply (*drule mp*)
apply *simp-all*
apply (*rule conjI*)
apply (*rule otnaux*)
apply *simp-all*
apply (*rule-tac p* = (y # z) **in** *OTNoTN*)
apply *simp-all*
apply (*metis FWCompilation.member.simps*(*2*))
apply (*simp add*: *onlyTwoNets-def*)

apply (*rule-tac* $y = y$ **in** *OTNConc*,*simp*)
done

lemma *OTNSEp*[*rule-format*]: *noDenyAll1 p* $\longrightarrow$ *OnlyTwoNets p* $\longrightarrow$ *OnlyTwoNets* (*separate p*)
apply (*rule separate.induct*) **back**
by (*simp-all add*: *OTNSepaux noDA1eq*)

lemma *nda*[*rule-format*]: *singleCombinators* (*a#p*) $\longrightarrow$ *noDenyAll p* $\longrightarrow$ *noDenyAll1* (*a # p*)
apply (*induct p*)
apply *simp-all*
apply (*case-tac a*, *simp-all*)
apply (*case-tac a*, *simp-all*)
done

lemma *nDAcharn*[*rule-format*]: *noDenyAll p* = ($\forall$ *r* $\in$ *set p*. $\neg$ *member DenyAll r*)
apply (*induct p*)
apply *simp-all*
done

lemma *nDAeqSet*: *set p* = *set s* $\Longrightarrow$ *noDenyAll p* = *noDenyAll s*
apply (*simp add*: *nDAcharn*)
done

lemma *nDASCaux*[*rule-format*]: *DenyAll* $\notin$ *set p* $\longrightarrow$ *singleCombinators p* $\longrightarrow$ *r* $\in$ *set p* $\longrightarrow$ $\neg$ *member DenyAll r*
apply (*case-tac r*)
apply *simp-all*
apply (*rule impI*)
apply (*rule impI*)
apply (*rule impI*)
apply (*rule FalseE*)
apply (*rule SCnotConc*)
apply *simp*
apply *simp*
done

lemma *nDASC*[*rule-format*]: *wellformed-policy1 p* $\longrightarrow$ *singleCombinators p* $\longrightarrow$ *noDenyAll1 p*
apply (*induct p*)
apply (*rule impI*)
apply *simp-all*
apply (*rule impI*)+
apply (*drule mp*)
apply (*erule waux2*)
apply (*drule mp*)
apply (*erule singleCombinatorsConc*)

apply (*rule nda*)
apply *simp*
apply (*simp add*: *nDAcharn*)
apply (*rule ballI*)
apply (*rule nDASCaux*)**apply** *simp-all*
apply (*erule singleCombinatorsConc*)
done

lemma *noDAAll*[*rule-format*]: *noDenyAll p* = (¬ *memberP DenyAll p*)
apply (*induct p*)
apply *simp-all*
done

lemma *memberPsep*[*symmetric*]: *memberP x p* = *memberP x* (*separate p*)
apply (*rule separate.induct*) **back**
apply *simp-all*
done

lemma *noDAsep*[*rule-format*]: *noDenyAll p* $\Longrightarrow$ *noDenyAll* (*separate p*)
apply (*simp add*:*noDAAll*)
apply (*subst memberPsep*)
apply *simp*
done

lemma *noDA1sep*[*rule-format*]: *noDenyAll1 p* $\longrightarrow$ *noDenyAll1* (*separate p*)
apply (*rule separate.induct*) **back**
apply *simp-all*
apply (*rule impI*)
apply (*rule noDAsep*)
apply *simp*
apply (*rule impI*)+
apply (*rule noDAsep*)
apply (*case-tac y*, *simp-all*)
apply (*rule impI*)+
apply (*rule noDAsep*)
apply (*case-tac y*, *simp-all*)
apply (*rule impI*)+
apply (*rule noDAsep*)
apply (*case-tac y*, *simp-all*)
done

lemma *isInAlternativeLista*: (*aa* ∈ *set* (*net-list-aux* [*a*]))$\Longrightarrow$ *aa* ∈ *set* (*net-list-aux* (*a* # *p*))
apply (*case-tac a*,*simp-all*)
apply *safe*
done

lemma *isInAlternativeListb*: (*aa* ∈ *set* (*net-list-aux p*))$\Longrightarrow$ *aa* ∈ *set* (*net-list-aux* (*a* # *p*))

apply (*case-tac a*,*simp-all*)
done

lemma *ANDSepaux*: *allNetsDistinct* $(x \# y \# z) \Longrightarrow$ *allNetsDistinct* $(x \oplus y \#$ $z)$
apply (*simp add*: *allNetsDistinct-def*)
apply (*rule allI*)+
apply (*rule impI*)
apply (*drule-tac* $x = a$ **in** *spec*, *drule-tac* $x = b$ **in** *spec*)
apply *simp*
apply (*drule mp*)
apply (*rule conjI*, *simp-all*)
apply (*metis isInAlternativeList*)+
done

lemma *netlistalternativeSeparateaux*: *net-list-aux* $[y]$ @ *net-list-aux* $z =$ *net-list-aux* $(y \# z)$
apply (*case-tac y*, *simp-all*)
done

lemma *netlistalternativeSeparate*: *net-list-aux* $p =$ *net-list-aux* (*separate* p)
apply (*rule separate.induct*) **back**
apply *simp-all*
apply (*simp-all add*: *netlistalternativeSeparateaux*)
done

lemma *ANDSepaux2*: $[\![$*allNetsDistinct* $(x \# y \# z)$; *allNetsDistinct* (*separate* $(y$ $\# z))]\!]$
$\Longrightarrow$ *allNetsDistinct* $(x \#$ *separate* $(y \# z))$
apply (*simp add*: *allNetsDistinct-def*)
apply (*rule allI*)+
apply (*rule impI*)
apply (*drule-tac* $x = a$ **in** *spec*)
apply (*rotate-tac* -1)
apply (*drule-tac* $x = b$ **in** *spec*)
apply (*simp*)
apply (*drule mp*)
apply (*rule conjI*)
apply (*case-tac* $a \in$ *set* (*net-list-aux* $[x]$))
apply *simp-all*
apply (*rule isInAlternativeLista*)
apply *simp*
apply (*rule isInAlternativeListb*)
apply (*subgoal-tac* $a \in$ *set* (*net-list-aux* (*separate* $(y \# z)$)))
apply (*metis netlistalternativeSeparate*)
apply (*metis netlistaux netlistalternativeSeparate*)
apply (*case-tac* $b \in$ *set* (*net-list-aux* $[x]$))
apply (*rule isInAlternativeLista*)
apply *simp*

apply (*rule isInAlternativeListb*)
apply (*subgoal-tac* $b \in$ *set* (*net-list-aux* (*separate* ($y\#z$))))
apply (*metis netlistalternativeSeparate*)
apply (*metis netlistaux netlistalternativeSeparate*)
done

lemma *ANDSep*[*rule-format*]: *allNetsDistinct p* $\longrightarrow$ *allNetsDistinct*(*separate p*)
apply (*rule separate.induct*) **back**
apply *simp-all*
apply (*metis ANDConc aNDDA separate.simps*(*1*))
apply (*metis ANDConc ANDSepaux ANDSepaux2*)
apply (*metis ANDConc ANDSepaux ANDSepaux2*)
apply (*metis ANDConc ANDSepaux ANDSepaux2*)
done

lemma *dom-id*: $\llbracket$*noDenyAll* ($a\#p$); *separated* ($a\#p$); $p \neq []$; $x \notin$ *dom* (C (*list2policy* p)); $x \in$ *dom* (C (a))$\rrbracket$
$\Longrightarrow x \notin$ *dom* (C (*list2policy* (*insertDenies* p)))
apply (*rule-tac* $a = a$ **in** *isSepaux*)
apply *simp-all*
apply (*rule idNMT*)
apply *simp*
apply (*rule noDAID*)
apply *simp*
apply (*rule conjI*)
apply (*rule allI*)
apply (*rule impI*)
apply (*rule id-aux4*)
apply *simp-all*
apply (*rule sepNetsID*)
apply *simp-all*
apply (*metis noDA1eq*)
apply (*simp add*: *C.simps*)
done

lemma *C-eq-iD-aux2*[*rule-format*]:
noDenyAll1 p $\longrightarrow$
separated p$\longrightarrow$
$p \neq []\longrightarrow$
$x \in$ *dom* (C (*list2policy* p))$\longrightarrow$
C(*list2policy* (*insertDenies* p)) x = C(*list2policy* p) x
proof (*induct p*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons y ys*) **thus** *?case* **using** *prems*
proof (*cases y*)
case *DenyAll* **thus** *?thesis* **using** *prems* **apply** *simp*

apply (*case-tac ys* = [])
apply *simp-all*
apply (*case-tac x* ∈ *dom* (*C* (*list2policy ys*)))
apply *simp-all*
apply (*metis Cdom2 Combinators.simps*(*1*) *DenyAll FWCompilation.member.simps*(*3*) *bar3 domID idNMT in-set-conv-decomp insert-absorb insert-code list2policyconc mem-def nMT-domMT noDA1C noDA1eq noDenyAll.simps*(*1*) *notMTpolicyimp-notMT notindom*)
apply (*metis DenyAll iD-isD idNMT list2policyconc nlpaux*)
done
next
case (*DenyAllFromTo a b*) **thus** *?thesis* **using** *prems* **apply** *simp*
apply (*rule impI*|*rule allI*|*rule conjI*|*simp*)+
apply (*case-tac ys* = [])
apply *simp-all*
apply (*metis Cdom2 ConcAssoc DenyAllFromTo*)
apply (*case-tac x* ∈ *dom* (*C* (*list2policy ys*)))
apply *simp-all*
apply (*drule mp*)
apply (*metis noDA1eq*)
apply (*case-tac x* ∈ *dom* (*C* (*list2policy* (*insertDenies ys*))))
apply (*metis Cdom2 DenyAllFromTo idNMT list2policyconc*)
apply (*metis domID*)
apply (*case-tac x* ∈ *dom* (*C* (*list2policy* (*insertDenies ys*))))
apply (*subgoal-tac C* (*list2policy* (*DenyAllFromTo a b* ⊕ *DenyAllFromTo b a* ⊕ *DenyAllFromTo a b* # *insertDenies ys*)) *x* = *Some* (*deny x*))
apply *simp-all*
apply (*subgoal-tac C* (*list2policy* (*DenyAllFromTo a b* # *ys*)) *x* = *C* ((*DenyAllFromTo a b*)) *x*)
apply (*simp add*: *PLemmas*, *simp split*: *if-splits*)
apply (*metis list2policyconc nlpaux*)
apply (*metis Combinators.simps*(*1*) *DenyAllFromTo FWCompilation.member.simps*(*3*) *dom-id domdConcStart mem-def noDenyAll.simps*(*1*) *separated.simps*(*1*))
apply (*metis Cdom2 ConcAssoc DenyAllFromTo domdConcStart l2p-aux2 list2policyconc nlpaux*)
done
next
case (*AllowPortFromTo a b c*) **thus** *?thesis* **using** *prems* **apply** *simp*
apply (*rule impI*|*rule allI*|*rule conjI*|*simp*)+
apply (*case-tac ys* = [])
apply *simp-all*
apply (*metis Cdom2 ConcAssoc AllowPortFromTo*)
apply (*case-tac x* ∈ *dom* (*C* (*list2policy ys*)))
apply *simp-all*
apply (*drule mp*)
apply (*metis noDA1eq*)
apply (*case-tac x* ∈ *dom* (*C* (*list2policy* (*insertDenies ys*))))
apply (*metis Cdom2 AllowPortFromTo idNMT list2policyconc*)
apply (*metis domID*)

apply (*subgoal-tac* $x \in$ *dom* (*C* (*AllowPortFromTo a b c*)))
apply (*case-tac* $x \notin$ *dom* (*C* (*list2policy* (*insertDenies ys*))))
apply *simp-all*
apply (*metis AllowPortFromTo Cdom2 ConcAssoc l2p-aux2 list2policyconc nl-paux*)
apply (*metis AllowPortFromTo Combinators.simps*(*3*) *FWCompilation.member.simps*(*4*) *dom-id mem-def noDenyAll.simps*(*1*) *separated.simps*(*1*))
apply (*metis AllowPortFromTo domdConcStart*)
done
next
case (*Conc a b*) **thus** *?thesis* **using** *prems* **apply** *simp*
apply (*rule impI*|*rule allI*|*rule conjI*|*simp*)+
apply (*case-tac ys* = [])
apply *simp-all*
apply (*metis Cdom2 ConcAssoc Conc*)
apply (*case-tac* $x \in$ *dom* (*C* (*list2policy ys*)))
apply *simp-all*
apply (*drule mp*)
apply (*metis noDA1eq*)
apply (*case-tac* $x \in$ *dom* (*C* ($a \oplus b$)))
apply (*case-tac* $x \notin$ *dom* (*C* (*list2policy* (*insertDenies ys*))))
apply *simp-all*
apply (*subst list2policyconc*)
apply (*rule idNMT*, *simp*)
apply (*metis domID*)
apply (*metis Cdom2 Conc idNMT list2policyconc*)
apply (*metis CConcEnd2 CConcStartA Cdom2 Conc aux0-4 domID domIff id-NMT in-set-conv-decomp l2p-aux2 list2policyconc mem-def nMT-domMT notMT-policyimpnotMT not-Cons-self notindom*)
apply (*case-tac* $x \in$ *dom* (*C* ($a \oplus b$)))
apply (*case-tac* $x \notin$ *dom* (*C* (*list2policy* (*insertDenies ys*))))
apply *simp-all*
apply (*subst list2policyconc*)
apply (*rule idNMT*, *simp*)
apply (*metis Cdom2 Conc ConcAssoc list2policyconc nlpaux*)
apply (*metis Conc FWCompilation.member.simps*(*1*) *dom-id mem-def noDenyAll.simps*(*1*) *separated.simps*(*1*))
apply (*metis Conc domdConcStart*)
done
qed
qed

lemma *C-eq-iD*: ⟦*separated p*; *noDenyAll1 p*; *wellformed-policy1-strong p*⟧ $\Longrightarrow$
C (*list2policy* (*insertDenies p*)) = *C* (*list2policy p*)
apply (*rule ext*)
apply (*rule C-eq-iD-aux2*)
apply *simp-all*
apply (*subgoal-tac DenyAll* $\in$ *set p*)
apply (*metis C-eq-RS1 DAAux append-is-Nil-conv domIff l2p-aux list.simps*(*1*)

mem-def nlpaux removeShadowRules1.simps(1) split-list-first)
apply (*erule wp1-aux1aa*)
done

lemma *wp1-alternativesep*[*rule-format*]: *wellformed-policy1-strong p* $\longrightarrow$ *wellformed-policy1-strong* (*separate p*)
apply (*rule impI*)
apply (*subst wp1n-tl*) **back**
apply *simp*
apply *simp*
apply (*rule sepDA*)
apply (*erule WP1n-DA-notinSet*)
done

lemma *noDAsort*[*rule-format*]: *noDenyAll1 p* $\longrightarrow$ *noDenyAll1* (*sort p l*)
apply (*case-tac p*)
apply *simp*
apply *simp*
apply (*case-tac a = DenyAll*)
apply *simp-all*
apply (*rule impI*)
apply (*subst nDAeqSet*)
defer *1*
apply *simp*
defer *1*
apply (*rule set-sort*)
apply (*rule impI*)
apply (*case-tac insort a* (*sort list l*) *l*)
apply *simp-all*
apply (*rule noDA1eq*)
apply (*subgoal-tac noDenyAll* (*a#list*))
defer *1*
apply (*case-tac a*, *simp*,*simp*)
apply *simp*
apply *simp*
apply (*subst nDAeqSet*)
defer *1*
apply *assumption*
apply (*metis sort.simps*(*2*) *set-sort*)
done

lemma *OTNSC*[*rule-format*]: *singleCombinators p* $\longrightarrow$ *OnlyTwoNets p*
apply (*induct p*)
apply *simp-all*
apply (*rule impI*)
apply (*drule mp*)
apply (*erule singleCombinatorsConc*)
apply (*case-tac a*, *simp-all*)
apply (*simp add*: *onlyTwoNets-def*)+

done

lemma *fMTaux*: $\neg$ *member DenyAll x* $\Longrightarrow$ *first-bothNet x* $\neq$ {}
apply (*metis bot-set-eq first-bothNetsd insert-not-empty*)
done

lemma *fl2*[*rule-format*]: *firstList* (*separate p*) = *firstList p*
apply (*rule separate.induct*)
apply *simp-all*
done

lemma *fl3*[*rule-format*]: *NetsCollected p* $\longrightarrow$ (*first-bothNet x* $\neq$ *firstList p* $\longrightarrow$($\forall$ *a*$\in$*set p*. *first-bothNet x* $\neq$ *first-bothNet a*))$\longrightarrow$ *NetsCollected* (*x#p*)
apply (*induct p*)
apply *simp-all*
done

lemma *sortedConc*[*rule-format*]: *sorted* (*a # p*) *l* $\longrightarrow$ *sorted p l*
apply (*induct p*)
apply *simp-all*
done

lemma *smalleraux2*:
{*a*,*b*} $\in$ *set l* $\Longrightarrow$ {*c*,*d*} $\in$ *set l* $\Longrightarrow$ {*a*,*b*} $\neq$ {*c*,*d*} $\Longrightarrow$
smaller (*DenyAllFromTo a b*) (*DenyAllFromTo c d*) *l* $\Longrightarrow$
$\neg$ *smaller* (*DenyAllFromTo c d*) (*DenyAllFromTo a b*) *l*
apply *simp*
apply (*rule conjI*)
apply (*rule impI*)
apply *simp*
apply (*metis*)
apply (*metis eq-imp-le mem-def pos-noteq*)
done

lemma *smalleraux2a*:
{*a*,*b*} $\in$ *set l* $\Longrightarrow$ {*c*,*d*} $\in$ *set l* $\Longrightarrow$ {*a*,*b*} $\neq$ {*c*,*d*} $\Longrightarrow$
smaller (*DenyAllFromTo a b*) (*AllowPortFromTo c d p*) *l* $\Longrightarrow$
$\neg$ *smaller* (*AllowPortFromTo c d p*) (*DenyAllFromTo a b*) *l*
apply *simp*
apply (*metis eq-imp-le mem-def pos-noteq*)
done

lemma *smalleraux2b*:
{*a*,*b*} $\in$ *set l* $\Longrightarrow$ {*c*,*d*} $\in$ *set l* $\Longrightarrow$ {*a*,*b*} $\neq$ {*c*,*d*} $\Longrightarrow$ *y* = *DenyAllFromTo a b* $\Longrightarrow$
smaller (*AllowPortFromTo c d p*) *y l* $\Longrightarrow$
$\neg$ *smaller y* (*AllowPortFromTo c d p*) *l*
apply *simp*

apply (*metis eq-imp-le mem-def pos-noteq*)
done

lemma *smalleraux2c*:
$\{a,b\} \in set\ l \Longrightarrow \{c,d\} \in set\ l \Longrightarrow \{a,b\} \neq \{c,d\} \Longrightarrow y = AllowPortFromTo\ a$
$b\ q \Longrightarrow$
smaller (*AllowPortFromTo c d p*) *y l* $\Longrightarrow$
$\neg$ *smaller y* (*AllowPortFromTo c d p*) *l*
apply *simp*
apply (*metis eq-imp-le mem-def pos-noteq*)
done

lemma *smalleraux3*:
assumes $x \in set\ l$
assumes $y \in set\ l$
assumes $x \neq y$
assumes $x = bothNet\ a$
assumes $y = bothNet\ b$
assumes *smaller a b l*
assumes *singleCombinators* [*a*]
assumes *singleCombinators* [*b*]
shows $\neg$ *smaller b a l*
proof (*cases a*)
case *DenyAll* **thus** *?thesis* **using** *prems* **by** (*case-tac b*,*simp-all*)
next
case (*DenyAllFromTo c d*) **thus** *?thesis*
proof (*cases b*)
case *DenyAll* **thus** *?thesis* **using** *prems* **by** *simp*
next
case (*DenyAllFromTo e f*) **thus** *?thesis* **using** *prems* **apply** *simp*
by (*metis Combinators.simps*(*13*) *DenyAllFromTo assms*(*1*) *assms*(*2*) *assms*(*3*)
eq-imp-le le-anti-sym pos-noteq)
next
case (*AllowPortFromTo e f g*) **thus** *?thesis* **using** *prems* **apply** *simp*
by (*metis assms*(*1*) *assms*(*2*) *assms*(*3*) *eq-imp-le pos-noteq*)
next
case (*Conc e f*) **thus** *?thesis* **using** *prems* **by** *simp*
qed
next
case (*AllowPortFromTo c d p*) **thus** *?thesis*
proof (*cases b*)
case *DenyAll* **thus** *?thesis* **using** *prems* **by** *simp*
next
case (*DenyAllFromTo e f*) **thus** *?thesis* **using** *prems* **apply** *simp*
by (*metis assms*(*1*) *assms*(*2*) *assms*(*3*) *eq-imp-le pos-noteq*)
next
case (*AllowPortFromTo e f g*) **thus** *?thesis* **using** *prems* **apply** *simp*
by (*metis assms*(*1*) *assms*(*2*) *assms*(*3*) *pos-noteq*)
next

case (*Conc e f*) **thus** *?thesis* **using** *prems* **by** *simp*
qed
next
case (*Conc c d*) **thus** *?thesis* **using** *prems* **by** *simp*
qed

lemma *smalleraux3a*:
$a \neq DenyAll \Longrightarrow b \neq DenyAll \Longrightarrow in\text{-}list\ b\ l \Longrightarrow in\text{-}list\ a\ l \Longrightarrow bothNet\ a \neq bothNet\ b \Longrightarrow$
$smaller\ a\ b\ l \Longrightarrow singleCombinators\ [a] \Longrightarrow singleCombinators\ [b] \Longrightarrow$
$\neg\ smaller\ b\ a\ l$
apply (*rule smalleraux3*)
apply *simp-all*
apply (*case-tac a*, *simp-all*)
apply (*case-tac b*, *simp-all*)
done

lemma *posaux*[*rule-format*]: $position\ a\ l < position\ b\ l \longrightarrow a \neq b$
apply (*induct l*)
apply *simp-all*
done

lemma *posaux6*[*rule-format*]: $a \in set\ l \longrightarrow b \in set\ l \longrightarrow a \neq b \longrightarrow$
$position\ a\ l \neq position\ b\ l$
apply (*induct l*)
apply *simp-all*
apply (*rule conjI*)
apply (*rule impI*)+
apply (*rule conjI*, *rule impI*,*simp*)
apply (*erule position-positive*)
apply (*metis position-positive*)
apply (*metis position-positive*)
done

lemma *notSmallerTransaux*[*rule-format*]:
$[\![x \neq DenyAll;\ r \neq DenyAll;\ singleCombinators\ [x];\ singleCombinators\ [y];\ singleCombinators\ [r];$
$\neg\ smaller\ y\ x\ l;\ smaller\ x\ y\ l;\ smaller\ x\ r\ l;\ smaller\ y\ r\ l;$
$in\text{-}list\ x\ l;\ in\text{-}list\ y\ l;\ in\text{-}list\ r\ l]\!] \Longrightarrow$
$\neg\ smaller\ r\ x\ l$
by (*metis FWCompilationProof.order-trans*)

lemma *notSmallerTrans*[*rule-format*]:
$x \neq DenyAll \longrightarrow r \neq DenyAll \longrightarrow singleCombinators\ (x\#y\#z) \longrightarrow$
$\neg\ smaller\ y\ x\ l \longrightarrow sorted\ (x\#y\#z)\ l \longrightarrow r \in set\ z \longrightarrow$
$all\text{-}in\text{-}list\ (x\#y\#z)\ l \longrightarrow \neg\ smaller\ r\ x\ l$
apply (*rule impI*)+

apply (*rule notSmallerTransaux*)
apply *simp-all*
apply (*metis singleCombinatorsConc singleCombinatorsStart*)
apply (*metis SCSubset equalityE mem-def remdups.simps*(*2*) *set-remdups single-CombinatorsConc singleCombinatorsStart*)
apply *metis*
apply (*metis FWCompilation.sorted.simps*(*3*) *in-set-in-list singleCombinatorsConc singleCombinatorsStart sortedConcStart sorted-is-smaller*)
apply (*metis FWCompilationProof.sorted-Cons all-in-list.simps*(*2*) *singleCombinatorsConc*)
apply *metis*
apply (*metis in-set-in-list*)
done

lemma *NCSaux1*[*rule-format*]:
noDenyAll p $\longrightarrow$ {*x*, *y*} $\in$ *set l* $\longrightarrow$ *all-in-list p l* $\longrightarrow$ *singleCombinators p* $\longrightarrow$
sorted (*DenyAllFromTo x y* # *p*) *l* $\longrightarrow$ {*x*, *y*} $\neq$ *firstList p* $\longrightarrow$ *DenyAllFromTo u v* $\in$ *set p* $\longrightarrow$
{*x*, *y*} $\neq$ {*u*, *v*}
proof (*cases p*)
case *Nil* **thus** *?thesis* **by** *simp* **next**
case (*Cons a p*) **thus** *?thesis* **using** *prems* **apply** *simp*
apply (*rule impI*)+
apply (*rule conjI*)
apply (*metis bothNet.simps*(*2*) *first-bothNet.simps*(*3*))
apply (*rule impI*)
apply (*subgoal-tac smaller* (*DenyAllFromTo x y*) (*DenyAllFromTo u v*) *l*)
apply (*subgoal-tac* $\neg$ *smaller* (*DenyAllFromTo u v*) (*DenyAllFromTo x y*) *l*)
apply (*rule notI*)
apply (*case-tac smaller* (*DenyAllFromTo u v*) (*DenyAllFromTo x y*) *l*)
apply (*simp del*: *smaller.simps*)
apply *simp*
apply (*case-tac x* = *u*)
apply *simp*
apply (*case-tac y* = *v*)
apply *simp*
apply (*subgoal-tac u* = *v*)
apply *simp*
apply *simp*
apply *simp*
apply (*rule-tac y* = *a* **and** *z* = *p* **in** *notSmallerTrans*)
apply (*simp-all del*: *smaller.simps*)
apply (*rule smalleraux3a*)
apply (*simp-all del*: *smaller.simps*)
apply (*case-tac a*, *simp-all del*: *smaller.simps*)
apply (*case-tac a*, *simp-all del*: *smaller.simps*)
apply (*rule-tac y* = *a* **in** *order-trans*)
apply *simp-all*
apply (*subgoal-tac in-list* (*DenyAllFromTo u v*) *l*)

apply *simp*
apply (*rule-tac* $p = p$ **in** *in-set-in-list*)
apply *simp*
apply (*case-tac* a, *simp-all del*: *smaller.simps*)
apply (*metis all-in-list.simps*(*2*) *sorted-Cons mem-def*)
done
qed

lemma *posaux3*[*rule-format*]: $a \in set\ l \longrightarrow b \in set\ l \longrightarrow a \neq b \longrightarrow position\ a\ l \neq position\ b\ l$
apply (*induct* l)
apply *simp-all*
apply (*rule conjI*)
apply (*rule impI*)+
apply (*rule conjI*)
apply (*rule impI*)
apply *simp-all*
apply (*metis position-positive*)+
done

lemma *posaux4*[*rule-format*]: $singleCombinators\ [a] \longrightarrow a\neq\ DenyAll \longrightarrow b \neq DenyAll \longrightarrow in\text{-}list\ a\ l \longrightarrow in\text{-}list\ b\ l \longrightarrow smaller\ a\ b\ \ l \longrightarrow x = (bothNet\ a) \longrightarrow y = (bothNet\ b) \longrightarrow position\ x\ l <= position\ y\ l$
proof (*cases* a)
 case *DenyAll* **then show** *?thesis* **by** *simp*
 next
 case (*DenyAllFromTo c d*) **thus** *?thesis*
  proof (*cases* b)
   case *DenyAll* **thus** *?thesis* **by** *simp* **next**
   case (*DenyAllFromTo e f*) **thus** *?thesis* **using** *prems*
    apply *simp*
    by (*metis bot-set-eq eq-imp-le*)
   next
   case (*AllowPortFromTo e f p*) **thus** *?thesis* **using** *prems* **by** *simp* **next**
   case (*Conc e f*) **thus** *?thesis* **using** *prems* **by** *simp*
 qed
 next
 case (*AllowPortFromTo c d p*) **thus** *?thesis*
  proof (*cases* b)
   case *DenyAll* **thus** *?thesis* **by** *simp* **next**
   case (*DenyAllFromTo e f*) **thus** *?thesis* **using** *prems* **by** *simp* **next**
   case (*AllowPortFromTo e f p*) **thus** *?thesis* **using** *prems* **by** *simp* **next**
   case (*Conc e f*) **thus** *?thesis* **using** *prems* **by** *simp*
 qed
 next
 case (*Conc c d*) **thus** *?thesis* **by** *simp*
qed

lemma *NCSaux2*[*rule-format*]:$noDenyAll\ p \longrightarrow \{a,\ b\} \in set\ l \longrightarrow\ \ all\text{-}in\text{-}list\ p$

$l \longrightarrow$*singleCombinators p* $\longrightarrow$ *sorted* (*DenyAllFromTo a b* # *p*) *l* $\longrightarrow$ $\{a, b\} \neq$ *firstList p* $\longrightarrow$ *AllowPortFromTo u v w* $\in$ *set p* $\longrightarrow$
$\{a, b\} \neq \{u, v\}$
apply (*case-tac p*)
apply *simp-all*
apply (*rule impI*)+
apply (*rule conjI*)
apply (*rule impI*)
apply (*rotate-tac* -1, *drule sym*)
apply *simp*
apply (*rule impI*)
apply (*subgoal-tac smaller* (*DenyAllFromTo a b*) (*AllowPortFromTo u v w*) *l*)
apply (*subgoal-tac* $\neg$ *smaller* (*AllowPortFromTo u v w*) (*DenyAllFromTo a b*) *l*)
defer *1*
apply (*rule-tac y* = *aa* **and** *z* = *list* **in** *notSmallerTrans*)
apply (*simp-all del*: *smaller.simps*)
apply (*rule smalleraux3a*)
apply (*simp-all del*: *smaller.simps*)
apply (*case-tac aa*, *simp-all del*: *smaller.simps*)
apply (*case-tac aa*, *simp-all del*: *smaller.simps*)
apply (*rule-tac y* = *aa* **in** *order-trans*)
apply (*simp-all del*: *smaller.simps*)
apply (*subgoal-tac in-list* (*AllowPortFromTo u v w*) *l*)
apply *simp*
apply (*rule-tac p* = *list* **in** *in-set-in-list*)
apply *simp*
apply *simp*
apply (*metis all-in-list.simps*(*2*) *sorted-Cons mem-def*)
apply (*rule-tac l* = *l* **in** *posaux*)
apply (*rule-tac y* = *position* (*first-bothNet aa*) *l* **in** *basic-trans-rules*(*22*))
apply *simp*
apply (*simp split*: *if-splits*)
apply (*case-tac aa*, *simp-all*)
apply (*case-tac a* = $\alpha 1$ $\wedge$ *b* = $\alpha 2$)
apply *simp-all*
apply (*case-tac a* = $\alpha 1$)
apply *simp-all*
apply (*rule basic-trans-rules*(*18*))
apply *simp*
apply (*rule posaux3*)
apply *simp*
apply *simp*
apply *simp*
apply (*rule basic-trans-rules*(*18*))
apply *simp*
apply (*rule posaux3*)
apply *simp*
apply *simp*
apply *simp*

apply (*rule basic-trans-rules*(*18*))
apply *simp*
apply (*rule posaux3*)
apply *simp*
apply *simp*
apply *simp*
apply (*rule basic-trans-rules*(*18*))
apply (*rule-tac a* = *DenyAllFromTo a b* **and** *b* = *aa* **in** *posaux4*)
apply *simp-all*
apply (*case-tac aa*,*simp-all*)
apply (*case-tac aa*, *simp-all*)
apply (*rule posaux3*)
apply *simp-all*
apply (*case-tac aa*, *simp-all*)
apply (*simp split*: *if-splits*)
apply (*rule-tac a* = *aa* **and** *b* = *AllowPortFromTo u v w* **in** *posaux4*)
apply *simp-all*
apply (*case-tac aa*,*simp-all*)
apply (*rule-tac p* = *list* **in** *sorted-is-smaller*)
apply *simp-all*
apply (*case-tac aa*, *simp-all*)
apply (*case-tac aa*, *simp-all*)
apply (*rule-tac a* = *aa* **and** *b* = *AllowPortFromTo u v w* **in** *posaux4*)
apply *simp-all*
apply (*case-tac aa*,*simp-all*)
apply (*subgoal-tac in-list* (*AllowPortFromTo u v w*) *l*)
apply *simp*
apply (*rule-tac p* = *list* **in** *in-set-in-list*)
apply *simp*
defer *1*
apply *simp-all*
apply (*metis all-in-list.simps*(*2*) *sorted-Cons mem-def*)
apply (*case-tac aa*, *simp-all*)
done

lemma *NCSaux3*[*rule-format*]:
noDenyAll p $\longrightarrow$ $\{a, b\} \in$ *set l* $\longrightarrow$ *all-in-list p l* $\longrightarrow$*singleCombinators p* $\longrightarrow$
sorted (*AllowPortFromTo a b w* # *p*) *l* $\longrightarrow$ $\{a, b\} \neq$ *firstList p* $\longrightarrow$ *DenyAllFromTo u v* $\in$ *set p* $\longrightarrow$
$\{a, b\} \neq \{u, v\}$
apply (*case-tac p*)
apply *simp-all*
apply (*rule impI*)+
apply (*rule conjI*)
apply (*rule impI*)
apply (*rotate-tac* −*1*, *drule sym*)
apply *simp*
apply (*rule impI*)
apply (*subgoal-tac smaller* (*AllowPortFromTo a b w*) (*DenyAllFromTo u v*) *l*)

apply (*subgoal-tac* ¬ *smaller* (*DenyAllFromTo u v*) (*AllowPortFromTo a b w*) *l*)
apply (*simp split*: *if-splits*)
apply (*rule-tac y* = *aa* **and** *z* = *list* **in** *notSmallerTrans*)
apply (*simp-all del*: *smaller.simps*)
apply (*rule smalleraux3a*)
apply (*simp-all del*: *smaller.simps*)
apply (*case-tac aa*, *simp-all del*: *smaller.simps*)
apply (*case-tac aa*, *simp-all del*: *smaller.simps*)
apply (*rule-tac y* = *aa* **in** *order-trans*)
apply (*simp-all del*: *smaller.simps*)
apply (*subgoal-tac in-list* (*DenyAllFromTo u v*) *l*)
apply *simp*
apply (*rule-tac p* = *list* **in** *in-set-in-list*)
apply *simp*
apply *simp*
apply (*rule-tac p* = *list* **in** *sorted-is-smaller*)
apply (*simp-all del*: *smaller.simps*)
apply (*subgoal-tac in-list* (*DenyAllFromTo u v*) *l*)
apply *simp*
apply (*rule-tac p* = *list* **in** *in-set-in-list*)
apply *simp*
apply *simp*
apply (*erule singleCombinatorsConc*)
done

lemma *NCSaux4* [*rule-format*]:
noDenyAll p $\longrightarrow$ $\{a, b\} \in$ *set l* $\longrightarrow$ *all-in-list p l* $\longrightarrow$ *singleCombinators p* $\longrightarrow$
sorted (*AllowPortFromTo a b c* # *p*) *l* $\longrightarrow$ $\{a, b\} \neq$ *firstList p* $\longrightarrow$ *AllowPortFromTo u v w* $\in$ *set p* $\longrightarrow$
$\{a, b\} \neq \{u, v\}$
apply (*case-tac p*)
apply *simp-all*
apply (*rule impI*)+
apply (*rule conjI*)
apply (*rule impI*)
apply (*rotate-tac* −*1*, *drule sym*)
apply *simp*
apply (*rule impI*)
apply (*subgoal-tac smaller* (*AllowPortFromTo a b c*) (*AllowPortFromTo u v w*) *l*)
apply (*subgoal-tac* ¬ *smaller* (*AllowPortFromTo u v w*) (*AllowPortFromTo a b c*) *l*)
apply (*simp split*: *if-splits*)
apply (*rule-tac y* = *aa* **and** *z* = *list* **in** *notSmallerTrans*)
apply (*simp-all del*: *smaller.simps*)
apply (*rule smalleraux3a*)
apply (*simp-all del*: *smaller.simps*)
apply (*case-tac aa*, *simp-all del*: *smaller.simps*)
apply (*case-tac aa*, *simp-all del*: *smaller.simps*)

apply (*case-tac aa*, *simp-all del*: *smaller.simps*)
apply (*rule-tac y* = *aa* **in** *order-trans*)
apply (*simp-all del*: *smaller.simps*)
apply (*subgoal-tac in-list* (*AllowPortFromTo u v w*) *l*)
apply *simp*
apply (*rule-tac p* = *list* **in** *in-set-in-list*)
apply *simp*
apply (*case-tac aa*, *simp-all del*: *smaller.simps*)
apply (*rule-tac p* = *list* **in** *sorted-is-smaller*)
apply (*simp-all del*: *smaller.simps*)
apply (*subgoal-tac in-list* (*AllowPortFromTo u v w*) *l*)
apply *simp*
apply (*rule-tac p* = *list* **in** *in-set-in-list*)
apply *simp*
apply *simp*
apply (*rule-tac y* = *aa* **in** *order-trans*)
apply (*simp-all del*: *smaller.simps*)
apply (*subgoal-tac in-list* (*AllowPortFromTo u v w*) *l*)
apply *simp*
apply (*rule-tac p* = *list* **in** *in-set-in-list*)
apply *simp*
apply *simp*
apply (*rule-tac p* = *list* **in** *sorted-is-smaller*)
apply (*simp-all del*: *smaller.simps*)
apply (*subgoal-tac in-list* (*AllowPortFromTo u v w*) *l*)
apply *simp*
apply (*rule-tac p* = *list* **in** *in-set-in-list*)
apply *simp-all*
done

lemma *NetsCollectedSorted*[*rule-format*]:
noDenyAll1 p $\longrightarrow$ *all-in-list p l* $\longrightarrow$ *singleCombinators p* $\longrightarrow$ *sorted p l* $\longrightarrow$ *NetsCollected p*
apply (*induct p*)
apply *simp*
apply (*rule impI*)+
apply (*drule mp*)
apply (*erule noDA1C*)
apply (*drule mp*)
apply *simp*
apply (*drule mp*)
apply (*erule singleCombinatorsConc*)
apply (*drule mp*)
apply (*erule sortedConc*)

apply (*rule fl3*)
apply *simp*
apply *simp*

apply (*case-tac a*)
apply *simp-all*
apply (*metis fMTaux noDA set-empty2*)
apply (*case-tac aa*)
apply *simp-all*
apply (*rule NCSaux1*, *simp-all*)
apply (*rule NCSaux2*, *simp-all*)
apply (*metis aux0-0*)
apply (*case-tac aa*)
apply *simp-all*
apply (*rule NCSaux3*,*simp-all*)
apply (*rule NCSaux4*,*simp-all*)
apply (*metis aux0-0*)
done

lemma *NetsCollectedSort*: *distinct p* $\Longrightarrow$*noDenyAll1 p* $\Longrightarrow$ *all-in-list p l* $\Longrightarrow$ *singleCombinators p* $\Longrightarrow$ *NetsCollected* (*sort p l*)
apply (*rule-tac l* = *l* **in** *NetsCollectedSorted*)
apply (*rule noDAsort*)
apply *simp-all*
apply (*rule-tac b*=*p* **in** *all-in-listSubset*)
apply *simp-all*
apply (*rule sort-is-sorted*)
apply *simp-all*
done

lemma *fBNsep*[*rule-format*]: ($\forall$ *a*$\in$*set z*. $\{b,c\} \neq$ *first-bothNet a*) $\longrightarrow$ ($\forall$ *a*$\in$*set* (*separate z*). $\{b,c\} \neq$ *first-bothNet a*)
apply (*rule separate.induct*) **back**
apply *simp*
apply (*rule impI*, *simp*)+
done

lemma *fBNsep1*[*rule-format*]: ($\forall$ *a*$\in$*set z*. *first-bothNet x* $\neq$ *first-bothNet a*) $\longrightarrow$ ($\forall$ *a*$\in$*set* (*separate z*). *first-bothNet x* $\neq$ *first-bothNet a*)
apply (*rule separate.induct*) **back**
apply *simp*
apply (*rule impI*, *simp*)+
done

lemma *NetsCollectedSepauxa*: $[\![\{b,c\} \neq$ *firstList z*; *noDenyAll1 z*;
($\forall$ *a*$\in$*set z*. $\{b,c\} \neq$ *first-bothNet a*); *NetsCollected* (*z*);

NetsCollected (*separate* (*z*)); {*b*,*c*} ≠ *firstList* (*separate* (*z*));
a ∈ *set* (*separate* (*z*))⟧
⟹ {*b*,*c*} ≠ *first-bothNet a*
apply (*rule fBNsep*)
apply *simp-all*
done

lemma *NetsCollectedSepaux*: ⟦*first-bothNet* (*x*::(′*a*,′*b*) *Combinators*) ≠ *first-bothNet y*; ¬ *member DenyAll y* ∧ *noDenyAll z*;
(∀ *a*∈*set z*. *first-bothNet x* ≠ *first-bothNet a*) ∧ *NetsCollected* (*y* # *z*);
NetsCollected (*separate* (*y* # *z*)); *first-bothNet x* ≠ *firstList* (*separate* (*y* # *z*));
a ∈ *set* (*separate* (*y* # *z*))⟧
⟹ *first-bothNet* (*x*::(′*a*,′*b*) *Combinators*) ≠ *first-bothNet* (*a*::(′*a*,′*b*) *Combinators*)

apply (*rule fBNsep1*)
apply *simp-all*
apply *auto*
done

lemma *NetsCollectedSep*[*rule-format*]: *noDenyAll1 p* ⟶ *NetsCollected p* ⟶ *NetsCollected* (*separate p*)
apply (*rule separate.induct*) **back**
apply *simp-all*
apply (*metis fMTaux noDA noDA1eq noDAsep set-empty2*)
apply (*rule conjI*|*rule impI*)+
apply *simp*
apply (*metis fBNsep set-ConsD*)
apply (*metis noDA1eq noDenyAll.simps*(*1*) *set-empty2*)
apply (*rule conjI*|*rule impI*)+
apply (*metis fBNsep mem-def set-ConsD*)
apply (*metis noDA1eq noDenyAll.simps*(*1*) *set-empty2*)
apply (*rule conjI*|*rule impI*)+
apply *simp*
apply (*metis NetsCollected.simps*(*1*) *NetsCollectedSepaux firstList.simps*(*1*) *fl2 fl3 noDA1eq noDenyAll.simps*(*1*))
apply (*metis noDA1eq noDenyAll.simps*(*1*))
done

lemma *OTNaux*: *onlyTwoNets a* ⟹ ¬ *member DenyAll a* ⟹
(*x*,*y*) ∈ *sdnets a* ⟹
(*x* = *first-srcNet a* ∧ *y* = *first-destNet a*) ∨ (*x* = *first-destNet a* ∧ *y* = *first-srcNet a*)
apply (*case-tac* (*x* = *first-srcNet a* ∧ *y* = *first-destNet a*))
apply *simp-all*
apply (*simp add*: *onlyTwoNets-def*)

apply (*case-tac* ($\exists$ *aa b*. *sdnets a* = {(*aa*, *b*)}))
apply *simp-all*
apply (*subgoal-tac sdnets a* = {(*first-srcNet a*,*first-destNet a*)})
apply *simp-all*
apply (*metis singletonE first-isIn*)
apply (*subgoal-tac sdnets a* = {(*first-srcNet a*,*first-destNet a*),(*first-destNet a*,*first-srcNet a*)})
apply *simp-all*
apply (*rule sdnets2*)
apply *simp-all*
done

lemma *sdnets-charn*: *onlyTwoNets a* $\Longrightarrow$ $\neg$ *member DenyAll a* $\Longrightarrow$
sdnets a = {(*first-srcNet a*,*first-destNet a*)} $\vee$ *sdnets a* = {(*first-srcNet a*, *first-destNet a*),(*first-destNet a*, *first-srcNet a*)}
apply (*case-tac sdnets a* = {(*first-srcNet a*, *first-destNet a*)})
apply *simp-all*
apply (*simp add*: *onlyTwoNets-def*)
apply (*case-tac* ($\exists$ *aa b*. *sdnets a* = {(*aa*, *b*)}))
apply *simp-all*
apply (*metis singletonE first-isIn*)
apply (*subgoal-tac sdnets a* = {(*first-srcNet a*,*first-destNet a*),(*first-destNet a*,*first-srcNet a*)})
apply *simp-all*
apply (*rule sdnets2*)
apply *simp-all*
done

lemma *first-bothNet-charn*[*rule-format*]: $\neg$ *member DenyAll a* $\longrightarrow$ *first-bothNet a* = {*first-srcNet a*, *first-destNet a*}
apply (*induct a*)
apply *simp-all*
done

lemma *sdnets-noteq*:⟦*onlyTwoNets a*; *onlyTwoNets aa*; *first-bothNet a* $\neq$ *first-bothNet aa*;
$\neg$ *member DenyAll a*; $\neg$ *member DenyAll aa*⟧
$\Longrightarrow$ *sdnets a* $\neq$ *sdnets aa*
apply (*insert sdnets-charn* [*of a*])
apply (*insert sdnets-charn* [*of aa*])
apply (*insert first-bothNet-charn* [*of a*])
apply (*insert first-bothNet-charn* [*of aa*])
apply *simp*
apply (*metis OTNaux first-bothNetsd first-isIn insert-absorb2 insert-commute*)
done

lemma *fbn-noteq*: ⟦*onlyTwoNets a*; *onlyTwoNets aa*; *first-bothNet a* $\neq$ *first-bothNet aa*;

$\neg$ *member DenyAll a*; $\neg$ *member DenyAll aa*; *allNetsDistinct* [*a*, *aa*]⟧
$\Longrightarrow$ *first-srcNet a* $\neq$ *first-srcNet aa* $\vee$ *first-srcNet a* $\neq$ *first-destNet aa* $\vee$ *first-destNet a* $\neq$ *first-srcNet aa* $\vee$ *first-destNet a* $\neq$ *first-destNet aa*
apply (*insert sdnets-charn* [*of a*])
apply (*insert sdnets-charn* [*of aa*])
apply *simp*
apply (*insert sdnets-noteq* [*of a aa*])
apply *simp*
apply (*rule impI*)+
apply *simp*
apply (*case-tac sdnets a* = {(*first-destNet aa*, *first-srcNet aa*)})
apply *simp-all*
apply (*case-tac sdnets aa* = {(*first-srcNet aa*, *first-destNet aa*)})
apply *simp-all*
done

lemma *NCisSD2aux*: ⟦*onlyTwoNets a*; *onlyTwoNets aa*; *first-bothNet a* $\neq$ *first-bothNet aa*;
$\neg$ *member DenyAll a*; $\neg$ *member DenyAll aa*; *allNetsDistinct* [*a*, *aa*]⟧
$\Longrightarrow$ *disjSD-2 a aa*
apply (*simp add*: *disjSD-2-def*)
apply (*rule allI*)+
apply (*rule impI*)
apply (*insert sdnets-charn* [*of a*])
apply (*insert sdnets-charn* [*of aa*])
apply *simp*
apply (*insert sdnets-noteq* [*of a aa*])
apply (*insert fbn-noteq* [*of a aa*])
apply *simp*
apply (*simp add*: *allNetsDistinct-def twoNetsDistinct-def*)
apply (*rule conjI*)
apply (*cases sdnets a* = {(*first-srcNet a*, *first-destNet a*)})
apply (*cases sdnets aa* = {(*first-srcNet aa*, *first-destNet aa*)})
apply *simp-all*
apply (*metis firstInNeta firstInNet alternativelistconc2*)
apply (*case-tac* (*c* = *first-srcNet aa* $\wedge$ *d* = *first-destNet aa*))
apply *simp-all*
apply (*case-tac* (*first-srcNet a*) $\neq$ (*first-srcNet aa*))
apply (*metis firstInNeta firstInNet alternativelistconc2*)
apply *simp*
apply (*subgoal-tac first-destNet a* $\neq$ *first-destNet aa*)
apply (*metis firstInNeta firstInNet alternativelistconc2*)
apply (*metis first-bothNetsd set-empty2*)
apply (*case-tac* (*first-destNet aa*) $\neq$ (*first-srcNet a*))
apply (*metis firstInNeta firstInNet alternativelistconc2*)
apply *simp*
apply (*case-tac first-destNet aa* $\neq$ *first-destNet a*)

apply *simp*
apply (*subgoal-tac first-srcNet aa* $\neq$ *first-destNet a*)
apply (*metis firstInNeta firstInNet alternativelistconc2*)
apply (*metis first-bothNetsd insert-commute set-empty2*)
apply (*metis firstInNeta firstInNet alternativelistconc2*)
apply (*case-tac* (*c* = *first-srcNet aa* $\wedge$ *d* = *first-destNet aa*))
apply *simp-all*
apply (*case-tac* (*ab* = *first-srcNet a* $\wedge$ *b* = *first-destNet a*))
apply *simp-all*
apply (*case-tac* (*first-srcNet a*) $\neq$ (*first-srcNet aa*))
apply (*metis firstInNeta firstInNet alternativelistconc2*)
apply *simp*
apply (*subgoal-tac first-destNet a* $\neq$ *first-destNet aa*)
apply (*metis firstInNeta firstInNet alternativelistconc2*)
apply (*metis first-bothNetsd set-empty2*)
apply (*case-tac* (*first-destNet aa*) $\neq$ (*first-srcNet a*))
apply (*metis firstInNeta firstInNet alternativelistconc2*)
apply *simp*
apply (*case-tac first-destNet aa* $\neq$ *first-destNet a*)
apply *simp*
apply (*subgoal-tac first-srcNet aa* $\neq$ *first-destNet a*)
apply (*metis firstInNeta firstInNet alternativelistconc2*)
apply (*metis first-bothNetsd insert-commute set-empty2*)
apply (*metis firstInNeta firstInNet alternativelistconc2*)
apply (*case-tac* (*ab* = *first-srcNet a* $\wedge$ *b* = *first-destNet a*))
apply *simp-all*
apply (*case-tac c* = *first-srcNet aa*)
apply *simp-all*
apply (*metis OTNaux*)
apply (*subgoal-tac c* = *first-destNet aa*)
apply *simp*
apply (*subgoal-tac d* = *first-srcNet aa*)
apply *simp*
apply (*case-tac* (*first-srcNet a*) $\neq$ (*first-destNet aa*))
apply (*metis firstInNeta firstInNet alternativelistconc2 alternativelistconc1*)
apply *simp*
apply (*subgoal-tac first-destNet a* $\neq$ *first-srcNet aa*)
apply (*metis firstInNeta firstInNet alternativelistconc2*)
apply (*metis first-bothNetsd set-empty2 insert-commute*)
apply (*metis OTNaux*)
apply (*metis OTNaux*)
apply (*case-tac c* = *first-srcNet aa*)
apply *simp-all*
apply (*metis OTNaux*)
apply (*subgoal-tac c* = *first-destNet aa*)
apply *simp*
apply (*subgoal-tac d* = *first-srcNet aa*)
apply *simp*
apply (*case-tac* (*first-destNet a*) $\neq$ (*first-destNet aa*))

apply (*metis firstInNeta firstInNet alternativelistconc2 alternativelistconc1*)
apply *simp*
apply (*subgoal-tac first-srcNet a* $\neq$ *first-srcNet aa*)
apply (*metis firstInNeta firstInNet alternativelistconc2*)
apply (*metis first-bothNetsd set-empty2 insert-commute*)
apply (*metis OTNaux*)
apply (*metis OTNaux*)
apply (*cases sdnets a* = {(*first-srcNet a*, *first-destNet a*)})
apply (*cases sdnets aa* = {(*first-srcNet aa*, *first-destNet aa*)})
apply *simp-all*
apply (*case-tac* (*c* = *first-srcNet aa* $\wedge$ *d* = *first-destNet aa*))
apply *simp-all*
apply (*case-tac* (*first-srcNet a*) $\neq$ (*first-destNet aa*))
apply *simp-all*
apply (*metis firstInNeta firstInNet alternativelistconc2 alternativelistconc1*)
apply (*subgoal-tac first-destNet a* $\neq$ *first-srcNet aa*)
apply (*metis firstInNeta firstInNet alternativelistconc2 alternativelistconc1*)
apply (*metis first-bothNetsd set-empty2 insert-commute*)
apply (*case-tac* (*c* = *first-srcNet aa* $\wedge$ *d* = *first-destNet aa*))
apply *simp-all*
apply (*case-tac* (*ab* = *first-srcNet a* $\wedge$ *b* = *first-destNet a*))
apply *simp-all*
apply (*case-tac* (*first-destNet a*) $\neq$ (*first-srcNet aa*))
apply (*metis firstInNeta firstInNet alternativelistconc2*)
apply *simp*
apply (*subgoal-tac first-srcNet a* $\neq$ *first-destNet aa*)
apply (*metis firstInNeta firstInNet alternativelistconc2*)
apply (*metis first-bothNetsd set-empty2 insert-commute*)
apply (*case-tac* (*first-srcNet aa*) $\neq$ (*first-srcNet a*))
apply (*metis firstInNeta firstInNet alternativelistconc2*)
apply *simp*
apply (*case-tac first-destNet aa* $\neq$ *first-destNet a*)
apply (*metis firstInNeta firstInNet alternativelistconc2 alternativelistconc1*)
apply *simp*
apply (*metis first-bothNetsd set-empty2*)
apply (*cases sdnets aa* = {(*first-srcNet aa*, *first-destNet aa*)})
apply *simp-all*
apply (*case-tac* (*c* = *first-srcNet aa* $\wedge$ *d* = *first-destNet aa*))
apply *simp-all*
apply (*case-tac* (*ab* = *first-srcNet a* $\wedge$ *b* = *first-destNet a*))
apply *simp-all*
apply (*case-tac* (*first-destNet a*) $\neq$ (*first-srcNet aa*))
apply (*metis firstInNeta firstInNet alternativelistconc2*)
apply *simp*
apply (*subgoal-tac first-srcNet a* $\neq$ *first-destNet aa*)
apply (*metis firstInNeta firstInNet alternativelistconc2*)
apply (*metis first-bothNetsd set-empty2 insert-commute*)
apply (*case-tac* (*first-srcNet aa*) $\neq$ (*first-srcNet a*))
apply (*metis firstInNeta firstInNet alternativelistconc2*)

apply *simp*
apply (*case-tac first-destNet aa* $\neq$ *first-destNet a*)
apply (*metis firstInNeta firstInNet alternativelistconc2 alternativelistconc1*)
apply *simp*
apply (*metis first-bothNetsd set-empty2*)
apply (*case-tac* (c = *first-srcNet aa* $\wedge$ d = *first-destNet aa*))
apply *simp-all*
apply (*case-tac* (*ab* = *first-srcNet a* $\wedge$ b = *first-destNet a*))
apply *simp-all*
apply (*case-tac* (*first-destNet a*) $\neq$ (*first-srcNet aa*))
apply (*metis firstInNeta firstInNet alternativelistconc2*)
apply *simp*
apply (*subgoal-tac first-srcNet a* $\neq$ *first-destNet aa*)
apply (*metis firstInNeta firstInNet alternativelistconc2*)
apply (*metis first-bothNetsd set-empty2 insert-commute*)
apply (*case-tac* (*first-srcNet aa*) $\neq$ (*first-srcNet a*))
apply (*metis firstInNeta firstInNet alternativelistconc2*)
apply *simp*
apply (*case-tac first-destNet aa* $\neq$ *first-destNet a*)
apply (*metis firstInNeta firstInNet alternativelistconc2 alternativelistconc1*)
apply *simp*
apply (*case-tac* (*ab* = *first-srcNet a* $\wedge$ b = *first-destNet a*))
apply *simp-all*
apply (*case-tac* (*first-destNet a*) $\neq$ (*first-srcNet aa*))
apply (*metis firstInNeta firstInNet alternativelistconc2*)
apply *simp*
apply (*subgoal-tac first-srcNet a* $\neq$ *first-srcNet aa*)
apply (*metis firstInNeta firstInNet alternativelistconc2*)
apply (*metis first-bothNetsd set-empty2 insert-commute*)
apply (*case-tac* (*first-srcNet aa*) $\neq$ (*first-destNet a*))
apply (*metis firstInNeta firstInNet alternativelistconc2*)
apply *simp*
apply (*case-tac first-destNet aa* $\neq$ *first-srcNet a*)
apply (*metis firstInNeta firstInNet alternativelistconc2 alternativelistconc1*)
apply *simp*
apply (*metis insert-commute set-empty2*)
done

lemma *ANDaux3*[*rule-format*]: $y \in$ *set xs* $\longrightarrow$ $a \in$ *set* (*net-list-aux* [y]) $\longrightarrow$ $a \in$ *set* (*net-list-aux xs*)
apply (*induct xs*)
apply *simp-all*
apply (*rule conjI*)
apply (*rule impI*)+
apply *simp*
apply (*metis isInAlternativeList*)
apply (*rule impI*)+
apply *simp*
apply (*erule isInAlternativeListb*)

done

lemma *ANDaux2*: *allNetsDistinct* (*x* # *xs*) $\Longrightarrow$ *y* $\in$ *set xs*
$\Longrightarrow$ *allNetsDistinct* [*x*,*y*]
apply (*simp add*: *allNetsDistinct-def*)
apply (*rule allI*)
apply (*rule allI*)
apply (*rule impI*)+
apply (*drule-tac x* = *a* **in** *spec*)
apply (*drule-tac x* = *b* **in** *spec*)
apply *simp*
apply (*drule mp*)
apply *simp-all*
apply (*rule conjI*)
apply (*case-tac a* $\in$ *set* (*net-list-aux* [*x*]))
apply (*erule isInAlternativeLista*)
apply (*rule isInAlternativeListb*)
apply (*rule ANDaux3*)
apply *simp-all*
apply (*metis netlistaux*)
apply (*case-tac b* $\in$ *set* (*net-list-aux* [*x*]))
apply (*erule isInAlternativeLista*)
apply (*rule isInAlternativeListb*)
apply (*rule ANDaux3*)
apply *simp-all*
apply (*metis netlistaux*)
done

lemma *NCisSD2*[*rule-format*]:
$[\![$¬ *member DenyAll a*; *OnlyTwoNets* (*a*#*p*); *NetsCollected2* (*a* # *p*); *NetsCollected* (*a*#*p*);*noDenyAll* (*p*); *allNetsDistinct* (*a* # *p*); *s* $\in$ *set p*$]\!]$ $\Longrightarrow$
disjSD-2 a s
apply (*case-tac p*)
apply *simp-all*
apply (*rule NCisSD2aux*)
apply *simp-all*
apply (*rule OTNoTN*)
apply *simp*
apply (*case-tac a*, *simp-all*)
apply (*rule OTNoTN*)
apply *simp*
apply (*metis FWCompilation.member.simps*(*2*) *noDA*)
apply *simp*
apply *metis*
apply (*metis noDA*)
apply (*rule ANDaux2*)
apply *simp-all*
apply *simp*
done

lemma *separatedNC*[*rule-format*]: *OnlyTwoNets p* ⟶ *NetsCollected2 p* ⟶ *NetsCollected p* ⟶ *noDenyAll1 p* ⟶ *allNetsDistinct p* ⟶ *separated p*
apply (*induct p*)
apply *simp-all*
apply (*case-tac a* = *DenyAll*)
apply *simp-all*
defer *1*
apply (*rule impI*)+
apply (*drule mp*)
apply (*erule OTNConc*)
apply (*drule mp*)
apply (*case-tac p*, *simp-all*)
apply (*drule mp*)
apply (*erule noDA1C*)
apply (*rule conjI*)
apply (*rule allI*)
apply (*rule impI*)
apply (*rule NCisSD2*)
apply *simp-all*
apply (*case-tac a*, *simp-all*)
apply (*case-tac a*, *simp-all*)
apply (*drule mp*)
apply (*erule ANDConc*)
apply *simp*
apply (*rule impI*)+
apply (*simp*)
apply (*drule mp*)
apply (*erule noDA1eq*)
apply (*drule mp*)
apply (*erule ANDConc*)
apply *simp*
apply (*simp add*: *disjSD-2-def*)
done

lemma *NC2Sep*[*rule-format*]: *noDenyAll1 p* ⟶ *NetsCollected2* (*separate p*)
apply (*rule separate.induct*) **back**
apply *simp-all*
apply (*rule impI*, *drule mp*)
apply (*erule noDA1eq*)
apply (*case-tac separate x* = [])
apply *simp-all*
apply (*case-tac x*, *simp-all*)
apply (*metis fMTaux firstList.simps*(*1*) *fl2 set-empty2*)
apply (*rule impI*)+
apply *simp*
apply (*drule mp*)
apply (*rule noDA1eq*)
apply (*case-tac y*, *simp-all*)

apply (*metis firstList.simps*(*1*) *fl2*)
apply (*rule impI*)+
apply *simp*
apply (*drule mp*)
apply (*rule noDA1eq*)
apply (*case-tac y*, *simp-all*)
apply (*metis firstList.simps*(*1*) *fl2*)
apply (*rule impI*)+
apply *simp*
apply (*drule mp*)
apply (*rule noDA1eq*)
apply (*case-tac y*, *simp-all*)
apply (*metis firstList.simps*(*1*) *fl2*)
done

lemma *separatedSep*[*rule-format*]: *OnlyTwoNets p* $\longrightarrow$ *NetsCollected2 p* $\longrightarrow$ *NetsCollected p* $\longrightarrow$ *noDenyAll1 p* $\longrightarrow$ *allNetsDistinct p* $\longrightarrow$
separated (*separate p*)
apply (*rule impI*)+
apply (*rule separatedNC*)
apply (*rule OTNSEp*)
apply *simp-all*
apply (*erule NC2Sep*)
apply (*erule NetsCollectedSep*)
apply *simp*
apply (*erule noDA1sep*)
apply (*erule ANDSep*)
done

lemmas *CLemmas* = *noneMTsep nMTSort noneMTRS2 noneMTrd nMTRS3 separatedSep noDAsort nDASC wp1-eq WP1rd wp1ID SC2 SCrd SCRS3 SCRiD SC1 aux0 aND-sort SC2 SCrd aND-RS2 ANDRS3 wellformed1-sorted wp1ID ANDiD ANDrd SC1 aND-RS1 SC3 ANDSep OTNSEp OTNSC noDA1sep wp1-alternativesep wellformed1-alternative-sorted distinct-RS2*

lemmas *C-eqLemmas-id* = *C-eq-Lemmas-sep CLemmas OTNSEp NC2Sep NetsCollectedSep NetsCollectedSort separatedNC*

lemma *C-eq-Until-InsertDenies*: $[\![$*DenyAll* $\in$ *set* (*policy2list p*); *all-in-list* (*policy2list p*) *l*; *allNetsDistinct* (*policy2list p*)$]\!]$ $\Longrightarrow$

C (*list2policy* ((*insertDenies* (*separate* (*sort* (*removeShadowRules2* (*remdups* (*removeShadowRules3* (*insertDeny* (*removeShadowRules1* (*policy2list p*)))))) *l*))))) = *C p*
apply (*subst C-eq-iD*)
apply (*simp-all add*: *C-eqLemmas-id*)
apply (*rule C-eq-until-separated*)
apply *simp-all*
done

lemma *rADnMT*[*rule-format*]: $p \neq [] \longrightarrow$ *removeAllDuplicates* $p \neq []$
apply (*induct p*)
apply *simp-all*
done

lemma *C-eq-RD-aux*[*rule-format*]: $C\ (p)\ x = C$ (*removeDuplicates* p) x
apply (*induct p*)
apply *simp-all*
apply (*rule conjI*, *rule impI*)
apply (*metis Cdom2 domIff nlpaux not-in-member*)
apply (*metis C.simps*(*4*) *CConcStartaux Cdom2 domIff*)
done

lemma *C-eq-RAD-aux*[*rule-format*]: $p \neq [] \longrightarrow C$ (*list2policy* p) $x = C$ (*list2policy* (*removeAllDuplicates* p)) x
apply (*induct p*)
apply *simp-all*
apply (*case-tac* $p = []$)
apply *simp-all*
apply (*metis C-eq-RD-aux*)
apply (*subst list2policyconc*)
apply *simp*
apply (*case-tac* $x \in$ *dom* (C (*list2policy* p)))
apply (*subst list2policyconc*)
apply (*rule rADnMT*)
apply *simp*
apply (*subst Cdom2*)
apply *simp*
apply (*drule sym*)
apply (*subst Cdom2*)
apply (*simp add*: *dom-def*)
apply *simp*
apply (*drule sym*)
apply (*subst nlpaux*)
apply *simp*
apply (*subst list2policyconc*)
apply (*rule rADnMT*)
apply *simp*
apply (*subst nlpaux*)
apply (*simp add*: *dom-def*)
apply (*rule C-eq-RD-aux*)
done

lemma *C-eq-RAD*: $p \neq [] \Longrightarrow C$ (*list2policy* p) $= C$ (*list2policy* (*removeAllDuplicates* p))
apply (*rule ext*)
apply (*erule C-eq-RAD-aux*)
done

lemma *C-eq-compile*:
$[\![$*DenyAll* $\in$ *set* (*policy2list p*); *all-in-list* (*policy2list p*) *l*; *allNetsDistinct* (*policy2list p*)$]\!]$
$\Longrightarrow$
C (*list2policy* (*removeAllDuplicates* (*insertDenies* (*separate* (*sort* (*removeShadowRules2* (*remdups* (*removeShadowRules3* (*insertDeny* (*removeShadowRules1* (*policy2list p*)))))) *l*))))) = *C p*
apply (*subst C-eq-RAD*[*symmetric*])
apply (*rule idNMT*)
apply (*simp add*: *C-eqLemmas-id*)
apply (*rule C-eq-Until-InsertDenies*)
apply *simp-all*
done

end

References

[1] A. D. Brucker, L. Brügger, P. Kearney, and B. Wolff. Verified firewall policy transformations for test case generation. In A. Cavalli and S. Ghosh, editors, *International Conference on Software Testing (ICST10)*, Lecture Notes in Computer Science. Springer-Verlag, 2010.

[2] A. D. Brucker, L. Brügger, and B. Wolff. Model-based firewall conformance testing. In K. Suzuki and T. Higashino, editors, *Testcom/FATES 2008*, number 5047 in Lecture Notes in Computer Science, pages 103–118. Springer-Verlag, 2008.

[3] A. D. Brucker and B. Wolff. Test-sequence generation with HOL-TestGen – with an application to firewall testing. In B. Meyer and Y. Gurevich, editors, *TAP 2007: Tests And Proofs*, number 4454 in Lecture Notes in Computer Science, pages 149–168. Springer-Verlag, 2007.

[4] D. von Bidder. *Specification-based Firewall Testing.* Ph.d. thesis, ETH Zurich, 2007. ETH Dissertation No. 17172. Diana von Bidder's maiden name is Diana Senn.