

HOL-TestGenFW

Achim D. Brucker Lukas Brügger Burkhart Wolff

November 19, 2012

Contents

1. Introduction	2
2. Installing and using HOL-TestGen/FW	3
3. Preliminaries	3
4. Packets and Networks	6
5. Address Representations	9
5.1. Datatype Addresses	10
5.2. Datatype Addresses with Ports	11
5.3. Integer Addresses	12
5.4. Integer Addresses with Ports	12
5.5. Integer Addresses with Ports and Protocols	13
5.6. IPv4 Addresses	14
6. Policies	15
6.1. Policy Core	15
6.2. Policy Combinators	15
6.3. Policy Combinators with Ports	16
6.4. Policy Combinators with Ports and Protocols	19
6.5. Ports	22
6.6. NAT	23
7. Network Address Translation	23
8. Policy Normalisation	26
8.1. Basics	27
8.2. Auxiliary definitions and functions.	28
8.3. Invariants	30
8.4. Transformations	31

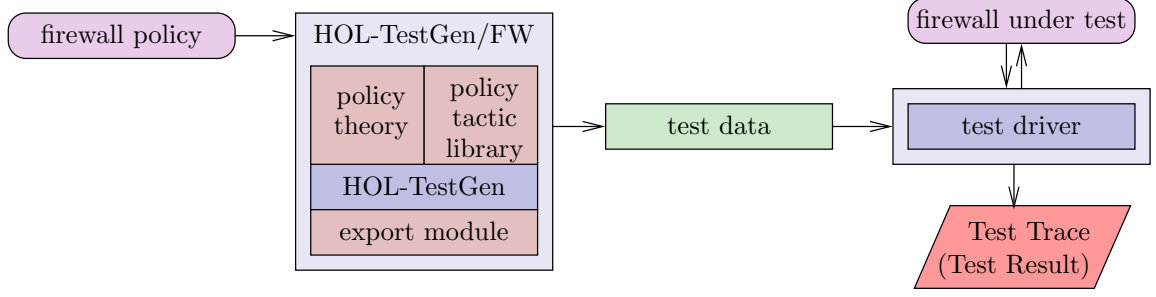


Figure 1: The HOL-TestGen/FW architecture.

8.5. IntPort	34
8.6. TCP_UDP_IntegerPort	36
9. Stateful Firewalls	38
9.1. Basic Constructs	38
9.2. FTP Protocol	40
9.2.1. The protocol syntax	40
9.2.2. The protocol policy specification	41
9.3. VoIP Protocol	45
9.4. FTP and VoIP Protocol	49
A. Appendix	56

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.

$\langle ML \rangle$

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*: $\llbracket x \in a; x \notin b \rrbracket \implies a \neq b$

$\langle proof \rangle$

lemma *setDistinct3*: $\llbracket x \neq y \rrbracket \implies$

$\{\{(a, b, c). a \in x\}\} \neq \{\{(a, b, c). a \in y\}\}$

$\langle proof \rangle$

lemma *setDistinct4*: $\llbracket (a::int) \neq x; a < b; x < y \rrbracket$

$\implies \{a..b\} \neq \{x..y\}$

$\langle proof \rangle$

lemma *setDistinct2*: $\llbracket x \neq y \rrbracket \implies$

$\{(a, b). a \in x\} \neq \{(a, b). a \in y\}$

$\langle proof \rangle$

lemma *setDistinct1*: $\llbracket x \neq y \rrbracket \implies$

$\{(a, b, c). a \in x\} \neq \{(a, b, c). a \in y\}$

$\langle proof \rangle$

lemma *AllowNMT*: $\llbracket x \neq \{\}; y \neq \{\} \rrbracket \implies$

$dom (Cp (AllowPortFromTo \{\{(a, b, c). a \in x\}\} \{\{(a, b, c). a \in y\}\} (p,i))) \neq \{\}$

$\langle proof \rangle$

lemma *domDA1Cp*: $\text{dom } (Cp \ (DenyAll \oplus P)) = UNIV$
 $\langle proof \rangle$

lemma *aux*: $\llbracket x \neq a; y \neq b; (x \neq y \wedge x \neq b) \vee (a \neq b \wedge a \neq y) \rrbracket \implies \{x, a\} \neq \{y, b\}$
 $\langle proof \rangle$

lemma *aux2*: $\{a, b\} = \{b, a\}$
 $\langle proof \rangle$

lemma *deny-comm[simp]*: $(deny() = X) = (X = deny())$
 $\langle proof \rangle$

lemma *allow-comm[simp]*: $(allow() = X) = (X = allow())$
 $\langle proof \rangle$

lemma *PO-add*: $PO \ P \implies P$
 $\langle proof \rangle$

lemma *dom-2-subset1*: $\text{dom } (r \text{ o-f } ((A \otimes_2 B) \text{ o } (\lambda x. (x, x)))) \subseteq \text{dom } A$
 $\langle proof \rangle$

lemma *dom-2-comm*: $\text{dom } (A \otimes_2 B) = \text{dom } (B \otimes_2 A)$
 $\langle proof \rangle$

lemma *dom-2-subset2*: $\text{dom } (r \text{ o-f } ((A \otimes_2 B) \text{ o } (\lambda x. (x, x)))) \subseteq \text{dom } B$
 $\langle proof \rangle$

lemma *dom-2*: $\text{dom } A = UNIV \implies$
 $\text{dom } (r \text{ o-f } ((A \otimes_2 B) \text{ o } (\lambda x. (x, x)))) = \text{dom } B$
 $\langle proof \rangle$

lemma *domDA1*: $\text{dom } (C \ (DenyAll \oplus P)) = UNIV$
 $\langle proof \rangle$

lemma *if-DA-simp*: $((if \ P \ \text{then } (deny\text{-all } x) \ \text{else } None) =$
 $None) = (\neg P)$
 $\langle proof \rangle$

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

<proof>

lemma *if-DA-simp3*: $((\text{if } P \text{ then deny-all } x \text{ else None}) = \text{Some } ae) = (P \wedge \text{deny-all } x = \text{Some } ae)$

<proof>

lemma *if-DA-simp4*: $((\text{if } P \text{ then Some (allow } x) \text{ else None}) = \text{None}) = (\neg P)$

<proof>

lemma *if-DA-simp5*: $((\text{if } P \text{ then Some (deny } x) \text{ else None}) = \text{None}) = (\neg P)$

<proof>

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

<proof>

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

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   ' $\alpha$  src = ' $\alpha$ 
```

```
type-synonym   ' $\alpha$  dest = ' $\alpha$ 
```

```
instance int :: adr <proof>
```

```
instance nat :: adr <proof>
```

```
instance fun :: (adr,adr) adr <proof>
```

```
instance prod :: (adr,adr) adr <proof>
```

The content is also specified with an unconstrained generic type:

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

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

```
definition src :: (' $\alpha$ ::adr, ' $\beta$ ) packet  $\Rightarrow$  ' $\alpha$ 
```

```
where src = fst o 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 <proof>
```

```

instance nat :: port <proof>
instance fun :: (port,port) port <proof>
instance prod :: (port,port) port <proof>

```

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 :: ('α::adr,'β) packet ⇒ id
where id = fst

```

```

definition dest :: ('α::adr,'β) packet ⇒ 'α dest
where dest = fst o snd o snd

```

```

definition content :: ('α::adr,'β) packet ⇒ 'β content
where content = snd o snd o snd

```

```

datatype protocol = tcp | udp

```

```

lemma either: [a ≠ tcp; a ≠ udp] ⇒ False
<proof>

```

```

lemma either2[simp]: (a ≠ tcp) = (a = udp)
<proof>

```

```

lemma either3[simp]: (a ≠ udp) = (a = tcp)
<proof>

```

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 :: ('α::adr,'β) packet ⇒ 'γ::port
consts dest-port :: ('α::adr,'β) packet ⇒ 'γ::port
consts src-protocol :: ('α::adr,'β) packet ⇒ protocol
consts dest-protocol :: ('α::adr,'β) packet ⇒ 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` (`⊆`) checks if an address is in a specific network.

```

definition
  in-subnet :: 'α::adr ⇒ 'α net ⇒ bool (infixl ⊆ 100) where
  in-subnet a S = (∃ s ∈ S. a ∈ s)

```


The following lemmas will be useful later.

lemma *in-subnet*:

$(a, e) \sqsubset \{\{(x1, y). P\ x1\ y\}\} = P\ a\ e$
 $\langle proof \rangle$

lemma *src-in-subnet*:

$src(q, (a, e), r, t) \sqsubset \{\{(x1, y). P\ x1\ y\}\} = P\ a\ e$
 $\langle proof \rangle$

lemma *dest-in-subnet*:

$dest(q, r, ((a), e), t) \sqsubset \{\{(x1, y). P\ x1\ y\}\} = P\ a\ e$
 $\langle proof \rangle$

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

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

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}}
```

end

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

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 <proof>

```

```

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
    <proof>

```

```

lemma dest-port : dest-port ((a,d,x,e)::(DatatypePort,'β) packet) = snd x
    <proof>

```

```

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  $src\text{-}port: src\text{-}port (a,x::adr_{ip},d,e) = snd\ x$ 
   $\langle proof \rangle$ 
```

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

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

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
adripp = address × port × protocol
```

```
instance protocol :: adr ⟨proof⟩
```

defs (overloaded)

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

```
lemma src-port: src-port (a,x::adripp,d,e) = fst (snd x)
⟨proof⟩
```

```
lemma dest-port: dest-port (a,d,x::adripp,e) = fst (snd x)
⟨proof⟩
```

Common test constraints:

```
definition port-positive :: (adripp, 'b) packet ⇒ bool where
port-positive x = (dest-port x > (0::port))
```

```
definition fix-values :: (adripp, DummyContent) packet ⇒ bool where
fix-values x = (src-port x = (1::port) ∧ src-protocol x = udp ∧ content x = data ∧ id x = 1)
```

lemmas $adr_{ipp} Lemmas = src\text{-}port\ dest\text{-}port\ src\text{-}port\text{-}def\ dest\text{-}port\text{-}def\ src\text{-}protocol\text{-}def\ dest\text{-}protocol\text{-}def\ subnet\text{-}of\text{-}def$

lemmas $adr_{ipp} TestConstraints = port\text{-}positive\text{-}def\ fix\text{-}values\text{-}def$

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 $src\text{-}port: src\text{-}port\ (a, (x::ipv4), d, e) = snd\ x$
 $\langle proof \rangle$

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

```

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 (' $\alpha$ , ' $\beta$ ) FWPolicy = (' $\alpha$ , ' $\beta$ ) 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 :: (' $\alpha$ , ' $\beta$ ) packet  $\Rightarrow$  (' $\alpha$ , ' $\beta$ ) 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 :: ' $\alpha$ ::adr net  $\Rightarrow$  ((' $\alpha$ , ' $\beta$ ) packet  $\mapsto$  unit) where

```

$allow-all-from\ src-net = \{pa.\ src\ pa \sqsubseteq src-net\} \triangleleft A_U$

definition

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

definition

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

definition

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

definition

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

definition

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

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:

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

definition

$allow\text{-}all\text{-}from\text{-}port :: 'α::adr\ net \Rightarrow 'γ::port \Rightarrow (('α, 'β)\ packet \mapsto unit)$ **where**
 $allow\text{-}all\text{-}from\text{-}port\ src\text{-}net\ s\text{-}port =$
 $\{pa.\ src\text{-}port\ pa = s\text{-}port\} \triangleleft allow\text{-}all\text{-}from\ src\text{-}net$

definition

$deny\text{-}all\text{-}from\text{-}port :: 'α::adr\ net \Rightarrow 'γ::port \Rightarrow (('α, 'β)\ packet \mapsto unit)$ **where**
 $deny\text{-}all\text{-}from\text{-}port\ src\text{-}net\ s\text{-}port =$
 $\{pa.\ src\text{-}port\ pa = s\text{-}port\} \triangleleft deny\text{-}all\text{-}from\ src\text{-}net$

definition

$allow\text{-}all\text{-}to\text{-}port :: 'α::adr\ net \Rightarrow 'γ::port \Rightarrow (('α, 'β)\ packet \mapsto unit)$ **where**
 $allow\text{-}all\text{-}to\text{-}port\ dest\text{-}net\ d\text{-}port =$
 $\{pa.\ dest\text{-}port\ pa = d\text{-}port\} \triangleleft allow\text{-}all\text{-}to\ dest\text{-}net$

definition

$deny\text{-}all\text{-}to\text{-}port :: 'α::adr\ net \Rightarrow 'γ::port \Rightarrow (('α, 'β)\ packet \mapsto unit)$ **where**
 $deny\text{-}all\text{-}to\text{-}port\ dest\text{-}net\ d\text{-}port =$
 $\{pa.\ dest\text{-}port\ pa = d\text{-}port\} \triangleleft deny\text{-}all\text{-}to\ dest\text{-}net$

definition

$allow\text{-}all\text{-}from\text{-}port\text{-}to :: 'α::adr\ net \Rightarrow 'γ::port \Rightarrow 'α::adr\ net \Rightarrow (('α, 'β)\ packet \mapsto unit)$
where
 $allow\text{-}all\text{-}from\text{-}port\text{-}to\ src\text{-}net\ s\text{-}port\ dest\text{-}net$
 $= \{pa.\ src\text{-}port\ pa = s\text{-}port\} \triangleleft allow\text{-}all\text{-}from\text{-}to\ src\text{-}net\ dest\text{-}net$

definition

$deny\text{-}all\text{-}from\text{-}port\text{-}to :: 'α::adr\ net \Rightarrow 'γ::port \Rightarrow 'α::adr\ net \Rightarrow (('α, 'β)\ packet \mapsto unit)$
where
 $deny\text{-}all\text{-}from\text{-}port\text{-}to\ src\text{-}net\ s\text{-}port\ dest\text{-}net$
 $= \{pa.\ src\text{-}port\ pa = s\text{-}port\} \triangleleft deny\text{-}all\text{-}from\text{-}to\ src\text{-}net\ dest\text{-}net$

definition

$allow\text{-}all\text{-}from\text{-}port\text{-}to\text{-}port :: 'α::adr\ net \Rightarrow 'γ::port \Rightarrow 'α::adr\ net \Rightarrow 'γ::port \Rightarrow$
 $((('α, 'β)\ packet \mapsto unit))$ **where**
 $allow\text{-}all\text{-}from\text{-}port\text{-}to\text{-}port\ src\text{-}net\ s\text{-}port\ dest\text{-}net\ d\text{-}port =$
 $\{pa.\ dest\text{-}port\ pa = d\text{-}port\} \triangleleft allow\text{-}all\text{-}from\text{-}port\text{-}to\ src\text{-}net\ s\text{-}port\ dest\text{-}net$

definition

$deny\text{-}all\text{-}from\text{-}port\text{-}to\text{-}port :: 'α::adr\ net \Rightarrow 'γ::port \Rightarrow 'α::adr\ net \Rightarrow$
 $'γ::port \Rightarrow (('α, 'β)\ packet \mapsto unit)$ **where**
 $deny\text{-}all\text{-}from\text{-}port\text{-}to\text{-}port\ src\text{-}net\ s\text{-}port\ dest\text{-}net\ d\text{-}port =$
 $\{pa.\ dest\text{-}port\ pa = d\text{-}port\} \triangleleft deny\text{-}all\text{-}from\text{-}port\text{-}to\ src\text{-}net\ s\text{-}port\ dest\text{-}net$

definition

$allow\text{-}all\text{-}from\text{-}to\text{-}port :: 'α::adr\ net \Rightarrow 'α::adr\ net \Rightarrow$
 $'γ::port \Rightarrow (('α, 'β)\ packet \mapsto unit)$ **where**
 $allow\text{-}all\text{-}from\text{-}to\text{-}port\ src\text{-}net\ dest\text{-}net\ d\text{-}port =$
 $\{pa.\ dest\text{-}port\ pa = d\text{-}port\} \triangleleft allow\text{-}all\text{-}from\text{-}to\ src\text{-}net\ dest\text{-}net$

definition

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

definition

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

definition

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

definition

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

definition

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

definition

$allow-from-ports-to :: 'γ::port\ set \Rightarrow 'α::adr\ net \Rightarrow 'α::adr\ net \Rightarrow$
 $((('α, 'β)\ packet \mapsto unit)\ \mathbf{where}$
 $allow-from-ports-to\ ports\ src-net\ dest-net =$
 $\{pa.\ src-port\ pa \in ports\} \triangleleft allow-all-from-to\ src-net\ dest-net$

definition

$allow-from-to-ports :: 'γ::port\ set \Rightarrow 'α::adr\ net \Rightarrow 'α::adr\ net \Rightarrow$
 $((('α, 'β)\ packet \mapsto unit)\ \mathbf{where}$
 $allow-from-to-ports\ ports\ src-net\ dest-net =$
 $\{pa.\ dest-port\ pa \in ports\} \triangleleft allow-all-from-to\ src-net\ dest-net$

definition

$deny-from-ports-to :: 'γ::port\ set \Rightarrow 'α::adr\ net \Rightarrow 'α::adr\ net \Rightarrow$
 $((('α, 'β)\ packet \mapsto unit)\ \mathbf{where}$
 $deny-from-ports-to\ ports\ src-net\ dest-net =$
 $\{pa.\ src-port\ pa \in ports\} \triangleleft deny-all-from-to\ src-net\ dest-net$

definition

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

definition

allow-all-from-port-tos :: $'\alpha::\text{adr net} \Rightarrow (' \gamma::\text{port}) \text{ set} \Rightarrow '\alpha::\text{adr net} \Rightarrow (('\alpha, '\beta) \text{ packet} \mapsto \text{unit})$
where
allow-all-from-port-tos *src-net s-port dest-net*
 $= \{pa. \text{ dest-port } pa \in \text{s-port}\} \triangleleft \text{allow-all-from-to } \text{src-net } \text{dest-net}$

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

$deny_all_from_port_to_port_prot :: protocol \Rightarrow 'α::adr\ net \Rightarrow 'γ::port \Rightarrow 'α::adr\ net \Rightarrow 'γ::port \Rightarrow (('α, 'β) packet \mapsto unit)$ **where**

$deny_all_from_port_to_port_prot\ p\ src_net\ s_port\ dest_net\ d_port =$
 $\{pa. dest_protocol\ pa = p\} \triangleleft deny_all_from_port_to_port\ src_net\ s_port\ dest_net\ d_port$

definition

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

definition

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

definition

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

definition

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

definition

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

definition

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

definition

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

definition

```

allow-from-to-ports-prot :: protocol => 'γ::port set => 'α::adr net => 'α::adr net =>
    (('α,'β) packet ↦ unit) where
allow-from-to-ports-prot p ports src-net dest-net =
    {pa. dest-protocol pa = p} ◁ 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 ↦ unit) where
deny-from-ports-to-prot p ports src-net dest-net =
    {pa. dest-protocol pa = p} ◁ 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 ↦ unit) where
deny-from-to-ports-prot p ports src-net dest-net =
    {pa. dest-protocol pa = p} ◁ 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*: $x \neq 80 \implies x \neq \text{http}$
 <proof>

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

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

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

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

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

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

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

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

definition $\text{src2poolPort} :: \text{int set} \Rightarrow (\text{adr}_{ip}, \beta) \text{ packet} \Rightarrow (\text{adr}_{ip}, \beta) \text{ packet set}$ **where**
 $\text{src2poolPort } t = (\lambda p. (\{(i, (s1, s2), (d1, d2), da). \\ (i = \text{id } p \wedge s1 \in t \wedge s2 = (\text{snd } (\text{src } p)) \wedge d1 = (\text{fst } (\text{dest } p)) \wedge \\ d2 = \text{snd } (\text{dest } p) \wedge da = \text{content } p)\}))$

definition $\text{src2poolPort-Protocol} :: \text{int set} \Rightarrow (\text{adr}_{ipp}, \beta) \text{ packet} \Rightarrow (\text{adr}_{ipp}, \beta) \text{ packet set}$ **where**
 $\text{src2poolPort-Protocol } t = (\lambda p. (\{(i, (s1, s2, s3), (d1, d2, d3), da). \\ (i = \text{id } p \wedge s1 \in t \wedge s2 = (\text{fst } (\text{snd } (\text{src } p))) \wedge s3 = \text{snd } (\text{snd } (\text{src } p)) \wedge \\ (d1, d2, d3) = \text{dest } p \wedge da = \text{content } p)\}))$

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

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

definition $\text{srcPat2poolPort-t} :: \text{int set} \Rightarrow (\text{adr}_{ip}, \beta) \text{ packet} \Rightarrow (\text{adr}_{ip}, \beta) \text{ packet set}$ **where**
 $\text{srcPat2poolPort-t } t = (\lambda p. (\{(i, (s1, s2), (d1, d2), da). \\ (i = \text{id } p \wedge s1 \in t \wedge d1 = (\text{fst } (\text{dest } p)) \wedge d2 = \text{snd } (\text{dest } p) \wedge da = \text{content } p)\}))$

definition $\text{srcPat2poolPort-Protocol-t} :: \text{int set} \Rightarrow (\text{adr}_{ipp}, \beta) \text{ packet} \Rightarrow (\text{adr}_{ipp}, \beta) \text{ packet set}$ **where**
 $\text{srcPat2poolPort-Protocol-t } t = (\lambda p. (\{(i, (s1, s2, s3), (d1, d2, d3), da). \\ (i = \text{id } p \wedge s1 \in t \wedge s3 = \text{src-protocol } p \wedge (d1, d2, d3) = \text{dest } p \wedge da = \text{content } p)\}))$

definition $\text{srcPat2pool-IntPort} :: \text{int set} \Rightarrow \text{int set} \Rightarrow (\text{adr}_{ip}, \beta) \text{ packet} \mapsto$
 $(\text{adr}_{ip}, \beta) \text{ packet set}$ **where**
 $\text{srcPat2pool-IntPort } \text{srcs } \text{transl} =$
 $\{x. (\text{fst } (\text{src } x)) \in \text{srcs}\} \triangleleft (A_f (\text{srcPat2poolPort-t } \text{transl}))$

definition $\text{srcPat2pool-IntProtocol} ::$
 $\text{int set} \Rightarrow \text{int set} \Rightarrow (\text{adr}_{ipp}, \beta) \text{ packet} \mapsto (\text{adr}_{ipp}, \beta) \text{ packet set}$ **where**
 $\text{srcPat2pool-IntProtocol } \text{srcs } \text{transl} =$
 $\{x. (\text{fst } (\text{src } x)) \in \text{srcs}\} \triangleleft (A_f (\text{srcPat2poolPort-Protocol-t } \text{transl}))$

The following lemmas are used for achieving a "standard" output format of NATted

packets for use in a firewall execution tool.

lemma *datasimp*: $\{(i, (s1, s2, s3), aba)\}.$

$$\begin{aligned} & \forall a \ aa \ b \ ba. \ aba = ((a, aa, b), ba) \longrightarrow i = i1 \wedge s1 = i101 \wedge \\ & s3 = iudp \wedge a = i110 \wedge aa = X606X3 \wedge b = X607X4 \wedge ba = data \} = \\ & \{(i, (s1, s2, s3), aba). \\ & \quad i = i1 \wedge s1 = i101 \wedge s3 = iudp \wedge (\lambda ((a,aa,b),ba). \ a = i110 \wedge aa = X606X3 \wedge \\ & \quad b = X607X4 \wedge ba = data) \ aba\} \\ & \langle proof \rangle \end{aligned}$$

lemma *datasimp2*: $\{(i, (s1, s2, s3), aba)\}.$

$$\begin{aligned} & \forall a \ aa \ b \ ba. \ aba = ((a, aa, b), ba) \longrightarrow i = i1 \wedge s1 = i132 \wedge s3 = iudp \wedge \\ & s2 = i1 \wedge a = i110 \wedge aa = i4 \wedge b = iudp \wedge ba = data \} \\ & = \{(i, (s1, s2, s3), aba). \\ & \quad i = i1 \wedge s1 = i132 \wedge s3 = iudp \wedge s2 = i1 \wedge (\lambda ((a,aa,b),ba). \ a = i110 \wedge \\ & \quad aa = i4 \wedge b = iudp \wedge ba = data) \ aba\} \\ & \langle proof \rangle \end{aligned}$$

lemma *datasimp3*: $\{(i, (s1, s2, s3), aba)\}.$

$$\begin{aligned} & ALL \ a \ aa \ b \ ba. \ aba = ((a, aa, b), ba) \dashrightarrow i = i1 \ \& \ i115 < s1 \ \& \ s1 < i124 \ \& \\ & s3 = iudp \ \& \ s2 = ii1 \ \& \ a = i110 \ \& \ aa = i3 \ \& \ b = itcp \ \& \ ba = data \} = \\ & \{(i, (s1, s2, s3), aba). \\ & \quad i = i1 \ \& \ i115 < s1 \ \& \ s1 < i124 \ \& \ s3 = iudp \ \& \ s2 = ii1 \ \& \ (\lambda ((a,aa,b),ba). \\ & \quad a = i110 \ \& \ aa = i3 \ \& \ b = itcp \ \& \ ba = data) \ aba\} \\ & \langle proof \rangle \end{aligned}$$

lemma *datasimp4*: $\{(i, (s1, s2, s3), aba)\}.$

$$\begin{aligned} & \forall a \ aa \ b \ ba. \ aba = ((a, aa, b), ba) \longrightarrow i = i1 \wedge s1 = i132 \wedge s3 = iudp \wedge \\ & s2 = ii1 \wedge a = i110 \wedge aa = i7 \wedge b = itcp \wedge ba = data \} = \\ & \{(i, (s1, s2, s3), aba). \\ & \quad i = i1 \wedge s1 = i132 \wedge s3 = iudp \wedge s2 = ii1 \wedge (\lambda ((a,aa,b),ba). \ a = i110 \wedge \\ & \quad aa = i7 \wedge b = itcp \wedge ba = data) \ aba\} \\ & \langle proof \rangle \end{aligned}$$

lemma *datasimp5*: $\{(i, (s1, s2, s3), aba)\}.$

$$\begin{aligned} & i = i1 \wedge s1 = i101 \wedge s3 = iudp \wedge (\lambda ((a,aa,b),ba). \ a = i110 \wedge aa = X606X3 \wedge \\ & b = X607X4 \wedge ba = data) \ aba \} \\ & = \{(i, (s1, s2, s3), (a,aa,b),ba). \\ & \quad i = i1 \wedge s1 = i101 \wedge s3 = iudp \wedge \ a = i110 \wedge aa = X606X3 \wedge \\ & \quad b = X607X4 \wedge ba = data \} \\ & \langle proof \rangle \end{aligned}$$

lemma *datasimp6*: $\{(i, (s1, s2, s3), aba)\}.$

$$\begin{aligned} & i = i1 \wedge s1 = i132 \wedge s3 = iudp \wedge s2 = i1 \wedge (\lambda ((a,aa,b),ba). \ a = i110 \wedge \\ & aa = i4 \wedge b = iudp \wedge ba = data) \ aba \} = \\ & \{(i, (s1, s2, s3), (a,aa,b),ba). \\ & \quad i = i1 \wedge s1 = i132 \wedge s3 = iudp \wedge s2 = i1 \wedge a = i110 \wedge aa = i4 \wedge b = iudp \wedge ba \\ & = data \} \\ & \langle proof \rangle \end{aligned}$$

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

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

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 = (zip (srcnets x) (destnets x))

fun (sequential) *normBothNets* **where**

```
|normBothNets ((a,b)#xs) = (if ((b,a)  $\in$  set xs)  $\vee$  (a,b)  $\in$  set xs)
    then (normBothNets xs)
    else (a,b)#(normBothNets xs)
|normBothNets x = x
```

fun *makeSets* **where**

```
|makeSets ((a,b)#xs) = ({a,b}#(makeSets xs))
|makeSets [] = []
```

```

fun bothNet where
  bothNet DenyAll = {}
| bothNet (DenyAllFromTo a b) = {a,b}
| bothNet (AllowPortFromTo a b p) = {a,b}
| bothNet (v  $\oplus$  va) = undefined

```

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

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

```

fun (sequential) first-srcNet where
  first-srcNet (x $\oplus$ y) = first-srcNet x
| first-srcNet x = srcNet x

```

```

fun (sequential) first-destNet where
  first-destNet (x $\oplus$ y) = first-destNet x
| first-destNet x = destNet x

```

```

fun (sequential) first-bothNet where
  first-bothNet (x $\oplus$ y) = first-bothNet x
| first-bothNet x = bothNet x

```

```

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

```

```

fun all-in-list where
  all-in-list [] l = True
| all-in-list (x#xs) l = (in-list x l  $\wedge$  all-in-list xs l)

```

```

fun (sequential) member where
  member a (x $\oplus$ xs) = ((member a x)  $\vee$  (member a xs))
| member a x = (a = x)

```

```

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

```

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

definition *subnetsOfAdr* **where** *subnetsOfAdr* a = {x. a \sqsubset 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*:: $((\alpha, \beta) \text{ Combinators}) \text{ list} \Rightarrow bool$ **where**
wellformed-policy1 $[] = True$
wellformed-policy1 $(x\#xs) = (DenyAll \notin (set \ xs))$

There is a DenyAll at the first position

fun *wellformed-policy1-strong*:: $((\alpha, \beta) \text{ Combinators}) \text{ list} \Rightarrow bool$
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* $a \ b = (\neg (\exists x. x \sqsubset a \wedge x \sqsubset b))$

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 $((x \oplus y) \# xs) = False$
singleCombinators $(x \# xs) = singleCombinators \ xs$

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

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

fun *OnlyTwoNets* **where**

```

OnlyTwoNets (DenyAll#xs) = OnlyTwoNets xs
| OnlyTwoNets (x#xs) = (onlyTwoNets x ∧ 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 ⊕ (list2FWpolicy y)

```

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

```

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

```

```

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

```

```

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

```

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

```

fun removeShadowRules2:: (('α, 'β) Combinators) list ⇒
  (('α, 'β) Combinators) list
where
  (removeShadowRules2 ((AllowPortFromTo x y p)#z)) =
    (if (((DenyAllFromTo x y) ∈ set z))
     then ((removeShadowRules2 z))
     else (((AllowPortFromTo x y p)#(removeShadowRules2 z))))
| removeShadowRules2 (x#y) = x#(removeShadowRules2 y)
| removeShadowRules2 [] = []

```

Sorting a pocli. We first need to define an ordering on rules. This ordering depends on the *Nets_List* of a policy.

```

fun smaller :: (('α, 'β) Combinators) ⇒
  (('α, 'β) Combinators) ⇒
  (('α) set) list ⇒ bool
where
  smaller DenyAll x l = True
| smaller x DenyAll l = False
| smaller x y l =
  ((x = y) ∨
   (if (bothNet x) = (bothNet y) then
    (case y of (DenyAllFromTo a b) ⇒ (x = DenyAllFromTo b a)
    | - ⇒ 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
  <proof>

```

```

lemma set-qsort [simp]: set (qsort xs l) = set xs

```


$\langle \text{proof} \rangle$

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

```
fun sort where
  sort []  $l = []$ 
  | sort ( $x\#xs$ )  $l = \text{insort } x\ (\text{sort } xs\ l)\ l$ 
```

```
fun sorted where
  sorted []  $l \longleftrightarrow \text{True}$  |
  sorted [ $x$ ]  $l \longleftrightarrow \text{True}$  |
  sorted ( $x\#y\#zs$ )  $l \longleftrightarrow \text{smaller } x\ y\ l \wedge \text{sorted } (y\#zs)\ l$ 
```

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

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

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

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

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

```
fun removeDuplicates where
  removeDuplicates ( $x\oplus xs$ ) = ( $\text{if member } x\ xs \text{ then } (\text{removeDuplicates } xs)$ 
    else  $x\oplus(\text{removeDuplicates } xs)$ )
  | removeDuplicates  $x = x$ 
```

```
fun removeAllDuplicates where
  removeAllDuplicates ( $x\#xs$ ) = ( $(\text{removeDuplicates } (x))\#(\text{removeAllDuplicates } xs)$ )
  | 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 ⊕ 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 ⊕ b) = (rotatePolicy b) ⊕ (rotatePolicy a)

```

```

lemma check: rev (policy2list (rotatePolicy p)) = policy2list p
<proof>

```

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::((adrip 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 ::

```
(adrip net, port) Combinators  $\Rightarrow$ 
(adrip 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 ::

```
(adrip net, port) Combinators  $\Rightarrow$ 
(adrip 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 $normalize = normalize'$

<proof>

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 $Dp :: (adr_{ipp}\ net, protocol \times port)\ Combinators \Rightarrow (adr_{ipp}, DummyContent)\ packet \mapsto unit$

where

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

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

fun $(sequential)\ wellformed-policy2Pr\ where$

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

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 $(sequential)\ wellformed-policy3Pr :: ((adr_{ipp}\ net, protocol \times port)\ Combinators)\ list \Rightarrow bool$
where

$wellformed_policy3Pr [] = True$
 $| wellformed_policy3Pr ((AllowPortFromTo a b p) \# xs) = ((\forall r. r \in set xs \longrightarrow$
 $\quad dom (Cp r) \cap dom (Cp (AllowPortFromTo a b p)) = \{\}) \wedge wellformed_policy3Pr xs)$
 $| wellformed_policy3Pr (x \# xs) = wellformed_policy3Pr xs$

definition

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

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

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

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

$normalize_manual_orderPrQ p l = removeAllDuplicates (insertDenies (separate$
 $\quad (qsort (removeShadowRules2 (remdups ((removeShadowRules3 Cp) (insertDeny$

(removeShadowRules1 (policy2list p)))))) ((l))))

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

lemma *normalizePr = normalizePr'*
<proof>

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

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

declare *Cp.simps [simp del]*

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

lemma *aux*: $\llbracket x \neq a; y \neq b; (x \neq y \wedge x \neq b) \vee (a \neq b \wedge a \neq y) \rrbracket \implies \{x, a\} \neq \{y, b\}$
<proof>

lemma *aux2*: $\{a, b\} = \{b, a\}$
<proof>

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

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

```

fun ids where
  ids a (x#xs) = (NetworkCore.id x ∈ a ∧ ids a xs)
| ids a [] = True

```

```

definition applyPolicy:: ('i × ('i ↦ 'o)) ↦ 'o
where    applyPolicy = (λ (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 = $(\lambda i p. (id\ p = i \wedge content\ p = ftp-init))$

definition

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

definition

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

definition

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

definition

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

definition

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

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

$$\begin{aligned}
&FTP\text{-}STA((i,s,d,ftp\text{-}port\text{-}request\ pr), (log\ ,\ pol)) = \\
&\quad (if\ before(Not\ o\ is\text{-}ftp\text{-}close\ i)(is\text{-}init\ i)\ log\ \wedge \\
&\quad\quad dest\text{-}port\ (i,s,d,ftp\text{-}port\text{-}request\ pr) = (21::port) \\
&\quad\quad then\ Some\ (((i,s,d,ftp\text{-}port\text{-}request\ pr)\#log\ , \\
&\quad\quad\quad (allow\text{-}from\text{-}to\text{-}port\ pr\ (subnet\text{-}of\ d)\ (subnet\text{-}of\ s)) \oplus\ pol)) \\
&\quad\quad else\ Some\ (((i,s,d,ftp\text{-}port\text{-}request\ pr)\#log\ ,pol))) \\
\\
&|FTP\text{-}STA((i,s,d,ftp\text{-}close), (log\ ,pol)) = \\
&\quad (if\ (\exists\ p.\ port\text{-}open\ log\ i\ p) \wedge dest\text{-}port\ (i,s,d,ftp\text{-}close) = (21::port) \\
&\quad\quad then\ Some\ ((i,s,d,ftp\text{-}close)\#log\ , \\
&\quad\quad\quad deny\text{-}from\text{-}to\text{-}port\ (last\text{-}opened\text{-}port\ i\ log)\ (subnet\text{-}of\ d)(subnet\text{-}of\ s) \oplus\ pol) \\
&\quad\quad else\ Some\ (((i,s,d,ftp\text{-}close)\#log\ ,\ pol))) \\
\\
&|FTP\text{-}STA(p,\ s) = Some\ (p\#(fst\ s),snd\ s)
\end{aligned}$$

fun $FTP\text{-}STD ::$
 $((adr_{ip},msg)\ history,\ adr_{ip},\ msg)\ FWStateTransition$
where
 $FTP\text{-}STD\ (p,s) = Some\ s$

definition $TRPolicy ::$ $(adr_{ip},msg)packet \times (adr_{ip},msg)history \times ((adr_{ip},msg)packet \mapsto unit)$
 $\mapsto (unit \times (adr_{ip},msg)history \times ((adr_{ip},msg)packet \mapsto unit))$
where $TRPolicy = ((FTP\text{-}STA,FTP\text{-}STD) \otimes_{\nabla} applyPolicy) \circ (\lambda(x,(y,z)).((x,z),(x,(y,z))))$

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

If required to contain the policy in the output

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

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* | *S1* | *S2* | *S3*

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

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 ((λ (*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 :: *adr_{ip}* *src* \Rightarrow *adr_{ip}* *dest* \Rightarrow *id* \Rightarrow *port* \Rightarrow
 adr_{ip} *src* \Rightarrow *adr_{ip}* *dest* \Rightarrow *id* \Rightarrow *port* \Rightarrow
 (*adr_{ip}*,*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*)) \wedge
 (*is-ftp S0 s2 d2 i2 p2* (*packet-with-id x i2*))}

lemma *subnetOf-lemma*: (*a::int*) \neq (*c::int*) $\implies \forall x \in \text{subnet-of } (a, b::\text{port}). (c, d) \notin x$
 <proof>

lemma *subnetOf-lemma2*: $\forall x \in \text{subnet-of } (a::\text{int}, b::\text{port}). (a, b) \in x$
 <proof>

lemma *subnetOf-lemma3*: ($\exists x. x \in \text{subnet-of } (a::\text{int}, b::\text{port})$)
 <proof>

lemma *subnetOf-lemma4*: $\exists x \in \text{subnet-of } (a::\text{int}, b::\text{port}). (a, c::\text{port}) \in x$
 <proof>

lemma *port-open-lemma*: $\neg (Ex (port-open \ [] (x::port)))$
 <proof>

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 adr_{ip}Lemmas 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

$$\text{Some } (\text{allow } (POL ((\text{allow-from-to-port } pr \ (\text{subnet-of } d) \ (\text{subnet-of } s)) \oplus \text{policy})),$$

$$(\text{ports}(i \mapsto pr)), (\text{allow-from-to-port } pr \ (\text{subnet-of } d) \ (\text{subnet-of } s))$$

$$\oplus \text{policy}))$$

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 ⇒
  Some(allow (POL (deny-from-to-port pr (subnet-of d) (subnet-of s) ⊕ policy)),
    ports(i:=None),
    deny-from-to-port pr (subnet-of d) (subnet-of s) ⊕ policy)
| None ⇒ 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 $'a\ voip\text{-}msg = ARQ$

$ACF\ 'a$	
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

$is\text{-}arq :: NetworkCore.id \Rightarrow ('a::adr, 'b\ voip\text{-}msg)\ packet \Rightarrow bool$ **where**
 $is\text{-}arq\ i\ p = (NetworkCore.id\ p = i \wedge content\ p = ARQ)$

definition

$is\text{-}fin :: id \Rightarrow ('a::adr, 'b\ voip\text{-}msg)\ packet \Rightarrow bool$ **where**
 $is\text{-}fin\ i\ p = (id\ p = i \wedge content\ p = Fin)$

definition

$is\text{-}connect :: id \Rightarrow port \Rightarrow ('a::adr, 'b\ voip\text{-}msg)\ packet \Rightarrow bool$ **where**
 $is\text{-}connect\ i\ port\ p = (id\ p = i \wedge content\ p = Connect\ port)$

definition

$is\text{-}setup :: id \Rightarrow port \Rightarrow ('a::adr, 'b\ voip\text{-}msg)\ packet \Rightarrow bool$ **where**

$is-setup\ i\ port\ p = (id\ p = i \wedge content\ p = Setup\ port)$

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

definition

$ports-open :: id \Rightarrow port \times port \Rightarrow (adr_{ip}, 'a\ voip-msg)\ history \Rightarrow bool$ **where**
 $ports-open\ i\ p\ L = ((not-before\ (is-fin\ i)\ (is-setup\ i\ (fst\ p)))\ L) \wedge$
 $not-before\ (is-fin\ i)\ (is-connect\ i\ (snd\ p))\ L)$

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

fun

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

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**

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

fun *VOIP-STA* ::

$((adr_{ip}, address\ voip-msg)\ history, adr_{ip}, address\ voip-msg)\ FWStateTransition$
where

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

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

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

| *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) ⊕ policy))

| *VOIP-STA* ((*a,c,d, Connect port*), (*InL, policy*)) =
 Some (((*a,c,d,Connect port*)#*InL*,
 allow-from-to-port port (subnet-of d) (subnet-of c) ⊕ policy))

| *VOIP-STA* ((*a,c,d,Fin*), (*InL,policy*)) =
 (*if* ∃ *p1 p2. ports-open a (p1,p2) InL* then (
 (*if 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)) ⊕*
 (*deny-from-to-port (snd (SOME p. ports-open a p InL))*
 (*subnet-of c) (subnet-of d)*) ⊕
 (*deny-from-to-port (fst (SOME p. ports-open a p InL))*
 (*subnet-of d) (subnet-of c)*) ⊕ *policy*)))
 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* ⇒

$port \Rightarrow (adr_{ip}, address\ voip\text{-}msg)\ history \Rightarrow bool$

where

$is\text{-}voip\ H\ s\ d\ g\ i\ p1\ p2\ [] = (H = S5)$
 $|is\text{-}voip\ H\ s\ d\ g\ i\ p1\ p2\ (x\#InL) =$
 $((\lambda (id, sr, de, co).$
 $((id = i \wedge$
 $(H = S4 \wedge ((sr = (s, 1719) \wedge de = (g, 1719) \wedge co = ARQ \wedge$
 $is\text{-}voip\ S5\ s\ d\ g\ i\ p1\ p2\ InL)))) \vee$
 $(H = S0 \wedge sr = (g, 1719) \wedge de = (s, 1719) \wedge co = ARJ \wedge$
 $is\text{-}voip\ S4\ s\ d\ g\ i\ p1\ p2\ InL) \vee$
 $(H = S3 \wedge sr = (g, 1719) \wedge de = (s, 1719) \wedge co = ACF\ d \wedge$
 $is\text{-}voip\ S4\ s\ d\ g\ i\ p1\ p2\ InL) \vee$
 $(H = S2 \wedge sr = (s, 1720) \wedge de = (d, 1720) \wedge co = Setup\ p1 \wedge$
 $is\text{-}voip\ S3\ s\ d\ g\ i\ p1\ p2\ InL) \vee$
 $(H = S1 \wedge sr = (d, 1720) \wedge de = (s, 1720) \wedge co = Connect\ p2 \wedge$
 $is\text{-}voip\ S2\ s\ d\ g\ i\ p1\ p2\ InL) \vee$
 $(H = S1 \wedge sr = (s, p1) \wedge de = (d, p2) \wedge co = Stream \wedge$
 $is\text{-}voip\ S1\ s\ d\ g\ i\ p1\ p2\ InL) \vee$
 $(H = S1 \wedge sr = (d, p2) \wedge de = (s, p1) \wedge co = Stream \wedge$
 $is\text{-}voip\ S1\ s\ d\ g\ i\ p1\ p2\ InL) \vee$
 $(H = S0 \wedge sr = (d, 1720) \wedge de = (s, 1720) \wedge co = Fin \wedge$
 $is\text{-}voip\ S1\ s\ d\ g\ i\ p1\ p2\ InL) \vee$
 $(H = S0 \wedge sr = (s, 1720) \wedge de = (d, 1720) \wedge co = Fin \wedge$
 $is\text{-}voip\ S1\ s\ d\ g\ i\ p1\ p2\ InL))))))\ x)$

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

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

end

9.4. FTP and VoIP Protocol

theory *FTPVOIP*

imports

FTP-WithPolicy VOIP

begin

datatype *ftpvoip* = *ARQ*
 $|$ *ACF int*
 $|$ *ARJ*
 $|$ *Setup port*
 $|$ *Connect port*

	<i>Stream</i>
	<i>Fin</i>
	<i>ftp-init</i>
	<i>ftp-port-request 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

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

definition

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

definition

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

definition

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

definition

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

definition

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

fun $FTPVOIP-are-other$ **where**

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

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 *FTPVOIP-FTP-STA* ::

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

where

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

|*FTPVOIP-FTP-STA* ((*i,s,d,ftp-close*), (*InL,policy*)) =
 (if (∃ *p*. *FTPVOIP-port-open InL i p*) ∧ *dest-port (i,s,d,ftp-close) = (21::port)*
 then *Some ((i,s,d,ftp-close)#InL, policy ++*
 deny-from-to-port (last-opened-port i InL) (subnet-of d) (subnet-of s))
 else *Some (((i,s,d,ftp-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* ⇒ (*'a::adr*, *ftpvoip*) *packet* ⇒ *bool* **where**
FTPVOIP-is-arq i p = (*NetworkCore.id p = i* ∧ *content p = ARQ*)

definition

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

definition

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

definition

FTPVOIP-is-setup :: *id* ⇒ *port* ⇒ (*'a::adr*, *ftpvoip*) *packet* ⇒ *bool* **where**
FTPVOIP-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

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

$$\text{not-before } (FTPVOIP\text{-is-fin } i) (FTPVOIP\text{-is-connect } i (snd\ p))\ L)$$

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

fun

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

definition $FTPVOIP\text{-subnet-of-adr} :: int \Rightarrow adr_{ip}\ net$ **where**
 $FTPVOIP\text{-subnet-of-adr } x = \{(a,b). a = x\}$

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

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

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

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

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

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

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

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

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

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

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

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)* =
 ((($\lambda (id, sr, de, co).$
 (((*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*)

definition

fun

where

definition

definition

where

55

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*: $\text{dom } X \neq \{\} \implies X \neq \emptyset$
 $\langle \text{proof} \rangle$

lemma *denyNMT*: $\text{deny-all} \neq \emptyset$
 $\langle \text{proof} \rangle$

lemma *wellformed-policy1-charn*[*rule-format*] : $\text{wellformed-policy1 } p \longrightarrow \text{DenyAll} \in \text{set } p \longrightarrow (\exists p'. p = \text{DenyAll} \# p' \wedge \text{DenyAll} \notin \text{set } p')$
 $\langle \text{proof} \rangle$

lemma *singleCombinatorsConc*: $\text{singleCombinators } (x \# xs) \implies \text{singleCombinators } xs$
 $\langle \text{proof} \rangle$

lemma *aux0-0*: $\text{singleCombinators } x \implies \neg (\exists a b. (a \oplus b) \in \text{set } x)$
 $\langle \text{proof} \rangle$

lemma *aux0-4*: $(a \in \text{set } x \vee a \in \text{set } y) = (a \in \text{set } (x @ y))$
 $\langle \text{proof} \rangle$

lemma *aux0-1*: $\llbracket \text{singleCombinators } xs; \text{singleCombinators } [x] \rrbracket \implies \text{singleCombinators } (x \# xs)$
 $\langle \text{proof} \rangle$

lemma *aux0-6*: $\llbracket \text{singleCombinators } xs; \neg (\exists a b. x = a \oplus b) \rrbracket \implies \text{singleCombinators } (x \# xs)$
 $\langle \text{proof} \rangle$

lemma *aux0-5*: $\neg (\exists a b. (a \oplus b) \in \text{set } x) \implies \text{singleCombinators } x$
 $\langle \text{proof} \rangle$

lemma *ANDConc*[*rule-format*]: $\text{allNetsDistinct } (a \# p) \longrightarrow \text{allNetsDistinct } (p)$
 $\langle \text{proof} \rangle$

lemma *aux6*: *twoNetsDistinct a1 a2 a b* \implies
 $\text{dom } (\text{deny-all-from-to } a1 \ a2) \cap \text{dom } (\text{deny-all-from-to } a \ b) = \{\}$
 $\langle \text{proof} \rangle$

lemma *aux5*[*rule-format*]: $(\text{DenyAllFromTo } a \ b) \in \text{set } p \longrightarrow a \in \text{set } (\text{net-list } p)$
 $\langle \text{proof} \rangle$

lemma *aux5a*[*rule-format*]: $(\text{DenyAllFromTo } b \ a) \in \text{set } p \longrightarrow a \in \text{set } (\text{net-list } p)$
 $\langle \text{proof} \rangle$

lemma *aux5c*[*rule-format*]:
 $(\text{AllowPortFromTo } a \ b \ po) \in \text{set } p \longrightarrow a \in \text{set } (\text{net-list } p)$
 $\langle \text{proof} \rangle$

lemma *aux5d*[*rule-format*]:
 $(\text{AllowPortFromTo } b \ a \ po) \in \text{set } p \longrightarrow a \in \text{set } (\text{net-list } p)$
 $\langle \text{proof} \rangle$

lemma *aux10*[*rule-format*]: $a \in \text{set } (\text{net-list } p) \longrightarrow a \in \text{set } (\text{net-list-aux } p)$
 $\langle \text{proof} \rangle$

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

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

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

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

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

lemma *aux7*: $\llbracket \text{DenyAllFromTo } a \ b \in \text{set } p; \text{allNetsDistinct } ((\text{DenyAllFromTo } c \ d) \# p);$
 $a \neq c \vee b \neq d \rrbracket \implies \text{twoNetsDistinct } a \ b \ c \ d$
 $\langle \text{proof} \rangle$

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

lemma *nDComm*: **assumes** *ab*: *netsDistinct a b* **shows** *ba*: *netsDistinct b a*
 $\langle \text{proof} \rangle$

lemma *tNDComm*:
assumes *abcd*: *twoNetsDistinct a b c d* **shows** *twoNetsDistinct c d a b*
 $\langle \text{proof} \rangle$

lemma *aux[rule-format]*: $a \in \text{set } (\text{removeShadowRules2 } p) \longrightarrow a \in \text{set } p$
 $\langle \text{proof} \rangle$

lemma *aux12*: $\llbracket a \in x; b \notin x \rrbracket \implies a \neq b$
 $\langle \text{proof} \rangle$

lemma *ND0aux1[rule-format]*: $\text{DenyAllFromTo } x \ y \in \text{set } b \implies$
 $x \in \text{set } (\text{net-list-aux } b)$
 $\langle \text{proof} \rangle$

lemma *ND0aux2[rule-format]*: $\text{DenyAllFromTo } x \ y \in \text{set } b \implies$
 $y \in \text{set } (\text{net-list-aux } b)$
 $\langle \text{proof} \rangle$

lemma *ND0aux3[rule-format]*: $\text{AllowPortFromTo } x \ y \ p \in \text{set } b \implies$
 $x \in \text{set } (\text{net-list-aux } b)$
 $\langle \text{proof} \rangle$

lemma *ND0aux4[rule-format]*: $\text{AllowPortFromTo } x \ y \ p \in \text{set } b \implies$
 $y \in \text{set } (\text{net-list-aux } b)$
 $\langle \text{proof} \rangle$

lemma *aNDSubsetaux*[*rule-format*]: *singleCombinators a* \longrightarrow *set a* \subseteq *set b* \longrightarrow
set (net-list-aux a) \subseteq *set (net-list-aux b)*
 <proof>

lemma *aNDSetsEqaux*[*rule-format*]: *singleCombinators a* \longrightarrow *singleCombinators b* \longrightarrow
set a = *set b* \longrightarrow *set (net-list-aux a)* = *set (net-list-aux b)*
 <proof>

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

lemma *aNDSetsEq*: $\llbracket \text{singleCombinators } a; \text{singleCombinators } b; \text{set } a = \text{set } b; \text{allNetsDistinct } b \rrbracket \Longrightarrow$ *allNetsDistinct a*
 <proof>

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

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

lemma *wp1-aux1a*[*rule-format*]: *xs* $\neq []$ \longrightarrow *wellformed-policy1-strong (xs @ [x])* \longrightarrow
wellformed-policy1-strong xs
 <proof>

lemma *wp1alternative-RS1*[*rule-format*]: *DenyAll* \in *set p* \longrightarrow
wellformed-policy1-strong (removeShadowRules1 p)
 <proof>

lemma *wellformed-eq*: *DenyAll* \in *set p* \longrightarrow
 ((*wellformed-policy1 p*) = (*wellformed-policy1-strong p*))
 <proof>

lemma *set-insort*: *set(insort x xs l)* = *insert x (set xs)*
 <proof>

lemma *set-sort*[*simp*]: *set(sort xs l)* = *set xs*
 <proof>

lemma *set-sortQ*: *set(qsort xs l)* = *set xs*
 <proof>

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

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

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

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

lemma *saux3*[rule-format]: $\text{DenyAllFromTo } a \ b \in \text{set list} \longrightarrow$
 $\text{DenyAllFromTo } c \ d \notin \text{set list} \longrightarrow (a \neq c) \vee (b \neq d)$
 <proof>

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

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

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

lemma *wellformed1-sorted-auxQ*[rule-format]: $\text{wellformed-policy1 } (p) \implies$
 $\text{wellformed-policy1 } (q\text{sort } p \ l)$
 <proof>

lemma *SR1Subset*: $\text{set } (\text{removeShadowRules1 } p) \subseteq \text{set } p$
 <proof>

lemma *SCSubset*[rule-format]: $\text{singleCombinators } b \longrightarrow \text{set } a \subseteq \text{set } b \longrightarrow$
 $\text{singleCombinators } a$
 <proof>

lemma *setInsert*[simp]: $\text{set list} \subseteq \text{insert } a \ (\text{set list})$
 <proof>

lemma *SC-RS1*[rule-format,simp]: $\text{singleCombinators } p \longrightarrow \text{allNetsDistinct } p \longrightarrow$
 $\text{singleCombinators } (\text{removeShadowRules1 } p)$
 <proof>

lemma *RS2Set*[rule-format]: $\text{set } (\text{removeShadowRules2 } p) \subseteq \text{set } p$

$\langle \text{proof} \rangle$

lemma *WP1*: $a \notin \text{set } \text{list} \implies a \notin \text{set } (\text{removeShadowRules2 } \text{list})$

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

lemma *SCConcEnd*: $\text{singleCombinators } (xs @ [xa]) \implies \text{singleCombinators } xs$

$\langle \text{proof} \rangle$

lemma *list2FWpolicyconc[rule-format]*: $a \neq [] \longrightarrow$

$(\text{list2FWpolicy } (xa \# a)) = (xa) \oplus (\text{list2FWpolicy } a)$

$\langle \text{proof} \rangle$

lemma *wp1n-tl [rule-format]*: $\text{wellformed-policy1-strong } p \longrightarrow$

$p = (\text{DenyAll} \# (\text{tl } p))$

$\langle \text{proof} \rangle$

lemma *foo2*:

$\llbracket a \notin \text{set } ps; a \notin \text{set } ss; \text{set } p = \text{set } s; p = (a \# (ps)); s = (a \# ss) \rrbracket \implies$

$\text{set } (ps) = \text{set } (ss)$

$\langle \text{proof} \rangle$

lemma *SCnotConc[rule-format,simp]*: $a \oplus b \in \text{set } p \longrightarrow \text{singleCombinators } p \longrightarrow \text{False}$

$\langle \text{proof} \rangle$

lemma *auxx8*: $\text{removeShadowRules1-alternative-rev } [x] = [x]$

$\langle \text{proof} \rangle$

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

$(\text{removeShadowRules1 } xs) @ [x]$

$\langle \text{proof} \rangle$

lemma *aux114*: $x \neq \text{DenyAll} \implies \text{removeShadowRules1-alternative-rev } (x \# xs) =$

$x \# (\text{removeShadowRules1-alternative-rev } xs)$

$\langle \text{proof} \rangle$

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

$= (\text{removeShadowRules1-alternative } xs) @ [x]$

$\langle \text{proof} \rangle$

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

<proof>

lemma *rSR1-eq*: *removeShadowRules1-alternative* = *removeShadowRules1*

<proof>

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

<proof>

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

<proof>

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

<proof>

lemma *wp1-aux1aa*[*rule-format*]: $\text{wellformed-policy1-strong } p \longrightarrow \text{DenyAll} \in \text{set } p$

<proof>

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

<proof>

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

<proof>

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

<proof>

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

<proof>

lemma *ANDConcEnd*: $\llbracket \text{allNetsDistinct } (xs \ @ \ [xa]); \text{singleCombinators } xs \rrbracket \implies$
 $\text{allNetsDistinct } xs$

<proof>

lemma *WP1ConcEnd*[*rule-format*]:
 $\text{wellformed-policy1 } (xs @ [xa]) \longrightarrow \text{wellformed-policy1 } xs$

<proof>

lemma *NDComm*: $\text{netsDistinct } a \ b = \text{netsDistinct } b \ a$

<proof>

lemma *wellformed1-sorted*[*simp*]:
assumes *wp1*: *wellformed-policy1* *p*
shows $\text{wellformed-policy1 } (\text{sort } p \ l)$

$\langle proof \rangle$

lemma *wellformed1-sortedQ[simp]*:
 assumes *wp1: wellformed-policy1 p*
 shows *wellformed-policy1 (qsort p l)*
 $\langle proof \rangle$

lemma *SC1[simp]*: *singleCombinators p \implies singleCombinators (removeShadowRules1 p)*
 $\langle proof \rangle$

lemma *SC2[simp]*: *singleCombinators p \implies singleCombinators (removeShadowRules2 p)*
 $\langle proof \rangle$

lemma *SC3[simp]*: *singleCombinators p \implies singleCombinators (sort p l)*
 $\langle proof \rangle$

lemma *SC3Q[simp]*: *singleCombinators p \implies singleCombinators (qsort p l)*
 $\langle proof \rangle$

lemma *aND-RS1[simp]*: $\llbracket \text{singleCombinators } p; \text{allNetsDistinct } p \rrbracket \implies$
 allNetsDistinct (removeShadowRules1 p)
 $\langle proof \rangle$

lemma *aND-RS2[simp]*: $\llbracket \text{singleCombinators } p; \text{allNetsDistinct } p \rrbracket \implies$
 allNetsDistinct (removeShadowRules2 p)
 $\langle proof \rangle$

lemma *aND-sort[simp]*: $\llbracket \text{singleCombinators } p; \text{allNetsDistinct } p \rrbracket \implies$
 allNetsDistinct (sort p l)
 $\langle proof \rangle$

lemma *aND-sortQ[simp]*: $\llbracket \text{singleCombinators } p; \text{allNetsDistinct } p \rrbracket \implies$
 allNetsDistinct (qsort p l)
 $\langle proof \rangle$

lemma *inRS2[rule-format,simp]*: $x \notin \text{set } p \longrightarrow x \notin \text{set (removeShadowRules2 } p)$
 $\langle proof \rangle$

lemma *distinct-RS2*[*rule-format,simp*]: $\text{distinct } p \longrightarrow$
 $\text{distinct } (\text{removeShadowRules2 } p)$
 <proof>

lemma *setPaireq*: $\{x, y\} = \{a, b\} \implies x = a \wedge y = b \vee x = b \wedge y = a$
 <proof>

lemma *position-positive*[*rule-format*]: $a \in \text{set } l \longrightarrow \text{position } a \ l > 0$
 <proof>

lemma *pos-noteq*[*rule-format*]:
 $a \in \text{set } l \longrightarrow b \in \text{set } l \longrightarrow c \in \text{set } l \longrightarrow a \neq b \longrightarrow$
 $(\text{position } a \ l) <= (\text{position } b \ l) \longrightarrow$
 $(\text{position } b \ l) <= (\text{position } c \ l) \longrightarrow$
 $a \neq c$
 <proof>

lemma *setPair-noteq*: $\{a, b\} \neq \{c, d\} \implies \neg ((a = c) \wedge (b = d))$
 <proof>

lemma *setPair-noteq-allow*: $\{a, b\} \neq \{c, d\} \implies \neg ((a = c) \wedge (b = d) \wedge P)$
 <proof>

lemma *order-trans*:
 $\llbracket \text{in-list } x \ l; \text{in-list } y \ l; \text{in-list } z \ l; \text{singleCombinators } [x];$
 $\text{singleCombinators } [y]; \text{singleCombinators } [z]; \text{smaller } x \ y \ l; \text{smaller } y \ z \ l \rrbracket \implies$
 $\text{smaller } x \ z \ l$
 <proof>

lemma *sortedConcStart*[*rule-format*]:
 $\text{sorted } (a \ \# \ aa \ \# \ p) \ l \longrightarrow \text{in-list } a \ l \longrightarrow \text{in-list } aa \ l \longrightarrow \text{all-in-list } p \ l \longrightarrow$
 $\text{singleCombinators } [a] \longrightarrow \text{singleCombinators } [aa] \longrightarrow \text{singleCombinators } p \longrightarrow$
 $\text{sorted } (a \ \# \ p) \ l$
 <proof>

lemma *singleCombinatorsStart*[*simp*]: $\text{singleCombinators } (x \ \# \ xs) \implies$
 $\text{singleCombinators } [x]$
 <proof>

lemma *sorted-is-smaller*[*rule-format*]:
 $\text{sorted } (a \ \# \ p) \ l \longrightarrow \text{in-list } a \ l \longrightarrow \text{in-list } b \ l \longrightarrow \text{all-in-list } p \ l \longrightarrow$
 $\text{singleCombinators } [a] \longrightarrow \text{singleCombinators } p \longrightarrow b \in \text{set } p \longrightarrow \text{smaller } a \ b \ l$
 <proof>

lemma *sortedConcEnd*[*rule-format*]: $\text{sorted } (a \ \# \ p) \ l \longrightarrow \text{in-list } a \ l \longrightarrow$
 $\text{all-in-list } p \ l \longrightarrow \text{singleCombinators } [a] \longrightarrow$
 $\text{singleCombinators } p \longrightarrow \text{sorted } p \ l$
 <proof>

lemma *in-set-in-list*[rule-format]: $a \in \text{set } p \longrightarrow \text{all-in-list } p \ l \longrightarrow \text{in-list } a \ l$
 <proof>

lemma *sorted-Consb*[rule-format]:
 $\text{all-in-list } (x\#xs) \ l \longrightarrow \text{singleCombinators } (x\#xs) \longrightarrow$
 $(\text{sorted } xs \ l \ \& \ (\text{ALL } y:\text{set } xs. \text{smaller } x \ y \ l)) \longrightarrow (\text{sorted } (x\#xs) \ l)$
 <proof>

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

lemma *smaller-antisym*: $\llbracket \neg \text{smaller } a \ b \ l; \text{in-list } a \ l; \text{in-list } b \ l;$
 $\text{singleCombinators } [a]; \text{singleCombinators } [b] \rrbracket \Longrightarrow$
 $\text{smaller } b \ a \ l$
 <proof>

lemma *set-insort-insert*: $\text{set } (\text{insort } x \ xs \ l) \subseteq \text{insert } x \ (\text{set } xs)$
 <proof>

lemma *all-in-listSubset*[rule-format]: $\text{all-in-list } b \ l \longrightarrow \text{singleCombinators } a \longrightarrow$
 $\text{set } a \subseteq \text{set } b \longrightarrow \text{all-in-list } a \ l$
 <proof>

lemma *singleCombinators-insort*: $\llbracket \text{singleCombinators } [x]; \text{singleCombinators } xs \rrbracket \Longrightarrow$
 $\text{singleCombinators } (\text{insort } x \ xs \ l)$
 <proof>

lemma *all-in-list-insort*: $\llbracket \text{all-in-list } xs \ l; \text{singleCombinators } (x\#xs);$
 $\text{in-list } x \ l \rrbracket \Longrightarrow \text{all-in-list } (\text{insort } x \ xs \ l) \ l$
 <proof>

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

lemma *is-in-insort*: $y \in \text{set } xs \Longrightarrow y \in \text{set } (\text{insort } x \ xs \ l)$
 <proof>

lemma *sorted-insorta*[rule-format]:
 $\text{sorted } (\text{insort } x \ xs \ l) \ l \longrightarrow \text{all-in-list } (x\#xs) \ l \longrightarrow \text{distinct } (x\#xs) \longrightarrow$
 $\text{singleCombinators } [x] \longrightarrow \text{singleCombinators } xs \longrightarrow \text{sorted } xs \ l$
 <proof>

lemma *sorted-insortb*[rule-format]:
 $\text{sorted } xs \ l \longrightarrow \text{all-in-list } (x\#xs) \ l \longrightarrow \text{distinct } (x\#xs) \longrightarrow$
 $\text{singleCombinators } [x] \longrightarrow \text{singleCombinators } xs \longrightarrow \text{sorted } (\text{insort } x \ xs \ l) \ l$
 <proof>

lemma *sorted-insort*: $\llbracket \text{all-in-list } (x\#xs) \ l; \text{distinct}(x\#xs); \text{singleCombinators } [x];$
 $\text{singleCombinators } xs \rrbracket \implies$
 $\text{sorted } (\text{insort } x \ xs \ l) \ l = \text{sorted } xs \ l$

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

lemma *sort-is-sorted[rule-format]*: $\text{all-in-list } p \ l \longrightarrow \text{distinct } p \longrightarrow$
 $\text{singleCombinators } p \longrightarrow \text{sorted } (\text{sort } p \ l) \ l$

$\langle \text{proof} \rangle$

lemma *smaller-sym[rule-format]*: $\text{all-in-list } [a] \ l \longrightarrow \text{smaller } a \ a \ l$

$\langle \text{proof} \rangle$

lemma *SC-sublist[rule-format]*: $\text{singleCombinators } xs \implies$
 $\text{singleCombinators } (qsort \ [y \leftarrow xs. \ P \ y] \ l)$

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

lemma *all-in-listAppend[rule-format]*:

$\text{all-in-list } (xs) \ l \longrightarrow \text{all-in-list } (ys) \ l \longrightarrow \text{all-in-list } (xs \ @ \ ys) \ l$

$\langle \text{proof} \rangle$

lemma *distinct-sortQ[rule-format]*: $\text{singleCombinators } xs \longrightarrow$

all-in-list xs l \longrightarrow *distinct xs* \longrightarrow *distinct (qsort xs l)*
 <proof>

lemma *singleCombinatorsAppend*[rule-format]:
singleCombinators xs \longrightarrow *singleCombinators ys* \longrightarrow *singleCombinators (xs @ ys)*
 <proof>

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* & ($\forall x \in \text{set } xs. \forall y \in \text{set } ys. \text{smaller } x \ y \ l$)))
 <proof>

lemma *sorted-append2*[rule-format]:
all-in-list xs l \longrightarrow *singleCombinators xs* \longrightarrow
all-in-list ys l \longrightarrow
singleCombinators ys \longrightarrow
 (*sorted xs l* & *sorted ys l* & ($\forall x \in \text{set } xs. \forall y \in \text{set } ys. \text{smaller } x \ y \ l$)) \longrightarrow (*sorted (xs@ys) l*)
 <proof>

lemma *sorted-append*[rule-format]:
all-in-list xs l \longrightarrow *singleCombinators xs* \longrightarrow
all-in-list ys l \longrightarrow
singleCombinators ys \longrightarrow (*sorted (xs@ys) l*) =
 (*sorted xs l* & *sorted ys l* & ($\forall x \in \text{set } xs. \forall y \in \text{set } ys. \text{smaller } x \ y \ l$))
 <proof>

lemma *sort-is-sortedQ*[rule-format]: *all-in-list p l* \longrightarrow
singleCombinators p \longrightarrow *sorted (qsort p l) l*
 <proof>

lemma *inSet-not-MT*: $a \in \text{set } p \implies p \neq []$
 <proof>

lemma *RS1n-assoc*: $x \neq \text{DenyAll} \implies \text{removeShadowRules1-alternative } xs \ @ \ [x] =$
removeShadowRules1-alternative (xs @ [x])
 <proof>

lemma *RS1n-nMT*[rule-format,simp]: $p \neq [] \longrightarrow \text{removeShadowRules1-alternative } p \neq []$
 <proof>

lemma *RS1N-DA*[simp]: *removeShadowRules1-alternative (a@[DenyAll])* = *[DenyAll]*
 <proof>

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

<proof>

lemma *mt-sym*: $dom\ a \cap dom\ b = \{\} \implies dom\ b \cap dom\ a = \{\}$

<proof>

lemma *DAnotTL*[rule-format]:

$xs \neq [] \longrightarrow \text{wellformed-policy1 } (xs @ [DenyAll]) \longrightarrow False$

<proof>

lemma *AND-tl*[rule-format]: *allNetsDistinct* (p) \longrightarrow *allNetsDistinct* ($tl\ p$)

<proof>

lemma *distinct-tl*[rule-format]: *distinct* $p \longrightarrow$ *distinct* ($tl\ p$)

<proof>

lemma *SC-tl*[rule-format]: *singleCombinators* (p) \longrightarrow *singleCombinators* ($tl\ p$)

<proof>

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

<proof>

lemma *wp1-tl*[rule-format]: $p \neq [] \wedge \text{wellformed-policy1 } p \longrightarrow$

$\text{wellformed-policy1 } (tl\ p)$

<proof>

lemma *wp1-eq*[rule-format]: *wellformed-policy1-strong* $p \implies \text{wellformed-policy1 } p$

<proof>

lemma *wellformed1-alternative-sorted*: *wellformed-policy1-strong* $p \implies$

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

<proof>

lemma *wp1n-RS2*[rule-format]: *wellformed-policy1-strong* $p \longrightarrow$

wellformed-policy1-strong ($\text{removeShadowRules2 } p$)

<proof>

lemma *RS2-NMT*[rule-format]: $p \neq [] \longrightarrow \text{removeShadowRules2 } p \neq []$

<proof>

lemma *wp1-alternative-not-mt[simp]: wellformed-policy1-strong $p \implies p \neq []$*
 <proof>

lemma *AIL1[rule-format,simp]: all-in-list $p\ l \longrightarrow$
 all-in-list (removeShadowRules1 p) l*
 <proof>

lemma *wp1ID: wellformed-policy1-strong (insertDeny (removeShadowRules1 p))*
 <proof>

lemma *dRD[simp]: distinct (remdups p)*
 <proof>

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

lemma *AILiD[rule-format,simp]: all-in-list $p\ l \longrightarrow$ all-in-list (insertDeny p) l*
 <proof>

lemma *SCrd[rule-format,simp]: singleCombinators $p \longrightarrow$ singleCombinators (remdups p)*
 <proof>

lemma *SCRiD[rule-format,simp]: singleCombinators $p \longrightarrow$
 singleCombinators (insertDeny p)*
 <proof>

lemma *WP1rd[rule-format,simp]: wellformed-policy1-strong $p \longrightarrow$
 wellformed-policy1-strong (remdups p)*
 <proof>

lemma *ANDrd[rule-format,simp]: singleCombinators $p \longrightarrow$ allNetsDistinct $p \longrightarrow$
 allNetsDistinct (remdups p)*
 <proof>

lemma *ANDiD[rule-format,simp]: allNetsDistinct $p \longrightarrow$
 allNetsDistinct (insertDeny p)*
 <proof>

lemma *mr-iD[rule-format]: wellformed-policy1-strong $p \longrightarrow$
 matching-rule $x\ p =$ matching-rule x (insertDeny p)*
 <proof>

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

wellformed-policy1-strong (*insertDeny* *p*)

<proof>

lemma *DAiniD*: *DenyAll* \in *set* (*insertDeny* *p*)

<proof>

lemma *p2lNmt*: *policy2list* *p* \neq []

<proof>

lemma *AIL2*[*rule-format,simp*]: *all-in-list* *p l* \longrightarrow
all-in-list (*removeShadowRules2* *p*) *l*

<proof>

lemma *SCConc*: $\llbracket \text{singleCombinators } x; \text{singleCombinators } y \rrbracket \Longrightarrow$
singleCombinators (*x@y*)

<proof>

lemma *SCp2l*: *singleCombinators* (*policy2list* *p*)

<proof>

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

<proof>

lemma *soadisj*: $\llbracket x \in \text{subnetsOfAdr } a; y \in \text{subnetsOfAdr } a \rrbracket \Longrightarrow \neg \text{netsDistinct } x \ y$

<proof>

lemma *not-member*: $\neg \text{member } a \ (x \oplus y) \Longrightarrow \neg \text{member } a \ x$

<proof>

lemma *soadisj2*: $(\forall \ a \ x \ y. \ x \in \text{subnetsOfAdr } a \wedge y \in \text{subnetsOfAdr } a \longrightarrow$
 $\neg \text{netsDistinct } x \ y)$

<proof>

lemma *ndFalse1*: $\llbracket (\forall \ a \ b \ c \ d. \ (a,b) \in A \wedge (c,d) \in B \longrightarrow \text{netsDistinct } a \ c);$
 $\exists (a, b) \in A. \ a \in \text{subnetsOfAdr } D;$
 $\exists (a, b) \in B. \ a \in \text{subnetsOfAdr } D \rrbracket$
 $\Longrightarrow \text{False}$

<proof>

lemma *ndFalse2*: $\llbracket (\forall \ a \ b \ c \ d. \ (a,b) \in A \wedge (c,d) \in B \longrightarrow \text{netsDistinct } b \ d);$
 $\exists (a, b) \in A. \ b \in \text{subnetsOfAdr } D;$
 $\exists (a, b) \in B. \ b \in \text{subnetsOfAdr } D \rrbracket$
 $\Longrightarrow \text{False}$

<proof>

lemma *tndFalse*: $\llbracket (\forall \ a \ b \ c \ d. \ (a,b) \in A \wedge (c,d) \in B \longrightarrow \text{twoNetsDistinct } a \ b \ c \ d);$

$\exists (a, b) \in A. a \in \text{subnetsOfAdr } (D::('a::\text{adr})) \wedge b \in \text{subnetsOfAdr } (F::'a);$
 $\exists (a, b) \in B. a \in \text{subnetsOfAdr } D \wedge b \in \text{subnetsOfAdr } F$
 $\implies \text{False}$
 $\langle \text{proof} \rangle$

lemma *sepnMT*[rule-format]: $p \neq [] \longrightarrow (\text{separate } p) \neq []$
 $\langle \text{proof} \rangle$

lemma *sepDA*[rule-format]: $\text{DenyAll} \notin \text{set } p \longrightarrow \text{DenyAll} \notin \text{set } (\text{separate } p)$
 $\langle \text{proof} \rangle$

lemma *setnMT*: $\text{set } a = \text{set } b \implies a \neq [] \implies b \neq []$
 $\langle \text{proof} \rangle$

lemma *sortnMT*: $p \neq [] \implies \text{sort } p \text{ l} \neq []$
 $\langle \text{proof} \rangle$

lemma *idNMT*[rule-format]: $p \neq [] \longrightarrow \text{insertDenies } p \neq []$
 $\langle \text{proof} \rangle$

lemma *OTNoTN*[rule-format]: $\text{OnlyTwoNets } p \longrightarrow x \neq \text{DenyAll} \longrightarrow x \in \text{set } p \longrightarrow$
 $\text{onlyTwoNets } x$
 $\langle \text{proof} \rangle$

lemma *first-isIn*[rule-format]:
 $\neg \text{member DenyAll } x \longrightarrow (\text{first-srcNet } x, \text{first-destNet } x) \in \text{sdnets } x$
 $\langle \text{proof} \rangle$

lemma *sdnets2*: $\llbracket \exists a \ b. \text{sdnets } x = \{(a, b), (b, a)\}; \neg \text{member DenyAll } x \rrbracket \implies$
 $\text{sdnets } x = \{(\text{first-srcNet } x, \text{first-destNet } x),$
 $(\text{first-destNet } x, \text{first-srcNet } x)\}$
 $\langle \text{proof} \rangle$

lemma *alternativelistconc1*[rule-format]: $a \in \text{set } (\text{net-list-aux } [x]) \longrightarrow$
 $a \in \text{set } (\text{net-list-aux } [x, y])$
 $\langle \text{proof} \rangle$

lemma *alternativelistconc2*[rule-format]: $a \in \text{set } (\text{net-list-aux } [x]) \longrightarrow$
 $a \in \text{set } (\text{net-list-aux } [y, x])$
 $\langle \text{proof} \rangle$

lemma *noDA*[rule-format]: $\text{noDenyAll } xs \longrightarrow s \in \text{set } xs \longrightarrow \neg \text{member DenyAll } s$
 $\langle \text{proof} \rangle$

lemma *isInAlternativeList*:
 $(aa \in \text{set } (\text{net-list-aux } [a]) \vee aa \in \text{set } (\text{net-list-aux } p))$
 $\implies aa \in \text{set } (\text{net-list-aux } (a \# p))$
 $\langle \text{proof} \rangle$

lemma *netlistaux*: $x \in \text{set } (\text{net-list-aux } (a \# p)) \implies$
 $x \in \text{set } (\text{net-list-aux } ([a])) \vee x \in \text{set } (\text{net-list-aux } (p))$
 <proof>

lemma *firstInNet*[rule-format]: $\neg \text{member DenyAll } a \longrightarrow$
 $\text{first-destNet } a \in \text{set } (\text{net-list-aux } (a \# p))$
 <proof>

lemma *firstInNeta*[rule-format]: $\neg \text{member DenyAll } a \longrightarrow$
 $\text{first-srcNet } a \in \text{set } (\text{net-list-aux } (a \# p))$
 <proof>

lemma *disjComm*: $\text{disjSD-2 } a \ b \implies \text{disjSD-2 } b \ a$
 <proof>

lemma *disjSD2aux*: $\llbracket \text{disjSD-2 } a \ b; \neg \text{member DenyAll } a; \neg \text{member DenyAll } b \rrbracket \implies$
 $\text{disjSD-2 } (\text{DenyAllFromTo } (\text{first-srcNet } a) (\text{first-destNet } a) \oplus$
 $\text{DenyAllFromTo } (\text{first-destNet } a) (\text{first-srcNet } a) \oplus a) \ b$
 <proof>

lemma *noDA1eq*[rule-format]: $\text{noDenyAll } p \longrightarrow \text{noDenyAll1 } p$
 <proof>

lemma *noDA1C*[rule-format]: $\text{noDenyAll1 } (a \# p) \longrightarrow \text{noDenyAll1 } p$
 <proof>

lemma *disjSD-2IDa*: $\llbracket \text{disjSD-2 } x \ y; \neg \text{member DenyAll } x; \neg \text{member DenyAll } y;$
 $a = (\text{first-srcNet } x); b = (\text{first-destNet } x) \rrbracket \implies$
 $\text{disjSD-2 } ((\text{DenyAllFromTo } a \ b) \oplus (\text{DenyAllFromTo } b \ a) \oplus x) \ y$
 <proof>

lemma *noDAID*[rule-format]: $\text{noDenyAll } p \longrightarrow \text{noDenyAll } (\text{insertDenies } p)$
 <proof>

lemma *isInIDo*[rule-format]: $\text{noDenyAll } p \longrightarrow s \in \text{set } (\text{insertDenies } p) \longrightarrow$
 $(\exists! a. s = (\text{DenyAllFromTo } (\text{first-srcNet } a) (\text{first-destNet } a)) \oplus$
 $(\text{DenyAllFromTo } (\text{first-destNet } a) (\text{first-srcNet } a)) \oplus a \wedge a \in \text{set } p)$
 <proof>

lemma *id-aux1*[rule-format]: $\text{DenyAllFromTo } (\text{first-srcNet } s) (\text{first-destNet } s) \oplus$
 $\text{DenyAllFromTo } (\text{first-destNet } s) (\text{first-srcNet } s) \oplus s \in \text{set } (\text{insertDenies } p)$
 $\longrightarrow s \in \text{set } p$
 <proof>

lemma *id-aux2*:
 $\llbracket \text{noDenyAll } p; (\forall s. s \in \text{set } p \longrightarrow \text{disjSD-2 } a \ s); \neg \text{member DenyAll } a;$
 $((\text{DenyAllFromTo } (\text{first-srcNet } s) (\text{first-destNet } s)) \oplus (\text{DenyAllFromTo } (\text{first-destNet } s) (\text{first-srcNet } s))) \rrbracket$

$(first_destNet\ s)\ (first_srcNet\ s) \oplus s \in set\ (insertDenies\ p) \implies$
 $disjSD-2\ a\ ((DenyAllFromTo\ (first_srcNet\ s)\ (first_destNet\ s)) \oplus$
 $(DenyAllFromTo\ (first_destNet\ s)\ (first_srcNet\ s)) \oplus s)$
 $\langle proof \rangle$

lemma *id-aux4*[rule-format]: $\llbracket noDenyAll\ p; (\forall s. s \in set\ p \longrightarrow$
 $disjSD-2\ a\ s); s \in set\ (insertDenies\ p); \neg member\ DenyAll\ a \rrbracket \implies disjSD-2\ a\ s$
 $\langle proof \rangle$

lemma *sepNetsID*[rule-format]: $noDenyAll1\ p \longrightarrow separated\ p \longrightarrow$
 $separated\ (insertDenies\ p)$
 $\langle proof \rangle$

lemma *aNDDA*[rule-format]: $allNetsDistinct\ p \longrightarrow allNetsDistinct(DenyAll\ \#p)$
 $\langle proof \rangle$

lemma *OTNConc*[rule-format]: $OnlyTwoNets\ (y\ \# z) \longrightarrow OnlyTwoNets\ z$
 $\langle proof \rangle$

lemma *first-bothNetsd*: $\neg member\ DenyAll\ x \implies$
 $first_bothNet\ x = \{first_srcNet\ x, first_destNet\ x\}$
 $\langle proof \rangle$

lemma *bNaux*:
 $\llbracket \neg member\ DenyAll\ x; \neg member\ DenyAll\ y; first_bothNet\ x = first_bothNet\ y \rrbracket$
 $\implies \{first_srcNet\ x, first_destNet\ x\} = \{first_srcNet\ y, first_destNet\ y\}$
 $\langle proof \rangle$

lemma *setPair*: $\{a, b\} = \{a, d\} \implies b = d$
 $\langle proof \rangle$

lemma *setPair1*: $\{a, b\} = \{d, a\} \implies b = d$
 $\langle proof \rangle$

lemma *setPair4*: $\{a, b\} = \{c, d\} \implies a \neq c \implies a = d$
 $\langle proof \rangle$

lemma *otnaux1*: $\{x, y, x, y\} = \{x, y\}$
 $\langle proof \rangle$

lemma *OTNIDaux4*: $\{x, y, x\} = \{y, x\}$
 $\langle proof \rangle$

lemma *setPair5*: $\{a, b\} = \{c, d\} \implies a \neq c \implies a = d$
 $\langle proof \rangle$

lemma *otnaux*:

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

lemma *OTNSepaux*: $\llbracket \text{onlyTwoNets } (a \oplus y) \wedge \text{OnlyTwoNets } z \longrightarrow \\ \text{OnlyTwoNets } (\text{separate } (a \oplus y \# z)); \\ \neg \text{member DenyAll } a; \\ \neg \text{member DenyAll } y; \text{noDenyAll } z; \\ \text{onlyTwoNets } a; \text{OnlyTwoNets } (y \# z); \text{first-bothNet } (a) = \text{first-bothNet } y \rrbracket \\ \implies \text{OnlyTwoNets } (\text{separate } (a \oplus y \# z))$
 <proof>

lemma *OTNSEp*[rule-format]: $\text{noDenyAll1 } p \longrightarrow \text{OnlyTwoNets } p \longrightarrow \\ \text{OnlyTwoNets } (\text{separate } p)$
 <proof>

lemma *nda*[rule-format]: $\text{singleCombinators } (a \# p) \longrightarrow \text{noDenyAll } p \longrightarrow \\ \text{noDenyAll1 } (a \# p)$
 <proof>

lemma *nDAcharn*[rule-format]: $\text{noDenyAll } p = (\forall r \in \text{set } p. \neg \text{member DenyAll } r)$
 <proof>

lemma *nDAeqSet*: $\text{set } p = \text{set } s \implies \text{noDenyAll } p = \text{noDenyAll } s$
 <proof>

lemma *nDASCaux*[rule-format]: $\text{DenyAll} \notin \text{set } p \longrightarrow \text{singleCombinators } p \longrightarrow \\ r \in \text{set } p \longrightarrow \neg \text{member DenyAll } r$
 <proof>

lemma *nDASC*[rule-format]: $\text{wellformed-policy1 } p \longrightarrow \text{singleCombinators } p \longrightarrow \\ \text{noDenyAll1 } p$
 <proof>

lemma *noDAAll*[rule-format]: $\text{noDenyAll } p = (\neg \text{memberP DenyAll } p)$
 <proof>

lemma *memberPsep*[symmetric]: $\text{memberP } x \ p = \text{memberP } x \ (\text{separate } p)$
 <proof>

lemma *noDAsep*[rule-format]: $\text{noDenyAll } p \implies \text{noDenyAll } (\text{separate } p)$
 <proof>

lemma *noDA1sep*[rule-format]: $\text{noDenyAll1 } p \longrightarrow \text{noDenyAll1 } (\text{separate } p)$
 <proof>

lemma *isInAlternativeLista*: $(aa \in \text{set } (\text{net-list-aux } [a])) \implies \\ aa \in \text{set } (\text{net-list-aux } (a \# p))$

<proof>

lemma *isInAlternativeListb*: $(aa \in \text{set } (\text{net-list-aux } p)) \implies$
 $aa \in \text{set } (\text{net-list-aux } (a \# p))$

<proof>

lemma *ANDSepaux*: $\text{allNetsDistinct } (x \# y \# z) \implies \text{allNetsDistinct } (x \oplus y \# z)$

<proof>

lemma *netlistalternativeSeparateaux*:
 $\text{net-list-aux } [y] @ \text{net-list-aux } z = \text{net-list-aux } (y \# z)$

<proof>

lemma *netlistalternativeSeparate*: $\text{net-list-aux } p = \text{net-list-aux } (\text{separate } p)$

<proof>

lemma *ANDSepaux2*: $\llbracket \text{allNetsDistinct } (x \# y \# z);$
 $\text{allNetsDistinct } (\text{separate } (y \# z)) \rrbracket$
 $\implies \text{allNetsDistinct } (x \# \text{separate } (y \# z))$

<proof>

lemma *ANDSep[rule-format]*: $\text{allNetsDistinct } p \longrightarrow \text{allNetsDistinct}(\text{separate } p)$

<proof>

lemma *wp1-alternativesep[rule-format]*: $\text{wellformed-policy1-strong } p \longrightarrow$
 $\text{wellformed-policy1-strong } (\text{separate } p)$

<proof>

lemma *noDAsort[rule-format]*: $\text{noDenyAll1 } p \longrightarrow \text{noDenyAll1 } (\text{sort } p \ l)$

<proof>

lemma *OTNSC[rule-format]*: $\text{singleCombinators } p \longrightarrow \text{OnlyTwoNets } p$

<proof>

lemma *fMTaux*: $\neg \text{member } \text{DenyAll } x \implies \text{first-bothNet } x \neq \{\}$

<proof>

lemma *fl2[rule-format]*: $\text{firstList } (\text{separate } p) = \text{firstList } p$

<proof>

lemma *fl3[rule-format]*: $\text{NetsCollected } p \longrightarrow (\text{first-bothNet } x \neq \text{firstList } p \longrightarrow$
 $(\forall a \in \text{set } p. \text{first-bothNet } x \neq \text{first-bothNet } a) \longrightarrow \text{NetsCollected } (x \# p))$

<proof>

lemma *sortedConc[rule-format]*: $\text{sorted } (a \# p) \ l \longrightarrow \text{sorted } p \ l$

<proof>

lemma *smalleraux2*:

$\{a,b\} \in \text{set } l \implies \{c,d\} \in \text{set } l \implies \{a,b\} \neq \{c,d\} \implies$
 $\text{smaller } (\text{DenyAllFromTo } a \ b) (\text{DenyAllFromTo } c \ d) \ l \implies$
 $\neg \text{smaller } (\text{DenyAllFromTo } c \ d) (\text{DenyAllFromTo } a \ b) \ l$
 ⟨proof⟩

lemma *smalleraux2a*:

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

lemma *smalleraux2b*:

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

lemma *smalleraux2c*:

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

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 $\text{singleCombinators } [a]$
assumes $\text{singleCombinators } [b]$
shows $\neg \text{smaller } b \ a \ l$
 ⟨proof⟩

lemma *smalleraux3a*:

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

lemma *posaux[rule-format]*: $\text{position } a \ l < \text{position } b \ l \longrightarrow a \neq b$
 ⟨proof⟩

lemma *posaux6[rule-format]*: $a \in \text{set } l \longrightarrow b \in \text{set } l \longrightarrow a \neq b \longrightarrow$
 $\text{position } a \ l \neq \text{position } b \ l$
 ⟨proof⟩

lemma *notSmallerTransaux*[rule-format]:

$\llbracket x \neq \text{DenyAll}; r \neq \text{DenyAll}; \text{singleCombinators } [x]; \text{singleCombinators } [y];$
 $\text{singleCombinators } [r]; \neg \text{smaller } y \ x \ l; \text{smaller } x \ y \ l; \text{smaller } x \ r \ l;$
 $\text{smaller } y \ r \ l; \text{in-list } x \ l; \text{in-list } y \ l; \text{in-list } r \ l \rrbracket \implies$
 $\neg \text{smaller } r \ x \ l$

<proof>

lemma *notSmallerTrans*[rule-format]:

$x \neq \text{DenyAll} \longrightarrow r \neq \text{DenyAll} \longrightarrow \text{singleCombinators } (x \# y \# z) \longrightarrow$
 $\neg \text{smaller } y \ x \ l \longrightarrow \text{sorted } (x \# y \# z) \ l \longrightarrow r \in \text{set } z \longrightarrow$
 $\text{all-in-list } (x \# y \# z) \ l \longrightarrow \neg \text{smaller } r \ x \ l$

<proof>

lemma *NCSaux1*[rule-format]:

$\text{noDenyAll } p \longrightarrow \{x, y\} \in \text{set } l \longrightarrow \text{all-in-list } p \ l \longrightarrow \text{singleCombinators } p \longrightarrow$
 $\text{sorted } (\text{DenyAllFromTo } x \ y \ \# \ p) \ l \longrightarrow \{x, y\} \neq \text{firstList } p \longrightarrow$
 $\text{DenyAllFromTo } u \ v \in \text{set } p \longrightarrow \{x, y\} \neq \{u, v\}$

<proof>

lemma *posaux3*[rule-format]:

$a \in \text{set } l \longrightarrow b \in \text{set } l \longrightarrow a \neq b \longrightarrow \text{position } a \ l \neq \text{position } b \ l$

<proof>

lemma *posaux4*[rule-format]: $\text{singleCombinators } [a] \longrightarrow a \neq \text{DenyAll} \longrightarrow$

$b \neq \text{DenyAll} \longrightarrow \text{in-list } a \ l \longrightarrow \text{in-list } b \ l \longrightarrow$
 $\text{smaller } a \ b \ l \longrightarrow x = (\text{bothNet } a) \longrightarrow$

$y = (\text{bothNet } b) \longrightarrow \text{position } x \ l \leq \text{position } y \ l$

<proof>

lemma *NCSaux2*[rule-format]:

$\text{noDenyAll } p \longrightarrow \{a, b\} \in \text{set } l \longrightarrow \text{all-in-list } p \ l \longrightarrow \text{singleCombinators } p \longrightarrow$
 $\text{sorted } (\text{DenyAllFromTo } a \ b \ \# \ p) \ l \longrightarrow \{a, b\} \neq \text{firstList } p \longrightarrow$
 $\text{AllowPortFromTo } u \ v \ w \in \text{set } p \longrightarrow \{a, b\} \neq \{u, v\}$

<proof>

lemma *NCSaux3*[rule-format]:

$\text{noDenyAll } p \longrightarrow \{a, b\} \in \text{set } l \longrightarrow \text{all-in-list } p \ l \longrightarrow \text{singleCombinators } p \longrightarrow$
 $\text{sorted } (\text{AllowPortFromTo } a \ b \ w \ \# \ p) \ l \longrightarrow \{a, b\} \neq \text{firstList } p \longrightarrow$
 $\text{DenyAllFromTo } u \ v \in \text{set } p \longrightarrow \{a, b\} \neq \{u, v\}$

<proof>

lemma *NCSaux4*[rule-format]:

$\text{noDenyAll } p \longrightarrow \{a, b\} \in \text{set } l \longrightarrow \text{all-in-list } p \ l \longrightarrow \text{singleCombinators } p \longrightarrow$
 $\text{sorted } (\text{AllowPortFromTo } a \ b \ c \ \# \ p) \ l \longrightarrow \{a, b\} \neq \text{firstList } p \longrightarrow$
 $\text{AllowPortFromTo } u \ v \ w \in \text{set } p \longrightarrow \{a, b\} \neq \{u, v\}$

<proof>

lemma *NetsCollectedSorted*[rule-format]:

noDenyAll1 p \longrightarrow *all-in-list p l* \longrightarrow *singleCombinators p* \longrightarrow *sorted p l* \longrightarrow
NetsCollected p

\langle *proof* \rangle

lemma *NetsCollectedSort*: *distinct p* \implies *noDenyAll1 p* \implies *all-in-list p l* \implies
singleCombinators p \implies *NetsCollected (sort p l)*

\langle *proof* \rangle

lemma *fBNsep*[rule-format]: $(\forall a \in \text{set } z. \{b, c\} \neq \text{first-bothNet } a) \longrightarrow$
 $(\forall a \in \text{set } (\text{separate } z). \{b, c\} \neq \text{first-bothNet } a)$

\langle *proof* \rangle

lemma *fBNsep1*[rule-format]: $(\forall a \in \text{set } z. \text{first-bothNet } x \neq \text{first-bothNet } a) \longrightarrow$
 $(\forall a \in \text{set } (\text{separate } z). \text{first-bothNet } x \neq \text{first-bothNet } a)$

\langle *proof* \rangle

lemma *NetsCollectedSepauxa*:

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

\langle *proof* \rangle

lemma *NetsCollectedSepaux*:

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

\langle *proof* \rangle

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

\langle *proof* \rangle

lemma *OTNaux*:

$onlyTwoNets\ a \implies \neg member\ DenyAll\ a \implies (x,y) \in sdnets\ a \implies$
 $(x = first-srcNet\ a \wedge y = first-destNet\ a) \vee$
 $(x = first-destNet\ a \wedge y = first-srcNet\ a)$
 <proof>

lemma *sdnets-charn*: $onlyTwoNets\ a \implies \neg member\ DenyAll\ a \implies$
 $sdnets\ a = \{(first-srcNet\ a, first-destNet\ a)\} \vee$
 $sdnets\ a = \{(first-srcNet\ a, first-destNet\ a), (first-destNet\ a, first-srcNet\ a)\}$
 <proof>

lemma *first-bothNet-charn*[rule-format]: $\neg member\ DenyAll\ a \longrightarrow$
 $first-bothNet\ a = \{first-srcNet\ a, first-destNet\ a\}$
 <proof>

lemma *sdnets-noteq*:
 $\llbracket onlyTwoNets\ a; onlyTwoNets\ aa; first-bothNet\ a \neq first-bothNet\ aa;$
 $\neg member\ DenyAll\ a; \neg member\ DenyAll\ aa \rrbracket$
 $\implies sdnets\ a \neq sdnets\ aa$
 <proof>

lemma *fbn-noteq*:
 $\llbracket onlyTwoNets\ a; onlyTwoNets\ aa; first-bothNet\ a \neq first-bothNet\ aa;$
 $\neg member\ DenyAll\ a; \neg member\ DenyAll\ aa; allNetsDistinct\ [a, aa] \rrbracket \implies$
 $first-srcNet\ a \neq first-srcNet\ aa \vee first-srcNet\ a \neq first-destNet\ aa \vee$
 $first-destNet\ a \neq first-srcNet\ aa \vee first-destNet\ a \neq first-destNet\ aa$
 <proof>

lemma *NCisSD2aux*:
 $\llbracket onlyTwoNets\ a; onlyTwoNets\ aa; first-bothNet\ a \neq first-bothNet\ aa;$
 $\neg member\ DenyAll\ a; \neg member\ DenyAll\ aa; allNetsDistinct\ [a, aa] \rrbracket \implies$
 $disjSD-2\ a\ aa$
 <proof>

lemma *ANDaux3*[rule-format]: $y \in set\ xs \longrightarrow a \in set\ (net-list-aux\ [y]) \longrightarrow$
 $a \in set\ (net-list-aux\ xs)$
 <proof>

lemma *ANDaux2*: $allNetsDistinct\ (x \# xs) \implies y \in set\ xs$
 $\implies allNetsDistinct\ [x,y]$
 <proof>

lemma *NCisSD2*[rule-format]:
 $\llbracket \neg member\ DenyAll\ a; OnlyTwoNets\ (a \# p); NetsCollected2\ (a \# p);$
 $NetsCollected\ (a \# p); noDenyAll\ (p); allNetsDistinct\ (a \# p); s \in set\ p \rrbracket \implies$
 $disjSD-2\ a\ s$
 <proof>

lemma *separatedNC*[rule-format]:

$OnlyTwoNets\ p \longrightarrow NetsCollected2\ p \longrightarrow NetsCollected\ p \longrightarrow noDenyAll1\ p \longrightarrow$
 $allNetsDistinct\ p \longrightarrow separated\ p$
 $\langle proof \rangle$

lemma $NC2Sep[rule-format]: noDenyAll1\ p \longrightarrow NetsCollected2\ (separate\ p)$
 $\langle proof \rangle$

lemma $separatedSep[rule-format]:$
 $OnlyTwoNets\ p \longrightarrow NetsCollected2\ p \longrightarrow NetsCollected\ p \longrightarrow noDenyAll1\ p \longrightarrow$
 $allNetsDistinct\ p \longrightarrow separated\ (separate\ p)$
 $\langle proof \rangle$

lemma $rADnMT[rule-format]: p \neq [] \longrightarrow removeAllDuplicates\ p \neq []$
 $\langle proof \rangle$

lemma $remDupsNMT[rule-format]: p \neq [] \longrightarrow$
 $remdups\ p \neq []$
 $\langle proof \rangle$

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

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

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

lemma $sets-distinct6: (m::int) < n \implies \{(a,b). a = n\} \neq \{(a,b). a = m\}$
 $\langle proof \rangle$

end

theory *NormalisationIntegerPortProof*
imports *NormalisationGenericProofs*
begin

Normalisation proofs which are specific to the IntegerPort address representation.

lemma $ConcAssoc: C((A \oplus B) \oplus D) = C(A \oplus (B \oplus D))$
 $\langle proof \rangle$

lemma *aux26[simp]: twoNetsDistinct a b c d \implies*
 $\text{dom } (C \text{ (AllowPortFromTo a b p)}) \cap \text{dom } (C \text{ (DenyAllFromTo c d)}) = \{\}$
<proof>

lemma *wp2-aux[rule-format]: wellformed-policy2 (xs @ [x]) \longrightarrow*
wellformed-policy2 xs
<proof>

lemma *Cdom2: x $\in \text{dom}(C b) \implies C (a \oplus b) x = (C b) x$*
<proof>

lemma *wp2Conc[rule-format]: wellformed-policy2 (x#xs) \implies wellformed-policy2 xs*
<proof>

lemma *DAimpliesMR-E[rule-format]: DenyAll $\in \text{set } p \longrightarrow$*
 $(\exists r. \text{applied-rule-rev } C x p = \text{Some } r)$
<proof>

lemma *DAimplieMR[rule-format]: DenyAll $\in \text{set } p \implies \text{applied-rule-rev } C x p \neq \text{None}$*
<proof>

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

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

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

lemma *CConcEnd[rule-format]: C a x = Some y \longrightarrow*
 $C (\text{list2FWpolicy } (xs @ [a])) x = \text{Some } y$
 $(\text{is ?P } xs)$
<proof>

lemma *CConcStartaux: $\llbracket C a x = \text{None} \rrbracket \implies (C aa ++ C a) x = C aa x$*
<proof>

lemma *CConcStart[rule-format]: xs $\neq [] \longrightarrow C a x = \text{None} \longrightarrow$*
 $C (\text{list2FWpolicy } (xs @ [a])) x = C (\text{list2FWpolicy } xs) x$
<proof>

lemma *mrNnt[simp]: applied-rule-rev C x p = Some a $\implies p \neq []$*
<proof>

lemma *mr-is-C[rule-format]*: *applied-rule-rev C x p = Some a* \longrightarrow
 $C (list2FWpolicy (p)) x = C a x$
 <proof>

lemma *CConcStart2*: $\llbracket p \neq []; x \notin dom (C a) \rrbracket \implies$
 $C (list2FWpolicy (p@[a])) x = C (list2FWpolicy p)x$
 <proof>

lemma *CConcEnd1*: $\llbracket q@p \neq []; x \notin dom (C a) \rrbracket \implies$
 $C (list2FWpolicy (q@p@[a])) x = C (list2FWpolicy (q@p))x$
 <proof>

lemma *CConcEnd2[rule-format]*: $x \in dom (C a) \longrightarrow$
 $C (list2FWpolicy (xs @ [a])) x = C a x$
 (is ?P xs)
 <proof>

lemma *bar3*: $x \in dom (C (list2FWpolicy (xs @ [xa]))) \implies$
 $x \in dom (C (list2FWpolicy xs)) \vee x \in dom (C xa)$
 <proof>

lemma *CeqEnd[rule-format,simp]*: $a \neq [] \longrightarrow x \in dom (C (list2FWpolicy a)) \longrightarrow$
 $C (list2FWpolicy (b@a)) x = (C (list2FWpolicy a)) x$
 <proof>

lemma *CConcStartA[rule-format,simp]*: $x \in dom (C a) \longrightarrow$
 $x \in dom (C (list2FWpolicy (a \# b)))$
 (is ?P b)
 <proof>

lemma *domConc*: $\llbracket x \in dom (C (list2FWpolicy b)); b \neq [] \rrbracket \implies$
 $x \in dom (C (list2FWpolicy (a@b)))$
 <proof>

lemma *CeqStart[rule-format,simp]*:
 $x \notin dom (C (list2FWpolicy a)) \longrightarrow a \neq [] \longrightarrow b \neq [] \longrightarrow$
 $C (list2FWpolicy (b@a)) x = (C (list2FWpolicy b)) x$
 <proof>

lemma *C-eq-if-mr-eq2*: $\llbracket applied-rule-rev C x a = Some r; applied-rule-rev C x b = Some r; a \neq []; b \neq [] \rrbracket \implies$
 $(C (list2FWpolicy a)) x = (C (list2FWpolicy b)) x$
 <proof>

lemma *nMRtoNone[rule-format]*: $p \neq [] \longrightarrow applied-rule-rev C x p = None \longrightarrow$

$C \text{ (list2FWpolicy } p) \ x = \text{None}$
 $\langle \text{proof} \rangle$

lemma *C-eq-if-mr-eq*:
 $\llbracket \text{applied-rule-rev } C \ x \ b = \text{applied-rule-rev } C \ x \ a; a \neq []; b \neq [] \rrbracket \implies$
 $(C \text{ (list2FWpolicy } a)) \ x = (C \text{ (list2FWpolicy } b)) \ x$
 $\langle \text{proof} \rangle$

lemma *notmatching-notdom*: $\text{applied-rule-rev } C \ x \ (p@[a]) \neq \text{Some } a \implies x \notin \text{dom } (C \ a)$
 $\langle \text{proof} \rangle$

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

lemma *foo3D*: $\llbracket \text{wellformed-policy1 } p; p = (\text{DenyAll}\#ps);$
 $\text{applied-rule-rev } C \ x \ p = \text{Some } \text{DenyAll}; r \in \text{set } ps \rrbracket \implies x \notin \text{dom } (C \ r)$
 $\langle \text{proof} \rangle$

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

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

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

lemma *mr-chnr[rule-format]*: $a \in \text{set } p \longrightarrow (x \in \text{dom } (C \ a)) \longrightarrow$
 $(\forall \ r. r \in \text{set } p \wedge x \in \text{dom } (C \ r) \longrightarrow r = a) \longrightarrow$
 $\text{applied-rule-rev } C \ x \ p = \text{Some } a$
 $\langle \text{proof} \rangle$

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

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

lemma *wp3tl[rule-format]*: $\text{wellformed-policy3 } p \longrightarrow \text{wellformed-policy3 } (tl \ p)$
 $\langle \text{proof} \rangle$

lemma *wp3Conc[rule-format]: wellformed-policy3 (a#p) \longrightarrow wellformed-policy3 p*
<proof>

lemma *foo98[rule-format]: applied-rule-rev C x (aa # p) = Some a \longrightarrow x \in dom (C r) \longrightarrow*
r \in set p
 \longrightarrow a \in set p
<proof>

lemma *mrMTNone[simp]: applied-rule-rev C x [] = None*
<proof>

lemma *DAAux[simp]: x \in dom (C DenyAll)*
<proof>

lemma *mrSet[rule-format]: applied-rule-rev C x p = Some r \longrightarrow r \in set p*
<proof>

lemma *mr-not-Conc: singleCombinators p \implies applied-rule-rev C x p \neq Some (a \oplus b)*
<proof>

lemma *foo25[rule-format]: wellformed-policy3 (p@[x]) \longrightarrow wellformed-policy3 p*
<proof>

lemma *mr-in-dom[rule-format]: applied-rule-rev C x p = Some a \longrightarrow x \in dom (C a)*
<proof>

lemma *wp3EndMT[rule-format]: wellformed-policy3 (p@[xs]) \longrightarrow*
AllowPortFromTo a b po \in set p \longrightarrow
dom (C (AllowPortFromTo a b po)) \cap dom (C xs) = {}
<proof>

lemma *foo29: [dom (C a) \neq {}; dom (C a) \cap dom (C b) = {}] \implies a \neq b*
<proof>

lemma *foo28: [AllowPortFromTo a b po \in set p;*
dom (C (AllowPortFromTo a b po)) \neq {}; (wellformed-policy3 (p@[x]))]
 \implies x \neq AllowPortFromTo a b po
<proof>

lemma *foo28a[rule-format]: x \in dom (C a) \implies dom (C a) \neq {}*
<proof>

lemma *allow-deny-dom[simp]: dom (C (AllowPortFromTo a b po)) \subseteq*
dom (C (DenyAllFromTo a b))

$\langle \text{proof} \rangle$

lemma *DenyAllowDisj*: $\text{dom } (C \text{ (AllowPortFromTo } a \ b \ p)) \neq \{\} \implies$
 $\text{dom } (C \text{ (DenyAllFromTo } a \ b)) \cap \text{dom } (C \text{ (AllowPortFromTo } a \ b \ p)) \neq \{\}$

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

lemma *deny-dom[simp]*: $\text{twoNetsDistinct } a \ b \ c \ d \implies \text{dom } (C \text{ (DenyAllFromTo } a \ b)) \cap$
 $\text{dom } (C \text{ (DenyAllFromTo } c \ d)) = \{\}$

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

lemma *DomInterAllowsMT*: $\llbracket \text{twoNetsDistinct } a \ b \ c \ d \rrbracket \implies$
 $\text{dom } (C \text{ (AllowPortFromTo } a \ b \ p)) \cap \text{dom } (C \text{ (AllowPortFromTo } c \ d \ po)) = \{\}$

$\langle \text{proof} \rangle$

lemma *DomInterAllowsMT-Ports*: $\llbracket p \neq po \rrbracket \implies$
 $\text{dom } (C \text{ (AllowPortFromTo } a \ b \ p)) \cap \text{dom } (C \text{ (AllowPortFromTo } c \ d \ po)) = \{\}$

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

lemma *DistinctNetsDenyAllow*:
 $\llbracket \text{DenyAllFromTo } b \ c \in \text{set } p; \text{AllowPortFromTo } a \ d \ po \in \text{set } p; \text{allNetsDistinct } p;$
 $\text{dom } (C \text{ (DenyAllFromTo } b \ c)) \cap \text{dom } (C \text{ (AllowPortFromTo } a \ d \ po)) \neq \{\} \rrbracket$
 $\implies b = a \wedge c = d$

$\langle \text{proof} \rangle$

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
 $\implies b = a \wedge c = d \wedge po = po$

$\langle \text{proof} \rangle$

lemma *WP2RS2[simp]*:

$\llbracket \text{singleCombinators } p;$
 $\text{distinct } p;$
 $\text{allNetsDistinct } p \rrbracket \implies$
 $\text{wellformed-policy2 } (\text{removeShadowRules2 } p)$

$\langle \text{proof} \rangle$

lemma *AD-aux*: $\llbracket \text{AllowPortFromTo } a \ b \ po \in \text{set } p; \text{DenyAllFromTo } c \ d \in \text{set } p;$

$\text{allNetsDistinct } p; \text{singleCombinators } p;$

$a \neq c \vee b \neq d \rrbracket$

$\implies \text{dom } (C \ (\text{AllowPortFromTo } a \ b \ po)) \cap \text{dom } (C \ (\text{DenyAllFromTo } c \ d)) = \{\}$

$\langle \text{proof} \rangle$

lemma *sorted-WP2[rule-format]*: $\text{sorted } p \ l \longrightarrow \text{all-in-list } p \ l \longrightarrow \text{distinct } p \longrightarrow$

$\text{allNetsDistinct } p \longrightarrow \text{singleCombinators } p \longrightarrow \text{wellformed-policy2 } p$

$\langle \text{proof} \rangle$

lemma *wellformed2-sorted[simp]*: $\llbracket \text{all-in-list } p \ l; \text{distinct } p; \text{allNetsDistinct } p;$

$\text{singleCombinators } p \rrbracket \implies \text{wellformed-policy2 } (\text{sort } p \ l)$

$\langle \text{proof} \rangle$

lemma *wellformed2-sortedQ[simp]*: $\llbracket \text{all-in-list } p \ l; \text{distinct } p; \text{allNetsDistinct } p;$

$\text{singleCombinators } p \rrbracket \implies \text{wellformed-policy2 } (\text{qsort } p \ l)$

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

lemma *C-eq-RS1n*:

$C(\text{list2FWpolicy } (\text{removeShadowRules1-alternative } p)) = C(\text{list2FWpolicy } p)$
 <proof>

lemma *C-eq-RS1[simp]*: $p \neq [] \implies$
 $C(\text{list2FWpolicy } (\text{removeShadowRules1 } p)) = C(\text{list2FWpolicy } p)$
 <proof>

lemma *EX-MR-aux[rule-format]*: $\text{applied-rule-rev } C \ x \ (\text{DenyAll } \# \ p) \neq \text{Some } \text{DenyAll} \longrightarrow$
 $(\exists y. \text{applied-rule-rev } C \ x \ p = \text{Some } y)$
 <proof>

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

lemma *mr-not-DA*:
 $\llbracket \text{wellformed-policy1-strong } s; \text{applied-rule-rev } C \ x \ p = \text{Some } (\text{DenyAllFromTo } a \ ab);$
 $\text{set } p = \text{set } s \rrbracket \implies \text{applied-rule-rev } C \ x \ s \neq \text{Some } \text{DenyAll}$
 <proof>

lemma *domsMT-notND-DD*:
 $\llbracket \text{dom } (C \ (\text{DenyAllFromTo } a \ b)) \cap \text{dom } (C \ (\text{DenyAllFromTo } c \ d)) \neq \{\} \rrbracket \implies$
 $\neg \text{netsDistinct } a \ c$
 <proof>

lemma *domsMT-notND-DD2*:
 $\llbracket \text{dom } (C \ (\text{DenyAllFromTo } a \ b)) \cap \text{dom } (C \ (\text{DenyAllFromTo } c \ d)) \neq \{\} \rrbracket \implies$
 $\neg \text{netsDistinct } b \ d$
 <proof>

lemma *domsMT-notND-DD3*:
 $\llbracket x \in \text{dom } (C \ (\text{DenyAllFromTo } a \ b)); x \in \text{dom } (C \ (\text{DenyAllFromTo } c \ d)) \rrbracket \implies$
 $\neg \text{netsDistinct } a \ c$
 <proof>

lemma *domsMT-notND-DD4*:
 $\llbracket x \in \text{dom } (C \ (\text{DenyAllFromTo } a \ b)); x \in \text{dom } (C \ (\text{DenyAllFromTo } c \ d)) \rrbracket \implies$
 $\neg \text{netsDistinct } b \ d$
 <proof>

lemma *NetsEq-if-sameP-DD*:
 $\llbracket \text{allNetsDistinct } p; u \in \text{set } p; v \in \text{set } p; u = (\text{DenyAllFromTo } a \ b);$
 $v = (\text{DenyAllFromTo } c \ d); x \in \text{dom } (C \ (u)); x \in \text{dom } (C \ (v)) \rrbracket \implies$
 $a = c \wedge b = d$

<proof>

lemma *rule-charn1*:
 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* = *AllowPortFromTo* *a* *b* *po* ∨ *r* = *DenyAllFromTo* *a* *b* ∨ *r* = *DenyAll*)
<proof>

lemma *noneMTsubset[rule-format]*: *noneMT* *C* *a* \longrightarrow *set* *b* \subseteq *set* *a* \longrightarrow *noneMT* *C* *b*
<proof>

lemma *nMTSort*: *noneMT* *C* *p* \implies *noneMT* *C* (*sort* *p* *l*)
<proof>

lemma *nMTSortQ*: *noneMT* *C* *p* \implies *noneMT* *C* (*qsort* *p* *l*)
<proof>

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*
<proof>

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*
<proof>

lemma *rule-charn2*:
 assumes *aND*: *allNetsDistinct* *p*
 and *wp1*: *wellformed-policy1* *p*
 and *SC*: *singleCombinators* *p*
 and *wp3*: *wellformed-policy3* *p*
 and *allow-in-list*: *AllowPortFromTo* *c* *d* *po* ∈ *set* *p*
 and *x-in-dom-allow*: *x* ∈ *dom* (*C* (*AllowPortFromTo* *c* *d* *po*))
 shows *applied-rule-rev* *C* *x* *p* = *Some* (*AllowPortFromTo* *c* *d* *po*)

$\langle \text{proof} \rangle$

lemma *rule-charn3*:

$\llbracket \text{wellformed-policy1 } p; \text{allNetsDistinct } p; \text{singleCombinators } p;$
 $\text{wellformed-policy3 } p; \text{applied-rule-rev } C \ x \ p = \text{Some } (\text{DenyAllFromTo } c \ d);$
 $\text{AllowPortFromTo } a \ b \ po \in \text{set } p \rrbracket \implies x \notin \text{dom } (C \ (\text{AllowPortFromTo } a \ b \ po))$

$\langle \text{proof} \rangle$

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*
 $\langle \text{proof} \rangle$

lemma *foo31a*: $\llbracket (\forall \ r. \ r \in \text{set } p \wedge x \in \text{dom } (C \ r) \longrightarrow$
 $(r = \text{AllowPortFromTo } a \ b \ po \vee r = \text{DenyAllFromTo } a \ b \vee r = \text{DenyAll}));$
 $\text{set } p = \text{set } s; r \in \text{set } s; x \in \text{dom } (C \ r) \rrbracket \implies$
 $(r = \text{AllowPortFromTo } a \ b \ po \vee r = \text{DenyAllFromTo } a \ b \vee r = \text{DenyAll})$
 $\langle \text{proof} \rangle$

lemma *aux4*[*rule-format*]:

applied-rule-rev *C* *x* (*a*#*p*) = *Some* *a* $\longrightarrow a \notin \text{set } (p) \longrightarrow \text{applied-rule-rev } C \ x \ p = \text{None}$
 $\langle \text{proof} \rangle$

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*
 $\langle \text{proof} \rangle$

lemma *rule-charnDAFT*:

$\llbracket \text{wellformed-policy1-strong } p; \text{allNetsDistinct } p; \text{singleCombinators } p;$
 $\text{wellformed-policy3 } p; \text{applied-rule-rev } C \ x \ p = \text{Some } (\text{DenyAllFromTo } a \ b);$
 $r \in \text{set } (\text{tl } p); x \in \text{dom } (C \ r) \rrbracket \implies r = \text{DenyAllFromTo } a \ b$
 $\langle \text{proof} \rangle$

lemma *mrDenyAll-is-unique*:

$\llbracket \text{wellformed-policy1-strong } p; \text{applied-rule-rev } C \ x \ p = \text{Some } \text{DenyAll};$
 $r \in \text{set } (\text{tl } p) \rrbracket \implies x \notin \text{dom } (C \ r)$
 $\langle \text{proof} \rangle$

theorem *C-eq-Sets-mr*:

assumes *sets-eq*: $\text{set } p = \text{set } s$
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>

lemma *C-eq-Sets*:

$\llbracket \text{singleCombinators } p; \text{wellformed-policy1-strong } p; \text{wellformed-policy1-strong } s;$
 $\text{wellformed-policy3 } p; \text{wellformed-policy3 } s; \text{allNetsDistinct } p; \text{set } p = \text{set } s \rrbracket \implies$
 $C (\text{list2FWpolicy } p) x = C (\text{list2FWpolicy } s) x$
<proof>

lemma *C-eq-sorted*: $\llbracket \text{distinct } p; \text{all-in-list } p l; \text{singleCombinators } p;$

$\text{wellformed-policy1-strong } p; \text{wellformed-policy3 } p; \text{allNetsDistinct } p \rrbracket \implies$
 $C (\text{list2FWpolicy } (\text{sort } p l)) = C (\text{list2FWpolicy } p)$

<proof>

lemma *C-eq-sortedQ*: $\llbracket \text{distinct } p; \text{all-in-list } p l; \text{singleCombinators } p;$

$\text{wellformed-policy1-strong } p; \text{wellformed-policy3 } p; \text{allNetsDistinct } p \rrbracket \implies$
 $C (\text{list2FWpolicy } (\text{qsort } p l)) = C (\text{list2FWpolicy } p)$

<proof>

lemma *C-eq-RS2-mr*: *applied-rule-rev* *C* *x* (*removeShadowRules2* *p*) = *applied-rule-rev* *C* *x* *p*

<proof>

lemma *C-eq-None[rule-format]*: $p \neq [] \implies \text{applied-rule-rev } C x p = \text{None} \longrightarrow$

$C (\text{list2FWpolicy } p) x = \text{None}$

<proof>

lemma *C-eq-None2*:

$\llbracket a \neq []; b \neq []; \text{applied-rule-rev } C x a = \text{None}; \text{applied-rule-rev } C x b = \text{None} \rrbracket \implies$
 $(C (\text{list2FWpolicy } a)) x = (C (\text{list2FWpolicy } b)) x$

<proof>

lemma *C-eq-RS2: wellformed-policy1-strong* $p \implies$
 $C \text{ (list2FWpolicy (removeShadowRules2 } p)) = C \text{ (list2FWpolicy } p)$
 $\langle \text{proof} \rangle$

lemma *noneMTRS2: noneMT* $C \text{ } p \implies \text{noneMT } C \text{ (removeShadowRules2 } p)$
 $\langle \text{proof} \rangle$

lemma *CconcNone: dom (C a) = {} ; p ≠ []* \implies
 $C \text{ (list2FWpolicy (a \# p)) } x = C \text{ (list2FWpolicy } p) \text{ } x$
 $\langle \text{proof} \rangle$

lemma *noneMTrd[rule-format]: noneMT* $C \text{ } p \longrightarrow \text{noneMT } C \text{ (remdups } p)$
 $\langle \text{proof} \rangle$

lemma *DARS3[rule-format]: DenyAll* $\notin \text{set } p \longrightarrow \text{DenyAll} \notin \text{set (removeShadowRules3 } C \text{ } p)$
 $\langle \text{proof} \rangle$

lemma *DAnMT: dom (C DenyAll) ≠ {}*
 $\langle \text{proof} \rangle$

lemma *DAnMT2: C DenyAll ≠ empty*
 $\langle \text{proof} \rangle$

lemma *wp1n-RS3[rule-format,simp]: wellformed-policy1-strong* $p \longrightarrow$
 $\text{wellformed-policy1-strong (removeShadowRules3 } C \text{ } p)$
 $\langle \text{proof} \rangle$

lemma *AILRS3[rule-format,simp]: all-in-list* $p \text{ } l \longrightarrow$
 $\text{all-in-list (removeShadowRules3 } C \text{ } p) \text{ } l$
 $\langle \text{proof} \rangle$

lemma *SCRS3[rule-format,simp]: singleCombinators* $p \longrightarrow$
 $\text{singleCombinators(removeShadowRules3 } C \text{ } p)$
 $\langle \text{proof} \rangle$

lemma *RS3subset: set (removeShadowRules3 C p) ⊆ set p*
 $\langle \text{proof} \rangle$

lemma *ANDRS3[simp]: [singleCombinators p; allNetsDistinct p]* \implies
 $\text{allNetsDistinct (removeShadowRules3 } C \text{ } p)$

$\langle \text{proof} \rangle$

lemma *nlpaux*: $x \notin \text{dom } (C \ b) \implies C \ (a \oplus b) \ x = C \ a \ x$

$\langle \text{proof} \rangle$

lemma *notindom*[*rule-format*]: $a \in \text{set } p \longrightarrow x \notin \text{dom } (C \ (\text{list2FWpolicy } p)) \longrightarrow x \notin \text{dom } (C \ a)$

$\langle \text{proof} \rangle$

lemma *C-eq-rd*[*rule-format*]: $p \neq [] \implies C \ (\text{list2FWpolicy } (\text{remdups } p)) = C \ (\text{list2FWpolicy } p)$

$\langle \text{proof} \rangle$

lemma *nMT-domMT*: $\llbracket \neg \text{notMTpolicy } C \ p; p \neq [] \rrbracket \implies r \notin \text{dom } (C \ (\text{list2FWpolicy } p))$

$\langle \text{proof} \rangle$

lemma *C-eq-RS3-aux*[*rule-format*]: $\text{notMTpolicy } C \ p \implies C \ (\text{list2FWpolicy } p) \ x = C \ (\text{list2FWpolicy } (\text{removeShadowRules3 } C \ p)) \ x$

$\langle \text{proof} \rangle$

lemma *C-eq-id*: $\text{wellformed-policy1-strong } p \implies C \ (\text{list2FWpolicy } (\text{insertDeny } p)) = C \ (\text{list2FWpolicy } p)$

$\langle \text{proof} \rangle$

lemma *C-eq-RS3*: $\text{notMTpolicy } C \ p \implies C \ (\text{list2FWpolicy } (\text{removeShadowRules3 } C \ p)) = C \ (\text{list2FWpolicy } p)$

$\langle \text{proof} \rangle$

lemma *NMPrd*[*rule-format*]: $\text{notMTpolicy } C \ p \longrightarrow \text{notMTpolicy } C \ (\text{remdups } p)$

$\langle \text{proof} \rangle$

lemma *NMPDA*[*rule-format*]: $\text{DenyAll} \in \text{set } p \longrightarrow \text{notMTpolicy } C \ p$

$\langle \text{proof} \rangle$

lemma *NMPiD*[*rule-format*]: $\text{notMTpolicy } C \ (\text{insertDeny } p)$

$\langle \text{proof} \rangle$

lemma *list2FWpolicy2list*[*rule-format*]: $C \ (\text{list2FWpolicy}(\text{policy2list } p)) = (C \ p)$

$\langle \text{proof} \rangle$

lemmas $C\text{-eq-Lemmas} = \text{noneMTRS2 noneMTrd SCp2l}$
 $\text{wp1n-RS2 wp1ID NMPiD wp1alternative-RS1}$
 $\text{p2lNmt list2FWpolicy2list wellformed-policy3-charn waux2 wp1-eq}$

lemmas $C\text{-eq-subst-Lemmas} = C\text{-eq-sorted } C\text{-eq-sortedQ } C\text{-eq-RS2 } C\text{-eq-rd } C\text{-eq-RS3 } C\text{-eq-id}$

lemma $C\text{-eq-All-untilSorted}$:

$\llbracket \text{DenyAll} \in \text{set } (\text{policy2list } p); \text{ all-in-list } (\text{policy2list } p) \text{ l};$
 $\text{allNetsDistinct } (\text{policy2list } p) \rrbracket \implies$
 $C(\text{list2FWpolicy } (\text{sort } (\text{removeShadowRules2 } (\text{remdups } (\text{removeShadowRules3 } C$
 $(\text{insertDeny } (\text{removeShadowRules1 } (\text{policy2list } p)))))) \text{ l})) = C \text{ p}$
 $\langle \text{proof} \rangle$

lemma $C\text{-eq-All-untilSortedQ}$:

$\llbracket \text{DenyAll} \in \text{set } (\text{policy2list } p); \text{ all-in-list } (\text{policy2list } p) \text{ l};$
 $\text{allNetsDistinct } (\text{policy2list } p) \rrbracket \implies$
 $C(\text{list2FWpolicy } (\text{qsort } (\text{removeShadowRules2 } (\text{remdups } (\text{removeShadowRules3 } C$
 $(\text{insertDeny } (\text{removeShadowRules1 } (\text{policy2list } p)))))) \text{ l})) = C \text{ p}$
 $\langle \text{proof} \rangle$

lemma $C\text{-eq-All-untilSorted-withSimps}$:

$\llbracket \text{DenyAll} \in \text{set } (\text{policy2list } p); \text{ all-in-list } (\text{policy2list } p) \text{ l};$
 $\text{allNetsDistinct } (\text{policy2list } p) \rrbracket \implies$
 $C(\text{list2FWpolicy } (\text{sort } (\text{removeShadowRules2 } (\text{remdups } (\text{removeShadowRules3 } C (\text{insertDeny}$
 $(\text{removeShadowRules1 } (\text{policy2list } p)))))) \text{ l})) = C \text{ p}$
 $\langle \text{proof} \rangle$

lemma $C\text{-eq-All-untilSorted-withSimpsQ}$:

$\llbracket \text{DenyAll} \in \text{set } (\text{policy2list } p); \text{ all-in-list } (\text{policy2list } p) \text{ l};$
 $\text{allNetsDistinct } (\text{policy2list } p) \rrbracket \implies$
 $C(\text{list2FWpolicy } (\text{qsort } (\text{removeShadowRules2 } (\text{remdups } (\text{removeShadowRules3 } C (\text{insertDeny}$
 $(\text{removeShadowRules1 } (\text{policy2list } p)))))) \text{ l})) = C \text{ p}$
 $\langle \text{proof} \rangle$

lemma $\text{InDomConc}[\text{rule-format}]: p \neq [] \longrightarrow x \in \text{dom } (C (\text{list2FWpolicy } (p))) \longrightarrow$
 $x \in \text{dom } (C (\text{list2FWpolicy } (a\#p)))$

$\langle \text{proof} \rangle$

lemma *not-in-member*[rule-format]: $\text{member } a \ b \longrightarrow x \notin \text{dom } (C \ b) \longrightarrow x \notin \text{dom } (C \ a)$
 <proof>

lemma *src-in-sdnets*[rule-format]: $\neg \text{member } \text{DenyAll } x \longrightarrow p \in \text{dom } (C \ x) \longrightarrow$
 $\text{subnetsOfAdr } (\text{src } p) \cap (\text{fst-set } (\text{sdnets } x)) \neq \{\}$
 <proof>

lemma *dest-in-sdnets*[rule-format]: $\neg \text{member } \text{DenyAll } x \longrightarrow p \in \text{dom } (C \ x) \longrightarrow$
 $\text{subnetsOfAdr } (\text{dest } p) \cap (\text{snd-set } (\text{sdnets } x)) \neq \{\}$
 <proof>

lemma *sdnets-in-subnets*[rule-format]: $p \in \text{dom } (C \ x) \longrightarrow \neg \text{member } \text{DenyAll } x \longrightarrow$
 $(\exists (a,b) \in \text{sdnets } x. a \in \text{subnetsOfAdr } (\text{src } p) \wedge b \in \text{subnetsOfAdr } (\text{dest } p))$
 <proof>

lemma *disjSD-no-p-in-both*[rule-format]:
 $\llbracket \text{disjSD-2 } x \ y; \neg \text{member } \text{DenyAll } x; \neg \text{member } \text{DenyAll } y;$
 $p \in \text{dom } (C \ x); p \in \text{dom } (C \ y) \rrbracket \Longrightarrow \text{False}$
 <proof>

lemma *list2FWpolicy-eq*: $zs \neq [] \Longrightarrow$
 $C \ (\text{list2FWpolicy } (x \oplus y \ \# \ z)) \ p = C \ (x \oplus \text{list2FWpolicy } (y \ \# \ z)) \ p$
 <proof>

lemma *dom-sep*[rule-format]: $x \in \text{dom } (C \ (\text{list2FWpolicy } p)) \longrightarrow$
 $x \in \text{dom } (C \ (\text{list2FWpolicy } (\text{separate } p)))$
 <proof>

lemma *domdConcStart*[rule-format]: $x \in \text{dom } (C \ (\text{list2FWpolicy } (a \ \# \ b))) \longrightarrow$
 $x \notin \text{dom } (C \ (\text{list2FWpolicy } b))$
 $\longrightarrow x \in \text{dom } (C \ (a))$
 <proof>

lemma *sep-dom2-aux*: $\llbracket x \in \text{dom } (C \ (\text{list2FWpolicy } (a \oplus y \ \# \ z))) \rrbracket$
 $\Longrightarrow x \in \text{dom } (C \ (a \oplus \text{list2FWpolicy } (y \ \# \ z)))$
 <proof>

lemma *sep-dom2-aux2*:
 $\llbracket (x \in \text{dom } (C \ (\text{list2FWpolicy } (\text{separate } (y \ \# \ z)))) \longrightarrow$
 $x \in \text{dom } (C \ (\text{list2FWpolicy } (y \ \# \ z)));$
 $x \in \text{dom } (C \ (\text{list2FWpolicy } (a \ \# \ \text{separate } (y \ \# \ z)))) \rrbracket$

$\implies x \in \text{dom } (C \text{ (list2FWpolicy } (a \oplus y \# z)))$
 $\langle \text{proof} \rangle$

lemma *sep-dom2*[rule-format]:
 $x \in \text{dom } (C \text{ (list2FWpolicy } (\text{separate } p))) \longrightarrow x \in \text{dom } (C \text{ (list2FWpolicy } (p)))$
 $\langle \text{proof} \rangle$

lemma *sepDom*: $\text{dom } (C \text{ (list2FWpolicy } p)) = \text{dom } (C \text{ (list2FWpolicy } (\text{separate } p)))$
 $\langle \text{proof} \rangle$

lemma *C-eq-s-ext*[rule-format]: $p \neq [] \longrightarrow$
 $C \text{ (list2FWpolicy } (\text{separate } p)) \ a = C \text{ (list2FWpolicy } p) \ a$
 $\langle \text{proof} \rangle$

lemma *C-eq-s*: $p \neq [] \implies C \text{ (list2FWpolicy } (\text{separate } p)) = C \text{ (list2FWpolicy } p)$
 $\langle \text{proof} \rangle$

lemma *sortnMTQ*: $p \neq [] \implies \text{qsort } p \ l \neq []$
 $\langle \text{proof} \rangle$

lemmas *C-eq-Lemmas-sep* =
C-eq-Lemmas sortnMT sortnMTQ RS2-NMT NMPrd NMPRS3

lemma *C-eq-until-separated*:
 $\llbracket \text{DenyAll} \in \text{set } (\text{policy2list } p); \text{all-in-list } (\text{policy2list } p) \ l; \text{allNetsDistinct } (\text{policy2list } p) \rrbracket \implies$
 $C \text{ (list2FWpolicy } (\text{separate } (\text{sort } (\text{removeShadowRules2 } (\text{remdups } (\text{removeShadowRules3 } C$
 $(\text{insertDeny } (\text{removeShadowRules1 } (\text{policy2list } p)))))) \ l))) = C \ p$
 $\langle \text{proof} \rangle$

lemma *C-eq-until-separatedQ*:
 $\llbracket \text{DenyAll} \in \text{set } (\text{policy2list } p); \text{all-in-list } (\text{policy2list } p) \ l; \text{allNetsDistinct } (\text{policy2list } p) \rrbracket \implies$
 $C \text{ (list2FWpolicy } (\text{separate } (\text{qsort } (\text{removeShadowRules2 } (\text{remdups } (\text{removeShadowRules3 } C$
 $(\text{insertDeny } (\text{removeShadowRules1 } (\text{policy2list } p)))))) \ l))) = C \ p$
 $\langle \text{proof} \rangle$

lemma *domID*[rule-format]: $p \neq [] \wedge x \in \text{dom}(C(\text{list2FWpolicy } p)) \longrightarrow$
 $x \in \text{dom } (C(\text{list2FWpolicy}(\text{insertDenies } p)))$
 $\langle \text{proof} \rangle$

lemma *DA-is-deny*:

$x \in \text{dom } (C (\text{DenyAllFromTo } a \ b \oplus \text{DenyAllFromTo } b \ a \oplus \text{DenyAllFromTo } a \ b)) \implies$
 $C (\text{DenyAllFromTo } a \ b \oplus \text{DenyAllFromTo } b \ a \oplus \text{DenyAllFromTo } a \ b) \ x = \text{Some } (\text{deny } ())$
 $\langle \text{proof} \rangle$

lemma *iDdomAux*[rule-format]:
 $p \neq [] \longrightarrow x \notin \text{dom } (C (\text{list2FWpolicy } p)) \longrightarrow$
 $x \in \text{dom } (C (\text{list2FWpolicy } (\text{insertDenies } p))) \longrightarrow$
 $C (\text{list2FWpolicy } (\text{insertDenies } p)) \ x = \text{Some } (\text{deny } ())$
 $\langle \text{proof} \rangle$

lemma *iD-isD*[rule-format]: $p \neq [] \longrightarrow x \notin \text{dom } (C (\text{list2FWpolicy } p))$
 $\longrightarrow C (\text{DenyAll} \oplus \text{list2FWpolicy } (\text{insertDenies } p)) \ x = C \text{DenyAll } x$
 $\langle \text{proof} \rangle$

lemma *inDomConc*: $\llbracket x \notin \text{dom } (C \ a); x \notin \text{dom } (C (\text{list2FWpolicy } p)) \rrbracket \implies$
 $x \notin \text{dom } (C (\text{list2FWpolicy } (a \# p)))$
 $\langle \text{proof} \rangle$

lemma *domsdisj*[rule-format]: $p \neq [] \longrightarrow (\forall \ x \ s. \ s \in \text{set } p \wedge x \in \text{dom } (C \ A) \longrightarrow$
 $x \notin \text{dom } (C \ s)) \longrightarrow y \in \text{dom } (C \ A) \longrightarrow$
 $y \notin \text{dom } (C (\text{list2FWpolicy } p))$
 $\langle \text{proof} \rangle$

lemma *isSepaux*:
 $\llbracket p \neq []; \text{noDenyAll } (a \# p); \text{separated } (a \ \# \ p);$
 $x \in \text{dom } (C (\text{DenyAllFromTo } (\text{first-srcNet } a) (\text{first-destNet } a) \oplus$
 $\text{DenyAllFromTo } (\text{first-destNet } a) (\text{first-srcNet } a) \oplus a)) \rrbracket \implies$
 $x \notin \text{dom } (C (\text{list2FWpolicy } p))$
 $\langle \text{proof} \rangle$

lemma *noneMTsep*[rule-format]: $\text{noneMT } C \ p \longrightarrow \text{noneMT } C \ (\text{separate } p)$
 $\langle \text{proof} \rangle$

lemma *dom-id*:
 $\llbracket \text{noDenyAll } (a \# p); \text{separated } (a \# p); p \neq []; x \notin \text{dom } (C (\text{list2FWpolicy } p));$
 $x \in \text{dom } (C \ (a)) \rrbracket$
 $\implies x \notin \text{dom } (C (\text{list2FWpolicy } (\text{insertDenies } p)))$
 $\langle \text{proof} \rangle$

lemma *C-eq-iD-aux2*[rule-format]:
 $\text{noDenyAll1 } p \longrightarrow$
 $\text{separated } p \longrightarrow$
 $p \neq [] \longrightarrow$
 $x \in \text{dom } (C (\text{list2FWpolicy } p)) \longrightarrow$

$C(\text{list2FWpolicy } (\text{insertDenies } p)) \ x = C(\text{list2FWpolicy } p) \ x$
 <proof>

lemma *C-eq-iD*: $\llbracket \text{separated } p; \text{noDenyAll1 } p; \text{wellformed-policy1-strong } p \rrbracket \implies$
 $C(\text{list2FWpolicy } (\text{insertDenies } p)) = C(\text{list2FWpolicy } p)$
 <proof>

lemma *noDAsortQ[rule-format]*: $\text{noDenyAll1 } p \longrightarrow \text{noDenyAll1 } (\text{qsort } p \ l)$
 <proof>

lemma *NetsCollectedSortQ*: $\text{distinct } p \implies \text{noDenyAll1 } p \implies \text{all-in-list } p \ l \implies$
 $\text{singleCombinators } p \implies \text{NetsCollected } (\text{qsort } p \ l)$
 <proof>

lemmas *CLemmas* = *nMTSort nMTSortQ noneMTRS2 noneMTrd*
noDAsort noDAsortQ nDASC wp1-eq wp1IID
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*: $\llbracket \text{DenyAll} \in \text{set } (\text{policy2list } p); \text{all-in-list}$
 $(\text{policy2list } p) \ l; \text{allNetsDistinct } (\text{policy2list } p) \rrbracket \implies$
 $C(\text{list2FWpolicy } ((\text{insertDenies } (\text{separate } (\text{sort } (\text{removeShadowRules2 } (\text{remdups}$
 $(\text{removeShadowRules3 } C(\text{insertDeny } (\text{removeShadowRules1 } (\text{policy2list } p)))))) \ l)))))) =$
 $C \ p$
 <proof>

lemma *C-eq-Until-InsertDeniesQ*: $\llbracket \text{DenyAll} \in \text{set } (\text{policy2list } p); \text{all-in-list}$
 $(\text{policy2list } p) \ l; \text{allNetsDistinct } (\text{policy2list } p) \rrbracket \implies$
 $C(\text{list2FWpolicy } ((\text{insertDenies } (\text{separate } (\text{qsort } (\text{removeShadowRules2 } (\text{remdups}$
 $(\text{removeShadowRules3 } C(\text{insertDeny } (\text{removeShadowRules1 } (\text{policy2list } p)))))) \ l)))))) =$
 $C \ p$
 <proof>

lemma *C-eq-RD-aux[rule-format]*: $C(p) \ x = C(\text{removeDuplicates } p) \ x$
 <proof>

lemma *C-eq-RAD-aux*[rule-format]:

$p \neq [] \longrightarrow C \text{ (list2FWpolicy } p) x = C \text{ (list2FWpolicy (removeAllDuplicates } p)) x$
 <proof>

lemma *C-eq-RAD*:

$p \neq [] \implies C \text{ (list2FWpolicy } p) = C \text{ (list2FWpolicy (removeAllDuplicates } p))$
 <proof>

lemma *C-eq-compile*:

$\llbracket \text{DenyAll} \in \text{set (policy2list } p); \text{all-in-list (policy2list } p) \text{ } l; \text{allNetsDistinct (policy2list } p) \rrbracket \implies$
 $C \text{ (list2FWpolicy (removeAllDuplicates (insertDenies (separate (sort$
 $\text{(removeShadowRules2 (remdups (removeShadowRules3 } C \text{ (insertDeny$
 $\text{(removeShadowRules1 (policy2list } p)))))) \text{ } l)))) = C p$
 <proof>

lemma *C-eq-compileQ*:

$\llbracket \text{DenyAll} \in \text{set (policy2list } p); \text{all-in-list (policy2list } p) \text{ } l; \text{allNetsDistinct (policy2list } p) \rrbracket \implies$
 $C \text{ (list2FWpolicy (removeAllDuplicates (insertDenies (separate (qsort$
 $\text{(removeShadowRules2 (remdups (removeShadowRules3 } C \text{ (insertDeny$
 $\text{(removeShadowRules1 (policy2list } p)))))) \text{ } l)))) = C p$
 <proof>

lemma *C-eq-normalize*:

$\llbracket \text{DenyAll} \in \text{set (policy2list } p); \text{allNetsDistinct (policy2list } p);$
 $\text{all-in-list (policy2list } p) \text{ (Nets-List } p) \rrbracket \implies$
 $C \text{ (list2FWpolicy (normalize } p)) = C p$
 <proof>

lemma *C-eq-normalizeQ*:

$\llbracket \text{DenyAll} \in \text{set (policy2list } p); \text{allNetsDistinct (policy2list } p);$
 $\text{all-in-list (policy2list } p) \text{ (Nets-List } p) \rrbracket \implies$
 $C \text{ (list2FWpolicy (normalizeQ } p)) = C p$
 <proof>

lemma *domSubset3*: $\text{dom } (C \text{ (DenyAll } \oplus x)) = \text{dom } (C \text{ (DenyAll)})$
 ⟨proof⟩

lemma *domSubset4*: $\text{dom } (C \text{ (DenyAllFromTo } x \ y \oplus \text{DenyAllFromTo } y \ x \oplus \text{AllowPortFromTo } x \ y \text{ dn})) =$
 $\text{dom } (C \text{ (DenyAllFromTo } x \ y \oplus \text{DenyAllFromTo } y \ x))$
 ⟨proof⟩

lemma *domSubset5*:
 $\text{dom } (C \text{ (DenyAllFromTo } x \ y \oplus \text{DenyAllFromTo } y \ x \oplus \text{AllowPortFromTo } y \ x \text{ dn})) =$
 $\text{dom } (C \text{ (DenyAllFromTo } x \ y \oplus \text{DenyAllFromTo } y \ x))$
 ⟨proof⟩

lemma *domSubset1*: $\text{dom } (C \text{ (DenyAllFromTo one two } \oplus \text{DenyAllFromTo two one } \oplus \text{AllowPortFromTo one two dn } \oplus x)) =$
 $\text{dom } (C \text{ (DenyAllFromTo one two } \oplus \text{DenyAllFromTo two one } \oplus x))$
 ⟨proof⟩

lemma *domSubset2*:
 $\text{dom } (C \text{ (DenyAllFromTo one two } \oplus \text{DenyAllFromTo two one } \oplus \text{AllowPortFromTo two one dn } \oplus x)) =$
 $\text{dom } (C \text{ (DenyAllFromTo one two } \oplus \text{DenyAllFromTo two one } \oplus x))$
 ⟨proof⟩

lemma *ConcAssoc2*: $C \text{ (} X \oplus Y \oplus ((A \oplus B) \oplus D)) = C \text{ (} X \oplus Y \oplus A \oplus B \oplus D)$
 ⟨proof⟩

lemma *ConcAssoc3*: $C \text{ (} X \oplus ((Y \oplus A) \oplus D)) = C \text{ (} X \oplus Y \oplus A \oplus D)$
 ⟨proof⟩

lemma *RS3-NMT[rule-format]*: $\text{DenyAll} \in \text{set } p \longrightarrow$
 $\text{removeShadowRules3 } C \ p \neq []$
 ⟨proof⟩

lemma *norm-notMT*: $\text{DenyAll} \in \text{set } (\text{policy2list } p) \implies \text{normalize } p \neq []$
 ⟨proof⟩

lemma *norm-notMTQ*: $\text{DenyAll} \in \text{set } (\text{policy2list } p) \implies \text{normalizeQ } p \neq []$
 ⟨proof⟩

lemma *domDA*: $\text{dom } (C \text{ (DenyAll } \oplus A)) = \text{dom } (C \text{ (DenyAll)})$
 ⟨proof⟩

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]: $p \in \text{dom } (C \text{ } a) \longrightarrow$
 $C \text{ (list2policyR } (a \# \text{ list})) \text{ } p = C \text{ } a \text{ } p$
 ⟨proof⟩

lemma *list2policyR-End*: $p \notin \text{dom } (C \text{ } a) \implies$
 $C \text{ (list2policyR } (a \# \text{ list})) \text{ } p = (C \text{ } a \oplus \text{ list2policy } (\text{map } C \text{ list})) \text{ } p$
 ⟨proof⟩

lemma *l2polR-eq-el*[rule-format]: $N \neq [] \longrightarrow$
 $C \text{ (list2policyR } N) \text{ } p = \text{ (list2policy } (\text{map } C \text{ } N)) \text{ } p$
 ⟨proof⟩

lemma *l2polR-eq*: $N \neq [] \implies$
 $C \text{ (list2policyR } N) = \text{ (list2policy } (\text{map } C \text{ } N))$
 ⟨proof⟩

lemma *list2FWpolicys-eq-el*[rule-format]:
 $\text{Filter} \neq [] \longrightarrow C \text{ (list2policyR } \text{Filter}) \text{ } p = C \text{ (list2FWpolicy } (\text{rev } \text{Filter})) \text{ } p$
 ⟨proof⟩

lemma *list2FWpolicys-eq*:
 $\text{Filter} \neq [] \implies$
 $C \text{ (list2policyR } \text{Filter}) = C \text{ (list2FWpolicy } (\text{rev } \text{Filter}))$
 ⟨proof⟩

lemma *list2FWpolicys-eq-sym*:
 $\text{Filter} \neq [] \implies$
 $C \text{ (list2policyR } (\text{rev } \text{Filter})) = C \text{ (list2FWpolicy } \text{Filter})$
 ⟨proof⟩

lemma *p-eq[rule-format]*: $p \neq [] \longrightarrow$
 $list2policy \ (map \ C \ (rev \ p)) = C \ (list2FWpolicy \ p)$
 $\langle proof \rangle$

lemma *p-eq2[rule-format]*: $normalize \ x \neq [] \longrightarrow$
 $C \ (list2FWpolicy \ (normalize \ x)) = C \ x \longrightarrow$
 $list2policy \ (map \ C \ (rev \ (normalize \ x))) = C \ x$
 $\langle proof \rangle$

lemma *p-eq2Q[rule-format]*: $normalizeQ \ x \neq [] \longrightarrow$
 $C \ (list2FWpolicy \ (normalizeQ \ x)) = C \ x \longrightarrow$
 $list2policy \ (map \ C \ (rev \ (normalizeQ \ x))) = C \ x$
 $\langle proof \rangle$

lemma *list2listNMT[rule-format]*: $x \neq [] \longrightarrow map \ sem \ x \neq []$
 $\langle proof \rangle$

lemma *Norm-Distr2*:
 $r \ o\text{-}f \ ((P \otimes_2 \ (list2policy \ Q)) \ o \ d) =$
 $(list2policy \ ((P \otimes_L \ Q) \ (op \otimes_2 \ r \ d)))$
 $\langle proof \rangle$

lemma *NATDistr*: $\llbracket N \neq []; F = C \ (list2policyR \ N) \rrbracket \implies$
 $((\lambda \ (x,y). \ x) \ o\text{-}f \ ((NAT \otimes_2 \ F) \ o \ (\lambda \ x. \ (x,x)))) =$
 $(list2policy \ ((NAT \otimes_L \ (map \ C \ N)) \ (op \otimes_2 \$
 $(\lambda \ (x,y). \ x) \ (\lambda \ x. \ (x,x)))))$
 $\langle proof \rangle$

lemma *C-eq-normalize-manual*:
 $\llbracket DenyAll \in set \ (policy2list \ p);$
 $allNetsDistinct \ (policy2list \ p);$
 $all\text{-}in\text{-}list \ (policy2list \ p) \ ll \rrbracket \implies$
 $C \ (list2FWpolicy \ (normalize\text{-}manual\text{-}order \ p \ l)) = C \ p$
 $\langle proof \rangle$

lemma *p-eq2-manualQ[rule-format]*: $normalize\text{-}manual\text{-}orderQ \ x \ l \neq [] \longrightarrow$
 $C \ (list2FWpolicy \ (normalize\text{-}manual\text{-}orderQ \ x \ l)) = C \ x \longrightarrow$

list2policy (map *C* (rev (normalize-manual-orderQ *x l*))) = *C x*
 <proof>

lemma *norm-notMT-manualQ*: *DenyAll* ∈ set (*policy2list p*) ⇒ *normalize-manual-orderQ p*
l ≠ []
 <proof>

lemma *C-eq-normalize-manualQ*:
 [[*DenyAll* ∈ set (*policy2list p*);
allNetsDistinct (*policy2list p*);
all-in-list (*policy2list p*) *l*]] ⇒
C (*list2FWpolicy* (*normalize-manual-orderQ p l*)) = *C p*
 <proof>

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*
 <proof>

lemma *norm-notMT-manual*: *DenyAll* ∈ set (*policy2list p*) ⇒ *normalize-manual-order p l* ≠ []
 <proof>

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 (λ *x*. (*x,x*)))) =
list2policy ((*NAT* ⊗_L (map *C* (rev (*normalize Filter*)))) (op ⊗₂) (λ (*x,y*). *x*) (λ *x*. (*x,x*)))
 <proof>

lemma *domSimpl[simp]*: dom (*C* (*A* ⊕ *DenyAll*)) = dom (*C* (*DenyAll*))
 <proof>

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*)

$\langle \text{proof} \rangle$

lemma *DAinRotate*: $\llbracket \text{DenyAll} \in \text{set} (\text{policy2list } p) \rrbracket$
 $\implies \text{DenyAll} \in \text{set} (\text{policy2list} (\text{rotatePolicy } p))$
 $\langle \text{proof} \rangle$

lemma *DAUniv*: $\text{dom} (\text{CRotate} (P \oplus \text{DenyAll})) = \text{UNIV}$
 $\langle \text{proof} \rangle$

lemma *p-eq2R[rule-format]*: $\text{normalize} (\text{rotatePolicy } x) \neq [] \longrightarrow$
 $C (\text{list2FWpolicy} (\text{normalize} (\text{rotatePolicy } x))) = \text{CRotate } x \longrightarrow$
 $\text{list2policy} (\text{map } C (\text{rev} (\text{normalize} (\text{rotatePolicy } x)))) = \text{CRotate } x$
 $\langle \text{proof} \rangle$

lemma *C-eq-normalizeRotate*:
 $\llbracket \text{DenyAll} \in \text{set} (\text{policy2list } p) \rrbracket$;
 $\text{allNetsDistinct} (\text{policy2list} (\text{rotatePolicy } p));$
 $\text{all-in-list} (\text{policy2list} (\text{rotatePolicy } p)) (\text{Nets-List} (\text{rotatePolicy } p)) \rrbracket \implies$
 $C (\text{list2FWpolicy} (\text{removeAllDuplicates} (\text{insertDenies} (\text{separate} (\text{sort}$
 $\quad (\text{removeShadowRules2} (\text{remdups} (\text{removeShadowRules3 } C (\text{insertDeny}$
 $\quad (\text{removeShadowRules1} (\text{policy2list} (\text{rotatePolicy } p)))))))$
 $(\text{Nets-List} (\text{rotatePolicy } p)))))) = \text{CRotate } p$
 $\langle \text{proof} \rangle$

lemma *C-eq-normalizeRotate2*:
 $\llbracket \text{DenyAll} \in \text{set} (\text{policy2list } p) \rrbracket$;
 $\text{allNetsDistinct} (\text{policy2list} (\text{rotatePolicy } p));$
 $\text{all-in-list} (\text{policy2list} (\text{rotatePolicy } p)) (\text{Nets-List} (\text{rotatePolicy } p)) \rrbracket \implies$
 $C (\text{list2FWpolicy} (\text{normalize} (\text{rotatePolicy } p))) = \text{CRotate } p$
 $\langle \text{proof} \rangle$

end

theory *NormalisationIPPProofs*
imports *NormalisationIntegerPortProof*
begin

Normalisation proofs which are specific to the IntegerProtocol address representation.

lemma *ConcAssoc*: $Cp((A \oplus B) \oplus D) = Cp(A \oplus (B \oplus D))$
 <proof>

lemma *aux26[simp]*: $twoNetsDistinct\ a\ b\ c\ d \implies$
 $dom\ (Cp\ (AllowPortFromTo\ a\ b\ p)) \cap dom\ (Cp\ (DenyAllFromTo\ c\ d)) = \{\}$
 <proof>

lemma *wp2-aux[rule-format]*: $wellformed-policy2Pr\ (xs\ @\ [x]) \longrightarrow$
 $wellformed-policy2Pr\ xs$
 <proof>

lemma *Cdom2*: $x \in dom(Cp\ b) \implies Cp\ (a \oplus b)\ x = (Cp\ b)\ x$
 <proof>

lemma *wp2Conc[rule-format]*: $wellformed-policy2Pr\ (x\ \#xs) \implies wellformed-policy2Pr\ xs$
 <proof>

lemma *DAimpliesMR-E[rule-format]*: $DenyAll \in set\ p \longrightarrow$
 $(\exists\ r.\ applied-rule-rev\ Cp\ x\ p = Some\ r)$
 <proof>

lemma *DAimplieMR[rule-format]*: $DenyAll \in set\ p \implies applied-rule-rev\ Cp\ x\ p \neq None$
 <proof>

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

lemma *MRList2*: $x \in dom\ (Cp\ a) \implies applied-rule-rev\ Cp\ x\ (c@b@[a]) = Some\ a$
 <proof>

lemma *MRList3*: $x \notin dom\ (Cp\ xa) \implies$
 $applied-rule-rev\ Cp\ x\ (a\ @\ b\ \#xs\ @\ [xa]) = applied-rule-rev\ Cp\ x\ (a\ @\ b\ \#xs)$
 <proof>

lemma *CConcEnd[rule-format]*: $Cp\ a\ x = Some\ y \longrightarrow$
 $Cp\ (list2FWpolicy\ (xs\ @\ [a]))\ x = Some\ y$
 (is ?P xs)
 <proof>

lemma *CConcStartaux*: $\llbracket Cp\ a\ x = None \rrbracket \implies (Cp\ aa\ ++\ Cp\ a)\ x = Cp\ aa\ x$
 <proof>

lemma *CConcStart[rule-format]*: $xs \neq [] \longrightarrow Cp\ a\ x = None \longrightarrow$
 $Cp\ (list2FWpolicy\ (xs\ @\ [a]))\ x = Cp\ (list2FWpolicy\ xs)\ x$
 <proof>

lemma *mrNnt[simp]: applied-rule-rev Cp x p = Some a \implies p \neq []*
<proof>

lemma *mr-is-C[rule-format]: applied-rule-rev Cp x p = Some a \longrightarrow*
Cp (list2FWpolicy (p)) x = Cp a x
<proof>

lemma *CConcStart2: [p \neq []; x \notin dom (Cp a)] \implies*
Cp (list2FWpolicy (p@[a])) x = Cp (list2FWpolicy p)x
<proof>

lemma *CConcEnd1: [q@[p] \neq []; x \notin dom (Cp a)] \implies*
Cp (list2FWpolicy (q@[p@[a]])) x = Cp (list2FWpolicy (q@[p]))x
<proof>

lemma *CConcEnd2[rule-format]: x \in dom (Cp a) \longrightarrow*
Cp (list2FWpolicy (xs @ [a])) x = Cp a x
(is ?P xs)
<proof>

lemma *bar3: x \in dom (Cp (list2FWpolicy (xs @ [xa]))) \implies*
x \in dom (Cp (list2FWpolicy xs)) \vee x \in dom (Cp xa)
<proof>

lemma *CeqEnd[rule-format,simp]: a \neq [] \longrightarrow x \in dom (Cp (list2FWpolicy a)) \longrightarrow*
Cp (list2FWpolicy (b@a)) x = (Cp (list2FWpolicy a)) x
<proof>

lemma *CConcStartA[rule-format,simp]: x \in dom (Cp a) \longrightarrow*
x \in dom (Cp (list2FWpolicy (a # b)))
(is ?P b)
<proof>

lemma *domConc: [x \in dom (Cp (list2FWpolicy b)); b \neq []] \implies*
x \in dom (Cp (list2FWpolicy (a@b)))
<proof>

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
<proof>

lemma *C-eq-if-mr-eq2: [applied-rule-rev Cp x a = Some r; applied-rule-rev Cp x b = Some r;*
a \neq []; b \neq []] \implies

$(Cp \text{ (list2FWpolicy } a)) \ x = (Cp \text{ (list2FWpolicy } b)) \ x$
 $\langle \text{proof} \rangle$

lemma *nMRtoNone*[rule-format]: $p \neq [] \longrightarrow \text{applied-rule-rev } Cp \ x \ p = \text{None} \longrightarrow$
 $Cp \text{ (list2FWpolicy } p) \ x = \text{None}$
 $\langle \text{proof} \rangle$

lemma *C-eq-if-mr-eq*:
 $\llbracket \text{applied-rule-rev } Cp \ x \ b = \text{applied-rule-rev } Cp \ x \ a; a \neq []; b \neq [] \rrbracket \implies$
 $(Cp \text{ (list2FWpolicy } a)) \ x = (Cp \text{ (list2FWpolicy } b)) \ x$
 $\langle \text{proof} \rangle$

lemma *notmatching-notdom*: $\text{applied-rule-rev } Cp \ x \ (p@[a]) \neq \text{Some } a \implies x \notin \text{dom } (Cp \ a)$
 $\langle \text{proof} \rangle$

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

lemma *foo3D*: $\llbracket \text{wellformed-policy1 } p; p = (\text{DenyAll} \# ps);$
 $\text{applied-rule-rev } Cp \ x \ p = \text{Some } \text{DenyAll}; r \in \text{set } ps \rrbracket \implies x \notin \text{dom } (Cp \ r)$
 $\langle \text{proof} \rangle$

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

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

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

lemma *mr-cha*[rule-format]: $a \in \text{set } p \longrightarrow (x \in \text{dom } (Cp \ a)) \longrightarrow$
 $(\forall \ r. r \in \text{set } p \wedge x \in \text{dom } (Cp \ r) \longrightarrow r = a) \longrightarrow$
 $\text{applied-rule-rev } Cp \ x \ p = \text{Some } a$
 $\langle \text{proof} \rangle$

lemma *foo8*: $\llbracket (\forall \ r. r \in \text{set } p \wedge x \in \text{dom } (Cp \ r) \longrightarrow r = a); \text{set } p = \text{set } s \rrbracket \implies$
 $(\forall \ r. r \in \text{set } s \wedge x \in \text{dom } (Cp \ r) \longrightarrow r = a)$
 $\langle \text{proof} \rangle$

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

lemma *wp3tl[rule-format]: wellformed-policy3Pr p \longrightarrow wellformed-policy3Pr (tl p)*
<proof>

lemma *wp3Conc[rule-format]: wellformed-policy3Pr (a#p) \longrightarrow wellformed-policy3Pr p*
<proof>

lemma *foo98[rule-format]: applied-rule-rev Cp x (aa # p) = Some a \longrightarrow x \in dom (Cp r) \longrightarrow*
r \in set p
 \longrightarrow a \in set p
<proof>

lemma *mrMTNone[simp]: applied-rule-rev Cp x [] = None*
<proof>

lemma *DAAux[simp]: x \in dom (Cp DenyAll)*
<proof>

lemma *mrSet[rule-format]: applied-rule-rev Cp x p = Some r \longrightarrow r \in set p*
<proof>

lemma *mr-not-Conc: singleCombinators p \implies applied-rule-rev Cp x p \neq Some (a \oplus b)*
<proof>

lemma *foo25[rule-format]: wellformed-policy3Pr (p@[x]) \longrightarrow wellformed-policy3Pr p*
<proof>

lemma *mr-in-dom[rule-format]: applied-rule-rev Cp x p = Some a \longrightarrow x \in dom (Cp a)*
<proof>

lemma *wp3EndMT[rule-format]: wellformed-policy3Pr (p@[xs]) \longrightarrow*
AllowPortFromTo a b po \in set p \longrightarrow
dom (Cp (AllowPortFromTo a b po)) \cap dom (Cp xs) = {}
<proof>

lemma *foo29: \llbracket dom (Cp a) \neq {}; dom (Cp a) \cap dom (Cp b) = {} $\rrbracket \implies a \neq b$*
<proof>

lemma *foo28: \llbracket AllowPortFromTo a b po \in set p;*
dom (Cp (AllowPortFromTo a b po)) \neq {}; (wellformed-policy3Pr (p@[x])) \rrbracket
 $\implies x \neq$ AllowPortFromTo a b po
<proof>

lemma *foo28a[rule-format]: x \in dom (Cp a) \implies dom (Cp a) \neq {}*

$\langle \text{proof} \rangle$

lemma *allow-deny-dom[simp]*: $\text{dom } (Cp \text{ (AllowPortFromTo } a \ b \ po)) \subseteq$
 $\text{dom } (Cp \text{ (DenyAllFromTo } a \ b))$

$\langle \text{proof} \rangle$

lemma *DenyAllowDisj*: $\text{dom } (Cp \text{ (AllowPortFromTo } a \ b \ p)) \neq \{\} \implies$
 $\text{dom } (Cp \text{ (DenyAllFromTo } a \ b)) \cap \text{dom } (Cp \text{ (AllowPortFromTo } a \ b \ p)) \neq \{\}$

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

lemma *deny-dom[simp]*: $\text{twoNetsDistinct } a \ b \ c \ d \implies \text{dom } (Cp \text{ (DenyAllFromTo } a \ b)) \cap$
 $\text{dom } (Cp \text{ (DenyAllFromTo } c \ d)) = \{\}$

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

lemma *DomInterAllowsMT*: $\llbracket \text{twoNetsDistinct } a \ b \ c \ d \rrbracket \implies$
 $\text{dom } (Cp \text{ (AllowPortFromTo } a \ b \ p)) \cap \text{dom } (Cp \text{ (AllowPortFromTo } c \ d \ po)) = \{\}$

$\langle \text{proof} \rangle$

lemma *DomInterAllowsMT-Ports*: $\llbracket p \neq po \rrbracket \implies$
 $\text{dom } (Cp \text{ (AllowPortFromTo } a \ b \ p)) \cap \text{dom } (Cp \text{ (AllowPortFromTo } c \ d \ po)) = \{\}$

$\langle \text{proof} \rangle$

lemma *wellformed-policy3-charn[rule-format]*:
 $\text{singleCombinators } p \longrightarrow \text{distinct } p \longrightarrow \text{allNetsDistinct } p \longrightarrow$
 $\text{wellformed-policy1 } p \longrightarrow \text{wellformed-policy2Pr } p \longrightarrow \text{wellformed-policy3Pr } p$

$\langle \text{proof} \rangle$

lemma *DistinctNetsDenyAllow*:

$\llbracket \text{DenyAllFromTo } b \ c \in \text{set } p; \text{AllowPortFromTo } a \ d \ po \in \text{set } p; \text{allNetsDistinct } p; \\ \text{dom } (Cp \ (\text{DenyAllFromTo } b \ c)) \cap \text{dom } (Cp \ (\text{AllowPortFromTo } a \ d \ po)) \neq \{\} \rrbracket \\ \implies b = a \wedge c = d$
 <proof>

lemma *DistinctNetsAllowAllow*:

$\llbracket \text{AllowPortFromTo } b \ c \ po \in \text{set } p; \text{AllowPortFromTo } a \ d \ po \in \text{set } p; \\ \text{allNetsDistinct } p; \text{dom } (Cp \ (\text{AllowPortFromTo } b \ c \ po)) \cap \\ \text{dom } (Cp \ (\text{AllowPortFromTo } a \ d \ po)) \neq \{\} \rrbracket \\ \implies b = a \wedge c = d \wedge po = po$
 <proof>

lemma *WP2RS2[simp]*:

$\llbracket \text{singleCombinators } p; \\ \text{distinct } p; \\ \text{allNetsDistinct } p \rrbracket \implies \\ \text{wellformed-policy2Pr } (\text{removeShadowRules2 } p)$
 <proof>

lemma *AD-aux*: $\llbracket \text{AllowPortFromTo } a \ b \ po \in \text{set } p; \text{DenyAllFromTo } c \ d \in \text{set } p; \\ \text{allNetsDistinct } p; \text{singleCombinators } p; \\ a \neq c \vee b \neq d \rrbracket \\ \implies \text{dom } (Cp \ (\text{AllowPortFromTo } a \ b \ po)) \cap \text{dom } (Cp \ (\text{DenyAllFromTo } c \ d)) = \{\}$
 <proof>

lemma *sorted-WP2[rule-format]*: $\text{sorted } p \ l \longrightarrow \text{all-in-list } p \ l \longrightarrow \text{distinct } p \longrightarrow \\ \text{allNetsDistinct } p \longrightarrow \text{singleCombinators } p \longrightarrow \text{wellformed-policy2Pr } p$
 <proof>

lemma *wellformed2-sorted[simp]*: $\llbracket \text{all-in-list } p \ l; \text{distinct } p; \text{allNetsDistinct } p; \\ \text{singleCombinators } p \rrbracket \implies \text{wellformed-policy2Pr } (\text{sort } p \ l)$
 <proof>

lemma *wellformed2-sortedQ[simp]*: $\llbracket \text{all-in-list } p \ l; \text{distinct } p; \text{allNetsDistinct } p; \\ \text{singleCombinators } p \rrbracket \implies \text{wellformed-policy2Pr } (\text{qsort } p \ l)$
 <proof>

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

$\langle proof \rangle$

lemma *C-eq-RS1n*:

$$Cp(list2FWpolicy (removeShadowRules1-alternative p)) = Cp(list2FWpolicy p)$$

$\langle proof \rangle$

lemma *C-eq-RS1[simp]*: $p \neq [] \implies$

$$Cp(list2FWpolicy (removeShadowRules1 p)) = Cp(list2FWpolicy p)$$

$\langle proof \rangle$

lemma *EX-MR-aux[rule-format]*: $applied-rule-rev Cp x (DenyAll \# p) \neq Some DenyAll \longrightarrow$
 $(\exists y. applied-rule-rev Cp x p = Some y)$

$\langle proof \rangle$

lemma *EX-MR* : $\llbracket applied-rule-rev Cp x p \neq (Some DenyAll); p = DenyAll \# ps \rrbracket \implies$
 $(applied-rule-rev Cp x p = applied-rule-rev Cp x ps)$

$\langle proof \rangle$

lemma *mr-not-DA*:

$$\llbracket wellformed-policy1-strong s; applied-rule-rev Cp x p = Some (DenyAllFromTo a ab);$$
$$set p = set s \rrbracket \implies applied-rule-rev Cp x s \neq Some DenyAll$$

$\langle proof \rangle$

lemma *domsMT-notND-DD*:

$$\llbracket dom (Cp (DenyAllFromTo a b)) \cap dom (Cp (DenyAllFromTo c d)) \neq \{\} \rrbracket \implies$$
$$\neg netsDistinct a c$$

$\langle proof \rangle$

lemma *domsMT-notND-DD2*:

$$\llbracket dom (Cp (DenyAllFromTo a b)) \cap dom (Cp (DenyAllFromTo c d)) \neq \{\} \rrbracket \implies$$
$$\neg netsDistinct b d$$

$\langle proof \rangle$

lemma *domsMT-notND-DD3*:

$$\llbracket x \in dom (Cp (DenyAllFromTo a b)); x \in dom (Cp (DenyAllFromTo c d)) \rrbracket \implies$$
$$\neg netsDistinct a c$$

$\langle proof \rangle$

lemma *domsMT-notND-DD4*:

$$\llbracket x \in dom (Cp (DenyAllFromTo a b)); x \in dom (Cp (DenyAllFromTo c d)) \rrbracket \implies$$
$$\neg netsDistinct b d$$

$\langle proof \rangle$

lemma *NetsEq-if-sameP-DD*:

[[*allNetsDistinct* p ; $u \in \text{set } p$; $v \in \text{set } p$; $u = (\text{DenyAllFromTo } a \ b)$;
 $v = (\text{DenyAllFromTo } c \ d)$; $x \in \text{dom } (\text{Cp } (u))$; $x \in \text{dom } (\text{Cp } (v))$]] \implies
 $a = c \wedge b = d$
 <proof>

lemma *rule-charn1*:

assumes *aND*: *allNetsDistinct* p
and *mr-is-allow*: *applied-rule-rev* $\text{Cp } x \ p = \text{Some } (\text{AllowPortFromTo } a \ b \ po)$
and *SC*: *singleCombinators* p
and *inp*: $r \in \text{set } p$
and *inDom*: $x \in \text{dom } (\text{Cp } r)$
shows $(r = \text{AllowPortFromTo } a \ b \ po \vee r = \text{DenyAllFromTo } a \ b \vee r = \text{DenyAll})$
 <proof>

lemma *noneMTsubset[rule-format]*: *noneMT* $\text{Cp } a \longrightarrow \text{set } b \subseteq \text{set } a \longrightarrow \text{noneMT } \text{Cp } b$
 <proof>

lemma *nMTSort*: *noneMT* $\text{Cp } p \implies \text{noneMT } \text{Cp } (\text{sort } p \ l)$
 <proof>

lemma *nMTSortQ*: *noneMT* $\text{Cp } p \implies \text{noneMT } \text{Cp } (q\text{sort } p \ l)$
 <proof>

lemma *wp3char[rule-format]*: *noneMT* $\text{Cp } xs \wedge \text{Cp } (\text{AllowPortFromTo } a \ b \ po) = \text{empty} \wedge$
wellformed-policy3Pr $(xs @ [\text{DenyAllFromTo } a \ b]) \longrightarrow$
 $\text{AllowPortFromTo } a \ b \ po \notin \text{set } xs$
 <proof>

lemma *wp3charn[rule-format]*:

assumes *domAllow*: $\text{dom } (\text{Cp } (\text{AllowPortFromTo } a \ b \ po)) \neq \{\}$
and *wp3*: *wellformed-policy3Pr* $(xs @ [\text{DenyAllFromTo } a \ b])$
shows *allowNotInList*: $\text{AllowPortFromTo } a \ b \ po \notin \text{set } xs$
 <proof>

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 ∈ set p*
and *x-in-dom-allow*: *x ∈ dom (Cp (AllowPortFromTo c d po))*
shows *applied-rule-rev Cp x p = Some (AllowPortFromTo c d po)*
 ⟨*proof*⟩

lemma *rule-charn3*:
 [[*wellformed-policy1 p*; *allNetsDistinct p*; *singleCombinators p*;
wellformed-policy3Pr p; *applied-rule-rev Cp x p = Some (DenyAllFromTo c d)*;
AllowPortFromTo a b po ∈ set p]] $\implies x \notin \text{dom } (Cp (AllowPortFromTo a b po))$
 ⟨*proof*⟩

lemma *rule-charn4*:
assumes *wp1*: *wellformed-policy1 p*
and *aND*: *allNetsDistinct p*
and *SC*: *singleCombinators p*
and *wp3*: *wellformed-policy3Pr p*
and *DA*: *DenyAll ∉ set p*
and *mr*: *applied-rule-rev Cp x p = Some (DenyAllFromTo a b)*
and *rinp*: *r ∈ set p*
and *xindom*: *x ∈ dom (Cp r)*
shows *r = DenyAllFromTo a b*
 ⟨*proof*⟩

lemma *foo31a*: [[$(\forall r. r \in \text{set } p \wedge x \in \text{dom } (Cp r) \longrightarrow$
 $(r = \text{AllowPortFromTo } a b po \vee r = \text{DenyAllFromTo } a b \vee r = \text{DenyAll}))$;
 $\text{set } p = \text{set } s$; $r \in \text{set } s$; $x \in \text{dom } (Cp r)$]] \implies
 $(r = \text{AllowPortFromTo } a b po \vee r = \text{DenyAllFromTo } a b \vee r = \text{DenyAll})$
 ⟨*proof*⟩

lemma *aux4*[*rule-format*]:
applied-rule-rev Cp x (a#p) = Some a $\longrightarrow a \notin \text{set } (p) \longrightarrow \text{applied-rule-rev Cp x p} = \text{None}$
 ⟨*proof*⟩

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*
 ⟨*proof*⟩

lemma *rule-charnDAFT*:
 [[*wellformed-policy1-strong p*; *allNetsDistinct p*; *singleCombinators p*;
wellformed-policy3Pr p; *applied-rule-rev Cp x p = Some (DenyAllFromTo a b)*;
 $r \in \text{set } (tl p)$; $x \in \text{dom } (Cp r)$]] $\implies r = \text{DenyAllFromTo } a b$
 ⟨*proof*⟩

lemma *mrDenyAll-is-unique*:

[[*wellformed-policy1-strong* p ; *applied-rule-rev* $Cp\ x\ p = \text{Some } \text{DenyAll}$;
 $r \in \text{set } (tl\ p)$]] $\implies x \notin \text{dom } (Cp\ r)$
 <proof>

theorem *C-eq-Sets-mr*:

assumes *sets-eq*: $\text{set } p = \text{set } s$
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* $Cp\ x\ p = \text{applied-rule-rev } Cp\ x\ s$
 <proof>

lemma *C-eq-Sets*:

[[*singleCombinators* p ; *wellformed-policy1-strong* p ; *wellformed-policy1-strong* s ;
wellformed-policy3Pr p ; *wellformed-policy3Pr* s ; *allNetsDistinct* p ; $\text{set } p = \text{set } s$]] \implies
 $Cp\ (\text{list2FWpolicy } p)\ x = Cp\ (\text{list2FWpolicy } s)\ x$
 <proof>

lemma *C-eq-sorted*: [[*distinct* p ; *all-in-list* $p\ l$; *singleCombinators* p ;

wellformed-policy1-strong p ; *wellformed-policy3Pr* p ; *allNetsDistinct* p]] \implies

$Cp\ (\text{list2FWpolicy } (\text{sort } p\ l)) = Cp\ (\text{list2FWpolicy } p)$

<proof>

lemma *C-eq-sortedQ*: [[*distinct* p ; *all-in-list* $p\ l$; *singleCombinators* p ;

wellformed-policy1-strong p ; *wellformed-policy3Pr* p ; *allNetsDistinct* p]] \implies

$Cp\ (\text{list2FWpolicy } (qsort\ p\ l)) = Cp\ (\text{list2FWpolicy } p)$

<proof>

lemma *C-eq-RS2-mr*: *applied-rule-rev* $Cp\ x\ (\text{removeShadowRules2 } p) = \text{applied-rule-rev } Cp\ x\ p$

<proof>

lemma *C-eq-None[rule-format]*: $p \neq [] \implies \text{applied-rule-rev } Cp\ x\ p = \text{None} \implies$

$Cp\ (\text{list2FWpolicy } p)\ x = \text{None}$

<proof>

lemma *C-eq-None2*:

$\llbracket a \neq []; b \neq []; \text{applied-rule-rev } Cp \ x \ a = \text{None}; \text{applied-rule-rev } Cp \ x \ b = \text{None} \rrbracket \implies$
 $(Cp \ (\text{list2FWpolicy } a)) \ x = (Cp \ (\text{list2FWpolicy } b)) \ x$
 $\langle \text{proof} \rangle$

lemma *C-eq-RS2*: *wellformed-policy1-strong* $p \implies$

$Cp \ (\text{list2FWpolicy } (\text{removeShadowRules2 } p)) = Cp \ (\text{list2FWpolicy } p)$
 $\langle \text{proof} \rangle$

lemma *noneMTRS2*: *noneMT* $Cp \ p \implies \text{noneMT } Cp \ (\text{removeShadowRules2 } p)$

$\langle \text{proof} \rangle$

lemma *CconcNone*: $\llbracket \text{dom } (Cp \ a) = \{\}; p \neq [] \rrbracket \implies$

$Cp \ (\text{list2FWpolicy } (a \ \# \ p)) \ x = Cp \ (\text{list2FWpolicy } p) \ x$
 $\langle \text{proof} \rangle$

lemma *noneMTrd[rule-format]*: *noneMT* $Cp \ p \longrightarrow \text{noneMT } Cp \ (\text{remdups } p)$

$\langle \text{proof} \rangle$

lemma *DARS3[rule-format]*: *DenyAll* $\notin \text{set } p \longrightarrow \text{DenyAll} \notin \text{set } (\text{removeShadowRules3 } Cp \ p)$

$\langle \text{proof} \rangle$

lemma *DAnMT*: *dom* $(Cp \ \text{DenyAll}) \neq \{\}$

$\langle \text{proof} \rangle$

lemma *DAnMT2*: $Cp \ \text{DenyAll} \neq \text{empty}$

$\langle \text{proof} \rangle$

lemma *wp1n-RS3[rule-format,simp]*: *wellformed-policy1-strong* $p \longrightarrow$

wellformed-policy1-strong $(\text{removeShadowRules3 } Cp \ p)$
 $\langle \text{proof} \rangle$

lemma *AILRS3[rule-format,simp]*: *all-in-list* $p \ l \longrightarrow$

all-in-list $(\text{removeShadowRules3 } Cp \ p) \ l$
 $\langle \text{proof} \rangle$

lemma *SCRS3[rule-format,simp]*: *singleCombinators* $p \longrightarrow$

singleCombinators $(\text{removeShadowRules3 } Cp \ p)$
 $\langle \text{proof} \rangle$

lemma *RS3subset*: *set* $(\text{removeShadowRules3 } Cp \ p) \subseteq \text{set } p$

$\langle \text{proof} \rangle$

lemma *ANDRS3[simp]*: $\llbracket \text{singleCombinators } p; \text{allNetsDistinct } p \rrbracket \implies$
 $\text{allNetsDistinct } (\text{removeShadowRules3 } Cp \ p)$
 $\langle \text{proof} \rangle$

lemma *nlpaux*: $x \notin \text{dom } (Cp \ b) \implies Cp \ (a \oplus b) \ x = Cp \ a \ x$
 $\langle \text{proof} \rangle$

lemma *notindom[rule-format]*: $a \in \text{set } p \longrightarrow x \notin \text{dom } (Cp \ (\text{list2FWpolicy } p)) \longrightarrow$
 $x \notin \text{dom } (Cp \ a)$
 $\langle \text{proof} \rangle$

lemma *C-eq-rd[rule-format]*: $p \neq [] \implies$
 $Cp \ (\text{list2FWpolicy } (\text{remdups } p)) = Cp \ (\text{list2FWpolicy } p)$
 $\langle \text{proof} \rangle$

lemma *nMT-domMT*: $\llbracket \neg \text{notMTpolicy } Cp \ p; p \neq [] \rrbracket \implies r \notin \text{dom } (Cp \ (\text{list2FWpolicy } p))$
 $\langle \text{proof} \rangle$

lemma *C-eq-RS3-aux[rule-format]*: $\text{notMTpolicy } Cp \ p \implies$
 $Cp \ (\text{list2FWpolicy } p) \ x = Cp \ (\text{list2FWpolicy } (\text{removeShadowRules3 } Cp \ p)) \ x$
 $\langle \text{proof} \rangle$

lemma *C-eq-id: wellformed-policy1-strong* $p \implies$
 $Cp(\text{list2FWpolicy } (\text{insertDeny } p)) = Cp \ (\text{list2FWpolicy } p)$
 $\langle \text{proof} \rangle$

lemma *C-eq-RS3*: $\text{notMTpolicy } Cp \ p \implies$
 $Cp(\text{list2FWpolicy } (\text{removeShadowRules3 } Cp \ p)) = Cp \ (\text{list2FWpolicy } p)$
 $\langle \text{proof} \rangle$

lemma *NMPrd[rule-format]*: $\text{notMTpolicy } Cp \ p \longrightarrow \text{notMTpolicy } Cp \ (\text{remdups } p)$
 $\langle \text{proof} \rangle$

lemma *NMPDA[rule-format]*: $\text{DenyAll} \in \text{set } p \longrightarrow \text{notMTpolicy } Cp \ p$
 $\langle \text{proof} \rangle$

lemma *NMPiD[rule-format]*: $\text{notMTpolicy } Cp \ (\text{insertDeny } p)$
 $\langle \text{proof} \rangle$

lemma *list2FWpolicy2list*[rule-format]: $Cp\ (list2FWpolicy(policy2list\ p)) = (Cp\ p)$
 <proof>

lemmas *C-eq-Lemmas* = noneMTRS2 noneMTrd SCp2l
 wp1n-RS2 wp1ID NMPiD wp1alternative-RS1
 p2lNmt list2FWpolicy2list wellformed-policy3-chaen 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*:

$\llbracket DenyAll \in set\ (policy2list\ p); all-in-list\ (policy2list\ p)\ l;$
 $allNetsDistinct\ (policy2list\ p) \rrbracket \implies$
 $Cp(list2FWpolicy\ (sort\ (removeShadowRules2\ (remdups\ (removeShadowRules3\ Cp$
 $(insertDeny\ (removeShadowRules1\ (policy2list\ p))))))\ l)) = Cp\ p$
 <proof>

lemma *C-eq-All-untilSortedQ*:

$\llbracket DenyAll \in set\ (policy2list\ p); all-in-list\ (policy2list\ p)\ l;$
 $allNetsDistinct\ (policy2list\ p) \rrbracket \implies$
 $Cp(list2FWpolicy\ (qsort\ (removeShadowRules2\ (remdups\ (removeShadowRules3\ Cp$
 $(insertDeny\ (removeShadowRules1\ (policy2list\ p))))))\ l)) = Cp\ p$
 <proof>

lemma *C-eq-All-untilSorted-withSimps*:

$\llbracket DenyAll \in set\ (policy2list\ p); all-in-list\ (policy2list\ p)\ l;$
 $allNetsDistinct\ (policy2list\ p) \rrbracket \implies$
 $Cp(list2FWpolicy\ (sort\ (removeShadowRules2\ (remdups\ (removeShadowRules3\ Cp\ (insertDeny$
 $(removeShadowRules1\ (policy2list\ p))))))\ l)) = Cp\ p$
 <proof>

lemma *C-eq-All-untilSorted-withSimpsQ*:

$\llbracket DenyAll \in set\ (policy2list\ p); all-in-list\ (policy2list\ p)\ l;$
 $allNetsDistinct\ (policy2list\ p) \rrbracket \implies$
 $Cp(list2FWpolicy\ (qsort\ (removeShadowRules2\ (remdups\ (removeShadowRules3\ Cp\ (insertDeny$
 $(removeShadowRules1\ (policy2list\ p))))))\ l)) = Cp\ p$
 <proof>

lemma *InDomConc*[rule-format]: $p \neq [] \longrightarrow x \in \text{dom } (Cp \text{ (list2FWpolicy } (p))) \longrightarrow$
 $x \in \text{dom } (Cp \text{ (list2FWpolicy } (a\#p)))$
 ⟨proof⟩

lemma *not-in-member*[rule-format]: $\text{member } a \ b \longrightarrow x \notin \text{dom } (Cp \ b) \longrightarrow x \notin \text{dom } (Cp \ a)$
 ⟨proof⟩

lemma *src-in-sdnets*[rule-format]: $\neg \text{member } \text{DenyAll } x \longrightarrow p \in \text{dom } (Cp \ x) \longrightarrow$
 $\text{subnetsOfAdr } (\text{src } p) \cap (\text{fst-set } (\text{sdnets } x)) \neq \{\}$
 ⟨proof⟩

lemma *dest-in-sdnets*[rule-format]: $\neg \text{member } \text{DenyAll } x \longrightarrow p \in \text{dom } (Cp \ x) \longrightarrow$
 $\text{subnetsOfAdr } (\text{dest } p) \cap (\text{snd-set } (\text{sdnets } x)) \neq \{\}$
 ⟨proof⟩

lemma *sdnets-in-subnets*[rule-format]: $p \in \text{dom } (Cp \ x) \longrightarrow \neg \text{member } \text{DenyAll } x \longrightarrow$
 $(\exists (a,b) \in \text{sdnets } x. a \in \text{subnetsOfAdr } (\text{src } p) \wedge b \in \text{subnetsOfAdr } (\text{dest } p))$
 ⟨proof⟩

lemma *disjSD-no-p-in-both*[rule-format]:
 $\llbracket \text{disjSD-2 } x \ y; \neg \text{member } \text{DenyAll } x; \neg \text{member } \text{DenyAll } y;$
 $p \in \text{dom } (Cp \ x); p \in \text{dom } (Cp \ y) \rrbracket \Longrightarrow \text{False}$
 ⟨proof⟩

lemma *list2FWpolicy-eq*: $zs \neq [] \Longrightarrow$
 $Cp \text{ (list2FWpolicy } (x \oplus y \# z)) \ p = Cp \text{ (list2FWpolicy } (y \# z)) \ p$
 ⟨proof⟩

lemma *dom-sep*[rule-format]: $x \in \text{dom } (Cp \text{ (list2FWpolicy } p)) \longrightarrow$
 $x \in \text{dom } (Cp \text{ (list2FWpolicy } (\text{separate } p)))$
 ⟨proof⟩

lemma *domdConcStart*[rule-format]: $x \in \text{dom } (Cp \text{ (list2FWpolicy } (a\#b))) \longrightarrow$
 $x \notin \text{dom } (Cp \text{ (list2FWpolicy } b))$
 $\longrightarrow x \in \text{dom } (Cp \ a)$
 ⟨proof⟩

lemma *sep-dom2-aux*: $\llbracket x \in \text{dom } (Cp \text{ (list2FWpolicy } (a \oplus y \# z))) \rrbracket$
 $\Longrightarrow x \in \text{dom } (Cp \ (a \oplus \text{list2FWpolicy } (y \# z)))$
 ⟨proof⟩

lemma *sep-dom2-aux2*:

$\llbracket (x \in \text{dom } (\text{Cp } (\text{list2FWpolicy } (\text{separate } (y \# z)))) \longrightarrow$
 $x \in \text{dom } (\text{Cp } (\text{list2FWpolicy } (y \# z)))$;
 $x \in \text{dom } (\text{Cp } (\text{list2FWpolicy } (a \# \text{separate } (y \# z)))) \rrbracket$
 $\implies x \in \text{dom } (\text{Cp } (\text{list2FWpolicy } (a \oplus y \# z)))$
 $\langle \text{proof} \rangle$

lemma *sep-dom2[rule-format]*:

$x \in \text{dom } (\text{Cp } (\text{list2FWpolicy } (\text{separate } p))) \longrightarrow x \in \text{dom } (\text{Cp } (\text{list2FWpolicy } (p)))$
 $\langle \text{proof} \rangle$

lemma *sepDom*: $\text{dom } (\text{Cp } (\text{list2FWpolicy } p)) = \text{dom } (\text{Cp } (\text{list2FWpolicy } (\text{separate } p)))$

$\langle \text{proof} \rangle$

lemma *C-eq-s-ext[rule-format]*: $p \neq [] \longrightarrow$

$\text{Cp } (\text{list2FWpolicy } (\text{separate } p)) \ a = \text{Cp } (\text{list2FWpolicy } p) \ a$

$\langle \text{proof} \rangle$

lemma *C-eq-s*: $p \neq [] \implies \text{Cp } (\text{list2FWpolicy } (\text{separate } p)) = \text{Cp } (\text{list2FWpolicy } p)$

$\langle \text{proof} \rangle$

lemma *sortnMTQ*: $p \neq [] \implies \text{qsort } p \ l \neq []$

$\langle \text{proof} \rangle$

lemmas *C-eq-Lemmas-sep* =

C-eq-Lemmas sortnMT sortnMTQ RS2-NMT NMPrd NMPRS3

lemma *C-eq-until-separated*:

$\llbracket \text{DenyAll} \in \text{set } (\text{policy2list } p); \text{all-in-list } (\text{policy2list } p) \ l;$
 $\text{allNetsDistinct } (\text{policy2list } p) \rrbracket \implies$
 $\text{Cp } (\text{list2FWpolicy } (\text{separate } (\text{sort } (\text{removeShadowRules2 } (\text{remdups } (\text{removeShadowRules3 } \text{Cp } (\text{insertDeny } (\text{removeShadowRules1 } (\text{policy2list } p)))))) \ l))) = \text{Cp } p$
 $\langle \text{proof} \rangle$

lemma *C-eq-until-separatedQ*:

$\llbracket \text{DenyAll} \in \text{set } (\text{policy2list } p); \text{all-in-list } (\text{policy2list } p) \ l;$
 $\text{allNetsDistinct } (\text{policy2list } p) \rrbracket \implies$
 $\text{Cp } (\text{list2FWpolicy } (\text{separate } (\text{qsort } (\text{removeShadowRules2 } (\text{remdups } (\text{removeShadowRules3 } \text{Cp } (\text{insertDeny } (\text{removeShadowRules1 } (\text{policy2list } p)))))) \ l))) = \text{Cp } p$
 $\langle \text{proof} \rangle$

lemma *domID[rule-format]*: $p \neq [] \wedge x \in \text{dom } (\text{Cp } (\text{list2FWpolicy } p)) \longrightarrow$

$x \in \text{dom } (\text{Cp } (\text{list2FWpolicy } (\text{insertDenies } p)))$

$\langle \text{proof} \rangle$

lemma *DA-is-deny*:

$x \in \text{dom } (\text{Cp } (\text{DenyAllFromTo } a \ b \oplus \text{DenyAllFromTo } b \ a \oplus \text{DenyAllFromTo } a \ b)) \implies$
 $\text{Cp } (\text{DenyAllFromTo } a \ b \oplus \text{DenyAllFromTo } b \ a \oplus \text{DenyAllFromTo } a \ b) \ x = \text{Some } (\text{deny } ())$
 $\langle \text{proof} \rangle$

lemma *iDdomAux*[rule-format]:

$p \neq [] \longrightarrow x \notin \text{dom } (\text{Cp } (\text{list2FWpolicy } p)) \longrightarrow$
 $x \in \text{dom } (\text{Cp } (\text{list2FWpolicy } (\text{insertDenies } p))) \longrightarrow$
 $\text{Cp } (\text{list2FWpolicy } (\text{insertDenies } p)) \ x = \text{Some } (\text{deny } ())$
 $\langle \text{proof} \rangle$

lemma *iD-isD*[rule-format]: $p \neq [] \longrightarrow x \notin \text{dom } (\text{Cp } (\text{list2FWpolicy } p))$

$\longrightarrow \text{Cp } (\text{DenyAll} \oplus \text{list2FWpolicy } (\text{insertDenies } p)) \ x = \text{Cp } \text{DenyAll } x$
 $\langle \text{proof} \rangle$

lemma *inDomConc*: $\llbracket x \notin \text{dom } (\text{Cp } a); x \notin \text{dom } (\text{Cp } (\text{list2FWpolicy } p)) \rrbracket \implies$

$x \notin \text{dom } (\text{Cp } (\text{list2FWpolicy } (a \# p)))$
 $\langle \text{proof} \rangle$

lemma *domsdisj*[rule-format]: $p \neq [] \longrightarrow (\forall \ x \ s. \ s \in \text{set } p \wedge x \in \text{dom } (\text{Cp } A) \longrightarrow$

$x \notin \text{dom } (\text{Cp } s)) \longrightarrow y \in \text{dom } (\text{Cp } A) \longrightarrow$
 $y \notin \text{dom } (\text{Cp } (\text{list2FWpolicy } p))$
 $\langle \text{proof} \rangle$

lemma *isSepaux*:

$\llbracket p \neq []; \text{noDenyAll } (a \# p); \text{separated } (a \ \# \ p);$
 $x \in \text{dom } (\text{Cp } (\text{DenyAllFromTo } (\text{first-srcNet } a) \ (\text{first-destNet } a) \oplus$
 $\text{DenyAllFromTo } (\text{first-destNet } a) \ (\text{first-srcNet } a) \oplus a)) \rrbracket \implies$
 $x \notin \text{dom } (\text{Cp } (\text{list2FWpolicy } p))$
 $\langle \text{proof} \rangle$

lemma *noneMTsep*[rule-format]: $\text{noneMT } \text{Cp } p \longrightarrow \text{noneMT } \text{Cp } (\text{separate } p)$

$\langle \text{proof} \rangle$

lemma *dom-id*:

$\llbracket \text{noDenyAll } (a \# p); \text{separated } (a \# p); p \neq []; x \notin \text{dom } (\text{Cp } (\text{list2FWpolicy } p));$
 $x \in \text{dom } (\text{Cp } (a)) \rrbracket$
 $\implies x \notin \text{dom } (\text{Cp } (\text{list2FWpolicy } (\text{insertDenies } p)))$
 $\langle \text{proof} \rangle$

lemma *C-eq-iD-aux2*[rule-format]:

$noDenyAll1\ p \longrightarrow$
 $separated\ p \longrightarrow$
 $p \neq [] \longrightarrow$
 $x \in dom\ (Cp\ (list2FWpolicy\ p)) \longrightarrow$
 $Cp(list2FWpolicy\ (insertDenies\ p))\ x = Cp(list2FWpolicy\ p)\ x$
 $\langle proof \rangle$

lemma *C-eq-iD*: $\llbracket separated\ p; noDenyAll1\ p; wellformed-policy1-strong\ p \rrbracket \implies$
 $Cp\ (list2FWpolicy\ (insertDenies\ p)) = Cp\ (list2FWpolicy\ p)$
 $\langle proof \rangle$

lemma *noDAsortQ[rule-format]*: $noDenyAll1\ p \longrightarrow noDenyAll1\ (qsort\ p\ l)$
 $\langle proof \rangle$

lemma *NetsCollectedSortQ*: $distinct\ p \implies noDenyAll1\ p \implies all-in-list\ p\ l \implies$
 $singleCombinators\ p \implies NetsCollected\ (qsort\ p\ l)$
 $\langle proof \rangle$

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*: $\llbracket DenyAll \in set\ (policy2list\ p); all-in-list$
 $(policy2list\ p)\ l; allNetsDistinct\ (policy2list\ p) \rrbracket \implies$
 $Cp\ (list2FWpolicy\ ((insertDenies\ (separate\ (sort\ (removeShadowRules2\ (remdups$
 $(removeShadowRules3\ Cp\ (insertDeny\ (removeShadowRules1\ (policy2list\ p))))))\ l)))))) =$
 $Cp\ p$
 $\langle proof \rangle$

lemma *C-eq-Until-InsertDeniesQ*: $\llbracket DenyAll \in set\ (policy2list\ p); all-in-list$
 $(policy2list\ p)\ l; allNetsDistinct\ (policy2list\ p) \rrbracket \implies$
 $Cp\ (list2FWpolicy\ ((insertDenies\ (separate\ (qsort\ (removeShadowRules2\ (remdups$
 $(removeShadowRules3\ Cp\ (insertDeny\ (removeShadowRules1\ (policy2list\ p))))))\ l)))))) =$
 $Cp\ p$
 $\langle proof \rangle$

lemma *C-eq-RD-aux*[rule-format]: $Cp\ (p)\ x = Cp\ (removeDuplicates\ p)\ x$
 <proof>

lemma *C-eq-RAD-aux*[rule-format]:
 $p \neq [] \longrightarrow Cp\ (list2FWpolicy\ p)\ x = Cp\ (list2FWpolicy\ (removeAllDuplicates\ p))\ x$
 <proof>

lemma *C-eq-RAD*:
 $p \neq [] \implies Cp\ (list2FWpolicy\ p) = Cp\ (list2FWpolicy\ (removeAllDuplicates\ p))$
 <proof>

lemma *C-eq-compile*:
 $\llbracket DenyAll \in set\ (policy2list\ p); all-in-list\ (policy2list\ p)\ l;$
 $allNetsDistinct\ (policy2list\ p) \rrbracket \implies$
 $Cp\ (list2FWpolicy\ (removeAllDuplicates\ (insertDenies\ (separate\ (sort$
 $(removeShadowRules2\ (remdups\ (removeShadowRules3\ Cp\ (insertDeny$
 $(removeShadowRules1\ (policy2list\ p))))))\ l)))) = Cp\ p$
 <proof>

lemma *C-eq-compileQ*:
 $\llbracket DenyAll \in set\ (policy2list\ p); all-in-list\ (policy2list\ p)\ l;$
 $allNetsDistinct\ (policy2list\ p) \rrbracket \implies$
 $Cp\ (list2FWpolicy\ (removeAllDuplicates\ (insertDenies\ (separate\ (qsort$
 $(removeShadowRules2\ (remdups\ (removeShadowRules3\ Cp\ (insertDeny$
 $(removeShadowRules1\ (policy2list\ p))))))\ l)))) = Cp\ p$
 <proof>

lemma *C-eq-normalizePr*:
 $\llbracket DenyAll \in set\ (policy2list\ p);$
 $allNetsDistinct\ (policy2list\ p);$
 $all-in-list\ (policy2list\ p)\ (Nets-List\ p) \rrbracket \implies$
 $Cp\ (list2FWpolicy\ (normalizePr\ p)) = Cp\ p$
 <proof>

lemma *C-eq-normalizePrQ*:
 $\llbracket DenyAll \in set\ (policy2list\ p);$
 $allNetsDistinct\ (policy2list\ p);$
 $all-in-list\ (policy2list\ p)\ (Nets-List\ p) \rrbracket \implies$
 $Cp\ (list2FWpolicy\ (normalizePrQ\ p)) = Cp\ p$
 <proof>

lemma *domSubset3*: $\text{dom } (Cp \ (DenyAll \oplus x)) = \text{dom } (Cp \ (DenyAll))$
 <proof>

lemma *domSubset4*: $\text{dom } (Cp \ (DenyAllFromTo \ x \ y \oplus DenyAllFromTo \ y \ x \oplus AllowPortFromTo \ x \ y \ dn)) =$
 $\text{dom } (Cp \ (DenyAllFromTo \ x \ y \oplus DenyAllFromTo \ y \ x))$
 <proof>

lemma *domSubset5*:
 $\text{dom } (Cp \ (DenyAllFromTo \ x \ y \oplus DenyAllFromTo \ y \ x \oplus AllowPortFromTo \ y \ x \ dn)) =$
 $\text{dom } (Cp \ (DenyAllFromTo \ x \ y \oplus DenyAllFromTo \ y \ x))$
 <proof>

lemma *domSubset1*: $\text{dom } (Cp \ (DenyAllFromTo \ one \ two \oplus DenyAllFromTo \ two \ one \oplus AllowPortFromTo \ one \ two \ dn \oplus x)) =$
 $\text{dom } (Cp \ (DenyAllFromTo \ one \ two \oplus DenyAllFromTo \ two \ one \oplus x))$
 <proof>

lemma *domSubset2*:
 $\text{dom } (Cp \ (DenyAllFromTo \ one \ two \oplus DenyAllFromTo \ two \ one \oplus AllowPortFromTo \ two \ one \ dn \oplus x)) =$
 $\text{dom } (Cp \ (DenyAllFromTo \ one \ two \oplus DenyAllFromTo \ two \ one \oplus x))$
 <proof>

lemma *ConcAssoc2*: $Cp \ (X \oplus Y \oplus ((A \oplus B) \oplus D)) = Cp \ (X \oplus Y \oplus A \oplus B \oplus D)$
 <proof>

lemma *ConcAssoc3*: $Cp \ (X \oplus ((Y \oplus A) \oplus D)) = Cp \ (X \oplus Y \oplus A \oplus D)$
 <proof>

lemma *RS3-NMT[rule-format]*: $DenyAll \in \text{set } p \longrightarrow$
 $\text{removeShadowRules3 } Cp \ p \neq []$
 <proof>

lemma *norm-notMT*: $DenyAll \in \text{set } (\text{policy2list } p) \implies \text{normalizePr } p \neq []$
 <proof>

lemma *norm-notMTQ*: $\text{DenyAll} \in \text{set } (\text{policy2list } p) \implies \text{normalizePrQ } p \neq []$
 <proof>

lemma *domDA*: $\text{dom } (\text{Cp } (\text{DenyAll} \oplus A)) = \text{dom } (\text{Cp } (\text{DenyAll}))$
 <proof>

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]: $p \in \text{dom } (\text{Cp } a) \longrightarrow$
 $\text{Cp } (\text{list2policyR } (a \# \text{list})) \text{ } p = \text{Cp } a \text{ } p$
 <proof>

lemma *list2policyR-End*: $p \notin \text{dom } (\text{Cp } a) \implies$
 $\text{Cp } (\text{list2policyR } (a \# \text{list})) \text{ } p = (\text{Cp } a \oplus \text{list2policy } (\text{map } \text{Cp } \text{list})) \text{ } p$
 <proof>

lemma *l2polR-eq-el*[rule-format]: $N \neq [] \longrightarrow$
 $\text{Cp } (\text{list2policyR } N) \text{ } p = (\text{list2policy } (\text{map } \text{Cp } N)) \text{ } p$
 <proof>

lemma *l2polR-eq*: $N \neq [] \implies$
 $\text{Cp } (\text{list2policyR } N) = (\text{list2policy } (\text{map } \text{Cp } N))$
 <proof>

lemma *list2FWpolicys-eq-el*[rule-format]:
 $\text{Filter} \neq [] \longrightarrow \text{Cp } (\text{list2policyR } \text{Filter}) \text{ } p = \text{Cp } (\text{list2FWpolicy } (\text{rev } \text{Filter})) \text{ } p$
 <proof>

lemma *list2FWpolicys-eq*:
 $\text{Filter} \neq [] \implies$
 $\text{Cp } (\text{list2policyR } \text{Filter}) = \text{Cp } (\text{list2FWpolicy } (\text{rev } \text{Filter}))$
 <proof>

lemma *list2FWpolicys-eq-sym*:
 $\text{Filter} \neq [] \implies$

$Cp \ (list2policyR \ (rev \ Filter)) = Cp \ (list2FWpolicy \ Filter)$
 $\langle proof \rangle$

lemma $p\text{-eq}[rule\text{-format}]$: $p \neq [] \longrightarrow$
 $list2policy \ (map \ Cp \ (rev \ p)) = Cp \ (list2FWpolicy \ p)$
 $\langle proof \rangle$

lemma $p\text{-eq2}[rule\text{-format}]$: $normalizePr \ x \neq [] \longrightarrow$
 $Cp \ (list2FWpolicy \ (normalizePr \ x)) = Cp \ x \longrightarrow$
 $list2policy \ (map \ Cp \ (rev \ (normalizePr \ x))) = Cp \ x$
 $\langle proof \rangle$

lemma $p\text{-eq2Q}[rule\text{-format}]$: $normalizePrQ \ x \neq [] \longrightarrow$
 $Cp \ (list2FWpolicy \ (normalizePrQ \ x)) = Cp \ x \longrightarrow$
 $list2policy \ (map \ Cp \ (rev \ (normalizePrQ \ x))) = Cp \ x$
 $\langle proof \rangle$

lemma $list2listNMT[rule\text{-format}]$: $x \neq [] \longrightarrow map \ sem \ x \neq []$
 $\langle proof \rangle$

lemma $Norm\text{-}Distr2$:
 $r \ o\text{-}f \ ((P \otimes_2 \ (list2policy \ Q)) \ o \ d) =$
 $(list2policy \ ((P \otimes_L \ Q) \ (op \otimes_2 \ r \ d)))$
 $\langle proof \rangle$

lemma $NATDistr$: $\llbracket N \neq [] ; F = Cp \ (list2policyR \ N) \rrbracket \implies$
 $((\lambda \ (x,y). \ x) \ o\text{-}f \ ((NAT \otimes_2 \ F) \ o \ (\lambda \ x. \ (x,x))) =$
 $(list2policy \ ((NAT \otimes_L \ (map \ Cp \ N)) \ (op \otimes_2)$
 $(\lambda \ (x,y). \ x) \ (\lambda \ x. \ (x,x)))))$
 $\langle proof \rangle$

lemma $C\text{-}eq\text{-}normalize\text{-}manual$:
 $\llbracket DenyAll \in set \ (policy2list \ p);$
 $allNetsDistinct \ (policy2list \ p);$
 $all\text{-}in\text{-}list \ (policy2list \ p) \ l \rrbracket \implies$
 $Cp \ (list2FWpolicy \ (normalize\text{-}manual\text{-}orderPr \ p \ l)) = Cp \ p$
 $\langle proof \rangle$

lemma *p-eq2-manualQ[rule-format]: normalize-manual-orderPrQ x l ≠ [] →*
Cp (list2FWpolicy (normalize-manual-orderPrQ x l)) = Cp x →
list2policy (map Cp (rev (normalize-manual-orderPrQ x l))) = Cp x
 ⟨proof⟩

lemma *norm-notMT-manualQ: DenyAll ∈ set (policy2list p) ⇒ normalize-manual-orderPrQ*
p l ≠ []
 ⟨proof⟩

lemma *C-eq-normalizePr-manualQ:*
 $\llbracket \text{DenyAll} \in \text{set } (\text{policy2list } p);$
 $\text{allNetsDistinct } (\text{policy2list } p);$
 $\text{all-in-list } (\text{policy2list } p) \text{ l} \rrbracket \Rightarrow$
 $\text{Cp } (\text{list2FWpolicy } (\text{normalize-manual-orderPrQ } p \text{ l})) = \text{Cp } p$
 ⟨proof⟩

lemma *p-eq2-manual[rule-format]: normalize-manual-orderPr x l ≠ [] →*
Cp (list2FWpolicy (normalize-manual-orderPr x l)) = Cp x →
list2policy (map Cp (rev (normalize-manual-orderPr x l))) = Cp x
 ⟨proof⟩

lemma *norm-notMT-manual: DenyAll ∈ set (policy2list p) ⇒ normalize-manual-orderPr p l*
≠ []
 ⟨proof⟩

As an example, how this theorems can be used for a concrete normalisation instantiation.

lemma *normalizePrNAT: $\llbracket \text{DenyAll} \in \text{set } (\text{policy2list } \text{Filter});$*
allNetsDistinct (policy2list Filter);
all-in-list (policy2list Filter) (Nets-List Filter) $\rrbracket \Rightarrow$
 $((\lambda (x,y). x) \circ \text{f } (((\text{NAT} \otimes_2 \text{Cp } \text{Filter}) \circ (\lambda x. (x,x)))))) =$
 $\text{list2policy } ((\text{NAT} \otimes_L (\text{map } \text{Cp } (\text{rev } (\text{normalizePr } \text{Filter})))) (\text{op } \otimes_2) (\lambda (x,y). x) (\lambda x.$
 $(x,x)))$
 ⟨proof⟩

lemma *dom.Simpl[simp]: dom (Cp (A ⊕ DenyAll)) = dom (Cp (DenyAll))*
 ⟨proof⟩

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.