

# 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](mailto:a.brucker@sheffield.ac.uk)

Information Security, ETH Zurich, 8092 Zurich, Switzerland  
[Lukas.A.Bruegger@gmail.com](mailto:Lukas.A.Bruegger@gmail.com)

‡Univ. Paris-Sud, Laboratoire LRI, UMR8623, 91405 Orsay, France France  
[burkhart.wolff@lri.fr](mailto:burkhart.wolff@lri.fr)

March 20, 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

<b>1</b>	<b>Introduction</b>	<b>5</b>
1.1	Motivation . . . . .	5
1.2	The Unified Policy Framework (UPF) . . . . .	5
<b>2</b>	<b>UPF Firewall</b>	<b>9</b>
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 . . . . .	72
2.3.4	Normalisation Proofs: Integer Protocol . . . . .	96
2.4	Stateful Network Protocols . . . . .	118
2.4.1	Stateful Protocols: Foundations . . . . .	119
2.4.2	The File Transfer Protocol (ftp) . . . . .	120
2.4.3	FTP enriched with a security policy . . . . .	124
2.4.4	A simple voice-over-ip model . . . . .	125
2.4.5	FTP and VoIP Protocol . . . . .	130
<b>3</b>	<b>Examples</b>	<b>137</b>
3.1	A Simple DMZ Setup . . . . .	137
3.1.1	DMZ Datatype . . . . .	137
3.1.2	DMZ: Integer . . . . .	139

3.2	Personal Firewall . . . . .	141
3.2.1	Personal Firewall: Integer . . . . .	141
3.2.2	Personal Firewall IPv4 . . . . .	142
3.2.3	Personal Firewall: Datatype . . . . .	143
3.3	Demonstrating Policy Transformations . . . . .	145
3.3.1	Transformation Example 1 . . . . .	145
3.3.2	Transforamtion Example 2 . . . . .	149
3.4	Example: NAT . . . . .	152
3.5	Voice over IP . . . . .	156

# 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.

$$\text{type\_synonym } \alpha \mapsto \beta = \alpha \multimap \beta \text{ decision}$$

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*  $\text{dom } p :: \alpha \multimap \beta \Rightarrow \alpha \text{ set}$  and a *range*  $\text{ran } p :: [\alpha \multimap \beta] \Rightarrow \beta \text{ set}$  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 *Integer*<sup>1</sup>.

---

<sup>1</sup>For 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 of the networks are relatively small. It suffices to provide the required types, a couple of lemmas, and - if required - a definition for the source and destination ports of a packet.

**end**

### 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**  $'\alpha$  *src* =  $'\alpha$

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

**instance** *int* :: *adr*  $\langle$ *proof* $\rangle$

**instance** *nat* :: *adr*  $\langle$ *proof* $\rangle$

**instance** *fun* :: (*adr,adr*) *adr* <*proof*>  
**instance** *prod* :: (*adr,adr*) *adr* <*proof*>

The content is also specified with an unconstrained generic type:

**type-synonym** *'β content* = *'β*

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** (*'α,'β*) *packet* = *id* × *'α src* × *'α dest* × *'β 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* :: (*'α::adr,'β*) *packet* ⇒ *'α*  
**where** *src* = *fst o snd*

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

**class** *port*

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

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

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

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

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

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

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

**lemma** *either*:  $\llbracket a \neq tcp; a \neq udp \rrbracket \implies False$   
*<proof>*

**lemma** *either2[simp]*:  $(a \neq tcp) = (a = udp)$   
*<proof>*

**lemma** *either3[simp]*:  $(a \neq udp) = (a = tcp)$   
*<proof>*

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

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

**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**  $\langle \sqsubset \rangle$  100) **where**

*in\_subnet*  $a S = (\exists s \in S. a \in s)$

The following lemmas will be useful later.

**lemma** *in\_subnet*:

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

*<proof>*

**lemma** *src-in\_subnet*:

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

*<proof>*

**lemma** *dest-in\_subnet*:

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

*<proof>*

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*  $\langle$ *proof* $\rangle$

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

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

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

lemma src-port : src-port ((a,x,d,e)::(DatatypePort, 'β) packet) = snd x
    ⟨proof⟩

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

lemmas DatatypePortLemmas = src-port dest-port src-port-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<sub>i</sub> = 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<sub>ip</sub> = address × port*

**overloading** *src-port-int* ≡ *src-port* :: ('α::adr,'β) packet ⇒ 'γ::port

**begin**

**definition**

*src-port-int* (*x*::(adr<sub>ip</sub>,β) packet) ≡ (snd o fst o snd) *x*

**end**

**overloading** *dest-port-int* ≡ *dest-port* :: ('α::adr,'β) packet ⇒ 'γ::port

**begin**

**definition**

*dest-port-int* (*x*::(adr<sub>ip</sub>,β) packet) ≡ (snd o fst o snd o snd) *x*

**end**

**overloading** *subnet-of-int* ≡ *subnet-of* :: 'α::adr ⇒ 'α net

**begin**

**definition**

*subnet-of-int* (*x*::(adr<sub>ip</sub>)) ≡ {(a,b::int). a = fst *x*}

**end**

**lemma** *src-port*: *src-port* (*a,x*::adr<sub>ip</sub>,d,e) = snd *x*

*<proof>*

**lemma** *dest-port*: *dest-port* (*a,d,x::adr<sub>ip</sub>,e*) = *snd x*

*<proof>*

**lemmas** *adr<sub>ip</sub>Lemmas* = *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**

*adr<sub>ipp</sub>* = *address* × *port* × *protocol*

**instance** *protocol* :: *adr* *<proof>*

**overloading** *src-port-int-TCPUDP* ≡ *src-port* :: (*'α::adr,'β*) *packet* ⇒ *'γ::port*

**begin**

**definition**

*src-port-int-TCPUDP* (*x::(adr<sub>ipp</sub>,β)*) *packet* ≡ (*fst o snd o fst o snd*) *x*

**end**

**overloading** *dest-port-int-TCPUDP* ≡ *dest-port* :: (*'α::adr,'β*) *packet* ⇒ *'γ::port*

**begin**

**definition**

*dest-port-int-TCPUDP* (*x::(adr<sub>ipp</sub>,β)*) *packet* ≡ (*fst o snd o fst o snd o snd*) *x*

**end**

**overloading** *subnet-of-int-TCPUDP* ≡ *subnet-of* :: *'α::adr* ⇒ *'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*  $\equiv src\text{-}protocol :: ('\alpha::adr, '\beta)\ packet \Rightarrow protocol$

**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*  $\equiv dest\text{-}protocol :: ('\alpha::adr, '\beta)\ packet \Rightarrow proto\text{-}col$

**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*:  $src\text{-}port\ (a, x::adr_{ipp}, d, e) = fst\ (snd\ x)$

*<proof>*

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

*<proof>*

Common test constraints:

**definition** *port-positive*  $:: (adr_{ipp}, '\beta)\ packet \Rightarrow bool$  **where**

*port-positive*  $x = (dest\text{-}port\ x > (0::port))$

**definition** *fix-values*  $:: (adr_{ipp}, DummyContent)\ packet \Rightarrow bool$  **where**

*fix-values*  $x = (src\text{-}port\ x = (1::port) \wedge src\text{-}protocol\ x = udp \wedge content\ x = data \wedge id\ x = 1)$

**lemmas** *adr<sub>ipp</sub>Lemmas* = *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** *adr<sub>ipp</sub>TestConstraints* = *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
  ⟨proof⟩

lemma dest-port: dest-port (a, d, (x::ipv4), e) = snd x
  ⟨proof⟩

```

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* <*proof*>

**overloading** *src-port-ipv4-TCPUDP* ≡ *src-port* :: ('α::*adr*, 'β) *packet* ⇒ '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* ⇒ '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* ⇒ 'α *net*

**begin**

**definition**

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

**end**

**overloading** *dest-protocol-ipv4-TCPUDP* ≡ *dest-protocol* :: ('α::*adr*, 'β) *packet* ⇒ *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*)  
 $\langle$ *proof* $\rangle$

**lemma** *dest-port*: *dest-port* (*a*, *d*, (*x*::*ipv4-TCPUDP*), *e*) = *fst* (*snd x*)  
 $\langle$ *proof* $\rangle$

**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** ( $'\alpha$ ,  $'\beta$ ) *FWPolicy* = ( $'\alpha$ ,  $'\beta$ ) *packet*  $\mapsto$  *unit*

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

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

**definition**  
*p-accept* :: ( $'\alpha$ ,  $'\beta$ ) *packet*  $\Rightarrow$  ( $'\alpha$ ,  $'\beta$ ) *FWPolicy*  $\Rightarrow$  *bool* **where**

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

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**

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

**definition**

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

**definition**

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

**definition**

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

**definition**

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

**definition**

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

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 the ones defined in the PolicyCombinators theory.

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

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

**definition**

$allow\_all\_from\_port :: '\alpha :: adr\ net \Rightarrow (' \gamma :: port) \Rightarrow ((' \alpha, '\beta) packet \mapsto unit)$  **where**  
 $allow\_all\_from\_port\ src\_net\ s\_port = \{pa.\ src\_port\ pa = s\_port\} \triangleleft allow\_all\_from\ src\_net$

**definition**

$deny\_all\_from\_port :: '\alpha :: adr\ net \Rightarrow (' \gamma :: port) \Rightarrow ((' \alpha, '\beta) packet \mapsto unit)$  **where**  
 $deny\_all\_from\_port\ src\_net\ s\_port = \{pa.\ src\_port\ pa = s\_port\} \triangleleft deny\_all\_from\ src\_net$

**definition**

$allow\_all\_to\_port :: '\alpha :: adr\ net \Rightarrow (' \gamma :: port) \Rightarrow ((' \alpha, '\beta) packet \mapsto unit)$  **where**  
 $allow\_all\_to\_port\ dest\_net\ d\_port = \{pa.\ dest\_port\ pa = d\_port\} \triangleleft allow\_all\_to\ dest\_net$

**definition**

$deny\_all\_to\_port :: '\alpha :: adr\ net \Rightarrow (' \gamma :: port) \Rightarrow ((' \alpha, '\beta) packet \mapsto unit)$  **where**  
 $deny\_all\_to\_port\ dest\_net\ d\_port = \{pa.\ dest\_port\ pa = d\_port\} \triangleleft deny\_all\_to\ dest\_net$

**definition**

$allow\_all\_from\_port\_to :: '\alpha :: adr\ net \Rightarrow (' \gamma :: port) \Rightarrow '\alpha :: adr\ net \Rightarrow ((' \alpha, '\beta) packet \mapsto unit)$   
**where**  
 $allow\_all\_from\_port\_to\ src\_net\ s\_port\ dest\_net$   
 $= \{pa.\ src\_port\ pa = s\_port\} \triangleleft allow\_all\_from\_to\ src\_net\ dest\_net$

**definition**

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

*deny-all-from-port-to src-net s-port dest-net*  
= {pa. src-port pa = s-port} ◁ *deny-all-from-to src-net dest-net*

**definition**

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

*allow-all-from-port-to-port src-net s-port dest-net d-port* =  
{pa. dest-port pa = d-port} ◁ *allow-all-from-port-to src-net s-port dest-net*

**definition**

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

*deny-all-from-port-to-port src-net s-port dest-net d-port* =  
{pa. dest-port pa = d-port} ◁ *deny-all-from-port-to src-net s-port dest-net*

**definition**

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

*allow-all-from-to-port src-net dest-net d-port* =  
{pa. dest-port pa = d-port} ◁ *allow-all-from-to src-net dest-net*

**definition**

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

*deny-all-from-to-port src-net dest-net d-port* =  
{pa. dest-port pa = d-port} ◁ *deny-all-from-to src-net dest-net*

**definition**

*allow-from-port-to* :: 'γ::port ⇒ 'α::adr net ⇒ 'α::adr net ⇒ (('α,'β) packet ↦ unit)  
**where**

*allow-from-port-to port src-net dest-net* =  
{pa. src-port pa = port} ◁ *allow-all-from-to src-net dest-net*

**definition**

*deny-from-port-to* :: 'γ::port ⇒ 'α::adr net ⇒ 'α::adr net ⇒ (('α,'β) packet ↦ unit)  
**where**

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

**definition**

*allow-from-to-port* :: 'γ::port ⇒ 'α::adr net ⇒ 'α::adr net ⇒ (('α,'β) packet ↦ unit)  
**where**

*allow-from-to-port* port src-net dest-net =  
 {pa. dest-port pa = port} ◁ *allow-all-from-to* src-net dest-net

**definition**

*deny-from-to-port* :: 'γ::port ⇒ 'α::adr net ⇒ 'α::adr net ⇒ (('α,'β) packet ↦ unit)

**where**

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

**definition**

*allow-from-ports-to* :: 'γ::port set ⇒ 'α::adr net ⇒ 'α::adr net ⇒  
 (('α,'β) packet ↦ unit) **where**

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

**definition**

*allow-from-to-ports* :: 'γ::port set ⇒ 'α::adr net ⇒ 'α::adr net ⇒  
 (('α,'β) packet ↦ unit) **where**

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

**definition**

*deny-from-ports-to* :: 'γ::port set ⇒ 'α::adr net ⇒ 'α::adr net ⇒  
 (('α,'β) packet ↦ unit) **where**

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

**definition**

*deny-from-to-ports* :: 'γ::port set ⇒ 'α::adr net ⇒ 'α::adr net ⇒  
 (('α,'β) packet ↦ unit) **where**

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

**definition**

*allow-all-from-port-tos*:: 'α::adr net ⇒ ('γ::port) set ⇒ 'α::adr net ⇒ (('α,'β) packet  
 ↦ unit)

**where**

*allow-all-from-port-tos* src-net s-port dest-net  
 = {pa. dest-port pa ∈ s-port} ◁ *allow-all-from-to* src-net dest-net

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

**lemmas** *PortCombinatorsCore* =

*allow-all-from-port-def* *deny-all-from-port-def* *allow-all-to-port-def*

```

deny-all-to-port-def allow-all-from-to-port-def
deny-all-from-to-port-def
allow-from-ports-to-def allow-from-to-ports-def
deny-from-ports-to-def deny-from-to-ports-def
allow-all-from-port-to-def deny-all-from-port-to-def
allow-from-port-to-def allow-from-to-port-def deny-from-to-port-def
deny-from-port-to-def allow-all-from-port-to-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 the ones defined in the *PolicyCombinators* theory.

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

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

**definition**

$allow\_all\_from\_port\_prot :: protocol \Rightarrow '\alpha :: adr\ net \Rightarrow (' \gamma :: port) \Rightarrow ((' \alpha, '\beta) packet \mapsto unit)$  **where**

$allow\_all\_from\_port\_prot\ p\ src\_net\ s\_port =$   
 $\{pa. dest\_protocol\ pa = p\} \triangleleft allow\_all\_from\_port\ src\_net\ s\_port$

**definition**

$deny\_all\_from\_port\_prot :: protocol \Rightarrow '\alpha :: adr\ net \Rightarrow (' \gamma :: port) \Rightarrow ((' \alpha, '\beta) packet \mapsto unit)$  **where**

$deny\_all\_from\_port\_prot\ p\ src\_net\ s\_port =$   
 $\{pa. dest\_protocol\ pa = p\} \triangleleft deny\_all\_from\_port\ src\_net\ s\_port$

**definition**

$allow\_all\_to\_port\_prot :: protocol \Rightarrow '\alpha :: adr\ net \Rightarrow (' \gamma :: port) \Rightarrow ((' \alpha, '\beta) packet \mapsto unit)$  **where**

$allow\_all\_to\_port\_prot\ p\ dest\_net\ d\_port =$

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

**definition**

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

**where**

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

**definition**

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

**where**

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

**definition**

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

**where**

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

**definition**

$\text{allow-all-from-port-to-port-prot} :: \text{protocol} \Rightarrow 'a::\text{adr net} \Rightarrow 'b::\text{port} \Rightarrow 'c::\text{adr net} \Rightarrow 'd::\text{port} \Rightarrow$

$(('a, 'b) \text{ packet} \mapsto \text{unit})$  **where**

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

**definition**

$\text{deny-all-from-port-to-port-prot} :: \text{protocol} \Rightarrow 'a::\text{adr net} \Rightarrow 'b::\text{port} \Rightarrow 'c::\text{adr net} \Rightarrow 'd::\text{port} \Rightarrow$

$'e::\text{port} \Rightarrow (('a, 'b) \text{ packet} \mapsto \text{unit})$  **where**

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

**definition**

$\text{allow-all-from-to-port-prot} :: \text{protocol} \Rightarrow 'a::\text{adr net} \Rightarrow 'b::\text{adr net} \Rightarrow 'c::\text{port} \Rightarrow (('a, 'b) \text{ packet} \mapsto \text{unit})$  **where**

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

**definition**

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

**definition**

$allow-from-port-to-prot \quad :: \text{protocol} \Rightarrow 'c::\text{port} \Rightarrow 'a::\text{adr net} \Rightarrow 'a::\text{adr net} \Rightarrow (( 'a, 'b)$   
 $\text{packet} \mapsto \text{unit})$

**where**

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

**definition**

$deny-from-port-to-prot \quad :: \text{protocol} \Rightarrow 'c::\text{port} \Rightarrow 'a::\text{adr net} \Rightarrow 'a::\text{adr net} \Rightarrow (( 'a, 'b)$   
 $\text{packet} \mapsto \text{unit})$

**where**

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

**definition**

$allow-from-to-port-prot \quad :: \text{protocol} \Rightarrow 'c::\text{port} \Rightarrow 'a::\text{adr net} \Rightarrow 'a::\text{adr net} \Rightarrow (( 'a, 'b)$   
 $\text{packet} \mapsto \text{unit})$

**where**

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

**definition**

$deny-from-to-port-prot \quad :: \text{protocol} \Rightarrow 'c::\text{port} \Rightarrow 'a::\text{adr net} \Rightarrow 'a::\text{adr net} \Rightarrow (( 'a, 'b)$   
 $\text{packet} \mapsto \text{unit})$

**where**

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

**definition**

$allow-from-ports-to-prot \quad :: \text{protocol} \Rightarrow 'c::\text{port set} \Rightarrow 'a::\text{adr net} \Rightarrow 'a::\text{adr net} \Rightarrow$   
 $(( 'a, 'b) \text{ packet} \mapsto \text{unit}) \textbf{ where}$

$allow-from-ports-to-prot \ p \ \text{ports} \ \text{src-net} \ \text{dest-net} =$   
 $\{pa. \ \text{dest-protocol} \ pa = p\} \triangleleft allow-from-ports-to \ \text{ports} \ \text{src-net} \ \text{dest-net}$

**definition**

$allow-from-to-ports-prot \quad :: \text{protocol} \Rightarrow 'c::\text{port set} \Rightarrow 'a::\text{adr net} \Rightarrow 'a::\text{adr net} \Rightarrow$   
 $(( 'a, 'b) \text{ packet} \mapsto \text{unit}) \textbf{ where}$

$allow-from-to-ports-prot \ p \ \text{ports} \ \text{src-net} \ \text{dest-net} =$   
 $\{pa. \ \text{dest-protocol} \ pa = p\} \triangleleft allow-from-to-ports \ \text{ports} \ \text{src-net} \ \text{dest-net}$

**definition**

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

**definition**

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

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

**lemmas** *ProtocolCombinatorsCore* =

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

**lemmas** *ProtocolCombinators* = *PortCombinators.PortCombinators ProtocolCombinatorsCore*

**end**

## 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*:  $x \neq 80 \implies x \neq \text{http}$   
 ⟨*proof*⟩

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

*<proof>*

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

**lemma** *smtp1: x ≠ 25 ⇒ x ≠ smtp*

*<proof>*

**lemma** *smtp2: x ≠ 25 ⇒ smtp ≠ x*

*<proof>*

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

**lemma** *ftp1: x ≠ 21 ⇒ x ≠ ftp*

*<proof>*

**lemma** *ftp2: x ≠ 21 ⇒ ftp ≠ x*

*<proof>*

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 :: 'α set ⇒ ('α::adr,'β) packet ⇒ ('α,'β) packet set* **where**  
*src2pool t = (λ p. ({(i,s,d,da). (i = id p ∧ s ∈ t ∧ d = dest p ∧ da = content p)}))*

**definition** *src2poolAP* **where**

*src2poolAP t = A<sub>f</sub> (src2pool t)*

**definition** *srcNat2pool :: 'α set ⇒ 'α set ⇒ ('α::adr,'β) packet ↦ ('α,'β) packet set*  
**where**

*srcNat2pool srcs transl = {x. src x ∈ srcs} ◁ (src2poolAP transl)*

**definition** *src2poolPort :: int set ⇒ (adr<sub>ip</sub>, 'β) packet ⇒ (adr<sub>ip</sub>, 'β) packet set* **where**  
*src2poolPort t = (λ p. ({(i,(s1,s2),(d1,d2),da).*

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

**definition** *src2poolPort-Protocol* :: *int set*  $\Rightarrow$  (*adr<sub>ipp</sub>*,*'β*) *packet*  $\Rightarrow$  (*adr<sub>ipp</sub>*,*'β*) *packet set* **where**

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

**definition** *srcNat2pool-IntPort* :: *address set*  $\Rightarrow$  *address set*  $\Rightarrow$  (*adr<sub>ip</sub>*,*'β*) *packet*  $\mapsto$  (*adr<sub>ip</sub>*,*'β*) *packet set* **where**

$$\begin{aligned} srcNat2pool-IntPort\ srcs\ transl &= \\ \{x.\ fst\ (src\ x) \in srcs\} &\triangleleft (A_f\ (src2poolPort\ transl)) \end{aligned}$$

**definition** *srcNat2pool-IntProtocolPort* :: *int set*  $\Rightarrow$  *int set*  $\Rightarrow$  (*adr<sub>ipp</sub>*,*'β*) *packet*  $\mapsto$  (*adr<sub>ipp</sub>*,*'β*) *packet set* **where**

$$\begin{aligned} srcNat2pool-IntProtocolPort\ srcs\ transl &= \\ \{x.\ (fst\ (src\ x)) \in srcs\} &\triangleleft (A_f\ (src2poolPort-Protocol\ transl)) \end{aligned}$$

**definition** *srcPat2poolPort-t* :: *int set*  $\Rightarrow$  (*adr<sub>ip</sub>*,*'β*) *packet*  $\Rightarrow$  (*adr<sub>ip</sub>*,*'β*) *packet set* **where**

$$\begin{aligned} srcPat2poolPort-t\ t &= (\lambda\ p.\ (\{(i,(s1,s2),(d1,d2),da).\ \\ (i = id\ p \wedge s1 \in t \wedge d1 = (fst\ (dest\ p)) \wedge d2 = snd\ (dest\ p) \wedge da = content\ p)\})) \end{aligned}$$

**definition** *srcPat2poolPort-Protocol-t* :: *int set*  $\Rightarrow$  (*adr<sub>ipp</sub>*,*'β*) *packet*  $\Rightarrow$  (*adr<sub>ipp</sub>*,*'β*) *packet set* **where**

$$\begin{aligned} srcPat2poolPort-Protocol-t\ t &= (\lambda\ p.\ (\{(i,(s1,s2,s3),(d1,d2,d3),da).\ \\ (i = id\ p \wedge s1 \in t \wedge s3 = src-protocol\ p \wedge (d1,d2,d3) = dest\ p \wedge da = content\ p)\})) \end{aligned}$$

**definition** *srcPat2pool-IntPort* :: *int set*  $\Rightarrow$  *int set*  $\Rightarrow$  (*adr<sub>ip</sub>*,*'β*) *packet*  $\mapsto$  (*adr<sub>ip</sub>*,*'β*) *packet set* **where**

$$\begin{aligned} srcPat2pool-IntPort\ srcs\ transl &= \\ \{x.\ (fst\ (src\ x)) \in srcs\} &\triangleleft (A_f\ (srcPat2poolPort-t\ transl)) \end{aligned}$$

**definition** *srcPat2pool-IntProtocol* ::

$$int\ set \Rightarrow int\ set \Rightarrow (adr_{ipp},'\beta)\ packet \mapsto (adr_{ipp},'\beta)\ packet\ set\ \mathbf{where}$$

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

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).\$

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

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

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

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

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

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

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

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

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

```

      b = X607X4 ∧ ba = data}
⟨proof⟩

lemma datasimp6: {(i, (s1, s2, s3), aba).
  i = i1 ∧ s1 = i132 ∧ s3 = iudp ∧ s2 = i1 ∧
  (λ ((a,aa,b),ba). a = i110 ∧ aa = i4 ∧ b = iudp ∧ ba = data) aba}
= {(i, (s1, s2, s3), (a,aa,b),ba).
  i = i1 ∧ s1 = i132 ∧ s3 = iudp ∧ s2 = i1 ∧ a = i110 ∧
  aa = i4 ∧ b = iudp ∧ ba = data}
⟨proof⟩

lemma datasimp7: {(i, (s1, s2, s3), aba).
  i = i1 ∧ i115 < s1 ∧ s1 < i124 ∧ s3 = iudp ∧ s2 = ii1 ∧
  (λ ((a,aa,b),ba). a = i110 ∧ aa = i3 ∧ b = itcp ∧ ba = data) aba}
= {(i, (s1, s2, s3), (a,aa,b),ba).
  i = i1 ∧ i115 < s1 ∧ s1 < i124 ∧ s3 = iudp ∧ s2 = ii1
  ∧ a = i110 ∧ aa = i3 ∧ b = itcp ∧ ba = data}
⟨proof⟩

lemma datasimp8: {(i, (s1, s2, s3), aba). i = i1 ∧ s1 = i132 ∧ s3 = iudp ∧ s2 = ii1
  ∧
  (λ ((a,aa,b),ba). a = i110 ∧ aa = i7 ∧ b = itcp ∧ ba = data) aba}
= {(i, (s1, s2, s3), (a,aa,b),ba). i = i1 ∧ s1 = i132 ∧ s3 = iudp
  ∧ s2 = ii1 ∧ a = i110 ∧ aa = i7 ∧ b = itcp ∧ ba = data}
⟨proof⟩

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
  NormalisationIPPProofs
  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 (' $\alpha$ , ' $\beta$ ) Combinators =  
  DenyAll  
  | DenyAllFromTo ' $\alpha$  ' $\alpha$   
  | AllowPortFromTo ' $\alpha$  ' $\alpha$  ' $\beta$   
  | Conc ((' $\alpha$ , ' $\beta$ ) Combinators) ((' $\alpha$ , ' $\beta$ ) Combinators) (infixr < $\oplus$ > 80)
```

And define the semantic interpretation of it. For technical reasons, we fix here the type to policies over IntegerPort addresses. However, we could easily provide definitions for other address types as well, using a generic 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 ( $v \oplus va$ ) = undefined
```

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

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

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

```
fun (sequential) net-list-aux where  
  net-list-aux [] = []
```

```

|net-list-aux (DenyAll#xs) = net-list-aux xs
|net-list-aux ((DenyAllFromTo x y)#xs) = x#y#(net-list-aux xs)
|net-list-aux ((AllowPortFromTo x y p)#xs) = x#y#(net-list-aux xs)
|net-list-aux ((x⊕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*  $x\ l = (bothNet\ x \in\ set\ l)$

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

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

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

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

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

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

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

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

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

## Invariants

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

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

There is a DenyAll at the first position

**fun** *wellformed-policy1-strong*:: ( $'\alpha,\ ' \beta$ ) *Combinators*) *list*  $\Rightarrow\ bool$   
**where**

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

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*  $a\ b\ c\ d = (\text{netsDistinct } a\ c \vee \text{netsDistinct } b\ d)$

**definition** *allNetsDistinct* **where**

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

**definition** *disjSD-2* **where**

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

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

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

**fun** *OnlyTwoNets* **where**

*OnlyTwoNets* ( $\text{DenyAll} \# xs$ ) = *OnlyTwoNets xs*

| *OnlyTwoNets* ( $x \# xs$ ) = (*onlyTwoNets x*  $\wedge$  *OnlyTwoNets xs*)

| *OnlyTwoNets* [] = *True*

**fun** *noDenyAll* **where**

*noDenyAll* ( $x \# xs$ ) = ( $(\neg \text{member } \text{DenyAll } x) \wedge \text{noDenyAll } xs$ )

| *noDenyAll* [] = *True*

**fun** *noDenyAll1* **where**

*noDenyAll1* ( $\text{DenyAll} \# xs$ ) = *noDenyAll xs*

| *noDenyAll1*  $xs = \text{noDenyAll } xs$

**fun** *separated* **where**

*separated* ( $x \# xs$ ) = ( $(\forall s. s \in \text{set } xs \longrightarrow \text{disjSD-2 } x\ s) \wedge \text{separated } xs$ )

| *separated* [] = *True*

**fun** *NetsCollected* **where**

*NetsCollected* ( $x\#xs$ ) = (((*first-bothNet*  $x \neq \text{firstList } xs$ )  $\longrightarrow$   
( $\forall a \in \text{set } xs. \text{first-bothNet } x \neq \text{first-bothNet } a$ )  $\wedge \text{NetsCollected } (xs)$ )  
| *NetsCollected* [] = *True*

**fun** *NetsCollected2* **where**

*NetsCollected2* ( $x\#xs$ ) = ( $xs = [] \vee (\text{first-bothNet } x \neq \text{firstList } xs \wedge$   
*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*  $\Rightarrow$

((('α, 'β) *Combinators*) *list* **where**  
*policy2list* ( $x \oplus y$ ) = (*concat* [(*policy2list*  $x$ ),(*policy2list*  $y$ )])  
| *policy2list*  $x = [x]$

**fun** *list2FWpolicy*::('α, 'β) *Combinators* *list*  $\Rightarrow$

((('α, 'β) *Combinators*) **where**  
*list2FWpolicy* [] = *undefined*  
| *list2FWpolicy* ( $x\#[]$ ) =  $x$   
| *list2FWpolicy* ( $x\#y$ ) =  $x \oplus (\text{list2FWpolicy } y)$

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

**fun** *removeShadowRules1* **where**

*removeShadowRules1* ( $x\#xs$ ) = (*if* (*DenyAll*  $\in \text{set } xs$ )  
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* ( $x\#xs$ )=  
 $x\#(\text{removeShadowRules1-alternative-rev } xs)$

**definition** *removeShadowRules1-alternative* **where**

*removeShadowRules1-alternative*  $p =$   
*rev* (*removeShadowRules1-alternative-rev* (*rev*  $p$ ))

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

**fun** *removeShadowRules2*:: (('α, 'β) Combinators) list ⇒  
 (('α, 'β) Combinators) list

**where**

(*removeShadowRules2* ((*AllowPortFromTo* x y p)#z)) =  
 (if (((*DenyAllFromTo* x y) ∈ set z))  
 then ((*removeShadowRules2* z))  
 else (((*AllowPortFromTo* x y p)#(*removeShadowRules2* z))))  
 | *removeShadowRules2* (x#y) = x#(*removeShadowRules2* y)  
 | *removeShadowRules2* [] = []

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

**fun** *smaller* :: ('α, 'β) Combinators ⇒  
 ('α, 'β) Combinators ⇒  
 (('α) set) list ⇒ bool

**where**

*smaller DenyAll* x l = True  
 | *smaller* x *DenyAll* l = False  
 | *smaller* x y l =  
 ((x = y) ∨ (if (bothNet x) = (bothNet y) then  
 (case y of (*DenyAllFromTo* a b) ⇒ (x = *DenyAllFromTo* b a)  
 | - ⇒ True)  
 else  
 (position (bothNet x) l <= position (bothNet y) l)))

We provide two different sorting algorithms: Quick Sort (*qsort*) and Insertion Sort (*sort*)

**fun** *qsort* **where**

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

**lemma** *qsort-permutes*:

set (*qsort* xs l) = set xs  
 ⟨proof⟩

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

⟨proof⟩

**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 = insert x (sort xs l) l

```

**fun sorted where**

```

  sorted [] l = True
| sorted [x] l = True
| sorted (x#y#zs) l = (smaller x y l ∧ sorted (y#zs) l)

```

**fun separate where**

```

  separate (DenyAll#x) = DenyAll#(separate x)
| separate (x#y#z) = (if (first-bothNet x = first-bothNet y)
                        then (separate ((x⊕y)#z))
                        else (x#(separate(y#z))))
| separate x = x

```

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

**fun insertDenies where**

```

  insertDenies (x#xs) = (case x of DenyAll ⇒ (DenyAll#(insertDenies xs))
                        | - ⇒ (DenyAllFromTo (first-srcNet x) (first-destNet x) ⊕
                                   (DenyAllFromTo (first-destNet x) (first-srcNet x)) ⊕ x)#
                           (insertDenies xs))
| insertDenies [] = []

```

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

**fun removeDuplicates where**

```

  removeDuplicates (x⊕xs) = (if member x xs then (removeDuplicates xs)
                             else x⊕(removeDuplicates xs))
| removeDuplicates x = x

```

**fun removeAllDuplicates where**

```

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

```

Insert a DenyAll at the beginning of a policy.

**fun insertDeny where**

```

  insertDeny (DenyAll#xs) = DenyAll#xs
| insertDeny xs = DenyAll#xs

```

**definition**  $sort' p l = sort l p$

**definition**  $qsort' p l = qsort l p$

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

**fun** *list2policyR*::((' $\alpha$ ', ' $\beta$ ') *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* :: (*adr<sub>ip</sub>* *net*, *port*) *Combinators*  $\Rightarrow$  (*adr<sub>ip</sub>*, *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* :: (*adr<sub>ip</sub>* *net*, *port*) *Combinators*  $\Rightarrow$  (*adr<sub>ip</sub>*, *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*  
 <*proof*>

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

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

An allow rule is disjunct with all rules appearing at the right of it. This invariant is

not necessary as it is a consequence from others, but facilitates some proofs.

**fun** (*sequential*) *wellformed-policy3*::((*adr<sub>ip</sub>* *net*,*port*) *Combinators*) *list*  $\Rightarrow$  *bool* **where**  
*wellformed-policy3* [] = *True*  
| *wellformed-policy3* ((*AllowPortFromTo* *a b p*)#*xs*) = (( $\forall$  *r*. *r*  $\in$  *set xs*  $\longrightarrow$   
 $\text{dom } (C \ r) \cap \text{dom } (C \ (\text{AllowPortFromTo } a \ b \ p)) = \{\}$ )  $\wedge$  *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<sub>ip</sub>* *net*, *port*) *Combinators*  $\Rightarrow$   
(*adr<sub>ip</sub>* *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<sub>ip</sub>* *net*, *port*) *Combinators*  $\Rightarrow$   
(*adr<sub>ip</sub>* *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'
  ⟨proof⟩
```

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

### TCP\_UDP\_IntegerPort

```
fun Cp :: (adripp net, protocol × port) Combinators ⇒
  (adripp, 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 :: (adripp net, protocol × port) Combinators ⇒
  (adripp, 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 :: ((adripp net, protocol × port) Combinators) list
⇒ bool where
```

```
  wellformed-policy3Pr [] = True
| wellformed-policy3Pr ((AllowPortFromTo a b p)#xs) = ((∀ r. r ∈ set xs →
  dom (Cp r) ∩ dom (Cp (AllowPortFromTo a b p)) = {}) ∧ wellformed-policy3Pr
xs)
| wellformed-policy3Pr (x#xs) = wellformed-policy3Pr xs
```

**definition**

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

**definition**  $normalizePr ::$ 

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

**definition**

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

**definition**

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

**definition**  $normalizePrQ ::$ 

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

**definition**

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

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

**lemma**  $normalizePr = normalizePr'$

$\langle proof \rangle$

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

**definition** *makeFUTPr* **where**

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

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

**lemmas** *PLemmas* = *C.simps* *Cp.simps* *dom-def* *PolicyCombinators.PolicyCombinators*

*PortCombinators.PortCombinatorsCore aux*  
    *ProtocolPortCombinators.ProtocolCombinatorsCore src-def dest-def*

*in-subnet-def*

*adr<sub>ipp</sub>Lemmas* *adr<sub>ipp</sub>Lemmas*

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

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

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

**lemma** *denyNMT*:  $deny\ all \neq \emptyset$   
 $\langle proof \rangle$

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

$\langle \text{proof} \rangle$

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

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

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

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

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

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

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

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

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

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

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

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

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

$\langle \text{proof} \rangle$

**lemma** *srcInNetListaux*[simp]:

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

**lemma** *destInNetListaux*[simp]:

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

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

**lemma** *nDComm*: **assumes** *ab*: *netsDistinct a b* **shows** *ba*: *netsDistinct b a*

$\langle \text{proof} \rangle$

**lemma** *tNDComm*:

**assumes** *abcd*: *twoNetsDistinct a b c d* **shows** *twoNetsDistinct c d a b*

$\langle \text{proof} \rangle$

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

**lemma** *aux12*:  $\llbracket a \in x; b \notin x \rrbracket \Longrightarrow a \neq b$   
 ⟨*proof*⟩

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

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

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

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

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

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

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

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

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

**lemma** *aux3*[*simp*]:  $\llbracket \text{singleCombinators } p; \text{singleCombinators } [a];$

$allNetsDistinct (a\#p) \Longrightarrow allNetsDistinct (a\#a\#p)$   
 ⟨proof⟩

**lemma**  $wp1aux1a[rule-format]: xs \neq [] \longrightarrow wellformed-policy1-strong (xs @ [x]) \longrightarrow wellformed-policy1-strong xs$   
 ⟨proof⟩

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

**lemma**  $wellformed-eq: DenyAll \in set p \longrightarrow ((wellformed-policy1 p) = (wellformed-policy1-strong p))$   
 ⟨proof⟩

**lemma**  $set-insort: set(insort x xs l) = insert x (set xs)$   
 ⟨proof⟩

**lemma**  $set-sort[simp]: set(sort xs l) = set xs$   
 ⟨proof⟩

**lemma**  $set-sortQ: set(qsort xs l) = set xs$   
 ⟨proof⟩

**lemma**  $aux79[rule-format]: y \in set (insort x a l) \longrightarrow y \neq x \longrightarrow y \in set a$   
 ⟨proof⟩

**lemma**  $aux80: \llbracket y \notin set p; y \neq x \rrbracket \Longrightarrow y \notin set (insort x (sort p l) l)$   
 ⟨proof⟩

**lemma**  $WP1Conca: DenyAll \notin set p \Longrightarrow wellformed-policy1 (a\#p)$   
 ⟨proof⟩

**lemma**  $saux[simp]: (insort DenyAll p l) = DenyAll\#p$   
 ⟨proof⟩

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

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

$\langle \text{proof} \rangle$

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

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

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

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

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

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

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

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

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

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

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

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

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

$(list2FWpolicy (xa \# a)) = (xa) \oplus (list2FWpolicy a)$

$\langle proof \rangle$

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

$\langle proof \rangle$

**lemma** *foo2*:  $a \notin set\ ps \implies$   
 $a \notin set\ ss \implies$   
 $set\ p = set\ s \implies$   
 $p = (a\#(ps)) \implies$   
 $s = (a\#ss) \implies$   
 $set\ (ps) = set\ (ss)$

$\langle proof \rangle$

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

$\langle proof \rangle$

**lemma** *auxx8*: *removeShadowRules1-alternative-rev*  $[x] = [x]$

$\langle proof \rangle$

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

$\langle proof \rangle$

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

$\langle proof \rangle$

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

$\langle proof \rangle$

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

$\langle proof \rangle$

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

$\langle proof \rangle$

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

$\langle proof \rangle$

**lemma** *domComm*:  $dom\ a \cap dom\ b = dom\ b \cap dom\ a$   
⟨proof⟩

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

**lemma** *wp1-aux1aa*[*rule-format*]:  $wellformed-policy1-strong\ p \longrightarrow DenyAll \in set\ p$   
⟨proof⟩

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

**lemma** *l2p-aux*[*rule-format*]:  $list \neq [] \longrightarrow$   
 $list2FWpolicy\ (a \# list) = a \oplus (list2FWpolicy\ list)$   
⟨proof⟩

**lemma** *l2p-aux2*[*rule-format*]:  $list = [] \implies list2FWpolicy\ (a \# list) = a$   
⟨proof⟩

**lemma** *aux7aa*:  
assumes  $1 : AllowPortFromTo\ a\ b\ po \in set\ p$   
and  $2 : allNetsDistinct\ ((AllowPortFromTo\ c\ d\ po) \# p)$   
and  $3 : a \neq c \vee b \neq d$   
shows  $twoNetsDistinct\ a\ b\ c\ d\ (is\ ?H)$   
⟨proof⟩

**lemma** *ANDConcEnd*:  $\llbracket allNetsDistinct\ (xs\ @\ [xa]);\ singleCombinators\ xs \rrbracket \implies$   
 $allNetsDistinct\ xs$   
⟨proof⟩

**lemma** *WP1ConcEnd*[*rule-format*]:  
 $wellformed-policy1\ (xs@[xa]) \longrightarrow wellformed-policy1\ xs$   
⟨proof⟩

**lemma** *NDComm*:  $netsDistinct\ a\ b = netsDistinct\ b\ a$   
⟨proof⟩

**lemma** *wellformed1-sorted*[*simp*]:  
assumes  $wp1 : wellformed-policy1\ p$   
shows  $wellformed-policy1\ (sort\ p\ l)$   
⟨proof⟩

**lemma** *wellformed1-sortedQ*[simp]:  
**assumes** *wp1*: *wellformed-policy1 p*  
**shows** *wellformed-policy1 (qsort p l)*  
⟨*proof*⟩

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

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

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

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

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

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

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

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

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

**lemma** *distinct-RS2*[rule-format,simp]: *distinct p*  $\longrightarrow$   
*distinct (removeShadowRules2 p)*  
⟨*proof*⟩

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

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

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

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

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

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

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

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

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

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

$\langle \text{proof} \rangle$

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

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

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

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

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

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

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

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

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

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

**lemma** *sorted-insorta*[rule-format]:  
**assumes**  $1 : \text{sorted } (\text{insort } x \ xs \ l) \ l$   
**and**  $2 : \text{all-in-list } (x\#xs) \ l$

**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>

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

**lemma** *sorted-insort*:  
 [[*all-in-list* (x#xs) l; *distinct*(x#xs); *singleCombinators* [x];  
*singleCombinators* xs]]  $\implies$   
 sorted (insort x xs l) l = sorted xs l  
 <proof>

**lemma** *distinct-insort*: *distinct* (insort x xs l) = (x  $\notin$  set xs  $\wedge$  *distinct* xs)  
 <proof>

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

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

**lemma** *smaller-sym*[rule-format]: *all-in-list* [a] l  $\longrightarrow$  *smaller* a a l  
 <proof>

**lemma** *SC-sublist*[rule-format]:  
*singleCombinators* xs  $\implies$  *singleCombinators* (qsort [y $\leftarrow$ xs. P y] l)  
 <proof>

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

**lemma** *SC-sublist2*[rule-format]:  
*singleCombinators* xs  $\longrightarrow$  *singleCombinators* ([y $\leftarrow$ xs. P y])  
 <proof>

**lemma** *all-in-list-sublist2*[rule-format]:

$singleCombinators\ xs \longrightarrow all-in-list\ xs\ l \longrightarrow all-in-list\ ([y \leftarrow xs.\ P\ y])\ l$   
 $\langle proof \rangle$

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

$all-in-list\ (xs)\ l \longrightarrow all-in-list\ (ys)\ l \longrightarrow all-in-list\ (xs\ @\ ys)\ l$   
 $\langle proof \rangle$

**lemma** *distinct-sortQ*[rule-format]:

$singleCombinators\ xs \longrightarrow all-in-list\ xs\ l \longrightarrow distinct\ xs \longrightarrow distinct\ (qsort\ xs\ l)$   
 $\langle proof \rangle$

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

$singleCombinators\ (xs) \longrightarrow singleCombinators\ (ys) \longrightarrow singleCombinators\ (xs\ @\ ys)$   
 $\langle proof \rangle$

**lemma** *sorted-append1*[rule-format]:

$all-in-list\ xs\ l \longrightarrow singleCombinators\ xs \longrightarrow$   
 $all-in-list\ ys\ l \longrightarrow singleCombinators\ ys \longrightarrow$   
 $(sorted\ (xs@ys)\ l \longrightarrow$   
 $(sorted\ xs\ l \ \&\ sorted\ ys\ l \ \&\ (\forall x \in set\ xs.\ \forall y \in set\ ys.\ smaller\ x\ y\ l)))$   
 $\langle proof \rangle$

**lemma** *sorted-append2*[rule-format]:

$all-in-list\ xs\ l \longrightarrow singleCombinators\ xs \longrightarrow$   
 $all-in-list\ ys\ l \longrightarrow singleCombinators\ ys \longrightarrow$   
 $(sorted\ xs\ l \ \&\ sorted\ ys\ l \ \&\ (\forall x \in set\ xs.\ \forall y \in set\ ys.\ smaller\ x\ y\ l)) \longrightarrow$   
 $(sorted\ (xs@ys)\ l)$   
 $\langle proof \rangle$

**lemma** *sorted-append*[rule-format]:

$all-in-list\ xs\ l \longrightarrow singleCombinators\ xs \longrightarrow$   
 $all-in-list\ ys\ l \longrightarrow singleCombinators\ ys \longrightarrow$   
 $(sorted\ (xs@ys)\ l) =$   
 $(sorted\ xs\ l \ \&\ sorted\ ys\ l \ \&\ (\forall x \in set\ xs.\ \forall y \in set\ ys.\ smaller\ x\ y\ l))$   
 $\langle proof \rangle$

**lemma** *sort-is-sortedQ*[rule-format]:

$all-in-list\ p\ l \longrightarrow singleCombinators\ p \longrightarrow sorted\ (qsort\ p\ l)\ l$   
 $\langle proof \rangle$

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

$\langle \text{proof} \rangle$

**lemma** *RS1n-assoc*:

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

**lemma** *wp1-tl[rule-format]*:

$$p \neq [] \wedge \text{wellformed-policy1 } p \longrightarrow \text{wellformed-policy1 } (tl \ p)$$

$\langle \text{proof} \rangle$

**lemma** *wp1-eq[rule-format]*:

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

$\langle \text{proof} \rangle$

**lemma** *wellformed1-alternative-sorted*:

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

$\langle \text{proof} \rangle$

**lemma** *wp1n-RS2[rule-format]*:

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

**lemma** *wp1-alternative-not-mt[simp]*:  $\text{wellformed-policy1-strong } p \implies p \neq []$

$\langle \text{proof} \rangle$

**lemma** *AIL1[rule-format,simp]*:  $\text{all-in-list } p \ l \longrightarrow$

$\text{all-in-list } (\text{removeShadowRules1 } p) \ l$

$\langle \text{proof} \rangle$

**lemma** *wp1ID*:  $\text{wellformed-policy1-strong } (\text{insertDeny } (\text{removeShadowRules1 } p))$

$\langle \text{proof} \rangle$

**lemma** *dRD[simp]*:  $\text{distinct } (\text{remdups } p)$

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

**lemma** *AILiD[rule-format,simp]*:  $\text{all-in-list } p \ l \longrightarrow \text{all-in-list } (\text{insertDeny } p) \ l$

$\langle \text{proof} \rangle$

**lemma** *SCrd[rule-format,simp]*:  $\text{singleCombinators } p \longrightarrow \text{singleCombinators}(\text{remdups } p)$

$\langle \text{proof} \rangle$

**lemma** *SCRiD[rule-format,simp]*:  $\text{singleCombinators } p \longrightarrow$

$\text{singleCombinators}(\text{insertDeny } p)$

$\langle \text{proof} \rangle$

**lemma** *WP1rd[rule-format,simp]*:

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

$\langle \text{proof} \rangle$

**lemma** *ANDrd[rule-format,simp]*:

*singleCombinators*  $p \longrightarrow$  *allNetsDistinct*  $p \longrightarrow$  *allNetsDistinct* (*remdups*  $p$ )  
*<proof>*

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

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

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

**lemma** *DAiniD*: *DenyAll*  $\in$  *set* (*insertDeny*  $p$ )  
*<proof>*

**lemma** *p2lNmt*: *policy2list*  $p \neq []$   
*<proof>*

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

**lemma** *SCConc*: *singleCombinators*  $x \Longrightarrow$  *singleCombinators*  $y \Longrightarrow$  *singleCombinators*  
 $(x@y)$   
*<proof>*

**lemma** *SCp2l*: *singleCombinators* (*policy2list*  $p$ )  
*<proof>*

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

**lemma** *soadisj*:  $x \in$  *subnetsOfAdr*  $a \Longrightarrow$   $y \in$  *subnetsOfAdr*  $a \Longrightarrow$   $\neg$  *netsDistinct*  $x \ y$   
*<proof>*

**lemma** *not-member*:  $\neg$  *member*  $a$  ( $x \oplus y$ )  $\Longrightarrow$   $\neg$  *member*  $a \ x$   
*<proof>*

**lemma** *soadisj2*:  $(\forall \ a \ x \ y. \ x \in$  *subnetsOfAdr*  $a \wedge \ y \in$  *subnetsOfAdr*  $a \longrightarrow$   $\neg$  *netsDistinct*  
 $x \ y)$   
*<proof>*

**lemma** *ndFalse1*:

$(\forall a b c d. (a,b) \in A \wedge (c,d) \in B \longrightarrow \text{netsDistinct } a \ c) \Longrightarrow$   
 $\exists (a, b) \in A. a \in \text{subnetsOfAdr } D \Longrightarrow$   
 $\exists (a, b) \in B. a \in \text{subnetsOfAdr } D \Longrightarrow \text{False}$   
*<proof>*

**lemma** *ndFalse2*:  $(\forall a b c d. (a,b) \in A \wedge (c,d) \in B \longrightarrow \text{netsDistinct } b \ d) \Longrightarrow$

$\exists (a, b) \in A. b \in \text{subnetsOfAdr } D \Longrightarrow$   
 $\exists (a, b) \in B. b \in \text{subnetsOfAdr } D \Longrightarrow \text{False}$   
*<proof>*

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

$\exists (a, b) \in A. a \in \text{subnetsOfAdr } (D::('a::\text{adr})) \wedge b \in \text{subnetsOfAdr } (F::'a) \Longrightarrow$   
 $\exists (a, b) \in B. a \in \text{subnetsOfAdr } D \wedge b \in \text{subnetsOfAdr } F$   
 $\Longrightarrow \text{False}$   
*<proof>*

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

*<proof>*

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

*<proof>*

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

*<proof>*

**lemma** *sortnMT*:  $p \neq [] \Longrightarrow \text{sort } p \ l \neq []$

*<proof>*

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

*<proof>*

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

*<proof>*

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

*<proof>*

**lemma** *sdnets2*:

$\exists a b. \text{sdnets } x = \{(a, b), (b, a)\} \Longrightarrow \neg \text{member } \text{DenyAll } x \Longrightarrow$   
 $\text{sdnets } x = \{(\text{first-srcNet } x, \text{first-destNet } x), (\text{first-destNet } x, \text{first-srcNet } x)\}$

$\langle proof \rangle$

**lemma** *alternativelistconc1*[rule-format]:

$a \in set (net-list-aux [x]) \longrightarrow a \in set (net-list-aux [x,y])$

$\langle proof \rangle$

**lemma** *alternativelistconc2*[rule-format]:

$a \in set (net-list-aux [x]) \longrightarrow a \in set (net-list-aux [y,x])$

$\langle proof \rangle$

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

$noDenyAll xs \longrightarrow s \in set xs \longrightarrow \neg member DenyAll s$

$\langle proof \rangle$

**lemma** *isInAlternativeList*:

$(aa \in set (net-list-aux [a]) \vee aa \in set (net-list-aux p)) \implies aa \in set (net-list-aux (a \# p))$

$\langle proof \rangle$

**lemma** *netlistaux*:

$x \in set (net-list-aux (a \# p)) \implies x \in set (net-list-aux ([a])) \vee x \in set (net-list-aux (p))$

$\langle proof \rangle$

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

$\neg member DenyAll a \longrightarrow first-destNet a \in set (net-list-aux (a \# p))$

$\langle proof \rangle$

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

$\neg member DenyAll a \longrightarrow first-srcNet a \in set (net-list-aux (a \# p))$

$\langle proof \rangle$

**lemma** *disjComm*:  $disjSD-2 a b \implies disjSD-2 b a$

$\langle proof \rangle$

**lemma** *disjSD2aux*:

$disjSD-2 a b \implies \neg member DenyAll a \implies \neg member DenyAll b \implies$

$disjSD-2 (DenyAllFromTo (first-srcNet a) (first-destNet a) \oplus$

$DenyAllFromTo (first-destNet a) (first-srcNet a) \oplus a)$

$b$

$\langle proof \rangle$

**lemma** *noDA1eq*[rule-format]:  $noDenyAll p \longrightarrow noDenyAll1 p$

$\langle proof \rangle$

**lemma** *noDA1C*[rule-format]:  $noDenyAll1 (a\#p) \longrightarrow noDenyAll1 p$   
 ⟨proof⟩

**lemma** *disjSD-2IDa*:

$disjSD-2 x y \implies$   
 $\neg member DenyAll x \implies$   
 $\neg member DenyAll y \implies$   
 $a = first-srcNet x \implies$   
 $b = first-destNet x \implies$   
 $disjSD-2 (DenyAllFromTo a b \oplus DenyAllFromTo b a \oplus x) y$   
 ⟨proof⟩

**lemma** *noDAID*[rule-format]:  $noDenyAll p \longrightarrow noDenyAll (insertDenies p)$   
 ⟨proof⟩

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

$noDenyAll p \longrightarrow s \in set (insertDenies p) \longrightarrow$   
 $(\exists! a. s = (DenyAllFromTo (first-srcNet a) (first-destNet a)) \oplus$   
 $(DenyAllFromTo (first-destNet a) (first-srcNet a)) \oplus a \wedge a \in set p)$   
 ⟨proof⟩

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

**lemma** *id-aux2*:

$noDenyAll p \implies$   
 $\forall s. s \in set p \longrightarrow disjSD-2 a s \implies$   
 $\neg member DenyAll a \implies$   
 $DenyAllFromTo (first-srcNet s) (first-destNet s) \oplus$   
 $DenyAllFromTo (first-destNet s) (first-srcNet s) \oplus s \in set (insertDenies p) \implies$   
 $disjSD-2 a (DenyAllFromTo (first-srcNet s) (first-destNet s) \oplus$   
 $DenyAllFromTo (first-destNet s) (first-srcNet s) \oplus s)$   
 ⟨proof⟩

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

$noDenyAll p \implies \forall s. s \in set p \longrightarrow disjSD-2 a s \implies$   
 $s \in set (insertDenies p) \implies \neg member DenyAll a \implies$   
 $disjSD-2 a s$   
 ⟨proof⟩

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

*noDenyAll1*  $p \longrightarrow \text{separated } p \longrightarrow \text{separated } (\text{insertDenies } p)$   
 $\langle \text{proof} \rangle$

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

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

**lemma** *first-bothNetsd*:  $\neg \text{member DenyAll } x \implies \text{first-bothNet } x = \{\text{first-srcNet } x, \text{first-destNet } x\}$   
 $\langle \text{proof} \rangle$

**lemma** *bNaux*:  
 $\neg \text{member DenyAll } x \implies \neg \text{member DenyAll } y \implies$   
 $\text{first-bothNet } x = \text{first-bothNet } y \implies$   
 $\{\text{first-srcNet } x, \text{first-destNet } x\} = \{\text{first-srcNet } y, \text{first-destNet } y\}$   
 $\langle \text{proof} \rangle$

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

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

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

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

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

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

**lemma** *otnaux*:  
 $\llbracket \text{first-bothNet } x = \text{first-bothNet } y; \neg \text{member DenyAll } x; \neg \text{member DenyAll } y;$   
 $\text{onlyTwoNets } y; \text{onlyTwoNets } x \rrbracket \implies$   
 $\text{onlyTwoNets } (x \oplus y)$   
 $\langle \text{proof} \rangle$

**lemma** *OTNSepaux*:

$onlyTwoNets (a \oplus y) \wedge OnlyTwoNets z \longrightarrow OnlyTwoNets (separate (a \oplus y \# z)) \implies$   
 $\neg member DenyAll a \implies \neg member DenyAll y \implies$   
 $noDenyAll z \implies onlyTwoNets a \implies OnlyTwoNets (y \# z) \implies first-bothNet a =$   
 $first-bothNet y \implies$   
 $OnlyTwoNets (separate (a \oplus y \# z))$   
 ⟨proof⟩

**lemma** *OTNSEp*[rule-format]:  
 $noDenyAll1 p \longrightarrow OnlyTwoNets p \longrightarrow OnlyTwoNets (separate p)$   
 ⟨proof⟩

**lemma** *nda*[rule-format]:  
 $singleCombinators (a\#p) \longrightarrow noDenyAll p \longrightarrow noDenyAll1 (a \# p)$   
 ⟨proof⟩

**lemma** *nDAcharn*[rule-format]:  $noDenyAll p = (\forall r \in set p. \neg member DenyAll r)$   
 ⟨proof⟩

**lemma** *nDAeqSet*:  $set p = set s \implies noDenyAll p = noDenyAll s$   
 ⟨proof⟩

**lemma** *nDASCaux*[rule-format]:  
 $DenyAll \notin set p \longrightarrow singleCombinators p \longrightarrow r \in set p \longrightarrow \neg member DenyAll r$   
 ⟨proof⟩

**lemma** *nDASC*[rule-format]:  
 $wellformed-policy1 p \longrightarrow singleCombinators p \longrightarrow noDenyAll1 p$   
 ⟨proof⟩

**lemma** *noDAAll*[rule-format]:  $noDenyAll p = (\neg memberP DenyAll p)$   
 ⟨proof⟩

**lemma** *memberPsep*[symmetric]:  $memberP x p = memberP x (separate p)$   
 ⟨proof⟩

**lemma** *noDAsep*[rule-format]:  $noDenyAll p \implies noDenyAll (separate p)$   
 ⟨proof⟩

**lemma** *noDA1sep*[rule-format]:  $noDenyAll1 p \longrightarrow noDenyAll1 (separate p)$   
 ⟨proof⟩

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

$\langle proof \rangle$

**lemma** *isInAlternativeListb*:

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

$\langle proof \rangle$

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

$\langle proof \rangle$

**lemma** *netlistalternativeSeparateaux*:

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

$\langle proof \rangle$

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

$\langle proof \rangle$

**lemma** *ANDSepaux2*:

$\text{allNetsDistinct}(x\#y\#z) \implies \text{allNetsDistinct}(\text{separate}(y\#z)) \implies \text{allNetsDistinct}(x\#\text{separate}(y\#z))$

$\langle proof \rangle$

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

$\langle proof \rangle$

**lemma** *wp1-alternativesep[rule-format]*:

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

$\langle proof \rangle$

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

$\langle proof \rangle$

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

$\langle proof \rangle$

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

$\langle proof \rangle$

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

$\langle proof \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

**lemma** *smalleraux2*:

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

$\langle \text{proof} \rangle$

**lemma** *smalleraux2a*:

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

$\langle \text{proof} \rangle$

**lemma** *smalleraux2b*:

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

$\langle \text{proof} \rangle$

**lemma** *smalleraux2c*:

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

$\langle \text{proof} \rangle$

**lemma** *smalleraux3*:

**assumes**  $x \in \text{set } l$  **and**  $y \in \text{set } l$  **and**  $x \neq y$  **and**  $x = \text{bothNet } a$  **and**  $y = \text{bothNet } b$   
**and**  $\text{smaller } a b l$  **and**  $\text{singleCombinators } [a]$  **and**  $\text{singleCombinators } [b]$   
**shows**  $\neg \text{smaller } b a l$

$\langle \text{proof} \rangle$

**lemma** *smalleraux3a*:

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

**lemma** *posaux6*[rule-format]:

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

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

$x \neq \text{DenyAll} \implies r \neq \text{DenyAll} \implies$   
 $\text{singleCombinators } [x] \implies \text{singleCombinators } [y] \implies \text{singleCombinators } [r] \implies$   
 $\neg \text{smaller } y \text{ } x \text{ } l \implies \text{smaller } x \text{ } y \text{ } l \implies \text{smaller } x \text{ } r \text{ } l \implies \text{smaller } y \text{ } r \text{ } l \implies$   
 $\text{in-list } x \text{ } l \implies \text{in-list } y \text{ } l \implies \text{in-list } r \text{ } l \implies \neg \text{smaller } r \text{ } x \text{ } l$   
 ⟨proof⟩

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

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

**lemma** *NCSaux1*[rule-format]:

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

**lemma** *posaux3*[rule-format]:  $a \in \text{set } l \longrightarrow b \in \text{set } l \longrightarrow a \neq b \longrightarrow \text{position } a \text{ } l \neq$   
 $\text{position } b \text{ } l$   
 ⟨proof⟩

**lemma** *posaux4*[rule-format]:

$\text{singleCombinators } [a] \longrightarrow a \neq \text{DenyAll} \longrightarrow b \neq \text{DenyAll} \longrightarrow \text{in-list } a \text{ } l \longrightarrow \text{in-list } b \text{ } l$   
 $\longrightarrow$   
 $\text{smaller } a \text{ } b \text{ } l \longrightarrow x = (\text{bothNet } a) \longrightarrow y = (\text{bothNet } b) \longrightarrow$   
 $\text{position } x \text{ } l \leq \text{position } y \text{ } l$   
 ⟨proof⟩

**lemma** *NCSaux2*[rule-format]:

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

**lemma** *NCSaux3*[rule-format]:

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

**lemma** *NCSaux4*[*rule-format*]:

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

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

$noDenyAll1\ p \longrightarrow all-in-list\ p\ l \longrightarrow singleCombinators\ p \longrightarrow sorted\ p\ l \longrightarrow NetsCol-$   
 $lected\ p$   
 ⟨*proof*⟩

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

⟨*proof*⟩

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

⟨*proof*⟩

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

⟨*proof*⟩

**lemma** *NetsCollectedSepauxa*:

$\{b, c\} \neq firstList\ z \implies noDenyAll1\ z \implies \forall a \in set\ z. \{b, c\} \neq first-bothNet\ a \implies NetsCol-$   
 $lected\ z \implies$

$NetsCollected\ (separate\ z) \implies \{b, c\} \neq firstList\ (separate\ z) \implies a \in set\ (separate$   
 $z) \implies$

$\{b, c\} \neq first-bothNet\ a$

⟨*proof*⟩

**lemma** *NetsCollectedSepaux*:

$first-bothNet\ (x::('a, 'b)Combinators) \neq first-bothNet\ y \implies \neg member\ DenyAll\ y \wedge$   
 $noDenyAll\ z \implies$

$(\forall a \in set\ z. first-bothNet\ x \neq first-bothNet\ a) \wedge NetsCollected\ (y \# z) \implies$

$NetsCollected\ (separate\ (y \# z)) \implies first-bothNet\ x \neq firstList\ (separate\ (y \# z))$

$\implies$

$a \in set\ (separate\ (y \# z)) \implies$

$first-bothNet\ (x::('a, 'b)Combinators) \neq first-bothNet\ (a::('a, 'b)Combinators)$

⟨*proof*⟩

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

*noDenyAll1*  $p \longrightarrow \text{NetsCollected } p \longrightarrow \text{NetsCollected (separate } p)$   
 ⟨proof⟩

**lemma** *OTNaux*:

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

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

**lemma** *first-bothNet-charn*[rule-format]:

$\neg \text{member DenyAll } a \longrightarrow \text{first-bothNet } a = \{\text{first-srcNet } a, \text{first-destNet } a\}$   
 ⟨proof⟩

**lemma** *sdnets-noteq*:

*onlyTwoNets*  $a \implies \text{onlyTwoNets } aa \implies \text{first-bothNet } a \neq \text{first-bothNet } aa \implies$   
 $\neg \text{member DenyAll } a \implies \neg \text{member DenyAll } aa \implies \text{sdnets } a \neq \text{sdnets } aa$   
 ⟨proof⟩

**lemma** *fbn-noteq*:

*onlyTwoNets*  $a \implies \text{onlyTwoNets } aa \implies \text{first-bothNet } a \neq \text{first-bothNet } aa \implies$   
 $\neg \text{member DenyAll } a \implies \neg \text{member DenyAll } aa \implies \text{allNetsDistinct } [a, aa] \implies$   
 $\text{first-srcNet } a \neq \text{first-srcNet } aa \vee \text{first-srcNet } a \neq \text{first-destNet } aa \vee$   
 $\text{first-destNet } a \neq \text{first-srcNet } aa \vee \text{first-destNet } a \neq \text{first-destNet } aa$   
 ⟨proof⟩

**lemma** *NCisSD2aux*:

**assumes** 1: *onlyTwoNets*  $a$  **and** 2 : *onlyTwoNets*  $aa$  **and** 3 : *first-bothNet*  $a \neq$   
*first-bothNet*  $aa$   
**and** 4:  $\neg \text{member DenyAll } a$  **and** 5:  $\neg \text{member DenyAll } aa$  **and** 6: *allNetsDistinct*  
 $[a, aa]$   
**shows** *disjSD-2*  $a$   $aa$   
 ⟨proof⟩

**lemma** *ANDaux3*[rule-format]:

$y \in \text{set } xs \longrightarrow a \in \text{set (net-list-aux } [y]) \longrightarrow a \in \text{set (net-list-aux } xs)$   
 ⟨proof⟩

**lemma** *ANDaux2*:

$allNetsDistinct (x \# xs) \implies y \in set\ xs \implies allNetsDistinct [x,y]$   
 ⟨proof⟩

**lemma** *NCisSD2*[rule-format]:  
 $\neg member\ DenyAll\ a \implies OnlyTwoNets\ (a\#\!p) \implies$   
 $NetsCollected2\ (a\ \#\ p) \implies NetsCollected\ (a\#\!p) \implies$   
 $noDenyAll\ (p) \implies allNetsDistinct\ (a\ \#\ p) \implies s \in set\ p \implies$   
 $disjSD-2\ a\ s$   
 ⟨proof⟩

**lemma** *separatedNC*[rule-format]:  
 $OnlyTwoNets\ p \longrightarrow NetsCollected2\ p \longrightarrow NetsCollected\ p \longrightarrow noDenyAll1\ p \longrightarrow$   
 $allNetsDistinct\ p \longrightarrow separated\ p$   
 ⟨proof⟩

**lemma** *separatedNC'*[rule-format]:  
 $OnlyTwoNets\ p \longrightarrow NetsCollected2\ p \longrightarrow NetsCollected\ p \longrightarrow noDenyAll1\ p \longrightarrow$   
 $allNetsDistinct\ p \longrightarrow separated\ p$   
 ⟨proof⟩

**lemma** *NC2Sep*[rule-format]:  $noDenyAll1\ p \longrightarrow NetsCollected2\ (separate\ p)$   
 ⟨proof⟩

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

**lemma** *rADnMT*[rule-format]:  $p \neq [] \longrightarrow removeAllDuplicates\ p \neq []$   
 ⟨proof⟩

**lemma** *remDupsNMT*[rule-format]:  $p \neq [] \longrightarrow remdups\ p \neq []$   
 ⟨proof⟩

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

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

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

**lemma** *sets-distinct6*:  $(m::int) < n \implies \{(a,b). a = n\} \neq \{(a,b). a = m\}$   
 ⟨proof⟩  
**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*:  $C((A \oplus B) \oplus D) = C(A \oplus (B \oplus D))$   
 ⟨proof⟩

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

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

**lemma** *Cdom2*:  $x \in dom(C\ b) \implies C\ (a \oplus b)\ x = (C\ b)\ x$   
 ⟨proof⟩

**lemma** *wp2Conc[rule-format]*:  $wellformed-policy2\ (x\ \#\ xs) \implies wellformed-policy2\ xs$   
 ⟨proof⟩

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

**lemma** *DAimplieMR[rule-format]*:  $DenyAll \in set\ p \implies applied-rule-rev\ C\ x\ p \neq None$   
 ⟨proof⟩

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

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

$\langle \text{proof} \rangle$

**lemma** *MRList3*:

$x \notin \text{dom} (C \ x a) \implies \text{applied-rule-rev } C \ x \ (a \ @ \ b \ \# \ xs \ @ \ [xa]) = \text{applied-rule-rev } C \ x \ (a \ @ \ b \ \# \ xs)$

$\langle \text{proof} \rangle$

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

$C \ a \ x = \text{Some } y \longrightarrow C \ (\text{list2FWpolicy } (xs \ @ \ [a])) \ x = \text{Some } y$   
(**is** *?P xs*)

$\langle \text{proof} \rangle$

**lemma** *CConcStartaux*:  $C \ a \ x = \text{None} \implies (C \ aa \ ++ \ C \ a) \ x = C \ aa \ x$

$\langle \text{proof} \rangle$

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

$xs \neq [] \longrightarrow C \ a \ x = \text{None} \longrightarrow C \ (\text{list2FWpolicy } (xs \ @ \ [a])) \ x = C \ (\text{list2FWpolicy } xs) \ x$

$\langle \text{proof} \rangle$

**lemma** *mrNnt*[*simp*]:  $\text{applied-rule-rev } C \ x \ p = \text{Some } a \implies p \neq []$

$\langle \text{proof} \rangle$

**lemma** *mr-is-C*[*rule-format*]:

$\text{applied-rule-rev } C \ x \ p = \text{Some } a \longrightarrow C \ (\text{list2FWpolicy } (p)) \ x = C \ a \ x$

$\langle \text{proof} \rangle$

**lemma** *CConcStart2*:

$p \neq [] \implies x \notin \text{dom} (C \ a) \implies C \ (\text{list2FWpolicy } (p \ @ \ [a])) \ x = C \ (\text{list2FWpolicy } p) \ x$

$\langle \text{proof} \rangle$

**lemma** *CConcEnd1*:

$q \ @ \ p \neq [] \implies x \notin \text{dom} (C \ a) \implies C \ (\text{list2FWpolicy } (q \ @ \ p \ @ \ [a])) \ x = C \ (\text{list2FWpolicy } (q \ @ \ p)) \ x$

$\langle \text{proof} \rangle$

**lemma** *CConcEnd2*[*rule-format*]:

$x \in \text{dom} (C \ a) \longrightarrow C \ (\text{list2FWpolicy } (xs \ @ \ [a])) \ x = C \ a \ x$  (**is** *?P xs*)

$\langle \text{proof} \rangle$

**lemma** *bar3*:

$x \in \text{dom} (C \ (\text{list2FWpolicy } (xs \ @ \ [xa]))) \implies x \in \text{dom} (C \ (\text{list2FWpolicy } xs)) \vee x \in$

$dom (C\ xa)$   
 $\langle proof \rangle$

**lemma** *CeqEnd*[*rule-format,simp*]:

$a \neq [] \longrightarrow x \in dom (C (list2FWpolicy\ a)) \longrightarrow C (list2FWpolicy(b@a))\ x = (C (list2FWpolicy\ a))\ x$   
 $\langle proof \rangle$

**lemma** *CConcStartA*[*rule-format,simp*]:

$x \in dom (C\ a) \longrightarrow x \in dom (C (list2FWpolicy (a \# b))) (is\ ?P\ b)$   
 $\langle proof \rangle$

**lemma** *domConc*:

$x \in dom (C (list2FWpolicy\ b)) \Longrightarrow b \neq [] \Longrightarrow x \in dom (C (list2FWpolicy (a @ b)))$   
 $\langle proof \rangle$

**lemma** *CeqStart*[*rule-format,simp*]:

$x \notin dom(C(list2FWpolicy\ a)) \longrightarrow a \neq [] \longrightarrow b \neq [] \longrightarrow C(list2FWpolicy(b@a))\ x = C(list2FWpolicy\ b)\ x$   
 $\langle proof \rangle$

**lemma** *C-eq-if-mr-eq2*:

$applied-rule-rev\ C\ x\ a = [r] \Longrightarrow$   
 $applied-rule-rev\ C\ x\ b = [r] \Longrightarrow a \neq [] \Longrightarrow b \neq [] \Longrightarrow$   
 $C (list2FWpolicy\ a)\ x = C (list2FWpolicy\ b)\ x$   
 $\langle proof \rangle$

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

$p \neq [] \longrightarrow applied-rule-rev\ C\ x\ p = None \longrightarrow C (list2FWpolicy\ p)\ x = None$   
 $\langle proof \rangle$

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

$applied-rule-rev\ C\ x\ b = applied-rule-rev\ C\ x\ a \Longrightarrow a \neq [] \Longrightarrow b \neq [] \Longrightarrow$   
 $C (list2FWpolicy\ a)\ x = C (list2FWpolicy\ b)\ x$   
 $\langle proof \rangle$

**lemma** *notmatching-notdom*:  $applied-rule-rev\ C\ x\ (p@[a]) \neq Some\ a \Longrightarrow x \notin dom (C\ a)$

$\langle proof \rangle$

**lemma** *foo3a*[*rule-format*]:

$applied-rule-rev\ C\ x\ (a@[b]@c) = Some\ b \longrightarrow r \in set\ c \longrightarrow b \notin set\ c \longrightarrow x \notin dom (C\ r)$   
 $\langle proof \rangle$

**lemma foo3D:**

$wellformed-policy1\ p \implies p = DenyAll \# ps \implies$   
 $applied-rule-rev\ C\ x\ p = \lfloor DenyAll \rfloor \implies r \in set\ ps \implies x \notin dom\ (C\ r)$   
 $\langle proof \rangle$

**lemma foo4[rule-format]:**

$set\ p = set\ s \wedge (\forall\ r.\ r \in set\ p \longrightarrow x \notin dom\ (C\ r)) \longrightarrow (\forall\ r.\ r \in set\ s \longrightarrow x \notin dom\ (C\ r))$   
 $\langle proof \rangle$

**lemma foo5b[rule-format]:**

$x \in dom\ (C\ b) \longrightarrow (\forall\ r.\ r \in set\ c \longrightarrow x \notin dom\ (C\ r)) \longrightarrow applied-rule-rev\ C\ x\ (b\#c)$   
 $=\ Some\ b$   
 $\langle proof \rangle$

**lemma mr-first:**

$x \in dom\ (C\ b) \implies \forall\ r.\ r \in set\ c \longrightarrow x \notin dom\ (C\ r) \implies s = b \# c \implies applied-rule-rev\ C\ x\ s = \lfloor b \rfloor$   
 $\langle proof \rangle$

**lemma mr-charn[rule-format]:**

$a \in set\ p \longrightarrow (x \in dom\ (C\ a)) \longrightarrow (\forall\ r.\ r \in set\ p \wedge x \in dom\ (C\ r) \longrightarrow r = a)$   
 $\longrightarrow$   
 $applied-rule-rev\ C\ x\ p = Some\ a$   
 $\langle proof \rangle$

**lemma foo8:**

$\forall\ r.\ r \in set\ p \wedge x \in dom\ (C\ r) \longrightarrow r = a \implies set\ p = set\ s \implies$   
 $\forall\ r.\ r \in set\ s \wedge x \in dom\ (C\ r) \longrightarrow r = a$   
 $\langle proof \rangle$

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

$applied-rule-rev\ C\ x\ (b\ \# \ p) = Some\ a \longrightarrow a \neq b \longrightarrow applied-rule-rev\ C\ x\ p = Some\ a$   
 $\langle proof \rangle$

**lemma wp3tl[rule-format]:**  $wellformed-policy3\ p \longrightarrow wellformed-policy3\ (tl\ p)$

$\langle proof \rangle$

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

$\langle proof \rangle$

**lemma** *foo98*[*rule-format*]:

*applied-rule-rev C x (aa # p) = Some a*  $\longrightarrow$  *x*  $\in$  *dom (C r)*  $\longrightarrow$  *r*  $\in$  *set p*  $\longrightarrow$  *a*  $\in$  *set p*  
*<proof>*

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

*<proof>*

**lemma** *DAAux*[*simp*]: *x*  $\in$  *dom (C DenyAll)*

*<proof>*

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

*<proof>*

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

*<proof>*

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

*<proof>*

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

*<proof>*

**lemma** *wp3EndMT*[*rule-format*]:

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

*dom (C (AllowPortFromTo a b po))*  $\cap$  *dom (C xs)* =  $\{\}$

*<proof>*

**lemma** *foo29*:  $\llbracket \text{dom (C a)} \neq \{\}; \text{dom (C a)} \cap \text{dom (C b)} = \{\} \rrbracket \implies a \neq b$  *<proof>*

**lemma** *foo28*:

*AllowPortFromTo a b po*  $\in$  *set p*  $\implies$  *dom (C (AllowPortFromTo a b po))*  $\neq$   $\{\}$   $\implies$

*wellformed-policy3 (p @ [x])*  $\implies x \neq$  *AllowPortFromTo a b po*

*<proof>*

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

**lemma** *allow-deny-dom*[*simp*]:

*dom (C (AllowPortFromTo a b po))*  $\subseteq$  *dom (C (DenyAllFromTo a b))*

*<proof>*

**lemma** *DenyAllowDisj*:

$$\begin{aligned} & \text{dom } (C \text{ (AllowPortFromTo } a \ b \ p)) \neq \{\} \implies \\ & \text{dom } (C \text{ (DenyAllFromTo } a \ b)) \cap \text{dom } (C \text{ (AllowPortFromTo } a \ b \ p)) \neq \{\} \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *foo31*:

$$\begin{aligned} & \forall r. r \in \text{set } p \wedge x \in \text{dom } (C \ r) \longrightarrow \\ & \quad r = \text{AllowPortFromTo } a \ b \ po \vee r = \text{DenyAllFromTo } a \ b \vee r = \text{DenyAll} \implies \\ & \quad \text{set } p = \text{set } s \implies \\ & \forall r. r \in \text{set } s \wedge x \in \text{dom } (C \ r) \longrightarrow r = \text{AllowPortFromTo } a \ b \ po \vee r = \text{DenyAllFromTo } \\ & a \ b \vee r = \text{DenyAll} \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *wp1-auxa*:

$$\begin{aligned} & \text{wellformed-policy1-strong } p \implies (\exists r. \text{applied-rule-rev } C \ x \ p = \text{Some } r) \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *deny-dom[simp]*:

$$\begin{aligned} & \text{twoNetsDistinct } a \ b \ c \ d \implies \text{dom } (C \text{ (DenyAllFromTo } a \ b)) \cap \text{dom } (C \text{ (DenyAllFromTo } \\ & c \ d)) = \{\} \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *domTrans*:  $\text{dom } a \subseteq \text{dom } b \implies \text{dom } b \cap \text{dom } c = \{\} \implies \text{dom } a \cap \text{dom } c = \{\}$   $\langle \text{proof} \rangle$

**lemma** *DomInterAllowsMT*:

$$\begin{aligned} & \text{twoNetsDistinct } a \ b \ c \ d \implies \\ & \text{dom } (C \text{ (AllowPortFromTo } a \ b \ p)) \cap \text{dom } (C \text{ (AllowPortFromTo } c \ d \ po)) = \{\} \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *DomInterAllowsMT-Ports*:

$$\begin{aligned} & p \neq po \implies \text{dom } (C \text{ (AllowPortFromTo } a \ b \ p)) \cap \text{dom } (C \text{ (AllowPortFromTo } c \ d \ po)) \\ & = \{\} \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *wellformed-policy3-charn[rule-format]*:

$$\begin{aligned} & \text{singleCombinators } p \longrightarrow \text{distinct } p \longrightarrow \text{allNetsDistinct } p \longrightarrow \\ & \text{wellformed-policy1 } p \longrightarrow \text{wellformed-policy2 } p \longrightarrow \text{wellformed-policy3 } p \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *DistinctNetsDenyAllow*:

$$\text{DenyAllFromTo } b \ c \in \text{set } p \implies$$

$AllowPortFromTo\ a\ d\ po \in set\ p \implies$   
 $allNetsDistinct\ p \implies dom\ (C\ (DenyAllFromTo\ b\ c)) \cap dom\ (C\ (AllowPortFromTo\ a\ d\ po)) \neq \{\}$   
 $b = a \wedge c = d$   
 <proof>

**lemma** *DistinctNetsAllowAllow*:

$AllowPortFromTo\ b\ c\ poo \in set\ p \implies$   
 $AllowPortFromTo\ a\ d\ po \in set\ p \implies$   
 $allNetsDistinct\ p \implies$   
 $dom\ (C\ (AllowPortFromTo\ b\ c\ poo)) \cap dom\ (C\ (AllowPortFromTo\ a\ d\ po)) \neq \{\}$   
 $\implies$   
 $b = a \wedge c = d \wedge poo = po$   
 <proof>

**lemma** *WP2RS2[simp]*:

$singleCombinators\ p \implies distinct\ p \implies allNetsDistinct\ p \implies$   
 $wellformed-policy2\ (removeShadowRules2\ p)$   
 <proof>

**lemma** *AD-aux*:

$AllowPortFromTo\ a\ b\ po \in set\ p \implies DenyAllFromTo\ c\ d \in set\ p \implies$   
 $allNetsDistinct\ p \implies singleCombinators\ p \implies a \neq c \vee b \neq d \implies$   
 $dom\ (C\ (AllowPortFromTo\ a\ b\ po)) \cap dom\ (C\ (DenyAllFromTo\ c\ d)) = \{\}$   
 <proof>

**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>

**lemma** *wellformed2-sorted[simp]*:

$all-in-list\ p\ l \implies distinct\ p \implies allNetsDistinct\ p \implies$   
 $singleCombinators\ p \implies wellformed-policy2\ (sort\ p\ l)$   
 <proof>

**lemma** *wellformed2-sortedQ[simp]*:  $\llbracket all-in-list\ p\ l; distinct\ p; allNetsDistinct\ p; singleCombinators\ p \rrbracket \implies wellformed-policy2\ (qsort\ p\ l)$   
 <proof>

**lemma** *C-DenyAll[simp]*:  $C\ (list2FWpolicy\ (xs\ @\ [DenyAll]))\ x = Some\ (deny\ ())$   
 <proof>

**lemma** *C-eq-RS1n*:

$C(list2FWpolicy\ (removeShadowRules1-alternative\ p)) = C(list2FWpolicy\ p)$

$\langle \text{proof} \rangle$

**lemma** *C-eq-RS1*[simp]:

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

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

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

**lemma** *EX-MR* :

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

**lemma** *mr-not-DA*:

$\text{wellformed-policy1-strong } s \implies$   
 $\text{applied-rule-rev } C \ x \ p = [\text{DenyAllFromTo } a \ ab] \implies \text{set } p = \text{set } s \implies$   
 $\text{applied-rule-rev } C \ x \ s \neq [\text{DenyAll}]$   
 $\langle \text{proof} \rangle$

**lemma** *domsMT-notND-DD*:

$\text{dom } (C (\text{DenyAllFromTo } a \ b)) \cap \text{dom } (C (\text{DenyAllFromTo } c \ d)) \neq \{\} \implies \neg \text{netsDistinct } a \ c$   
 $\langle \text{proof} \rangle$

**lemma** *domsMT-notND-DD2*:

$\text{dom } (C (\text{DenyAllFromTo } a \ b)) \cap \text{dom } (C (\text{DenyAllFromTo } c \ d)) \neq \{\} \implies \neg \text{netsDistinct } b \ d$   
 $\langle \text{proof} \rangle$

**lemma** *domsMT-notND-DD3*:

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

**lemma** *domsMT-notND-DD4*:

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

**lemma** *NetsEq-if-sameP-DD*:

$\text{allNetsDistinct } p \implies u \in \text{set } p \implies v \in \text{set } p \implies u = \text{DenyAllFromTo } a \ b \implies$

$v = \text{DenyAllFromTo } c \ d \implies x \in \text{dom } (C \ u) \implies x \in \text{dom } (C \ v) \implies a = c \wedge b = d$   
 ⟨proof⟩

**lemma** *rule-charn1*:

**assumes** *aND*: *allNetsDistinct* *p*  
**and** *mr-is-allow*: *applied-rule-rev* *C* *x* *p* = *Some* (*AllowPortFromTo* *a* *b* *po*)  
**and** *SC*: *singleCombinators* *p*  
**and** *inp*: *r* ∈ *set* *p*  
**and** *inDom*: *x* ∈ *dom* (*C* *r*)  
**shows** (*r* = *AllowPortFromTo* *a* *b* *po* ∨ *r* = *DenyAllFromTo* *a* *b* ∨ *r* = *DenyAll*)  
 ⟨proof⟩

**lemma** *none-MT-rulessubset*[*rule-format*]:

*none-MT-rules* *C* *a*  $\longrightarrow$  *set* *b*  $\subseteq$  *set* *a*  $\longrightarrow$  *none-MT-rules* *C* *b*  
 ⟨proof⟩

**lemma** *nMTSort*: *none-MT-rules* *C* *p*  $\implies$  *none-MT-rules* *C* (*sort* *p* *l*)

⟨proof⟩

**lemma** *nMTSortQ*: *none-MT-rules* *C* *p*  $\implies$  *none-MT-rules* *C* (*qsort* *p* *l*)

⟨proof⟩

**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*  
 ⟨proof⟩

**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*  
 ⟨proof⟩

**lemma** *rule-charn2*:

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

**lemma rule-charn3:**

$wellformed-policy1\ p \implies allNetsDistinct\ p \implies singleCombinators\ p \implies$   
 $wellformed-policy3\ p \implies applied-rule-rev\ C\ x\ p = [DenyAllFromTo\ c\ d] \implies$   
 $AllowPortFromTo\ a\ b\ po \in set\ p \implies x \notin dom\ (C\ (AllowPortFromTo\ a\ b\ po))$   
 $\langle proof \rangle$

**lemma rule-charn4:**

**assumes**  $wp1: wellformed-policy1\ p$   
**and**  $aND: allNetsDistinct\ p$   
**and**  $SC: singleCombinators\ p$   
**and**  $wp3: wellformed-policy3\ p$   
**and**  $DA: DenyAll \notin set\ p$   
**and**  $mr: applied-rule-rev\ C\ x\ p = Some\ (DenyAllFromTo\ a\ b)$   
**and**  $rinp: r \in set\ p$   
**and**  $xindom: x \in dom\ (C\ r)$   
**shows**  $r = DenyAllFromTo\ a\ b$   
 $\langle proof \rangle$

**lemma foo31a:**

$\forall r. r \in set\ p \wedge x \in dom\ (C\ r) \longrightarrow r = AllowPortFromTo\ a\ b\ po \vee r = DenyAllFromTo\ a\ b \vee r = DenyAll \implies$   
 $set\ p = set\ s \implies r \in set\ s \implies x \in dom\ (C\ r) \implies$   
 $r = AllowPortFromTo\ a\ b\ po \vee r = DenyAllFromTo\ a\ b \vee r = DenyAll$   
 $\langle proof \rangle$

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

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

**lemma mrDA-tl:**

**assumes**  $mr-DA: applied-rule-rev\ C\ x\ p = Some\ DenyAll$   
**and**  $wp1n: wellformed-policy1-strong\ p$   
**shows**  $applied-rule-rev\ C\ x\ (tl\ p) = None$   
 $\langle proof \rangle$

**lemma rule-charnDAFT:**

$wellformed-policy1-strong\ p \implies allNetsDistinct\ p \implies singleCombinators\ p \implies$   
 $wellformed-policy3\ p \implies applied-rule-rev\ C\ x\ p = [DenyAllFromTo\ a\ b] \implies r \in set$   
 $(tl\ p) \implies$   
 $x \in dom\ (C\ r) \implies r = DenyAllFromTo\ a\ b$   
 $\langle proof \rangle$

**lemma mrDenyAll-is-unique:**

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

**theorem** *C-eq-Sets-mr*:

**assumes** *sets-eq*:  $\text{set } p = \text{set } s$   
**and** *SC*:  $\text{singleCombinators } p$   
**and** *wp1-p*:  $\text{wellformed-policy1-strong } p$   
**and** *wp1-s*:  $\text{wellformed-policy1-strong } s$   
**and** *wp3-p*:  $\text{wellformed-policy3 } p$   
**and** *wp3-s*:  $\text{wellformed-policy3 } s$   
**and** *aND*:  $\text{allNetsDistinct } p$   
**shows**  $\text{applied-rule-rev } C \ x \ p = \text{applied-rule-rev } C \ x \ s$   
 <proof>

**lemma** *C-eq-Sets*:

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

**lemma** *C-eq-sorted*:

$\text{distinct } p \implies \text{all-in-list } p \ l \implies \text{singleCombinators } p \implies \text{wellformed-policy1-strong } p \implies$   
 $\text{wellformed-policy3 } p \implies \text{allNetsDistinct } p \implies$   
 $C \ (\text{list2FWpolicy } (\text{FWNormalisationCore.sort } p \ l)) = C \ (\text{list2FWpolicy } p)$   
 <proof>

**lemma** *C-eq-sortedQ*:

$\text{distinct } p \implies \text{all-in-list } p \ l \implies \text{singleCombinators } p \implies \text{wellformed-policy1-strong } p \implies$   
 $\text{wellformed-policy3 } p \implies \text{allNetsDistinct } p \implies$   
 $C \ (\text{list2FWpolicy } (\text{qsort } p \ l)) = C \ (\text{list2FWpolicy } p)$   
 <proof>

**lemma** *C-eq-RS2-mr*:  $\text{applied-rule-rev } C \ x \ (\text{removeShadowRules2 } p) = \text{applied-rule-rev } C \ x \ p$   
 <proof>

**lemma** *C-eq-None[rule-format]*:

$p \neq [] \implies \text{applied-rule-rev } C \ x \ p = \text{None} \implies C \ (\text{list2FWpolicy } p) \ x = \text{None}$   
 <proof>

**lemma** *C-eq-None2*:

$a \neq [] \implies b \neq [] \implies \text{applied-rule-rev } C \ x \ a = \perp \implies \text{applied-rule-rev } C \ x \ b = \perp \implies$   
 $C \ (\text{list2FWpolicy } a) \ x = C \ (\text{list2FWpolicy } b) \ x$   
 $\langle \text{proof} \rangle$

**lemma** *C-eq-RS2*:

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

**lemma** *none-MT-rulesRS2*:

$\text{none-MT-rules } C \ p \implies \text{none-MT-rules } C \ (\text{removeShadowRules2 } p)$   
 $\langle \text{proof} \rangle$

**lemma** *CconcNone*:

$\text{dom } (C \ a) = \{\} \implies p \neq [] \implies C \ (\text{list2FWpolicy } (a \ # \ p)) \ x = C \ (\text{list2FWpolicy } p) \ x$   
 $\langle \text{proof} \rangle$

**lemma** *none-MT-rulesrd*[*rule-format*]:

$\text{none-MT-rules } C \ p \longrightarrow \text{none-MT-rules } C \ (\text{remdups } p)$   
 $\langle \text{proof} \rangle$

**lemma** *DARS3*[*rule-format*]:

$\text{DenyAll} \notin \text{set } p \longrightarrow \text{DenyAll} \notin \text{set } (\text{rm-MT-rules } C \ p)$   
 $\langle \text{proof} \rangle$

**lemma** *DAnMT*:  $\text{dom } (C \ \text{DenyAll}) \neq \{\}$

$\langle \text{proof} \rangle$

**lemma** *DAnMT2*:  $C \ \text{DenyAll} \neq \text{Map.empty}$

$\langle \text{proof} \rangle$

**lemma** *wp1n-RS3*[*rule-format,simp*]:

$\text{wellformed-policy1-strong } p \longrightarrow \text{wellformed-policy1-strong } (\text{rm-MT-rules } C \ p)$   
 $\langle \text{proof} \rangle$

**lemma** *AILRS3*[*rule-format,simp*]:

$\text{all-in-list } p \ l \longrightarrow \text{all-in-list } (\text{rm-MT-rules } C \ p) \ l$   
 $\langle \text{proof} \rangle$

**lemma** *SCRS3*[*rule-format,simp*]:

$\text{singleCombinators } p \longrightarrow \text{singleCombinators}(\text{rm-MT-rules } C \ p)$   
 $\langle \text{proof} \rangle$

**lemma** *RS3subset*:  $set (rm\text{-}MT\text{-}rules\ C\ p) \subseteq set\ p$   
 ⟨proof⟩

**lemma** *ANDRS3[simp]*:  
 $singleCombinators\ p \implies allNetsDistinct\ p \implies allNetsDistinct\ (rm\text{-}MT\text{-}rules\ C\ p)$   
 ⟨proof⟩

**lemma** *nlpaux*:  $x \notin dom\ (C\ b) \implies C\ (a \oplus b)\ x = C\ a\ x$   
 ⟨proof⟩

**lemma** *notindom[rule-format]*:  
 $a \in set\ p \longrightarrow x \notin dom\ (C\ (list2FWpolicy\ p)) \longrightarrow x \notin dom\ (C\ a)$   
 ⟨proof⟩

**lemma** *C-eq-rd[rule-format]*:  
 $p \neq [] \implies C\ (list2FWpolicy\ (remdups\ p)) = C\ (list2FWpolicy\ p)$   
 ⟨proof⟩

**lemma** *nMT-domMT*:  
 $\neg not\text{-}MT\ C\ p \implies p \neq [] \implies r \notin dom\ (C\ (list2FWpolicy\ p))$   
 ⟨proof⟩

**lemma** *C-eq-RS3-aux[rule-format]*:  
 $not\text{-}MT\ C\ p \implies C\ (list2FWpolicy\ p)\ x = C\ (list2FWpolicy\ (rm\text{-}MT\text{-}rules\ C\ p))\ x$   
 ⟨proof⟩

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

**lemma** *C-eq-RS3*:  
 $not\text{-}MT\ C\ p \implies C\ (list2FWpolicy\ (rm\text{-}MT\text{-}rules\ C\ p)) = C\ (list2FWpolicy\ p)$   
 ⟨proof⟩

**lemma** *NMPrd[rule-format]*:  $not\text{-}MT\ C\ p \longrightarrow not\text{-}MT\ C\ (remdups\ p)$   
 ⟨proof⟩

**lemma** *NMPDA[rule-format]*:  $DenyAll \in set\ p \longrightarrow not\text{-}MT\ C\ p$   
 ⟨proof⟩

**lemma** *NMPiD[rule-format]*:  $not\text{-}MT\ C\ (insertDeny\ p)$   
 ⟨proof⟩

**lemma** *list2FWpolicy2list[rule-format]*:  $C (list2FWpolicy(policy2list p)) = (C p)$   
 ⟨proof⟩

**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*:

$DenyAll \in set(policy2list p) \implies all-in-list(policy2list p) l \implies allNetsDistinct(policy2list p) \implies$   
 $C (list2FWpolicy$   
      $(FWNormalisationCore.sort$   
        $(removeShadowRules2 (remdups (rm-MT-rules C$   
          $(insertDeny (removeShadowRules1 (policy2list p)))))) l) =$   
 $C p$   
 ⟨proof⟩

**lemma** *C-eq-All-untilSortedQ*:

$DenyAll \in set(policy2list p) \implies all-in-list(policy2list p) l \implies allNetsDistinct(policy2list p) \implies$   
 $C (list2FWpolicy$   
      $(qsort (removeShadowRules2 (remdups (rm-MT-rules C$   
        $(insertDeny (removeShadowRules1 (policy2list p)))))) l) =$   
 $C p$   
 ⟨proof⟩

**lemma** *C-eq-All-untilSorted-withSimps*:

$DenyAll \in set(policy2list p) \implies all-in-list(policy2list p) l \implies allNetsDistinct (policy2list p) \implies$   
 $C (list2FWpolicy$   
      $(FWNormalisationCore.sort$   
        $(removeShadowRules2 (remdups (rm-MT-rules C$   
          $(insertDeny (removeShadowRules1 (policy2list p)))))) l) =$   
 $C p$   
 ⟨proof⟩

**lemma** *C-eq-All-untilSorted-withSimpsQ*:

$DenyAll \in set(policy2list p) \implies all-in-list(policy2list p) l \implies allNetsDistinct(policy2list p) \implies$

$C$  (*list2FWpolicy*  
 (qsort (removeShadowRules2 (remdups (rm-MT-rules  $C$   
 (insertDeny (removeShadowRules1 (policy2list  $p$ )))))) l)) =  
 $C$   $p$   
 ⟨proof⟩

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

$p \neq [] \longrightarrow x \in \text{dom} (C (\text{list2FWpolicy} (p))) \longrightarrow x \in \text{dom} (C (\text{list2FWpolicy} (a\#p)))$   
 ⟨proof⟩

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

⟨proof⟩

**lemma** *src-in-sdnets*[rule-format]:

$\neg \text{member DenyAll } x \longrightarrow p \in \text{dom} (C \ x) \longrightarrow \text{subnetsOfAdr} (\text{src } p) \cap (\text{fst-set} (\text{sdnets } x)) \neq \{\}$   
 ⟨proof⟩

**lemma** *dest-in-sdnets*[rule-format]:

$\neg \text{member DenyAll } x \longrightarrow p \in \text{dom} (C \ x) \longrightarrow \text{subnetsOfAdr} (\text{dest } p) \cap (\text{snd-set} (\text{sdnets } x)) \neq \{\}$   
 ⟨proof⟩

**lemma** *sdnets-in-sbnets*[rule-format]:

$p \in \text{dom} (C \ x) \longrightarrow \neg \text{member DenyAll } x \longrightarrow$   
 $(\exists (a,b) \in \text{sdnets } x. a \in \text{subnetsOfAdr} (\text{src } p) \wedge b \in \text{subnetsOfAdr} (\text{dest } p))$   
 ⟨proof⟩

**lemma** *disjSD-no-p-in-both*[rule-format]:

$\text{disjSD-2 } x \ y \Longrightarrow \neg \text{member DenyAll } x \Longrightarrow \neg \text{member DenyAll } y \Longrightarrow p \in \text{dom}(C \ x)$   
 $\Longrightarrow p \in \text{dom}(C \ y) \Longrightarrow$   
 $\text{False}$   
 ⟨proof⟩

**lemma** *list2FWpolicy-eq*:

$zs \neq [] \Longrightarrow C (\text{list2FWpolicy} (x \oplus y \# z)) \ p = C (x \oplus \text{list2FWpolicy} (y \# z)) \ p$   
 ⟨proof⟩

**lemma** *dom-sep*[rule-format]:

$x \in \text{dom} (C (\text{list2FWpolicy } p)) \longrightarrow x \in \text{dom} (C (\text{list2FWpolicy}(\text{separate } p)))$   
 ⟨proof⟩

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

$x \in \text{dom } (C (\text{list2FWpolicy } (a\#b))) \longrightarrow x \notin \text{dom } (C (\text{list2FWpolicy } b)) \longrightarrow x \in \text{dom } (C (a))$   
*<proof>*

**lemma** *sep-dom2-aux*:

$x \in \text{dom } (C (\text{list2FWpolicy } (a \oplus y \# z))) \implies x \in \text{dom } (C (a \oplus \text{list2FWpolicy } (y \# z)))$   
*<proof>*

**lemma** *sep-dom2-aux2*:

$x \in \text{dom } (C (\text{list2FWpolicy } (\text{separate } (y \# z)))) \longrightarrow x \in \text{dom } (C (\text{list2FWpolicy } (y \# z))) \implies$   
 $x \in \text{dom } (C (\text{list2FWpolicy } (a \# \text{separate } (y \# z)))) \implies x \in \text{dom } (C (\text{list2FWpolicy } (a \oplus y \# z)))$   
*<proof>*

**lemma** *sep-dom2*[*rule-format*]:

$x \in \text{dom } (C (\text{list2FWpolicy } (\text{separate } p))) \longrightarrow x \in \text{dom } (C (\text{list2FWpolicy } (p)))$   
*<proof>*

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

*<proof>*

**lemma** *C-eq-s-ext*[*rule-format*]:

$p \neq [] \longrightarrow C (\text{list2FWpolicy } (\text{separate } p)) a = C (\text{list2FWpolicy } p) a$   
*<proof>*

**lemma** *C-eq-s*:

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

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

*<proof>*

**lemmas** *C-eq-Lemmas-sep* =

*C-eq-Lemmas sortnMT sortnMTQ RS2-NMT NMPrd not-MTimpnotMT*

**lemma** *C-eq-until-separated*:

$\text{DenyAll} \in \text{set}(\text{policy2list } p) \implies \text{all-in-list}(\text{policy2list } p)l \implies \text{allNetsDistinct}$   
 $(\text{policy2list } p) \implies$   
 $C (\text{list2FWpolicy}$   
 $(\text{separate}$   
 $(\text{FWNormalisationCore.sort}$

$$(removeShadowRules2 (remdups (rm-MT-rules C (insertDeny (removeShadowRules1 (policy2list p)))))) l)) =$$

$C p$   
 $\langle proof \rangle$

**lemma** *C-eq-until-separatedQ*:

$$DenyAll \in set(policy2list p) \implies all-in-list(policy2list p)l \implies allNetsDistinct(policy2list p) \implies$$

$$C (list2FWpolicy (separate (qsort (removeShadowRules2 (remdups (rm-MT-rules C (insertDeny (removeShadowRules1 (policy2list p)))))) l)) =$$

$C p$   
 $\langle proof \rangle$

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

**lemma** *DA-is-deny*:

$$x \in dom(C(DenyAllFromTo a b \oplus DenyAllFromTo b a \oplus DenyAllFromTo a b)) \implies C(DenyAllFromTo a b \oplus DenyAllFromTo b a \oplus DenyAllFromTo a b) x = Some(deny())$$

$\langle proof \rangle$

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

$$p \neq [] \longrightarrow x \notin dom(C(list2FWpolicy p)) \longrightarrow x \in dom(C(list2FWpolicy(insertDenies p))) \longrightarrow C(list2FWpolicy(insertDenies p)) x = Some(deny())$$

$\langle proof \rangle$

**lemma** *iD-isD[rule-format]*:

$$p \neq [] \longrightarrow x \notin dom(C(list2FWpolicy p)) \longrightarrow C(DenyAll \oplus list2FWpolicy(insertDenies p)) x = C DenyAll x$$

$\langle proof \rangle$

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

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

$$p \neq [] \longrightarrow (\forall x s. s \in set p \wedge x \in dom(C A) \longrightarrow x \notin dom(C s)) \longrightarrow y \in dom(C A) \longrightarrow$$

$y \notin \text{dom} (C (\text{list2FWpolicy } p))$   
 ⟨proof⟩

**lemma** *isSepaux*:

$p \neq [] \implies \text{noDenyAll} (a \# p) \implies \text{separated} (a \# p) \implies$   
 $x \in \text{dom} (C (\text{DenyAllFromTo} (\text{first-srcNet } a) (\text{first-destNet } a) \oplus$   
 $\text{DenyAllFromTo} (\text{first-destNet } a) (\text{first-srcNet } a) \oplus a)) \implies$   
 $x \notin \text{dom} (C (\text{list2FWpolicy } p))$   
 ⟨proof⟩

**lemma** *none-MT-rulessep[rule-format]*:  $\text{none-MT-rules } C \ p \longrightarrow \text{none-MT-rules } C$   
*(separate p)*  
 ⟨proof⟩

**lemma** *dom-id*:

$\text{noDenyAll}(a\#p) \implies \text{separated}(a\#p) \implies p \neq [] \implies x \notin \text{dom}(C(\text{list2FWpolicy } p)) \implies$   
 $x \in \text{dom} (C \ a) \implies$   
 $x \notin \text{dom} (C (\text{list2FWpolicy} (\text{insertDenies } p)))$   
 ⟨proof⟩

**lemma** *C-eq-iD-aux2[rule-format]*:

$\text{noDenyAll1 } p \longrightarrow \text{separated } p \longrightarrow p \neq [] \longrightarrow x \in \text{dom} (C (\text{list2FWpolicy } p)) \longrightarrow$   
 $C(\text{list2FWpolicy} (\text{insertDenies } p)) \ x = C(\text{list2FWpolicy } p) \ x$   
 ⟨proof⟩

**lemma** *C-eq-iD*:

$\text{separated } p \implies \text{noDenyAll1 } p \implies \text{wellformed-policy1-strong } p \implies$   
 $C (\text{list2FWpolicy} (\text{insertDenies } p)) = C (\text{list2FWpolicy } p)$   
 ⟨proof⟩

**lemma** *noDASortQ[rule-format]*:  $\text{noDenyAll1 } p \longrightarrow \text{noDenyAll1} (\text{qsort } p \ l)$   
 ⟨proof⟩

**lemma** *NetsCollectedSortQ*:

$\text{distinct } p \implies \text{noDenyAll1 } p \implies \text{all-in-list } p \ l \implies \text{singleCombinators } p \implies$   
 $\text{NetsCollected} (\text{qsort } p \ l)$   
 ⟨proof⟩

**lemmas** *CLemmas* = *nMTSort nMTSortQ 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) ⇒ all-in-list(policy2list p)l ⇒ allNetsDistinct(policy2list p) ⇒*  
*C (list2FWpolicy*  
*(insertDenies*  
*(separate*  
*(FWNormalisationCore.sort*  
*(removeShadowRules2 (remdups (rm-MT-rules C*  
*(insertDeny (removeShadowRules1 (policy2list p)))))) l)))) =*  
*C p*  
 ⟨proof⟩

**lemma** *C-eq-Until-InsertDeniesQ:*

*DenyAll ∈ set(policy2list p) ⇒ all-in-list(policy2list p)l ⇒ allNetsDistinct(policy2list p) ⇒*  
*C(list2FWpolicy*  
*(insertDenies*  
*(separate (qsort (removeShadowRules2 (remdups (rm-MT-rules C*  
*(insertDeny (removeShadowRules1 (policy2list p)))))) l)))) =*  
*C p*  
 ⟨proof⟩

**lemma** *C-eq-RD-aux[rule-format]: C (p) x = C (removeDuplicates p) x*  
 ⟨proof⟩

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

*p ≠ [] → C (list2FWpolicy p) x = C (list2FWpolicy (removeAllDuplicates p)) x*  
 ⟨proof⟩

**lemma** *C-eq-RAD:*

*p ≠ [] ⇒ C (list2FWpolicy p) = C (list2FWpolicy (removeAllDuplicates p))*  
 ⟨proof⟩

**lemma** *C-eq-compile:*

*DenyAll ∈ set(policy2list p) ⇒ all-in-list(policy2list p)l ⇒ allNetsDistinct(policy2list p) ⇒*  
*C (list2FWpolicy*  
*(removeAllDuplicates*  
*(insertDenies*  
*(separate*

$(FWNormalisationCore.sort$   
 $(removeShadowRules2 (remdups (rm-MT-rules C$   
 $(insertDeny (removeShadowRules1 (policy2list p)))))) l)))) =$   
 $C p$   
 $\langle proof \rangle$

**lemma** *C-eq-compileQ:*

$DenyAll \in set(policy2list p) \implies all-in-list(policy2list p)l \implies allNetsDistinct(policy2list$   
 $p) \implies$   
 $C (list2FWpolicy$   
 $(removeAllDuplicates$   
 $(insertDenies$   
 $(separate$   
 $(qsort (removeShadowRules2 (remdups (rm-MT-rules C$   
 $(insertDeny (removeShadowRules1 (policy2list p)))))) l)))) =$   
 $C p$   
 $\langle proof \rangle$

**lemma** *C-eq-normalize:*

$DenyAll \in set (policy2list p) \implies allNetsDistinct (policy2list p) \implies$   
 $all-in-list(policy2list p)(Nets-List p) \implies$   
 $C (list2FWpolicy (normalize p)) = C p$   
 $\langle proof \rangle$

**lemma** *C-eq-normalizeQ:*

$DenyAll \in set (policy2list p) \implies allNetsDistinct (policy2list p) \implies$   
 $all-in-list (policy2list p) (Nets-List p) \implies$   
 $C (list2FWpolicy (normalizeQ p)) = C p$   
 $\langle proof \rangle$

**lemma** *domSubset3:*  $dom (C (DenyAll \oplus x)) = dom (C (DenyAll))$

$\langle proof \rangle$

**lemma** *domSubset4:*

$dom (C (DenyAllFromTo x y \oplus DenyAllFromTo y x \oplus AllowPortFromTo x y dn)) =$   
 $dom (C (DenyAllFromTo x y \oplus DenyAllFromTo y x))$   
 $\langle proof \rangle$

**lemma** *domSubset5:*

$dom (C (DenyAllFromTo x y \oplus DenyAllFromTo y x \oplus AllowPortFromTo y x dn)) =$   
 $dom (C (DenyAllFromTo x y \oplus DenyAllFromTo y x))$   
 $\langle proof \rangle$

**lemma** *domSubset1:*

$dom (C (DenyAllFromTo one two \oplus DenyAllFromTo two one \oplus AllowPortFromTo one two dn \oplus x)) =$

$dom (C (DenyAllFromTo one two \oplus DenyAllFromTo two one \oplus x))$

$\langle proof \rangle$

**lemma** *domSubset2*:

$dom (C (DenyAllFromTo one two \oplus DenyAllFromTo two one \oplus AllowPortFromTo two one dn \oplus x)) =$

$dom (C (DenyAllFromTo one two \oplus DenyAllFromTo two one \oplus x))$

$\langle proof \rangle$

**lemma** *ConcAssoc2*:  $C (X \oplus Y \oplus ((A \oplus B) \oplus D)) = C (X \oplus Y \oplus A \oplus B \oplus D)$

$\langle proof \rangle$

**lemma** *ConcAssoc3*:  $C (X \oplus ((Y \oplus A) \oplus D)) = C (X \oplus Y \oplus A \oplus D)$

$\langle proof \rangle$

**lemma** *RS3-NMT*[*rule-format*]:

$DenyAll \in set\ p \longrightarrow rm\text{-}MT\text{-}rules\ C\ p \neq []$

$\langle proof \rangle$

**lemma** *norm-notMT*:  $DenyAll \in set (policy2list\ p) \implies normalize\ p \neq []$

$\langle proof \rangle$

**lemma** *norm-notMTQ*:  $DenyAll \in set (policy2list\ p) \implies normalizeQ\ p \neq []$

$\langle proof \rangle$

**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*]:  $p \in dom (C\ a) \longrightarrow$

$C (list2policyR (a \# list))\ p = C\ a\ p$

$\langle proof \rangle$

**lemma** *list2policyR-End*:  $p \notin dom (C\ a) \implies$

$C (list2policyR (a \# list))\ p = (C\ a \oplus list2policy (map\ C\ list))\ p$

$\langle proof \rangle$

**lemma** *l2polR-eq-el*[rule-format]:

$N \neq [] \longrightarrow C(\text{list2policyR } N) p = (\text{list2policy } (\text{map } C N)) p$   
 ⟨proof⟩

**lemma** *l2polR-eq*:

$N \neq [] \implies C(\text{list2policyR } N) = (\text{list2policy } (\text{map } C N))$   
 ⟨proof⟩

**lemma** *list2FWpolicys-eq-el*[rule-format]:

$\text{Filter} \neq [] \longrightarrow C(\text{list2policyR } \text{Filter}) p = C(\text{list2FWpolicy } (\text{rev } \text{Filter})) p$   
 ⟨proof⟩

**lemma** *list2FWpolicys-eq*:

$\text{Filter} \neq [] \implies C(\text{list2policyR } \text{Filter}) = C(\text{list2FWpolicy } (\text{rev } \text{Filter}))$   
 ⟨proof⟩

**lemma** *list2FWpolicys-eq-sym*:

$\text{Filter} \neq [] \implies C(\text{list2policyR } (\text{rev } \text{Filter})) = C(\text{list2FWpolicy } \text{Filter})$   
 ⟨proof⟩

**lemma** *p-eq*[rule-format]:

$p \neq [] \longrightarrow \text{list2policy } (\text{map } C (\text{rev } p)) = C(\text{list2FWpolicy } p)$   
 ⟨proof⟩

**lemma** *p-eq2*[rule-format]:

$\text{normalize } x \neq [] \longrightarrow C(\text{list2FWpolicy}(\text{normalize } x)) = C x \longrightarrow$   
 $\text{list2policy}(\text{map } C (\text{rev}(\text{normalize } x))) = C x$   
 ⟨proof⟩

**lemma** *p-eq2Q*[rule-format]:

$\text{normalizeQ } x \neq [] \longrightarrow C(\text{list2FWpolicy } (\text{normalizeQ } x)) = C x \longrightarrow$   
 $\text{list2policy } (\text{map } C (\text{rev } (\text{normalizeQ } x))) = C x$   
 ⟨proof⟩

**lemma** *list2listNMT*[rule-format]:  $x \neq [] \longrightarrow \text{map sem } x \neq []$

⟨proof⟩

**lemma** *Norm-Distr2*:

$r \circ f ((P \otimes_2 (\text{list2policy } Q)) \circ d) = (\text{list2policy } ((P \otimes_L Q) (\otimes_2) r d))$   
 ⟨proof⟩

**lemma** *NATDistr*:

$N \neq [] \implies F = C(\text{list2policyR } N) \implies$   
 $(\lambda(x, y). x) \circ_f (\text{NAT } \otimes_2 F \circ (\lambda x. (x, x))) =$

$list2policy ((NAT \otimes_L map C N) (\otimes_2) (\lambda(x, y). x) (\lambda x. (x, x)))$   
 ⟨proof⟩

**lemma** *C-eq-normalize-manual*:

$DenyAll \in set(policy2list p) \implies allNetsDistinct(policy2list p) \implies all-in-list(policy2list p) l \implies$   
 $C (list2FWpolicy (normalize-manual-order p l)) = C p$   
 ⟨proof⟩

**lemma** *p-eq2-manualQ[rule-format]*:

$normalize-manual-orderQ x l \neq [] \longrightarrow C(list2FWpolicy (normalize-manual-orderQ x l)) = C x \longrightarrow$   
 $list2policy (map C (rev (normalize-manual-orderQ x l))) = C x$   
 ⟨proof⟩

**lemma** *norm-notMT-manualQ*:  $DenyAll \in set (policy2list p) \implies normalize-manual-orderQ p l \neq []$   
 ⟨proof⟩

**lemma** *C-eq-normalize-manualQ*:

$DenyAll \in set(policy2list p) \implies allNetsDistinct(policy2list p) \implies all-in-list(policy2list p) l \implies$   
 $C (list2FWpolicy (normalize-manual-orderQ p l)) = C p$   
 ⟨proof⟩

**lemma** *p-eq2-manual[rule-format]*:

$normalize-manual-order x l \neq [] \longrightarrow C (list2FWpolicy (normalize-manual-order x l)) = C x \longrightarrow$   
 $list2policy (map C (rev (normalize-manual-order x l))) = C x$   
 ⟨proof⟩

**lemma** *norm-notMT-manual*:  $DenyAll \in set (policy2list p) \implies normalize-manual-order p l \neq []$   
 ⟨proof⟩

As an example, how this theorems can be used for a concrete normalisation instantiation.

**lemma** *normalizeNAT*:

$DenyAll \in set (policy2list Filter) \implies allNetsDistinct (policy2list Filter) \implies$   
 $all-in-list (policy2list Filter) (Nets-List Filter) \implies$   
 $(\lambda(x, y). x) \circ_f (NAT \otimes_2 C Filter \circ (\lambda x. (x, x))) =$   
 $list2policy ((NAT \otimes_L map C (rev (FWNormalisationCore.normalize Filter))) (\otimes_2)$   
 $(\lambda(x, y). x) (\lambda x. (x, x)))$

$\langle proof \rangle$

**lemma** *domSimpl[simp]*:  $dom (C (A \oplus DenyAll)) = dom (C (DenyAll))$

$\langle proof \rangle$

The following 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 needs to combine several normalisation strategies.

**lemma** *CRotate-eq-rotateC*:  $CRotate p = C (rotatePolicy p)$

$\langle proof \rangle$

**lemma** *DAinRotate*:

$DenyAll \in set (policy2list p) \implies DenyAll \in set (policy2list (rotatePolicy p))$

$\langle proof \rangle$

**lemma** *DAUniv*:  $dom (CRotate (P \oplus DenyAll)) = UNIV$

$\langle proof \rangle$

**lemma** *p-eq2R[rule-format]*:

$normalize (rotatePolicy x) \neq [] \longrightarrow C(list2FWpolicy(normalize (rotatePolicy x))) = CRotate x \longrightarrow$

$list2policy (map C (rev (normalize (rotatePolicy x)))) = CRotate x$

$\langle proof \rangle$

**lemma** *C-eq-normalizeRotate*:

$DenyAll \in set (policy2list p) \implies allNetsDistinct (policy2list (rotatePolicy p)) \implies$

$all-in-list (policy2list (rotatePolicy p)) (Nets-List (rotatePolicy p)) \implies$

$C (list2FWpolicy$

$(removeAllDuplicates$

$(insertDenies$

$(separate$

$(sort(removeShadowRules2(remdups(rm-MT-rules C$

$(insertDeny(removeShadowRules1(policy2list(rotatePolicy p))))))$

$(Nets-List (rotatePolicy p)))))) =$

$CRotate p$

$\langle proof \rangle$

**lemma** *C-eq-normalizeRotate2*:

$DenyAll \in set (policy2list p) \implies$

$allNetsDistinct (policy2list (rotatePolicy p)) \implies$

$all-in-list (policy2list (rotatePolicy p)) (Nets-List (rotatePolicy p)) \implies$

$C (list2FWpolicy (FWNormalisationCore.normalize (rotatePolicy p))) = CRotate p$

$\langle proof \rangle$

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*:  $Cp((A \oplus B) \oplus D) = Cp(A \oplus (B \oplus D))$

*<proof>*

**lemma** *aux26[simp]*:

$twoNetsDistinct\ a\ b\ c\ d \implies dom\ (Cp\ (AllowPortFromTo\ a\ b\ p)) \cap dom\ (Cp\ (DenyAllFromTo\ c\ d)) = \{\}$

*<proof>*

**lemma** *wp2-aux[rule-format]*:

$wellformed-policy2Pr\ (xs\ @\ [x]) \longrightarrow wellformed-policy2Pr\ xs$

*<proof>*

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

*<proof>*

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

*<proof>*

**lemma** *DAimpliesMR-E[rule-format]*:  $DenyAll \in set\ p \longrightarrow$

$(\exists\ r.\ applied-rule-rev\ Cp\ x\ p = Some\ r)$

*<proof>*

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

*<proof>*

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

*<proof>*

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

*<proof>*

**lemma** *MRList3*:

$x \notin \text{dom}(Cp\ xa) \implies \text{applied-rule-rev}\ Cp\ x\ (a@b\#\#xs@[xa]) = \text{applied-rule-rev}\ Cp\ x\ (a@b\#\#xs)$   
 $\langle \text{proof} \rangle$

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

$Cp\ a\ x = \text{Some}\ y \longrightarrow Cp\ (\text{list2FWpolicy}\ (xs\ @\ [a]))\ x = \text{Some}\ y\ (\text{is}\ ?P\ xs)$   
 $\langle \text{proof} \rangle$

**lemma** *CConcStartaux*:  $Cp\ a\ x = \text{None} \implies (Cp\ aa\ ++\ Cp\ a)\ x = Cp\ aa\ x$

$\langle \text{proof} \rangle$

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

$xs \neq [] \longrightarrow Cp\ a\ x = \text{None} \longrightarrow Cp\ (\text{list2FWpolicy}\ (xs\ @\ [a]))\ x = Cp\ (\text{list2FWpolicy}\ xs)\ x$   
 $\langle \text{proof} \rangle$

**lemma** *mrNnt[simp]*:  $\text{applied-rule-rev}\ Cp\ x\ p = \text{Some}\ a \implies p \neq []$

$\langle \text{proof} \rangle$

**lemma** *mr-is-C[rule-format]*:

$\text{applied-rule-rev}\ Cp\ x\ p = \text{Some}\ a \longrightarrow Cp\ (\text{list2FWpolicy}\ (p))\ x = Cp\ a\ x$   
 $\langle \text{proof} \rangle$

**lemma** *CConcStart2*:

$p \neq [] \implies x \notin \text{dom}\ (Cp\ a) \implies Cp(\text{list2FWpolicy}\ (p@[a]))\ x = Cp\ (\text{list2FWpolicy}\ p)\ x$   
 $\langle \text{proof} \rangle$

**lemma** *CConcEnd1*:

$q@p \neq [] \implies x \notin \text{dom}\ (Cp\ a) \implies Cp(\text{list2FWpolicy}(q@p@[a]))\ x = Cp\ (\text{list2FWpolicy}\ (q@p))\ x$   
 $\langle \text{proof} \rangle$

**lemma** *CConcEnd2[rule-format]*:

$x \in \text{dom}\ (Cp\ a) \longrightarrow Cp\ (\text{list2FWpolicy}\ (xs\ @\ [a]))\ x = Cp\ a\ x\ (\text{is}\ ?P\ xs)$   
 $\langle \text{proof} \rangle$

**lemma** *bar3*:

$x \in \text{dom}\ (Cp\ (\text{list2FWpolicy}\ (xs\ @\ [xa]))) \implies x \in \text{dom}\ (Cp\ (\text{list2FWpolicy}\ xs)) \vee x \in \text{dom}\ (Cp\ xa)$   
 $\langle \text{proof} \rangle$

**lemma** *CeqEnd[rule-format,simp]*:

$a \neq [] \longrightarrow x \in \text{dom } (Cp(\text{list2FWpolicy } a)) \longrightarrow Cp(\text{list2FWpolicy}(b@a)) x = (Cp(\text{list2FWpolicy } a)) x$   
 ⟨proof⟩

**lemma** *CConcStartA*[rule-format,simp]:

$x \in \text{dom } (Cp a) \longrightarrow x \in \text{dom } (Cp (\text{list2FWpolicy } (a \# b)))$  (is ?P b)  
 ⟨proof⟩

**lemma** *domConc*:

$x \in \text{dom } (Cp (\text{list2FWpolicy } b)) \implies b \neq [] \implies x \in \text{dom } (Cp (\text{list2FWpolicy } (a@b)))$   
 ⟨proof⟩

**lemma** *CeqStart*[rule-format,simp]:

$x \notin \text{dom } (Cp (\text{list2FWpolicy } a)) \longrightarrow a \neq [] \longrightarrow b \neq [] \longrightarrow Cp (\text{list2FWpolicy } (b@a)) x = (Cp (\text{list2FWpolicy } b)) x$   
 ⟨proof⟩

**lemma** *C-eq-if-mr-eq2*:

$\text{applied-rule-rev } Cp x a = \text{Some } r \implies \text{applied-rule-rev } Cp x b = \text{Some } r \implies a \neq [] \implies b \neq [] \implies (Cp (\text{list2FWpolicy } a)) x = (Cp (\text{list2FWpolicy } b)) x$   
 ⟨proof⟩

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

$p \neq [] \longrightarrow \text{applied-rule-rev } Cp x p = \text{None} \longrightarrow Cp (\text{list2FWpolicy } p) x = \text{None}$   
 ⟨proof⟩

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

$\text{applied-rule-rev } Cp x b = \text{applied-rule-rev } Cp x a \implies a \neq [] \implies b \neq [] \implies (Cp (\text{list2FWpolicy } a)) x = (Cp (\text{list2FWpolicy } b)) x$   
 ⟨proof⟩

**lemma** *notmatching-notdom*:

$\text{applied-rule-rev } Cp x (p@[a]) \neq \text{Some } a \implies x \notin \text{dom } (Cp a)$   
 ⟨proof⟩

**lemma** *foo3a*[rule-format]:

$\text{applied-rule-rev } Cp x (a@[b]@c) = \text{Some } b \longrightarrow r \in \text{set } c \longrightarrow b \notin \text{set } c \longrightarrow x \notin \text{dom } (Cp r)$   
 ⟨proof⟩

**lemma** *foo3D*:

$\text{wellformed-policy1 } p \implies p = \text{DenyAll} \# ps \implies \text{applied-rule-rev } Cp x p = \text{Some } \text{DenyAll} \implies r \in \text{set } ps \implies$

$x \notin \text{dom} (Cp r)$   
 $\langle \text{proof} \rangle$

**lemma** *foo4*[*rule-format*]:

$\text{set } p = \text{set } s \wedge (\forall r. r \in \text{set } p \longrightarrow x \notin \text{dom} (Cp r)) \longrightarrow (\forall r. r \in \text{set } s \longrightarrow x \notin \text{dom} (Cp r))$   
 $\langle \text{proof} \rangle$

**lemma** *foo5b*[*rule-format*]:

$x \in \text{dom} (Cp b) \longrightarrow (\forall r. r \in \text{set } c \longrightarrow x \notin \text{dom} (Cp r)) \longrightarrow \text{applied-rule-rev } Cp x (b\#c) = \text{Some } b$   
 $\langle \text{proof} \rangle$

**lemma** *mr-first*:

$x \in \text{dom} (Cp b) \Longrightarrow (\forall r. r \in \text{set } c \longrightarrow x \notin \text{dom} (Cp r)) \Longrightarrow s = b\#c \Longrightarrow$   
 $\text{applied-rule-rev } Cp x s = \text{Some } b$   
 $\langle \text{proof} \rangle$

**lemma** *mr-charn*[*rule-format*]:

$a \in \text{set } p \longrightarrow (x \in \text{dom} (Cp a)) \longrightarrow (\forall r. r \in \text{set } p \wedge x \in \text{dom} (Cp r) \longrightarrow r = a)$   
 $\longrightarrow$   
 $\text{applied-rule-rev } Cp x p = \text{Some } a$   
 $\langle \text{proof} \rangle$

**lemma** *foo8*:

$\forall r. r \in \text{set } p \wedge x \in \text{dom} (Cp r) \longrightarrow r = a \Longrightarrow \text{set } p = \text{set } s \Longrightarrow$   
 $\forall r. r \in \text{set } s \wedge x \in \text{dom} (Cp r) \longrightarrow r = a$   
 $\langle \text{proof} \rangle$

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

$\text{applied-rule-rev } Cp x (b \# p) = \text{Some } a \longrightarrow a \neq b \longrightarrow \text{applied-rule-rev } Cp x p = \text{Some } a$   
 $\langle \text{proof} \rangle$

**lemma** *wp3tl*[*rule-format*]: *wellformed-policy3Pr*  $p \longrightarrow \text{wellformed-policy3Pr} (tl p)$

$\langle \text{proof} \rangle$

**lemma** *wp3Conc*[*rule-format*]: *wellformed-policy3Pr*  $(a\#p) \longrightarrow \text{wellformed-policy3Pr } p$

$\langle \text{proof} \rangle$

**lemma** *foo98*[*rule-format*]:

$\text{applied-rule-rev } Cp x (aa \# p) = \text{Some } a \longrightarrow x \in \text{dom} (Cp r) \longrightarrow r \in \text{set } p \longrightarrow a \in$

*set p*  
⟨*proof*⟩

**lemma** *mrMTNone[simp]*: *applied-rule-rev Cp x [] = None*  
⟨*proof*⟩

**lemma** *DAAux[simp]*: *x ∈ dom (Cp DenyAll)*  
⟨*proof*⟩

**lemma** *mrSet[rule-format]*: *applied-rule-rev Cp x p = Some r ⟶ r ∈ set p*  
⟨*proof*⟩

**lemma** *mr-not-Conc*: *singleCombinators p ⟹ applied-rule-rev Cp x p ≠ Some (a⊕b)*  
⟨*proof*⟩

**lemma** *foo25[rule-format]*: *wellformed-policy3Pr (p@[x]) ⟶ wellformed-policy3Pr p*  
⟨*proof*⟩

**lemma** *mr-in-dom[rule-format]*: *applied-rule-rev Cp x p = Some a ⟶ x ∈ dom (Cp a)*  
⟨*proof*⟩

**lemma** *wp3EndMT[rule-format]*:  
*wellformed-policy3Pr (p@[xs]) ⟶ AllowPortFromTo a b po ∈ set p ⟶*  
*dom (Cp (AllowPortFromTo a b po)) ∩ dom (Cp xs) = {}*  
⟨*proof*⟩

**lemma** *foo29*: *dom (Cp a) ≠ {} ⟹ dom (Cp a) ∩ dom (Cp b) = {} ⟹ a ≠ b*  
⟨*proof*⟩

**lemma** *foo28*:  
*AllowPortFromTo a b po ∈ set p ⟹ dom(Cp(AllowPortFromTo a b po)) ≠ {} ⟹*  
*(wellformed-policy3Pr(p@[x])) ⟹*  
*x ≠ AllowPortFromTo a b po*  
⟨*proof*⟩

**lemma** *foo28a[rule-format]*: *x ∈ dom (Cp a) ⟹ dom (Cp a) ≠ {}*  
⟨*proof*⟩

**lemma** *allow-deny-dom[simp]*:  
*dom (Cp (AllowPortFromTo a b po)) ⊆ dom (Cp (DenyAllFromTo a b))*  
⟨*proof*⟩

**lemma** *DenyAllowDisj*:

$$\begin{aligned} & \text{dom } (Cp \text{ (AllowPortFromTo } a \ b \ p)) \neq \{\} \implies \\ & \text{dom } (Cp \text{ (DenyAllFromTo } a \ b)) \cap \text{dom } (Cp \text{ (AllowPortFromTo } a \ b \ p)) \neq \{\} \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *foo31*:

$$\begin{aligned} & \forall r. r \in \text{set } p \wedge x \in \text{dom } (Cp \ r) \longrightarrow \\ & \quad (r = \text{AllowPortFromTo } a \ b \ po \vee r = \text{DenyAllFromTo } a \ b \vee r = \text{DenyAll}) \implies \\ & \quad \text{set } p = \text{set } s \implies \\ & \quad (\forall r. r \in \text{set } s \wedge x \in \text{dom}(Cp \ r) \longrightarrow r = \text{AllowPortFromTo } a \ b \ po \vee r = \text{DenyAllFromTo } \\ & \quad a \ b \vee r = \text{DenyAll}) \\ & \langle \text{proof} \rangle \end{aligned}$$

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

$\langle \text{proof} \rangle$

**lemma** *deny-dom[simp]*:

$$\begin{aligned} & \text{twoNetsDistinct } a \ b \ c \ d \implies \text{dom } (Cp \text{ (DenyAllFromTo } a \ b)) \cap \text{dom } (Cp \\ & \text{(DenyAllFromTo } c \ d)) = \{\} \\ & \langle \text{proof} \rangle \end{aligned}$$

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

$\langle \text{proof} \rangle$

**lemma** *DomInterAllowsMT*:

$$\begin{aligned} & \text{twoNetsDistinct } a \ b \ c \ d \implies \text{dom } (Cp(\text{AllowPortFromTo } a \ b \ p)) \cap \\ & \text{dom}(Cp(\text{AllowPortFromTo } c \ d \ po)) = \{\} \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *DomInterAllowsMT-Ports*:

$$\begin{aligned} & p \neq po \implies \text{dom } (Cp \text{ (AllowPortFromTo } a \ b \ p)) \cap \text{dom } (Cp \text{ (AllowPortFromTo } c \ d \\ & \text{po})) = \{\} \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *wellformed-policy3-charn[rule-format]*:

$$\begin{aligned} & \text{singleCombinators } p \longrightarrow \text{distinct } p \longrightarrow \text{allNetsDistinct } p \longrightarrow \\ & \text{wellformed-policy1 } p \longrightarrow \text{wellformed-policy2Pr } p \longrightarrow \text{wellformed-policy3Pr } p \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *DistinctNetsDenyAllow*:

$$\text{DenyAllFromTo } b \ c \in \text{set } p \implies \text{AllowPortFromTo } a \ d \ po \in \text{set } p \implies \text{allNetsDistinct } p \implies$$

$dom (Cp (DenyAllFromTo b c)) \cap dom (Cp (AllowPortFromTo a d po)) \neq \{\} \implies$   
 $b = a \wedge c = d$   
 ⟨proof⟩

**lemma** *DistinctNetsAllowAllow*:

$AllowPortFromTo b c poo \in set p \implies AllowPortFromTo a d po \in set p \implies$   
 $allNetsDistinct p \implies dom(Cp(AllowPortFromTo b c poo)) \cap$   
 $dom(Cp(AllowPortFromTo a d po)) \neq \{\} \implies$   
 $b = a \wedge c = d \wedge poo = po$   
 ⟨proof⟩

**lemma** *WP2RS2[simp]*:

$singleCombinators p \implies distinct p \implies allNetsDistinct p \implies$   
 $wellformed-policy2Pr (removeShadowRules2 p)$   
 ⟨proof⟩

**lemma** *AD-aux*:

$AllowPortFromTo a b po \in set p \implies DenyAllFromTo c d \in set p \implies$   
 $allNetsDistinct p \implies singleCombinators p \implies a \neq c \vee b \neq d \implies$   
 $dom (Cp (AllowPortFromTo a b po)) \cap dom (Cp (DenyAllFromTo c d)) = \{\}$   
 ⟨proof⟩

**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⟩

**lemma** *wellformed2-sorted[simp]*:

$all-in-list p l \implies distinct p \implies allNetsDistinct p \implies singleCombinators p \implies$   
 $wellformed-policy2Pr (sort p l)$   
 ⟨proof⟩

**lemma** *wellformed2-sortedQ[simp]*:

$all-in-list p l \implies distinct p \implies allNetsDistinct p \implies singleCombinators p \implies$   
 $wellformed-policy2Pr (qsort p l)$   
 ⟨proof⟩

**lemma** *C-DenyAll[simp]*:  $Cp (list2FWpolicy (xs @ [DenyAll])) x = Some (deny ())$

⟨proof⟩

**lemma** *C-eq-RS1n*:

$Cp(list2FWpolicy (removeShadowRules1-alternative p)) = Cp(list2FWpolicy p)$   
 ⟨proof⟩

**lemma** *C-eq-RS1[simp]*:

$p \neq [] \implies Cp(list2FWpolicy (removeShadowRules1 p)) = Cp(list2FWpolicy p)$   
 $\langle proof \rangle$

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

$applied-rule-rev Cp x (DenyAll \# p) \neq Some DenyAll \longrightarrow (\exists y. applied-rule-rev Cp x p = Some y)$   
 $\langle proof \rangle$

**lemma** *EX-MR :*

$applied-rule-rev Cp x p \neq (Some DenyAll) \implies p = DenyAll \# ps \implies$   
 $(applied-rule-rev Cp x p = applied-rule-rev Cp x ps)$   
 $\langle proof \rangle$

**lemma** *mr-not-DA:*

$wellformed-policy1-strong s \implies applied-rule-rev Cp x p = Some (DenyAllFromTo a ab) \implies$   
 $set p = set s \implies applied-rule-rev Cp x s \neq Some DenyAll$   
 $\langle proof \rangle$

**lemma** *domsMT-notND-DD:*

$dom (Cp (DenyAllFromTo a b)) \cap dom (Cp (DenyAllFromTo c d)) \neq \{\} \implies \neg$   
 $netsDistinct a c$   
 $\langle proof \rangle$

**lemma** *domsMT-notND-DD2:*

$dom (Cp (DenyAllFromTo a b)) \cap dom (Cp (DenyAllFromTo c d)) \neq \{\} \implies \neg$   
 $netsDistinct b d$   
 $\langle proof \rangle$

**lemma** *domsMT-notND-DD3:*

$x \in dom (Cp (DenyAllFromTo a b)) \implies x \in dom (Cp (DenyAllFromTo c d)) \implies \neg$   
 $netsDistinct a c$   
 $\langle proof \rangle$

**lemma** *domsMT-notND-DD4:*

$x \in dom (Cp (DenyAllFromTo a b)) \implies x \in dom (Cp (DenyAllFromTo c d)) \implies \neg$   
 $netsDistinct b d$   
 $\langle proof \rangle$

**lemma** *NetsEq-if-sameP-DD:*

$allNetsDistinct p \implies u \in set p \implies v \in set p \implies u = (DenyAllFromTo a b) \implies$   
 $v = (DenyAllFromTo c d) \implies x \in dom (Cp (u)) \implies x \in dom (Cp (v)) \implies$

$a = c \wedge b = d$   
 ⟨proof⟩

**lemma** *rule-charn1*:

**assumes** *aND* : *allNetsDistinct* *p*  
**and** *mr-is-allow* : *applied-rule-rev* *Cp* *x* *p* = *Some* (*AllowPortFromTo* *a* *b* *po*)  
**and** *SC* : *singleCombinators* *p*  
**and** *inp* :  $r \in \text{set } p$   
**and** *inDom* :  $x \in \text{dom } (Cp \ r)$   
**shows**  $(r = \text{AllowPortFromTo } a \ b \ po \vee r = \text{DenyAllFromTo } a \ b \vee r = \text{DenyAll})$   
 ⟨proof⟩

**lemma** *none-MT-rulessubset*[*rule-format*]:

*none-MT-rules* *Cp* *a*  $\longrightarrow$  *set* *b*  $\subseteq$  *set* *a*  $\longrightarrow$  *none-MT-rules* *Cp* *b*  
 ⟨proof⟩

**lemma** *nMTSort*: *none-MT-rules* *Cp* *p*  $\implies$  *none-MT-rules* *Cp* (*sort* *p* *l*)

⟨proof⟩

**lemma** *nMTSortQ*: *none-MT-rules* *Cp* *p*  $\implies$  *none-MT-rules* *Cp* (*qsort* *p* *l*)

⟨proof⟩

**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*

⟨proof⟩

**lemma** *wp3charn*[*rule-format*]:

**assumes** *domAllow*:  $\text{dom } (Cp \ (\text{AllowPortFromTo } a \ b \ po)) \neq \{\}$   
**and** *wp3*: *wellformed-policy3Pr* (*xs* @ [*DenyAllFromTo* *a* *b*])  
**shows** *allowNotInList*: *AllowPortFromTo* *a* *b* *po*  $\notin$  *set* *xs*  
 ⟨proof⟩

**lemma** *rule-charn2*:

**assumes** *aND*: *allNetsDistinct* *p*  
**and** *wp1*: *wellformed-policy1* *p*  
**and** *SC*: *singleCombinators* *p*  
**and** *wp3*: *wellformed-policy3Pr* *p*  
**and** *allow-in-list*: *AllowPortFromTo* *c* *d* *po*  $\in$  *set* *p*  
**and** *x-in-dom-allow*:  $x \in \text{dom } (Cp \ (\text{AllowPortFromTo } c \ d \ po))$   
**shows** *applied-rule-rev* *Cp* *x* *p* = *Some* (*AllowPortFromTo* *c* *d* *po*)  
 ⟨proof⟩

**lemma rule-charn3:**

$wellformed-policy1\ p \implies allNetsDistinct\ p \implies singleCombinators\ p \implies$   
 $wellformed-policy3Pr\ p \implies applied-rule-rev\ Cp\ x\ p = Some\ (DenyAllFromTo\ c\ d) \implies$

$AllowPortFromTo\ a\ b\ po \in set\ p \implies x \notin dom\ (Cp\ (AllowPortFromTo\ a\ b\ po))$   
 $\langle proof \rangle$

**lemma rule-charn4:**

**assumes**  $wp1:$   $wellformed-policy1\ p$   
**and**  $aND:$   $allNetsDistinct\ p$   
**and**  $SC:$   $singleCombinators\ p$   
**and**  $wp3:$   $wellformed-policy3Pr\ p$   
**and**  $DA:$   $DenyAll \notin set\ p$   
**and**  $mr:$   $applied-rule-rev\ Cp\ x\ p = Some\ (DenyAllFromTo\ a\ b)$   
**and**  $rinp:$   $r \in set\ p$   
**and**  $xindom:$   $x \in dom\ (Cp\ r)$   
**shows**  $r = DenyAllFromTo\ a\ b$

$\langle proof \rangle$

**lemma foo31a:**

$(\forall\ r. r \in set\ p \wedge x \in dom\ (Cp\ r) \longrightarrow$   
 $(r = AllowPortFromTo\ a\ b\ po \vee r = DenyAllFromTo\ a\ b \vee r = DenyAll)) \implies$   
 $set\ p = set\ s \implies r \in set\ s \implies x \in dom\ (Cp\ r) \implies$   
 $(r = AllowPortFromTo\ a\ b\ po \vee r = DenyAllFromTo\ a\ b \vee r = DenyAll)$   
 $\langle proof \rangle$

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

$applied-rule-rev\ Cp\ x\ (a\#p) = Some\ a \longrightarrow a \notin set\ (p) \longrightarrow applied-rule-rev\ Cp\ x\ p =$   
 $None$   
 $\langle proof \rangle$

**lemma mrDA-tl:**

**assumes**  $mr-DA:$   $applied-rule-rev\ Cp\ x\ p = Some\ DenyAll$   
**and**  $wp1n:$   $wellformed-policy1-strong\ p$   
**shows**  $applied-rule-rev\ Cp\ x\ (tl\ p) = None$   
 $\langle proof \rangle$

**lemma rule-charnDAFT:**

$wellformed-policy1-strong\ p \implies allNetsDistinct\ p \implies singleCombinators\ p \implies$   
 $wellformed-policy3Pr\ p \implies applied-rule-rev\ Cp\ x\ p = Some\ (DenyAllFromTo\ a\ b)$   
 $\implies$   
 $r \in set\ (tl\ p) \implies x \in dom\ (Cp\ r) \implies$   
 $r = DenyAllFromTo\ a\ b$   
 $\langle proof \rangle$

**lemma** *mrDenyAll-is-unique*:

*wellformed-policy1-strong*  $p \implies \text{applied-rule-rev } Cp \ x \ p = \text{Some } \text{DenyAll} \implies r \in \text{set}$   
(*tl*  $p$ )  $\implies$   
 $x \notin \text{dom } (Cp \ r)$   
(*proof*)

**theorem** *C-eq-Sets-mr*:

**assumes** *sets-eq*:  $\text{set } p = \text{set } s$   
**and** *SC*:  $\text{singleCombinators } p$   
**and** *wp1-p*:  $\text{wellformed-policy1-strong } p$   
**and** *wp1-s*:  $\text{wellformed-policy1-strong } s$   
**and** *wp3-p*:  $\text{wellformed-policy3Pr } p$   
**and** *wp3-s*:  $\text{wellformed-policy3Pr } s$   
**and** *aND*:  $\text{allNetsDistinct } p$   
**shows**  $\text{applied-rule-rev } Cp \ x \ p = \text{applied-rule-rev } Cp \ x \ s$   
(*proof*)

**lemma** *C-eq-Sets*:

$\text{singleCombinators } p \implies \text{wellformed-policy1-strong } p \implies \text{wellformed-policy1-strong } s$   
 $\implies$   
 $\text{wellformed-policy3Pr } p \implies \text{wellformed-policy3Pr } s \implies \text{allNetsDistinct } p \implies \text{set } p =$   
 $\text{set } s \implies$   
 $Cp \ (\text{list2FWpolicy } p) \ x = Cp \ (\text{list2FWpolicy } s) \ x$   
(*proof*)

**lemma** *C-eq-sorted*:

$\text{distinct } p \implies \text{all-in-list } p \ l \implies \text{singleCombinators } p \implies$   
 $\text{wellformed-policy1-strong } p \implies \text{wellformed-policy3Pr } p \implies \text{allNetsDistinct } p \implies$   
 $Cp \ (\text{list2FWpolicy } (\text{sort } p \ l)) = Cp \ (\text{list2FWpolicy } p)$   
(*proof*)

**lemma** *C-eq-sortedQ*:

$\text{distinct } p \implies \text{all-in-list } p \ l \implies \text{singleCombinators } p \implies$   
 $\text{wellformed-policy1-strong } p \implies \text{wellformed-policy3Pr } p \implies \text{allNetsDistinct } p \implies$   
 $Cp \ (\text{list2FWpolicy } (\text{qsort } p \ l)) = Cp \ (\text{list2FWpolicy } p)$   
(*proof*)

**lemma** *C-eq-RS2-mr*:  $\text{applied-rule-rev } Cp \ x \ (\text{removeShadowRules2 } p) = \text{applied-rule-rev}$   
 $Cp \ x \ p$   
(*proof*)

**lemma** *C-eq-None[rule-format]*:

$p \neq [] \longrightarrow \text{applied-rule-rev } Cp \ x \ p = \text{None} \longrightarrow Cp \ (\text{list2FWpolicy } p) \ x = \text{None}$

$\langle \text{proof} \rangle$

**lemma** *C-eq-None2*:

$a \neq [] \implies b \neq [] \implies \text{applied-rule-rev } Cp \ x \ a = \text{None} \implies \text{applied-rule-rev } Cp \ x \ b = \text{None} \implies$

$(Cp \ (\text{list2FWpolicy } a)) \ x = (Cp \ (\text{list2FWpolicy } b)) \ x$

$\langle \text{proof} \rangle$

**lemma** *C-eq-RS2*:

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

$Cp \ (\text{list2FWpolicy } (\text{removeShadowRules2 } p)) = Cp \ (\text{list2FWpolicy } p)$

$\langle \text{proof} \rangle$

**lemma** *none-MT-rulesRS2*:  $\text{none-MT-rules } Cp \ p \implies \text{none-MT-rules } Cp \ (\text{removeShadowRules2 } p)$

$\langle \text{proof} \rangle$

**lemma** *CconcNone*:

$\text{dom } (Cp \ a) = \{\} \implies p \neq [] \implies Cp \ (\text{list2FWpolicy } (a \# p)) \ x = Cp \ (\text{list2FWpolicy } p) \ x$

$\langle \text{proof} \rangle$

**lemma** *none-MT-rulesrd[rule-format]*:  $\text{none-MT-rules } Cp \ p \longrightarrow \text{none-MT-rules } Cp \ (\text{remdups } p)$

$\langle \text{proof} \rangle$

**lemma** *DARS3[rule-format]*:  $\text{DenyAll} \notin \text{set } p \longrightarrow \text{DenyAll} \notin \text{set } (\text{rm-MT-rules } Cp \ p)$

$\langle \text{proof} \rangle$

**lemma** *DAnMT*:  $\text{dom } (Cp \ \text{DenyAll}) \neq \{\}$

$\langle \text{proof} \rangle$

**lemma** *DAnMT2*:  $Cp \ \text{DenyAll} \neq \text{Map.empty}$

$\langle \text{proof} \rangle$

**lemma** *wp1n-RS3[rule-format,simp]*:

$\text{wellformed-policy1-strong } p \longrightarrow \text{wellformed-policy1-strong } (\text{rm-MT-rules } Cp \ p)$

$\langle \text{proof} \rangle$

**lemma** *AILRS3[rule-format,simp]*:

$\text{all-in-list } p \ l \longrightarrow \text{all-in-list } (\text{rm-MT-rules } Cp \ p) \ l$

$\langle \text{proof} \rangle$

**lemma** *SCRS3[rule-format,simp]*:

$singleCombinators\ p \longrightarrow singleCombinators(rm-MT-rules\ Cp\ p)$   
 $\langle proof \rangle$

**lemma** *RS3subset*:  $set\ (rm-MT-rules\ Cp\ p) \subseteq set\ p$   
 $\langle proof \rangle$

**lemma** *ANDRS3[simp]*:  
 $singleCombinators\ p \Longrightarrow allNetsDistinct\ p \Longrightarrow allNetsDistinct\ (rm-MT-rules\ Cp\ p)$   
 $\langle proof \rangle$

**lemma** *nlpaux*:  $x \notin dom\ (Cp\ b) \Longrightarrow Cp\ (a \oplus b)\ x = Cp\ a\ x$   
 $\langle proof \rangle$

**lemma** *notindom[rule-format]*:  
 $a \in set\ p \longrightarrow x \notin dom\ (Cp\ (list2FWpolicy\ p)) \longrightarrow x \notin dom\ (Cp\ a)$   
 $\langle proof \rangle$

**lemma** *C-eq-rd[rule-format]*:  
 $p \neq [] \Longrightarrow Cp\ (list2FWpolicy\ (remdups\ p)) = Cp\ (list2FWpolicy\ p)$   
 $\langle proof \rangle$

**lemma** *nMT-domMT*:  
 $\neg not-MT\ Cp\ p \Longrightarrow p \neq [] \Longrightarrow r \notin dom\ (Cp\ (list2FWpolicy\ p))$   
 $\langle proof \rangle$

**lemma** *C-eq-RS3-aux[rule-format]*:  
 $not-MT\ Cp\ p \Longrightarrow Cp\ (list2FWpolicy\ p)\ x = Cp\ (list2FWpolicy\ (rm-MT-rules\ Cp\ p))\ x$   
 $\langle proof \rangle$

**lemma** *C-eq-id*:  
 $wellformed-policy1-strong\ p \Longrightarrow Cp\ (list2FWpolicy\ (insertDeny\ p)) = Cp\ (list2FWpolicy\ p)$   
 $\langle proof \rangle$

**lemma** *C-eq-RS3*:  
 $not-MT\ Cp\ p \Longrightarrow Cp\ (list2FWpolicy\ (rm-MT-rules\ Cp\ p)) = Cp\ (list2FWpolicy\ p)$   
 $\langle proof \rangle$

**lemma** *NMPrd[rule-format]*:  $not-MT\ Cp\ p \longrightarrow not-MT\ Cp\ (remdups\ p)$   
 $\langle proof \rangle$

**lemma** *NMPDA[rule-format]*:  $DenyAll \in set\ p \longrightarrow not-MT\ Cp\ p$   
 $\langle proof \rangle$

**lemma** *NMPiD*[*rule-format*]: *not-MT Cp (insertDeny p)*  
 ⟨*proof*⟩

**lemma** *list2FWpolicy2list*[*rule-format*]:  
 $Cp (list2FWpolicy(policy2list p)) = (Cp p)$   
 ⟨*proof*⟩

**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*:  
 $DenyAll \in set(policy2list p) \implies all-in-list(policy2list p) l \implies allNetsDistinct(policy2list p) \implies$   
 $Cp(list2FWpolicy (sort (removeShadowRules2 (remdups (rm-MT-rules Cp$   
*(insertDeny*  
 $(removeShadowRules1 (policy2list p)))))) l)) =$   
 $Cp p$   
 ⟨*proof*⟩

**lemma** *C-eq-All-untilSortedQ*:  
 $DenyAll \in set(policy2list p) \implies all-in-list(policy2list p) l \implies allNetsDis-$   
*tinct(policy2list p) \implies*  
 $Cp(list2FWpolicy (qsort (removeShadowRules2 (remdups (rm-MT-rules Cp$   
*(insertDeny*  
 $(removeShadowRules1 (policy2list p)))))) l)) =$   
 $Cp p$   
 ⟨*proof*⟩

**lemma** *C-eq-All-untilSorted-withSimps*:  
 $DenyAll \in set (policy2list p) \implies all-in-list (policy2list p) l \implies$   
 $allNetsDistinct (policy2list p) \implies$   
 $Cp(list2FWpolicy(sort(removeShadowRules2(remdups(rm-MT-rules Cp (insertDeny$   
 $(removeShadowRules1(policy2list p)))))) l)) =$   
 $Cp p$   
 ⟨*proof*⟩

**lemma** *C-eq-All-untilSorted-withSimpsQ*:  
 $DenyAll \in set (policy2list p) \implies all-in-list (policy2list p) l \implies$

$allNetsDistinct (policy2list p) \implies$   
 $Cp(list2FWpolicy(qsort(removeShadowRules2(remdups(rm-MT-rules Cp (insertDeny$   
 $(removeShadowRules1 (policy2list p)))))) l) =$   
 $Cp p$   
 $\langle proof \rangle$

**lemma** *InDomConc*[rule-format]:  $p \neq [] \longrightarrow x \in dom (Cp (list2FWpolicy (p))) \longrightarrow$   
 $x \in dom (Cp (list2FWpolicy (a\#p)))$   
 $\langle proof \rangle$

**lemma** *not-in-member*[rule-format]:  $member a b \longrightarrow x \notin dom (Cp b) \longrightarrow x \notin dom (Cp$   
 $a)$   
 $\langle proof \rangle$

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

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

**lemma** *sdnets-in-subnets*[rule-format]:  
 $p \in dom (Cp x) \longrightarrow \neg member DenyAll x \longrightarrow$   
 $(\exists (a,b) \in sdnets x. a \in subnetsOfAdr (src p) \wedge b \in subnetsOfAdr (dest p))$   
 $\langle proof \rangle$

**lemma** *disjSD-no-p-in-both*[rule-format]:  
 $\llbracket disjSD-2 x y; \neg member DenyAll x; \neg member DenyAll y;$   
 $p \in dom (Cp x); p \in dom (Cp y) \rrbracket \implies False$   
 $\langle proof \rangle$

**lemma** *list2FWpolicy-eq*:  
 $zs \neq [] \implies Cp (list2FWpolicy (x \oplus y \# z)) p = Cp (x \oplus list2FWpolicy (y \# z)) p$   
 $\langle proof \rangle$

**lemma** *dom-sep*[rule-format]:  
 $x \in dom (Cp (list2FWpolicy p)) \longrightarrow x \in dom (Cp (list2FWpolicy (separate p)))$   
 $\langle proof \rangle$

**lemma** *domdConcStart*[rule-format]:  
 $x \in dom (Cp (list2FWpolicy (a\#b))) \longrightarrow x \notin dom (Cp (list2FWpolicy b)) \longrightarrow x \in$

$dom (Cp (a))$   
 ⟨proof⟩

**lemma sep-dom2-aux:**

$x \in dom (Cp (list2FWpolicy (a \oplus y \# z))) \implies x \in dom (Cp (a \oplus list2FWpolicy (y \# z)))$   
 ⟨proof⟩

**lemma sep-dom2-aux2:**

$(x \in dom (Cp (list2FWpolicy (separate (y \# z)))) \longrightarrow x \in dom (Cp (list2FWpolicy (y \# z)))) \implies$   
 $x \in dom (Cp (list2FWpolicy (a \# separate (y \# z)))) \implies$   
 $x \in dom (Cp (list2FWpolicy (a \oplus y \# z)))$   
 ⟨proof⟩

**lemma sep-dom2[rule-format]:**

$x \in dom (Cp (list2FWpolicy (separate p))) \longrightarrow x \in dom (Cp (list2FWpolicy (p)))$   
 ⟨proof⟩

**lemma sepDom:**  $dom (Cp (list2FWpolicy p)) = dom (Cp (list2FWpolicy (separate p)))$   
 ⟨proof⟩

**lemma C-eq-s-ext[rule-format]:**

$p \neq [] \longrightarrow Cp (list2FWpolicy (separate p)) a = Cp (list2FWpolicy p) a$   
 ⟨proof⟩

**lemma C-eq-s:**  $p \neq [] \implies Cp (list2FWpolicy (separate p)) = Cp (list2FWpolicy p)$   
 ⟨proof⟩

**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:**

$DenyAll \in set (policy2list p) \implies all-in-list (policy2list p) l \implies allNetsDistinct (policy2list p) \implies$   
 $Cp (list2FWpolicy (separate (sort (removeShadowRules2 (remdups (rm-MT-rules Cp (insertDeny (removeShadowRules1 (policy2list p)))))) l))) =$   
 $Cp p$   
 ⟨proof⟩

**lemma C-eq-until-separatedQ:**

$DenyAll \in set (policy2list p) \implies all-in-list (policy2list p) l \implies$   
 $allNetsDistinct (policy2list p) \implies$

$$Cp(list2FWpolicy(separate(qsort($$
  

$$removeShadowRules2(remdups (rm-MT-rules Cp$$
  

$$(insertDeny (removeShadowRules1 (policy2list p)))))) l))) =$$
  

$$Cp p$$
  
 <proof>

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

$$p \neq [] \wedge x \in dom(Cp(list2FWpolicy p)) \longrightarrow x \in dom (Cp(list2FWpolicy(insertDenies$$
  

$$p)))$$
  
 <proof>

**lemma** *DA-is-deny:*

$$x \in dom (Cp (DenyAllFromTo a b \oplus DenyAllFromTo b a \oplus DenyAllFromTo a b))$$
  

$$\implies$$
  

$$Cp (DenyAllFromTo a b \oplus DenyAllFromTo b a \oplus DenyAllFromTo a b) x = Some (deny$$
  

$$())$$
  
 <proof>

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

$$p \neq [] \longrightarrow x \notin dom (Cp (list2FWpolicy p)) \longrightarrow$$
  

$$x \in dom (Cp (list2FWpolicy (insertDenies p))) \longrightarrow$$
  

$$Cp (list2FWpolicy (insertDenies p)) x = Some (deny ())$$
  
 <proof>

**lemma** *iD-isD[rule-format]:*

$$p \neq [] \longrightarrow x \notin dom (Cp (list2FWpolicy p)) \longrightarrow$$
  

$$Cp (DenyAll \oplus list2FWpolicy (insertDenies p)) x = Cp DenyAll x$$
  
 <proof>

**lemma** *inDomConc:*

$$x \notin dom (Cp a) \implies x \notin dom (Cp (list2FWpolicy p)) \implies x \notin dom (Cp$$
  

$$(list2FWpolicy(a\#p)))$$
  
 <proof>

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

$$p \neq [] \longrightarrow (\forall x s. s \in set p \wedge x \in dom (Cp A) \longrightarrow x \notin dom (Cp s)) \longrightarrow y \in dom$$
  

$$(Cp A) \longrightarrow$$
  

$$y \notin dom (Cp (list2FWpolicy p))$$
  
 <proof>

**lemma** *isSepaux:*

$$p \neq [] \implies noDenyAll (a\#p) \implies separated (a \# p) \implies$$
  

$$x \in dom (Cp (DenyAllFromTo (first-srcNet a) (first-destNet a) \oplus$$
  

$$DenyAllFromTo (first-destNet a) (first-srcNet a) \oplus a)) \implies$$

$x \notin \text{dom} (Cp (\text{list2FWpolicy } p))$   
 ⟨proof⟩

**lemma** *none-MT-rulessep*[rule-format]: *none-MT-rules*  $Cp\ p \longrightarrow \text{none-MT-rules } Cp$   
 (*separate*  $p$ )  
 ⟨proof⟩

**lemma** *dom-id*:  
 $\text{noDenyAll} (a\#p) \Longrightarrow \text{separated} (a\#p) \Longrightarrow p \neq [] \Longrightarrow$   
 $x \notin \text{dom} (Cp (\text{list2FWpolicy } p)) \Longrightarrow x \in \text{dom} (Cp (a)) \Longrightarrow$   
 $x \notin \text{dom} (Cp (\text{list2FWpolicy} (\text{insertDenies } p)))$   
 ⟨proof⟩

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

**lemma** *C-eq-iD*:  
 $\text{separated } p \Longrightarrow \text{noDenyAll1 } p \Longrightarrow \text{wellformed-policy1-strong } p \Longrightarrow$   
 $Cp(\text{list2FWpolicy} (\text{insertDenies } p)) = Cp (\text{list2FWpolicy } p)$   
 ⟨proof⟩

**lemma** *noDAsortQ*[rule-format]: *noDenyAll1*  $p \longrightarrow \text{noDenyAll1} (q\text{sort } p\ l)$   
 ⟨proof⟩

**lemma** *NetsCollectedSortQ*:  
 $\text{distinct } p \Longrightarrow \text{noDenyAll1 } p \Longrightarrow \text{all-in-list } p\ l \Longrightarrow \text{singleCombinators } p \Longrightarrow$   
 $\text{NetsCollected} (q\text{sort } p\ l)$   
 ⟨proof⟩

**lemmas**  $C\text{Lemmas} = \text{nMTSort } \text{nMTSortQ } \text{none-MT-rulesRS2 } \text{none-MT-rulesrd}$   
 $\text{noDAsort } \text{noDAsortQ } \text{nDASC } \text{wp1-eq } \text{wp1ID } \text{SCp2l } \text{ANDSep } \text{wp1n-RS2}$

$\text{OTNSEp } \text{OTNSC } \text{noDA1sep } \text{wp1-alternativesep } \text{wellformed-eq}$   
 $\text{wellformed1-alternative-sorted}$

**lemmas**  $C\text{-eqLemmas-id} = C\text{Lemmas } \text{NC2Sep } \text{NetsCollectedSep}$   
 $\text{NetsCollectedSort } \text{NetsCollectedSortQ } \text{separatedNC}$

**lemma** *C-eq-Until-InsertDenies*:  
 $\text{DenyAll} \in \text{set}(\text{policy2list } p) \Longrightarrow \text{all-in-list}(\text{policy2list } p)\ l \Longrightarrow \text{allNetsDistinct}$   
 $(\text{policy2list } p) \Longrightarrow$   
 $Cp (\text{list2FWpolicy} ((\text{insertDenies}(\text{separate}(\text{sort}(\text{removeShadowRules2}$   
 $(\text{remdups}(\text{rm-MT-rules } Cp (\text{insertDeny } (\text{removeShadowRules1 } (\text{policy2list}$

$p)))))) l)))) =$   
 $Cp\ p$   
 $\langle proof \rangle$

**lemma** *C-eq-Until-InsertDeniesQ*:

$DenyAll \in set\ (policy2list\ p) \implies all-in-list\ (policy2list\ p)\ l \implies$   
 $allNetsDistinct\ (policy2list\ p) \implies$   
 $Cp\ (list2FWpolicy\ ((insertDenies\ (separate\ (qsort\ (removeShadowRules2$   
 $(remdups\ (rm-MT-rules\ Cp\ (insertDeny\ (removeShadowRules1\ (policy2list$   
 $p)))))) l)))) =$   
 $Cp\ p$   
 $\langle proof \rangle$

**lemma** *C-eq-RD-aux[rule-format]*:  $Cp\ (p)\ x = Cp\ (removeDuplicates\ p)\ x$   
 $\langle proof \rangle$

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

$p \neq [] \implies Cp\ (list2FWpolicy\ p)\ x = Cp\ (list2FWpolicy\ (removeAllDuplicates\ p))\ x$   
 $\langle proof \rangle$

**lemma** *C-eq-RAD*:

$p \neq [] \implies Cp\ (list2FWpolicy\ p) = Cp\ (list2FWpolicy\ (removeAllDuplicates\ p))$   
 $\langle proof \rangle$

**lemma** *C-eq-compile*:

$DenyAll \in set\ (policy2list\ p) \implies all-in-list\ (policy2list\ p)\ l \implies$   
 $allNetsDistinct\ (policy2list\ p) \implies$   
 $Cp\ (list2FWpolicy\ (removeAllDuplicates\ (insertDenies\ (separate$   
 $(sort\ (removeShadowRules2\ (remdups\ (rm-MT-rules\ Cp\ (insertDeny$   
 $(removeShadowRules1\ (policy2list\ p)))))) l)))) = Cp\ p$   
 $\langle proof \rangle$

**lemma** *C-eq-compileQ*:

$DenyAll \in set\ (policy2list\ p) \implies all-in-list\ (policy2list\ p)\ l \implies allNetsDis-$   
 $tinct\ (policy2list\ p) \implies$   
 $Cp\ (list2FWpolicy\ (removeAllDuplicates\ (insertDenies\ (separate\ (qsort$   
 $(removeShadowRules2\ (remdups\ (rm-MT-rules\ Cp\ (insertDeny$   
 $(removeShadowRules1\ (policy2list\ p)))))) l)))) = Cp\ p$   
 $\langle proof \rangle$

**lemma** *C-eq-normalizePr*:

$DenyAll \in set\ (policy2list\ p) \implies allNetsDistinct\ (policy2list\ p) \implies$   
 $all-in-list\ (policy2list\ p)\ (Nets-List\ p) \implies$   
 $Cp\ (list2FWpolicy\ (normalizePr\ p)) = Cp\ p$

$\langle \text{proof} \rangle$

**lemma** *C-eq-normalizePrQ*:

$\text{DenyAll} \in \text{set} (\text{policy2list } p) \implies \text{allNetsDistinct} (\text{policy2list } p) \implies$   
 $\text{all-in-list} (\text{policy2list } p) (\text{Nets-List } p) \implies$   
 $\text{Cp} (\text{list2FWpolicy} (\text{normalizePrQ } p)) = \text{Cp } p$   
 $\langle \text{proof} \rangle$

**lemma** *domSubset3*:  $\text{dom} (\text{Cp} (\text{DenyAll} \oplus x)) = \text{dom} (\text{Cp} (\text{DenyAll}))$

$\langle \text{proof} \rangle$

**lemma** *domSubset4*:

$\text{dom} (\text{Cp} (\text{DenyAllFromTo } x \ y \oplus \text{DenyAllFromTo } y \ x \oplus \text{AllowPortFromTo } x \ y \ \text{dn})) =$   
 $\text{dom} (\text{Cp} (\text{DenyAllFromTo } x \ y \oplus \text{DenyAllFromTo } y \ x))$   
 $\langle \text{proof} \rangle$

**lemma** *domSubset5*:

$\text{dom} (\text{Cp} (\text{DenyAllFromTo } x \ y \oplus \text{DenyAllFromTo } y \ x \oplus \text{AllowPortFromTo } y \ x \ \text{dn})) =$   
 $\text{dom} (\text{Cp} (\text{DenyAllFromTo } x \ y \oplus \text{DenyAllFromTo } y \ x))$   
 $\langle \text{proof} \rangle$

**lemma** *domSubset1*:

$\text{dom} (\text{Cp} (\text{DenyAllFromTo } \text{one } \ \text{two} \oplus \text{DenyAllFromTo } \text{two } \ \text{one} \oplus \text{AllowPortFromTo } \text{one } \ \text{two} \ \text{dn} \oplus x)) =$   
 $\text{dom} (\text{Cp} (\text{DenyAllFromTo } \text{one } \ \text{two} \oplus \text{DenyAllFromTo } \text{two } \ \text{one} \oplus x))$   
 $\langle \text{proof} \rangle$

**lemma** *domSubset2*:

$\text{dom} (\text{Cp} (\text{DenyAllFromTo } \text{one } \ \text{two} \oplus \text{DenyAllFromTo } \text{two } \ \text{one} \oplus \text{AllowPortFromTo } \text{two } \ \text{one} \ \text{dn} \oplus x)) =$   
 $\text{dom} (\text{Cp} (\text{DenyAllFromTo } \text{one } \ \text{two} \oplus \text{DenyAllFromTo } \text{two } \ \text{one} \oplus x))$   
 $\langle \text{proof} \rangle$

**lemma** *ConcAssoc2*:  $\text{Cp} (X \oplus Y \oplus ((A \oplus B) \oplus D)) = \text{Cp} (X \oplus Y \oplus A \oplus B \oplus D)$

$\langle \text{proof} \rangle$

**lemma** *ConcAssoc3*:  $\text{Cp} (X \oplus ((Y \oplus A) \oplus D)) = \text{Cp} (X \oplus Y \oplus A \oplus D)$

$\langle \text{proof} \rangle$

**lemma** *RS3-NMT[rule-format]*:  $\text{DenyAll} \in \text{set } p \longrightarrow$

$\text{rm-MT-rules } \text{Cp } p \neq []$

$\langle \text{proof} \rangle$

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

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

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

**lemmas** *domain-reasoningPr* = *domDA ConcAssoc2 domSubset1 domSubset2*  
*domSubset3 domSubset4 domSubset5 domSubsetDistr1*  
*domSubsetDistr2 domSubsetDistrA domSubsetDistrD coerc-assoc ConcAssoc*  
*ConcAssoc3*

The following lemmas help with the normalisation

**lemma** *list2policyR-Start*[*rule-format*]:  $p \in \text{dom} (\text{Cp } a) \longrightarrow$   
 $\text{Cp} (\text{list2policyR } (a \# \text{list})) p = \text{Cp } a p$   
 ⟨proof⟩

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

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

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

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

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

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

$\langle \text{proof} \rangle$

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

**lemma**  $p\text{-eq2}[rule\text{-format}]$ :  $normalizePr x \neq [] \longrightarrow$   
 $Cp (list2FWpolicy (normalizePr x)) = Cp x \longrightarrow$   
 $list2policy (map Cp (rev (normalizePr x))) = Cp x$   
 $\langle \text{proof} \rangle$

**lemma**  $p\text{-eq2Q}[rule\text{-format}]$ :  $normalizePrQ x \neq [] \longrightarrow$   
 $Cp (list2FWpolicy (normalizePrQ x)) = Cp x \longrightarrow$   
 $list2policy (map Cp (rev (normalizePrQ x))) = Cp x$   
 $\langle \text{proof} \rangle$

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

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

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

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

**lemma**  $p\text{-}eq2\text{-}manualQ[rule\text{-format}]$ :  
 $normalize\text{-}manual\text{-}orderPrQ x l \neq [] \longrightarrow$   
 $Cp (list2FWpolicy (normalize\text{-}manual\text{-}orderPrQ x l)) = Cp x \longrightarrow$   
 $list2policy (map Cp (rev (normalize\text{-}manual\text{-}orderPrQ x l))) = Cp x$   
 $\langle \text{proof} \rangle$

**lemma**  $norm\text{-}notMT\text{-}manualQ$ :  $DenyAll \in set (policy2list p) \implies normal\text{-}$

*ize-manual-orderPrQ p l ≠ []*  
 ⟨proof⟩

**lemma** *C-eq-normalizePr-manualQ*:

*DenyAll ∈ set (policy2list p) ⇒*  
*allNetsDistinct (policy2list p) ⇒*  
*all-in-list (policy2list p) l ⇒*  
*Cp (list2FWpolicy (normalize-manual-orderPrQ p l)) = Cp p*  
 ⟨proof⟩

**lemma** *p-eq2-manual[rule-format]: normalize-manual-orderPr x l ≠ [] →*

*Cp (list2FWpolicy (normalize-manual-orderPr x l)) = Cp x →*  
*list2policy (map Cp (rev (normalize-manual-orderPr x l))) = Cp x*  
 ⟨proof⟩

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

⟨proof⟩

As an example, how this theorems can be used for a concrete normalisation instantiation.

**lemma** *normalizePrNAT*:

*DenyAll ∈ set (policy2list Filter) ⇒*  
*allNetsDistinct (policy2list Filter) ⇒*  
*all-in-list (policy2list Filter) (Nets-List Filter) ⇒*  
*((λ (x,y). x) o-f (((NAT ⊗<sub>2</sub> Cp Filter) o (λx. (x,x)))))) =*  
*list2policy (((NAT ⊗<sub>L</sub> (map Cp (rev (normalizePr Filter)))) (⊗<sub>2</sub>) (λ (x,y). x) (λ x.*  
*(x,x)))*  
 ⟨proof⟩

**lemma** *domSimpl[simp]: dom (Cp (A ⊕ DenyAll)) = dom (Cp (DenyAll))*

⟨proof⟩

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

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

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

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

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

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

```

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 ∧ ids1 i xs)
|ids1 i [] = True

```

```

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

```

```

definition applyPolicy:: ('i × ('i ↦ 'o)) ↦ 'o
where   applyPolicy = (λ (x,z). z x)

```

**end**

## 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**

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

**definition**

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

**definition**

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

**definition**

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

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

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

**The protocol policy specification**

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

Of course, not every message changes the policy. In such cases, we do not have to model different cases, one is enough. In our example, only messages 2 and 4 need special

transitions. The default says that if the policy accepts the packet, it is added to the history, otherwise it is simply dropped. The policy remains the same in both cases.

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

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

| *last-opened-port*  $i$   $(x\#xs) = last\text{-}opened\text{-}port\ i\ xs$

| *last-opened-port*  $x\ [] = undefined$

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

**where**

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

| *FTP-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-STA*  $(p, s) = Some\ (p\#\text{(fst}\ s),\ snd\ s)$

**fun** *FTP-STD* ::  $((adr_{ip},msg)\ history, adr_{ip}, msg)\ FWStateTransition$

**where** *FTP-STD*  $(p,s) = Some\ s$

**definition** *TRPolicy* ::  $(adr_{ip},msg)\ packet \times (adr_{ip},msg)\ history \times ((adr_{ip},msg)\ packet$   
 $\mapsto unit)$

$\mapsto (unit \times (adr_{ip},msg)\ history \times ((adr_{ip},msg)\ packet \mapsto$   
 $unit))$

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

**definition** *TRPolicyMon*

**where** *TRPolicyMon* = *policy2MON*(*TRPolicy*)

If required to contain the policy in the output

**definition** *TRPolicyMon'*

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

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**  $ftp\text{-}states = S0 \mid S1 \mid S2 \mid S3$

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

**fun**

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

**where**

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

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

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

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

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

**definition**

$ftp\text{-}2\text{-}interleaved :: adr_{ip} \ src \Rightarrow adr_{ip} \ dest \Rightarrow id \Rightarrow port \Rightarrow$

```

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

```

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

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

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

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

**lemma port-open-lemma:**  $\neg (Ex (\text{port-open } [] (x::port)))$   
 ⟨proof⟩

**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<sub>ip</sub>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 ⇒ 'a **where** POL x = x

Variant 2 takes the policy into the output

**fun** FTP-STP ::

$((id \rightarrow port), adr_{ip}, msg) \text{ FWStateTransitionP}$   
**where**

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

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

$|FTP\text{-}STP p x = (if\ p\text{-}accept\ p\ (snd\ x)$   
 $then\ Some\ (allow\ (POL\ (snd\ x)),((fst\ x),snd\ x))$   
 $else\ Some\ (deny\ (POL\ (snd\ x)),(fst\ x,snd\ x))$

**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**

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

**definition**

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

**definition**

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

**definition**

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

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

**definition**

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

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

**fun**

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

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**

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

**fun** *VOIP-STA* ::

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

$VOIP\text{-}STA\ ((a,c,d,ARQ), (InL, policy)) =$

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

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

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

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

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

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

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 | S1 | S2 | S3 | S4 | S5

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

**fun is-voip** :: voip-states  $\Rightarrow$  address  $\Rightarrow$  address  $\Rightarrow$  address  $\Rightarrow$  id  $\Rightarrow$  port  $\Rightarrow$   
 port  $\Rightarrow$  (adr<sub>ip</sub>, 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) =  
 ((( $\lambda$  (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  $\wedge$

$$(H = S0 \wedge sr = (s, 1720) \wedge de = (d, 1720) \wedge co = Fin \wedge is-voip\ S1\ s\ d\ g\ i\ p1\ p2\ InL))))))\ x)$$

Finally, *NB-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*  $\Rightarrow$   
 (*adr<sub>ip</sub>*, *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<sub>ip</sub>*, *ftpvoip*) *packet*  $\Rightarrow$  *bool* **where**  
*FTPVOIP-is-init* = ( $\lambda$  *i p*. (*id p* = *i*  $\wedge$  *content p* = *ftp-init*))

**definition**

*FTPVOIP-is-port-request* :: *id*  $\Rightarrow$  *port*  $\Rightarrow$  (*adr<sub>ip</sub>*, *ftpvoip*) *packet*  $\Rightarrow$  *bool* **where**

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

**definition**

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

**definition**

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

**definition**

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

**definition**

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

**fun FTPVOIP-are-other where**

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

**fun last-opened-port where**

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

**fun FTPVOIP-FTP-STA ::**

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

**where**

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

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

*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<sub>ip</sub>*, *ftpvoip*) *history*, *adr<sub>ip</sub>*, *ftpvoip*) *FWStateTransition*

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

**definition**

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

**definition**

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

**definition**

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

**definition**

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

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

**definition**

*FTPVOIP-ports-open* :: *id*  $\Rightarrow$  *port*  $\times$  *port*  $\Rightarrow$  (*adr<sub>ip</sub>*, *ftpvoip*) *history*  $\Rightarrow$  *bool* **where**  
*FTPVOIP-ports-open* *i p L* = ((*not-before* (*FTPVOIP-is-fin* *i*) (*FTPVOIP-is-setup* *i* (*fst p*)) *L*)  $\wedge$   
*not-before* (*FTPVOIP-is-fin* *i*) (*FTPVOIP-is-connect* *i* (*snd p*))  
*L*)

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

**fun**

*FTPVOIP-src-is-initiator* :: *id*  $\Rightarrow$  *adr<sub>ip</sub>*  $\Rightarrow$  (*adr<sub>ip</sub>*,*ftpvoip*) *history*  $\Rightarrow$  *bool* **where**  
*FTPVOIP-src-is-initiator* *i a* [] = *False*  
 $|FTPVOIP-src-is-initiator$  *i a* (*p#S*) = (((*id p* = *i*)  $\wedge$   
 $(\exists$  *port*. *content p* = *Setup port*)  $\wedge$

$$((fst (src p) = fst a)) \vee \\ (FTPVOIP-src-is-initiator i a S)$$

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

**fun** *FTPVOIP-VOIP-STA* ::

*((adr<sub>ip</sub>, ftpvoip) history, adr<sub>ip</sub>, ftpvoip)* *FWStateTransition*

**where**

*FTPVOIP-VOIP-STA* *((a,c,d,ARQ), (InL, policy))* =

*Some* *((a,c,d, ARQ)#InL,*

*(allow-from-to-port (1719::port)(subnet-of d) (subnet-of c))*  $\oplus$  *policy*)

*|FTPVOIP-VOIP-STA* *((a,c,d,ARJ), (InL, policy))* =

*(if (not-before (FTPVOIP-is-fin a) (FTPVOIP-is-arq a) InL)*

*then Some* *((a,c,d,ARJ)#InL,*

*deny-from-to-port (14::port) (subnet-of c) (subnet-of d)*  $\oplus$  *policy*)

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

*|FTPVOIP-VOIP-STA* *((a,c,d,ACF callee), (InL, policy))* =

*Some* *((a,c,d,ACF callee)#InL,*

*allow-from-to-port (1720::port) (subnet-of-adr callee) (subnet-of d)*  $\oplus$

*allow-from-to-port (1720::port) (subnet-of d) (subnet-of-adr callee)*  $\oplus$

*deny-from-to-port (1719::port) (subnet-of d) (subnet-of c)*  $\oplus$

*policy*)

*|FTPVOIP-VOIP-STA* *((a,c,d, Setup port), (InL, policy))* =

*Some* *((a,c,d,Setup port)#InL,*

*allow-from-to-port port (subnet-of d) (subnet-of c)*  $\oplus$  *policy*)

*|FTPVOIP-VOIP-STA* *((a,c,d, ftpvoip.Connect port), (InL, policy))* =

*Some* *((a,c,d,ftpvoip.Connect port)#InL,*

*allow-from-to-port port (subnet-of d) (subnet-of c)*  $\oplus$  *policy*)

*|FTPVOIP-VOIP-STA* *((a,c,d,Fin), (InL,policy))* =

*(if*  $\exists$  *p1 p2. FTPVOIP-ports-open a (p1,p2) InL* *then* (

*(if* *FTPVOIP-src-is-initiator a c InL*

*then* *(Some* *((a,c,d,Fin)#InL,*

*(deny-from-to-port (1720::int) (subnet-of c) (subnet-of d) )*  $\oplus$

*(deny-from-to-port (snd (SOME p. FTPVOIP-ports-open a p InL))*

*(subnet-of c) (subnet-of d))*  $\oplus$

*(deny-from-to-port (fst (SOME p. FTPVOIP-ports-open a p InL))*

*(subnet-of d) (subnet-of c))*  $\oplus$  *policy*))

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

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

**fun** *FTPVOIP-VOIP-STD* ::  
 ((*adr<sub>ip</sub>*, *ftpvoip*) *history*, *adr<sub>ip</sub>*, *ftpvoip*) *FWStateTransition*  
**where**  
     *FTPVOIP-VOIP-STD* (*p,s*) = *Some s*

**definition** *FTP-VOIP-STA* :: ((*adr<sub>ip</sub>*, *ftpvoip*) *history*, *adr<sub>ip</sub>*, *ftpvoip*) *FWStateTransition*  
*sition*  
**where**  
     *FTP-VOIP-STA* = (( $\lambda(x,x).$  *Some x*)  $\circ_m$  ((*FTPVOIP-FTP-STA*  $\otimes_S$  *FTPVOIP-VOIP-STA*  $\circ$  ( $\lambda (p,x).$  (*p,x,x*))))))

**definition** *FTP-VOIP-STD* :: ((*adr<sub>ip</sub>*, *ftpvoip*) *history*, *adr<sub>ip</sub>*, *ftpvoip*) *FWStateTransition*  
*sition*  
**where**  
     *FTP-VOIP-STD* = ( $\lambda(x,x).$  *Some x*)  $\circ_m$  ((*FTPVOIP-FTP-STD*  $\otimes_S$  *FTPVOIP-VOIP-STD*  $\circ$  ( $\lambda (p,x).$  (*p,x,x*))))

**definition** *FTPVOIP-TRPolicy* **where**  
     *FTPVOIP-TRPolicy* = *policy2MON* (  
         (((*FTP-VOIP-STA,FTP-VOIP-STD*)  $\otimes_{\nabla}$  *applyPolicy*)  $\circ$  ( $\lambda (x,(y,z)).$   
         ((*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 adr<sub>ip</sub>Lemmas*

**datatype** *ftp-states2* = *FS0* | *FS1* | *FS2* | *FS3*  
**datatype** *voip-states2* = *V0* | *V1* | *V2* | *V3* | *V4* | *V5*

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

**fun** *FTPVOIP-is-voip* :: *voip-states2*  $\Rightarrow$  *address*  $\Rightarrow$  *address*  $\Rightarrow$  *address*  $\Rightarrow$  *id*  $\Rightarrow$  *port*  $\Rightarrow$   
*port*  $\Rightarrow$  (*adr<sub>ip</sub>*, *ftpvoip*) *history*  $\Rightarrow$  *bool*

**where**

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

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

**definition**

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

**fun**

*FTPVOIP-is-ftp* :: *ftp-states2*  $\Rightarrow$  *adr<sub>ip</sub>*  $\Rightarrow$  *adr<sub>ip</sub>*  $\Rightarrow$  *id*  $\Rightarrow$  *port*  $\Rightarrow$   
(*adr<sub>ip</sub>*, *ftpvoip*) *history*  $\Rightarrow$  *bool*

**where**

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

$$\begin{aligned}
& (H=FS2 \wedge sr = c \wedge de = s \wedge co = ftp-init \wedge FTPVOIP-is-ftp FS3 c s i p InL) \vee \\
& (H=FS1 \wedge sr = c \wedge de = s \wedge co = ftp-port-request p \wedge FTPVOIP-is-ftp FS2 c s i \\
& p InL) \vee \\
& (H=FS1 \wedge sr = s \wedge de = (fst c,p) \wedge co = ftp-data \wedge FTPVOIP-is-ftp FS1 c s i p \\
& InL) \vee \\
& (H=FS0 \wedge sr = c \wedge de = s \wedge co = ftp-close \wedge FTPVOIP-is-ftp FS1 c s i p InL) \\
& )))) x))
\end{aligned}$$

**definition**

*FTPVOIP-NB-ftp* ::  $adr_{ip} src \Rightarrow adr_{ip} dest \Rightarrow id \Rightarrow port \Rightarrow (adr_{ip}, ftpvoip)$  history set **where**

*FTPVOIP-NB-ftp*  $s d i p = \{x. (FTPVOIP-is-ftp FS0 s d i p x)\}$

**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* *<proof>*  
**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 mailservers 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:
- Intranet (Company intern network)
  - DMZ (demilitarised zone, servers, etc), containing at least two distinct servers “mail” and “web”
  - Internet (“all others”)
- Policy:
- allow http(s) from Intranet to Internet
  - deny all traffic from Internet to Intranet
  - allow imaps and smtp from intranet to mailservers
  - allow smtp from Internet to mailservers
  - allow http(s) from Internet to webservers
  - deny everything else

**definition**

*intranet* :: *adr<sub>ip</sub> net* **where**  
*intranet* = { {(a,b) . (a > 1 ∧ a < 4) } }

**definition**

*dmz* :: *adr<sub>ip</sub> net* **where**  
*dmz* = { {(a,b) . (a > 6) ∧ (a < 11) } }

**definition**

*mail* :: *adr<sub>ip</sub> net* **where**  
*mail* = { {(a,b) . a = 7 } }

**definition**

*web* :: *adr<sub>ip</sub> net* **where**  
*web* = { {(a,b) . a = 8 } }

**definition**

*internet* :: *adr<sub>ip</sub> net* **where**  
*internet* = { {(a,b) . ¬ ( (a > 1 ∧ a < 4) ∨ (a > 6) ∧ (a < 11) ) } }

**definition**

*Intranet-mail-port* :: (*adr<sub>ip</sub>, 'b*) *FWPolicy* **where**  
*Intranet-mail-port* = (*allow-from-to-ports* {21::port,14} *intranet mail*)

**definition**

*Intranet-Internet-port* :: (*adr<sub>ip</sub>, 'b*) *FWPolicy* **where**  
*Intranet-Internet-port* = *allow-from-to-ports* {80::port,90} *intranet internet*

**definition**

*Internet-web-port* :: (*adr<sub>ip</sub>, 'b*) *FWPolicy* **where**  
*Internet-web-port* = (*allow-from-to-ports* {80::port,90} *internet web*)

**definition**

*Internet-mail-port* :: (*adr<sub>ip</sub>, 'b*) *FWPolicy* **where**  
*Internet-mail-port* = (*allow-all-from-port-to internet* (21::port) *dmz* )

**definition**

*policyPort* :: (*adr<sub>ip</sub>, 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 mailservers and the dmz. Therefore, the constraint looks as follows.

**definition**

*not-in-same-net* :: (*adr<sub>ip</sub>, DummyContent*) *packet* ⇒ *bool* **where**

$$\begin{aligned} \text{not-in-same-net } x = & ((\text{src } x \sqsubset \text{internet} \longrightarrow \neg \text{dest } x \sqsubset \text{internet}) \wedge \\ & (\text{src } x \sqsubset \text{intranet} \longrightarrow \neg \text{dest } x \sqsubset \text{intranet}) \wedge \\ & (\text{src } x \sqsubset \text{dmz} \longrightarrow \neg \text{dest } x \sqsubset \text{dmz})) \end{aligned}$$

```

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**

$PC :: (\text{adr}_{ip} \text{ net}) \text{ where}$   
 $PC = \{\{(a,b). a = 3\}\}$

**definition**

*Internet* :: *adr<sub>ip</sub>* net **where**  
*Internet* =  $\{(a,b). \neg (a = 3)\}$

**definition**

*not-in-same-net* :: (*adr<sub>ip</sub>*,*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<sub>ip</sub>*,*DummyContent*) *FWPolicy* **where**  
*strictPolicy* = *deny-all* ++ *allow-all-from-to* *PC* *Internet*

**definition**

*PortPolicy* :: (*adr<sub>ip</sub>*,*DummyContent*) *FWPolicy* **where**  
*PortPolicy* = *deny-all* ++ *allow-from-ports-to* {*http*,*smtp*,*ftp*} *PC* *Internet*

**definition**

*PortPolicyBig* :: (*adr<sub>ip</sub>*,*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-def* *PortPolicy-def* *PC-def*  
*Internet-def* *PortPolicyBig-def* *src-def* *dest-def*  
*adr<sub>ip</sub>Lemmas* *content-def*  
*PortCombinators* *in-subnet-def* *PortPolicyBig-def* *id-def*

**declare** *Ports* [*simp* *add*]

**definition** *wellformed-packet*::(*adr<sub>ip</sub>*,*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*) **where**

*PC* =  $\{\{(a,b,c,d),e\}. a = 1 \wedge b = 3 \wedge c = 5 \wedge d = 2\}\}$

**definition**

*Internet* :: *ipv4 net* **where**

*Internet* =  $\{\{(a,b,c,d),e\}. \neg (a = 1 \wedge b = 3 \wedge c = 5 \wedge d = 2)\}\}$

**definition**

*not-in-same-net* :: (*ipv4, 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* :: (*ipv4, DummyContent*) *FWPolicy* **where**

*strictPolicy* = *deny-all ++ allow-all-from-to PC Internet*

**definition**

*PortPolicy* :: (*ipv4, DummyContent*) *FWPolicy* **where**

*PortPolicy* = *deny-all ++ allow-from-ports-to {80::port,24,21} PC Internet*

**definition**

*PortPolicyBig* :: (*ipv4, DummyContent*) *FWPolicy* **where**

*PortPolicyBig* = *deny-all ++ allow-from-port-to (80::port) PC Internet ++ allow-from-port-to (24::port) PC Internet ++ allow-from-port-to (21::port) PC Internet*

**lemmas** *policyLemmas* = *strictPolicy-def PortPolicy-def PC-def*

*Internet-def PortPolicyBig-def src-def dest-def*

*IPv4.src-port*

*IPv4.dest-port PolicyCombinators*

*PortCombinators in-subnet-def PortPolicyBig-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 <proof>

```

**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 PortPoli-
cyBig-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 :: adrip net where
FWLink = { {(a,b). a = 1} }

```

```

definition
any :: adrip net where
any = { {(a,b). a > 5} }

```

```

definition
i4 :: adrip net where
i4 = { {(a,b). a = 2 } }

```

```

definition
i27 :: adrip net where
i27 = { {(a,b). a = 3 } }

```

**definition**

*eth-intern*:: *adr<sub>ip</sub> net* **where**  
*eth-intern* =  $\{\{(a,b). a = 4\}\}$

**definition**

*eth-private*:: *adr<sub>ip</sub> net* **where**  
*eth-private* =  $\{\{(a,b). a = 5\}\}$

**definition**

*MG2* :: (*adr<sub>ip</sub> net,port*) *Combinators* **where**  
*MG2* = *AllowPortFromTo i27 any 1*  $\oplus$   
*AllowPortFromTo i27 any 2*  $\oplus$   
*AllowPortFromTo i27 any 3*

**definition**

*MG3* :: (*adr<sub>ip</sub> net,port*) *Combinators* **where**  
*MG3* = *AllowPortFromTo any FWLink 1*

**definition**

*MG4* :: (*adr<sub>ip</sub> net,port*) *Combinators* **where**  
*MG4* = *AllowPortFromTo FWLink FWLink 4*

**definition**

*MG7* :: (*adr<sub>ip</sub> net,port*) *Combinators* **where**  
*MG7* = *AllowPortFromTo FWLink i4 6*  $\oplus$   
*AllowPortFromTo FWLink i4 7*

**definition**

*MG8* :: (*adr<sub>ip</sub> net,port*) *Combinators* **where**  
*MG8* = *AllowPortFromTo FWLink i4 6*  $\oplus$   
*AllowPortFromTo FWLink i4 7*

**definition**

*DG3*:: (*adr<sub>ip</sub> net,port*) *Combinators* **where**  
*DG3* = *AllowPortFromTo any any 7*

**definition**

*Policy* = *DenyAll*  $\oplus$  *MG8*  $\oplus$  *MG7*  $\oplus$  *MG4*  $\oplus$  *MG3*  $\oplus$  *MG2*  $\oplus$  *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<sub>ip</sub>*, *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*}

⟨*proof*⟩

**lemma** *sets-distinct2*: (*m*::*int*) ≠ *n* ⇒ {(*a*,*b*). *a* = *n*} ≠ {(*a*,*b*). *a* = *m*}

⟨*proof*⟩

**lemma** *sets-distinct3*: {((*a*::*int*),(*b*::*int*)). *a* = *n*} ≠ {(*a*,*b*). *a* > *n*}

⟨*proof*⟩

**lemma** *sets-distinct4*: {((*a*::*int*),(*b*::*int*)). *a* > *n*} ≠ {(*a*,*b*). *a* = *n*}

⟨*proof*⟩

**lemma** *aux*: [ *a* ∈ *c*; *a* ∉ *d*; *c* = *d* ] ⇒ *False*

⟨*proof*⟩

**lemma** *sets-distinct5*: (*s*::*int*) < *g* ⇒ {(*a*::*int*, *b*::*int*). *a* = *s*} ≠ {(*a*::*int*, *b*::*int*). *g* < *a*}

⟨*proof*⟩

**lemma sets-distinct6:**  $(s::int) < g \implies \{(a::int, b::int). g < a\} \neq \{(a::int, b::int). a = s\}$   
 <proof>

**lemma distinctNets:**  $FWLink \neq any \wedge FWLink \neq i4 \wedge FWLink \neq i27 \wedge FWLink \neq eth-intern \wedge FWLink \neq eth-private \wedge$   
 $any \neq FWLink \wedge any \neq i4 \wedge any \neq i27 \wedge any \neq eth-intern \wedge any \neq eth-private$   
 $\wedge i4 \neq FWLink \wedge$   
 $i4 \neq any \wedge i4 \neq i27 \wedge i4 \neq eth-intern \wedge i4 \neq eth-private \wedge i27 \neq FWLink \wedge i27$   
 $\neq any \wedge$   
 $i27 \neq i4 \wedge i27 \neq eth-intern \wedge i27 \neq eth-private \wedge eth-intern \neq FWLink \wedge eth-intern$   
 $\neq any \wedge$   
 $eth-intern \neq i4 \wedge eth-intern \neq i27 \wedge eth-intern \neq eth-private \wedge eth-private \neq$   
 $FWLink \wedge$   
 $eth-private \neq any \wedge eth-private \neq i4 \wedge eth-private \neq i27 \wedge eth-private \neq eth-intern$   
 <proof>

**lemma aux5:**  $\llbracket x \neq a; y \neq b; (x \neq y \wedge x \neq b) \vee (a \neq b \wedge a \neq y) \rrbracket \implies \{x,a\} \neq \{y,b\}$   
 <proof>

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

**lemma ANDex:** *allNetsDistinct (policy2list Policy)*  
 <proof>

**fun** (sequential) *numberOfRules* **where**  
*numberOfRules* (a $\oplus$ b) = *numberOfRules* a + *numberOfRules* b  
*numberOfRules* a = (1::int)

**fun** *numberOfRulesList* **where**  
*numberOfRulesList* (x#xs) = ((*numberOfRules* x)#(*numberOfRulesList* xs))  
*numberOfRulesList* [] = []

**lemma all-in-list:** *all-in-list (policy2list Policy) (Nets-List Policy)*  
 <proof>

**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<sub>ip</sub> net* **where**

$FWLink = \{(a,b). a = 1\}$

**definition**

*any* :: *adr<sub>ip</sub> net* **where**

$any = \{(a,b). a > 5\}$

**definition**

*i4-32* :: *adr<sub>ip</sub> net* **where**

$i4-32 = \{(a,b). a = 2\}$

**definition**

*i10-32* :: *adr<sub>ip</sub> net* **where**

$i10-32 = \{(a,b). a = 3\}$

**definition**

*eth-intern* :: *adr<sub>ip</sub> net* **where**

$eth-intern = \{(a,b). a = 4\}$

**definition**

*eth-private* :: *adr<sub>ip</sub> net* **where**

$eth-private = \{(a,b). a = 5\}$

**definition**

*D1a* :: (*adr<sub>ip</sub> net, port*) *Combinators* **where**

$D1a = AllowPortFromTo\ eth-intern\ any\ 1 \oplus$

$AllowPortFromTo\ eth-intern\ any\ 2$

**definition**

*D1b* :: (*adr<sub>ip</sub> net, port*) *Combinators* **where**

$D1b = AllowPortFromTo\ eth-private\ any\ 1 \oplus$

$AllowPortFromTo\ eth-private\ any\ 2$

**definition**

*D2a* :: (*adr<sub>ip</sub> net, port*) *Combinators* **where**

$D2a = AllowPortFromTo \text{ any } i4-32 \ 21$

**definition**

$D2b :: (adr_{ip} \ net, \ port) \ Combinators \ \mathbf{where}$

$D2b = AllowPortFromTo \text{ any } i10-32 \ 21 \oplus$

$AllowPortFromTo \text{ any } i10-32 \ 43$

**definition**

$Policy :: (adr_{ip} \ net, \ port) \ Combinators \ \mathbf{where}$

$Policy = DenyAll \oplus D2b \oplus D2a \oplus D1b \oplus D1a$

**lemmas**  $PolicyLemmas = Policy-def \ D1a-def \ D1b-def \ D2a-def \ D2b-def$

**lemmas**  $PolicyL = Policy-def$

$FWLink-def$

$any-def$

$i10-32-def$

$i4-32-def$

$eth-intern-def$

$eth-private-def$

$D1a-def \ D1b-def \ D2a-def \ D2b-def$

**consts**  $fixID :: id$

**consts**  $fixContent :: DummyContent$

**definition**  $fixElements \ p = (id \ p = fixID \wedge \ content \ p = fixContent)$

**lemmas**  $fixDefs = fixElements-def \ NetworkCore.id-def \ NetworkCore.content-def$

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

$\langle proof \rangle$

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

$\langle proof \rangle$

**lemma**  $sets-distinct3: \{((a::int),(b::int)). \ a = n\} \neq \{(a,b). \ a > n\}$

$\langle proof \rangle$

**lemma**  $sets-distinct4: \{((a::int),(b::int)). \ a > n\} \neq \{(a,b). \ a = n\}$

$\langle proof \rangle$

**lemma**  $aux: \llbracket a \in c; \ a \notin d; \ c = d \rrbracket \implies False$

$\langle proof \rangle$

**lemma** *sets-distinct5*:  $(s::int) < g \implies \{(a::int, b::int). a = s\} \neq \{(a::int, b::int). g < a\}$   
 ⟨proof⟩

**lemma** *sets-distinct6*:  $(s::int) < g \implies \{(a::int, b::int). g < a\} \neq \{(a::int, b::int). a = s\}$   
 ⟨proof⟩

**lemma** *distinctNets*:  $FWLink \neq any \wedge FWLink \neq i4-32 \wedge FWLink \neq i10-32 \wedge FWLink \neq eth-intern \wedge FWLink \neq eth-private \wedge any \neq FWLink \wedge any \neq i4-32 \wedge any \neq i10-32 \wedge any \neq eth-intern \wedge any \neq eth-private \wedge i4-32 \neq FWLink \wedge i4-32 \neq any \wedge i4-32 \neq i10-32 \wedge i4-32 \neq eth-intern \wedge i4-32 \neq eth-private \wedge i10-32 \neq FWLink \wedge i10-32 \neq any \wedge i10-32 \neq i4-32 \wedge i10-32 \neq eth-intern \wedge i10-32 \neq eth-private \wedge eth-intern \neq FWLink \wedge eth-intern \neq any \wedge eth-intern \neq i4-32 \wedge eth-intern \neq i10-32 \wedge eth-intern \neq eth-private \wedge eth-private \neq FWLink \wedge eth-private \neq any \wedge eth-private \neq i4-32 \wedge eth-private \neq i10-32 \wedge eth-private \neq eth-intern$   
 ⟨proof⟩

**lemma** *aux5*:  $\llbracket x \neq a; y \neq b; (x \neq y \wedge x \neq b) \vee (a \neq b \wedge a \neq y) \rrbracket \implies \{x, a\} \neq \{y, b\}$   
 ⟨proof⟩

**lemma** *aux2*:  $\{a, b\} = \{b, a\}$   
 ⟨proof⟩

**lemma** *ANDex*: *allNetsDistinct* (*policy2list* *Policy*)  
 ⟨proof⟩

**fun** (*sequential*) *numberOfRules* **where**  
*numberOfRules* ( $a \oplus b$ ) = *numberOfRules* *a* + *numberOfRules* *b*  
 | *numberOfRules* *a* = (1::int)

**fun** *numberOfRulesList* **where**  
*numberOfRulesList* ( $x \# xs$ ) = ((*numberOfRules* *x*)#(*numberOfRulesList* *xs*))  
 | *numberOfRulesList* [] = []

**lemma** *all-in-list*: *all-in-list* (*policy2list* *Policy*) (*Nets-List* *Policy*)  
 ⟨proof⟩

**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<sub>ip</sub> net* **where**

*subnet1* =  $\{(d,e). d > 1 \wedge d < 256\}$

**definition** *subnet2* :: *adr<sub>ip</sub> net* **where**

*subnet2* =  $\{(d,e). d > 500 \wedge d < 1256\}$

**definition**

*accross-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<sub>ip</sub>, DummyContent*) *FWPolicy* **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* **where**

*nat-0* = (*A<sub>f</sub>*( $\lambda x. \{x\}$ ))

**lemmas** *UnfoldPolicy0* =*filter-def nat-0-def*

*NATLemmas*

*ProtocolPortCombinators.ProtocolCombinators*

*adr<sub>ip</sub>Lemmas*

*packet-defs accross-subnets-def*

*subnet1-def subnet2-def*

**lemmas** *subnets* = *subnet1-def subnet2-def*

**definition** *Adr11* :: *int set*

**where** *Adr11* =  $\{d. d > 2 \wedge d < 3\}$

**definition** *Adr21* :: *int set* **where**

*Adr21* =  $\{d. d > 502 \wedge d < 503\}$

**definition** *nat-1* **where**

$nat-1 = nat-0 ++ (srcPat2pool-IntPort\ Adr11\ Adr21)$

**definition** *policy-1* **where**

$policy-1 = ((\lambda\ (x,y).\ x)\ o-f\ ((nat-1\ \otimes_2\ filter)\ o\ (\lambda\ x.\ (x,x))))$

**lemmas**  $UnfoldPolicy1 = UnfoldPolicy0\ nat-1-def\ Adr11-def\ Adr21-def\ policy-1-def$

**definition** *Adr12* **:: int 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**

$policy-7 = ((\lambda (x,y). x) \text{ o-f } ((nat-7 \otimes_2 filter) \text{ o } (\lambda x. (x,x))))$

**lemmas**  $UnfoldPolicy7 = UnfoldPolicy6 \text{ nat-7-def } Adr17-def \text{ } Adr27-def \text{ } policy-7-def$

**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**

$policy-8 = ((\lambda (x,y). x) \text{ o-f } ((nat-8 \otimes_2 filter) \text{ o } (\lambda x. (x,x))))$

**lemmas**  $UnfoldPolicy8 = UnfoldPolicy7 \text{ nat-8-def } Adr18-def \text{ } Adr28-def \text{ } policy-8-def$

**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**

$policy-9 = ((\lambda (x,y). x) \text{ o-f } ((nat-9 \otimes_2 filter) \text{ o } (\lambda x. (x,x))))$

**lemmas**  $UnfoldPolicy9 = UnfoldPolicy8 \text{ nat-9-def } Adr19-def \text{ } Adr29-def \text{ } policy-9-def$

**definition** *Adr110* **:: int set**

**where**  $Adr110 = \{d. d > 20 \wedge d < 30\}$

**definition** *Adr210* :: *int set* **where**  
*Adr210* = {*d*. *d* > 520 ∧ *d* < 530}

**definition** *nat-10* **where**  
*nat-10* = *nat-9* ++ (*srcPat2pool-IntPort* *Adr110* *Adr210*)

**definition** *policy-10* **where**  
*policy-10* = ((λ (*x,y*). *x*) o-f  
 ((*nat-10* ⊗<sub>2</sub> *filter*) o (λ *x*. (*x,x*))))

**lemmas** *UnfoldPolicy10* = *UnfoldPolicy9* *nat-10-def* *Adr110-def* *Adr210-def* *policy-10-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<sub>ip</sub> net* **where**  
*intranet* = {{(a,e) . a = 3}}

**definition**  
*internet* :: *adr<sub>ip</sub> net* **where**  
*internet* = {{(a,c) . a > 4}}

**definition**  
*gatekeeper* :: *adr<sub>ip</sub> net* **where**  
*gatekeeper* = {{(a,c) . a = 4}}

**definition**  
*voip-policy* :: (*adr<sub>ip</sub>,address voip-msg*) *FWPolicy* **where**  
*voip-policy* = *A<sub>U</sub>*

The next two constants check if an address is in the Intranet or in the Internet re-

spectively.

**definition**

*is-in-intranet* :: *address*  $\Rightarrow$  *bool* **where**  
*is-in-intranet* *a* = (*a* = 3)

**definition**

*is-gatekeeper* :: *address*  $\Rightarrow$  *bool* **where**  
*is-gatekeeper* *a* = (*a* = 4)

**definition**

*is-in-internet* :: *address*  $\Rightarrow$  *bool* **where**  
*is-in-internet* *a* = (*a* > 4)

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

**definition**

$\sigma$ -0-voip :: (*adr<sub>ip</sub>*, *address voip-msg*) *history*  $\times$   
(*adr<sub>ip</sub>*, *address voip-msg*) *FWPolicy*

**where**

$\sigma$ -0-voip = ( $\square$ , *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-voip* :: (*adr<sub>ip</sub>*, *address voip-msg*) *history*  $\Rightarrow$  *bool* **where**

*accept-voip* *t* = ( $\exists$  *c s g i p1 p2*. *t*  $\in$  *NB-voip c s g i p1 p2  $\wedge$  *is-in-intranet* *c*  
 $\wedge$  *is-in-internet* *s*  
 $\wedge$  *is-gatekeeper* *g*)*

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

*packet-with-id*  $\square$  *i* =  $\square$   
|*packet-with-id* (*x#xs*) *i* =  
(*if id* *x* = *i* *then* (*x#(packet-with-id xs i)*) *else* (*packet-with-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*  $\square$  = *True*

**lemmas** *ST-simps* = *Let-def valid-SE-def unit-SE-def bind-SE-def*

*subnet-of-int-def p-accept-def content-def*  
*is-in-intranet-def is-in-internet-def intranet-def internet-def exI*  
*subnetOf-lemma subnetOf-lemma2 subnetOf-lemma3 subnetOf-lemma4 voip-policy-def*  
*NetworkCore.id-def is-arq-def is-fin-def*  
*is-connect-def is-setup-def ports-open-def subnet-of-adr-def*

*VOIP.NB-voip-def  $\sigma$ -0-voip-def PLemmas VOIP-TRPolicy-def  
policy2MON-def applyPolicy-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.