

Formal Network Models and Their Application to Firewall Policies (UPF-Firewall)

Achim D. Brucker* Lukas Brügger† Burkhardt Wolff‡

*Department of Computer Science, The University of Sheffield, Sheffield, UK
a.brucker@sheffield.ac.uk

Information Security, ETH Zurich, 8092 Zurich, Switzerland
Lukas.A.Bruegger@gmail.com

‡Univ. Paris-Sud, Laboratoire LRI, UMR8623, 91405 Orsay, France France
burkhart.wolff@lri.fr

March 17, 2025

Abstract

We present a formal model of network protocols and their application to modeling firewall policies. The formalization is based on the *Unified Policy Framework* (UPF). The formalization was originally developed with for generating test cases for testing the security configuration actual firewall and router (middle-boxes) using HOL-TestGen. Our work focuses on modeling application level protocols on top of tcp/ip.

Contents

1	Introduction	5
1.1	Motivation	5
1.2	The Unified Policy Framework (UPF)	5
2	UPF Firewall	9
2.1	Network Models	9
2.1.1	Packets and Networks	10
2.1.2	Datatype Addresses	13
2.1.3	Datatype Addresses with Ports	13
2.1.4	Integer Addresses	14
2.1.5	Integer Addresses with Ports	15
2.1.6	Integer Addresses with Ports and Protocols	16
2.1.7	Formalizing IPv4 Addresses	17
2.1.8	IPv4 with Ports and Protocols	19
2.2	Network Policies: Packet Filter	20
2.2.1	Policy Core	20
2.2.2	Policy Combinators	21
2.2.3	Policy Combinators with Ports	22
2.2.4	Policy Combinators with Ports and Protocols	25
2.2.5	Ports	28
2.2.6	Network Address Translation	29
2.3	Firewall Policy Normalisation	32
2.3.1	Policy Normalisation: Core Definitions	33
2.3.2	Normalisation Proofs (Generic)	45
2.3.3	Normalisation Proofs: Integer Port	99
2.3.4	Normalisation Proofs: Integer Protocol	143
2.4	Stateful Network Protocols	187
2.4.1	Stateful Protocols: Foundations	187
2.4.2	The File Transfer Protocol (ftp)	189
2.4.3	FTP enriched with a security policy	193
2.4.4	A simple voice-over-ip model	194
2.4.5	FTP and VoIP Protocol	199
3	Examples	207
3.1	A Simple DMZ Setup	207
3.1.1	DMZ Datatype	207
3.1.2	DMZ: Integer	209

3.2	Personal Firewall	211
3.2.1	Personal Firewall: Integer	211
3.2.2	Personal Firewall IPv4	212
3.2.3	Personal Firewall: Datatype	213
3.3	Demonstrating Policy Transformations	215
3.3.1	Transformation Example 1	215
3.3.2	Transforamtion Example 2	219
3.4	Example: NAT	222
3.5	Voice over IP	227

1 Introduction

1.1 Motivation

Because of its connected life, the modern world is increasingly depending on secure implementations and configurations of network infrastructures. As building blocks of the latter, firewalls are playing a central role in ensuring the overall *security* of networked applications.

Firewalls, routers applying network-address-translation (NAT) and similar networking systems suffer from the same quality problems as other complex software. Jennifer Rexford mentioned in her keynote at POPL 2012 that high-end firewalls consist of more than 20 million lines of code comprising components written in Ada as well as LISP. However, the testing techniques discussed here are of wider interest to all network infrastructure operators that need to ensure the security and reliability of their infrastructures across system changes such as system upgrades or hardware replacements. This is because firewalls and routers are active network elements that can filter and rewrite network traffic based on configurable rules. The *configuration* by appropriate rule sets implements a security policy or links networks together.

Thus, it is, firstly, important to test both the implementation of a firewall and, secondly, the correct configuration for each use. To address this problem, we model firewall policies formally in Isabelle/HOL. This formalization is based on the Unified Policy Framework (UPF) [6]. This formalization allows to express access control policies on the network level using a combinator-based language that is close to textbook-style specifications of firewall rules. To actually test the implementation as well as the configuration of a firewall, we use HOL-TestGen [1, 2, 5] to generate test cases that can be used to validate the compliance of real network middleboxes (e.g., firewalls, routers). In this document, we focus on the Isabelle formalization of network access control policies. For details of the overall approach, we refer the reader elsewhere [7]

1.2 The Unified Policy Framework (UPF)

Our formalization of firewall policies is based on the Unified Policy Framework (UPF). In this section, we briefly introduce UPF, for all details we refer the reader to) [6].

UPF is a generic framework for policy modeling with the primary goal of being used for test case generation. The interested reader is referred to [4] for an application of UPF to large scale access control policies in the health care domain; a comprehensive treatment is also contained in the reference manual coming with the distribution on the HOL-TestGen website (<http://www.brucker.ch/projects/hol-testgen/>). UPF is based on

the following four principles:

1. policies are represented as *functions* (rather than relations),
2. policy combination avoids conflicts by construction,
3. the decision type is three-valued (allow, deny, undefined),
4. the output type does not only contain the decision but also a ‘slot’ for arbitrary result data.

Formally, the concept of a policy is specified as a partial function from some input to a decision value and additional some output. *Partial* functions are used because elementary policies are described by partial system behavior, which are glued together by operators such as function override and functional composition.

$$\text{type_synonym } \alpha \mapsto \beta = \alpha \rightarrow \beta \text{ decision}$$

where the enumeration type decision is

$$\text{datatype } \alpha \text{ decision} = \text{allow } \alpha \mid \text{deny } \alpha$$

As policies are partial functions or ‘maps’, the notions of a *domain* $\text{dom } p :: \alpha \rightarrow \beta \Rightarrow \alpha \text{ set}$ and a *range* $\text{ran } p :: [\alpha \rightarrow \beta] \Rightarrow \beta \text{ set}$ can be inherited from the Isabelle library.

Inspired by the Z notation [8], there is the concept of *domain restriction* $_ \triangleleft _$ and *range restriction* $_ \triangleright _$, defined as:

$$\begin{aligned} \text{definition } _ \triangleleft _ &:: \alpha \text{ set} \Rightarrow \alpha \mapsto \beta \Rightarrow \alpha \mapsto \beta \\ \text{where } S \triangleleft p &= \lambda x. \text{ if } x \in S \text{ then } p x \text{ else } \perp \\ \text{definition } _ \triangleright _ &:: \alpha \mapsto \beta \Rightarrow \beta \text{ decision set} \Rightarrow \alpha \mapsto \beta \\ \text{where } p \triangleright S &= \lambda x. \text{ if } (\text{the}(p x)) \in S \text{ then } p x \text{ else } \perp \end{aligned}$$

The operator ‘the’ strips off the Some, if it exists. Otherwise the range restriction is underspecified.

There are many operators that change the result of applying the policy to a particular element. The essential one is the *update*:

$$p(x \mapsto t) = \lambda y. \text{ if } y = x \text{ then } |t| \text{ else } p y$$

Next, there are three categories of elementary policies in UPF, relating to the three possible decision values:

- The empty policy; undefined for all elements: $\emptyset = \lambda x. \perp$
- A policy allowing everything, written as $A_f f$, or A_U if the additional output is unit (defined as $\lambda x. [\text{allow}()]$).
- A policy denying everything, written as $D_f f$, or D_U if the additional output is unit.

The most often used approach to define individual rules is to define a rule as a refinement of one of the elementary policies, by using a domain restriction. As an example,

$$\{(Alice, obj1, read)\} \triangleleft A_U$$

Finally, rules can be combined to policies in three different ways:

- Override operators: used for policies of the same type, written as $_ \oplus_i _$.
- Parallel combination operators: used for the parallel composition of policies of potentially different type, written as $_ \otimes_i _$.
- Sequential combination operators: used for the sequential composition of policies of potentially different type, written as $_ \circ_i _$.

All three combinators exist in four variants, depending on how the decisions of the constituent policies are to be combined. For example, the $_ \otimes_2 _$ operator is the parallel combination operator where the decision of the second policy is used.

Several interesting algebraic properties are proved for UPF operators. For example, distributivity

$$(P_1 \oplus P_2) \otimes P_3 = (P_1 \otimes P_3) \oplus (P_2 \otimes P_3)$$

Other UPF concepts are introduced in this paper on-the-fly when needed.

2 UPF Firewall

```
theory
  UPF-Firewall
  imports
    PacketFilter/PacketFilter
    NAT/NAT
    FWNormalisation/FWNormalisation
    StatefulFW/StatefulFW
begin
```

This is the main entry point for specifications of firewall policies.

```
end
```

2.1 Network Models

```
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.

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

- 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 pf 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

2.1.1 Packets and Networks

theory

NetworkCore

imports

Main

begin

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

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

The ID is 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 ' α src = ' α
type-synonym ' α dest = ' α

instance *int* ::*adr* ..

instance *nat* ::*adr* ..

```
instance fun :: (adr,adr) adr ..
instance prod :: (adr,adr) adr ..
```

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
```

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 ..
instance nat :: port ..
instance fun :: (port,port) port ..
instance prod :: (port,port) port ..
```

A packet therefore has two parameters, the first being the address, the second the content. For the sake of simplicity, we do not allow to have a different address representation format for the source and the destination of a packet.

To access the different parts of a packet directly, we define a couple of projectors:

```
definition id :: (' $\alpha$ ::adr,' $\beta$ ) packet  $\Rightarrow$  id
where id = fst
```

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

```
definition content :: (' $\alpha$ ::adr,' $\beta$ ) packet  $\Rightarrow$  ' $\beta$  content
where content = snd o snd o snd
```

```
datatype protocol = tcp | udp
```

```
lemma either:  $\llbracket a \neq \text{tcp}; a \neq \text{udp} \rrbracket \implies \text{False}$ 
by (case-tac a,simp-all)
```

```
lemma either2[simp]:  $(a \neq \text{tcp}) = (a = \text{udp})$ 
by (case-tac a,simp-all)
```

```
lemma either3[simp]:  $(a \neq \text{udp}) = (a = \text{tcp})$ 
by (case-tac a,simp-all)
```

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

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

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

```
type-synonym ' $\alpha$  net = ' $\alpha$  set set
```

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

definition

```
in-subnet :: ' $\alpha$ ::adr  $\Rightarrow$  ' $\alpha$  net  $\Rightarrow$  bool (infixl  $\sqsubset$  100) where
in-subnet a S = ( $\exists$  s  $\in$  S. a  $\in$  s)
```

The following lemmas will be useful later.

lemma in-subnet:

```
(a, e)  $\sqsubset$   $\{\{(x_1,y). P x_1 y\}\} = P a e$ 
by (simp add: in-subnet-def)
```

lemma src-in-subnet:

```
src(q,(a,e),r,t)  $\sqsubset$   $\{\{(x_1,y). P x_1 y\}\} = P a e$ 
by (simp add: in-subnet-def in-subnet src-def)
```

lemma dest-in-subnet:

```
dest (q,r,((a),e),t)  $\sqsubset$   $\{\{(x_1,y). P x_1 y\}\} = P a e$ 
by (simp add: in-subnet-def in-subnet dest-def)
```

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

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

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

end

2.1.2 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
```

2.1.3 Datatype Addresses with Ports

```
theory
  DatatypePort
  imports
    NetworkCore
begin

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

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

  type-synonym
    port = int
  type-synonym
    DatatypePort = (DatatypeAddress × port)

  instance DatatypeAddress :: adr ..

  definition
    dmz::DatatypePort net where
```

```

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

overloading src-port-datatype  $\equiv$  src-port :: (' $\alpha$ ::addr,' $\beta$ ) packet  $\Rightarrow$  ' $\gamma$ ::port
begin
definition
  src-port-datatype (x::(DatatypePort,' $\beta$ ) packet)  $\equiv$  (snd o fst o snd) x
end

overloading dest-port-datatype  $\equiv$  dest-port :: (' $\alpha$ ::addr,' $\beta$ ) packet  $\Rightarrow$  ' $\gamma$ ::port
begin
definition
  dest-port-datatype (x::(DatatypePort,' $\beta$ ) packet)  $\equiv$  (snd o fst o snd o snd) x
end

overloading subnet-of-datatype  $\equiv$  subnet-of :: ' $\alpha$ ::addr  $\Rightarrow$  ' $\alpha$  net
begin
definition
  subnet-of-datatype (x::DatatypePort)  $\equiv$  {{(a,b::int). a = fst x}}
end

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

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

lemmas DatatypePortLemmas = src-port dest-port src-port-datatype-def
dest-port-datatype-def

end

```

2.1.4 Integer Addresses

```

theory
  IntegerAddress
imports
  NetworkCore
begin

```

A theory where addresses are modelled as Integers.

type-synonym

$adr_i = int$

end

2.1.5 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

$adr_{ip} = address \times port$

overloading $src\text{-}port\text{-}int \equiv src\text{-}port :: ('\alpha::addr, '\beta) packet \Rightarrow '\gamma::port$

begin

definition

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

end

overloading $dest\text{-}port\text{-}int \equiv dest\text{-}port :: ('\alpha::addr, '\beta) packet \Rightarrow '\gamma::port$

begin

definition

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

end

overloading $subnet\text{-}of\text{-}int \equiv subnet\text{-}of :: '\alpha::addr \Rightarrow '\alpha net$

begin

definition

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

end

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

```

by (simp add: src-port-int-def in-subnet)
lemma dest-port: dest-port (a,d,x::adrip,e) = snd x
by (simp add: dest-port-int-def in-subnet)
lemmas adripLemmas = src-port dest-port src-port-int-def dest-port-int-def
end

```

2.1.6 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 ..

```

```

overloading src-port-int-TCPUDP ≡ src-port :: ('α::addr,'β) packet ⇒ 'γ::port
begin
definition
  src-port-int-TCPUDP (x::(adripp,'β) packet) ≡ (fst o snd o fst o snd) x
end

```

```

overloading dest-port-int-TCPUDP ≡ dest-port :: ('α::addr,'β) packet ⇒ 'γ::port
begin
definition
  dest-port-int-TCPUDP (x::(adripp,'β) packet) ≡ (fst o snd o fst o snd o snd) x
end

```

```

overloading subnet-of-int-TCPUDP ≡ subnet-of :: 'α::addr ⇒ 'α net
begin

```

```

definition
  subnet-of-int-TCPUDP ( $x::(adr_{ipp})$ )  $\equiv \{\{(a,b,c). a = fst\ } x\}::adr_{ipp}$  net
end

overloading src-protocol-int-TCPUDP  $\equiv$  src-protocol :: (' $\alpha$ ::adr, ' $\beta$ ') packet  $\Rightarrow$  protocol
begin
definition
  src-protocol-int-TCPUDP ( $x::(adr_{ipp},'\beta)$  packet)  $\equiv (snd\ o\ snd\ o\ fst\ o\ snd)\ x$ 
end

overloading dest-protocol-int-TCPUDP  $\equiv$  dest-protocol :: (' $\alpha$ ::adr, ' $\beta$ ') packet  $\Rightarrow$  proto-
col
begin
definition
  dest-protocol-int-TCPUDP ( $x::(adr_{ipp},'\beta)$  packet)  $\equiv (snd\ o\ snd\ o\ fst\ o\ snd\ o\ snd)\ x$ 
end

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

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

  Common test constraints:

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

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

lemmas adrippLemmas  $=$  src-port    dest-port    src-port-int-TCPUDP-def
dest-port-int-TCPUDP-def    src-protocol-int-TCPUDP-def    dest-protocol-int-TCPUDP-def    sub-
net-of-int-TCPUDP-def

lemmas adrippTestConstraints  $=$  port-positive-def fix-values-def

end

```

2.1.7 Formalizing 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-ip = (int × int × int × int)

type-synonym
  port = int

type-synonym
  ipv4 = (ipv4-ip × port)

overloading src-port-ipv4 ≡ src-port :: ('α::adr,'β) packet ⇒ 'γ::port
begin
definition
  src-port-ipv4 (x::(ipv4,β) packet) ≡ (snd o fst o snd) x
end

overloading dest-port-ipv4 ≡ dest-port :: ('α::adr,'β) packet ⇒ 'γ::port
begin
definition
  dest-port-ipv4 (x::(ipv4,β) packet) ≡ (snd o fst o snd o snd) x
end

overloading subnet-of-ipv4 ≡ subnet-of :: 'α::adr ⇒ 'α net
begin
definition
  subnet-of-ipv4 (x::ipv4) ≡ {{(a,b::int). a = fst x}}
end

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

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

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

```

```

lemmas IPv4Lemmas = src-port dest-port src-port-ipv4-def dest-port-ipv4-def
end

```

2.1.8 IPv4 with Ports and Protocols

theory

IPv4-TCPUDP

imports *IPv4*

begin

type-synonym

ipv4-TCPUDP = (*ipv4-ip* × *port* × *protocol*)

instance *protocol* :: *adr* ..

overloading *src-port-ipv4-TCPUDP* ≡ *src-port* :: (' α ::*adr*, ' β ') *packet* \Rightarrow ' γ ::*port*

begin

definition

src-port-ipv4-TCPUDP (*x*::(*ipv4-TCPUDP*, ' β ') *packet*) ≡ (*fst o snd o fst o snd*) *x*

end

overloading *dest-port-ipv4-TCPUDP* ≡ *dest-port* :: (' α ::*adr*, ' β ') *packet* \Rightarrow ' γ ::*port*

begin

definition

dest-port-ipv4-TCPUDP (*x*::(*ipv4-TCPUDP*, ' β ') *packet*) ≡ (*fst o snd o fst o snd o snd*) *x*

end

overloading *subnet-of-ipv4-TCPUDP* ≡ *subnet-of* :: ' α ::*adr* \Rightarrow ' α *net*

begin

definition

subnet-of-ipv4-TCPUDP (*x*::(*ipv4-TCPUDP*, ' β ') *packet*) ≡ {{(a,b). a = *fst x*}}::(*ipv4-TCPUDP* *net*)

end

overloading *dest-protocol-ipv4-TCPUDP* ≡ *dest-protocol* :: (' α ::*adr*, ' β ') *packet* \Rightarrow *protocol*

begin

definition

dest-protocol-ipv4-TCPUDP (*x*::(*ipv4-TCPUDP*, ' β ') *packet*) ≡ (*snd o snd o fst o snd o snd*) *x*

end

```

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

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

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

lemmas Ipv4-TCPUDPLemmas = src-port dest-port src-port-ipv4-TCPUDP-def
dest-port-ipv4-TCPUDP-def
dest-protocol-ipv4-TCPUDP-def subnet-of-ipv4-TCPUDP-def
end

```

2.2 Network Policies: Packet Filter

```

theory
  PacketFilter
imports
  NetworkModels
  ProtocolPortCombinators
  Ports
begin
end

```

2.2.1 Policy Core

```

theory
  PolicyCore
imports
  NetworkCore
  UPF.UPF
begin

```

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

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

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

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

definition
p-accept :: (' α , ' β) packet \Rightarrow (' α , ' β) FWPolicy \Rightarrow bool **where**

$p\text{-accept } p \text{ pol} = (\text{pol } p = [\text{allow } ()])$

end

2.2.2 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 :: ' $\alpha::\text{adr net} \Rightarrow ((\alpha, \beta) \text{ packet} \mapsto \text{unit})$ ' **where**

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

definition

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

deny-all-from src-net = $\{\text{pa. src pa} \sqsubseteq \text{src-net}\} \triangleleft D_U$

definition

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

allow-all-to dest-net = $\{\text{pa. dest pa} \sqsubseteq \text{dest-net}\} \triangleleft A_U$

definition

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

deny-all-to dest-net = $\{\text{pa. dest pa} \sqsubseteq \text{dest-net}\} \triangleleft D_U$

definition

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

allow-all-from-to src-net dest-net =

$\{\text{pa. src pa} \sqsubseteq \text{src-net} \wedge \text{dest pa} \sqsubseteq \text{dest-net}\} \triangleleft A_U$

definition

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

deny-all-from-to src-net dest-net = $\{\text{pa. src pa} \sqsubseteq \text{src-net} \wedge \text{dest pa} \sqsubseteq \text{dest-net}\} \triangleleft D_U$

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

2.2.3 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 ones defined in the PolicyCombinators theory.

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

- $\text{src_port} :: ('\alpha, '\beta)\text{packet} \Rightarrow ('\gamma :: \text{port})$
- $\text{dest_port} :: ('\alpha, '\beta)\text{packet} \Rightarrow ('\gamma :: \text{port})$

definition

allow-all-from-port :: '\alpha::adr net \Rightarrow (' γ ::port) \Rightarrow (('\alpha, '\beta) packet \mapsto unit) where

allow-all-from-port src-net s-port = {pa. src-port pa = s-port} \triangleleft allow-all-from src-net

definition

deny-all-from-port :: '\alpha::adr net \Rightarrow '\gamma::port \Rightarrow (('\alpha, '\beta) packet \mapsto unit) where

deny-all-from-port src-net s-port = {pa. src-port pa = s-port} \triangleleft deny-all-from src-net

definition

allow-all-to-port :: '\alpha::adr net \Rightarrow '\gamma::port \Rightarrow (('\alpha, '\beta) packet \mapsto unit) where

allow-all-to-port dest-net d-port = {pa. dest-port pa = d-port} \triangleleft allow-all-to dest-net

definition

deny-all-to-port :: '\alpha::adr net \Rightarrow '\gamma::port \Rightarrow (('\alpha, '\beta) packet \mapsto unit) where

deny-all-to-port dest-net d-port = {pa. dest-port pa = d-port} \triangleleft deny-all-to dest-net

definition

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

where

allow-all-from-port-to src-net s-port dest-net

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

definition

deny-all-from-port-to::' α ::adr net \Rightarrow ' γ ::port \Rightarrow ' α ::adr net \Rightarrow ((α , β) packet \mapsto unit)

where

deny-all-from-port-to src-net s-port dest-net

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

definition

*allow-all-from-port-to-port::' 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 src-net s-port dest-net d-port =

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

definition

*deny-all-from-port-to-port :: ' 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 src-net s-port dest-net d-port =

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

definition

*allow-all-from-to-port :: ' α ::adr net \Rightarrow ' α ::adr net \Rightarrow ' γ ::port \Rightarrow ((α , β) packet \mapsto unit) **where***

allow-all-from-to-port src-net dest-net d-port =

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

definition

*deny-all-from-to-port :: ' α ::adr net \Rightarrow ' α ::adr net \Rightarrow ' γ ::port \Rightarrow ((α , β) packet \mapsto unit) **where***

deny-all-from-to-port src-net dest-net d-port =

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

definition

*allow-from-port-to :: ' γ ::port \Rightarrow ' α ::adr net \Rightarrow ' α ::adr net \Rightarrow ((α , β) packet \mapsto unit) **where***

allow-from-port-to port src-net dest-net =

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

definition

*deny-from-port-to :: ' γ ::port \Rightarrow ' α ::adr net \Rightarrow ' α ::adr net \Rightarrow ((α , β) packet \mapsto unit) **where***

deny-from-port-to port src-net dest-net =

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

definition

*allow-from-to-port :: ' γ ::port \Rightarrow ' α ::adr net \Rightarrow ' α ::adr net \Rightarrow ((α , β) packet \mapsto unit) **where***

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

definition

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

definition

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

definition

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

definition

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

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

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

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

```

2.2.4 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 ones defined in the *PolicyCombinators* theory.

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

- $\text{src_port} :: ('\alpha, '\beta)\text{packet} \Rightarrow ('\gamma :: \text{port})$
- $\text{dest_port} :: ('\alpha, '\beta)\text{packet} \Rightarrow ('\gamma :: \text{port})$

definition

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

allow-all-from-port-prot p src-net s-port =
 $\{pa. \text{dest-protocol } pa = p\} \triangleleft \text{allow-all-from-port src-net s-port}$

definition

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

deny-all-from-port-prot p src-net s-port =
 $\{pa. \text{dest-protocol } pa = p\} \triangleleft \text{deny-all-from-port src-net s-port}$

definition

allow-all-to-port-prot :: *protocol* $=> ' \alpha :: \text{adr net} \Rightarrow ' \gamma :: \text{port} \Rightarrow (('\alpha, '\beta) \text{ packet} \mapsto \text{unit})$

where

allow-all-to-port-prot p dest-net d-port =

$\{pa. \text{dest-protocol } pa = p\} \triangleleft \text{allow-all-to-port dest-net } d\text{-port}$

definition

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

where

$\text{deny-all-to-port-prot } p \text{ dest-net } d\text{-port} =$

$\{pa. \text{dest-protocol } pa = p\} \triangleleft \text{deny-all-to-port dest-net } d\text{-port}$

definition

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

where

$\text{allow-all-from-port-to-prot } p \text{ src-net } s\text{-port dest-net} =$

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

definition

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

where

$\text{deny-all-from-port-to-prot } p \text{ src-net } s\text{-port dest-net} =$

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

definition

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

where

$\text{allow-all-from-port-to-port-prot } p \text{ src-net } s\text{-port dest-net } d\text{-port} =$

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

definition

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

where

$\text{deny-all-from-port-to-port-prot } p \text{ src-net } s\text{-port dest-net } d\text{-port} =$

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

definition

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

where

$\text{allow-all-from-to-port-prot } p \text{ src-net dest-net } d\text{-port} =$

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

definition

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

definition

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

definition

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

definition

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

definition

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

definition

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

definition

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

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

2.2.5 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.

definition http:int **where** http = 80

```
lemma http1: x ≠ 80 ⇒ x ≠ http
    by (simp add: http-def)
```

```
lemma http2: x ≠ 80 ⇒ http ≠ x
```

```

by (simp add: http-def)
definition smtp::int where smtp = 25

lemma smtp1: x ≠ 25  $\Rightarrow$  x ≠ smtp
by (simp add: smtp-def)

lemma smtp2: x ≠ 25  $\Rightarrow$  smtp ≠ x
by (simp add: smtp-def)

definition ftp::int where ftp = 21

lemma ftp1: x ≠ 21  $\Rightarrow$  x ≠ ftp
by (simp add: ftp-def)

lemma ftp2: x ≠ 21  $\Rightarrow$  ftp ≠ x
by (simp add: ftp-def)

```

And so on for all desired port numbers.

```

lemmas Ports = http1 http2 ftp1 ftp2 smtp1 smtp2

end

```

2.2.6 Network Address Translation

```

theory
  NAT
imports
  ../PacketFilter/PacketFilter
begin

definition src2pool :: ' $\alpha$  set  $\Rightarrow$  (' $\alpha$ ::adr, ' $\beta$ ) packet set where
  src2pool t =  $(\lambda p. (\{(i,s,d,da). (i = id p \wedge s \in t \wedge d = dest p \wedge da = content p)\}))$ 

definition src2poolAP where
  src2poolAP t = A_f (src2pool t)

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

definition src2poolPort :: 'int set  $\Rightarrow$  (adr_ip, ' $\beta$ ) packet  $\Rightarrow$  (adr_ip, ' $\beta$ ) packet set where
  src2poolPort t =  $(\lambda p. (\{(i,(s1,s2),(d1,d2),da).$ 

```

$$(i = id p \wedge s1 \in t \wedge s2 = (snd (src p)) \wedge d1 = (fst (dest p)) \wedge d2 = snd (dest p) \wedge da = content p\}))$$

definition $src2poolPort\text{-Protocol} :: int\ set \Rightarrow (adr_{ipp},'\beta)\ packet \Rightarrow (adr_{ipp},'\beta)\ packet$
set where

$$src2poolPort\text{-Protocol } t = (\lambda p. (\{(i,(s1,s2,s3),(d1,d2,d3), da).\\ (i = id p \wedge s1 \in t \wedge s2 = (fst (snd (src p))) \wedge s3 = snd (snd (src p)) \wedge (d1,d2,d3) = dest p \wedge da = content p\}\}))$$

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

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

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

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

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

definition $srcPat2pool\text{-IntProtocol} :: int\ set \Rightarrow int\ set \Rightarrow (adr_{ipp},'\beta)\ packet \mapsto (adr_{ipp},'\beta)\ packet$
set where
 $srcPat2pool\text{-IntProtocol } srcs\ transl = \{x. (fst (src x)) \in srcs\} \triangleleft (A_f (srcPat2poolPort\text{-Protocol-t } transl))$

The following lemmas are used for achieving a normalized output format of packages after applying NAT. This is used, e.g., by our firewall execution tool.

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

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

by auto

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

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

by auto

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

$$\begin{aligned}
& \forall a aa b ba. aba = ((a, aa, b), ba) \longrightarrow i = i1 \wedge i115 < s1 \wedge s1 < \\
& i124 \wedge \\
& s3 = iudp \wedge s2 = ii1 \wedge a = i110 \wedge aa = i3 \wedge b = itcp \wedge ba = \\
& \text{data}\} \\
= & \{(i, (s1, s2, s3), aba). \\
& \quad i = i1 \wedge i115 < s1 \wedge s1 < i124 \wedge s3 = iudp \wedge s2 = ii1 \wedge \\
& (\lambda ((a,aa,b),ba). a = i110 \& aa = i3 \& b = itcp \& ba = \text{data}) \text{ aba}\}
\end{aligned}$$

by auto

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

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

by auto

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

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

```

 $b = X607X4 \wedge ba = data\}$ 
by auto

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

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

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

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

```

2.3 Firewall Policy Normalisation

```

theory
FWNormalisation
imports
NormalisationIPPPProofs
ElementaryRules

```

```
begin
```

```
end
```

2.3.1 Policy Normalisation: Core Definitions

```
theory
```

```
FWNormalisationCore
```

```
imports
```

```
.. / PacketFilter / PacketFilter
```

```
begin
```

This theory contains all the definitions used for policy normalisation as described in [3, 7].

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.

Basics

We define a very simple policy language:

```
datatype ('α,'β) Combinators =
  DenyAll
  | DenyAllFromTo 'α 'α
  | AllowPortFromTo 'α 'α 'β
  | Conc (('α,'β) Combinators) (('α,'β) Combinators) (infixr ⊕ 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 constants for the type definition and a primitive recursive definition for each desired address model.

Auxiliary definitions and functions.

This section 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 ⊕ va) = undefined

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

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

fun destnets where
  destnets DenyAll = []
|destnets (DenyAllFromTo x y) = [y]
|destnets (AllowPortFromTo x y p) = [y]
|(destnets (x ⊕ 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⊕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) ∈ set xs) ∨ (a,b) ∈ 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 ⊕ 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⊕y) = first-srcNet x

```

| *first-srcNet* *x* = *srcNet* *x*

fun (*sequential*) *first-destNet* **where**

```

first-destNet (x⊕y) = first-destNet x

```

| *first-destNet* *x* = *destNet* *x*

fun (*sequential*) *first-bothNet* **where**

```

first-bothNet (x⊕y) = first-bothNet x

```

| *first-bothNet* *x* = *bothNet* *x*

fun (*sequential*) *in-list* **where**

```

in-list DenyAll l = True
|in-list x l = (bothNet x ∈ set l)

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

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

fun sdnets where
  sdnets DenyAll = {}
| sdnets (DenyAllFromTo a b) = {(a,b)}
| sdnets (AllowPortFromTo a b c) = {(a,b)}
| sdnets (a ⊕ b) = sdnets a ∪ sdnets b

definition packet-Nets where packet-Nets x a b = ((src x ⊑ a ∧ dest x ⊑ b) ∨
  (src x ⊑ b ∧ dest x ⊑ a))

definition subnetsOfAdr where subnetsOfAdr a = {x. a ⊑ x}

definition fst-set where fst-set s = {a. ∃ b. (a,b) ∈ s}

definition snd-set where snd-set s = {a. ∃ b. (b,a) ∈ s}

fun memberP where
  memberP r (x#xs) = (member r x ∨ memberP r xs)
|memberP r [] = False

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


```

Invariants

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

```

fun wellformed-policy1:: (('α, 'β) Combinators) list ⇒ bool where
  wellformed-policy1 [] = True
| wellformed-policy1 (x#xs) = (DenyAll ∉ (set xs))

```

There is a DenyAll at the first position

```

fun wellformed-policy1-strong:: (('α, 'β) Combinators) list ⇒ 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  $\wedge$  OnlyTwoNets xs)
| OnlyTwoNets [] = True
```

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

```
fun noDenyAll1 where
noDenyAll1 (DenyAll#xs) = noDenyAll xs
| noDenyAll1 xs = noDenyAll xs
```

```
fun separated where
separated (x#xs) = (( $\forall$  s. s  $\in$  set xs  $\longrightarrow$  disjSD-2 x s)  $\wedge$  separated xs)
| separated [] = True
```

```

fun NetsCollected where
  NetsCollected (x#xs) = (((first-bothNet x ≠ firstList xs) →
    (forall 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

```

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:: (( $\alpha$ ,  $\beta$ ) Combinators) list  $\Rightarrow$ 
    (( $\alpha$ ,  $\beta$ ) Combinators) list
where
  (removeShadowRules2 ((AllowPortFromTo x y p) $\#$ z)) =
    (if (((DenyAllFromTo x y)  $\in$  set z))
     then ((removeShadowRules2 z))
     else (((AllowPortFromTo x y p) $\#$ (removeShadowRules2 z))))
| removeShadowRules2 (x $\#$ y) = x $\#$ (removeShadowRules2 y)
| removeShadowRules2 [] = []

```

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

```

fun smaller :: ( $\alpha$ ,  $\beta$ ) Combinators  $\Rightarrow$ 
    ( $\alpha$ ,  $\beta$ ) Combinators  $\Rightarrow$ 
    (( $\alpha$ ) set) list  $\Rightarrow$  bool
where
  smaller DenyAll x l = True
| smaller x DenyAll l = False
| smaller x y l =
  ((x = y)  $\vee$  (if (bothNet x) = (bothNet y) then
   (case y of (DenyAllFromTo a b)  $\Rightarrow$  (x = DenyAllFromTo b a)
   | -  $\Rightarrow$  True)
  else
    (position (bothNet x) l  $\leq$  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 $\leftarrow$ xs.  $\neg$  (smaller x y l)] l) @ [x] @ (qsort [y $\leftarrow$ xs. smaller x y l] l)

```

```

lemma qsort-permutes:
  set (qsort xs l) = set xs
  apply (induct xs l rule: qsort.induct)
  by (auto)

```

```

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

```

```

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

```

```

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

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

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

```

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

```

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

```

Remove duplicate rules. This is especially necessary as insertDenies might have inserted duplicate rules. The second function is supposed to work on a list of policies. Only rules which are duplicated within the same policy are removed.

```

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

```

```

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

```

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::(( $\alpha$ ,  $\beta$ ) Combinators) list  $\Rightarrow$ 
    (( $\alpha$ ,  $\beta$ ) Combinators) where
    list2policyR (x#[]) = x
    |list2policyR (x#y) = (list2policyR y)  $\oplus$  x
    |list2policyR [] = undefined

```

We provide the definitions for two address representations.

IntPort

```

fun C :: (adrip net, port) Combinators  $\Rightarrow$  (adrip, DummyContent) packet  $\mapsto$  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  $\oplus$  y) = C x ++ C y

fun CRotate :: (adrip net, port) Combinators  $\Rightarrow$  (adrip, DummyContent) packet  $\mapsto$  unit
where
    CRotate DenyAll = C DenyAll
    |CRotate (DenyAllFromTo x y) = C (DenyAllFromTo x y)
    |CRotate (AllowPortFromTo x y p) = C (AllowPortFromTo x y p)
    |CRotate (x  $\oplus$  y) = ((CRotate y) ++ ((CRotate x)))

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

```

```

lemma check: rev (policy2list (rotatePolicy p)) = policy2list p
  apply (induct p)
  by (simp-all)

```

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

```

fun (sequential) wellformed-policy2 where
    wellformed-policy2 [] = True
    |wellformed-policy2 (DenyAll#xs) = wellformed-policy2 xs
    |wellformed-policy2 (x#xs) = (( $\forall$  c a b. c = DenyAllFromTo a b  $\wedge$  c  $\in$  set xs  $\longrightarrow$ 

```

$$Map.dom (C x) \cap Map.dom (C c) = \{\} \wedge wellformed-policy2 xs$$

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

```
fun (sequential) wellformed-policy3::((adr_ip net,port) Combinators) list  $\Rightarrow$  bool where
  wellformed-policy3 [] = True
  | wellformed-policy3 ((AllowPortFromTo a b p)#xs) = (( $\forall$  r. r  $\in$  set xs  $\longrightarrow$ 
    dom (C r)  $\cap$  dom (C (AllowPortFromTo a b p)) = {})  $\wedge$  wellformed-policy3 xs)
  | wellformed-policy3 (x#xs) = wellformed-policy3 xs
```

definition

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

definition normalize ::

```
(adr_ip net, port) Combinators  $\Rightarrow$ 
  (adr_ip net, port) Combinators list
```

where

```
normalize p = (removeAllDuplicates (insertDenies (separate (sort
  (removeShadowRules2 (remdups ((rm-MT-rules C) (insertDeny
  (removeShadowRules1 (policy2list p))))))) ((Nets-List p))))))
```

definition

```
normalize-manual-order p l = removeAllDuplicates (insertDenies (separate
  (sort (removeShadowRules2 (remdups ((rm-MT-rules C) (insertDeny
  (removeShadowRules1 (policy2list p))))))) ((l)))))
```

definition normalizeQ ::

```
(adr_ip net, port) Combinators  $\Rightarrow$ 
  (adr_ip net, port) Combinators list
```

where

```
normalizeQ p = (removeAllDuplicates (insertDenies (separate (qsort
  (removeShadowRules2 (remdups ((rm-MT-rules C) (insertDeny
  (removeShadowRules1 (policy2list p))))))) ((Nets-List p))))))
```

definition

```
normalize-manual-orderQ p l = removeAllDuplicates (insertDenies (separate
  (qsort (removeShadowRules2 (remdups ((rm-MT-rules C) (insertDeny
    (removeShadowRules1 (policy2list p)))))) ((l)))))
```

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

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

declare C.simps [simp del]
```

TCP_UDP_IntegerPort

```
fun Cp :: (adr_ipp net, protocol × port) Combinators ⇒
  (adr_ipp, DummyContent) packet ↪ 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 ⊕ y) = Cp x ++ Cp y
```

```
fun Dp :: (adr_ipp net, protocol × port) Combinators ⇒
  (adr_ipp, DummyContent) packet ↪ 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 ⊕ y) = Cp (y ⊕ 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) = ((∀ c a b. c = DenyAllFromTo a b ∧ c ∈ set xs →
  Map.dom (Cp x) ∩ Map.dom (Cp c) = {}) ∧ wellformed-policy2Pr xs)
```

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

```
fun (sequential) wellformed-policy3Pr::((adr_ipp net, protocol × port) Combinators) list
⇒ bool where
  wellformed-policy3Pr [] = True
| wellformed-policy3Pr ((AllowPortFromTo a b p)#xs) = ((∀ r. r ∈ set xs →
```

$\text{dom } (\text{Cp } r) \cap \text{dom } (\text{Cp } (\text{AllowPortFromTo } a b p)) = \{\}) \wedge \text{wellformed-policy3Pr}$
 $xs)$
 $| \text{wellformed-policy3Pr } (x\#xs) = \text{wellformed-policy3Pr } xs$

definition

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

definition $\text{normalizePr} ::$

$(\text{adr}_{ipp} \text{ net}, \text{protocol} \times \text{port}) \text{ Combinators}$
 $\Rightarrow (\text{adr}_{ipp} \text{ net}, \text{protocol} \times \text{port}) \text{ Combinators list where}$
 $\text{normalizePr } p = (\text{removeAllDuplicates } (\text{insertDenies } (\text{separate } (\text{sort}$
 $(\text{removeShadowRules2 } (\text{remdups } ((\text{rm-MT-rules } \text{Cp}) \ (\text{insertDeny}$
 $(\text{removeShadowRules1 } (\text{policy2list } p))))))) \ ((\text{Nets-List } p))))$

definition

$\text{normalize-manual-orderPr } p l = \text{removeAllDuplicates } (\text{insertDenies } (\text{separate }$
 $(\text{sort } (\text{removeShadowRules2 } (\text{remdups } ((\text{rm-MT-rules } \text{Cp}) \ (\text{insertDeny}$
 $(\text{removeShadowRules1 } (\text{policy2list } p))))))) \ ((l))))$

definition

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

definition $\text{normalizePrQ} ::$

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

definition

$\text{normalize-manual-orderPrQ } p l = \text{removeAllDuplicates } (\text{insertDenies } (\text{separate }$
 $(\text{qsort } (\text{removeShadowRules2 } (\text{remdups } ((\text{rm-MT-rules } \text{Cp}) \ (\text{insertDeny}$
 $(\text{removeShadowRules1 } (\text{policy2list } p))))))) \ ((l))))$

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

```

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

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

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

declare Cp.simps [simp del]

lemmas PLemmas = C.simps Cp.simps dom-def PolicyCombinators
  PortCombinators.PortCombinatorsCore aux
  ProtocolPortCombinators.ProtocolCombinatorsCore src-def dest-def
  in-subnet-def
  adr_ipppLemmas adr_ipppLemmas

lemma aux: [|x ≠ a; y≠b; (x ≠ y ∧ x ≠ b) ∨ (a ≠ b ∧ a ≠ y)|] ==> {x,a} ≠ {y,b}
  by (auto)

lemma aux2: {a,b} = {b,a}
  by auto
end

```

2.3.2 Normalisation Proofs (Generic)

```

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: dom X ≠ {} ==> X ≠ ∅
  by auto

lemma denyNMT: deny-all ≠ ∅
  apply (rule domNMT)
  by (simp add: deny-all-def dom-def)

```

```

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

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

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

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

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

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

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

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

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

lemma aux5[rule-format]: (DenyAllFromTo a b)  $\in$  set p  $\longrightarrow$  a  $\in$  set (net-list p)

```

```

by (rule net-list-aux.induct,simp-all)

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

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

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

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

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

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

lemma tND1:  $\llbracket \text{allNetsDistinct } p; x \in \text{set } p; y \in \text{set } p; a = \text{srcNet } x;$ 
 $b = \text{destNet } x; c = \text{srcNet } y; d = \text{destNet } y; a \neq c;$ 
 $\text{singleCombinators}[x]; x \neq \text{DenyAll}; \text{singleCombinators}[y];$ 
 $y \neq \text{DenyAll} \rrbracket \implies \text{twoNetsDistinct } a b c d$ 
by (simp add: allNetsDistinct-def twoNetsDistinct-def)

```

```

lemma tND2: [[allNetsDistinct p; x ∈ set p; y ∈ set p; a = srcNet x;
    b = destNet x; c = srcNet y; d = destNet y; b ≠ d;
    singleCombinators[x]; x ≠ DenyAll; singleCombinators[y];
    y ≠ DenyAll] ⇒ twoNetsDistinct a b c d
by (simp add: allNetsDistinct-def twoNetsDistinct-def)

lemma tND: [[allNetsDistinct p; x ∈ set p; y ∈ set p; a = srcNet x;
    b = destNet x; c = srcNet y; d = destNet y; a ≠ c ∨ b ≠ d;
    singleCombinators[x]; x ≠ DenyAll; singleCombinators[y]; y ≠ DenyAll]
    ⇒ twoNetsDistinct a b c d
apply (case-tac a ≠ c, simp-all)
apply (erule-tac x = x and y = y in tND1, simp-all)
apply (erule-tac x = x and y = y in tND2, simp-all)
done

lemma aux7: [[DenyAllFromTo a b ∈ set p; allNetsDistinct ((DenyAllFromTo c d) # p);
    a ≠ c ∨ b ≠ d] ⇒ twoNetsDistinct a b c d
apply (erule-tac x = DenyAllFromTo a b and y = DenyAllFromTo c d in tND)
by simp-all

lemma aux7a: [[DenyAllFromTo a b ∈ set p;
    allNetsDistinct ((AllowPortFromTo c d po) # p); a ≠ c ∨ b ≠ d] ⇒
    twoNetsDistinct a b c d
apply (erule-tac x = DenyAllFromTo a b and
    y = AllowPortFromTo c d po in tND)
by simp-all

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

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

lemma aux[rule-format]: a ∈ set (removeShadowRules2 p) → a ∈ set p
apply (case-tac a)
by (rule removeShadowRules2.induct, simp-all)+

lemma aux12: [a ∈ x; b ∉ x] ⇒ a ≠ b
by auto

lemma ND0aux1[rule-format]: DenyAllFromTo x y ∈ set b ⇒

```

$x \in \text{set}(\text{net-list-aux } b)$
by (*metis aux5 net-list.simps set-remdups*)

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

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

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

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

apply (*induct a*)
apply *simp-all*
apply *clarify*
apply (*drule mp, erule singleCombinatorsConc*)
subgoal for $a a' x$
apply (*case-tac a*)
apply (*simp-all add: contra-subsetD*)
apply (*metis contra-subsetD*)
apply (*metis ND0aux1 ND0aux2 contra-subsetD*)
apply (*metis ND0aux3 ND0aux4 contra-subsetD*)
done
done

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

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

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

apply (*simp add: allNetsDistinct-def*)
apply (*rule allI*)+
apply (*rule impI*)+
subgoal for $x y$

```

apply (drule-tac  $x = x$  in spec, drule-tac  $x = y$  in spec)
  using aNDSubsetaux by blast
done

lemma aNDSetsEq:  $\llbracket \text{singleCombinators } a; \text{singleCombinators } b; \text{set } a = \text{set } b;$ 
 $\text{allNetsDistinct } b \rrbracket \implies \text{allNetsDistinct } a$ 
apply (simp add: allNetsDistinct-def)
apply (rule allI)+
apply (rule impI)+
subgoal for  $x y$ 
  apply (drule-tac  $x = x$  in spec, drule-tac  $x = y$  in spec)
    using aNDSetsEqaux by auto
done

lemma SCConca:  $\llbracket \text{singleCombinators } p; \text{singleCombinators } [a] \rrbracket \implies$ 
 $\text{singleCombinators } (a\#p)$ 
by(metis aux0-1)

lemma aux3[simp]:  $\llbracket \text{singleCombinators } p; \text{singleCombinators } [a];$ 
 $\text{allNetsDistinct } (a\#p) \rrbracket \implies \text{allNetsDistinct } (a\#a\#p)$ 
by (metis aNDSetsEq aux0-1 insert-absorb2 list.set(2))

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

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

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

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

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

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

```

```

lemma aux79[rule-format]:  $y \in \text{set}(\text{inser}t x a l) \rightarrow y \neq x \rightarrow y \in \text{set} a$ 
  apply (induct a)
  by auto

lemma aux80:  $\llbracket y \notin \text{set} p; y \neq x \rrbracket \implies y \notin \text{set}(\text{inser}t x (\text{sort} p l) l)$ 
  apply (metis aux79 set-sort)
  done

lemma WP1Conca:  $\text{DenyAll} \notin \text{set} p \implies \text{wellformed-policy1}(a\#p)$ 
  by (case-tac a,simp-all)

lemma saux[simp]:  $(\text{inser}t \text{DenyAll} p l) = \text{DenyAll}\#p$ 
  by (induct-tac p,simp-all)

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

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

lemma waux3[rule-format]:  $\llbracket x \neq a; x \notin \text{set} p \rrbracket \implies x \notin \text{set}(\text{inser}t a p l)$ 
  by (metis aux79)

lemma wellformed1-sorted-aux[rule-format]:  $\text{wellformed-policy1}(x\#p) \implies$ 
   $\text{wellformed-policy1}(\text{inser}t x p l)$ 
  by (metis NormalisationGenericProofs.set-inser list.set(2) saux waux2 wellformed-eq
    wellformed-policy1-strong.simps(2))

lemma wellformed1-sorted-auxQ[rule-format]:  $\text{wellformed-policy1}(p) \implies$ 
   $\text{wellformed-policy1}(\text{qsort} p l)$ 
  proof (induct p)
    case Nil show ?case by simp
  next
    case (Cons a S) then show ?case
      apply simp-all
      apply (cases a,simp-all)
      by (metis Combinators.simps append-Cons append-Nil qsort.simps(2) set-ConsD
        set-qsort waux2)+
  qed

```

```

lemma SR1Subset: set (removeShadowRules1 p) ⊆ set p
  apply (induct-tac p, simp-all)
  subgoal for x xs
    apply (case-tac x, simp-all)
      apply(auto)
    done
  done

lemma SCSubset[rule-format]: singleCombinators b → set a ⊆ set b →
  singleCombinators a
proof (induct a)
  case Nil thus ?case by simp
next
  case (Cons x xs) thus ?case
    by (meson aux0-0 aux0-5 subsetCE)
qed

lemma setInsert[simp]: set list ⊆ insert a (set list)
  by auto

lemma SC-RS1[rule-format,simp]: singleCombinators p → allNetsDistinct p →
  singleCombinators (removeShadowRules1 p)
  apply (induct-tac p)
  apply simp-all
  using ANDConc singleCombinatorsConc by blast

lemma RS2Set[rule-format]: set (removeShadowRules2 p) ⊆ set p
  apply(induct p, simp-all)
  subgoal for a as
    apply(case-tac a)
    by(auto)
  done

lemma WP1: a ∉ set list ==> a ∉ set (removeShadowRules2 list)
  using RS2Set [of list] by blast

lemma denyAllDom[simp]: x ∈ dom (deny-all)
  by (simp add: UPFDefs(24) domI)

lemma lCdom2: (list2FWpolicy (a @ (b @ c))) = (list2FWpolicy ((a@b)@c))
  by auto

lemma SCConcEnd: singleCombinators (xs @ [xa]) ==> singleCombinators xs

```

```

apply (induct xs, simp-all)
subgoal for a as
  by (case-tac a , simp-all)
done

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

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

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

lemma SCnotConc[rule-format,simp]:  $a \oplus b \in \text{set } p \rightarrow \text{singleCombinators } p \rightarrow \text{False}$ 
  apply (induct p, simp-all)
  subgoal for p ps
    by(case-tac p, simp-all)
  done

lemma auxx8: removeShadowRules1-alternative-rev [x] = [x]
  by (case-tac x, simp-all)

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

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

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

```

```

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

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

lemma rSR1-eq: removeShadowRules1-alternative = removeShadowRules1
  apply (rule ext)
  apply (simp add: removeShadowRules1-alternative-def)
  subgoal for x
    apply (rule-tac xs = x in rev-induct)
    apply simp-all
    subgoal for x xs
      apply (case-tac x = DenyAll, simp-all)
      apply (metis RS1End aux114 rev.simps(2))
      done
    done
  done

lemma domInterMT[rule-format]: [|dom a ∩ dom b = {}; x ∈ dom a|] ⇒ x ∉ dom b
  by auto

lemma domComm: dom a ∩ dom b = dom b ∩ dom a
  by auto

lemma r-not-DA-in-tl[rule-format]:
  wellformed-policy1-strong p → a ∈ set p → a ≠ DenyAll → a ∈ set (tl p)
  by (induct p,simp-all)

lemma wp1-aux1aa[rule-format]: wellformed-policy1-strong p → DenyAll ∈ set p
  by (induct p,simp-all)

lemma mauxa: (∃ r. a b = |r|) = (a b ≠ ⊥)
  by auto

lemma l2p-aux[rule-format]: list ≠ [] →
  list2FWpolicy (a # list) = a ⊕ (list2FWpolicy list)
  by (induct list, simp-all)

lemma l2p-aux2[rule-format]: list = [] ⇒ list2FWpolicy (a # list) = a
  by simp

lemma aux7aa:
  assumes 1 : AllowPortFromTo a b poo ∈ set p

```

```

and 2 : allNetsDistinct ((AllowPortFromTo c d po) # p)
and 3 : a ≠ c ∨ b ≠ d
shows twoNetsDistinct a b c d (is ?H)
proof(cases a ≠ c) print-cases
  case True assume *:a ≠ c show ?H
    by (meson 1 2 True allNetsDistinct-def aux5c list.set-intros(1)
        list.set-intros(2) twoNetsDistinct-def)
next
  case False assume *:¬(a ≠ c) show twoNetsDistinct a b c d
    by (meson 1 2 3 False allNetsDistinct-def aux5d list.set-intros(1)
        list.set-intros(2) twoNetsDistinct-def)
qed

lemma ANDConcEnd: [ allNetsDistinct (xs @ [xa]); singleCombinators xs] ==>
  allNetsDistinct xs
by (rule aNDSubset, auto)

lemma WP1ConcEnd[rule-format]:
  wellformed-policy1 (xs@[xa]) —> wellformed-policy1 xs
by (induct xs, simp-all)

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

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

lemma wellformed1-sortedQ[simp]:
  assumes wp1: wellformed-policy1 p

```

```

shows      wellformed-policy1 (qsort p l)
proof (cases p)
  case Nil thus ?thesis by simp
next
  case (Cons x xs) thus ?thesis
  proof (cases x = DenyAll)
    case True thus ?thesis using assms Cons by simp
  next
    case False thus ?thesis using assms
    by (metis Cons set-qsort False waux2 wellformed-eq
        wellformed-policy1-strong.simps(2))
qed
qed

lemma SC1[simp]: singleCombinators p ==>singleCombinators (removeShadowRules1 p)
  by (erule SCSsubset) (rule SR1Subset)

lemma SC2[simp]: singleCombinators p ==>singleCombinators (removeShadowRules2 p)
  by (erule SCSsubset) (rule RS2Set)

lemma SC3[simp]: singleCombinators p ==> singleCombinators (sort p l)
  by (erule SCSsubset) simp

lemma SC3Q[simp]: singleCombinators p ==> singleCombinators (qsort p l)
  by (erule SCSsubset) simp

lemma aND-RS1[simp]: [[singleCombinators p; allNetsDistinct p]] ==>
  allNetsDistinct (removeShadowRules1 p)
  apply (rule aNDSubset)
  apply (erule SC-RS1, simp-all)
  apply (rule SR1Subset)
  done

lemma aND-RS2[simp]: [[singleCombinators p; allNetsDistinct p]] ==>
  allNetsDistinct (removeShadowRules2 p)
  apply (rule aNDSubset)
  apply (erule SC2, simp-all)
  apply (rule RS2Set)
  done

lemma aND-sort[simp]: [[singleCombinators p; allNetsDistinct p]] ==>
  allNetsDistinct (sort p l)

```

```

apply (rule aNDSubset)
by (erule SC3, simp-all)

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

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

lemma distinct-RS2[rule-format,simp]:  $\text{distinct } p \implies \text{distinct } (\text{removeShadowRules2 } p)$ 
apply (induct p)
apply simp-all
apply clarify
subgoal for a p
apply (case-tac a)
by auto
done

lemma setPaireq:  $\{x, y\} = \{a, b\} \implies x = a \wedge y = b \vee x = b \wedge y = a$ 
by (metis doubleton-eq-iff)

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

lemma pos-noteq[rule-format]:
 $a \in \text{set } l \implies b \in \text{set } l \implies c \in \text{set } l \implies$ 
 $a \neq b \implies \text{position } a \text{ } l \leq \text{position } b \text{ } l \implies \text{position } b \text{ } l \leq \text{position } c \text{ } l \implies$ 
 $a \neq c$ 
proof(induct l)
case Nil show ?case by simp
next
case (Cons a R) show ?case
by (metis (no-types, lifting) Cons.hyps One-nat-def Suc-le-mono le-antisym
length-greater-0-conv list.size(3) nat.inject position.simps(2)
position-positive set-ConsD)
qed

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

```

```

by auto

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

lemma order-trans:
 $\llbracket \text{in-list } x \text{ } l; \text{in-list } y \text{ } l; \text{in-list } z \text{ } l; \text{singleCombinators } [x];$ 
 $\text{singleCombinators } [y]; \text{singleCombinators } [z]; \text{smaller } x \text{ } y \text{ } l; \text{smaller } y \text{ } z \text{ } l \rrbracket \implies$ 
 $\text{smaller } x \text{ } z \text{ } l$ 
apply (case-tac  $x$ , simp-all)
apply (case-tac  $z$ , simp-all)
apply (case-tac  $y$ , simp-all)
apply (case-tac  $y$ , simp-all)
apply (rule conjI|rule impI)+
apply (simp add: setPaireq)
apply (rule conjI|rule impI)+
apply (simp-all split: if-splits)
apply metis+
apply auto[1]
apply (simp add: setPaireq)
apply (rule impI, case-tac  $y$ , simp-all)
apply (simp-all split: if-splits, metis, simp-all add: setPair-noteq setPair-noteq-allow)
apply (case-tac  $z$ , simp-all)
apply (case-tac  $y$ , simp-all)
apply (intro impI|rule conjI)+
apply (simp-all split: if-splits)
apply (simp add: setPair-noteq)
apply (erule pos-noteq, simp-all)
apply auto[1]
apply (rule conjI, simp add: setPair-noteq-allow)
apply (erule pos-noteq, simp-all)
apply auto[1]
apply (rule impI, rule disjI2)
apply (case-tac  $y$ , simp-all split: if-splits )
apply metis
apply (simp-all add: setPair-noteq-allow)
done

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

```

```

apply simp-all
apply (rule impI)+
apply simp
apply (rule-tac y = aa in order-trans)
  apply simp-all
subgoal for p ps
  apply (case-tac p, simp-all)
  done
done

lemma singleCombinatorsStart[simp]: singleCombinators (x#xs)  $\Rightarrow$ 
  singleCombinators [x]
by (case-tac x, simp-all)

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

lemma sortedConcEnd[rule-format]: sorted (a # p) l  $\rightarrow$  in-list a l  $\rightarrow$ 
  all-in-list p l  $\rightarrow$  singleCombinators [a]  $\rightarrow$ 
  singleCombinators p  $\rightarrow$  sorted p l
apply (induct p)
apply (auto intro: singleCombinatorsConc sortedConcStart)
done

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

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

lemma sorted-Cons: [[all-in-list (x#xs) l; singleCombinators (x#xs)]]  $\Rightarrow$ 
  (sorted xs l & ( $\forall y \in$  set xs. smaller x y l)) = (sorted (x#xs) l)
apply auto
apply (rule sorted-Consb, simp-all)
apply (metis singleCombinatorsConc singleCombinatorsStart sortedConcEnd)

```

```

apply (erule sorted-is-smaller)
apply (auto intro: singleCombinatorsConc singleCombinatorsStart in-set-in-list)
done

lemma smaller-antisym:  $\neg \text{smaller } a b l; \text{in-list } a l; \text{in-list } b l;$   

 $\text{singleCombinators}[a]; \text{singleCombinators}[b] \implies \text{smaller } b a l$ 
apply (case-tac a)
  apply simp-all
apply (case-tac b)
  apply simp-all
apply (simp-all split: if-splits)
apply (rule setPaireq)
apply simp
apply (case-tac b)
  apply simp-all
apply (simp-all split: if-splits)
done

lemma set-insort-insert:  $\text{set}(\text{insort } x xs l) \subseteq \text{insert } x (\text{set } xs)$ 
by (induct xs) auto

lemma all-in-listSubset[rule-format]:  $\text{all-in-list } b l \longrightarrow \text{singleCombinators } a \longrightarrow$   

 $\text{set } a \subseteq \text{set } b \longrightarrow \text{all-in-list } a l$ 
by (induct-tac a) (auto intro: in-set-in-list singleCombinatorsConc)

lemma singleCombinators-insort:  $\text{singleCombinators}[x]; \text{singleCombinators}[xs] \implies$   

 $\text{singleCombinators}(\text{insort } x xs l)$ 
by (metis NormalisationGenericProofs.set-insort aux0-0 aux0-1 aux0-5 list.simps(15))

lemma all-in-list-insort:  $\text{all-in-list } xs l; \text{singleCombinators}(x \# xs);$   

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

lemma sorted-ConsA:  $\text{all-in-list}(x \# xs) l; \text{singleCombinators}(x \# xs) \implies$   

 $(\text{sorted}(x \# xs) l) = (\text{sorted } xs l \ \& \ (\forall y \in \text{set } xs. \text{smaller } x y l))$ 
by (metis sorted-Cons)

lemma is-in-insort:  $y \in \text{set } xs \implies y \in \text{set}(\text{insort } x xs l)$ 

```

```

by (simp add: NormalisationGenericProofs.set-insort)

lemma sorted-insorta[rule-format]:
assumes 1 : sorted (insort x xs l) l
  and 2 : all-in-list (x#xs) l
  and 3 : all-in-list (x#xs) l
  and 4 : distinct (x#xs)
  and 5 : singleCombinators [x]
  and 6 : singleCombinators xs
shows sorted xs l
proof (insert 1 2 3 4 5 6, induct xs)
  case Nil show ?case by simp
next
  case (Cons a xs)
  then show ?case
    apply simp
    apply (auto intro: is-in-insort sorted-ConsA set-insort singleCombinators-insort
           singleCombinatorsConc sortedConcEnd all-in-list-insort) [1]
    apply(cases smaller x a l, simp-all)
      by (metis NormalisationGenericProofs.set-insort NormalisationGenericProofs.sorted-Cons
           all-in-list.simps(2) all-in-list-insort aux0-1 insert-iff singleCombinatorsConc
           singleCombinatorsStart singleCombinators-insort)
qed

lemma sorted-insortb[rule-format]:
sorted xs l → all-in-list (x#xs) l → distinct (x#xs) →
singleCombinators [x] → singleCombinators xs → sorted (insort x xs l) l
proof (induct xs)
  case Nil show ?case by simp-all
next
  case (Cons a xs)
  have * : sorted (a # xs) l ⇒ all-in-list (x # a # xs) l ⇒
    distinct (x # a # xs) ⇒ singleCombinators [x] ⇒
    singleCombinators (a # xs) ⇒ sorted (insort x xs l) l
  apply(insert Cons.hyps, simp-all)
  apply(metis sorted-Cons all-in-list.simps(2) singleCombinatorsConc)
  done
  show ?case
    apply (insert Cons.hyps)
    apply (rule impI)+
    apply (insert *, auto intro!: sorted-Consb)
  proof (rule-tac b = x#xs in all-in-listSubset)

```

```

show in-list x l  $\implies$  all-in-list xs l  $\implies$  all-in-list (x # xs) l
    by simp-all
next
    show singleCombinators [x]  $\implies$ 
        singleCombinators (a # xs)  $\implies$ 
        FWNormalisationCore.sorted (FWNormalisationCore.insert x xs l) l
     $\implies$ 
        singleCombinators (FWNormalisationCore.insert x xs l)
    apply (rule singleCombinators-insert, simp-all)
    by (erule singleCombinatorsConc)
next
    show set (FWNormalisationCore.insert x xs l)  $\subseteq$  set (x # xs)
        using NormalisationGenericProofs.set-insert-insert by auto
next
    show singleCombinators [x]  $\implies$ 
        singleCombinators (a # xs)  $\implies$ 
        singleCombinators (a # FWNormalisationCore.insert x xs l)
    by (metis SCCConca singleCombinatorsConc singleCombinatorsStart
        singleCombinators-insert)
next
    fix y
    show FWNormalisationCore.sorted (a # xs) l  $\implies$ 
        singleCombinators [x]  $\implies$  singleCombinators (a # xs)  $\implies$ 
        in-list x l  $\implies$  in-list a l  $\implies$  all-in-list xs l  $\implies$ 
         $\neg$  smaller x a l  $\implies$  y  $\in$  set (FWNormalisationCore.insert x xs l)  $\implies$ 
        smaller a y l
    by (metis NormalisationGenericProofs.set-insert in-set-in-list insert-iff
        singleCombinatorsConc singleCombinatorsStart smaller-antisym
        sorted-is-smaller)
qed
qed

lemma sorted-insert:
     $\llbracket \text{all-in-list } (x \# xs) l; \text{distinct}(x \# xs); \text{singleCombinators } [x];$ 
     $\text{singleCombinators } xs \rrbracket \implies$ 
    sorted (insert x xs l) l = sorted xs l
    by (auto intro: sorted-inserta sorted-insertb)

lemma distinct-insert: distinct (insert x xs l) = (x  $\notin$  set xs  $\wedge$  distinct xs)
    by (induct xs) (auto simp:set-insert)

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

```

```

lemma sort-is-sorted[rule-format]:
  all-in-list p l → distinct p → singleCombinators p → sorted (sort p l) l
  apply (induct p)
  apply simp
  by (metis (no-types, lifting) NormalisationGenericProofs.distinct-sort
    NormalisationGenericProofs.set-sort SC3 all-in-list.simps(2) all-in-listSubset
    distinct.simps(2) set-subset-Cons singleCombinatorsConc singleCombinatorsStart
    sort.simps(2) sorted-insortb)

lemma smaller-sym[rule-format]: all-in-list [a] l → smaller a a l
  by (case-tac a,simp-all)

lemma SC-sublist[rule-format]:
  singleCombinators xs ⇒ singleCombinators (qsort [y←xs. P y] l)
  by (auto intro: SCSsubset)

lemma all-in-list-sublist[rule-format]:
  singleCombinators xs → all-in-list xs l → all-in-list (qsort [y←xs. P y] l) l
  by (auto intro: all-in-listSubset SC-sublist)

lemma SC-sublist2[rule-format]:
  singleCombinators xs → singleCombinators ([y←xs. P y])
  by (auto intro: SCSsubset)

lemma all-in-list-sublist2[rule-format]:
  singleCombinators xs → all-in-list xs l → all-in-list ( [y←xs. P y]) l
  by (auto intro: all-in-listSubset SC-sublist2)

lemma all-in-listAppend[rule-format]:
  all-in-list (xs) l → all-in-list (ys) l → all-in-list (xs @ ys) l
  by (induct xs) simp-all

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

lemma singleCombinatorsAppend[rule-format]:
  singleCombinators (xs) → singleCombinators (ys) → singleCombinators (xs @ ys)
  apply (induct xs, auto)
  subgoal for a as

```

```

apply (case-tac a,simp-all)
done
subgoal for a as
apply (case-tac a,simp-all)
done
done

lemma sorted-append1[rule-format]:
all-in-list xs l  $\rightarrow$  singleCombinators xs  $\rightarrow$ 
all-in-list ys l  $\rightarrow$  singleCombinators ys  $\rightarrow$ 
(sorted (xs@ys) l  $\rightarrow$ 
(sorted xs l & sorted ys l & ( $\forall x \in set xs. \forall y \in set ys. smaller x y l$ ))
apply(induct xs)
apply(simp-all)
by (metis NormalisationGenericProofs.sorted-Cons all-in-list.simps(2))
all-in-listAppend aux0-1
aux0-4 singleCombinatorsAppend singleCombinatorsConc singleCombinatorsStartlemma sorted-append2[rule-format]:
all-in-list xs l  $\rightarrow$  singleCombinators xs  $\rightarrow$ 
all-in-list ys l  $\rightarrow$  singleCombinators ys  $\rightarrow$ 
(sorted xs l & sorted ys l & ( $\forall x \in set xs. \forall y \in set ys. smaller x y l$ ))  $\rightarrow$ 
(sorted (xs@ys) l)
apply (induct xs)
apply simp-all
by (metis NormalisationGenericProofs.sorted-Cons all-in-list.simps(2))
all-in-listAppend aux0-1
aux0-4 singleCombinatorsAppend singleCombinatorsConc singleCombinatorsStart

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

lemma sort-is-sortedQ[rule-format]:
all-in-list p l  $\rightarrow$  singleCombinators p  $\rightarrow$  sorted (qsort p l) l
proof (induct p l rule: qsort.induct) print-cases
case 1 show ?case by simp

```

```

next
case 2 fix  $x:(a,b)$  Combinators fix  $xs:(a,b)$  Combinators list fix  $l$ 
show all-in-list [ $y \leftarrow xs . \neg smaller x y l$ ]  $l \rightarrow$ 
    singleCombinators [ $y \leftarrow xs . \neg smaller x y l$ ]  $\rightarrow$ 
    sorted (qsort [ $y \leftarrow xs . \neg smaller x y l$ ]  $l$ )  $l \Rightarrow$ 
    all-in-list [ $y \leftarrow xs . smaller x y l$ ]  $l \rightarrow$ 
    singleCombinators [ $y \leftarrow xs . smaller x y l$ ]  $\rightarrow$ 
    sorted (qsort [ $y \leftarrow xs . smaller x y l$ ]  $l$ )  $l \Rightarrow$ 
    all-in-list( $x \# xs$ )  $l \rightarrow$  singleCombinators( $x \# xs$ )  $\rightarrow$  sorted (qsort( $x \# xs$ )
 $l$ )  $l$ 
apply (intro impI)
apply (simp-all add: SC-sublist all-in-list-sublist all-in-list-sublist2
    singleCombinatorsConc SC-sublist2)
proof (subst sorted-append)
show in-list  $x l \wedge$  all-in-list  $xs l \Rightarrow$ 
    singleCombinators ( $x \# xs$ )  $\Rightarrow$ 
    all-in-list (qsort [ $y \leftarrow xs . \neg smaller x y l$ ]  $l$ )  $l$ 
by (metis all-in-list-sublist singleCombinatorsConc)
next
show in-list  $x l \wedge$  all-in-list  $xs l \Rightarrow$ 
    singleCombinators ( $x \# xs$ )  $\Rightarrow$ 
    singleCombinators (qsort [ $y \leftarrow xs . \neg smaller x y l$ ]  $l$ )
apply (auto simp: SC-sublist all-in-list-sublist SC-sublist2
    all-in-list-sublist2 sorted-Cons sorted-append not-le)
apply (metis SC3Q SC-sublist2 singleCombinatorsConc)
done
next
show sorted (qsort [ $y \leftarrow xs . \neg smaller x y l$ ]  $l$ )  $l \Rightarrow$ 
    sorted (qsort [ $y \leftarrow xs . smaller x y l$ ]  $l$ )  $l \Rightarrow$ 
    in-list  $x l \wedge$  all-in-list  $xs l \Rightarrow$  singleCombinators ( $x \# xs$ )  $\Rightarrow$ 
    all-in-list ( $x \# qsort [y \leftarrow xs . smaller x y l]$ )  $l$ 
using all-in-list.simps(2) all-in-list-sublist singleCombinatorsConc by blast
next
show sorted (qsort [ $y \leftarrow xs . smaller x y l$ ]  $l$ )  $l \Rightarrow$ 
    in-list  $x l \wedge$  all-in-list  $xs l \Rightarrow$  singleCombinators ( $x \# xs$ )  $\Rightarrow$ 
    singleCombinators ( $x \# qsort [y \leftarrow xs . smaller x y l]$ )  $l$ 
using SC-sublist aux0-1 singleCombinatorsConc singleCombinatorsStart by blast
next
show sorted (qsort [ $y \leftarrow xs . \neg smaller x y l$ ]  $l$ )  $l \Rightarrow$ 
    sorted (qsort [ $y \leftarrow xs . smaller x y l$ ]  $l$ )  $l \Rightarrow$ 
    in-list  $x l \wedge$  all-in-list  $xs l \Rightarrow$ 
    singleCombinators ( $x \# xs$ )  $\Rightarrow$ 
    FWNormalisationCore.sorted (qsort [ $y \leftarrow xs . \neg smaller x y l$ ]  $l$ )  $l \wedge$ 
    FWNormalisationCore.sorted ( $x \# qsort [y \leftarrow xs . smaller x y l]$ )  $l \wedge$ 

```

```

 $(\forall x' \in set (qsort [y \leftarrow xs . \neg smaller x y l] l).$ 
 $\forall y \in set (x \# qsort [y \leftarrow xs . smaller x y l] l). smaller x' y l)$ 
apply(auto)[1]
apply (metis (mono-tags, lifting) SC-sublist all-in-list.simps(2)
  all-in-list-sublist aux0-1 mem-Collect-eq set-filter set-qsort
  singleCombinatorsConc singleCombinatorsStart sorted-Consb)
apply (metis aux0-0 aux0-6 in-set-in-list singleCombinatorsConc
  singleCombinatorsStart smaller-antisym)
by (metis (no-types, lifting) NormalisationGenericProofs.order-trans aux0-0
  aux0-6 in-set-in-list
  singleCombinatorsConc singleCombinatorsStart smaller-antisym)

qed
qed

```

```

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

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

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

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

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

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

```

```

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

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

lemma distinct-tl[rule-format]: distinct p → distinct (tl p)
  by (induct p, simp-all)

lemma SC-tl[rule-format]: singleCombinators (p) → singleCombinators (tl p)
  by (induct p, simp-all) (auto intro: singleCombinatorsConc)

lemma Conc-not-MT: p = x#xs ⇒ p ≠ []
  by auto

lemma wp1-tl[rule-format]:
  p ≠ [] ∧ wellformed-policy1 p → wellformed-policy1 (tl p)
  by (induct p) (auto intro: waux2)

lemma wp1-eq[rule-format]:
  wellformed-policy1-strong p ⇒ wellformed-policy1 p
  apply (case-tac DenyAll ∈ set p)
  apply (subst wellformed-eq)
  apply (auto elim: waux2)
  done

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

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

lemma RS2-NMT[rule-format]: p ≠ [] → removeShadowRules2 p ≠ []
  apply (induct p, simp-all)
  subgoal for a p
    apply (case-tac p ≠ [], simp-all)
    apply (case-tac a, simp-all) +
  done

```

done

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

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

lemma *wp1ID*: *wellformed-policy1-strong* (*insertDeny* (*removeShadowRules1* p))
apply (induct p , simp-all)
subgoal for $a\ p$
apply (case-tac a , simp-all)
done
done

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

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

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

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

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

```

lemma WP1rd[rule-format,simp]:
  wellformed-policy1-strong p → wellformed-policy1-strong (remdups p)
  by (induct p, simp-all)

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

lemma ANDiD[rule-format,simp]:
  allNetsDistinct p → allNetsDistinct (insertDeny p)
  apply (induct p, simp-all)
  apply (simp add: allNetsDistinct-def)
  apply (auto intro: ANDConc)
  subgoal for a p
    apply (case-tac a, simp-all add: allNetsDistinct-def)
    done
  done

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

lemma WP1iD[rule-format,simp]: wellformed-policy1-strong p →
  wellformed-policy1-strong (insertDeny p)
  by (induct p, simp-all)

lemma DAiniD: DenyAll ∈ set (insertDeny p)
  apply (induct p, simp-all)
  subgoal for a p
    apply (case-tac a, simp-all)
    done
  done

lemma p2lNmt: policy2list p ≠ []
  by (rule policy2list.induct, simp-all)

lemma AIL2[rule-format,simp]:
  all-in-list p l → all-in-list (removeShadowRules2 p) l
  apply (induct-tac p, simp-all)
  subgoal for a p

```

```

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

lemma SCConc: singleCombinators x  $\Rightarrow$  singleCombinators y  $\Rightarrow$  singleCombinators (x@y)
apply (rule aux0-5)
apply (metis aux0-0 aux0-4)
done

lemma SCp2l: singleCombinators (policy2list p)
by (induct-tac p) (auto intro: SCConc)

lemma subnetAux: Dd  $\cap$  A  $\neq \{\}$   $\Rightarrow$  A  $\subseteq$  B  $\Rightarrow$  Dd  $\cap$  B  $\neq \{\}$ 
by auto

lemma soadisj: x  $\in$  subnetsOfAdr a  $\Rightarrow$  y  $\in$  subnetsOfAdr a  $\Rightarrow$   $\neg$  netsDistinct x y
by (simp add: subnetsOfAdr-def netsDistinct-def, auto)

lemma not-member:  $\neg$  member a (x⊕y)  $\Rightarrow$   $\neg$  member a x
by auto

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

lemma ndFalse1:
 $(\forall a b c d. (a,b) \in A \wedge (c,d) \in B \longrightarrow \text{netsDistinct } a c) \Rightarrow$ 
 $\exists (a, b) \in A. a \in \text{subnetsOfAdr } D \Rightarrow$ 
 $\exists (a, b) \in B. a \in \text{subnetsOfAdr } D \Rightarrow \text{False}$ 
apply (auto simp: soadisj)
using soadisj2 by blast

lemma ndFalse2:  $(\forall a b c d. (a,b) \in A \wedge (c,d) \in B \longrightarrow \text{netsDistinct } b d) \Rightarrow$ 
 $\exists (a, b) \in A. b \in \text{subnetsOfAdr } D \Rightarrow$ 
 $\exists (a, b) \in B. b \in \text{subnetsOfAdr } D \Rightarrow \text{False}$ 
apply (auto simp: soadisj)
using soadisj2 by blast

lemma tndFalse:  $(\forall a b c d. (a,b) \in A \wedge (c,d) \in B \longrightarrow \text{twoNetsDistinct } a b c d) \Rightarrow$ 
 $\exists (a, b) \in A. a \in \text{subnetsOfAdr } (D::('a::adr)) \wedge b \in \text{subnetsOfAdr } (F::'a) \Rightarrow$ 
 $\exists (a, b) \in B. a \in \text{subnetsOfAdr } D \wedge b \in \text{subnetsOfAdr } F$ 
 $\Rightarrow \text{False}$ 
apply (simp add: twoNetsDistinct-def)

```

```

apply (auto simp: ndFalse1 ndFalse2)
apply (metis soadisj)
done

lemma sepnMT[rule-format]:  $p \neq [] \rightarrow (\text{separate } p) \neq []$ 
  by (induct p rule: separate.induct) simp-all

lemma sepDA[rule-format]:  $\text{DenyAll} \notin \text{set } p \rightarrow \text{DenyAll} \notin \text{set } (\text{separate } p)$ 
  by (induct p rule: separate.induct) simp-all

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

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

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

lemma OTNoTN[rule-format]:  $\text{OnlyTwoNets } p \rightarrow x \neq \text{DenyAll} \rightarrow x \in \text{set } p \rightarrow$ 
 $\text{onlyTwoNets } x$ 
  apply (induct p, simp-all, rename-tac a p)
  apply (intro impI conjI, simp)
  subgoal for a p
    apply(case-tac a, simp-all)
    done
  subgoal for a p
    apply(case-tac a, simp-all)
    done
  done

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

lemma sdnets2:
   $\exists a b. \text{sdnets } x = \{(a, b), (b, a)\} \Rightarrow \neg \text{member DenyAll } x \Rightarrow$ 
   $\text{sdnets } x = \{(\text{first-srcNet } x, \text{first-destNet } x), (\text{first-destNet } x, \text{first-srcNet } x)\}$ 
proof -
  have * :  $\exists a b. \text{sdnets } x = \{(a, b), (b, a)\} \Rightarrow \neg \text{member DenyAll } x$ 

```

```

 $\implies (\text{first-srcNet } x, \text{first-destNet } x) \in \text{sdnets } x$ 
by (erule first-isIn)
show  $\exists a b. \text{sdnets } x = \{(a, b), (b, a)\} \implies \neg \text{member DenyAll } x \implies$ 
 $\text{sdnets } x = \{(\text{first-srcNet } x, \text{first-destNet } x), (\text{first-destNet } x, \text{first-srcNet } x)\}$ 
using * by auto
qed

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

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

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

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

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

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

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

```

```

lemma disjComm:  $\text{disjSD-2 } a \ b \implies \text{disjSD-2 } b \ a$ 
  apply (simp add: disjSD-2-def)
  apply (intro allI impI conjI)
  using tNDComm apply blast
  by (meson tNDComm twoNetsDistinct-def)

lemma disjSD2aux:
 $\text{disjSD-2 } a \ b \implies \neg \text{member DenyAll } a \implies \neg \text{member DenyAll } b \implies$ 
 $\text{disjSD-2 } (\text{DenyAllFromTo } (\text{first-srcNet } a) (\text{first-destNet } a) \oplus$ 
 $\quad \text{DenyAllFromTo } (\text{first-destNet } a) (\text{first-srcNet } a) \oplus a)$ 
 $\quad b$ 
  apply (drule disjComm,rule disjComm)
  apply (simp add: disjSD-2-def)
  using first-isIn by blast

lemma noDA1eq[rule-format]:  $\text{noDenyAll } p \longrightarrow \text{noDenyAll1 } p$ 
  apply (induct p, simp,rename-tac a p)
  subgoal for a p
    apply(case-tac a, simp-all)
    done
  done

lemma noDA1C[rule-format]:  $\text{noDenyAll1 } (a \# p) \longrightarrow \text{noDenyAll1 } p$ 
  by (case-tac a, simp-all,rule impI, rule noDA1eq, simp)+

lemma disjSD-2IDA:
 $\text{disjSD-2 } x \ y \implies$ 
 $\neg \text{member DenyAll } x \implies$ 
 $\neg \text{member DenyAll } y \implies$ 
 $a = \text{first-srcNet } x \implies$ 
 $b = \text{first-destNet } x \implies$ 
 $\text{disjSD-2 } (\text{DenyAllFromTo } a \ b \oplus \text{DenyAllFromTo } b \ a \oplus x) \ y$ 
  by(simp add:disjSD2aux)

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

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

```

```

( $\text{DenyAllFromTo}(\text{first-destNet } a) (\text{first-srcNet } a)) \oplus a \wedge a \in \text{set } p$ )
apply (induct p, simp, rename-tac a p)
subgoal for a p
  apply (case-tac a = DenyAll, simp)
  apply (case-tac a, auto)
  done
done

lemma id-aux1[rule-format]:  $\text{DenyAllFromTo}(\text{first-srcNet } s) (\text{first-destNet } s) \oplus$ 
 $\text{DenyAllFromTo}(\text{first-destNet } s) (\text{first-srcNet } s) \oplus s \in \text{set}(\text{insertDenies } p)$ 
 $\longrightarrow s \in \text{set } p$ 
apply (induct p, simp-all, rename-tac a p)
subgoal for a p
  apply (case-tac a, simp-all)
  done
done

lemma id-aux2:
 $\text{noDenyAll } p \implies$ 
 $\forall s. s \in \text{set } p \implies \text{disjSD-2 } a s \implies$ 
 $\neg \text{member DenyAll } a \implies$ 
 $\text{DenyAllFromTo}(\text{first-srcNet } s) (\text{first-destNet } s) \oplus$ 
 $\text{DenyAllFromTo}(\text{first-destNet } s) (\text{first-srcNet } s) \oplus s \in \text{set}(\text{insertDenies } p) \implies$ 
 $\text{disjSD-2 } a (\text{DenyAllFromTo}(\text{first-srcNet } s) (\text{first-destNet } s) \oplus$ 
 $\text{DenyAllFromTo}(\text{first-destNet } s) (\text{first-srcNet } s) \oplus s)$ 
by (metis disjComm disjSD2aux isInIDo noDA)

```

```

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

```

```

lemma sepNetsID[rule-format]:
 $\text{noDenyAll1 } p \longrightarrow \text{separated } p \longrightarrow \text{separated}(\text{insertDenies } p)$ 
apply (induct p, simp)
apply (rename-tac a p, auto)
using noDA1C apply blast
subgoal for a p

```

```

apply (case-tac a = DenyAll, auto)
  apply (simp add: disjSD-2-def)
  apply (case-tac a,auto)
    apply (rule disjSD-2IDA, simp-all, rule id-aux4, simp-all, metis noDA noDAID) +
  done
done

lemma aNDDA[rule-format]: allNetsDistinct p  $\rightarrow$  allNetsDistinct(DenyAll#p)
  by (case-tac p,auto simp: allNetsDistinct-def)

lemma OTNConc[rule-format]: OnlyTwoNets (y # z)  $\rightarrow$  OnlyTwoNets z
  by (case-tac y, simp-all)

lemma first-bothNetsd:  $\neg$  member DenyAll x  $\Rightarrow$  first-bothNet x = {first-srcNet x, first-destNet x}
  by (induct x) simp-all

lemma bNaux:
   $\neg$  member DenyAll x  $\Rightarrow$   $\neg$  member DenyAll y  $\Rightarrow$ 
  first-bothNet x = first-bothNet y  $\Rightarrow$ 
  {first-srcNet x, first-destNet x} = {first-srcNet y, first-destNet y}
  by (simp add: first-bothNetsd)

lemma setPair: {a,b} = {a,d}  $\Rightarrow$  b = d
  by (metis setPaireq)

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

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

lemma otnaux1: {x, y, x, y} = {x,y}
  by auto

lemma OTNIDaux4: {x,y,x} = {y,x}
  by auto

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

lemma otnaux:
   $\llbracket$  first-bothNet x = first-bothNet y;  $\neg$  member DenyAll x;  $\neg$  member DenyAll y;
  onlyTwoNets y; onlyTwoNets x  $\rrbracket$   $\Rightarrow$ 

```

```

onlyTwoNets ( $x \oplus y$ )
apply (simp add: onlyTwoNets-def)
apply (subgoal-tac {first-srcNet x, first-destNet x} =
         {first-srcNet y, first-destNet y})
apply (case-tac ( $\exists a b. sdnets y = \{(a, b)\}$ ))
apply simp-all
apply (case-tac ( $\exists a b. sdnets x = \{(a, b)\}$ ))
apply simp-all
apply (subgoal-tac sdnets x = {(first-srcNet x, first-destNet x)})
apply (subgoal-tac sdnets y = {(first-srcNet y, first-destNet y)})
apply simp
apply (case-tac first-srcNet x = first-srcNet y)
apply simp-all
apply (rule disjI1)
apply (rule setPair)
apply simp
apply (subgoal-tac first-srcNet x = first-destNet y)
apply simp
apply (subgoal-tac first-destNet x = first-srcNet y)
apply simp
apply (rule-tac x =first-srcNet y in exI, rule-tac x =first-destNet y in exI,simp)
apply (rule setPair1)
apply simp
apply (rule setPair4)
apply simp-all
apply (metis first-isIn singletonE)
apply (metis first-isIn singletonE)
apply (subgoal-tac sdnets x = {(first-srcNet x, first-destNet x),
         (first-destNet x, first-srcNet x)})
apply (subgoal-tac sdnets y = {(first-srcNet y, first-destNet y)})
apply simp
apply (case-tac first-srcNet x = first-srcNet y)
apply simp-all
apply (subgoal-tac first-destNet x = first-destNet y)
apply simp
apply (rule setPair)
apply simp
apply (subgoal-tac first-srcNet x = first-destNet y)
apply simp
apply (subgoal-tac first-destNet x = first-srcNet y)
apply simp
apply (rule-tac x =first-srcNet y in exI, rule-tac x =first-destNet y in exI)
apply (metis OTNIDaux4 insert-commute)
apply (rule setPair1)

```

```

apply simp
apply (rule setPair5)
apply assumption
apply simp
apply (metis first-isIn singletonE)
apply (rule sdnets2)
apply simp-all
apply (case-tac ( $\exists a\ b.\ sdnets\ x = \{(a,\ b)\}$ ))
apply simp-all
apply (subgoal-tac sdnets  $x = \{(first-srcNet\ x,\ first-destNet\ x)\}$ )
apply (subgoal-tac sdnets  $y = \{(first-srcNet\ y,\ first-destNet\ y),$ 
 $(first-destNet\ y,\ first-srcNet\ y)\}$ )
apply simp
apply (case-tac first-srcNet  $x = first-srcNet\ y$ )
apply simp-all
apply (subgoal-tac first-destNet  $x = first-destNet\ y$ )
apply simp
apply (rule-tac  $x = first-srcNet\ y$  in exI, rule-tac  $x = first-destNet\ y$  in exI)
apply (metis OTNIDaux4 insert-commute)
apply (rule setPair)
apply simp
apply (subgoal-tac first-srcNet  $x = first-destNet\ y$ )
apply simp
apply (subgoal-tac first-destNet  $x = first-srcNet\ y$ )
apply simp
apply (rule setPair1)
apply simp
apply (rule setPair4)
apply assumption
apply simp
apply (rule sdnets2)
apply simp
apply simp
apply (metis singletonE first-isIn)
apply (subgoal-tac sdnets  $x = \{(first-srcNet\ x,\ first-destNet\ x),$ 
 $(first-destNet\ x,\ first-srcNet\ x)\}$ )
apply (subgoal-tac sdnets  $y = \{(first-srcNet\ y,\ first-destNet\ y),$ 
 $(first-destNet\ y,\ first-srcNet\ y)\}$ )
apply simp
apply (case-tac first-srcNet  $x = first-srcNet\ y$ )
apply simp-all
apply (subgoal-tac first-destNet  $x = first-destNet\ y$ )
apply simp
apply (rule-tac  $x = first-srcNet\ y$  in exI, rule-tac  $x = first-destNet\ y$  in exI)

```

```

apply (rule otiaux1)
apply (rule setPair)
apply simp
apply (subgoal-tac first-srcNet x = first-destNet y)
apply simp
apply (subgoal-tac first-destNet x = first-srcNet y)
apply simp
apply (rule-tac x =first-srcNet y in exI, rule-tac x = first-destNet y in exI)
apply (metis OTNIDaux4 insert-commute)
apply (rule setPair1)
apply simp
apply (rule setPair4)
apply assumption
apply simp
apply (rule sdnets2,simp-all)+
apply (rule bNaux, simp-all)
done

```

lemma OTNSepaux:

```

onlyTwoNets (a ⊕ y) ∧ OnlyTwoNets z → OnlyTwoNets (separate (a ⊕ y # z)) ⇒
¬ member DenyAll a ⇒ ¬ member DenyAll y ⇒
noDenyAll z ⇒ onlyTwoNets a ⇒ OnlyTwoNets (y # z) ⇒ first-bothNet a =
first-bothNet y ⇒
OnlyTwoNets (separate (a ⊕ y # z))
apply (drule mp)
apply simp-all
apply (rule conjI)
apply (rule otiaux)
apply simp-all
apply (rule-tac p = (y # z) in OTNoTN)
apply simp-all
apply (metis member.simps(2))
apply (simp add: onlyTwoNets-def)
apply (rule-tac y = y in OTNCconc,simp)
done

```

lemma OTNSEp[rule-format]:

```

noDenyAll1 p → OnlyTwoNets p → OnlyTwoNets (separate p)
apply (induct p rule: separate.induct)
by (simp-all add: OTNSepaux noDA1eq)

```

lemma nda[rule-format]:

```

singleCombinators (a#p) → noDenyAll p → noDenyAll1 (a # p)
apply (induct p,simp-all)

```

```

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

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

lemma nDAeqSet: set p = set s  $\implies$  noDenyAll p = noDenyAll s
  by (simp add: nDAcharn)

lemma nDASCaux[rule-format]:
  DenyAll  $\notin$  set p  $\longrightarrow$  singleCombinators p  $\longrightarrow$  r  $\in$  set p  $\longrightarrow$   $\neg member DenyAll r$ 
  apply (case-tac r, simp-all)
  using SCnotConc by blast

lemma nDASC[rule-format]:
  wellformed-policy1 p  $\longrightarrow$  singleCombinators p  $\longrightarrow$  noDenyAll1 p
  apply (induct p, simp-all)
  using nDASCaux nDAcharn nda singleCombinatorsConc by blast

lemma noDAAll[rule-format]: noDenyAll p = ( $\neg memberP DenyAll p$ )
  by (induct p) simp-all

lemma memberPsep[symmetric]: memberP x p = memberP x (separate p)
  by (induct p rule: separate.induct) simp-all

lemma noDAssep[rule-format]: noDenyAll p  $\implies$  noDenyAll (separate p)
  by (simp add: noDAAll, subst memberPsep, simp)

lemma noDA1sep[rule-format]: noDenyAll1 p  $\longrightarrow$  noDenyAll1 (separate p)
  by (induct p rule: separate.induct, simp-all add: noDAssep)

lemma isInAlternativeLista:
  (aa  $\in$  set (net-list-aux [a]))  $\implies$  aa  $\in$  set (net-list-aux (a # p))
  by (case-tac a, auto)

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

lemma ANDSepaux: allNetsDistinct (x # y # z)  $\implies$  allNetsDistinct (x  $\oplus$  y # z)
  apply (simp add: allNetsDistinct-def)
  apply (intro allI impI, rename-tac a b)
  subgoal for a b

```

```

apply (drule-tac  $x = a$  in spec, drule-tac  $x = b$  in spec)
by (meson isInAlternativeList)
done

lemma netlistalternativeSeparateaux:
  net-list-aux [y] @ net-list-aux z = net-list-aux (y # z)
  by (case-tac y, simp-all)

lemma netlistalternativeSeparate: net-list-aux p = net-list-aux (separate p)
  by (induct p rule:separate.induct, simp-all) (simp-all add: netlistalternativeSeparateaux)

lemma ANDSepaux2:
  allNetsDistinct(x#y#z)  $\implies$  allNetsDistinct(separate(y#z))  $\implies$  allNetsDistinct(x#separate(y#z))
  apply (simp add: allNetsDistinct-def)
  by (metis isInAlternativeList netlistalternativeSeparate netlistaux)

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

lemma wp1-alternativesep[rule-format]:
  wellformed-policy1-strong p  $\longrightarrow$  wellformed-policy1-strong (separate p)
  by (metis sepDA separate.simps(1) wellformed-policy1-strong.simps(2) wp1n-tl)

lemma noDAsort[rule-format]: noDenyAll1 p  $\longrightarrow$  noDenyAll1 (sort p l)
  apply (case-tac p,simp-all)
  subgoal for a as
    apply (case-tac a = DenyAll, auto)
    using NormalisationGenericProofs.set-sort nDAeqSet apply blast
  proof -
    fix a::('a,'b)Combinators fix list
    have * : a  $\neq$  DenyAll  $\implies$  noDenyAll1 (a # list)  $\implies$  noDenyAll (a#list) by
    (case-tac a, simp-all)
    show a  $\neq$  DenyAll  $\implies$  noDenyAll1 (a # list)  $\implies$  noDenyAll1 (insort a (sort list
    l) l)
    apply(cases insort a (sort list l) l, simp-all)
    by (metis * NormalisationGenericProofs.set-insort NormalisationGener-

```

```

icProofs.set-sort
list.simps(15) nDAeqSet noDA1eq
qed
done

lemma OTNSC[rule-format]: singleCombinators p —> OnlyTwoNets p
apply (induct p,simp-all)
apply (rename-tac a p)
apply (rule impI,drule mp)
apply (erule singleCombinatorsConc)
subgoal for a b
  apply (case-tac a, simp-all)
  apply (simp add: onlyTwoNets-def)+
done
done

lemma fMTaux: ¬ member DenyAll x ==> first-bothNet x ≠ {}
by (metis first-bothNetsd insert-commute insert-not-empty)

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

lemma fl3[rule-format]: NetsCollected p —> (first-bothNet x ≠ firstList p —>
(∀ a∈set p. first-bothNet x ≠ first-bothNet a))—> NetsCollected (x#p)
by (induct p) simp-all

lemma sortedConc[rule-format]: sorted (a # p) l —> sorted p l
by (induct p) simp-all

lemma smalleraux2:
{a,b} ∈ set l ==> {c,d} ∈ set l ==> {a,b} ≠ {c,d} ==>
smaller (DenyAllFromTo a b) (DenyAllFromTo c d) l ==>
¬ smaller (DenyAllFromTo c d) (DenyAllFromTo a b) l
by (metis bothNet.simps(2) pos-noteq smaller.simps(5))

lemma smalleraux2a:
{a,b} ∈ set l ==> {c,d} ∈ set l ==> {a,b} ≠ {c,d} ==>
smaller (DenyAllFromTo a b) (AllowPortFromTo c d p) l ==>
¬ smaller (AllowPortFromTo c d p) (DenyAllFromTo a b) l
by (simp) (metis eq-imp-le pos-noteq)

lemma smalleraux2b:
{a,b} ∈ set l ==> {c,d} ∈ set l ==> {a,b} ≠ {c,d} ==> y = DenyAllFromTo a b ==>

```

```

smaller (AllowPortFromTo c d p) y l  $\implies$ 
 $\neg$  smaller y (AllowPortFromTo c d p) l
by (simp) (metis eq-imp-le pos-noteq)

lemma smalleraux2c:
{a,b} ∈ set l  $\implies$  {c,d} ∈ set l  $\implies$  {a,b} ≠ {c,d}  $\implies$  y = AllowPortFromTo a b q  $\implies$ 
smaller (AllowPortFromTo c d p) y l  $\implies$   $\neg$  smaller y (AllowPortFromTo c d p) l
by (simp) (metis pos-noteq)

lemma smalleraux3:
assumes x ∈ set l and y ∈ set l and x ≠ y and x = bothNet a and y = bothNet b
and smaller a b l and singleCombinators [a] and singleCombinators [b]
shows  $\neg$  smaller b a l
proof (cases a)
  case DenyAll thus ?thesis using assms by (case-tac b,simp-all)
next
  case (DenyAllFromTo c d) thus ?thesis
  proof (cases b)
    case DenyAll thus ?thesis using assms DenyAll DenyAllFromTo by simp
  next
    case (DenyAllFromTo e f) thus ?thesis using assms DenyAllFromTo
      by (metis DenyAllFromTo ⟨a = DenyAllFromTo c d⟩ bothNet.simps(2) smaller-
aux2)
    next
      case (AllowPortFromTo e f g) thus ?thesis
        using assms DenyAllFromTo AllowPortFromTo by simp (metis eq-imp-le pos-noteq)
    next
      case (Conc e f) thus ?thesis using assms by simp
    qed
  next
    case (AllowPortFromTo c d p) thus ?thesis
    proof (cases b)
      case DenyAll thus ?thesis using assms AllowPortFromTo DenyAll by simp
    next
      case (DenyAllFromTo e f) thus ?thesis
        using assms by simp (metis AllowPortFromTo DenyAllFromTo bothNet.simps(3)
smalleraux2a)
    next
      case (AllowPortFromTo e f g) thus ?thesis
        using assms by(simp)(metis AllowPortFromTo ⟨a = AllowPortFromTo c d p⟩
bothNet.simps(3) smalleraux2c)
    next
      case (Conc e f) thus ?thesis using assms by simp
    qed

```

```

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

lemma smalleraux3a:
   $a \neq \text{DenyAll} \implies b \neq \text{DenyAll} \implies \text{in-list } b \ l \implies \text{in-list } a \ l \implies$ 
   $\text{bothNet } a \neq \text{bothNet } b \implies \text{smaller } a \ b \ l \implies \text{singleCombinators } [a] \implies$ 
   $\text{singleCombinators } [b] \implies \neg \text{smaller } b \ a \ l$ 
  apply (rule smalleraux3,simp-all)
  apply (case-tac a, simp-all)
  apply (case-tac b, simp-all)
done

lemma posaux[rule-format]: position a l < position b l  $\longrightarrow$   $a \neq b$ 
by (induct l, simp-all)

lemma posaux6[rule-format]:
   $a \in \text{set } l \longrightarrow b \in \text{set } l \longrightarrow a \neq b \longrightarrow \text{position } a \ l \neq \text{position } b \ l$ 
by (induct l) (simp-all add: position-positive)

lemma notSmallerTransaux[rule-format]:
   $x \neq \text{DenyAll} \implies r \neq \text{DenyAll} \implies$ 
   $\text{singleCombinators } [x] \implies \text{singleCombinators } [y] \implies \text{singleCombinators } [r] \implies$ 
   $\neg \text{smaller } y \ x \ l \implies \text{smaller } x \ y \ l \implies \text{smaller } x \ r \ l \implies \text{smaller } y \ r \ l \implies$ 
   $\text{in-list } x \ l \implies \text{in-list } y \ l \implies \text{in-list } r \ l \implies \neg \text{smaller } r \ x \ l$ 
by (metis order-trans)

lemma notSmallerTrans[rule-format]:
   $x \neq \text{DenyAll} \longrightarrow r \neq \text{DenyAll} \longrightarrow \text{singleCombinators } (x \# y \# z) \longrightarrow$ 
   $\neg \text{smaller } y \ x \ l \longrightarrow \text{sorted } (x \# y \# z) \ l \longrightarrow r \in \text{set } z \longrightarrow$ 
   $\text{all-in-list } (x \# y \# z) \ l \longrightarrow \neg \text{smaller } r \ x \ l$ 
  apply (rule impI)+
  apply (rule notSmallerTransaux, simp-all)
    apply (metis singleCombinatorsConc singleCombinatorsStart)
    apply (metis SCSsubset equalityE remdups.simps(2) set-remdups
            singleCombinatorsConc singleCombinatorsStart)
    apply metis
    apply (metis sorted.simps(3) in-set-in-list singleCombinatorsConc
            singleCombinatorsStart sortedConcStart sorted-is-smaller)
  apply (metis sorted-Cons all-in-list.simps(2)
          singleCombinatorsConc)
  apply (metis,metis in-set-in-list)
done

```

```

lemma NCSaux1[rule-format]:
  noDenyAll p → {x, y} ∈ set l → all-in-list p l → singleCombinators p →
  sorted (DenyAllFromTo x y # p) l → {x, y} ≠ firstList p →
  DenyAllFromTo u v ∈ set p → {x, y} ≠ {u, v}
proof (cases p)
  case Nil thus ?thesis by simp
next
  case (Cons a list)
  then show ?thesis apply simp
    apply (intro impI conjI)
    apply (metis bothNet.simps(2) first-bothNet.simps(3))
  proof -
    assume 1: {x, y} ∈ set l and 2: in-list a l ∧ all-in-list list l
    and 3 : singleCombinators (a # list)
    and 4 : smaller (DenyAllFromTo x y) a l ∧ sorted (a # list) l
    and 5 : DenyAllFromTo u v ∈ set list
    and 6 : ¬ member DenyAll a ∧ noDenyAll list
    have * : smaller ((DenyAllFromTo x y)::((a,b)Combinators)) (DenyAllFromTo u
v) l
      apply (insert 1 2 3 4 5, rule-tac y = a in order-trans, simp-all)
      using in-set-in-list apply fastforce
      by (simp add: sorted-ConsA)

    have ** :{x, y} ≠ first-bothNet a ==>
      ¬ smaller ((DenyAllFromTo u v)::(a, b) Combinators) (DenyAllFromTo
x y) l
      apply (insert 1 2 3 4 5 6,
        rule-tac y = a and z = list in notSmallerTrans,
        simp-all del: smaller.simps)
      apply (rule smallerAux3a,simp-all del: smaller.simps)
      apply (case-tac a, simp-all del: smaller.simps)
      by (metis aux0-0 first-bothNet.elims list.set-intros(1))
    show {x, y} ≠ first-bothNet a ==> {x, y} ≠ {u, v}
      using 3 ** by force
  qed
qed
qed

lemma posaux3[rule-format]:a ∈ set l → b ∈ set l → a ≠ b → position a l ≠
position b l
apply (induct l, auto)
by(metis position-positive)+

lemma posaux4[rule-format]:
  singleCombinators [a] → a ≠ DenyAll → b ≠ DenyAll → in-list a l → in-list b l

```

```

→
smaller a b l → x = (bothNet a) → y = (bothNet b) →
position x l <= position y l
proof (cases a)
  case DenyAll then show ?thesis by simp
next
  case (DenyAllFromTo c d) thus ?thesis
  proof (cases b)
    case DenyAll thus ?thesis by simp
  next
    case (DenyAllFromTo e f) thus ?thesis using DenyAllFromTo
      by (auto simp: eq-imp-le` a = DenyAllFromTo c d`)
  next
    case (AllowPortFromTo e f p) thus ?thesis using a = DenyAllFromTo c d by
      simp
  next
    case (Conc e f) thus ?thesis using Conc` a = DenyAllFromTo c d` by simp
  qed
next
  case (AllowPortFromTo c d p) thus ?thesis
  proof (cases b)
    case DenyAll thus ?thesis by simp
  next
    case (DenyAllFromTo e f) thus ?thesis using AllowPortFromTo by simp
  next
    case (AllowPortFromTo e f p2) thus ?thesis using a = AllowPortFromTo c d p
    by simp
  next
    case (Conc e f) thus ?thesis using AllowPortFromTo by simp
  qed
next
  case (Conc c d) thus ?thesis by simp
qed

```

lemma *NCSaux2[rule-format]*:

```

noDenyAll p → {a, b} ∈ set l → all-in-list p l → singleCombinators p →
  sorted (DenyAllFromTo a b # p) l → {a, b} ≠ firstList p →
  AllowPortFromTo u v w ∈ set p → {a, b} ≠ {u, v}
proof (cases p)
  case Nil then show ?thesis by simp
next
  case (Cons aa list)
  have * : {a, b} ∈ set l ⇒ in-list aa l ∧ all-in-list list l ⇒
    singleCombinators (aa # list) ⇒ AllowPortFromTo u v w ∈ set list

```

```

⇒
smaller (DenyAllFromTo a b) aa l ∧ sorted (aa # list) l ⇒
smaller (DenyAllFromTo a b) (AllowPortFromTo u v w) l
apply (rule-tac y = aa in order-trans,simp-all del: smaller.simps)
using in-set-in-list apply fastforce
using NormalisationGenericProofs.sorted-Cons all-in-list.simps(2) by blast

have **: AllowPortFromTo u v w ∈ set list ⇒
in-list aa l ⇒ all-in-list list l ⇒
in-list (AllowPortFromTo u v w) l
apply (rule-tac p = list in in-set-in-list)
apply simp-all
done
assume p = aa # list
then show ?thesis
apply simp
apply (intro impI conjI,hypsubst, simp)
apply (subgoal-tac smaller (DenyAllFromTo a b) (AllowPortFromTo u v w) l)
apply (subgoal-tac ¬ smaller (AllowPortFromTo u v w) (DenyAllFromTo a b) l)
apply (rule-tac l = l in posaux)
apply (rule-tac y = position (first-bothNet aa) l in basic-trans-rules(22))
apply (simp-all split: if-splits)
apply (case-tac aa, simp-all)
subgoal for x x'
apply (case-tac a = x ∧ b = x', simp-all)
apply (case-tac a = x, simp-all)
apply (simp add: order.not-eq-order-implies-strict posaux6)
apply (simp add: order.not-eq-order-implies-strict posaux6)
done
apply (simp add: order.not-eq-order-implies-strict posaux6)
apply (rule basic-trans-rules(18))
apply (rule-tac a = DenyAllFromTo a b and b = aa in posaux4, simp-all)
apply (case-tac aa,simp-all)
apply (case-tac aa, simp-all)
apply (rule posaux3, simp-all)
apply (case-tac aa, simp-all)
apply (rule-tac a = aa and b = AllowPortFromTo u v w in posaux4, simp-all)
apply (case-tac aa,simp-all)
apply (rule-tac p = list in sorted-is-smaller, simp-all)
apply (case-tac aa, simp-all)
apply (case-tac aa, simp-all)
apply (rule-tac a = aa and b = AllowPortFromTo u v w in posaux4, simp-all)
apply (case-tac aa,simp-all)
using ** apply auto[1]

```

```

apply (metis all-in-list.simps(2) sorted-Cons)
apply (case-tac aa, simp-all)
apply (metis ** bothNet.simps(3) in-list.simps(3) posaux6)
using * by force
qed

lemma NCSaux3[rule-format]:
noDenyAll p → {a, b} ∈ set l → all-in-list p l → singleCombinators p →
sorted (AllowPortFromTo a b w # p) l → {a, b} ≠ firstList p →
DenyAllFromTo u v ∈ set p → {a, b} ≠ {u, v}
apply (case-tac p, simp-all,intro impI conjI,hypsubst,simp)

proof -
fix aa:('a, 'b) Combinators fix list:('a, 'b) Combinators list
assume 1 : ¬ member DenyAll aa ∧ noDenyAll list and 2: {a, b} ∈ set l
and 3 : in-list aa l ∧ all-in-list list l and 4: singleCombinators (aa # list)
and 5 : smaller (AllowPortFromTo a b w) aa l ∧ sorted (aa # list) l
and 6 : {a, b} ≠ first-bothNet aa and 7: DenyAllFromTo u v ∈ set list
have *: ¬ smaller (DenyAllFromTo u v) (AllowPortFromTo a b w) l
apply (insert 1 2 3 4 5 6 7, rule-tac y = aa and z = list in notSmallerTrans)
apply (simp-all del: smaller.simps)
apply (rule smalleraux3a,simp-all del: smaller.simps)
apply (case-tac aa, simp-all del: smaller.simps)
apply (case-tac aa, simp-all del: smaller.simps)
done
have **: smaller (AllowPortFromTo a b w) (DenyAllFromTo u v) l
apply (insert 1 2 3 4 5 6 7,rule-tac y = aa in order-trans,simp-all del: smaller.simps)
apply (subgoal-tac in-list (DenyAllFromTo u v) l, simp)
apply (rule-tac p = list in in-set-in-list, simp-all)
apply (rule-tac p = list in sorted-is-smaller,simp-all del: smaller.simps)
apply (subgoal-tac in-list (DenyAllFromTo u v) l, simp)
apply (rule-tac p = list in in-set-in-list, simp-all)
apply (erule singleCombinatorsConc)
done
show      {a, b} ≠ {u, v} by (insert * **, simp split: if-splits)
qed

```

```

lemma NCSaux4[rule-format]:
noDenyAll p → {a, b} ∈ set l → all-in-list p l → singleCombinators p →
sorted (AllowPortFromTo a b c # p) l → {a, b} ≠ firstList p →
AllowPortFromTo u v w ∈ set p → {a, b} ≠ {u, v}
apply (cases p, simp-all)
apply (intro impI conjI)
apply (hypsubst,simp-all)
proof -

```

```

fix aa::('a, 'b) Combinators fix list::('a, 'b) Combinators list
assume 1 :  $\neg$  member DenyAll aa  $\wedge$  noDenyAll list and 2: {a, b}  $\in$  set l
      and 3 : in-list aa l  $\wedge$  all-in-list list l and 4: singleCombinators (aa # list)
      and 5 : smaller (AllowPortFromTo a b c) aa l  $\wedge$  sorted (aa # list) l
      and 6 : {a, b}  $\neq$  first-bothNet aa and 7: AllowPortFromTo u v w  $\in$  set list
have *:  $\neg$  smaller (AllowPortFromTo u v w) (AllowPortFromTo a b c) l
apply (insert 1 2 3 4 5 6 7, rule-tac y = aa and z = list in notSmallerTrans)
      apply (simp-all del: smaller.simps)
apply (rule smalleraux3a,simp-all del: smaller.simps)
      apply (case-tac aa, simp-all del: smaller.simps)
apply (case-tac aa, simp-all del: smaller.simps)
done
have **: smaller (AllowPortFromTo a b c) (AllowPortFromTo u v w) l
apply(insert 1 2 3 4 5 6 7)
apply (case-tac aa, simp-all del: smaller.simps)
apply (rule-tac y = aa in order-trans,simp-all del: smaller.simps)
apply (subgoal-tac in-list (AllowPortFromTo u v w) l, simp)
apply (rule-tac p = list in in-set-in-list, simp)
apply (case-tac aa, simp-all del: smaller.simps)
apply (rule-tac p = list in sorted-is-smaller,simp-all del: smaller.simps)
apply (subgoal-tac in-list (AllowPortFromTo u v w) l, simp)
apply (rule-tac p = list in in-set-in-list, simp, simp)
apply (rule-tac y = aa in order-trans,simp-all del: smaller.simps)
apply (subgoal-tac in-list (AllowPortFromTo u v w) l, simp)
using in-set-in-list apply blast
by (metis all-in-list.simps(2) bothNet.simps(3) in-list.simps(3)
      singleCombinators.simps(5) sorted-ConsA)
show {a, b}  $\neq$  {u, v} by (insert * **, simp-all split: if-splits)
qed

```

```

lemma NetsCollectedSorted[rule-format]:
noDenyAll1 p  $\longrightarrow$  all-in-list p l  $\longrightarrow$  singleCombinators p  $\longrightarrow$  sorted p l  $\longrightarrow$  NetsCollected p
apply (induct p)
apply simp
apply (intro impI,drule mp,erule noDA1C,drule mp,simp)
apply (drule mp,erule singleCombinatorsConc)
apply (drule mp,erule sortedConc)
proof -
fix a:: ('a, 'b) Combinators fix p:: ('a, 'b) Combinators list
assume 1: noDenyAll1 (a # p) and 2:all-in-list (a # p) l
      and 3: singleCombinators (a # p) and 4: sorted (a # p) l and 5: NetsCollected p
show NetsCollected (a # p)

```

```

apply(insert 1 2 3 4 5, rule fl3)
  apply(simp, rename-tac aa)
proof (cases a)
  case DenyAll
  fix aa::('a, 'b) Combinators
  assume 6: aa ∈ set p
  show first-bothNet a ≠ first-bothNet aa
    apply(insert 1 2 3 4 5 6 ‹a = DenyAll›, simp-all)
    using fMTaux noDA by blast
next
  case (DenyAllFromTo x21 x22)
  fix aa::('a, 'b) Combinators
  assume 6: first-bothNet a ≠ firstList p and 7 :aa ∈ set p
  show first-bothNet a ≠ first-bothNet aa
    apply(insert 1 2 3 4 5 6 7 ‹a = DenyAllFromTo x21 x22›)
    apply(case-tac aa, simp-all)
      apply(meson NCSaux1)
      apply(meson NCSaux2)
    using SCnotConc by auto[1]
next
  case (AllowPortFromTo x31 x32 x33)
  fix aa::('a, 'b) Combinators
  assume 6: first-bothNet a ≠ firstList p and 7 :aa ∈ set p
  show first-bothNet a ≠ first-bothNet aa
    apply(insert 1 2 3 4 6 7 ‹a = AllowPortFromTo x31 x32 x33›)
    apply(case-tac aa, simp-all)
      apply(meson NCSaux3)
      apply(meson NCSaux4)
    using SCnotConc by auto
next
  case (Conc x41 x42)
  fix aa::('a, 'b) Combinators
  show first-bothNet a ≠ first-bothNet aa
    by(insert 3 4 ‹a = x41 ⊕ x42›, simp)
qed
qed

lemma NetsCollectedSort: distinct p ==> noDenyAll1 p ==> all-in-list p l ==>
  singleCombinators p ==> NetsCollected (sort p l)
  apply(rule-tac l = l in NetsCollectedSorted, rule noDAsort, simp-all)
    apply(rule-tac b=p in all-in-listSubset)
  by(auto intro: sort-is-sorted)

lemma fBNsep[rule-format]: (∀ a∈set z. {b,c} ≠ first-bothNet a) —>

```

```


$$(\forall a \in set (separate z). \{b,c\} \neq first\text{-}bothNet a)$$

apply (induct z rule: separate.induct, simp)
by (rule impI, simp)+

lemma fBNsep1[rule-format]:  $(\forall a \in set z. first\text{-}bothNet x \neq first\text{-}bothNet a) \rightarrow$ 
 $(\forall a \in set (separate z). first\text{-}bothNet x \neq first\text{-}bothNet a)$ 
apply (induct z rule: separate.induct, simp)
by (rule impI, simp)+

lemma NetsCollectedSepauxa:
 $\{b,c\} \neq firstList z \Rightarrow noDenyAll1 z \Rightarrow \forall a \in set z. \{b,c\} \neq first\text{-}bothNet a \Rightarrow NetsCollected z \Rightarrow$ 
 $NetsCollected (separate z) \Rightarrow \{b, c\} \neq firstList (separate z) \Rightarrow a \in set (separate z) \Rightarrow$ 
 $\{b, c\} \neq first\text{-}bothNet a$ 
by (rule fBNsep) simp-all

lemma NetsCollectedSepaux:
 $first\text{-}bothNet (x::('a,'b)Combinators) \neq first\text{-}bothNet y \Rightarrow \neg member DenyAll y \wedge$ 
 $noDenyAll z \Rightarrow$ 
 $(\forall a \in set z. first\text{-}bothNet x \neq first\text{-}bothNet a) \wedge NetsCollected (y \# z) \Rightarrow$ 
 $NetsCollected (separate (y \# z)) \Rightarrow first\text{-}bothNet x \neq firstList (separate (y \# z))$ 
 $\Rightarrow$ 
 $a \in set (separate (y \# z)) \Rightarrow$ 
 $first\text{-}bothNet (x::('a,'b)Combinators) \neq first\text{-}bothNet (a::('a,'b)Combinators)$ 
by (rule fBNsep1) auto

lemma NetsCollectedSep[rule-format]:
 $noDenyAll1 p \rightarrow NetsCollected p \rightarrow NetsCollected (separate p)$ 
proof (induct p rule: separate.induct, simp-all, goal-cases)
fix x::('a, 'b) Combinators list
case 1 then show ?case
by (metis fMTaux noDA noDA1eq noDAsep)
next
fix v va y fix z::('a, 'b) Combinators list
case 2 then show ?case
apply (intro conjI impI, simp)
apply (metis NetsCollectedSepaux fl3 noDA1eq noDenyAll.simps(1))
by (metis noDA1eq noDenyAll.simps(1))
next
fix v va vb y fix z::('a, 'b) Combinators list
case 3 then show ?case
apply (intro conjI impI)

```

```

apply (metis NetsCollectedSepaux fl3 noDA1eq noDenyAll.simps(1))
by (metis noDA1eq noDenyAll.simps(1))
next
  fix v va y fix z::('a, 'b) Combinators list
  case 4 then show ?case
    by (metis NetsCollectedSepaux fl3 noDA1eq noDenyAll.simps(1))
qed

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)
  apply (case-tac (x = first-srcNet a  $\wedge$  y = first-destNet a),simp-all add: onlyT-
woNets-def)
  apply (case-tac ( $\exists$  aa b. sdnets a = {(aa, b)}), simp-all)
  apply (subgoal-tac sdnets a = {(first-srcNet a,first-destNet a)}, simp-all)
  apply (metis singletonE first-isIn)
  apply (subgoal-tacsdnets a = {(first-srcNet a,first-destNet a),(first-destNet a,
first-srcNet a)})
  by(auto intro!: sdnets2)

lemma sdnets-charn: onlyTwoNets a  $\implies$   $\neg$  member DenyAll a  $\implies$ 
  sdnets a = {(first-srcNet a,first-destNet a)}  $\vee$ 
  sdnets a = {(first-srcNet a, first-destNet a),(first-destNet a, first-srcNet a)}
  apply (case-tac sdnets a = {(first-srcNet a, first-destNet a)}, simp-all add: onlyT-
woNets-def)
  apply (case-tac ( $\exists$  aa b. sdnets a = {(aa, b)}), simp-all)
  apply (metis singletonE first-isIn)
  apply (subgoal-tac sdnets a = {(first-srcNet a,first-destNet a),(first-destNet
a,first-srcNet a)})
  by( auto intro!: sdnets2)

lemma first-bothNet-charn[rule-format]:
   $\neg$  member DenyAll a  $\longrightarrow$  first-bothNet a = {first-srcNet a, first-destNet a}
  by (induct a) simp-all

lemma sdnets-noteq:
  onlyTwoNets a  $\implies$  onlyTwoNets aa  $\implies$  first-bothNet a  $\neq$  first-bothNet aa  $\implies$ 
   $\neg$  member DenyAll a  $\implies$   $\neg$  member DenyAll aa  $\implies$  sdnets a  $\neq$  sdnets aa
  apply (insert sdnets-charn [of a])
  apply (insert sdnets-charn [of aa])
  apply (insert first-bothNet-charn [of a])
  apply (insert first-bothNet-charn [of aa])

```

by(metis OTNaux first-isIn insert-absorb2 insert-commute)

lemma fbn-noteq:

onlyTwoNets a \implies onlyTwoNets aa \implies first-bothNet a \neq first-bothNet aa \implies
 \neg member DenyAll a \implies \neg member DenyAll aa \implies allNetsDistinct [a, aa] \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
apply (insert sdnets-charn [of a])
apply (insert sdnets-charn [of aa])
by (metis first-bothNet-charn)

lemma NCisSD2aux:

assumes 1: onlyTwoNets a **and** 2 : onlyTwoNets aa **and** 3 : first-bothNet a \neq first-bothNet aa
and 4: \neg member DenyAll a **and** 5: \neg member DenyAll aa **and** 6: allNetsDistinct [a, aa]

shows disjSD-2 a aa

apply (insert 1 2 3 4 5 6)

apply (simp add: disjSD-2-def)

apply (intro allI impI)

apply (insert sdnets-charn [of a] sdnets-charn [of aa], simp)

apply (insert sdnets-noteq [of a aa] fbn-noteq [of a aa], simp)

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

proof –

fix ab b c d

assume 7: $\forall ab\ b. ab \neq b \wedge ab \in set(net-list-aux[a,aa]) \wedge b \in set(net-list-aux [a,aa]) \longrightarrow netsDistinct ab\ b$

and 8: $(ab, b) \in sdnets\ a \wedge (c, d) \in sdnets\ aa$

and 9: $sdnets\ a = \{(first-srcNet\ a, first-destNet\ a)\} \vee$

$sdnets\ a = \{(first-srcNet\ a, first-destNet\ a), (first-destNet\ a, first-srcNet\ a)\}$

and 10: $sdnets\ aa = \{(first-srcNet\ aa, first-destNet\ aa)\} \vee$

$sdnets\ aa = \{(first-srcNet\ aa, first-destNet\ aa), (first-destNet\ aa, first-srcNet\ aa)\}$

and 11: $sdnets\ a \neq sdnets\ aa$

and 12: $first-destNet\ a = first-srcNet\ aa \longrightarrow first-srcNet\ a = first-destNet\ aa \longrightarrow first-destNet\ aa \neq first-srcNet\ aa$

show (netsDistinct ab c \vee netsDistinct b d) \wedge (netsDistinct ab d \vee netsDistinct b c)

proof (rule conjI)

show netsDistinct ab c \vee netsDistinct b d

apply(insert 7 8 9 10 11 12)

apply (cases sdnets a = {(first-srcNet a, first-destNet a)})

apply (cases sdnets aa = {(first-srcNet aa, first-destNet aa)}, simp-all)

```

apply (metis 4 5 firstInNeta firstInNet alternativelistconc2)
apply (case-tac (c = first-srcNet aa ∧ d = first-destNet aa), simp-all)
apply (case-tac (first-srcNet a) ≠ (first-srcNet aa), simp-all)
apply (metis 4 5 firstInNeta alternativelistconc2)
apply (subgoal-tac first-destNet a ≠ first-destNet aa)
apply (metis 4 5 firstInNet alternativelistconc2)
apply (metis 3 4 5 first-bothNetsd)
apply (case-tac (first-destNet aa) ≠ (first-srcNet a), simp-all)
apply (metis 4 5 firstInNeta firstInNet alternativelistconc2)
apply (case-tac first-destNet aa ≠ first-destNet a, simp-all)
apply (subgoal-tac first-srcNet aa ≠ first-destNet a)
apply (metis 4 5 firstInNeta firstInNet alternativelistconc2)
apply (metis 3 4 5 first-bothNetsd insert-commute)
apply (metis 5 firstInNeta firstInNet alternativelistconc2)
apply (case-tac (c = first-srcNet aa ∧ d = first-destNet aa), simp-all)
apply (case-tac (ab = first-srcNet a ∧ b = first-destNet a), simp-all)
apply (case-tac (first-srcNet a) ≠ (first-srcNet aa), simp-all)
apply (metis 4 5 firstInNeta alternativelistconc2)
apply (subgoal-tac first-destNet a ≠ first-destNet aa)
apply (metis 4 5 firstInNet alternativelistconc2)
apply (metis 3 4 5 first-bothNetsd )
apply (case-tac (first-destNet aa) ≠ (first-srcNet a), simp-all)
apply (metis 4 5 firstInNeta firstInNet alternativelistconc2)
apply (case-tac first-destNet aa ≠ first-destNet a, simp)
apply (subgoal-tac first-srcNet aa ≠ first-destNet a)
apply (metis 4 5 firstInNeta firstInNet alternativelistconc2)
apply (metis 3 4 5 first-bothNetsd insert-commute)
apply (metis)
proof -
  assume 14 : (ab = first-srcNet a ∧ b = first-destNet a ∨ ab = first-destNet a ∧
b = first-srcNet a) ∧ (c, d) ∈ sdnets aa
  and 15 : sdnets a = {(first-srcNet a, first-destNet a), (first-destNet a,
first-srcNet a)}
  and 16 : sdnets aa = {(first-srcNet aa, first-destNet aa)} ∨ sdnets aa =
{(first-srcNet aa, first-destNet aa), (first-destNet aa, first-srcNet aa)}
  and 17 : {(first-srcNet a, first-destNet a), (first-destNet a, first-srcNet a)} ≠
sdnets aa
  and 18 : first-destNet a = first-srcNet aa → first-srcNet a = first-destNet
aa → first-destNet aa ≠ first-srcNet aa
  and 19 : first-destNet a ≠ first-srcNet a
  and 20 : c = first-srcNet aa → d ≠ first-destNet aa
show netsDistinct ab c ∨ netsDistinct b d
apply (case-tac (ab = first-srcNet a ∧ b = first-destNet a), simp-all)
apply (case-tac c = first-srcNet aa, simp-all)

```

```

apply (metis 2 5 14 20 OTNaux)
apply (subgoal-tac c = first-destNet aa, simp)
apply (subgoal-tac d = first-srcNet aa, simp)
apply (case-tac (first-srcNet a) ≠ (first-destNet aa),simp-all)
  apply (metis 4 5 7 firstInNeta firstInNet alternativelistconc2)
apply (subgoal-tac first-destNet a ≠ first-srcNet aa)
  apply (metis 4 5 7 firstInNeta firstInNet alternativelistconc2)
apply (metis 3 4 5 first-bothNetsd insert-commute)
apply (metis 2 5 14 OTNaux)
apply (metis 2 5 14 OTNaux)
apply (case-tac c = first-srcNet aa, simp-all)
  apply (metis 2 5 14 20 OTNaux)
apply (subgoal-tac c = first-destNet aa, simp)
apply (subgoal-tac d = first-srcNet aa, simp)
apply (case-tac (first-destNet a) ≠ (first-destNet aa),simp-all)
  apply (metis 4 5 7 14 firstInNet alternativelistconc2)
apply (subgoal-tac first-srcNet a ≠ first-srcNet aa)
  apply (metis 4 5 7 14 firstInNeta alternativelistconc2)
apply (metis 3 4 5 first-bothNetsd insert-commute)
apply (metis 2 5 14 OTNaux)
apply (metis 2 5 14 OTNaux)
done

qed
next
show netsDistinct ab d ∨ netsDistinct b c
apply (insert 1 2 3 4 5 6 7 8 9 10 11 12)
apply (cases sdnets a = {(first-srcNet a, first-destNet a)})
apply (cases sdnets aa = {(first-srcNet aa, first-destNet aa)}, simp-all)
apply (case-tac (c = first-srcNet aa ∧ d = first-destNet aa), simp-all)
apply (case-tac (first-srcNet a) ≠ (first-destNet aa), simp-all)
  apply (metis firstInNeta firstInNet alternativelistconc2)
apply (subgoal-tac first-destNet a ≠ first-srcNet aa)
  apply (metis firstInNeta firstInNet alternativelistconc2)
apply (metis first-bothNetsd insert-commute)
apply (case-tac (c = first-srcNet aa ∧ d = first-destNet aa), simp-all)
apply (case-tac (ab = first-srcNet a ∧ b = first-destNet a), simp-all)
apply (case-tac (first-destNet a) ≠ (first-srcNet aa),simp-all)
  apply (metis firstInNeta firstInNet alternativelistconc2)
apply (subgoal-tac first-srcNet a ≠ first-destNet aa)
  apply (metis firstInNeta firstInNet alternativelistconc2)
apply (metis first-bothNetsd insert-commute)
apply (case-tac (first-srcNet aa) ≠ (first-srcNet a),simp-all)
  apply (metis firstInNeta alternativelistconc2)
apply (case-tac first-destNet aa ≠ first-destNet a,simp-all)

```

```

apply (metis firstInNet alternativelistconc2)
apply (metis first-bothNetsd)
proof –
assume 13:  $\forall ab b. ab \neq b \wedge ab \in set(net-list-aux[a,aa]) \wedge b \in set(net-list-aux[a,aa])$ 
 $\longrightarrow netsDistinct ab b$ 
and 14 :  $(ab = first-srcNet a \wedge b = first-destNet a) \vee$ 
 $ab = first-destNet a \wedge b = first-srcNet a) \wedge (c, d) \in sdnets aa$ 
and 15 :  $sdnets a = \{(first-srcNet a, first-destNet a),$ 
 $(first-destNet a, first-srcNet a)\}$ 
and 16 :  $sdnets aa = \{(first-srcNet aa, first-destNet aa)\} \vee$ 
 $sdnets aa = \{(first-srcNet aa, first-destNet aa),$ 
 $(first-destNet aa, first-srcNet aa)\}$ 
and 17 :  $\{(first-srcNet a, first-destNet a),$ 
 $(first-destNet a, first-srcNet a)\} \neq sdnets aa$ 
show  $first-destNet a \neq first-srcNet a \implies netsDistinct ab d \vee netsDistinct b c$ 
apply (insert 1 2 3 4 5 6 13 14 15 16 17)
apply (cases sdnets aa = {(first-srcNet aa, first-destNet aa)}, simp-all)
apply (case-tac (c = first-srcNet aa \wedge d = first-destNet aa), simp-all)
apply (case-tac (ab = first-srcNet a \wedge b = first-destNet a), simp-all)
apply (case-tac (first-destNet a) \neq (first-srcNet aa), simp-all)
apply (metis firstInNeta firstInNet alternativelistconc2)
apply (subgoal-tac first-srcNet a \neq first-destNet aa)
apply (metis firstInNeta firstInNet alternativelistconc2)
apply (metis first-bothNetsd insert-commute)
apply (case-tac (first-srcNet aa) \neq (first-srcNet a), simp-all)
apply (metis firstInNeta alternativelistconc2)
apply (case-tac first-destNet aa \neq first-destNet a, simp-all)
apply (metis firstInNet alternativelistconc2)
apply (metis first-bothNetsd)
proof –
assume 20:  $\{(first-srcNet a, first-destNet a), (first-destNet a, first-srcNet a)\} \neq$ 
 $\{(first-srcNet aa, first-destNet aa), (first-destNet aa, first-srcNet aa)\}$ 
and 21:  $first-destNet a \neq first-srcNet a$ 
show  $netsDistinct ab d \vee netsDistinct b c$ 
apply (case-tac (c = first-srcNet aa \wedge d = first-destNet aa), simp-all)
apply (case-tac (ab = first-srcNet a \wedge b = first-destNet a), simp-all)
apply (case-tac (first-destNet a) \neq (first-srcNet aa), simp-all)
apply (metis 4 5 7 firstInNeta firstInNet alternativelistconc2)
apply (subgoal-tac first-srcNet a \neq first-destNet aa)
apply (metis 4 5 7 firstInNeta firstInNet alternativelistconc2)
apply (metis 20 insert-commute)
apply (case-tac (first-srcNet aa) \neq (first-srcNet a), simp-all)
apply (metis 4 5 13 14 firstInNeta alternativelistconc2)
apply (case-tac first-destNet aa \neq first-destNet a, simp-all)

```

```

apply (metis 4 5 13 14 firstInNet alternativelistconc2)
apply (case-tac (ab = first-srcNet a ∧ b = first-destNet a), simp-all)
apply (case-tac (first-destNet a) ≠ (first-srcNet aa), simp-all)
apply (metis 20)
apply (subgoal-tac first-srcNet a ≠ first-srcNet aa)
apply (metis 20)
apply (metis 21)
apply (case-tac (first-srcNet aa) ≠ (first-destNet a))
apply (metis (no-types, lifting) 2 3 4 5 7 14 OTNaux
      firstInNet firstInNeta first-bothNetsd isInAlternativeList)
by (metis 2 4 5 7 20 14 OTNaux doubleton-eq-iff firstInNet
      firstInNeta isInAlternativeList)

qed
qed
qed
qed

```

lemma ANDaux3[rule-format]:

$y \in \text{set } xs \rightarrow a \in \text{set} (\text{net-list-aux } [y]) \rightarrow a \in \text{set} (\text{net-list-aux } xs)$
by (induct xs) (simp-all add: isInAlternativeList)

lemma ANDaux2:

$\text{allNetsDistinct } (x \# xs) \Rightarrow y \in \text{set } xs \Rightarrow \text{allNetsDistinct } [x,y]$
apply (simp add: allNetsDistinct-def)
by (meson ANDaux3 isInAlternativeList netlistaux)

lemma NCisSD2[rule-format]:

$\neg \text{member DenyAll } a \Rightarrow \text{OnlyTwoNets } (a \# p) \Rightarrow$
 $\text{NetsCollected2 } (a \# p) \Rightarrow \text{NetsCollected } (a \# p) \Rightarrow$
 $\text{noDenyAll } (p) \Rightarrow \text{allNetsDistinct } (a \# p) \Rightarrow s \in \text{set } p \Rightarrow$
 $\text{disjSD-2 } a s$
by (metis ANDaux2 FWNormalisationCore.member.simps(2) NCisSD2aux NetsCollected.simps(1)
 $\text{NetsCollected2.simps(1) OTNConc OTNoTN empty-iff empty-set list.set-intros(1)}$
 $\text{noDA})$

lemma separatedNC[rule-format]:

$\text{OnlyTwoNets } p \rightarrow \text{NetsCollected2 } p \rightarrow \text{NetsCollected } p \rightarrow \text{noDenyAll1 } p \rightarrow$
 $\text{allNetsDistinct } p \rightarrow \text{separated } p$
proof (induct p, simp-all, rename-tac a b, case-tac a = DenyAll, simp-all, goal-cases)
fix a **fix** p::('a set set, 'b) Combinators list
show OnlyTwoNets p → NetsCollected2 p → NetsCollected p → noDenyAll1 p
 \rightarrow

```


$$\text{allNetsDistinct } p \rightarrow \text{separated } p \implies a \neq \text{DenyAll} \implies \text{OnlyTwoNets } (a \# p)$$


$$\rightarrow$$


$$\begin{aligned} & \text{first-bothNet } a \neq \text{firstList } p \wedge \text{NetsCollected2 } p \rightarrow \\ & (\forall aa \in \text{set } p. \text{first-bothNet } a \neq \text{first-bothNet } aa) \wedge \text{NetsCollected } p \rightarrow \\ & \text{noDenyAll1 } (a \# p) \rightarrow \text{allNetsDistinct } (a \# p) \rightarrow (\forall s. s \in \text{set } p \rightarrow \\ & \text{disjSD-2 } a s) \wedge \text{separated } p \end{aligned}$$

apply (intro impI,drule mp, erule OTNConc,drule mp)
apply (case-tac p, simp-all)
apply (drule mp,erule noDA1C, intro conjI allI impI NCisSD2, simp-all)
apply (case-tac a, simp-all)
apply (case-tac a, simp-all)
using ANDConc by auto

next
fix a:('a set set,'b) Combinators fix p ::('a set set,'b) Combinators list
show OnlyTwoNets p → NetsCollected2 p → NetsCollected p → noDenyAll1 p

$$\rightarrow$$


$$\begin{aligned} & \text{allNetsDistinct } p \rightarrow \text{separated } p \implies a = \text{DenyAll} \implies \text{OnlyTwoNets } p \rightarrow \\ & \{\} \neq \text{firstList } p \wedge \text{NetsCollected2 } p \rightarrow (\forall a \in \text{set } p. \{\} \neq \text{first-bothNet } a) \wedge \text{NetsCollected } p \rightarrow \\ & \text{noDenyAll } p \rightarrow \text{allNetsDistinct } (\text{DenyAll} \# p) \rightarrow \\ & (\forall s. s \in \text{set } p \rightarrow \text{disjSD-2 DenyAll } s) \wedge \text{separated } p \end{aligned}$$

by (simp add: ANDConc disjSD-2-def noDA1eq)
qed

lemma separatedNC'[rule-format]:
OnlyTwoNets p → NetsCollected2 p → NetsCollected p → noDenyAll1 p →
allNetsDistinct p → separated p
proof (induct p)
case Nil show ?case by simp
next
case (Cons a p) then show ?case
apply simp
proof (cases a = DenyAll) print-cases
case True
then show OnlyTwoNets (a # p) → first-bothNet a ≠ firstList p ∧ NetsCollected2 p →

$$\begin{aligned} & (\forall aa \in \text{set } p. \text{first-bothNet } a \neq \text{first-bothNet } aa) \wedge \text{NetsCollected } p \rightarrow \\ & \text{noDenyAll1 } (a \# p) \rightarrow \text{allNetsDistinct } (a \# p) \rightarrow \\ & (\forall s. s \in \text{set } p \rightarrow \text{disjSD-2 } a s) \wedge \text{separated } p \end{aligned}$$

apply(insert Cons.hyps `a = DenyAll`)
apply (intro impI,drule mp, erule OTNConc,drule mp)
apply (case-tac p, simp-all)
apply (case-tac a, simp-all)
apply (case-tac a, simp-all)

```

```

  by (simp add: ANDConc disjSD-2-def noDA1eq)
next
  case False
  then show OnlyTwoNets (a # p) —> first-bothNet a ≠ firstList p ∧ NetsCollected2
p —>
  (forall aa ∈ set p. first-bothNet a ≠ first-bothNet aa) ∧ NetsCollected p —>
  noDenyAll1 (a # p) —> allNetsDistinct (a # p) —> (∀ s. s ∈ set p
—>
  disjSD-2 a s) ∧ separated p
  apply(insert Cons.hyps ‹a ≠ DenyAll›)
  by (metis NetsCollected.simps(1) NetsCollected2.simps(1) separated.simps(1) separatedNC)
qed
qed

lemma NC2Sep[rule-format]: noDenyAll1 p —> NetsCollected2 (separate p)
proof (induct p rule: separate.induct, simp-all, goal-cases)
  fix x :: ('a, 'b) Combinators list
  case 1 then show ?case
    by (metis fMTaux firstList.simps(1) fl2 noDA1eq noDenyAll.elims(2) separate.simps(5))
next
  fix v va fix y:: ('a, 'b) Combinators fix z
  case 2 then show ?case
    by (simp add: fl2 noDA1eq)
next
  fix v va vb fix y:: ('a, 'b) Combinators fix z
  case 3 then show ?case
    by (simp add: fl2 noDA1eq)
next
  fix v va fix y:: ('a, 'b) Combinators fix z
  case 4 then show ?case
    by (simp add: fl2 noDA1eq)
qed

lemma separatedSep[rule-format]:
  OnlyTwoNets p —> NetsCollected2 p —> NetsCollected p —>
  noDenyAll1 p —> allNetsDistinct p —> separated (separate p)
  by (simp add: ANDSep NC2Sep NetsCollectedSep OTNSEp noDA1sep separatedNC)

lemma rADnMT[rule-format]: p ≠ [] —> removeAllDuplicates p ≠ []
  by (induct p) simp-all

```

```

lemma remDupsNMT[rule-format]:  $p \neq [] \longrightarrow \text{remdups } p \neq []$ 
  by (metis remdups_eq_nil_iff)

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

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

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

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

```

2.3.3 Normalisation Proofs: Integer Port

theory

NormalisationIntegerPortProof

imports

NormalisationGenericProofs

begin

Normalisation proofs which are specific to the IntegerPort address representation.

```

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

```

```

lemma aux26[simp]: twoNetsDistinct a b c d  $\implies$ 
  dom (C (AllowPortFromTo a b p))  $\cap$  dom (C (DenyAllFromTo c d)) = {}
  apply (auto simp: PLemmas twoNetsDistinct-def netsDistinct-def)[1]
  by auto

```

```

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

```

```

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

```

```

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

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

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

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

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

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

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

lemma CConcStartaux: C a x = None  $\implies$  (C aa ++ C a) x = C aa x
  by (simp add: PLemmas)

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

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

```

```

lemma mr-is-C[rule-format]:
  applied-rule-rev C x p = Some a → C (list2FWpolicy (p)) x = C a x
  apply (simp add: applied-rule-rev-def)
  apply (rule rev-induct,auto)
  apply (metis CConcEnd)
  apply (metis CConcEnd)
  by (metis CConcStart applied-rule-rev-def mrNnt option.exhaust)

lemma CConcStart2:
  p ≠ [] ⇒ x ∉ dom (C a) ⇒ C (list2FWpolicy (p @ [a])) x = C (list2FWpolicy p) x
  by (erule CConcStart,simp add: PLemmas)

lemma CConcEnd1:
  q @ p ≠ [] ⇒ x ∉ dom (C a) ⇒ C (list2FWpolicy (q @ p @ [a])) x = C (list2FWpolicy (q @ p)) x
  apply (subst lCdom2)
  by (rule CConcStart2, simp-all)

lemma CConcEnd2[rule-format]:
  x ∈ dom (C a) → C (list2FWpolicy (xs @ [a])) x = C a x (is ?P xs)
  apply (rule-tac P = ?P in list2FWpolicy.induct)
  by (auto simp:C.simps)

lemma bar3:
  x ∈ dom (C (list2FWpolicy (xs @ [xa]))) ⇒ x ∈ dom (C (list2FWpolicy xs)) ∨ x ∈ dom (C xa)
  by auto (metis CConcStart eq-Nil-appendI l2p-aux2 option.simps(3))

lemma CeqEnd[rule-format,simp]:
  a ≠ [] → x ∈ dom (C (list2FWpolicy a)) → C (list2FWpolicy(b@a)) x = (C (list2FWpolicy a)) x
  apply (rule rev-induct,simp-all)
  subgoal for xa xs
    apply (case-tac xs ≠ [], simp-all)
    apply (case-tac x ∈ dom (C xa))
    apply (metis CConcEnd2 MRList2 mr-is-C )
    apply (metis CConcEnd1 CConcStart2 Nil-is-append-conv bar3 )
    apply (metis MRList2 eq-Nil-appendI mr-is-C )
    done
  done

lemma CConcStartA[rule-format,simp]:

```

```

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

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

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

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

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

lemma C-eq-if-mr-eq:
 $\text{applied-rule-rev } C x b = \text{applied-rule-rev } C x a \implies a \neq [] \implies b \neq [] \implies$ 
 $C (\text{list2FWpolicy} a) x = C (\text{list2FWpolicy} b) x$ 
apply (cases applied-rule-rev C x a = None, simp-all)
apply (subst nMRtoNone,simp-all)
apply (subst nMRtoNone, simp-all)
by (auto intro: C-eq-if-mr-eq2)

lemma notmatching-notdom: applied-rule-rev C x (p@[a]) ≠ Some a  $\implies x \notin \text{dom } (C a)$ 
by (simp add: applied-rule-rev-def split: if-splits)

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

```

```
(C r)
  apply (rule rev-induct)
  apply simp-all
  apply (intro impI conjI, simp)
  subgoal for xa xs
    apply (rule-tac p = a @ b # xs in notmatching-notdom,simp-all)
    done
  by (metis MRLList2 MRLList3 append-Cons option.inject)
```

lemma foo3D:

```
wellformed-policy1 p ==> p = DenyAll # ps ==>
  applied-rule-rev C x p = [DenyAll] ==> r in set ps ==> x notin dom (C r)
  by (rule-tac a = [] and b = DenyAll and c = ps in foo3a, simp-all)
```

lemma foo4[rule-format]:

```
set p = set s ∧ (∀ r. r in set p → x notin dom (C r)) → (∀ r. r in set s → x notin dom (C r))
  by simp
```

lemma foo5b[rule-format]:

```
x in dom (C b) → (∀ r. r in set c → x notin dom (C r)) → applied-rule-rev C x (b#c)
= Some b
  apply (simp add: applied-rule-rev-def)
  apply (rule-tac xs = c in rev-induct, simp-all)
  done
```

lemma mr-first:

```
x in dom (C b) ==> ∀ r. r in set c → x notin dom (C r) ==> s = b # c ==> applied-rule-rev
C x s = [b]
  by (simp add: foo5b)
```

lemma mr-charn[rule-format]:

```
a in set p → (x in dom (C a)) → (∀ r. r in set p ∧ x in dom (C r) → r = a)
  →
  applied-rule-rev C x p = Some a
unfolding applied-rule-rev-def
  apply (rule-tac xs = p in rev-induct)
  apply(simp)
  by(safe,auto)
```

lemma foo8:

```
∀ r. r in set p ∧ x in dom (C r) → r = a ==> set p = set s ==>
  ∀ r. r in set s ∧ x in dom (C r) → r = a
  by auto
```

```

lemma mrConcEnd[rule-format]:
  applied-rule-rev C x (b # p) = Some a  $\rightarrow$  a  $\neq$  b  $\rightarrow$  applied-rule-rev C x p = Some
  a
  apply (simp add: applied-rule-rev-def)
  by (rule-tac xs = p in rev-induct,auto)

lemma wp3tl[rule-format]: wellformed-policy3 p  $\rightarrow$  wellformed-policy3 (tl p)
  apply (induct p, simp-all)
  subgoal for a p
    apply(case-tac a, simp-all)
    done
  done

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

lemma foo98[rule-format]:
  applied-rule-rev C x (aa # p) = Some a  $\rightarrow$  x  $\in$  dom (C r)  $\rightarrow$  r  $\in$  set p  $\rightarrow$  a  $\in$ 
  set p
  unfolding applied-rule-rev-def
  apply (rule rev-induct, simp-all)
  subgoal for xa xs
    apply (case-tac r = xa, simp-all)
    done
  done

lemma mrMTNone[simp]: applied-rule-rev C x [] = None
  by (simp add: applied-rule-rev-def)

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

lemma mrSet[rule-format]: applied-rule-rev C x p = Some r  $\rightarrow$  r  $\in$  set p
  unfolding applied-rule-rev-def
  by (rule-tac xs=p in rev-induct, simp-all)

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

```

```

lemma foo25[rule-format]: wellformed-policy3 (p@[x]) —> wellformed-policy3 p
by (induct p, simp-all, case-tac a, simp-all)

lemma mr-in-dom[rule-format]: applied-rule-rev C x p = Some a —> x ∈ dom (C a)
apply (rule-tac xs = p in rev-induct)
by (auto simp: applied-rule-rev-def)

lemma wp3EndMT[rule-format]:
wellformed-policy3 (p@[xs]) —> AllowPortFromTo a b po ∈ set p —>
dom (C (AllowPortFromTo a b po)) ∩ dom (C xs) = {}
apply (induct p,simp-all)
apply (intro impI,drule mp,erule wp3Conc)
by clarify auto

lemma foo29: [|dom (C a) ≠ {}; dom (C a) ∩ dom (C b) = {}|] ==> a ≠ b by auto

lemma foo28:
AllowPortFromTo a b po ∈ set p ==> dom (C (AllowPortFromTo a b po)) ≠ {} ==>
wellformed-policy3 (p @ [x]) ==> x ≠ AllowPortFromTo a b po
by (metis foo29 C.simps(3) wp3EndMT)

lemma foo28a[rule-format]: x ∈ dom (C a) ==> dom (C a) ≠ {} by auto

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

lemma DenyAllowDisj:
dom (C (AllowPortFromTo a b p)) ≠ {} ==>
dom (C (DenyAllFromTo a b)) ∩ dom (C (AllowPortFromTo a b p)) ≠ {}
by (metis Int-absorb1 allow-deny-dom)

lemma foo31:
∀ r. r ∈ set p ∧ x ∈ dom (C r) —>
r = AllowPortFromTo a b po ∨ r = DenyAllFromTo a b ∨ r = DenyAll ==>
set p = set s ==>
∀ r. r ∈ set s ∧ x ∈ dom (C r) —> r = AllowPortFromTo a b po ∨ r = DenyAllFromTo
a b ∨ r = DenyAll
by auto

lemma wp1-auxa:

```

wellformed-policy1-strong $p \Rightarrow (\exists r. \text{applied-rule-rev } C x p = \text{Some } r)$
apply (*rule DAimpliesMR-E*)
by (*erule wp1-aux1aa*)

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

lemma *domTrans*: $\text{dom } a \subseteq \text{dom } b \Rightarrow \text{dom } b \cap \text{dom } c = \{\} \Rightarrow \text{dom } a \cap \text{dom } c = \{\}$ **by** *auto*

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

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

lemma *wellformed-policy3-charn[rule-format]*:
 $\text{singleCombinators } p \rightarrow \text{distinct } p \rightarrow \text{allNetsDistinct } p \rightarrow$
 $\text{wellformed-policy1 } p \rightarrow \text{wellformed-policy2 } p \rightarrow \text{wellformed-policy3 } p$
apply (*induct-tac* p)
apply *simp-all*
apply (*auto intro: singleCombinatorsConc ANDConc waux2 wp2Conc*)
subgoal for a **list**
apply (*case-tac* a , *simp-all*, *clarify*)
apply (*metis C.elims DomInterAllowsMT DomInterAllowsMT-Ports aux0-0 aux7aa inf-commute*)
done
done

lemma *DistinctNetsDenyAllow*:
 $\text{DenyAllFromTo } b c \in \text{set } p \Rightarrow$
 $\text{AllowPortFromTo } a d po \in \text{set } p \Rightarrow$

```

 $allNetsDistinct p \implies \text{dom } (C (\text{DenyAllFromTo } b c)) \cap \text{dom } (C (\text{AllowPortFromTo } a d po)) \neq \{\} \implies$ 
 $b = a \wedge c = d$ 
unfolding allNetsDistinct-def
apply (frule-tac  $x = b$  in spec)
apply (drule-tac  $x = d$  in spec)
apply (drule-tac  $x = a$  in spec)
apply (drule-tac  $x = c$  in spec)
apply (simp,metis Int-commute ND0aux1 ND0aux3 NDComm aux26 twoNetsDistinct-def ND0aux2 ND0aux4)
done

```

lemma *DistinctNetsAllowAllow*:

```

 $\text{AllowPortFromTo } b c \text{ poo} \in \text{set } p \implies$ 
 $\text{AllowPortFromTo } a d \text{ po} \in \text{set } p \implies$ 
 $allNetsDistinct p \implies$ 
 $\text{dom } (C (\text{AllowPortFromTo } b c \text{ poo})) \cap \text{dom } (C (\text{AllowPortFromTo } a d \text{ po})) \neq \{\}$ 
 $\implies$ 
 $b = a \wedge c = d \wedge \text{poo} = \text{po}$ 
unfolding allNetsDistinct-def
apply (frule-tac  $x = b$  in spec)
apply (drule-tac  $x = d$  in spec)
apply (drule-tac  $x = a$  in spec)
apply (drule-tac  $x = c$  in spec)
apply (simp,metis DomInterAllowsMT DomInterAllowsMT-Ports ND0aux3 ND0aux4 NDComm twoNetsDistinct-def)
done

```

lemma *WP2RS2[simp]*:

```

 $\text{singleCombinators } p \implies \text{distinct } p \implies allNetsDistinct p \implies$ 
 $\text{wellformed-policy2 } (\text{removeShadowRules2 } p)$ 
proof (induct p)
  case Nil thus ?case by simp
next
  case (Cons  $x xs$ )
    have wp-xs: wellformed-policy2 (removeShadowRules2  $xs$ )
      by (metis Cons ANDConc distinct.simps(2) singleCombinatorsConc)
    show ?case
    proof (cases  $x$ )
      case DenyAll thus ?thesis using wp-xs by simp
next
  case (DenyAllFromTo  $a b$ ) thus ?thesis
    using wp-xs Cons by (simp,metis DenyAllFromTo aux aux7 tNDComm deny-dom)

```

```

next
  case (AllowPortFromTo a b p) thus ?thesis
    using wp-xs by (simp, metis aux26 AllowPortFromTo Cons(4) aux aux7a tND-Comm)
next
  case (Conc a b) thus ?thesis
    by (metis Conc Cons(2) singleCombinators.simps(2))
qed
qed

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

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

```

```

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

lemma wellformed2-sorted[simp]:
  all-in-list p l ==> distinct p ==> allNetsDistinct p ==>
    singleCombinators p ==> wellformed-policy2 (sort p l)
  apply (rule sorted-WP2, erule sort-is-sorted, simp-all)
  apply (auto elim: all-in-listSubset intro: SC3 singleCombinatorsConc sorted-insort)
done

lemma wellformed2-sortedQ[simp]: [| all-in-list p l; distinct p; allNetsDistinct p;
  singleCombinators p |] ==> wellformed-policy2 (qsort p l)
  apply (rule sorted-WP2, erule sort-is-sortedQ, simp-all)
  apply (auto elim: all-in-listSubset intro: SC3Q singleCombinatorsConc distinct-sortQ)
done

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

lemma C-eq-RS1n:
  C(list2FWpolicy (removeShadowRules1-alternative p)) = C(list2FWpolicy p)
proof (cases p)print-cases
  case Nil then show ?thesis apply(simp-all)
  by (metis list2FWpolicy.simps(1) rSR1-eq removeShadowRules1.simps(2))
next
  case (Cons x list) show ?thesis
    apply (rule rev-induct)
    apply (metis rSR1-eq removeShadowRules1.simps(2))
    subgoal for x xs
      apply (case-tac xs = [], simp-all)
      unfolding removeShadowRules1-alternative-def
      apply (case-tac x, simp-all)
      by (metis (no-types, opaque-lifting) CConcEnd2 CConcStart C-DenyAll RS1n-nMT
aux114
        domIff removeShadowRules1-alternative-def
        removeShadowRules1-alternative-rev.simps(2) rev.simps(2))
    done
qed

```

```

lemma C-eq-RS1[simp]:

$$p \neq [] \implies C(\text{list2FWpolicy} (\text{removeShadowRules1 } p)) = C(\text{list2FWpolicy } p)$$

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

lemma EX-MR-aux[rule-format]:

$$\text{applied-rule-rev } C x (\text{DenyAll} \# p) \neq \text{Some DenyAll} \longrightarrow (\exists y. \text{applied-rule-rev } C x p = \text{Some } y)$$

apply (simp add: applied-rule-rev-def)
apply (rule-tac xs = p in rev-induct, simp-all)
done

lemma EX-MR :

$$\text{applied-rule-rev } C x p \neq [\text{DenyAll}] \implies p = \text{DenyAll} \# ps \implies$$


$$\text{applied-rule-rev } C x p = \text{applied-rule-rev } C x ps$$

apply auto
apply (subgoal-tac applied-rule-rev C x (DenyAll#ps) ≠ None, auto)
apply (metis mrConcEnd)
by (metis DAimpliesMR-E list.sel(1) hd-in-set list.simps(3) not-Some-eq)

lemma mr-not-DA:

$$\text{wellformed-policy1-strong } s \implies$$


$$\text{applied-rule-rev } C x p = [\text{DenyAllFromTo } a ab] \implies \text{set } p = \text{set } s \implies$$


$$\text{applied-rule-rev } C x s \neq [\text{DenyAll}]$$

apply (subst wp1n-tl, simp-all)
apply (subgoal-tac x ∈ dom (C (DenyAllFromTo a ab)))
apply (subgoal-tac DenyAllFromTo a ab ∈ set (tl s))
apply (metis wp1n-tl foo98 wellformed-policy1-strong.simps(2))
using mrSet r-not-DA-in-tl apply blast
by (simp add: mr-in-dom)

lemma domsMT-notND-DD:

$$\text{dom } (C (\text{DenyAllFromTo } a b)) \cap \text{dom } (C (\text{DenyAllFromTo } c d)) \neq \{\} \implies \neg \text{nets-Distinct } a c$$

using deny-dom twoNetsDistinct-def by blast

lemma domsMT-notND-DD2:

$$\text{dom } (C (\text{DenyAllFromTo } a b)) \cap \text{dom } (C (\text{DenyAllFromTo } c d)) \neq \{\} \implies \neg \text{nets-Distinct } b d$$

using deny-dom twoNetsDistinct-def by blast

lemma domsMT-notND-DD3:

$$x \in \text{dom } (C (\text{DenyAllFromTo } a b)) \implies x \in \text{dom } (C (\text{DenyAllFromTo } c d)) \implies \neg \text{netsDistinct } a c$$

by(auto intro!:domsMT-notND-DD)

```

```

lemma domsMT-notND-DD4:
   $x \in \text{dom} (C (\text{DenyAllFromTo } a b)) \Rightarrow x \in \text{dom} (C (\text{DenyAllFromTo } c d)) \Rightarrow \neg \text{netsDistinct } b d$ 
  by (auto intro!: domsMT-notND-DD2)

lemma NetsEq-if-sameP-DD:
   $\text{allNetsDistinct } p \Rightarrow u \in \text{set } p \Rightarrow v \in \text{set } p \Rightarrow u = \text{DenyAllFromTo } a b \Rightarrow$ 
   $v = \text{DenyAllFromTo } c d \Rightarrow x \in \text{dom} (C u) \Rightarrow x \in \text{dom} (C v) \Rightarrow a = c \wedge b = d$ 
  apply (simp add: allNetsDistinct-def)
  by (metis ND0aux1 ND0aux2 domsMT-notND-DD3 domsMT-notND-DD4)

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 (cases r)
    case DenyAll show ?thesis by (metis DenyAll)
  next
    case (DenyAllFromTo x y) show ?thesis
      by (metis AD-aux DenyAllFromTo SC aND domInterMT inDom inp mrSet
            mr-in-dom mr-is-allow)
  next
    case (AllowPortFromTo x y b) show ?thesis
      by (metis (no-types, lifting) AllowPortFromTo DistinctNetsAllowAllow aND dom-
            InterMT
            inDom inp mrSet mr-in-dom mr-is-allow)
  next
    case (Conc x y) thus ?thesis using assms by (metis aux0-0)
  qed

lemma none-MT-rulesSubset[rule-format]:
  none-MT-rules C a → set b ⊆ set a → none-MT-rules C b
  by (induct b,simp-all) (metis notMTnMT)

lemma nMTSort: none-MT-rules C p ⇒ none-MT-rules C (sort p l)
  by (metis set-sort nMTeqSet)

lemma nMTSortQ: none-MT-rules C p ⇒ none-MT-rules C (qsort p l)
  by (metis set-sortQ nMTeqSet)

```

```

lemma wp3char[rule-format]:
  none-MT-rules C xs  $\wedge$  C (AllowPortFromTo a b po)= $\emptyset$   $\wedge$  well-formed-policy3(xs@[DenyAllFromTo a b])  $\longrightarrow$ 
  AllowPortFromTo a b po  $\notin$  set xs
  apply (induct xs,simp-all)
  by (metis domNMT wp3Conc)

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

lemma rule-charn2:
  assumes aND: allNetsDistinct p
  and wp1: wellformed-policy1 p
  and SC: singleCombinators p
  and wp3: wellformed-policy3 p
  and allow-in-list: AllowPortFromTo c d po  $\in$  set p
  and x-in-dom-allow: x  $\in$  dom (C (AllowPortFromTo c d po))
  shows applied-rule-rev C x p = Some (AllowPortFromTo c d po)
  proof (insert assms, induct p rule: rev-induct)
    case Nil show ?case using Nil by simp
  next
    case (snoc y ys)
    show ?case using snoc
      apply (case-tac y = (AllowPortFromTo c d po), simp-all )
      apply (simp add: applied-rule-rev-def)
      apply (subgoal-tac ys  $\neq \[]$ )
      apply (subgoal-tac applied-rule-rev C x ys = Some (AllowPortFromTo c d po))
      defer 1
      apply (metis ANDConcEnd SCCConcEnd WP1ConcEnd foo25)
      apply (metis inSet-not-MT)
    proof (cases y)
      case DenyAll thus ?thesis using DenyAll snoc
        apply simp
        by (metis DAnotTL DenyAll inSet-not-MT policy2list.simps(2))
    next

```

```

case (DenyAllFromTo a b) thus ?thesis using snoc apply simp
  apply (simp-all add: applied-rule-rev-def)
  apply (rule conjI)
  apply (metis domInterMT wp3EndMT)
  apply (rule impI)
  by (metis ANDConcEnd DenyAllFromTo SCCConcEnd WP1ConcEnd foo25)
next
  case (AllowPortFromTo a1 a2 b) thus ?thesis
    using AllowPortFromTo snoc apply simp
    apply (simp-all add: applied-rule-rev-def)
    apply (rule conjI)
    apply (metis domInterMT wp3EndMT)
    by (metis ANDConcEnd AllowPortFromTo SCCConcEnd WP1ConcEnd foo25
      x-in-dom-allow)
next
  case (Conc a b) thus ?thesis using Conc snoc apply simp
    by (metis Conc aux0-0 in-set-conv-decomp)
qed
qed

lemma rule-charn3:
  wellformed-policy1 p  $\implies$  allNetsDistinct p  $\implies$  singleCombinators p  $\implies$ 
  wellformed-policy3 p  $\implies$  applied-rule-rev C x p = [DenyAllFromTo c d]  $\implies$ 
  AllowPortFromTo a b po ∈ set p  $\implies$  x ∉ dom (C (AllowPortFromTo a b po))
  by (clarify, auto simp: rule-charn2 dom-def)

lemma rule-charn4:
  assumes wp1: wellformed-policy1 p
  and aND: allNetsDistinct p
  and SC: singleCombinators p
  and wp3: wellformed-policy3 p
  and DA: DenyAll ∉ set p
  and mr: applied-rule-rev C x p = Some (DenyAllFromTo a b)
  and rinp: r ∈ set p
  and xindom: x ∈ dom (C r)
  shows r = DenyAllFromTo a b
proof (cases r)
  case DenyAll thus ?thesis using DenyAll assms by simp
next
  case (DenyAllFromTo c d) thus ?thesis using assms apply simp
  apply (erule-tac x = x and p = p and v = (DenyAllFromTo a b) and
    u = (DenyAllFromTo c d) in NetsEq-if-sameP-DD)
    apply simp-all
  apply (erule mrSet)

```

```

by (erule mr-in-dom)
next
  case (AllowPortFromTo c d e) thus ?thesis using assms apply simp
    apply (subgoal-tac x ∉ dom (C (AllowPortFromTo c d e)))
      apply simp
    apply (rule-tac p = p in rule-charn3)
    by (auto intro: SCnotConc)
next
  case (Conc a b) thus ?thesis using assms apply simp
    by (metis Conc aux0-0)
qed

lemma foo31a:
  ∀ r. r ∈ set p ∧ x ∈ dom (C r) → r = AllowPortFromTo a b po ∨ r = DenyAllFromTo a b ∨ r = DenyAll ⇒
    set p = set s ⇒ r ∈ set s ⇒ x ∈ dom (C r) ⇒
    r = AllowPortFromTo a b po ∨ r = DenyAllFromTo a b ∨ r = DenyAll
  by auto

lemma aux4[rule-format]:
  applied-rule-rev C x (a#p) = Some a → a ∉ set (p) → applied-rule-rev C x p = None
  apply (rule rev-induct,simp-all)
  by (metis aux0-4 empty-iff empty-set insert-iff list.simps(15) mrSet mreq-end3)

lemma mrDA-tl:
  assumes mr-DA: applied-rule-rev C x p = Some DenyAll
  and wp1n: wellformed-policy1-strong p
  shows applied-rule-rev C x (tl p) = None
  apply (rule aux4 [where a = DenyAll])
  apply (metis wp1n-tl mr-DA wp1n)
  by (metis WP1n-DA-notinSet wp1n)

lemma rule-charnDAFT:
  wellformed-policy1-strong p ⇒ allNetsDistinct p ⇒ singleCombinators p ⇒
  wellformed-policy3 p ⇒ applied-rule-rev C x p = [DenyAllFromTo a b] ⇒ r ∈ set (tl p) ⇒
  x ∈ dom (C r) ⇒ r = DenyAllFromTo a b
  apply (subgoal-tac p = DenyAll#(tl p))
  apply (metis AND-tl Combinators.distinct(1) SC-tl list.sel(3) mrConcEnd
rule-charn4 waux2 wellformed-policy1-charn wp1-aux1aa wp1-eq wp3tl)
  using wp1n-tl by blast

lemma mrDenyAll-is-unique:

```

```

[wellformed-policy1-strong p; applied-rule-rev C x p = Some DenyAll;
 r ∈ set (tl p)] ⇒ x ∉ dom (C r)
apply (rule-tac a = [] and b = DenyAll and c = tl p in foo3a, simp-all)
apply (metis wp1n-tl)
by (metis WP1n-DA-notinSet)

```

theorem *C-eq-Sets-mr*:

```

assumes sets-eq: set p = set s
and SC:      singleCombinators p
and wp1-p:   wellformed-policy1-strong p
and wp1-s:   wellformed-policy1-strong s
and wp3-p:   wellformed-policy3 p
and wp3-s:   wellformed-policy3 s
and aND:     allNetsDistinct p
shows applied-rule-rev C x p = applied-rule-rev C x s
proof (cases applied-rule-rev C x p)
  case None
  have DA: DenyAll ∈ set p using wp1-p by (auto simp: wp1-aux1aa)
  have notDA: DenyAll ∉ set p using None by (auto simp: DAimplieMR)
  thus ?thesis using DA by (contradiction)
next
  case (Some y) thus ?thesis
  proof (cases y)
    have tl-p: p = DenyAll#(tl p) by (metis wp1-p wp1n-tl)
    have tl-s: s = DenyAll#(tl s) by (metis wp1-s wp1n-tl)
    have tl-eq: set (tl p) = set (tl s)
      by (metis list.sel(3) WP1n-DA-notinSet sets-eq foo2
           wellformed-policy1-charn wp1-aux1aa wp1-eq wp1-p wp1-s)
    { case DenyAll
      have mr-p-is-DenyAll: applied-rule-rev C x p = Some DenyAll
        by (simp add: DenyAll Some)
      hence x-notin-tl-p: ∀ r. r ∈ set (tl p) → x ∉ dom (C r) using wp1-p
        by (auto simp: mrDenyAll-is-unique)
      hence x-notin-tl-s: ∀ r. r ∈ set (tl s) → x ∉ dom (C r) using tl-eq
        by auto
      hence mr-s-is-DenyAll: applied-rule-rev C x s = Some DenyAll using tl-s
        by (auto simp: mr-first)
      thus ?thesis using mr-p-is-DenyAll by simp
    }
    {case (DenyAllFromTo a b)
      have mr-p-is-DAFT: applied-rule-rev C x p = Some (DenyAllFromTo a b)
        by (simp add: DenyAllFromTo Some)
      have DA-notin-tl: DenyAll ∉ set (tl p)
        by (metis WP1n-DA-notinSet wp1-p)
    }
  
```

```

have mr-tl-p: applied-rule-rev C x p = applied-rule-rev C x (tl p)
  by (metis Combinators.simps(4) DenyAllFromTo Some mrConcEnd tl-p)
have dom-tl-p:  $\bigwedge r. r \in set(tl p) \wedge x \in dom(C r) \implies r = (\text{DenyAllFromTo } a b)$ 
  using wp1-p AND SC wp3-p mr-p-is-DAFT
  by (auto simp: rule-charnDAFT)
hence dom-tl-s:  $\bigwedge r. r \in set(tl s) \wedge x \in dom(C r) \implies r = (\text{DenyAllFromTo } a$ 
b)
  using tl-eq by auto
have DAFT-in-tl-s: DenyAllFromTo a b  $\in set(tl s)$  using mr-tl-p
  by (metis DenyAllFromTo mrSet mr-p-is-DAFT tl-eq)
have x-in-dom-DAFT:  $x \in dom(C(\text{DenyAllFromTo } a b))$ 
  by (metis mr-p-is-DAFT DenyAllFromTo mr-in-dom)
hence mr-tl-s-is-DAFT: applied-rule-rev C x (tl s) = Some (DenyAllFromTo a b)
  using DAFT-in-tl-s dom-tl-s by (metis mr-charn)
hence mr-s-is-DAFT: applied-rule-rev C x s = Some (DenyAllFromTo a b)
  using tl-s
  by (metis DA-notin-tl DenyAllFromTo EX-MR mrDA-tl
    not-Some-eq tl-eq wellformed-policy1-strong.simps(2))
thus ?thesis using mr-p-is-DAFT by simp
}

{case (AllowPortFromTo a b c)
  have wp1s: wellformed-policy1 s by (metis wp1-eq wp1-s)
  have mr-p-is-A: applied-rule-rev C x p = Some (AllowPortFromTo a b c)
    by (simp add: AllowPortFromTo Some)
  hence A-in-s: AllowPortFromTo a b c  $\in set s$  using sets-eq
    by (auto intro: mrSet)
  have x-in-dom-A:  $x \in dom(C(\text{AllowPortFromTo } a b c))$ 
    by (metis mr-p-is-A AllowPortFromTo mr-in-dom)
  have SCs: singleCombinators s using SC sets-eq
    by (auto intro: SCSubset)
  hence ANDs: allNetsDistinct s using AND sets-eq SC
    by (auto intro: aNDSetsEq)
  hence mr-s-is-A: applied-rule-rev C x s = Some (AllowPortFromTo a b c)
    using A-in-s wp1s mr-p-is-A AND SCs wp3-s x-in-dom-A
    by (simp add: rule-charn2)
  thus ?thesis using mr-p-is-A by simp
}
  case (Conc a b) thus ?thesis by (metis Some mr-not-Conc SC)
qed
qed

```

lemma $C\text{-eq}\text{-Sets}$:

$$\text{singleCombinators } p \implies \text{wellformed-policy1-strong } p \implies \text{wellformed-policy1-strong } s \implies$$

```

wellformed-policy3 p ==> wellformed-policy3 s ==> allNetsDistinct p ==> set p = set
s ==>
  C (list2FWpolicy p) x = C (list2FWpolicy s) x
  by(auto intro: C-eq-if-mr-eq C-eq-Sets-mr [symmetric])

lemma C-eq-sorted:
  distinct p ==> all-in-list p l ==> singleCombinators p ==> wellformed-policy1-strong p
  ==>
    wellformed-policy3 p ==> allNetsDistinct p ==>
    C (list2FWpolicy (FWNormalisationCore.sort p l)) = C (list2FWpolicy p)
    apply (rule ext)
    by (auto intro: C-eq-Sets simp: nMTSort wellformed1-alternative-sorted
        wellformed-policy3-charn wp1-eq)

lemma C-eq-sortedQ:
  distinct p ==> all-in-list p l ==> singleCombinators p ==> wellformed-policy1-strong p
  ==>
    wellformed-policy3 p ==> allNetsDistinct p ==>
    C (list2FWpolicy (qsort p l)) = C (list2FWpolicy p)
    apply (rule ext)
    apply (auto intro!: C-eq-Sets simp: nMTSortQ wellformed1-alternative-sorted distinct-sortQ
        wellformed-policy3-charn wp1-eq)
    by (metis set-qsort wellformed1-sortedQ wellformed-eq wp1-aux1aa)

lemma C-eq-RS2-mr: applied-rule-rev C x (removeShadowRules2 p)= applied-rule-rev
C x p
proof (induct p)
  case Nil thus ?case by simp
next
  case (Cons y ys) thus ?case
  proof (cases ys = [])
    case True thus ?thesis by (cases y, simp-all)
  next
    case False thus ?thesis
    proof (cases y)
      case DenyAll thus ?thesis by (simp, metis Cons DenyAll mreq-end2)
    next
      case (DenyAllFromTo a b) thus ?thesis
      by (simp, metis Cons DenyAllFromTo mreq-end2)
    next
      case (AllowPortFromTo a b p) thus ?thesis
      proof (cases DenyAllFromTo a b ∈ set ys)
        case True thus ?thesis using AllowPortFromTo Cons

```

```

apply (cases applied-rule-rev C x ys = None, simp-all)
  apply (subgoal-tac x  $\notin$  dom (C (AllowPortFromTo a b p)))
    apply (subst mrconcNone, simp-all)
    apply (simp add: applied-rule-rev-def )
    apply (rule contra-subsetD [OF allow-deny-dom])
    apply (erule mrNoneMT,simp)
    apply (metis AllowPortFromTo mrconc)
    done
next
  case False thus ?thesis using False Cons AllowPortFromTo
    by (simp, metis AllowPortFromTo Cons mreq-end2) qed
next
  case (Conc a b) thus ?thesis
    by (metis Cons mreq-end2 removeShadowRules2.simps(4))
  qed
qed
qed

lemma C-eq-None[rule-format]:
 $p \neq [] \rightarrow \text{applied-rule-rev } C x p = \text{None} \rightarrow C (\text{list2FWpolicy } p) x = \text{None}$ 
  apply (simp add: applied-rule-rev-def)
  apply (rule rev-induct, simp-all)
  apply (intro impI, simp)
  subgoal for xa xs
    apply (case-tac xs  $\neq []$ )
      apply (simp-all add: dom-def)
      done
  done

lemma C-eq-None2:
 $a \neq [] \Rightarrow b \neq [] \Rightarrow \text{applied-rule-rev } C x a = \perp \Rightarrow \text{applied-rule-rev } C x b = \perp \Rightarrow C (\text{list2FWpolicy } a) x = C (\text{list2FWpolicy } b) x$ 
  by (auto simp: C-eq-None)

lemma C-eq-RS2:
 $\text{wellformed-policy1-strong } p \Rightarrow C (\text{list2FWpolicy } (\text{removeShadowRules2 } p)) = C (\text{list2FWpolicy } p)$ 
  apply (rule ext)
  by (metis C-eq-RS2-mr C-eq-if-mr-eq wellformed-policy1-strong.simps(1) wp1n-RS2)

lemma none-MT-rulesRS2:
 $\text{none-MT-rules } C p \Rightarrow \text{none-MT-rules } C (\text{removeShadowRules2 } p)$ 
  by (auto simp: RS2Set none-MT-rulessubset)

```

```

lemma CconcNone:
  dom (C a) = {}  $\implies$  p  $\neq$  []  $\implies$  C (list2FWpolicy (a # p)) x = C (list2FWpolicy p) x
  apply (case-tac p = [], simp-all)
  apply (case-tac x  $\in$  dom (C (list2FWpolicy(p))))
  apply (metis Cdom2 list2FWpolicyconc)
  apply (metis C.simps(4) map-add-dom-app-simps(2) inSet-not-MT list2FWpolicy-conc set-empty2)
  done

lemma none-MT-rulesrd[rule-format]:
  none-MT-rules C p  $\longrightarrow$  none-MT-rules C (remdups p)
  by (induct p, simp-all)

lemma DARS3[rule-format]:
  DenyAll  $\notin$  set p  $\longrightarrow$  DenyAll  $\notin$  set (rm-MT-rules C p)
  by (induct p, simp-all)

lemma DAnMT: dom (C DenyAll)  $\neq$  {}
  by (simp add: dom-def C.simps PolicyCombinators.PolicyCombinators)

lemma DAnMT2: C DenyAll  $\neq$  Map.empty
  by (metis DAAux dom-eq-empty-conv empty-iff)

lemma wp1n-RS3[rule-format,simp]:
  wellformed-policy1-strong p  $\longrightarrow$  wellformed-policy1-strong (rm-MT-rules C p)
  by (induct p, simp-all add: DARS3 DAnMT)

lemma AILRS3[rule-format,simp]:
  all-in-list p l  $\longrightarrow$  all-in-list (rm-MT-rules C p) l
  by (induct p, simp-all)

lemma SCRS3[rule-format,simp]:
  singleCombinators p  $\longrightarrow$  singleCombinators(rm-MT-rules C p)
  apply (induct p, simp-all)
  subgoal for a p
    apply(case-tac a, simp-all)
    done
  done

lemma RS3subset: set (rm-MT-rules C p)  $\subseteq$  set p
  by (induct p, auto)

lemma ANDRS3[simp]:
  singleCombinators p  $\implies$  allNetsDistinct p  $\implies$  allNetsDistinct (rm-MT-rules C p)

```

```

using RS3subset SCRS3 aNDSubset by blast

lemma nlpaux:  $x \notin \text{dom } (C b) \implies C (a \oplus b) x = C a x$ 
by (metis C.simps(4) map-add-dom-app-simps(3))

lemma notindom[rule-format]:
 $a \in \text{set } p \implies x \notin \text{dom } (C (\text{list2FWpolicy } p)) \implies x \notin \text{dom } (C a)$ 
apply (induct p, simp-all)
by (metis CConcStartA Cdom2 domIff empty-if empty-set l2p-aux)

lemma C-eq-rd[rule-format]:
 $p \neq [] \implies C (\text{list2FWpolicy } (\text{remdups } p)) = C (\text{list2FWpolicy } p)$ 
proof (rule ext,induct p)
  case Nil thus ?case by simp
next
  case (Cons y ys) thus ?case
  proof (cases ys = [])
    case True thus ?thesis by simp
  next
    case False thus ?thesis using Cons
    apply (simp) apply (rule conjI, rule impI)
    apply (cases x ∈ dom (C (list2FWpolicy ys)))
    apply (metis Cdom2 False list2FWpolicyconc)
    apply (metis False domIff list2FWpolicyconc nlpaux notindom)
    apply (rule impI)
    apply (cases x ∈ dom (C (list2FWpolicy ys)))
    apply (subgoal-tac x ∈ dom (C (list2FWpolicy (remdups ys))))
    apply (metis Cdom2 False list2FWpolicyconc remdups-eq-nil-iff)
    apply (metis domIff)
    apply (subgoal-tac x ∉ dom (C (list2FWpolicy (remdups ys))))
    apply (metis False list2FWpolicyconc nlpaux remdups-eq-nil-iff)
    apply (metis domIff)
    done
  qed
qed
qed

lemma nMT-domMT:
 $\neg \text{not-MT } C \ p \implies p \neq [] \implies r \notin \text{dom } (C (\text{list2FWpolicy } p))$ 
proof (induct p)
  case Nil thus ?case by simp
next
  case (Cons x xs) thus ?case
  proof (simp split: if-splits)
    apply (cases xs = [],simp-all)
  qed
qed

```

```

by (metis CconcNone domIff)
qed

lemma C-eq-RS3-aux[rule-format]:
  not-MT C p ==> C (list2FWpolicy p) x = C (list2FWpolicy (rm-MT-rules C p)) x
proof (induct p)
  case Nil thus ?case by simp
next
  case (Cons y ys)
  thus ?case
  proof (cases not-MT C ys)
    case True thus ?thesis using Cons
      apply (simp) apply(rule conjI, rule impI, simp)
      apply (metis CconcNone True not-MTimpnotMT)
      apply (rule impI, simp)
      apply (cases x ∈ dom (C (list2FWpolicy ys)))
      apply (subgoal-tac x ∈ dom (C (list2FWpolicy (rm-MT-rules C ys))))
      apply (metis Cdom2 NMPrm l2p-aux not-MTimpnotMT)
      apply (simp add: domIff)
      apply (subgoal-tac x ∉ dom (C (list2FWpolicy (rm-MT-rules C ys))))
      apply (metis l2p-aux l2p-aux2 nlpaux)
      apply (metis domIff)
      done
  next
  case False thus ?thesis using Cons False
  proof (cases ys = [])
    case True thus ?thesis using Cons by (simp) (rule impI, simp)
  next
  case False thus ?thesis
    using Cons False ← not-MT C ys apply (simp)
    by (metis SR3nMT l2p-aux list2FWpolicy.simps(2) nMT-domMT nlpaux)
  qed
qed
qed

```

lemma C-eq-id:

wellformed-policy1-strong p ==> C(list2FWpolicy (insertDeny p)) = C (list2FWpolicy p)

by (rule ext) (auto intro: C-eq-if-mr-eq elim: mr-iD)

lemma C-eq-RS3:

not-MT C p ==> C(list2FWpolicy (rm-MT-rules C p)) = C (list2FWpolicy p)

by (rule ext) (erule C-eq-RS3-aux[symmetric])

```

lemma NMPrd[rule-format]: not-MT C p  $\longrightarrow$  not-MT C (remdups p)
  by (induct p) (auto simp: NMPcharr)

lemma NMPDA[rule-format]: DenyAll ∈ set p  $\longrightarrow$  not-MT C p
  by (induct p, simp-all add: DAnMT)

lemma NMPiD[rule-format]: not-MT C (insertDeny p)
  by (simp add: DAiniD NMPDA)

lemma list2FWpolicy2list[rule-format]: C (list2FWpolicy(policy2list p)) = (C p)
  apply (rule ext)
  apply (induct-tac p, simp-all)
  by (metis (no-types, lifting) Cdom2 CeqEnd CeqStart domIff nlpaux p2lNmt)

lemmas C-eq-Lemmas = none-MT-rulesRS2 none-MT-rulesrd SCp2l wp1n-RS2
wp1ID NMPiD wp1-eq
wp1alternative-RS1 p2lNmt list2FWpolicy2list wellformed-policy3-charn
waux2

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

lemma C-eq-All-untilSorted:
  DenyAll ∈ set(policy2list p)  $\Longrightarrow$  all-in-list(policy2list p) l  $\Longrightarrow$  allNetsDistinct(policy2list p)  $\Longrightarrow$ 
  C (list2FWpolicy
    (FWNormalisationCore.sort
      (removeShadowRules2 (remdups (rm-MT-rules C
        (insertDeny (removeShadowRules1 (policy2list p)))))) l)) =
  C p
  apply (subst C-eq-sorted,simp-all add: C-eq-Lemmas)
  apply (subst C-eq-RS2,simp-all add: C-eq-Lemmas)
  apply (subst C-eq-rd,simp-all add: C-eq-Lemmas)
  apply (subst C-eq-RS3,simp-all add: C-eq-Lemmas)
  apply (subst C-eq-id,simp-all add: C-eq-Lemmas)
  done

lemma C-eq-All-untilSortedQ:
  DenyAll ∈ set(policy2list p)  $\Longrightarrow$  all-in-list(policy2list p) l  $\Longrightarrow$  allNetsDistinct(policy2list p)  $\Longrightarrow$ 
  C (list2FWpolicy
    (qsort (removeShadowRules2 (remdups (rm-MT-rules C
      (insertDeny (removeShadowRules1 (policy2list p)))))) l)) =
  C p

```

```

apply (subst C-eq-sortedQ,simp-all add: C-eq-Lemmas)
apply (subst C-eq-RS2,simp-all add: C-eq-Lemmas)
apply (subst C-eq-rd,simp-all add: C-eq-Lemmas)
apply (subst C-eq-RS3,simp-all add: C-eq-Lemmas)
apply (subst C-eq-id,simp-all add: C-eq-Lemmas)
done

lemma C-eq-All-untilSorted-withSimps:
  DenyAll ∈ set(policy2list p)  $\Rightarrow$  all-in-list(policy2list p) l  $\Rightarrow$  allNetsDistinct (policy2list p)  $\Rightarrow$ 
    C (list2FWpolicy
      (FWNormalisationCore.sort
        (removeShadowRules2 (remdup (rm-MT-rules C
          (insertDeny (removeShadowRules1 (policy2list p)))))) l)) =
    C p
  by (simp-all add: C-eq-Lemmas C-eq-subst-Lemmas)

lemma C-eq-All-untilSorted-withSimpsQ:
  DenyAll ∈ set(policy2list p)  $\Rightarrow$  all-in-list(policy2list p) l  $\Rightarrow$  allNetsDistinct (policy2list p)  $\Rightarrow$ 
    C (list2FWpolicy
      (qsort (removeShadowRules2 (remdup (rm-MT-rules C
        (insertDeny (removeShadowRules1 (policy2list p)))))) l)) =
    C p
  by (simp-all add: C-eq-Lemmas C-eq-subst-Lemmas)

lemma InDomConc[rule-format]:
  p ≠ []  $\rightarrow$  x ∈ dom (C (list2FWpolicy (p)))  $\rightarrow$  x ∈ dom (C (list2FWpolicy (a#p)))
  apply (induct p, simp-all)
  subgoal for a' p
    apply (case-tac p = [], simp-all add: dom-def C.simps)
    done
  done

lemma not-in-member[rule-format]: member a b  $\rightarrow$  x ∉ dom (C b)  $\rightarrow$  x ∉ dom (C a)
  by (induct b) (simp-all add: dom-def C.simps)

lemma src-in-sdnets[rule-format]:
   $\neg$  member DenyAll x  $\rightarrow$  p ∈ dom (C x)  $\rightarrow$  subnetsOfAdr (src p) ∩ (fst-set (sdnets x)) ≠ {}
  apply (induct rule: Combinators.induct)
  apply (simp-all add: fst-set-def subnetsOfAdr-def PLemmas fst-set-def)

```

```

apply (intro impI)
subgoal for x1 x2
  apply (case-tac p ∈ dom (C x2))
    apply (rule subnetAux)
      apply (auto simp: PLemmas fst-set-def)
    done
  done

lemma dest-in-sdnets[rule-format]:
  ¬ member DenyAll x → p ∈ dom (C x) → subnetsOfAdr (dest p) ∩ (snd-set (sdnets x)) ≠ {}
  apply (induct rule: Combinators.induct)
    apply (simp-all add: snd-set-def subnetsOfAdr-def PLemmas)
  apply (intro impI)
  apply (simp add: snd-set-def)
  subgoal for x1 x2
    apply (case-tac p ∈ dom (C x2))
      apply (rule subnetAux)
        apply (auto simp: PLemmas)
      done
    done

lemma sdnets-in-subnets[rule-format]:
  p ∈ dom (C x) → ¬ member DenyAll x →
  (∃ (a,b)∈sdnets x. a ∈ subnetsOfAdr (src p) ∧ b ∈ subnetsOfAdr (dest p))
  apply (rule Combinators.induct)
    apply (simp-all add: PLemmas subnetsOfAdr-def)
  apply (intro impI, simp)
  subgoal for x1 x2
    apply (case-tac p ∈ dom (C (x2)))
      apply (auto simp: PLemmas subnetsOfAdr-def)
      done
    done

lemma disjSD-no-p-in-both[rule-format]:
  disjSD-2 x y ⇒ ¬ member DenyAll x ⇒ ¬ member DenyAll y ⇒ p ∈ dom(C x)
  ⇒ p ∈ dom(C y) ⇒
  False
  apply (rule-tac A = sdnets x and B = sdnets y and D = src p and F = dest p in
  tndFalse)
  by (auto simp: dest-in-sdnets src-in-sdnets sdnets-in-subnets disjSD-2-def)

lemma list2FWpolicy-eq:
  zs ≠ [] ⇒ C (list2FWpolicy (x ⊕ y # z)) p = C (x ⊕ list2FWpolicy (y # z)) p

```

```

by (metis ConcAssoc l2p-aux list2FWpolicy.simps(2))

lemma dom-sep[rule-format]:
   $x \in \text{dom } (C (\text{list2FWpolicy } p)) \longrightarrow x \in \text{dom } (C (\text{list2FWpolicy}(\text{separate } p)))$ 
proof (induct p rule: separate.induct, simp-all, goal-cases)
  case (1 v va y z) then show ?case
    apply (intro conjI impI)
    apply (simp, drule mp)
    apply (case-tac x ∈ dom (C (DenyAllFromTo v va)))
    apply (metis CConcStartA domIff l2p-aux2 list2FWpolicyconc not-Cons-self )
    apply (metis Conc-not-MT domIff list2FWpolicy-eq, simp)
    by (metis InDomConc domIff list.simps(3) list2FWpolicyconc nlpaux sepnMT)
  next
  case (2 v va vb y z)
  assume * : {v, va} = first-bothNet y ⟹
     $x \in \text{dom } (C (\text{list2FWpolicy} (\text{AllowPortFromTo } v \text{ va vb} \oplus y \# z))) \longrightarrow$ 
     $x \in \text{dom } (C (\text{list2FWpolicy} (\text{separate} (\text{AllowPortFromTo } v \text{ va vb} \oplus y \# z))))$ 
  and **: {v, va} ≠ first-bothNet y ⟹
     $x \in \text{dom}(C(\text{list2FWpolicy}(y\#z))) \longrightarrow x \in \text{dom}$ 
    ( $C(\text{list2FWpolicy}(\text{separate}(y\#z)))$ )
  show ?case
    apply (insert * **, rule conjI)+
    apply (simp, case-tac x ∈ dom (C (AllowPortFromTo v va vb)))
    apply (metis CConcStartA domIff l2p-aux2 list2FWpolicyconc not-Cons-self )
    apply (subgoal-tac x ∈ dom (C (list2FWpolicy (y # z))))
    apply (metis CConcStartA Cdom2 domIff l2p-aux2 list2FWpolicyconc nlpaux)
    apply (simp add: dom-def C.simps)
    apply (intro impI, simp-all)
    apply (case-tac x ∈ dom (C (AllowPortFromTo v va vb)), simp-all)
    by (metis Cdom2 domIff l2p-aux list2FWpolicy.simps(3) nlpaux sepnMT)
  next
  case (3 v va y z)
  assume * : (first-bothNet v = first-bothNet y ⟹
     $x \in \text{dom } (C (\text{list2FWpolicy} ((v \oplus va) \oplus y \# z))) \longrightarrow$ 
     $x \in \text{dom } (C (\text{list2FWpolicy} (\text{separate} ((v \oplus va) \oplus y \# z))))$ )
  and ** : (first-bothNet v ≠ first-bothNet y ⟹
     $x \in \text{dom}(C(\text{list2FWpolicy}(y\#z))) \longrightarrow x \in \text{dom } (C (\text{list2FWpolicy} (\text{separate} (y \# z))))$ )
  show ?case
    apply (insert * **, rule conjI | rule impI)+
    apply (simp, drule mp)
    apply (case-tac x ∈ dom (C ((v ⊕ va))))
    apply (metis C.simps(4) CConcStartA ConcAssoc domIff list2FWpolicy2list

```

```

list2FWpolicyconc p2lNmt)
  apply simp-all
  apply (metis Conc-not-MT domIff list2FWpolicy-eq)
  by (metis CConcStartA Conc-not-MT InDomConc domIff nlpaux sepnMT)
qed

lemma domdConcStart[rule-format]:
   $x \in \text{dom} (C (\text{list2FWpolicy} (a \# b))) \rightarrow x \notin \text{dom} (C (\text{list2FWpolicy} b)) \rightarrow x \in \text{dom} (C (a))$ 
  by (induct b, simp-all) (auto simp: PLemmas)

lemma sep-dom2-aux:
   $x \in \text{dom} (C (\text{list2FWpolicy} (a \oplus y \# z))) \implies x \in \text{dom} (C (a \oplus \text{list2FWpolicy} (y \# z)))$ 
  by (auto)[1] (metis list2FWpolicy-eq p2lNmt)

lemma sep-dom2-aux2:
   $x \in \text{dom} (C (\text{list2FWpolicy} (\text{separate} (y \# z)))) \rightarrow x \in \text{dom} (C (\text{list2FWpolicy} (y \# z))) \implies$ 
   $x \in \text{dom} (C (\text{list2FWpolicy} (a \# \text{separate} (y \# z)))) \implies x \in \text{dom} (C (\text{list2FWpolicy} (a \oplus y \# z)))$ 
  by (metis CConcStartA InDomConc domdConcStart list.simps(2) list2FWpolicy.simps(2) list2FWpolicyconc)

lemma sep-dom2[rule-format]:
   $x \in \text{dom} (C (\text{list2FWpolicy} (\text{separate} p))) \rightarrow x \in \text{dom} (C (\text{list2FWpolicy} (p)))$ 
  by (rule separate.induct) (simp-all add: sep-dom2-aux sep-dom2-aux2)

lemma sepDom:  $\text{dom} (C (\text{list2FWpolicy} p)) = \text{dom} (C (\text{list2FWpolicy} (\text{separate} p)))$ 
  apply (rule equalityI)
  by (rule subsetI, (erule dom-sep|erule sep-dom2))+

lemma C-eq-s-ext[rule-format]:
   $p \neq [] \rightarrow C (\text{list2FWpolicy} (\text{separate} p)) a = C (\text{list2FWpolicy} p) a$ 
proof (induct rule: separate.induct, goal-cases)
  case (1 x) thus ?case
    apply simp
    apply (cases x = [])
    apply (metis l2p-aux2 separate.simps(5))
    apply simp
    apply (cases a ∈ dom (C (list2FWpolicy x)))
    apply (subgoal-tac a ∈ dom (C (list2FWpolicy (separate x))))
    apply (metis Cdom2 list2FWpolicyconc sepDom sepnMT)
    apply (metis sepDom)

```

```

apply (subgoal-tac a  $\notin$  dom (C (list2FWpolicy (separate x)))))

apply (subst list2FWpolicyconc,simp add: sepMT)
apply (subst list2FWpolicyconc,simp add: sepMT)
apply (metis nlpaux sepDom)
apply (metis sepDom)
done

next
case ( $\lambda v \text{ va } y \text{ z}$ ) thus ?case
apply (cases z =  $\emptyset$ , simp-all)
apply (rule conjI|rule impI|simp)+
apply (subst list2FWpolicyconc)
apply (metis not-Cons-self sepMT)
apply (metis C.simps(4) CConcStartaux Cdom2 domIff)
apply (rule conjI|rule impI|simp)+
apply (erule list2FWpolicy-eq)
apply (rule impI, simp)
apply (subst list2FWpolicyconc)
apply (metis list.simps(2) sepMT)
by (metis C.simps(4) CConcStartaux Cdom2 domIff)
next
case ( $\lambda v \text{ va } vb \text{ y } z$ ) thus ?case
apply (cases z =  $\emptyset$ , simp-all)
apply (rule conjI|rule impI|simp)+
apply (subst list2FWpolicyconc)
apply (metis not-Cons-self sepMT)
apply (metis C.simps(4) CConcStartaux Cdom2 domIff)
apply (rule conjI|rule impI|simp)+
apply (erule list2FWpolicy-eq)
apply (rule impI, simp)
apply (subst list2FWpolicyconc)
apply (metis list.simps(2) sepMT)
by (metis C.simps(4) CConcStartaux Cdom2 domIff)
next
case ( $\lambda v \text{ va } y \text{ z}$ ) thus ?case
apply (cases z =  $\emptyset$ , simp-all)
apply (rule conjI|rule impI|simp)+
apply (subst list2FWpolicyconc)
apply (metis not-Cons-self sepMT)
apply (metis C.simps(4) CConcStartaux Cdom2 domIff)
apply (rule conjI|rule impI|simp)+
apply (erule list2FWpolicy-eq)
apply (rule impI, simp)
apply (subst list2FWpolicyconc)
apply (metis list.simps(2) sepMT)

```

```

by (metis C.simps(4) CConcStartaux Cdom2 domIff)
next
  case 5 thus ?case by simp
next
  case 6 thus ?case by simp
next
  case 7 thus ?case by simp
next
  case 8 thus ?case by simp
qed

lemma C-eq-s:
 $p \neq [] \implies C(\text{list2FWpolicy}(\text{separate } p)) = C(\text{list2FWpolicy } p)$ 
apply (rule ext) using C-eq-s-ext by blast

lemma sortnMTQ:  $p \neq [] \implies \text{qsort } p \neq []$ 
  by (metis set-sortQ setnMT)

lemmas C-eq-Lemmas-sep =
  C-eq-Lemmas sortnMT sortnMTQ RS2-NMT NMPrd not-MTimpnotMT

lemma C-eq-until-separated:
 $\text{DenyAll} \in \text{set}(\text{policy2list } p) \implies \text{all-in-list}(\text{policy2list } p)l \implies \text{allNetsDistinct}(\text{policy2list } p) \implies$ 
 $C(\text{list2FWpolicy}(\text{separate}(\text{FWNormalisationCore.sort}(\text{removeShadowRules2}(\text{remdups}(\text{rm-MT-rules } C(\text{insertDeny}(\text{removeShadowRules1}(\text{policy2list } p))))))))) =$ 
 $C p$ 
  by (simp add: C-eq-All-untilSorted-withSimps C-eq-s wellformed1-alternative-sorted wp1ID wp1n-RS2)

lemma C-eq-until-separatedQ:
 $\text{DenyAll} \in \text{set}(\text{policy2list } p) \implies \text{all-in-list}(\text{policy2list } p)l \implies \text{allNetsDistinct}(\text{policy2list } p) \implies$ 
 $C(\text{list2FWpolicy}(\text{separate}(\text{qsort}(\text{removeShadowRules2}(\text{remdups}(\text{rm-MT-rules } C(\text{insertDeny}(\text{removeShadowRules1}(\text{policy2list } p))))))))) =$ 
 $C p$ 
  by (simp add: C-eq-All-untilSorted-withSimpsQ C-eq-s sortnMTQ wp1ID wp1n-RS2)

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

```

```

proof(induct p)
  case Nil then show ?case by simp
next
  case (Cons a p) then show ?case
  proof(cases p=[],goal-cases)
    case 1 then show ?case
      apply(simp) apply(rule impI)
      apply(cases a, simp-all)
      apply(simp-all add: C.simps dom-def)+
      by auto
next
  case 2 then show ?case
  proof(cases x ∈ dom(C(list2FWpolicy p)), goal-cases)
    case 1 then show ?case
      apply simp apply(rule impI)
      apply(cases a, simp-all)
      using InDomConc idNMT apply blast
      apply(rule InDomConc, simp-all add: idNMT)+
      done
next
  case 2 then show ?case
    apply simp apply(rule impI)
  proof(cases x ∈ dom(C(list2FWpolicy(insertDenies p))), goal-cases)
    case 1 then show ?case
    proof(induct a)
      case DenyAll then show ?case by simp
    next
      case (DenyAllFromTo src dest) then show ?case
        apply simp by(rule InDomConc, simp add: idNMT)
    next
      case (AllowPortFromTo src dest port) then show ?case
        apply simp by(rule InDomConc, simp add: idNMT)
    next
      case (Conc - -) then show ?case
        apply simp by(rule InDomConc, simp add: idNMT)
    qed
next
  case 2 then show ?case
  proof(induct a)
    case DenyAll then show ?case by simp
  next
    case (DenyAllFromTo src dest) then show ?case
      by(simp,metis domIff CConcStartA list2FWpolicyconc nlpaux Cdom2)
  next

```

```

case (AllowPortFromTo src dest port) then show ?case
  by(simp,metis domIff CConcStartA list2FWpolicyconc nlpaux Cdom2)
next
  case (Conc - -) then show ?case
    by (simp,metis CConcStartA Cdom2 domIff domdConcStart)
  qed
  qed
  qed
  qed
  qed
lemma DA-is-deny:

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

  apply (case-tac  $x \in \text{dom} (C (\text{DenyAllFromTo } a b))$ )
  apply (simp-all add: PLemmas)
  apply (simp-all split: if-splits)
done

lemma iDdomAux[rule-format]:

$$p \neq [] \longrightarrow x \notin \text{dom} (C (\text{list2FWpolicy } p)) \longrightarrow x \in \text{dom} (C (\text{list2FWpolicy } (\text{insertDenies } p))) \longrightarrow C (\text{list2FWpolicy } (\text{insertDenies } p)) x = \text{Some} (\text{deny} ())$$

proof (induct p)
  case Nil thus ?case by simp
next
  case (Cons y ys) thus ?case
  proof (cases y)
    case DenyAll then show ?thesis by simp
  next
    case (DenyAllFromTo a b) then show ?thesis using DenyAllFromTo Cons
      apply simp
      apply (intro impI)
      proof (cases ys = [], goal-cases)
        case 1 then show ?case by (simp add: DA-is-deny)
      next
        case 2 then show ?case
          apply simp
          apply (drule mp)
          apply (metis DenyAllFromTo InDomConc )
          apply (cases x  $\in$   $\text{dom} (C (\text{list2FWpolicy } (\text{insertDenies } ys)))$ , simp-all)
            apply (metis Cdom2 DenyAllFromTo idNMT list2FWpolicyconc)
          apply (subgoal-tac C (list2FWpolicy (DenyAllFromTo a b  $\oplus$ 

```

```

 $DenyAllFromTo b a \oplus DenyAllFromTo a b \# insertDenies ys)$ )
x =
C ((DenyAllFromTo a b \oplus DenyAllFromTo b a \oplus DenyAllFromTo
a b)) x )
apply simp
apply (rule DA-is-deny)
apply (metis DenyAllFromTo domdConcStart)
apply (metis DenyAllFromTo l2p-aux2 list2FWpolicyconc nlpaux)
done
qed
next
case (AllowPortFromTo a b c) then show ?thesis using Cons AllowPortFromTo
proof (cases ys = [], goal-cases)
case 1 then show ?case
apply simp
apply (intro impI)
apply (subgoal-tac x \in dom (C (DenyAllFromTo a b \oplus DenyAllFromTo b a)))
apply (simp-all add: PLemmas)
apply (simp split: if-splits, auto)
done
next
case 2 then show ?case
apply simp
apply (intro impI)
apply (drule mp)
apply (metis AllowPortFromTo InDomConc)
apply (cases x \in dom (C (list2FWpolicy (insertDenies ys))))
apply simp-all
apply (metis AllowPortFromTo Cdom2 idNMT list2FWpolicyconc)
apply (subgoal-tac C (list2FWpolicy (DenyAllFromTo a b \oplus
DenyAllFromTo b a \oplus AllowPortFromTo a b c \# insertDenies
ys))) x =
C ((DenyAllFromTo a b \oplus DenyAllFromTo b a)) x )
apply simp
defer 1
apply (metis AllowPortFromTo CConcStartA ConcAssoc idNMT list2FWpoli-
cyconc nlpaux)
apply (simp add: PLemmas, simp split: if-splits, auto)
done
qed
next
case (Conc a b) then show ?thesis
proof (cases ys = [], goal-cases)
case 1 then show ?case

```

```

apply simp
apply (rule impI)+
apply (subgoal-tac  $x \in \text{dom } (C (\text{DenyAllFromTo } (\text{first-srcNet } a)$ 
 $(\text{first-destNet } a) \oplus \text{DenyAllFromTo } (\text{first-destNet } a) (\text{first-srcNet }$ 
 $a)))$ )
apply (simp-all add: PLemmas)
apply (simp split: if-splits, auto)
done
next
case 2 then show ?case
apply simp
apply (intro impI)
apply (cases  $x \in \text{dom } (C (\text{list2FWpolicy } (\text{insertDenies } ys)))$ )
apply (metis Cdom2 Conc Cons InDomConc idNMT list2FWpolicyconc)
apply (subgoal-tac  $C (\text{list2FWpolicy } (\text{DenyAllFromTo } (\text{first-srcNet } a)$ 
 $(\text{first-destNet } a) \oplus \text{DenyAllFromTo } (\text{first-destNet } a) (\text{first-srcNet }$ 
 $a)$ )
 $\oplus a \oplus b \# \text{insertDenies } ys)) x =$ 
 $C ((\text{DenyAllFromTo } (\text{first-srcNet } a) (\text{first-destNet } a) \oplus$ 
 $\text{DenyAllFromTo } (\text{first-destNet } a) (\text{first-srcNet } a) \oplus a \oplus$ 
 $b)) x$ )
apply simp
defer 1
apply (metis Conc l2p-aux2 list2FWpolicyconc nlpaux)
apply (subgoal-tac  $C((\text{DenyAllFromTo } (\text{first-srcNet } a)$ 
 $(\text{first-destNet } a) \oplus \text{DenyAllFromTo } (\text{first-destNet } a)$ 
 $(\text{first-srcNet } a) \oplus a \oplus b)) x =$ 
 $C((\text{DenyAllFromTo } (\text{first-srcNet } a)(\text{first-destNet } a) \oplus$ 
 $\text{DenyAllFromTo } (\text{first-destNet } a)(\text{first-srcNet } a))) x$ )
apply simp
defer 1
apply (metis CConcStartA Conc ConcAssoc nlpaux)
apply (simp add: PLemmas, simp split: if-splits, auto)
done
qed
qed
qed

```

lemma iD-isD[rule-format]:

```

 $p \neq [] \rightarrow x \notin \text{dom } (C (\text{list2FWpolicy } p)) \rightarrow$ 
 $C (\text{DenyAll} \oplus \text{list2FWpolicy } (\text{insertDenies } p)) x = C \text{ DenyAll } x$ 
apply (case-tac  $x \in \text{dom } (C (\text{list2FWpolicy } (\text{insertDenies } p)))$ )
apply (simp add: Cdom2 PLemmas(1) deny-all-def iDdomAux)

```

by (*simp add: nlpaux*)

```

lemma inDomConc:[ $x \notin \text{dom } (C a); x \notin \text{dom } (C (\text{list2FWpolicy } p))$ ]  $\implies$ 
     $x \notin \text{dom } (C (\text{list2FWpolicy}(a \# p)))$ 
by (metis domdConcStart)

```

```

lemma domsdisj[rule-format]:
     $p \neq [] \implies (\forall x s. s \in \text{set } p \wedge x \in \text{dom } (C A) \implies x \notin \text{dom } (C s)) \implies y \in \text{dom } (C A) \implies y \notin \text{dom } (C (\text{list2FWpolicy } p))$ 
proof (induct p)
    case Nil show ?case by simp
next
    case (Cons a p) then show ?case
        apply (case-tac  $p = []$ )
        apply fastforce
        by (meson domdConcStart list.set-intros(1) list.set-intros(2))
qed

```

```

lemma isSepaux:
     $p \neq [] \implies \text{noDenyAll } (a \# p) \implies \text{separated } (a \# p) \implies$ 
     $x \in \text{dom } (C (\text{DenyAllFromTo } (\text{first-srcNet } a) (\text{first-destNet } a) \oplus$ 
         $\text{DenyAllFromTo } (\text{first-destNet } a) (\text{first-srcNet } a) \oplus a)) \implies$ 
     $x \notin \text{dom } (C (\text{list2FWpolicy } p))$ 
    apply (rule-tac  $A = (\text{DenyAllFromTo } (\text{first-srcNet } a) (\text{first-destNet } a) \oplus$ 
         $\text{DenyAllFromTo } (\text{first-destNet } a) (\text{first-srcNet } a) \oplus a)$  in domsdisj)
    apply simp-all
    by (metis Combinators.distinct(1) FWNormalisationCore.member.simps(1)
        FWNormalisationCore.member.simps(3) disjSD2aux disjSD-no-p-in-both noDA)

```

```

lemma none-MT-rulessep[rule-format]: none-MT-rules C p  $\implies$  none-MT-rules C
(separate p)
apply (induct p rule: separate.induct)
by (simp-all add: C.simps map-add-le-mapE map-le-antisym)

```

```

lemma dom-id:
     $\text{noDenyAll } (a \# p) \implies \text{separated } (a \# p) \implies p \neq [] \implies x \notin \text{dom } (C (\text{list2FWpolicy } p)) \implies$ 
 $x \in \text{dom } (C a) \implies$ 
     $x \notin \text{dom } (C (\text{list2FWpolicy } (\text{insertDenies } p)))$ 
    apply (rule-tac  $a = a$  in isSepaux, simp-all)
    using idNMT apply blast
    using noDAID apply blast
    using id-aux4 noDA1eq sepNetsID apply blast

```

```

by (metis list.set-intros(1) list.set-intros(2) list2FWpolicy.simps(2) list2FWpolicy.simps(3) notindom)

lemma C-eq-iD-aux2[rule-format]:
  noDenyAll1 p --> separated p --> p ≠ [] --> x ∈ dom (C (list2FWpolicy p)) -->
  C(list2FWpolicy (insertDenies p)) x = C(list2FWpolicy p) x
proof (induct p)
  case Nil thus ?case by simp
next
  case (Cons y ys) thus ?case using Cons
  proof (cases y)
    case DenyAll thus ?thesis using Cons DenyAll apply simp
      apply (case-tac ys = [], simp-all)
      apply (case-tac x ∈ dom (C (list2FWpolicy ys)), simp-all)
      apply (metis Cdom2 domID idNMT list2FWpolicyconc noDA1eq)
      apply (metis DenyAll iD-isD idNMT list2FWpolicyconc nlpaux)
      done
  next
  case (DenyAllFromTo a b) thus ?thesis using Cons apply simp
    apply (rule impI|rule allI|rule conjI|simp)+
    apply (case-tac ys = [], simp-all)
    apply (metis Cdom2 ConcAssoc DenyAllFromTo)
    apply (case-tac x ∈ dom (C (list2FWpolicy ys)), simp-all)
    apply (simp add: Cdom2 domID idNMT l2p-aux noDA1eq)
    apply (case-tac x ∈ dom (C (list2FWpolicy (insertDenies ys)))))
    apply (meson Combinators.distinct(1) FWNormalisationCore.member.simps(3))
  dom-id domdConcStart
    noDenyAll.simps(1) separated.simps(1))
  by (metis Cdom2 DenyAllFromTo domIff dom-def domdConcStart l2p-aux l2p-aux2
  nlpaux)
  next
  case (AllowPortFromTo a b c) thus ?thesis
    using AllowPortFromTo Cons apply simp
    apply (rule impI|rule allI|rule conjI|simp)+
    apply (case-tac ys = [], simp-all)
    apply (metis Cdom2 ConcAssoc AllowPortFromTo)
    apply (case-tac x ∈ dom (C (list2FWpolicy ys)), simp-all)
    apply (simp add: Cdom2 domID idNMT list2FWpolicyconc noDA1eq)
    apply (case-tac x ∈ dom (C (list2FWpolicy (insertDenies ys)))))
    apply (meson Combinators.distinct(3) FWNormalisationCore.member.simps(4))
  dom-id domdConcStart noDenyAll.simps(1) separated.simps(1))
  by (metis Cdom2 ConcAssoc l2p-aux list2FWpolicy.simps(2) nlpaux)
  next
  case (Conc a b) thus ?thesis using Cons Conc

```

```

apply simp
apply (rule impI|rule allI|rule conjI|simp)+
apply (case-tac ys = [], simp-all)
apply (metis Cdom2 ConcAssoc Conc)
apply (case-tac x ∈ dom (C (list2FWpolicy ys)),simp-all)
apply (simp add: Cdom2 domID idNMT list2FWpolicyconc noDA1eq)
apply (case-tac x ∈ dom (C (a ⊕ b)))
apply (case-tac x ∉ dom (C (list2FWpolicy (insertDenies ys))),simp-all)
apply (simp add: Cdom2 domIff idNMT list2FWpolicyconc nlpaux)
apply (metis FWNormalisationCore.member.simps(1) dom-id noDenyAll.simps(1)
separated.simps(1))
by (simp add: inDomConc)
qed
qed

lemma C-eq-iD:
separated p  $\implies$  noDenyAll1 p  $\implies$  wellformed-policy1-strong p  $\implies$ 
C (list2FWpolicy (insertDenies p)) = C (list2FWpolicy p)
by (rule ext) (metis CConcStartA C-eq-iD-aux2 DAAux wp1-alternative-not-mt
wp1n-tl)

lemma noDAsortQ[rule-format]: noDenyAll1 p  $\longrightarrow$  noDenyAll1 (qsort p l)
apply (case-tac p,simp-all, rename-tac a list)
subgoal for a list
apply (case-tac a = DenyAll,simp-all)
using nDAeqSet set-sortQ apply blast
apply (rule impI,rule noDA1eq)
apply (subgoal-tac noDenyAll (a#list))
apply (metis append-Cons append-Nil nDAeqSet qsort.simps(2) set-sortQ)
by (case-tac a, simp-all)
done

lemma NetsCollectedSortQ:
distinct p  $\implies$  noDenyAll1 p  $\implies$  all-in-list p l  $\implies$  singleCombinators p  $\implies$ 
NetsCollected (qsort p l)
by (metis NetsCollectedSorted SC3Q all-in-list.elims(2) all-in-list.simps(1)
all-in-list.simps(2)
all-in-listAppend all-in-list-sublist noDAsortQ qsort.simps(1) qsort.simps(2)
singleCombinatorsConc sort-is-sortedQ)

lemmas CLemmas = nMTSort nMTSortQ none-MT-rulesRS2 none-MT-rulesrd
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:*

DenyAll \in *set(policy2list p)* \implies *all-in-list(policy2list p)l* \implies *allNetsDistinct(policy2list p)* \implies
C (list2FWpolicy (insertDenies (separate (FWNormalisationCore.sort (removeShadowRules2 (remdups (rm-MT-rules C (insertDeny (removeShadowRules1 (policy2list p)))))) l)))) = C p
apply (*subst C-eq-iD, simp-all add: C-eqLemmas-id*)
apply (*rule C-eq-until-separated, simp-all*)
done

lemma *C-eq-Until-InsertDeniesQ:*

DenyAll \in *set(policy2list p)* \implies *all-in-list(policy2list p)l* \implies *allNetsDistinct(policy2list p)* \implies
C(list2FWpolicy (insertDenies (separate (qsort (removeShadowRules2 (remdups (rm-MT-rules C (insertDeny (removeShadowRules1 (policy2list p)))))) l)))) = C p
apply (*subst C-eq-iD, simp-all add: C-eqLemmas-id*)
apply (*metis WP1rd set-qsort wellformed1-sortedQ wellformed-eq wp1ID wp1-alternativesep wp1-aux1aa wp1n-RS2 wp1n-RS3*)
by (*rule C-eq-until-separatedQ, simp-all*)

lemma *C-eq-RD-aux[rule-format]: C (p) x = C (removeDuplicates p) x*

apply (*induct p, simp-all*)
by (*metis Cdom2 domIff nlpaux not-in-member*)

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

p $\neq \emptyset$ \longrightarrow *C (list2FWpolicy p) x = C (list2FWpolicy (removeAllDuplicates p)) x*
proof (*induct p*)
case *Nil* **show** ?case **by** *simp*
next
case (*Cons a p*) **show** ?case

```

apply (case-tac  $p = []$ , simp-all)
  apply (metis C-eq-RD-aux)
  apply (subst list2FWpolicyconc,simp)
  apply (case-tac  $x \in \text{dom}(C(\text{list2FWpolicy } p))$ )
    apply (simp add: Cdom2 Cons.hyps domIff l2p-aux rADnMT)
  by (metis C-eq-RD-aux Cons.hyps domIff list2FWpolicyconc nlpaux rADnMT)
qed

lemma C-eq-RAD:
 $p \neq [] \implies C(\text{list2FWpolicy } p) = C(\text{list2FWpolicy}(\text{removeAllDuplicates } p))$ 
by (rule ext,erule C-eq-RAD-aux)

lemma C-eq-compile:
 $\text{DenyAll} \in \text{set}(\text{policy2list } p) \implies \text{all-in-list}(\text{policy2list } p)l \implies \text{allNetsDistinct}(\text{policy2list } p)$ 
 $\implies C(\text{list2FWpolicy}(\text{removeAllDuplicates}(\text{insertDenies}(\text{separate}(\text{FWNormalisationCore.sort}(\text{removeShadowRules2}(\text{remdups}(\text{rm-MT-rules } C(\text{insertDeny}(\text{removeShadowRules1}(\text{policy2list } p))))))) l)))) = C p$ 
apply (subst C-eq-RAD[symmetric])
  apply (rule idNMT,simp add: C-eqLemmas-id)
by (rule C-eq-Until-InsertDenies, simp-all)

lemma C-eq-compileQ:
 $\text{DenyAll} \in \text{set}(\text{policy2list } p) \implies \text{all-in-list}(\text{policy2list } p)l \implies \text{allNetsDistinct}(\text{policy2list } p)$ 
 $\implies C(\text{list2FWpolicy}(\text{removeAllDuplicates}(\text{insertDenies}(\text{separate}(\text{qsort}(\text{removeShadowRules2}(\text{remdups}(\text{rm-MT-rules } C(\text{insertDeny}(\text{removeShadowRules1}(\text{policy2list } p))))))) l)))) = C p$ 
apply (subst C-eq-RAD[symmetric],rule idNMT)
  apply (metis WP1rd sepnMT sortnMTQ wellformed-policy1-strong.simps(1) wp1ID wp1n-RS2 wp1n-RS3)
by (rule C-eq-Until-InsertDeniesQ, simp-all)

lemma C-eq-normalize:
 $\text{DenyAll} \in \text{set}(\text{policy2list } p) \implies \text{allNetsDistinct}(\text{policy2list } p) \implies$ 
 $\text{all-in-list}(\text{policy2list } p)(\text{Nets-List } p) \implies$ 

```

$C (\text{list2FWpolicy} (\text{normalize } p)) = C p$

unfolding `normalize-def`

by (`simp add: C-eq-compile`)

lemma `C-eq-normalizeQ`:

$\text{DenyAll} \in \text{set} (\text{policy2list } p) \implies \text{allNetsDistinct} (\text{policy2list } p) \implies$

$\text{all-in-list} (\text{policy2list } p) (\text{Nets-List } p) \implies$

$C (\text{list2FWpolicy} (\text{normalizeQ } p)) = C p$

by (`simp add: normalizeQ-def C-eq-compileQ`)

lemma `domSubset3`: $\text{dom} (C (\text{DenyAll} \oplus x)) = \text{dom} (C (\text{DenyAll}))$

by (`simp add: PLemmas split-tupled-all split: option.splits`)

lemma `domSubset4`:

$\text{dom} (C (\text{DenyAllFromTo } x y \oplus \text{DenyAllFromTo } y x \oplus \text{AllowPortFromTo } x y dn)) =$

$\text{dom} (C (\text{DenyAllFromTo } x y \oplus \text{DenyAllFromTo } y x))$

by (`auto simp: PLemmas split: option.splits decision.splits`)

lemma `domSubset5`:

$\text{dom} (C (\text{DenyAllFromTo } x y \oplus \text{DenyAllFromTo } y x \oplus \text{AllowPortFromTo } y x dn)) =$

$\text{dom} (C (\text{DenyAllFromTo } x y \oplus \text{DenyAllFromTo } y x))$

by (`auto simp: PLemmas split: option.splits decision.splits`)

lemma `domSubset1`:

$\text{dom} (C (\text{DenyAllFromTo } \text{one two} \oplus \text{DenyAllFromTo } \text{two one} \oplus \text{AllowPortFromTo } \text{one two dn} \oplus x)) =$

$\text{dom} (C (\text{DenyAllFromTo } \text{one two} \oplus \text{DenyAllFromTo } \text{two one} \oplus x))$

by (`simp add: PLemmas split: option.splits decision.splits`) (`auto simp: allow-all-def deny-all-def`)

lemma `domSubset2`:

$\text{dom} (C (\text{DenyAllFromTo } \text{one two} \oplus \text{DenyAllFromTo } \text{two one} \oplus \text{AllowPortFromTo } \text{two one dn} \oplus x)) =$

$\text{dom} (C (\text{DenyAllFromTo } \text{one two} \oplus \text{DenyAllFromTo } \text{two one} \oplus x))$

by (`simp add: PLemmas split: option.splits decision.splits`) (`auto simp: allow-all-def deny-all-def`)

lemma `ConcAssoc2`: $C (X \oplus Y \oplus ((A \oplus B) \oplus D)) = C (X \oplus Y \oplus A \oplus B \oplus D)$

by (`simp add: C.simps`)

lemma `ConcAssoc3`: $C (X \oplus ((Y \oplus A) \oplus D)) = C (X \oplus Y \oplus A \oplus D)$

by (`simp add: C.simps`)

lemma `RS3-NMT[rule-format]`:

$\text{DenyAll} \in \text{set } p \rightarrow \text{rm-MT-rules } C p \neq []$
by (induct-tac p) (simp-all add: PLemmas)

lemma norm-notMT: $\text{DenyAll} \in \text{set } (\text{policy2list } p) \implies \text{normalize } p \neq []$
by (simp add: DAiniD RS2-NMT RS3-NMT idNMT normalize-def rADnMT sepNMT sortnMT)

lemma norm-notMTQ: $\text{DenyAll} \in \text{set } (\text{policy2list } p) \implies \text{normalizeQ } p \neq []$
by (simp add: DAiniD RS2-NMT RS3-NMT idNMT normalizeQ-def rADnMT sepNMT sortnMTQ)

lemmas domDA = NormalisationIntegerPortProof.domSubset3

lemmas domain-reasoning = domDA ConcAssoc2 domSubset1 domSubset2
domSubset3 domSubset4 domSubset5 domSubsetDistr1
domSubsetDistr2 domSubsetDistrA domSubsetDistrD coerc-assoc
ConcAssoc
ConcAssoc3

The following lemmas help with the normalisation

lemma list2policyR-Start[rule-format]: $p \in \text{dom } (C a) \rightarrow$
 $C (\text{list2policyR } (a \# \text{list})) p = C a p$
by (induct a # list rule:list2policyR.induct) (auto simp: C.simps dom-def map-add-def)

lemma list2policyR-End: $p \notin \text{dom } (C a) \implies$
 $C (\text{list2policyR } (a \# \text{list})) p = (C a \oplus \text{list2policy } (\text{map } C \text{ list})) p$
by (rule list2policyR.induct)
(simp-all add: C.simps dom-def map-add-def list2policy-def split: option.splits)

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

proof (induct N)
case Nil show ?case by (simp-all add: list2policy-def)

next

case (Cons a N) then show ?case
apply (case-tac p ∈ dom (C a), simp-all add: domStart list2policy-def)
apply (rule list2policyR-Start, simp-all)
apply (rule list2policyR.induct, simp-all)
apply (simp-all add: C.simps dom-def map-add-def)
apply (simp split: option.splits)
done

qed

```

lemma l2polR-eq:

$$N \neq [] \implies C(\text{list2policyR } N) = (\text{list2policy} (\text{map } C N))$$

by (auto simp: list2policy-def l2polR-eq-el)

lemma list2FWpolicys-eq-el[rule-format]:

$$\text{Filter} \neq [] \longrightarrow C(\text{list2policyR Filter}) p = C(\text{list2FWpolicy} (\text{rev Filter})) p$$

proof (induct Filter) print-cases
  case Nil show ?case by (simp)
next
  case (Cons a list) then show ?case
    apply simp-all
    apply (case-tac list = [], simp-all)
    apply (case-tac p ∈ dom (C a), simp-all)
    apply (rule list2policyR-Start, simp-all)
    by (metis C.simps(4) l2polR-eq list2policyR-End nlpaux)
qed

lemma list2FWpolicys-eq:

$$\text{Filter} \neq [] \implies C(\text{list2policyR Filter}) = C(\text{list2FWpolicy} (\text{rev Filter}))$$

by (rule ext, erule list2FWpolicys-eq-el)

lemma list2FWpolicys-eq-sym:

$$\text{Filter} \neq [] \implies C(\text{list2policyR} (\text{rev Filter})) = C(\text{list2FWpolicy Filter})$$

by (metis list2FWpolicys-eq rev-is-Nil-conv rev-rev-ident)

lemma p-eq[rule-format]:

$$p \neq [] \longrightarrow \text{list2policy} (\text{map } C (\text{rev } p)) = C(\text{list2FWpolicy } p)$$

by (metis l2polR-eq list2FWpolicys-eq-sym rev.simps(1) rev-rev-ident)

lemma p-eq2[rule-format]:

$$\text{normalize } x \neq [] \longrightarrow C(\text{list2FWpolicy} (\text{normalize } x)) = C x \longrightarrow$$


$$\text{list2policy} (\text{map } C (\text{rev} (\text{normalize } x))) = C x$$

by (simp add: p-eq)

lemma p-eq2Q[rule-format]:

$$\text{normalizeQ } x \neq [] \longrightarrow C(\text{list2FWpolicy} (\text{normalizeQ } x)) = C x \longrightarrow$$


$$\text{list2policy} (\text{map } C (\text{rev} (\text{normalizeQ } x))) = C x$$

by (simp add: p-eq)

lemma list2listNMT[rule-format]:  $x \neq [] \longrightarrow \text{map sem } x \neq []$ 
by (case-tac x) simp-all

lemma Norm-Distr2:

$$r o-f ((P \otimes_2 (\text{list2policy } Q)) o d) = (\text{list2policy} ((P \otimes_L Q) (\otimes_2 r d)))$$


```

by (rule ext, rule Norm-Distr-2)

lemma NATDistr:

$$\begin{aligned} N \neq [] \implies F = C(\text{list2policyR } N) \implies \\ (\lambda(x, y). x) \circ_f (\text{NAT} \otimes_2 F \circ (\lambda x. (x, x))) = \\ \text{list2policy}((\text{NAT} \otimes_L \text{map } C N) (\otimes_2) (\lambda(x, y). x) (\lambda x. (x, x))) \\ \text{apply (simp add: l2polR-eq)} \\ \text{apply (rule ext)} \\ \text{apply (rule Norm-Distr-2)} \\ \text{done} \end{aligned}$$

lemma C-eq-normalize-manual:

$$\begin{aligned} \text{DenyAll} \in \text{set}(\text{policy2list } p) \implies \text{allNetsDistinct}(\text{policy2list } p) \implies \text{all-in-list}(\text{policy2list } p) l \implies \\ C(\text{list2FWpolicy}(\text{normalize-manual-order } p l)) = C p \\ \text{by (simp add: normalize-manual-order-def C-eq-compile)} \end{aligned}$$

lemma p-eq2-manualQ[rule-format]:

$$\begin{aligned} \text{normalize-manual-orderQ } x l \neq [] \longrightarrow C(\text{list2FWpolicy}(\text{normalize-manual-orderQ } x l)) = C x \longrightarrow \\ \text{list2policy}(\text{map } C(\text{rev}(\text{normalize-manual-orderQ } x l))) = C x \\ \text{by (simp add: p-eq)} \end{aligned}$$

lemma norm-notMT-manualQ: $\text{DenyAll} \in \text{set}(\text{policy2list } p) \implies \text{normalize-manual-orderQ } p l \neq []$

$$\text{by (simp add: DAiniD RS2-NMT RS3-NMT idNMT normalize-manual-orderQ-def rADnMT sepnMT sortnMTQ)}$$

lemma C-eq-normalize-manualQ:

$$\begin{aligned} \text{DenyAll} \in \text{set}(\text{policy2list } p) \implies \text{allNetsDistinct}(\text{policy2list } p) \implies \text{all-in-list}(\text{policy2list } p) l \implies \\ C(\text{list2FWpolicy}(\text{normalize-manual-orderQ } p l)) = C p \\ \text{by (simp add: normalize-manual-orderQ-def C-eq-compileQ)} \end{aligned}$$

lemma p-eq2-manual[rule-format]:

$$\begin{aligned} \text{normalize-manual-order } x l \neq [] \longrightarrow C(\text{list2FWpolicy}(\text{normalize-manual-order } x l)) \\ = C x \longrightarrow \\ \text{list2policy}(\text{map } C(\text{rev}(\text{normalize-manual-order } x l))) = C x \\ \text{by (simp add: p-eq)} \end{aligned}$$

lemma norm-notMT-manual: $\text{DenyAll} \in \text{set}(\text{policy2list } p) \implies \text{normalize-manual-order } p l \neq []$

$$\text{by (simp add: RS2-NMT idNMT normalize-manual-order-def rADnMT sepnMT sortnMT wp1ID)}$$

As an example, how this theorems can be used for a concrete normalisation instantiation.

lemma *normalizeNAT*:

$$\begin{aligned} \text{DenyAll} \in \text{set } (\text{policy2list } \text{Filter}) &\implies \text{allNetsDistinct } (\text{policy2list } \text{Filter}) \implies \\ \text{all-in-list } (\text{policy2list } \text{Filter}) \text{ (Nets-List Filter)} &\implies \\ (\lambda(x, y). x) \text{ of } (\text{NAT} \otimes_2 C \text{ Filter} \circ (\lambda x. (x, x))) = \\ \text{list2policy } ((\text{NAT} \otimes_L \text{map } C \text{ (rev } (\text{FWNormalisationCore.normalize Filter}))) \text{ (}\otimes_2\text{)}) \end{aligned}$$

$$(\lambda(x, y). x) \text{ (}\lambda x. (x, x)\text{))}$$

by (*simp add: C-eq-normalize NATDistr list2FWpolicys-eq-sym norm-notMT*)

lemma *domSimpl[simp]*: $\text{dom } (C (A \oplus \text{DenyAll})) = \text{dom } (C (\text{DenyAll}))$

by (*simp add: PLemmas*)

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*: $CRotate p = C (\text{rotatePolicy } p)$

by (*induct p rule: rotatePolicy.induct*) (*simp-all add: C.simps map-add-def*)

lemma *DAinRotate*:

$$\text{DenyAll} \in \text{set } (\text{policy2list } p) \implies \text{DenyAll} \in \text{set } (\text{policy2list } (\text{rotatePolicy } p))$$

apply (*induct p,simp-all*)

subgoal for *p1 p2*

apply (*case-tac DenyAll* ∈ *set (policy2list p1)*,*simp-all*)

done

done

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

by (*metis CRotate.simps(1) CRotate.simps(4) CRotate-eq-rotateC DAAux PLemmas(4) UNIV-eq-I domSubset3*)

lemma *p-eq2R[rule-format]*:

$$\begin{aligned} \text{normalize } (\text{rotatePolicy } x) \neq [] &\longrightarrow C(\text{list2FWpolicy}(\text{normalize } (\text{rotatePolicy } x))) = \\ CRotate x &\longrightarrow \end{aligned}$$

$$\text{list2policy } (\text{map } C \text{ (rev } (\text{normalize } (\text{rotatePolicy } x)))) = CRotate x$$

by (*simp add: p-eq*)

lemma *C-eq-normalizeRotate*:

$$\text{DenyAll} \in \text{set } (\text{policy2list } p) \implies \text{allNetsDistinct } (\text{policy2list } (\text{rotatePolicy } p)) \implies$$

$$\text{all-in-list } (\text{policy2list } (\text{rotatePolicy } p)) \text{ (Nets-List } (\text{rotatePolicy } p)) \implies$$

$$C \text{ (list2FWpolicy}$$

$$\text{ (removeAllDuplicates}$$

```

(insertDenies
  (separate
    (sort(removeShadowRules2(remdups(rm-MT-rules C
      (insertDeny(removeShadowRules1(policy2list(rotatePolicy p)))))))
      (Nets-List (rotatePolicy p))))))
= CRotate p
by (simp add: CRotate-eq-rotateC C-eq-compile DAinRotate)

lemma C-eq-normalizeRotate2:
  DenyAll ∈ set (policy2list p) ==>
  allNetsDistinct (policy2list (rotatePolicy p)) ==>
  all-in-list (policy2list (rotatePolicy p)) (Nets-List (rotatePolicy p)) ==>
  C (list2FWpolicy (FWNormalisationCore.normalize (rotatePolicy p))) = CRotate p
by (simp add: normalize-def, erule C-eq-normalizeRotate,simp-all)

end

```

2.3.4 Normalisation Proofs: Integer Protocol

```

theory
  NormalisationIPPProofs
imports
  NormalisationIntegerPortProof
begin

  Normalisation proofs which are specific to the IntegerProtocol address representation.

lemma ConcAssoc: Cp((A ⊕ B) ⊕ D) = Cp(A ⊕ (B ⊕ D))
by (simp add: Cp.simps)

lemma aux26[simp]:
  twoNetsDistinct a b c d ==> dom (Cp (AllowPortFromTo a b p)) ∩ dom (Cp
  (DenyAllFromTo c d)) = {}
by(auto simp:twoNetsDistinct-def netsDistinct-def PLemmas, auto)

lemma wp2-aux[rule-format]:
  wellformed-policy2Pr (xs @ [x]) —> wellformed-policy2Pr xs
  apply(induct xs, simp-all)
  subgoal for a as
    apply(case-tac a, simp-all)
    done
  done

lemma Cdom2: x ∈ dom(Cp b) ==> Cp (a ⊕ b) x = (Cp b) x
by (auto simp: Cp.simps)

```

```

lemma wp2Conc[rule-format]: wellformed-policy2Pr (x#xs)  $\implies$  wellformed-policy2Pr xs
by (case-tac x,simp-all)

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

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

lemma MRLList1[rule-format]: x  $\in$  dom (Cp a)  $\implies$  applied-rule-rev Cp x (b@[a]) = Some a
by (simp add: applied-rule-rev-def)

lemma MRLList2: x  $\in$  dom (Cp a)  $\implies$  applied-rule-rev Cp x (c@[b@[a]]) = Some a
by (simp add: applied-rule-rev-def)

lemma MRLList3:
x  $\notin$  dom(Cp xa)  $\implies$  applied-rule-rev Cp x (a@[b#xs@[xa]]) = applied-rule-rev Cp x (a @ b # xs)
by (simp add: applied-rule-rev-def)

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

lemma CConcStartaux: Cp a x = None  $\implies$  (Cp aa ++ Cp a) x = Cp aa x
by (simp add: PLemmas)

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

lemma mrNnt[simp]: applied-rule-rev Cp x p = Some a  $\implies$  p  $\neq$  []
by (simp add: applied-rule-rev-def)(auto)

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

```

```

apply (simp add: applied-rule-rev-def)
apply (rule rev-induct, simp-all, safe)
  apply (metis CConcEnd )
  apply (metis CConcEnd)
by (metis CConcStart applied-rule-rev-def mrNnt option.exhaust)

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

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

lemma CConcEnd2[rule-format]:
 $x \in \text{dom}(\text{Cp } a) \longrightarrow \text{Cp}(\text{list2FWpolicy}(xs @ [a])) x = \text{Cp } a x \text{ (is ?P xs)}$ 
by (rule-tac P = ?P in list2FWpolicy.induct) (auto simp:Cp.simps)

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

lemma CeqEnd[rule-format,simp]:
 $a \neq [] \longrightarrow x \in \text{dom}(\text{Cp}(\text{list2FWpolicy } a)) \longrightarrow \text{Cp}(\text{list2FWpolicy}(b@a)) x = (\text{Cp}(\text{list2FWpolicy } a)) x$ 
proof (induct rule: rev-induct)
  case Nil show ?case by simp
next
  case (snoc xa xs) show ?case
    apply (case-tac xs  $\neq []$ , simp-all)
    apply (case-tac x  $\in$  dom (Cp xa))
    apply (metis CConcEnd2 MRLList2 mr-is-C )
    apply (metis snoc.hyps CConcEnd1 CConcStart2 Nil-is-append-conv bar3 )
    by (metis MRLList2 eq-Nil-appendI mr-is-C )
qed

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

lemma domConc:
 $x \in \text{dom}(\text{Cp}(\text{list2FWpolicy } b)) \implies b \neq [] \implies x \in \text{dom}(\text{Cp}(\text{list2FWpolicy } (a@b)))$ 

```

```

by (auto simp: PLemmas)

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

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

lemma nMRtoNone[rule-format]:
 $p \neq [] \rightarrow \text{applied-rule-rev } \text{Cp } x p = \text{None} \rightarrow \text{Cp} (\text{list2FWpolicy } p) x = \text{None}$ 
proof (induct rule: rev-induct)
  case Nil show ?case by simp
next
  case (snoc xa xs) show ?case
    apply (case-tac xs = [], simp-all)
    by (simp-all add: snoc.hyps applied-rule-rev-def dom-def)
qed

lemma C-eq-if-mr-eq:
 $\text{applied-rule-rev } \text{Cp } x b = \text{applied-rule-rev } \text{Cp } x a \Rightarrow a \neq [] \Rightarrow b \neq [] \Rightarrow$ 
 $(\text{Cp} (\text{list2FWpolicy } a)) x = (\text{Cp} (\text{list2FWpolicy } b)) x$ 
apply (cases applied-rule-rev Cp x a = None, simp-all)
apply (subst nMRtoNone,simp-all)
apply (subst nMRtoNone,simp-all)
by (auto intro: C-eq-if-mr-eq2)

lemma notmatching-notdom:
 $\text{applied-rule-rev } \text{Cp } x (p@[a]) \neq \text{Some } a \Rightarrow x \notin \text{dom } (\text{Cp } a)$ 
by (simp add: applied-rule-rev-def split: if-splits)

lemma foo3a[rule-format]:
 $\text{applied-rule-rev } \text{Cp } x (a@[b]@c) = \text{Some } b \rightarrow r \in \text{set } c \rightarrow b \notin \text{set } c \rightarrow x \notin \text{dom } (\text{Cp } r)$ 
proof (induct rule: rev-induct)
  case Nil show ?case by simp
next
  case (snoc xa xs) show ?case
    apply simp-all
    apply (rule impI|rule conjI|simp)+
```

```

apply (rule-tac  $p = a @ b \# xs$  in notmatching-notdom,simp-all)
  by (metis Cons-eq-appendI NormalisationIPPProofs.MRList2 NormalisationIPPProofs.MRList3
append-Nil option.inject snoc.hyps)
qed

lemma foo3D:
  wellformed-policy1  $p \Rightarrow p = \text{DenyAll} \# ps \Rightarrow \text{applied-rule-rev } Cp x p = \text{Some DenyAll}$ 
 $\Rightarrow r \in \text{set } ps \Rightarrow$ 
 $x \notin \text{dom } (Cp r)$ 
  by (rule-tac  $a = []$  and  $b = \text{DenyAll}$  and  $c = ps$  in foo3a, simp-all)

lemma foo4[rule-format]:
  set  $p = \text{set } s \wedge (\forall r. r \in \text{set } p \rightarrow x \notin \text{dom } (Cp r)) \rightarrow (\forall r. r \in \text{set } s \rightarrow x \notin \text{dom } (Cp r))$ 
  by simp

lemma foo5b[rule-format]:
   $x \in \text{dom } (Cp b) \rightarrow (\forall r. r \in \text{set } c \rightarrow x \notin \text{dom } (Cp r)) \rightarrow \text{applied-rule-rev } Cp x (b \# c) = \text{Some } b$ 
  apply (simp add: applied-rule-rev-def)
  apply (rule-tac  $xs = c$  in rev-induct, simp-all)
  done

lemma mr-first:
   $x \in \text{dom } (Cp b) \Rightarrow (\forall r. r \in \text{set } c \rightarrow x \notin \text{dom } (Cp r)) \Rightarrow s = b \# c \Rightarrow$ 
  applied-rule-rev  $Cp x s = \text{Some } b$ 
  by (simp add: foo5b)

lemma mr-charn[rule-format]:
   $a \in \text{set } p \rightarrow (x \in \text{dom } (Cp a)) \rightarrow (\forall r. r \in \text{set } p \wedge x \in \text{dom } (Cp r) \rightarrow r = a)$ 
 $\rightarrow$ 
  applied-rule-rev  $Cp x p = \text{Some } a$ 
  apply(rule-tac  $xs = p$  in rev-induct)
  apply(simp-all only:applied-rule-rev-def)
  apply(simp,safe,simp-all)
  by(auto)

lemma foo8:
   $\forall r. r \in \text{set } p \wedge x \in \text{dom } (Cp r) \rightarrow r = a \Rightarrow \text{set } p = \text{set } s \Rightarrow$ 
   $\forall r. r \in \text{set } s \wedge x \in \text{dom } (Cp r) \rightarrow r = a$ 
  by auto

lemma mrConcEnd[rule-format]:

```

```

applied-rule-rev Cp x (b # p) = Some a → a ≠ b → applied-rule-rev Cp x p =
Some a
apply (simp add: applied-rule-rev-def)
apply (rule-tac xs = p in rev-induct,simp-all)
by auto

lemma wp3tl[rule-format]: wellformed-policy3Pr p → wellformed-policy3Pr (tl p)
apply (induct p, simp-all)
subgoal for a as
  apply(case-tac a, simp-all)
  done
done

lemma wp3Conc[rule-format]: wellformed-policy3Pr (a#p) → wellformed-policy3Pr p
by (induct p, simp-all, case-tac a, simp-all)

lemma foo98[rule-format]:
  applied-rule-rev Cp x (aa # p) = Some a → x ∈ dom (Cp r) → r ∈ set p → a ∈
set p
  unfolding applied-rule-rev-def
proof (induct rule: rev-induct)
  case Nil show ?case by simp
next
  case (snoc xa xs) then show ?case
    by simp-all (case-tac r = xa, simp-all)
qed

lemma mrMTNone[simp]: applied-rule-rev Cp x [] = None
by (simp add: applied-rule-rev-def)

lemma DAAux[simp]: x ∈ dom (Cp DenyAll)
by (simp add: dom-def PolicyCombinators.PolicyCombinators Cp.simps)

lemma mrSet[rule-format]: applied-rule-rev Cp x p = Some r → r ∈ set p
  unfolding applied-rule-rev-def
  by (rule-tac xs=p in rev-induct) simp-all

lemma mr-not-Conc: singleCombinators p ⇒ applied-rule-rev Cp x p ≠ Some (a⊕b)
by (auto simp: mrSet dest: mrSet elim: SCnotConc)

lemma foo25[rule-format]: wellformed-policy3Pr (p@[x]) → wellformed-policy3Pr p

```

```

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

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

lemma wp3EndMT[rule-format]:
  wellformed-policy3Pr (p@[xs])  $\longrightarrow$  AllowPortFromTo a b po ∈ set p  $\longrightarrow$ 
    dom (Cp (AllowPortFromTo a b po)) ∩ dom (Cp xs) = {}
  apply (induct p, simp-all)
  by (metis NormalisationIPPProofs.wp3Conc aux0-4 inf-commute list.set-intros(1)
    wellformed-policy3Pr.simps(2))

lemma foo29: dom (Cp a) ≠ {}  $\Longrightarrow$  dom (Cp a) ∩ dom (Cp b) = {}  $\Longrightarrow$  a ≠ b
  by auto

lemma foo28:
  AllowPortFromTo a b po ∈ set p  $\Longrightarrow$  dom(Cp(AllowPortFromTo a b po)) ≠ {}  $\Longrightarrow$ 
  (wellformed-policy3Pr(p@[x]))  $\Longrightarrow$ 
  x ≠ AllowPortFromTo a b po
  by (metis foo29 Cp.simps(3) wp3EndMT)

lemma foo28a[rule-format]: x ∈ dom (Cp a)  $\Longrightarrow$  dom (Cp a) ≠ {}
  by auto

lemma allow-deny-dom[simp]:
  dom (Cp (AllowPortFromTo a b po)) ⊆ dom (Cp (DenyAllFromTo a b))
  by (simp-all add: twoNetsDistinct-def netsDistinct-def PLemmas) auto

lemma DenyAllowDisj:
  dom (Cp (AllowPortFromTo a b p)) ≠ {}  $\Longrightarrow$ 
  dom (Cp (DenyAllFromTo a b)) ∩ dom (Cp (AllowPortFromTo a b p)) ≠ {}
  by (metis Int-absorb1 allow-deny-dom)

lemma foo31:
   $\forall r. r \in \text{set } p \wedge x \in \text{dom } (\text{Cp } r) \longrightarrow$ 
    (r = AllowPortFromTo a b po ∨ r = DenyAllFromTo a b ∨ r = DenyAll)  $\Longrightarrow$ 
    set p = set s  $\Longrightarrow$ 
    ( $\forall r. r \in \text{set } s \wedge x \in \text{dom } (\text{Cp } r) \longrightarrow r = \text{AllowPortFromTo a b po} \vee r = \text{DenyAllFromTo a b} \vee r = \text{DenyAll}$ )

```

```

by auto

lemma wp1-auxa: wellformed-policy1-strong  $\Rightarrow (\exists r. \text{applied-rule-rev } Cp x p = \text{Some } r)$ 
  apply (rule DAimpliesMR-E)
  by (erule wp1-aux1aa)

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

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

lemma DomInterAllowsMT:
  twoNetsDistinct a b c d  $\Rightarrow \text{dom } (Cp (\text{AllowPortFromTo } a b p)) \cap \text{dom } (Cp (\text{AllowPortFromTo } c d po)) = \{\}$ 
  apply (case-tac p = po, simp-all)
  apply (rule-tac b = Cp (DenyAllFromTo a b) in domTrans, simp-all)
  apply (metis domComm aux26 tNDComm)
  apply (simp add: twoNetsDistinct-def netsDistinct-def PLemmas)
  by (auto simp: prod-eqI)

lemma DomInterAllowsMT-Ports:
   $p \neq po \Rightarrow \text{dom } (Cp (\text{AllowPortFromTo } a b p)) \cap \text{dom } (Cp (\text{AllowPortFromTo } c d po)) = \{\}$ 
  apply (simp add: twoNetsDistinct-def netsDistinct-def PLemmas)
  by (auto simp: prod-eqI)

lemma wellformed-policy3-charn[rule-format]:
  singleCombinators p  $\rightarrow$  distinct p  $\rightarrow$  allNetsDistinct p  $\rightarrow$ 
  wellformed-policy1 p  $\rightarrow$  wellformed-policy2Pr p  $\rightarrow$  wellformed-policy3Pr p
proof (induct p)
  case Nil show ?case by simp
next
  case (Cons a p) then show ?case
    apply (auto intro: singleCombinatorsConc ANDConc waux2 wp2Conc)
    apply (case-tac a,simp-all, clarify)
    subgoal for a b c d r
      apply (case-tac r,simp-all)
      apply (metis Int-commute)
      apply (metis DomInterAllowsMT aux7aa DomInterAllowsMT-Ports)

```

```

apply (metis aux0-0 )
done
done
qed

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

lemma DistinctNetsAllowAllow:
 $\text{AllowPortFromTo } b \in \text{set } p \implies \text{AllowPortFromTo } a \in \text{set } p \implies$ 
 $\text{allNetsDistinct } p \implies \text{dom}(\text{Cp}(\text{AllowPortFromTo } b)) \cap \text{dom}(\text{Cp}(\text{AllowPortFromTo } a)) \neq \{\} \implies$ 
 $b = a \wedge c = d \wedge \text{poo} = \text{po}$ 
apply (simp add: allNetsDistinct-def)
apply (frule-tac x = b in spec)
apply (drule-tac x = d in spec)
apply (drule-tac x = a in spec)
apply (drule-tac x = c in spec)
apply (metis DomInterAllowsMT DomInterAllowsMT-Ports ND0aux3 ND0aux4 ND-
Comm twoNetsDistinct-def)
done

lemma WP2RS2[simp]:
 $\text{singleCombinators } p \implies \text{distinct } p \implies \text{allNetsDistinct } p \implies$ 
 $\text{wellformed-policy2Pr } (\text{removeShadowRules2 } p)$ 
proof (induct p)
case Nil
then show ?case by simp
next
case (Cons x xs)
have wp-xs: wellformed-policy2Pr (removeShadowRules2 xs)
by (metis Cons ANDConc distinct.simps(2) singleCombinatorsConc)
show ?case

```

```

proof (cases x)
  case DenyAll thus ?thesis using wp-xs by simp
next
  case (DenyAllFromTo a b) thus ?thesis
    using wp-xs Cons
    by (simp,metis DenyAllFromTo aux aux7 tNDComm deny-dom)
next
  case (AllowPortFromTo a b p) thus ?thesis
    using wp-xs
    by (simp, metis aux26 AllowPortFromTo Cons(4) aux aux7a tNDComm)
next
  case (Conc a b) thus ?thesis
    by (metis Conc Cons(2) singleCombinators.simps(2))
qed
qed

```

lemma *AD-aux*:

```

AllowPortFromTo a b po  $\in$  set p  $\implies$  DenyAllFromTo c d  $\in$  set p  $\implies$ 
allNetsDistinct p  $\implies$  singleCombinators p  $\implies$  a  $\neq$  c  $\vee$  b  $\neq$  d  $\implies$ 
dom (Cp (AllowPortFromTo a b po))  $\cap$  dom (Cp (DenyAllFromTo c d)) = {}
by (rule aux26,rule-tac x =AllowPortFromTo a b po and y = DenyAllFromTo c d in tND) auto

```

lemma *sorted-WP2[rule-format]*:

```

sorted p l  $\longrightarrow$  all-in-list p l  $\longrightarrow$  distinct p  $\longrightarrow$  allNetsDistinct p  $\longrightarrow$  singleCombinators p  $\longrightarrow$ 
wellformed-policy2Pr p
proof (induct p)
  case Nil thus ?case by simp
next
  case (Cons a p) thus ?case
proof (cases a)
  case DenyAll thus ?thesis
    using Cons by (auto intro: ANDConc singleCombinatorsConc sortedConcEnd)
next
  case (DenyAllFromTo c d) thus ?thesis
    using Cons apply (simp, intro impI conjI allI impI deny-dom)
    by (auto intro: aux7 tNDComm ANDConc singleCombinatorsConc sortedConcEnd)
next
  case (AllowPortFromTo c d e) thus ?thesis
    using Cons apply simp
    apply (intro impI conjI allI, rename-tac aa b)
    apply (rule aux26)

```

```

subgoal for aa b
  apply (rule-tac  $x = \text{AllowPortFromTo } c d e$  and  $y = \text{DenyAllFromTo } aa b$  in
 $tND,$ 
  assumption,simp-all)
  apply (subgoal-tac smaller (AllowPortFromTo c d e) (DenyAllFromTo aa b) l)
  apply (simp split: if-splits)
  apply metis
  apply (erule sorted-is-smaller, simp-all)
  apply (metis bothNet.simps(2) in-list.simps(2) in-set-in-list)
  done
  by (auto intro: aux7 tNDComm ANDConc singleCombinatorsConc sortedConcEnd)
next
  case (Conc a b) thus ?thesis using Cons by simp
qed
qed

lemma wellformed2-sorted[simp]:
  all-in-list p l  $\implies$  distinct p  $\implies$  allNetsDistinct p  $\implies$  singleCombinators p  $\implies$ 
  wellformed-policy2Pr (sort p l)
  by (metis distinct-sort set-sort sorted-WP2 SC3 aND-sort all-in-listSubset order-refl
  sort-is-sorted)

lemma wellformed2-sortedQ[simp]:
  all-in-list p l  $\implies$  distinct p  $\implies$  allNetsDistinct p  $\implies$  singleCombinators p  $\implies$ 
  wellformed-policy2Pr (qsort p l)
  by (metis sorted-WP2 SC3Q aND-sortQ all-in-listSubset distinct-sortQ set-sortQ
  sort-is-sortedQ subsetI)

lemma C-DenyAll[simp]: Cp (list2FWpolicy (xs @ [DenyAll])) x = Some (deny ())
  by (auto simp: PLemmas)

lemma C-eq-RS1n:
  Cp(list2FWpolicy (removeShadowRules1-alternative p)) = Cp(list2FWpolicy p)
proof (cases p)
  case Nil then show ?thesis
    by (simp, metis list2FWpolicy.simps(1) rSR1-eq removeShadowRules1.simps(2))
next
  case (Cons a list) then show ?thesis
    apply (hypsubst, simp)
    apply (thin-tac  $p = a \# list$ )
proof (induct rule: rev-induct)
  case Nil show ?case by (metis rSR1-eq removeShadowRules1.simps(2))
next
  case (snoc x xs) show ?case

```

```

apply (case-tac xs = [], simp-all)
apply (simp add: removeShadowRules1-alternative-def)
apply (insert snoc.hyps, case-tac x, simp-all)
apply (rule ext, rename-tac xa)
apply (case-tac x = DenyAll, simp-all add: PLemmas)
apply (rule-tac t = removeShadowRules1-alternative (xs @ [x]) and
      s = (removeShadowRules1-alternative xs)@[x] in subst)
apply (erule RS1n-assoc)
subgoal for a
  apply (case-tac a ∈ dom (Cp x), simp-all)
  done
done
qed
qed

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

lemma EX-MR-aux[rule-format]:
  applied-rule-rev Cp x (DenyAll # p) ≠ Some DenyAll → (exists y. applied-rule-rev Cp x p = Some y)
  by (simp add: applied-rule-rev-def) (rule-tac xs = p in rev-induct, simp-all)

lemma EX-MR :
  applied-rule-rev Cp x p ≠ (Some DenyAll) ⇒ p = DenyAll#ps ⇒
  (applied-rule-rev Cp x p = applied-rule-rev Cp x ps)
  apply (auto, subgoal-tac applied-rule-rev Cp x (DenyAll#ps) ≠ None, auto)
  apply (metis mrConcEnd)
  by (metis DAImpliesMR-E list.sel(1) hd-in-set list.simps(3) not-Some-eq)

lemma mr-not-DA:
  wellformed-policy1-strong s ⇒ applied-rule-rev Cp x p = Some (DenyAllFromTo a ab) ⇒
  set p = set s ⇒ applied-rule-rev Cp x s ≠ Some DenyAll
  apply (subst wp1n-tl, simp-all)
  by (metis (mono-tags, lifting) Combinators.distinct(1) foo98
       mrSet mr-in-dom WP1n-DA-notinSet set-ConsD wp1n-tl)

lemma domsMT-notND-DD:
  dom (Cp (DenyAllFromTo a b)) ∩ dom (Cp (DenyAllFromTo c d)) ≠ {} ⇒ ¬
  netsDistinct a c
  by (erule contrapos-nn) (simp add: Cp.simps aux6 twoNetsDistinct-def)

```

```

lemma domsMT-notND-DD2:
   $\text{dom}(\text{Cp}(\text{DenyAllFromTo } a \ b)) \cap \text{dom}(\text{Cp}(\text{DenyAllFromTo } c \ d)) \neq \{\} \implies \neg \text{netsDistinct } b \ d$ 
  by (erule contrapos-nn) (simp add: Cp.simps aux6 twoNetsDistinct-def)

lemma domsMT-notND-DD3:
   $x \in \text{dom}(\text{Cp}(\text{DenyAllFromTo } a \ b)) \implies x \in \text{dom}(\text{Cp}(\text{DenyAllFromTo } c \ d)) \implies \neg \text{netsDistinct } a \ c$ 
  by (auto intro!: domsMT-notND-DD)

lemma domsMT-notND-DD4:
   $x \in \text{dom}(\text{Cp}(\text{DenyAllFromTo } a \ b)) \implies x \in \text{dom}(\text{Cp}(\text{DenyAllFromTo } c \ d)) \implies \neg \text{netsDistinct } b \ d$ 
  by (auto intro!: domsMT-notND-DD2)

lemma NetsEq-if-sameP-DD:
   $\text{allNetsDistinct } p \implies u \in \text{set } p \implies v \in \text{set } p \implies u = (\text{DenyAllFromTo } a \ b) \implies$ 
   $v = (\text{DenyAllFromTo } c \ d) \implies x \in \text{dom}(\text{Cp}(u)) \implies x \in \text{dom}(\text{Cp}(v)) \implies$ 
   $a = c \wedge b = d$ 
  unfolding allNetsDistinct-def
  by (simp)(metis allNetsDistinct-def ND0aux1 ND0aux2 domsMT-notND-DD3
  domsMT-notND-DD4)

lemma rule-charn1:
  assumes aND : allNetsDistinct p
  and mr-is-allow : applied-rule-rev Cp x p = Some (AllowPortFromTo a b po)
  and SC : singleCombinators p
  and inp : r ∈ set p
  and inDom : x ∈ dom(Cp r)
  shows (r = AllowPortFromTo a b po ∨ r = DenyAllFromTo a b ∨ r = DenyAll)
  proof (cases r)
    case DenyAll show ?thesis by (metis DenyAll)
  next
    case (DenyAllFromTo x y) show ?thesis
      by (metis DenyAllFromTo NormalisationIPPProofs.AD-aux NormalisationIPPProofs.mrSet
      NormalisationIPPProofs.mr-in-dom SC aND domInterMT inDom inp mr-is-allow)
  next
    case (AllowPortFromTo x y b) show ?thesis
      by (metis (mono-tags, lifting) AllowPortFromTo NormalisationIPPProofs.DistinctNetsAllowAllow
      NormalisationIPPProofs.mrSet NormalisationIPPProofs.mr-in-dom aND dom-
      InterMT inDom
      inp mr-is-allow)

```

```

next
  case (Conc x y) thus ?thesis using assms by (metis aux0-0)
qed

lemma none-MT-rulessubset[rule-format]:
  none-MT-rules Cp a  $\longrightarrow$  set b  $\subseteq$  set a  $\longrightarrow$  none-MT-rules Cp b
  by (induct b,simp-all) (metis notMTnMT)

lemma nMTSort: none-MT-rules Cp p  $\implies$  none-MT-rules Cp (sort p l)
  by (metis set-sort nMTeqSet)

lemma nMTSortQ: none-MT-rules Cp p  $\implies$  none-MT-rules Cp (qsort p l)
  by (metis set-sortQ nMTeqSet)

lemma wp3char[rule-format]: none-MT-rules Cp xs  $\wedge$  Cp (AllowPortFromTo a b po)
= Map.empty  $\wedge$ 
  wellformed-policy3Pr (xs @ [DenyAllFromTo a b])  $\longrightarrow$ 
    AllowPortFromTo a b po  $\notin$  set xs
  by (induct xs, simp-all) (metis domNMT wp3Conc)

lemma wp3charrn[rule-format]:
  assumes domAllow: dom (Cp (AllowPortFromTo a b po))  $\neq \{\}$ 
  and wp3: wellformed-policy3Pr (xs @ [DenyAllFromTo a b])
  shows allowNotInList: AllowPortFromTo a b po  $\notin$  set xs
  apply (insert assms)
  proof (induct xs)
    case Nil show ?case by simp
  next
    case (Cons x xs) show ?case using Cons
      by (simp,auto intro: wp3Conc) (auto simp: DenyAllowDisj domAllow)
  qed

lemma rule-charn2:
  assumes aND: allNetsDistinct p
  and wp1: wellformed-policy1 p
  and SC: singleCombinators p
  and wp3: wellformed-policy3Pr p
  and allow-in-list: AllowPortFromTo c d po  $\in$  set p
  and x-in-dom-allow: x  $\in$  dom (Cp (AllowPortFromTo c d po))
  shows applied-rule-rev Cp x p = Some (AllowPortFromTo c d po)
  proof (insert assms, induct p rule: rev-induct)
    case Nil show ?case using Nil by simp
  next
    case (snoc y ys) show ?case using snoc

```

```

apply simp
apply (case-tac y = (AllowPortFromTo c d po))
  apply (simp add: applied-rule-rev-def)
apply simp-all
apply (subgoal-tac ys ≠ [])
  apply (subgoal-tac applied-rule-rev Cp x ys = Some (AllowPortFromTo c d po))
    defer 1
    apply (metis ANDConcEnd SCCConcEnd WP1ConcEnd foo25)
    apply (metis inSet-not-MT)
proof (cases y)
  case DenyAll thus ?thesis using DenyAll snoc
    apply simp
    by (metis DAnotTL DenyAll inSet-not-MT policy2list.simps(2))
next
  case (DenyAllFromTo a b) thus ?thesis using snoc apply simp
    apply (simp-all add: applied-rule-rev-def)
    apply (rule conjI)
    apply (metis domInterMT wp3EndMT)
    apply (rule impI)
    by (metis ANDConcEnd DenyAllFromTo SCCConcEnd WP1ConcEnd foo25)
next
  case (AllowPortFromTo a1 a2 b) thus ?thesis using AllowPortFromTo snoc apply simp
    apply (simp-all add: applied-rule-rev-def)
    apply (rule conjI)
    apply (metis domInterMT wp3EndMT)
    by (metis ANDConcEnd AllowPortFromTo SCCConcEnd WP1ConcEnd foo25
x-in-dom-allow)
next
  case (Conc a b) thus ?thesis
    using Conc snoc apply simp
    by (metis Conc aux0-0 in-set-conv-decomp)
qed
qed

```

lemma rule-charn3:

$\text{wellformed-policy1 } p \implies \text{allNetsDistinct } p \implies \text{singleCombinators } p \implies$
 $\text{wellformed-policy3Pr } p \implies \text{applied-rule-rev } Cp x p = \text{Some } (\text{DenyAllFromTo } c d) \implies$
 $\text{AllowPortFromTo } a b po \in \text{set } p \implies x \notin \text{dom } (Cp (\text{AllowPortFromTo } a b po))$
by (clarify) (simp add: NormalisationIPPProofs.rule-charn2 domI)

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 (cases r)
  case DenyAll thus ?thesis using DenyAll assms by simp
next
  case (DenyAllFromTo c d) thus ?thesis
    using assms apply simp
    apply (erule-tac x = x and p = p and v = (DenyAllFromTo a b) and
           u = (DenyAllFromTo c d) in NetsEq-if-sameP-DD, simp-all)
    apply (erule mrSet)
    by (erule mr-in-dom)
next
  case (AllowPortFromTo c d e) thus ?thesis
    using assms apply simp
    apply (subgoal-tac x ∉ dom (Cp (AllowPortFromTo c d e)), simp)
    by (rule-tac p = p in rule-charn3, auto intro: SCnotConc)
next
  case (Conc a b) thus ?thesis
    using assms apply simp
    by (metis Conc aux0-0)
qed

```

```

lemma foo31a:
  ( ∀ r. r ∈ set p ∧ x ∈ dom (Cp r) →
    (r = AllowPortFromTo a b po ∨ r = DenyAllFromTo a b ∨ r = DenyAll)) ⇒
    set p = set s ⇒ r ∈ set s ⇒ x ∈ dom (Cp r) ⇒
    (r = AllowPortFromTo a b po ∨ r = DenyAllFromTo a b ∨ r = DenyAll)
  by auto

```

```

lemma aux4[rule-format]:
  applied-rule-rev Cp x (a#p) = Some a → a ∉ set (p) → applied-rule-rev Cp x p =
  None
  by (rule rev-induct, simp-all) (intro impI, simp add: applied-rule-rev-def split: if-splits)

```

```

lemma mrDA-tl:
  assumes mr-DA: applied-rule-rev Cp x p = Some DenyAll
  and      wp1n:  wellformed-policy1-strong p
  shows     applied-rule-rev Cp x (tl p) = None

```

```

apply (rule aux4 [where a = DenyAll])
  apply (metis wp1n-tl mr-DA wp1n)
  by (metis WP1n-DA-notinSet wp1n)

lemma rule-charnDAFT:
  wellformed-policy1-strong p  $\Rightarrow$  allNetsDistinct p  $\Rightarrow$  singleCombinators p  $\Rightarrow$ 
    wellformed-policy3Pr p  $\Rightarrow$  applied-rule-rev Cp x p = Some (DenyAllFromTo a b)
   $\Rightarrow$ 
    r  $\in$  set (tl p)  $\Rightarrow$  x  $\in$  dom (Cp r)  $\Rightarrow$ 
    r = DenyAllFromTo a b
  apply (subgoal-tac p = DenyAll#(tl p))
  apply (metis (no-types, lifting) ANDConc Combinators.distinct(1) Normalisation-IPPProofs.mrConcEnd
    NormalisationIPPProofs.rule-charn4 NormalisationIPPProofs.wp3Conc
    WP1n-DA-notinSet
    singleCombinatorsConc waux2)
  using wp1n-tl by auto

```

```

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

```

```

theorem C-eq-Sets-mr:
  assumes sets-eq: set p = set s
  and SC: singleCombinators p
  and wp1-p: wellformed-policy1-strong p
  and wp1-s: wellformed-policy1-strong s
  and wp3-p: wellformed-policy3Pr p
  and wp3-s: wellformed-policy3Pr s
  and aND: allNetsDistinct p
  shows applied-rule-rev Cp x p = applied-rule-rev Cp x s
proof (cases applied-rule-rev Cp x p)
  case None
  have DA: DenyAll  $\in$  set p using wp1-p by (auto simp: wp1-aux1aa)
  have notDA: DenyAll  $\notin$  set p using None by (auto simp: DAimplieMR)
  thus ?thesis using DA by (contradiction)
next
  case (Some y) thus ?thesis
  proof (cases y)
    have tl-p: p = DenyAll#(tl p) by (metis wp1-p wp1n-tl)

```

```

have tl-s:  $s = \text{DenyAll\#}(tl\ s)$  by (metis wp1-s wp1n-tl)
have tl-eq:  $\text{set}\ (tl\ p) = \text{set}\ (tl\ s)$ 
  by (metis list.sel(3) WP1n-DA-notinSet sets-eq foo2
       wellformed-policy1-charn wp1-aux1aa wp1-eq wp1-p wp1-s)
{
  case DenyAll
  have mr-p-is-DenyAll: applied-rule-rev  $Cp\ x\ p = \text{Some}\ \text{DenyAll}$ 
    by (simp add: DenyAll Some)
  hence x-notin-tl-p:  $\forall r. r \in \text{set}\ (tl\ p) \longrightarrow x \notin \text{dom}\ (Cp\ r)$  using wp1-p
    by (auto simp: mrDenyAll-is-unique)
  hence x-notin-tl-s:  $\forall r. r \in \text{set}\ (tl\ s) \longrightarrow x \notin \text{dom}\ (Cp\ r)$  using tl-eq
    by auto
  hence mr-s-is-DenyAll: applied-rule-rev  $Cp\ x\ s = \text{Some}\ \text{DenyAll}$  using tl-s
    by (auto simp: mr-first)
  thus ?thesis using mr-p-is-DenyAll by simp
next
  case (DenyAllFromTo a b)
  have mr-p-is-DAFT: applied-rule-rev  $Cp\ x\ p = \text{Some}\ (\text{DenyAllFromTo}\ a\ b)$ 
    by (simp add: DenyAllFromTo Some)
  have DA-notin-tl:  $\text{DenyAll} \notin \text{set}\ (tl\ p)$ 
    by (metis WP1n-DA-notinSet wp1-p)
  have mr-tl-p: applied-rule-rev  $Cp\ x\ p = \text{applied-rule-rev}\ Cp\ x\ (tl\ p)$ 
    by (metis Combinators.simps(4) DenyAllFromTo Some mrConcEnd tl-p)
  have dom-tl-p:  $\bigwedge r. r \in \text{set}\ (tl\ p) \wedge x \in \text{dom}\ (Cp\ r) \implies$ 
     $r = (\text{DenyAllFromTo}\ a\ b)$ 
    using wp1-p AND SC wp3-p mr-p-is-DAFT
    by (auto simp: rule-charnDAFT)
  hence dom-tl-s:  $\bigwedge r. r \in \text{set}\ (tl\ s) \wedge x \in \text{dom}\ (Cp\ r) \implies$ 
     $r = (\text{DenyAllFromTo}\ a\ b)$ 
    using tl-eq by auto
  have DAFT-in-tl-s:  $\text{DenyAllFromTo}\ a\ b \in \text{set}\ (tl\ s)$  using mr-tl-p
    by (metis DenyAllFromTo mrSet mr-p-is-DAFT tl-eq)
  have x-in-dom-DAFT:  $x \in \text{dom}\ (Cp\ (\text{DenyAllFromTo}\ a\ b))$ 
    by (metis mr-p-is-DAFT DenyAllFromTo mr-in-dom)
  hence mr-tl-s-is-DAFT: applied-rule-rev  $Cp\ x\ (tl\ s) = \text{Some}\ (\text{DenyAllFromTo}\ a\ b)$ 
    using DAFT-in-tl-s dom-tl-s by (metis mr-charn)
  hence mr-s-is-DAFT: applied-rule-rev  $Cp\ x\ s = \text{Some}\ (\text{DenyAllFromTo}\ a\ b)$ 
    using tl-s
    by (metis DA-notin-tl DenyAllFromTo EX-MR mrDA-tl
          not-Some-eq tl-eq wellformed-policy1-strong.simps(2))
  thus ?thesis using mr-p-is-DAFT by simp
next
  case (AllowPortFromTo a b c)
  have wp1s: wellformed-policy1 s by (metis wp1-eq wp1-s)

```

```

have mr-p-is-A: applied-rule-rev Cp x p = Some (AllowPortFromTo a b c)
  by (simp add: AllowPortFromTo Some)
hence A-in-s: AllowPortFromTo a b c ∈ set s using sets-eq
  by (auto intro: mrSet)
have x-in-dom-A: x ∈ dom (Cp (AllowPortFromTo a b c))
  by (metis mr-p-is-A AllowPortFromTo mr-in-dom)
have SCs: singleCombinators s using SC sets-eq
  by (auto intro: SCSubset)
hence ANDs: allNetsDistinct s using aND sets-eq SC
  by (auto intro: aNDSetsEq)
hence mr-s-is-A: applied-rule-rev Cp x s = Some (AllowPortFromTo a b c)
  using A-in-s wp1s mr-p-is-A aND SCs wp3-s x-in-dom-A
  by (simp add: rule-charn2)
thus ?thesis using mr-p-is-A by simp
}
next
  case (Conc a b) thus ?thesis by (metis Some mr-not-Conc SC)
qed
qed

```

lemma C-eq-Sets:

$$\begin{aligned} \text{singleCombinators } p &\implies \text{wellformed-policy1-strong } p \implies \text{wellformed-policy1-strong } s \\ &\implies \text{wellformed-policy3Pr } p \implies \text{wellformed-policy3Pr } s \implies \text{allNetsDistinct } p \implies \text{set } p = \\ &\quad \text{set } s \implies \\ &Cp (\text{list2FWpolicy } p) x = Cp (\text{list2FWpolicy } s) x \\ &\text{by (metis C-eq-Sets-mr C-eq-if-mr-eq wellformed-policy1-strong.simps(1))} \end{aligned}$$

lemma C-eq-sorted:

$$\begin{aligned} \text{distinct } p &\implies \text{all-in-list } p l \implies \text{singleCombinators } p \implies \\ &\text{wellformed-policy1-strong } p \implies \text{wellformed-policy3Pr } p \implies \text{allNetsDistinct } p \implies \\ &Cp (\text{list2FWpolicy } (\text{sort } p l)) = Cp (\text{list2FWpolicy } p) \\ &\text{by (rule ext)} \\ &\quad (\text{meson distinct-sort set-sort C-eq-Sets wellformed2-sorted wellformed-policy3-charn} \\ &\quad \text{SC3 aND-sort} \\ &\quad \text{wellformed1-alternative-sorted wp1-eq}) \end{aligned}$$

lemma C-eq-sortedQ:

$$\begin{aligned} \text{distinct } p &\implies \text{all-in-list } p l \implies \text{singleCombinators } p \implies \\ &\text{wellformed-policy1-strong } p \implies \text{wellformed-policy3Pr } p \implies \text{allNetsDistinct } p \implies \\ &Cp (\text{list2FWpolicy } (\text{qsort } p l)) = Cp (\text{list2FWpolicy } p) \\ &\text{by (rule ext)} \\ &\quad (\text{metis C-eq-Sets wellformed2-sortedQ wellformed-policy3-charn SC3Q aND-sortQ} \\ &\quad \text{distinct-sortQ}) \end{aligned}$$

```

set-sortQ wellformed1-sorted-auxQ wellformed-eq wp1-aux1aa)

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

lemma C-eq-None[rule-format]:
  p ≠ [] → applied-rule-rev Cp x p = None → Cp (list2FWpolicy p) x = None
  unfolding applied-rule-rev-def
proof(induct rule: rev-induct)
  case Nil show ?case by simp

```

```

next
case (snoc xa xs) show ?case
  apply (insert snoc.hyps, intro impI, simp)
  apply (case-tac xs ≠ [])
  apply (metis CConcStart2 option.simps(3))
  by (metis append-Nil domIff l2p-aux2 option.distinct(1))
qed

lemma C-eq-None2:
  a ≠ []  $\implies$  b ≠ []  $\implies$  applied-rule-rev Cp x a = None  $\implies$  applied-rule-rev Cp x b = None  $\implies$ 
  (Cp (list2FWpolicy a)) x = (Cp (list2FWpolicy b)) x
  by (auto simp: C-eq-None)

lemma C-eq-RS2:
  wellformed-policy1-strong p  $\implies$ 
  Cp (list2FWpolicy (removeShadowRules2 p)) = Cp (list2FWpolicy p)
  apply (rule ext)
  by (metis C-eq-RS2-mr C-eq-if-mr-eq RS2-NMT wp1-alternative-not-mt)

lemma none-MT-rulesRS2: none-MT-rules Cp p  $\implies$  none-MT-rules Cp (removeShadowRules2 p)
  by (auto simp: RS2Set none-MT-rulessubset)

lemma CconcNone:
  dom (Cp a) = {}  $\implies$  p ≠ []  $\implies$  Cp (list2FWpolicy (a # p)) x = Cp (list2FWpolicy p) x
  apply (case-tac p = [], simp-all)
  apply (case-tac x ∈ dom (Cp (list2FWpolicy(p))))
  apply (metis Cdom2 list2FWpolicyconc)
  apply (metis Cp.simps(4) map-add-dom-app-simps(2) inSet-not-MT list2FWpolicy-conc set-empty2)
  done

lemma none-MT-rulesrd[rule-format]: none-MT-rules Cp p  $\longrightarrow$  none-MT-rules Cp (remdups p)
  by (induct p, simp-all)

lemma DARS3[rule-format]: DenyAll ∉ set p  $\longrightarrow$  DenyAll ∉ set (rm-MT-rules Cp p)
  by (induct p, simp-all)

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

```

```

lemma DAnMT2:  $Cp \text{ DenyAll} \neq \text{Map.empty}$ 
  by (metis DAAux dom-eq-empty-conv empty-iff)

lemma wp1n-RS3[rule-format,simp]:
  wellformed-policy1-strong p  $\longrightarrow$  wellformed-policy1-strong (rm-MT-rules Cp p)
  apply (induct p, simp-all)
  apply (rule conjI | rule impI | simp)+
  apply (metis DAnMT)
  apply (metis DARS3)
  done

lemma AILRS3[rule-format,simp]:
  all-in-list p l  $\longrightarrow$  all-in-list (rm-MT-rules Cp p) l
  by (induct p, simp-all)

lemma SCRS3[rule-format,simp]:
  singleCombinators p  $\longrightarrow$  singleCombinators(rm-MT-rules Cp p)
  apply (induct p, simp-all)
  subgoal for a p
    apply (case-tac a, simp-all)
    done
  done

lemma RS3subset: set (rm-MT-rules Cp p)  $\subseteq$  set p
  by (induct p, auto)

lemma ANDRS3[simp]:
  singleCombinators p  $\implies$  allNetsDistinct p  $\implies$  allNetsDistinct (rm-MT-rules Cp p)
  by (rule-tac b = p in aNDSubset, simp-all add:RS3subset)

lemma nlpaux:  $x \notin \text{dom } (Cp b) \implies Cp (a \oplus b) x = Cp a x$ 
  by (metis Cp.simps(4) map-add-dom-app-simps(3))

lemma notindom[rule-format]:
   $a \in \text{set p} \longrightarrow x \notin \text{dom } (Cp (\text{list2FWpolicy p})) \longrightarrow x \notin \text{dom } (Cp a)$ 
  proof (induct p)
    case Nil show ?case by simp
  next
    case (Cons a p) then show ?case
      apply (simp-all,intro conjI impI)
      apply (metis CConcStartA)
      apply simp
      apply (metis Cdom2 List.set-simps(2) domIff insert-absorb list.simps(2) list2FW-policyconc set-empty)

```

```

done
qed

lemma C-eq-rd[rule-format]:
   $p \neq [] \implies Cp(\text{list2FWpolicy}(\text{remdups } p)) = Cp(\text{list2FWpolicy } p)$ 
proof (rule ext, induct p)
  case Nil thus ?case by simp
next
  case (Cons y ys) thus ?case
  proof (cases ys = [])
    case True thus ?thesis by simp
next
  case False thus ?thesis
    using Cons apply simp
    apply (intro conjI impI)
    apply (metis Cdom2 nlpaux notindom domIff l2p-aux)
    by (metis (no-types, lifting) Cdom2 nlpaux domIff l2p-aux remDupsNMT)
qed
qed

lemma nMT-domMT:
   $\neg \text{not-MT } Cp \ p \implies p \neq [] \implies r \notin \text{dom}(Cp(\text{list2FWpolicy } p))$ 
proof (induct p)
  case Nil thus ?case by simp
next
  case (Cons x xs) thus ?case
    apply (simp split: if-splits)
    apply (cases xs = [],simp-all )
    by (metis CconcNone domIff)
qed

lemma C-eq-RS3-aux[rule-format]:
   $\text{not-MT } Cp \ p \implies Cp(\text{list2FWpolicy } p) \ x = Cp(\text{list2FWpolicy}(\text{rm-MT-rules } Cp \ p))$ 
  x
proof (induct p)
  case Nil thus ?case by simp
next
  case (Cons y ys) thus ?case
  proof (cases not-MT Cp ys)
    case True thus ?thesis
      using Cons apply simp
      apply (intro conjI impI, simp)
      apply (metis CconcNone True not-MTimpnotMT)
      apply (cases x ∈ dom (Cp (list2FWpolicy ys)))

```

```

apply (subgoal-tac  $x \in \text{dom} (\text{Cp} (\text{list2FWpolicy} (\text{rm-MT-rules} \text{ Cp} ys))))$ 
  apply (metis (mono-tags) Cons-eq-appendI NMPrm CeqEnd append-Nil
not-MTimpnotMT)
  apply (simp add: domIff)
  apply (subgoal-tac  $x \notin \text{dom} (\text{Cp} (\text{list2FWpolicy} (\text{rm-MT-rules} \text{ Cp} ys))))$ 
  apply (metis l2p-aux l2p-aux2 nlpaux)
  by (metis domIff)
next
  case False thus ?thesis
    using Cons False
  proof (cases ys = [])
    case True thus ?thesis using Cons by (simp) (rule impI, simp)
next
  case False thus ?thesis
    using Cons False  $\leftarrow \neg \text{not-MT} \text{ Cp} ys$  apply (simp)
    apply (intro conjI impI| simp)+
    apply (subgoal-tac rm-MT-rules Cp ys = [])
    apply (subgoal-tac  $x \notin \text{dom} (\text{Cp} (\text{list2FWpolicy} ys))$ )
    apply simp-all
    apply (metis l2p-aux nlpaux)
    apply (erule nMT-domMT, simp-all)
    by (metis SR3nMT)
  qed
  qed
qed

```

lemma C-eq-id:
 $\text{wellformed-policy1-strong } p \implies \text{Cp}(\text{list2FWpolicy} (\text{insertDeny } p)) = \text{Cp}(\text{list2FWpolicy } p)$
by (rule ext) (metis insertDeny.simps(1) wp1n-tl)

lemma C-eq-RS3:
 $\text{not-MT } \text{Cp } p \implies \text{Cp}(\text{list2FWpolicy} (\text{rm-MT-rules} \text{ Cp } p)) = \text{Cp} (\text{list2FWpolicy } p)$
by (rule ext) (erule C-eq-RS3-aux[symmetric])

lemma NMPrd[rule-format]: $\text{not-MT } \text{Cp } p \longrightarrow \text{not-MT } \text{Cp } (\text{remdups } p)$
by (induct p, simp-all) (auto simp: NMPcharn)

lemma NMPDA[rule-format]: $\text{DenyAll} \in \text{set } p \longrightarrow \text{not-MT } \text{Cp } p$
by (induct p, simp-all add: DAnMT)

lemma NMPiD[rule-format]: $\text{not-MT } \text{Cp } (\text{insertDeny } p)$
by (insert DAiniD [of p]) (erule NMPDA)

```

lemma list2FWpolicy2list[rule-format]:
   $Cp(\text{list2FWpolicy}(\text{policy2list } p)) = (Cp \ p)$ 
  apply (rule ext)
  apply (induct-tac  $p$ , simp-all)
  subgoal for  $x \ x1 \ x2$ 
    apply (case-tac  $x \in \text{dom} (Cp(x2))$ )
      apply (metis Cdom2 CeqEnd domIff p2lNmt)
      apply (metis CeqStart domIff p2lNmt nlpaux)
      done
  done

lemmas C-eq-Lemmas = none-MT-rulesRS2 none-MT-rulesrd SCp2l wp1n-RS2
wp1ID NMPiD waux2
          wp1alternative-RS1 p2lNmt list2FWpolicy2list wellformed-policy3-charn
wp1-eq

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

lemma C-eq-All-untilSorted:
  DenyAll $\in$ set(policy2list  $p$ )  $\implies$  all-in-list(policy2list  $p$ )  $l \implies$  allNetsDistinct(policy2list  $p$ )  $\implies$ 
     $Cp(\text{list2FWpolicy} (\text{sort} (\text{removeShadowRules2} (\text{remdups} (\text{rm-MT-rules} \ Cp (\text{insertDeny} (\text{removeShadowRules1} (\text{policy2list } p))))))) \ l)) =$ 
     $Cp \ p$ 
    apply (subst C-eq-sorted,simp-all add: C-eq-Lemmas)
    apply (subst C-eq-RS2,simp-all add: C-eq-Lemmas)
    apply (subst C-eq-rd,simp-all add: C-eq-Lemmas)
    apply (subst C-eq-RS3,simp-all add: C-eq-Lemmas)
    apply (subst C-eq-id,simp-all add: C-eq-Lemmas)
    done

lemma C-eq-All-untilSortedQ:
  DenyAll $\in$ set(policy2list  $p$ )  $\implies$  all-in-list(policy2list  $p$ )  $l \implies$  allNetsDistinct(policy2list  $p$ )  $\implies$ 
     $Cp(\text{list2FWpolicy} (\text{qsort} (\text{removeShadowRules2} (\text{remdups} (\text{rm-MT-rules} \ Cp (\text{insertDeny} (\text{removeShadowRules1} (\text{policy2list } p))))))) \ l)) =$ 
     $Cp \ p$ 
    apply (subst C-eq-sortedQ,simp-all add: C-eq-Lemmas)
    apply (subst C-eq-RS2,simp-all add: C-eq-Lemmas)
    apply (subst C-eq-rd,simp-all add: C-eq-Lemmas)
    apply (subst C-eq-RS3,simp-all add: C-eq-Lemmas)

```

```

apply (subst C-eq-id,simp-all add: C-eq-Lemmas)
done

lemma C-eq-All-untilSorted-withSimps:
  DenyAll ∈ set (policy2list p)  $\Rightarrow$  all-in-list (policy2list p) l  $\Rightarrow$ 
  allNetsDistinct (policy2list p)  $\Rightarrow$ 
  Cp(list2FWpolicy(sort(removeShadowRules2(remdups(rm-MT-rules Cp (insertDeny
    (removeShadowRules1(policy2list p)))))) l)) =
  Cp p
  by (simp-all add: C-eq-Lemmas C-eq-subst-Lemmas)

lemma C-eq-All-untilSorted-withSimpsQ:
  DenyAll ∈ set (policy2list p)  $\Rightarrow$  all-in-list (policy2list p) l  $\Rightarrow$ 
  allNetsDistinct (policy2list p)  $\Rightarrow$ 
  Cp(list2FWpolicy(qsort(removeShadowRules2(remdups(rm-MT-rules Cp (insertDeny
    (removeShadowRules1(policy2list p)))))) l)) =
  Cp p
  by (simp-all add: C-eq-Lemmas C-eq-subst-Lemmas)

lemma InDomConc[rule-format]:  $p \neq [] \rightarrow x \in \text{dom} (\text{Cp} (\text{list2FWpolicy} (p))) \rightarrow$ 
   $x \in \text{dom} (\text{Cp} (\text{list2FWpolicy} (a\#p)))$ 
  apply (induct p, simp-all)
  subgoal for a p
    apply (case-tac p = [], simp-all add: dom-def Cp.simps)
    done
  done

lemma not-in-member[rule-format]: member a b  $\rightarrow$   $x \notin \text{dom} (\text{Cp} b) \rightarrow x \notin \text{dom} (\text{Cp}$ 
  a)
  by (induct b)(simp-all add: dom-def Cp.simps)

lemma src-in-sdnets[rule-format]:
   $\neg \text{member DenyAll } x \rightarrow p \in \text{dom} (\text{Cp } x) \rightarrow \text{subnetsOfAdr} (\text{src } p) \cap (\text{fst-set} (\text{sdnets}$ 
  x))  $\neq \{\}$ 
  apply (induct rule: Combinators.induct)
    apply (simp-all add: fst-set-def subnetsOfAdr-def PLemmas, rename-tac x1 x2)
  apply (intro impI)
  apply (simp add: fst-set-def)
  subgoal for x1 x2
    apply (case-tac p ∈ dom (Cp x2))
    apply (rule subnetAux)
    apply (auto simp: PLemmas)
  done
  done

```

```

lemma dest-in-sdnets[rule-format]:
   $\neg \text{member } \text{DenyAll } x \longrightarrow p \in \text{dom} (\text{Cp } x) \longrightarrow \text{subnetsOfAdr} (\text{dest } p) \cap (\text{snd-set} (\text{sdnets } x)) \neq \{\}$ 
  apply (induct rule: Combinators.induct)
    apply (simp-all add: snd-set-def subnetsOfAdr-def PLemmas, rename-tac x1 x2)
  apply (intro impI, simp add: snd-set-def)
  subgoal for x1 x2
    apply (case-tac p ∈ dom (Cp x2))
    apply (rule subnetAux)
    apply (auto simp: PLemmas)
  done
done

lemma sdnets-in-subnets[rule-format]:
 $p \in \text{dom} (\text{Cp } x) \longrightarrow \neg \text{member } \text{DenyAll } x \longrightarrow$ 
 $(\exists (a,b) \in \text{sdnets } x. a \in \text{subnetsOfAdr} (\text{src } p) \wedge b \in \text{subnetsOfAdr} (\text{dest } p))$ 
  apply (rule Combinators.induct)
    apply (simp-all add: PLemmas subnetsOfAdr-def)
  apply (intro impI, simp)
  subgoal for x1 x2
    apply (case-tac p ∈ dom (Cp (x2)))
    apply (auto simp: PLemmas subnetsOfAdr-def)
  done
done

lemma disjSD-no-p-in-both[rule-format]:
 $\llbracket \text{disjSD-2 } x \ y; \neg \text{member } \text{DenyAll } x; \neg \text{member } \text{DenyAll } y;$ 
 $p \in \text{dom} (\text{Cp } x); p \in \text{dom} (\text{Cp } y) \rrbracket \implies \text{False}$ 
  apply (rule-tac A = sdnets x and B = sdnets y and D = src p and F = dest p in tndFalse)
  by (auto simp: dest-in-sdnets src-in-sdnets sdnets-in-subnets disjSD-2-def)

lemma list2FWpolicy-eq:
 $zs \neq [] \implies \text{Cp} (\text{list2FWpolicy} (x \oplus y \# z)) \ p = \text{Cp} (x \oplus \text{list2FWpolicy} (y \# z)) \ p$ 
  by (metis ConcAssoc l2p-aux list2FWpolicy.simps(2))

lemma dom-sep[rule-format]:
 $x \in \text{dom} (\text{Cp} (\text{list2FWpolicy } p)) \longrightarrow x \in \text{dom} (\text{Cp} (\text{list2FWpolicy} (\text{separate } p)))$ 
proof (induct p rule: separate.induct,simp-all, goal-cases)
  case (1 v va y z) then show ?case
    apply (intro conjI impI)
    apply (simp, drule mp)
    apply (case-tac x ∈ dom (Cp (DenyAllFromTo v va)))

```

```

apply (metis CConcStartA domIff l2p-aux2 list2FWpolicyconc not-Cons-self )
apply (subgoal-tac  $x \in \text{dom} (\text{Cp} (\text{list2FWpolicy} (y \# z)))$ )
  apply (metis CConcStartA Cdom2 domIff l2p-aux2 list2FWpolicyconc nlpaux)
  apply (subgoal-tac  $x \in \text{dom} (\text{Cp} (\text{list2FWpolicy} ((\text{DenyAllFromTo } v \text{ va}) \# y \# z)))$ )
    apply (simp add: dom-def Cp.simps,simp-all)
apply (case-tac  $x \in \text{dom} (\text{Cp} (\text{DenyAllFromTo } v \text{ va}))$ , simp-all)
apply (subgoal-tac  $x \in \text{dom} (\text{Cp} (\text{list2FWpolicy} (y \# z)))$ )
  apply (metis InDomConc sepnMT list.simps(2))
apply (subgoal-tac  $x \in \text{dom} (\text{Cp} (\text{list2FWpolicy} ((\text{DenyAllFromTo } v \text{ va}) \# y \# z)))$ )
  by (simp-all add: dom-def Cp.simps)
next
case ( $\lambda v \text{ va } vb \text{ y } z$ ) then show ?case
apply (intro impI conjI,simp)
apply (case-tac  $x \in \text{dom} (\text{Cp} (\text{AllowPortFromTo } v \text{ va } vb))$ )
  apply (metis CConcStartA domIff l2p-aux2 list2FWpolicyconc not-Cons-self )
  apply (subgoal-tac  $x \in \text{dom} (\text{Cp} (\text{list2FWpolicy} (y \# z)))$ )
    apply (metis CConcStartA Cdom2 InDomConc domIff l2p-aux2 list2FWpolicyconc nlpaux)
      apply (simp add: dom-def Cp.simps, simp-all)
apply (case-tac  $x \in \text{dom} (\text{Cp} (\text{AllowPortFromTo } v \text{ va } vb))$ , simp-all)
apply (subgoal-tac  $x \in \text{dom} (\text{Cp} (\text{list2FWpolicy} (y \# z))),simp)
  apply (metis Conc-not-MT InDomConc sepnMT)
apply (metis domIff nlpaux)
done
next
case ( $\lambda v \text{ va } y \text{ z}$ ) then show ?case
apply (intro conjI impI,simp)
apply (drule mp)
apply (case-tac  $x \in \text{dom} (\text{Cp} ((v \oplus va)))$ )
  apply (metis Cp.simps(4) CConcStartA ConcAssoc domIff list2FWpolicy2list list2FWpolicyconc p2lNmt)
defer 1
apply simp-all
apply (case-tac  $x \in \text{dom} (\text{Cp} ((v \oplus va))),simp-all)
apply (drule mp)
apply (simp add: Cp.simps dom-def)
apply (metis InDomConc list.simps(2) sepnMT)
apply (subgoal-tac  $x \in \text{dom} (\text{Cp} (\text{list2FWpolicy} (y \# z)))$ )
apply (case-tac  $x \in \text{dom} (\text{Cp} y),simp-all$ )
  apply (metis CConcStartA Cdom2 ConcAssoc domIff)
  apply (metis InDomConc domIff l2p-aux2 list2FWpolicyconc nlpaux)
apply (case-tac  $x \in \text{dom} (\text{Cp} y),simp-all$ )
  by (metis domIff nlpaux)
qed$$ 
```

```

lemma domdConcStart[rule-format]:
   $x \in \text{dom}(\text{Cp}(\text{list2FWpolicy}(a \# b))) \rightarrow x \notin \text{dom}(\text{Cp}(\text{list2FWpolicy } b)) \rightarrow x \in \text{dom}(\text{Cp}(a))$ 
  by (induct b, simp-all) (auto simp: PLemmas)

lemma sep-dom2-aux:
   $x \in \text{dom}(\text{Cp}(\text{list2FWpolicy}(a \oplus y \# z))) \Rightarrow x \in \text{dom}(\text{Cp}(a \oplus \text{list2FWpolicy}(y \# z)))$ 
  by auto (metis list2FWpolicy-eq p2lNmt)

lemma sep-dom2-aux2:
   $(x \in \text{dom}(\text{Cp}(\text{list2FWpolicy}(\text{separate}(y \# z)))) \rightarrow x \in \text{dom}(\text{Cp}(\text{list2FWpolicy}(y \# z)))) \Rightarrow$ 
   $x \in \text{dom}(\text{Cp}(\text{list2FWpolicy}(a \# \text{separate}(y \# z)))) \Rightarrow$ 
   $x \in \text{dom}(\text{Cp}(\text{list2FWpolicy}(a \oplus y \# z)))$ 
  by (metis CConcStartA InDomConc domdConcStart list.simps(2) list2FWpolicy.simps(2) list2FWpolicyconc)

lemma sep-dom2[rule-format]:
   $x \in \text{dom}(\text{Cp}(\text{list2FWpolicy}(\text{separate } p))) \rightarrow x \in \text{dom}(\text{Cp}(\text{list2FWpolicy}(p)))$ 
  by (rule separate.induct) (simp-all add: sep-dom2-aux sep-dom2-aux2)

lemma sepDom:  $\text{dom}(\text{Cp}(\text{list2FWpolicy } p)) = \text{dom}(\text{Cp}(\text{list2FWpolicy}(\text{separate } p)))$ 
  by (rule equalityI) (rule subsetI, (erule dom-sep|erule sep-dom2))+

lemma C-eq-s-ext[rule-format]:
   $p \neq [] \rightarrow \text{Cp}(\text{list2FWpolicy}(\text{separate } p)) a = \text{Cp}(\text{list2FWpolicy } p) a$ 
proof (induct rule: separate.induct, goal-cases)
  case (1 x) thus ?case
    apply (cases x = [],simp-all)
    apply (cases a ∈ dom (Cp (list2FWpolicy x)))
    apply (subgoal-tac a ∈ dom (Cp (list2FWpolicy (separate x))))
    apply (metis Cdom2 list2FWpolicyconc sepDom sepMT)
    apply (metis sepDom)
    by (metis nlpaux sepDom list2FWpolicyconc sepMT)
  next
  case (2 v va y z) thus ?case
    apply (cases z = [],simp-all)
    apply (intro conjI impI|simp)+
    apply (simp add: PLemmas(8) UPFDefs(8) list2FWpolicyconc sepMT)
    by (metis (mono-tags, lifting) Conc-not-MT Cdom2 list2FWpolicy-eq nlpaux sepDom l2p-aux sepMT)
  next

```

```

case (3 v va vb y z) thus ?case
  apply (cases z = [], simp-all)
    apply (simp add: PLemmas(8) UPFDefs(8) list2FWpolicyconc sepnMT)
    by (metis (no-types, opaque-lifting) Conc-not-MT Cdom2 nlpaux domIff l2p-aux sepnMT)
next
  case (4 v va y z) thus ?case
    apply (cases z = [], simp-all)
      apply (simp add: PLemmas(8) UPFDefs(8) l2p-aux sepnMT)
      by (metis (no-types, lifting) ConcAssoc PLemmas(8) UPFDefs(8) list.distinct(1)
          list2FWpolicyconc sepnMT)
next
  case 5 thus ?case by simp
next
  case 6 thus ?case by simp
next
  case 7 thus ?case by simp
next
  case 8 thus ?case by simp
qed

```

lemma C-eq-s: $p \neq [] \implies Cp(\text{list2FWpolicy}(\text{separate } p)) = Cp(\text{list2FWpolicy } p)$
by (rule ext) (simp add: C-eq-s-ext)

lemmas sortnMTQ = NormalisationIntegerPortProof.C-eq-Lemmas-sep(14)
lemmas C-eq-Lemmas-sep = C-eq-Lemmas sortnMT sortnMTQ RS2-NMT NMPrd
not-MTimpnotMT

lemma C-eq-until-separated:
 $\text{DenyAll} \in \text{set}(\text{policy2list } p) \implies \text{all-in-list}(\text{policy2list } p) l \implies \text{allNetsDistinct}(\text{policy2list } p) \implies$
 $Cp(\text{list2FWpolicy}(\text{separate}(\text{sort}(\text{removeShadowRules2}(\text{remdups}(\text{rm-MT-rules } Cp(\text{insertDeny}(\text{removeShadowRules1}(\text{policy2list } p))))))))) =$
 $Cp p$
by (simp add: C-eq-All-untilSorted-withSimps C-eq-s wellformed1-alternative-sorted
wp1ID wp1n-RS2)

lemma C-eq-until-separatedQ:
 $\text{DenyAll} \in \text{set}(\text{policy2list } p) \implies \text{all-in-list}(\text{policy2list } p) l \implies$
 $\text{allNetsDistinct}(\text{policy2list } p) \implies$
 $Cp(\text{list2FWpolicy}(\text{separate}(\text{qsort}(\text{removeShadowRules2}(\text{remdups}(\text{rm-MT-rules } Cp(\text{insertDeny}(\text{removeShadowRules1}(\text{policy2list } p))))))))) =$
 $Cp p$

```

by (simp add: C-eq-All-untilSorted-withSimpsQ C-eq-s setnMT wp1ID wp1n-RS2)

lemma domID[rule-format]:
   $p \neq [] \wedge x \in \text{dom}(\text{Cp}(\text{list2FWpolicy } p)) \longrightarrow x \in \text{dom}(\text{Cp}(\text{list2FWpolicy}(\text{insertDenies } p)))$ 
proof(induct p)
  case Nil then show ?case by simp
next
  case (Cons a p) then show ?case
  proof(cases p=[], goal-cases)
    case 1 then show ?case
      apply(simp) apply(rule impI)
      apply(cases a, simp-all)
      apply(simp-all add: Cp.simps dom-def)+
      by auto
  next
    case 2 then show ?case
    proof(cases x ∈ dom(Cp(list2FWpolicy p)), goal-cases)
      case 1 then show ?case
        apply simp apply(rule impI)
        apply(cases a, simp-all)
        apply(metis InDomConc idNMT)
        apply(rule InDomConc, simp-all add: idNMT)+
        done
    next
      case 2 then show ?case
        apply simp apply(rule impI)
        proof(cases x ∈ dom(Cp(list2FWpolicy(insertDenies p))), goal-cases)
          case 1 then show ?case
            proof(induct a)
              case DenyAll then show ?case by simp
            next
              case (DenyAllFromTo src dest) then show ?case
                by simp (rule InDomConc, simp add: idNMT)
            next
              case (AllowPortFromTo src dest port) then show ?case
                by simp (rule InDomConc, simp add: idNMT)
            next
              case (Conc - -) then show ?case
                by simp(rule InDomConc, simp add: idNMT)
            qed
        next
          case 2 then show ?case
        proof(induct a)
    qed
  qed
qed

```

```

case DenyAll then show ?case by simp
next
  case (DenyAllFromTo src dest) then show ?case
    by(simp,metis domIff CCConcStartA list2FWpolicyconc nlpaux Cdom2)
next
  case (AllowPortFromTo src dest port) then show ?case
    by(simp,metis domIff CCConcStartA list2FWpolicyconc nlpaux Cdom2)
next
  case (Conc - -) then show ?case
    by simp (metis CCConcStartA Cdom2 Conc(5) ConcAssoc domIff domdConc-
Start)
    qed
  qed
  qed
  qed
  qed
lemma DA-is-deny:
  
$$x \in \text{dom} (\text{Cp} (\text{DenyAllFromTo } a \ b \oplus \text{DenyAllFromTo } b \ a \oplus \text{DenyAllFromTo } a \ b)) \implies \text{Cp} (\text{DenyAllFromTo } a \ b \oplus \text{DenyAllFromTo } b \ a \oplus \text{DenyAllFromTo } a \ b) \ x = \text{Some} (\text{deny} ())$$

  by (case-tac  $x \in \text{dom} (\text{Cp} (\text{DenyAllFromTo } a \ b))$ ) (simp-all add: PLemmas split: if-splits)

lemma iDdomAux[rule-format]:
  
$$p \neq [] \implies x \notin \text{dom} (\text{Cp} (\text{list2FWpolicy } p)) \implies x \in \text{dom} (\text{Cp} (\text{list2FWpolicy} (\text{insertDenies } p))) \implies \text{Cp} (\text{list2FWpolicy} (\text{insertDenies } p)) \ x = \text{Some} (\text{deny} ())$$

proof (induct p)
  case Nil thus ?case by simp
next
  case (Cons y ys) thus ?case
    proof (cases y)
      case DenyAll then show ?thesis by simp
    next
      case (DenyAllFromTo a b) then show ?thesis
        using DenyAllFromTo Cons apply simp
        apply (rule impI)+
      proof (cases ys = [], goal-cases)
        case 1 then show ?case by (simp add: DA-is-deny)
      next
        case 2 then show ?case
        apply simp

```

```

apply (drule mp)
  apply (metis DenyAllFromTo InDomConc )
  apply (cases x ∈ dom (Cp (list2FWpolicy (insertDenies ys))),simp-all)
    apply (metis Cdom2 DenyAllFromTo idNMT list2FWpolicyconc)
    apply (subgoal-tac Cp (list2FWpolicy (DenyAllFromTo a b ⊕
                                              DenyAllFromTo b a ⊕ DenyAllFromTo a b#insertDenies
                                              ys)) x =
              Cp ((DenyAllFromTo a b ⊕ DenyAllFromTo b a ⊕
                  DenyAllFromTo a b)) x )
      apply (metis DA-is-deny DenyAllFromTo domdConcStart)
      apply (metis DenyAllFromTo l2p-aux2 list2FWpolicyconc nlpaux)
      done
  qed
next
  case (AllowPortFromTo a b c) then show ?thesis using Cons AllowPortFromTo
  proof (cases ys = [], goal-cases)
    case 1 then show ?case
      apply (simp,intro impI)
      apply (subgoal-tac x ∈ dom (Cp (DenyAllFromTo a b ⊕ DenyAllFromTo b a)))
        apply (auto simp: PLemmas split: if-splits)
        done
    next
    case 2 then show ?case
      apply (simp, intro impI)
      apply (drule mp)
      apply (metis AllowPortFromTo InDomConc)
      apply (cases x ∈ dom (Cp (list2FWpolicy (insertDenies ys))),simp-all)
        apply (metis AllowPortFromTo Cdom2 idNMT list2FWpolicyconc)
        apply (subgoal-tac Cp (list2FWpolicy (DenyAllFromTo a b ⊕
                                              DenyAllFromTo b a ⊕
                                              AllowPortFromTo a b c#insertDenies ys)) x =
              Cp ((DenyAllFromTo a b ⊕ DenyAllFromTo b a)) x )
          apply (auto simp: PLemmas split: if-splits)[1]
          by (metis AllowPortFromTo CConcStartA ConcAssoc idNMT list2FWpolicyconc
              nlpaux)
      qed
    next
    case (Conc a b) then show ?thesis
    proof (cases ys = [], goal-cases)
      case 1 then show ?case
        apply (simp,intro impI)
        apply (subgoal-tac x ∈ dom (Cp (DenyAllFromTo (first-srcNet a) (first-destNet
          a) ⊕
          DenyAllFromTo (first-destNet a) (first-srcNet a)))))


```

```

    by (auto simp: PLemmas split: if-splits)
next
  case 2 then show ?case
    apply(simp,intro impI)
    apply(cases x ∈ dom (Cp (list2FWpolicy (insertDenies ys))))
    apply (metis Cdom2 Conc Cons InDomConc idNMT list2FWpolicyconc)
    apply (subgoal-tac Cp (list2FWpolicy(DenyAllFromTo (first-srcNet a)(first-destNet
a) ⊕
DenyAllFromTo (first-destNet a) (first-srcNet
a)⊕
a ⊕ b#insertDenies ys)) x =
Cp ((DenyAllFromTo(first-srcNet a) (first-destNet a) ⊕
DenyAllFromTo (first-destNet a)(first-srcNet a) ⊕ a ⊕
b)) x)
      apply simp
      defer 1
      apply (metis Conc l2p-aux2 list2FWpolicyconc nlpaux)
      apply (subgoal-tac Cp((DenyAllFromTo(first-srcNet a)(first-destNet a) ⊕
DenyAllFromTo (first-destNet a)(first-srcNet a)⊕ a ⊕ b))
x =
Cp((DenyAllFromTo (first-srcNet a)(first-destNet a)⊕
DenyAllFromTo (first-destNet a) (first-srcNet a))) x )
      apply simp
      defer 1
      apply (metis CCConcStartA Conc ConcAssoc nlpaux)
      by (auto simp: PLemmas split: if-splits)
qed
qed
qed

```

lemma *iD-isD[rule-format]:*

```

 $p \neq [] \rightarrow x \notin \text{dom} (\text{Cp} (\text{list2FWpolicy } p)) \rightarrow$ 
 $\text{Cp} (\text{DenyAll} \oplus \text{list2FWpolicy} (\text{insertDenies } p)) x = \text{Cp} \text{ DenyAll } x$ 
apply (case-tac x ∈ dom (Cp (list2FWpolicy (insertDenies p))))
apply (simp add: Cp.simps(1) Cdom2 iDdomAux deny-all-def)
using NormalisationIPPProofs.nlpaux
by blast

```

lemma *inDomConc:*

```

 $x \notin \text{dom} (\text{Cp } a) \implies x \notin \text{dom} (\text{Cp} (\text{list2FWpolicy } p)) \implies x \notin \text{dom} (\text{Cp}$ 
 $(\text{list2FWpolicy}(a \# p)))$ 
by (metis domdConcStart)

```

lemma *domsdisj[rule-format]:*

```

 $p \neq [] \rightarrow (\forall x s. s \in set p \wedge x \in dom (Cp A) \rightarrow x \notin dom (Cp s)) \rightarrow y \in dom (Cp A) \rightarrow$ 
 $y \notin dom (Cp (list2FWpolicy p))$ 
proof (induct p)
  case Nil show ?case by simp
next
  case (Cons a p) show ?case
    apply (case-tac p = [], simp)
    apply (rule-tac x = y in spec)
    apply (simp add: split-tupled-all)
    by (metis Cons.hyps inDomConc list.set-intros(1) list.set-intros(2))
qed

```

lemma *isSepaux*:

```

 $p \neq [] \Rightarrow noDenyAll (a \# p) \Rightarrow separated (a \# p) \Rightarrow$ 
 $x \in dom (Cp (DenyAllFromTo (first-srcNet a) (first-destNet a) \oplus$ 
 $DenyAllFromTo (first-destNet a) (first-srcNet a) \oplus a)) \Rightarrow$ 
 $x \notin dom (Cp (list2FWpolicy p))$ 
apply (rule-tac A = (DenyAllFromTo (first-srcNet a) (first-destNet a) \oplus
DenyAllFromTo (first-destNet a) (first-srcNet a) \oplus a) in domsdisj,
simp-all)
apply (rule notI)
subgoal for xa s
  apply (rule-tac p = xa and x = (DenyAllFromTo (first-srcNet a) (first-destNet a) \oplus
DenyAllFromTo (first-destNet a) (first-srcNet a) \oplus a)
and y = s in disjSD-no-p-in-both, simp-all)
  using disjSD2aux noDA apply blast
  using noDA
  by blast
done

```

lemma *none-MT-rulessep*[rule-format]: *none-MT-rules* Cp p \rightarrow *none-MT-rules* Cp (separate p)
by (*induct p rule: separate.induct*) (simp-all add: Cp.simps map-add-le-mapE
map-le-antisym)

lemma *dom-id*:

```

noDenyAll (a \# p) \Rightarrow separated (a \# p) \Rightarrow p \neq [] \Rightarrow
x \notin dom (Cp (list2FWpolicy p)) \Rightarrow x \in dom (Cp (a)) \Rightarrow
x \notin dom (Cp (list2FWpolicy (insertDenies p)))
apply (rule-tac a = a in isSepaux, simp-all)
using idNMT apply blast
using noDAID apply blast
using id-aux4 noDA1eq sepNetsID apply blast

```

```

by (simp add: NormalisationIPPProofs.Cdom2 domIff)

lemma C-eq-iD-aux2[rule-format]:
  noDenyAll1 p --> separated p --> p ≠ [] --> x ∈ dom (Cp (list2FWpolicy p)) -->
  Cp(list2FWpolicy (insertDenies p)) x = Cp(list2FWpolicy p) x
proof (induct p)
  case Nil thus ?case by simp
next
  case (Cons y ys) thus ?case
    using Cons proof (cases y)
    case DenyAll thus ?thesis using Cons DenyAll apply simp
      apply (case-tac ys = [], simp-all)
      apply (case-tac x ∈ dom (Cp (list2FWpolicy ys)), simp-all)
      apply (metis Cdom2 domID idNMT list2FWpolicyconc noDA1eq)
      apply (metis DenyAll iD-isD idNMT list2FWpolicyconc nlpaux)
      done
  next
    case (DenyAllFromTo a b) thus ?thesis
      using Cons apply simp
      apply (rule impI|rule allI|rule conjI|simp)+
      apply (case-tac ys = [], simp-all)
      apply (metis Cdom2 ConcAssoc DenyAllFromTo)
      apply (case-tac x ∈ dom (Cp (list2FWpolicy ys)), simp-all)
      apply (drule mp)
      apply (metis noDA1eq)
      apply (case-tac x ∈ dom (Cp (list2FWpolicy (insertDenies ys))))
      apply (metis Cdom2 DenyAllFromTo idNMT list2FWpolicyconc)
      apply (metis domID)
      apply (case-tac x ∈ dom (Cp (list2FWpolicy (insertDenies ys))))
      apply (subgoal-tac Cp (list2FWpolicy (DenyAllFromTo a b ⊕ DenyAllFromTo b
a ⊕
          DenyAllFromTo a b # insertDenies ys)) x = Some (deny ()))
      apply simp-all
      apply (subgoal-tac Cp (list2FWpolicy (DenyAllFromTo a b # ys)) x =
          Cp ((DenyAllFromTo a b)) x)
      apply (simp add: PLemmas, simp split: if-splits)
      apply (metis list2FWpolicyconc nlpaux)
      apply (metis Cdom2 DenyAllFromTo iD-isD iDdomAux idNMT list2FWpolicyconc)
      apply (metis Cdom2 DenyAllFromTo domIff idNMT list2FWpolicyconc nlpaux)
      done
  next
    case (AllowPortFromTo a b c) thus ?thesis
      using AllowPortFromTo Cons apply simp
      apply (rule impI|rule allI|rule conjI|simp)+

```

```

apply (case-tac ys = [], simp-all)
apply (metis Cdom2 ConcAssoc AllowPortFromTo)
apply (case-tac x ∈ dom (Cp (list2FWpolicy ys)), simp-all)
apply (drule mp)
apply (metis noDA1eq)
apply (case-tac x ∈ dom (Cp (list2FWpolicy (insertDenies ys))))
apply (metis Cdom2 AllowPortFromTo idNMT list2FWpolicyconc)
apply (metis domID)
apply (subgoal-tac x ∈ dom (Cp (AllowPortFromTo a b c)))
apply (case-tac x ∉ dom (Cp (list2FWpolicy (insertDenies ys))), simp-all)
apply (metis AllowPortFromTo Cdom2 ConcAssoc l2p-aux2 list2FWpolicyconc
nlpaux)
apply (meson Combinators.distinct(3) FWNormalisationCore.member.simps(4)
NormalisationIPPProofs.dom-id noDenyAll.simps(1) separated.simps(1))
apply (metis AllowPortFromTo domdConcStart)
done

next
case (Conc a b) thus ?thesis
using Cons Conc apply simp
apply (intro impI allI conjI|simp)+
apply (case-tac ys = [], simp-all)
apply (metis Cdom2 ConcAssoc Conc)
apply (case-tac x ∈ dom (Cp (list2FWpolicy ys)), simp-all)
apply (drule mp)
apply (metis noDA1eq)
apply (case-tac x ∈ dom (Cp (a ⊕ b)))
apply (case-tac x ∉ dom (Cp (list2FWpolicy (insertDenies ys))), simp-all)
apply (subst list2FWpolicyconc)
apply (rule idNMT, simp)
apply (metis domID)
apply (metis Cdom2 Conc idNMT list2FWpolicyconc)
apply (metis Cdom2 Conc domIff idNMT list2FWpolicyconc )
apply (case-tac x ∈ dom (Cp (a ⊕ b)))
apply (case-tac x ∉ dom (Cp (list2FWpolicy (insertDenies ys))), simp-all)
apply (subst list2FWpolicyconc)
apply (rule idNMT, simp)
apply (metis Cdom2 Conc ConcAssoc list2FWpolicyconc nlpaux)
apply (metis (lifting, no-types) FWNormalisationCore.member.simps(1) Normal-
isationIPPProofs.dom-id noDenyAll.simps(1) separated.simps(1))
apply (metis Conc domdConcStart)
done

qed
qed

```

```

lemma C-eq-iD:
  separated p ==> noDenyAll1 p ==> wellformed-policy1-strong p ==>
  Cp(list2FWpolicy (insertDenies p)) = Cp (list2FWpolicy p)
  by (rule ext) (metis CConcStartA C-eq-iD-aux2 DAAux wp1-alternative-not-mt
wp1n-tl)

lemma noDAsortQ[rule-format]: noDenyAll1 p --> noDenyAll1 (qsort p l)
proof (cases p)
  case Nil then show ?thesis by simp
next
  case (Cons a list) show ?thesis
    apply (insert `p = a # list`, simp-all)
  proof (cases a = DenyAll)
    case True
      assume * : a = DenyAll
      show noDenyAll1(a # list) -->
        noDenyAll1(qsort[y←list . ¬ smaller a y l] l @ a # qsort [y←list . smaller
a y l] l)
        using noDAsortQ by fastforce
    next
    case False
      assume * : a ≠ DenyAll
      have ** : noDenyAll1 (a # list) ==> noDenyAll (a # list) by(case-tac a,simp-all
add:*)
      show noDenyAll1(a # list) -->
        noDenyAll1(qsort[y←list . ¬ smaller a y l] l @ a # qsort [y←list . smaller
a y l] l)
        apply (insert *,rule impI)
        apply (rule noDA1eq, frule **)
        by (metis append-Cons append-Nil nDAeqSet qsort.simps(2) set-sortQ)
qed
qed

lemma NetsCollectedSortQ:
  distinct p ==> noDenyAll1 p ==> all-in-list p l ==> singleCombinators p ==>
  NetsCollected (qsort p l)
  by(metis C-eqLemmas-id(22))

lemmas CLemmas = nMTSort nMTSortQ none-MT-rulesRS2 none-MT-rulesrd
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:
  DenyAll ∈ set(policy2list p)  $\implies$  all-in-list(policy2list p) l  $\implies$  allNetsDistinct
  (policy2list p)  $\implies$ 
    Cp (list2FWpolicy((insertDenies(separate(sort(removeShadowRules2
      (remdups(rm-MT-rules Cp (insertDeny (removeShadowRules1 (policy2list
      p))))))) l)))) =
    Cp p
  by (subst C-eq-iD,simp-all add: C-eqLemmas-id) (rule C-eq-until-separated, simp-all)

lemma C-eq-Until-InsertDeniesQ:
  DenyAll ∈ set (policy2list p)  $\implies$  all-in-list (policy2list p) l  $\implies$ 
  allNetsDistinct (policy2list p)  $\implies$ 
    Cp (list2FWpolicy ((insertDenies (separate (qsort (removeShadowRules2
      (remdups (rm-MT-rules Cp (insertDeny (removeShadowRules1 (policy2list
      p))))))) l)))) =
    Cp p
  apply (subst C-eq-iD, simp-all add: C-eqLemmas-id)
  apply (metis WP1rd set-qsort wellformed1-sortedQ wellformed-eq wp1ID
  wp1-alternativesep
    wp1-aux1aa wp1n-RS2 wp1n-RS3)
  apply (rule C-eq-until-separatedQ)
  by simp-all

lemma C-eq-RD-aux[rule-format]: Cp (p) x = Cp (removeDuplicates p) x
  apply (induct p, simp-all)
  apply (intro conjI impI)
  by (metis Cdom2 domIff nlpaux not-in-member) (metis Cp.simps(4) CConcStartaux
  Cdom2 domIff)

lemma C-eq-RAD-aux[rule-format]:
  p ≠ []  $\longrightarrow$  Cp (list2FWpolicy p) x = Cp (list2FWpolicy (removeAllDuplicates p)) x
proof (induct p)
  case Nil show ?case by simp
next
  case (Cons a p) then show ?case
  apply simp-all
  apply (case-tac p = [], simp-all)
  apply (metis C-eq-RD-aux)
  apply (subst list2FWpolicyconc, simp)
  apply (case-tac x ∈ dom (Cp (list2FWpolicy p)))
  apply (subst list2FWpolicyconc)
  apply (rule rADnMT, simp)

```

```

apply (subst Cdom2,simp)
apply (simp add: NormalisationIPPProofs.Cdom2 domIff)
by (metis C-eq-RD-aux nlpaux domIff list2FWpolicyconc rADnMT)
qed

lemma C-eq-RAD:
 $p \neq [] \implies Cp(\text{list2FWpolicy } p) = Cp(\text{list2FWpolicy}(\text{removeAllDuplicates } p))$ 
by (rule ext) (erule C-eq-RAD-aux)

lemma C-eq-compile:
 $\text{DenyAll} \in \text{set}(\text{policy2list } p) \implies \text{all-in-list}(\text{policy2list } p) l \implies$ 
 $\text{allNetsDistinct}(\text{policy2list } p) \implies$ 
 $Cp(\text{list2FWpolicy}(\text{removeAllDuplicates}(\text{insertDenies}(\text{separate}(\text{sort}(\text{removeShadowRules2}(\text{remdups}(\text{rm-MT-rules } Cp(\text{insertDeny}(\text{removeShadowRules1}(\text{policy2list } p))))))) l)))) = Cp p$ 
by (metis C-eq-RAD C-eq-Until-InsertDenies removeAllDuplicates.simps(2))

lemma C-eq-compileQ:
 $\text{DenyAll} \in \text{set}(\text{policy2list } p) \implies \text{all-in-list}(\text{policy2list } p) l \implies \text{allNetsDistinct}(\text{policy2list } p) \implies$ 
 $Cp(\text{list2FWpolicy}(\text{removeAllDuplicates}(\text{insertDenies}(\text{separate}(\text{qsort}(\text{removeShadowRules2}(\text{remdups}(\text{rm-MT-rules } Cp(\text{insertDeny}(\text{removeShadowRules1}(\text{policy2list } p))))))) l)))) = Cp p$ 
apply (subst C-eq-RAD[symmetric])
apply (rule idNMT)
apply (metis WP1rd sepnMT sortnMTQ wellformed-policy1-strong.simps(1) wp1ID
wp1n-RS2 wp1n-RS3)
apply (rule C-eq-Until-InsertDeniesQ, simp-all)
done

lemma C-eq-normalizePr:
 $\text{DenyAll} \in \text{set}(\text{policy2list } p) \implies \text{allNetsDistinct}(\text{policy2list } p) \implies$ 
 $\text{all-in-list}(\text{policy2list } p) (\text{Nets-List } p) \implies$ 
 $Cp(\text{list2FWpolicy}(\text{normalizePr } p)) = Cp p$ 
unfolding normalizePrQ-def
by (simp add: C-eq-compile normalizePr-def)

lemma C-eq-normalizePrQ:
 $\text{DenyAll} \in \text{set}(\text{policy2list } p) \implies \text{allNetsDistinct}(\text{policy2list } p) \implies$ 
 $\text{all-in-list}(\text{policy2list } p) (\text{Nets-List } p) \implies$ 
 $Cp(\text{list2FWpolicy}(\text{normalizePrQ } p)) = Cp p$ 
unfolding normalizePrQ-def
using C-eq-compileQ by auto

```

```

lemma domSubset3:  $\text{dom}(\text{Cp}(\text{DenyAll} \oplus x)) = \text{dom}(\text{Cp}(\text{DenyAll}))$ 
  by (simp add: PLemmas split-tupled-all split: option.splits)

lemma domSubset4:
 $\text{dom}(\text{Cp}(\text{DenyAllFromTo } x y \oplus \text{DenyAllFromTo } y x \oplus \text{AllowPortFromTo } x y dn)) =$ 
 $\text{dom}(\text{Cp}(\text{DenyAllFromTo } x y \oplus \text{DenyAllFromTo } y x))$ 
  by (simp add: PLemmas split: option.splits decision.splits) auto

lemma domSubset5:
 $\text{dom}(\text{Cp}(\text{DenyAllFromTo } x y \oplus \text{DenyAllFromTo } y x \oplus \text{AllowPortFromTo } y x dn)) =$ 
 $\text{dom}(\text{Cp}(\text{DenyAllFromTo } x y \oplus \text{DenyAllFromTo } y x))$ 
  by (simp add: PLemmas split: option.splits decision.splits) auto

lemma domSubset1:
 $\text{dom}(\text{Cp}(\text{DenyAllFromTo } \text{one two} \oplus \text{DenyAllFromTo } \text{two one} \oplus \text{AllowPortFromTo } \text{one two dn} \oplus x)) =$ 
 $\text{dom}(\text{Cp}(\text{DenyAllFromTo } \text{one two} \oplus \text{DenyAllFromTo } \text{two one} \oplus x))$ 
  by (simp add: PLemmas allow-all-def deny-all-def split: option.splits decision.splits)
  auto

lemma domSubset2:
 $\text{dom}(\text{Cp}(\text{DenyAllFromTo } \text{one two} \oplus \text{DenyAllFromTo } \text{two one} \oplus \text{AllowPortFromTo } \text{two one dn} \oplus x)) =$ 
 $\text{dom}(\text{Cp}(\text{DenyAllFromTo } \text{one two} \oplus \text{DenyAllFromTo } \text{two one} \oplus x))$ 
  by (simp add: PLemmas allow-all-def deny-all-def split: option.splits decision.splits)
  auto

lemma ConcAssoc2:  $\text{Cp}(X \oplus Y \oplus ((A \oplus B) \oplus D)) = \text{Cp}(X \oplus Y \oplus A \oplus B \oplus D)$ 
  by (simp add: Cp.simps)

lemma ConcAssoc3:  $\text{Cp}(X \oplus ((Y \oplus A) \oplus D)) = \text{Cp}(X \oplus Y \oplus A \oplus D)$ 
  by (simp add: Cp.simps)

lemma RS3-NMT[rule-format]:  $\text{DenyAll} \in \text{set } p \rightarrow$ 
   $\text{rm-MT-rules } \text{Cp } p \neq []$ 
  by (induct-tac p) (simp-all add: PLemmas)

lemma norm-notMT:  $\text{DenyAll} \in \text{set } (\text{policy2list } p) \implies \text{normalizePr } p \neq []$ 
  unfolding normalizePrQ-def
  by (simp add: DAiniD RS3-NMT RS2-NMT idNMT normalizePr-def rADnMT sep-nMT sortnMT)

```

```

lemma norm-notMTQ: DenyAll ∈ set (policy2list p)  $\implies$  normalizePrQ p  $\neq []$ 
  unfolding normalizePrQ-def
  by (simp add: DAiniD RS3-NMT sortnMTQ RS2-NMT idNMT rADnMT sepnMT)

```

```

lemma domDA: dom (Cp (DenyAll  $\oplus$  A)) = dom (Cp (DenyAll))
  by (rule domSubset3)

```

```

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 ∈ dom (Cp a)  $\longrightarrow$ 
  Cp (list2policyR (a # list)) p = Cp a p
  by (induct a # list rule:list2policyR.induct)
    (auto simp: Cp.simps dom-def map-add-def)

```

```

lemma list2policyR-End: p  $\notin$  dom (Cp a)  $\implies$ 
  Cp (list2policyR (a # list)) p = (Cp a  $\oplus$  list2policy (map Cp list)) p
  by (rule list2policyR.induct)
    (simp-all add: Cp.simps dom-def map-add-def list2policy-def split: option.splits)

```

```

lemma l2polR-eq-el[rule-format]: N  $\neq []$   $\longrightarrow$ 
  Cp( list2policyR N) p = (list2policy (map Cp N)) p
proof (induct N)
  case Nil show ?case by simp
next
  case (Cons a p) show ?case
    apply (insert Cons.hyps, simp-all add: list2policy-def)
    by (metis list2policyR-End list2policyR-Start domStart list2policy-def)
qed

```

```

lemma l2polR-eq:
  N  $\neq [] \implies$  Cp( list2policyR N) = (list2policy (map Cp N))
  by (auto simp: list2policy-def l2polR-eq-el )

```

```

lemma list2FWpolicyS-eq-el[rule-format]:
  Filter  $\neq [] \longrightarrow$  Cp (list2policyR Filter) p = Cp (list2FWpolicy (rev Filter)) p
  apply (induct-tac Filter)
  apply simp-all
subgoal for a list
  apply (case-tac list = [])
  apply simp-all

```

```

apply (case-tac  $p \in \text{dom} (Cp\ a)$ )
  apply simp-all
  apply (rule list2policyR-Start)
  apply simp-all
apply (subgoal-tac  $Cp (\text{list2policyR} (a \# \text{list})) p = Cp (\text{list2policyR} \text{list}) p$ )
  apply (subgoal-tac  $Cp (\text{list2FWpolicy} (\text{rev} \text{list} @ [a])) p = Cp (\text{list2FWpolicy} (\text{rev} \text{list})) p$ )
    apply simp
    apply (rule CConcStart2)
    apply simp
    apply simp
apply (case-tac list,simp-all)
apply (simp-all add: Cp.simps dom-def map-add-def)
done
done

lemma list2FWpolicys-eq:
 $\text{Filter} \neq [] \implies Cp (\text{list2policyR} \text{Filter}) = Cp (\text{list2FWpolicy} (\text{rev} \text{Filter}))$ 
by (rule ext, erule list2FWpolicys-eq-el)

lemma list2FWpolicys-eq-sym:
 $\text{Filter} \neq [] \implies Cp (\text{list2policyR} (\text{rev} \text{Filter})) = Cp (\text{list2FWpolicy} \text{Filter})$ 
by (metis list2FWpolicys-eq rev-is-Nil-conv rev-rev-ident)

lemma p-eq[rule-format]:  $p \neq [] \implies \text{list2policy} (\text{map} \ Cp (\text{rev} \ p)) = Cp (\text{list2FWpolicy} \ p)$ 
by (metis l2polR-eq list2FWpolicys-eq-sym rev.simps(1) rev-rev-ident)

lemma p-eq2[rule-format]:  $\text{normalizePr} \ x \neq [] \implies Cp (\text{list2FWpolicy} (\text{normalizePr} \ x)) = Cp \ x \implies \text{list2policy} (\text{map} \ Cp (\text{rev} (\text{normalizePr} \ x))) = Cp \ x$ 
by (simp add: p-eq)

lemma p-eq2Q[rule-format]:  $\text{normalizePrQ} \ x \neq [] \implies Cp (\text{list2FWpolicy} (\text{normalizePrQ} \ x)) = Cp \ x \implies \text{list2policy} (\text{map} \ Cp (\text{rev} (\text{normalizePrQ} \ x))) = Cp \ x$ 
by (simp add: p-eq)

lemma list2listNMT[rule-format]:  $x \neq [] \implies \text{map sem} \ x \neq []$ 
by (case-tac x) (simp-all)

lemma Norm-Distr2:

```

$r \circ-f ((P \otimes_2 (\text{list2policy } Q)) \circ d) =$
 $(\text{list2policy } ((P \otimes_L Q) (\otimes_2 r d)))$
by (rule ext, rule Norm-Distr-2)

lemma NATDistr:

$N \neq [] \implies F = Cp (\text{list2policyR } N) \implies$
 $((\lambda (x,y). x) \circ-f ((\text{NAT} \otimes_2 F) \circ (\lambda x. (x,x)))) =$
 $(\text{list2policy } ((\text{NAT} \otimes_L (\text{map } Cp N)) (\otimes_2$
 $(\lambda (x,y). x) (\lambda x. (x,x))))))$
by (simp add: l2polR-eq) (rule ext, rule Norm-Distr-2)

lemma C-eq-normalize-manual:

$\text{DenyAll} \in \text{set } (\text{policy2list } p) \implies \text{allNetsDistinct } (\text{policy2list } p) \implies$
 $\text{all-in-list } (\text{policy2list } p) l \implies$
 $Cp (\text{list2FWpolicy } (\text{normalize-manual-orderPr } p l)) = Cp p$
unfolding normalize-manual-orderPr-def
by (simp-all add: C-eq-compile)

lemma p-eq2-manualQ[rule-format]:

$\text{normalize-manual-orderPrQ } x l \neq [] \implies$
 $Cp (\text{list2FWpolicy } (\text{normalize-manual-orderPrQ } x l)) = Cp x \implies$
 $\text{list2policy } (\text{map } Cp (\text{rev } (\text{normalize-manual-orderPrQ } x l))) = Cp x$
by (simp add: p-eq)

lemma norm-notMT-manualQ: $\text{DenyAll} \in \text{set } (\text{policy2list } p) \implies \text{normalize-manual-orderPrQ } p l \neq []$
by (simp add: DAiniD RS3-NMT sortnMTQ RS2-NMT idNMT normalize-manual-orderPrQ-def rADnMT sepnMT)

lemma C-eq-normalizePr-manualQ:

$\text{DenyAll} \in \text{set } (\text{policy2list } p) \implies$
 $\text{allNetsDistinct } (\text{policy2list } p) \implies$
 $\text{all-in-list } (\text{policy2list } p) l \implies$
 $Cp (\text{list2FWpolicy } (\text{normalize-manual-orderPrQ } p l)) = Cp p$
by (simp add: normalize-manual-orderPrQ-def C-eq-compileQ)

lemma p-eq2-manual[rule-format]: $\text{normalize-manual-orderPr } x l \neq [] \implies$
 $Cp (\text{list2FWpolicy } (\text{normalize-manual-orderPr } x l)) = Cp x \implies$
 $\text{list2policy } (\text{map } Cp (\text{rev } (\text{normalize-manual-orderPr } x l))) = Cp x$
by (simp add: p-eq)

lemma norm-notMT-manual: $\text{DenyAll} \in \text{set } (\text{policy2list } p) \implies \text{normalize-manual-orderPr } p l \neq []$
unfolding normalize-manual-orderPr-def

```

by (simp add: idNMT rADnMT wellformed1-alternative-sorted wp1ID
wp1-alternatives sep wp1n-RS2)

```

As an example, how this theorems can be used for a concrete normalisation instantiation.

lemma normalizePrNAT:

```

DenyAll ∈ set (policy2list Filter) ==>
allNetsDistinct (policy2list Filter) ==>
all-in-list (policy2list Filter) (Nets-List Filter) ==>
((λ (x,y). x) o-f (((NAT ⊗2 Cp Filter) o (λx. (x,x))))) =
list2policy (((NAT ⊗L (map Cp (rev (normalizePr Filter)))) (⊗2) (λ (x,y). x) (λ x. (x,x))))
by (simp add: C-eq-normalizePr NATDistr list2FWpollicys-eq-sym norm-notMT)

```

```

lemma domSimpl[simp]: dom (Cp (A ⊕ DenyAll)) = dom (Cp (DenyAll))
by (simp add: PLemmas)

```

end

2.4 Stateful Network Protocols

```

theory
  StatefulFW
imports
  FTPVOIP
begin

```

end

2.4.1 Stateful Protocols: Foundations

```

theory
  StatefulCore
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 does not help here either, as the opening and closing of the ports takes place on the application layer. Therefore, a firewall needs to have some knowledge of the application protocols being run and track the states of these protocols. We next model this behaviour.

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

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

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

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.

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

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

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

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

```

fun packet-with-id where
  packet-with-id [] i = []
| packet-with-id (x#xs) i = (if id x = i then (x#(packet-with-id xs i)) else (packet-with-id
  xs i))

fun ids1 where
  ids1 i (x#xs) = (id x = i  $\wedge$  ids1 i xs)
| ids1 i [] = True

fun ids where
  ids a (x#xs) = (NetworkCore.id x  $\in$  a  $\wedge$  ids a xs)
| ids a [] = True

definition applyPolicy:: ('i  $\times$  ('i  $\mapsto$  'o))  $\mapsto$  'o

```

```

where       $applyPolicy = (\lambda (x,z). z\ x)$ 

end

```

2.4.2 The File Transfer Protocol (ftp)

```

theory
  FTP
imports
  StatefulCore
begin

```

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 $msg = ftp-init \mid ftp-port-request\ port \mid ftp-data \mid ftp-close \mid ftp-other$

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

$$\begin{aligned} \text{is-ftp-data} :: id \Rightarrow (\text{adr}_{ip}, \text{msg}) \text{ packet} \Rightarrow \text{bool where} \\ \text{is-ftp-data} = (\lambda i p. (id p = i \wedge \text{content } p = \text{ftp-data})) \end{aligned}$$
definition

$$\begin{aligned} \text{is-ftp-close} :: id \Rightarrow (\text{adr}_{ip}, \text{msg}) \text{ packet} \Rightarrow \text{bool where} \\ \text{is-ftp-close} = (\lambda i p. (id p = i \wedge \text{content } p = \text{ftp-close})) \end{aligned}$$
definition

$$\begin{aligned} \text{port-open} :: (\text{adr}_{ip}, \text{msg}) \text{ history} \Rightarrow id \Rightarrow port \Rightarrow \text{bool where} \\ \text{port-open} = (\lambda L a p. (\text{not-before}(\text{is-ftp-close } a) (\text{is-ftp-port-request } a p) L)) \end{aligned}$$
definition

$$\begin{aligned} \text{is-ftp-other} :: id \Rightarrow (\text{adr}_{ip}, \text{msg}) \text{ packet} \Rightarrow \text{bool where} \\ \text{is-ftp-other} = (\lambda i p. (id p = i \wedge \text{content } p = \text{ftp-other})) \end{aligned}$$
fun *are-ftp-other* **where**

$$\begin{aligned} \text{are-ftp-other } i (x \# xs) &= (\text{is-ftp-other } i x \wedge \text{are-ftp-other } i xs) \\ |\text{are-ftp-other } i [] &= \text{True} \end{aligned}$$
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**

$$\begin{aligned} \text{last-opened-port } i ((j, s, d, \text{ftp-port-request } p) \# xs) &= (\text{if } i = j \text{ then } p \text{ else } \text{last-opened-port } i xs) \\ |\text{last-opened-port } i (x \# xs) &= \text{last-opened-port } i xs \\ |\text{last-opened-port } x [] &= \text{undefined} \end{aligned}$$
fun *FTP-STA* :: $((\text{adr}_{ip}, \text{msg}) \text{ history}, \text{adr}_{ip}, \text{msg}) \text{ FWStateTransition}$
where

$$\begin{aligned} \text{FTP-STA } ((i, s, d, \text{ftp-port-request } pr), (log, pol)) &= \\ (\text{if before}(\text{Not } o \text{ is-ftp-close } i) (\text{is-init } i) \log \wedge \\ \text{dest-port } (i, s, d, \text{ftp-port-request } pr) &= (21 :: \text{port})) \end{aligned}$$

*then Some (((i,s,d,ftp-port-request pr) # log,
 $(allow\text{-}from\text{-}to\text{-}port pr (subnet\text{-}of d) (subnet\text{-}of s)) \oplus pol)$)
*else Some (((i,s,d,ftp-port-request pr) # log, pol)))**

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

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

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

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 $\text{ftp-states} = S0 \mid S1 \mid S2 \mid S3$

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

fun

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

where

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

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

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

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

$\text{ftp-2-interleaved} :: \text{adr}_{ip} \text{ src} \Rightarrow \text{adr}_{ip} \text{ dest} \Rightarrow \text{id} \Rightarrow \text{port} \Rightarrow \text{adr}_{ip} \text{ src} \Rightarrow \text{adr}_{ip} \text{ dest} \Rightarrow \text{id} \Rightarrow \text{port} \Rightarrow (\text{adr}_{ip}, \text{msg}) \text{ history set}$ **where**
 $\text{ftp-2-interleaved } s1 d1 i1 p1 s2 d2 i2 p2 = \{x. (\text{is-ftp } S0 s1 d1 i1 p1 (\text{packet-with-id } x i1)) \wedge (\text{is-ftp } S0 s2 d2 i2 p2 (\text{packet-with-id } x i2))\}$

lemma $\text{subnetOf-lemma}: (a::int) \neq (c::int) \implies \forall x \in \text{subnet-of } (a, b::port). (c, d) \notin x$
by (rule *ballI*, simp add: *subnet-of-int-def*)

lemma $\text{subnetOf-lemma2}: \forall x \in \text{subnet-of } (a::int, b::port). (a, b) \in x$
by (rule *ballI*, simp add: *subnet-of-int-def*)

lemma $\text{subnetOf-lemma3}: (\exists x. x \in \text{subnet-of } (a::int, b::port))$
by (rule *exI*, simp add: *subnet-of-int-def*)

```

lemma subnetOf-lemma4:  $\exists x \in \text{subnet-of } (a::int, b::port). (a, c::port) \in x$ 
  by (rule bexI, simp-all add: subnet-of-int-def)

lemma port-open-lemma:  $\neg (\exists x \text{ (port-open } [] (x::port)))$ 
  by (simp add: port-open-def)

lemmas FTPLemmas = TRPolicy-def applyPolicy-def policy2MON-def
  Let-def in-subnet-def src-def
  dest-def subnet-of-int-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

```

2.4.3 FTP enriched with a security policy

theory

FTP-WithPolicy

imports

FTP

begin

FTP where the policy is part of the output.

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

Variant 2 takes the policy into the output

fun FTP-STP ::

$((id \rightarrow port), adr_{ip}, msg) FWStateTransitionP$
where

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

$|FTP-STP (i,s,d,ftp-close) (ports,policy) =$
 $(if (p-accept (i,s,d,ftp-close) policy) then$

```

case ports i of
  Some pr =>
    Some(allow (POL (deny-from-to-port pr (subnet-of d) (subnet-of s)  $\oplus$  policy)),
      ports(i:=None),
      deny-from-to-port pr (subnet-of d) (subnet-of s)  $\oplus$  policy)
  | None => 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

```

2.4.4 A simple voice-over-ip model

```

theory VOIP
  imports StatefulCore
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. H.323 poses many problems to firewalls. These problems include:

- 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:

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

- Setup *port*
- Connect *port*
- Stream (dynamic ports). The call itself. In reality, several connections are used here.
- Fin (port 1720).

The two main differences to FTP are:

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

We model the protocol as seen from a firewall at the caller, namely we are not interested in the messages from the callee to its gatekeeper. Incoming calls are not modelled either, they would require a different set of state transitions.

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

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

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

definition

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

definition

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

definition

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

definition

```

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

```

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

definition

```

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

```

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

fun

```

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

```

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

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

```

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)) ⊕ policy))

```

```

| VOIP-STA ((a,c,d,ARJ), (InL, policy)) =
    (if (not-before (is-fin a) (is-arg a) InL)
        then Some (((a,c,d,ARJ)#InL,
                    deny-from-to-port (14::port) (subnet-of c) (subnet-of d) ⊕ 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) ⊕
           allow-from-to-port (1720::port) (subnet-of d) (subnet-of-adr callee) ⊕
           deny-from-to-port (1719::port) (subnet-of d) (subnet-of c) ⊕

```

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**

$VOIP-TRPolicy = policy2MON ($
 $((VOIP-STA, VOIP-STD) \otimes_{\nabla} applyPolicy) o (\lambda (x, (y, z)). ((x, z), (x, (y, z))))$

For a full protocol run, six states are needed.

datatype $voip-states = S0 \mid S1 \mid S2 \mid S3 \mid S4 \mid S5$

The constant $is\text{-}voip$ 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  $\Rightarrow$  address  $\Rightarrow$  address  $\Rightarrow$  address  $\Rightarrow$  id  $\Rightarrow$  port  $\Rightarrow$ 
    port  $\Rightarrow$  (adrip, address voip-msg) history  $\Rightarrow$  bool
where
  is-voip H s d g i p1 p2 [] = (H = S5)
  | is-voip H s d g i p1 p2 (x#InL) =
    (((λ (id, sr, de, co).
      (((id = i  $\wedge$ 
        (H = S4  $\wedge$  ((sr = (s, 1719)  $\wedge$  de = (g, 1719)  $\wedge$  co = ARQ  $\wedge$ 
          is-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-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-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-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-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-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-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-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-voip S1 s d g i p1 p2 InL)))))) x)
```

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

end

2.4.5 FTP and VoIP Protocol

```

theory
  FTPVOIP
imports
  FTP-WithPolicy VOIP
begin

datatype ftpvoip = ARQ
  | ACF int
  | ARJ
  | Setup port
  | Connect port
  | Stream
  | Fin
  | ftp-init
  | ftp-port-request port
  | ftp-data
  | ftp-close
  | other

```

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

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

definition

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

definition

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

definition

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

definition

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

definition

FTPVOIP-port-open :: $(adr_{ip}, \text{ ftpvoip }) \text{ history} \Rightarrow id \Rightarrow port \Rightarrow \text{bool}$ **where**

$$\begin{aligned} \text{FTPVOIP-port-open} &= (\lambda L a p. (not-before (\text{FTPVOIP-is-close } a) \\ &(\text{FTPVOIP-is-port-request } a p)) L)) \end{aligned}$$

definition

FTPVOIP-is-other :: id \Rightarrow (adr_{ip}, ftpvoip) 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 \wedge
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 ($\exists p.$ FTPVOIP-port-open InL i p) \wedge 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

$$\begin{aligned} \text{FTPVOIP-is-arq} :: \text{NetworkCore.id} \Rightarrow ('a::\text{adr}, \text{ftpvoip}) \text{ packet} \Rightarrow \text{bool where} \\ \text{FTPVOIP-is-arq } i \text{ } p = (\text{NetworkCore.id } p = i \wedge \text{content } p = \text{ARQ}) \end{aligned}$$
definition

$$\begin{aligned} \text{FTPVOIP-is-fin} :: \text{id} \Rightarrow ('a::\text{adr}, \text{ftpvoip}) \text{ packet} \Rightarrow \text{bool where} \\ \text{FTPVOIP-is-fin } i \text{ } p = (\text{id } p = i \wedge \text{content } p = \text{Fin}) \end{aligned}$$
definition

$$\begin{aligned} \text{FTPVOIP-is-connect} :: \text{id} \Rightarrow \text{port} \Rightarrow ('a::\text{adr}, \text{ftpvoip}) \text{ packet} \Rightarrow \text{bool where} \\ \text{FTPVOIP-is-connect } i \text{ } \text{port } p = (\text{id } p = i \wedge \text{content } p = \text{Connect port}) \end{aligned}$$
definition

$$\begin{aligned} \text{FTPVOIP-is-setup} :: \text{id} \Rightarrow \text{port} \Rightarrow ('a::\text{adr}, \text{ftpvoip}) \text{ packet} \Rightarrow \text{bool where} \\ \text{FTPVOIP-is-setup } i \text{ } \text{port } p = (\text{id } p = i \wedge \text{content } p = \text{Setup port}) \end{aligned}$$

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

definition

$$\begin{aligned} \text{FTPVOIP-ports-open} :: \text{id} \Rightarrow \text{port} \times \text{port} \Rightarrow (\text{adr}_{ip}, \text{ftpvoip}) \text{ history} \Rightarrow \text{bool where} \\ \text{FTPVOIP-ports-open } i \text{ } p \text{ } L = ((\text{not-before } (\text{FTPVOIP-is-fin } i)) \text{ } (\text{FTPVOIP-is-setup } i \text{ } (\text{fst } p)) \text{ } L) \wedge \\ \text{not-before } (\text{FTPVOIP-is-fin } i) \text{ } (\text{FTPVOIP-is-connect } i \text{ } (\text{snd } p)) \\ L \end{aligned}$$

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

fun

$$\begin{aligned} \text{FTPVOIP-src-is-initiator} :: \text{id} \Rightarrow \text{adr}_{ip} \Rightarrow (\text{adr}_{ip}, \text{ftpvoip}) \text{ history} \Rightarrow \text{bool where} \\ \text{FTPVOIP-src-is-initiator } i \text{ } a \text{ } [] = \text{False} \\ |\text{FTPVOIP-src-is-initiator } i \text{ } a \text{ } (p \# S) = (((\text{id } p = i) \wedge \\ (\exists \text{ port. content } p = \text{Setup port}) \wedge \\ ((\text{fst } (\text{src } p) = \text{fst } a))) \vee \\ (\text{FTPVOIP-src-is-initiator } i \text{ } a \text{ } S)) \end{aligned}$$
definition $\text{FTPVOIP-subnet-of-addr} :: \text{int} \Rightarrow \text{adr}_{ip} \text{ net where}$

$$\text{FTPVOIP-subnet-of-addr } x = \{\{(a, b). a = x\}\}$$
fun $\text{FTPVOIP-VOIP-STA} ::$

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

where

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

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

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

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

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

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

$| \ FTPVOIP\text{-}VOIP\text{-}STA (p, (InL, policy)) =$

```

Some ((p#InL,policy))

fun FTPVOIP-VOIP-STD ::  

((adrip, ftpvoip) history, adrip, ftpvoip) FWStateTransition  

where  

FTPVOIP-VOIP-STD (p,s) = Some s

definition FTP-VOIP-STA :: ((adrip, ftpvoip) history, adrip, ftpvoip) FWStateTransition  

where  

FTP-VOIP-STA = ((λ(x,x). Some x) om ((FTPVOIP-FTP-STA ⊗S FTPVOIP-VOIP-STA o (λ(p,x). (p,x,x)))))

definition FTP-VOIP-STD :: ((adrip, ftpvoip) history, adrip, ftpvoip) FWStateTransition  

where  

FTP-VOIP-STD = (λ(x,x). Some x) om ((FTPVOIP-FTP-STD ⊗S FTPVOIP-VOIP-STD o (λ(p,x). (p,x,x)))))

definition FTPVOIP-TRPolicy where  

FTPVOIP-TRPolicy = policy2MON (  

(((FTP-VOIP-STA,FTP-VOIP-STD) ⊗▽ applyPolicy) o (λ(x,(y,z)).  

((x,z),(x,(y,z))))))

lemmas FTPVOIP-ST-simps = Let-def in-subnet-def src-def dest-def  

subnet-of-int-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 adripLemmas

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 ⇒ address ⇒ address ⇒ address ⇒ id ⇒ port ⇒  

port ⇒ (adrip, ftpvoip) history ⇒ bool
where  

FTPVOIP-is-voip H s d g i p1 p2 [] = (H = V5)  

| FTPVOIP-is-voip H s d g i p1 p2 (x#InL) =  

(((λ(id,sr,de,co).  

(((id = i ∧  

(H = V4 ∧ ((sr = (s,1719) ∧ de = (g,1719) ∧ co = ARQ ∧
```

$$\begin{aligned}
& \text{FTPVOIP-is-voip } V5 s d g i p1 p2 InL))) \vee \\
& (H = V0 \wedge sr = (g, 1719) \wedge de = (s, 1719) \wedge co = ARJ \wedge \\
& \quad \text{FTPVOIP-is-voip } V4 s d g i p1 p2 InL) \vee \\
& (H = V3 \wedge sr = (g, 1719) \wedge de = (s, 1719) \wedge co = ACF d \wedge \\
& \quad \text{FTPVOIP-is-voip } V4 s d g i p1 p2 InL) \vee \\
& (H = V2 \wedge sr = (s, 1720) \wedge de = (d, 1720) \wedge co = Setup p1 \wedge \\
& \quad \text{FTPVOIP-is-voip } V3 s d g i p1 p2 InL) \vee \\
& (H = V1 \wedge sr = (d, 1720) \wedge de = (s, 1720) \wedge co = Connect p2 \wedge \\
& \quad \text{FTPVOIP-is-voip } V2 s d g i p1 p2 InL) \vee \\
& (H = V1 \wedge sr = (s, p1) \wedge de = (d, p2) \wedge co = Stream \wedge \\
& \quad \text{FTPVOIP-is-voip } V1 s d g i p1 p2 InL) \vee \\
& (H = V1 \wedge sr = (d, p2) \wedge de = (s, p1) \wedge co = Stream \wedge \\
& \quad \text{FTPVOIP-is-voip } V1 s d g i p1 p2 InL) \vee \\
& (H = V0 \wedge sr = (d, 1720) \wedge de = (s, 1720) \wedge co = Fin \wedge \\
& \quad \text{FTPVOIP-is-voip } V1 s d g i p1 p2 InL) \vee \\
& (H = V0 \wedge sr = (s, 1720) \wedge de = (d, 1720) \wedge co = Fin \wedge \\
& \quad \text{FTPVOIP-is-voip } V1 s d g i p1 p2 InL)))))) x)
\end{aligned}$$

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

definition

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

fun

$$\begin{aligned}
& \text{FTPVOIP-is-ftp} :: \text{ftp-states2} \Rightarrow adr_{ip} \Rightarrow adr_{ip} \Rightarrow id \Rightarrow port \Rightarrow \\
& \quad (adr_{ip}, \text{ftpvoip}) \text{ history} \Rightarrow \text{bool}
\end{aligned}$$

where

$$\begin{aligned}
& \text{FTPVOIP-is-ftp } H c s i p [] = (H=FS3) \\
| & \text{FTPVOIP-is-ftp } H c s i p (x \# InL) = (snd s = 21 \wedge ((\lambda (id, sr, de, co). (((id = i \wedge (\\
& \quad (H=FS2 \wedge sr = c \wedge de = s \wedge co = \text{ftp-init} \wedge \text{FTPVOIP-is-ftp FS3 } c s i p InL) \vee \\
& \quad (H=FS1 \wedge sr = c \wedge de = s \wedge co = \text{ftp-port-request } p \wedge \text{FTPVOIP-is-ftp FS2 } c s i \\
& \quad p InL) \vee \\
& \quad (H=FS1 \wedge sr = s \wedge de = (fst c, p) \wedge co = \text{ftp-data} \wedge \text{FTPVOIP-is-ftp FS1 } c s i p \\
& \quad InL) \vee \\
& \quad (H=FS0 \wedge sr = c \wedge de = s \wedge co = \text{ftp-close} \wedge \text{FTPVOIP-is-ftp FS1 } c s i p InL) \\
& \quad))))) x))
\end{aligned}$$

definition

$$\begin{aligned}
& \text{FTPVOIP-NB-ftp} :: adr_{ip} \text{ src} \Rightarrow adr_{ip} \text{ dest} \Rightarrow id \Rightarrow port \Rightarrow (adr_{ip}, \text{ftpvoip}) \text{ history} \\
& \text{set where}
\end{aligned}$$

$$\text{FTPVOIP-NB-ftp } s d i p = \{x. (\text{FTPVOIP-is-ftp FS0 } s d i p x)\}$$

definition

ftp-voip-interleaved :: $adr_{ip} \ src \Rightarrow adr_{ip} \ dest \Rightarrow id \Rightarrow port \Rightarrow address \Rightarrow address \Rightarrow address \Rightarrow id \Rightarrow port \Rightarrow port \Rightarrow (adr_{ip}, \ ftpvoip) \ history \ set$

where

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

end

3 Examples

```
theory
  Examples
  imports
    DMZ / DMZ
    Voice-over-IP / Voice-over-IP
    Transformation / Transformation
    NAT-FW / NAT-FW
    PersonalFirewall / PersonalFirewall
begin
end
```

3.1 A Simple DMZ Setup

```
theory
  DMZ
  imports
    DMZDatatype
    DMZInteger
begin
end
```

3.1.1 DMZ Datatype

```
theory
  DMZDatatype
  imports
    ../../UPF-Firewall
begin
```

This is the fourth scenario, slightly more complicated than the previous one, as we now also model specific servers within one network. Therefore, we could not use anymore the modelling using datatype synonym, but only use the one where an address is modelled as an integer (with ports).

Just for comparison, this theory is the same scenario with datatype synonym anyway, but with four distinct networks instead of one contained in another. As there is no corresponding network model included, we need to define a custom one.

```

datatype Adr = Intranet | Internet | Mail | Web | DMZ
instance Adr::adr ..
type-synonym port = int
type-synonym Networks = Adr × port

```

definition

```

intranet::Networks net where
intranet = {{(a,b). a= Intranet}}

```

definition

```

dmz :: Networks net where
dmz = {{(a,b). a= DMZ}}

```

definition

```

mail :: Networks net where
mail = {{(a,b). a=Mail}}

```

definition

```

web :: Networks net where
web = {{(a,b). a=Web}}

```

definition

```

internet :: Networks net where
internet = {{(a,b). a= Internet}}

```

definition

```

Intranet-mail-port :: (Networks ,DummyContent) FWPolicy where
Intranet-mail-port = (allow-from-ports-to {21::port,14} intranet mail)

```

definition

```

Intranet-Internet-port :: (Networks,DummyContent) FWPolicy where
Intranet-Internet-port = allow-from-ports-to {80::port,90} intranet internet

```

definition

```

Internet-web-port :: (Networks,DummyContent) FWPolicy where
Internet-web-port = (allow-from-ports-to {80::port,90} internet web)

```

definition

```

Internet-mail-port :: (Networks,DummyContent) FWPolicy where
Internet-mail-port = (allow-all-from-port-to internet (21::port) dmz)

```

definition

```

policyPort :: (Networks, DummyContent) FWPolicy where
policyPort = deny-all ++
    Intranet-Internet-port ++
    Intranet-mail-port ++
    Internet-mail-port ++
    Internet-web-port

```

We only want to create test cases which are sent between the three main networks: e.g. not between the mailserver and the dmz. Therefore, the constraint looks as follows.

%

definition

```
not-in-same-net :: (Networks,DummyContent) packet ⇒ bool where
not-in-same-net x = ((src x ⊑ internet → ¬ dest x ⊑ internet) ∧
                      (src x ⊑ intranet → ¬ dest x ⊑ intranet) ∧
                      (src x ⊑ dmz → ¬ dest x ⊑ dmz))
```

```
lemmas PolicyLemmas = dmz-def internet-def intranet-def mail-def web-def
Internet-web-port-def Internet-mail-port-def
Intranet-Internet-port-def Intranet-mail-port-def
src-def dest-def src-port dest-port in-subnet-def
```

end

3.1.2 DMZ: Integer

theory

DMZInteger

imports

..../UPF-Firewall

begin

This scenario is slightly more complicated than the SimpleDMZ one, as we now also model specific servers within one network. Therefore, we cannot use anymore the modelling using datatype synonym, but only use the one where an address is modelled as an integer (with ports).

The scenario is the following:

- | | |
|-----------|---|
| Networks: | <ul style="list-style-type: none"> • Intranet (Company intern network) • DMZ (demilitarised zone, servers, etc), containing at least two distinct servers “mail” and “web” • Internet (“all others”) |
| Policy: | <ul style="list-style-type: none"> • allow http(s) from Intranet to Internet • deny all traffic from Internet to Intranet • allow imaps and smtp from intranet to mailserver • allow smtp from Internet to mailserver • allow http(s) from Internet to webserver • deny everything else |

definition

intranet::*adr_ip net where*
intranet = {{(a,b) . (a > 1 \wedge a < 4)}}

definition

dmz :: *adr_ip net where*
dmz = {{(a,b). (a > 6) \wedge (a < 11)}}

definition

mail :: *adr_ip net where*
mail = {{(a,b). a = 7}}

definition

web :: *adr_ip net where*
web = {{(a,b). a = 8}}

definition

internet :: *adr_ip net where*
internet = {{(a,b). \neg ((a > 1 \wedge a < 4) \vee (a > 6) \wedge (a < 11))}}

definition

Intranet-mail-port :: (*adr_ip,'b*) *FWPolicy where*
Intranet-mail-port = (allow-from-to-ports {21::port,14} *intranet mail*)

definition

Intranet-Internet-port :: (*adr_ip,'b*) *FWPolicy where*
Intranet-Internet-port = allow-from-to-ports {80::port,90} *intranet internet*

definition

Internet-web-port :: (*adr_ip,'b*) *FWPolicy where*
Internet-web-port = (allow-from-to-ports {80::port,90} *internet web*)

definition

Internet-mail-port :: (*adr_ip,'b*) *FWPolicy where*
Internet-mail-port = (allow-all-from-port-to *internet* (21::port) *dmz*)

definition

policyPort :: (*adr_ip, DummyContent*) *FWPolicy where*
policyPort = deny-all ++
 Intranet-Internet-port ++
 Intranet-mail-port ++
 Internet-mail-port ++
 Internet-web-port

We only want to create test cases which are sent between the three main networks:
e.g. not between the mailserver and the dmz. Therefore, the constraint looks as follows.

definition

not-in-same-net :: (*adr_ip,DummyContent*) *packet* \Rightarrow *bool where*

```

not-in-same-net x = ((src x ⊑ internet → ¬ dest x ⊑ internet) ∧
                      (src x ⊑ intranet → ¬ dest x ⊑ intranet) ∧
                      (src x ⊑ dmz → ¬ dest x ⊑ dmz))

lemmas PolicyLemmas = policyPort-def dmz-def internet-def intranet-def mail-def
web-def
Intranet-Internet-port-def Intranet-mail-port-def Internet-web-port-def
Internet-mail-port-def src-def dest-def IntegerPort.src-port
in-subnet-def IntegerPort.dest-port

end

```

3.2 Personal Firewall

```

theory
PersonalFirewall
imports
PersonalFirewallInt
PersonalFirewallIpv4
PersonalFirewallDatatype
begin
end

```

3.2.1 Personal Firewall: Integer

```

theory
PersonalFirewallInt
imports
..../UPF-Firewall
begin

```

The most basic firewall scenario; there is a personal PC on one side and the Internet on the other. There are two policies: the first one allows all traffic from the PC to the Internet and denies all coming into the PC. The second policy only allows specific ports from the PC. This scenario comes in three variants: the first one specifies the allowed protocols directly, the second together with their respective port numbers, the third one only with the port numbers.

Definitions of the subnets

```

definition
PC :: (adr_ip net) where
PC = {{(a,b). a = 3}}

```

definition

Internet :: adr_{ip} net **where**
 $Internet = \{(a,b). \neg (a = 3)\}$

definition

not-in-same-net :: $(adr_{ip}, DummyContent)$ packet \Rightarrow bool **where**
 $not-in-same-net x = ((src x \sqsubset PC \longrightarrow dest x \sqsubset Internet) \wedge (src x \sqsubset Internet \longrightarrow dest x \sqsubset PC))$

Definitions of the policies

definition

strictPolicy :: $(adr_{ip}, DummyContent)$ FWPolicy **where**
 $strictPolicy = deny-all ++ allow-all-from-to PC Internet$

definition

PortPolicy :: $(adr_{ip}, DummyContent)$ FWPolicy **where**
 $PortPolicy = deny-all ++ allow-from-ports-to \{http, smtp, ftp\} PC Internet$

definition

PortPolicyBig :: $(adr_{ip}, DummyContent)$ FWPolicy **where**
 $PortPolicyBig = deny-all ++$
 $allow-from-port-to http PC Internet ++$
 $allow-from-port-to smtp PC Internet ++$
 $allow-from-port-to ftp PC Internet$

lemmas $policyLemmas = strictPolicy\text{-def } PortPolicy\text{-def } PC\text{-def}$
 $Internet\text{-def } PortPolicyBig\text{-def } src\text{-def } dest\text{-def}$
 $adr_{ip}\text{Lemmas content\text{-def}}$
 $PortCombinators\text{ in\text{-}subnet\text{-}def } PortPolicyBig\text{-def id\text{-}def}$

declare Ports [simp add]

definition *wellformed-packet*:: $(adr_{ip}, DummyContent)$ packet \Rightarrow bool **where**
 $wellformed-packet p = (content p = data)$

end

3.2.2 Personal Firewall IPv4

theory

PersonalFirewallIpv4

imports

$..//UPF-Firewall$

begin

The most basic firewall scenario; there is a personal PC on one side and the Internet

on the other. There are two policies: the first one allows all traffic from the PC to the Internet and denies all coming into the PC. The second policy only allows specific ports from the PC. This scenario comes in three variants: the first one specifies the allowed protocols directly, the second together with their respective port numbers, the third one only with the port numbers.

Definitions of the subnets

definition

$PC :: (ipv4\ net) \text{ where}$

$PC = \{(a,b,c,d). a = 1 \wedge b = 3 \wedge c = 5 \wedge d = 2\}\}$

definition

$Internet :: ipv4\ net \text{ where}$

$Internet = \{(a,b,c,d). \neg(a = 1 \wedge b = 3 \wedge c = 5 \wedge d = 2)\}\}$

definition

$not-in-same-net :: (ipv4, DummyContent) \text{ packet} \Rightarrow \text{bool where}$

$not-in-same-net x = ((src\ x \sqsubset PC \rightarrow dest\ x \sqsubset Internet) \wedge (src\ x \sqsubset Internet \rightarrow dest\ x \sqsubset PC))$

Definitions of the policies

definition

$strictPolicy :: (ipv4, DummyContent) \text{ FWPolicy where}$

$strictPolicy = \text{deny-all} ++ \text{allow-all-from-to } PC\ Internet$

definition

$PortPolicy :: (ipv4, DummyContent) \text{ FWPolicy where}$

$PortPolicy = \text{deny-all} ++ \text{allow-from-ports-to } \{80::port, 24, 21\} \text{ PC Internet}$

definition

$PortPolicyBig :: (ipv4, DummyContent) \text{ FWPolicy where}$

$PortPolicyBig = \text{deny-all} ++ \text{allow-from-port-to } (80::port) \text{ PC Internet} ++ \text{allow-from-port-to } (24::port) \text{ PC Internet} ++ \text{allow-from-port-to } (21::port) \text{ PC Internet}$

lemmas $policyLemmas = strictPolicy\text{-def } PortPolicy\text{-def } PC\text{-def}$

$\text{Internet-def } PortPolicyBig\text{-def } src\text{-def } dest\text{-def}$

$IPv4.\text{src-port}$

$IPv4.\text{dest-port } PolicyCombinators$

$PortCombinators \text{ in-subnet-def } PortPolicyBig\text{-def}$

end

3.2.3 Personal Firewall: Datatype

theory

```

PersonalFirewallDatatype
imports
  ../../UPF-Firewall
begin

```

The most basic firewall scenario; there is a personal PC on one side and the Internet on the other. There are two policies: the first one allows all traffic from the PC to the Internet and denies all coming into the PC. The second policy only allows specific ports from the PC. This scenario comes in three variants: the first one specifies the allowed protocols directly, the second together with their respective port numbers, the third one only with the port numbers.

```
datatype Adr = pc | internet
```

```
type-synonym DatatypeTwoNets = Adr × int
```

```
instance Adr::adr ..
```

definition

```
PC :: DatatypeTwoNets net where
PC = {{(a,b). a = pc}}
```

definition

```
Internet :: DatatypeTwoNets net where
Internet = {{(a,b). a = internet}}
```

definition

```
not-in-same-net :: (DatatypeTwoNets,DummyContent) packet ⇒ bool where
not-in-same-net x = ((src x ⊑ PC → dest x ⊑ Internet) ∧ (src x ⊑ Internet →
dest x ⊑ PC))
```

Definitions of the policies

In fact, the short definitions wouldn't have to be written down - they are the automatically simplified versions of their big counterparts.

definition

```
strictPolicy :: (DatatypeTwoNets,DummyContent) FWPolicy where
strictPolicy = deny-all ++ allow-all-from-to PC Internet
```

definition

```
PortPolicy :: (DatatypeTwoNets,'b) FWPolicy where
PortPolicy = deny-all ++ allow-from-ports-to {80::port,24,21} PC Internet
```

definition

```
PortPolicyBig :: (DatatypeTwoNets,'b) FWPolicy where
PortPolicyBig =
```

```

allow-from-port-to (80::port) PC Internet ⊕
allow-from-port-to (24::port) PC Internet ⊕
allow-from-port-to (21::port) PC Internet ⊕
deny-all

lemmas policyLemmas = strictPolicy-def PortPolicy-def PC-def Internet-def PortPolicyBig-def src-def
                           PolicyCombinators PortCombinators in-subnet-def

end

```

3.3 Demonstrating Policy Transformations

```

theory
  Transformation
imports
  Transformation01
  Transformation02
begin
end

```

3.3.1 Transformation Example 1

```

theory
  Transformation01
imports
  ../../UPF-Firewall
begin

definition
  FWLink :: adr_ip net where
  FWLink = {{(a,b). a = 1}}

```

```

definition
  any :: adr_ip net where
  any = {{(a,b). a > 5}}

```

```

definition
  i4:: adr_ip net where
  i4 = {{(a,b). a = 2 }}

```

```

definition
  i27:: adr_ip net where
  i27 = {{(a,b). a = 3 }}

```

definition

eth-intern:: adr_{ip} net where
eth-intern = {{(a,b). a = 4 }}

definition

eth-private:: adr_{ip} net where
eth-private = {{(a,b). a = 5 }}

definition

MG2 :: (adr_{ip} net,port) Combinators where
MG2 = AllowPortFromTo i27 any 1 ⊕
AllowPortFromTo i27 any 2 ⊕
AllowPortFromTo i27 any 3

definition

MG3 :: (adr_{ip} net,port) Combinators where
MG3 = AllowPortFromTo any FWLink 1

definition

MG4 :: (adr_{ip} net,port) Combinators where
MG4 = AllowPortFromTo FWLink FWLink 4

definition

MG7 :: (adr_{ip} net,port) Combinators where
MG7 = AllowPortFromTo FWLink i4 6 ⊕
AllowPortFromTo FWLink i4 7

definition

MG8 :: (adr_{ip} net,port) Combinators where
MG8 = AllowPortFromTo FWLink i4 6 ⊕
AllowPortFromTo FWLink i4 7

definition

DG3:: (adr_{ip} net,port) Combinators where
DG3 = AllowPortFromTo any any 7

definition

Policy = DenyAll ⊕ MG8 ⊕ MG7 ⊕ MG4 ⊕ MG3 ⊕ MG2 ⊕ DG3

lemmas *PolicyLemmas = Policy-def*
FWLink-def

```

any-def
i27-def
i4-def
eth-intern-def
eth-private-def
MG2-def MG3-def MG4-def MG7-def MG8-def
DG3-def

```

lemmas *PolicyL* = MG2-def MG3-def MG4-def MG7-def MG8-def DG3-def *Policy-def*

definition

```

not-in-same-net :: (adr_ip,DummyContent) packet ⇒ bool where
not-in-same-net x = (((src x ⊑ i27) → (¬(dest x ⊑ i27))) ∧
                      ((src x ⊑ i4) → (¬(dest x ⊑ i4))) ∧
                      ((src x ⊑ eth-intern) → (¬(dest x ⊑ eth-intern))) ∧
                      ((src x ⊑ eth-private) → (¬(dest x ⊑ eth-private))))

```

consts *fixID* :: *id*

consts *fixContent* :: *DummyContent*

definition *fixElements p* = (*id p* = *fixID* ∧ *content p* = *fixContent*)

lemmas *fixDefs* = *fixElements-def NetworkCore.id-def NetworkCore.content-def*

lemma *sets-distinct1*: (*n::int*) ≠ *m* ⇒ {(*a,b*). *a* = *n*} ≠ {(*a,b*). *a* = *m*}

by auto

lemma *sets-distinct2*: (*m::int*) ≠ *n* ⇒ {(*a,b*). *a* = *n*} ≠ {(*a,b*). *a* = *m*}

by auto

lemma *sets-distinct3*: {((*a::int*),(*b::int*)). *a* = *n*} ≠ {(*a,b*). *a* > *n*}

by auto

lemma *sets-distinct4*: {((*a::int*),(*b::int*)). *a* > *n*} ≠ {(*a,b*). *a* = *n*}

by auto

lemma *aux*: [a ∈ *c*; *a* ∉ *d*; *c* = *d*] ⇒ False

by auto

lemma *sets-distinct5*: (*s::int*) < *g* ⇒ {(*a::int, b::int*). *a* = *s*} ≠ {(*a::int, b::int*). *g* < *a*}

apply (auto simp: *sets-distinct3*)[1]

```

apply (subgoal-tac (s,4) ∈ {(a::int,b::int). a = (s)})
apply (subgoal-tac (s,4) ∉ {(a::int,b::int). g < a})
apply (erule aux)
apply assumption+
apply simp
by blast

lemma sets-distinct6: (s::int) < g ==> {(a::int, b::int). g < a} ≠ {(a::int, b::int). a = s}
apply (rule not-sym)
apply (rule sets-distinct5)
by simp

lemma distinctNets: FWLink ≠ any ∧ FWLink ≠ i4 ∧ FWLink ≠ i27 ∧ FWLink ≠ eth-intern ∧ FWLink ≠ eth-private ∧ any ≠ FWLink ∧ any ≠ i4 ∧ any ≠ i27 ∧ any ≠ eth-intern ∧ any ≠ eth-private ∧ i4 ≠ FWLink ∧ i4 ≠ any ∧ i4 ≠ i27 ∧ i4 ≠ eth-intern ∧ i4 ≠ eth-private ∧ i27 ≠ FWLink ∧ i27 ≠ any ∧ i27 ≠ eth-intern ∧ i27 ≠ eth-private ∧ eth-intern ≠ FWLink ∧ eth-intern ≠ any ∧ eth-intern ≠ i4 ∧ eth-intern ≠ i27 ∧ eth-intern ≠ eth-private ∧ eth-private ≠ FWLink ∧ eth-private ≠ any ∧ eth-private ≠ i4 ∧ eth-private ≠ i27 ∧ eth-private ≠ eth-intern
by (simp add: PolicyLemmas sets-distinct1 sets-distinct2 sets-distinct3 sets-distinct4 sets-distinct5 sets-distinct6)

lemma aux5: [|x ≠ a; y≠b; (x ≠ y ∧ x ≠ b) ∨ (a ≠ b ∧ a ≠ y)|] ==> {x,a} ≠ {y,b}
by auto

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

lemma ANDex: allNetsDistinct (policy2list Policy)
apply (simp add: PolicyL allNetsDistinct-def distinctNets)
by (auto simp: PLemmas PolicyLemmas netsDistinct-def sets-distinct5 sets-distinct6)

fun (sequential) numberOfRowsInSection where
  numberOfRowsInSection (a⊕b) = numberOfRowsInSection a + numberOfRowsInSection b
  |numberOfRules a = (1::int)

fun numberOfRowsInSectionList where
  numberOfRowsInSectionList (x#xs) = ((numberOfRules x) #(numberOfRulesList xs))
  |numberOfRulesList [] = []

```

```

lemma all-in-list: all-in-list (policy2list Policy) (Nets-List Policy)
  apply (simp add: PolicyL)
  apply (unfold Nets-List-def)
  apply (unfold bothNets-def)
  apply (insert distinctNets)
  by simp

lemmas normalizeUnfold =  normalize-def Policy-def Nets-List-def bothNets-def aux
aux2 bothNets-def

end

```

3.3.2 Transformation Example 2

```

theory
  Transformation02
imports
  ../../UPF-Firewall
begin

definition
  FWLink :: adr_ip net where
  FWLink = {{(a,b). a = 1} }

definition
  any :: adr_ip net where
  any = {{(a,b). a > 5} }

definition
  i4-32:: adr_ip net where
  i4-32 = {{(a,b). a = 2 } }

definition
  i10-32:: adr_ip net where
  i10-32 = {{(a,b). a = 3 } }

definition
  eth-intern:: adr_ip net where
  eth-intern = {{(a,b). a = 4 } }

definition
  eth-private:: adr_ip net where
  eth-private = {{(a,b). a = 5 } }

```

definition

$D1a :: (adr_{ip} \ net, \ port) \ Combinators \ where$
 $D1a = AllowPortFromTo \ eth-intern \ any \ 1 \oplus$
 $\quad AllowPortFromTo \ eth-intern \ any \ 2$

definition

$D1b :: (adr_{ip} \ net, \ port) \ Combinators \ where$
 $D1b = AllowPortFromTo \ eth-private \ any \ 1 \oplus$
 $\quad AllowPortFromTo \ eth-private \ any \ 2$

definition

$D2a :: (adr_{ip} \ net, \ port) \ Combinators \ where$
 $D2a = AllowPortFromTo \ any \ i4\text{-}32 \ 21$

definition

$D2b :: (adr_{ip} \ net, \ port) \ Combinators \ where$
 $D2b = AllowPortFromTo \ any \ i10\text{-}32 \ 21 \oplus$
 $\quad AllowPortFromTo \ any \ i10\text{-}32 \ 43$

definition

$Policy :: (adr_{ip} \ net, \ port) \ Combinators \ where$
 $Policy = DenyAll \oplus D2b \oplus D2a \oplus D1b \oplus D1a$

lemmas $PolicyLemmas = Policy\text{-}def \ D1a\text{-}def \ D1b\text{-}def \ D2a\text{-}def \ D2b\text{-}def$

lemmas $PolicyL = \begin{aligned} & Policy\text{-}def \\ & FWLink\text{-}def \\ & any\text{-}def \\ & i10\text{-}32\text{-}def \\ & i4\text{-}32\text{-}def \\ & eth\text{-}intern\text{-}def \\ & eth\text{-}private\text{-}def \\ & D1a\text{-}def \ D1b\text{-}def \ D2a\text{-}def \ D2b\text{-}def \end{aligned}$

consts $fixID :: id$
consts $fixContent :: DummyContent$

definition $fixElements p = (id \ p = fixID \wedge content \ p = fixContent)$

lemmas $fixDefs = fixElements\text{-}def \ NetworkCore.id\text{-}def \ NetworkCore.content\text{-}def$

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

```

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

lemma sets-distinct3:  $\{((a::int), (b::int)). a = n\} \neq \{(a,b). a > n\}$ 
  by auto

lemma sets-distinct4:  $\{((a::int), (b::int)). a > n\} \neq \{(a,b). a = n\}$ 
  by auto

lemma aux:  $\llbracket a \in c; a \notin d; c = d \rrbracket \implies False$ 
  by auto

lemma sets-distinct5:  $(s::int) < g \implies \{(a::int, b::int). a = s\} \neq \{(a::int, b::int). g < a\}$ 
  apply (auto simp: sets-distinct3)
  apply (subgoal-tac (s,4)  $\in \{(a::int, b::int). a = (s)\}$ )
  apply (subgoal-tac (s,4)  $\notin \{(a::int, b::int). g < a\}$ )
  apply (erule aux)
  apply assumption+
  apply simp
  by blast

lemma sets-distinct6:  $(s::int) < g \implies \{(a::int, b::int). g < a\} \neq \{(a::int, b::int). a = s\}$ 
  apply (rule not-sym)
  apply (rule sets-distinct5)
  by simp

lemma distinctNets:  $FWLink \neq \text{any} \wedge FWLink \neq i4\text{-}32 \wedge FWLink \neq i10\text{-}32 \wedge$ 
 $FWLink \neq \text{eth-intern} \wedge FWLink \neq \text{eth-private} \wedge \text{any} \neq FWLink \wedge \text{any} \neq$ 
 $i4\text{-}32 \wedge \text{any} \neq i10\text{-}32 \wedge \text{any} \neq \text{eth-intern} \wedge \text{any} \neq \text{eth-private} \wedge i4\text{-}32 \neq$ 
 $FWLink \wedge i4\text{-}32 \neq \text{any} \wedge i4\text{-}32 \neq i10\text{-}32 \wedge i4\text{-}32 \neq \text{eth-intern} \wedge i4\text{-}32 \neq$ 
 $\text{eth-private} \wedge i10\text{-}32 \neq FWLink \wedge i10\text{-}32 \neq \text{any} \wedge i10\text{-}32 \neq i4\text{-}32 \wedge i10\text{-}32$ 
 $\neq \text{eth-intern} \wedge i10\text{-}32 \neq \text{eth-private} \wedge \text{eth-intern} \neq FWLink \wedge \text{eth-intern} \neq$ 
 $\text{any} \wedge \text{eth-intern} \neq i4\text{-}32 \wedge \text{eth-intern} \neq i10\text{-}32 \wedge \text{eth-intern} \neq$ 
 $\text{eth-private} \wedge \text{eth-private} \neq FWLink \wedge \text{eth-private} \neq \text{any} \wedge \text{eth-private} \neq$ 
 $i4\text{-}32 \wedge \text{eth-private} \neq i10\text{-}32 \wedge \text{eth-private} \neq \text{eth-intern}$ 
  by (simp add: PolicyL sets-distinct1 sets-distinct2 sets-distinct3
    sets-distinct4 sets-distinct5 sets-distinct6)

lemma aux5:  $\llbracket x \neq a; y \neq b; (x \neq y \wedge x \neq b) \vee (a \neq b \wedge a \neq y) \rrbracket \implies \{x,a\} \neq \{y,b\}$ 
  by auto

```

```

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

lemma ANDex: allNetsDistinct (policy2list Policy)
  apply (simp add: PolicyLemmas allNetsDistinct-def distinctNets)
  apply (simp add: PolicyL)
  by (auto simp: PLemmas PolicyL netsDistinct-def sets-distinct5 sets-distinct6
sets-distinct1
           sets-distinct2)

fun (sequential) numberOfRowsInSection where
  numberOfRowsInSection (a⊕b) = numberOfRowsInSection a + numberOfRowsInSection b
|numberOfRules a = (1::int)

fun numberOfRowsInSectionList where
  numberOfRowsInSectionList (x#xs) = ((numberOfRules x) #(numberOfRulesList xs))
|numberOfRulesList [] = []

lemma all-in-list: all-in-list (policy2list Policy) (Nets-List Policy)
  apply (simp add: PolicyLemmas)
  apply (unfold Nets-List-def)
  apply (unfold bothNets-def)
  apply (insert distinctNets)
  by simp

lemmas normalizeUnfold = normalize-def PolicyL Nets-List-def bothNets-def aux
aux2 bothNets-def sets-distinct1 sets-distinct2 sets-distinct3 sets-distinct4 sets-distinct5
sets-distinct6 aux5 aux2

end

```

3.4 Example: NAT

```

theory
  NAT-FW
  imports
    ../../UPF-Firewall
begin

definition subnet1 :: adr_ip net where
  subnet1 = {{(d,e). d > 1 ∧ d < 256} }

definition subnet2 :: adr_ip net where

```

$subnet2 = \{(d,e). d > 500 \wedge d < 1256\}\}$

definition

$across-subnets x \equiv ((src x \sqsubset subnet1 \wedge (dest x \sqsubset subnet2)) \vee (src x \sqsubset subnet2 \wedge (dest x \sqsubset subnet1)))$

definition

$filter :: (adr_{ip}, DummyContent) FWPolicy \text{ where}$
 $filter = allow-from-port-to (1::port) subnet1 subnet2 ++$
 $allow-from-port-to (2::port) subnet1 subnet2 ++$
 $allow-from-port-to (3::port) subnet1 subnet2 ++ deny-all$

definition

$nat-0 \text{ where}$
 $nat-0 = (A_f(\lambda x. \{x\}))$

lemmas $UnfoldPolicy0 = filter\text{-def } nat-0\text{-def}$
NATLemmas
ProtocolPortCombinators.*ProtocolCombinators*
adr_{ip}Lemmas
packet-defs *across-subnets-def*
subnet1-def *subnet2-def*

lemmas $subnets = subnet1\text{-def } subnet2\text{-def}$

definition $Adr11 :: int \text{ set}$
where $Adr11 = \{d. d > 2 \wedge d < 3\}$

definition $Adr21 :: int \text{ set }$
where $Adr21 = \{d. d > 502 \wedge d < 503\}$

definition $nat-1 \text{ where}$
 $nat-1 = nat-0 ++ (srcPat2pool-IntPort Adr11 Adr21)$

definition $policy-1 \text{ where}$
 $policy-1 = ((\lambda (x,y). x) o-f ((nat-1 \otimes_2 filter) o (\lambda x. (x,x))))$

lemmas $UnfoldPolicy1 = UnfoldPolicy0 \ nat-1\text{-def } Adr11\text{-def } Adr21\text{-def } policy-1\text{-def}$

definition $Adr12 :: int \text{ set}$
where $Adr12 = \{d. d > 4 \wedge d < 6\}$

```

definition Adr22 :: int set where
  Adr22 = {d. d > 504  $\wedge$  d < 506}

definition nat-2 where
  nat-2 = nat-1 ++ (srcPat2pool-IntPort Adr12 Adr22)

definition policy-2 where
  policy-2 = (( $\lambda$  (x,y). x) o-f
  ((nat-2  $\otimes_2$  filter) o ( $\lambda$  x. (x,x)))

lemmas UnfoldPolicy2 = UnfoldPolicy1 nat-2-def Adr12-def Adr22-def policy-2-def

definition Adr13 :: int set
where Adr13 = {d. d > 6  $\wedge$  d < 9}

definition Adr23 :: int set where
  Adr23 = {d. d > 506  $\wedge$  d < 509}

definition nat-3 where
  nat-3 = nat-2 ++ (srcPat2pool-IntPort Adr13 Adr23)

definition policy-3 where
  policy-3 = (( $\lambda$  (x,y). x) o-f
  ((nat-3  $\otimes_2$  filter) o ( $\lambda$  x. (x,x)))

lemmas UnfoldPolicy3 = UnfoldPolicy2 nat-3-def Adr13-def Adr23-def policy-3-def

definition Adr14 :: int set
where Adr14 = {d. d > 8  $\wedge$  d < 12}

definition Adr24 :: int set where
  Adr24 = {d. d > 508  $\wedge$  d < 512}

definition nat-4 where
  nat-4 = nat-3 ++ (srcPat2pool-IntPort Adr14 Adr24)

definition policy-4 where
  policy-4 = (( $\lambda$  (x,y). x) o-f
  ((nat-4  $\otimes_2$  filter) o ( $\lambda$  x. (x,x)))

lemmas UnfoldPolicy4 = UnfoldPolicy3 nat-4-def Adr14-def Adr24-def policy-4-def

definition Adr15 :: int set
where Adr15 = {d. d > 10  $\wedge$  d < 15}

```

```

definition Adr25 :: int set where
  Adr25 = {d. d > 510  $\wedge$  d < 515}

definition nat-5 where
  nat-5 = nat-4 ++ (srcPat2pool-IntPort Adr15 Adr25)

definition policy-5 where
  policy-5 = (( $\lambda$  (x,y). x) o-f
  ((nat-5  $\otimes_2$  filter) o ( $\lambda$  x. (x,x)))))

lemmas UnfoldPolicy5 = UnfoldPolicy4 nat-5-def Adr15-def Adr25-def policy-5-def

definition Adr16 :: int set
where Adr16 = {d. d > 12  $\wedge$  d < 18}

definition Adr26 :: int set where
  Adr26 = {d. d > 512  $\wedge$  d < 518}

definition nat-6 where
  nat-6 = nat-5 ++ (srcPat2pool-IntPort Adr16 Adr26)

definition policy-6 where
  policy-6 = (( $\lambda$  (x,y). x) o-f
  ((nat-6  $\otimes_2$  filter) o ( $\lambda$  x. (x,x)))))

lemmas UnfoldPolicy6 = UnfoldPolicy5 nat-6-def Adr16-def Adr26-def policy-6-def

definition Adr17 :: int set
where Adr17 = {d. d > 14  $\wedge$  d < 21}

definition Adr27 :: int set where
  Adr27 = {d. d > 514  $\wedge$  d < 521}

definition nat-7 where
  nat-7 = nat-6 ++ (srcPat2pool-IntPort Adr17 Adr27)

definition policy-7 where
  policy-7 = (( $\lambda$  (x,y). x) o-f
  ((nat-7  $\otimes_2$  filter) o ( $\lambda$  x. (x,x)))))

lemmas UnfoldPolicy7 = UnfoldPolicy6 nat-7-def Adr17-def Adr27-def policy-7-def

definition Adr18 :: int set

```

where $Adr18 = \{d. d > 16 \wedge d < 24\}$

definition $Adr28 :: int set$ **where**
 $Adr28 = \{d. d > 516 \wedge d < 524\}$

definition $nat-8$ **where**
 $nat-8 = nat-7 ++ (srcPat2pool-IntPort Adr18 Adr28)$

definition $policy-8$ **where**
 $policy-8 = ((\lambda (x,y). x) o-f$
 $((nat-8 \otimes_2 filter) o (\lambda x. (x,x))))$

lemmas $UnfoldPolicy8 = UnfoldPolicy7\ nat-8\text{-def} Adr18\text{-def} Adr28\text{-def} policy-8\text{-def}$

definition $Adr19 :: int set$
where $Adr19 = \{d. d > 18 \wedge d < 27\}$

definition $Adr29 :: int set$ **where**
 $Adr29 = \{d. d > 518 \wedge d < 527\}$

definition $nat-9$ **where**
 $nat-9 = nat-8 ++ (srcPat2pool-IntPort Adr19 Adr29)$

definition $policy-9$ **where**
 $policy-9 = ((\lambda (x,y). x) o-f$
 $((nat-9 \otimes_2 filter) o (\lambda x. (x,x))))$

lemmas $UnfoldPolicy9 = UnfoldPolicy8\ nat-9\text{-def} Adr19\text{-def} Adr29\text{-def} policy-9\text{-def}$

definition $Adr110 :: int set$
where $Adr110 = \{d. d > 20 \wedge d < 30\}$

definition $Adr210 :: int set$ **where**
 $Adr210 = \{d. d > 520 \wedge d < 530\}$

definition $nat-10$ **where**
 $nat-10 = nat-9 ++ (srcPat2pool-IntPort Adr110 Adr210)$

definition $policy-10$ **where**
 $policy-10 = ((\lambda (x,y). x) o-f$
 $((nat-10 \otimes_2 filter) o (\lambda x. (x,x))))$

lemmas $UnfoldPolicy10 = UnfoldPolicy9\ nat-10\text{-def} Adr110\text{-def} Adr210\text{-def} policy-10\text{-def}$

```
end
```

3.5 Voice over IP

theory

Voice-over-IP

imports

.../../UPF-Firewall

begin

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

definition

intranet :: adr_{ip} net where

intranet = {{(a,e) . a = 3}}

definition

internet :: adr_{ip} net where

internet = {{(a,c). a > 4}}

definition

gatekeeper :: adr_{ip} net where

gatekeeper = {{(a,c). a = 4}}

definition

voip-policy :: (adr_{ip},address voip-msg) FWPolicy where

voip-policy = AU

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

definition

is-in-intranet :: address ⇒ bool where

is-in-intranet a = (a = 3)

definition

is-gatekeeper :: address ⇒ bool where

is-gatekeeper a = (a = 4)

definition

is-in-internet :: address ⇒ bool where

is-in-internet a = (a > 4)

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

definition

$\sigma\text{-}0\text{-}voip} :: (adr_{ip}, address \text{ voip-msg}) history \times (adr_{ip}, address \text{ voip-msg}) FWPolicy$

where

$\sigma\text{-}0\text{-}voip} = ([] , voip-policy)$

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

definition $accept\text{-}voip} :: (adr_{ip}, address \text{ voip-msg}) history \Rightarrow \text{bool}$ **where**

$accept\text{-}voip} t = (\exists c s g i p1 p2. t \in NB\text{-}voip c s g i p1 p2 \wedge is\text{-}in\text{-}intranet c \wedge is\text{-}in\text{-}internet s \wedge is\text{-}gatekeeper g)$

fun $packet\text{-}with\text{-}id$ **where**

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

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

fun $ids1$ **where**

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

lemmas $ST\text{-}simp = Let\text{-}def valid\text{-}SE\text{-}def unit\text{-}SE\text{-}def bind\text{-}SE\text{-}def$
 $subnet\text{-}of\text{-}int\text{-}def p\text{-}accept\text{-}def content\text{-}def$
 $is\text{-}in\text{-}intranet\text{-}def is\text{-}in\text{-}internet\text{-}def intranet\text{-}def internet\text{-}def exI$
 $subnetOf\text{-}lemma subnetOf\text{-}lemma2 subnetOf\text{-}lemma3 subnetOf\text{-}lemma4 voip\text{-}policy\text{-}def$
 $NetworkCore.id\text{-}def is\text{-}arq\text{-}def is\text{-}fin\text{-}def$
 $is\text{-}connect\text{-}def is\text{-}setup\text{-}def ports\text{-}open\text{-}def subnet\text{-}of\text{-}adr\text{-}def$
 $VOIP.NB\text{-}voip\text{-}def \sigma\text{-}0\text{-}voip\text{-}def PLemmas VOIP\text{-}TRPolicy\text{-}def$
 $policy2MON\text{-}def applyPolicy\text{-}def$

end

Bibliography

- [1] A. D. Brucker and B. Wolff. Interactive testing using HOL-TestGen. In W. Grieskamp and C. Weise, editors, *Formal Approaches to Testing of Software*, number 3997 in Lecture Notes in Computer Science. Springer-Verlag, 2005. ISBN 3-540-25109-X. doi: [10.1007/11759744_7](https://doi.org/10.1007/11759744_7).
- [2] A. D. Brucker and B. Wolff. On theorem prover-based testing. *Formal Aspects of Computing*, 25(5):683–721, 2013. ISSN 0934-5043. doi: [10.1007/s00165-012-0222-y](https://doi.org/10.1007/s00165-012-0222-y).
- [3] 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.
- [4] A. D. Brucker, L. Brügger, P. Kearney, and B. Wolff. An approach to modular and testable security models of real-world health-care applications. pages 133–142. ACM Press, 2011. ISBN 978-1-4503-0688-1. doi: [10.1145/1998441.1998461](https://doi.org/10.1145/1998441.1998461).
- [5] A. D. Brucker, L. Brügger, and B. Wolff. Hol-testgen/fw: An environment for specification-based firewall conformance testing. In Z. Liu, J. Woodcock, and H. Zhu, editors, *International Colloquium on Theoretical Aspects of Computing (ICTAC)*, number 8049 in Lecture Notes in Computer Science, pages 112–121. Springer-Verlag, 2013. ISBN 978-3-642-39717-2. doi: [10.1007/978-3-642-39718-9_7](https://doi.org/10.1007/978-3-642-39718-9_7).
- [6] A. D. Brucker, L. Brügger, and B. Wolff. The unified policy framework (upf). *Archive of Formal Proofs*, sep 2014. ISSN 2150-914x. URL <https://www.brucker.ch/bibliography/abstract/brucker.ea-upf-2014>. <http://www.isa-afp.org/entries/UPF.shtml>, Formal proof development.
- [7] A. D. Brucker, L. Brügger, and B. Wolff. Formal firewall conformance testing: An application of test and proof techniques. *Software Testing, Verification & Reliability (STVR)*, 25(1):34–71, 2015. doi: [10.1002/stvr.1544](https://doi.org/10.1002/stvr.1544). URL <https://www.brucker.ch/bibliography/abstract/brucker.ea-formal-fw-testing-2014>.
- [8] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, Inc., Upper Saddle River, NJ, USA, 2nd edition, 1992. ISBN 0-139-78529-9.