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

Achim D. Brucker[*] Lukas Brügger[†] Burkhart Wolff[‡]

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

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

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

March 17, 2025

Abstract

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

Contents

1 Introduction

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.

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

where the enumeration type decision is

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

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

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

$$\begin{array}{lll}
\text{definition} & _ \lhd _ & :: \alpha\,\text{set} \Rightarrow \alpha \mapsto \beta \Rightarrow \alpha \mapsto \beta \\
\text{where} & S \lhd p & = \lambda x.\ \text{if}\, x \in S\, \text{then}\, p\, x\, \text{else}\, \bot \\
\text{definition} & _ \rhd _ & :: \alpha \mapsto \beta \Rightarrow \beta\ \text{decision set} \Rightarrow \alpha \mapsto \beta \\
\text{where} & p \rhd S & = \lambda x.\ \text{if} \big(\text{the}\,(p\, x)\big) \in S\, \text{then}\, p\, x\, \text{else}\, \bot
\end{array}$$

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*:

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

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

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

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

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[1].

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

- adr_i: An address in an Integer.
- adr_ip: An address is a pair of an Integer and a port (which is again an Integer).
- adr_ipp: An address is a triple consisting of two Integers modelling the IP address and the port number, and the specification of the network protocol
- IPv4: An address is a pair. The first element is a four-tuple of Integers, modelling an IPv4 address, the second element is an Integer denoting the port number.
- IPv4_TCPUDP: The same as above, but including additionally the specification of the network protocol.

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

end

2.1.1 Packets and Networks

theory
NetworkCore
imports
Main
begin

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

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

The ID is an integer:

type-synonym *id* = *int*

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

class *adr*

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

instance *int* ::*adr* ⟨*proof*⟩
instance *nat* ::*adr* ⟨*proof*⟩

instance *fun* :: (*adr*,*adr*) *adr* ⟨*proof*⟩
instance *prod* :: (*adr*,*adr*) *adr* ⟨*proof*⟩

The content is also specified with an unconstrained generic type:

type-synonym $'\beta$ *content* = $'\beta$

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

datatype *DummyContent* = *data*

Finally, a packet is:

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

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

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

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

class *port*

instance *int* ::*port* ⟨*proof*⟩
instance *nat* :: *port* ⟨*proof*⟩
instance *fun* :: (*port*,*port*) *port* ⟨*proof*⟩
instance *prod* :: (*port*,*port*) *port* ⟨*proof*⟩

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

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

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

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

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

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

lemma *either*: $[\![a \neq tcp;a \neq udp]\!] \Longrightarrow False$
⟨*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 (⊏) checks if an address is in a specific network.

definition
in-subnet :: $'\alpha::adr \Rightarrow {'\alpha}\ net \Rightarrow bool$ (**infixl** ‹⊏› *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 = $(DatatypeAddress \times port)$

instance *DatatypeAddress* :: *adr* ⟨*proof*⟩

definition
dmz::*DatatypePort net* **where**

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

overloading *src-port-datatype* ≡ *src-port* :: ($'\alpha$::*adr*,$'\beta$) *packet* ⇒ $'\gamma$::*port*
begin
definition
src-port-datatype (*x*::(*DatatypePort*,$'\beta$) *packet*) ≡ (*snd o fst o snd*) *x*
end

overloading *dest-port-datatype* ≡ *dest-port* :: ($'\alpha$::*adr*,$'\beta$) *packet* ⇒ $'\gamma$::*port*
begin
definition
dest-port-datatype (*x*::(*DatatypePort*,$'\beta$) *packet*) ≡(*snd o fst o snd o snd*) *x*
end

overloading *subnet-of-datatype* ≡ *subnet-of* :: $'\alpha$::*adr* ⇒ $'\alpha$ *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*,$'\beta$) *packet*) = *snd x*
⟨*proof*⟩

lemma *dest-port* : *dest-port* ((*a*,*d*,*x*,*e*)::(*DatatypePort*,$'\beta$) *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
$adr_i = int$

end

2.1.5 Integer Addresses with Ports

theory
IntegerPort
imports
NetworkCore
begin

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

type-synonym
$address = int$

type-synonym
$port = int$

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

overloading *src-port-int* $\equiv$ *src-port* :: $('\alpha::adr,'\beta)$ *packet* $\Rightarrow$ $'\gamma::port$
begin
definition
src-port-int $(x::(adr_{ip},'\beta)\ packet) \equiv (snd\ o\ fst\ \ o\ snd)\ x$
end

overloading *dest-port-int* $\equiv$ *dest-port* :: $('\alpha::adr,'\beta)$ *packet* $\Rightarrow$ $'\gamma::port$
begin
definition
dest-port-int $(x::(adr_{ip},'\beta)\ packet) \equiv (snd\ o\ fst\ o\ \ snd\ o\ snd)\ x$
end

overloading *subnet-of-int* $\equiv$ *subnet-of* :: $'\alpha::adr \Rightarrow\ '\alpha\ net$
begin
definition
subnet-of-int $(x::(adr_{ip})) \equiv \{\{(a,b::int).\ a = fst\ x\}\}$
end

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

⟨*proof*⟩

lemma *dest-port*: *dest-port* $(a,d,x{::}adr_{ip},e) = snd\ x$
⟨*proof*⟩

lemmas $adr_{ip}Lemmas$ = *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_{ipp} = *address* × *port* × *protocol*

instance *protocol* :: *adr* ⟨*proof*⟩

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

overloading *dest-port-int-TCPUDP* ≡ *dest-port* :: $('\alpha{::}adr,'\beta)$ *packet* ⇒ $'\gamma{::}port$
begin
definition
dest-port-int-TCPUDP $(x{::}(adr_{ipp},'\beta)\ packet) \equiv (fst\ o\ snd\ o\ fst\ o\ \ snd\ o\ snd)\ x$
end

overloading *subnet-of-int-TCPUDP* ≡ *subnet-of* :: $'\alpha{::}adr \Rightarrow {'\alpha}\ net$
begin

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

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

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

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

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

Common test constraints:

definition *port-positive* $:: (adr_{ipp},'b)\ packet \Rightarrow bool$ **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 $adr_{ipp}Lemmas$ = *src-port dest-port src-port-int-TCPUDP-def dest-port-int-TCPUDP-def src-protocol-int-TCPUDP-def dest-protocol-int-TCPUDP-def subnet-of-int-TCPUDP-def*

lemmas $adr_{ipp}TestConstraints$ = *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
$ipv4\text{-}ip = (int \times int \times int \times int)$

type-synonym
$port = int$

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

overloading $src\text{-}port\text{-}ipv4 \equiv src\text{-}port :: ('\alpha::adr,'\beta)\ packet \Rightarrow '\gamma::port$
begin
definition
$src\text{-}port\text{-}ipv4\ (x::(ipv4,'\beta)\ packet) \equiv (snd\ o\ fst\ o\ snd)\ x$
end

overloading $dest\text{-}port\text{-}ipv4 \equiv dest\text{-}port :: ('\alpha::adr,'\beta)\ packet \Rightarrow '\gamma::port$
begin
definition
$dest\text{-}port\text{-}ipv4\ (x::(ipv4,'\beta)\ packet) \equiv (snd\ o\ fst\ o\ snd\ o\ snd)\ x$
end

overloading $subnet\text{-}of\text{-}ipv4 \equiv subnet\text{-}of :: '\alpha::adr \Rightarrow '\alpha\ net$
begin
definition
$subnet\text{-}of\text{-}ipv4\ (x::ipv4) \equiv \{\{(a,b::int).\ a = fst\ x\}\}$
end

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

lemma *src-port*: $src\text{-}port\ (a,(x::ipv4),d,e) = snd\ x$
⟨*proof*⟩

lemma *dest-port*: $dest\text{-}port\ (a,d,(x::ipv4),e) = snd\ x$
⟨*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* :: ($'\alpha$::*adr*,$'\beta$) *packet* ⇒ $'\gamma$::*port*
begin
definition
src-port-ipv4-TCPUDP (*x*::(*ipv4-TCPUDP*,$'\beta$) *packet*) ≡ (*fst o snd o fst o snd*) *x*
end

overloading *dest-port-ipv4-TCPUDP* ≡ *dest-port* :: ($'\alpha$::*adr*,$'\beta$) *packet* ⇒ $'\gamma$::*port*
begin
definition
dest-port-ipv4-TCPUDP (*x*::(*ipv4-TCPUDP*,$'\beta$) *packet*) ≡ (*fst o snd o fst o snd o snd*) *x*
end

overloading *subnet-of-ipv4-TCPUDP* ≡ *subnet-of* :: $'\alpha$::*adr* ⇒ $'\alpha$ *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* :: ($'\alpha$::*adr*,$'\beta$) *packet* ⇒ *protocol*
begin
definition
dest-protocol-ipv4-TCPUDP (*x*::(*ipv4-TCPUDP*,$'\beta$) *packet*) ≡ (*snd o snd o fst o snd o snd*) *x*
end

definition *subnet-of-ip* :: *ipv4-ip* ⇒ *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*)
  ⟨*proof*⟩

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

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* ↦ *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* ⇒ ($'\alpha$, $'\beta$) *FWPolicy* ⇒ *bool* **where**

$p\text{-}accept\ p\ pol =\ (pol\ p = \lfloor allow\ ()\rfloor)$

end

2.2.2 Policy Combinators

theory
PolicyCombinators
imports
PolicyCore
begin

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

definition
allow-all-from :: $'\alpha$::*adr net* $\Rightarrow$ (($'\alpha$,$'\beta$) *packet* $\mapsto$ *unit*) **where**
allow-all-from src-net = $\{pa.\ src\ pa \sqsubset src\text{-}net\} \lhd A_U$

definition
deny-all-from :: $'\alpha$::*adr net* $\Rightarrow$ (($'\alpha$,$'\beta$) *packet* $\mapsto$ *unit*) **where**
deny-all-from src-net = $\{pa.\ src\ pa \sqsubset src\text{-}net\} \lhd D_U$

definition
allow-all-to :: $'\alpha$::*adr net* $\Rightarrow$ (($'\alpha$,$'\beta$) *packet* $\mapsto$ *unit*) **where**
allow-all-to dest-net = $\{pa.\ dest\ pa \sqsubset dest\text{-}net\} \lhd A_U$

definition
deny-all-to :: $'\alpha$::*adr net* $\Rightarrow$ (($'\alpha$,$'\beta$) *packet* $\mapsto$ *unit*) **where**
deny-all-to dest-net = $\{pa.\ dest\ pa \sqsubset dest\text{-}net\} \lhd D_U$

definition
allow-all-from-to :: $'\alpha$::*adr net* $\Rightarrow$ $'\alpha$::*adr net* $\Rightarrow$ (($'\alpha$,$'\beta$) *packet* $\mapsto$ *unit*) **where**
allow-all-from-to src-net dest-net =
$\{pa.\ src\ pa \sqsubset src\text{-}net \land dest\ pa \sqsubset dest\text{-}net\} \lhd A_U$

definition
deny-all-from-to :: $'\alpha$::*adr net* $\Rightarrow$ $'\alpha$::*adr net* $\Rightarrow$ (($'\alpha$,$'\beta$) *packet* $\mapsto$ *unit*) **where**
deny-all-from-to src-net dest-net = $\{pa.\ src\ pa \sqsubset src\text{-}net \land dest\ pa \sqsubset dest\text{-}net\} \lhd D_U$

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

definition
deny-from-to-port :: $'\gamma$::*port* ⇒ $'\alpha$::*adr net* ⇒ $'\alpha$::*adr net* ⇒ (($'\alpha$,$'\beta$) *packet* ↦ *unit*)
where
deny-from-to-port port src-net dest-net =
$\{pa.\ dest\text{-}port\ pa = port\} \lhd$ *deny-all-from-to src-net dest-net*

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

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

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

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

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

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

lemmas *PortCombinatorsCore* =
allow-all-from-port-def deny-all-from-port-def allow-all-to-port-def

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

lemmas *PortCombinators* = *PortCombinatorsCore PolicyCombinators*

end

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* ⇒ ′α::*adr net* ⇒ (′γ::*port*) ⇒ ((′α,′β) *packet* ↦ *unit*) **where**
allow-all-from-port-prot p src-net s-port =
{*pa. dest-protocol pa* = *p*} ◁ *allow-all-from-port src-net s-port*

definition
deny-all-from-port-prot :: *protocol* =>′α::*adr net* ⇒ ′γ::*port* ⇒ ((′α,′β) *packet* ↦ *unit*) **where**
deny-all-from-port-prot p src-net s-port =
{*pa. dest-protocol pa* = *p*} ◁ *deny-all-from-port src-net s-port*

definition
allow-all-to-port-prot :: *protocol* =>′α::*adr net* ⇒ ′γ::*port* ⇒ ((′α,′β) *packet* ↦ *unit*)
where
allow-all-to-port-prot p dest-net d-port =

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

definition
deny-all-to-port-prot :: *protocol* =>$'\alpha$::*adr net* ⇒ $'\gamma$::*port* ⇒ (($'\alpha$,$'\beta$) *packet* ↦ *unit*)
where
deny-all-to-port-prot p dest-net d-port =
$\{pa.\ dest\text{-}protocol\ pa = p\} \triangleleft$ *deny-all-to-port dest-net d-port*

definition
allow-all-from-port-to-prot:: *protocol* =>$'\alpha$::*adr net* ⇒ $'\gamma$::*port* ⇒ $'\alpha$::*adr net* ⇒ (($'\alpha$,$'\beta$) *packet* ↦ *unit*)
where
allow-all-from-port-to-prot p src-net s-port dest-net =
$\{pa.\ dest\text{-}protocol\ pa = p\} \triangleleft$ *allow-all-from-port-to src-net s-port dest-net*

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

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

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

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

definition

deny-all-from-to-port-prot :: *protocol* =>$'\alpha$::*adr net* ⇒ $'\alpha$::*adr net* ⇒ $'\gamma$::*port* ⇒ (($'\alpha$,$'\beta$) *packet* ↦ *unit*) **where**
deny-all-from-to-port-prot p src-net dest-net d-port =
{*pa*. *dest-protocol pa* = *p*} ◁ *deny-all-from-to-port src-net dest-net d-port*

definition
allow-from-port-to-prot :: *protocol* =>$'\gamma$::*port* ⇒ $'\alpha$::*adr net* ⇒ $'\alpha$::*adr net* ⇒ (($'\alpha$,$'\beta$) *packet* ↦ *unit*)
where
allow-from-port-to-prot p port src-net dest-net =
{*pa*. *dest-protocol pa* = *p*} ◁ *allow-from-port-to port src-net dest-net*

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

definition
allow-from-to-port-prot :: *protocol* =>$'\gamma$::*port* ⇒ $'\alpha$::*adr net* ⇒ $'\alpha$::*adr net* ⇒ (($'\alpha$,$'\beta$) *packet* ↦ *unit*)
where
allow-from-to-port-prot p port src-net dest-net =
{*pa*. *dest-protocol pa* = *p*} ◁ *allow-from-to-port port src-net dest-net*

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

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

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

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

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

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

lemmas *ProtocolCombinatorsCore* =
allow-all-from-port-prot-def deny-all-from-port-prot-def allow-all-to-port-prot-def
deny-all-to-port-prot-def allow-all-from-to-port-prot-def
deny-all-from-to-port-prot-def
allow-from-ports-to-prot-def allow-from-to-ports-prot-def
deny-from-ports-to-prot-def deny-from-to-ports-prot-def
allow-all-from-port-to-prot-def deny-all-from-port-to-prot-def
allow-from-port-to-prot-def allow-from-to-port-prot-def deny-from-to-port-prot-def
deny-from-port-to-prot-def

lemmas *ProtocolCombinators* = *PortCombinators.PortCombinators ProtocolCombinatorsCore*

end

2.2.5 Ports

theory *Ports*
imports
Main
begin

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

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

lemma *http1*: $x \neq 80 \Longrightarrow x \neq http$
⟨*proof*⟩

lemma *http2*: $x \neq 80 \Longrightarrow http \neq x$

⟨*proof*⟩

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

lemma *smtp1*: $x \neq 25 \Longrightarrow x \neq smtp$
⟨*proof*⟩

lemma *smtp2*: $x \neq 25 \Longrightarrow smtp \neq x$
⟨*proof*⟩

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

lemma *ftp1*: $x \neq 21 \Longrightarrow x \neq ftp$
⟨*proof*⟩

lemma *ftp2*: $x \neq 21 \Longrightarrow ftp \neq x$
⟨*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* :: $'\alpha\ set \Rightarrow ('\alpha{::}adr,'\beta)\ packet \Rightarrow ('\alpha,'\beta)\ packet\ set$ **where**
$src2pool\ t = (\lambda\ p.\ (\{(i,s,d,da).\ (i = id\ p \wedge s \in t \wedge d = dest\ p \wedge da = content\ p)\}))$

definition *src2poolAP* **where**
$src2poolAP\ t = A_f\ (src2pool\ t)$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

lemma *datasimp3*: $\{(i,\ (s1,\ s2,\ s3),\ aba).$
$\forall\ a\ aa\ b\ ba.\ aba = ((a,\ aa,\ b),\ ba) \longrightarrow i = i1 \wedge i115 < s1 \wedge s1 <$
$i124 \wedge$
$s3 = iudp \wedge s2 = ii1 \wedge a = i110 \wedge aa = i3 \wedge b = itcp \wedge ba =$
$data\}$
$= \{(i,\ (s1,\ s2,\ s3),\ aba).$
$i = i1 \wedge i115 < s1 \wedge s1 < i124 \wedge s3 = iudp \wedge s2 = ii1 \wedge$
$(\lambda\ ((a,aa,b),ba).\ a = i110\ \&\ aa = i3\ \&\ b = itcp\ \&\ ba = data)\ aba\}$
$\langle proof \rangle$

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

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

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

$\langle proof \rangle$

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

$\langle proof \rangle$

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

$\langle proof \rangle$

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

$\langle proof \rangle$

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

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

end

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** ‹⊕› *80*)

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

Auxiliary definitions and functions.

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

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

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

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

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

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

|*net-list-aux* (*DenyAll*#*xs*) = *net-list-aux xs*
|*net-list-aux* ((*DenyAllFromTo x y*)#*xs*) = *x*#*y*#(*net-list-aux xs*)
|*net-list-aux* ((*AllowPortFromTo x y p*)#*xs*) = *x*#*y*#(*net-list-aux xs*)
|*net-list-aux* ((*x*⊕*y*)#*xs*) = (*net-list-aux* [*x*])@(*net-list-aux* [*y*])@(*net-list-aux xs*)

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

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

fun (*sequential*) *normBothNets* **where**
normBothNets ((*a*,*b*)#*xs*) = (*if* ((*b*,*a*) ∈ *set xs*) ∨ (*a*,*b*) ∈ *set* (*xs*)
then (*normBothNets xs*)
else (*a*,*b*)#(*normBothNets xs*))
|*normBothNets x* = *x*

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

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

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

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

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

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

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

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

$in\text{-}list\ DenyAll\ l = True$
$|in\text{-}list\ x\ l = (bothNet\ x \in set\ l)$

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

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

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

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

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

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

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

Invariants

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

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

There is a DenyAll at the first position

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

wellformed-policy1-strong [] = *False*
| *wellformed-policy1-strong* (*x*#*xs*) = (*x*=*DenyAll* ∧ (*DenyAll* ∉ (*set xs*)))

All two networks are either disjoint or equal.

definition *netsDistinct* **where** *netsDistinct a b* = (¬ (∃ *x*. *x* ⊏ *a* ∧ *x* ⊏ *b*))

definition *twoNetsDistinct* **where**
twoNetsDistinct a b c d = (*netsDistinct a c* ∨ *netsDistinct b d*)

definition *allNetsDistinct* **where**
allNetsDistinct p = (∀ *a b*. (*a* ≠ *b* ∧ *a* ∈ *set* (*net-list p*) ∧
b ∈ *set* (*net-list p*)) ⟶ *netsDistinct a b*)

definition *disjSD-2* **where**
disjSD-2 x y = (∀ *a b c d*. ((*a*,*b*)∈*sdnets x* ∧ (*c*,*d*) ∈*sdnets y* ⟶
(*twoNetsDistinct a b c d* ∧ *twoNetsDistinct a b d c*)))

The policy is given as a list of single rules.

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

definition *onlyTwoNets* **where**
onlyTwoNets x = ((∃ *a b*. (*sdnets x* = {(*a*,*b*)})) ∨ (∃ *a b*. *sdnets x* = {(*a*,*b*),(*b*,*a*)}))

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

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

fun *noDenyAll* **where**
noDenyAll (*x*#*xs*) = ((¬ *member DenyAll x*) ∧ *noDenyAll xs*)
|*noDenyAll* [] = *True*

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

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

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

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

Transformations

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

fun $policy2list{::}('\alpha,\ '\beta)\ Combinators \Rightarrow$
$(('\alpha,\ '\beta)\ Combinators)\ list$ **where**
$policy2list\ (x \oplus y) = (concat\ [(policy2list\ x),(policy2list\ y)])$
$|policy2list\ x = [x]$

fun $list2FWpolicy{::}(('\alpha,\ '\beta)\ Combinators)\ list \Rightarrow$
$(('\alpha,\ '\beta)\ Combinators)$ **where**
$list2FWpolicy\ [] = undefined$
$|list2FWpolicy\ (x\#[]) = x$
$|list2FWpolicy\ (x\#y) = x \oplus (list2FWpolicy\ y)$

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

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

fun *removeShadowRules1-alternative-rev* **where**
$removeShadowRules1\text{-}alternative\text{-}rev\ [] = []$
$|\ removeShadowRules1\text{-}alternative\text{-}rev\ (DenyAll\#xs) = [DenyAll]$
$|\ removeShadowRules1\text{-}alternative\text{-}rev\ [x] = [x]$
$|\ removeShadowRules1\text{-}alternative\text{-}rev\ (x\#xs)=$
$x\#(removeShadowRules1\text{-}alternative\text{-}rev\ xs)$

definition *removeShadowRules1-alternative* **where**
$removeShadowRules1\text{-}alternative\ p =$
$rev\ (removeShadowRules1\text{-}alternative\text{-}rev\ (rev\ p))$

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

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

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

fun *smaller* :: $('\alpha, '\beta)$ *Combinators* $\Rightarrow$
$('\alpha, '\beta)$ *Combinators* $\Rightarrow$
$(('\alpha)$ *set*) *list* $\Rightarrow$ *bool*
where
smaller DenyAll x l = *True*
| *smaller x DenyAll l* = *False*
| *smaller x y l* =
((*x* = *y*) $\vee$ (*if* (*bothNet x*) = (*bothNet y*) *then*
(*case y of* (*DenyAllFromTo a b*) $\Rightarrow$ (*x* = *DenyAllFromTo b a*)
| - $\Rightarrow$ *True*)
else
(*position* (*bothNet x*) *l* <= *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 = $insort$ x $(sort$ xs $l)$ l

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

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

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

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

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

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

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

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_{ip} *net*, *port*) *Combinators* $\Rightarrow$ (adr_{ip},*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_{ip} *net*, *port*) *Combinators* $\Rightarrow$ (adr_{ip},*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*$_{ip}$ *net*,*port*) *Combinators*) *list* ⇒ *bool* **where**
wellformed-policy3 [] = *True*
| *wellformed-policy3* ((*AllowPortFromTo a b p*)#*xs*) = ((∀ *r*. *r* ∈ *set xs* ⟶
dom (*C r*) ∩ *dom* (*C* (*AllowPortFromTo a b p*)) = {}) ∧ *wellformed-policy3 xs*)
| *wellformed-policy3* (*x*#*xs*) = *wellformed-policy3 xs*

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

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

definition *normalize* ::
(*adr*$_{ip}$ *net*, *port*) *Combinators* ⇒
(*adr*$_{ip}$ *net*, *port*) *Combinators list*
where
normalize p = (*removeAllDuplicates* (*insertDenies* (*separate* (*sort*
(*removeShadowRules2* (*remdups* ((*rm-MT-rules C*) (*insertDeny*
(*removeShadowRules1* (*policy2list p*)))))) ((*Nets-List p*))))))

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

definition *normalizeQ* ::
(*adr*$_{ip}$ *net*, *port*) *Combinators* ⇒
(*adr*$_{ip}$ *net*, *port*) *Combinators list*
where
normalizeQ p = (*removeAllDuplicates* (*insertDenies* (*separate* (*qsort*
(*removeShadowRules2* (*remdups* ((*rm-MT-rules C*) (*insertDeny*
(*removeShadowRules1* (*policy2list p*)))))) ((*Nets-List p*))))))

definition
normalize-manual-orderQ p l = *removeAllDuplicates* (*insertDenies* (*separate*

(*qsort* (*removeShadowRules2* (*remdups* ((*rm-MT-rules C*) (*insertDeny* (*removeShadowRules1* (*policy2list p*)))))) ((*l*)))))

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

lemma *normalize* = *normalize'*
⟨*proof*⟩

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

TCP_UDP_IntegerPort

fun *Cp* :: (adr_{ipp} *net*, *protocol* × *port*) *Combinators* ⇒
(adr_{ipp},*DummyContent*) *packet* ↦ *unit*
where
Cp DenyAll = *deny-all*
|*Cp* (*DenyAllFromTo x y*) = *deny-all-from-to x y*
|*Cp* (*AllowPortFromTo x y p*) = *allow-from-to-port-prot* (*fst p*) (*snd p*) *x y*
|*Cp* (*x* ⊕ *y*) = *Cp x* ++ *Cp y*

fun *Dp* :: (adr_{ipp} *net*, *protocol* × *port*) *Combinators* ⇒
(adr_{ipp},*DummyContent*) *packet* ↦ *unit*
where
Dp DenyAll = *Cp DenyAll*
|*Dp* (*DenyAllFromTo x y*) = *Cp* (*DenyAllFromTo x y*)
|*Dp* (*AllowPortFromTo x y p*) = *Cp* (*AllowPortFromTo x y p*)
|*Dp* (*x* ⊕ *y*) = *Cp* (*y* ⊕ *x*)

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

fun (*sequential*) *wellformed-policy2Pr* **where**
wellformed-policy2Pr [] = *True*
| *wellformed-policy2Pr* (*DenyAll#xs*) = *wellformed-policy2Pr xs*
| *wellformed-policy2Pr* (*x#xs*) = ((∀ *c a b*. *c* = *DenyAllFromTo a b* ∧ *c* ∈ *set xs* ⟶
Map.dom (*Cp x*) ∩ *Map.dom* (*Cp c*) = {}) ∧ *wellformed-policy2Pr xs*)

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

fun (*sequential*) *wellformed-policy3Pr*::((adr_{ipp} *net*, *protocol* × *port*) *Combinators*) *list* ⇒ *bool* **where**
wellformed-policy3Pr [] = *True*
| *wellformed-policy3Pr* ((*AllowPortFromTo a b p*)#*xs*) = ((∀ *r*. *r* ∈ *set xs* ⟶
dom (*Cp r*) ∩ *dom* (*Cp* (*AllowPortFromTo a b p*)) = {}) ∧ *wellformed-policy3Pr xs*)
| *wellformed-policy3Pr* (*x#xs*) = *wellformed-policy3Pr xs*

definition
normalizePr′ :: (adr_{ipp} *net*, *protocol* × *port*) *Combinators*
⇒ (adr_{ipp} *net*, *protocol* × *port*) *Combinators list* **where**
normalizePr′ p = (*removeAllDuplicates o insertDenies o separate o*
(*sort′* (*Nets-List p*)) *o removeShadowRules2 o remdups o*
(*rm-MT-rules Cp*) *o insertDeny o removeShadowRules1 o*
policy2list) *p*

definition *normalizePr* ::
(adr_{ipp} *net*, *protocol* × *port*) *Combinators*
⇒ (adr_{ipp} *net*, *protocol* × *port*) *Combinators list* **where**
normalizePr p = (*removeAllDuplicates* (*insertDenies* (*separate* (*sort*
(*removeShadowRules2* (*remdups* ((*rm-MT-rules Cp*) (*insertDeny*
(*removeShadowRules1* (*policy2list p*)))))) ((*Nets-List p*))))))

definition
normalize-manual-orderPr p l = *removeAllDuplicates* (*insertDenies* (*separate*
(*sort* (*removeShadowRules2* (*remdups* ((*rm-MT-rules Cp*) (*insertDeny*
(*removeShadowRules1* (*policy2list p*)))))) ((*l*)))))

definition
normalizePrQ′ :: (adr_{ipp} *net*, *protocol* × *port*) *Combinators*
⇒ (adr_{ipp} *net*, *protocol* × *port*) *Combinators list* **where**
normalizePrQ′ p = (*removeAllDuplicates o insertDenies o separate o*
(*qsort′* (*Nets-List p*)) *o removeShadowRules2 o remdups o*
(*rm-MT-rules Cp*) *o insertDeny o removeShadowRules1 o*
policy2list) *p*

definition *normalizePrQ* ::
(adr_{ipp} *net*, *protocol* × *port*) *Combinators*
⇒ (adr_{ipp} *net*, *protocol* × *port*) *Combinators list* **where**
normalizePrQ p = (*removeAllDuplicates* (*insertDenies* (*separate* (*qsort*
(*removeShadowRules2* (*remdups* ((*rm-MT-rules Cp*) (*insertDeny*
(*removeShadowRules1* (*policy2list p*)))))) ((*Nets-List p*))))))

definition
normalize-manual-orderPrQ p l = *removeAllDuplicates* (*insertDenies* (*separate*
(*qsort* (*removeShadowRules2* (*remdups* ((*rm-MT-rules Cp*) (*insertDeny*
(*removeShadowRules1* (*policy2list p*)))))) ((*l*)))))

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

lemma *normalizePr* = *normalizePr′*

⟨*proof*⟩

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

definition *makeFUTPr* **where**
makeFUTPr FUT p x n =
(*packet-Nets x* (*fst* (*normBothNets* (*bothNets p*)!*n*))
(*snd*(*normBothNets* (*bothNets p*)!*n*)) ⟶
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_{ipp}*Lemmas* adr_{ipp}*Lemmas*

lemma *aux*: ⟦$x \neq a$; $y \neq b$; $(x \neq y \wedge x \neq b) \vee (a \neq b \wedge a \neq y)$⟧ $\Longrightarrow \{x,a\} \neq \{y,b\}$
⟨*proof*⟩

lemma *aux2*: $\{a,b\} = \{b,a\}$
⟨*proof*⟩
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* $X \neq \{\} \Longrightarrow X \neq \emptyset$
⟨*proof*⟩

lemma *denyNMT*: *deny-all* $\neq \emptyset$
⟨*proof*⟩

lemma *wellformed-policy1-charn*[*rule-format*]:
wellformed-policy1 p ⟶ *DenyAll* ∈ *set p* ⟶ (∃ *p*′. *p* = *DenyAll* # *p*′ ∧ *DenyAll* ∉ *set p*′)

⟨*proof*⟩

lemma *singleCombinatorsConc*: *singleCombinators* (*x#xs*) $\Longrightarrow$ *singleCombinators xs*
⟨*proof*⟩

lemma *aux0-0*: *singleCombinators x* $\Longrightarrow \neg (\exists\ a\ b.\ (a \oplus b) \in set\ x)$
⟨*proof*⟩

lemma *aux0-4*: $(a \in set\ x \vee a \in set\ y) = (a \in set\ (x@y))$
⟨*proof*⟩

lemma *aux0-1*: ⟦*singleCombinators xs*; *singleCombinators* [*x*]⟧ $\Longrightarrow$
singleCombinators (*x#xs*)
⟨*proof*⟩

lemma *aux0-6*: ⟦*singleCombinators xs*; $\neg (\exists\ a\ b.\ x = a \oplus b)$⟧ $\Longrightarrow$
singleCombinators(*x#xs*)
⟨*proof*⟩

lemma *aux0-5*: $\neg (\exists\ a\ b.\ (a \oplus b) \in set\ x) \Longrightarrow$ *singleCombinators x*
⟨*proof*⟩

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

lemma *aux6*: *twoNetsDistinct a1 a2 a b* $\Longrightarrow$
dom (*deny-all-from-to a1 a2*) $\cap$ *dom* (*deny-all-from-to a b*) $= \{\}$
⟨*proof*⟩

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

lemma *aux5a*[*rule-format*]: (*DenyAllFromTo b a*) $\in$ *set p* $\longrightarrow$ *a* $\in$ *set* (*net-list p*)
⟨*proof*⟩

lemma *aux5c*[*rule-format*]:
(*AllowPortFromTo a b po*) $\in$ *set p* $\longrightarrow$ *a* $\in$ *set* (*net-list p*)
⟨*proof*⟩

lemma *aux5d*[*rule-format*]:
(*AllowPortFromTo b a po*) $\in$ *set p* $\longrightarrow$ *a* $\in$ *set* (*net-list p*)
⟨*proof*⟩

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

⟨*proof*⟩

lemma *srcInNetListaux*[*simp*]:
$[\![x \in set\ p;\ singleCombinators[x];\ x \neq DenyAll]\!] \Longrightarrow srcNet\ x \in set\ (net\text{-}list\text{-}aux\ p)$
⟨*proof*⟩

lemma *destInNetListaux*[*simp*]:
$[\![x \in set\ p;\ singleCombinators[x];\ x \neq DenyAll]\!] \Longrightarrow destNet\ x \in set\ (net\text{-}list\text{-}aux\ p)$
⟨*proof*⟩

lemma *tND1*: $[\![allNetsDistinct\ p;\ x \in set\ p;\ y \in set\ p;\ a = srcNet\ x;$
$b = destNet\ x;\ c = srcNet\ y;\ d = destNet\ y;\ a \neq c;$
$singleCombinators[x];\ x \neq DenyAll;\ singleCombinators[y];$
$y \neq DenyAll]\!] \Longrightarrow twoNetsDistinct\ a\ b\ c\ d$
⟨*proof*⟩

lemma *tND2*: $[\![allNetsDistinct\ p;\ x \in set\ p;\ y \in set\ p;\ a = srcNet\ x;$
$b = destNet\ x;\ c = srcNet\ y;\ d = destNet\ y;\ b \neq d;$
$singleCombinators[x];\ x \neq DenyAll;\ singleCombinators[y];$
$y \neq DenyAll]\!] \Longrightarrow twoNetsDistinct\ a\ b\ c\ d$
⟨*proof*⟩

lemma *tND*: $[\![allNetsDistinct\ p;\ x \in set\ p;\ y \in set\ p;\ a = srcNet\ x;$
$b = destNet\ x;\ c = srcNet\ y;\ d = destNet\ y;\ a \neq c \lor b \neq d;$
$singleCombinators[x];\ x \neq DenyAll;\ singleCombinators[y];\ y \neq DenyAll]\!]$
$\Longrightarrow twoNetsDistinct\ a\ b\ c\ d$
⟨*proof*⟩

lemma *aux7*: $[\![DenyAllFromTo\ a\ b \in set\ p;\ allNetsDistinct\ ((DenyAllFromTo\ c\ d)\#p);$
$a\neq c\lor b\neq d]\!] \Longrightarrow twoNetsDistinct\ a\ b\ c\ d$
⟨*proof*⟩

lemma *aux7a*: $[\![DenyAllFromTo\ a\ b\ \in set\ p;$
$allNetsDistinct\ ((AllowPortFromTo\ c\ d\ po)\#p);\ a \neq c\lor\ b \neq d]\!] \Longrightarrow$
$twoNetsDistinct\ a\ b\ c\ d$
⟨*proof*⟩

lemma *nDComm*: **assumes** *ab*: *netsDistinct a b* **shows** *ba*: *netsDistinct b a*
⟨*proof*⟩

lemma *tNDComm*:
assumes *abcd*: *twoNetsDistinct a b c d* **shows** *twoNetsDistinct c d a b*
⟨*proof*⟩

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

lemma *aux12*: $[\![a \in x;\ b \notin x]\!] \Longrightarrow a \neq b$
⟨*proof*⟩

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

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

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

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

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

lemma *aNDSetsEqaux*[*rule-format*]: $singleCombinators\ a \longrightarrow singleCombinators\ b \longrightarrow$

$set\ a = set\ b \longrightarrow set\ (net\text{-}list\text{-}aux\ a) = set\ (net\text{-}list\text{-}aux\ b)$
⟨*proof*⟩

lemma *aNDSubset*: $[\![singleCombinators\ a;set\ a \subseteq set\ b;\ allNetsDistinct\ b]\!] \Longrightarrow$
$allNetsDistinct\ a$
⟨*proof*⟩

lemma *aNDSetsEq*: $[\![singleCombinators\ a;\ singleCombinators\ b;\ set\ a = set\ b;$
$allNetsDistinct\ b]\!] \Longrightarrow allNetsDistinct\ a$
⟨*proof*⟩

lemma *SCConca*: $[\![singleCombinators\ p;\ singleCombinators\ [a]]\!] \Longrightarrow$
$singleCombinators\ (a\#p)$
⟨*proof*⟩

lemma *aux3*[*simp*]: $[\![singleCombinators\ p;\ singleCombinators\ [a];$

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

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

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

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

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

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

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

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

lemma *aux80*: ⟦$y \notin$ *set p*; $y \neq x$⟧ $\Longrightarrow$ $y \notin$ *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*

$\langle proof \rangle$

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

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

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

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

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

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

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

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

lemma *WP1*: $a \notin set\ list \Longrightarrow a \notin set\ (removeShadowRules2\ list)$
$\langle proof \rangle$

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

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

lemma *SCConcEnd*: $singleCombinators\ (xs\ @\ [xa]) \Longrightarrow singleCombinators\ xs$
$\langle proof \rangle$

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

$(list2FWpolicy\ (xa\ \#\ a)) = (xa) \oplus (list2FWpolicy\ a)$
⟨*proof*⟩

lemma *wp1n-tl* [*rule-format*]: $wellformed\text{-}policy1\text{-}strong\ p \longrightarrow$
$p = (DenyAll\#(tl\ p))$
⟨*proof*⟩

lemma *foo2*: $a \notin set\ ps \Longrightarrow$
$a \notin set\ ss \Longrightarrow$
$set\ p = set\ s \Longrightarrow$
$p = (a\#(ps)) \Longrightarrow$
$s = (a\#ss) \Longrightarrow$
$set\ (ps) = set\ (ss)$
⟨*proof*⟩

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

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

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

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

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

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

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

lemma *rSR1-eq*: $removeShadowRules1\text{-}alternative = removeShadowRules1$
⟨*proof*⟩

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

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

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

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

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

lemma *ANDConcEnd*: $[\![\ allNetsDistinct\ (xs\ @\ [xa]);\ singleCombinators\ xs]\!] \Longrightarrow$
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* ⟹ *singleCombinators* (*removeShadowRules1 p*)
  ⟨*proof*⟩

lemma *SC2*[*simp*]: *singleCombinators p* ⟹ *singleCombinators* (*removeShadowRules2 p*)
  ⟨*proof*⟩

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

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

lemma *aND-RS1*[*simp*]: ⟦*singleCombinators p*; *allNetsDistinct p*⟧ ⟹
    allNetsDistinct (*removeShadowRules1 p*)
  ⟨*proof*⟩

lemma *aND-RS2*[*simp*]: ⟦*singleCombinators p*; *allNetsDistinct p*⟧ ⟹
    allNetsDistinct (*removeShadowRules2 p*)
  ⟨*proof*⟩

lemma *aND-sort*[*simp*]: ⟦*singleCombinators p*; *allNetsDistinct p*⟧ ⟹
    allNetsDistinct (*sort p l*)
  ⟨*proof*⟩

lemma *aND-sortQ*[*simp*]: ⟦*singleCombinators p*; *allNetsDistinct p*⟧ ⟹
    allNetsDistinct (*qsort p l*)
  ⟨*proof*⟩

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

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

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

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

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

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

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

lemma *order-trans*:
⟦*in-list x l*; *in-list y l*; *in-list z l*; *singleCombinators* [*x*];
singleCombinators [*y*]; *singleCombinators* [*z*]; *smaller x y l*; *smaller y z l*⟧ $\Longrightarrow$
smaller x z l
⟨*proof*⟩

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

lemma *singleCombinatorsStart*[*simp*]: *singleCombinators* (*x*#*xs*) $\Longrightarrow$
singleCombinators [*x*]
⟨*proof*⟩

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

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

⟨*proof*⟩

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

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

lemma *sorted-Cons*: $[\![all\text{-}in\text{-}list\ (x\#xs)\ l;\ singleCombinators\ (x\#xs)]\!] \Longrightarrow$
$(sorted\ xs\ l\ \&\ (\forall y \in set\ xs.\ smaller\ x\ y\ l)) = (sorted\ (x\#xs)\ l)$
⟨*proof*⟩

lemma *smaller-antisym*: $[\![\neg\ smaller\ a\ b\ l;\ in\text{-}list\ a\ l;\ in\text{-}list\ b\ l;$
$singleCombinators[a];\ singleCombinators\ [b]]\!] \Longrightarrow$
$smaller\ b\ a\ l$
⟨*proof*⟩

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

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

lemma *singleCombinators-insort*: $[\![singleCombinators\ [x];\ singleCombinators\ xs]\!] \Longrightarrow$
$singleCombinators\ (insort\ x\ xs\ l)$
⟨*proof*⟩

lemma *all-in-list-insort*: $[\![all\text{-}in\text{-}list\ xs\ l;\ singleCombinators\ (x\#xs);$
$in\text{-}list\ x\ l]\!] \Longrightarrow all\text{-}in\text{-}list\ (insort\ x\ xs\ l)\ l$
⟨*proof*⟩

lemma *sorted-ConsA*: $[\![all\text{-}in\text{-}list\ (x\#xs)\ l;\ singleCombinators\ (x\#xs)]\!] \Longrightarrow$
$(sorted\ (x\#xs)\ l) = (sorted\ xs\ l\ \&\ (\forall y \in set\ xs.\ smaller\ x\ y\ l))$
⟨*proof*⟩

lemma *is-in-insort*: $y \in set\ xs \Longrightarrow y \in set\ (insort\ x\ xs\ l)$
⟨*proof*⟩

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

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 ⟶ *all-in-list* (*x#xs*) *l* ⟶ *distinct* (*x#xs*) ⟶
singleCombinators [*x*] ⟶ *singleCombinators xs* ⟶ *sorted* (*insort x xs l*) *l*
⟨*proof*⟩

lemma *sorted-insort*:
⟦*all-in-list* (*x#xs*) *l*; *distinct*(*x#xs*); *singleCombinators* [*x*];
singleCombinators xs⟧ ⟹
sorted (*insort x xs l*) *l* = *sorted xs l*
⟨*proof*⟩

lemma *distinct-insort*: *distinct* (*insort x xs l*) = (*x* ∉ *set xs* ∧ *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 ⟶ *distinct p* ⟶ *singleCombinators p* ⟶ *sorted* (*sort p l*) *l*
⟨*proof*⟩

lemma *smaller-sym*[*rule-format*]: *all-in-list* [*a*] *l* ⟶ *smaller a a l*
⟨*proof*⟩

lemma *SC-sublist*[*rule-format*]:
singleCombinators xs ⟹ *singleCombinators* (*qsort* [*y*←*xs*. *P y*] *l*)
⟨*proof*⟩

lemma *all-in-list-sublist*[*rule-format*]:
singleCombinators xs ⟶ *all-in-list xs l* ⟶ *all-in-list* (*qsort* [*y*←*xs*. *P y*] *l*) *l*
⟨*proof*⟩

lemma *SC-sublist2*[*rule-format*]:
singleCombinators xs ⟶ *singleCombinators* ([*y*←*xs*. *P y*])
⟨*proof*⟩

lemma *all-in-list-sublist2*[*rule-format*]:
singleCombinators xs ⟶ *all-in-list xs l* ⟶ *all-in-list* ([*y*←*xs*. *P y*]) *l*
⟨*proof*⟩

lemma *all-in-listAppend*[*rule-format*]:
all-in-list (*xs*) *l* ⟶ *all-in-list* (*ys*) *l* ⟶ *all-in-list* (*xs* @ *ys*) *l*
⟨*proof*⟩

lemma *distinct-sortQ*[*rule-format*]:
singleCombinators xs ⟶ *all-in-list xs l* ⟶ *distinct xs* ⟶ *distinct* (*qsort xs l*)
⟨*proof*⟩

lemma *singleCombinatorsAppend*[*rule-format*]:
singleCombinators (*xs*) ⟶ *singleCombinators* (*ys*) ⟶ *singleCombinators* (*xs* @ *ys*)
⟨*proof*⟩

lemma *sorted-append1*[*rule-format*]:
all-in-list xs l ⟶ *singleCombinators xs* ⟶
all-in-list ys l ⟶ *singleCombinators ys* ⟶
(*sorted* (*xs*@*ys*) *l* ⟶
(*sorted xs l* & *sorted ys l* & (∀ *x* ∈ *set xs*. ∀ *y* ∈ *set ys*. *smaller x y l*)))
⟨*proof*⟩

lemma *sorted-append2*[*rule-format*]:
all-in-list xs l⟶ *singleCombinators xs* ⟶
all-in-list ys l ⟶ *singleCombinators ys* ⟶
(*sorted xs l* & *sorted ys l* & (∀ *x* ∈ *set xs*. ∀ *y* ∈ *set ys*. *smaller x y l*)) ⟶
(*sorted* (*xs*@*ys*) *l*)
⟨*proof*⟩

lemma *sorted-append*[*rule-format*]:
all-in-list xs l ⟶ *singleCombinators xs* ⟶
all-in-list ys l ⟶ *singleCombinators ys* ⟶
(*sorted* (*xs*@*ys*) *l*) =
(*sorted xs l* & *sorted ys l* & (∀ *x* ∈ *set xs*. ∀ *y* ∈ *set ys*. *smaller x y l*))
⟨*proof*⟩

lemma *sort-is-sortedQ*[*rule-format*]:
all-in-list p l ⟶ *singleCombinators p* ⟶ *sorted* (*qsort p l*) *l*
⟨*proof*⟩

lemma *inSet-not-MT*: *a* ∈ *set p* ⟹ *p* ≠ []

⟨*proof*⟩

lemma *RS1n-assoc*:
$x \neq DenyAll \Longrightarrow$ *removeShadowRules1-alternative xs* @ [*x*] =
removeShadowRules1-alternative (*xs* @ [*x*])
⟨*proof*⟩

lemma *RS1n-nMT*[*rule-format*,*simp*]: $p \neq [] \longrightarrow$ *removeShadowRules1-alternative* $p \neq []$
⟨*proof*⟩

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

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

lemma *mt-sym*: *dom a* $\cap$ *dom b* = {} $\Longrightarrow$ *dom b* $\cap$ *dom a* = {}
⟨*proof*⟩

lemma *DAnotTL*[*rule-format*]:
$xs \neq [] \longrightarrow$ *wellformed-policy1* (*xs* @ [*DenyAll*]) $\longrightarrow$ *False*
⟨*proof*⟩

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

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

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

lemma *Conc-not-MT*: $p = x\#xs \Longrightarrow p \neq []$
⟨*proof*⟩

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

lemma *wp1-eq*[*rule-format*]:
wellformed-policy1-strong p $\Longrightarrow$ *wellformed-policy1 p*

⟨*proof*⟩

lemma *wellformed1-alternative-sorted*:
wellformed-policy1-strong p $\Longrightarrow$ *wellformed-policy1-strong* (*sort p l*)
⟨*proof*⟩

lemma *wp1n-RS2*[*rule-format*]:
wellformed-policy1-strong p $\longrightarrow$ *wellformed-policy1-strong* (*removeShadowRules2 p*)
⟨*proof*⟩

lemma *RS2-NMT*[*rule-format*]: *p* $\neq$ [] $\longrightarrow$ *removeShadowRules2 p* $\neq$ []
⟨*proof*⟩

lemma *wp1-alternative-not-mt*[*simp*]: *wellformed-policy1-strong p* $\Longrightarrow$ *p* $\neq$ []
⟨*proof*⟩

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

lemma *wp1ID*: *wellformed-policy1-strong* (*insertDeny* (*removeShadowRules1 p*))
⟨*proof*⟩

lemma *dRD*[*simp*]: *distinct* (*remdups p*)
⟨*proof*⟩

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

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

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

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

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

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

singleCombinators p ⟶ *allNetsDistinct p* ⟶ *allNetsDistinct* (*remdups p*)
⟨*proof*⟩

lemma *ANDiD*[*rule-format*,*simp*]:
allNetsDistinct p ⟶ *allNetsDistinct* (*insertDeny p*)
⟨*proof*⟩

lemma *mr-iD*[*rule-format*]:
wellformed-policy1-strong p ⟶ *matching-rule x p* = *matching-rule x* (*insertDeny p*)
⟨*proof*⟩

lemma *WP1iD*[*rule-format*,*simp*]: *wellformed-policy1-strong p* ⟶
wellformed-policy1-strong (*insertDeny p*)
⟨*proof*⟩

lemma *DAiniD*: *DenyAll* ∈ *set* (*insertDeny p*)
⟨*proof*⟩

lemma *p2lNmt*: *policy2list p* ≠ []
⟨*proof*⟩

lemma *AIL2*[*rule-format*,*simp*]:
all-in-list p l ⟶ *all-in-list* (*removeShadowRules2 p*) *l*
⟨*proof*⟩

lemma *SCConc*: *singleCombinators x* ⟹ *singleCombinators y* ⟹ *singleCombinators* (*x*@*y*)
⟨*proof*⟩

lemma *SCp2l*: *singleCombinators* (*policy2list p*)
⟨*proof*⟩

lemma *subnetAux*: *Dd* ∩ *A* ≠ {} ⟹ *A* ⊆ *B* ⟹ *Dd* ∩ *B* ≠ {}
⟨*proof*⟩

lemma *soadisj*: *x* ∈ *subnetsOfAdr a* ⟹ *y* ∈ *subnetsOfAdr a* ⟹ ¬ *netsDistinct x y*
⟨*proof*⟩

lemma *not-member*: ¬ *member a* (*x*⊕*y*) ⟹ ¬ *member a x*
⟨*proof*⟩

lemma *soadisj2*: (∀ *a x y*. *x* ∈ *subnetsOfAdr a* ∧ *y* ∈ *subnetsOfAdr a* ⟶ ¬ *netsDistinct x y*)
⟨*proof*⟩

lemma *ndFalse1*:
$(\forall a\ b\ c\ d.\ (a,b) \in A\ \wedge (c,d) \in B \longrightarrow netsDistinct\ a\ c) \Longrightarrow$
$\exists (a,\ b) \in A.\ a \in subnetsOfAdr\ D \Longrightarrow$
$\exists (a,\ b) \in B.\ a \in subnetsOfAdr\ D \Longrightarrow False$
⟨*proof*⟩

lemma *ndFalse2*: $(\forall a\ b\ c\ d.\ (a,b) \in A\ \wedge (c,d) \in B \longrightarrow netsDistinct\ b\ d) \Longrightarrow$
$\exists (a,\ b) \in A.\ b \in subnetsOfAdr\ D \Longrightarrow$
$\exists (a,\ b) \in B.\ b \in subnetsOfAdr\ D \Longrightarrow False$
⟨*proof*⟩

lemma *tndFalse*: $(\forall a\ b\ c\ d.\ (a,b) \in A\ \wedge (c,d) \in B \longrightarrow twoNetsDistinct\ a\ b\ c\ d) \Longrightarrow$
$\exists (a,\ b) \in A.\ a \in subnetsOfAdr\ (D::('a::adr)) \wedge b \in subnetsOfAdr\ (F::'a) \Longrightarrow$
$\exists (a,\ b) \in B.\ a \in subnetsOfAdr\ D \wedge b \in subnetsOfAdr\ F$
$\Longrightarrow False$
⟨*proof*⟩

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

lemma *sepDA*[*rule-format*]: $DenyAll \notin set\ p \longrightarrow DenyAll \notin set\ (separate\ p)$
⟨*proof*⟩

lemma *setnMT*: $set\ a = set\ b \Longrightarrow a \neq [] \Longrightarrow b \neq []$
⟨*proof*⟩

lemma *sortnMT*: $p \neq [] \Longrightarrow sort\ p\ l \neq []$
⟨*proof*⟩

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

lemma *OTNoTN*[*rule-format*]: $OnlyTwoNets\ p \longrightarrow x \neq DenyAll \longrightarrow x \in set\ p \longrightarrow onlyTwoNets\ x$
⟨*proof*⟩

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

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

⟨*proof*⟩

lemma *alternativelistconc1*[*rule-format*]:
a ∈ *set* (*net-list-aux* [*x*]) ⟶ *a* ∈ *set* (*net-list-aux* [*x*,*y*])
⟨*proof*⟩

lemma *alternativelistconc2*[*rule-format*]:
a ∈ *set* (*net-list-aux* [*x*]) ⟶ *a* ∈ *set* (*net-list-aux* [*y*,*x*])
⟨*proof*⟩

lemma *noDA*[*rule-format*]:
noDenyAll xs ⟶ *s* ∈ *set xs* ⟶ ¬ *member DenyAll s*
⟨*proof*⟩

lemma *isInAlternativeList*:
(*aa* ∈ *set* (*net-list-aux* [*a*]) ∨ *aa* ∈ *set* (*net-list-aux p*)) ⟹ *aa* ∈ *set* (*net-list-aux* (*a* # *p*))
⟨*proof*⟩

lemma *netlistaux*:
x ∈ *set* (*net-list-aux* (*a* # *p*))⟹ *x* ∈ *set* (*net-list-aux* ([*a*])) ∨ *x* ∈ *set* (*net-list-aux* (*p*))
⟨*proof*⟩

lemma *firstInNet*[*rule-format*]:
¬ *member DenyAll a* ⟶ *first-destNet a* ∈ *set* (*net-list-aux* (*a* # *p*))
⟨*proof*⟩

lemma *firstInNeta*[*rule-format*]:
¬ *member DenyAll a* ⟶ *first-srcNet a* ∈ *set* (*net-list-aux* (*a* # *p*))
⟨*proof*⟩

lemma *disjComm*: *disjSD-2 a b* ⟹ *disjSD-2 b a*
⟨*proof*⟩

lemma *disjSD2aux*:
disjSD-2 a b ⟹ ¬ *member DenyAll a* ⟹ ¬ *member DenyAll b* ⟹
disjSD-2 (*DenyAllFromTo* (*first-srcNet a*) (*first-destNet a*) ⊕
DenyAllFromTo (*first-destNet a*) (*first-srcNet a*) ⊕ *a*)
b
⟨*proof*⟩

lemma *noDA1eq*[*rule-format*]: *noDenyAll p* ⟶ *noDenyAll1 p*
⟨*proof*⟩

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

lemma *disjSD-2IDa*:
disjSD-2 x y $\Longrightarrow$
$\neg$ *member DenyAll x* $\Longrightarrow$
$\neg$ *member DenyAll y* $\Longrightarrow$
a = *first-srcNet x* $\Longrightarrow$
b = *first-destNet x* $\Longrightarrow$
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 $\Longrightarrow$
$\forall$ *s*. *s* $\in$ *set p* $\longrightarrow$ *disjSD-2 a s* $\Longrightarrow$
$\neg$ *member DenyAll a* $\Longrightarrow$
DenyAllFromTo (*first-srcNet s*) (*first-destNet s*) $\oplus$
DenyAllFromTo (*first-destNet s*) (*first-srcNet s*) $\oplus$ *s* $\in$ *set* (*insertDenies p*) $\Longrightarrow$
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 $\Longrightarrow$ $\forall$ *s*. *s* $\in$ *set p* $\longrightarrow$ *disjSD-2 a s* $\Longrightarrow$
s $\in$ *set* (*insertDenies p*) $\Longrightarrow$ $\neg$ *member DenyAll a* $\Longrightarrow$
disjSD-2 a s
⟨*proof*⟩

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

noDenyAll1 p ⟶ *separated p* ⟶ *separated* (*insertDenies p*)
⟨*proof*⟩

lemma *aNDDA*[*rule-format*]: *allNetsDistinct p* ⟶ *allNetsDistinct*(*DenyAll*#*p*)
⟨*proof*⟩

lemma *OTNConc*[*rule-format*]: *OnlyTwoNets* (*y* # *z*) ⟶ *OnlyTwoNets z*
⟨*proof*⟩

lemma *first-bothNetsd*: ¬ *member DenyAll x* ⟹ *first-bothNet x* = {*first-srcNet x*, *first-destNet x*}
⟨*proof*⟩

lemma *bNaux*:
¬ *member DenyAll x* ⟹ ¬ *member DenyAll y* ⟹
first-bothNet x = *first-bothNet y* ⟹
{*first-srcNet x*, *first-destNet x*} = {*first-srcNet y*, *first-destNet y*}
⟨*proof*⟩

lemma *setPair*: $\{a,b\} = \{a,d\} \Longrightarrow b = d$
⟨*proof*⟩

lemma *setPair1*: $\{a,b\} = \{d,a\} \Longrightarrow b = d$
⟨*proof*⟩

lemma *setPair4*: $\{a,b\} = \{c,d\} \Longrightarrow a \neq c \Longrightarrow a = d$
⟨*proof*⟩

lemma *otnaux1*: $\{x, y, x, y\} = \{x,y\}$
⟨*proof*⟩

lemma *OTNIDaux4*: $\{x,y,x\} = \{y,x\}$
⟨*proof*⟩

lemma *setPair5*: $\{a,b\} = \{c,d\} \Longrightarrow a \neq c \Longrightarrow a = d$
⟨*proof*⟩

lemma *otnaux*:
⟦*first-bothNet x* = *first-bothNet y*; ¬ *member DenyAll x*; ¬ *member DenyAll y*;
onlyTwoNets y; *onlyTwoNets x*⟧ ⟹
onlyTwoNets (*x* ⊕ *y*)
⟨*proof*⟩

lemma *OTNSepaux*:

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

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

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

⟨*proof*⟩

lemma *isInAlternativeListb*:
(*aa* ∈ *set* (*net-list-aux p*))⟹ *aa* ∈ *set* (*net-list-aux* (*a* # *p*))
⟨*proof*⟩

lemma *ANDSepaux*: *allNetsDistinct* (*x* # *y* # *z*) ⟹ *allNetsDistinct* (*x* ⊕ *y* # *z*)
⟨*proof*⟩

lemma *netlistalternativeSeparateaux*:
net-list-aux [*y*] @ *net-list-aux z* = *net-list-aux* (*y* # *z*)
⟨*proof*⟩

lemma *netlistalternativeSeparate*: *net-list-aux p* = *net-list-aux* (*separate p*)
⟨*proof*⟩

lemma *ANDSepaux2*:
allNetsDistinct(*x*#*y*#*z*) ⟹ *allNetsDistinct*(*separate*(*y*#*z*)) ⟹ *allNetsDistinct*(*x*#*separate*(*y*#*z*))
⟨*proof*⟩

lemma *ANDSep*[*rule-format*]: *allNetsDistinct p* ⟶ *allNetsDistinct*(*separate p*)
⟨*proof*⟩

lemma *wp1-alternativesep*[*rule-format*]:
wellformed-policy1-strong p ⟶ *wellformed-policy1-strong* (*separate p*)
⟨*proof*⟩

lemma *noDAsort*[*rule-format*]: *noDenyAll1 p* ⟶ *noDenyAll1* (*sort p l*)
⟨*proof*⟩

lemma *OTNSC*[*rule-format*]: *singleCombinators p* ⟶ *OnlyTwoNets p*
⟨*proof*⟩

lemma *fMTaux*: ¬ *member DenyAll x* ⟹ *first-bothNet x* ≠ {}
⟨*proof*⟩

lemma *fl2*[*rule-format*]: *firstList* (*separate p*) = *firstList p*
⟨*proof*⟩

lemma *fl3*[*rule-format*]: *NetsCollected p* ⟶ (*first-bothNet x* ≠ *firstList p* ⟶
(∀ *a*∈*set p*. *first-bothNet x* ≠ *first-bothNet a*))⟶ *NetsCollected* (*x*#*p*)

⟨*proof*⟩

lemma *sortedConc*[*rule-format*]: *sorted* (*a* # *p*) *l* ⟶ *sorted p l*
⟨*proof*⟩

lemma *smalleraux2*:
$\{a,b\} \in set\ l \Longrightarrow \{c,d\} \in set\ l \Longrightarrow \{a,b\} \neq \{c,d\} \Longrightarrow$
smaller (*DenyAllFromTo a b*) (*DenyAllFromTo c d*) *l* ⟹
¬ *smaller* (*DenyAllFromTo c d*) (*DenyAllFromTo a b*) *l*
⟨*proof*⟩

lemma *smalleraux2a*:
$\{a,b\} \in set\ l \Longrightarrow \{c,d\} \in set\ l \Longrightarrow \{a,b\} \neq \{c,d\} \Longrightarrow$
smaller (*DenyAllFromTo a b*) (*AllowPortFromTo c d p*) *l* ⟹
¬ *smaller* (*AllowPortFromTo c d p*) (*DenyAllFromTo a b*) *l*
⟨*proof*⟩

lemma *smalleraux2b*:
$\{a,b\} \in set\ l \Longrightarrow \{c,d\} \in set\ l \Longrightarrow \{a,b\} \neq \{c,d\} \Longrightarrow y = DenyAllFromTo\ a\ b \Longrightarrow$
smaller (*AllowPortFromTo c d p*) *y l* ⟹
¬ *smaller y* (*AllowPortFromTo c d p*) *l*
⟨*proof*⟩

lemma *smalleraux2c*:
$\{a,b\} \in set\ l \Longrightarrow \{c,d\} \in set\ l \Longrightarrow \{a,b\} \neq \{c,d\} \Longrightarrow y = AllowPortFromTo\ a\ b\ q \Longrightarrow$
smaller (*AllowPortFromTo c d p*) *y l* ⟹ ¬ *smaller y* (*AllowPortFromTo c d p*) *l*
⟨*proof*⟩

lemma *smalleraux3*:
assumes *x* ∈ *set l* **and** *y* ∈ *set l* **and** *x* ≠ *y* **and** *x* = *bothNet a* **and** *y* = *bothNet b*
and *smaller a b l* **and** *singleCombinators* [*a*] **and** *singleCombinators* [*b*]
shows ¬ *smaller b a l*
⟨*proof*⟩

lemma *smalleraux3a*:
a ≠ *DenyAll* ⟹ *b* ≠ *DenyAll* ⟹ *in-list b l* ⟹ *in-list a l* ⟹
bothNet a ≠ *bothNet b* ⟹ *smaller a b l* ⟹ *singleCombinators* [*a*] ⟹
singleCombinators [*b*] ⟹ ¬ *smaller b a l*
⟨*proof*⟩

lemma *posaux*[*rule-format*]: *position a l* < *position b l* ⟶ *a* ≠ *b*
⟨*proof*⟩

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

$a \in set\ l \longrightarrow b \in set\ l \longrightarrow a \neq b \longrightarrow position\ a\ l \neq\ position\ b\ l$
⟨*proof*⟩

lemma *notSmallerTransaux*[*rule-format*]:
$x \neq DenyAll \Longrightarrow r \neq DenyAll \Longrightarrow$
$singleCombinators\ [x] \Longrightarrow\ singleCombinators\ [y] \Longrightarrow\ singleCombinators\ [r] \Longrightarrow$
$\neg\ smaller\ y\ x\ l \Longrightarrow smaller\ x\ y\ l \Longrightarrow smaller\ x\ r\ l \Longrightarrow smaller\ y\ r\ l \Longrightarrow$
$in\text{-}list\ x\ l \Longrightarrow in\text{-}list\ y\ l \Longrightarrow in\text{-}list\ r\ l \Longrightarrow \neg\ smaller\ r\ x\ l$
⟨*proof*⟩

lemma *notSmallerTrans*[*rule-format*]:
$x \neq DenyAll \longrightarrow r \neq DenyAll \longrightarrow singleCombinators\ (x\#y\#z) \longrightarrow$
$\neg\ smaller\ y\ x\ l \longrightarrow sorted\ (x\#y\#z)\ l \longrightarrow r \in set\ z \longrightarrow$
$all\text{-}in\text{-}list\ (x\#y\#z)\ l \longrightarrow \neg\ smaller\ r\ x\ l$
⟨*proof*⟩

lemma *NCSaux1*[*rule-format*]:
$noDenyAll\ p \longrightarrow \{x,\ y\} \in set\ l \longrightarrow\ all\text{-}in\text{-}list\ p\ l \longrightarrow singleCombinators\ p \longrightarrow$
$sorted\ (DenyAllFromTo\ x\ y\ \#\ p)\ l \longrightarrow \{x,\ y\} \neq firstList\ p \longrightarrow$
$DenyAllFromTo\ u\ v \in set\ p \longrightarrow \{x,\ y\} \neq \{u,\ v\}$
⟨*proof*⟩

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

lemma *posaux4*[*rule-format*]:
$singleCombinators\ [a] \longrightarrow a \neq DenyAll \longrightarrow b \neq DenyAll \longrightarrow in\text{-}list\ a\ l \longrightarrow in\text{-}list\ b\ l \longrightarrow$
$smaller\ a\ b\ \ l \longrightarrow x = (bothNet\ a) \longrightarrow\ y = (bothNet\ b) \longrightarrow$
$position\ x\ l <= position\ y\ l$
⟨*proof*⟩

lemma *NCSaux2*[*rule-format*]:
$noDenyAll\ p \longrightarrow \{a,\ b\} \in set\ l \longrightarrow all\text{-}in\text{-}list\ p\ l \longrightarrow singleCombinators\ p \longrightarrow$
$sorted\ (DenyAllFromTo\ a\ b\ \#\ p)\ l \longrightarrow \{a,\ b\} \neq firstList\ p \longrightarrow$
$AllowPortFromTo\ u\ v\ w \in set\ p \longrightarrow\ \{a,\ b\} \neq \{u,\ v\}$
⟨*proof*⟩

lemma *NCSaux3*[*rule-format*]:
$noDenyAll\ p \longrightarrow \{a,\ b\} \in set\ l \longrightarrow\ all\text{-}in\text{-}list\ p\ l \longrightarrow singleCombinators\ p \longrightarrow$
$sorted\ (AllowPortFromTo\ a\ b\ w\ \#\ p)\ l \longrightarrow \{a,\ b\} \neq firstList\ p \longrightarrow$
$DenyAllFromTo\ u\ v \in set\ p \longrightarrow \{a,\ b\} \neq \{u,\ v\}$
⟨*proof*⟩

lemma *NCSaux4* [*rule-format*]:
noDenyAll p ⟶ {*a*, *b*} ∈ *set l* ⟶ *all-in-list p l* ⟶ *singleCombinators p* ⟶ *sorted* (*AllowPortFromTo a b c* # *p*) *l* ⟶ {*a*, *b*} ≠ *firstList p* ⟶ *AllowPortFromTo u v w* ∈ *set p* ⟶ {*a*, *b*} ≠ {*u*, *v*}
⟨*proof*⟩

lemma *NetsCollectedSorted*[*rule-format*]:
noDenyAll1 p ⟶ *all-in-list p l* ⟶ *singleCombinators p* ⟶ *sorted p l* ⟶ *NetsCollected p*
⟨*proof*⟩

lemma *NetsCollectedSort*: *distinct p* ⟹*noDenyAll1 p* ⟹ *all-in-list p l* ⟹
singleCombinators p ⟹ *NetsCollected* (*sort p l*)
⟨*proof*⟩

lemma *fBNsep*[*rule-format*]: (∀ *a*∈*set z*. {*b*,*c*} ≠ *first-bothNet a*) ⟶
(∀ *a*∈*set* (*separate z*). {*b*,*c*} ≠ *first-bothNet a*)
⟨*proof*⟩

lemma *fBNsep1* [*rule-format*]: (∀ *a*∈*set z*. *first-bothNet x* ≠ *first-bothNet a*) ⟶
(∀ *a*∈*set* (*separate z*). *first-bothNet x* ≠ *first-bothNet a*)
⟨*proof*⟩

lemma *NetsCollectedSepauxa*:
{*b*,*c*}≠*firstList z* ⟹ *noDenyAll1 z* ⟹ ∀ *a*∈*set z*. {*b*,*c*}≠*first-bothNet a* ⟹ *NetsCollected z* ⟹
NetsCollected (*separate z*) ⟹ {*b*, *c*} ≠ *firstList* (*separate z*) ⟹ *a* ∈ *set* (*separate z*) ⟹
{*b*, *c*} ≠ *first-bothNet a*
⟨*proof*⟩

lemma *NetsCollectedSepaux*:
first-bothNet (*x*::(′*a*,′*b*)*Combinators*) ≠ *first-bothNet y* ⟹ ¬ *member DenyAll y* ∧ *noDenyAll z* ⟹
(∀ *a*∈*set z*. *first-bothNet x* ≠ *first-bothNet a*) ∧ *NetsCollected* (*y* # *z*) ⟹
NetsCollected (*separate* (*y* # *z*)) ⟹ *first-bothNet x* ≠ *firstList* (*separate* (*y* # *z*)) ⟹
a ∈ *set* (*separate* (*y* # *z*)) ⟹
first-bothNet (*x*::(′*a*,′*b*)*Combinators*) ≠ *first-bothNet* (*a*::(′*a*,′*b*)*Combinators*)
⟨*proof*⟩

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

noDenyAll1 p ⟶ *NetsCollected p* ⟶ *NetsCollected* (*separate p*)
⟨*proof*⟩

lemma *OTNaux*:
onlyTwoNets a ⟹ ¬ *member DenyAll a* ⟹ (*x*,*y*) ∈ *sdnets a* ⟹
(*x* = *first-srcNet a* ∧ *y* = *first-destNet a*) ∨ (*x* = *first-destNet a* ∧ *y* = *first-srcNet a*)
⟨*proof*⟩

lemma *sdnets-charn*: *onlyTwoNets a* ⟹ ¬ *member DenyAll a* ⟹
sdnets a = {(*first-srcNet a*,*first-destNet a*)} ∨
sdnets a = {(*first-srcNet a*, *first-destNet a*),(*first-destNet a*, *first-srcNet a*)}
⟨*proof*⟩

lemma *first-bothNet-charn*[*rule-format*]:
¬ *member DenyAll a* ⟶ *first-bothNet a* = {*first-srcNet a*, *first-destNet a*}
⟨*proof*⟩

lemma *sdnets-noteq*:
onlyTwoNets a ⟹ *onlyTwoNets aa* ⟹ *first-bothNet a* ≠ *first-bothNet aa* ⟹
¬ *member DenyAll a* ⟹ ¬ *member DenyAll aa* ⟹ *sdnets a* ≠ *sdnets aa*
⟨*proof*⟩

lemma *fbn-noteq*:
onlyTwoNets a ⟹ *onlyTwoNets aa* ⟹ *first-bothNet a* ≠ *first-bothNet aa* ⟹
¬ *member DenyAll a* ⟹ ¬ *member DenyAll aa* ⟹ *allNetsDistinct* [*a*, *aa*] ⟹
first-srcNet a ≠ *first-srcNet aa* ∨ *first-srcNet a* ≠ *first-destNet aa* ∨
first-destNet a ≠ *first-srcNet aa* ∨ *first-destNet a* ≠ *first-destNet aa*
⟨*proof*⟩

lemma *NCisSD2aux*:
assumes *1*: *onlyTwoNets a* **and** *2* : *onlyTwoNets aa* **and** *3* : *first-bothNet a* ≠ *first-bothNet aa*
and *4*: ¬ *member DenyAll a* **and** *5*: ¬ *member DenyAll aa* **and** *6*: *allNetsDistinct* [*a*, *aa*]
shows *disjSD-2 a aa*
⟨*proof*⟩

lemma *ANDaux3*[*rule-format*]:
y ∈ *set xs* ⟶ *a* ∈ *set* (*net-list-aux* [*y*]) ⟶ *a* ∈ *set* (*net-list-aux xs*)
⟨*proof*⟩

lemma *ANDaux2*:

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

lemma *NCisSD2*[*rule-format*]:
¬ *member DenyAll a* $\Longrightarrow$ *OnlyTwoNets* (*a*#*p*) $\Longrightarrow$
NetsCollected2 (*a* # *p*) $\Longrightarrow$ *NetsCollected* (*a*#*p*) $\Longrightarrow$
noDenyAll (*p*) $\Longrightarrow$ *allNetsDistinct* (*a* # *p*) $\Longrightarrow$ *s* $\in$ *set p* $\Longrightarrow$
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* $\Longrightarrow$ $\{(a,b).\ a = n\} \neq \{(a,b).\ a = m\}$
⟨*proof*⟩

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

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

lemma *sets-distinct6*: $(m::int) < n \Longrightarrow \{(a,b).\ a = n\} \neq \{(a,b).\ a = m\}$
⟨*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* $\Longrightarrow$
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*: $x \in dom(C\ b) \Longrightarrow C\ (a \oplus b)\ x = (C\ b)\ x$
⟨*proof*⟩

lemma *wp2Conc*[*rule-format*]: *wellformed-policy2* (*x#xs*) $\Longrightarrow$ *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* $\Longrightarrow$ *applied-rule-rev C x p* $\neq$ *None*
⟨*proof*⟩

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

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

⟨proof⟩

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

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

lemma *CConcStartaux*: $C\ a\ x = None \Longrightarrow (C\ aa\ {+}{+}\ C\ a)\ x = C\ aa\ x$
⟨proof⟩

lemma *CConcStart*[*rule-format*]:
$xs \neq [] \longrightarrow C\ a\ x = None \longrightarrow C\ (list2FWpolicy\ (xs\ @\ [a]))\ x = C\ (list2FWpolicy\ xs)\ x$
⟨proof⟩

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

lemma *mr-is-C*[*rule-format*]:
$applied\text{-}rule\text{-}rev\ C\ x\ p = Some\ a \longrightarrow C\ (list2FWpolicy\ (p))\ x = C\ a\ x$
⟨proof⟩

lemma *CConcStart2*:
$p \neq [] \Longrightarrow x \notin dom\ (C\ a) \Longrightarrow C\ (list2FWpolicy\ (p\ @\ [a]))\ x = C\ (list2FWpolicy\ p)\ x$
⟨proof⟩

lemma *CConcEnd1*:
$q\ @\ p \neq [] \Longrightarrow x \notin dom\ (C\ a) \Longrightarrow C\ (list2FWpolicy\ (q\ @\ p\ @\ [a]))\ x = C\ (list2FWpolicy\ (q\ @\ p))\ x$
⟨proof⟩

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

lemma *bar3*:
$x \in dom\ (C\ (list2FWpolicy\ (xs\ @\ [xa]))) \Longrightarrow x \in dom\ (C\ (list2FWpolicy\ xs)) \lor x \in$

dom (*C xa*)
⟨*proof*⟩

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

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

lemma *domConc*:
$x \in$ *dom* (*C* (*list2FWpolicy b*)) $\Longrightarrow b \neq [] \Longrightarrow x \in$ *dom* (*C* (*list2FWpolicy* (*a* @ *b*)))
⟨*proof*⟩

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

lemma *C-eq-if-mr-eq2*:
applied-rule-rev C x a = $\lfloor r \rfloor \Longrightarrow$
applied-rule-rev C x b = $\lfloor r \rfloor \Longrightarrow a \neq [] \Longrightarrow b \neq [] \Longrightarrow$
C (*list2FWpolicy a*) *x* = *C* (*list2FWpolicy b*) *x*
⟨*proof*⟩

lemma *nMRtoNone*[*rule-format*]:
$p \neq [] \longrightarrow$ *applied-rule-rev C x p* = *None* $\longrightarrow$ *C* (*list2FWpolicy p*) *x* = *None*
⟨*proof*⟩

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*
⟨*proof*⟩

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

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*)
⟨*proof*⟩

lemma *foo3D*:
  $wellformed\text{-}policy1\ p \Longrightarrow p = DenyAll\ \#\ ps \Longrightarrow$
   $applied\text{-}rule\text{-}rev\ C\ x\ p = \lfloor DenyAll \rfloor \Longrightarrow r \in set\ ps \Longrightarrow x \notin dom\ (C\ r)$
  ⟨*proof*⟩

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

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

lemma *mr-first*:
  $x \in dom\ (C\ b) \Longrightarrow \forall r.\ r \in set\ c \longrightarrow x \notin dom\ (C\ r) \Longrightarrow s = b\ \#\ c \Longrightarrow applied\text{-}rule\text{-}rev\ C\ x\ s = \lfloor b \rfloor$
  ⟨*proof*⟩

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

lemma *foo8*:
  $\forall r.\ r \in set\ p \wedge x \in dom\ (C\ r) \longrightarrow r = a \Longrightarrow set\ p = set\ s \Longrightarrow$
   $\forall r.\ r \in set\ s \wedge x \in dom\ (C\ r) \longrightarrow r = a$
  ⟨*proof*⟩

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

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

lemma *wp3Conc*[*rule-format*]: $wellformed\text{-}policy3\ (a\#p) \longrightarrow wellformed\text{-}policy3\ p$
  ⟨*proof*⟩

lemma *foo98*[*rule-format*]:
applied-rule-rev C x (*aa* # *p*) = *Some a* ⟶ *x* ∈ *dom* (*C r*) ⟶ *r* ∈ *set p* ⟶ *a* ∈ *set p*
⟨*proof*⟩

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

lemma *DAAux*[*simp*]: *x* ∈ *dom* (*C DenyAll*)
⟨*proof*⟩

lemma *mrSet*[*rule-format*]: *applied-rule-rev C x p* = *Some r* ⟶ *r* ∈ *set p*
⟨*proof*⟩

lemma *mr-not-Conc*: *singleCombinators p* ⟹ *applied-rule-rev C x p* ≠ *Some* (*a*⊕*b*)
⟨*proof*⟩

lemma *foo25*[*rule-format*]: *wellformed-policy3* (*p*@[*x*]) ⟶ *wellformed-policy3 p*
⟨*proof*⟩

lemma *mr-in-dom*[*rule-format*]: *applied-rule-rev C x p* = *Some a* ⟶ *x* ∈ *dom* (*C a*)
⟨*proof*⟩

lemma *wp3EndMT*[*rule-format*]:
wellformed-policy3 (*p*@[*xs*]) ⟶ *AllowPortFromTo a b po* ∈ *set p* ⟶
dom (*C* (*AllowPortFromTo a b po*)) ∩ *dom* (*C xs*) = {}
⟨*proof*⟩

lemma *foo29*: ⟦*dom* (*C a*) ≠ {}; *dom* (*C a*) ∩ *dom* (*C b*) = {}⟧ ⟹ *a* ≠ *b* ⟨*proof*⟩

lemma *foo28*:
AllowPortFromTo a b po ∈ *set p* ⟹ *dom* (*C* (*AllowPortFromTo a b po*)) ≠ {} ⟹
wellformed-policy3 (*p* @ [*x*]) ⟹ *x* ≠ *AllowPortFromTo a b po*
⟨*proof*⟩

lemma *foo28a*[*rule-format*]: *x* ∈ *dom* (*C a*) ⟹ *dom* (*C a*) ≠ {} ⟨*proof*⟩

lemma *allow-deny-dom*[*simp*]:
dom (*C* (*AllowPortFromTo a b po*)) ⊆ *dom* (*C* (*DenyAllFromTo a b*))
⟨*proof*⟩

lemma *DenyAllowDisj*:
dom (*C* (*AllowPortFromTo a b p*)) $\neq$ {} $\Longrightarrow$
dom (*C* (*DenyAllFromTo a b*)) $\cap$ *dom* (*C* (*AllowPortFromTo a b p*)) $\neq$ {}
⟨*proof*⟩

lemma *foo31*:
$\forall$ *r*. *r* $\in$ *set p* $\wedge$ *x* $\in$ *dom* (*C r*) $\longrightarrow$
r = *AllowPortFromTo a b po* $\vee$ *r* = *DenyAllFromTo a b* $\vee$ *r* = *DenyAll* $\Longrightarrow$
set p = *set s* $\Longrightarrow$
$\forall$ *r*. *r* $\in$ *set s* $\wedge$ *x* $\in$ *dom* (*C r*) $\longrightarrow$ *r*=*AllowPortFromTo a b po* $\vee$ *r* = *DenyAllFromTo a b* $\vee$ *r* = *DenyAll*
⟨*proof*⟩

lemma *wp1-auxa*:
wellformed-policy1-strong p$\Longrightarrow$($\exists$ *r*. *applied-rule-rev C x p* = *Some r*)
⟨*proof*⟩

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

lemma *domTrans*: *dom a* $\subseteq$ *dom b* $\Longrightarrow$ *dom b* $\cap$ *dom c* = {} $\Longrightarrow$ *dom a* $\cap$ *dom c* = {} ⟨*proof*⟩

lemma *DomInterAllowsMT*:
twoNetsDistinct a b c d $\Longrightarrow$
dom (*C* (*AllowPortFromTo a b p*)) $\cap$ *dom* (*C* (*AllowPortFromTo c d po*)) = {}
⟨*proof*⟩

lemma *DomInterAllowsMT-Ports*:
p $\neq$ *po* $\Longrightarrow$ *dom* (*C* (*AllowPortFromTo a b p*)) $\cap$ *dom* (*C* (*AllowPortFromTo c d po*)) = {}
⟨*proof*⟩

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

lemma *DistinctNetsDenyAllow*:
DenyAllFromTo b c $\in$ *set p* $\Longrightarrow$

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

lemma *DistinctNetsAllowAllow*:
$AllowPortFromTo\ b\ c\ poo \in set\ p \Longrightarrow$
$AllowPortFromTo\ a\ d\ po \in set\ p \Longrightarrow$
$allNetsDistinct\ p \Longrightarrow$
$dom\ (C\ (AllowPortFromTo\ b\ c\ poo)) \cap dom\ (C\ (AllowPortFromTo\ a\ d\ po)) \neq \{\} \Longrightarrow$
$b = a \wedge c = d \wedge poo = po$
$\langle proof \rangle$

lemma *WP2RS2*[*simp*]:
$singleCombinators\ p \Longrightarrow distinct\ p \Longrightarrow allNetsDistinct\ p \Longrightarrow$
$wellformed\text{-}policy2\ (removeShadowRules2\ p)$
$\langle proof \rangle$

lemma *AD-aux*:
$AllowPortFromTo\ a\ b\ po \in set\ p \Longrightarrow DenyAllFromTo\ c\ d \in set\ p \Longrightarrow$
$allNetsDistinct\ p \Longrightarrow singleCombinators\ p \Longrightarrow a \neq c \vee b \neq d \Longrightarrow$
$dom\ (C\ (AllowPortFromTo\ a\ b\ po)) \cap dom\ (C\ (DenyAllFromTo\ c\ d)) = \{\}$
$\langle proof \rangle$

lemma *sorted-WP2*[*rule-format*]: $sorted\ p\ l \longrightarrow all\text{-}in\text{-}list\ p\ l \longrightarrow distinct\ p \longrightarrow$
$allNetsDistinct\ p \longrightarrow singleCombinators\ p \longrightarrow wellformed\text{-}policy2\ p$
$\langle proof \rangle$

lemma *wellformed2-sorted*[*simp*]:
$all\text{-}in\text{-}list\ p\ l \Longrightarrow distinct\ p \Longrightarrow allNetsDistinct\ p \Longrightarrow$
$singleCombinators\ p \Longrightarrow wellformed\text{-}policy2\ (sort\ p\ l)$
$\langle proof \rangle$

lemma *wellformed2-sortedQ*[*simp*]: $[\![all\text{-}in\text{-}list\ p\ l;\ distinct\ p;\ allNetsDistinct\ p;$
$singleCombinators\ p]\!] \Longrightarrow wellformed\text{-}policy2\ (qsort\ p\ l)$
$\langle proof \rangle$

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

lemma *C-eq-RS1n*:
$C(list2FWpolicy\ (removeShadowRules1\text{-}alternative\ p)) = C(list2FWpolicy\ p)$

⟨*proof*⟩

lemma *C-eq-RS1*[*simp*]:
p ≠ [] ⟹ *C*(*list2FWpolicy* (*removeShadowRules1 p*)) = *C*(*list2FWpolicy p*)
⟨*proof*⟩

lemma *EX-MR-aux*[*rule-format*]:
applied-rule-rev C x (*DenyAll* # *p*) ≠ *Some DenyAll* ⟶ (∃ *y*. *applied-rule-rev C x p* = *Some y*)
⟨*proof*⟩

lemma *EX-MR* :
applied-rule-rev C x p ≠ ⌊*DenyAll*⌋ ⟹ *p* = *DenyAll* # *ps* ⟹
applied-rule-rev C x p = *applied-rule-rev C x ps*
⟨*proof*⟩

lemma *mr-not-DA*:
wellformed-policy1-strong s ⟹
applied-rule-rev C x p = ⌊*DenyAllFromTo a ab*⌋ ⟹ *set p* = *set s* ⟹
applied-rule-rev C x s ≠ ⌊*DenyAll*⌋
⟨*proof*⟩

lemma *domsMT-notND-DD*:
dom (*C* (*DenyAllFromTo a b*)) ∩ *dom* (*C* (*DenyAllFromTo c d*)) ≠ {} ⟹ ¬ *netsDistinct a c*
⟨*proof*⟩

lemma *domsMT-notND-DD2*:
dom (*C* (*DenyAllFromTo a b*)) ∩ *dom* (*C* (*DenyAllFromTo c d*)) ≠ {} ⟹ ¬ *netsDistinct b d*
⟨*proof*⟩

lemma *domsMT-notND-DD3*:
x ∈ *dom* (*C* (*DenyAllFromTo a b*)) ⟹ *x* ∈ *dom* (*C* (*DenyAllFromTo c d*)) ⟹ ¬ *netsDistinct a c*
⟨*proof*⟩

lemma *domsMT-notND-DD4*:
x ∈ *dom* (*C* (*DenyAllFromTo a b*)) ⟹ *x* ∈ *dom* (*C* (*DenyAllFromTo c d*)) ⟹ ¬ *netsDistinct b d*
⟨*proof*⟩

lemma *NetsEq-if-sameP-DD*:
allNetsDistinct p ⟹ *u* ∈ *set p* ⟹ *v* ∈ *set p* ⟹ *u* = *DenyAllFromTo a b* ⟹

$v = DenyAllFromTo\ c\ d \Longrightarrow x \in dom\ (C\ u) \Longrightarrow x \in dom\ (C\ v) \Longrightarrow a = c \wedge b = d$
⟨*proof*⟩

lemma *rule-charn1*:
assumes *aND*: *allNetsDistinct p*
and *mr-is-allow*: $applied\text{-}rule\text{-}rev\ C\ x\ p = Some\ (AllowPortFromTo\ a\ b\ po)$
and *SC*: *singleCombinators p*
and *inp*: $r \in set\ p$
and *inDom*: $x \in dom\ (C\ r)$
shows $(r = AllowPortFromTo\ a\ b\ po \vee r = DenyAllFromTo\ a\ b \vee r = DenyAll)$
⟨*proof*⟩

lemma *none-MT-rulessubset*[*rule-format*]:
$none\text{-}MT\text{-}rules\ C\ a \longrightarrow set\ b \subseteq set\ a \longrightarrow none\text{-}MT\text{-}rules\ C\ b$
⟨*proof*⟩

lemma *nMTSort*: $none\text{-}MT\text{-}rules\ C\ p \Longrightarrow none\text{-}MT\text{-}rules\ C\ (sort\ p\ l)$
⟨*proof*⟩

lemma *nMTSortQ*: $none\text{-}MT\text{-}rules\ C\ p \Longrightarrow none\text{-}MT\text{-}rules\ C\ (qsort\ p\ l)$
⟨*proof*⟩

lemma *wp3char*[*rule-format*]:
$none\text{-}MT\text{-}rules\ C\ xs \wedge C\ (AllowPortFromTo\ a\ b\ po) = \emptyset \wedge well\text{-}formed\text{-}policy3(xs@[DenyAllFromTo\ a\ b]) \longrightarrow$
$AllowPortFromTo\ a\ b\ po \notin set\ xs$
⟨*proof*⟩

lemma *wp3charn*[*rule-format*]:
assumes *domAllow*: $dom\ (C\ (AllowPortFromTo\ a\ b\ po)) \neq \{\}$
and *wp3*: $wellformed\text{-}policy3\ (xs\ @\ [DenyAllFromTo\ a\ b])$
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*: $AllowPortFromTo\ c\ d\ po \in set\ p$
and *x-in-dom-allow*: $x \in dom\ (C\ (AllowPortFromTo\ c\ d\ po))$
shows $applied\text{-}rule\text{-}rev\ C\ x\ p = Some\ (AllowPortFromTo\ c\ d\ po)$
⟨*proof*⟩

lemma *rule-charn3*:
wellformed-policy1 p $\Longrightarrow$ *allNetsDistinct p* $\Longrightarrow$ *singleCombinators p* $\Longrightarrow$
wellformed-policy3 p $\Longrightarrow$ *applied-rule-rev C x p* = $\lfloor$*DenyAllFromTo c d*$\rfloor$ $\Longrightarrow$
AllowPortFromTo a b po $\in$ *set p* $\Longrightarrow$ *x* $\notin$ *dom* (*C* (*AllowPortFromTo a b po*))
⟨*proof*⟩

lemma *rule-charn4*:
assumes *wp1*: *wellformed-policy1 p*
and *aND*: *allNetsDistinct p*
and *SC*: *singleCombinators p*
and *wp3*: *wellformed-policy3 p*
and *DA*: *DenyAll* $\notin$ *set p*
and *mr*: *applied-rule-rev C x p* = *Some* (*DenyAllFromTo a b*)
and *rinp*: *r* $\in$ *set p*
and *xindom*: *x* $\in$ *dom* (*C r*)
shows *r* = *DenyAllFromTo a b*
⟨*proof*⟩

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* $\Longrightarrow$
set p = *set s* $\Longrightarrow$ *r* $\in$ *set s* $\Longrightarrow$ *x* $\in$ *dom* (*C r*) $\Longrightarrow$
r = *AllowPortFromTo a b po* $\vee$ *r* = *DenyAllFromTo a b* $\vee$ *r* = *DenyAll*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *rule-charnDAFT*:
wellformed-policy1-strong p $\Longrightarrow$ *allNetsDistinct p* $\Longrightarrow$ *singleCombinators p* $\Longrightarrow$
wellformed-policy3 p $\Longrightarrow$ *applied-rule-rev C x p* = $\lfloor$*DenyAllFromTo a b*$\rfloor$ $\Longrightarrow$ *r* $\in$ *set* (*tl p*) $\Longrightarrow$
x $\in$ *dom* (*C r*) $\Longrightarrow$ *r* = *DenyAllFromTo a b*
⟨*proof*⟩

lemma *mrDenyAll-is-unique*:

$\llbracket$*wellformed-policy1-strong p*; *applied-rule-rev C x p = Some DenyAll*;
r ∈ *set* (*tl p*)$\rrbracket$ ⟹ *x* ∉ *dom* (*C r*)
⟨*proof*⟩

theorem *C-eq-Sets-mr*:
assumes *sets-eq*: *set p = set s*
and *SC*: *singleCombinators p*
and *wp1-p*: *wellformed-policy1-strong p*
and *wp1-s*: *wellformed-policy1-strong s*
and *wp3-p*: *wellformed-policy3 p*
and *wp3-s*: *wellformed-policy3 s*
and *aND*: *allNetsDistinct p*
shows *applied-rule-rev C x p = applied-rule-rev C x s*
⟨*proof*⟩

lemma *C-eq-Sets*:
singleCombinators p ⟹ *wellformed-policy1-strong p* ⟹ *wellformed-policy1-strong s* ⟹
wellformed-policy3 p ⟹ *wellformed-policy3 s* ⟹ *allNetsDistinct p* ⟹ *set p = set s* ⟹
C (*list2FWpolicy p*) *x* = *C* (*list2FWpolicy s*) *x*
⟨*proof*⟩

lemma *C-eq-sorted*:
distinct p ⟹ *all-in-list p l* ⟹ *singleCombinators p* ⟹ *wellformed-policy1-strong p* ⟹
wellformed-policy3 p ⟹ *allNetsDistinct p* ⟹
C (*list2FWpolicy* (*FWNormalisationCore.sort p l*)) = *C* (*list2FWpolicy p*)
⟨*proof*⟩

lemma *C-eq-sortedQ*:
distinct p ⟹ *all-in-list p l* ⟹ *singleCombinators p* ⟹ *wellformed-policy1-strong p* ⟹
wellformed-policy3 p ⟹ *allNetsDistinct p* ⟹
C (*list2FWpolicy* (*qsort p l*)) = *C* (*list2FWpolicy p*)
⟨*proof*⟩

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

lemma *C-eq-None*[*rule-format*]:
p ≠ [] −−> *applied-rule-rev C x p = None* ⟶ *C* (*list2FWpolicy p*) *x = None*
⟨*proof*⟩

lemma *C-eq-None2*:
$a \neq [] \Longrightarrow b \neq [] \Longrightarrow$ *applied-rule-rev* $C\ x\ a = \bot \Longrightarrow$ *applied-rule-rev* $C\ x\ b = \bot \Longrightarrow$
C (*list2FWpolicy* a) $x = C$ (*list2FWpolicy* b) x
⟨*proof*⟩

lemma *C-eq-RS2*:
wellformed-policy1-strong $p \Longrightarrow C$ (*list2FWpolicy* (*removeShadowRules2* p))$= C$ (*list2FWpolicy* p)
⟨*proof*⟩

lemma *none-MT-rulesRS2*:
none-MT-rules $C\ p \Longrightarrow$ *none-MT-rules* C (*removeShadowRules2* p)
⟨*proof*⟩

lemma *CconcNone*:
dom $(C\ a) = \{\} \Longrightarrow p \neq [] \Longrightarrow C$ (*list2FWpolicy* $(a \# p)$) $x = C$ (*list2FWpolicy* p) x
⟨*proof*⟩

lemma *none-MT-rulesrd*[*rule-format*]:
none-MT-rules $C\ p \longrightarrow$ *none-MT-rules* C (*remdups* p)
⟨*proof*⟩

lemma *DARS3*[*rule-format*]:
DenyAll $\notin$ *set* $p \longrightarrow$ *DenyAll* $\notin$ *set* (*rm-MT-rules* $C\ p$)
⟨*proof*⟩

lemma *DAnMT*: *dom* (C *DenyAll*) $\neq \{\}$
⟨*proof*⟩

lemma *DAnMT2*: C *DenyAll* $\neq$ *Map.empty*
⟨*proof*⟩

lemma *wp1n-RS3*[*rule-format,simp*]:
wellformed-policy1-strong $p \longrightarrow$ *wellformed-policy1-strong* (*rm-MT-rules* $C\ p$)
⟨*proof*⟩

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

lemma *SCRS3*[*rule-format,simp*]:
singleCombinators $p \longrightarrow$ *singleCombinators*(*rm-MT-rules* $C\ p$)
⟨*proof*⟩

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

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

lemma *nlpaux*: $x \notin$ *dom* (C b) $\Longrightarrow$ 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*]:
$p \neq []$ $\Longrightarrow$ C (*list2FWpolicy* (*remdups p*)) = C (*list2FWpolicy p*)
⟨*proof*⟩

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

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

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

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

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

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

lemma *NMPiD*[*rule-format*]: *not-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∈*set*(*policy2list p*) ⟹ *all-in-list*(*policy2list p*) *l* ⟹ *allNetsDistinct*(*policy2list p*) ⟹
C (*list2FWpolicy*
(*FWNormalisationCore.sort*
(*removeShadowRules2* (*remdups* (*rm-MT-rules C*
(*insertDeny* (*removeShadowRules1* (*policy2list p*)))))) *l*)) =
C p
⟨*proof*⟩

lemma *C-eq-All-untilSortedQ*:
DenyAll∈*set*(*policy2list p*) ⟹ *all-in-list*(*policy2list p*) *l* ⟹ *allNetsDistinct*(*policy2list p*) ⟹
C (*list2FWpolicy*
(*qsort* (*removeShadowRules2* (*remdups* (*rm-MT-rules C*
(*insertDeny* (*removeShadowRules1* (*policy2list p*)))))) *l*)) =
C p
⟨*proof*⟩

lemma *C-eq-All-untilSorted-withSimps*:
DenyAll∈*set*(*policy2list p*) ⟹*all-in-list*(*policy2list p*) *l* ⟹ *allNetsDistinct* (*policy2list p*) ⟹
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∈*set*(*policy2list p*)⟹*all-in-list*(*policy2list p*) *l* ⟹ *allNetsDistinct*(*policy2list p*) ⟹

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

lemma *InDomConc*[*rule-format*]:
$p \neq [] \longrightarrow x \in dom\ (C\ (list2FWpolicy\ (p))) \longrightarrow x \in\ dom\ (C\ (list2FWpolicy\ (a\#p)))$
⟨*proof*⟩

lemma *not-in-member*[*rule-format*]: $member\ a\ b \longrightarrow x \notin dom\ (C\ b) \longrightarrow x \notin dom\ (C\ a)$
⟨*proof*⟩

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

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

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

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

lemma *list2FWpolicy-eq*:
$zs \neq [] \Longrightarrow\ C\ (list2FWpolicy\ (x \oplus y\ \#\ z))\ p = C\ (x \oplus list2FWpolicy\ (y\ \#\ z))\ p$
⟨*proof*⟩

lemma *dom-sep*[*rule-format*]:
$x \in dom\ (C\ (list2FWpolicy\ p)) \longrightarrow x \in dom\ (C\ (list2FWpolicy(separate\ p)))$
⟨*proof*⟩

lemma *domdConcStart*[*rule-format*]:
$x \in dom\ (C\ (list2FWpolicy\ (a\#b))) \longrightarrow x \notin dom\ (C\ (list2FWpolicy\ b)) \longrightarrow x \in dom\ (C\ (a))$
⟨*proof*⟩

lemma *sep-dom2-aux*:
$x \in dom\ (C\ (list2FWpolicy\ (a \oplus y\ \#\ z))) \Longrightarrow x \in dom\ (C\ (a \oplus list2FWpolicy\ (y\ \#\ z)))$
⟨*proof*⟩

lemma *sep-dom2-aux2*:
$x \in dom\ (C\ (list2FWpolicy\ (separate\ (y\ \#\ z)))) \longrightarrow x \in dom\ (C\ (list2FWpolicy\ (y\ \#\ z))) \Longrightarrow$
$x \in dom\ (C\ (list2FWpolicy\ (a\ \#\ separate\ (y\ \#\ z)))) \Longrightarrow x \in dom\ (C\ (list2FWpolicy\ (a \oplus y\ \#\ z)))$
⟨*proof*⟩

lemma *sep-dom2*[*rule-format*]:
$x \in dom\ (C\ (list2FWpolicy\ (separate\ p))) \longrightarrow x \in dom\ (C\ (list2FWpolicy(\ p)))$
⟨*proof*⟩

lemma *sepDom*: $dom\ (C\ (list2FWpolicy\ p)) = dom\ (C\ (list2FWpolicy\ (separate\ p)))$
⟨*proof*⟩

lemma *C-eq-s-ext*[*rule-format*]:
$p \neq [] \longrightarrow\ C\ (list2FWpolicy\ (separate\ p))\ a\ = C\ (list2FWpolicy\ p)\ a$
⟨*proof*⟩

lemma *C-eq-s*:
$p \neq [] \Longrightarrow C\ (list2FWpolicy\ (separate\ p)) = C\ (list2FWpolicy\ p)$
⟨*proof*⟩

lemma *sortnMTQ*: $p \neq [] \Longrightarrow qsort\ p\ l \neq []$
⟨*proof*⟩

lemmas *C-eq-Lemmas-sep* =
C-eq-Lemmas sortnMT sortnMTQ RS2-NMT NMPrd not-MTimpnotMT

lemma *C-eq-until-separated*:
$DenyAll \in set(policy2list\ \ p) \Longrightarrow all\text{-}in\text{-}list(policy2list\ \ p)l \Longrightarrow allNetsDistinct\ (policy2list\ p) \Longrightarrow$
$C\ (list2FWpolicy$
$(separate$
$(FWNormalisationCore.sort$

$(removeShadowRules2\ (remdups\ (rm\text{-}MT\text{-}rules\ C$
$(insertDeny\ (removeShadowRules1\ (policy2list\ p)))))\ l))) =$
$C\ p$
⟨proof⟩

lemma *C-eq-until-separatedQ*:
$DenyAll \in set(policy2list\ p) \Longrightarrow all\text{-}in\text{-}list(policy2list\ p) l \Longrightarrow allNetsDistinct(policy2list\ p) \Longrightarrow$
$C\ (list2FWpolicy$
$(separate\ (qsort\ (removeShadowRules2\ (remdups\ (rm\text{-}MT\text{-}rules\ C$
$(insertDeny\ (removeShadowRules1\ (policy2list\ p)))))\ l))) =$
$C\ p$
⟨proof⟩

lemma *domID*[*rule-format*]: $p \neq [] \wedge x \in dom(C(list2FWpolicy\ p)) \longrightarrow$
$x \in dom\ (C(list2FWpolicy(insertDenies\ p)))$
⟨proof⟩

lemma *DA-is-deny*:
$x \in dom\ (C\ (DenyAllFromTo\ a\ b \oplus DenyAllFromTo\ b\ a \oplus DenyAllFromTo\ a\ b)) \Longrightarrow$
$C\ (DenyAllFromTo\ a\ b \oplus DenyAllFromTo\ b\ a \oplus DenyAllFromTo\ a\ b)\ x = Some\ (deny\ ())$
⟨proof⟩

lemma *iDdomAux*[*rule-format*]:
$p \neq [] \longrightarrow x \notin dom\ (C\ (list2FWpolicy\ p)) \longrightarrow$
$x \in dom\ (C\ (list2FWpolicy\ (insertDenies\ p))) \longrightarrow$
$C\ (list2FWpolicy\ (insertDenies\ p))\ x = Some\ (deny\ ())$
⟨proof⟩

lemma *iD-isD*[*rule-format*]:
$p \neq [] \longrightarrow x \notin dom\ (C\ (list2FWpolicy\ p)) \longrightarrow$
$C\ (DenyAll \oplus list2FWpolicy\ (insertDenies\ p))\ x = C\ DenyAll\ x$
⟨proof⟩

lemma *inDomConc*: $[\![\ x \notin dom\ (C\ a);\ x \notin dom\ (C\ (list2FWpolicy\ p))]\!] \Longrightarrow$
$x \notin dom\ (C\ (list2FWpolicy(a\#p)))$
⟨proof⟩

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

$y \notin dom\ (C\ (list2FWpolicy\ p))$
⟨*proof*⟩

lemma *isSepaux*:
$p \neq [] \Longrightarrow noDenyAll\ (a\ \#\ p) \Longrightarrow separated\ (a\ \#\ p) \Longrightarrow$
$x \in dom\ (C\ (DenyAllFromTo\ (first\text{-}srcNet\ a)\ (first\text{-}destNet\ a) \oplus$
$DenyAllFromTo\ (first\text{-}destNet\ a)\ (first\text{-}srcNet\ a) \oplus a)) \Longrightarrow$
$x \notin dom\ (C\ (list2FWpolicy\ p))$
⟨*proof*⟩

lemma *none-MT-rulessep*[*rule-format*]: $none\text{-}MT\text{-}rules\ C\ p \longrightarrow none\text{-}MT\text{-}rules\ C\ (separate\ p)$
⟨*proof*⟩

lemma *dom-id*:
$noDenyAll(a\#p) \Longrightarrow separated(a\#p) \Longrightarrow p \neq [] \Longrightarrow x \notin dom(C(list2FWpolicy\ p)) \Longrightarrow x \in dom\ (C\ a) \Longrightarrow$
$x \notin dom\ (C\ (list2FWpolicy\ (insertDenies\ p)))$
⟨*proof*⟩

lemma *C-eq-iD-aux2*[*rule-format*]:
$noDenyAll1\ p \longrightarrow separated\ p \longrightarrow p \neq [] \longrightarrow x \in dom\ (C\ (list2FWpolicy\ p)) \longrightarrow$
$C(list2FWpolicy\ (insertDenies\ p))\ x = C(list2FWpolicy\ p)\ x$
⟨*proof*⟩

lemma *C-eq-iD*:
$separated\ p \Longrightarrow noDenyAll1\ p \Longrightarrow wellformed\text{-}policy1\text{-}strong\ p \Longrightarrow$
$C\ (list2FWpolicy\ (insertDenies\ p)) = C\ (list2FWpolicy\ p)$
⟨*proof*⟩

lemma *noDAsortQ*[*rule-format*]: $noDenyAll1\ p \longrightarrow noDenyAll1\ (qsort\ p\ l)$
⟨*proof*⟩

lemma *NetsCollectedSortQ*:
$distinct\ p \Longrightarrow noDenyAll1\ p \Longrightarrow all\text{-}in\text{-}list\ p\ l \Longrightarrow singleCombinators\ p \Longrightarrow$
$NetsCollected\ (qsort\ p\ l)$
⟨*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*
⟨*proof*⟩

lemma *C-eq-compileQ*:
DenyAll∈*set*(*policy2list* *p*) ⟹ *all-in-list*(*policy2list* *p*)*l* ⟹ *allNetsDistinct*(*policy2list* *p*) ⟹
C (*list2FWpolicy*
  (*removeAllDuplicates*
    (*insertDenies*
      (*separate*
        (*qsort* (*removeShadowRules2* (*remdups* (*rm-MT-rules* *C*
          (*insertDeny* (*removeShadowRules1* (*policy2list* *p*)))))) *l*))))) =
C *p*
⟨*proof*⟩

lemma *C-eq-normalize*:
DenyAll ∈ *set* (*policy2list* *p*) ⟹ *allNetsDistinct* (*policy2list* *p*) ⟹
all-in-list(*policy2list* *p*)(*Nets-List* *p*) ⟹
C (*list2FWpolicy* (*normalize* *p*)) = *C* *p*
⟨*proof*⟩

lemma *C-eq-normalizeQ*:
DenyAll ∈ *set* (*policy2list* *p*) ⟹ *allNetsDistinct* (*policy2list* *p*) ⟹
all-in-list (*policy2list* *p*) (*Nets-List* *p*) ⟹
C (*list2FWpolicy* (*normalizeQ* *p*)) = *C* *p*
⟨*proof*⟩

lemma *domSubset3*: *dom* (*C* (*DenyAll* ⊕ *x*)) = *dom* (*C* (*DenyAll*))
⟨*proof*⟩

lemma *domSubset4*:
dom (*C* (*DenyAllFromTo* *x* *y* ⊕ *DenyAllFromTo* *y* *x* ⊕ *AllowPortFromTo* *x* *y* *dn*)) =
dom (*C* (*DenyAllFromTo* *x* *y* ⊕ *DenyAllFromTo* *y* *x*))
⟨*proof*⟩

lemma *domSubset5*:
dom (*C* (*DenyAllFromTo* *x* *y* ⊕ *DenyAllFromTo* *y* *x* ⊕ *AllowPortFromTo* *y* *x* *dn*)) =
dom (*C* (*DenyAllFromTo* *x* *y* ⊕ *DenyAllFromTo* *y* *x*))
⟨*proof*⟩

lemma *domSubset1*:

dom (*C* (*DenyAllFromTo one two* ⊕ *DenyAllFromTo two one* ⊕ *AllowPortFromTo one two dn* ⊕ *x*)) =
dom (*C* (*DenyAllFromTo one two* ⊕ *DenyAllFromTo two one* ⊕ *x*))
⟨*proof*⟩

lemma *domSubset2*:
dom (*C* (*DenyAllFromTo one two* ⊕ *DenyAllFromTo two one* ⊕ *AllowPortFromTo two one dn* ⊕ *x*)) =
dom (*C* (*DenyAllFromTo one two* ⊕ *DenyAllFromTo two one* ⊕ *x*))
⟨*proof*⟩

lemma *ConcAssoc2*: *C* (*X* ⊕ *Y* ⊕ ((*A* ⊕ *B*) ⊕ *D*)) = *C* (*X* ⊕ *Y* ⊕ *A* ⊕ *B* ⊕ *D*)
⟨*proof*⟩

lemma *ConcAssoc3*: *C* (*X* ⊕ ((*Y* ⊕ *A*) ⊕ *D*)) = *C* (*X* ⊕ *Y* ⊕ *A* ⊕ *D*)
⟨*proof*⟩

lemma *RS3-NMT*[*rule-format*]:
DenyAll ∈ *set p* ⟶ *rm-MT-rules C p* ≠ []
⟨*proof*⟩

lemma *norm-notMT*: *DenyAll* ∈ *set* (*policy2list p*) ⟹ *normalize p* ≠ []
⟨*proof*⟩

lemma *norm-notMTQ*: *DenyAll* ∈ *set* (*policy2list p*) ⟹ *normalizeQ p* ≠ []
⟨*proof*⟩

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* ∈ *dom* (*C a*) ⟶
C (*list2policyR* (*a* # *list*)) *p* = *C a p*
⟨*proof*⟩

lemma *list2policyR-End*: *p* ∉ *dom* (*C a*) ⟹
C (*list2policyR* (*a* # *list*)) *p* = (*C a* ⨁ *list2policy* (*map C list*)) *p*
⟨*proof*⟩

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

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

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

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

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

lemma *p-eq*[*rule-format*]:
$p \neq [] \longrightarrow \textit{list2policy}\ (\textit{map}\ C\ (\textit{rev}\ p)) = C\ (\textit{list2FWpolicy}\ p)$
⟨*proof*⟩

lemma *p-eq2*[*rule-format*]:
$\textit{normalize}\ x \neq [] \longrightarrow C(\textit{list2FWpolicy}(\textit{normalize}\ x)) = C\ x \longrightarrow$
$\textit{list2policy}(\textit{map}\ C\ (\textit{rev}(\textit{normalize}\ x))) = C\ x$
⟨*proof*⟩

lemma *p-eq2Q*[*rule-format*]:
$\textit{normalizeQ}\ x \neq [] \longrightarrow C\ (\textit{list2FWpolicy}\ (\textit{normalizeQ}\ x)) = C\ x \longrightarrow$
$\textit{list2policy}\ (\textit{map}\ C\ (\textit{rev}\ (\textit{normalizeQ}\ x))) = C\ x$
⟨*proof*⟩

lemma *list2listNMT*[*rule-format*]: $x \neq [] \longrightarrow \textit{map sem}\ x \neq []$
⟨*proof*⟩

lemma *Norm-Distr2*:
$r\ \textit{o-f}\ ((P \bigotimes_2 (\textit{list2policy}\ Q))\ o\ d) = (\textit{list2policy}\ ((P \bigotimes_L Q)\ (\bigotimes_2)\ r\ d))$
⟨*proof*⟩

lemma *NATDistr*:
$N \neq [] \Longrightarrow F = C\ (\textit{list2policyR}\ N) \Longrightarrow$
$(\lambda(x, y).\ x)\ o_f\ (\textit{NAT} \bigotimes_2 F \circ (\lambda x.\ (x, x))) =$

list2policy ((*NAT* $\bigotimes_L$ *map C N*) ($\bigotimes_2$) (λ(*x*, *y*). *x*) (λ*x*. (*x*, *x*)))
⟨*proof*⟩

lemma *C-eq-normalize-manual*:
DenyAll∈*set*(*policy2list p*) ⟹ *allNetsDistinct*(*policy2list p*) ⟹ *all-in-list*(*policy2list p*) *l* ⟹
C (*list2FWpolicy* (*normalize-manual-order p l*)) = *C p*
⟨*proof*⟩

lemma *p-eq2-manualQ*[*rule-format*]:
normalize-manual-orderQ x l ≠ [] ⟶ *C*(*list2FWpolicy* (*normalize-manual-orderQ x l*)) = *C x* ⟶
list2policy (*map C* (*rev* (*normalize-manual-orderQ x l*))) = *C x*
⟨*proof*⟩

lemma *norm-notMT-manualQ*: *DenyAll* ∈ *set* (*policy2list p*) ⟹ *normalize-manual-orderQ p l* ≠ []
⟨*proof*⟩

lemma *C-eq-normalize-manualQ*:
DenyAll∈*set*(*policy2list p*) ⟹ *allNetsDistinct*(*policy2list p*) ⟹ *all-in-list*(*policy2list p*) *l* ⟹
C (*list2FWpolicy* (*normalize-manual-orderQ p l*)) = *C p*
⟨*proof*⟩

lemma *p-eq2-manual*[*rule-format*]:
normalize-manual-order x l ≠ [] ⟶ *C* (*list2FWpolicy* (*normalize-manual-order x l*)) = *C x* ⟶
list2policy (*map C* (*rev* (*normalize-manual-order x l*))) = *C x*
⟨*proof*⟩

lemma *norm-notMT-manual*: *DenyAll* ∈ *set* (*policy2list p*) ⟹ *normalize-manual-order p l* ≠ []
⟨*proof*⟩

As an example, how this theorems can be used for a concrete normalisation instantiation.

lemma *normalizeNAT*:
DenyAll ∈ *set* (*policy2list Filter*) ⟹ *allNetsDistinct* (*policy2list Filter*) ⟹
all-in-list (*policy2list Filter*) (*Nets-List Filter*) ⟹
(λ(*x*, *y*). *x*) o_f (*NAT* $\bigotimes_2$ *C Filter* ∘ (λ*x*. (*x*, *x*))) =
list2policy ((*NAT* $\bigotimes_L$ *map C* (*rev* (*FWNormalisationCore.normalize Filter*))) ($\bigotimes_2$)

(λ(*x*, *y*). *x*) (λ*x*. (*x*, *x*)))

⟨*proof*⟩

lemma *domSimpl*[*simp*]: $dom\ (C\ (A \oplus DenyAll)) = dom\ (C\ (DenyAll))$
⟨*proof*⟩

The followin theorems can be applied when prepending the usual normalisation with an additional step and using another semantical interpretation function. This is a general recipe which can be applied whenever one nees to combine several normalisation strategies.

lemma *CRotate-eq-rotateC*: $CRotate\ p = C\ (rotatePolicy\ p)$
⟨*proof*⟩

lemma *DAinRotate*:
$DenyAll \in set\ (policy2list\ p) \Longrightarrow DenyAll \in set\ (policy2list\ (rotatePolicy\ p))$
⟨*proof*⟩

lemma *DAUniv*: $dom\ (CRotate\ (P \oplus DenyAll)) = UNIV$
⟨*proof*⟩

lemma *p-eq2R*[*rule-format*]:
$normalize\ (rotatePolicy\ x) \neq [] \longrightarrow C(list2FWpolicy(normalize\ (rotatePolicy\ x))) = CRotate\ x \longrightarrow$
$list2policy\ (map\ C\ (rev\ (normalize\ (rotatePolicy\ x)))) = CRotate\ x$
⟨*proof*⟩

lemma *C-eq-normalizeRotate*:
$DenyAll \in set\ (policy2list\ p) \Longrightarrow allNetsDistinct\ (policy2list\ (rotatePolicy\ p)) \Longrightarrow$
$all\text{-}in\text{-}list\ (policy2list\ (rotatePolicy\ p))\ (Nets\text{-}List\ (rotatePolicy\ p)) \Longrightarrow$
$C\ (list2FWpolicy$
$(removeAllDuplicates$
$(insertDenies$
$(separate$
$(sort(removeShadowRules2(remdups(rm\text{-}MT\text{-}rules\ C$
$(insertDeny(removeShadowRules1(policy2list(rotatePolicy\ p)))))))$
$(Nets\text{-}List\ (rotatePolicy\ p))))))) =$
$CRotate\ p$
⟨*proof*⟩

lemma *C-eq-normalizeRotate2*:
$DenyAll \in set\ (policy2list\ p) \Longrightarrow$
$allNetsDistinct\ (policy2list\ (rotatePolicy\ p)) \Longrightarrow$
$all\text{-}in\text{-}list\ (policy2list\ (rotatePolicy\ p))\ (Nets\text{-}List\ (rotatePolicy\ p)) \Longrightarrow$
$C\ (list2FWpolicy\ (FWNormalisationCore.normalize\ (rotatePolicy\ p))) = CRotate\ p$
⟨*proof*⟩

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 $\Longrightarrow$ *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*: $x \in dom(Cp\ b) \Longrightarrow Cp\ (a \oplus b)\ x = (Cp\ b)\ x$
⟨*proof*⟩

lemma *wp2Conc*[*rule-format*]: *wellformed-policy2Pr* (*x*#*xs*) $\Longrightarrow$ *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* $\Longrightarrow$ *applied-rule-rev Cp x p* $\neq$ *None*
⟨*proof*⟩

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

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

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

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

lemma *CConcStartaux*: $Cp\ a\ x = None \Longrightarrow (Cp\ aa\ {+}{+}\ Cp\ a)\ x = Cp\ aa\ x$
⟨*proof*⟩

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

lemma *mrNnt*[*simp*]: *applied-rule-rev Cp x p* $= Some\ a \Longrightarrow p \neq []$
⟨*proof*⟩

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

lemma *CConcStart2*:
$p \neq [] \Longrightarrow x \notin dom\ (Cp\ a) \Longrightarrow Cp(list2FWpolicy\ (p@[a]))\ x = Cp\ (list2FWpolicy\ p)x$
⟨*proof*⟩

lemma *CConcEnd1*:
$q@p \neq [] \Longrightarrow x \notin dom\ (Cp\ a) \Longrightarrow Cp(list2FWpolicy(q@p@[a]))\ x = Cp\ (list2FWpolicy\ (q@p))x$
⟨*proof*⟩

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

lemma *bar3*:
$x \in dom\ (Cp\ (list2FWpolicy\ (xs\ @\ [xa]))) \Longrightarrow x \in dom\ (Cp\ (list2FWpolicy\ xs)) \vee x \in dom\ (Cp\ xa)$
⟨*proof*⟩

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

$a \neq [] \longrightarrow x \in dom\ (Cp(list2FWpolicy\ a)) \longrightarrow Cp(list2FWpolicy(b@a))\ x = (Cp(list2FWpolicy\ a))\ x$
⟨proof⟩

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

lemma *domConc*:
$x \in dom\ (Cp\ (list2FWpolicy\ b)) \Longrightarrow b \neq [] \Longrightarrow x \in dom\ (Cp\ (list2FWpolicy\ (a@b)))$
⟨proof⟩

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

lemma *C-eq-if-mr-eq2*:
$applied\text{-}rule\text{-}rev\ Cp\ x\ a = Some\ r \Longrightarrow applied\text{-}rule\text{-}rev\ Cp\ x\ b = Some\ r \Longrightarrow a{\neq}[] \Longrightarrow b{\neq}[] \Longrightarrow$
$(Cp\ (list2FWpolicy\ a))\ x = (Cp\ (list2FWpolicy\ b))\ x$
⟨proof⟩

lemma *nMRtoNone*[*rule-format*]:
$p \neq [] \longrightarrow applied\text{-}rule\text{-}rev\ Cp\ x\ p = None \longrightarrow Cp\ (list2FWpolicy\ p)\ x = None$
⟨proof⟩

lemma *C-eq-if-mr-eq*:
$applied\text{-}rule\text{-}rev\ Cp\ x\ b = applied\text{-}rule\text{-}rev\ Cp\ x\ a \Longrightarrow a \neq [] \Longrightarrow b \neq [] \Longrightarrow$
$(Cp\ (list2FWpolicy\ a))\ x = (Cp\ (list2FWpolicy\ b))\ x$
⟨proof⟩

lemma *notmatching-notdom*:
$applied\text{-}rule\text{-}rev\ Cp\ x\ (p@[a]) \neq Some\ a \Longrightarrow x \notin dom\ (Cp\ a)$
⟨proof⟩

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

lemma *foo3D*:
$wellformed\text{-}policy1\ p \Longrightarrow p{=}DenyAll\#ps \Longrightarrow applied\text{-}rule\text{-}rev\ Cp\ x\ p = Some\ DenyAll \Longrightarrow r{\in}set\ ps \Longrightarrow$

$x \notin dom\ (Cp\ r)$
⟨*proof*⟩

lemma *foo4*[*rule-format*]:
$set\ p = set\ s \wedge (\forall\ r.\ r \in set\ p \longrightarrow x \notin dom\ (Cp\ r)) \longrightarrow (\forall\ r\ .r \in set\ s \longrightarrow x \notin dom\ (Cp\ r))$
⟨*proof*⟩

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

lemma *mr-first*:
$x \in dom\ (Cp\ b) \Longrightarrow (\forall\ r.\ r \in set\ c \longrightarrow x \notin dom\ (Cp\ r)) \Longrightarrow s = b\#c \Longrightarrow$
$applied\text{-}rule\text{-}rev\ Cp\ x\ s = Some\ b$
⟨*proof*⟩

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

lemma *foo8*:
$\forall\ r.\ r \in set\ p \wedge x \in dom\ (Cp\ r) \longrightarrow r = a \Longrightarrow set\ p = set\ s \Longrightarrow$
$\forall\ r.\ r \in set\ s \wedge x \in dom\ (Cp\ r) \longrightarrow r = a$
⟨*proof*⟩

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

lemma *wp3tl*[*rule-format*]: $wellformed\text{-}policy3Pr\ p \longrightarrow wellformed\text{-}policy3Pr\ (tl\ p)$
⟨*proof*⟩

lemma *wp3Conc*[*rule-format*]: $wellformed\text{-}policy3Pr\ (a\#p) \longrightarrow wellformed\text{-}policy3Pr\ p$
⟨*proof*⟩

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

set p
⟨*proof*⟩

lemma *mrMTNone*[*simp*]: *applied-rule-rev Cp x* [] = *None*
⟨*proof*⟩

lemma *DAAux*[*simp*]: $x \in$ *dom* (*Cp DenyAll*)
⟨*proof*⟩

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

lemma *mr-not-Conc*: *singleCombinators p* $\Longrightarrow$ *applied-rule-rev Cp x p* $\neq$ *Some* ($a \oplus b$)
⟨*proof*⟩

lemma *foo25*[*rule-format*]: *wellformed-policy3Pr* (*p*@[*x*]) $\longrightarrow$ *wellformed-policy3Pr p*
⟨*proof*⟩

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

lemma *wp3EndMT*[*rule-format*]:
wellformed-policy3Pr (*p*@[*xs*]) $\longrightarrow$ *AllowPortFromTo a b po* $\in$ *set p* $\longrightarrow$
dom (*Cp* (*AllowPortFromTo a b po*)) $\cap$ *dom* (*Cp xs*) = {}
⟨*proof*⟩

lemma *foo29*: *dom* (*Cp a*) $\neq$ {} $\Longrightarrow$ *dom* (*Cp a*) $\cap$ *dom* (*Cp b*) = {} $\Longrightarrow$ $a \neq b$
⟨*proof*⟩

lemma *foo28*:
AllowPortFromTo a b po$\in$*set p* $\Longrightarrow$ *dom*(*Cp*(*AllowPortFromTo a b po*))$\neq${} $\Longrightarrow$
(*wellformed-policy3Pr*(*p*@[*x*])) $\Longrightarrow$
$x \neq$ *AllowPortFromTo a b po*
⟨*proof*⟩

lemma *foo28a*[*rule-format*]: $x \in$ *dom* (*Cp a*) $\Longrightarrow$ *dom* (*Cp a*) $\neq$ {}
⟨*proof*⟩

lemma *allow-deny-dom*[*simp*]:
dom (*Cp* (*AllowPortFromTo a b po*)) $\subseteq$ *dom* (*Cp* (*DenyAllFromTo a b*))
⟨*proof*⟩

lemma *DenyAllowDisj*:
$dom\ (Cp\ (AllowPortFromTo\ a\ b\ p)) \neq \{\} \Longrightarrow$
$dom\ (Cp\ (DenyAllFromTo\ a\ b)) \cap dom\ (Cp\ (AllowPortFromTo\ a\ b\ p))\ \neq \{\}$
⟨*proof*⟩

lemma *foo31*:
$\forall\ r.\ r \in set\ p \wedge x \in dom\ (Cp\ r) \longrightarrow$
$(r = AllowPortFromTo\ a\ b\ po \vee r = DenyAllFromTo\ a\ b \vee r = DenyAll) \Longrightarrow$
$set\ p = set\ s \Longrightarrow$
$(\forall\ r.\ r{\in}set\ s \wedge x{\in}dom(Cp\ r) \longrightarrow r{=}AllowPortFromTo\ a\ b\ po \vee r{=}DenyAllFromTo\ a\ b \vee r = DenyAll)$
⟨*proof*⟩

lemma *wp1-auxa*: $wellformed\text{-}policy1\text{-}strong\ p {\Longrightarrow} (\exists\ r.\ applied\text{-}rule\text{-}rev\ Cp\ x\ p = Some\ r)$
⟨*proof*⟩

lemma *deny-dom*[*simp*]:
$twoNetsDistinct\ a\ b\ c\ d \Longrightarrow dom\ (Cp\ (DenyAllFromTo\ a\ b)) \cap dom\ (Cp\ (DenyAllFromTo\ c\ d)) = \{\}$
⟨*proof*⟩

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

lemma *DomInterAllowsMT*:
$twoNetsDistinct\ a\ b\ c\ d \Longrightarrow dom\ (Cp(AllowPortFromTo\ a\ b\ p)) \cap dom(Cp(AllowPortFromTo\ c\ d\ po)){=}\{\}$
⟨*proof*⟩

lemma *DomInterAllowsMT-Ports*:
$p \neq po \Longrightarrow dom\ (Cp\ (AllowPortFromTo\ a\ b\ p)) \cap dom\ (Cp\ (AllowPortFromTo\ c\ d\ po)) = \{\}$
⟨*proof*⟩

lemma *wellformed-policy3-charn*[*rule-format*]:
$singleCombinators\ p \longrightarrow distinct\ p \longrightarrow allNetsDistinct\ p \longrightarrow$
$wellformed\text{-}policy1\ p \longrightarrow wellformed\text{-}policy2Pr\ p\ \longrightarrow wellformed\text{-}policy3Pr\ p$
⟨*proof*⟩

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

dom (*Cp* (*DenyAllFromTo b c*)) ∩ *dom* (*Cp* (*AllowPortFromTo a d po*)) ≠ {}⟹
b = *a* ∧ *c* = *d*
⟨*proof*⟩

lemma *DistinctNetsAllowAllow*:
AllowPortFromTo b c poo ∈ *set p* ⟹ *AllowPortFromTo a d po* ∈ *set p* ⟹
allNetsDistinct p ⟹ *dom*(*Cp*(*AllowPortFromTo b c poo*)) ∩ *dom*(*Cp*(*AllowPortFromTo a d po*)) ≠ {} ⟹
b = *a* ∧ *c* = *d* ∧ *poo* = *po*
⟨*proof*⟩

lemma *WP2RS2*[*simp*]:
singleCombinators p ⟹ *distinct p* ⟹ *allNetsDistinct p* ⟹
wellformed-policy2Pr (*removeShadowRules2 p*)
⟨*proof*⟩

lemma *AD-aux*:
AllowPortFromTo a b po ∈ *set p* ⟹ *DenyAllFromTo c d* ∈ *set p* ⟹
allNetsDistinct p ⟹ *singleCombinators p* ⟹ *a* ≠ *c* ∨ *b* ≠ *d* ⟹
dom (*Cp* (*AllowPortFromTo a b po*)) ∩ *dom* (*Cp* (*DenyAllFromTo c d*)) = {}
⟨*proof*⟩

lemma *sorted-WP2*[*rule-format*]:
sorted p l ⟶ *all-in-list p l* ⟶ *distinct p* ⟶ *allNetsDistinct p* ⟶ *singleCombinators p* ⟶
wellformed-policy2Pr p
⟨*proof*⟩

lemma *wellformed2-sorted*[*simp*]:
all-in-list p l ⟹ *distinct p* ⟹ *allNetsDistinct p* ⟹ *singleCombinators p* ⟹
wellformed-policy2Pr (*sort p l*)
⟨*proof*⟩

lemma *wellformed2-sortedQ*[*simp*]:
all-in-list p l ⟹ *distinct p* ⟹ *allNetsDistinct p* ⟹ *singleCombinators p* ⟹
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 ≠ [] ⟹ *Cp*(*list2FWpolicy* (*removeShadowRules1 p*)) = *Cp*(*list2FWpolicy p*)
⟨*proof*⟩

lemma *EX-MR-aux*[*rule-format*]:
applied-rule-rev Cp x (*DenyAll* # *p*) ≠ *Some DenyAll* ⟶ (∃ *y*. *applied-rule-rev Cp x p* = *Some y*)
⟨*proof*⟩

lemma *EX-MR* :
applied-rule-rev Cp x p ≠ (*Some DenyAll*) ⟹ *p* = *DenyAll*#*ps* ⟹
(*applied-rule-rev Cp x p* = *applied-rule-rev Cp x ps*)
⟨*proof*⟩

lemma *mr-not-DA*:
wellformed-policy1-strong s ⟹ *applied-rule-rev Cp x p* = *Some* (*DenyAllFromTo a ab*) ⟹
set p = *set s* ⟹ *applied-rule-rev Cp x s* ≠ *Some DenyAll*
⟨*proof*⟩

lemma *domsMT-notND-DD*:
dom (*Cp* (*DenyAllFromTo a b*)) ∩ *dom* (*Cp* (*DenyAllFromTo c d*)) ≠ {} ⟹ ¬ *netsDistinct a c*
⟨*proof*⟩

lemma *domsMT-notND-DD2*:
dom (*Cp* (*DenyAllFromTo a b*)) ∩ *dom* (*Cp* (*DenyAllFromTo c d*)) ≠ {} ⟹ ¬ *netsDistinct b d*
⟨*proof*⟩

lemma *domsMT-notND-DD3*:
x ∈ *dom* (*Cp* (*DenyAllFromTo a b*)) ⟹ *x* ∈ *dom* (*Cp* (*DenyAllFromTo c d*)) ⟹ ¬ *netsDistinct a c*
⟨*proof*⟩

lemma *domsMT-notND-DD4*:
x ∈ *dom* (*Cp* (*DenyAllFromTo a b*)) ⟹ *x* ∈ *dom* (*Cp* (*DenyAllFromTo c d*)) ⟹ ¬ *netsDistinct b d*
⟨*proof*⟩

lemma *NetsEq-if-sameP-DD*:
allNetsDistinct p ⟹ *u*∈ *set p* ⟹ *v*∈ *set p* ⟹ *u* = (*DenyAllFromTo a b*) ⟹
v = (*DenyAllFromTo c d*) ⟹ *x* ∈ *dom* (*Cp* (*u*)) ⟹ *x* ∈ *dom* (*Cp* (*v*)) ⟹

$a = c \wedge b = d$
⟨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* : $r \in$ *set p*
and *inDom* : $x \in$ *dom (Cp r)*
shows (r = *AllowPortFromTo a b po* $\vee$ r = *DenyAllFromTo a b* $\vee$ r = *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* $\Longrightarrow$ *none-MT-rules Cp (sort p l)*
⟨proof⟩

lemma *nMTSortQ*: *none-MT-rules Cp p* $\Longrightarrow$ *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*: *dom (Cp (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*: $x \in$ *dom (Cp (AllowPortFromTo c d po))*
shows *applied-rule-rev Cp x p = Some (AllowPortFromTo c d po)*
⟨proof⟩

lemma *rule-charn3*:
wellformed-policy1 p $\Longrightarrow$ *allNetsDistinct p* $\Longrightarrow$ *singleCombinators p* $\Longrightarrow$
wellformed-policy3Pr p $\Longrightarrow$ *applied-rule-rev Cp x p* = *Some* (*DenyAllFromTo c d*) $\Longrightarrow$

AllowPortFromTo a b po $\in$ *set p* $\Longrightarrow$ *x* $\notin$ *dom* (*Cp* (*AllowPortFromTo a b po*))
⟨*proof*⟩

lemma *rule-charn4*:
assumes *wp1*: *wellformed-policy1 p*
and *aND*: *allNetsDistinct p*
and *SC*: *singleCombinators p*
and *wp3*: *wellformed-policy3Pr p*
and *DA*: *DenyAll* $\notin$ *set p*
and *mr*: *applied-rule-rev Cp x p* = *Some* (*DenyAllFromTo a b*)
and *rinp*: *r* $\in$ *set p*
and *xindom*: *x* $\in$ *dom* (*Cp r*)
shows *r* = *DenyAllFromTo a b*
⟨*proof*⟩

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*)) $\Longrightarrow$
set p = *set s* $\Longrightarrow$ *r* $\in$ *set s* $\Longrightarrow$ *x* $\in$ *dom* (*Cp r*) $\Longrightarrow$
(*r* = *AllowPortFromTo a b po* $\vee$ *r* = *DenyAllFromTo a b* $\vee$ *r* = *DenyAll*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *mrDA-tl*:
assumes *mr-DA*: *applied-rule-rev Cp x p* = *Some DenyAll*
and *wp1n*: *wellformed-policy1-strong p*
shows *applied-rule-rev Cp x* (*tl p*) = *None*
⟨*proof*⟩

lemma *rule-charnDAFT*:
wellformed-policy1-strong p $\Longrightarrow$ *allNetsDistinct p* $\Longrightarrow$ *singleCombinators p* $\Longrightarrow$
wellformed-policy3Pr p $\Longrightarrow$ *applied-rule-rev Cp x p* = *Some* (*DenyAllFromTo a b*) $\Longrightarrow$
r $\in$ *set* (*tl p*) $\Longrightarrow$ *x* $\in$ *dom* (*Cp r*) $\Longrightarrow$
r = *DenyAllFromTo a b*
⟨*proof*⟩

lemma *mrDenyAll-is-unique*:
wellformed-policy1-strong p $\Longrightarrow$ *applied-rule-rev Cp x p* = *Some DenyAll* $\Longrightarrow$ *r* $\in$ *set* (*tl p*) $\Longrightarrow$
x $\notin$ *dom* (*Cp r*)
⟨*proof*⟩

theorem *C-eq-Sets-mr*:
assumes *sets-eq*: *set p* = *set s*
and *SC*: *singleCombinators p*
and *wp1-p*: *wellformed-policy1-strong p*
and *wp1-s*: *wellformed-policy1-strong s*
and *wp3-p*: *wellformed-policy3Pr p*
and *wp3-s*: *wellformed-policy3Pr s*
and *aND*: *allNetsDistinct p*
shows *applied-rule-rev Cp x p* = *applied-rule-rev Cp x s*
⟨*proof*⟩

lemma *C-eq-Sets*:
singleCombinators p $\Longrightarrow$ *wellformed-policy1-strong p* $\Longrightarrow$ *wellformed-policy1-strong s* $\Longrightarrow$
wellformed-policy3Pr p $\Longrightarrow$ *wellformed-policy3Pr s* $\Longrightarrow$ *allNetsDistinct p* $\Longrightarrow$ *set p* = *set s* $\Longrightarrow$
Cp (*list2FWpolicy p*) *x* = *Cp* (*list2FWpolicy s*) *x*
⟨*proof*⟩

lemma *C-eq-sorted*:
distinct p $\Longrightarrow$ *all-in-list p l* $\Longrightarrow$ *singleCombinators p* $\Longrightarrow$
wellformed-policy1-strong p$\Longrightarrow$ *wellformed-policy3Pr p*$\Longrightarrow$ *allNetsDistinct p* $\Longrightarrow$
Cp (*list2FWpolicy* (*sort p l*))= *Cp* (*list2FWpolicy p*)
⟨*proof*⟩

lemma *C-eq-sortedQ*:
distinct p $\Longrightarrow$ *all-in-list p l* $\Longrightarrow$ *singleCombinators p* $\Longrightarrow$
wellformed-policy1-strong p $\Longrightarrow$ *wellformed-policy3Pr p* $\Longrightarrow$ *allNetsDistinct p* $\Longrightarrow$
Cp (*list2FWpolicy* (*qsort p l*))= *Cp* (*list2FWpolicy p*)
⟨*proof*⟩

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

lemma *C-eq-None*[*rule-format*]:
p $\neq$ [] $\longrightarrow$ *applied-rule-rev Cp x p* = *None* $\longrightarrow$ *Cp* (*list2FWpolicy p*) *x* = *None*

⟨*proof*⟩

lemma *C-eq-None2*:
a ≠ [] ⟹ *b* ≠ [] ⟹ *applied-rule-rev Cp x a* = *None* ⟹ *applied-rule-rev Cp x b* = *None* ⟹
(*Cp* (*list2FWpolicy a*)) *x* = (*Cp* (*list2FWpolicy b*)) *x*
⟨*proof*⟩

lemma *C-eq-RS2*:
wellformed-policy1-strong p ⟹
Cp (*list2FWpolicy* (*removeShadowRules2 p*))= *Cp* (*list2FWpolicy p*)
⟨*proof*⟩

lemma *none-MT-rulesRS2*: *none-MT-rules Cp p* ⟹ *none-MT-rules Cp* (*removeShadowRules2 p*)
⟨*proof*⟩

lemma *CconcNone*:
dom (*Cp a*) = {} ⟹ *p* ≠ [] ⟹ *Cp* (*list2FWpolicy* (*a* # *p*)) *x* = *Cp* (*list2FWpolicy p*) *x*
⟨*proof*⟩

lemma *none-MT-rulesrd*[*rule-format*]: *none-MT-rules Cp p* ⟶ *none-MT-rules Cp* (*remdups p*)
⟨*proof*⟩

lemma *DARS3*[*rule-format*]:*DenyAll* ∉ *set p*⟶*DenyAll* ∉ *set* (*rm-MT-rules Cp p*)
⟨*proof*⟩

lemma *DAnMT*: *dom* (*Cp DenyAll*) ≠ {}
⟨*proof*⟩

lemma *DAnMT2*: *Cp DenyAll* ≠ *Map.empty*
⟨*proof*⟩

lemma *wp1n-RS3*[*rule-format*,*simp*]:
wellformed-policy1-strong p ⟶ *wellformed-policy1-strong* (*rm-MT-rules Cp p*)
⟨*proof*⟩

lemma *AILRS3*[*rule-format*,*simp*]:
all-in-list p l ⟶ *all-in-list* (*rm-MT-rules Cp p*) *l*
⟨*proof*⟩

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

singleCombinators p ⟶ *singleCombinators*(*rm-MT-rules Cp p*)
⟨*proof*⟩

lemma *RS3subset*: *set* (*rm-MT-rules Cp p*) ⊆ *set p*
⟨*proof*⟩

lemma *ANDRS3*[*simp*]:
singleCombinators p ⟹ *allNetsDistinct p* ⟹ *allNetsDistinct* (*rm-MT-rules Cp p*)
⟨*proof*⟩

lemma *nlpaux*: *x* ∉ *dom* (*Cp b*) ⟹ *Cp* (*a* ⊕ *b*) *x* = *Cp a x*
⟨*proof*⟩

lemma *notindom*[*rule-format*]:
a ∈ *set p* ⟶ *x* ∉ *dom* (*Cp* (*list2FWpolicy p*)) ⟶ *x* ∉ *dom* (*Cp a*)
⟨*proof*⟩

lemma *C-eq-rd*[*rule-format*]:
p ≠ [] ⟹ *Cp* (*list2FWpolicy* (*remdups p*)) = *Cp* (*list2FWpolicy p*)
⟨*proof*⟩

lemma *nMT-domMT*:
¬ *not-MT Cp p* ⟹ *p* ≠ [] ⟹ *r* ∉ *dom* (*Cp* (*list2FWpolicy p*))
⟨*proof*⟩

lemma *C-eq-RS3-aux*[*rule-format*]:
not-MT Cp p ⟹ *Cp* (*list2FWpolicy p*) *x* = *Cp* (*list2FWpolicy* (*rm-MT-rules Cp p*)) *x*
⟨*proof*⟩

lemma *C-eq-id*:
wellformed-policy1-strong p ⟹ *Cp*(*list2FWpolicy* (*insertDeny p*)) = *Cp* (*list2FWpolicy p*)
⟨*proof*⟩

lemma *C-eq-RS3*:
not-MT Cp p ⟹ *Cp*(*list2FWpolicy* (*rm-MT-rules Cp p*)) = *Cp* (*list2FWpolicy p*)
⟨*proof*⟩

lemma *NMPrd*[*rule-format*]: *not-MT Cp p* ⟶ *not-MT Cp* (*remdups p*)
⟨*proof*⟩

lemma *NMPDA*[*rule-format*]: *DenyAll* ∈ *set p* ⟶ *not-MT Cp p*
⟨*proof*⟩

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∈*set*(*policy2list p*) ⟹ *all-in-list*(*policy2list p*) *l* ⟹ *allNetsDistinct*(*policy2list p*) ⟹
Cp(*list2FWpolicy* (*sort* (*removeShadowRules2* (*remdups* (*rm-MT-rules Cp* (*insertDeny*
(*removeShadowRules1* (*policy2list p*)))))) *l*)) =
Cp p
⟨*proof*⟩

lemma *C-eq-All-untilSortedQ*:
DenyAll∈ *set*(*policy2list p*) ⟹ *all-in-list*(*policy2list p*) *l* ⟹ *allNetsDistinct*(*policy2list p*) ⟹
Cp(*list2FWpolicy* (*qsort* (*removeShadowRules2* (*remdups* (*rm-MT-rules Cp* (*insertDeny*
(*removeShadowRules1* (*policy2list p*)))))) *l*)) =
Cp p
⟨*proof*⟩

lemma *C-eq-All-untilSorted-withSimps*:
DenyAll ∈ *set* (*policy2list p*) ⟹ *all-in-list* (*policy2list p*) *l* ⟹
allNetsDistinct (*policy2list p*) ⟹
Cp(*list2FWpolicy*(*sort*(*removeShadowRules2*(*remdups*(*rm-MT-rules Cp* (*insertDeny*
(*removeShadowRules1*(*policy2list p*)))))) *l*)) =
Cp p
⟨*proof*⟩

lemma *C-eq-All-untilSorted-withSimpsQ*:
DenyAll ∈ *set* (*policy2list p*) ⟹ *all-in-list* (*policy2list p*) *l* ⟹

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

lemma *InDomConc*[*rule-format*]: $p \neq [] \longrightarrow x \in dom\ (Cp\ (list2FWpolicy\ (p))) \longrightarrow x \in dom\ (Cp\ (list2FWpolicy\ (a\#p)))$
⟨*proof*⟩

lemma *not-in-member*[*rule-format*]: $member\ a\ b \longrightarrow x \notin dom\ (Cp\ b) \longrightarrow x \notin dom\ (Cp\ a)$
⟨*proof*⟩

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

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

lemma *sdnets-in-subnets*[*rule-format*]:
$p \in dom\ (Cp\ x) \longrightarrow \neg\ member\ DenyAll\ x \longrightarrow$
$(\exists\ (a,b) \in sdnets\ x.\ a \in subnetsOfAdr\ (src\ p) \wedge b \in subnetsOfAdr\ (dest\ p))$
⟨*proof*⟩

lemma *disjSD-no-p-in-both*[*rule-format*]:
$[\![disjSD\text{-}2\ x\ y;\ \neg\ member\ DenyAll\ x;\ \neg\ member\ DenyAll\ y;$
$p \in dom\ (Cp\ x);\ p \in dom\ (Cp\ y)]\!] \Longrightarrow False$
⟨*proof*⟩

lemma *list2FWpolicy-eq*:
$zs \neq [] \Longrightarrow Cp\ (list2FWpolicy\ (x \oplus y\ \#\ z))\ p = Cp\ (x \oplus list2FWpolicy\ (y\ \#\ z))\ p$
⟨*proof*⟩

lemma *dom-sep*[*rule-format*]:
$x \in dom\ (Cp\ (list2FWpolicy\ p)) \longrightarrow x \in dom\ (Cp\ (list2FWpolicy(separate\ p)))$
⟨*proof*⟩

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 ∈ *dom* (*Cp* (*list2FWpolicy* (*a* ⊕ *y* # *z*))) ⟹ *x* ∈ *dom* (*Cp* (*a* ⊕ *list2FWpolicy* (*y* # *z*)))
⟨*proof*⟩

lemma *sep-dom2-aux2*:
(*x* ∈ *dom* (*Cp* (*list2FWpolicy* (*separate* (*y* # *z*)))) ⟶ *x* ∈ *dom* (*Cp* (*list2FWpolicy* (*y* # *z*)))) ⟹
x ∈ *dom* (*Cp* (*list2FWpolicy* (*a* # *separate* (*y* # *z*)))) ⟹
x ∈ *dom* (*Cp* (*list2FWpolicy* (*a* ⊕ *y* # *z*)))
⟨*proof*⟩

lemma *sep-dom2*[*rule-format*]:
x ∈ *dom* (*Cp* (*list2FWpolicy* (*separate p*))) ⟶ *x* ∈ *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 ≠ [] ⟶ *Cp* (*list2FWpolicy* (*separate p*)) *a* = *Cp* (*list2FWpolicy p*) *a*
⟨*proof*⟩

lemma *C-eq-s*: *p* ≠ [] ⟹ *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∈*set*(*policy2list p*) ⟹ *all-in-list*(*policy2list p*) *l* ⟹ *allNetsDistinct*(*policy2list p*) ⟹
Cp (*list2FWpolicy* (*separate* (*sort* (*removeShadowRules2* (*remdups* (*rm-MT-rules Cp* (*insertDeny* (*removeShadowRules1* (*policy2list p*)))))) *l*))) =
Cp p
⟨*proof*⟩

lemma *C-eq-until-separatedQ*:
DenyAll ∈ *set* (*policy2list p*) ⟹ *all-in-list* (*policy2list p*) *l* ⟹
allNetsDistinct (*policy2list p*) ⟹

$Cp(list2FWpolicy(separate(qsort($
$removeShadowRules2(remdups\ (rm\text{-}MT\text{-}rules\ Cp$
$(insertDeny\ (removeShadowRules1\ (policy2list\ p)))))) \ l))) =$
$Cp\ p$
$\langle proof\rangle$

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

lemma *DA-is-deny*:
$x \in dom\ (Cp\ (DenyAllFromTo\ a\ b \oplus DenyAllFromTo\ b\ a \oplus DenyAllFromTo\ a\ b)) \Longrightarrow$
$Cp\ (DenyAllFromTo\ a\ b \oplus DenyAllFromTo\ b\ a \oplus DenyAllFromTo\ a\ b)\ x = Some\ (deny\ ())$
$\langle proof\rangle$

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

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

lemma *inDomConc*:
$x \notin dom\ (Cp\ a) \Longrightarrow x \notin dom\ (Cp\ (list2FWpolicy\ p)) \Longrightarrow x \notin dom\ (Cp\ (list2FWpolicy(a\#p)))$
$\langle proof\rangle$

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

lemma *isSepaux*:
$p \neq [] \Longrightarrow noDenyAll\ (a\#p) \Longrightarrow separated\ (a\ \#\ p) \Longrightarrow$
$x \in dom\ (Cp\ (DenyAllFromTo\ (first\text{-}srcNet\ a)\ (first\text{-}destNet\ a) \oplus$
$DenyAllFromTo\ (first\text{-}destNet\ a)\ (first\text{-}srcNet\ a) \oplus a)) \Longrightarrow$

$x \notin$ *dom* (*Cp* (*list2FWpolicy p*))
⟨*proof*⟩

lemma *none-MT-rulessep*[*rule-format*]: *none-MT-rules Cp p* ⟶ *none-MT-rules Cp* (*separate p*)
⟨*proof*⟩

lemma *dom-id*:
noDenyAll (*a#p*) ⟹ *separated* (*a#p*) ⟹ $p \neq []$ ⟹
$x \notin$ *dom* (*Cp* (*list2FWpolicy p*)) ⟹ $x \in$ *dom* (*Cp* (*a*)) ⟹
$x \notin$ *dom* (*Cp* (*list2FWpolicy* (*insertDenies p*)))
⟨*proof*⟩

lemma *C-eq-iD-aux2*[*rule-format*]:
noDenyAll1 p ⟶ *separated p*⟶ $p \neq []$⟶ $x \in$ *dom* (*Cp* (*list2FWpolicy p*))⟶
Cp(*list2FWpolicy* (*insertDenies p*)) *x* = *Cp*(*list2FWpolicy p*) *x*
⟨*proof*⟩

lemma *C-eq-iD*:
separated p ⟹ *noDenyAll1 p* ⟹ *wellformed-policy1-strong p* ⟹
Cp(*list2FWpolicy* (*insertDenies p*)) = *Cp* (*list2FWpolicy p*)
⟨*proof*⟩

lemma *noDAsortQ*[*rule-format*]: *noDenyAll1 p* ⟶ *noDenyAll1* (*qsort p l*)
⟨*proof*⟩

lemma *NetsCollectedSortQ*:
distinct p ⟹*noDenyAll1 p* ⟹ *all-in-list p l* ⟹ *singleCombinators p* ⟹
NetsCollected (*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*)⟹
Cp (*list2FWpolicy*((*insertDenies*(*separate*(*sort*(*removeShadowRules2*
(*remdups*(*rm-MT-rules Cp* (*insertDeny* (*removeShadowRules1* (*policy2list*

p)))))) *l*))))) =
Cp p
⟨*proof*⟩

lemma *C-eq-Until-InsertDeniesQ*:
DenyAll ∈ *set* (*policy2list p*) ⟹ *all-in-list* (*policy2list p*) *l* ⟹
allNetsDistinct (*policy2list p*) ⟹
Cp (*list2FWpolicy* ((*insertDenies* (*separate* (*qsort* (*removeShadowRules2*
(*remdups* (*rm-MT-rules Cp* (*insertDeny* (*removeShadowRules1* (*policy2list*
p)))))) *l*))))) =
Cp p
⟨*proof*⟩

lemma *C-eq-RD-aux*[*rule-format*]: *Cp* (*p*) *x* = *Cp* (*removeDuplicates p*) *x*
⟨*proof*⟩

lemma *C-eq-RAD-aux*[*rule-format*]:
p ≠ [] ⟶ *Cp* (*list2FWpolicy p*) *x* = *Cp* (*list2FWpolicy* (*removeAllDuplicates p*)) *x*
⟨*proof*⟩

lemma *C-eq-RAD*:
p ≠ [] ⟹ *Cp* (*list2FWpolicy p*) = *Cp* (*list2FWpolicy* (*removeAllDuplicates p*))
⟨*proof*⟩

lemma *C-eq-compile*:
DenyAll ∈ *set* (*policy2list p*) ⟹ *all-in-list* (*policy2list p*) *l* ⟹
allNetsDistinct (*policy2list p*) ⟹
Cp (*list2FWpolicy* (*removeAllDuplicates* (*insertDenies* (*separate*
(*sort* (*removeShadowRules2* (*remdups* (*rm-MT-rules Cp* (*insertDeny*
(*removeShadowRules1* (*policy2list p*)))))) *l*))))) = *Cp p*
⟨*proof*⟩

lemma *C-eq-compileQ*:
DenyAll∈*set*(*policy2list p*) ⟹ *all-in-list*(*policy2list p*) *l* ⟹ *allNetsDistinct*(*policy2list p*) ⟹
Cp (*list2FWpolicy* (*removeAllDuplicates* (*insertDenies* (*separate* (*qsort*
(*removeShadowRules2* (*remdups* (*rm-MT-rules Cp* (*insertDeny*
(*removeShadowRules1* (*policy2list p*)))))) *l*))))) = *Cp p*
⟨*proof*⟩

lemma *C-eq-normalizePr*:
DenyAll ∈ *set* (*policy2list p*) ⟹ *allNetsDistinct* (*policy2list p*) ⟹
all-in-list (*policy2list p*) (*Nets-List p*) ⟹
Cp (*list2FWpolicy* (*normalizePr p*)) = *Cp p*

⟨*proof*⟩

lemma *C-eq-normalizePrQ*:
DenyAll $\in$ *set* (*policy2list p*) $\Longrightarrow$ *allNetsDistinct* (*policy2list p*) $\Longrightarrow$
all-in-list (*policy2list p*) (*Nets-List p*) $\Longrightarrow$
Cp (*list2FWpolicy* (*normalizePrQ p*)) = *Cp p*
⟨*proof*⟩

lemma *domSubset3*: *dom* (*Cp* (*DenyAll* $\oplus$ *x*)) = *dom* (*Cp* (*DenyAll*))
⟨*proof*⟩

lemma *domSubset4*:
dom (*Cp* (*DenyAllFromTo x y* $\oplus$ *DenyAllFromTo y x* $\oplus$ *AllowPortFromTo x y dn*)) =

dom (*Cp* (*DenyAllFromTo x y* $\oplus$ *DenyAllFromTo y x*))
⟨*proof*⟩

lemma *domSubset5*:
dom (*Cp* (*DenyAllFromTo x y* $\oplus$ *DenyAllFromTo y x* $\oplus$ *AllowPortFromTo y x dn*)) =

dom (*Cp* (*DenyAllFromTo x y* $\oplus$ *DenyAllFromTo y x*))
⟨*proof*⟩

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

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

lemma *ConcAssoc2*: *Cp* ($X \oplus Y \oplus ((A \oplus B) \oplus D)$) = *Cp* ($X \oplus Y \oplus A \oplus B \oplus D$)
⟨*proof*⟩

lemma *ConcAssoc3*: *Cp* ($X \oplus ((Y \oplus A) \oplus D)$) = *Cp* ($X \oplus Y \oplus A \oplus D$)
⟨*proof*⟩

lemma *RS3-NMT*[*rule-format*]: *DenyAll* $\in$ *set p* $\longrightarrow$
rm-MT-rules Cp p $\neq$ []
⟨*proof*⟩

lemma *norm-notMT*: $DenyAll \in set\ (policy2list\ p) \Longrightarrow normalizePr\ p \neq []$
⟨*proof*⟩

lemma *norm-notMTQ*: $DenyAll \in set\ (policy2list\ p) \Longrightarrow normalizePrQ\ p \neq []$
⟨*proof*⟩

lemma *domDA*: $dom\ (Cp\ (DenyAll \oplus A)) = dom\ (Cp\ (DenyAll))$
⟨*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*]: $p \in dom\ (Cp\ a) \longrightarrow$
$Cp\ (list2policyR\ (a\ \#\ list))\ p = Cp\ a\ p$
⟨*proof*⟩

lemma *list2policyR-End*: $p \notin dom\ (Cp\ a) \Longrightarrow$
$Cp\ (list2policyR\ (a\ \#\ list))\ p = (Cp\ a \bigoplus list2policy\ (map\ Cp\ list))\ p$
⟨*proof*⟩

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

lemma *l2polR-eq*:
$N \neq [] \Longrightarrow Cp(\ list2policyR\ N) = (list2policy\ (map\ Cp\ N))$
⟨*proof*⟩

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

lemma *list2FWpolicys-eq*:
$Filter \neq [] \Longrightarrow$
$Cp\ (list2policyR\ Filter) = Cp\ (list2FWpolicy\ (rev\ Filter))$
⟨*proof*⟩

lemma *list2FWpolicys-eq-sym*:
$Filter \neq [] \Longrightarrow$
$Cp\ (list2policyR\ (rev\ Filter)) = Cp\ (list2FWpolicy\ Filter)$

⟨*proof*⟩

lemma *p-eq*[*rule-format*]: $p \neq [] \longrightarrow$
list2policy (*map Cp* (*rev p*)) = *Cp* (*list2FWpolicy p*)
⟨*proof*⟩

lemma *p-eq2*[*rule-format*]: *normalizePr* $x \neq [] \longrightarrow$
Cp (*list2FWpolicy* (*normalizePr x*)) = *Cp* $x \longrightarrow$
list2policy (*map Cp* (*rev* (*normalizePr x*))) = *Cp x*
⟨*proof*⟩

lemma *p-eq2Q*[*rule-format*]: *normalizePrQ* $x \neq [] \longrightarrow$
Cp (*list2FWpolicy* (*normalizePrQ x*)) = *Cp* $x \longrightarrow$
list2policy (*map Cp* (*rev* (*normalizePrQ x*))) = *Cp x*
⟨*proof*⟩

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

lemma *Norm-Distr2*:
r o-f ((*P* $\bigotimes_2$ (*list2policy Q*)) *o d*) =
(*list2policy* ((*P* $\bigotimes_L$ *Q*) ($\bigotimes_2$) *r d*))
⟨*proof*⟩

lemma *NATDistr*:
$N \neq [] \Longrightarrow F$ = *Cp* (*list2policyR N*) $\Longrightarrow$
((λ (*x*,*y*). *x*) *o-f* ((*NAT* $\bigotimes_2$ *F*) *o* (λ *x*. (*x*,*x*))) =
(*list2policy* (((*NAT* $\bigotimes_L$ (*map Cp N*)) ($\bigotimes_2$)
(λ (*x*,*y*). *x*) (λ *x*. (*x*,*x*))))))
⟨*proof*⟩

lemma *C-eq-normalize-manual*:
DenyAll $\in$ *set* (*policy2list p*) $\Longrightarrow$ *allNetsDistinct* (*policy2list p*) $\Longrightarrow$
all-in-list (*policy2list p*) *l* $\Longrightarrow$
Cp (*list2FWpolicy* (*normalize-manual-orderPr p l*)) = *Cp p*
⟨*proof*⟩

lemma *p-eq2-manualQ*[*rule-format*]:
normalize-manual-orderPrQ x l $\neq [] \longrightarrow$
Cp (*list2FWpolicy* (*normalize-manual-orderPrQ x l*)) = *Cp* $x \longrightarrow$
list2policy (*map Cp* (*rev* (*normalize-manual-orderPrQ x l*))) = *Cp x*
⟨*proof*⟩

lemma *norm-notMT-manualQ*: *DenyAll* $\in$ *set* (*policy2list p*) $\Longrightarrow$ *normal-*

ize-manual-orderPrQ p l $\neq$ []
⟨*proof*⟩

lemma *C-eq-normalizePr-manualQ*:
DenyAll $\in$ *set* (*policy2list p*) $\Longrightarrow$
allNetsDistinct (*policy2list p*) $\Longrightarrow$
all-in-list (*policy2list p*) *l* $\Longrightarrow$
Cp (*list2FWpolicy* (*normalize-manual-orderPrQ p l*)) = *Cp p*
⟨*proof*⟩

lemma *p-eq2-manual*[*rule-format*]: *normalize-manual-orderPr x l* $\neq$ [] $\longrightarrow$
Cp (*list2FWpolicy* (*normalize-manual-orderPr x l*)) = *Cp x* $\longrightarrow$
list2policy (*map Cp* (*rev* (*normalize-manual-orderPr x l*))) = *Cp x*
⟨*proof*⟩

lemma *norm-notMT-manual*: *DenyAll* $\in$ *set* (*policy2list p*) $\Longrightarrow$ *normalize-manual-orderPr p l* $\neq$ []
⟨*proof*⟩

As an example, how this theorems can be used for a concrete normalisation instantiation.

lemma *normalizePrNAT*:
DenyAll $\in$ *set* (*policy2list Filter*) $\Longrightarrow$
allNetsDistinct (*policy2list Filter*) $\Longrightarrow$
all-in-list (*policy2list Filter*) (*Nets-List Filter*) $\Longrightarrow$
((λ (*x*,*y*). *x*) *o-f* (((*NAT* $\bigotimes_2$ *Cp Filter*) *o* (λ*x*. (*x*,*x*))))) =
list2policy ((*NAT* $\bigotimes_L$ (*map Cp* (*rev* (*normalizePr Filter*)))) ($\bigotimes_2$) (λ (*x*,*y*). *x*) (λ *x*. (*x*,*x*)))
⟨*proof*⟩

lemma *domSimpl*[*simp*]: *dom* (*Cp* (*A* $\oplus$ *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* $= '\alpha \times (('\beta,'\gamma)$ *packet* $\mapsto$ *unit*$)$

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

type-synonym $('\alpha,'\beta,'\gamma)$ *FWStateTransitionP* $=$
$('\beta,'\gamma)$ *packet* $\Rightarrow$ $(((('\beta,'\gamma)$ *packet* $\mapsto$ *unit*$)$ *decision*, $('\alpha,'\beta,'\gamma)$ *FWState*$)$ MON_{SE}

type-synonym $('\alpha,'\beta,'\gamma)$ *FWStateTransition* $=$
$((('\beta,'\gamma)$ *packet* $\times$ $('\alpha,'\beta,'\gamma)$ *FWState*$) \rightharpoonup ('\alpha,'\beta,'\gamma)$ *FWState*

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

type-synonym $('\beta,'\gamma)$ *history* $= ('\beta,'\gamma)$ *packet list*

fun *packet-with-id* **where**
packet-with-id [] i = []
|*packet-with-id* ($x\#xs$) i = (*if id* x = i *then* ($x\#$(*packet-with-id xs i*)) *else* (*packet-with-id xs i*))

fun *ids1* **where**
ids1 i ($x\#xs$) = (*id* x = i $\wedge$ *ids1 i xs*)
|*ids1 i* [] = *True*

fun *ids* **where**
ids a ($x\#xs$) = (*NetworkCore.id* $x \in a$ $\wedge$ *ids a xs*)
|*ids a* [] = *True*

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

end

2.4.2 The File Transfer Protocol (ftp)

theory
FTP
imports
StatefulCore
begin

The protocol syntax

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

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

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

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

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

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

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

definition
is-init :: *id* ⇒ (adr_{ip}, *msg*)*packet* ⇒ *bool* **where**
is-init = (λ *i p*. (*id p* = *i* ∧ *content p* = *ftp-init*))

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

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

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

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

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

fun *are-ftp-other* **where**
are-ftp-other i (*x#xs*) = (*is-ftp-other i x* ∧ *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-port-request p)#xs) = (if i=j then p else last-opened-port i xs)
| *last-opened-port i (x#xs) = last-opened-port i xs*
| *last-opened-port x [] = undefined*

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

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

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

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

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

definition *TRPolicy* :: (adr_{ip}, msg)*packet* $\times$ (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-STA*,*FTP-STD*) $\bigotimes_\nabla$ *applyPolicy*) *o* $(\lambda(x,(y,z)).((x,z),(x,(y,z))))$

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

If required to contain the policy in the output

definition $TRPolicy_{Mon}{}'$

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

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

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

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

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

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

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

fun
is-ftp :: *ftp-states* ⇒ adr_{ip} ⇒ adr_{ip} ⇒ *id* ⇒ *port* ⇒
(adr_{ip},*msg*) *history* ⇒ *bool*
where
is-ftp H c s i p [] = (*H=S3*)
|*is-ftp H c s i p* (*x#InL*) = (*snd s* = *21* ∧((λ (*id*,*sr*,*de*,*co*). (((*id* = *i* ∧ (
(*H=ftp-states.S2* ∧ *sr* = *c* ∧ *de* = *s* ∧ *co* = *ftp-init* ∧ *is-ftp S3 c s i p InL*) ∨
(*H=ftp-states.S1* ∧ *sr* = *c* ∧ *de* = *s* ∧ *co* = *ftp-port-request p* ∧ *is-ftp S2 c s i p InL*) ∨
(*H=ftp-states.S1* ∧ *sr* = *s* ∧ *de* = (*fst c*,*p*) ∧ *co*= *ftp-data* ∧ *is-ftp S1 c s i p InL*) ∨
(*H=ftp-states.S0* ∧ *sr* = *c* ∧ *de* = *s* ∧ *co* = *ftp-close* ∧ *is-ftp S1 c s i p InL*)))))) *x*))

definition *is-single-ftp-run* :: adr_{ip} *src* ⇒ adr_{ip} *dest* ⇒ *id* ⇒ *port* ⇒ (adr_{ip},*msg*) *history set*
where *is-single-ftp-run s d i p* = {*x*. (*is-ftp S0 s d i p x*)}

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

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

definition
ftp-2-interleaved :: adr_{ip} *src* ⇒ adr_{ip} *dest* ⇒ *id* ⇒ *port* ⇒

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

lemma *subnetOf-lemma*: (*a*::*int*) ≠ (*c*::*int*) ⟹ ∀ *x*∈*subnet-of* (*a*, *b*::*port*). (*c*, *d*) ∉ *x*
⟨*proof*⟩

lemma *subnetOf-lemma2*: ∀ *x*∈*subnet-of* (*a*::*int*, *b*::*port*). (*a*, *b*) ∈ *x*
⟨*proof*⟩

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

lemma *subnetOf-lemma4*: ∃ *x*∈*subnet-of* (*a*::*int*, *b*::*port*). (*a*, *c*::*port*) ∈ *x*
⟨*proof*⟩

lemma *port-open-lemma*: ¬ (*Ex* (*port-open* [] (*x*::*port*)))
⟨*proof*⟩

lemmas *FTPLemmas* = *TRPolicy-def applyPolicy-def policy2MON-def*
Let-def in-subnet-def src-def
dest-def subnet-of-int-def
is-init-def p-accept-def port-open-def is-ftp-data-def is-ftp-close-def
is-ftp-port-request-def content-def PortCombinators
exI subnetOf-lemma subnetOf-lemma2 subnetOf-lemma3 subnetOf-lemma4

NetworkCore.id-def adr_{ip}*Lemmas port-open-lemma*
bind-SE-def unit-SE-def valid-SE-def
end

2.4.3 FTP enriched with a security policy

theory
FTP-WithPolicy
imports
FTP
begin

FTP where the policy is part of the output.

definition *POL* :: ′*a* ⇒ ′*a* **where** *POL x* = *x*

Variant 2 takes the policy into the output

fun *FTP-STP* ::

((*id* ⇀ *port*), adr_{ip}, *msg*) *FWStateTransitionP*
where

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

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

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

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

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

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

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

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 $VOIP\text{-}STA$ ::
$((adr_{ip},address\ voip\text{-}msg)\ history,\ adr_{ip},\ address\ voip\text{-}msg)\ FWStateTransition$
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)) ⊕ 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) ⊕ policy))
                         else Some (((a,c,d,ARJ)#InL,policy)))

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


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


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


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

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

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

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

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

definition *VOIP-TRPolicy* **where**
VOIP-TRPolicy = policy2MON (
$((VOIP\text{-}STA, VOIP\text{-}STD) \bigotimes_{\nabla} applyPolicy)\ o\ (\lambda\ (x,(y,z)).\ ((x,z),(x,(y,z)))))$

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* ⇒ *address* ⇒ *address* ⇒ *address* ⇒ *id* ⇒ *port* ⇒
    port ⇒ $(adr_{ip}$, *address voip-msg*) *history* ⇒ *bool*
where
is-voip H s d g i p1 p2 [] = (H = S5)
|*is-voip H s d g i p1 p2 (x#InL) =*
(((λ (*id,sr,de,co*).
(((*id = i* ∧
(*H = S4* ∧ ((*sr = (s,1719)* ∧ *de = (g,1719)* ∧ *co = ARQ* ∧
  is-voip S5 s d g i p1 p2 InL))) ∨
(*H = S0* ∧ *sr = (g,1719)* ∧ *de = (s,1719)* ∧ *co = ARJ* ∧
  is-voip S4 s d g i p1 p2 InL) ∨
(*H = S3* ∧ *sr = (g,1719)* ∧ *de = (s,1719)* ∧ *co = ACF d* ∧
  is-voip S4 s d g i p1 p2 InL) ∨
(*H = S2* ∧ *sr = (s,1720)* ∧ *de = (d,1720)* ∧ *co = Setup p1* ∧
  is-voip S3 s d g i p1 p2 InL) ∨
(*H = S1* ∧ *sr = (d,1720)* ∧ *de = (s,1720)* ∧ *co = Connect p2* ∧
  is-voip S2 s d g i p1 p2 InL) ∨
(*H = S1* ∧ *sr = (s,p1)* ∧ *de = (d,p2)* ∧ *co = Stream* ∧
  is-voip S1 s d g i p1 p2 InL) ∨
(*H = S1* ∧ *sr = (d,p2)* ∧ *de = (s,p1)* ∧ *co = Stream* ∧
  is-voip S1 s d g i p1 p2 InL) ∨
(*H = S0* ∧ *sr = (d,1720)* ∧ *de = (s,1720)* ∧ *co = Fin* ∧

is-voip S1 s d g i p1 p2 InL) ∨
(*H* = *S0* ∧ *sr* = (*s*,*1720*) ∧ *de* = (*d*,*1720*) ∧ *co* = *Fin* ∧
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* ⇒ *address* ⇒ *address* ⇒ *id* ⇒ *port* ⇒ *port* ⇒
(adr_{ip}, *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* ⇒ (adr_{ip}, *ftpvoip*) *packet* ⇒ *bool* **where**
FTPVOIP-is-init = (λ *i p*. (*id p* = *i* ∧ *content p* = *ftp-init*))

definition
FTPVOIP-is-port-request :: *id* ⇒ *port* ⇒(adr_{ip}, *ftpvoip*) *packet* ⇒ *bool* **where**

FTPVOIP-is-port-request = (λ *i port p*. (*id p* = *i* ∧ *content p* = *ftp-port-request port*))

definition
FTPVOIP-is-data :: *id* ⇒ (adr_{ip}, *ftpvoip*) *packet* ⇒ *bool* **where**
FTPVOIP-is-data = (λ *i p*. (*id p* = *i* ∧ *content p* = *ftp-data*))

definition
FTPVOIP-is-close :: *id* ⇒ (adr_{ip}, *ftpvoip*) *packet* ⇒ *bool* **where**
FTPVOIP-is-close = (λ *i p*. (*id p* = *i* ∧ *content p* = *ftp-close*))

definition
FTPVOIP-port-open :: (adr_{ip}, *ftpvoip*) *history* ⇒ *id* ⇒ *port* ⇒ *bool* **where**
FTPVOIP-port-open = (λ *L a p*. (*not-before* (*FTPVOIP-is-close a*) (*FTPVOIP-is-port-request a p*) *L*))

definition
FTPVOIP-is-other :: *id* ⇒ (adr_{ip}, *ftpvoip*) *packet* ⇒ *bool* **where**
FTPVOIP-is-other = (λ *i p*. (*id p* = *i* ∧ *content p* = *other*))

fun *FTPVOIP-are-other* **where**
FTPVOIP-are-other i (*x#xs*) = (*FTPVOIP-is-other i x* ∧ *FTPVOIP-are-other i xs*)
|*FTPVOIP-are-other i* [] = *True*

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

fun *FTPVOIP-FTP-STA* ::
((adr_{ip}, *ftpvoip*) *history*, adr_{ip}, *ftpvoip*) *FWStateTransition*
where

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

|*FTPVOIP-FTP-STA* ((*i*,*s*,*d*,*ftp-close*), (*InL*,*policy*)) =
(*if* (∃ *p*. *FTPVOIP-port-open InL i p*) ∧ *dest-port* (*i*,*s*,*d*,*ftp-close*) = (*21*::*port*)
then Some ((*i*,*s*,*d*,*ftp-close*)#*InL*, *policy* ++

deny-from-to-port (*last-opened-port i InL*) (*subnet-of d*) (*subnet-of s*))
else Some (((*i*,*s*,*d*,*ftp-close*)#*InL*, *policy*)))

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

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

definition
FTPVOIP-is-arq :: *NetworkCore.id* ⇒ (′*a*::*adr*, *ftpvoip*) *packet* ⇒ *bool* **where**
FTPVOIP-is-arq i p = (*NetworkCore.id p* = *i* ∧ *content p* = *ARQ*)

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

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

definition
FTPVOIP-is-setup :: *id* ⇒ *port* ⇒ (′*a*::*adr*, *ftpvoip*) *packet* ⇒ *bool* **where**
FTPVOIP-is-setup i port p = (*id p* = *i* ∧ *content p* = *Setup port*)

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

definition
FTPVOIP-ports-open :: *id* ⇒ *port* × *port* ⇒ (*adr*$_{ip}$, *ftpvoip*) *history* ⇒ *bool* **where**
FTPVOIP-ports-open i p L = ((*not-before* (*FTPVOIP-is-fin i*) (*FTPVOIP-is-setup i* (*fst p*)) *L*) ∧
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* ⇒ *adr*$_{ip}$ ⇒ (*adr*$_{ip}$,*ftpvoip*) *history* ⇒ *bool* **where**
FTPVOIP-src-is-initiator i a [] = *False*
|*FTPVOIP-src-is-initiator i a* (*p*#*S*) = (((*id p* = *i*) ∧
(∃ *port*. *content p* = *Setup port*) ∧

((*fst* (*src p*) = *fst a*))) ∨
(*FTPVOIP-src-is-initiator i a S*))

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

fun *FTPVOIP-VOIP-STA* ::
((*adr_{ip}*, *ftpvoip*) *history*, *adr_{ip}*, *ftpvoip*) *FWStateTransition*
where
FTPVOIP-VOIP-STA ((*a*,*c*,*d*,*ARQ*), (*InL*, *policy*)) =
Some (((*a*,*c*,*d*, *ARQ*)#*InL*,
(*allow-from-to-port* (*1719*::*port*)(*subnet-of d*) (*subnet-of c*)) ⨁ *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*) ⨁ *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*) ⨁
allow-from-to-port (*1720*::*port*) (*subnet-of d*) (*subnet-of-adr callee*) ⨁
deny-from-to-port (*1719*::*port*) (*subnet-of d*) (*subnet-of c*) ⨁
policy))

|*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*) ⨁ *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*) ⨁ *policy*))

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

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

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

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

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

definition *FTP-VOIP-STA* :: ((*adr*$_{ip}$, *ftpvoip*) *history*, *adr*$_{ip}$, *ftpvoip*) *FWStateTransition*
where
FTP-VOIP-STA = ((λ(*x*,*x*). *Some x*) $\circ_m$ ((*FTPVOIP-FTP-STA* $\bigotimes_S$ *FTPVOIP-VOIP-STA o* (λ (*p*,*x*). (*p*,*x*,*x*)))))

definition *FTP-VOIP-STD* :: ((*adr*$_{ip}$, *ftpvoip*) *history*, *adr*$_{ip}$, *ftpvoip*) *FWStateTransition*
where
FTP-VOIP-STD = (λ(*x*,*x*). *Some x*) $\circ_m$ ((*FTPVOIP-FTP-STD* $\bigotimes_S$ *FTPVOIP-VOIP-STD o* (λ (*p*,*x*). (*p*,*x*,*x*))))

definition *FTPVOIP-TRPolicy* **where**
FTPVOIP-TRPolicy = *policy2MON* (
(((*FTP-VOIP-STA*,*FTP-VOIP-STD*) $\bigotimes_\nabla$ *applyPolicy*) *o* (λ (*x*,(*y*,*z*)). ((*x*,*z*),(*x*,(*y*,*z*))))))

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

datatype *ftp-states2* = *FS0* | *FS1* | *FS2* | *FS3*
datatype *voip-states2* = *V0* | *V1* | *V2* | *V3* | *V4* | *V5*

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

fun *FTPVOIP-is-voip* :: *voip-states2* ⇒ *address* ⇒ *address* ⇒ *address* ⇒ *id* ⇒ *port* ⇒
port ⇒ (adr_{ip}, *ftpvoip*) *history* ⇒ *bool*
where
FTPVOIP-is-voip H s d g i p1 p2 [] = (*H* = *V5*)
|*FTPVOIP-is-voip H s d g i p1 p2* (*x*#*InL*) =
(((λ (*id*,*sr*,*de*,*co*).
(((*id* = *i* ∧
(*H* = *V4* ∧ ((*sr* = (*s*,*1719*) ∧ *de* = (*g*,*1719*) ∧ *co* = *ARQ* ∧
FTPVOIP-is-voip V5 s d g i p1 p2 InL))) ∨
(*H* = *V0* ∧ *sr* = (*g*,*1719*) ∧ *de* = (*s*,*1719*) ∧ *co* = *ARJ* ∧
FTPVOIP-is-voip V4 s d g i p1 p2 InL) ∨
(*H* = *V3* ∧ *sr* = (*g*,*1719*) ∧ *de* = (*s*,*1719*) ∧ *co* = *ACF d* ∧
FTPVOIP-is-voip V4 s d g i p1 p2 InL) ∨
(*H* = *V2* ∧ *sr* = (*s*,*1720*) ∧ *de* = (*d*,*1720*) ∧ *co* = *Setup p1* ∧
FTPVOIP-is-voip V3 s d g i p1 p2 InL) ∨
(*H* = *V1* ∧ *sr* = (*d*,*1720*) ∧ *de* = (*s*,*1720*) ∧ *co* = *Connect p2* ∧
FTPVOIP-is-voip V2 s d g i p1 p2 InL) ∨
(*H* = *V1* ∧ *sr* = (*s*,*p1*) ∧ *de* = (*d*,*p2*) ∧ *co* = *Stream* ∧
FTPVOIP-is-voip V1 s d g i p1 p2 InL) ∨
(*H* = *V1* ∧ *sr* = (*d*,*p2*) ∧ *de* = (*s*,*p1*) ∧ *co* = *Stream* ∧
FTPVOIP-is-voip V1 s d g i p1 p2 InL) ∨
(*H* = *V0* ∧ *sr* = (*d*,*1720*) ∧ *de* = (*s*,*1720*) ∧ *co* = *Fin* ∧
FTPVOIP-is-voip V1 s d g i p1 p2 InL) ∨
(*H* = *V0* ∧ *sr* = (*s*,*1720*) ∧ *de* = (*d*,*1720*) ∧ *co* = *Fin* ∧
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* ⇒ *address* ⇒ *address* ⇒ *id* ⇒ *port* ⇒ *port* ⇒
(adr_{ip}, *ftpvoip*) *history set* **where**
FTPVOIP-NB-voip s d g i p1 p2= {*x*. (*FTPVOIP-is-voip V0 s d g i p1 p2 x*)}

fun
FTPVOIP-is-ftp :: *ftp-states2* ⇒ adr_{ip} ⇒ adr_{ip} ⇒ *id* ⇒ *port* ⇒
(adr_{ip}, *ftpvoip*) *history* ⇒ *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* ∧((λ (*id*,*sr*,*de*,*co*). (((*id* = *i* ∧ (

$(H{=}FS2 \wedge sr = c \wedge de = s \wedge co = \textit{ftp-init} \wedge \textit{FTPVOIP-is-ftp FS3 c s i p InL}) \vee$
$(H{=}FS1 \wedge sr = c \wedge de = s \wedge co = \textit{ftp-port-request p} \wedge \textit{FTPVOIP-is-ftp FS2 c s i p InL}) \vee$
$(H{=}FS1 \wedge sr = s \wedge de = (\textit{fst c,p}) \wedge co = \textit{ftp-data} \wedge \textit{FTPVOIP-is-ftp FS1 c s i p InL}) \vee$
$(H{=}FS0 \wedge sr = c \wedge de = s \wedge co = \textit{ftp-close} \wedge \textit{FTPVOIP-is-ftp FS1 c s i p InL})$
$))))) x))$

definition
$\textit{FTPVOIP-NB-ftp} :: adr_{ip}\ src \Rightarrow adr_{ip}\ dest \Rightarrow id \Rightarrow port \Rightarrow (adr_{ip}, \textit{ftpvoip})\ history\ set$ **where**
$\textit{FTPVOIP-NB-ftp s d i p} = \{x.\ (\textit{FTPVOIP-is-ftp FS0 s d i p x})\}$

definition
$\textit{ftp-voip-interleaved} :: adr_{ip}\ src \Rightarrow adr_{ip}\ dest \Rightarrow id \Rightarrow port \Rightarrow$
$address \Rightarrow address \Rightarrow address \Rightarrow id \Rightarrow port \Rightarrow port \Rightarrow$
$(adr_{ip}, \textit{ftpvoip})\ history\ set$

where
$\textit{ftp-voip-interleaved s1 d1 i1 p1 vs vd vg vi vp1 vp2} =$
$\{x.\ (\textit{FTPVOIP-is-ftp FS0 s1 d1 i1 p1}\ (\textit{packet-with-id x i1})) \wedge$
$(\textit{FTPVOIP-is-voip V0 vs vd vg vi vp1 vp2}\ (\textit{packet-with-id x vi}))\}$

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 mailserver and the dmz. Therefore, the constraint looks as follows. %

definition
not-in-same-net :: (*Networks*,*DummyContent*) *packet* ⇒ *bool* **where**
$$\textit{not-in-same-net}\ x\ =\ ((\textit{src}\ x \sqsubset \textit{internet} \longrightarrow \neg\ \textit{dest}\ x \sqsubset \textit{internet}) \wedge (\textit{src}\ x \sqsubset \textit{intranet} \longrightarrow \neg\ \textit{dest}\ x \sqsubset \textit{intranet}) \wedge (\textit{src}\ x \sqsubset \textit{dmz} \longrightarrow \neg\ \textit{dest}\ x \sqsubset \textit{dmz}))$$

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 trafic from Internet to Intranet
- allo imaps and smtp from intranet to mailserver
- allow smtp from Internet to mailserver
- allow http(s) from Internet to webserver
- deny everything else

definition
$intranet::adr_{ip}\ net$ **where**
$intranet = \{\{(a,b)\ .\ (a > 1 \wedge a < 4)\ \}\}$
definition
$dmz :: adr_{ip}\ net$ **where**
$dmz = \{\{(a,b).\ (a > 6) \wedge (a < 11)\}\}$
definition
$mail :: adr_{ip}\ net$ **where**
$mail = \{\{(a,b).\ a = 7\}\}$
definition
$web :: adr_{ip}\ net$ **where**
$web = \{\{(a,b).\ a = 8\ \}\}$
definition
$internet :: adr_{ip}\ net$ **where**
$internet = \{\{(a,b).\ \neg\ (\ (a > 1 \wedge a < 4) \vee (a > 6) \wedge (a < 11))\quad \}\}$

definition
$Intranet\text{-}mail\text{-}port :: (adr_{ip},'b)\ FWPolicy$ **where**
$Intranet\text{-}mail\text{-}port = (allow\text{-}from\text{-}to\text{-}ports\ \{21::port,14\}\ intranet\ mail)$

definition
$Intranet\text{-}Internet\text{-}port :: (adr_{ip},'b)\ FWPolicy$ **where**
$Intranet\text{-}Internet\text{-}port = allow\text{-}from\text{-}to\text{-}ports\ \{80::port,90\}\ intranet\ internet$

definition
$Internet\text{-}web\text{-}port :: (adr_{ip},'b)\ FWPolicy$ **where**
$Internet\text{-}web\text{-}port = (allow\text{-}from\text{-}to\text{-}ports\ \{80::port,90\}\ internet\ web)$

definition
$Internet\text{-}mail\text{-}port :: (adr_{ip},'b)\ FWPolicy$ **where**
$Internet\text{-}mail\text{-}port =\ (allow\text{-}all\text{-}from\text{-}port\text{-}to\ internet\ (21::port)\ dmz\ )$

definition
$policyPort :: (adr_{ip},\ DummyContent)\ FWPolicy$ **where**
$policyPort = deny\text{-}all\ ++$
$Intranet\text{-}Internet\text{-}port\ ++$
$Intranet\text{-}mail\text{-}port\ ++$
$Internet\text{-}mail\text{-}port\ ++$
$Internet\text{-}web\text{-}port$

We only want to create test cases which are sent between the three main networks: e.g. not between the mailserver and the dmz. Therefore, the constraint looks as follows.

definition
$not\text{-}in\text{-}same\text{-}net :: (adr_{ip},DummyContent)\ packet \Rightarrow bool$ **where**

$$\begin{aligned} \textit{not-in-same-net } x \ = \ & ((\textit{src } x \sqsubset \textit{internet} \longrightarrow \neg \textit{ dest } x \sqsubset \textit{internet}) \wedge \\ & (\textit{src } x \sqsubset \textit{intranet} \longrightarrow \neg \textit{ dest } x \sqsubset \textit{intranet}) \wedge \\ & (\textit{src } x \sqsubset \textit{dmz} \longrightarrow \neg \textit{ dest } x \sqsubset \textit{dmz})) \end{aligned}$$

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
$PC :: (adr_{ip}\ net)$ **where**
$PC = \{\{(a,b).\ a = 3\}\}$

definition

Internet :: adr_{ip} *net* **where**
Internet = {{(*a*,*b*). ¬ (*a* = *3*)}}

definition
not-in-same-net :: (adr_{ip},*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

definition
strictPolicy :: (adr_{ip},*DummyContent*) *FWPolicy* **where**
strictPolicy = *deny-all* ++ *allow-all-from-to PC Internet*

definition
PortPolicy :: (adr_{ip},*DummyContent*) *FWPolicy* **where**
PortPolicy = *deny-all* ++ *allow-from-ports-to* {*http*,*smtp*,*ftp*} *PC Internet*

definition
PortPolicyBig :: (adr_{ip},*DummyContent*) *FWPolicy* **where**
PortPolicyBig = *deny-all* ++
allow-from-port-to http PC Internet ++
allow-from-port-to smtp PC Internet ++
allow-from-port-to ftp PC Internet

lemmas *policyLemmas* = *strictPolicy-def PortPolicy-def PC-def*
Internet-def PortPolicyBig-def src-def dest-def
adr_{ip}*Lemmas content-def*
PortCombinators in-subnet-def PortPolicyBig-def id-def

declare *Ports* [*simp add*]

definition *wellformed-packet*::(adr_{ip},*DummyContent*) *packet* ⇒ *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* ⇒ *bool* **where**
not-in-same-net x = ((*src x* ⊏ *PC* ⟶ *dest x* ⊏ *Internet*) ∧ (*src x* ⊏ *Internet* ⟶ *dest x* ⊏ *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 $Adr = pc \mid internet$

type-synonym $DatatypeTwoNets = Adr \times int$

instance $Adr{::}adr$ $\langle proof \rangle$

definition
$PC :: DatatypeTwoNets\ net$ **where**
$PC = \{\{(a,b).\ a = pc\}\}$

definition
$Internet :: DatatypeTwoNets\ net$ **where**
$Internet = \{\{(a,b).\ a = internet\}\}$

definition
$not\text{-}in\text{-}same\text{-}net :: (DatatypeTwoNets,DummyContent)\ packet \Rightarrow bool$ **where**
$not\text{-}in\text{-}same\text{-}net\ x = ((src\ x \sqsubset PC \longrightarrow dest\ x \sqsubset Internet) \wedge (src\ x \sqsubset Internet \longrightarrow dest\ x \sqsubset PC))$

Definitions of the policies

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**
$strictPolicy = deny\text{-}all ++ allow\text{-}all\text{-}from\text{-}to\ PC\ Internet$

definition
$PortPolicy :: (DatatypeTwoNets,'b)\ FWPolicy$ **where**
$PortPolicy = deny\text{-}all ++ allow\text{-}from\text{-}ports\text{-}to\ \{80{::}port,24,21\}\ PC\ Internet$

definition
$PortPolicyBig :: (DatatypeTwoNets,'b)\ FWPolicy$ **where**
$PortPolicyBig =$

allow-from-port-to (*80*::*port*) *PC Internet* ⊕
allow-from-port-to (*24*::*port*) *PC Internet* ⊕
allow-from-port-to (*21*::*port*) *PC Internet* ⊕
deny-all

lemmas *policyLemmas* = *strictPolicy-def PortPolicy-def PC-def Internet-def PortPolicyBig-def src-def*
PolicyCombinators PortCombinators in-subnet-def

end

3.3 Demonstrating Policy Transformations

theory
Transformation
imports
Transformation01
Transformation02
begin
end

3.3.1 Transformation Example 1

theory
Transformation01
imports
../../UPF−Firewall
begin

definition
FWLink :: adr_{ip} *net* **where**
FWLink = {{(*a*,*b*). *a* = *1*}}

definition
any :: adr_{ip} *net* **where**
any = {{(*a*,*b*). *a* > *5*}}

definition
i4:: adr_{ip} *net* **where**
i4 = {{(*a*,*b*). *a* = *2* }}

definition
i27:: adr_{ip} *net* **where**
i27 = {{(*a*,*b*). *a* = *3* }}

definition
eth-intern:: adr_{ip} *net* **where**
eth-intern = {{(*a*,*b*). *a* = *4* }}

definition
eth-private:: adr_{ip} *net* **where**
eth-private = {{(*a*,*b*). *a* = *5* }}

definition

MG2 :: (adr_{ip} *net*,*port*) *Combinators* **where**
MG2 = *AllowPortFromTo i27 any 1* ⊕
AllowPortFromTo i27 any 2 ⊕
AllowPortFromTo i27 any 3

definition
MG3 :: (adr_{ip} *net*,*port*) *Combinators* **where**
MG3 = *AllowPortFromTo any FWLink 1*

definition
MG4 :: (adr_{ip} *net*,*port*) *Combinators* **where**
MG4 = *AllowPortFromTo FWLink FWLink 4*

definition
MG7 :: (adr_{ip} *net*,*port*) *Combinators* **where**
MG7 = *AllowPortFromTo FWLink i4 6* ⊕
AllowPortFromTo FWLink i4 7

definition
MG8 :: (adr_{ip} *net*,*port*) *Combinators* **where**
MG8 = *AllowPortFromTo FWLink i4 6* ⊕
AllowPortFromTo FWLink i4 7

definition
DG3:: (adr_{ip} *net*,*port*) *Combinators* **where**
DG3 = *AllowPortFromTo any any 7*

definition
Policy = *DenyAll* ⊕ *MG8* ⊕ *MG7* ⊕ *MG4* ⊕ *MG3* ⊕ *MG2* ⊕ *DG3*

lemmas *PolicyLemmas* = *Policy-def*
FWLink-def

any-def
i27-def
i4-def
eth-intern-def
eth-private-def
MG2-def MG3-def MG4-def MG7-def MG8-def
DG3-def

lemmas *PolicyL* = *MG2-def MG3-def MG4-def MG7-def MG8-def DG3-def Policy-def*

definition
not-in-same-net :: (adr_{ip},*DummyContent*) *packet* ⇒ *bool* **where**
not-in-same-net x = $(((src\ x \sqsubset i27) \longrightarrow (\neg (dest\ x \sqsubset i27))) \wedge$
$((src\ x \sqsubset i4) \longrightarrow (\neg (dest\ x \sqsubset i4))) \wedge$
$((src\ x \sqsubset eth\text{-}intern) \longrightarrow (\neg (dest\ x \sqsubset eth\text{-}intern))) \wedge$
$((src\ x \sqsubset eth\text{-}private) \longrightarrow (\neg (dest\ x \sqsubset eth\text{-}private))))$

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*: $(n{::}int) \neq m \Longrightarrow \{(a,b).\ a = n\} \neq \{(a,b).\ a = m\}$
⟨*proof*⟩

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

lemma *sets-distinct3*: $\{((a{::}int),(b{::}int)).\ a = n\} \neq \{(a,b).\ a > n\}$
⟨*proof*⟩

lemma *sets-distinct4*: $\{((a{::}int),(b{::}int)).\ a > n\} \neq \{(a,b).\ a = n\}$
⟨*proof*⟩

lemma *aux*: $[\![a \in c;\ a \notin d;\ c = d]\!] \Longrightarrow False$
⟨*proof*⟩

lemma *sets-distinct5*: $(s{::}int) < g \Longrightarrow \{(a{::}int,\ b{::}int).\ a = s\} \neq \{(a{::}int,\ b{::}int).\ g < a\}$
⟨*proof*⟩

lemma *sets-distinct6*: (*s*::*int*) < *g* ⟹ {(*a*::*int*, *b*::*int*). *g* < *a*} ≠ {(*a*::*int*, *b*::*int*). *a* = *s*}
⟨*proof*⟩

lemma *distinctNets*: *FWLink* ≠ *any* ∧ *FWLink* ≠ *i4* ∧ *FWLink* ≠ *i27* ∧ *FWLink* ≠ *eth-intern* ∧ *FWLink* ≠ *eth-private* ∧
any ≠ *FWLink* ∧ *any* ≠ *i4* ∧ *any* ≠ *i27* ∧ *any* ≠ *eth-intern* ∧ *any* ≠ *eth-private* ∧ *i4* ≠ *FWLink* ∧
i4 ≠ *any* ∧ *i4* ≠ *i27* ∧ *i4* ≠ *eth-intern* ∧ *i4* ≠ *eth-private* ∧ *i27* ≠ *FWLink* ∧ *i27* ≠ *any* ∧
i27 ≠ *i4* ∧ *i27* ≠ *eth-intern* ∧ *i27* ≠ *eth-private* ∧ *eth-intern* ≠ *FWLink* ∧ *eth-intern* ≠ *any* ∧
eth-intern ≠ *i4* ∧ *eth-intern* ≠ *i27* ∧ *eth-intern* ≠ *eth-private* ∧ *eth-private* ≠ *FWLink* ∧
eth-private ≠ *any* ∧ *eth-private* ≠ *i4* ∧ *eth-private* ≠ *i27* ∧ *eth-private* ≠ *eth-intern*
⟨*proof*⟩

lemma *aux5*: ⟦*x* ≠ *a*; *y*≠*b*; (*x* ≠ *y* ∧ *x* ≠ *b*) ∨ (*a* ≠ *b* ∧ *a* ≠ *y*)⟧ ⟹ {*x*,*a*} ≠ {*y*,*b*}
⟨*proof*⟩

lemma *aux2*: {*a*,*b*} = {*b*,*a*}
⟨*proof*⟩

lemma *ANDex*: *allNetsDistinct* (*policy2list Policy*)
⟨*proof*⟩

fun (*sequential*) *numberOfRules* **where**
numberOfRules (*a*⊕*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 Transforamtion Example 2

theory
  Transformation02
  imports
    ../../UPF−Firewall
begin

definition
  FWLink :: adr_{ip} *net* **where**
  FWLink = {{(*a*,*b*). *a* = *1*}}

definition
  any :: adr_{ip} *net* **where**
  any = {{(*a*,*b*). *a* > *5*}}

definition
  i4-32:: adr_{ip} *net* **where**
  i4-32 = {{(*a*,*b*). *a* = *2* }}

definition
  i10-32:: adr_{ip} *net* **where**
  i10-32 = {{(*a*,*b*). *a* = *3* }}

definition
  eth-intern:: adr_{ip} *net* **where**
  eth-intern = {{(*a*,*b*). *a* = *4* }}

definition
  eth-private:: adr_{ip} *net* **where**
  eth-private = {{(*a*,*b*). *a* = *5* }}

definition
  D1a :: (adr_{ip} *net*, *port*) *Combinators* **where**
  D1a = *AllowPortFromTo eth-intern any 1* ⊕
      AllowPortFromTo eth-intern any 2

definition
  D1b :: (adr_{ip} *net*, *port*) *Combinators* **where**
  D1b = *AllowPortFromTo eth-private any 1* ⊕
      AllowPortFromTo eth-private any 2

definition
  D2a :: (adr_{ip} *net*, *port*) *Combinators* **where**

D2a = *AllowPortFromTo any i4-32 21*

definition
D2b :: (*adr*$_{ip}$ *net*, *port*) *Combinators* **where**
D2b = *AllowPortFromTo any i10-32 21* ⊕
AllowPortFromTo any i10-32 43

definition
Policy :: (*adr*$_{ip}$ *net*, *port*) *Combinators* **where**
Policy = *DenyAll* ⊕ *D2b* ⊕ *D2a* ⊕ *D1b* ⊕ *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* ∧ *content p* = *fixContent*)

lemmas *fixDefs* = *fixElements-def NetworkCore.id-def NetworkCore.content-def*

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

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

lemma *sets-distinct3*: $\{((a{::}int),(b{::}int)).\ a = n\} \neq \{(a,b).\ a > n\}$
⟨*proof*⟩

lemma *sets-distinct4*: $\{((a{::}int),(b{::}int)).\ a > n\} \neq \{(a,b).\ a = n\}$
⟨*proof*⟩

lemma *aux*: $[\![a \in c;\ a \notin d;\ c = d]\!] \Longrightarrow False$
⟨*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* ⟹ {(*a*::*int*, *b*::*int*). *g* < *a*} ≠ {(*a*::*int*, *b*::*int*). *a* = *s*}
⟨*proof*⟩

lemma *distinctNets*: *FWLink* ≠ *any* ∧ *FWLink* ≠ *i4-32* ∧ *FWLink* ≠ *i10-32* ∧ *FWLink* ≠ *eth-intern* ∧ *FWLink* ≠ *eth-private* ∧ *any* ≠ *FWLink* ∧ *any* ≠ *i4-32* ∧ *any* ≠ *i10-32* ∧ *any* ≠ *eth-intern* ∧ *any* ≠ *eth-private* ∧ *i4-32* ≠ *FWLink* ∧ *i4-32* ≠ *any* ∧ *i4-32* ≠ *i10-32* ∧ *i4-32* ≠ *eth-intern* ∧ *i4-32* ≠ *eth-private* ∧ *i10-32* ≠ *FWLink* ∧ *i10-32* ≠ *any* ∧ *i10-32* ≠ *i4-32* ∧ *i10-32* ≠ *eth-intern* ∧ *i10-32* ≠ *eth-private* ∧ *eth-intern* ≠ *FWLink* ∧ *eth-intern* ≠ *any* ∧ *eth-intern* ≠ *i4-32* ∧ *eth-intern* ≠ *i10-32* ∧ *eth-intern* ≠ *eth-private* ∧ *eth-private* ≠ *FWLink* ∧ *eth-private* ≠ *any* ∧ *eth-private* ≠ *i4-32* ∧ *eth-private* ≠ *i10-32* ∧ *eth-private* ≠ *eth-intern*
⟨*proof*⟩

lemma *aux5*: ⟦*x* ≠ *a*; *y*≠*b*; (*x* ≠ *y* ∧ *x* ≠ *b*) ∨ (*a* ≠ *b* ∧ *a* ≠ *y*)⟧ ⟹ {*x*,*a*} ≠ {*y*,*b*}
⟨*proof*⟩

lemma *aux2*: {*a*,*b*} = {*b*,*a*}
⟨*proof*⟩

lemma *ANDex*: *allNetsDistinct* (*policy2list Policy*)
⟨*proof*⟩

fun (*sequential*) *numberOfRules* **where**
numberOfRules (*a*⊕*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_{ip} *net* **where**
 subnet1 = {{(*d*,*e*). *d* > *1* ∧ *d* < *256*}}

definition *subnet2* :: adr_{ip} *net* **where**
 subnet2 = {{(*d*,*e*). *d* > *500* ∧ *d* < *1256*}}

definition
  accross-subnets x ≡
 ((*src x* ⊏ *subnet1* ∧ (*dest x* ⊏ *subnet2*)) ∨
  (*src x* ⊏ *subnet2* ∧ (*dest x* ⊏ *subnet1*)))

definition
  filter :: (adr_{ip}, *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_f(λ*x*. {*x*}))

lemmas *UnfoldPolicy0* =*filter-def nat-0-def*
          NATLemmas
          ProtocolPortCombinators.ProtocolCombinators
          adr_{ip}*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* ∧ *d* < *3*}

definition *Adr21* :: *int set* **where**
 Adr21 = {*d*. *d* > *502* ∧ *d* < *503*}

definition *nat-1* **where**
nat-1 = *nat-0* ++ (*srcPat2pool-IntPort Adr11 Adr21*)

definition *policy-1* **where**
policy-1 = ((λ (*x*,*y*). *x*) *o-f*
((*nat-1* $\bigotimes_2$ *filter*) *o* (λ *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 = ((λ (*x*,*y*). *x*) *o-f*
((*nat-2* $\bigotimes_2$ *filter*) *o* (λ *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 = ((λ (*x*,*y*). *x*) *o-f*
((*nat-3* $\bigotimes_2$ *filter*) *o* (λ *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 = ((λ (*x*,*y*). *x*) *o-f*
((*nat-4* $\bigotimes_2$ *filter*) *o* (λ *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* ∧ *d* < *15*}

definition *Adr25* :: *int set* **where**
Adr25 = {*d*. *d* > *510* ∧ *d* < *515*}

definition *nat-5* **where**
nat-5 = *nat-4* ++ (*srcPat2pool-IntPort Adr15 Adr25*)

definition *policy-5* **where**
policy-5 = ((λ (*x*,*y*). *x*) *o-f*
((*nat-5* $\bigotimes_2$ *filter*) *o* (λ *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* ∧ *d* < *18*}

definition *Adr26* :: *int set* **where**
Adr26 = {*d*. *d* > *512* ∧ *d* < *518*}

definition *nat-6* **where**
nat-6 = *nat-5* ++ (*srcPat2pool-IntPort Adr16 Adr26*)

definition *policy-6* **where**
policy-6 = ((λ (*x*,*y*). *x*) *o-f*
((*nat-6* $\bigotimes_2$ *filter*) *o* (λ *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* ∧ *d* < *21*}

definition *Adr27* :: *int set* **where**

Adr27 = {*d*. *d* > *514* ∧ *d* < *521*}

definition *nat-7* **where**
nat-7 = *nat-6* ++ (*srcPat2pool-IntPort Adr17 Adr27*)

definition *policy-7* **where**
policy-7 = ((λ (*x*,*y*). *x*) *o-f*
((*nat-7* $\bigotimes_2$ *filter*) *o* (λ *x*. (*x*,*x*))))

lemmas *UnfoldPolicy7* = *UnfoldPolicy6 nat-7-def Adr17-def Adr27-def policy-7-def*

definition *Adr18* :: *int set*
where *Adr18* = {*d*. *d* > *16* ∧ *d* < *24*}

definition *Adr28* :: *int set* **where**
Adr28 = {*d*. *d* > *516* ∧ *d* < *524*}

definition *nat-8* **where**
nat-8 = *nat-7* ++ (*srcPat2pool-IntPort Adr18 Adr28*)

definition *policy-8* **where**
policy-8 = ((λ (*x*,*y*). *x*) *o-f*
((*nat-8* $\bigotimes_2$ *filter*) *o* (λ *x*. (*x*,*x*))))

lemmas *UnfoldPolicy8* = *UnfoldPolicy7 nat-8-def Adr18-def Adr28-def policy-8-def*

definition *Adr19* :: *int set*
where *Adr19* = {*d*. *d* > *18* ∧ *d* < *27*}

definition *Adr29* :: *int set* **where**
Adr29 = {*d*. *d* > *518* ∧ *d* < *527*}

definition *nat-9* **where**
nat-9 = *nat-8* ++ (*srcPat2pool-IntPort Adr19 Adr29*)

definition *policy-9* **where**
policy-9 = ((λ (*x*,*y*). *x*) *o-f*
((*nat-9* $\bigotimes_2$ *filter*) *o* (λ *x*. (*x*,*x*))))

lemmas *UnfoldPolicy9* = *UnfoldPolicy8 nat-9-def Adr19-def Adr29-def policy-9-def*

definition *Adr110* :: *int set*
where *Adr110* = {*d*. *d* > *20* ∧ *d* < *30*}

definition *Adr210* :: *int set* **where**
Adr210 = $\{d.\ d > 520 \wedge d < 530\}$

definition *nat-10* **where**
nat-10 = *nat-9* ++ (*srcPat2pool-IntPort Adr110 Adr210*)

definition *policy-10* **where**
policy-10 = $((\lambda\ (x,y).\ x)$ *o-f*
$((nat\text{-}10 \bigotimes_2 filter)\ o\ (\lambda\ x.\ (x,x))))$

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 definining the networks and the policy. We use a rather simple policy which allows only FTP connections starting from the Intranet and going to the Internet, and deny everything else.

definition
intranet :: adr_{ip} *net* **where**
intranet = $\{\{(a,e)\ .\ a = 3\}\}$

definition
internet :: adr_{ip} *net* **where**
internet = $\{\{(a,c).\ a > 4\}\}$

definition
gatekeeper :: adr_{ip} *net* **where**
gatekeeper = $\{\{(a,c).\ a = 4\}\}$

definition
voip-policy :: (adr_{ip},*address voip-msg*) *FWPolicy* **where**
voip-policy = A_U

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

definition
is-in-intranet :: *address* ⇒ *bool* **where**
is-in-intranet a = (*a* = *3*)

definition
is-gatekeeper :: *address* ⇒ *bool* **where**
is-gatekeeper a = (*a* = *4*)

definition
is-in-internet :: *address* ⇒ *bool* **where**
is-in-internet a = (*a* > *4*)

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

definition
σ-0-voip :: $(adr_{ip}$, *address voip-msg*) *history* ×
$(adr_{ip}$, *address voip-msg*) *FWPolicy*
where
σ-0-voip = ([],*voip-policy*)

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

definition *accept-voip* :: $(adr_{ip}$, *address voip-msg*) *history* ⇒ *bool* **where**
accept-voip t = (∃ *c s g i p1 p2*. *t* ∈ *NB-voip c s g i p1 p2* ∧ *is-in-intranet c*
∧ *is-in-internet s*
∧ *is-gatekeeper g*)

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*))

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* ∧ *ids1 i xs*)
|*ids1 i* [] = *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 σ-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.

[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.

[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.

[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.

[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. 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.