

HOL-TestGenFW

Achim D. Brucker Lukas Brügger Burkhardt Wolff

November 19, 2012

Contents

1. Introduction	2
2. Installing and using HOL-TestGen/FW	3
3. Preliminaries	3
4. Packets and Networks	8
5. Address Representations	11
5.1. Datatype Addresses	12
5.2. Datatype Addresses with Ports	13
5.3. Integer Addresses	14
5.4. Integer Addresses with Ports	14
5.5. Integer Addresses with Ports and Protocols	15
5.6. IPv4 Addresses	16
6. Policies	17
6.1. Policy Core	17
6.2. Policy Combinators	17
6.3. Policy Combinators with Ports	18
6.4. Policy Combinators with Ports and Protocols	21
6.5. Ports	24
6.6. NAT	25
7. Network Address Translation	25
8. Policy Normalisation	29
8.1. Basics	29
8.2. Auxiliary definitions and functions.	30
8.3. Invariants	32
8.4. Transformations	33

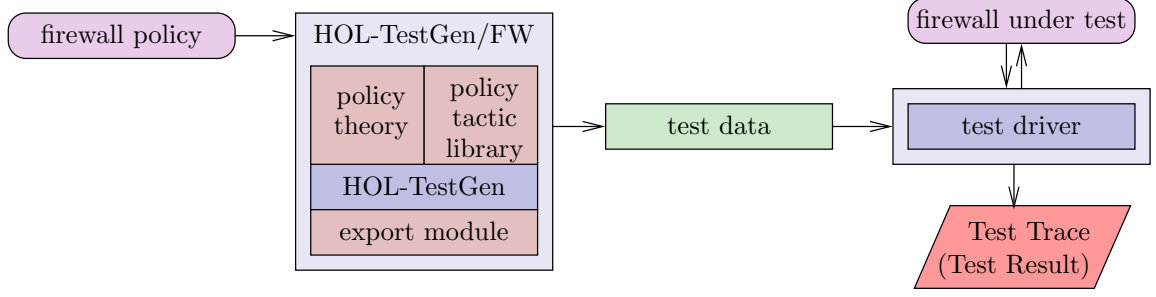


Figure 1: The HOL-TestGen/FW architecture.

8.5. IntPort	36
8.6. TCP_UDP_IntegerPort	38
9. Stateful Firewalls	41
9.1. Basic Constructs	41
9.2. FTP Protocol	42
9.2.1. The protocol syntax	42
9.2.2. The protocol policy specification	44
9.3. VoIP Protocol	48
9.4. FTP and VoIP Protocol	52
A. Appendix	59

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. Furthermore the UPF heap must have been built before. The latter can be done by going into the `add-ons/security/src/` directory and call

```
isabelle make
```

from the command line.

To build the firewall extension, please 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 jedit -l HOL-TestGenFW
```

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

```
isabelle jedit -l HOLCF-TestGenFW
```

3. Preliminaries

```
theory
  FWTesting
imports
  PacketFilter/PacketFilter
  NAT/NAT
  FWNormalisation/FWNormalisation
  StatefulFW/StatefulFW
begin
```

declare *dom-eq-empty-conv* [*simp del*]

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.

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

```

```

ML <<
fun conj-ts-tac ctxt =
  (REPEAT(CHANGED(TRYALL((rtac @{thm conjI}) THEN' (rtac @{thm impI})))));
>>

```

```

method-setup conj-ts =
  <<
    Scan.succeed (fn ctxt => SIMPLE-METHOD
      (conj-ts-tac ctxt))>> Prepares the firewall test theorem

```

```

ML <<
fun export-fw-data ctxt filename dataname =

```

```

let
  val thy = ProofContext.theory-of ctxt
  val thms = Global-Theory.get-thms thy (dataname^.test-data)
  fun generate () =
    let
      val thmsstrings = String.concat (map (fn t => ((Syntax.string-of-term ctxt o
prop-of) t) ^\n
                                     ) thms)
    in
      thmsstrings
    end

  val test-data-str = Print-Mode.setmp [] (generate) ();
  val - = File.write (Path.explode filename) test-data-str;

in () end
>>

```

```

ML<<
val - =
  Outer-Syntax.command export-fw-data export fw test data to an external file
    Keyword.diag
    (Parse.name -- Parse.name
     >> (fn (filename,name) =>
        (Toplevel.keep (fn state => export-fw-data (Toplevel.context-of
state) filename name)))));
>>

```

This theory contains a collection of (rather unsorted lemmas) which are of general use when processing test specification (including Normalization) of the Firewall Testing heap.

definition *policyID* **where**
policyID = ($\lambda x. (Some (allow (x)))$)

lemma *setDistinct*: $\llbracket x \in a; x \notin b \rrbracket \implies a \neq b$
apply *auto*
done

lemma *setDistinct3*: $\llbracket x \neq y \rrbracket \implies$
 $\{\{(a, b, c). a \in x\}\} \neq \{\{(a, b, c). a \in y\}\}$
apply *auto*
done

lemma *setDistinct4*: $\llbracket (a::int) \neq x; a < b; x < y \rrbracket$
 $\implies \{a..b\} \neq \{x..y\}$
apply (*case-tac* $a < x$)

```

apply simp-all
apply (rule-tac  $x = a$  in setDistinct)
apply simp-all
apply (rule not-sym)
apply (rule-tac  $x = x$  in setDistinct)
apply simp-all
done

```

```

lemma setDistinct2:  $\llbracket x \neq y \rrbracket \implies$ 
 $\{(a, b). a \in x\} \neq \{(a, b). a \in y\}$ 
apply auto
done

```

```

lemma setDistinct1:  $\llbracket x \neq y \rrbracket \implies$ 
 $\{(a, b, c). a \in x\} \neq \{(a, b, c). a \in y\}$ 
apply auto
done

```

```

lemma AllowNMT:  $\llbracket x \neq \{\}; y \neq \{\} \rrbracket \implies$ 
 $\text{dom } (Cp \ (AllowPortFromTo \ \{\{(a, b, c). a \in x\}\ \{\{(a, b, c). a \in y\}\ (p, i))\}) \neq \{\}$ 
apply (simp add: PLemmas)
apply auto
done

```

```

lemma domDA1Cp:  $\text{dom } (Cp \ (DenyAll \oplus P)) = UNIV$ 
apply (simp add: PLemmas)
apply (auto split: option.splits simp: deny-all-def)
done

```

```

lemma aux:  $\llbracket x \neq a; y \neq b; (x \neq y \wedge x \neq b) \vee (a \neq b \wedge a \neq y) \rrbracket \implies \{x, a\} \neq \{y, b\}$ 
apply auto
done

```

```

lemma aux2:  $\{a, b\} = \{b, a\}$ 
apply auto
done

```

```

lemma deny-comm[simp]:  $(deny() = X) = (X = deny())$ 
by auto

```

```

lemma allow-comm[simp]:  $(allow() = X) = (X = allow())$ 
by auto

```

lemma *PO-add*: $PO\ P \implies P$
by (*simp add: PO-def*)

lemma *dom-2-subset1*: $dom\ (r\ o-f\ ((A \otimes_2 B) \ o\ (\lambda\ x.\ (x,x)))) \subseteq dom\ A$
apply (*simp add: dom-def prod-2-def policy-range-comp-def*)
apply (*simp split: option.splits decision.splits*)
apply *auto*
done

lemma *dom-2-comm*: $dom\ (A \otimes_2 B) = dom\ (B \otimes_2 A)$
apply (*simp add: dom-def prod-2-def policy-range-comp-def*)
apply (*simp split: option.splits decision.splits prod.splits*)
oops

lemma *dom-2-subset2*: $dom\ (r\ o-f\ ((A \otimes_2 B) \ o\ (\lambda\ x.\ (x,x)))) \subseteq dom\ B$
apply (*rule subsetI*)
apply (*simp add: dom-def prod-2-def policy-range-comp-def*)
apply (*simp split: option.splits decision.splits*)
done

lemma *dom-2*: $dom\ A = UNIV \implies$
 $dom\ (r\ o-f\ ((A \otimes_2 B) \ o\ (\lambda\ x.\ (x,x)))) = dom\ B$
apply (*rule equalityI*)
apply (*rule dom-2-subset2*)
apply (*rule subsetI*)
apply (*simp add: dom-def prod-2-def policy-range-comp-def*)
apply (*simp split: option.splits decision.splits*)
apply *auto*
done

lemma *domDA1*: $dom\ (C\ (DenyAll \oplus P)) = UNIV$
apply (*simp add: PLemmas*)
apply (*auto simp: deny-all-def C.simps*)
apply (*auto split: option.splits simp: deny-all-def*)
done

lemma *if-DA-simp*: $((if\ P\ then\ (deny-all\ x)\ else\ None) =$
 $None) = (\neg\ P)$
apply (*simp add: deny-all-def*)
done

lemma *if-DA-simp2*: $(\exists\ y.\ (if\ P\ then\ deny-all\ x\ else\ None) = Some\ y) = P$
apply *auto*

```

apply (simp add: deny-all-def)
done

```

```

lemma if-DA-simp3: ((if P then deny-all x else None) =
    Some ae) = (P  $\wedge$  deny-all x = Some ae)
apply (simp add: deny-all-def)
done

```

```

lemma if-DA-simp4: ((if P
    then Some (allow x)
    else None) =
    None) = ( $\neg$  P)
apply auto
done

```

```

lemma if-DA-simp5: ((if P
    then Some (deny x)
    else None) =
    None) = ( $\neg$  P)
apply auto
done

```

```

lemma setDistinct6:  $\llbracket x \neq y \rrbracket \implies$ 
     $\{(a, b, c). a = x\} \neq \{(a, b, c). a = y\}$ 
by auto

```

```

definition ip2int :: (int  $\times$  int  $\times$  int  $\times$  int)  $\Rightarrow$  int where
    ip2int x = (((fst x)*16777216) + ((fst (snd x))*65536) +
        ((fst (snd (snd x)))*256) + snd (snd (snd x)))

```

```

lemmas if-DA = if-DA-simp if-DA-simp2 if-DA-simp3 if-DA-simp4 if-DA-simp5

```

```

declare if-DA [simp]

```

```

end

```

4. Packets and Networks

```

theory NetworkCore
imports Main UPF
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 an integer:

type-synonym $id = int$

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:

class adr

type-synonym $'\alpha \text{ src} = '\alpha$

type-synonym $'\alpha \text{ dest} = '\alpha$

instance $int :: adr \dots$

instance $nat :: adr \dots$

instance $fun :: (adr, adr) \rightarrow adr \dots$

instance $prod :: (adr, adr) \rightarrow adr \dots$

The content is also specified with an unconstrained generic type:

type-synonym $'\beta \text{ content} = '\beta$

For applications where 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 $DummyContent = data$

Finally, a packet is:

type-synonym $(' \alpha, ' \beta) \text{ packet} = id \times ' \alpha \text{ src} \times ' \alpha \text{ dest} \times ' \beta \text{ content}$

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 are modelled as part of the address. Additionally, stateful firewalls often determine the protocol by the content of a packet.

definition $src :: (' \alpha :: adr, ' \beta) \text{ packet} \Rightarrow ' \alpha$

where $src = fst \circ snd$

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.

class *port*

instance *int* :: *port* ..

instance *nat* :: *port* ..

instance *fun* :: (*port*,*port*) *port* ..

instance *prod* :: (*port*,*port*) *port* ..

A packet therefore has two parameters, the first being the address, the second the content. 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.

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

definition *id* :: ($\alpha :: \text{adr}, \beta$) *packet* \Rightarrow *id*

where *id* = *fst*

definition *dest* :: ($\alpha :: \text{adr}, \beta$) *packet* \Rightarrow α *dest*

where *dest* = *fst* o *snd* o *snd*

definition *content* :: ($\alpha :: \text{adr}, \beta$) *packet* \Rightarrow β *content*

where *content* = *snd* o *snd* o *snd*

datatype *protocol* = *tcp* | *udp*

lemma *either*: $\llbracket a \neq \text{tcp}; a \neq \text{udp} \rrbracket \Longrightarrow \text{False}$

by (*case-tac a, simp-all*)

lemma *either2*[*simp*]: $(a \neq \text{tcp}) = (a = \text{udp})$

by (*case-tac a, simp-all*)

lemma *either3*[*simp*]: $(a \neq \text{udp}) = (a = \text{tcp})$

by (*case-tac a, simp-all*)

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* :: ($\alpha :: \text{adr}, \beta$) *packet* \Rightarrow $\gamma :: \text{port}$

consts *dest-port* :: ($\alpha :: \text{adr}, \beta$) *packet* \Rightarrow $\gamma :: \text{port}$

consts *src-protocol* :: ($\alpha :: \text{adr}, \beta$) *packet* \Rightarrow *protocol*

consts *dest-protocol* :: ($\alpha :: \text{adr}, \beta$) *packet* \Rightarrow *protocol*

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

type-synonym α *net* = α *set set*

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

definition

$in_subnet :: 'α::adr \Rightarrow 'α\ net \Rightarrow bool$ (**infixl** \sqsubset 100) **where**
 $in_subnet\ a\ S = (\exists\ s \in S. a \in s)$

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* :: $'α::adr \Rightarrow 'α\ net$

lemmas *packet-defs* = *in-subnet-def id-def content-def src-def dest-def*

end

5. Address Representations

theory

NetworkModels

imports

DatatypeAddress

DatatypePort

IntegerAddress

IntegerPort

IntegerPort-TCPUDP

IPv4

IPv4-TCPUDP

begin

One can think of many different possible address representations. In this distribution,

we include seven 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 Integer¹.
- **adr_i**: An address in an Integer.
- **adr_ip**: An address is a pair of an Integer and a port (which is again an Integer).
- **adr_ipp**: An address is a triple consisting of two Integers modelling the IP address and the port number, and the specification of the network protocol
- **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.
- **IPv4_TCPUDP**: The same as above, but including additionally the specification of the network protocol.

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

end

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}}
```

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

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
```

```
type-synonym
```

```
port = int
```

```
type-synonym
```

```
DatatypePort = (DatatypeAddress × port)
```

```
instance DatatypeAddress :: adr ..
```

```
definition
```

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

```
definition
```

```
intranet::DatatypePort net where
intranet = {{(a,b). a = intranet-adr}}
```

```
definition
```

```
internet::DatatypePort net where
internet = {{(a,b). a = internet-adr}}
```

```
defs (overloaded)
```

```
src-port-def: src-port (x::(DatatypePort,'β) packet) ≡ (snd o fst o snd) x
dest-port-def: dest-port (x::(DatatypePort,'β) packet) ≡ (snd o fst o snd o snd) x
subnet-of-def: subnet-of (x::DatatypePort) ≡ {{(a,b). a = fst x}}
```

```
lemma src-port : src-port ((a,x,d,e)::(DatatypePort,'β) packet) = snd x
by (simp add: src-port-def in-subnet)
```

```
lemma dest-port : dest-port ((a,d,x,e)::(DatatypePort,'β) packet) = snd x
by (simp add: dest-port-def in-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.

```
type-synonym
   $adr_i = int$ 
```

```
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.

```
type-synonym
   $address = int$ 
```

```
type-synonym
   $port = int$ 
```

```
type-synonym
   $adr_{ip} = address \times port$ 
```

```
defs (overloaded)
```

```
 $src\text{-}port\text{-}def: src\text{-}port\ (x::(adr_{ip},'\beta)\ packet) \equiv (snd\ o\ fst\ o\ snd)\ x$ 
```

```
 $dest\text{-}port\text{-}def: dest\text{-}port\ (x::(adr_{ip},'\beta)\ packet) \equiv (snd\ o\ fst\ o\ snd\ o\ snd)\ x$ 
```

```
 $subnet\text{-}of\text{-}def: subnet\text{-}of\ (x::(adr_{ip})) \equiv \{\{(a,b). a = fst\ x\}\}$ 
```

```
lemma  $src\text{-}port: src\text{-}port\ (a,x::adr_{ip},d,e) = snd\ x$ 
by ( $simp\ add: src\text{-}port\text{-}def\ in\ subnet$ )
```

```
lemma  $dest\text{-}port: dest\text{-}port\ (a,d,x::adr_{ip},e) = snd\ x$ 
```

```

    by (simp add: dest-port-def in-subnet)

lemmas  $adr_{ip}$ Lemmas = src-port dest-port src-port-def dest-port-def

end

```

5.5. Integer Addresses with Ports and Protocols

```

theory IntegerPort-TCPUDP
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.

```

type-synonym
  address = int

```

```

type-synonym
  port = int

```

```

type-synonym
   $adr_{ipp}$  = address  $\times$  port  $\times$  protocol

```

```

instance protocol ::  $adr$  ..

```

```

defs (overloaded)

```

```

src-port-def: src-port ( $x::(adr_{ipp},'\beta)$  packet)  $\equiv$  (fst o snd o fst o snd) x
dest-port-def: dest-port ( $x::(adr_{ipp},'\beta)$  packet)  $\equiv$  (fst o snd o fst o snd o snd) x
subnet-of-def: subnet-of ( $x::(adr_{ipp})$ )  $\equiv$   $\{\{(a,b,c). a = \text{fst } x\}\}$ 
src-protocol-def: src-protocol ( $x::(adr_{ipp},'\beta)$  packet)  $\equiv$  (snd o snd o fst o snd) x
dest-protocol-def: dest-protocol ( $x::(adr_{ipp},'\beta)$  packet)  $\equiv$  (snd o snd o fst o snd o snd) x

```

```

lemma src-port: src-port ( $a,x::adr_{ipp},d,e$ ) = fst (snd x)
  by (simp add: src-port-def in-subnet)

```

```

lemma dest-port: dest-port ( $a,d,x::adr_{ipp},e$ ) = fst (snd x)
  by (simp add: dest-port-def in-subnet)

```

Common test constraints:

```

definition port-positive :: ( $adr_{ipp},'b$ ) packet  $\Rightarrow$  bool where
  port-positive x = (dest-port x > (0::port))

```

```

definition fix-values :: ( $adr_{ipp},DummyContent$ ) packet  $\Rightarrow$  bool where
  fix-values x = (src-port x = (1::port)  $\wedge$  src-protocol x = udp  $\wedge$  content x = data  $\wedge$  id x = 1)

```

lemmas adr_{ipp} *Lemmas* = *src-port dest-port src-port-def dest-port-def src-protocol-def dest-protocol-def subnet-of-def*

lemmas adr_{ipp} *TestConstraints* = *port-positive-def fix-values-def*

end

5.6. 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.

type-synonym
 $ipv4\text{-}ip = (int \times int \times int \times int)$

type-synonym
 $port = int$

type-synonym
 $ipv4 = (ipv4\text{-}ip \times port)$

defs (overloaded)
 $src\text{-}port\text{-}def: src\text{-}port (x::(ipv4, '\beta) packet) \equiv (snd \ o \ fst \ o \ snd) \ x$
defs (overloaded)
 $dest\text{-}port\text{-}def: dest\text{-}port (x::(ipv4, '\beta) packet) \equiv (snd \ o \ fst \ o \ snd \ o \ snd) \ x$
defs (overloaded)
 $subnet\text{-}of\text{-}def: subnet\text{-}of (x::ipv4) \equiv \{\{(a,b). \ a = \ fst \ x\}\}$

definition $subnet\text{-}of\text{-}ip :: ipv4\text{-}ip \Rightarrow ipv4 \ net$
where $subnet\text{-}of\text{-}ip \ ip = \{\{(a,b). \ (a = \ ip)\}\}$

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

lemma $dest\text{-}port: dest\text{-}port (a,d,(x::ipv4),e) = snd \ x$
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 UPF
begin

```

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

```

type-synonym ('α, 'β) FWPolicy = ('α, 'β) packet  $\mapsto$  unit

```

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 :: ('α, 'β) packet  $\Rightarrow$  ('α, 'β) FWPolicy  $\Rightarrow$  bool where
    p-accept p pol = (pol p = [allow ()])

```

```

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

$allow-all-from :: 'α::adr\ net \Rightarrow (('α, 'β)\ packet \mapsto unit)\ \mathbf{where}$
 $allow-all-from\ src-net = \{pa.\ src\ pa \sqsubset src-net\} \triangleleft A_U$

definition

$deny-all-from :: 'α::adr\ net \Rightarrow (('α, 'β)\ packet \mapsto unit)\ \mathbf{where}$
 $deny-all-from\ src-net = \{pa.\ src\ pa \sqsubset src-net\} \triangleleft D_U$

definition

$allow-all-to :: 'α::adr\ net \Rightarrow (('α, 'β)\ packet \mapsto unit)\ \mathbf{where}$
 $allow-all-to\ dest-net = \{pa.\ dest\ pa \sqsubset dest-net\} \triangleleft A_U$

definition

$deny-all-to :: 'α::adr\ net \Rightarrow (('α, 'β)\ packet \mapsto unit)\ \mathbf{where}$
 $deny-all-to\ dest-net = \{pa.\ dest\ pa \sqsubset dest-net\} \triangleleft D_U$

definition

$allow-all-from-to :: 'α::adr\ net \Rightarrow 'α::adr\ net \Rightarrow (('α, 'β)\ packet \mapsto unit)\ \mathbf{where}$
 $allow-all-from-to\ src-net\ dest-net =$
 $\{pa.\ src\ pa \sqsubset src-net \wedge dest\ pa \sqsubset dest-net\} \triangleleft A_U$

definition

$deny-all-from-to :: 'α::adr\ net \Rightarrow 'α::adr\ net \Rightarrow (('α, 'β)\ packet \mapsto unit)\ \mathbf{where}$
 $deny-all-from-to\ src-net\ dest-net =$
 $\{pa.\ src\ pa \sqsubset src-net \wedge dest\ pa \sqsubset dest-net\} \triangleleft D_U$

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

lemmas $PolicyCombinators = allow-all-from-def\ deny-all-from-def$
 $allow-all-to-def\ deny-all-to-def\ allow-all-from-to-def$
 $deny-all-from-to-def\ UPFDefs$

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 *PolicyCombinators* 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

$allow_all_from_port :: ' \alpha :: adr\ net \Rightarrow (' \gamma :: port) \Rightarrow ((' \alpha, '\beta) packet \mapsto unit)$ **where**
 $allow_all_from_port\ src_net\ s_port =$
 $\{pa. src_port\ pa = s_port\} \triangleleft allow_all_from\ src_net$

definition

$deny_all_from_port :: ' \alpha :: adr\ net \Rightarrow (' \gamma :: port) \Rightarrow ((' \alpha, '\beta) packet \mapsto unit)$ **where**
 $deny_all_from_port\ src_net\ s_port =$
 $\{pa. src_port\ pa = s_port\} \triangleleft deny_all_from\ src_net$

definition

$allow_all_to_port :: ' \alpha :: adr\ net \Rightarrow (' \gamma :: port) \Rightarrow ((' \alpha, '\beta) packet \mapsto unit)$ **where**
 $allow_all_to_port\ dest_net\ d_port =$
 $\{pa. dest_port\ pa = d_port\} \triangleleft allow_all_to\ dest_net$

definition

$deny_all_to_port :: ' \alpha :: adr\ net \Rightarrow (' \gamma :: port) \Rightarrow ((' \alpha, '\beta) packet \mapsto unit)$ **where**
 $deny_all_to_port\ dest_net\ d_port =$
 $\{pa. dest_port\ pa = d_port\} \triangleleft deny_all_to\ dest_net$

definition

$allow_all_from_port_to :: ' \alpha :: adr\ net \Rightarrow (' \gamma :: port) \Rightarrow ' \alpha :: adr\ net \Rightarrow ((' \alpha, '\beta) packet \mapsto unit)$
where
 $allow_all_from_port_to\ src_net\ s_port\ dest_net$
 $= \{pa. src_port\ pa = s_port\} \triangleleft allow_all_from_to\ src_net\ dest_net$

definition

$deny_all_from_port_to :: ' \alpha :: adr\ net \Rightarrow (' \gamma :: port) \Rightarrow ' \alpha :: adr\ net \Rightarrow ((' \alpha, '\beta) packet \mapsto unit)$
where
 $deny_all_from_port_to\ src_net\ s_port\ dest_net$
 $= \{pa. src_port\ pa = s_port\} \triangleleft deny_all_from_to\ src_net\ dest_net$

definition

$allow_all_from_port_to_port :: ' \alpha :: adr\ net \Rightarrow (' \gamma :: port) \Rightarrow ' \alpha :: adr\ net \Rightarrow (' \gamma :: port) \Rightarrow$
 $((' \alpha, '\beta) packet \mapsto unit)$ **where**
 $allow_all_from_port_to_port\ src_net\ s_port\ dest_net\ d_port =$
 $\{pa. dest_port\ pa = d_port\} \triangleleft allow_all_from_port_to\ src_net\ s_port\ dest_net$

definition

$deny_all_from_port_to_port :: ' \alpha :: adr\ net \Rightarrow (' \gamma :: port) \Rightarrow ' \alpha :: adr\ net \Rightarrow$
 $' \gamma :: port \Rightarrow ((' \alpha, '\beta) packet \mapsto unit)$ **where**
 $deny_all_from_port_to_port\ src_net\ s_port\ dest_net\ d_port =$
 $\{pa. dest_port\ pa = d_port\} \triangleleft deny_all_from_port_to\ src_net\ s_port\ dest_net$

definition

$allow_all_from_to_port :: ' \alpha :: adr\ net \Rightarrow ' \alpha :: adr\ net \Rightarrow$
 $' \gamma :: port \Rightarrow ((' \alpha, '\beta) packet \mapsto unit)$ **where**

allow-all-from-to-port *src-net* *dest-net* *d-port* =
 $\{pa. \text{dest-port } pa = d\text{-port}\} \triangleleft \text{allow-all-from-to } src\text{-net } dest\text{-net}$

definition

deny-all-from-to-port :: $'\alpha::adr \text{ net} \Rightarrow '\alpha::adr \text{ net} \Rightarrow '\gamma::port \Rightarrow$
 $(('\alpha, '\beta) \text{ packet} \mapsto unit)$ **where**

deny-all-from-to-port *src-net* *dest-net* *d-port* = $\{pa. \text{dest-port } pa = d\text{-port}\} \triangleleft \text{deny-all-from-to}$
src-net *dest-net*

definition

allow-from-port-to :: $'\gamma::port \Rightarrow '\alpha::adr \text{ net} \Rightarrow '\alpha::adr \text{ net} \Rightarrow (('\alpha, '\beta) \text{ packet} \mapsto unit)$

where

allow-from-port-to *port* *src-net* *dest-net* =
 $\{pa. \text{src-port } pa = port\} \triangleleft \text{allow-all-from-to } src\text{-net } dest\text{-net}$

definition

deny-from-port-to :: $'\gamma::port \Rightarrow '\alpha::adr \text{ net} \Rightarrow '\alpha::adr \text{ net} \Rightarrow (('\alpha, '\beta) \text{ packet} \mapsto unit)$

where

deny-from-port-to *port* *src-net* *dest-net* =
 $\{pa. \text{src-port } pa = port\} \triangleleft \text{deny-all-from-to } src\text{-net } dest\text{-net}$

definition

allow-from-to-port :: $'\gamma::port \Rightarrow '\alpha::adr \text{ net} \Rightarrow '\alpha::adr \text{ net} \Rightarrow (('\alpha, '\beta) \text{ packet} \mapsto unit)$

where

allow-from-to-port *port* *src-net* *dest-net* =
 $\{pa. \text{dest-port } pa = port\} \triangleleft \text{allow-all-from-to } src\text{-net } dest\text{-net}$

definition

deny-from-to-port :: $'\gamma::port \Rightarrow '\alpha::adr \text{ net} \Rightarrow '\alpha::adr \text{ net} \Rightarrow (('\alpha, '\beta) \text{ packet} \mapsto unit)$

where

deny-from-to-port *port* *src-net* *dest-net* =
 $\{pa. \text{dest-port } pa = port\} \triangleleft \text{deny-all-from-to } src\text{-net } dest\text{-net}$

definition

allow-from-ports-to :: $'\gamma::port \text{ set} \Rightarrow '\alpha::adr \text{ net} \Rightarrow '\alpha::adr \text{ net} \Rightarrow$
 $(('\alpha, '\beta) \text{ packet} \mapsto unit)$ **where**

allow-from-ports-to *ports* *src-net* *dest-net* =
 $\{pa. \text{src-port } pa \in ports\} \triangleleft \text{allow-all-from-to } src\text{-net } dest\text{-net}$

definition

allow-from-to-ports :: $'\gamma::port \text{ set} \Rightarrow '\alpha::adr \text{ net} \Rightarrow '\alpha::adr \text{ net} \Rightarrow$
 $(('\alpha, '\beta) \text{ packet} \mapsto unit)$ **where**

allow-from-to-ports *ports* *src-net* *dest-net* =
 $\{pa. \text{dest-port } pa \in ports\} \triangleleft \text{allow-all-from-to } src\text{-net } dest\text{-net}$

definition

deny-from-ports-to :: $'\gamma::port \text{ set} \Rightarrow '\alpha::adr \text{ net} \Rightarrow '\alpha::adr \text{ net} \Rightarrow$
 $(('\alpha, '\beta) \text{ packet} \mapsto unit)$ **where**

deny-from-ports-to ports src-net dest-net =
{pa. src-port pa ∈ ports} ◁ deny-all-from-to src-net dest-net

definition

deny-from-to-ports :: 'γ::port set ⇒ 'α::adr net ⇒ 'α::adr net ⇒
(('α,'β) packet ↦ unit) where
deny-from-to-ports ports src-net dest-net =
{pa. dest-port pa ∈ ports} ◁ deny-all-from-to src-net dest-net

definition

allow-all-from-port-tos:: 'α::adr net ⇒ ('γ::port) set ⇒ 'α::adr net ⇒ (('α,'β) packet ↦ unit)
where
allow-all-from-port-tos src-net s-port dest-net
= {pa. dest-port pa ∈ s-port} ◁ allow-all-from-to src-net dest-net

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

lemmas *PortCombinatorsCore =*

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
deny-from-ports-to-def deny-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 allow-all-from-port-tos-def

lemmas *PortCombinators =*

PortCombinatorsCore PolicyCombinators

end

6.4. Policy Combinators with Ports and Protocols

theory *ProtocolPortCombinators*

imports *PortCombinators*

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

- $src_port :: ('α, 'β)packet \Rightarrow ('γ :: port)$
- $dest_port :: ('α, 'β)packet \Rightarrow ('γ :: port)$

definition

$allow_all_from_port_prot :: protocol \Rightarrow 'α::adr\ net \Rightarrow ('γ::port) \Rightarrow (('α, 'β)\ packet \mapsto unit)$
where
 $allow_all_from_port_prot\ p\ src_net\ s_port =$
 $\{pa.\ dest_protocol\ pa = p\} \triangleleft allow_all_from_port\ src_net\ s_port$

definition

$deny_all_from_port_prot :: protocol \Rightarrow 'α::adr\ net \Rightarrow ('γ::port) \Rightarrow (('α, 'β)\ packet \mapsto unit)$
where
 $deny_all_from_port_prot\ p\ src_net\ s_port =$
 $\{pa.\ dest_protocol\ pa = p\} \triangleleft deny_all_from_port\ src_net\ s_port$

definition

$allow_all_to_port_prot :: protocol \Rightarrow 'α::adr\ net \Rightarrow ('γ::port) \Rightarrow (('α, 'β)\ packet \mapsto unit)$ **where**
 $allow_all_to_port_prot\ p\ dest_net\ d_port =$
 $\{pa.\ dest_protocol\ pa = p\} \triangleleft allow_all_to_port\ dest_net\ d_port$

definition

$deny_all_to_port_prot :: protocol \Rightarrow 'α::adr\ net \Rightarrow ('γ::port) \Rightarrow (('α, 'β)\ packet \mapsto unit)$ **where**
 $deny_all_to_port_prot\ p\ dest_net\ d_port =$
 $\{pa.\ dest_protocol\ pa = p\} \triangleleft deny_all_to_port\ dest_net\ d_port$

definition

$allow_all_from_port_to_port_prot :: protocol \Rightarrow 'α::adr\ net \Rightarrow ('γ::port) \Rightarrow 'α::adr\ net \Rightarrow (('α, 'β)\ packet \mapsto unit)$
where
 $allow_all_from_port_to_port_prot\ p\ src_net\ s_port\ dest_net =$
 $\{pa.\ dest_protocol\ pa = p\} \triangleleft allow_all_from_port_to\ src_net\ s_port\ dest_net$

definition

$deny_all_from_port_to_port_prot :: protocol \Rightarrow 'α::adr\ net \Rightarrow ('γ::port) \Rightarrow 'α::adr\ net \Rightarrow (('α, 'β)\ packet \mapsto unit)$
where
 $deny_all_from_port_to_port_prot\ p\ src_net\ s_port\ dest_net =$
 $\{pa.\ dest_protocol\ pa = p\} \triangleleft deny_all_from_port_to\ src_net\ s_port\ dest_net$

definition

$allow_all_from_port_to_port_prot :: protocol \Rightarrow 'α::adr\ net \Rightarrow ('γ::port) \Rightarrow 'α::adr\ net \Rightarrow ('γ::port) \Rightarrow (('α, 'β)\ packet \mapsto unit)$ **where**
 $allow_all_from_port_to_port_prot\ p\ src_net\ s_port\ dest_net\ d_port =$
 $\{pa.\ dest_protocol\ pa = p\} \triangleleft allow_all_from_port_to_port\ src_net\ s_port\ dest_net\ d_port$

definition

$deny_all_from_port_to_port_prot :: protocol \Rightarrow 'α::adr\ net \Rightarrow ('γ::port) \Rightarrow 'α::adr\ net \Rightarrow$

$\gamma::port \Rightarrow ((\alpha, \beta) \text{ packet} \mapsto \text{unit})$ **where**
 $\text{deny-all-from-port-to-port-prot } p \text{ src-net s-port dest-net d-port} =$
 $\{pa. \text{ dest-protocol } pa = p\} \triangleleft \text{deny-all-from-port-to-port src-net s-port dest-net d-port}$

definition

$\text{allow-all-from-to-port-prot} :: \text{protocol} \Rightarrow \alpha::\text{adr net} \Rightarrow \alpha::\text{adr net} \Rightarrow$
 $\gamma::port \Rightarrow ((\alpha, \beta) \text{ packet} \mapsto \text{unit})$ **where**
 $\text{allow-all-from-to-port-prot } p \text{ src-net dest-net d-port} =$
 $\{pa. \text{ dest-protocol } pa = p\} \triangleleft \text{allow-all-from-to-port src-net dest-net d-port}$

definition

$\text{deny-all-from-to-port-prot} :: \text{protocol} \Rightarrow \alpha::\text{adr net} \Rightarrow \alpha::\text{adr net} \Rightarrow \gamma::port \Rightarrow$
 $((\alpha, \beta) \text{ packet} \mapsto \text{unit})$ **where**
 $\text{deny-all-from-to-port-prot } p \text{ src-net dest-net d-port} =$
 $\{pa. \text{ dest-protocol } pa = p\} \triangleleft \text{deny-all-from-to-port src-net dest-net d-port}$

definition

$\text{allow-from-port-to-prot} :: \text{protocol} \Rightarrow \gamma::port \Rightarrow \alpha::\text{adr net} \Rightarrow \alpha::\text{adr net} \Rightarrow ((\alpha, \beta) \text{ packet}$
 $\mapsto \text{unit})$
where
 $\text{allow-from-port-to-prot } p \text{ port src-net dest-net} =$
 $\{pa. \text{ dest-protocol } pa = p\} \triangleleft \text{allow-from-port-to port src-net dest-net}$

definition

$\text{deny-from-port-to-prot} :: \text{protocol} \Rightarrow \gamma::port \Rightarrow \alpha::\text{adr net} \Rightarrow \alpha::\text{adr net} \Rightarrow ((\alpha, \beta) \text{ packet}$
 $\mapsto \text{unit})$
where
 $\text{deny-from-port-to-prot } p \text{ port src-net dest-net} =$
 $\{pa. \text{ dest-protocol } pa = p\} \triangleleft \text{deny-from-port-to port src-net dest-net}$

definition

$\text{allow-from-to-port-prot} :: \text{protocol} \Rightarrow \gamma::port \Rightarrow \alpha::\text{adr net} \Rightarrow \alpha::\text{adr net} \Rightarrow ((\alpha, \beta) \text{ packet}$
 $\mapsto \text{unit})$
where
 $\text{allow-from-to-port-prot } p \text{ port src-net dest-net} =$
 $\{pa. \text{ dest-protocol } pa = p\} \triangleleft \text{allow-from-to-port port src-net dest-net}$

definition

$\text{deny-from-to-port-prot} :: \text{protocol} \Rightarrow \gamma::port \Rightarrow \alpha::\text{adr net} \Rightarrow \alpha::\text{adr net} \Rightarrow ((\alpha, \beta) \text{ packet}$
 $\mapsto \text{unit})$
where
 $\text{deny-from-to-port-prot } p \text{ port src-net dest-net} =$
 $\{pa. \text{ dest-protocol } pa = p\} \triangleleft \text{deny-from-to-port port src-net dest-net}$

definition

$\text{allow-from-ports-to-prot} :: \text{protocol} \Rightarrow \gamma::port \text{ set} \Rightarrow \alpha::\text{adr net} \Rightarrow \alpha::\text{adr net} \Rightarrow$
 $((\alpha, \beta) \text{ packet} \mapsto \text{unit})$ **where**
 $\text{allow-from-ports-to-prot } p \text{ ports src-net dest-net} =$
 $\{pa. \text{ dest-protocol } pa = p\} \triangleleft \text{allow-from-ports-to ports src-net dest-net}$

definition

allow-from-to-ports-prot :: *protocol* => ' γ ::*port set* => ' α ::*adr net* => ' α ::*adr net* =>
 ((α, β) *packet* \mapsto *unit*) **where**
allow-from-to-ports-prot *p ports src-net dest-net* =
 {*pa. dest-protocol pa = p*} \triangleleft *allow-from-to-ports ports src-net dest-net*

definition

deny-from-ports-to-prot :: *protocol* => ' γ ::*port set* => ' α ::*adr net* => ' α ::*adr net* =>
 ((α, β) *packet* \mapsto *unit*) **where**
deny-from-ports-to-prot *p ports src-net dest-net* =
 {*pa. dest-protocol pa = p*} \triangleleft *deny-from-ports-to ports src-net dest-net*

definition

deny-from-to-ports-prot :: *protocol* => ' γ ::*port set* => ' α ::*adr net* => ' α ::*adr net* =>
 ((α, β) *packet* \mapsto *unit*) **where**
deny-from-to-ports-prot *p ports src-net dest-net* =
 {*pa. dest-protocol pa = p*} \triangleleft *deny-from-to-ports ports src-net dest-net*

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

lemmas *ProtocolCombinatorsCore* =

allow-all-from-port-prot-def deny-all-from-port-prot-def allow-all-to-port-prot-def
deny-all-to-port-prot-def allow-all-from-to-port-prot-def
deny-all-from-to-port-prot-def
allow-from-ports-to-prot-def allow-from-to-ports-prot-def
deny-from-ports-to-prot-def deny-from-to-ports-prot-def
allow-all-from-port-to-prot-def deny-all-from-port-to-prot-def
allow-from-port-to-prot-def allow-from-to-port-prot-def deny-from-to-port-prot-def
deny-from-port-to-prot-def

lemmas *ProtocolCombinators* = *PortCombinators.PortCombinators*
ProtocolCombinatorsCore

end

6.5. 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*: $x \neq 80 \implies x \neq \text{http}$

by (*simp add: http-def*)

lemma *http2*: $x \neq 80 \implies \text{http} \neq x$

by (*simp add: http-def*)

definition *smtp::int* **where** *smtp* = 25

lemma *smtp1*: $x \neq 25 \implies x \neq \text{smtp}$

by (*simp add: smtp-def*)

lemma *smtp2*: $x \neq 25 \implies \text{smtp} \neq x$

by (*simp add: smtp-def*)

definition *ftp::int* **where** *ftp* = 21

lemma *ftp1*: $x \neq 21 \implies x \neq \text{ftp}$

by (*simp add: ftp-def*)

lemma *ftp2*: $x \neq 21 \implies \text{ftp} \neq x$

by (*simp add: ftp-def*)

And so on for all desired port numbers.

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

end

6.6. NAT

7. Network Address Translation

theory *NAT*

imports *../PacketFilter/PacketFilter*

begin

definition *src2pool* :: $'\alpha$ *set* $\Rightarrow (' \alpha :: \text{adr}, ' \beta)$ *packet* $\Rightarrow (' \alpha, ' \beta)$ *packet set* **where**

src2pool *t* = $(\lambda p. (\{(i, s, d, da). (i = \text{id } p \wedge s \in t \wedge d = \text{dest } p \wedge da = \text{content } p)\}))$

definition *src2poolAP* **where**

$src2poolAP\ t = A_f\ (src2pool\ t)$

definition $srcNat2pool :: 'α\ set \Rightarrow 'α\ set \Rightarrow ('α::adr, 'β)\ packet \mapsto ('α, 'β)\ packet\ set$ **where**
 $srcNat2pool\ srcs\ transl = \{x.\ src\ x \in srcs\} \triangleleft (src2poolAP\ transl)$

definition $src2poolPort :: int\ set \Rightarrow (adr_{ip}, 'β)\ packet \Rightarrow (adr_{ip}, 'β)\ packet\ set$ **where**
 $src2poolPort\ t = (\lambda\ p.\ (\{(i, (s1, s2), (d1, d2), da).\$
 $(i = id\ p \wedge s1 \in t \wedge s2 = (snd\ (src\ p)) \wedge d1 = (fst\ (dest\ p)) \wedge$
 $d2 = snd\ (dest\ p) \wedge da = content\ p)\}))$

definition $src2poolPort-Protocol :: int\ set \Rightarrow (adr_{ipp}, 'β)\ packet \Rightarrow (adr_{ipp}, 'β)\ packet\ set$ **where**
 $src2poolPort-Protocol\ t = (\lambda\ p.\ (\{(i, (s1, s2, s3), (d1, d2, d3), da).\$
 $(i = id\ p \wedge s1 \in t \wedge s2 = (fst\ (snd\ (src\ p))) \wedge s3 = snd\ (snd\ (src\ p)) \wedge$
 $(d1, d2, d3) = dest\ p \wedge da = content\ p)\}))$

definition $srcNat2pool-IntPort :: address\ set \Rightarrow address\ set \Rightarrow$
 $(adr_{ip}, 'β)\ packet \mapsto (adr_{ip}, 'β)\ packet\ set$ **where**
 $srcNat2pool-IntPort\ srcs\ transl =$
 $\{x.\ fst\ (src\ x) \in srcs\} \triangleleft (A_f\ (src2poolPort\ transl))$

definition $srcNat2pool-IntProtocolPort :: int\ set \Rightarrow int\ set \Rightarrow$
 $(adr_{ipp}, 'β)\ packet \mapsto (adr_{ipp}, 'β)\ packet\ set$ **where**
 $srcNat2pool-IntProtocolPort\ srcs\ transl =$
 $\{x.\ (fst\ ((src\ x))) \in srcs\} \triangleleft (A_f\ (src2poolPort-Protocol\ transl))$

definition $srcPat2poolPort-t :: int\ set \Rightarrow (adr_{ip}, 'β)\ packet \Rightarrow (adr_{ip}, 'β)\ packet\ set$ **where**
 $srcPat2poolPort-t\ t = (\lambda\ p.\ (\{(i, (s1, s2), (d1, d2), da).\$
 $(i = id\ p \wedge s1 \in t \wedge d1 = (fst\ (dest\ p)) \wedge d2 = snd\ (dest\ p) \wedge da = content\ p)\}))$

definition $srcPat2poolPort-Protocol-t :: int\ set \Rightarrow (adr_{ipp}, 'β)\ packet \Rightarrow (adr_{ipp}, 'β)\ packet\ set$ **where**
 $srcPat2poolPort-Protocol-t\ t = (\lambda\ p.\ (\{(i, (s1, s2, s3), (d1, d2, d3), da).\$
 $(i = id\ p \wedge s1 \in t \wedge s3 = src\ protocol\ p \wedge (d1, d2, d3) = dest\ p \wedge da = content\ p)\}))$

definition $srcPat2pool-IntPort :: int\ set \Rightarrow int\ set \Rightarrow (adr_{ip}, 'β)\ packet \mapsto$
 $(adr_{ip}, 'β)\ packet\ set$ **where**
 $srcPat2pool-IntPort\ srcs\ transl =$
 $\{x.\ (fst\ (src\ x)) \in srcs\} \triangleleft (A_f\ (srcPat2poolPort-t\ transl))$

definition $srcPat2pool-IntProtocol ::$
 $int\ set \Rightarrow int\ set \Rightarrow (adr_{ipp}, 'β)\ packet \mapsto (adr_{ipp}, 'β)\ packet\ set$ **where**

$srcPat2pool-IntProtocol\ srcs\ transl =$
 $\{x. (fst\ (src\ x)) \in srcs\} \triangleleft (A_f\ (srcPat2poolPort-Protocol-t\ transl))$

The following lemmas are used for achieving a "standard" output format of NATted packets for use in a firewall execution tool.

lemma *datasimp*: $\{(i, (s1, s2, s3), aba).$
 $\forall a\ aa\ b\ ba. aba = ((a, aa, b), ba) \longrightarrow i = i1 \wedge s1 = i101 \wedge$
 $s3 = iudp \wedge a = i110 \wedge aa = X606X3 \wedge b = X607X4 \wedge ba = data\} =$
 $\{(i, (s1, s2, s3), aba).$
 $i = i1 \wedge s1 = i101 \wedge s3 = iudp \wedge (\lambda ((a,aa,b),ba). a = i110 \wedge aa = X606X3 \wedge$
 $b = X607X4 \wedge ba = data) aba\}$
by *auto*

lemma *datasimp2*: $\{(i, (s1, s2, s3), aba).$
 $\forall a\ aa\ b\ ba. aba = ((a, aa, b), ba) \longrightarrow i = i1 \wedge s1 = i132 \wedge s3 = iudp \wedge$
 $s2 = i1 \wedge a = i110 \wedge aa = i4 \wedge b = iudp \wedge ba = data\} =$
 $\{(i, (s1, s2, s3), aba).$
 $i = i1 \wedge s1 = i132 \wedge s3 = iudp \wedge s2 = i1 \wedge (\lambda ((a,aa,b),ba). a = i110 \wedge$
 $aa = i4 \wedge b = iudp \wedge ba = data) aba\}$
by *auto*

lemma *datasimp3*: $\{(i, (s1, s2, s3), aba).$
 $ALL\ a\ aa\ b\ ba. aba = ((a, aa, b), ba) \longrightarrow i = i1 \ \&\ i115 < s1 \ \&\ s1 < i124 \ \&$
 $s3 = iudp \ \&\ s2 = ii1 \ \&\ a = i110 \ \&\ aa = i3 \ \&\ b = itcp \ \&\ ba = data\} =$
 $\{(i, (s1, s2, s3), aba).$
 $i = i1 \ \&\ i115 < s1 \ \&\ s1 < i124 \ \&\ s3 = iudp \ \&\ s2 = ii1 \ \&\ (\lambda ((a,aa,b),ba).$
 $a = i110 \ \&\ aa = i3 \ \&\ b = itcp \ \&\ ba = data) aba\}$
apply *auto*
done

lemma *datasimp4*: $\{(i, (s1, s2, s3), aba).$
 $\forall a\ aa\ b\ ba. aba = ((a, aa, b), ba) \longrightarrow i = i1 \wedge s1 = i132 \wedge s3 = iudp \wedge$
 $s2 = ii1 \wedge a = i110 \wedge aa = i7 \wedge b = itcp \wedge ba = data\} =$
 $\{(i, (s1, s2, s3), aba).$
 $i = i1 \wedge s1 = i132 \wedge s3 = iudp \wedge s2 = ii1 \wedge (\lambda ((a,aa,b),ba). a = i110 \wedge$
 $aa = i7 \wedge b = itcp \wedge ba = data) aba\}$
apply *auto*
done

lemma *datasimp5*: $\{(i, (s1, s2, s3), aba).$
 $i = i1 \wedge s1 = i101 \wedge s3 = iudp \wedge (\lambda ((a,aa,b),ba). a = i110 \wedge aa = X606X3 \wedge$
 $b = X607X4 \wedge ba = data) aba\}$
 $= \{(i, (s1, s2, s3), (a,aa,b),ba).$
 $i = i1 \wedge s1 = i101 \wedge s3 = iudp \wedge a = i110 \wedge aa = X606X3 \wedge$
 $b = X607X4 \wedge ba = data\}$
apply *auto*
done

lemma *datasimp6*: $\{(i, (s1, s2, s3), aba).$
 $i = i1 \wedge s1 = i132 \wedge s3 = iudp \wedge s2 = i1 \wedge (\lambda ((a,aa,b),ba). a = i110 \wedge$
 $aa = i4 \wedge b = iudp \wedge ba = data) aba\} =$
 $\{(i, (s1, s2, s3), (a,aa,b),ba).$
 $i = i1 \wedge s1 = i132 \wedge s3 = iudp \wedge s2 = i1 \wedge a = i110 \wedge aa = i4 \wedge b = iudp \wedge ba$
 $= data\}$
apply *auto*
done

lemma *datasimp7*: $\{(i, (s1, s2, s3), aba).$
 $i = i1 \ \& \ i115 < s1 \ \& \ s1 < i124 \ \& \ s3 = iudp \ \& \ s2 = ii1 \ \& \ (\lambda ((a,aa,b),ba). a =$
 $i110 \ \&$
 $aa = i3 \ \& \ b = itcp \ \& \ ba = data) aba\} =$
 $\{(i, (s1, s2, s3), (a,aa,b),ba).$
 $i = i1 \ \& \ i115 < s1 \ \& \ s1 < i124 \ \& \ s3 = iudp \ \& \ s2 = ii1 \ \& \ a = i110 \ \& \ aa = i3 \ \& \ b = itcp$
 $\ \& \ ba = data\}$
apply *auto*
done

lemma *datasimp8*: $\{(i, (s1, s2, s3), aba).$
 $i = i1 \wedge s1 = i132 \wedge s3 = iudp \wedge s2 = ii1 \wedge (\lambda ((a,aa,b),ba). a = i110 \wedge aa =$
 $i7 \wedge$
 $b = itcp \wedge ba = data) aba\}$
 $= \{(i, (s1, s2, s3), (a,aa,b),ba).$
 $i = i1 \wedge s1 = i132 \wedge s3 = iudp \wedge s2 = ii1 \wedge a = i110 \wedge aa = i7 \wedge b = itcp \wedge$
 $ba = data\}$
apply *auto*
done

lemmas *datasimps* = *datasimp datasimp2 datasimp3 datasimp4*
datasimp5 datasimp6 datasimp7 datasimp8

lemmas *NATLemmas* = *src2pool-def src2poolPort-def*
src2poolPort-Protocol-def src2poolAP-def srcNat2pool-def
srcNat2pool-IntProtocolPort-def srcNat2pool-IntPort-def
srcPat2poolPort-t-def srcPat2poolPort-Protocol-t-def
srcPat2pool-IntPort-def srcPat2pool-IntProtocol-def

end

8. Policy Normalisation

```
theory
  FWNormalisationCore
imports
  ../PacketFilter/PacketFilter
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.

8.1. 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.

8.2. Auxiliary definitions and functions.

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

```

fun srcNet where
  | srcNet (DenyAllFromTo  $x$   $y$ ) =  $x$ 
  | srcNet (AllowPortFromTo  $x$   $y$   $p$ ) =  $x$ 
  | srcNet DenyAll = undefined
  | srcNet ( $v \oplus va$ ) = undefined

```

```

fun destNet where
  | destNet (DenyAllFromTo  $x$   $y$ ) =  $y$ 
  | destNet (AllowPortFromTo  $x$   $y$   $p$ ) =  $y$ 
  | destNet DenyAll = undefined
  | destNet ( $v \oplus va$ ) = undefined

```

```

fun srcnets where
  | srcnets DenyAll = []
  | srcnets (DenyAllFromTo  $x$   $y$ ) = [ $x$ ]
  | srcnets (AllowPortFromTo  $x$   $y$   $p$ ) = [ $x$ ]
  | (srcnets ( $x \oplus y$ )) = (srcnets  $x$ )@(srcnets  $y$ )

```

```

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

```

```

fun net-list where net-list  $p$  = remdups (net-list-aux  $p$ )

```

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

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

definition *packet-Nets* **where** $packet-Nets\ x\ a\ b = ((src\ x \sqsubseteq a \wedge dest\ x \sqsubseteq b) \vee (src\ x \sqsubseteq b \wedge dest\ x \sqsubseteq a))$

definition *subnetsOfAdr* **where** $subnetsOfAdr\ a = \{x. a \sqsubseteq x\}$

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

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

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

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

8.3. Invariants

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

fun *wellformed-policy1*:: $((\alpha, \beta)\ Combinators)\ list \Rightarrow bool$ **where**
 $wellformed-policy1\ [] = True$
| $wellformed-policy1\ (x\#xs) = (DenyAll \notin (set\ xs))$

There is a DenyAll at the first position

fun *wellformed-policy1-strong*:: $((\alpha, \beta)\ Combinators)\ list \Rightarrow bool$
where
 $wellformed-policy1-strong\ [] = False$
| $wellformed-policy1-strong\ (x\#xs) = (x=DenyAll \wedge (DenyAll \notin (set\ xs)))$

All two networks are either disjoint or equal.

definition *netsDistinct* **where** $netsDistinct\ a\ b = (\neg (\exists x. x \sqsubseteq a \wedge x \sqsubseteq b))$

definition *twoNetsDistinct* **where**
 $twoNetsDistinct\ a\ b\ c\ d = (netsDistinct\ a\ c \vee netsDistinct\ b\ d)$

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

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

The policy is given as a list of single rules.

```
fun singleCombinators where
  singleCombinators [] = True
| singleCombinators ((x⊕y)#xs) = False
| singleCombinators (x#xs) = singleCombinators xs
```

```
definition onlyTwoNets where
  onlyTwoNets x = ((∃ a b. (sdnets x = {(a,b)})) ∨ (∃ 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 ∧ OnlyTwoNets xs)
| OnlyTwoNets [] = True
```

```
fun noDenyAll where
  noDenyAll (x#xs) = ((¬ member DenyAll x) ∧ 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
```

8.4. Transformations

The following two functions transform a policy into a list of single rules and vice-versa - by staying on the combinator level.

```
fun policy2list::('α, 'β) Combinators ⇒
  (('α, 'β) Combinators) list where
  policy2list (x ⊕ y) = (concat [(policy2list x),(policy2list y)])
| policy2list x = [x]
```

```
fun list2FWpolicy::(('α, 'β) Combinators) list ⇒
  (('α, 'β) Combinators) where
```

```

list2FWpolicy [] = undefined
| list2FWpolicy (x#[]) = x
| list2FWpolicy (x#y) = x  $\oplus$  (list2FWpolicy y)

```

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

```

fun removeShadowRules1 where
  removeShadowRules1 (x#xs) = (if (DenyAll  $\in$  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:: (('α, 'β) Combinators) list  $\Rightarrow$ 
  (('α, 'β) 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 policy. We first need to define an ordering on rules. This ordering depends on the *Nets_List* of a policy.

```

fun smaller :: (('α, 'β) Combinators)  $\Rightarrow$ 
  (('α, 'β) Combinators)  $\Rightarrow$ 
  (('α) set) list  $\Rightarrow$  bool
where
  smaller DenyAll x l = True
| smaller x DenyAll l = False
| smaller x y l =
  ((x = y)  $\vee$ 
   (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 provide two different sorting algorithms: Quick Sort (qsort) and Insertion Sort (sort)

```
fun qsort where
  qsort [] l = []
| qsort (x#xs) l = (qsort [y←xs. ¬ (smaller x y l)] l) @ [x] @ (qsort [y←xs. smaller x y l] l)
```

```
lemma qsort-permutes:
  set (qsort xs l) = set xs
apply (induct xs l rule: qsort.induct)
apply (simp-all)
apply auto
done
```

```
lemma set-qsort [simp]: set (qsort xs l) = set xs
apply (induct xs l rule: qsort.induct)
apply (simp-all)
apply auto
done
```

```
fun insert where
  insert a [] l = [a]
| insert a (x#xs) l = (if (smaller a x l) then a#x#xs else x#(insert a xs l))
```

```
fun sort where
  sort [] l = []
| sort (x#xs) l = insert x (sort xs l) l
```

```
fun sorted where
  sorted [] l ↔ True |
  sorted [x] l ↔ True |
  sorted (x#y#zs) l ↔ smaller x y l ∧ sorted (y#zs) l
```

```
fun separate where
  separate (DenyAll#x) = DenyAll#(separate x)
| separate (x#y#z) = (if (first-bothNet x = first-bothNet y)
    then (separate ((x⊕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 ⇒ (DenyAll#(insertDenies xs))
    | - ⇒ (DenyAllFromTo (first-srcNet x) (first-destNet x) ⊕
      (DenyAllFromTo (first-destNet x) (first-srcNet x) ⊕ 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⊕xs) = (if member x xs then (removeDuplicates xs)
                             else x⊕(removeDuplicates xs))
| removeDuplicates x = x
```

```
fun removeAllDuplicates where
  removeAllDuplicates (x#xs) = ((removeDuplicates (x))#(removeAllDuplicates xs))
| removeAllDuplicates x = x
```

Insert a DenyAll at the beginning of a policy.

```
fun insertDeny where
  insertDeny (DenyAll#xs) = DenyAll#xs
| insertDeny xs = DenyAll#xs
```

```
definition sort' p l = sort l p
```

```
definition qsort' p l = qsort l p
```

```
declare dom-eq-empty-conv [simp del]
```

```
fun list2policyR::(('α, 'β) Combinators) list ⇒
  (('α, 'β) Combinators) where
  list2policyR (x#[]) = x
| list2policyR (x#y) = (list2policyR y) ⊕ x
| list2policyR [] = undefined
```

We provide the definitions for two address representations.

8.5. IntPort

```
fun C :: (adrip net, port) Combinators ⇒ (adrip, DummyContent) packet ↦ unit
where
  C DenyAll = deny-all
| C (DenyAllFromTo x y) = deny-all-from-to x y
| C (AllowPortFromTo x y p) = allow-from-to-port p x y
| C (x ⊕ y) = C x ++ C y
```

```
fun CRotate :: (adrip net, port) Combinators ⇒ (adrip, DummyContent) packet ↦ unit
where
```

```

CRotate DenyAll = C DenyAll
| CRotate (DenyAllFromTo x y) = C (DenyAllFromTo x y)
| CRotate (AllowPortFromTo x y p) = C (AllowPortFromTo x y p)
| CRotate (x  $\oplus$  y) = ((CRotate y) ++ ((CRotate x)))

```

fun rotatePolicy **where**

```

rotatePolicy DenyAll = DenyAll
| rotatePolicy (DenyAllFromTo a b) = DenyAllFromTo a b
| rotatePolicy (AllowPortFromTo a b p) = AllowPortFromTo a b p
| rotatePolicy (a  $\oplus$  b) = (rotatePolicy b)  $\oplus$  (rotatePolicy a)

```

lemma check: rev (policy2list (rotatePolicy p)) = policy2list p

apply (induct p)

apply simp

apply simp-all

done

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

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

```

wellformed-policy2 [] = True
| wellformed-policy2 (DenyAll#xs) = wellformed-policy2 xs
| wellformed-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-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::((adr_ip net,port) Combinators) list \Rightarrow bool **where**

```

wellformed-policy3 [] = True
| wellformed-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-policy3 xs)
| wellformed-policy3 (x#xs) = wellformed-policy3 xs

```

definition

```

normalize' p = (removeAllDuplicates o insertDenies o separate o
(sort' (Nets-List p)) o removeShadowRules2 o remdups o
(removeShadowRules3 C) o insertDeny o removeShadowRules1 o
policy2list) p

```

definition

```

normalizeQ' p = (removeAllDuplicates o insertDenies o separate o
(qsort' (Nets-List p)) o removeShadowRules2 o remdups o
(removeShadowRules3 C) o insertDeny o removeShadowRules1 o
policy2list) p

```

definition *normalize* ::

(*adr_{ip}* *net*, *port*) *Combinators* \Rightarrow
 (*adr_{ip}* *net*, *port*) *Combinators list*

where

normalize p = (*removeAllDuplicates* (*insertDenies* (*separate* (*sort*
 (*removeShadowRules2* (*remdups* ((*removeShadowRules3* *C*) (*insertDeny*
 (*removeShadowRules1* (*policy2list p*)))))) ((*Nets-List p*))))))

definition

normalize-manual-order p l = *removeAllDuplicates* (*insertDenies* (*separate*
 (*sort* (*removeShadowRules2* (*remdups* ((*removeShadowRules3* *C*) (*insertDeny*
 (*removeShadowRules1* (*policy2list p*)))))) ((*l*))))))

definition *normalizeQ* ::

(*adr_{ip}* *net*, *port*) *Combinators* \Rightarrow
 (*adr_{ip}* *net*, *port*) *Combinators list*

where

normalizeQ p = (*removeAllDuplicates* (*insertDenies* (*separate* (*qsort*
 (*removeShadowRules2* (*remdups* ((*removeShadowRules3* *C*) (*insertDeny*
 (*removeShadowRules1* (*policy2list p*)))))) ((*Nets-List p*))))))

definition

normalize-manual-orderQ p l = *removeAllDuplicates* (*insertDenies* (*separate*
 (*qsort* (*removeShadowRules2* (*remdups* ((*removeShadowRules3* *C*) (*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*)

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

8.6. TCP_UDP_IntegerPort

fun *Cp* :: (*adr_{ipp}* *net*, *protocol* \times *port*) *Combinators* \Rightarrow
 (*adr_{ipp}*, *DummyContent*) *packet* \mapsto *unit*

where

Cp DenyAll = *deny-all*
 | *Cp* (*DenyAllFromTo x y*) = *deny-all-from-to x y*
 | *Cp* (*AllowPortFromTo x y p*) = *allow-from-to-port-prot* (*fst p*) (*snd p*) *x y*
 | *Cp* (*x* \oplus *y*) = *Cp x* ++ *Cp y*

fun *Dp* :: (*adr_{ipp}* *net*, *protocol* \times *port*) *Combinators* \Rightarrow

$(adr_{ipp}, DummyContent) \text{ packet} \mapsto unit$

where

$Dp \text{ DenyAll} = Cp \text{ DenyAll}$
 $| Dp \text{ (DenyAllFromTo } x \ y) = Cp \text{ (DenyAllFromTo } x \ y)$
 $| Dp \text{ (AllowPortFromTo } x \ y \ p) = Cp \text{ (AllowPortFromTo } x \ y \ p)$
 $| Dp \text{ (} x \oplus y) = Cp \text{ (} y \oplus x)$

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

fun (sequential) wellformed-policy2Pr **where**

$wellformed-policy2Pr [] = True$
 $| wellformed-policy2Pr \text{ (DenyAll}\#xs) = wellformed-policy2Pr \ xs$
 $| wellformed-policy2Pr \text{ (} x\#xs) = ((\forall \ c \ a \ b. \ c = \text{DenyAllFromTo } a \ b \wedge c \in \text{set } xs \longrightarrow$
 $\quad Map.dom \ (Cp \ x) \cap Map.dom \ (Cp \ c) = \{\}) \wedge wellformed-policy2Pr \ xs)$

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-policy3Pr::((adr_{ipp} net, protocol × port) Combinators) list ⇒ bool **where**

$wellformed-policy3Pr [] = True$
 $| wellformed-policy3Pr \text{ ((AllowPortFromTo } a \ b \ p)\#xs) = ((\forall \ r. \ r \in \text{set } xs \longrightarrow$
 $\quad dom \ (Cp \ r) \cap dom \ (Cp \text{ (AllowPortFromTo } a \ b \ p)) = \{\}) \wedge wellformed-policy3Pr \ xs)$
 $| wellformed-policy3Pr \text{ (} x\#xs) = wellformed-policy3Pr \ xs$

definition

$normalizePr' :: (adr_{ipp} \text{ net}, protocol \times port) \text{ Combinators}$
 $\Rightarrow (adr_{ipp} \text{ net}, protocol \times port) \text{ Combinators list}$ **where**
 $normalizePr' \ p = (removeAllDuplicates \ o \ insertDenies \ o \ separate \ o$
 $\quad (sort' \ (Nets-List \ p)) \ o \ removeShadowRules2 \ o \ remdups \ o$
 $\quad (removeShadowRules3 \ Cp) \ o \ insertDeny \ o \ removeShadowRules1 \ o$
 $\quad policy2list) \ p$

definition normalizePr ::

$(adr_{ipp} \text{ net}, protocol \times port) \text{ Combinators}$
 $\Rightarrow (adr_{ipp} \text{ net}, protocol \times port) \text{ Combinators list}$ **where**
 $normalizePr \ p = (removeAllDuplicates \ (insertDenies \ (separate \ (sort$
 $\quad (removeShadowRules2 \ (remdups \ ((removeShadowRules3 \ Cp) \ (insertDeny$
 $\quad (removeShadowRules1 \ (policy2list \ p)))))) \ ((Nets-List \ p))))))$

definition

$normalize-manual-orderPr \ p \ l = removeAllDuplicates \ (insertDenies \ (separate$
 $\quad (sort \ (removeShadowRules2 \ (remdups \ ((removeShadowRules3 \ Cp) \ (insertDeny$
 $\quad (removeShadowRules1 \ (policy2list \ p)))))) \ ((l))))$

definition

$normalizePrQ' :: (adr_{ipp} \text{ net}, protocol \times port) \text{ Combinators}$
 $\Rightarrow (adr_{ipp} \text{ net}, protocol \times port) \text{ Combinators list where}$
 $normalizePrQ' p = (removeAllDuplicates \ o \ insertDenies \ o \ separate \ o$
 $\quad (qsort' (Nets-List \ p)) \ o \ removeShadowRules2 \ o \ remdups \ o$
 $\quad (removeShadowRules3 \ Cp) \ o \ insertDeny \ o \ removeShadowRules1 \ o$
 $\quad policy2list) \ p$

definition $normalizePrQ ::$

$(adr_{ipp} \text{ net}, protocol \times port) \text{ Combinators}$
 $\Rightarrow (adr_{ipp} \text{ net}, protocol \times port) \text{ Combinators list where}$
 $normalizePrQ \ p = (removeAllDuplicates \ (insertDenies \ (separate \ (qsort$
 $\quad (removeShadowRules2 \ (remdups \ ((removeShadowRules3 \ Cp) \ (insertDeny$
 $\quad (removeShadowRules1 \ (policy2list \ p)))))) \ ((Nets-List \ p))))$

definition

$normalize-manual-orderPrQ \ p \ l = removeAllDuplicates \ (insertDenies \ (separate$
 $\quad (qsort \ (removeShadowRules2 \ (remdups \ ((removeShadowRules3 \ Cp) \ (insertDeny$
 $\quad (removeShadowRules1 \ (policy2list \ p)))))) \ ((l))))$

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

lemma $normalizePr = normalizePr'$

by $(rule \ ext, \ simp \ add: \ normalizePr-def \ normalizePr'-def \ sort'-def)$

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

definition $makeFUTPr \text{ where}$

$makeFUTPr \ FUT \ p \ x \ n =$
 $\quad (packet-Nets \ x \ (fst \ (normBothNets \ (bothNets \ p)!n))$
 $\quad \quad (snd \ (normBothNets \ (bothNets \ p)!n)) \longrightarrow$
 $\quad FUT \ x = Cp \ ((normalizePr \ p)!Suc \ n) \ x)$

declare $Cp.simps \ [simp \ del]$

lemmas $PLemmas = C.simps \ Cp.simps \ dom-def \ PolicyCombinators.PolicyCombinators$
 $\quad PortCombinators.PortCombinatorsCore \ aux$
 $\quad ProtocolPortCombinators.ProtocolCombinatorsCore \ src-def \ dest-def \ in-subnet-def$
 $\quad adr_{ip}Lemmas \ adr_{ipp}Lemmas$

lemma $aux: \llbracket x \neq a; y \neq b; (x \neq y \wedge x \neq b) \vee (a \neq b \wedge a \neq y) \rrbracket \Longrightarrow \{x, a\} \neq \{y, b\}$
by $(auto)$


```

lemma aux2:  $\{a,b\} = \{b,a\}$ 
  by auto

```

```

end

```

9. Stateful Firewalls

9.1. Basic Constructs

```

theory Stateful
imports ../PacketFilter/PacketFilter LTL-alike
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 tuple from some memory to be refined later and the current policy.

```

type-synonym ( $'\alpha, '\beta, '\gamma$ ) FWState =  $'\alpha \times (('\beta, '\gamma) \text{ packet} \mapsto \text{unit})$ 

```

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. We provide two types of firewall monads: one

type-synonym $(\alpha, \beta, \gamma) \text{ FWStateTransitionP} = (\beta, \gamma) \text{ packet} \Rightarrow$
 $((\beta, \gamma) \text{ packet} \mapsto \text{unit}) \text{ decision}, (\alpha, \beta, \gamma) \text{ FWState}) \text{ MON}_{SE}$

type-synonym $(\alpha, \beta, \gamma) \text{ FWStateTransition} =$
 $((\beta, \gamma) \text{ packet} \times (\alpha, \beta, \gamma) \text{ FWState}) \rightarrow (\alpha, \beta, \gamma) \text{ FWState}$

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

type-synonym $(\beta, \gamma) \text{ history} = (\beta, \gamma) \text{ packet list}$

fun *packet-with-id* **where**

packet-with-id $[] \ i = []$
 $| \text{packet-with-id } (x \# xs) \ i = (\text{if } id \ x = i \text{ then } (x \# (\text{packet-with-id } xs \ i)) \text{ else } (\text{packet-with-id } xs \ i))$

fun *ids1* **where**

ids1 $i \ (x \# xs) = (id \ x = i \wedge ids1 \ i \ xs)$
 $| ids1 \ i \ [] = \text{True}$

fun *ids* **where**

ids $a \ (x \# xs) = (\text{NetworkCore.id } x \in a \wedge ids \ a \ xs)$
 $| ids \ a \ [] = \text{True}$

definition *applyPolicy*:: $(i \times (i \mapsto o)) \mapsto o$
where $\text{applyPolicy} = (\lambda (x, z). z \ x)$

end

9.2. FTP Protocol

theory *FTP*

imports

Stateful

begin

9.2.1. The protocol syntax

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. *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-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 *msg* = *ftp-init* | *ftp-port-request port* | *ftp-data* | *ftp-close* | *ftp-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* \Rightarrow (*adr_{ip}*, *msg*) *packet* \Rightarrow *bool* **where**
is-init = (λ *i p*. (*id p* = *i* \wedge *content p* = *ftp-init*))

definition

is-ftp-port-request :: *id* \Rightarrow *port* \Rightarrow (*adr_{ip}*, *msg*) *packet* \Rightarrow *bool* **where**
is-ftp-port-request = (λ *i port p*. (*id p* = *i* \wedge *content p* = *ftp-port-request port*))

definition

is-ftp-data :: *id* \Rightarrow (*adr_{ip}*, *msg*) *packet* \Rightarrow *bool* **where**
is-ftp-data = (λ *i p*. (*id p* = *i* \wedge *content p* = *ftp-data*))

definition

is-ftp-close :: *id* \Rightarrow (*adr_{ip}*, *msg*) *packet* \Rightarrow *bool* **where**
is-ftp-close = (λ *i p*. (*id p* = *i* \wedge *content p* = *ftp-close*))

definition

port-open :: (*adr_{ip}*, *msg*) *history* \Rightarrow *id* \Rightarrow *port* \Rightarrow *bool* **where**
port-open = (λ *L a p*. (*not-before* (*is-ftp-close a*) (*is-ftp-port-request a p*) *L*))

definition

is-ftp-other :: *id* \Rightarrow (*adr_{ip}*, *msg*) *packet* \Rightarrow *bool* **where**
is-ftp-other = (λ *i p*. (*id p* = *i* \wedge *content p* = *ftp-other*))

fun *are-ftp-other* **where**

are-ftp-other i (x#xs) = (*is-ftp-other i x* \wedge *are-ftp-other i xs*)
are-ftp-other i [] = *True*

9.2.2. The protocol policy specification

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 *last-opened-port* **where**

last-opened-port *i* ((*j,s,d,ftp-port-request* *p*)#*xs*) = (if *i=j* then *p* else *last-opened-port* *i* *xs*)
| *last-opened-port* *i* (*x*#*xs*) = *last-opened-port* *i* *xs*
| *last-opened-port* *x* [] = *undefined*

fun *FTP-STA* :: ((*adr_{ip}*,*msg*) *history*, *adr_{ip}*, *msg*) *FWStateTransition*

where

FTP-STA ((*i,s,d,ftp-port-request* *pr*), (*log* , *pol*)) =
(if before(*Not* *o* *is-ftp-close* *i*)(*is-init* *i*) *log* ∧
dest-port (*i,s,d,ftp-port-request* *pr*) = (21::*port*)
then Some (((*i,s,d,ftp-port-request* *pr*)#*log* ,
(*allow-from-to-port* *pr* (*subnet-of* *d*) (*subnet-of* *s*)) ⊕ *pol*))
else Some (((*i,s,d,ftp-port-request* *pr*)#*log* ,*pol*)))

| *FTP-STA* ((*i,s,d,ftp-close*), (*log* ,*pol*)) =
(if (∃ *p*. *port-open* *log* *i* *p*) ∧ *dest-port* (*i,s,d,ftp-close*) = (21::*port*)
then Some ((*i,s,d,ftp-close*)#*log* ,
deny-from-to-port (*last-opened-port* *i* *log*) (*subnet-of* *d*)(*subnet-of* *s*) ⊕ *pol*)
else Some (((*i,s,d,ftp-close*)#*log* , *pol*)))

| *FTP-STA* (*p*, *s*) = Some (*p*#(*fst* *s*),*snd* *s*)

fun *FTP-STD* ::

((*adr_{ip}*,*msg*) *history*, *adr_{ip}*, *msg*) *FWStateTransition*

where

FTP-STD (*p*,*s*) = Some *s*

definition *TRPolicy* :: (*adr_{ip}*,*msg*)*packet* × (*adr_{ip}*,*msg*)*history* × ((*adr_{ip}*,*msg*)*packet* ↦ *unit*)

where $TRPolicy = ((FTP-STA, FTP-STD) \otimes_{\nabla} applyPolicy) \circ (\lambda(x, (y, z)). ((x, z), (x, (y, z))))$

definition $TRPolicy_{Mon}$

where $TRPolicy_{Mon} = policy2MON(TRPolicy)$

If required to contain the policy in the output

definition $TRPolicy_{Mon}'$

where $TRPolicy_{Mon}' = policy2MON(((\lambda(x, y, z). (z, (y, z))) \circ f TRPolicy))$

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 $ftp-states = S0 \mid S1 \mid S2 \mid S3$

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.

fun

$is-ftp :: ftp-states \Rightarrow adr_{ip} \Rightarrow adr_{ip} \Rightarrow id \Rightarrow port \Rightarrow$
 $(adr_{ip}, msg) history \Rightarrow bool$

where

$is-ftp H c s i p [] = (H = S3)$

$is-ftp H c s i p (x \# InL) = (snd s = 21 \wedge ((\lambda(id, sr, de, co). (((id = i \wedge ($
 $(H = ftp-states.S2 \wedge sr = c \wedge de = s \wedge co = ftp-init \wedge is-ftp S3 c s i p InL) \vee$
 $(H = ftp-states.S1 \wedge sr = c \wedge de = s \wedge co = ftp-port-request p \wedge is-ftp S2 c s i p InL) \vee$
 $(H = ftp-states.S1 \wedge sr = s \wedge de = (fst c, p) \wedge co = ftp-data \wedge is-ftp S1 c s i p InL) \vee$
 $(H = ftp-states.S0 \wedge sr = c \wedge de = s \wedge co = ftp-close \wedge is-ftp S1 c s i p InL)))))) x))$

definition $is-single-ftp-run :: adr_{ip} src \Rightarrow adr_{ip} dest \Rightarrow id \Rightarrow port \Rightarrow (adr_{ip}, msg) history set$

where $is-single-ftp-run s d i p = \{x. (is-ftp S0 s d i p x)\}$

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.

The following definition returns the set of all possible interleaving of two correct FTP protocol runs.

definition

```

ftp-2-interleaved :: adrip src ⇒ adrip dest ⇒ id ⇒ port ⇒
    adrip src ⇒ adrip dest ⇒ id ⇒ port ⇒
    (adrip,msg) history set where
ftp-2-interleaved s1 d1 i1 p1 s2 d2 i2 p2 =
    {x. (is-ftp S0 s1 d1 i1 p1 (packet-with-id x i1)) ∧
        (is-ftp S0 s2 d2 i2 p2 (packet-with-id x i2))}

```

```

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: ∃ x∈subnet-of (a::int, b::port). (a, c::port) ∈ x
apply (rule bexI)
apply (simp-all add: IntegerPort.subnet-of-def)
done

```

```

lemma port-open-lemma: ¬ (Ex (port-open [] (x::port)))
apply (simp add: port-open-def)
done

```

```

lemmas FTPLemmas = TRPolicy-def applyPolicy-def policy2MON-def
Let-def in-subnet-def src-def
dest-def subnet-of-def
is-init-def p-accept-def port-open-def is-ftp-data-def is-ftp-close-def
is-ftp-port-request-def
content-def PortCombinators
exI subnetOf-lemma
subnetOf-lemma2 subnetOf-lemma3 subnetOf-lemma4
NetworkCore.id-def adripLemmas port-open-lemma
bind-SE-def unit-SE-def valid-SE-def

```

end

theory *FTP-WithPolicy*
imports
FTP
begin

FTP where the policy is part of the output.

definition *POL* :: 'a \Rightarrow 'a
where *POL* *x* = *x*

Variant 2 takes the policy into the output

fun *FTP-STP* ::
 ((*id* \rightarrow *port*), *adr_{ip}*, *msg*) *FWStateTransitionP*
where

FTP-STP (*i,s,d,ftp-port-request pr*) (*ports, policy*) =
 (if *p-accept* (*i,s,d,ftp-port-request pr*) *policy* then
 Some (*allow* (*POL* ((*allow-from-to-port pr* (*subnet-of d*) (*subnet-of s*)) \oplus *policy*)),
 (*(ports(i \rightarrow pr)),(allow-from-to-port pr* (*subnet-of d*) (*subnet-of s*))
 \oplus *policy*))
 else (*Some* (*deny* (*POL policy*),(*ports,policy*))))

|*FTP-STP* (*i,s,d,ftp-close*) (*ports,policy*) =
 (if (*p-accept* (*i,s,d,ftp-close*) *policy*) then
 case *ports i* of
 Some pr \Rightarrow
 Some(*allow* (*POL* (*deny-from-to-port pr* (*subnet-of d*) (*subnet-of s*) \oplus *policy*)),
 ports(i:=None),
 deny-from-to-port pr (*subnet-of d*) (*subnet-of s*) \oplus *policy*)
 |*None* \Rightarrow *Some*(*allow* (*POL policy*), *ports, policy*)
 else *Some* (*deny* (*POL policy*), *ports, policy*)

|*FTP-STP* *p x* = (if *p-accept p* (*snd x*)
 then *Some* (*allow* (*POL* (*snd x*)),(*(fst x),snd x*))
 else *Some* (*deny* (*POL* (*snd x*)),(*(fst x,snd x)*))

end

9.3. VoIP Protocol

```
theory VOIP
imports Stateful
begin
```

After the FTP-Protocol which was rather simple we show the strength of the model with a more current and especially much more complicated example, namely Voice over IP (VoIP). VoIP is standardized by the ITU-T under the name H.323, which can be seen as an "umbrella standard" which aggregates standards for multimedia conferencing over packet-based networks (for a good overview of the protocol suite, see [?]). H.323 poses many problems to firewalls. These problems include (taken from [?]):

- An H.323 call is made up of many different simultaneous connections.
- Most connections are made to dynamic ports.
- The addresses and port numbers are exchanged within the data stream of the next higher connection.
- Calls can be initiated from outside the firewall.

Again we only consider a simplified VoIP scenario with the following seven messages which are grouped into four subprotocols (see Figure ??):

- Registration and Admission (H.225, port 1719): The caller contacts its gatekeeper with a call request. The gatekeeper either rejects or confirms the request, returning the address of the callee in the latter case.
 - Admission Request (ARQ)
 - Admission Reject (ARJ)
 - Admission Confirm (ACF) 'a
- Call Signaling (Q.931, port 1720) The caller and the callee agree on the dynamic ports over which the call will take place.
 - Setup *port*
 - Connect *port*
- Stream (dynamic ports). The call itself. In reality, several connections are used here.
- Fin (port 1720).

The two main differences to FTP are:

- In VoIP, we deal with three different entities: the caller, the callee, and the gatekeeper.
- We do not know in advance which entity will close the connection.

We model the protocol as seen from a firewall at the caller, namely we are not interested

in the messages from the callee to its gatekeeper. Incoming calls are not modelled either, they would require a different set of state transitions.

The content of a packet now consists of one of the seven messages or a default one. It is parameterized with the type of the address that the gatekeeper returns.

```
datatype 'a voip-msg = ARQ
    | ACF 'a
    | ARJ
    | Setup port
    | Connect port
    | Stream
    | Fin
    | other
```

As before, we need operators which check if a packet contains a specific content and ID, respectively if such a packet has appeared in the trace.

definition

```
is-arq :: NetworkCore.id ⇒ ('a::adr, 'b voip-msg) packet ⇒ bool where
is-arq i p = (NetworkCore.id p = i ∧ content p = ARQ)
```

definition

```
is-fin :: id ⇒ ('a::adr, 'b voip-msg) packet ⇒ bool where
is-fin i p = (id p = i ∧ content p = Fin)
```

definition

```
is-connect :: id ⇒ port ⇒ ('a::adr, 'b voip-msg) packet ⇒ bool where
is-connect i port p = (id p = i ∧ content p = Connect port)
```

definition

```
is-setup :: id ⇒ port ⇒ ('a::adr, 'b voip-msg) packet ⇒ bool where
is-setup i port p = (id p = i ∧ content p = Setup port)
```

We need also an operator *ports-open* to get access to the two dynamic ports.

definition

```
ports-open :: id ⇒ port × port ⇒ (adrip, 'a voip-msg) history ⇒ bool where
ports-open i p L = ((not-before (is-fin i) (is-setup i (fst p)) L) ∧
    not-before (is-fin i) (is-connect i (snd p)) L)
```

As we do not know which entity closes the connection, we define an operator which checks if the closer is the caller.

fun

```
src-is-initiator :: id ⇒ adrip ⇒ (adrip, 'b voip-msg) history ⇒ bool where
src-is-initiator i a [] = False
|src-is-initiator i a (p#S) = (((id p = i) ∧
    (∃ port. content p = Setup port) ∧
    ((fst (src p) = fst a))) ∨
    (src-is-initiator i a S))
```

The first state transition is for those messages which do not change the policy. In this scenario, this only happens for the Stream messages.

definition *subnet-of-adr* **where**

$$\text{subnet-of-adr } x = \{(a,b). a = x\}$$

fun *VOIP-STA* ::

((*adr_{ip}*, *address voip-msg*) *history*, *adr_{ip}*, *address voip-msg*) *FWStateTransition*

where

$$\begin{aligned} \text{VOIP-STA } ((a,c,d,ARQ), (InL, policy)) = \\ \text{Some } (((a,c,d, ARQ)\#InL, \\ (\text{allow-from-to-port } (1719::port)(\text{subnet-of } d) (\text{subnet-of } c)) \oplus policy)) \end{aligned}$$

$$\begin{aligned} | \text{VOIP-STA } ((a,c,d,ARJ), (InL, policy)) = \\ (\text{if } (\text{not-before } (\text{is-fin } a) (\text{is-arg } a) InL) \\ \text{then Some } (((a,c,d,ARJ)\#InL, \\ \text{deny-from-to-port } (14::port) (\text{subnet-of } c) (\text{subnet-of } d) \oplus policy)) \\ \text{else Some } (((a,c,d,ARJ)\#InL,policy))) \end{aligned}$$

$$\begin{aligned} | \text{VOIP-STA } ((a,c,d,ACF \text{ callee}), (InL, policy)) = \\ \text{Some } (((a,c,d,ACF \text{ callee})\#InL, \\ \text{allow-from-to-port } (1720::port) (\text{subnet-of-adr } \text{callee}) (\text{subnet-of } d) \oplus \\ \text{allow-from-to-port } (1720::port) (\text{subnet-of } d) (\text{subnet-of-adr } \text{callee}) \oplus \\ \text{deny-from-to-port } (1719::port) (\text{subnet-of } d) (\text{subnet-of } c) \oplus \\ policy)) \end{aligned}$$

$$\begin{aligned} | \text{VOIP-STA } ((a,c,d, Setup \text{ port}), (InL, policy)) = \\ \text{Some } (((a,c,d,Setup \text{ port})\#InL, \\ \text{allow-from-to-port } port (\text{subnet-of } d) (\text{subnet-of } c) \oplus policy)) \end{aligned}$$

$$\begin{aligned} | \text{VOIP-STA } ((a,c,d, Connect \text{ port}), (InL, policy)) = \\ \text{Some } (((a,c,d,Connect \text{ port})\#InL, \\ \text{allow-from-to-port } port (\text{subnet-of } d) (\text{subnet-of } c) \oplus policy)) \end{aligned}$$

$$\begin{aligned} | \text{VOIP-STA } ((a,c,d,Fin), (InL,policy)) = \\ (\text{if } \exists p1 p2. \text{ports-open } a (p1,p2) InL \text{ then } (\\ (\text{if } \text{src-is-initiator } a c InL \\ \text{then } (\text{Some } (((a,c,d,Fin)\#InL, \\ (\text{deny-from-to-port } (1720::int) (\text{subnet-of } c) (\text{subnet-of } d)) \oplus \\ (\text{deny-from-to-port } (snd (SOME p. \text{ports-open } a p InL)) \\ (\text{subnet-of } c) (\text{subnet-of } d)) \oplus \\ (\text{deny-from-to-port } (fst (SOME p. \text{ports-open } a p InL)) \\ (\text{subnet-of } d) (\text{subnet-of } c)) \oplus policy)))) \end{aligned}$$

```

      else (Some (((a,c,d,Fin)#InL,
(deny-from-to-port (1720::int) (subnet-of c) (subnet-of d) ) ⊕
(deny-from-to-port (fst (SOME p. ports-open a p InL))
(subnet-of c) (subnet-of d)) ⊕
(deny-from-to-port (snd (SOME p. ports-open a p InL))
(subnet-of d) (subnet-of c)) ⊕ policy))))))

    else
      (Some (((a,c,d,Fin)#InL,policy))))

```

```

| VOIP-STA (p, (InL, policy)) =
  Some ((p#InL,policy))

```

```

fun VOIP-STD where
  VOIP-STD (p,s) = Some s

```

```

definition VOIP-TRPolicy where
  VOIP-TRPolicy = policy2MON (
    ((VOIP-STA,VOIP-STD) ⊗∇ applyPolicy) o (λ (x,(y,z)). ((x,z),(x,(y,z)))))

```

For a full protocol run, six states are needed.

```

datatype voip-states = S0 | S1 | S2 | S3 | S4 | S5

```

The constant *is-voip* checks if a trace corresponds to a legal VoIP protocol, given the IP-addresses of the three entities, the ID, and the two dynamic ports.

```

fun is-voip :: voip-states ⇒ address ⇒ address ⇒ address ⇒ id ⇒ port ⇒
  port ⇒ (adrip, address voip-msg) history ⇒ bool

```

where

```

  is-voip H s d g i p1 p2 [] = (H = S5)
  is-voip H s d g i p1 p2 (x#InL) =
    (((λ (id,sr,de,co).
      (((id = i ∧
        (H = S4 ∧ ((sr = (s,1719) ∧ de = (g,1719) ∧ co = ARQ ∧
          is-voip S5 s d g i p1 p2 InL)))) ∨
        (H = S0 ∧ sr = (g,1719) ∧ de = (s,1719) ∧ co = ARJ ∧
          is-voip S4 s d g i p1 p2 InL) ∨
        (H = S3 ∧ sr = (g,1719) ∧ de = (s,1719) ∧ co = ACF d ∧
          is-voip S4 s d g i p1 p2 InL) ∨
        (H = S2 ∧ sr = (s,1720) ∧ de = (d,1720) ∧ co = Setup p1 ∧
          is-voip S3 s d g i p1 p2 InL) ∨
        (H = S1 ∧ sr = (d,1720) ∧ de = (s,1720) ∧ co = Connect p2 ∧
          is-voip S2 s d g i p1 p2 InL) ∨
        (H = S1 ∧ sr = (s,p1) ∧ de = (d,p2) ∧ co = Stream ∧
          is-voip S1 s d g i p1 p2 InL) ∨
        (H = S1 ∧ sr = (d,p2) ∧ de = (s,p1) ∧ co = Stream ∧

```

$$\begin{aligned}
& is-voip\ S1\ s\ d\ g\ i\ p1\ p2\ InL) \vee \\
& (H = S0 \wedge sr = (d,1720) \wedge de = (s,1720) \wedge co = Fin \wedge \\
& \quad is-voip\ S1\ s\ d\ g\ i\ p1\ p2\ InL) \vee \\
& (H = S0 \wedge sr = (s,1720) \wedge de = (d,1720) \wedge co = Fin \wedge \\
& \quad is-voip\ S1\ s\ d\ g\ i\ p1\ p2\ InL))))))\ x)
\end{aligned}$$

Finally, *NB-voip* returns the set of protocol traces which correspond to a correct protocol run given the three addresses, the ID, and the two dynamic ports.

definition

$$\begin{aligned}
NB-voip &:: address \Rightarrow address \Rightarrow address \Rightarrow id \Rightarrow port \Rightarrow port \Rightarrow \\
&\quad (adr_{ip}, address\ voip\text{-}msg)\ history\ set\ \mathbf{where} \\
NB-voip\ s\ d\ g\ i\ p1\ p2 &= \{x. (is-voip\ S0\ s\ d\ g\ i\ p1\ p2\ x)\}
\end{aligned}$$

end

9.4. FTP and VoIP Protocol

theory *FTPVOIP*

imports

FTP-WithPolicy VOIP

begin

datatype *ftpvoip* = *ARQ*

<i>ACF</i>	<i>int</i>
<i>ARJ</i>	
<i>Setup</i>	<i>port</i>
<i>Connect</i>	<i>port</i>
<i>Stream</i>	
<i>Fin</i>	
<i>ftp-init</i>	
<i>ftp-port-request</i>	<i>port</i>
<i>ftp-data</i>	
<i>ftp-close</i>	
<i>other</i>	

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

$$\begin{aligned}
FTPVOIP\text{-}is\text{-}init &:: id \Rightarrow (adr_{ip},\ ftpvoip)\ packet \Rightarrow bool\ \mathbf{where} \\
FTPVOIP\text{-}is\text{-}init &= (\lambda\ i\ p. (id\ p = i \wedge content\ p = ftp\text{-}init))
\end{aligned}$$

definition

$$FTPVOIP\text{-}is\text{-}port\text{-}request :: id \Rightarrow port \Rightarrow (adr_{ip},\ ftpvoip)\ packet \Rightarrow bool\ \mathbf{where}$$

$FTPVOIP\text{-}is\text{-}port\text{-}request = (\lambda i \text{ port } p. (id \ p = i \wedge content \ p = ftp\text{-}port\text{-}request \ port))$

definition

$FTPVOIP\text{-}is\text{-}data :: id \Rightarrow (adr_{ip}, ftpvoip) \text{ packet} \Rightarrow bool$ **where**
 $FTPVOIP\text{-}is\text{-}data = (\lambda i \ p. (id \ p = i \wedge content \ p = ftp\text{-}data))$

definition

$FTPVOIP\text{-}is\text{-}close :: id \Rightarrow (adr_{ip}, ftpvoip) \text{ packet} \Rightarrow bool$ **where**
 $FTPVOIP\text{-}is\text{-}close = (\lambda i \ p. (id \ p = i \wedge content \ p = ftp\text{-}close))$

definition

$FTPVOIP\text{-}port\text{-}open :: (adr_{ip}, ftpvoip) \text{ history} \Rightarrow id \Rightarrow port \Rightarrow bool$ **where**
 $FTPVOIP\text{-}port\text{-}open = (\lambda L \ a \ p. (not\text{-}before \ (FTPVOIP\text{-}is\text{-}close \ a) \ (FTPVOIP\text{-}is\text{-}port\text{-}request \ a \ p) \ L))$

definition

$FTPVOIP\text{-}is\text{-}other :: id \Rightarrow (adr_{ip}, ftpvoip) \text{ packet} \Rightarrow bool$ **where**
 $FTPVOIP\text{-}is\text{-}other = (\lambda i \ p. (id \ p = i \wedge content \ p = other))$

fun $FTPVOIP\text{-}are\text{-}other$ **where**

$FTPVOIP\text{-}are\text{-}other \ i \ (x\#xs) = (FTPVOIP\text{-}is\text{-}other \ i \ x \wedge FTPVOIP\text{-}are\text{-}other \ i \ xs)$
 $|FTPVOIP\text{-}are\text{-}other \ i \ [] = True$

fun $last\text{-}opened\text{-}port$ **where**

$last\text{-}opened\text{-}port \ i \ ((j,s,d,ftp\text{-}port\text{-}request \ p)\#xs) = (if \ i=j \ then \ p \ else \ last\text{-}opened\text{-}port \ i \ xs)$
 $| \ last\text{-}opened\text{-}port \ i \ (x\#xs) = last\text{-}opened\text{-}port \ i \ xs$
 $| \ last\text{-}opened\text{-}port \ x \ [] = undefined$

fun $FTPVOIP\text{-}FTP\text{-}STA ::$

$((adr_{ip}, ftpvoip) \text{ history}, adr_{ip}, ftpvoip) \text{ FWStateTransition}$

where

$FTPVOIP\text{-}FTP\text{-}STA \ ((i,s,d,ftp\text{-}port\text{-}request \ pr), (InL, policy)) =$
 $(if \ not\text{-}before \ (FTPVOIP\text{-}is\text{-}close \ i) \ (FTPVOIP\text{-}is\text{-}init \ i) \ InL \wedge$
 $dest\text{-}port \ (i,s,d,ftp\text{-}port\text{-}request \ pr) = (21::port) \ then$
 $Some \ (((i,s,d,ftp\text{-}port\text{-}request \ pr)\#InL, policy) ++$
 $(allow\text{-}from\text{-}to\text{-}port \ pr \ (subnet\text{-}of \ d) \ (subnet\text{-}of \ s))))$
 $else \ Some \ (((i,s,d,ftp\text{-}port\text{-}request \ pr)\#InL,policy)))$

$|FTPVOIP\text{-}FTP\text{-}STA \ ((i,s,d,ftp\text{-}close), (InL,policy)) =$
 $(if \ (\exists \ p. \ FTPVOIP\text{-}port\text{-}open \ InL \ i \ p) \wedge \ dest\text{-}port \ (i,s,d,ftp\text{-}close) = (21::port)$
 $then \ Some \ ((i,s,d,ftp\text{-}close)\#InL, policy) ++$
 $deny\text{-}from\text{-}to\text{-}port \ (last\text{-}opened\text{-}port \ i \ InL) \ (subnet\text{-}of \ d) \ (subnet\text{-}of \ s))$
 $else \ Some \ (((i,s,d,ftp\text{-}close)\#InL, policy)))$

$|FTPVOIP-FTP-STA (p, s) = Some (p\#(fst s), snd s)$

fun *FTPVOIP-FTP-STD* ::
 ((*adr_{ip}*, *ftpvoip*) *history*, *adr_{ip}*, *ftpvoip*) *FWStateTransition*
where
FTPVOIP-FTP-STD (*p*, *s*) = *Some s*

definition

FTPVOIP-is-arq :: *NetworkCore.id* \Rightarrow (*'a::adr*, *ftpvoip*) *packet* \Rightarrow *bool* **where**
FTPVOIP-is-arq *i p* = (*NetworkCore.id p* = *i* \wedge *content p* = *ARQ*)

definition

FTPVOIP-is-fin :: *id* \Rightarrow (*'a::adr*, *ftpvoip*) *packet* \Rightarrow *bool* **where**
FTPVOIP-is-fin *i p* = (*id p* = *i* \wedge *content p* = *Fin*)

definition

FTPVOIP-is-connect :: *id* \Rightarrow *port* \Rightarrow (*'a::adr*, *ftpvoip*) *packet* \Rightarrow *bool* **where**
FTPVOIP-is-connect *i port p* = (*id p* = *i* \wedge *content p* = *Connect port*)

definition

FTPVOIP-is-setup :: *id* \Rightarrow *port* \Rightarrow (*'a::adr*, *ftpvoip*) *packet* \Rightarrow *bool* **where**
FTPVOIP-is-setup *i port p* = (*id p* = *i* \wedge *content p* = *Setup port*)

We need also an operator *ports-open* to get access to the two dynamic ports.

definition

FTPVOIP-ports-open :: *id* \Rightarrow *port* \times *port* \Rightarrow (*adr_{ip}*, *ftpvoip*) *history* \Rightarrow *bool* **where**
FTPVOIP-ports-open *i p L* = ((*not-before* (*FTPVOIP-is-fin* *i*) (*FTPVOIP-is-setup* *i* (*fst p*))

L) \wedge
not-before (*FTPVOIP-is-fin* *i*) (*FTPVOIP-is-connect* *i* (*snd p*)) *L*)

As we do not know which entity closes the connection, we define an operator which checks if the closer is the caller.

fun

FTPVOIP-src-is-initiator :: *id* \Rightarrow *adr_{ip}* \Rightarrow (*adr_{ip}*, *ftpvoip*) *history* \Rightarrow *bool* **where**
FTPVOIP-src-is-initiator *i a* [] = *False*
 $|FTPVOIP-src-is-initiator$ *i a* (*p#S*) = (((*id p* = *i*) \wedge
 (\exists *port*. *content p* = *Setup port*) \wedge
 ((*fst* (*src p*) = *fst a*))) \vee
 (*FTPVOIP-src-is-initiator* *i a S*)

definition *FTPVOIP-subnet-of-adr* :: *int* \Rightarrow *adr_{ip}* *net* **where**

FTPVOIP-subnet-of-adr *x* = { {(*a*, *b*). *a* = *x*}}

fun *FTPVOIP-VOIP-STA* ::

((*adr_{ip}*, *ftpvoip*) *history*, *adr_{ip}*, *ftpvoip*) *FWStateTransition*
where

FTPVOIP-VOIP-STA ((*a,c,d,ARQ*), (*InL*, *policy*)) =
Some (((*a,c,d, ARQ*)#*InL*,
 (allow-from-to-port (1719::port)(subnet-of *d*) (subnet-of *c*)) \oplus *policy*))

|*FTPVOIP-VOIP-STA* ((*a,c,d,ARJ*), (*InL*, *policy*)) =
 (if (not-before (*FTPVOIP-is-fin* *a*) (*FTPVOIP-is-arq* *a*) *InL*)
 then *Some* (((*a,c,d,ARJ*)#*InL*,
 deny-from-to-port (14::port) (subnet-of *c*) (subnet-of *d*) \oplus *policy*))
 else *Some* (((*a,c,d,ARJ*)#*InL,policy*)))

|*FTPVOIP-VOIP-STA* ((*a,c,d,ACF callee*), (*InL*, *policy*)) =
Some (((*a,c,d,ACF callee*)#*InL*,
 allow-from-to-port (1720::port) (subnet-of-adr *callee*) (subnet-of *d*) \oplus
 allow-from-to-port (1720::port) (subnet-of *d*) (subnet-of-adr *callee*) \oplus
 deny-from-to-port (1719::port) (subnet-of *d*) (subnet-of *c*) \oplus
policy))

|*FTPVOIP-VOIP-STA* ((*a,c,d, Setup port*), (*InL*, *policy*)) =
Some (((*a,c,d,Setup port*)#*InL*,
 allow-from-to-port *port* (subnet-of *d*) (subnet-of *c*) \oplus *policy*))

|*FTPVOIP-VOIP-STA* ((*a,c,d, ftpvoip.Connect port*), (*InL*, *policy*)) =
Some (((*a,c,d,ftpvoip.Connect port*)#*InL*,
 allow-from-to-port *port* (subnet-of *d*) (subnet-of *c*) \oplus *policy*))

|*FTPVOIP-VOIP-STA* ((*a,c,d,Fin*), (*InL,policy*)) =
 (if \exists *p1 p2*. *FTPVOIP-ports-open* *a* (*p1,p2*) *InL* then (
 (if *FTPVOIP-src-is-initiator* *a c InL*
 then (*Some* (((*a,c,d,Fin*)#*InL*,
 (deny-from-to-port (1720::int) (subnet-of *c*) (subnet-of *d*)) \oplus
 (deny-from-to-port (*snd* (*SOME p. FTPVOIP-ports-open* *a p InL*))
 (subnet-of *c*) (subnet-of *d*)) \oplus
 (deny-from-to-port (*fst* (*SOME p. FTPVOIP-ports-open* *a p InL*))
 (subnet-of *d*) (subnet-of *c*)) \oplus *policy*)))

 else (*Some* (((*a,c,d,Fin*)#*InL*,
 (deny-from-to-port (1720::int) (subnet-of *c*) (subnet-of *d*)) \oplus
 (deny-from-to-port (*fst* (*SOME p. FTPVOIP-ports-open* *a p InL*))
 (subnet-of *c*) (subnet-of *d*)) \oplus
 (deny-from-to-port (*snd* (*SOME p. FTPVOIP-ports-open* *a p InL*))
 (subnet-of *d*) (subnet-of *c*)) \oplus *policy*))))

else
 (*Some* (((*a,c,d,Fin*)#*InL,policy*))))

| *FTPVOIP-VOIP-STA* (*p*, (*InL*, *policy*)) =
 Some ((*p*#*InL,policy*))

fun *FTPVOIP-VOIP-STD* ::
 ((*adr_{ip}*, *ftpvoip*) *history*, *adr_{ip}*, *ftpvoip*) *FWStateTransition*
where
FTPVOIP-VOIP-STD (*p,s*) = *Some s*

definition *FTP-VOIP-STA* :: ((*adr_{ip}*, *ftpvoip*) *history*, *adr_{ip}*, *ftpvoip*) *FWStateTransition*
where
FTP-VOIP-STA = ($\lambda(x,x).$ *Some x*) *o-m* ((*FTPVOIP-FTP-STA* \otimes_S *FTPVOIP-VOIP-STA*
o ($\lambda (p,x).$ (*p,x,x*))))

definition *FTP-VOIP-STD* :: ((*adr_{ip}*, *ftpvoip*) *history*, *adr_{ip}*, *ftpvoip*) *FWStateTransition*
where
FTP-VOIP-STD = ($\lambda(x,x).$ *Some x*) *o-m* ((*FTPVOIP-FTP-STD* \otimes_S *FTPVOIP-VOIP-STD*
o ($\lambda (p,x).$ (*p,x,x*))))

definition *FTPVOIP-TRPolicy* **where**
FTPVOIP-TRPolicy = *policy2MON* (
 (((*FTP-VOIP-STA,FTP-VOIP-STD*) \otimes_{∇} *applyPolicy*) *o* ($\lambda (x,(y,z)). ((x,z),(x,(y,z))))$)))

lemmas *FTPVOIP-ST-simps* = *Let-def in-subnet-def src-def dest-def*
subnet-of-def id-def FTPVOIP-port-open-def
FTPVOIP-is-init-def FTPVOIP-is-data-def FTPVOIP-is-port-request-def FTPVOIP-is-close-def
p-accept-def content-def PortCombinators exI
NetworkCore.id-def adr_{ip}Lemmas

datatype *ftp-states2* = *FS0* | *FS1* | *FS2* | *FS3*

datatype *voip-states2* = *V0* | *V1* | *V2* | *V3* | *V4* | *V5*

The constant *is-voip* checks if a trace corresponds to a legal VoIP protocol, given the IP-addresses of the three entities, the ID, and the two dynamic ports.

fun *FTPVOIP-is-voip* :: *voip-states2* \Rightarrow *address* \Rightarrow *address* \Rightarrow *address* \Rightarrow *id* \Rightarrow *port* \Rightarrow *port* \Rightarrow (*adr_{ip}*, *ftpvoip*) *history* \Rightarrow *bool*

where

FTPVOIP-is-voip *H s d g i p1 p2 []* = (*H* = *V5*)
FTPVOIP-is-voip *H s d g i p1 p2 (x#InL)* =
 (((λ (*id*,*sr*,*de*,*co*).
 (((*id* = *i* \wedge
 (*H* = *V4* \wedge ((*sr* = (*s*,1719) \wedge *de* = (*g*,1719) \wedge *co* = *ARQ* \wedge
FTPVOIP-is-voip *V5 s d g i p1 p2 InL*))) \vee
 (*H* = *V0* \wedge *sr* = (*g*,1719) \wedge *de* = (*s*,1719) \wedge *co* = *ARJ* \wedge
FTPVOIP-is-voip *V4 s d g i p1 p2 InL*) \vee
 (*H* = *V3* \wedge *sr* = (*g*,1719) \wedge *de* = (*s*,1719) \wedge *co* = *ACF d* \wedge
FTPVOIP-is-voip *V4 s d g i p1 p2 InL*) \vee
 (*H* = *V2* \wedge *sr* = (*s*,1720) \wedge *de* = (*d*,1720) \wedge *co* = *Setup p1* \wedge
FTPVOIP-is-voip *V3 s d g i p1 p2 InL*) \vee
 (*H* = *V1* \wedge *sr* = (*d*,1720) \wedge *de* = (*s*,1720) \wedge *co* = *Connect p2* \wedge
FTPVOIP-is-voip *V2 s d g i p1 p2 InL*) \vee
 (*H* = *V1* \wedge *sr* = (*s*,*p1*) \wedge *de* = (*d*,*p2*) \wedge *co* = *Stream* \wedge
FTPVOIP-is-voip *V1 s d g i p1 p2 InL*) \vee
 (*H* = *V1* \wedge *sr* = (*d*,*p2*) \wedge *de* = (*s*,*p1*) \wedge *co* = *Stream* \wedge
FTPVOIP-is-voip *V1 s d g i p1 p2 InL*) \vee
 (*H* = *V0* \wedge *sr* = (*d*,1720) \wedge *de* = (*s*,1720) \wedge *co* = *Fin* \wedge
FTPVOIP-is-voip *V1 s d g i p1 p2 InL*) \vee
 (*H* = *V0* \wedge *sr* = (*s*,1720) \wedge *de* = (*d*,1720) \wedge *co* = *Fin* \wedge
FTPVOIP-is-voip *V1 s d g i p1 p2 InL*)))))) *x*)

Finally, *NB-voip* returns the set of protocol traces which correspond to a correct protocol run given the three addresses, the ID, and the two dynamic ports.

definition

FTPVOIP-NB-voip :: *address* \Rightarrow *address* \Rightarrow *address* \Rightarrow *id* \Rightarrow *port* \Rightarrow *port* \Rightarrow (*adr_{ip}*, *ftpvoip*) *history set* **where**
FTPVOIP-NB-voip *s d g i p1 p2* = {*x*. (*FTPVOIP-is-voip* *V0 s d g i p1 p2 x*)}

fun

FTPVOIP-is-ftp :: *ftp-states2* \Rightarrow *adr_{ip}* \Rightarrow *adr_{ip}* \Rightarrow *id* \Rightarrow *port* \Rightarrow (*adr_{ip}*, *ftpvoip*) *history* \Rightarrow *bool*

where

FTPVOIP-is-ftp *H c s i p []* = (*H*=*FS3*)
FTPVOIP-is-ftp *H c s i p (x#InL)* = (*snd s* = 21 \wedge ((λ (*id*,*sr*,*de*,*co*). (((*id* = *i* \wedge (*H*=*FS2* \wedge *sr* = *c* \wedge *de* = *s* \wedge *co* = *ftp-init* \wedge *FTPVOIP-is-ftp* *FS3 c s i p InL*) \vee

$$\begin{aligned}
& (H=FS1 \wedge sr = c \wedge de = s \wedge co = ftp\text{-}port\text{-}request\ p \wedge FTPVOIP\text{-}is\text{-}ftp\ FS2\ c\ s\ i\ p\ InL) \\
\vee \\
& (H=FS1 \wedge sr = s \wedge de = (fst\ c,p) \wedge co = ftp\text{-}data \wedge FTPVOIP\text{-}is\text{-}ftp\ FS1\ c\ s\ i\ p\ InL) \vee \\
& (H=FS0 \wedge sr = c \wedge de = s \wedge co = ftp\text{-}close \wedge FTPVOIP\text{-}is\text{-}ftp\ FS1\ c\ s\ i\ p\ InL)))))\ x))
\end{aligned}$$

definition

$$\begin{aligned}
FTPVOIP\text{-}NB\text{-}ftp &:: adr_{ip}\ src \Rightarrow adr_{ip}\ dest \Rightarrow id \Rightarrow port \Rightarrow \\
&\quad (adr_{ip}, ftpvoip)\ history\ set\ \mathbf{where} \\
FTPVOIP\text{-}NB\text{-}ftp\ s\ d\ i\ p &= \{x. (FTPVOIP\text{-}is\text{-}ftp\ FS0\ s\ d\ i\ p\ x)\}
\end{aligned}$$

definition

$$\begin{aligned}
ftp\text{-}voip\text{-}interleaved &:: adr_{ip}\ src \Rightarrow adr_{ip}\ dest \Rightarrow id \Rightarrow port \Rightarrow \\
&\quad address \Rightarrow address \Rightarrow address \Rightarrow id \Rightarrow port \Rightarrow port \Rightarrow \\
&\quad (adr_{ip}, ftpvoip)\ history\ set
\end{aligned}$$

where

$$\begin{aligned}
ftp\text{-}voip\text{-}interleaved\ s1\ d1\ i1\ p1\ vs\ vd\ vg\ vi\ vp1\ vp2 &= \\
\{x. (FTPVOIP\text{-}is\text{-}ftp\ FS0\ s1\ d1\ i1\ p1\ (packet\text{-}with\text{-}id\ x\ i1)) \wedge \\
&\quad (FTPVOIP\text{-}is\text{-}voip\ V0\ vs\ vd\ vg\ vi\ vp1\ vp2\ (packet\text{-}with\text{-}id\ x\ vi))\}
\end{aligned}$$

end

A. Appendix

```
theory NormalisationGenericProofs
imports FWNormalisationCore
begin
```

This theory contains the generic proofs of the normalisation procedure, i.e. those which are independent from the concrete semantical interpretation function.

```
lemma domNMT:  $\text{dom } X \neq \{\} \implies X \neq \emptyset$ 
apply auto
done
```

```
lemma denyNMT:  $\text{deny-all} \neq \emptyset$ 
apply (rule domNMT)
apply (simp add: deny-all-def dom-def)
done
```

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

```
lemma singleCombinatorsConc:  $\text{singleCombinators } (x \# xs) \implies \text{singleCombinators } xs$ 
by (case-tac x,simp-all)
```

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

```
lemma aux0-4:  $(a \in \text{set } x \vee a \in \text{set } y) = (a \in \text{set } (x @ y))$ 
by auto
```

```
lemma aux0-1:  $\llbracket \text{singleCombinators } xs; \text{singleCombinators } [x] \rrbracket \implies$ 
 $\text{singleCombinators } (x \# xs)$ 
by (case-tac x,simp-all)
```

```
lemma aux0-6:  $\llbracket \text{singleCombinators } xs; \neg (\exists a b. x = a \oplus b) \rrbracket \implies$ 
 $\text{singleCombinators } (x \# xs)$ 
apply (rule aux0-1,simp-all)
apply (case-tac x,simp-all)
apply auto
done
```

```

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

```

```

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

```

```

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

```

```

lemma aux5[rule-format]:  $(\text{DenyAllFromTo } a \ b) \in \text{set } p \longrightarrow a \in \text{set } (\text{net-list } p)$ 
by (rule net-list-aux.induct,simp-all)

```

```

lemma aux5a[rule-format]:  $(\text{DenyAllFromTo } b \ a) \in \text{set } p \longrightarrow a \in \text{set } (\text{net-list } p)$ 
by (rule net-list-aux.induct,simp-all)

```

```

lemma aux5c[rule-format]:
 $(\text{AllowPortFromTo } a \ b \ po) \in \text{set } p \longrightarrow a \in \text{set } (\text{net-list } p)$ 
by (rule net-list-aux.induct,simp-all)

```

```

lemma aux5d[rule-format]:
 $(\text{AllowPortFromTo } b \ a \ po) \in \text{set } p \longrightarrow a \in \text{set } (\text{net-list } p)$ 
by (rule net-list-aux.induct,simp-all)

```

```

lemma aux10[rule-format]:  $a \in \text{set } (\text{net-list } p) \longrightarrow a \in \text{set } (\text{net-list-aux } p)$ 
by simp

```

```

lemma srcInNetListaux[simp]:  $\llbracket x \in \text{set } p; \text{singleCombinators}[x]; x \neq \text{DenyAll} \rrbracket \implies$ 
 $\text{srcNet } x \in \text{set } (\text{net-list-aux } p)$ 
apply (induct p)
apply simp-all
apply (case-tac x = a, simp-all)
apply (case-tac a, simp-all)+
done

```

lemma *destInNetListaux*[simp]: $\llbracket x \in \text{set } p; \text{singleCombinators}[x]; x \neq \text{DenyAll} \rrbracket \implies$
 $\text{destNet } x \in \text{set } (\text{net-list-aux } p)$

apply (*induct* *p*)
apply *simp-all*
apply (*case-tac* $x = a$, *simp-all*)
apply (*case-tac* a , *simp-all*)
done

lemma *tND1*: $\llbracket \text{allNetsDistinct } p; x \in \text{set } p; y \in \text{set } p; a = \text{srcNet } x;$
 $b = \text{destNet } x; c = \text{srcNet } y; d = \text{destNet } y; a \neq c;$
 $\text{singleCombinators}[x]; x \neq \text{DenyAll}; \text{singleCombinators}[y];$
 $y \neq \text{DenyAll} \rrbracket \implies \text{twoNetsDistinct } a \ b \ c \ d$

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

lemma *tND2*: $\llbracket \text{allNetsDistinct } p; x \in \text{set } p; y \in \text{set } p; a = \text{srcNet } x;$
 $b = \text{destNet } x; c = \text{srcNet } y; d = \text{destNet } y; b \neq d;$
 $\text{singleCombinators}[x]; x \neq \text{DenyAll}; \text{singleCombinators}[y];$
 $y \neq \text{DenyAll} \rrbracket \implies \text{twoNetsDistinct } a \ b \ c \ d$

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

lemma *tND*: $\llbracket \text{allNetsDistinct } p; x \in \text{set } p; y \in \text{set } p; a = \text{srcNet } x;$
 $b = \text{destNet } x; c = \text{srcNet } y; d = \text{destNet } y; a \neq c \vee b \neq d;$
 $\text{singleCombinators}[x]; x \neq \text{DenyAll}; \text{singleCombinators}[y]; y \neq \text{DenyAll} \rrbracket$
 $\implies \text{twoNetsDistinct } a \ b \ c \ d$

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*: $\llbracket \text{DenyAllFromTo } a \ b \in \text{set } p; \text{allNetsDistinct } ((\text{DenyAllFromTo } c \ d) \# p);$
 $a \neq c \vee b \neq d \rrbracket \implies \text{twoNetsDistinct } a \ b \ c \ d$

apply (*erule-tac* $x = \text{DenyAllFromTo } a \ b$ **and** $y = \text{DenyAllFromTo } c \ d$ **in** *tND*)
apply *simp-all*
done

lemma *aux7a*: $\llbracket \text{DenyAllFromTo } a \ b \in \text{set } p;$
 $\text{allNetsDistinct } ((\text{AllowPortFromTo } c \ d \ po) \# p); a \neq c \vee b \neq d \rrbracket \implies$
 $\text{twoNetsDistinct } a \ b \ c \ d$

apply (*erule-tac* $x = \text{DenyAllFromTo } a \ b$ **and**
 $y = \text{AllowPortFromTo } c \ d \ po$ **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]*: $a \in \text{set } (\text{removeShadowRules2 } p) \longrightarrow a \in \text{set } p$
apply (*case-tac a*)
by (*rule removeShadowRules2.induct, simp-all*)⁺

lemma *aux12*: $\llbracket a \in x; b \notin x \rrbracket \Longrightarrow a \neq b$
by *auto*

lemma *ND0aux1[rule-format]*: $\text{DenyAllFromTo } x \ y \in \text{set } b \Longrightarrow$
 $x \in \text{set } (\text{net-list-aux } b)$
by (*metis aux5 net-list.simps set-remdups*)

lemma *ND0aux2[rule-format]*: $\text{DenyAllFromTo } x \ y \in \text{set } b \Longrightarrow$
 $y \in \text{set } (\text{net-list-aux } b)$
by (*metis aux5a net-list.simps set-remdups*)

lemma *ND0aux3[rule-format]*: $\text{AllowPortFromTo } x \ y \ p \in \text{set } b \Longrightarrow$
 $x \in \text{set } (\text{net-list-aux } b)$
by (*metis aux5c net-list.simps set-remdups*)

lemma *ND0aux4[rule-format]*: $\text{AllowPortFromTo } x \ y \ p \in \text{set } b \Longrightarrow$
 $y \in \text{set } (\text{net-list-aux } b)$
by (*metis aux5d net-list.simps set-remdups*)

lemma *aNDSubsetaux[rule-format]*: $\text{singleCombinators } a \longrightarrow \text{set } a \subseteq \text{set } b \longrightarrow$
 $\text{set } (\text{net-list-aux } a) \subseteq \text{set } (\text{net-list-aux } b)$

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*)
apply (*metis ND0aux3 ND0aux4 contra-subsetD*)
done

lemma *aNDSetsEqaux*[*rule-format*]: *singleCombinators a* \longrightarrow *singleCombinators b* \longrightarrow
 $\text{set } a = \text{set } b \longrightarrow \text{set } (\text{net-list-aux } a) = \text{set } (\text{net-list-aux } b)$

apply (*rule impI*) +
apply (*rule equalityI*)
apply (*rule aNDSubsetaux, simp-all*) +
done

lemma *aNDSubset*: $\llbracket \text{singleCombinators } a; \text{set } a \subseteq \text{set } b; \text{allNetsDistinct } b \rrbracket \Longrightarrow$
 $\text{allNetsDistinct } a$

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*: $\llbracket \text{singleCombinators } a; \text{singleCombinators } b; \text{set } a = \text{set } b; \text{allNetsDistinct } b \rrbracket \Longrightarrow$
 $\text{allNetsDistinct } a$

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*)
done

lemma *SCConca*: $\llbracket \text{singleCombinators } p; \text{singleCombinators } [a] \rrbracket \Longrightarrow$
 $\text{singleCombinators } (a \# p)$

by (*case-tac a, simp-all*)

lemma *aux3*[*simp*]: $\llbracket \text{singleCombinators } p; \text{singleCombinators } [a]; \text{allNetsDistinct } (a \# p) \rrbracket \Longrightarrow$
 $\text{allNetsDistinct } (a \# a \# p)$

apply (*insert aNDSubset[of (a#a#p) (a#p)]*)
by (*simp add: SCConca*)

lemma *wp1-aux1a*[*rule-format*]: $xs \neq [] \longrightarrow \text{wellformed-policy1-strong } (xs @ [x]) \longrightarrow$
 $\text{wellformed-policy1-strong } xs$

by (*induct xs, simp-all*)

lemma *wp1alternative-RS1*[*rule-format*]: $\text{DenyAll} \in \text{set } p \longrightarrow$
 $\text{wellformed-policy1-strong } (\text{removeShadowRules1 } p)$

by (*induct p, simp-all*)

lemma *wellformed-eq*: $\text{DenyAll} \in \text{set } p \longrightarrow$
 $((\text{wellformed-policy1 } p) = (\text{wellformed-policy1-strong } p))$

by (*induct p,simp-all*)

lemma *set-insort*: $\text{set}(\text{insort } x \text{ } xs \text{ } l) = \text{insert } x (\text{set } xs)$

by (*induct xs*) *auto*

lemma *set-sort[simp]*: $\text{set}(\text{sort } xs \text{ } l) = \text{set } xs$

by (*induct xs*) (*simp-all add:set-insort*)

lemma *set-sortQ*: $\text{set}(\text{qsort } xs \text{ } l) = \text{set } xs$

by *simp*

lemma *aux79[rule-format]*: $y \in \text{set } (\text{insort } x \text{ } a \text{ } l) \longrightarrow y \neq x \longrightarrow y \in \text{set } a$

apply (*induct a*)

by *auto*

lemma *aux80*: $\llbracket y \notin \text{set } p; y \neq x \rrbracket \implies y \notin \text{set } (\text{insort } x (\text{sort } p \text{ } l) \text{ } l)$

apply (*metis aux79 set-sort*)

done

lemma *WP1Conca*: $\text{DenyAll} \notin \text{set } p \implies \text{wellformed-policy1 } (a\#p)$

by (*case-tac a,simp-all*)

lemma *saux[simp]*: $(\text{insort } \text{DenyAll } p \text{ } l) = \text{DenyAll}\#p$

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

lemma *saux3[rule-format]*: $\text{DenyAllFromTo } a \text{ } b \in \text{set list} \longrightarrow$

$\text{DenyAllFromTo } c \text{ } d \notin \text{set list} \longrightarrow (a \neq c) \vee (b \neq d)$

by *blast*

lemma *waux2[rule-format]*: $(\text{DenyAll} \notin \text{set } xs) \longrightarrow \text{wellformed-policy1 } xs$

by (*induct-tac xs,simp-all*)

lemma *waux3[rule-format]*: $\llbracket x \neq a; x \notin \text{set } p \rrbracket \implies x \notin \text{set } (\text{insort } a \text{ } p \text{ } l)$

by (*metis aux79*)

lemma *wellformed1-sorted-aux[rule-format]*: $\text{wellformed-policy1 } (x\#p) \implies$

$\text{wellformed-policy1 } (\text{insort } x \text{ } p \text{ } l)$

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

by (*rule waux2,rule waux3, simp-all*)**+**

lemma *wellformed1-sorted-auxQ[rule-format]*: $\text{wellformed-policy1 } (p) \implies$

$\text{wellformed-policy1 } (\text{qsort } p \text{ } l)$

apply (*induct p*)

apply *simp-all*

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

apply (*metis append-Cons append-Nil mem-def member-rec(1) member-set not-Cons-self qsort.simps(2)*)


```

      set-qsort srcnets.simps(1) srcnets.simps(2) waux2)
apply (metis Combinators.simps(6) append-Cons append-Nil mem-def member-rec(1) member-set
      qsort.simps(2) set-qsort waux2)
by (metis Combinators.simps(8) append-Cons append-Nil mem-def member-rec(1) member-set
      qsort.simps(2) set-sortQ waux2)

```

```

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

```

```

proof (induct a)
  case Nil thus ?case by simp
next
  case (Cons x xs) thus ?case
    proof (cases x)
      case goal1 thus ?thesis by simp
    next
      case goal2 thus ?thesis by simp
    next
      case goal3 thus ?thesis by simp
    next
      case (Conc c d)
      have f:  $c \oplus e \in \text{set } b \longrightarrow \neg \text{singleCombinators } b$ 
        by (rule singleCombinators.induct,simp-all)
      from this show ?thesis
      apply simp
      by (metis Conc aux0-0)
    qed
  qed

```

```

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*: $a \notin \text{set } list \implies a \notin \text{set } (\text{removeShadowRules2 } list)$
apply (*insert RS2Set [of list]*)
apply *blast*
done

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

lemma *lCdom2*: $(\text{list2FWpolicy } (a @ (b @ c))) = (\text{list2FWpolicy } ((a @ b) @ c))$
by *auto*

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

lemma *list2FWpolicyconc*[*rule-format*]: $a \neq [] \longrightarrow$
 $(\text{list2FWpolicy } (xa \# a)) = (xa) \oplus (\text{list2FWpolicy } a)$
by (*induct a, simp-all*)

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

lemma *foo2*:
 $\llbracket a \notin \text{set } ps; a \notin \text{set } ss; \text{set } p = \text{set } s; p = (a \# (ps)); s = (a \# ss) \rrbracket \implies$
 $\text{set } (ps) = \text{set } (ss)$
by *auto*

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

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

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

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

by (*case-tac* x , *simp-all*)

lemma *aux115*[*rule-format*]: $x \neq \text{DenyAll} \implies \text{removeShadowRules1-alternative } (xs @ [x])$
 $= (\text{removeShadowRules1-alternative } xs) @ [x]$

apply (*simp add*: *removeShadowRules1-alternative-def aux114*)
done

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

lemma *rSR1-eq*: $\text{removeShadowRules1-alternative} = \text{removeShadowRules1}$
apply (*rule ext*)
apply (*simp add*: *removeShadowRules1-alternative-def*)
apply (*rule-tac* $xs = x$ **in** *rev-induct*)
apply *simp-all*
apply (*case-tac* $xa = \text{DenyAll}$, *simp-all*)
apply (*metis RS1End aux114 rev.simps*(2))
done

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

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

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

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

lemma *mauxa*: $(\exists r. a \ b = \lfloor r \rfloor) = (a \ b \neq \perp)$
by *auto*

lemma *l2p-aux*[*rule-format*]: $\text{list} \neq [] \longrightarrow$
 $\text{list2FWpolicy } (a \ \# \ \text{list}) = a \oplus (\text{list2FWpolicy } \text{list})$
by (*induct* list , *simp-all*)

lemma *l2p-aux2*[*rule-format*]: $\text{list} = [] \implies \text{list2FWpolicy } (a \ \# \ \text{list}) = a$
by *simp*

lemma *aux7aa*: $\llbracket \text{AllowPortFromTo } a \ b \ \text{poo} \in \text{set } p;$
 $\text{allNetsDistinct } ((\text{AllowPortFromTo } c \ d \ \text{po}) \ \# \ p); a \neq c \vee b \neq d \rrbracket \implies$
 $\text{twoNetsDistinct } a \ b \ c \ d$
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)+
done

```

```

lemma ANDConcEnd:  $\llbracket \text{allNetsDistinct } (xs \text{ @ } [xa]); \text{singleCombinators } xs \rrbracket \implies$ 
  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 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 assms Cons by simp
  next
  case False thus ?thesis using assms
  by (metis Cons set-sort False waux2 wellformed-eq
    wellformed-policy1-strong.simps(2))
  qed
qed

```

```

lemma wellformed1-sortedQ[simp]:
  assumes wp1: wellformed-policy1 p
  shows      wellformed-policy1 (qsort 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 assms Cons by simp
  next
  case False thus ?thesis using assms

```

```

    by (metis Cons set-qsort False waux2 wellformed-eq
        wellformed-policy1-strong.simps(2))
  qed
qed

```

```

lemma SC1[simp]: singleCombinators p  $\implies$  singleCombinators (removeShadowRules1 p)
by (erule SCSubset) (rule SR1Subset)

```

```

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

```

```

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

```

```

lemma SC3Q[simp]: singleCombinators p  $\implies$  singleCombinators (qsort p l)
by (erule SCSubset) simp

```

```

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

```

```

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

```

```

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

```

```

lemma aND-sortQ[simp]:  $\llbracket$ singleCombinators p; allNetsDistinct p $\rrbracket \implies$ 
    allNetsDistinct (qsort p l)
apply (rule aNDSubset)
by (erule SC3Q, simp-all)

```

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

lemma *distinct-RS2*[*rule-format,simp*]: $\text{distinct } p \longrightarrow \text{distinct } (\text{removeShadowRules2 } p)$
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 doubleton-eq-iff*)

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

lemma *pos-noteq*[*rule-format*]:
 $a \in \text{set } l \longrightarrow b \in \text{set } l \longrightarrow c \in \text{set } l \longrightarrow a \neq b \longrightarrow$
 $(\text{position } a \ l) <= (\text{position } b \ l) \longrightarrow$
 $(\text{position } b \ l) <= (\text{position } c \ l) \longrightarrow$
 $a \neq c$
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*:
 $\llbracket \text{in-list } x \ l; \text{in-list } y \ l; \text{in-list } z \ l; \text{singleCombinators } [x];$
 $\text{singleCombinators } [y]; \text{singleCombinators } [z]; \text{smaller } x \ y \ l; \text{smaller } y \ z \ l \rrbracket \Longrightarrow$
 $\text{smaller } x \ z \ l$
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  $\longrightarrow$  in-list a l  $\longrightarrow$  in-list aa l  $\longrightarrow$  all-in-list p l  $\longrightarrow$ 
  singleCombinators [a]  $\longrightarrow$  singleCombinators [aa]  $\longrightarrow$  singleCombinators p  $\longrightarrow$ 
  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 ( $x \# xs$ )  $\implies$ 
singleCombinators [ $x$ ]
by (case-tac  $x$ , simp-all)

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

lemma sortedConcEnd[rule-format]:  $sorted\ (a \# p)\ l \longrightarrow in-list\ a\ l \longrightarrow$ 
 $all-in-list\ p\ l \longrightarrow singleCombinators\ [a] \longrightarrow$ 
 $singleCombinators\ p \longrightarrow sorted\ p\ l$ 
apply (induct  $p$ )
apply (auto intro: singleCombinatorsConc sortedConcStart)
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-Consb[rule-format]:
 $all-in-list\ (x \# xs)\ l \longrightarrow singleCombinators\ (x \# xs) \longrightarrow$ 
 $(sorted\ xs\ l \ \&\ (ALL\ y: set\ xs.\ smaller\ x\ y\ l)) \longrightarrow (sorted\ (x \# xs)\ l)$ 
apply(induct  $xs$  arbitrary:  $x$ )
apply simp
apply (auto simp: order-trans)
done

lemma sorted-Cons:  $\llbracket all-in-list\ (x \# xs)\ l; singleCombinators\ (x \# xs) \rrbracket \implies$ 
 $(sorted\ xs\ l \ \&\ (ALL\ y: set\ xs.\ smaller\ x\ y\ l)) = (sorted\ (x \# xs)\ l)$ 
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:  $\llbracket \neg\ smaller\ a\ b\ l; in-list\ a\ l; in-list\ b\ l;$ 
 $singleCombinators[a]; singleCombinators\ [b] \rrbracket \implies$ 
 $smaller\ b\ a\ l$ 
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:  $\text{set } (\text{insort } x \text{ } xs \text{ } l) \subseteq \text{insert } x \text{ } (\text{set } xs)$ 
by (induct xs) auto

lemma all-in-listSubset[rule-format]:  $\text{all-in-list } b \text{ } l \longrightarrow \text{singleCombinators } a \longrightarrow$ 
 $\text{set } a \subseteq \text{set } b \longrightarrow \text{all-in-list } a \text{ } l$ 
by (induct-tac a) (auto intro: in-set-in-list singleCombinatorsConc)

lemma singleCombinators-insort:  $\llbracket \text{singleCombinators } [x]; \text{singleCombinators } xs \rrbracket \Longrightarrow$ 
 $\text{singleCombinators } (\text{insort } x \text{ } xs \text{ } l)$ 
by (metis SCSubset SCConca set-insort set.simps(2) subset-refl)

lemma all-in-list-insort:  $\llbracket \text{all-in-list } xs \text{ } l; \text{singleCombinators } (x \# xs);$ 
 $\text{in-list } x \text{ } l \rrbracket \Longrightarrow \text{all-in-list } (\text{insort } x \text{ } xs \text{ } l) \text{ } l$ 
apply (rule-tac b = x \# xs in all-in-listSubset)
apply simp-all
apply (metis singleCombinatorsConc singleCombinatorsStart
 $\text{singleCombinators-insort}$ )
apply (rule set-insort-insert)
done

lemma sorted-ConsA:  $\llbracket \text{all-in-list } (x \# xs) \text{ } l; \text{singleCombinators } (x \# xs) \rrbracket \Longrightarrow$ 
 $(\text{sorted } (x \# xs) \text{ } l) = (\text{sorted } xs \text{ } l \ \& \ (\text{ALL } y:\text{set } xs. \text{smaller } x \text{ } y \text{ } l))$ 
by (metis sorted-Cons)

lemma is-in-insort:  $y \in \text{set } xs \Longrightarrow y \in \text{set } (\text{insort } x \text{ } xs \text{ } l)$ 
by (metis ListMem-iff insert set-insort set.simps(2))

lemma sorted-insorta[rule-format]:
 $\text{sorted } (\text{insort } x \text{ } xs \text{ } l) \text{ } l \longrightarrow \text{all-in-list } (x \# xs) \text{ } l \longrightarrow \text{distinct } (x \# xs) \longrightarrow$ 
 $\text{singleCombinators } [x] \longrightarrow \text{singleCombinators } xs \longrightarrow \text{sorted } xs \text{ } l$ 
apply (induct xs)
apply simp-all
apply (rule impI)+
apply simp
apply (auto intro: is-in-insort sorted-ConsA set-insort singleCombinators-insort
 $\text{singleCombinatorsConc sortedConcEnd all-in-list-insort}$ )
apply (metis sort.simps(2) set-sort SCSubset all-in-list-insort set-subset-Cons
 $\text{singleCombinators.simps(3) singleCombinatorsConc singleCombinatorsStart}$ )

```

```

    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  $\longrightarrow$  all-in-list (x#xs) l  $\longrightarrow$  distinct (x#xs)  $\longrightarrow$ 
  singleCombinators [x]  $\longrightarrow$  singleCombinators xs  $\longrightarrow$  sorted (insort x xs l) l
  apply (induct xs)
  apply simp-all
  apply (rule impI)+
  apply (subgoal-tac sorted (insort x xs l) l)
  apply simp-all
  defer 1
  apply (metis 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  $\in$  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:  $\llbracket$ all-in-list (x#xs) l; distinct(x#xs); singleCombinators [x];
  singleCombinators xs $\rrbracket \implies$ 
  sorted (insort x xs l) l = sorted xs l
  by (auto intro: sorted-insorta sorted-insortb)

```

lemma *distinct-insort*: $\text{distinct } (\text{insort } x \text{ } xs \text{ } l) = (x \notin \text{set } xs \wedge \text{distinct } xs)$
by (*induct xs*) (*auto simp: set-insort*)

lemma *distinct-sort[simp]*: $\text{distinct } (\text{sort } xs \text{ } l) = \text{distinct } xs$
by (*induct xs*) (*simp-all add: distinct-insort*)

lemma *sort-is-sorted[rule-format]*: $\text{all-in-list } p \text{ } l \longrightarrow \text{distinct } p \longrightarrow$
 $\text{singleCombinators } p \longrightarrow \text{sorted } (\text{sort } p \text{ } l) \text{ } l$
apply (*induct p*)
apply (*auto intro: SC3 all-in-listSubset singleCombinatorsConc sorted-insort*)
apply (*subst sorted-insort*)
apply (*auto intro: singleCombinatorsConc all-in-listSubset SC3*)
apply (*erule all-in-listSubset*)
by (*auto intro: SC3 singleCombinatorsConc sorted-insort*)

lemma *smaller-sym[rule-format]*: $\text{all-in-list } [a] \text{ } l \longrightarrow \text{smaller } a \text{ } a \text{ } l$
apply (*case-tac a, simp-all*)
done

lemma *SC-sublist[rule-format]*: $\text{singleCombinators } xs \Longrightarrow$
 $\text{singleCombinators } (qsort \text{ } [y \leftarrow xs. P \text{ } y] \text{ } l)$
apply (*auto intro: SCSubset*)
done

lemma *all-in-list-sublist[rule-format]*: $\text{singleCombinators } xs \longrightarrow \text{all-in-list } xs \text{ } l \longrightarrow$
 $\text{all-in-list } (qsort \text{ } [y \leftarrow xs. P \text{ } y] \text{ } l) \text{ } l$
apply (*auto intro: all-in-listSubset SC-sublist*)
done

lemma *SC-sublist2[rule-format]*: $\text{singleCombinators } xs \longrightarrow$
 $\text{singleCombinators } ([y \leftarrow xs. P \text{ } y])$
apply (*auto intro: SCSubset*)
done

lemma *all-in-list-sublist2[rule-format]*: $\text{singleCombinators } xs \longrightarrow \text{all-in-list } xs \text{ } l \longrightarrow$
 $\text{all-in-list } ([y \leftarrow xs. P \text{ } y]) \text{ } l$
apply (*auto intro: all-in-listSubset SC-sublist2*)
done

lemma *all-in-listAppend[rule-format]*:
 $\text{all-in-list } (xs) \text{ } l \longrightarrow \text{all-in-list } (ys) \text{ } l \longrightarrow \text{all-in-list } (xs @ ys) \text{ } l$

```

apply (induct xs)
apply simp-all
done

```

```

lemma distinct-sortQ[rule-format]: singleCombinators xs  $\longrightarrow$ 
all-in-list xs l  $\longrightarrow$  distinct xs  $\longrightarrow$  distinct (qsort xs l)
apply (induct xs l rule: qsort.induct)
apply simp-all
apply (rule impI)+
apply (auto simp: SC-sublist2 singleCombinatorsConc all-in-list-sublist2)
done

```

```

lemma singleCombinatorsAppend[rule-format]:
singleCombinators (xs)  $\longrightarrow$  singleCombinators (ys)  $\longrightarrow$  singleCombinators (xs @ ys)
apply (induct xs)
apply simp-all
apply (rule impI)+
apply simp-all
apply (drule mp)+
apply (case-tac a, simp-all)+
done

```

```

lemma sorted-append1[rule-format]:
all-in-list xs l  $\longrightarrow$  singleCombinators xs  $\longrightarrow$ 
all-in-list ys l  $\longrightarrow$ 
singleCombinators ys  $\longrightarrow$ 
(sorted (xs@ys) l  $\longrightarrow$  (sorted xs l & sorted ys l & ( $\forall x \in \text{set } xs. \forall y \in \text{set } ys. \text{smaller } x \ y \ l$ )))
apply (induct xs)
apply simp-all
apply (rule impI)+
apply simp
apply (drule mp)+
apply (case-tac a, simp-all)
apply (drule mp)+
apply (erule sortedConcEnd)
apply simp-all
apply (rule all-in-listAppend, simp-all)
apply (rule singleCombinatorsAppend, simp-all)
apply (case-tac a, simp-all)
apply (rule conjI)
apply (smt all-in-list.simps(2) all-in-listAppend append-Cons aux0-4 sorted-Cons singleCombinatorsAppend)
by (metis all-in-list.simps(2) all-in-listAppend append-Cons aux0-4 sorted-Cons singleCombinatorsAppend)

```

```

lemma sorted-append2[rule-format]:

```

```

all-in-list xs l → singleCombinators xs →
all-in-list ys l →
singleCombinators ys →
(sorted xs l & sorted ys l & (∀ x ∈ set xs. ∀ y ∈ set ys. smaller x y l)) → (sorted (xs@ys) l)
apply (induct xs)
apply simp-all
apply (rule impI)+
apply simp
apply (drule mp)+
apply (case-tac a,simp-all)
apply (drule mp)
apply (rule-tac a = a in sortedConcEnd)
apply simp-all
apply (case-tac a,simp-all)
by (smt all-in-list.simps(2) all-in-listAppend append-Cons aux0-4 sorted-Cons singleCombinatorsAppend sorted-Consb)

```

```

lemma sorted-append[rule-format]:
all-in-list xs l → singleCombinators xs →
all-in-list ys l →
singleCombinators ys → (sorted (xs@ys) l) =
(sorted xs l & sorted ys l & (∀ x ∈ set xs. ∀ y ∈ set ys. smaller x y l))
apply (rule impI)+
apply (rule iffI)
apply (rule sorted-append1,simp-all)
apply (rule sorted-append2,simp-all)
done

```

```

lemma sort-is-sortedQ[rule-format]: all-in-list p l →
singleCombinators p → sorted (qsort p l) l
apply (induct p l rule: qsort.induct)
apply simp-all
apply (rule impI)+
apply (simp-all add: SC-sublist all-in-list-sublist all-in-list-sublist2 singleCombinatorsConc SC-sublist2)
apply (subst sorted-append)
apply (metis all-in-list-sublist singleCombinatorsConc)
apply (auto simp add: SC-sublist all-in-list-sublist SC-sublist2 all-in-list-sublist2 sorted-Cons sorted-append not-le less-imp-le )
apply (metis SC3Q SC-sublist2 singleCombinatorsConc)
apply (metis all-in-list-sublist singleCombinatorsConc)
apply (metis SC3Q SC-sublist2 aux0-1 singleCombinatorsConc singleCombinatorsStart)
apply (rule sorted-Consb)
apply (metis all-in-list.simps(2) all-in-list-sublist singleCombinatorsConc)
apply (metis SC3Q SC-sublist2 aux0-1 singleCombinatorsConc singleCombinatorsStart)
apply (rule conjI)
apply simp-all
apply (rule smaller-antisym)

```

```

apply simp-all
apply (metis in-set-in-list)
apply (erule SCSubset)
apply auto
apply (subgoal-tac smaller xa x l)
apply (rule-tac y = x in order-trans)
apply simp-all
apply (metis in-set-in-list)
apply (metis in-set-in-list)
apply (erule SCSubset)
apply simp
apply (erule SCSubset)
apply simp
apply (rule smaller-antisym)
apply simp-all
apply (metis in-set-in-list)
apply (erule SCSubset)
apply simp
done

```

```

lemma inSet-not-MT:  $a \in \text{set } p \implies p \neq []$ 
by auto

```

```

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

```

```

lemma RS1n-nMT[rule-format, simp]:  $p \neq [] \longrightarrow \text{removeShadowRules1-alternative } p \neq []$ 
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]:  $\text{removeShadowRules1-alternative } (a @ [\text{DenyAll}]) = [\text{DenyAll}]$ 
by (simp add: removeShadowRules1-alternative-def)

```

```

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

```

```

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

```

```

lemma DAnotTL[rule-format]:

```

$xs \neq [] \longrightarrow \text{wellformed-policy1 } (xs @ [DenyAll]) \longrightarrow \text{False}$
by (*induct xs, simp-all*)

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

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

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

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

lemma *wp1-tl[rule-format]*: $p \neq [] \wedge \text{wellformed-policy1 } p \longrightarrow \text{wellformed-policy1 } (tl\ p)$
apply (*induct p*)
apply *simp-all*
apply (*auto intro: waux2*)
done

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

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

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

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

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

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

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

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

```

```

lemma SCrd[rule-format,simp]:singleCombinators  $p \longrightarrow \text{singleCombinators}(\text{remdups } p)$ 
apply (induct  $p$ , simp-all)
apply (case-tac  $a$ )
apply 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 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 *ANDiD*[*rule-format, simp*]: $\text{allNetsDistinct } p \longrightarrow \text{allNetsDistinct } (\text{insertDeny } p)$
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 *mr-iD*[*rule-format*]: $\text{wellformed-policy1-strong } p \longrightarrow \text{matching-rule } x \text{ } p = \text{matching-rule } x (\text{insertDeny } p)$
by (*induct* p , *simp-all*)

lemma *WP1iD*[*rule-format, simp*]: $\text{wellformed-policy1-strong } p \longrightarrow \text{wellformed-policy1-strong } (\text{insertDeny } p)$
by (*induct* p , *simp-all*)

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

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

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

lemma *SCConc*: $\llbracket \text{singleCombinators } x; \text{singleCombinators } y \rrbracket \Longrightarrow \text{singleCombinators } (x@y)$
apply (*rule* *aux0-5*)
apply (*metis* *aux0-0* *aux0-4*)
done

lemma *SCp2l*: $\text{singleCombinators } (\text{policy2list } p)$
apply (*induct-tac* p)
apply *simp-all*
apply (*rule* *SCConc*)
apply *simp-all*
done

lemma *subnetAux*: $Dd \cap A \neq \{\} \Longrightarrow A \subseteq B \Longrightarrow Dd \cap B \neq \{\}$

apply *auto*
done

lemma *soadisj*: $\llbracket x \in \text{subnetsOfAdr } a; y \in \text{subnetsOfAdr } a \rrbracket \implies \neg \text{netsDistinct } x \ y$
by (*simp add: subnetsOfAdr-def netsDistinct-def, auto simp: PLemmas*)

lemma *not-member*: $\neg \text{member } a \ (x \oplus y) \implies \neg \text{member } a \ x$
apply *auto*
done

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

lemma *ndFalse1*: $\llbracket (\forall \ a \ b \ c \ d. (a,b) \in A \wedge (c,d) \in B \longrightarrow \text{netsDistinct } a \ c);$
 $\exists (a, b) \in A. a \in \text{subnetsOfAdr } D;$
 $\exists (a, b) \in B. a \in \text{subnetsOfAdr } D \rrbracket$
 $\implies \text{False}$
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*: $\llbracket (\forall \ a \ b \ c \ d. (a,b) \in A \wedge (c,d) \in B \longrightarrow \text{netsDistinct } b \ d);$
 $\exists (a, b) \in A. b \in \text{subnetsOfAdr } D;$
 $\exists (a, b) \in B. b \in \text{subnetsOfAdr } D \rrbracket$
 $\implies \text{False}$
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*: $\llbracket (\forall \ a \ b \ c \ d. (a,b) \in A \wedge (c,d) \in B \longrightarrow \text{twoNetsDistinct } a \ b \ c \ d);$
 $\exists (a, b) \in A. a \in \text{subnetsOfAdr } (D::('a::\text{adr})) \wedge b \in \text{subnetsOfAdr } (F::'a);$
 $\exists (a, b) \in B. a \in \text{subnetsOfAdr } D \wedge b \in \text{subnetsOfAdr } F \rrbracket$
 $\implies \text{False}$
apply (*simp add: twoNetsDistinct-def*)
apply (*auto simp: ndFalse1 ndFalse2*)
apply (*metis soadisj*)
done

lemma *sepnMT*[*rule-format*]: $p \neq [] \longrightarrow (\text{separate } p) \neq []$
apply (*rule separate.induct*) **back back back**

by *simp-all*

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

lemma *setnMT*: $\text{set } a = \text{set } b \implies a \neq [] \implies b \neq []$
by *auto*

lemma *sortnMT*: $p \neq [] \implies \text{sort } p \text{ } l \neq []$
by (*metis set-sort setnMT*)

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

lemma *OTNoTN*[*rule-format*]: $\text{OnlyTwoNets } p \longrightarrow x \neq \text{DenyAll} \longrightarrow x \in \text{set } p \longrightarrow \text{onlyTwoNets } x$
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*]:
 $\neg \text{member DenyAll } x \longrightarrow (\text{first-srcNet } x, \text{first-destNet } x) \in \text{sdnets } x$
by (*induct x, case-tac x, simp-all*)

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

apply (*erule first-isIn*)
done

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

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

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

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

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

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

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

lemma *disjComm*: $\text{disjSD-2 } a \ b \implies \text{disjSD-2 } b \ a$
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:  $\llbracket \text{disjSD-2 } a \ b; \neg \text{member DenyAll } a; \neg \text{member DenyAll } b \rrbracket \implies$ 
 $\text{disjSD-2 } (\text{DenyAllFromTo } (\text{first-srcNet } a) (\text{first-destNet } a) \oplus$ 
 $\text{DenyAllFromTo } (\text{first-destNet } a) (\text{first-srcNet } a) \oplus a) \ b$ 
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,
 $\text{drule-tac } x = \text{first-srcNet } a \text{ in spec,}$ 
 $\text{drule-tac } x = \text{first-destNet } a \text{ in spec, auto intro: first-isIn})+$ 
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:  $\llbracket \text{disjSD-2 } x \ y; \neg \text{member DenyAll } x; \neg \text{member DenyAll } y;$ 
 $a = (\text{first-srcNet } x); b = (\text{first-destNet } x) \rrbracket \implies$ 
 $\text{disjSD-2 } ((\text{DenyAllFromTo } a \ b) \oplus (\text{DenyAllFromTo } b \ a) \oplus x) \ y$ 
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*]: $\text{noDenyAll } p \longrightarrow s \in \text{set } (\text{insertDenies } p) \longrightarrow$
 $(\exists! a. s = (\text{DenyAllFromTo } (\text{first-srcNet } a) (\text{first-destNet } a)) \oplus$
 $(\text{DenyAllFromTo } (\text{first-destNet } a) (\text{first-srcNet } a)) \oplus a \wedge a \in \text{set } p)$

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*]: $\text{DenyAllFromTo } (\text{first-srcNet } s) (\text{first-destNet } s) \oplus$
 $\text{DenyAllFromTo } (\text{first-destNet } s) (\text{first-srcNet } s) \oplus s \in \text{set } (\text{insertDenies } p)$
 $\longrightarrow s \in \text{set } p$

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

lemma *id-aux2*:

$\llbracket \text{noDenyAll } p; (\forall s. s \in \text{set } p \longrightarrow \text{disjSD-2 } a \ s); \neg \text{member } \text{DenyAll } a;$
 $((\text{DenyAllFromTo } (\text{first-srcNet } s) (\text{first-destNet } s)) \oplus (\text{DenyAllFromTo } (\text{first-destNet } s) (\text{first-srcNet } s)) \oplus s) \in \text{set } (\text{insertDenies } p) \rrbracket \implies$
 $\text{disjSD-2 } a ((\text{DenyAllFromTo } (\text{first-srcNet } s) (\text{first-destNet } s)) \oplus$
 $(\text{DenyAllFromTo } (\text{first-destNet } s) (\text{first-srcNet } s)) \oplus s)$

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*]: $\llbracket \text{noDenyAll } p; (\forall s. s \in \text{set } p \longrightarrow$
 $\text{disjSD-2 } a \ s); s \in \text{set } (\text{insertDenies } p); \neg \text{member } \text{DenyAll } a \rrbracket \implies \text{disjSD-2 } a \ s$

apply (*subgoal-tac* $\exists a. s =$
 $\text{DenyAllFromTo } (\text{first-srcNet } a) (\text{first-destNet } a) \oplus$
 $\text{DenyAllFromTo } (\text{first-destNet } a) (\text{first-srcNet } a) \oplus a \wedge$
 $a \in \text{set } p)$

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  $a = \text{DenyAll}$ )
apply simp-all
apply (simp add: disjSD-2-def)
apply (case-tac  $a, \text{simp-all}$ )
apply auto
apply (rule disjSD-2IDa, simp-all, rule id-aux4, simp-all, metis noDA noDAID)+
done

```

```

lemma aNDDA[rule-format]: allNetsDistinct  $p \longrightarrow$  allNetsDistinct( $\text{DenyAll}\#p$ )
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  $y, \text{simp-all}$ )
done

```

```

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

```

```

lemma bNaux:
 $\llbracket \neg \text{member DenyAll } x; \neg \text{member DenyAll } y; \text{first-bothNet } x = \text{first-bothNet } y \rrbracket$ 
 $\implies \{\text{first-srcNet } x, \text{first-destNet } x\} = \{\text{first-srcNet } y, \text{first-destNet } y\}$ 
apply (simp add: first-bothNetsd)
done

```

```

lemma setPair:  $\{a, b\} = \{a, d\} \implies b = d$ 
apply (metis setPaireq)
done

```

```

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

```

done

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

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

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

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

lemma *otnaux*:

$\llbracket \text{first-bothNet } x = \text{first-bothNet } y; \neg \text{member DenyAll } x; \neg \text{member DenyAll } y;$
 $\text{onlyTwoNets } y; \text{onlyTwoNets } x \rrbracket \implies$
 $\text{onlyTwoNets } (x \oplus y)$

apply (*simp add: onlyTwoNets-def*)

apply (*subgoal-tac* $\{\text{first-srcNet } x, \text{first-destNet } x\} =$
 $\{\text{first-srcNet } y, \text{first-destNet } y\}$)

apply (*case-tac* $(\exists a b. \text{sdnets } y = \{(a, b)\})$)

apply *simp-all*

apply (*case-tac* $(\exists a b. \text{sdnets } x = \{(a, b)\})$)

apply *simp-all*

apply (*subgoal-tac* $\text{sdnets } x = \{(\text{first-srcNet } x, \text{first-destNet } x)\}$)

apply (*subgoal-tac* $\text{sdnets } y = \{(\text{first-srcNet } y, \text{first-destNet } y)\}$)

apply *simp*

apply (*case-tac* $\text{first-srcNet } x = \text{first-srcNet } y$)

apply *simp-all*

apply (*rule disjI1*)

apply (*rule setPair*)

apply *simp*

apply (*subgoal-tac* $\text{first-srcNet } x = \text{first-destNet } y$)

apply *simp*

apply (*subgoal-tac* $\text{first-destNet } x = \text{first-srcNet } y$)

apply *simp*

apply (*rule-tac* $x = \text{first-srcNet } y$ **in** *exI*,
 $\text{rule-tac } x = \text{first-destNet } y$ **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* $\text{sdnets } x = \{(\text{first-srcNet } x, \text{first-destNet } x),$
 $(\text{first-destNet } x, \text{first-srcNet } x)\}$)

apply (*subgoal-tac* $\text{sdnets } y = \{(\text{first-srcNet } y, \text{first-destNet } y)\}$)


```

apply simp
apply (case-tac first-srcNet  $x = \text{first-srcNet } y$ )
apply simp-all
apply (subgoal-tac first-destNet  $x = \text{first-destNet } y$ )
apply simp
apply (rule setPair)
apply simp
apply (subgoal-tac first-srcNet  $x = \text{first-destNet } y$ )
apply simp
apply (subgoal-tac first-destNet  $x = \text{first-srcNet } y$ )
apply simp
apply (rule-tac  $x = \text{first-srcNet } y$  in exI,
      rule-tac  $x = \text{first-destNet } y$  in exI)
apply (metis OTNIDaux4 insert-commute )
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 ( $\exists a b. \text{sdnets } x = \{(a, b)\}$ ))
apply simp-all
apply (subgoal-tac sdnets  $x = \{(\text{first-srcNet } x, \text{first-destNet } x)\}$ )
apply (subgoal-tac sdnets  $y = \{(\text{first-srcNet } y, \text{first-destNet } y),$ 
       $(\text{first-destNet } y, \text{first-srcNet } y)\}$ )

apply simp
apply (case-tac first-srcNet  $x = \text{first-srcNet } y$ )
apply simp-all
apply (subgoal-tac first-destNet  $x = \text{first-destNet } y$ )
apply simp
apply (rule-tac  $x = \text{first-srcNet } y$  in exI,
      rule-tac  $x = \text{first-destNet } y$  in exI)
apply (metis OTNIDaux4 insert-commute )
apply (rule setPair)
apply simp
apply (subgoal-tac first-srcNet  $x = \text{first-destNet } y$ )
apply simp
apply (subgoal-tac first-destNet  $x = \text{first-srcNet } y$ )
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 OTNIDaux4 insert-commute)
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:  $\llbracket onlyTwoNets\ (a \oplus y) \wedge OnlyTwoNets\ z \longrightarrow$ 
   $OnlyTwoNets\ (separate\ (a \oplus y\ \# z));$ 
   $\neg member\ DenyAll\ a;$ 
   $\neg member\ DenyAll\ y; noDenyAll\ z;$ 
   $onlyTwoNets\ a; OnlyTwoNets\ (y\ \# z); first-bothNet\ (a) = first-bothNet\ y \rrbracket$ 
   $\implies OnlyTwoNets\ (separate\ (a \oplus y\ \# z))$ 
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 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  $\implies$  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 = ( $\neg$  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  $\implies$  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  $\in$  set (net-list-aux [a])) $\implies$ 
    aa  $\in$  set (net-list-aux (a # p))
apply (case-tac a,simp-all)
apply safe
done

```

```

lemma isInAlternativeListb: (aa  $\in$  set (net-list-aux p)) $\implies$ 
    aa  $\in$  set (net-list-aux (a # p))
apply (case-tac a,simp-all)
done

```

```

lemma ANDSepaux: allNetsDistinct (x # y # z)  $\implies$  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:  $\llbracket$ allNetsDistinct (x # y # z);
  allNetsDistinct (separate (y # z)) $\rrbracket$ 
 $\implies$  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  $a \in \text{set } (\text{net-list-aux } [x])$ )
apply simp-all
apply (rule isInAlternativeLista)
apply simp
apply (rule isInAlternativeListb)
apply (subgoal-tac  $a \in \text{set } (\text{net-list-aux } (\text{separate } (y \# z)))$ )
apply (metis netlistalternativeSeparate)
apply (metis netlistaux netlistalternativeSeparate)
apply (case-tac  $b \in \text{set } (\text{net-list-aux } [x])$ )
apply (rule isInAlternativeLista)
apply simp
apply (rule isInAlternativeListb)
apply (subgoal-tac  $b \in \text{set } (\text{net-list-aux } (\text{separate } (y \# z)))$ )
apply (metis netlistalternativeSeparate)
apply (metis netlistaux netlistalternativeSeparate)
done

```

```

lemma ANDSep[rule-format]: allNetsDistinct  $p \longrightarrow \text{allNetsDistinct}(\text{separate } p)$ 
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 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  $p \longrightarrow \text{noDenyAll1 } (\text{sort } p \ l)$ 
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  $p \longrightarrow \text{OnlyTwoNets } p$ 
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:  $\neg \text{member DenyAll } x \implies \text{first-bothNet } x \neq \{\}$ 
apply (metis first-bothNetsd insert-commute 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  $p \longrightarrow (\text{first-bothNet } x \neq \text{firstList } p \longrightarrow$ 
 $(\forall a \in \text{set } p. \text{first-bothNet } x \neq \text{first-bothNet } a)) \longrightarrow \text{NetsCollected } (x \# p)$ 
apply (induct  $p$ )
apply simp-all
done

```

```

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

```

```

lemma smalleraux2:
 $\{a, b\} \in \text{set } l \implies \{c, d\} \in \text{set } l \implies \{a, b\} \neq \{c, d\} \implies$ 
 $\text{smaller } (\text{DenyAllFromTo } a \ b) \ (\text{DenyAllFromTo } c \ d) \ l \implies$ 
 $\neg \text{smaller } (\text{DenyAllFromTo } c \ d) \ (\text{DenyAllFromTo } a \ b) \ l$ 
apply simp
apply (rule conjI)
apply (rule impI)
apply simp
apply (metis)
apply (metis pos-noteq)
done

```

```

lemma smalleraux2a:
 $\{a, b\} \in \text{set } l \implies \{c, d\} \in \text{set } l \implies \{a, b\} \neq \{c, d\} \implies$ 
 $\text{smaller } (\text{DenyAllFromTo } a \ b) \ (\text{AllowPortFromTo } c \ d \ p) \ l \implies$ 
 $\neg \text{smaller } (\text{AllowPortFromTo } c \ d \ p) \ (\text{DenyAllFromTo } a \ b) \ l$ 
apply simp
apply (metis eq-imp-le pos-noteq)
done

```

lemma *smalleraux2b*:

$\{a,b\} \in \text{set } l \implies \{c,d\} \in \text{set } l \implies \{a,b\} \neq \{c,d\} \implies y = \text{DenyAllFromTo } a \ b \implies$
 $\text{smaller } (\text{AllowPortFromTo } c \ d \ p) \ y \ l \implies$
 $\neg \text{smaller } y \ (\text{AllowPortFromTo } c \ d \ p) \ l$

apply *simp*

apply (*metis eq-imp-le pos-noteq*)

done

lemma *smalleraux2c*:

$\{a,b\} \in \text{set } l \implies \{c,d\} \in \text{set } l \implies \{a,b\} \neq \{c,d\} \implies y = \text{AllowPortFromTo } a \ b \ q \implies$
 $\text{smaller } (\text{AllowPortFromTo } c \ d \ p) \ y \ l \implies \neg \text{smaller } y \ (\text{AllowPortFromTo } c \ d \ p) \ l$

apply *simp*

apply (*metis pos-noteq*)

done

lemma *smalleraux3*:

assumes $x \in \text{set } l$

assumes $y \in \text{set } l$

assumes $x \neq y$

assumes $x = \text{bothNet } a$

assumes $y = \text{bothNet } b$

assumes $\text{smaller } a \ b \ l$

assumes *singleCombinators* [a]

assumes *singleCombinators* [b]

shows $\neg \text{smaller } b \ a \ l$

proof (*cases a*)

case *DenyAll* **thus** *?thesis* **using** *assms* **by** (*case-tac b, simp-all*)

next

case (*DenyAllFromTo c d*) **thus** *?thesis*

proof (*cases b*)

case *DenyAll* **thus** *?thesis* **using** *assms DenyAll DenyAllFromTo* **by** *simp*

next

case (*DenyAllFromTo e f*) **thus** *?thesis* **using** *assms DenyAllFromTo* **apply** *simp*

by (*metis DenyAllFromTo* $\langle a = \text{DenyAllFromTo } c \ d \rangle$ *bothNet.simps(2)* *smalleraux2*)

next

case (*AllowPortFromTo e f g*) **thus** *?thesis* **using** *assms DenyAllFromTo AllowPortFromTo*

apply *simp*

by (*metis eq-imp-le pos-noteq*)

next

case (*Conc e f*) **thus** *?thesis* **using** *assms* **by** *simp*

qed

next

case (*AllowPortFromTo c d p*) **thus** *?thesis*

proof (*cases b*)

case *DenyAll* **thus** *?thesis* **using** *assms AllowPortFromTo DenyAll* **by** *simp*

next

case (*DenyAllFromTo e f*) **thus** *?thesis* **using** *assms* **apply** *simp*


```

by (metis AllowPortFromTo DenyAllFromTo bothNet.simps(3) smalleraux2a)
  next
  case (AllowPortFromTo e f g) thus ?thesis using assms apply simp
by (metis AllowPortFromTo ⟨a = AllowPortFromTo c d p⟩ bothNet.simps(3) smalleraux2c)
  next
  case (Conc e f) thus ?thesis using assms by simp
qed
next
case (Conc c d) thus ?thesis using assms by simp
qed

```

lemma smalleraux3a:

$a \neq \text{DenyAll} \implies b \neq \text{DenyAll} \implies \text{in-list } b \ l \implies \text{in-list } a \ l \implies$
 $\text{bothNet } a \neq \text{bothNet } b \implies \text{smaller } a \ b \ l \implies \text{singleCombinators } [a] \implies$
 $\text{singleCombinators } [b] \implies \neg \text{smaller } b \ a \ l$

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]:

$\llbracket x \neq \text{DenyAll}; r \neq \text{DenyAll}; \text{singleCombinators } [x]; \text{singleCombinators } [y];$

$\text{singleCombinators } [r]; \neg \text{smaller } y \ x \ l; \text{smaller } x \ y \ l; \text{smaller } x \ r \ l;$

$\text{smaller } y \ r \ l; \text{in-list } x \ l; \text{in-list } y \ l; \text{in-list } r \ l \rrbracket \implies$

$\neg \text{smaller } r \ x \ l$

by (metis order-trans)

lemma notSmallerTrans[rule-format]:

$x \neq \text{DenyAll} \longrightarrow r \neq \text{DenyAll} \longrightarrow \text{singleCombinators } (x \# y \# z) \longrightarrow$

```

   $\neg$  smaller  $y\ x\ l \longrightarrow$  sorted  $(x\#y\#z)\ l \longrightarrow r \in \text{set } z \longrightarrow$ 
  all-in-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 remdups.simps(2) set-remdups
  singleCombinatorsConc singleCombinatorsStart)
apply metis
apply (metis sorted.simps(3) in-set-in-list singleCombinatorsConc
  singleCombinatorsStart sortedConcStart sorted-is-smaller)
apply (metis sorted-Cons all-in-list.simps(2)
  singleCombinatorsConc)
apply metis
apply (metis in-set-in-list)
done

lemma NCSaux1[rule-format]:
  noDenyAll  $p \longrightarrow \{x, y\} \in \text{set } l \longrightarrow$  all-in-list  $p\ l \longrightarrow$  singleCombinators  $p \longrightarrow$ 
  sorted  $(\text{DenyAllFromTo } x\ y\ \# p)\ l \longrightarrow \{x, y\} \neq \text{firstList } p \longrightarrow$ 
  DenyAllFromTo  $u\ v \in \text{set } p \longrightarrow \{x, y\} \neq \{u, v\}$ 
proof (cases p)
case Nil thus ?thesis by simp next
case (Cons a p) thus ?thesis using assms apply simp
  apply (rule impI)+
  apply (rule conjI)
  apply (metis bothNet.simps(2) first-bothNet.simps(3))
  apply (rule impI)
  apply (subgoal-tac smaller  $(\text{DenyAllFromTo } x\ y)\ (\text{DenyAllFromTo } u\ v)\ l)$ 
apply (subgoal-tac  $\neg$  smaller  $(\text{DenyAllFromTo } u\ v)\ (\text{DenyAllFromTo } x\ y)\ l)$ 
apply (rule notI)
apply (case-tac smaller  $(\text{DenyAllFromTo } u\ v)\ (\text{DenyAllFromTo } x\ y)\ l)$ 
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 metis
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)
done
qed

```

```

lemma posaux3[rule-format]:
   $a \in \text{set } l \longrightarrow b \in \text{set } l \longrightarrow a \neq b \longrightarrow \text{position } a\ l \neq \text{position } b\ l$ 
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  $[a] \longrightarrow a \neq \text{DenyAll} \longrightarrow$ 
   $b \neq \text{DenyAll} \longrightarrow \text{in-list } a\ l \longrightarrow \text{in-list } b\ l \longrightarrow$ 
   $\text{smaller } a\ b\ l \longrightarrow x = (\text{bothNet } a) \longrightarrow$ 
   $y = (\text{bothNet } b) \longrightarrow \text{position } x\ l \leq \text{position } y\ l$ 

```

```

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 assms DenyAllFromTo
  by (auto simp: eq-imp-le  $\langle a = \text{DenyAllFromTo } c\ d \rangle$ )
next
case (AllowPortFromTo  $e\ f\ p$ ) thus ?thesis using assms  $\langle a = \text{DenyAllFromTo } c\ d \rangle$  by simp
next
case (Conc  $e\ f$ ) thus ?thesis using assms Conc  $\langle a = \text{DenyAllFromTo } c\ d \rangle$  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 assms AllowPortFromTo by simp next
case (AllowPortFromTo  $e\ f\ p2$ ) thus ?thesis using assms  $\langle a = \text{AllowPortFromTo } c\ d\ p \rangle$ 
by simp next
case (Conc  $e\ f$ ) thus ?thesis using assms AllowPortFromTo by simp
qed

```

```

next
case (Conc c d) thus ?thesis by simp
qed

lemma NCSaux2[rule-format]:
  noDenyAll p  $\longrightarrow$   $\{a, b\} \in \text{set } l \longrightarrow \text{all-in-list } p \ l \longrightarrow \text{singleCombinators } p \longrightarrow$ 
  sorted (DenyAllFromTo a b # p) l  $\longrightarrow$   $\{a, b\} \neq \text{firstList } p \longrightarrow$ 
  AllowPortFromTo u v w  $\in \text{set } p \longrightarrow \{a, b\} \neq \{u, v\}$ 
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)
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)
apply (case-tac aa, simp-all)
done

lemma NCSaux3[rule-format]:
  noDenyAll p  $\longrightarrow$   $\{a, b\} \in \text{set } l \longrightarrow \text{all-in-list } p \text{ } l \longrightarrow \text{singleCombinators } p \longrightarrow$ 
  sorted (AllowPortFromTo a b w \# p) l  $\longrightarrow \{a, b\} \neq \text{firstList } p \longrightarrow$ 
  DenyAllFromTo u v  $\in \text{set } p \longrightarrow \{a, b\} \neq \{u, v\}$ 
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  $\neg$  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  $\neg$  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)
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  $\implies$  noDenyAll1 p  $\implies$  all-in-list p l  $\implies$ 
singleCombinators p  $\implies$  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]: ( $\forall a \in \text{set } z. \{b, c\} \neq \text{first-bothNet } a$ )  $\longrightarrow$ 
( $\forall a \in \text{set } (\text{separate } z). \{b, c\} \neq \text{first-bothNet } a$ )
apply (rule separate.induct) back
apply simp
apply (rule impI, simp)+
done

```

```

lemma fBNsepI[rule-format]: ( $\forall a \in \text{set } z. \text{first-bothNet } x \neq \text{first-bothNet } a$ )  $\longrightarrow$ 
( $\forall a \in \text{set } (\text{separate } z). \text{first-bothNet } x \neq \text{first-bothNet } a$ )
apply (rule separate.induct) back
apply simp
apply (rule impI, simp)+
done

```


lemma *NetsCollectedSepaux*:

$\llbracket \{b,c\} \neq \text{firstList } z; \text{noDenyAll1 } z;$
 $(\forall a \in \text{set } z. \{b,c\} \neq \text{first-bothNet } a); \text{NetsCollected } (z);$
 $\text{NetsCollected } (\text{separate } (z)); \{b,c\} \neq \text{firstList } (\text{separate } (z));$
 $a \in \text{set } (\text{separate } (z)) \rrbracket \implies$
 $\{b,c\} \neq \text{first-bothNet } a$

apply (*rule fBNsep*)

apply *simp-all*

done

lemma *NetsCollectedSepaux*:

$\llbracket \text{first-bothNet } (x::('a,'b)\text{Combinators}) \neq \text{first-bothNet } y; \neg \text{member DenyAll } y \wedge$
 $\text{noDenyAll } z;$
 $(\forall a \in \text{set } z. \text{first-bothNet } x \neq \text{first-bothNet } a) \wedge \text{NetsCollected } (y \# z);$
 $\text{NetsCollected } (\text{separate } (y \# z)); \text{first-bothNet } x \neq \text{firstList } (\text{separate } (y \# z));$
 $a \in \text{set } (\text{separate } (y \# z)) \rrbracket \implies$
 $\text{first-bothNet } (x::('a,'b)\text{Combinators}) \neq \text{first-bothNet } (a::('a,'b)\text{Combinators})$

apply (*rule fBNsep1*)

apply *simp-all*

apply *auto*

done

lemma *NetsCollectedSep*[*rule-format*]: $\text{noDenyAll1 } p \longrightarrow \text{NetsCollected } p \longrightarrow$
 $\text{NetsCollected } (\text{separate } p)$

apply (*rule separate.induct*) **back**

apply *simp-all*

apply (*metis fMTaux noDA noDA1eq noDAsep*)

apply (*rule conjI*|*rule impI*)**+**

apply *simp*

apply (*metis fBNsep set-ConsD*)

apply (*metis noDA1eq noDenyAll.simps*(1))

apply (*rule conjI*|*rule impI*)**+**

apply (*metis fBNsep set-ConsD*)

apply (*metis noDA1eq noDenyAll.simps*(1))

apply (*rule conjI*|*rule impI*)**+**

apply *simp*

apply (*metis NetsCollected.simps*(1) *NetsCollectedSepaux firstList.simps*(1) *fl2 fl3*
 $\text{noDA1eq noDenyAll.simps}$ (1))

apply (*metis noDA1eq noDenyAll.simps*(1))

done

lemma *OTNaux*:

$\text{onlyTwoNets } a \implies \neg \text{member DenyAll } a \implies (x,y) \in \text{sdnets } a \implies$
 $(x = \text{first-srcNet } a \wedge y = \text{first-destNet } a) \vee$
 $(x = \text{first-destNet } a \wedge y = \text{first-srcNet } a)$

apply (*case-tac* ($x = \text{first-srcNet } a \wedge y = \text{first-destNet } a$))

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  $a \implies \neg member\ DenyAll\ a \implies$ 
  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:
   $\llbracket onlyTwoNets\ a; onlyTwoNets\ aa; first-bothNet\ a \neq first-bothNet\ aa;$ 
   $\neg member\ DenyAll\ a; \neg member\ DenyAll\ aa \rrbracket$ 
   $\implies sdnets\ a \neq sdnets\ aa$ 
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:
   $\llbracket onlyTwoNets\ a; onlyTwoNets\ aa; first-bothNet\ a \neq first-bothNet\ aa;$ 

```

```

     $\neg \text{member } \text{DenyAll } a; \neg \text{member } \text{DenyAll } aa; \text{allNetsDistinct } [a, aa]] \implies$ 
     $\text{first-srcNet } a \neq \text{first-srcNet } aa \vee \text{first-srcNet } a \neq \text{first-destNet } aa \vee$ 
     $\text{first-destNet } a \neq \text{first-srcNet } aa \vee \text{first-destNet } a \neq \text{first-destNet } aa$ 
    apply (insert sdnets-chan [of a])
    apply (insert sdnets-chan [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;
     $\neg \text{member } \text{DenyAll } a; \neg \text{member } \text{DenyAll } aa; \text{allNetsDistinct } [a, aa]] \implies$ 
    disjSD-2 a aa
    apply (simp add: disjSD-2-def)
    apply (rule allI)+
    apply (rule impI)
    apply (insert sdnets-chan [of a])
    apply (insert sdnets-chan [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 alternativelistconc2)
    apply simp
    apply (subgoal-tac first-destNet a  $\neq$  first-destNet aa)
    apply (metis firstInNet alternativelistconc2)
    apply (metis first-bothNetsd)
    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)
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 alternativelistconc2)
apply simp
apply (subgoal-tac first-destNet a  $\neq$  first-destNet aa)
apply (metis firstInNet alternativelistconc2)
apply (metis first-bothNetsd)
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)
apply (metis)
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)
apply simp
apply (subgoal-tac first-destNet a  $\neq$  first-srcNet aa)
apply (metis firstInNeta firstInNet alternativelistconc2)
apply (metis first-bothNetsd 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 firstInNet alternativelistconc2)
apply simp

```

```

apply (subgoal-tac first-srcNet a  $\neq$  first-srcNet aa)
apply (metis firstInNeta alternativelistconc2)
apply (metis first-bothNetsd 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)
apply (subgoal-tac first-destNet a  $\neq$  first-srcNet aa)
apply (metis firstInNeta firstInNet alternativelistconc2)
apply (metis first-bothNetsd 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 insert-commute)
apply (case-tac (first-srcNet aa  $\neq$  (first-srcNet a)))
apply (metis firstInNeta alternativelistconc2)
apply simp
apply (case-tac first-destNet aa  $\neq$  first-destNet a)
apply (metis firstInNet alternativelistconc2)
apply simp
apply (metis first-bothNetsd)
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 insert-commute)
apply (case-tac (first-srcNet aa  $\neq$  (first-srcNet a)))
apply (metis firstInNeta alternativelistconc2)
apply simp
apply (case-tac first-destNet aa  $\neq$  first-destNet a)

```

```

apply (metis firstInNet alternativelistconc2)
apply simp
apply (metis first-bothNetsd)
apply (case-tac ( $c = \text{first-srcNet } aa \wedge d = \text{first-destNet } aa$ ))
apply simp-all
apply (case-tac ( $ab = \text{first-srcNet } a \wedge b = \text{first-destNet } a$ ))
apply simp-all
apply (case-tac ( $\text{first-destNet } a \neq \text{first-srcNet } aa$ ))
apply (metis firstInNeta firstInNet alternativelistconc2)
apply simp
apply (subgoal-tac  $\text{first-srcNet } a \neq \text{first-destNet } aa$ )
apply (metis firstInNeta firstInNet alternativelistconc2)
apply (metis insert-commute)
apply (case-tac ( $\text{first-srcNet } aa \neq \text{first-srcNet } a$ ))
apply (metis firstInNeta alternativelistconc2)
apply simp
apply (case-tac  $\text{first-destNet } aa \neq \text{first-destNet } a$ )
apply (metis firstInNet alternativelistconc2)
apply simp
apply (case-tac ( $ab = \text{first-srcNet } a \wedge b = \text{first-destNet } a$ ))
apply simp-all
apply (case-tac ( $\text{first-destNet } a \neq \text{first-srcNet } aa$ ))
apply (metis firstInNeta firstInNet alternativelistconc2)
apply simp
apply (subgoal-tac  $\text{first-srcNet } a \neq \text{first-srcNet } aa$ )
apply (metis firstInNeta firstInNet)
apply (metis)
apply (case-tac ( $\text{first-srcNet } aa \neq \text{first-destNet } a$ ))
apply (metis firstInNeta firstInNet alternativelistconc2)
apply simp
apply (case-tac  $\text{first-destNet } aa \neq \text{first-srcNet } a$ )
apply (metis firstInNeta firstInNet alternativelistconc2 )
apply simp
apply (metis insert-commute)
done

```

lemma *ANDaux3*[*rule-format*]: $y \in \text{set } xs \longrightarrow a \in \text{set } (\text{net-list-aux } [y]) \longrightarrow$
 $a \in \text{set } (\text{net-list-aux } xs)$

```

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)  $\implies$  y  $\in$  set xs
   $\implies$  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]:
 $\llbracket \neg \text{member DenyAll } a; \text{OnlyTwoNets } (a \# p); \text{NetsCollected2 } (a \# p);$ 
 $\text{NetsCollected } (a \# p); \text{noDenyAll } (p); \text{allNetsDistinct } (a \# p); s \in \text{set } p \rrbracket \implies$ 
 $\text{disjSD-2 } a \ s$ 
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 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 )
  apply (rule impI)+
  apply simp
  apply (drule mp)
  apply (erule 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

```

```

lemma rADnMT[rule-format]: p  $\neq \square \longrightarrow$  removeAllDuplicates p  $\neq \square$ 
apply (induct p)
apply simp-all
done

```

```

lemma remDupsNMT[rule-format]: p  $\neq \square \longrightarrow$ 
  remdups p  $\neq \square$ 
by (metis remdups-eq-nil-iff)

```

```

lemma sets-distinct1: (n::int)  $\neq m \implies \{(a,b). a = n\} \neq \{(a,b). a = m\}$ 
by auto

```

```

lemma sets-distinct2: (m::int)  $\neq n \implies \{(a,b). a = n\} \neq \{(a,b). a = m\}$ 
by auto

```

```

lemma sets-distinct5: (n::int)  $< m \implies \{(a,b). a = n\} \neq \{(a,b). a = m\}$ 
by auto

```

lemma *sets-distinct6*: $(m::int) < n \implies \{(a,b). a = n\} \neq \{(a,b). a = m\}$
by *auto*

end

theory *NormalisationIntegerPortProof*
imports *NormalisationGenericProofs*
begin

Normalisation proofs which are specific to the IntegerPort address representation.

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

lemma *aux26*[*simp*]: $twoNetsDistinct\ a\ b\ c\ d \implies$
 $dom\ (C\ (AllowPortFromTo\ a\ b\ p)) \cap dom\ (C\ (DenyAllFromTo\ c\ d)) = \{\}$
by (*auto simp: PLemmas twoNetsDistinct-def netsDistinct-def*) *auto*

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

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

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

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

lemma *MRList1*[*rule-format*]: $x \in dom\ (C\ a) \implies applied-rule-rev\ C\ x\ (b@[a]) = Some\ a$

by (*simp add: applied-rule-rev-def*)

lemma *MRList2*: $x \in \text{dom } (C \ a) \implies \text{applied-rule-rev } C \ x \ (c@b@[a]) = \text{Some } a$
by (*simp add: applied-rule-rev-def*)

lemma *MRList3*: $x \notin \text{dom } (C \ x a) \implies$
 $\text{applied-rule-rev } C \ x \ (a \ @ \ b \ \# \ xs \ @ \ [xa]) = \text{applied-rule-rev } C \ x \ (a \ @ \ b \ \# \ xs)$
by (*simp add: applied-rule-rev-def*)

lemma *CConcEnd*[*rule-format*]: $C \ a \ x = \text{Some } y \longrightarrow$
 $C \ (\text{list2FWpolicy } (xs \ @ \ [a])) \ x = \text{Some } y$
(is ?P xs)
apply (*rule-tac P = ?P in list2FWpolicy.induct*)
by (*simp-all add: C.simps*)

lemma *CConcStartaux*: $\llbracket C \ a \ x = \text{None} \rrbracket \implies (C \ aa \ ++ \ C \ a) \ x = C \ aa \ x$
by (*simp add: PLemmas*)

lemma *CConcStart*[*rule-format*]: $xs \neq [] \longrightarrow C \ a \ x = \text{None} \longrightarrow$
 $C \ (\text{list2FWpolicy } (xs \ @ \ [a])) \ x = C \ (\text{list2FWpolicy } xs) \ x$
apply (*rule list2FWpolicy.induct*)
by (*simp-all add: PLemmas*)

lemma *mrNnt*[*simp*]: $\text{applied-rule-rev } C \ x \ p = \text{Some } a \implies p \neq []$
apply (*simp add: applied-rule-rev-def*)
by *auto*

lemma *mr-is-C*[*rule-format*]: $\text{applied-rule-rev } C \ x \ p = \text{Some } a \longrightarrow$
 $C \ (\text{list2FWpolicy } (p)) \ x = C \ a \ x$
apply (*simp add: applied-rule-rev-def*)
apply (*rule rev-induct*)
apply *simp-all*
apply *safe*
apply (*metis CConcEnd rotate-simps*)
apply (*metis CConcEnd*)
by (*metis CConcStart applied-rule-rev-def mrNnt option.exhaust*)

lemma *CConcStart2*: $\llbracket p \neq []; x \notin \text{dom } (C \ a) \rrbracket \implies$
 $C \ (\text{list2FWpolicy } (p@[a])) \ x = C \ (\text{list2FWpolicy } p) \ x$
by (*erule CConcStart, simp add: PLemmas*)

lemma *CConcEnd1*: $\llbracket q@p \neq []; x \notin \text{dom } (C \ a) \rrbracket \implies$
 $C \ (\text{list2FWpolicy } (q@p@[a])) \ x = C \ (\text{list2FWpolicy } (q@p)) \ x$
apply (*subst lCdom2*)
by (*rule CConcStart2, simp-all*)

lemma *CConcEnd2*[*rule-format*]: $x \in \text{dom } (C \ a) \longrightarrow$
 $C \ (\text{list2FWpolicy } (xs \ @ \ [a])) \ x = C \ a \ x$

(is ?P xs)
apply (rule-tac P = ?P in list2FWpolicy.induct)
by (auto simp: C.simps)

lemma bar3: $x \in \text{dom } (C (\text{list2FWpolicy } (xs @ [xa]))) \implies$
 $x \in \text{dom } (C (\text{list2FWpolicy } xs)) \vee x \in \text{dom } (C xa)$
apply auto
by (metis CConcStart eq-Nil-appendI l2p-aux2 option.simps(3))

lemma CeqEnd[rule-format,simp]: $a \neq [] \longrightarrow x \in \text{dom } (C (\text{list2FWpolicy } a)) \longrightarrow$
 $C (\text{list2FWpolicy } (b@a)) x = (C (\text{list2FWpolicy } a)) x$
apply (rule rev-induct,simp-all)
apply (case-tac xs $\neq []$, simp-all)
apply (case-tac $x \in \text{dom } (C xa)$)
apply (metis CConcEnd2 MRLList2 mr-is-C rotate-simps)
apply (metis CConcEnd1 CConcStart2 Nil-is-append-conv bar3 rotate-simps)
apply (metis MRLList2 eq-Nil-appendI mr-is-C rotate-simps)
done

lemma CConcStartA[rule-format,simp]: $x \in \text{dom } (C a) \longrightarrow$
 $x \in \text{dom } (C (\text{list2FWpolicy } (a \# b)))$
(is ?P b)
apply (rule-tac P = ?P in list2FWpolicy.induct)
apply (simp-all add: C.simps)
done

lemma domConc: $\llbracket x \in \text{dom } (C (\text{list2FWpolicy } b)); b \neq [] \rrbracket \implies$
 $x \in \text{dom } (C (\text{list2FWpolicy } (a@b)))$
by (auto simp: PLemmas)

lemma CeqStart[rule-format,simp]:
 $x \notin \text{dom } (C (\text{list2FWpolicy } a)) \longrightarrow a \neq [] \longrightarrow b \neq [] \longrightarrow$
 $C (\text{list2FWpolicy } (b@a)) x = (C (\text{list2FWpolicy } b)) x$
apply (rule list2FWpolicy.induct,simp-all)
apply (auto simp: list2FWpolicyconc PLemmas)
done

lemma C-eq-if-mr-eq2: $\llbracket \text{applied-rule-rev } C x a = \text{Some } r; \text{applied-rule-rev } C x b = \text{Some } r;$
 $a \neq []; b \neq [] \rrbracket \implies$
 $(C (\text{list2FWpolicy } a)) x = (C (\text{list2FWpolicy } b)) x$
by (metis mr-is-C)

lemma nMRtoNone[rule-format]: $p \neq [] \longrightarrow \text{applied-rule-rev } C x p = \text{None} \longrightarrow$
 $C (\text{list2FWpolicy } p) x = \text{None}$
apply (rule rev-induct, simp-all)
apply (case-tac xs = [], simp-all)

by (*simp-all add: applied-rule-rev-def dom-def*)

lemma *C-eq-if-mr-eq*:

$\llbracket \text{applied-rule-rev } C \ x \ b = \text{applied-rule-rev } C \ x \ a; a \neq []; b \neq [] \rrbracket \implies$
 $(C \ (\text{list2FWpolicy } a)) \ x = (C \ (\text{list2FWpolicy } b)) \ x$

apply (*cases applied-rule-rev C 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 *notmatching-notdom*: *applied-rule-rev C x (p@[a]) ≠ Some a $\implies x \notin \text{dom } (C \ a)$*

by (*simp add: applied-rule-rev-def split: if-splits*)

lemma *foo3a[rule-format]*: *applied-rule-rev C x (a@[b]@c) = Some b $\longrightarrow r \in \text{set } c \longrightarrow$*
b $\notin \text{set } c \longrightarrow x \notin \text{dom } (C \ r)$

apply (*rule rev-induct*)

apply *simp-all*

apply (*rule impI|rule conjI|simp*)**+**

apply (*rule-tac p = a @ b # xs in notmatching-notdom, 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 notmatching-notdom, simp-all*)

done

lemma *foo3D*: $\llbracket \text{wellformed-policy1 } p; p = (\text{DenyAll} \# ps) \rrbracket$

applied-rule-rev C x p = Some DenyAll; r $\in \text{set } ps \rrbracket \implies x \notin \text{dom } (C \ r)$

by (*rule-tac a = [] and b = DenyAll and c = ps in foo3a, simp-all*)

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

by *simp*

lemma *foo5b[rule-format]*: *x $\in \text{dom } (C \ b) \longrightarrow (\forall \ r. r \in \text{set } c \longrightarrow x \notin \text{dom } (C \ r)) \longrightarrow$*
applied-rule-rev C x (b#c) = Some b

apply (*simp add: applied-rule-rev-def*)

apply (*rule-tac xs = c in rev-induct, simp-all*)

done

lemma *mr-first*: $\llbracket x \in \text{dom } (C \ b); (\forall \ r. r \in \text{set } c \longrightarrow x \notin \text{dom } (C \ r)); s = b \# c \rrbracket \implies$
applied-rule-rev C x s = Some b

by (*simp add: foo5b*)

lemma *mr-chaen[rule-format]*: *a $\in \text{set } p \longrightarrow (x \in \text{dom } (C \ a)) \longrightarrow$*

$$(\forall r. r \in \text{set } p \wedge x \in \text{dom } (C \ r) \longrightarrow r = a) \longrightarrow$$

$$\text{applied-rule-rev } C \ x \ p = \text{Some } a$$
apply (*rule-tac* $xs = p$ **in** *rev-induct*)
by (*simp-all* *add*: *applied-rule-rev-def*)

lemma *foo8*: $\llbracket (\forall r. r \in \text{set } p \wedge x \in \text{dom } (C \ r) \longrightarrow r = a); \text{set } p = \text{set } s \rrbracket \Longrightarrow$

$$(\forall r. r \in \text{set } s \wedge x \in \text{dom } (C \ r) \longrightarrow r = a)$$
by *auto*

lemma *mrConcEnd*[*rule-format*]: *applied-rule-rev* $C \ x \ (b \ \# \ p) = \text{Some } a \longrightarrow a \neq b \longrightarrow$

$$\text{applied-rule-rev } C \ x \ p = \text{Some } a$$
apply (*simp* *add*: *applied-rule-rev-def*)
apply (*rule-tac* $xs = p$ **in** *rev-induct*, *simp-all*)
by *auto*

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

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

lemma *foo98*[*rule-format*]: *applied-rule-rev* $C \ x \ (aa \ \# \ p) = \text{Some } a \longrightarrow x \in \text{dom } (C \ r) \longrightarrow$

$$r \in \text{set } p$$

$$\longrightarrow a \in \text{set } p$$
apply (*simp* *add*: *applied-rule-rev-def*)
apply (*rule* *rev-induct*)
apply *simp-all*
apply (*case-tac* $r = xa$, *simp-all*)
done

lemma *mrMTNone*[*simp*]: *applied-rule-rev* $C \ x \ [] = \text{None}$
by (*simp* *add*: *applied-rule-rev-def*)

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

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

lemma *mr-not-Conc*: *singleCombinators* $p \Longrightarrow \text{applied-rule-rev } C \ x \ p \neq \text{Some } (a \oplus b)$
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]: *applied-rule-rev* $C\ x\ p = \text{Some } a \longrightarrow x \in \text{dom } (C\ a)$
apply (*rule-tac* $xs = p$ **in** *rev-induct*)
by (*auto simp: applied-rule-rev-def*)

lemma *wp3EndMT*[rule-format]: *wellformed-policy3* ($p@[xs]$) \longrightarrow
 $\text{AllowPortFromTo } a\ b\ po \in \text{set } p \longrightarrow$
 $\text{dom } (C\ (\text{AllowPortFromTo } a\ b\ po)) \cap \text{dom } (C\ xs) = \{\}$
apply (*induct* p , *simp-all*)
apply (*rule impI*)
apply (*drule mp*)
apply (*erule wp3Conc*)
by *clarify auto*

lemma *foo29*: $\llbracket \text{dom } (C\ a) \neq \{\}; \text{dom } (C\ a) \cap \text{dom } (C\ b) = \{\} \rrbracket \Longrightarrow a \neq b$
by *auto*

lemma *foo28*: $\llbracket \text{AllowPortFromTo } a\ b\ po \in \text{set } p;$
 $\text{dom } (C\ (\text{AllowPortFromTo } a\ b\ po)) \neq \{\}; (\text{wellformed-policy3 } (p@[x])) \rrbracket$
 $\Longrightarrow x \neq \text{AllowPortFromTo } a\ b\ po$
by (*metis foo29 C.simps(3) wp3EndMT*)

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

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

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

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

lemma *wp1-axa*: *wellformed-policy1-strong* $p \Longrightarrow (\exists\ r. \text{applied-rule-rev } C\ x\ p = \text{Some } r)$

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

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

lemma *domTrans*: $\llbracket dom\ a \subseteq dom\ b; dom(b) \cap dom\ (c) = \{\} \rrbracket \implies dom(a) \cap dom(c) = \{\}$
by *auto*

lemma *DomInterAllowsMT*: $\llbracket twoNetsDistinct\ a\ b\ c\ d \rrbracket \implies$
 $dom\ (C\ (AllowPortFromTo\ a\ b\ p)) \cap dom\ (C\ (AllowPortFromTo\ c\ d\ po)) = \{\}$
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*: $\llbracket p \neq po \rrbracket \implies$
 $dom\ (C\ (AllowPortFromTo\ a\ b\ p)) \cap dom\ (C\ (AllowPortFromTo\ c\ d\ po)) = \{\}$
by (*simp add: twoNetsDistinct-def netsDistinct-def PLemmas*) *auto*

lemma *wellformed-policy3-chn*[*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 *DistinctNetsDenyAllow*:
 $\llbracket DenyAllFromTo\ b\ c \in set\ p; AllowPortFromTo\ a\ d\ po \in set\ p; allNetsDistinct\ p;$
 $dom\ (C\ (DenyAllFromTo\ b\ c)) \cap dom\ (C\ (AllowPortFromTo\ a\ d\ po)) \neq \{\} \rrbracket$


```

     $\implies b = a \wedge c = d$ 
  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:
   $\llbracket \text{AllowPortFromTo } b \ c \ po \in \text{set } p; \text{AllowPortFromTo } a \ d \ po \in \text{set } p; \\ \text{allNetsDistinct } p; \text{dom } (C \ (\text{AllowPortFromTo } b \ c \ po)) \cap \\ \text{dom } (C \ (\text{AllowPortFromTo } a \ d \ po)) \neq \{\} \rrbracket \\ \implies b = a \wedge c = d \wedge po = po$ 
  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]:
   $\llbracket \text{singleCombinators } p; \\ \text{distinct } p; \\ \text{allNetsDistinct } p \rrbracket \implies \\ \text{wellformed-policy2 } (\text{removeShadowRules2 } p)$ 
  proof (induct p)
    case Nil thus ?case by simp
  next
    case (Cons x xs)
    have wp-xs: wellformed-policy2 (removeShadowRules2 xs) using assms
  by (metis Cons ANDConc distinct.simps(2) singleCombinatorsConc)
    show ?case
    proof (cases x)
      case DenyAll thus ?thesis using wp-xs by simp
    next
      case (DenyAllFromTo a b) thus ?thesis
        using wp-xs Cons
        by (simp, metis DenyAllFromTo aux aux7 tNDComm deny-dom)
    next
      case (AllowPortFromTo a b p) thus ?thesis
        using assms wp-xs
        by (simp, metis aux26 AllowPortFromTo Cons(4) aux aux7a tNDComm)
    next

```

```

      case (Conc a b) thus ?thesis
        using assms by (metis Conc Cons(2) singleCombinators.simps(2))
    qed
  qed

```

```

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

```

```

lemma sorted-WP2[rule-format]:  $\text{sorted } p \ l \longrightarrow \text{all-in-list } p \ l \longrightarrow \text{distinct } p \longrightarrow$ 
   $\text{allNetsDistinct } p \longrightarrow \text{singleCombinators } p \longrightarrow \text{wellformed-policy2 } p$ 
proof (induct p)
  case Nil thus ?case by simp
next
  case (Cons a p) thus ?case
    proof (cases a)
      case DenyAll thus ?thesis using assms Cons
        by (auto intro: ANDConc singleCombinatorsConc sortedConcEnd)
    next
      case (DenyAllFromTo c d) thus ?thesis using assms Cons
        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 Cons assms
        apply simp
        apply (rule impI|rule conjI|rule allI)+
        apply (rule aux26)
        apply (rule-tac  $x = \text{AllowPortFromTo } c \ d \ e$  and
           $y = \text{DenyAllFromTo } aa \ b$  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 bothNet.simps(2) in-list.simps(2)
        in-set-in-list)
by (auto intro: aux7 tNDComm ANDConc singleCombinatorsConc sortedConcEnd)
next
case (Conc a b) thus ?thesis using Cons by simp
qed
qed

```

```

lemma wellformed2-sorted[simp]:  $\llbracket \text{all-in-list } p \text{ } l; \text{ distinct } p; \text{ allNetsDistinct } p; \text{ singleCombinators } p \rrbracket \implies \text{wellformed-policy2 } (\text{sort } p \text{ } l)$ 
apply (rule sorted-WP2)
apply (erule sort-is-sorted, simp-all)
apply (erule all-in-listSubset)
apply (auto intro: SC3 singleCombinatorsConc sorted-insort)
done

```

```

lemma wellformed2-sortedQ[simp]:  $\llbracket \text{all-in-list } p \text{ } l; \text{ distinct } p; \text{ allNetsDistinct } p; \text{ singleCombinators } p \rrbracket \implies \text{wellformed-policy2 } (q\text{sort } p \text{ } l)$ 
apply (rule sorted-WP2)
apply (erule sort-is-sortedQ, simp-all)
apply (erule all-in-listSubset)
apply (auto intro: SC3Q singleCombinatorsConc distinct-sortQ)
done

```

```

lemma C-DenyAll[simp]:  $C (\text{list2FWpolicy } (xs @ [\text{DenyAll}])) x = \text{Some } (\text{deny } ())$ 
by (auto simp: PLemmas)

```

```

lemma C-eq-RS1n:
   $C(\text{list2FWpolicy } (\text{removeShadowRules1-alternative } p)) = C(\text{list2FWpolicy } p)$ 
apply (case-tac p = [])
apply simp-all
apply (metis list2FWpolicy.simps(1) 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: PLemmas)
apply (rule-tac t = removeShadowRules1-alternative (xs @ [x]) and
         $s = (\text{removeShadowRules1-alternative } xs)@[x] \text{ in } \text{subst}$ )

```

```

apply (erule RS1n-assoc)
apply (case-tac  $xa \in \text{dom } (C \ x)$ )
apply simp-all
done

```

```

lemma C-eq-RS1[simp]:  $p \neq [] \implies$ 
 $C(\text{list2FWpolicy } (\text{removeShadowRules1 } p)) = C(\text{list2FWpolicy } p)$ 
by (metis rSR1-eq C-eq-RS1n)

```

```

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

```

```

lemma EX-MR :  $\llbracket \text{applied-rule-rev } C \ x \ p \neq (\text{Some } \text{DenyAll}); p = \text{DenyAll} \# ps \rrbracket \implies$ 
 $(\text{applied-rule-rev } C \ x \ p = \text{applied-rule-rev } C \ x \ ps)$ 
apply auto

```

```

apply (subgoal-tac applied-rule-rev C x (DenyAll#ps) ≠ None)
apply auto
apply (metis mrConcEnd)
by (metis DAimpliesMR-E hd.simps hd-in-set list.simps(3) not-Some-eq)

```

```

lemma mr-not-DA:
 $\llbracket \text{wellformed-policy1-strong } s; \text{applied-rule-rev } C \ x \ p = \text{Some } (\text{DenyAllFromTo } a \ ab);$ 
 $\text{set } p = \text{set } s \rrbracket \implies \text{applied-rule-rev } C \ x \ s \neq \text{Some } \text{DenyAll}$ 
apply (subst wp1n-tl, simp-all)
apply (subgoal-tac x ∈ dom (C (DenyAllFromTo a ab)))
apply (subgoal-tac DenyAllFromTo a ab ∈ 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 ∈ set p, simp)
apply (erule mrSet)
apply (erule mr-in-dom)
done

```

```

lemma domsMT-notND-DD:
 $\llbracket \text{dom } (C \ (\text{DenyAllFromTo } a \ b)) \cap \text{dom } (C \ (\text{DenyAllFromTo } c \ d)) \neq \{\} \rrbracket \implies$ 
 $\neg \text{netsDistinct } a \ c$ 
apply (erule contrapos-nn)
apply (simp add: C.simps)
apply (erule aux6)
apply (simp add: twoNetsDistinct-def)
done

```

lemma *domsMT-notND-DD2*:

$\llbracket \text{dom } (C \text{ (DenyAllFromTo } a \ b)) \cap \text{dom } (C \text{ (DenyAllFromTo } c \ d)) \neq \{\} \rrbracket \implies$
 $\neg \text{netsDistinct } b \ d$

apply (*erule contrapos-nn*)

apply (*simp add: C.simps*)

apply (*rule aux6*)

apply (*simp add: twoNetsDistinct-def*)

done

lemma *domsMT-notND-DD3*:

$\llbracket x \in \text{dom } (C \text{ (DenyAllFromTo } a \ b)); x \in \text{dom } (C \text{ (DenyAllFromTo } c \ d)) \rrbracket \implies$
 $\neg \text{netsDistinct } a \ c$

apply (*rule domsMT-notND-DD*)

apply *auto*

done

lemma *domsMT-notND-DD4*:

$\llbracket x \in \text{dom } (C \text{ (DenyAllFromTo } a \ b)); x \in \text{dom } (C \text{ (DenyAllFromTo } c \ d)) \rrbracket \implies$
 $\neg \text{netsDistinct } b \ d$

apply (*rule domsMT-notND-DD2*)

apply *auto*

done

lemma *NetsEq-if-sameP-DD*:

$\llbracket \text{allNetsDistinct } p; u \in \text{set } p; v \in \text{set } p; u = (\text{DenyAllFromTo } a \ b);$
 $v = (\text{DenyAllFromTo } c \ d); x \in \text{dom } (C \text{ (} u \text{)}); x \in \text{dom } (C \text{ (} v \text{)}) \rrbracket \implies$
 $a = c \wedge b = d$

apply (*simp add: allNetsDistinct-def*)

apply (*metis ND0aux1 ND0aux2 domsMT-notND-DD3 domsMT-notND-DD4 mem-def*)

done

lemma *rule-cha1*:

assumes *aND*: *allNetsDistinct p*

and *mr-is-allow*: *applied-rule-rev C x p = Some (AllowPortFromTo a b po)*

and *SC*: *singleCombinators p*

and *inp*: *r ∈ set p*

and *inDom*: *x ∈ dom (C r)*

shows $(r = \text{AllowPortFromTo } a \ b \ po \vee r = \text{DenyAllFromTo } a \ b \vee r = \text{DenyAll})$

proof (*cases r*)

case *DenyAll* **show** *?thesis* **by** (*metis DenyAll*)

next

case $(\text{DenyAllFromTo } x \ y)$ **show** *?thesis* **using** *Cons assms DenyAllFromTo*

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 set-empty2*)

next

```

case (AllowPortFromTo x y b) show ?thesis using assms AllowPortFromTo Cons
  apply (simp)
  apply (rule DistinctNetsAllowAllow)
  apply (simp-all)
  apply (metis mrSet)
  by (metis Int-iff mr-in-dom inSet-not-MT set-empty2)
next
  case (Conc x y) thus ?thesis using assms by (metis aux0-0)
qed

```

```

lemma noneMTsubset[rule-format]: noneMT C a  $\longrightarrow$  set b  $\subseteq$  set a  $\longrightarrow$  noneMT C b
apply (induct b,simp-all)
by (metis notMTnMT)

```

```

lemma nMTSort: noneMT C p  $\implies$  noneMT C (sort p l)
by (metis set-sort nMTeqSet)

```

```

lemma nMTSortQ: noneMT C p  $\implies$  noneMT C (qsort p l)
by (metis set-sortQ nMTeqSet)

```

```

lemma wp3char[rule-format]: noneMT C xs  $\wedge$  C (AllowPortFromTo a b po) = empty  $\wedge$ 
  wellformed-policy3 (xs @ [DenyAllFromTo a b])  $\longrightarrow$ 
  AllowPortFromTo a b po  $\notin$  set xs
apply (induct xs)
apply simp-all
by (metis domNMT wp3Conc)

```

```

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

```

```

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  $\in$  set p
and x-in-dom-allow:  $x \in \text{dom } (C \text{ (AllowPortFromTo c d po)})$ 
shows applied-rule-rev C x p = Some (AllowPortFromTo c d po)
proof (insert assms, induct p rule; rev-induct)
  case Nil show ?case using Nil by simp
next
case (snoc y ys) show ?case using snoc
  apply simp
  apply (case-tac y = (AllowPortFromTo c d po))
  apply (simp add: applied-rule-rev-def)
  apply simp-all
  apply (subgoal-tac ys  $\neq$  [])
  apply (subgoal-tac applied-rule-rev C x ys = Some (AllowPortFromTo c d po))
  defer 1
  apply (metis ANDConcEnd SCConcEnd WP1ConcEnd foo25)
  apply (metis inSet-not-MT)
  proof (cases y)
    case DenyAll thus ?thesis using DenyAll snoc
    apply simp
    by (metis DAnotTL DenyAll inSet-not-MT policy2list.simps(2))
    next
    case (DenyAllFromTo a b) thus ?thesis using snoc apply simp
    apply (simp-all add: applied-rule-rev-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 AllowPortFromTo snoc apply simp
    apply (simp-all add: applied-rule-rev-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 Conc snoc apply simp
    by (metis Conc aux0-0 in-set-conv-decomp)
  qed
qed

lemma rule-charn3:
   $\llbracket \text{wellformed-policy1 } p; \text{allNetsDistinct } p; \text{singleCombinators } p; \\ \text{wellformed-policy3 } p; \text{applied-rule-rev } C \text{ x p} = \text{Some } (\text{DenyAllFromTo } c \text{ d}); \\ \text{AllowPortFromTo } a \text{ b po} \in \text{set } p \rrbracket \implies x \notin \text{dom } (C \text{ (AllowPortFromTo } a \text{ b po)})$ 
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: applied-rule-rev C 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 DenyAll assms by simp
next
  case (DenyAllFromTo c d) thus ?thesis using assms 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 assms 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 assms apply simp
    by (metis Conc aux0-0)
qed

```

```

lemma foo31a:  $\llbracket (\forall r. r \in \text{set } p \wedge x \in \text{dom } (C r) \longrightarrow$ 
  (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)  $\rrbracket \implies$ 
  (r = AllowPortFromTo a b po  $\vee$  r = DenyAllFromTo a b  $\vee$  r = DenyAll)
by auto

```

```

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

```



```

lemma mrDA-tl:
  assumes mr-DA: applied-rule-rev C x p = Some DenyAll
  and wp1n: wellformed-policy1-strong p
  shows applied-rule-rev C 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:
   $\llbracket \text{wellformed-policy1-strong } p; \text{allNetsDistinct } p; \text{singleCombinators } p; \\ \text{wellformed-policy3 } p; \text{applied-rule-rev } C \ x \ p = \text{Some } (\text{DenyAllFromTo } a \ b); \\ r \in \text{set } (tl \ p); x \in \text{dom } (C \ r) \rrbracket \implies r = \text{DenyAllFromTo } a \ b$ 
  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(4) EX-MR option.inject)
  apply (metis wp1n-tl)
  done

lemma mrDenyAll-is-unique:
   $\llbracket \text{wellformed-policy1-strong } p; \text{applied-rule-rev } C \ x \ p = \text{Some } \text{DenyAll}; \\ r \in \text{set } (tl \ p) \rrbracket \implies x \notin \text{dom } (C \ r)$ 
  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 applied-rule-rev C x p = applied-rule-rev C x s
  proof (cases applied-rule-rev C x p)

  case None
    have DA: DenyAll ∈ set p using wp1-p by (auto simp: wp1-aux1aa)
    have notDA: DenyAll ∉ 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 sets-eq foo2
        wellformed-policy1-charn wp1-aux1aa wp1-eq wp1-p wp1-s)
  {
case DenyAll
  have mr-p-is-DenyAll: applied-rule-rev C x p = Some DenyAll
    by (simp add: DenyAll Some)
  hence x-notin-tl-p:  $\forall r. r \in \text{set } (tl\ p) \longrightarrow x \notin \text{dom } (C\ r)$  using wp1-p
    by (auto simp: mrDenyAll-is-unique)
  hence x-notin-tl-s:  $\forall r. r \in \text{set } (tl\ s) \longrightarrow x \notin \text{dom } (C\ r)$  using tl-eq
    by auto
  hence mr-s-is-DenyAll: applied-rule-rev C 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: applied-rule-rev C 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: applied-rule-rev C x p = applied-rule-rev C x (tl p)
    by (metis Combinators.simps(4) DenyAllFromTo Some mrConcEnd tl-p)
  have dom-tl-p:  $\bigwedge r. r \in \text{set } (tl\ p) \wedge x \in \text{dom } (C\ r) \implies$ 
    r = (DenyAllFromTo a b)
    using wp1-p aND SC wp3-p mr-p-is-DAFT
    by (auto simp: rule-charnDAFT)
  hence dom-tl-s:  $\bigwedge r. r \in \text{set } (tl\ s) \wedge x \in \text{dom } (C\ r) \implies$ 
    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:  $x \in \text{dom } (C\ (\text{DenyAllFromTo } a\ b))$ 
    by (metis mr-p-is-DAFT DenyAllFromTo mr-in-dom)
  hence mr-tl-s-is-DAFT: applied-rule-rev C 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: applied-rule-rev C x s = Some (DenyAllFromTo a b)
    using tl-s
    by (metis DA-notin-tl DenyAllFromTo EX-MR mrDA-tl
        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: applied-rule-rev C x p = Some (AllowPortFromTo a b c)

```

```

    by (simp add: AllowPortFromTo Some)
  hence A-in-s: AllowPortFromTo a b c ∈ set s using sets-eq
    by (auto intro: mrSet)
  have x-in-dom-A: x ∈ 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: aNDSetsEq)
  hence mr-s-is-A: applied-rule-rev C 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]] ==>
  C (list2FWpolicy p) x = C (list2FWpolicy s) x
  apply (rule C-eq-if-mr-eq)
  apply (rule C-eq-Sets-mr [symmetric])
  apply simp-all
done

```

lemma *C-eq-sorted*: [[distinct p; all-in-list p l; singleCombinators p;
wellformed-policy1-strong p; wellformed-policy3 p; allNetsDistinct p]] ==>
C (list2FWpolicy (sort p l)) = C (list2FWpolicy p)

```

  apply (rule ext)
  apply (rule C-eq-Sets)
  apply (auto simp: nMTSort wellformed1-alternative-sorted
    wellformed-policy3-charn wp1-eq)
done

```

lemma *C-eq-sortedQ*: [[distinct p; all-in-list p l; singleCombinators p;
wellformed-policy1-strong p; wellformed-policy3 p; allNetsDistinct p]] ==>
C (list2FWpolicy (qsort p l)) = C (list2FWpolicy p)

```

  apply (rule ext)
  apply (rule C-eq-Sets)
  apply (auto simp: nMTSortQ wellformed1-alternative-sorted distinct-sortQ
    wellformed-policy3-charn wp1-eq)
by (metis mem-def member-rec(1) member-set wellformed1-sortedQ wellformed-eq wp1-eq
  set-qsort wp1n-tl)

```

```

lemma C-eq-RS2-mr: applied-rule-rev C x (removeShadowRules2 p)= applied-rule-rev C 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 ∈ set ys)
  case True thus ?thesis using AllowPortFromTo Cons
    apply (cases applied-rule-rev C x ys = None, simp-all)
    apply (subgoal-tac x ∉ dom (C (AllowPortFromTo a b p)))
    apply (subst mrconcNone, simp-all)
    apply (simp add: applied-rule-rev-def)
    apply (rule contra-subsetD [OF allow-deny-dom])
    apply (erule mrNoneMT, simp)
    apply (metis AllowPortFromTo mrconc)
    done
  next
  case False thus ?thesis using False Cons AllowPortFromTo
    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 C-eq-None[rule-format]: p ≠ [] --> applied-rule-rev C x p = None →
  C (list2FWpolicy p) x = None
apply (simp add: applied-rule-rev-def)
apply (rule rev-induct, simp-all)
apply (rule impI)+
apply simp
apply (case-tac xs ≠ [])
apply (simp-all add: dom-def)
done

```

lemma *C-eq-None2:*

$\llbracket a \neq []; b \neq []; \text{applied-rule-rev } C \ x \ a = \text{None}; \text{applied-rule-rev } C \ x \ b = \text{None} \rrbracket \implies$
 $(C \ (\text{list2FWpolicy } a)) \ x = (C \ (\text{list2FWpolicy } b)) \ x$
by (auto simp: C-eq-None)

lemma C-eq-RS2: *wellformed-policy1-strong* $p \implies$
 $C \ (\text{list2FWpolicy } (\text{removeShadowRules2 } p)) = C \ (\text{list2FWpolicy } p)$
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 noneMTRS2: *noneMT* $C \ p \implies \text{noneMT } C \ (\text{removeShadowRules2 } p)$
by (auto simp: RS2Set noneMTsubset)

lemma CconcNone: $\llbracket \text{dom } (C \ a) = \{\}; p \neq [] \rrbracket \implies$
 $C \ (\text{list2FWpolicy } (a \ \# \ p)) \ x = C \ (\text{list2FWpolicy } p) \ x$
apply (case-tac $p = []$, simp-all)
apply (case-tac $x \in \text{dom } (C \ (\text{list2FWpolicy } (p)))$)
apply (metis Cdom2 list2FWpolicyconc)
apply (metis C.simps(4) map-add-dom-app-simps(2) inSet-not-MT list2FWpolicyconc set-empty2)
done

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

lemma DARS3[rule-format]: *DenyAll* $\notin \text{set } p \longrightarrow \text{DenyAll} \notin \text{set } (\text{removeShadowRules3 } C \ p)$
by (induct p , simp-all)

lemma DAnMT: $\text{dom } (C \ \text{DenyAll}) \neq \{\}$
by (simp add: dom-def C.simps PolicyCombinators.PolicyCombinators)

lemma DAnMT2: $C \ \text{DenyAll} \neq \text{empty}$
by (metis DAAux dom-eq-empty-conv empty-iff)

lemma wp1n-RS3[rule-format,simp]: *wellformed-policy1-strong* $p \longrightarrow$
 $\text{wellformed-policy1-strong } (\text{removeShadowRules3 } C \ p)$
apply (induct p , simp-all)
apply (rule conjI | rule impI | simp)+
apply (metis DAnMT)
apply (metis DARS3)
done

```

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

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

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

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

lemma nlpaux:  $x \notin \text{dom } (C\ b) \Longrightarrow C\ (a \oplus b)\ x = C\ a\ x$ 
by (metis  $C.\text{simps}(4)$  map-add-dom-app-simps(3))

lemma notindom[rule-format]:  $a \in \text{set } p \longrightarrow x \notin \text{dom } (C\ (\text{list2FWpolicy } p)) \longrightarrow$ 
                                      $x \notin \text{dom } (C\ a)$ 
apply (induct  $p$ )
apply simp-all
apply (rule conjI | rule impI) +
apply (metis CConcStartA)
apply (rule impI) +
apply simp
apply (metis  $C.\text{dom2}$   $\text{List.set.simps}(2)$  domIff insert-absorb  $\text{list.simps}(2)$  list2FWpolicyconc
set-empty)
done

lemma C-eq-rd[rule-format]:  $p \neq [] \Longrightarrow$ 
                                      $C\ (\text{list2FWpolicy } (\text{remdups } p)) = C\ (\text{list2FWpolicy } p)$ 
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 Cons apply simp
        apply (rule conjI, rule impI)
        apply (cases  $x \in \text{dom } (C\ (\text{list2FWpolicy } ys))$ )
        apply (metis  $C.\text{dom2}$  False list2FWpolicyconc)
    qed

```

```

    apply (metis False domIff list2FWpolicyconc nlpaux notindom)
    apply (rule impI)
    apply (cases x ∈ dom (C (list2FWpolicy ys)))
    apply (subgoal-tac x ∈ dom (C (list2FWpolicy (remdups ys))))
    apply (metis Cdom2 False list2FWpolicyconc remdups-eq-nil-iff)
    apply (metis domIff)
    apply (subgoal-tac x ∉ dom (C (list2FWpolicy (remdups ys))))
    apply (metis False list2FWpolicyconc nlpaux remdups-eq-nil-iff)
    apply (metis domIff)
  done
qed
qed

lemma nMT-domMT:  $\llbracket \neg \text{notMTpolicy } C \ p; p \neq [] \rrbracket \implies r \notin \text{dom } (C \ (\text{list2FWpolicy } p))$ 
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 )
  by (metis CconcNone domIff)
qed

lemma C-eq-RS3-aux[rule-format]:  $\text{notMTpolicy } C \ p \implies$ 

$$C \ (\text{list2FWpolicy } p) \ x = C \ (\text{list2FWpolicy } (\text{removeShadowRules3 } C \ p)) \ x$$

proof (induct p)
  case Nil thus ?case by simp next
  case (Cons y ys) thus ?case
    proof (cases notMTpolicy C ys)
      case True thus ?thesis using Cons apply simp
        apply (rule conjI, rule impI, simp)
        apply (metis CconcNone True notMTpolicyimpnotMT)
        apply (rule impI, simp)
        apply (cases x ∈ dom (C (list2FWpolicy ys)))
        apply (subgoal-tac x ∈ dom (C (list2FWpolicy (removeShadowRules3 C ys))))
        apply (metis Cdom2 NMPRS3 True l2p-aux notMTpolicyimpnotMT)
        apply (simp add: domIff)
        apply (subgoal-tac x ∉ dom (C (list2FWpolicy (removeShadowRules3 C ys))))
        apply (metis l2p-aux l2p-aux2 nlpaux)
        apply (metis domIff)
      done
    next
    case False thus ?thesis using Cons False
      proof (cases ys = [])
        case True thus ?thesis using Cons by (simp) (rule impI, simp) next
        case False thus ?thesis using Cons False  $\langle \neg \text{notMTpolicy } C \ ys \rangle$ 
          apply (simp)
      done
    next
  done

```

```

    apply (rule conjI | rule impI | simp)+
    apply (subgoal-tac removeShadowRules3 C ys = [])
    apply (subgoal-tac x ∉ dom (C (list2FWpolicy ys)))
    apply simp-all
    apply (metis l2p-aux nlpaux)
    apply (erule nMT-domMT, simp-all)
    by (metis SR3nMT)
  qed
qed
qed

```

```

lemma C-eq-id: wellformed-policy1-strong p ⇒
  C(list2FWpolicy (insertDeny p)) = C (list2FWpolicy p)
apply (rule ext)
apply (rule C-eq-if-mr-eq)
apply simp-all
apply (erule mr-iD)
done

```

```

lemma C-eq-RS3: notMTpolicy C p ⇒
  C(list2FWpolicy (removeShadowRules3 C p)) = C (list2FWpolicy p)
apply (rule ext)
by (erule C-eq-RS3-aux[symmetric])

```

```

lemma NMPrd[rule-format]: notMTpolicy C p ⟶ notMTpolicy C (remdups p)
apply (induct p, simp-all)
by (auto simp: NMPcham)

```

```

lemma NMPDA[rule-format]: DenyAll ∈ set p ⟶ notMTpolicy C p
by (induct p, simp-all add: DAnMT)

```

```

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

```

```

lemma list2FWpolicy2list[rule-format]: C (list2FWpolicy(policy2list p)) = (C p)
apply (rule ext)
apply (induct-tac p, simp-all)
apply (case-tac x ∈ dom (C (Combinators2)))

```



```

apply (metis Cdom2 CeqEnd domIff p2lNmt)
apply (metis CeqStart domIff p2lNmt nlpaux)
done

```

```

lemmas C-eq-Lemmas = noneMTRS2 noneMTrd SCp2l
      wp1n-RS2 wp1ID NMPiD wp1alternative-RS1
      p2lNmt list2FWpolicy2list wellformed-policy3-charn waux2 wp1-eq

```

```

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

```

```

lemma C-eq-All-untilSorted:
  [[DenyAll ∈ set (policy2list p); all-in-list (policy2list p) l;
    allNetsDistinct (policy2list p)]] ⇒
  C(list2FWpolicy (sort (removeShadowRules2 (remdups (removeShadowRules3 C
    (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-untilSortedQ:
  [[DenyAll ∈ set (policy2list p); all-in-list (policy2list p) l;
    allNetsDistinct (policy2list p)]] ⇒
  C(list2FWpolicy (qsort (removeShadowRules2 (remdups (removeShadowRules3 C
    (insertDeny (removeShadowRules1 (policy2list p)))))) l)) = C p
apply (subst C-eq-sortedQ)
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*:
 $\llbracket \text{DenyAll} \in \text{set } (\text{policy2list } p); \text{all-in-list } (\text{policy2list } p) \text{ } l; \text{allNetsDistinct } (\text{policy2list } p) \rrbracket \implies$
 $C(\text{list2FWpolicy } (\text{sort } (\text{removeShadowRules2 } (\text{remdups } (\text{removeShadowRules3 } C (\text{insertDeny } (\text{removeShadowRules1 } (\text{policy2list } p)))))) \text{ } l)) = C \text{ } p$
by (*simp-all add: C-eq-Lemmas C-eq-subst-Lemmas*)

lemma *C-eq-All-untilSorted-withSimpsQ*:
 $\llbracket \text{DenyAll} \in \text{set } (\text{policy2list } p); \text{all-in-list } (\text{policy2list } p) \text{ } l; \text{allNetsDistinct } (\text{policy2list } p) \rrbracket \implies$
 $C(\text{list2FWpolicy } (\text{qsort } (\text{removeShadowRules2 } (\text{remdups } (\text{removeShadowRules3 } C (\text{insertDeny } (\text{removeShadowRules1 } (\text{policy2list } p)))))) \text{ } l)) = C \text{ } p$
by (*simp-all add: C-eq-Lemmas C-eq-subst-Lemmas*)

lemma *InDomConc[rule-format]*: $p \neq [] \longrightarrow x \in \text{dom } (C (\text{list2FWpolicy } (p))) \longrightarrow$
 $x \in \text{dom } (C (\text{list2FWpolicy } (a\#p)))$
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]*: $\text{member } a \text{ } b \longrightarrow x \notin \text{dom } (C \text{ } b) \longrightarrow x \notin \text{dom } (C \text{ } a)$
apply (*induct b*)
apply (*simp-all add: dom-def C.simps*)
done

lemma *src-in-sdnets[rule-format]*: $\neg \text{member DenyAll } x \longrightarrow p \in \text{dom } (C \text{ } x) \longrightarrow$
 $\text{subnetsOfAdr } (\text{src } p) \cap (\text{fst-set } (\text{sdnets } x)) \neq \{\}$
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]:  $\neg \text{member DenyAll } x \longrightarrow p \in \text{dom } (C \ x) \longrightarrow$ 
 $\text{subnetsOfAdr } (\text{dest } p) \cap (\text{snd-set } (\text{sdnets } x)) \neq \{\}$ 
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  $p \in \text{dom } (C \ \text{Combinators2})$ )
apply simp-all
apply (rule subnetAux)
apply assumption
apply (auto simp: PLemmas)
done

```

```

lemma sdnets-in-subnets[rule-format]:  $p \in \text{dom } (C \ x) \longrightarrow \neg \text{member DenyAll } x \longrightarrow$ 
 $(\exists (a,b) \in \text{sdnets } x. a \in \text{subnetsOfAdr } (\text{src } p) \wedge b \in \text{subnetsOfAdr } (\text{dest } p))$ 
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  $p \in \text{dom } (C \ (\text{Combinators2}))$ )
apply simp-all
apply (auto simp: PLemmas subnetsOfAdr-def)
done

```

```

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

```

```

lemma list2FWpolicy-eq:  $zs \neq [] \Longrightarrow$ 
 $C \ (\text{list2FWpolicy } (x \oplus y \ \# \ z)) \ p = C \ (x \oplus \text{list2FWpolicy } (y \ \# \ z)) \ p$ 
by (metis ConcAssoc l2p-aux list2FWpolicy.simps(2))

```

```

lemma dom-sep[rule-format]:  $x \in \text{dom } (C \ (\text{list2FWpolicy } p)) \longrightarrow$ 
 $x \in \text{dom } (C \ (\text{list2FWpolicy } (\text{separate } p)))$ 
apply (rule separate.induct) back
apply simp-all
apply (rule conjI)
apply (rule impI)+

```

```

apply simp
apply (thin-tac False  $\implies$  ?S)
apply (drule mp)
apply (case-tac  $x \in \text{dom } (C \text{ (DenyAllFromTo } v \text{ } va)))$ )
apply (metis CConcStartA domIff l2p-aux2
      list2FWpolicyconc not-Cons-self )
apply (subgoal-tac  $x \in \text{dom } (C \text{ (list2FWpolicy (y \#z))))$ )
apply (metis CConcStartA Cdom2 InDomConc domIff l2p-aux2 list2FWpolicyconc nlpaux)
apply (subgoal-tac  $x \in \text{dom } (C \text{ (list2FWpolicy ((DenyAllFromTo } v \text{ } va)\#y\#z))))$ )
apply (simp add: dom-def C.simps)
apply simp
apply simp
apply (rule impI) +
apply simp
apply (thin-tac False  $\implies$  ?S)
apply (case-tac  $x \in \text{dom } (C \text{ (DenyAllFromTo } v \text{ } va)))$ )
apply simp-all
apply (subgoal-tac  $x \in \text{dom } (C \text{ (list2FWpolicy (y \#z))))$ )
apply (metis InDomConc sepnMT list.simps(2))
apply (subgoal-tac  $x \in \text{dom } (C \text{ (list2FWpolicy ((DenyAllFromTo } v \text{ } va)\#y\#z))))$ )
apply (simp add: dom-def C.simps)
apply simp
apply (rule impI | rule conjI) +
apply simp
apply (case-tac  $x \in \text{dom } (C \text{ (AllowPortFromTo } v \text{ } va \text{ } vb)))$ )
apply (metis CConcStartA domIff l2p-aux2
      list2FWpolicyconc not-Cons-self )
apply (subgoal-tac  $x \in \text{dom } (C \text{ (list2FWpolicy (y \#z))))$ )
apply simp
apply (metis CConcStartA Cdom2 InDomConc domIff l2p-aux2 list2FWpolicyconc nlpaux)
apply (simp add: dom-def C.simps)
apply (rule impI) +
apply simp-all
apply (case-tac  $x \in \text{dom } (C \text{ (AllowPortFromTo } v \text{ } va \text{ } vb)))$ )
apply simp-all
apply (metis Cdom2 domIff l2p-aux list2FWpolicy.simps(3) nlpaux sepnMT)
apply (rule conjI | rule impI) +
apply simp
apply (thin-tac False  $\implies$  ?S)
apply (drule mp)
apply (case-tac  $x \in \text{dom } (C \text{ ((v } \oplus \text{ } va))))$ )
apply (metis C.simps(4) CConcStartA ConcAssoc domIff
      list2FWpolicy2list list2FWpolicyconc p2lNmt)
defer 1
apply simp-all
apply (rule impI) +
apply simp
apply (thin-tac False  $\implies$  ?S)
apply (case-tac  $x \in \text{dom } (C \text{ ((v } \oplus \text{ } va))))$ )

```

```

apply (metis CConcStartA)
apply (drule mp)
apply (simp add: C.simps dom-def)
apply (metis InDomConc list.simps(2)sepnMT)
apply (subgoal-tac  $x \in \text{dom } (C (\text{list2FWpolicy } (y \# z)))$ )
apply (case-tac  $x \in \text{dom } (C y)$ )
apply simp-all
apply (metis CConcStartA Cdom2 ConcAssoc domIff)
apply (metis InDomConc domIff l2p-aux2 list2FWpolicyconc nlpaux)
apply (case-tac  $x \in \text{dom } (C y)$ )
apply simp-all
apply (metis domIff nlpaux)
done

lemma domdConcStart[rule-format]:  $x \in \text{dom } (C (\text{list2FWpolicy } (a \# b))) \longrightarrow$ 
 $x \notin \text{dom } (C (\text{list2FWpolicy } b))$ 
 $\longrightarrow x \in \text{dom } (C (a))$ 
apply (induct b, simp-all)
apply (auto simp: PLemmas)
done

lemma sep-dom2-aux:  $\llbracket x \in \text{dom } (C (\text{list2FWpolicy } (a \oplus y \# z))) \rrbracket$ 
 $\implies x \in \text{dom } (C (a \oplus \text{list2FWpolicy } (y \# z)))$ 
apply auto
by (metis list2FWpolicy-eq p2lNmt)

lemma sep-dom2-aux2:
 $\llbracket (x \in \text{dom } (C (\text{list2FWpolicy } (\text{separate } (y \# z)))) \longrightarrow$ 
 $x \in \text{dom } (C (\text{list2FWpolicy } (y \# z)));$ 
 $x \in \text{dom } (C (\text{list2FWpolicy } (a \# \text{separate } (y \# z)))) \rrbracket$ 
 $\implies x \in \text{dom } (C (\text{list2FWpolicy } (a \oplus y \# z)))$ 
by (metis CConcStartA InDomConc domdConcStart list.simps(2) list2FWpolicy.simps(2) list2FWpolicyconc)

lemma sep-dom2[rule-format]:
 $x \in \text{dom } (C (\text{list2FWpolicy } (\text{separate } p))) \longrightarrow x \in \text{dom } (C (\text{list2FWpolicy } (p)))$ 
apply (rule separate.induct)
by (simp-all add: sep-dom2-aux sep-dom2-aux2)

lemma sepDom:  $\text{dom } (C (\text{list2FWpolicy } p)) = \text{dom } (C (\text{list2FWpolicy } (\text{separate } p)))$ 
apply (rule equalityI)
by (rule subsetI, (erule dom-sep|erule sep-dom2))+

lemma C-eq-s-ext[rule-format]:  $p \neq [] \longrightarrow$ 
 $C (\text{list2FWpolicy } (\text{separate } p)) a = C (\text{list2FWpolicy } p) a$ 
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 ∈ dom (C (list2FWpolicy x)))
apply (subgoal-tac a ∈ dom (C (list2FWpolicy (separate x))))
apply (metis Cdom2 list2FWpolicyconc sepDom sepnMT)
apply (metis sepDom)
apply (subgoal-tac a ∉ dom (C (list2FWpolicy (separate x))))
apply (subst list2FWpolicyconc)
apply (simp add: sepnMT)
apply (subst list2FWpolicyconc)
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 list2FWpolicyconc)
  apply (metis not-Cons-self sepnMT)
  apply (metis C.simps(4) CConcStartaux Cdom2 domIff)
  apply (rule conjI|rule impI|simp)+
  apply (erule list2FWpolicy-eq)
  apply (rule impI, simp)
  apply (subst list2FWpolicyconc)
  apply (metis list.simps(2) sepnMT)
  by (metis C.simps(4) CConcStartaux Cdom2 domIff)
next
case goal3 thus ?case
apply simp
  apply (cases z = [])
  apply simp-all
  apply (rule conjI|rule impI|simp)+
  apply (subst list2FWpolicyconc)
  apply (metis not-Cons-self sepnMT)
  apply (metis C.simps(4) CConcStartaux Cdom2 domIff)
  apply (rule conjI|rule impI|simp)+
  apply (erule list2FWpolicy-eq)
  apply (rule impI, simp)
  apply (subst list2FWpolicyconc)
  apply (metis list.simps(2) sepnMT)
  by (metis C.simps(4) CConcStartaux Cdom2 domIff)
next
case goal4 thus ?case
apply simp
  apply (cases z = [])

```

```

apply simp-all
apply (rule conjI|rule impI|simp)+
apply (subst list2FWpolicyconc)
  apply (metis not-Cons-self sepnMT)
apply (metis C.simps(4) CConcStartaux Cdom2 domIff)
apply (rule conjI|rule impI|simp)+
apply (erule list2FWpolicy-eq)
apply (rule impI, simp)
apply (subst list2FWpolicyconc)
apply (metis list.simps(2) sepnMT)
by (metis C.simps(4) CConcStartaux Cdom2 domIff)
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:  $p \neq [] \implies C (list2FWpolicy (separate p)) = C (list2FWpolicy p)$ 
apply (rule ext)
apply (rule C-eq-s-ext)
apply simp
done

lemma sortnMTQ:  $p \neq [] \implies qsort p l \neq []$ 
by (metis set-sortQ setnMT)

lemmas C-eq-Lemmas-sep =
  C-eq-Lemmas sortnMT sortnMTQ RS2-NMT NMPrd NMPRS3

lemma C-eq-until-separated:
 $\llbracket DenyAll \in set (policy2list p); all-in-list (policy2list p) l; allNetsDistinct (policy2list p) \rrbracket \implies$ 
 $C (list2FWpolicy (separate (sort (removeShadowRules2 (remdups (removeShadowRules3 C (insertDeny (removeShadowRules1 (policy2list p)))))) l))) = C p$ 
apply (subst C-eq-s)
apply (simp-all add: C-eq-Lemmas-sep)
apply (rule C-eq-All-untilSorted)
apply simp-all
done

lemma C-eq-until-separatedQ:
 $\llbracket DenyAll \in set (policy2list p); all-in-list (policy2list p) l; allNetsDistinct (policy2list p) \rrbracket \implies$ 
 $C (list2FWpolicy (separate (qsort (removeShadowRules2 (remdups (removeShadowRules3 C (insertDeny (removeShadowRules1 (policy2list p)))))) l))) = C p$ 

```

```

apply (subst C-eq-s)
apply (simp-all add: C-eq-Lemmas-sep)
apply (rule C-eq-All-untilSortedQ)
apply simp-all
done

```

```

lemma domID[rule-format]:  $p \neq [] \wedge x \in \text{dom}(C(\text{list2FWpolicy } p)) \longrightarrow$ 
 $x \in \text{dom}(C(\text{list2FWpolicy}(\text{insertDenies } p)))$ 
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-all add: C.simps dom-def)+
        by auto
    next
      case goal2 then show ?case
        proof(cases  $x \in \text{dom}(C(\text{list2FWpolicy } p))$ )
          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  $x \in \text{dom}(C(\text{list2FWpolicy}(\text{insertDenies } p)))$ )
                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 list2FWpolicyconc nlpaux Cdom2)
    next
      case (AllowPortFromTo src dest port) then show ?case
    by(simp,metis domIff CConcStartA list2FWpolicyconc nlpaux Cdom2)
    next
      case (Conc -) then show ?case
    apply simp
  by (metis CConcStartA Cdom2 Conc(5) ConcAssoc domIff domdConcStart goal2(2))
    qed
  qed
qed
qed
qed

```

lemma *DA-is-deny*:

```

   $x \in \text{dom } (C (\text{DenyAllFromTo } a \ b \oplus \text{DenyAllFromTo } b \ a \oplus \text{DenyAllFromTo } a \ b)) \implies$ 
   $C (\text{DenyAllFromTo } a \ b \oplus \text{DenyAllFromTo } b \ a \oplus \text{DenyAllFromTo } a \ b) \ x = \text{Some } (\text{deny } ())$ 
  apply (case-tac  $x \in \text{dom } (C (\text{DenyAllFromTo } a \ b))$ )
  apply (simp-all add: PLemmas)
  apply (simp-all split: if-splits)
done

```

lemma *iDdomAux*[rule-format]:

```

   $p \neq [] \longrightarrow x \notin \text{dom } (C (\text{list2FWpolicy } p)) \longrightarrow$ 
   $x \in \text{dom } (C (\text{list2FWpolicy } (\text{insertDenies } p))) \longrightarrow$ 
   $C (\text{list2FWpolicy } (\text{insertDenies } p)) \ x = \text{Some } (\text{deny } ())$ 
  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 DenyAllFromTo Cons
        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  $x \in \text{dom } (C (\text{list2FWpolicy } (\text{insertDenies } ys)))$ )
            apply simp-all
            apply (metis Cdom2 DenyAllFromTo goal2(5) idNMT list2FWpolicyconc)
            apply (subgoal-tac  $C (\text{list2FWpolicy } (\text{DenyAllFromTo } a \ b \oplus$ 

```

```

    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 list2FWpolicyconc nlpaux)
  done
qed
next
case (AllowPortFromTo a b c) then show ?thesis using Cons AllowPortFromTo
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 (list2FWpolicy (insertDenies ys))))
  apply simp-all
  apply (metis AllowPortFromTo Cdom2 goal2(4) idNMT list2FWpolicyconc)
  apply (subgoal-tac C (list2FWpolicy (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
    list2FWpolicyconc 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 ∈ dom (C (list2FWpolicy (insertDenies ys))))
apply (metis Cdom2 Conc Cons InDomConc goal2(2) idNMT list2FWpolicyconc)
apply (subgoal-tac C (list2FWpolicy (DenyAllFromTo (first-srcNet a)
(first-destNet a) ⊕ DenyAllFromTo (first-destNet a) (first-srcNet a)
⊕ a ⊕ b#insertDenies ys)) x =
C ((DenyAllFromTo (first-srcNet a) (first-destNet a) ⊕
DenyAllFromTo (first-destNet a) (first-srcNet a) ⊕ a ⊕ b)) x )
apply simp
defer 1
apply (metis Conc l2p-aux2 list2FWpolicyconc nlpaux)
apply (subgoal-tac C ((DenyAllFromTo (first-srcNet a)
(first-destNet a) ⊕ DenyAllFromTo (first-destNet a)
(first-srcNet a) ⊕ a ⊕ b)) x = C ((DenyAllFromTo (first-srcNet a)
(first-destNet a) ⊕ 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 ≠ [] ⟶ x ∉ dom (C (list2FWpolicy p))
  ⟶ C (DenyAll ⊕ list2FWpolicy (insertDenies p)) x = C DenyAll x
apply (case-tac x ∈ dom (C (list2FWpolicy (insertDenies p))))
apply (rule impI)+
apply (metis C.simps(1) deny-all-def iDdomAux Cdom2)
apply (rule impI)+
apply (subst nlpaux)
apply simp-all
done

lemma inDomConc: [ x ∉ dom (C a); x ∉ dom (C (list2FWpolicy p)) ] ⟹
  x ∉ dom (C (list2FWpolicy(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 (list2FWpolicy 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:
   $\llbracket p \neq []; \text{noDenyAll } (a \# p); \text{separated } (a \# p);$ 
   $x \in \text{dom } (C (\text{DenyAllFromTo } (\text{first-srcNet } a) (\text{first-destNet } a) \oplus$ 
     $\text{DenyAllFromTo } (\text{first-destNet } a) (\text{first-srcNet } a) \oplus a)) \rrbracket \implies$ 
   $x \notin \text{dom } (C (\text{list2FWpolicy } p))$ 
apply (rule-tac  $A = (\text{DenyAllFromTo } (\text{first-srcNet } a) (\text{first-destNet } a) \oplus$ 
   $\text{DenyAllFromTo } (\text{first-destNet } a) (\text{first-srcNet } a) \oplus a)$  in domsdisj)
apply simp-all
apply (rule notI)
apply (rule-tac  $p = xa$  and  $x = (\text{DenyAllFromTo } (\text{first-srcNet } a) (\text{first-destNet } a)$ 
   $\oplus \text{DenyAllFromTo } (\text{first-destNet } a) (\text{first-srcNet } a) \oplus a)$  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 noneMTsep[rule-format]: noneMT  $C \ p \longrightarrow \text{noneMT } C \ (\text{separate } p)$ 
apply (rule separate.induct) back
apply (simp-all add:  $C.\text{simps}$  map-add-le-mapE map-le-antisym)
done

```

```

lemma dom-id:
   $\llbracket \text{noDenyAll } (a \# p); \text{separated } (a \# p); p \neq []; x \notin \text{dom } (C (\text{list2FWpolicy } p));$ 
   $x \in \text{dom } (C (a)) \rrbracket$ 
   $\implies x \notin \text{dom } (C (\text{list2FWpolicy } (\text{insertDenies } p)))$ 
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  $\in \text{dom } (C \text{ (list2FWpolicy } p)) \longrightarrow$ 
  C(list2FWpolicy (insertDenies p)) x = C(list2FWpolicy p) x
proof (induct p)
case Nil thus ?case by simp
next
case (Cons y ys) thus ?case using Cons
proof (cases y)
case DenyAll thus ?thesis using Cons DenyAll apply simp
  apply (case-tac ys = [])
  apply simp-all
  apply (case-tac x  $\in \text{dom } (C \text{ (list2FWpolicy } ys))$ )
  apply simp-all
apply (metis Cdom2 domID idNMT list2FWpolicyconc noDA1eq)
apply (metis DenyAll iD-isD idNMT list2FWpolicyconc nlpaux)
done
next
case (DenyAllFromTo a b) thus ?thesis using Cons 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  $\in \text{dom } (C \text{ (list2FWpolicy } ys))$ )
  apply simp-all
  apply (drule mp)
  apply (metis noDA1eq)
  apply (case-tac x  $\in \text{dom } (C \text{ (list2FWpolicy (insertDenies ys))})$ )
  apply (metis Cdom2 DenyAllFromTo idNMT list2FWpolicyconc)
  apply (metis domID)
  apply (case-tac x  $\in \text{dom } (C \text{ (list2FWpolicy (insertDenies ys))})$ )
  apply (subgoal-tac C (list2FWpolicy (DenyAllFromTo a b  $\oplus$  DenyAllFromTo b a  $\oplus$ 
    DenyAllFromTo a b  $\#$  insertDenies ys)) x = Some (deny ()))
  apply simp-all
  apply (subgoal-tac C (list2FWpolicy (DenyAllFromTo a b  $\#$  ys)) x =
    C ((DenyAllFromTo a b)) x)
  apply (simp add: PLemmas, simp split: if-splits)
  apply (metis list2FWpolicyconc nlpaux)
  apply (metis Cdom2 DenyAllFromTo iD-isD iDdomAux idNMT list2FWpolicyconc)
apply (metis Cdom2 DenyAllFromTo domIff idNMT list2FWpolicyconc nlpaux)
done
next

```

```

case (AllowPortFromTo a b c) thus ?thesis using AllowPortFromTo Cons 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 (list2FWpolicy ys)))
  apply simp-all
  apply (drule mp)
  apply (metis noDA1eq)
  apply (case-tac x ∈ dom (C (list2FWpolicy (insertDenies ys))))
  apply (metis Cdom2 AllowPortFromTo idNMT list2FWpolicyconc)
  apply (metis domID)
  apply (subgoal-tac x ∈ dom (C (AllowPortFromTo a b c)))
  apply (case-tac x ∉ dom (C (list2FWpolicy (insertDenies ys))))
  apply simp-all
  apply (metis AllowPortFromTo Cdom2 ConcAssoc l2p-aux2 list2FWpolicyconc nlpaux)
  apply (metis AllowPortFromTo Combinators.simps(6) member.simps(4)
    dom-id noDenyAll.simps(1) separated.simps(1))
  apply (metis AllowPortFromTo domdConcStart)
  done
next
case (Conc a b) thus ?thesis using Cons Conc
  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 ∈ dom (C (list2FWpolicy ys)))
  apply simp-all
  apply (drule mp)
  apply (metis noDA1eq)
  apply (case-tac x ∈ dom (C (a ⊕ b)))
  apply (case-tac x ∉ dom (C (list2FWpolicy (insertDenies ys))))
  apply simp-all
  apply (subst list2FWpolicyconc)
  apply (rule idNMT, simp)
  apply (metis domID)
  apply (metis Cdom2 Conc idNMT list2FWpolicyconc)
  apply (metis Cdom2 Conc domIff idNMT list2FWpolicyconc)
  apply (case-tac x ∈ dom (C (a ⊕ b)))
  apply (case-tac x ∉ dom (C (list2FWpolicy (insertDenies ys))))
  apply simp-all
  apply (subst list2FWpolicyconc)
  apply (rule idNMT, simp)
  apply (metis Cdom2 Conc ConcAssoc list2FWpolicyconc nlpaux)
  apply (metis Conc member.simps(1) dom-id
    noDenyAll.simps(1) separated.simps(1))
  apply (metis Conc domdConcStart)
  done

```

qed
qed

lemma *C-eq-iD*: $\llbracket \text{separated } p; \text{noDenyAll1 } p; \text{wellformed-policy1-strong } p \rrbracket \implies$
 $C (\text{list2FWpolicy } (\text{insertDenies } p)) = C (\text{list2FWpolicy } p)$
apply (rule ext)
apply (rule C-eq-iD-aux2)
apply simp-all
apply (subgoal-tac $\text{DenyAll} \in \text{set } p$)
apply (metis CConcStartA DAAux wp1n-tl)
apply (erule wp1-aux1aa)
done

lemma *noDAsortQ*[rule-format]: $\text{noDenyAll1 } p \longrightarrow \text{noDenyAll1 } (\text{qsort } p \ l)$
apply (case-tac p)
apply simp
apply simp
apply (case-tac $a = \text{DenyAll}$)
apply simp-all
apply (rule impI)
apply (subst nDAeqSet)
defer 1
apply simp
defer 1
apply (rule set-sortQ)
apply (rule impI)
apply (rule noDA1eq)
apply (subgoal-tac $\text{noDenyAll } (a \# \text{list})$)
defer 1
apply (case-tac $a, \text{simp}, \text{simp}$)
apply simp
apply simp
apply (subst nDAeqSet)
defer 1
apply assumption
by (metis append-Cons append-Nil qsort.simps(2) set-qsort)

lemma *NetsCollectedSortQ*: $\text{distinct } p \implies \text{noDenyAll1 } p \implies \text{all-in-list } p \ l \implies$
 $\text{singleCombinators } p \implies \text{NetsCollected } (\text{qsort } p \ l)$
apply (rule-tac $l = l$ **in** NetsCollectedSorted)
apply (rule noDAsortQ)
apply simp-all
apply (rule-tac $b=p$ **in** all-in-listSubset)
apply simp-all
apply (rule sort-is-sortedQ)

apply *simp-all*
done

lemmas *CLemmas* = *nMTSort nMTSortQ noneMTRS2 noneMTrd*
noDAsort noDAsortQ nDASC wp1-eq wp1ID
SCp2l ANDSep wp1n-RS2
OTNSEp OTNSC noDA1sep wp1-alternativesep wellformed-eq
wellformed1-alternative-sorted

lemmas *C-eqLemmas-id* = *CLemmas NC2Sep NetsCollectedSep*
NetsCollectedSort NetsCollectedSortQ separatedNC

lemma *C-eq-Until-InsertDenies*: $\llbracket \text{DenyAll} \in \text{set} (\text{policy2list } p); \text{all-in-list} (\text{policy2list } p) \text{ } l; \text{allNetsDistinct} (\text{policy2list } p) \rrbracket \implies$
 $C (\text{list2FWpolicy } ((\text{insertDenies } (\text{separate } (\text{sort } (\text{removeShadowRules2 } (\text{remdups } (\text{removeShadowRules3 } C (\text{insertDeny } (\text{removeShadowRules1 } (\text{policy2list } p)))))) \text{ } l)))))) =$
 $C \text{ } p$
apply (*subst C-eq-iD*)
apply (*simp-all add: C-eqLemmas-id*)
apply (*rule C-eq-until-separated*)
apply *simp-all*
done

lemma *C-eq-Until-InsertDeniesQ*: $\llbracket \text{DenyAll} \in \text{set} (\text{policy2list } p); \text{all-in-list} (\text{policy2list } p) \text{ } l; \text{allNetsDistinct} (\text{policy2list } p) \rrbracket \implies$
 $C (\text{list2FWpolicy } ((\text{insertDenies } (\text{separate } (\text{qsort } (\text{removeShadowRules2 } (\text{remdups } (\text{removeShadowRules3 } C (\text{insertDeny } (\text{removeShadowRules1 } (\text{policy2list } p)))))) \text{ } l)))))) =$
 $C \text{ } p$
apply (*subst C-eq-iD*)
apply (*simp-all add: C-eqLemmas-id*)
apply (*metis WP1rd set-qsort wellformed1-sortedQ wellformed-eq wp1ID wp1-alternativesep wp1-aux1aa wp1n-RS2 wp1n-RS3*)
apply (*rule C-eq-until-separatedQ*)
apply *simp-all*
done

lemma *C-eq-RD-aux*[*rule-format*]: $C (p) \text{ } x = C (\text{removeDuplicates } p) \text{ } x$
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]:
   $p \neq [] \longrightarrow C \text{ (list2FWpolicy } p) = C \text{ (list2FWpolicy (removeAllDuplicates } p))$   $x$ 
apply (induct  $p$ )
apply simp-all
apply (case-tac  $p = []$ )
apply simp-all
apply (metis C-eq-RD-aux)
apply (subst list2FWpolicyconc)
apply simp
apply (case-tac  $x \in \text{dom } (C \text{ (list2FWpolicy } p))$ )
apply (subst list2FWpolicyconc)
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 list2FWpolicyconc)
apply (rule rADnMT)
apply simp
apply (subst nlpaux)
apply (simp add: dom-def)
apply (rule C-eq-RD-aux)
done

```

```

lemma C-eq-RAD:
   $p \neq [] \implies C \text{ (list2FWpolicy } p) = C \text{ (list2FWpolicy (removeAllDuplicates } p))$ 
apply (rule ext)
apply (erule C-eq-RAD-aux)
done

```

```

lemma C-eq-compile:
   $\llbracket \text{DenyAll} \in \text{set } (\text{policy2list } p); \text{all-in-list } (\text{policy2list } p) \text{ } l; \text{allNetsDistinct } (\text{policy2list } p) \rrbracket \implies$ 
   $C \text{ (list2FWpolicy (removeAllDuplicates (insertDenies (separate (sort$ 
   $(\text{removeShadowRules2 (remdups (removeShadowRules3 } C \text{ (insertDeny$ 
   $(\text{removeShadowRules1 } (\text{policy2list } p)))))) \text{ } l)))))) = C \text{ } p$ 
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

lemma *C-eq-compileQ*:

$\llbracket \text{DenyAll} \in \text{set } (\text{policy2list } p); \text{all-in-list } (\text{policy2list } p) \text{ } l; \\ \text{allNetsDistinct } (\text{policy2list } p) \rrbracket \implies$
 $C \text{ (list2FWpolicy (removeAllDuplicates (insertDenies (separate (qsort$
 $(\text{removeShadowRules2 (remdups (removeShadowRules3 } C \text{ (insertDeny$
 $(\text{removeShadowRules1 (policy2list } p)))))) \text{ } l)))))) = C \text{ } p$
apply (*subst C-eq-RAD[symmetric]*)
apply (*rule idNMT*)
apply (*metis WP1rd sepnMT sortnMTQ wellformed-policy1-strong.simps(1) wp1ID wp1n-RS2*
wp1n-RS3)
apply (*rule C-eq-Until-InsertDeniesQ*)
apply *simp-all*
done

lemma *C-eq-normalize*:

$\llbracket \text{DenyAll} \in \text{set } (\text{policy2list } p); \\ \text{allNetsDistinct } (\text{policy2list } p); \\ \text{all-in-list } (\text{policy2list } p) \text{ (Nets-List } p) \rrbracket \implies$
 $C \text{ (list2FWpolicy (normalize } p)) = C \text{ } p$
apply (*simp add: normalize-def*)
apply (*rule C-eq-compile*)
apply *simp-all*
done

lemma *C-eq-normalizeQ*:

$\llbracket \text{DenyAll} \in \text{set } (\text{policy2list } p); \\ \text{allNetsDistinct } (\text{policy2list } p); \\ \text{all-in-list } (\text{policy2list } p) \text{ (Nets-List } p) \rrbracket \implies$
 $C \text{ (list2FWpolicy (normalizeQ } p)) = C \text{ } p$
apply (*simp add: normalizeQ-def*)
apply (*rule C-eq-compileQ*)
apply *simp-all*
done

lemma *domSubset3*: $\text{dom } (C \text{ (DenyAll } \oplus \text{ } x)) = \text{dom } (C \text{ (DenyAll)})$

apply (*simp add: PLemmas split-tupled-all split: option.splits*)
done

lemma *domSubset4*: $\text{dom } (C \text{ (DenyAllFromTo } x \ y \oplus \text{ DenyAllFromTo } y \ x \oplus \text{ AllowPortFromTo } x \ y \text{ dn})) =$
 $\text{dom } (C \text{ (DenyAllFromTo } x \ y \oplus \text{ DenyAllFromTo } y \ x))$
apply (*simp add: PLemmas*)
apply (*simp split: option.splits decision.splits*)
apply *auto*
done

lemma *domSubset5*:
 $\text{dom } (C \text{ (DenyAllFromTo } x \ y \oplus \text{ DenyAllFromTo } y \ x \oplus \text{ AllowPortFromTo } y \ x \text{ dn})) =$
 $\text{dom } (C \text{ (DenyAllFromTo } x \ y \oplus \text{ DenyAllFromTo } y \ x))$
apply (*simp add: PLemmas*)
apply (*simp split: option.splits decision.splits*)
apply *auto*
done

lemma *domSubset1*: $\text{dom } (C \text{ (DenyAllFromTo one two } \oplus \text{ DenyAllFromTo two one } \oplus \text{ AllowPortFromTo one two dn } \oplus \ x)) =$
 $\text{dom } (C \text{ (DenyAllFromTo one two } \oplus \text{ DenyAllFromTo two one } \oplus \ x))$
apply (*simp add: PLemmas*)
apply (*simp split: option.splits decision.splits*)
apply (*simp add: allow-all-def deny-all-def*)
apply *auto*
done

lemma *domSubset2*:
 $\text{dom } (C \text{ (DenyAllFromTo one two } \oplus \text{ DenyAllFromTo two one } \oplus \text{ AllowPortFromTo two one dn } \oplus \ x)) =$
 $\text{dom } (C \text{ (DenyAllFromTo one two } \oplus \text{ DenyAllFromTo two one } \oplus \ x))$
apply (*simp add: PLemmas*)
apply (*simp split: option.splits decision.splits*)
apply (*simp add: allow-all-def deny-all-def*)
apply *auto*
done

lemma *ConcAssoc2*: $C \text{ (} X \oplus Y \oplus ((A \oplus B) \oplus D)) = C \text{ (} X \oplus Y \oplus A \oplus B \oplus D)$
apply (*simp add: C.simps*)
done

lemma *ConcAssoc3*: $C \text{ (} X \oplus ((Y \oplus A) \oplus D)) = C \text{ (} X \oplus Y \oplus A \oplus D)$

apply (*simp add: C.simps*)
done

lemma *RS3-NMT*[*rule-format*]: $\text{DenyAll} \in \text{set } p \longrightarrow$
 $\text{removeShadowRules3 } C \ p \neq []$
apply (*induct-tac p*)
apply (*simp-all add: PLemmas*)
done

lemma *norm-notMT*: $\text{DenyAll} \in \text{set } (\text{policy2list } p) \implies \text{normalize } p \neq []$
apply (*simp add: normalize-def*)
apply (*rule rADnMT*)
apply (*rule idNMT*)
apply (*rule sepnMT*)
apply (*rule sortnMT*)
apply (*rule RS2-NMT*)
apply (*rule remDupsNMT*)
apply (*rule RS3-NMT*)
apply (*rule DAiniD*)
done

lemma *norm-notMTQ*: $\text{DenyAll} \in \text{set } (\text{policy2list } p) \implies \text{normalizeQ } p \neq []$
apply (*simp add: normalizeQ-def*)
apply (*rule rADnMT*)
apply (*rule idNMT*)
apply (*rule sepnMT*)
apply (*rule sortnMTQ*)
apply (*rule RS2-NMT*)
apply (*rule remDupsNMT*)
apply (*rule RS3-NMT*)
apply (*rule DAiniD*)
done

lemma *domDA*: $\text{dom } (C (\text{DenyAll} \oplus A)) = \text{dom } (C (\text{DenyAll}))$
apply (*rule domSubset3*)
done

lemmas *domain-reasoning* = *domDA ConcAssoc2 domSubset1 domSubset2*
domSubset3 domSubset4 domSubset5 domSubsetDistr1
domSubsetDistr2 domSubsetDistrA domSubsetDistrD coerc-assoc ConcAssoc
ConcAssoc3

The following lemmas help with the normalisation

lemma *list2policyR-Start*[*rule-format*]: $p \in \text{dom } (C \ a) \longrightarrow$
 $C (\text{list2policyR } (a \ \# \ \text{list})) \ p = C \ a \ p$

```

apply (rule list2policyR.induct) back
apply (auto simp: C.simps dom-def map-add-def)
done

```

```

lemma list2policyR-End:  $p \notin \text{dom } (C \ a) \implies$ 
   $C \ (list2policyR \ (a \# \text{list})) \ p = (C \ a \oplus \text{list2policy} \ (\text{map } C \ \text{list})) \ p$ 
apply (rule list2policyR.induct)
apply (simp-all add: C.simps dom-def map-add-def list2policy-def split: option.splits)
done

```

```

lemma l2polR-eq-el[rule-format]:  $N \neq [] \longrightarrow$ 
   $C \ (list2policyR \ N) \ p = (list2policy \ (\text{map } C \ N)) \ p$ 
apply (induct-tac N)
apply (simp-all add: list2policy-def)
apply (case-tac  $p \in \text{dom } (C \ a)$ )
apply (simp add: domStart)
apply (rule list2policyR-Start)
apply simp-all
apply (rule list2policyR.induct)
apply simp-all
apply (simp-all add: C.simps dom-def map-add-def)
apply (simp split: option.splits)
done

```

```

lemma l2polR-eq:  $N \neq [] \implies$ 
   $C \ (list2policyR \ N) = (list2policy \ (\text{map } C \ N))$ 
by (auto simp: list2policy-def l2polR-eq-el )

```

```

lemma list2FWpolicys-eq-el[rule-format]:
   $\text{Filter} \neq [] \longrightarrow C \ (list2policyR \ \text{Filter}) \ p = C \ (list2FWpolicy \ (\text{rev } \text{Filter})) \ p$ 
apply (induct-tac Filter)
apply simp-all
apply (case-tac  $\text{list} = []$ )
apply simp-all
apply (case-tac  $p \in \text{dom } (C \ a)$ )
apply simp-all
apply (rule list2policyR-Start)
apply simp-all
apply (subgoal-tac  $C \ (list2policyR \ (a \# \text{list})) \ p = C \ (list2policyR \ \text{list}) \ p$ )
apply (subgoal-tac  $C \ (list2FWpolicy \ (\text{rev } \text{list} \ @ \ [a])) \ p = C \ (list2FWpolicy \ (\text{rev } \text{list})) \ p$ )
apply simp
apply (rule CConcStart2)
apply simp

```

```

apply simp
apply (thin-tac ?S)
apply (case-tac list,simp-all)
apply (simp-all add: C.simps dom-def map-add-def)
done

```

```

lemma list2FWpolicys-eq:
  Filter  $\neq [] \implies$ 
     $C (list2policyR\ Filter) = C (list2FWpolicy (rev\ Filter))$ 
by (rule ext, erule list2FWpolicys-eq-el)

```

```

lemma list2FWpolicys-eq-sym:
  Filter  $\neq [] \implies$ 
     $C (list2policyR (rev\ Filter)) = C (list2FWpolicy\ Filter)$ 
by (metis list2FWpolicys-eq rev-is-Nil-conv rev-rev-ident)

```

```

lemma p-eq[rule-format]:  $p \neq [] \longrightarrow$ 
   $list2policy (map\ C (rev\ p)) = C (list2FWpolicy\ p)$ 
by (metis l2polR-eq list2FWpolicys-eq-sym rev.simps(1) rev-rev-ident)

```

```

lemma p-eq2[rule-format]: normalize  $x \neq [] \longrightarrow$ 
   $C (list2FWpolicy (normalize\ x)) = C\ x \longrightarrow$ 
   $list2policy (map\ C (rev (normalize\ x))) = C\ x$ 
apply (simp add: p-eq)
done

```

```

lemma p-eq2Q[rule-format]: normalizeQ  $x \neq [] \longrightarrow$ 
   $C (list2FWpolicy (normalizeQ\ x)) = C\ x \longrightarrow$ 
   $list2policy (map\ C (rev (normalizeQ\ x))) = C\ x$ 
apply (simp add: p-eq)
done

```

```

lemma list2listNMT[rule-format]:  $x \neq [] \longrightarrow map\ sem\ x \neq []$ 
apply (case-tac x)
apply simp-all
done

```

```

lemma Norm-Distr2:
   $r\ o-f\ ((P \otimes_2 (list2policy\ Q))\ o\ d) =$ 
   $(list2policy\ ((P \otimes_L\ Q)\ (op\ \otimes_2)\ r\ d))$ 
apply (rule ext, rule Norm-Distr-2)
done

```

lemma *NATDistr*: $\llbracket N \neq []; F = C \text{ (list2policyR } N) \rrbracket \implies$
 $((\lambda (x,y). x) \text{ o-f } ((NAT \otimes_2 F) \text{ o } (\lambda x. (x,x)))) =$
 $(\text{list2policy } (((NAT \otimes_L (\text{map } C N)) (\text{op } \otimes_2)$
 $(\lambda (x,y). x) (\lambda x. (x,x))))))$
apply *simp*
apply (*simp add: l2polR-eq*)
apply (*rule ext*)
apply (*rule Norm-Distr-2*)
done

lemma *C-eq-normalize-manual*:
 $\llbracket \text{DenyAll} \in \text{set } (\text{policy2list } p);$
 $\text{allNetsDistinct } (\text{policy2list } p);$
 $\text{all-in-list } (\text{policy2list } p) \text{ l} \rrbracket \implies$
 $C \text{ (list2FWpolicy (normalize-manual-order } p \text{ l))} = C p$
apply (*simp add: normalize-manual-order-def*)
apply (*rule C-eq-compile*)
apply *simp-all*
done

lemma *p-eq2-manualQ*[*rule-format*]: $\text{normalize-manual-orderQ } x \text{ l} \neq [] \longrightarrow$
 $C \text{ (list2FWpolicy (normalize-manual-orderQ } x \text{ l))} = C x \longrightarrow$
 $\text{list2policy } (\text{map } C \text{ (rev (normalize-manual-orderQ } x \text{ l))}) = C x$
apply (*simp add: p-eq*)
done

lemma *norm-notMT-manualQ*: $\text{DenyAll} \in \text{set } (\text{policy2list } p) \implies \text{normalize-manual-orderQ } p$
 $\text{l} \neq []$
apply (*simp add: normalize-manual-orderQ-def*)
apply (*rule rADnMT*)
apply (*rule idNMT*)
apply (*rule sepnMT*)
apply (*rule sortnMTQ*)
apply (*rule RS2-NMT*)
apply (*rule remDupsNMT*)
apply (*rule RS3-NMT*)
apply (*rule DAiniD*)
done

lemma *C-eq-normalize-manualQ*:
 $\llbracket \text{DenyAll} \in \text{set } (\text{policy2list } p);$

```

allNetsDistinct (policy2list p);
all-in-list (policy2list p) l]] ==>
C (list2FWpolicy (normalize-manual-orderQ p l)) = C p
apply (simp add: normalize-manual-orderQ-def)
apply (rule C-eq-compileQ)
apply simp-all
done

```

```

lemma p-eq2-manual[rule-format]: normalize-manual-order x l ≠ [] →
  C (list2FWpolicy (normalize-manual-order x l)) = C x →
  list2policy (map C (rev (normalize-manual-order x l))) = C x
apply (simp add: p-eq)
done

```

```

lemma norm-notMT-manual: DenyAll ∈ set (policy2list p) ==> normalize-manual-order p l ≠
[]
apply (simp add: normalize-manual-order-def)
apply (rule rADnMT)
apply (rule idNMT)
apply (rule sepnMT)
apply (rule sortnMT)
apply (rule RS2-NMT)
apply (rule remDupsNMT)
apply (rule RS3-NMT)
apply (rule DAiniD)
done

```

As an example, how this theorems can be used for a concrete normalisation instantiation.

```

lemma normalizeNAT: [[DenyAll ∈ set (policy2list Filter);
allNetsDistinct (policy2list Filter);
all-in-list (policy2list Filter) (Nets-List Filter)]] ==>
((λ (x,y). x) o-f (((NAT ⊗2 C Filter) o (λx. (x,x)))) =
list2policy ((NAT ⊗L (map C (rev (normalize Filter)))) (op ⊗2) (λ (x,y). x) (λ x. (x,x)))
apply (rule NATDistr)
apply simp-all
apply (metis norm-notMT)
by (metis C-eq-normalize list2FWpolicys-eq-sym norm-notMT)

```

```

lemma domSimpl[simp]: dom (C (A ⊕ DenyAll)) = dom (C (DenyAll))
apply (simp add: PLemmas)
done

```


The followin theorems can be applied when prepending the usual normalisation with an additional step and using another semantical interpretation function. This is a general recipe which can be applied whenever one nees to combine several normalisation strategies.

lemma *CRotate-eq-rotateC*: $CRotate\ p = C\ (rotatePolicy\ p)$
apply (*induct* p *rule*: *rotatePolicy.induct*)
apply *simp-all*
apply (*simp add*: $C.simps\ map-add-def$)
done

lemma *DAinRotate*: $\llbracket DenyAll \in set\ (policy2list\ p) \rrbracket$
 $\implies DenyAll \in set\ (policy2list\ (rotatePolicy\ p))$
apply (*induct* p , *simp-all*)
apply (*case-tac* $DenyAll \in set\ (policy2list\ p1)$, *simp-all*)
done

lemma *DAUniv*: $dom\ (CRotate\ (P \oplus DenyAll)) = UNIV$
apply (*simp add*: *PLemmas*)
apply (*auto simp*: *PLemmas*)
apply (*simp split*: *option.splits decision.splits*)
apply *auto*
apply (*simp add*: *deny-all-def*)
done

lemma *p-eq2R[rule-format]*: $normalize\ (rotatePolicy\ x) \neq [] \longrightarrow$
 $C\ (list2FWpolicy\ (normalize\ (rotatePolicy\ x))) = CRotate\ x \longrightarrow$
 $list2policy\ (map\ C\ (rev\ (normalize\ (rotatePolicy\ x)))) = CRotate\ x$
apply (*simp add*: *p-eq*)
done

lemma *C-eq-normalizeRotate*:
 $\llbracket DenyAll \in set\ (policy2list\ p) \rrbracket$;
 $allNetsDistinct\ (policy2list\ (rotatePolicy\ p))$;
 $all-in-list\ (policy2list\ (rotatePolicy\ p))\ (Nets-List\ (rotatePolicy\ p)) \implies$
 $C\ (list2FWpolicy\ (removeAllDuplicates\ (insertDenies\ (separate\ (sort\ (removeShadowRules2\ (remdups\ (removeShadowRules3\ C\ (insertDeny\ (removeShadowRules1\ (policy2list\ (rotatePolicy\ p))))))\ (Nets-List\ (rotatePolicy\ p)))))) = CRotate\ p$
apply (*simp add*: *CRotate-eq-rotateC*)
apply (*subst* *C-eq-RAD[symmetric]*)
apply (*rule idNMT*)
apply (*simp add*: *C-eqLemmas-id*)
apply (*rule C-eq-Until-InsertDenies*)

```

apply simp-all
apply (erule DAinRotate)
done

```

```

lemma C-eq-normalizeRotate2:
   $\llbracket \text{DenyAll} \in \text{set } (\text{policy2list } p);$ 
   $\text{allNetsDistinct } (\text{policy2list } (\text{rotatePolicy } p));$ 
   $\text{all-in-list } (\text{policy2list } (\text{rotatePolicy } p)) (\text{Nets-List } (\text{rotatePolicy } p)) \rrbracket \implies$ 
   $C (\text{list2FWpolicy } (\text{normalize } (\text{rotatePolicy } p))) = C\text{Rotate } p$ 
apply (simp add: normalize-def, erule C-eq-normalizeRotate, simp-all)
done

```

end

```

theory NormalisationIPPPProofs
imports NormalisationIntegerPortProof
begin

```

Normalisation proofs which are specific to the IntegerProtocol address representation.

```

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

```

```

lemma aux26[simp]:  $\text{twoNetsDistinct } a \ b \ c \ d \implies$ 
   $\text{dom } (Cp (\text{AllowPortFromTo } a \ b \ p)) \cap \text{dom } (Cp (\text{DenyAllFromTo } c \ d)) = \{\}$ 
by (auto simp: PLemmas twoNetsDistinct-def netsDistinct-def) auto

```

```

lemma wp2-aux[rule-format]:  $\text{wellformed-policy2Pr } (xs \ @ \ [x]) \longrightarrow$ 
   $\text{wellformed-policy2Pr } xs$ 
apply (induct xs, simp-all)
apply (case-tac a, simp-all)
done

```

```

lemma Cdom2:  $x \in \text{dom}(Cp \ b) \implies Cp \ (a \oplus b) \ x = (Cp \ b) \ x$ 
by (auto simp: Cp.simps)

```

```

lemma wp2Conc[rule-format]:  $\text{wellformed-policy2Pr } (x \# xs) \implies \text{wellformed-policy2Pr } xs$ 
by (case-tac x, simp-all)

```

```

lemma DAimpliesMR-E[rule-format]:  $\text{DenyAll} \in \text{set } p \longrightarrow$ 
   $(\exists \ r. \text{applied-rule-rev } Cp \ x \ p = \text{Some } r)$ 
apply (simp add: applied-rule-rev-def)

```

apply (*rule-tac* $xs = p$ **in** *rev-induct*)
apply *simp-all*
by (*metis* $Cp.simps(1)$ *denyAllDom*)

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

lemma *MRList1*[*rule-format*]: $x \in dom\ (Cp\ a) \implies applied_rule_rev\ Cp\ x\ (b@[a]) = Some\ a$
by (*simp add*: *applied-rule-rev-def*)

lemma *MRList2*: $x \in dom\ (Cp\ a) \implies applied_rule_rev\ Cp\ x\ (c@b@[a]) = Some\ a$
by (*simp add*: *applied-rule-rev-def*)

lemma *MRList3*: $x \notin dom\ (Cp\ xa) \implies$
 $applied_rule_rev\ Cp\ x\ (a\ @\ b\ \# \ xs\ @\ [xa]) = applied_rule_rev\ Cp\ x\ (a\ @\ b\ \# \ xs)$
by (*simp add*: *applied-rule-rev-def*)

lemma *CConcEnd*[*rule-format*]: $Cp\ a\ x = Some\ y \longrightarrow$
 $Cp\ (list2FWpolicy\ (xs\ @\ [a]))\ x = Some\ y$
(is *?P xs*)
apply (*rule-tac* $P = ?P$ **in** *list2FWpolicy.induct*)
by (*simp-all add*: $Cp.simps$)

lemma *CConcStartaux*: $\llbracket Cp\ a\ x = None \rrbracket \implies (Cp\ aa\ ++\ Cp\ a)\ x = Cp\ aa\ x$
by (*simp add*: *PLemmas*)

lemma *CConcStart*[*rule-format*]: $xs \neq [] \longrightarrow Cp\ a\ x = None \longrightarrow$
 $Cp\ (list2FWpolicy\ (xs\ @\ [a]))\ x = Cp\ (list2FWpolicy\ xs)\ x$
apply (*rule* *list2FWpolicy.induct*)
by (*simp-all add*: *PLemmas*)

lemma *mrNnt*[*simp*]: $applied_rule_rev\ Cp\ x\ p = Some\ a \implies p \neq []$
apply (*simp add*: *applied-rule-rev-def*)
by *auto*

lemma *mr-is-C*[*rule-format*]: $applied_rule_rev\ Cp\ x\ p = Some\ a \longrightarrow$
 $Cp\ (list2FWpolicy\ (p))\ x = Cp\ a\ x$
apply (*simp add*: *applied-rule-rev-def*)
apply (*rule* *rev-induct*)
apply *simp-all*
apply *safe*
apply (*metis* *CConcEnd rotate-simps*)
apply (*metis* *CConcEnd*)
by (*metis* *CConcStart applied-rule-rev-def mrNnt option.exhaust*)

lemma *CConcStart2*: $\llbracket p \neq []; x \notin dom\ (Cp\ a) \rrbracket \implies$

$$Cp (list2FWpolicy (p@[a])) x = Cp (list2FWpolicy p)x$$
by (erule CConcStart,simp add: PLemmas)

lemma CConcEnd1: $\llbracket q@p \neq []; x \notin dom (Cp a) \rrbracket \implies$

$$Cp (list2FWpolicy (q@p@[a])) x = Cp (list2FWpolicy (q@p))x$$
apply (subst lCdom2)
by (rule CConcStart2, simp-all)

lemma CConcEnd2[rule-format]: $x \in dom (Cp a) \longrightarrow$

$$Cp (list2FWpolicy (xs @ [a])) x = Cp a x$$
(is ?P xs)
apply (rule-tac P = ?P in list2FWpolicy.induct)
by (auto simp:Cp.simps)

lemma bar3: $x \in dom (Cp (list2FWpolicy (xs @ [xa]))) \implies$

$$x \in dom (Cp (list2FWpolicy xs)) \vee x \in dom (Cp xa)$$
apply auto
by (metis CConcStart eq-Nil-appendI l2p-aux2 option.simps(3))

lemma CeqEnd[rule-format,simp]: $a \neq [] \longrightarrow x \in dom (Cp (list2FWpolicy a)) \longrightarrow$

$$Cp (list2FWpolicy (b@a)) x = (Cp (list2FWpolicy a)) x$$
apply (rule rev-induct,simp-all)
apply (case-tac xs $\neq []$, simp-all)
apply (case-tac $x \in dom (Cp xa)$)
apply (metis CConcEnd2 MRLList2 mr-is-C rotate-simps)
apply (metis CConcEnd1 CConcStart2 Nil-is-append-conv bar3 rotate-simps)
apply (metis MRLList2 eq-Nil-appendI mr-is-C rotate-simps)
done

lemma CConcStartA[rule-format,simp]: $x \in dom (Cp a) \longrightarrow$

$$x \in dom (Cp (list2FWpolicy (a \# b)))$$
(is ?P b)
apply (rule-tac P = ?P in list2FWpolicy.induct)
apply (simp-all add: Cp.simps)
done

lemma domConc: $\llbracket x \in dom (Cp (list2FWpolicy b)); b \neq [] \rrbracket \implies$

$$x \in dom (Cp (list2FWpolicy (a@b)))$$
by (auto simp: PLemmas)

lemma CeqStart[rule-format,simp]:

$$x \notin dom (Cp (list2FWpolicy a)) \longrightarrow a \neq [] \longrightarrow b \neq [] \longrightarrow$$

$$Cp (list2FWpolicy (b@a)) x = (Cp (list2FWpolicy b)) x$$
apply (rule list2FWpolicy.induct,simp-all)
apply (auto simp: list2FWpolicyconc PLemmas)
done

lemma *C-eq-if-mr-eq2*: $\llbracket \text{applied-rule-rev } Cp \ x \ a = \text{Some } r; \text{applied-rule-rev } Cp \ x \ b = \text{Some } r; \text{ } a \neq []; b \neq [] \rrbracket \implies$
 $(Cp \ (list2FWpolicy \ a)) \ x = (Cp \ (list2FWpolicy \ b)) \ x$
by (*metis mr-is-C*)

lemma *nMRtoNone*[*rule-format*]: $p \neq [] \longrightarrow \text{applied-rule-rev } Cp \ x \ p = \text{None} \longrightarrow$
 $Cp \ (list2FWpolicy \ p) \ x = \text{None}$
apply (*rule rev-induct, simp-all*)
apply (*case-tac xs = [], simp-all*)
by (*simp-all add: applied-rule-rev-def dom-def*)

lemma *C-eq-if-mr-eq*:
 $\llbracket \text{applied-rule-rev } Cp \ x \ b = \text{applied-rule-rev } Cp \ x \ a; a \neq []; b \neq [] \rrbracket \implies$
 $(Cp \ (list2FWpolicy \ a)) \ x = (Cp \ (list2FWpolicy \ b)) \ x$
apply (*cases applied-rule-rev Cp 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 *notmatching-notdom*: $\text{applied-rule-rev } Cp \ x \ (p@[a]) \neq \text{Some } a \implies x \notin \text{dom } (Cp \ a)$
by (*simp add: applied-rule-rev-def split: if-splits*)

lemma *foo3a*[*rule-format*]: $\text{applied-rule-rev } Cp \ x \ (a@[b]@c) = \text{Some } b \longrightarrow r \in \text{set } c \longrightarrow$
 $b \notin \text{set } c \longrightarrow x \notin \text{dom } (Cp \ r)$
apply (*rule rev-induct*)
apply *simp-all*
apply (*rule impI|rule conjI|simp*)
apply (*rule-tac p = a @ b # xs in notmatching-notdom, 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 notmatching-notdom, simp-all*)
done

lemma *foo3D*: $\llbracket \text{wellformed-policy1 } p; p = (\text{DenyAll} \# ps); \text{applied-rule-rev } Cp \ x \ p = \text{Some } \text{DenyAll}; r \in \text{set } ps \rrbracket \implies x \notin \text{dom } (Cp \ r)$
by (*rule-tac a = [] and b = DenyAll and c = ps in foo3a, simp-all*)

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

lemma *foo5b*[*rule-format*]: $x \in \text{dom } (Cp \ b) \longrightarrow (\forall \ r. r \in \text{set } c \longrightarrow x \notin \text{dom } (Cp \ r)) \longrightarrow$

$applied\text{-}rule\text{-}rev\ Cp\ x\ (b\#c) = Some\ b$
apply (*simp add: applied-rule-rev-def*)
apply (*rule-tac xs = c in rev-induct, simp-all*)
done

lemma *mr-first*: $\llbracket x \in dom\ (Cp\ b); (\forall\ r. r \in set\ c \longrightarrow x \notin dom\ (Cp\ r)); s = b\#c \rrbracket \Longrightarrow$
 $applied\text{-}rule\text{-}rev\ Cp\ x\ s = Some\ b$
by (*simp add: foo5b*)

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

lemma *foo8*: $\llbracket (\forall\ r. r \in set\ p \wedge x \in dom\ (Cp\ r) \longrightarrow r = a); set\ p = set\ s \rrbracket \Longrightarrow$
 $(\forall\ r. r \in set\ s \wedge x \in dom\ (Cp\ r) \longrightarrow r = a)$
by *auto*

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

lemma *wp3tl*[*rule-format*]: $wellformed\text{-}policy3Pr\ p \longrightarrow wellformed\text{-}policy3Pr\ (tl\ p)$
by (*induct p, simp-all, case-tac a, simp-all*)

lemma *wp3Conc*[*rule-format*]: $wellformed\text{-}policy3Pr\ (a\#p) \longrightarrow wellformed\text{-}policy3Pr\ p$
by (*induct p, simp-all, case-tac a, simp-all*)

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

lemma *mrMTNone*[*simp*]: $applied\text{-}rule\text{-}rev\ Cp\ x\ [] = None$
by (*simp add: applied-rule-rev-def*)

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

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

lemma *mr-not-Conc*: *singleCombinators* $p \implies \text{applied-rule-rev } Cp\ x\ p \neq \text{Some } (a \oplus b)$
apply (*auto simp: mrSet*)
apply (*drule mrSet*)
apply (*erule SCnotConc, simp*)
done

lemma *foo25*[rule-format]: *wellformed-policy3Pr* $(p@[x]) \longrightarrow \text{wellformed-policy3Pr } p$
by (*induct p, simp-all, case-tac a, simp-all*)

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

lemma *wp3EndMT*[rule-format]: *wellformed-policy3Pr* $(p@[xs]) \longrightarrow$
 $\text{AllowPortFromTo } a\ b\ po \in \text{set } p \longrightarrow$
 $\text{dom } (Cp\ (\text{AllowPortFromTo } a\ b\ po)) \cap \text{dom } (Cp\ xs) = \{\}$
apply (*induct p, simp-all*)
apply (*rule impI*)
apply (*drule mp*)
apply (*erule wp3Conc*)
by *clarify auto*

lemma *foo29*: $\llbracket \text{dom } (Cp\ a) \neq \{\}; \text{dom } (Cp\ a) \cap \text{dom } (Cp\ b) = \{\} \rrbracket \implies a \neq b$
by *auto*

lemma *foo28*: $\llbracket \text{AllowPortFromTo } a\ b\ po \in \text{set } p;$
 $\text{dom } (Cp\ (\text{AllowPortFromTo } a\ b\ po)) \neq \{\}; (\text{wellformed-policy3Pr } (p@[x])) \rrbracket$
 $\implies x \neq \text{AllowPortFromTo } a\ b\ po$
by (*metis foo29 Cp.simps(3) wp3EndMT*)

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

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

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

lemma *foo31*: $\llbracket (\forall r. r \in \text{set } p \wedge x \in \text{dom } (Cp \ r) \longrightarrow$
 $(r = \text{AllowPortFromTo } a \ b \ po \vee r = \text{DenyAllFromTo } a \ b \vee r = \text{DenyAll})) ;$
 $\text{set } p = \text{set } s \rrbracket \Longrightarrow$
 $\llbracket (\forall r. r \in \text{set } s \wedge x \in \text{dom } (Cp \ r) \longrightarrow$
 $(r = \text{AllowPortFromTo } a \ b \ po \vee r = \text{DenyAllFromTo } a \ b \vee r = \text{DenyAll}))$
by *auto*

lemma *wp1-auxa*: *wellformed-policy1-strong* $p \Longrightarrow (\exists r. \text{applied-rule-rev } Cp \ x \ p = \text{Some } r)$
apply (*rule DAimpliesMR-E*)
by (*erule wp1-aux1aa*)

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

lemma *domTrans*: $\llbracket \text{dom } a \subseteq \text{dom } b ; \text{dom}(b) \cap \text{dom } (c) = \{\} \rrbracket \Longrightarrow \text{dom}(a) \cap \text{dom}(c) = \{\}$
by *auto*

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

lemma *DomInterAllowsMT-Ports*: $\llbracket p \neq po \rrbracket \Longrightarrow$
 $\text{dom } (Cp \ (\text{AllowPortFromTo } a \ b \ p)) \cap \text{dom } (Cp \ (\text{AllowPortFromTo } c \ d \ po)) = \{\}$
apply (*simp add: twoNetsDistinct-def netsDistinct-def PLemmas*)
by (*auto simp: prod-eqI*)

lemma *wellformed-policy3-charn[rule-format]*:
 $\text{singleCombinators } p \longrightarrow \text{distinct } p \longrightarrow \text{allNetsDistinct } p \longrightarrow$
 $\text{wellformed-policy1 } p \longrightarrow \text{wellformed-policy2Pr } p \longrightarrow \text{wellformed-policy3Pr } p$
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 DistinctNetsDenyAllow:
  
$$\llbracket \text{DenyAllFromTo } b \ c \in \text{set } p; \text{ AllowPortFromTo } a \ d \ po \in \text{set } p; \text{ allNetsDistinct } p; \\ \text{dom } (Cp \ (\text{DenyAllFromTo } b \ c)) \cap \text{dom } (Cp \ (\text{AllowPortFromTo } a \ d \ po)) \neq \{\} \rrbracket \\ \implies b = a \wedge c = d$$

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:
  
$$\llbracket \text{AllowPortFromTo } b \ c \ poo \in \text{set } p; \text{ AllowPortFromTo } a \ d \ po \in \text{set } p; \\ \text{allNetsDistinct } p; \text{dom } (Cp \ (\text{AllowPortFromTo } b \ c \ poo)) \cap \\ \text{dom } (Cp \ (\text{AllowPortFromTo } a \ d \ po)) \neq \{\} \rrbracket \\ \implies b = a \wedge c = d \wedge poo = po$$

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]:
  
$$\llbracket \text{singleCombinators } p; \\ \text{distinct } p; \\ \text{allNetsDistinct } p \rrbracket \implies \\ \text{wellformed-policy2Pr } (\text{removeShadowRules2 } p)$$

proof (induct p)
  case Nil thus ?case by simp
next
  case (Cons x xs)
    have wp-xs: wellformed-policy2Pr (removeShadowRules2 xs) using assms
  by (metis Cons ANDConc distinct.simps(2) singleCombinatorsConc)

```

```

show ?case
proof (cases x)
  case DenyAll thus ?thesis using wp-xs by simp
next
  case (DenyAllFromTo a b) thus ?thesis
    using wp-xs Cons
    by (simp,metis DenyAllFromTo aux aux7 tNDComm deny-dom)
next
  case (AllowPortFromTo a b p) thus ?thesis
    using assms wp-xs
    by (simp, metis aux26 AllowPortFromTo Cons(4) aux aux7a tNDComm)
next
  case (Conc a b) thus ?thesis
    using assms by (metis Conc Cons(2) singleCombinators.simps(2))
qed
qed

```

```

lemma AD-aux:  $\llbracket \text{AllowPortFromTo } a \ b \ po \in \text{set } p ; \text{DenyAllFromTo } c \ d \in \text{set } p ;$ 
 $\text{allNetsDistinct } p ; \text{singleCombinators } p ;$ 
 $a \neq c \vee b \neq d \rrbracket$ 
 $\implies \text{dom } (Cp \ (\text{AllowPortFromTo } a \ b \ po)) \cap \text{dom } (Cp \ (\text{DenyAllFromTo } c \ d)) = \{\}$ 
apply (rule aux26)
apply (rule-tac x = AllowPortFromTo a b po and y = DenyAllFromTo c d in tND)
apply auto
done

```

```

lemma sorted-WP2[rule-format]: sorted p l  $\longrightarrow$  all-in-list p l  $\longrightarrow$  distinct p  $\longrightarrow$ 
 $\text{allNetsDistinct } p \longrightarrow \text{singleCombinators } p \longrightarrow \text{wellformed-policy2Pr } p$ 
proof (induct p)
  case Nil thus ?case by simp
next
  case (Cons a p) thus ?case
    proof (cases a)
      case DenyAll thus ?thesis using assms Cons
        by (auto intro: ANDConc singleCombinatorsConc sortedConcEnd)
    next
      case (DenyAllFromTo c d) thus ?thesis using assms Cons
        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
    end
  done

```

```

next
case (AllowPortFromTo c d e) thus ?thesis using Cons assms
  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 bothNet.simps(2) in-list.simps(2)
        in-set-in-list)
  by (auto intro: aux7 tNDComm ANDConc singleCombinatorsConc sortedConcEnd)
next
case (Conc a b) thus ?thesis using Cons by simp
qed
qed

```

```

lemma wellformed2-sorted[simp]:  $\llbracket \text{all-in-list } p \text{ } l; \text{ distinct } p; \text{ allNetsDistinct } p; \text{ singleCombinators } p \rrbracket \implies \text{wellformed-policy2Pr } (\text{sort } p \text{ } l)$ 
  apply (rule sorted-WP2)
  apply (erule sort-is-sorted, simp-all)
  apply (erule all-in-listSubset)
  apply (auto intro: SC3 singleCombinatorsConc sorted-insort)
done

```

```

lemma wellformed2-sortedQ[simp]:  $\llbracket \text{all-in-list } p \text{ } l; \text{ distinct } p; \text{ allNetsDistinct } p; \text{ singleCombinators } p \rrbracket \implies \text{wellformed-policy2Pr } (q\text{sort } p \text{ } l)$ 
  apply (rule sorted-WP2)
  apply (erule sort-is-sortedQ, simp-all)
  apply (erule all-in-listSubset)
  apply (auto intro: SC3Q singleCombinatorsConc distinct-sortQ)
done

```

```

lemma C-DenyAll[simp]:  $C_p(\text{list2FWpolicy } (xs @ [\text{DenyAll}])) x = \text{Some } (\text{deny } ())$ 
  by (auto simp: PLemmas)

```

```

lemma C-eq-RS1n:
   $C_p(\text{list2FWpolicy } (\text{removeShadowRules1-alternative } p)) = C_p(\text{list2FWpolicy } p)$ 
  apply (case-tac p = [])

```

```

apply simp-all
apply (metis list2FWpolicy.simps(1) 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: 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 ∈ dom (Cp x))
apply simp-all
done

lemma C-eq-RS1[simp]: p ≠ [] ⇒
      Cp(list2FWpolicy (removeShadowRules1 p)) = Cp(list2FWpolicy p)
by (metis rSR1-eq C-eq-RS1n)

lemma EX-MR-aux[rule-format]: applied-rule-rev Cp x (DenyAll # p) ≠ Some DenyAll ⟶
      (∃ y. applied-rule-rev Cp x p = Some y)
apply (simp add: applied-rule-rev-def)
apply (rule-tac xs = p in rev-induct, simp-all)
done

lemma EX-MR : [applied-rule-rev Cp x p ≠ (Some DenyAll); p = DenyAll#ps] ⟶
      (applied-rule-rev Cp x p = applied-rule-rev Cp x ps)

apply auto

apply (subgoal-tac applied-rule-rev Cp x (DenyAll#ps) ≠ None)
apply auto
apply (metis mrConcEnd)
by (metis DAimpliesMR-E hd.simps hd-in-set list.simps(3) not-Some-eq)

lemma mr-not-DA:
  [wellformed-policy1-strong s; applied-rule-rev Cp x p = Some (DenyAllFromTo a ab);
   set p = set s] ⟶ applied-rule-rev Cp x s ≠ Some DenyAll
apply (subst wp1n-tl, simp-all)
apply (subgoal-tac x ∈ dom (Cp (DenyAllFromTo a ab)))
apply (subgoal-tac DenyAllFromTo a ab ∈ 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 ∈ set p, simp)

```

apply (erule mrSet)
apply (erule mr-in-dom)
done

lemma domsMT-notND-DD:
 $\llbracket \text{dom } (Cp \text{ (DenyAllFromTo } a \ b)) \cap \text{dom } (Cp \text{ (DenyAllFromTo } c \ d)) \neq \{\} \rrbracket \implies$
 $\neg \text{netsDistinct } a \ c$
apply (erule contrapos-nn)
apply (simp add: Cp.simps)
apply (rule aux6)
apply (simp add: twoNetsDistinct-def)
done

lemma domsMT-notND-DD2:
 $\llbracket \text{dom } (Cp \text{ (DenyAllFromTo } a \ b)) \cap \text{dom } (Cp \text{ (DenyAllFromTo } c \ d)) \neq \{\} \rrbracket \implies$
 $\neg \text{netsDistinct } b \ d$
apply (erule contrapos-nn)
apply (simp add: Cp.simps)
apply (rule aux6)
apply (simp add: twoNetsDistinct-def)
done

lemma domsMT-notND-DD3:
 $\llbracket x \in \text{dom } (Cp \text{ (DenyAllFromTo } a \ b)); x \in \text{dom } (Cp \text{ (DenyAllFromTo } c \ d)) \rrbracket \implies$
 $\neg \text{netsDistinct } a \ c$
apply (rule domsMT-notND-DD)
apply auto
done

lemma domsMT-notND-DD4:
 $\llbracket x \in \text{dom } (Cp \text{ (DenyAllFromTo } a \ b)); x \in \text{dom } (Cp \text{ (DenyAllFromTo } c \ d)) \rrbracket \implies$
 $\neg \text{netsDistinct } b \ d$
apply (rule domsMT-notND-DD2)
apply auto
done

lemma NetsEq-if-sameP-DD:
 $\llbracket \text{allNetsDistinct } p; u \in \text{set } p; v \in \text{set } p; u = (\text{DenyAllFromTo } a \ b);$
 $v = (\text{DenyAllFromTo } c \ d); x \in \text{dom } (Cp \text{ (} u)); x \in \text{dom } (Cp \text{ (} v)) \rrbracket \implies$
 $a = c \wedge b = d$
apply (simp add: allNetsDistinct-def)
apply (metis ND0aux1 ND0aux2 domsMT-notND-DD3 domsMT-notND-DD4 mem-def)
done

lemma rule-charn1:
assumes aND: allNetsDistinct p
and mr-is-allow: applied-rule-rev Cp x p = Some (AllowPortFromTo a b po)

```

and SC: singleCombinators p
and inp:  $r \in \text{set } p$ 
and inDom:  $x \in \text{dom } (Cp \ r)$ 
shows  $(r = \text{AllowPortFromTo } a \ b \ po \vee r = \text{DenyAllFromTo } a \ b \vee r = \text{DenyAll})$ 
proof (cases r)
  case DenyAll show ?thesis by (metis DenyAll)
next
  case (DenyAllFromTo x y) show ?thesis using Cons assms DenyAllFromTo
    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 set-empty2)
next
  case (AllowPortFromTo x y b) show ?thesis using assms AllowPortFromTo Cons
    apply (simp)
    apply (rule DistinctNetsAllowAllow)
    apply (simp-all)
    apply (metis mrSet)
    by (metis Int-iff mr-in-dom inSet-not-MT set-empty2)
next
  case (Conc x y) thus ?thesis using assms by (metis aux0-0)
qed

```

```

lemma noneMTsubset[rule-format]:  $\text{noneMT } Cp \ a \longrightarrow \text{set } b \subseteq \text{set } a \longrightarrow \text{noneMT } Cp \ b$ 
apply (induct b, simp-all)
by (metis notMTnMT)

```

```

lemma nMTSort:  $\text{noneMT } Cp \ p \Longrightarrow \text{noneMT } Cp \ (\text{sort } p \ l)$ 
by (metis set-sort nMTeqSet)

```

```

lemma nMTSortQ:  $\text{noneMT } Cp \ p \Longrightarrow \text{noneMT } Cp \ (q\text{sort } p \ l)$ 
by (metis set-sortQ nMTeqSet)

```

```

lemma wp3char[rule-format]:  $\text{noneMT } Cp \ xs \wedge Cp \ (\text{AllowPortFromTo } a \ b \ po) = \text{empty} \wedge$ 
   $\text{wellformed-policy3Pr } (xs \ @ \ [\text{DenyAllFromTo } a \ b]) \longrightarrow$ 
   $\text{AllowPortFromTo } a \ b \ po \notin \text{set } xs$ 
apply (induct xs)
apply simp-all
by (metis domNMT wp3Conc)

```

```

lemma wp3charn[rule-format]:
assumes domAllow:  $\text{dom } (Cp \ (\text{AllowPortFromTo } a \ b \ po)) \neq \{\}$ 

```

```

and wp3: wellformed-policy3Pr (xs @ [DenyAllFromTo a b])
shows allowNotInList: AllowPortFromTo a b po  $\notin$  set xs
apply (insert assms)
proof (induct xs)
  case Nil show ?case by simp
next
  case (Cons x xs) show ?case using Cons
  by (simp, auto intro: wp3Conc) (auto simp: DenyAllowDisj domAllow)
qed

```

```

lemma rule-charn2:
  assumes aND: allNetsDistinct p
  and wp1: wellformed-policy1 p
  and SC: singleCombinators p
  and wp3: wellformed-policy3Pr p
  and allow-in-list: AllowPortFromTo c d po  $\in$  set p
  and x-in-dom-allow:  $x \in \text{dom } (Cp \text{ (AllowPortFromTo c d po)})$ 
  shows applied-rule-rev Cp x p = Some (AllowPortFromTo c d po)
  proof (insert assms, induct p rule: rev-induct)
    case Nil show ?case using Nil by simp
  next
    case (snoc y ys) show ?case using snoc
      apply simp
      apply (case-tac y = (AllowPortFromTo c d po))
      apply (simp add: applied-rule-rev-def)
      apply simp-all
      apply (subgoal-tac ys  $\neq []$ )
      apply (subgoal-tac applied-rule-rev Cp x ys = Some (AllowPortFromTo c d po))
      defer 1
      apply (metis ANDConcEnd SCConcEnd WP1ConcEnd foo25)
      apply (metis inSet-not-MT)
      proof (cases y)
        case DenyAll thus ?thesis using DenyAll snoc
          apply simp
          by (metis DAnotTL DenyAll inSet-not-MT policy2list.simps(2))
      next
        case (DenyAllFromTo a b) thus ?thesis using snoc apply simp
      apply (simp-all add: applied-rule-rev-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 AllowPortFromTo snoc apply simp
      apply (simp-all add: applied-rule-rev-def)
      apply (rule conjI)
      apply (metis domInterMT wp3EndMT)
      by (metis ANDConcEnd AllowPortFromTo SCConcEnd WP1ConcEnd foo25 x-in-dom-allow)
  end

```

```

next
case (Conc a b) thus ?thesis using Conc snoc apply simp
  by (metis Conc aux0-0 in-set-conv-decomp)
qed
qed

lemma rule-charn3:
   $\llbracket \text{wellformed-policy1 } p; \text{allNetsDistinct } p; \text{singleCombinators } p; \\ \text{wellformed-policy3Pr } p; \text{applied-rule-rev } Cp \ x \ p = \text{Some } (\text{DenyAllFromTo } c \ d); \\ \text{AllowPortFromTo } a \ b \ po \in \text{set } p \rrbracket \implies x \notin \text{dom } (Cp \ (\text{AllowPortFromTo } a \ b \ po))$ 
  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-policy3Pr p
  and DA: DenyAll  $\notin$  set p
  and mr: applied-rule-rev Cp x p = Some (DenyAllFromTo a b)
  and rinp: r  $\in$  set p
  and xindom: x  $\in$  dom (Cp r)
  shows r = DenyAllFromTo a b
  proof (cases r)
    case DenyAll thus ?thesis using DenyAll assms by simp
  next
    case (DenyAllFromTo c d) thus ?thesis using assms 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 assms apply simp
      apply (subgoal-tac x  $\notin$  dom (Cp (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 assms apply simp
      by (metis Conc aux0-0)
  qed
qed

lemma foo31a:  $\llbracket (\forall \ r. \ r \in \text{set } p \wedge x \in \text{dom } (Cp \ r) \longrightarrow \\ (r = \text{AllowPortFromTo } a \ b \ po \vee r = \text{DenyAllFromTo } a \ b \vee r = \text{DenyAll}); \\ \text{set } p = \text{set } s; \ r \in \text{set } s; \ x \in \text{dom } (Cp \ r) \rrbracket \implies \\ (r = \text{AllowPortFromTo } a \ b \ po \vee r = \text{DenyAllFromTo } a \ b \vee r = \text{DenyAll})$ 
  by auto

```



```

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

```

```

lemma mrDA-tl:
  assumes mr-DA: applied-rule-rev Cp x p = Some DenyAll
  and wp1n: wellformed-policy1-strong p
  shows applied-rule-rev Cp 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:
   $\llbracket$ wellformed-policy1-strong p; allNetsDistinct p; singleCombinators p;
  wellformed-policy3Pr p; applied-rule-rev Cp x p = Some (DenyAllFromTo a b);
  r  $\in$  set (tl p); x  $\in$  dom (Cp r) $\rrbracket \implies$  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(4) EX-MR option.inject)
apply (metis wp1n-tl)
done

```

```

lemma mrDenyAll-is-unique:
   $\llbracket$ wellformed-policy1-strong p; applied-rule-rev Cp x p = Some DenyAll;
  r  $\in$  set (tl p) $\rrbracket \implies$  x  $\notin$  dom (Cp 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-policy3Pr p*
and *wp3-s*: *wellformed-policy3Pr s*
and *aND*: *allNetsDistinct p*

shows *applied-rule-rev Cp x p = applied-rule-rev Cp x s*

proof (*cases applied-rule-rev Cp x p*)

case *None*

have *DA*: *DenyAll ∈ set p* **using** *wp1-p* **by** (*auto simp: wp1-aux1aa*)
have *notDA*: *DenyAll ∉ 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 sets-eq foo2*
wellformed-policy1-charn wp1-aux1aa wp1-eq wp1-p wp1-s)

{

case *DenyAll*

have *mr-p-is-DenyAll*: *applied-rule-rev Cp x p = Some DenyAll*
by (*simp add: DenyAll Some*)
hence *x-notin-tl-p*: $\forall r. r \in \text{set } (tl\ p) \longrightarrow x \notin \text{dom } (Cp\ r)$ **using** *wp1-p*
by (*auto simp: mrDenyAll-is-unique*)
hence *x-notin-tl-s*: $\forall r. r \in \text{set } (tl\ s) \longrightarrow x \notin \text{dom } (Cp\ r)$ **using** *tl-eq*
by *auto*
hence *mr-s-is-DenyAll*: *applied-rule-rev Cp 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*: *applied-rule-rev Cp x p = Some (DenyAllFromTo a b)*
by (*simp add: DenyAllFromTo Some*)
have *DA-notin-tl*: *DenyAll ∉ set (tl p)*
by (*metis WP1n-DA-notinSet wp1-p*)
have *mr-tl-p*: *applied-rule-rev Cp x p = applied-rule-rev Cp x (tl p)*
by (*metis Combinators.simps(4) DenyAllFromTo Some mrConcEnd tl-p*)
have *dom-tl-p*: $\bigwedge r. r \in \text{set } (tl\ p) \wedge x \in \text{dom } (Cp\ r) \implies$
 $r = (DenyAllFromTo\ a\ b)$
using *wp1-p aND SC wp3-p mr-p-is-DAFT*
by (*auto simp: rule-charnDAFT*)
hence *dom-tl-s*: $\bigwedge r. r \in \text{set } (tl\ s) \wedge x \in \text{dom } (Cp\ r) \implies$
 $r = (DenyAllFromTo\ a\ b)$
using *tl-eq* **by** *auto*
have *DAFT-in-tl-s*: *DenyAllFromTo a b ∈ set (tl s)* **using** *mr-tl-p*
by (*metis DenyAllFromTo mrSet mr-p-is-DAFT tl-eq*)
have *x-in-dom-DAFT*: $x \in \text{dom } (Cp\ (DenyAllFromTo\ a\ b))$

```

    by (metis mr-p-is-DAFT DenyAllFromTo mr-in-dom)
  hence mr-tl-s-is-DAFT: applied-rule-rev Cp 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: applied-rule-rev Cp x s = Some (DenyAllFromTo a b)
    using tl-s
    by (metis DA-notin-tl DenyAllFromTo EX-MR mrDA-tl
      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: applied-rule-rev Cp x p = Some (AllowPortFromTo a b c)
    by (simp add: AllowPortFromTo Some)
  hence A-in-s: AllowPortFromTo a b c ∈ set s using sets-eq
    by (auto intro: mrSet)
  have x-in-dom-A: x ∈ dom (Cp (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: aNDSetsEq)
  hence mr-s-is-A: applied-rule-rev Cp 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
}
}
qed
qed

```

lemma *C-eq-Sets*:

```

[[singleCombinators p; wellformed-policy1-strong p; wellformed-policy1-strong s;
wellformed-policy3Pr p; wellformed-policy3Pr s; allNetsDistinct p; set p = set s]] ==>
  Cp (list2FWpolicy p) x = Cp (list2FWpolicy s) x
  apply (rule C-eq-if-mr-eq)
  apply (rule C-eq-Sets-mr [symmetric])
  apply simp-all
done

```

lemma *C-eq-sorted*: [[distinct p; all-in-list p l; singleCombinators p;
wellformed-policy1-strong p; wellformed-policy3Pr p; allNetsDistinct p]] ==>
Cp (list2FWpolicy (sort p l)) = Cp (list2FWpolicy p)

```

  apply (rule ext)
  apply (rule C-eq-Sets)
  apply (auto simp: nMTSort wellformed1-alternative-sorted
    wellformed-policy3-charn wp1-eq)
done

```

```

lemma C-eq-sortedQ:  $\llbracket \text{distinct } p; \text{all-in-list } p \text{ } l; \text{singleCombinators } p; \\ \text{wellformed-policy1-strong } p; \text{wellformed-policy3Pr } p; \text{allNetsDistinct } p \rrbracket \implies \\ \text{Cp } (\text{list2FWpolicy } (\text{qsort } p \text{ } l)) = \text{Cp } (\text{list2FWpolicy } p) \\ \text{apply } (\text{rule ext}) \\ \text{apply } (\text{rule C-eq-Sets}) \\ \text{apply } (\text{auto simp: nMTSortQ wellformed1-alternative-sorted distinct-sortQ} \\ \text{wellformed-policy3-charn wp1-eq}) \\ \text{by } (\text{metis mem-def member-rec(1) member-set wellformed1-sortedQ wellformed-eq wp1-eq} \\ \text{set-qsort wp1n-tl})$ 
```

```

lemma C-eq-RS2-mr:  $\text{applied-rule-rev } \text{Cp } x \text{ } (\text{removeShadowRules2 } p) = \text{applied-rule-rev } \text{Cp } x \text{ } p$ 
proof (induct p)
case Nil thus ?thesis by simp next
case (Cons y ys) thus ?thesis
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 AllowPortFromTo Cons
  apply (cases applied-rule-rev Cp x ys = None, simp-all)
  apply (subgoal-tac x  $\notin$  dom (Cp (AllowPortFromTo a b p)))
  apply (subst mrconcNone, simp-all)
  apply (simp add: applied-rule-rev-def )
  apply (rule contra-subsetD [OF allow-deny-dom])
  apply (erule mrNoneMT, simp)
  apply (metis AllowPortFromTo mrconc)
  done
next
case False thus ?thesis using False Cons AllowPortFromTo
  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 *C-eq-None*[rule-format]: $p \neq [] \longrightarrow \text{applied-rule-rev } Cp \ x \ p = \text{None} \longrightarrow$
 $Cp \ (\text{list2FWpolicy } p) \ x = \text{None}$

apply (*simp add: applied-rule-rev-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*:

$\llbracket a \neq []; b \neq []; \text{applied-rule-rev } Cp \ x \ a = \text{None}; \text{applied-rule-rev } Cp \ x \ b = \text{None} \rrbracket \implies$
 $(Cp \ (\text{list2FWpolicy } a)) \ x = (Cp \ (\text{list2FWpolicy } b)) \ x$

by (*auto simp: C-eq-None*)

lemma *C-eq-RS2: wellformed-policy1-strong* $p \implies$

$Cp \ (\text{list2FWpolicy } (\text{removeShadowRules2 } p)) = Cp \ (\text{list2FWpolicy } p)$

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 *noneMTRS2: noneMT* $Cp \ p \implies \text{noneMT } Cp \ (\text{removeShadowRules2 } p)$

by (*auto simp: RS2Set noneMTsubset*)

lemma *CconcNone*: $\llbracket \text{dom } (Cp \ a) = \{\}; p \neq [] \rrbracket \implies$

$Cp \ (\text{list2FWpolicy } (a \ \# \ p)) \ x = Cp \ (\text{list2FWpolicy } p) \ x$

apply (*case-tac p = [], simp-all*)

apply (*case-tac x $\in \text{dom } (Cp \ (\text{list2FWpolicy } p))$*)

apply (*metis Cdom2 list2FWpolicyconc*)

apply (*metis Cp.simps(4) map-add-dom-app-simps(2) inSet-not-MT list2FWpolicyconc set-empty2*)

done

lemma *noneMTrd*[rule-format]: $\text{noneMT } Cp \ p \longrightarrow \text{noneMT } Cp \ (\text{remdups } p)$

by (*induct p, simp-all*)

lemma *DARS3*[rule-format]: $\text{DenyAll} \notin \text{set } p \longrightarrow \text{DenyAll} \notin \text{set } (\text{removeShadowRules3 } Cp \ p)$

by (*induct p, simp-all*)

lemma *DAnMT*: $\text{dom } (Cp \ \text{DenyAll}) \neq \{\}$

by (*simp add: dom-def Cp.simps PolicyCombinators.PolicyCombinators*)

lemma *DAnMT2*: $Cp \ \text{DenyAll} \neq \text{empty}$

by (*metis DAAux dom-eq-empty-conv empty-iff*)

```

lemma wp1n-RS3[rule-format,simp]: wellformed-policy1-strong  $p \longrightarrow$ 
      wellformed-policy1-strong (removeShadowRules3  $Cp$   $p$ )
apply (induct  $p$ , simp-all)
apply (rule conjI | rule impI | simp) +
apply (metis DAnMT)
apply (metis DARS3)
done

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

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

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

lemma ANDRS3[simp]:  $\llbracket$ singleCombinators  $p$ ; allNetsDistinct  $p$  $\rrbracket \Longrightarrow$ 
      allNetsDistinct (removeShadowRules3  $Cp$   $p$ )
apply (rule-tac  $b = p$  in aNDSsubset)
apply simp-all
apply (rule RS3subset)
done

lemma nlpaur:  $x \notin \text{dom } (Cp\ b) \Longrightarrow Cp\ (a \oplus b)\ x = Cp\ a\ x$ 
by (metis  $Cp.\text{sims}(4)$  map-add-dom-app-sims(3))

lemma notindom[rule-format]:  $a \in \text{set } p \longrightarrow x \notin \text{dom } (Cp\ (\text{list2FWpolicy } p)) \longrightarrow$ 
       $x \notin \text{dom } (Cp\ a)$ 
apply (induct  $p$ )
apply simp-all
apply (rule conjI | rule impI) +
apply (metis CConcStartA)
apply (rule impI) +
apply simp
apply (metis  $C\text{dom}2$  List.set.sims(2) domIff insert-absorb list.sims(2) list2FWpolicyconc
set-empty)
done

```

```

lemma C-eq-rd[rule-format]:  $p \neq [] \implies$ 
       $Cp \ (list2FWpolicy \ (remdups \ p)) = Cp \ (list2FWpolicy \ p)$ 
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 Cons apply simp
      apply (rule conjI, rule impI)
      apply (cases x  $\in$  dom (Cp (list2FWpolicy ys)))
      apply (metis Cdom2 False list2FWpolicyconc)
      apply (metis False domIff list2FWpolicyconc nlpaux notindom)
      apply (rule impI)
      apply (cases x  $\in$  dom (Cp (list2FWpolicy ys)))
      apply (subgoal-tac x  $\in$  dom (Cp (list2FWpolicy (remdups ys))))
      apply (metis Cdom2 False list2FWpolicyconc remdups-eq-nil-iff)
      apply (metis domIff)
      apply (subgoal-tac x  $\notin$  dom (Cp (list2FWpolicy (remdups ys))))
      apply (metis False list2FWpolicyconc nlpaux remdups-eq-nil-iff)
      apply (metis domIff)
    done
  qed
qed

```

```

lemma nMT-domMT:  $\llbracket \neg \text{notMTpolicy } Cp \ p; p \neq [] \rrbracket \implies r \notin \text{dom} \ (Cp \ (list2FWpolicy \ p))$ 
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 )
by (metis CconcNone domIff)
qed

```

```

lemma C-eq-RS3-aux[rule-format]:  $\text{notMTpolicy } Cp \ p \implies$ 
       $Cp \ (list2FWpolicy \ p) \ x = Cp \ (list2FWpolicy \ (\text{removeShadowRules3 } Cp \ p)) \ x$ 
proof (induct p)
  case Nil thus ?case by simp next
  case (Cons y ys) thus ?case
    proof (cases notMTpolicy Cp ys)
      case True thus ?thesis using Cons apply simp
      apply (rule conjI, rule impI, simp)
      apply (metis CconcNone True notMTpolicyimpnotMT)
      apply (rule impI, simp)
      apply (cases x  $\in$  dom (Cp (list2FWpolicy ys)))
      apply (subgoal-tac x  $\in$  dom (Cp (list2FWpolicy (removeShadowRules3 Cp ys))))
    done
  qed

```

```

apply (metis Cdom2 NMPRS3 True l2p-aux notMTpolicyimpnotMT)
apply (simp add: domIff)
apply (subgoal-tac  $x \notin \text{dom } (Cp \text{ (list2FWpolicy (removeShadowRules3 } Cp \text{ ys))))$ )
apply (metis l2p-aux l2p-aux2 nlpaux)
apply (metis domIff)
done
next
case False thus ?thesis using Cons False
  proof (cases ys = [])
    case True thus ?thesis using Cons by (simp) (rule impI, simp) next
    case False thus ?thesis using Cons False (¬ notMTpolicy Cp ys)
      apply (simp)
      apply (rule conjI | rule impI | simp)+
      apply (subgoal-tac removeShadowRules3 Cp ys = [])
      apply (subgoal-tac  $x \notin \text{dom } (Cp \text{ (list2FWpolicy ys))))$ 
      apply simp-all
      apply (metis l2p-aux nlpaux)
      apply (erule nMT-domMT, simp-all)
      by (metis SR3nMT)
  qed
qed
qed

```

```

lemma C-eq-id: wellformed-policy1-strong  $p \implies$ 
   $Cp(\text{list2FWpolicy (insertDeny } p)) = Cp \text{ (list2FWpolicy } p)$ 
apply (rule ext)
apply (rule C-eq-if-mr-eq)
apply simp-all
apply (erule mr-iD)
done

```

```

lemma C-eq-RS3: notMTpolicy Cp  $p \implies$ 
   $Cp(\text{list2FWpolicy (removeShadowRules3 } Cp \text{ } p)) = Cp \text{ (list2FWpolicy } p)$ 
apply (rule ext)
by (erule C-eq-RS3-aux[symmetric])

```

```

lemma NMPrd[rule-format]: notMTpolicy Cp  $p \longrightarrow \text{notMTpolicy } Cp \text{ (remdups } p)$ 
apply (induct p, simp-all)
by (auto simp: NMPcham)

```

```

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

```



```

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

```

```

lemma list2FWpolicy2list[rule-format]: Cp (list2FWpolicy(policy2list p)) = (Cp p)
apply (rule ext)
apply (induct-tac p, simp-all)
apply (case-tac x ∈ dom (Cp (Combinators2)))
apply (metis Cdom2 CeqEnd domIff p2lNmt)
apply (metis CeqStart domIff p2lNmt nlpaux)
done

```

```

lemmas C-eq-Lemmas = noneMTRS2 noneMTrd SCp2l
      wp1n-RS2 wp1ID NMPiD wp1alternative-RS1
      p2lNmt list2FWpolicy2list wellformed-policy3-charn waux2 wp1-eq

```

```

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

```

```

lemma C-eq-All-untilSorted:
  ⟦DenyAll ∈ set (policy2list p); all-in-list (policy2list p) l;
  allNetsDistinct (policy2list p)⟧ ⟹
  Cp(list2FWpolicy (sort (removeShadowRules2 (remdups (removeShadowRules3 Cp
    (insertDeny (removeShadowRules1 (policy2list p)))))) l)) = Cp 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-untilSortedQ:
  ⟦DenyAll ∈ set (policy2list p); all-in-list (policy2list p) l;
  allNetsDistinct (policy2list p)⟧ ⟹
  Cp(list2FWpolicy (qsort (removeShadowRules2 (remdups (removeShadowRules3 Cp
    (insertDeny (removeShadowRules1 (policy2list p)))))) l)) = Cp p
apply (subst C-eq-sortedQ)
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:
   $\llbracket \text{DenyAll} \in \text{set } (\text{policy2list } p); \text{all-in-list } (\text{policy2list } p) \text{ } l; \\ \text{allNetsDistinct } (\text{policy2list } p) \rrbracket \implies \\ \text{Cp}(\text{list2FWpolicy } (\text{sort } (\text{removeShadowRules2 } (\text{remdups } (\text{removeShadowRules3 } \text{Cp } (\text{insertDeny} \\ (\text{removeShadowRules1 } (\text{policy2list } p)))))) \text{ } l)) = \text{Cp } p \\ \text{by } (\text{simp-all add: C-eq-Lemmas C-eq-subst-Lemmas})$ 
```

```

lemma C-eq-All-untilSorted-withSimpsQ:
   $\llbracket \text{DenyAll} \in \text{set } (\text{policy2list } p); \text{all-in-list } (\text{policy2list } p) \text{ } l; \\ \text{allNetsDistinct } (\text{policy2list } p) \rrbracket \implies \\ \text{Cp}(\text{list2FWpolicy } (\text{qsort } (\text{removeShadowRules2 } (\text{remdups } (\text{removeShadowRules3 } \text{Cp } (\text{insertDeny} \\ (\text{removeShadowRules1 } (\text{policy2list } p)))))) \text{ } l)) = \text{Cp } p \\ \text{by } (\text{simp-all add: C-eq-Lemmas C-eq-subst-Lemmas})$ 
```

```

lemma InDomConc[rule-format]:  $p \neq [] \longrightarrow x \in \text{dom } (\text{Cp } (\text{list2FWpolicy } (p))) \longrightarrow \\ x \in \text{dom } (\text{Cp } (\text{list2FWpolicy } (a\#p)))$ 
apply (induct p)
apply simp-all
apply (case-tac p = [])
apply (simp-all add: dom-def Cp.simps)
done

```

```

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

```

```

lemma src-in-sdnets[rule-format]:  $\neg \text{member DenyAll } x \longrightarrow p \in \text{dom } (\text{Cp } x) \longrightarrow \\ \text{subnetsOfAdr } (\text{src } p) \cap (\text{fst-set } (\text{sdnets } x)) \neq \{\}$ 
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  $p \in \text{dom } (Cp \text{ Combinators2})$ )
apply simp-all
apply (rule subnetAux)
apply assumption
apply (auto simp: PLemmas)
done

lemma dest-in-sdnets[rule-format]:  $\neg \text{member } \text{DenyAll } x \longrightarrow p \in \text{dom } (Cp \ x) \longrightarrow$ 
 $\text{subnetsOfAdr } (\text{dest } p) \cap (\text{snd-set } (\text{sdnets } x)) \neq \{\}$ 
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  $p \in \text{dom } (Cp \text{ Combinators2})$ )
apply simp-all
apply (rule subnetAux)
apply assumption
apply (auto simp: PLemmas)
done

lemma sdnets-in-subnets[rule-format]:  $p \in \text{dom } (Cp \ x) \longrightarrow \neg \text{member } \text{DenyAll } x \longrightarrow$ 
 $(\exists (a,b) \in \text{sdnets } x. a \in \text{subnetsOfAdr } (\text{src } p) \wedge b \in \text{subnetsOfAdr } (\text{dest } p))$ 
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  $p \in \text{dom } (Cp \ (\text{Combinators2}))$ )
apply simp-all
apply (auto simp: PLemmas subnetsOfAdr-def)
done

lemma disjSD-no-p-in-both[rule-format]:
 $\llbracket \text{disjSD-2 } x \ y; \neg \text{member } \text{DenyAll } x; \neg \text{member } \text{DenyAll } y;$ 
 $p \in \text{dom } (Cp \ x); p \in \text{dom } (Cp \ y) \rrbracket \Longrightarrow \text{False}$ 
apply (rule-tac  $A = \text{sdnets } x$  and  $B = \text{sdnets } y$  and  $D = \text{src } p$ 
and  $F = \text{dest } p$  in tndFalse)
by (auto simp: dest-in-sdnets src-in-sdnets sdnets-in-subnets disjSD-2-def)

```

```

lemma list2FWpolicy-eq:  $zs \neq [] \implies$ 
       $Cp (list2FWpolicy (x \oplus y \# z)) p = Cp (x \oplus list2FWpolicy (y \# z)) p$ 
by (metis ConcAssoc l2p-aux list2FWpolicy.simps(2))

lemma dom-sep[rule-format]:  $x \in dom (Cp (list2FWpolicy p)) \longrightarrow$ 
       $x \in dom (Cp (list2FWpolicy (separate p)))$ 
apply (rule separate.induct) back
apply simp-all
apply (rule conjI)
apply (rule impI)+
apply simp
apply (thin-tac False  $\implies$  ?S)
apply (drule mp)
apply (case-tac x  $\in$  dom (Cp (DenyAllFromTo v va)))
apply (metis CConcStartA domIff l2p-aux2
      list2FWpolicyconc not-Cons-self)
apply (subgoal-tac x  $\in$  dom (Cp (list2FWpolicy (y #z))))
apply (metis CConcStartA Cdom2 InDomConc domIff l2p-aux2 list2FWpolicyconc nlpaux)
apply (subgoal-tac x  $\in$  dom (Cp (list2FWpolicy ((DenyAllFromTo v va)#y#z))))
apply (simp add: dom-def Cp.simps)
apply simp
apply simp
apply (rule impI)+
apply simp
apply (thin-tac False  $\implies$  ?S)
apply (case-tac x  $\in$  dom (Cp (DenyAllFromTo v va)))
apply simp-all
apply (subgoal-tac x  $\in$  dom (Cp (list2FWpolicy (y #z))))
apply (metis InDomConc sepnMT list.simps(2))
apply (subgoal-tac x  $\in$  dom (Cp (list2FWpolicy ((DenyAllFromTo v va)#y#z))))
apply (simp add: dom-def Cp.simps)
apply simp
apply (rule impI | rule conjI)+
apply simp
apply (case-tac x  $\in$  dom (Cp (AllowPortFromTo v va vb)))
apply (metis CConcStartA domIff l2p-aux2
      list2FWpolicyconc not-Cons-self)
apply (subgoal-tac x  $\in$  dom (Cp (list2FWpolicy (y #z))))
apply simp
apply (metis CConcStartA Cdom2 InDomConc domIff l2p-aux2 list2FWpolicyconc nlpaux)
apply (simp add: dom-def Cp.simps)
apply (rule impI)+
apply simp-all
apply (thin-tac False  $\implies$  ?S)
apply (case-tac x  $\in$  dom (Cp (AllowPortFromTo v va vb)))
apply simp-all
apply (subgoal-tac x  $\in$  dom (Cp (list2FWpolicy (y #z))))

```

```

apply simp
apply (metis Conc-not-MT InDomConc sepnMT)
apply (metis domIff nlpaux)
apply (rule conjI | rule impI)+
apply simp
apply (thin-tac False  $\implies$  ?S)
apply (drule mp)
apply (case-tac  $x \in \text{dom } (Cp ((v \oplus va))))$ )
apply (metis Cp.simps(4) CConcStartA ConcAssoc domIff
  list2FWpolicy2list list2FWpolicyconc p2lNmt)
defer 1
apply simp-all
apply (rule impI)+
apply simp
apply (thin-tac False  $\implies$  ?S)
apply (case-tac  $x \in \text{dom } (Cp ((v \oplus va))))$ )
apply (metis CConcStartA)
apply (drule mp)
apply (simp add: Cp.simps dom-def)
apply (metis InDomConc list.simps(2) sepnMT)
apply (subgoal-tac  $x \in \text{dom } (Cp (\text{list2FWpolicy } (y \# z)))$ )
apply (case-tac  $x \in \text{dom } (Cp y)$ )
apply simp-all
apply (metis CConcStartA Cdom2 ConcAssoc domIff)
apply (metis InDomConc domIff l2p-aux2 list2FWpolicyconc nlpaux)
apply (case-tac  $x \in \text{dom } (Cp y)$ )
apply simp-all
apply (metis domIff nlpaux)
done

```

```

lemma domdConcStart[rule-format]:  $x \in \text{dom } (Cp (\text{list2FWpolicy } (a \# b))) \implies$ 
   $x \notin \text{dom } (Cp (\text{list2FWpolicy } b))$ 
   $\implies x \in \text{dom } (Cp (a))$ 
apply (induct b, simp-all)
apply (auto simp: PLemmas)
done

```

```

lemma sep-dom2-aux:  $\llbracket x \in \text{dom } (Cp (\text{list2FWpolicy } (a \oplus y \# z))) \rrbracket$ 
   $\implies x \in \text{dom } (Cp (a \oplus \text{list2FWpolicy } (y \# z)))$ 
apply auto
by (metis list2FWpolicy-eq p2lNmt)

```

```

lemma sep-dom2-aux2:
   $\llbracket (x \in \text{dom } (Cp (\text{list2FWpolicy } (\text{separate } (y \# z)))) \implies$ 
     $x \in \text{dom } (Cp (\text{list2FWpolicy } (y \# z)))$ ;
     $x \in \text{dom } (Cp (\text{list2FWpolicy } (a \# \text{separate } (y \# z)))) \rrbracket$ 
   $\implies x \in \text{dom } (Cp (\text{list2FWpolicy } (a \oplus y \# z)))$ 
by (metis CConcStartA InDomConc domdConcStart list.simps(2) list2FWpolicy.simps(2) list2FWpolicyconc)

```

```

lemma sep-dom2[rule-format]:
   $x \in \text{dom} (Cp (\text{list2FWpolicy} (\text{separate } p))) \longrightarrow x \in \text{dom} (Cp (\text{list2FWpolicy} (p)))$ 
apply (rule separate.induct)
by (simp-all add: sep-dom2-aux sep-dom2-aux2)

lemma sepDom:  $\text{dom} (Cp (\text{list2FWpolicy } p)) = \text{dom} (Cp (\text{list2FWpolicy} (\text{separate } p)))$ 
apply (rule equalityI)
by (rule subsetI, (erule dom-sep|erule sep-dom2))+

lemma C-eq-s-ext[rule-format]:  $p \neq [] \longrightarrow$ 
   $Cp (\text{list2FWpolicy} (\text{separate } p)) \ a = Cp (\text{list2FWpolicy } p) \ a$ 
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  $a \in \text{dom} (Cp (\text{list2FWpolicy } x))$ )
  apply (subgoal-tac  $a \in \text{dom} (Cp (\text{list2FWpolicy} (\text{separate } x)))$ )
  apply (metis Cdom2 list2FWpolicyconc sepDom sepnMT)
  apply (metis sepDom)
  apply (subgoal-tac  $a \notin \text{dom} (Cp (\text{list2FWpolicy} (\text{separate } x)))$ )
  apply (subst list2FWpolicyconc)
  apply (simp add: sepnMT)
  apply (subst list2FWpolicyconc)
  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 list2FWpolicyconc)
  apply (metis not-Cons-self sepnMT)
  apply (metis Cp.simps(4) CConcStartaux Cdom2 domIff)
  apply (rule conjI|rule impI|simp)+
  apply (erule list2FWpolicy-eq)
  apply (rule impI, simp)
  apply (subst list2FWpolicyconc)
  apply (metis list.simps(2) sepnMT)
  by (metis Cp.simps(4) CConcStartaux Cdom2 domIff)
next
case goal3 thus ?case
apply simp

```

```

    apply (cases z = [])
    apply simp-all
    apply (rule conjI|rule impI|simp)+
    apply (subst list2FWpolicyconc)
    apply (metis not-Cons-self sepnMT)
    apply (metis Cp.simps(4) CConcStartaux Cdom2 domIff)
    apply (rule conjI|rule impI|simp)+
    apply (erule list2FWpolicy-eq)
    apply (rule impI, simp)
    apply (subst list2FWpolicyconc)
    apply (metis list.simps(2) sepnMT)
    by (metis Cp.simps(4) CConcStartaux Cdom2 domIff)
next
case goal4 thus ?case
apply simp
  apply (cases z = [])
  apply simp-all
  apply (rule conjI|rule impI|simp)+
  apply (subst list2FWpolicyconc)
  apply (metis not-Cons-self sepnMT)
  apply (metis Cp.simps(4) CConcStartaux Cdom2 domIff)
  apply (rule conjI|rule impI|simp)+
  apply (erule list2FWpolicy-eq)
  apply (rule impI, simp)
  apply (subst list2FWpolicyconc)
  apply (metis list.simps(2) sepnMT)
  by (metis Cp.simps(4) CConcStartaux Cdom2 domIff)
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:  $p \neq [] \implies Cp (list2FWpolicy (separate p)) = Cp (list2FWpolicy p)$ 
apply (rule ext)
apply (rule C-eq-s-ext)
apply simp
done

lemma sortnMTQ:  $p \neq [] \implies qsort p l \neq []$ 
by (metis set-sortQ setnMT)

lemmas C-eq-Lemmas-sep =
  C-eq-Lemmas sortnMT sortnMTQ RS2-NMT NMPrd NMPRS3

```

lemma *C-eq-until-separated*:
 $\llbracket \text{DenyAll} \in \text{set } (\text{policy2list } p); \text{all-in-list } (\text{policy2list } p) \text{ l};$
 $\text{allNetsDistinct } (\text{policy2list } p) \rrbracket \implies$
 $\text{Cp } (\text{list2FWpolicy } (\text{separate } (\text{sort } (\text{removeShadowRules2 } (\text{remdups } (\text{removeShadowRules3 } \text{Cp}$
 $(\text{insertDeny } (\text{removeShadowRules1 } (\text{policy2list } p)))))) \text{ l})) = \text{Cp } p$
apply (*subst C-eq-s*)
apply (*simp-all add: C-eq-Lemmas-sep*)
apply (*rule C-eq-All-untilSorted*)
apply *simp-all*
done

lemma *C-eq-until-separatedQ*:
 $\llbracket \text{DenyAll} \in \text{set } (\text{policy2list } p); \text{all-in-list } (\text{policy2list } p) \text{ l};$
 $\text{allNetsDistinct } (\text{policy2list } p) \rrbracket \implies$
 $\text{Cp } (\text{list2FWpolicy } (\text{separate } (\text{qsort } (\text{removeShadowRules2 } (\text{remdups } (\text{removeShadowRules3 } \text{Cp}$
 $(\text{insertDeny } (\text{removeShadowRules1 } (\text{policy2list } p)))))) \text{ l})) = \text{Cp } p$
apply (*subst C-eq-s*)
apply (*simp-all add: C-eq-Lemmas-sep*)
apply (*rule C-eq-All-untilSortedQ*)
apply *simp-all*
done

lemma *domID[rule-format]*: $p \neq [] \wedge x \in \text{dom}(\text{Cp}(\text{list2FWpolicy } p)) \longrightarrow$
 $x \in \text{dom } (\text{Cp}(\text{list2FWpolicy}(\text{insertDenies } p)))$

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-all add: Cp.simps dom-def*) +
by *auto*
next
case *goal2* **then show** ?*case*
proof(*cases x ∈ dom(Cp(list2FWpolicy 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 (Cp (list2FWpolicy (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 list2FWpolicyconc nlpaux Cdom2)
  next
    case (AllowPortFromTo src dest port) then show ?case
  by(simp,metis domIff CConcStartA list2FWpolicyconc nlpaux Cdom2)
  next
    case (Conc - -) then show ?case
  apply simp
by (metis CConcStartA Cdom2 Conc(5) ConcAssoc domIff domdConcStart goal2(2))
  qed
qed
qed
qed
qed

```

lemma *DA-is-deny*:

```

 $x \in \text{dom } (Cp \ (DenyAllFromTo \ a \ b \oplus DenyAllFromTo \ b \ a \oplus DenyAllFromTo \ a \ b)) \implies$ 
 $Cp \ (DenyAllFromTo \ a \ b \oplus DenyAllFromTo \ b \ a \oplus DenyAllFromTo \ a \ b) \ x = \text{Some } (\text{deny } ())$ 
apply (case-tac  $x \in \text{dom } (Cp \ (DenyAllFromTo \ a \ b))$ )
apply (simp-all add: PLemmas)
apply (simp-all split: if-splits)
done

```

lemma *iDdomAux*[*rule-format*]:

```

 $p \neq [] \longrightarrow x \notin \text{dom } (Cp \ (\text{list2FWpolicy } p)) \longrightarrow$ 
 $x \in \text{dom } (Cp \ (\text{list2FWpolicy } (\text{insertDenies } p))) \longrightarrow$ 
 $Cp \ (\text{list2FWpolicy } (\text{insertDenies } p)) \ x = \text{Some } (\text{deny } ())$ 
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 DenyAllFromTo Cons
    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 ∈ dom (Cp (list2FWpolicy (insertDenies ys))))
        apply simp-all
        apply (metis Cdom2 DenyAllFromTo goal2(5) idNMT list2FWpolicyconc)
        apply (subgoal-tac Cp (list2FWpolicy (DenyAllFromTo a b ⊕
          DenyAllFromTo b a ⊕ DenyAllFromTo a b#insertDenies ys)) x =
          Cp ((DenyAllFromTo a b ⊕ DenyAllFromTo b a ⊕ DenyAllFromTo a b)) x)
        apply simp
        apply (rule DA-is-deny)
        apply (metis DenyAllFromTo domdConcStart goal2(4))
        apply (metis DenyAllFromTo l2p-aux2 list2FWpolicyconc nlpaux)
        done
      qed
    next
    case (AllowPortFromTo a b c) then show ?thesis using Cons AllowPortFromTo
      proof (cases ys = [])
        case goal1 then show ?case
          apply simp
          apply (rule impI)+
          apply (subgoal-tac x ∈ dom (Cp (DenyAllFromTo a b ⊕ 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 ∈ dom (Cp (list2FWpolicy (insertDenies ys))))
          apply simp-all
          apply (metis AllowPortFromTo Cdom2 goal2(4) idNMT list2FWpolicyconc)
          apply (subgoal-tac Cp (list2FWpolicy (DenyAllFromTo a b ⊕
            DenyAllFromTo b a ⊕ AllowPortFromTo a b c#insertDenies ys)) x =
            Cp ((DenyAllFromTo a b ⊕ DenyAllFromTo b a)) x)
          apply simp
          defer 1
          apply (metis AllowPortFromTo CConcStartA ConcAssoc goal2(4) idNMT
            list2FWpolicyconc 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  $x \in \text{dom } (Cp (\text{DenyAllFromTo } (\text{first-srcNet } a) (\text{first-destNet } a) \oplus \text{DenyAllFromTo } (\text{first-destNet } a) (\text{first-srcNet } a))))$ )
    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  $x \in \text{dom } (Cp (\text{list2FWpolicy } (\text{insertDenies } ys)))$ )
    apply (metis Cdom2 Conc Cons InDomConc goal2(2) idNMT list2FWpolicyconc)
    apply (subgoal-tac  $Cp (\text{list2FWpolicy } (\text{DenyAllFromTo } (\text{first-srcNet } a) (\text{first-destNet } a) \oplus \text{DenyAllFromTo } (\text{first-destNet } a) (\text{first-srcNet } a) \oplus a \oplus b \# \text{insertDenies } ys))$   $x = Cp ((\text{DenyAllFromTo } (\text{first-srcNet } a) (\text{first-destNet } a) \oplus \text{DenyAllFromTo } (\text{first-destNet } a) (\text{first-srcNet } a) \oplus a \oplus b))$   $x$ )
    apply simp
    defer 1
    apply (metis Conc l2p-aux2 list2FWpolicyconc nlpaux)
    apply (subgoal-tac  $Cp ((\text{DenyAllFromTo } (\text{first-srcNet } a) (\text{first-destNet } a) \oplus \text{DenyAllFromTo } (\text{first-destNet } a) (\text{first-srcNet } a) \oplus a \oplus b))$   $x = Cp ((\text{DenyAllFromTo } (\text{first-srcNet } a) (\text{first-destNet } a) \oplus \text{DenyAllFromTo } (\text{first-destNet } a) (\text{first-srcNet } a))$   $x$ )
    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]:  $p \neq [] \longrightarrow x \notin \text{dom } (Cp (\text{list2FWpolicy } p)) \longrightarrow Cp (\text{DenyAll} \oplus \text{list2FWpolicy } (\text{insertDenies } p)) x = Cp \text{DenyAll } x$ 
  apply (case-tac  $x \in \text{dom } (Cp (\text{list2FWpolicy } (\text{insertDenies } p)))$ )
  apply (rule impI)+
  apply (metis Cp.simps(1) deny-all-def iDdomAux Cdom2)
  apply (rule impI)+
  apply (subst nlpaux)
  apply simp-all
done

```

```

lemma inDomConc:  $\llbracket x \notin \text{dom } (Cp \ a); x \notin \text{dom } (Cp \ (\text{list2FWpolicy } p)) \rrbracket \implies$ 
 $x \notin \text{dom } (Cp \ (\text{list2FWpolicy}(a \# p)))$ 
by (metis domdConcStart)

lemma domsdisj[rule-format]:  $p \neq [] \longrightarrow (\forall \ x \ s. s \in \text{set } p \wedge x \in \text{dom } (Cp \ A) \longrightarrow$ 
 $x \notin \text{dom } (Cp \ s)) \longrightarrow y \in \text{dom } (Cp \ A) \longrightarrow$ 
 $y \notin \text{dom } (Cp \ (\text{list2FWpolicy } p))$ 

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:
 $\llbracket p \neq []; \text{noDenyAll } (a \# p); \text{separated } (a \# p);$ 
 $x \in \text{dom } (Cp \ (\text{DenyAllFromTo } (\text{first-srcNet } a) \ (\text{first-destNet } a) \oplus$ 
 $\text{DenyAllFromTo } (\text{first-destNet } a) \ (\text{first-srcNet } a) \oplus a)) \rrbracket \implies$ 
 $x \notin \text{dom } (Cp \ (\text{list2FWpolicy } p))$ 
apply (rule-tac A = ( $\text{DenyAllFromTo } (\text{first-srcNet } a) \ (\text{first-destNet } a) \oplus$ 
 $\text{DenyAllFromTo } (\text{first-destNet } a) \ (\text{first-srcNet } a) \oplus a$ ) in domsdisj)
apply simp-all
apply (rule notI)
apply (rule-tac p = xa and  $x = (\text{DenyAllFromTo } (\text{first-srcNet } a) \ (\text{first-destNet } a)$ 
 $\oplus \text{DenyAllFromTo } (\text{first-destNet } a) \ (\text{first-srcNet } a) \oplus a)$  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 noneMTsep[rule-format]: noneMT Cp p  $\longrightarrow$  noneMT Cp (separate p)
apply (rule separate.induct) back
apply (simp-all add: Cp.simps map-add-le-mapE map-le-antisym)
done

```

```

lemma dom-id:
  [[noDenyAll (a#p); separated (a#p); p ≠ []; x ∉ dom (Cp (list2FWpolicy p));
   x ∈ dom (Cp (a))]]
    ⇒ x ∉ dom (Cp (list2FWpolicy (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: Cp.simps)
done

lemma C-eq-iD-aux2[rule-format]:
  noDenyAll1 p →
  separated p →
  p ≠ [] →
  x ∈ dom (Cp (list2FWpolicy p)) →
  Cp(list2FWpolicy (insertDenies p)) x = Cp(list2FWpolicy p) x
proof (induct p)
case Nil thus ?case by simp
next
case (Cons y ys) thus ?case using Cons
proof (cases y)
case DenyAll thus ?thesis using Cons DenyAll apply simp
  apply (case-tac ys = [])
  apply simp-all
  apply (case-tac x ∈ dom (Cp (list2FWpolicy ys)))
  apply simp-all
apply (metis Cdom2 domID idNMT list2FWpolicyconc noDA1eq)
apply (metis DenyAll iD-isD idNMT list2FWpolicyconc nlpaux)
  done
next
case (DenyAllFromTo a b) thus ?thesis using Cons 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 (Cp (list2FWpolicy ys)))
  apply simp-all
  apply (drule mp)

```

```

apply (metis noDA1eq)
apply (case-tac  $x \in \text{dom } (Cp \text{ (list2FWpolicy (insertDenies ys))})$ )
apply (metis Cdom2 DenyAllFromTo idNMT list2FWpolicyconc)
apply (metis domID)
apply (case-tac  $x \in \text{dom } (Cp \text{ (list2FWpolicy (insertDenies ys))})$ )
apply (subgoal-tac  $Cp \text{ (list2FWpolicy (DenyAllFromTo } a \ b \oplus \text{ DenyAllFromTo } b \ a \oplus$ 
     $\text{DenyAllFromTo } a \ b \ \# \text{ insertDenies ys)) } x = \text{Some (deny ())}$ )
apply simp-all
apply (subgoal-tac  $Cp \text{ (list2FWpolicy (DenyAllFromTo } a \ b \ \# \text{ ys)) } x =$ 
     $Cp \text{ ((DenyAllFromTo } a \ b)) \ x$ )
apply (simp add: PLemmas, simp split: if-splits)
apply (metis list2FWpolicyconc nlpaux)
apply (metis Cdom2 DenyAllFromTo iD-isD iDdomAux idNMT list2FWpolicyconc)
apply (metis Cdom2 DenyAllFromTo domIff idNMT list2FWpolicyconc nlpaux)
done
next
case (AllowPortFromTo a b c) thus ?thesis using AllowPortFromTo Cons 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  $x \in \text{dom } (Cp \text{ (list2FWpolicy ys))})$ 
apply simp-all
apply (drule mp)
apply (metis noDA1eq)
apply (case-tac  $x \in \text{dom } (Cp \text{ (list2FWpolicy (insertDenies ys))})$ )
apply (metis Cdom2 AllowPortFromTo idNMT list2FWpolicyconc)
apply (metis domID)
apply (subgoal-tac  $x \in \text{dom } (Cp \text{ (AllowPortFromTo } a \ b \ c))$ )
apply (case-tac  $x \notin \text{dom } (Cp \text{ (list2FWpolicy (insertDenies ys))})$ )
apply simp-all
apply (metis AllowPortFromTo Cdom2 ConcAssoc l2p-aux2 list2FWpolicyconc nlpaux)
apply (metis AllowPortFromTo Combinators.simps(6) member.simps(4)
    dom-id noDenyAll.simps(1) separated.simps(1))
apply (metis AllowPortFromTo domdConcStart)
done
next
case (Conc a b) thus ?thesis using Cons Conc
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  $x \in \text{dom } (Cp \text{ (list2FWpolicy ys))})$ 
apply simp-all
apply (drule mp)
apply (metis noDA1eq)
apply (case-tac  $x \in \text{dom } (Cp \text{ (} a \oplus b))$ )
apply (case-tac  $x \notin \text{dom } (Cp \text{ (list2FWpolicy (insertDenies ys))})$ )

```

```

apply simp-all
apply (subst list2FWpolicyconc)
apply (rule idNMT, simp)
apply (metis domID)
apply (metis Cdom2 Conc idNMT list2FWpolicyconc)
apply (metis Cdom2 Conc domIff idNMT list2FWpolicyconc )
apply (case-tac x ∈ dom (Cp (a ⊕ b)))
apply (case-tac x ∉ dom (Cp (list2FWpolicy (insertDenies ys))))
apply simp-all
apply (subst list2FWpolicyconc)
apply (rule idNMT, simp)
apply (metis Cdom2 Conc ConcAssoc list2FWpolicyconc nlpaux)
apply (metis Conc member.simps(1) dom-id
        noDenyAll.simps(1) separated.simps(1))
apply (metis Conc domdConcStart)
done
qed
qed

lemma C-eq-iD:  $\llbracket \text{separated } p; \text{noDenyAll1 } p; \text{wellformed-policy1-strong } p \rrbracket \implies$ 
 $Cp (\text{list2FWpolicy } (\text{insertDenies } p)) = Cp (\text{list2FWpolicy } p)$ 
apply (rule ext)
apply (rule C-eq-iD-aux2)
apply simp-all
apply (subgoal-tac DenyAll ∈ set p)
apply (metis CConcStartA DAAux wp1n-tl)
apply (erule wp1-aux1aa)
done

lemma noDAsortQ[rule-format]:  $\text{noDenyAll1 } p \longrightarrow \text{noDenyAll1 } (q\text{sort } p \ l)$ 
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-sortQ)
apply (rule impI)
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
by (metis append-Cons append-Nil qsort.simps(2) set-qsort)

```

```

lemma NetsCollectedSortQ: distinct p  $\implies$  noDenyAll1 p  $\implies$  all-in-list p l  $\implies$ 
    singleCombinators p  $\implies$  NetsCollected (qsort p l)
apply (rule-tac l = l in NetsCollectedSorted)
apply (rule noDASortQ)
apply simp-all
apply (rule-tac b=p in all-in-listSubset)
apply simp-all
apply (rule sort-is-sortedQ)
apply simp-all
done

```

```

lemmas CLemmas = nMTSort nMTSortQ noneMTRS2 noneMTrd
    noDASort noDASortQ nDASC wp1-eq wp1ID
    SCp2l ANDSep wp1n-RS2
    OTNSEp OTNSC noDA1sep wp1-alternativesep wellformed-eq
    wellformed1-alternative-sorted

```

```

lemmas C-eqLemmas-id = CLemmas NC2Sep NetsCollectedSep
    NetsCollectedSort NetsCollectedSortQ separatedNC

```

```

lemma C-eq-Until-InsertDenies:  $\llbracket \text{DenyAll} \in \text{set } (\text{policy2list } p); \text{all-in-list } (\text{policy2list } p) \text{ l}; \text{allNetsDistinct } (\text{policy2list } p) \rrbracket \implies$ 
    Cp (list2FWpolicy ((insertDenies (separate (sort (removeShadowRules2 (remdups
    (removeShadowRules3 Cp (insertDeny (removeShadowRules1 (policy2list p)))))) l)))))) =
    Cp p
apply (subst C-eq-iD)
apply (simp-all add: C-eqLemmas-id)
apply (rule C-eq-until-separated)
apply simp-all
done

```

```

lemma C-eq-Until-InsertDeniesQ:  $\llbracket \text{DenyAll} \in \text{set } (\text{policy2list } p); \text{all-in-list } (\text{policy2list } p) \text{ l}; \text{allNetsDistinct } (\text{policy2list } p) \rrbracket \implies$ 
    Cp (list2FWpolicy ((insertDenies (separate (qsort (removeShadowRules2 (remdups
    (removeShadowRules3 Cp (insertDeny (removeShadowRules1 (policy2list p)))))) l)))))) =
    Cp p
apply (subst C-eq-iD)
apply (simp-all add: C-eqLemmas-id)

```



```

apply (metis WP1rd set-qsort wellformed1-sortedQ wellformed-eq wp1ID wp1-alternativesep wp1-aux1aa
wp1n-RS2 wp1n-RS3)
apply (rule C-eq-until-separatedQ)
apply simp-all
done

```

```

lemma C-eq-RD-aux[rule-format]: Cp (p) x = Cp (removeDuplicates p) x
apply (induct p)
apply simp-all
apply (rule conjI, rule impI)
apply (metis Cdom2 domIff nlpaux not-in-member)
apply (metis Cp.simps(4) CConcStartaux Cdom2 domIff)
done

```

```

lemma C-eq-RAD-aux[rule-format]:
  p ≠ [] ⟶ Cp (list2FWpolicy p) x = Cp (list2FWpolicy (removeAllDuplicates p)) x
apply (induct p)
apply simp-all
apply (case-tac p = [])
apply simp-all
apply (metis C-eq-RD-aux)
apply (subst list2FWpolicyconc)
apply simp
apply (case-tac x ∈ dom (Cp (list2FWpolicy p)))
apply (subst list2FWpolicyconc)
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 list2FWpolicyconc)
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 ≠ [] ⟹ Cp (list2FWpolicy p) = Cp (list2FWpolicy (removeAllDuplicates p))
apply (rule ext)
apply (erule C-eq-RAD-aux)

```

done

lemma *C-eq-compile*:

```
[[DenyAll ∈ set (policy2list p); all-in-list (policy2list p) l;  
  allNetsDistinct (policy2list p)]] ⇒  
Cp (list2FWpolicy (removeAllDuplicates (insertDenies (separate (sort  
(removeShadowRules2 (remdups (removeShadowRules3 Cp (insertDeny  
(removeShadowRules1 (policy2list p)))))) l)))) = Cp 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
```

lemma *C-eq-compileQ*:

```
[[DenyAll ∈ set (policy2list p); all-in-list (policy2list p) l;  
  allNetsDistinct (policy2list p)]] ⇒  
Cp (list2FWpolicy (removeAllDuplicates (insertDenies (separate (qsort  
(removeShadowRules2 (remdups (removeShadowRules3 Cp (insertDeny  
(removeShadowRules1 (policy2list p)))))) l)))) = Cp p  
apply (subst C-eq-RAD[symmetric])  
apply (rule idNMT)  
apply (metis WP1rd sepnMT sortnMTQ wellformed-policy1-strong.simps(1) wp1ID wp1n-RS2  
wp1n-RS3)  
apply (rule C-eq-Until-InsertDeniesQ)  
apply simp-all  
done
```

lemma *C-eq-normalizePr*:

```
[[DenyAll ∈ set (policy2list p);  
  allNetsDistinct (policy2list p);  
  all-in-list (policy2list p) (Nets-List p)]] ⇒  
Cp (list2FWpolicy (normalizePr p)) = Cp p  
apply (simp add: normalizePr-def)  
apply (rule C-eq-compile)  
apply simp-all  
done
```

lemma *C-eq-normalizePrQ*:

```
[[DenyAll ∈ set (policy2list p);
```

```

allNetsDistinct (policy2list p);
all-in-list (policy2list p) (Nets-List p)]  $\implies$ 
Cp (list2FWpolicy (normalizePrQ p)) = Cp p
apply (simp add: normalizePrQ-def)
apply (rule C-eq-compileQ)
apply simp-all
done

```

```

lemma domSubset3: dom (Cp (DenyAll  $\oplus$  x)) = dom (Cp (DenyAll))
apply (simp add: PLemmas split-tupled-all split: option.splits)
done

```

```

lemma domSubset4: dom (Cp (DenyAllFromTo x y  $\oplus$  DenyAllFromTo y x  $\oplus$  AllowPortFromTo
x y dn)) =
      dom (Cp (DenyAllFromTo x y  $\oplus$  DenyAllFromTo y x))
apply (simp add: PLemmas )
apply (simp split: option.splits decision.splits)
apply auto
done

```

```

lemma domSubset5:
  dom (Cp (DenyAllFromTo x y  $\oplus$  DenyAllFromTo y x  $\oplus$  AllowPortFromTo y x dn)) =
    dom (Cp (DenyAllFromTo x y  $\oplus$  DenyAllFromTo y x))
apply (simp add: PLemmas)
apply (simp split: option.splits decision.splits)
apply auto
done

```

```

lemma domSubset1: dom (Cp (DenyAllFromTo one two  $\oplus$  DenyAllFromTo two one  $\oplus$  Allow-
PortFromTo one two dn  $\oplus$  x)) =
      dom (Cp (DenyAllFromTo one two  $\oplus$  DenyAllFromTo two one  $\oplus$  x))
apply (simp add: PLemmas)
apply (simp split: option.splits decision.splits)
apply (simp add: allow-all-def deny-all-def)
apply auto
done

```

```

lemma domSubset2:
  dom (Cp (DenyAllFromTo one two  $\oplus$  DenyAllFromTo two one  $\oplus$  AllowPortFromTo two one
dn  $\oplus$  x)) =

```

```

      dom (Cp (DenyAllFromTo one two  $\oplus$  DenyAllFromTo two one  $\oplus$  x))
apply (simp add: PLemmas)
apply (simp split: option.splits decision.splits)
apply (simp add: allow-all-def deny-all-def)
apply auto
done

lemma ConcAssoc2: Cp (X  $\oplus$  Y  $\oplus$  ((A  $\oplus$  B)  $\oplus$  D)) = Cp (X  $\oplus$  Y  $\oplus$  A  $\oplus$  B  $\oplus$  D)
apply (simp add: Cp.simps)
done

lemma ConcAssoc3: Cp (X  $\oplus$  ((Y  $\oplus$  A)  $\oplus$  D)) = Cp (X  $\oplus$  Y  $\oplus$  A  $\oplus$  D)
apply (simp add: Cp.simps)
done

lemma RS3-NMT[rule-format]: DenyAll  $\in$  set p  $\longrightarrow$ 
  removeShadowRules3 Cp p  $\neq$  []
apply (induct-tac p)
apply (simp-all add: PLemmas)
done

lemma norm-notMT: DenyAll  $\in$  set (policy2list p)  $\implies$  normalizePr p  $\neq$  []
apply (simp add: normalizePr-def)
apply (rule rADnMT)
apply (rule idNMT)
apply (rule sepnMT)
apply (rule sortnMT)
apply (rule RS2-NMT)
apply (rule remDupsNMT)
apply (rule RS3-NMT)
apply (rule DAiniD)
done

lemma norm-notMTQ: DenyAll  $\in$  set (policy2list p)  $\implies$  normalizePrQ p  $\neq$  []
apply (simp add: normalizePrQ-def)
apply (rule rADnMT)
apply (rule idNMT)
apply (rule sepnMT)
apply (rule sortnMTQ)
apply (rule RS2-NMT)
apply (rule remDupsNMT)
apply (rule RS3-NMT)
apply (rule DAiniD)
done

```

```

lemma domDA: dom (Cp (DenyAll  $\oplus$  A)) = dom (Cp (DenyAll))
apply (rule domSubset3)
done

```

```

lemmas domain-reasoningPr = domDA ConcAssoc2 domSubset1 domSubset2
  domSubset3 domSubset4 domSubset5 domSubsetDistr1
  domSubsetDistr2 domSubsetDistrA domSubsetDistrD coerc-assoc ConcAssoc
  ConcAssoc3

```

The following lemmas help with the normalisation

```

lemma list2policyR-Start[rule-format]: p  $\in$  dom (Cp a)  $\longrightarrow$ 
  Cp (list2policyR (a # list)) p = Cp a p
apply (rule list2policyR.induct) back
apply (auto simp: Cp.simps dom-def map-add-def)
done

```

```

lemma list2policyR-End: p  $\notin$  dom (Cp a)  $\implies$ 
  Cp (list2policyR (a # list)) p = (Cp a  $\oplus$  list2policy (map Cp list)) p
apply (rule list2policyR.induct)
apply (simp-all add: Cp.simps dom-def map-add-def list2policy-def split: option.splits)
done

```

```

lemma l2polR-eq-el[rule-format]: N  $\neq$  []  $\longrightarrow$ 
  Cp( list2policyR N) p = (list2policy (map Cp N)) p
apply (induct-tac N)
apply (simp-all add: list2policy-def)
apply (case-tac p  $\in$  dom (Cp a))
apply (simp add: domStart)
apply (rule list2policyR-Start)
apply simp-all
apply (rule list2policyR.induct)
apply simp-all
apply (simp-all add: Cp.simps dom-def map-add-def)
apply (simp split: option.splits)
done

```

```

lemma l2polR-eq: N  $\neq$  []  $\implies$ 
  Cp( list2policyR N) = (list2policy (map Cp N))
by (auto simp: list2policy-def l2polR-eq-el )

```

```

lemma list2FWpolicys-eq-el[rule-format]:

```

```

  Filter ≠ [] → Cp (list2policyR Filter) p = Cp (list2FWpolicy (rev Filter)) p
apply (induct-tac Filter)
apply simp-all
apply (case-tac list = [])
apply simp-all
apply (case-tac p ∈ dom (Cp a))
apply simp-all
apply (rule list2policyR-Start)
apply simp-all
apply (subgoal-tac Cp (list2policyR (a # list)) p = Cp (list2policyR list) p)
apply (subgoal-tac Cp (list2FWpolicy (rev list @ [a])) p = Cp (list2FWpolicy (rev list)) p)
apply simp
apply (rule CConcStart2)
apply simp
apply simp
apply (thin-tac ?S)
apply (case-tac list, simp-all)
apply (simp-all add: Cp.simps dom-def map-add-def)
done

```

lemma list2FWpolicys-eq:

```

  Filter ≠ [] ⇒
    Cp (list2policyR Filter) = Cp (list2FWpolicy (rev Filter))
by (rule ext, erule list2FWpolicys-eq-el)

```

lemma list2FWpolicys-eq-sym:

```

  Filter ≠ [] ⇒
    Cp (list2policyR (rev Filter)) = Cp (list2FWpolicy Filter)
by (metis list2FWpolicys-eq rev-is-Nil-conv rev-rev-ident)

```

lemma p-eq[rule-format]: p ≠ [] →

```

  list2policy (map Cp (rev p)) = Cp (list2FWpolicy p)
by (metis l2polR-eq list2FWpolicys-eq-sym rev.simps(1) rev-rev-ident)

```

lemma p-eq2[rule-format]: normalizePr x ≠ [] →

```

  Cp (list2FWpolicy (normalizePr x)) = Cp x →
  list2policy (map Cp (rev (normalizePr x))) = Cp x
apply (simp add: p-eq)
done

```

lemma p-eq2Q[rule-format]: normalizePrQ x ≠ [] →

```

  Cp (list2FWpolicy (normalizePrQ x)) = Cp x →
  list2policy (map Cp (rev (normalizePrQ x))) = Cp x
apply (simp add: p-eq)

```

done

lemma *list2listNMT*[*rule-format*]: $x \neq [] \longrightarrow \text{map sem } x \neq []$
apply (*case-tac* *x*)
apply *simp-all*
done

lemma *Norm-Distr2*:
 $r \text{ o-f } ((P \otimes_2 (\text{list2policy } Q)) \text{ o } d) =$
 $(\text{list2policy } ((P \otimes_L Q) (\text{op } \otimes_2) r d))$
apply (*rule ext*, *rule Norm-Distr-2*)
done

lemma *NATDistr*: $\llbracket N \neq []; F = \text{Cp } (\text{list2policyR } N) \rrbracket \Longrightarrow$
 $((\lambda (x,y). x) \text{ o-f } ((\text{NAT} \otimes_2 F) \text{ o } (\lambda x. (x,x)))) =$
 $(\text{list2policy } ((\text{NAT} \otimes_L (\text{map } \text{Cp } N)) (\text{op } \otimes_2)$
 $(\lambda (x,y). x) (\lambda x. (x,x)))))$
apply *simp*
apply (*simp add: l2polR-eq*)
apply (*rule ext*)
apply (*rule Norm-Distr-2*)
done

lemma *C-eq-normalize-manual*:
 $\llbracket \text{DenyAll} \in \text{set } (\text{policy2list } p);$
 $\text{allNetsDistinct } (\text{policy2list } p);$
 $\text{all-in-list } (\text{policy2list } p) \text{ l} \rrbracket \Longrightarrow$
 $\text{Cp } (\text{list2FWpolicy } (\text{normalize-manual-orderPr } p \text{ l})) = \text{Cp } p$
apply (*simp add: normalize-manual-orderPr-def*)
apply (*rule C-eq-compile*)
apply *simp-all*
done

lemma *p-eq2-manualQ*[*rule-format*]: $\text{normalize-manual-orderPrQ } x \text{ l} \neq [] \longrightarrow$
 $\text{Cp } (\text{list2FWpolicy } (\text{normalize-manual-orderPrQ } x \text{ l})) = \text{Cp } x \longrightarrow$
 $\text{list2policy } (\text{map } \text{Cp } (\text{rev } (\text{normalize-manual-orderPrQ } x \text{ l}))) = \text{Cp } x$
apply (*simp add: p-eq*)
done

lemma *norm-notMT-manualQ*: $\text{DenyAll} \in \text{set } (\text{policy2list } p) \Longrightarrow \text{normalize-manual-orderPrQ}$
 $p \text{ l} \neq []$

```

apply (simp add: normalize-manual-orderPrQ-def)
apply (rule rADnMT)
apply (rule idNMT)
apply (rule sepnMT)
apply (rule sortnMTQ)
apply (rule RS2-NMT)
apply (rule remDupsNMT)
apply (rule RS3-NMT)
apply (rule DAiniD)
done

```

```

lemma C-eq-normalizePr-manualQ:
   $\llbracket \text{DenyAll} \in \text{set } (\text{policy2list } p);$ 
   $\text{allNetsDistinct } (\text{policy2list } p);$ 
   $\text{all-in-list } (\text{policy2list } p) \text{ } l \rrbracket \implies$ 
   $C_p (\text{list2FWpolicy } (\text{normalize-manual-orderPrQ } p \text{ } l)) = C_p p$ 
apply (simp add: normalize-manual-orderPrQ-def)
apply (rule C-eq-compileQ)
apply simp-all
done

```

```

lemma p-eq2-manual[rule-format]:  $\text{normalize-manual-orderPr } x \text{ } l \neq [] \longrightarrow$ 
   $C_p (\text{list2FWpolicy } (\text{normalize-manual-orderPr } x \text{ } l)) = C_p x \longrightarrow$ 
   $\text{list2policy } (\text{map } C_p (\text{rev } (\text{normalize-manual-orderPr } x \text{ } l))) = C_p x$ 
apply (simp add: p-eq)
done

```

```

lemma norm-notMT-manual:  $\text{DenyAll} \in \text{set } (\text{policy2list } p) \implies \text{normalize-manual-orderPr } p \text{ } l$ 
 $\neq []$ 
apply (simp add: normalize-manual-orderPr-def)
apply (rule rADnMT)
apply (rule idNMT)
apply (rule sepnMT)
apply (rule sortnMT)
apply (rule RS2-NMT)
apply (rule remDupsNMT)
apply (rule RS3-NMT)
apply (rule DAiniD)
done

```

As an example, how this theorems can be used for a concrete normalisation instantiation.

```

lemma normalizePrNAT:  $\llbracket \text{DenyAll} \in \text{set } (\text{policy2list } \text{Filter});$ 
   $\text{allNetsDistinct } (\text{policy2list } \text{Filter});$ 
   $\text{all-in-list } (\text{policy2list } \text{Filter}) (\text{Nets-List } \text{Filter}) \rrbracket \implies$ 

```



```

((λ (x,y). x) o-f (((NAT ⊗2 Cp Filter) o (λx. (x,x)))) =
  list2policy ((NAT ⊗L (map Cp (rev (normalizePr Filter)))) (op ⊗2) (λ (x,y). x) (λ x.
(x,x)))
apply (rule NATDistr)
apply simp-all
apply (metis norm-notMT)
by (metis C-eq-normalizePr list2FWpolicys-eq-sym norm-notMT)

```

```

lemma domSimpl[simp]: dom (Cp (A ⊕ DenyAll)) = dom (Cp (DenyAll))
apply (simp add: PLemmas)
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.