Internet Draft
Expires May 17, 1999
draft-ietf-rps-rpsl-v2-00.txt

Cengiz Alaettinoglu
USC/Information Sciences Institute
Curtis Villamizar
ANS
Elise Gerich
At Home Network
David Kessens
Qwest
David Meyer
University of Oregon
Tony Bates
Cisco Systems
Daniel Karrenberg
RIPE
Marten Terpstra
Bay Networks
November 17, 1998

Routing Policy Specification Language (RPSL)

Status of this Memo

RPSL allows a network operator to be able to specify routing policies at various levels in the Internet hierarchy; for example at the Autonomous System (AS) level. At the same time, policies can be specified with sufficient detail in RPSL so that low level router configurations can be generated from them. RPSL is extensible; new routing protocols and new protocol features can be introduced at any time.

This document is an Internet Draft, and can be found as draft-ietf-rps-rpsl-v2-00.txt in any standard internet drafts repository. Internet Drafts are working documents of the Internet Engineering Task Force (IETF), its Areas, and its Working Groups. Note that other groups may also distribute working documents as Internet Drafts.

Internet Drafts are draft documents valid for a maximum of six months. Internet Drafts may be updated, replaced, or obsoleted by other documents at any time. It is not appropriate to use Internet Drafts as reference material, or to cite them other than as a "working draft" or "work in progress."

Please check the I-D abstract listing contained in each Internet Draft directory to learn the current status of this or any other Internet Draft.

Contents

1 Introduction

This Internet Draft is the reference document for the Routing Policy Specification Language (RPSL). RPSL allows a network operator to be able to specify routing policies at various levels in the Internet hierarchy; for example at the Autonomous System (AS) level. At the same time, policies can be specified with sufficient detail in RPSL so that low level router configurations can be generated from them. RPSL is extensible; new routing protocols and new protocol features can be introduced at any time.

RPSL is a replacement for the current Internet policy specification language known as RIPE-181 [5] or RFC-1786 [6]. RIPE-81 [7] was the first language deployed in the Internet for specifying routing policies. It was later replaced by RIPE-181 [5]. Through operational use of RIPE-181 it has become apparent that certain policies cannot be specified and a need for an enhanced and more generalized language is needed. RPSL addresses RIPE-181's limitations.

RPSL was designed so that a view of the global routing policy can be contained in a single cooperatively maintained distributed database to improve the integrity of Internet's routing. RPSL is not designed to be a router configuration language. RPSL is designed so that router configurations can be generated from the description of the policy for one autonomous system (`aut-num` class) combined with the description of a router (`inet-rtr` class), mainly providing router ID, autonomous system number of the router, interfaces and peers of the router, and combined with a global database mappings from AS sets to ASes (`as-set` class), and from origin ASes and route sets to route prefixes (`route` and `route-set` classes). The accurate population of the RPSL database can help contribute toward such goals as router configurations that protect against accidental (or malicious) distribution of inaccurate routing information, verification of Internet's routing, and aggregation boundaries beyond a single AS.

RPSL is object oriented; that is, objects contain pieces of policy and administrative information. These objects are registered in the Internet Routing Registry (IRR) by the authorized organizations. The registration process is beyond the scope of this document. Please refer to [1, 16, 3] for more details on the IRR.

In the following sections, we present the classes that are used to define various policy and administrative objects. The "*mntner*" class defines entities authorized to add, delete and modify a set of objects. The "*person*" and "*role*" classes describes technical and administrative contact personnel. Autonomous systems (ASes) are specified using the "*aut-num*" class. Routes are specified using the "*route*" class. Sets of ASes and routes can be defined using the "*as-set*" and "*route-set*" classes. The "*dictionary*" class provides the extensibility to the language. The "*inet-rtr*" class is used to specify routers. Many of these classes were originally defined in earlier documents [5, 12, 15, 11, 4] and have all been enhanced.

This document is self-contained. However, the reader is encouraged to read RIPE-181 [6] and the associated documents [12, 15, 11, 4] as they provide significant background as to the motivation and underlying principles behind RIPE-181 and consequently, RPSL. For a tutorial on RPSL, the reader should read the RPSL applications document [3].

2 RPSL Names, Reserved Words, and Representation

Each class has a set of attributes which store a piece of information about the objects of the class. Attributes can be mandatory or optional: A mandatory attribute has to be defined for all objects of the class; optional attributes can be skipped. Attributes can also be single or multiple valued. Each object is uniquely identified by a set of attributes, referred to as the class "key".

The value of an attribute has a type. The following types are most widely used. Note that RPSL is case insensitive and only the characters from the ASCII character set can be used.

⟨*object-name*⟩ Many objects in RPSL have a name. An ⟨*object-name*⟩ is made up of letters, digits, the character underscore "_", and the character hyphen "-"; the first character of a name must be a letter, and the last character of a name must be a letter or a digit. The following words are reserved by RPSL, and they can not be used as names:

```
any as-any rs-any peeras
and or not
atomic from to at action accept announce except refine
networks into inbound outbound
```

Names starting with certain prefixes are reserved for certain object types. Names starting with "`as-`" are reserved for as set names. Names starting with "`rs-`" are reserved for route set names.

⟨*as-number*⟩ An AS number x is represented as the string "`AS`x". That is, the AS 226 is represented as `AS226`.

⟨*ipv4-address*⟩ An IPv4 address is represented as a sequence of four integers in the range from 0 to 255 separated by the character dot ".". For example, `128.9.128.5` represents a valid IPv4 address. In the rest of this document, we may refer to IPv4 addresses as IP addresses.

⟨*address-prefix*⟩ An address prefix is represented as an IPv4 address followed by the character slash "/" followed by an integer in the range from 0 to 32. The following are valid address prefixes: `128.9.128.5/32`, `128.9.0.0/16`, `0.0.0.0/0`; and the following address prefixes are invalid: `0/0`, `128.9/16` since `0` or `128.9` are not strings containing four integers.

⟨*address-prefix-range*⟩ An address prefix range is an address prefix followed by an optional range operator. The range operators are:

`^-` is the *exclusive more specifics* operator; it stands for the more specifics of the address prefix excluding the address prefix itself. For example, `128.9.0.0/16^-` contains all the more specifics of `128.9.0.0/16` excluding `128.9.0.0/16`.

`^+` is the *inclusive more specifics* operator; it stands for the more specifics of the address prefix including the address prefix itself. For example, `5.0.0.0/8^+` contains all the more specifics of `5.0.0.0/8` including `5.0.0.0/8`.

`^n` where `n` is an integer, stands for all the length `n` specifics of the address prefix. For example, `30.0.0.0/8^16` contains all the more specifics of `30.0.0.0/8` which are of length 16 such as `30.9.0.0/16`.

`^n-m` where `n` and `m` are integers, stands for all the length `n` to length `m` specifics of the address prefix. For example, `30.0.0.0/8^24-32` contains all the more specifics of `30.0.0.0/8` which are of length 24 to 32 such as `30.9.9.96/28`.

Range operators can also be applied to address prefix sets. In this case, they distribute over the members of the set. For example, for a `route-set` (defined later) `rs-foo`, `rs-foo^+` contains all the inclusive more specifics of all the prefixes in `rs-foo`.

It is an error to follow a range operator with another one (e.g. `30.0.0.0/8^24-28^+` is an error). However, a range operator can be applied to an address prefix set that has address prefix ranges in it (e.g. {`30.0.0.0/8^24-28`}^27-30 is *not* an error). In this case, the outer operator `^n-m` distributes over the inner operator `^k-l` and becomes the operator `^max(n,k)-m` if `m` is greater than or equal to `max(n,k)`, or otherwise, the prefix is deleted from the set. Note that the operator `^n` is equivalent to `^n-n`; `prefix/l^+` is equivalent to `prefix/l^l-32`; `prefix/l^-` is equivalent to `prefix/l^(l+1)-32`; {`prefix/l^n-m`}`^+` is equivalent to {`prefix/l^n-32`}; and {`prefix/l^n-m`}`^-` is equivalent to {`prefix/l^(n+1)-32`}. For example,

```
{128.9.0.0/16^+}^-      == {128.9.0.0/16^-}
{128.9.0.0/16^-}^+      == {128.9.0.0/16^-}
{128.9.0.0/16^17}^24    == {128.9.0.0/16^24}
{128.9.0.0/16^20-24}^26-28 == {128.9.0.0/16^26-28}
{128.9.0.0/16^20-24}^22-28 == {128.9.0.0/16^22-28}
{128.9.0.0/16^20-24}^18-28 == {128.9.0.0/16^20-28}
{128.9.0.0/16^20-24}^18-22 == {128.9.0.0/16^20-22}
{128.9.0.0/16^20-24}^18-19 == {}
```

⟨*date*⟩ A date is represented as an eight digit integer of the form `YYYYMMDD` where `YYYY` represents the year, `MM` represents the month of the year (01 through 12), and `DD` represents the day of the month (01 through 31). All dates are in UTC unless otherwise specified. For example, June 24, 1996 is represented as 19960624.

⟨*email-address*⟩ is as described in RFC-822[9].

⟨*dns-name*⟩ is as described in RFC-1034[17].

⟨*nic-handle*⟩ is a uniquely assigned identifier[14] used by routing, address allocation, and other registries to unambiguously refer to contact information. `person` and `role` classes map NIC handles to actual person names, and contact information.

⟨*free-form*⟩ is a sequence of ASCII characters.

⟨`X`-*name*⟩ is a name of an object of type `X`. That is ⟨`mntner`-*name*⟩ is a name of a `mntner` object.

⟨*registry-name*⟩ is a name of an IRR registry. The routing registries are listed in Appendix A.

A value of an attribute may also be a *list* of one of these types. A list is represented by separating the list members by commas ",". For example, "`AS1, AS2, AS3, AS4`" is a list of AS numbers. Note that being list valued and being multiple valued are orthogonal. A multiple valued attribute has more than one value, each of which may or may not be a list. On the other hand a single valued attribute may have a list value.

An RPSL object is textually represented as a list of attribute-value pairs. Each attribute-value pair is written on a separate line. The attribute name starts at column 0, followed by character ":" and followed by the value of the attribute. The object's representation ends when a blank line is encountered. An attribute's value can be split over multiple lines, by having a space, a tab or a plus ('+') character as the *first* character of the continuation lines. The character "+" for line continuation allows attribute values to contain blank lines. More spaces may optionally be used after the continuation character to increase readability. The order of attribute-value pairs is significant.

An object's description may contain comments. A comment can be anywhere in an object's definition, it starts at the first "#" character on a line and ends at the first end-of-line character. White space characters can be used to improve readability.

An integer can be specified using (1) the C programming language notation (e.g. 1, 12345); (2) sequence of four 1-octet integers (in the range from 0 to 255) separated by the character dot "." (e.g. 1.1.1.1, 255.255.0.0), in this case a 4-octet integer is formed by concatenating these 1-octet integers in the most significant to least significant order; (3) sequence of two 2-octet integers (in the range from 0 to 65535) separated by the character colon ":" (e.g. 3561:70, 3582:10), in this case a 4-octet integer is formed by concatenating these 2-octet integers in the most significant to least significant order.

3 Contact Information

The `mntner`, `person` and `role` classes, `admin-c`, `tech-c`, `mnt-by`, `changed`, and `source` attributes of all classes describe contact information. The `mntner` class also specifies authenticaiton information required to create, delete and update other objects. These classes do not specify routing policies and each registry may have different or additional requirements on them. Here we present the common denominator for completeness which is the RIPE database implementation [16]. Please consult your routing registry for the latest specification of these classes and attributes. The "Routing Policy System Security" document [20] describes the authenticaiton and authorization model in more detail.

3.1 `mntner` Class

The `mntner` class specifies authenticaiton information required to create, delete and update RPSL objects. A provider, before he/she can create RPSL objects, first needs to create a `mntner` object.

The attributes of the mntner class are shown in Figure 1. The mntner class was first described in [12].

Attribute	Value	Type
mntner	⟨*object-name*⟩	mandatory, single-valued, class key
descr	⟨*free-form*⟩	mandatory, single-valued
auth	see description in text	mandatory, multi-valued
upd-to	⟨*email-address*⟩	mandatory, multi-valued
mnt-nfy	⟨*email-address*⟩	optional, multi-valued
tech-c	⟨*nic-handle*⟩	mandatory, multi-valued
admin-c	⟨*nic-handle*⟩	optional, multi-valued
remarks	⟨*free-form*⟩	optional, multi-valued
notify	⟨*email-address*⟩	optional, multi-valued
mnt-by	list of ⟨mntner-*name*⟩	mandatory, multi-valued
changed	⟨*email-address*⟩ ⟨*date*⟩	mandatory, multi-valued
source	⟨*registry-name*⟩	mandatory, single-valued

Figure 1: mntner Class Attributes

The mntner attribute is mandatory and is the class key. Its value is an RPSL name. The auth attribute specifies the scheme that will be used to identify and authenticate update requests from this maintainer. It has the following syntax:

```
auth: <scheme-id> <auth-info>

E.g.
        auth: NONE
        auth: CRYPT-PW dhjsdfhruewf
        auth: MAIL-FROM .*@ripe\.net
```

The <scheme-id>'s currently defined are: NONE, MAIL-FROM, PGP-KEY and CRYPT-PW. The <auth-info> is additional information required by a particular scheme: in the case of MAIL-FROM, it is a regular expression matching valid email addresses; in the case of CRYPT-PW, it is a password in UNIX crypt format; and in the case of PGP-KEY, it is a pointer to key-certif object [22] containing the PGP public key of the user. If multiple auth attributes are specified, an update request satisfying *any* one of them is authenticated to be from the maintainer.

The upd-to attribute is an email address. On an unauthorized update attempt of an object maintained by this maintainer, an email message will be sent to this address. The mnt-nfy attribute is an email address. A notification message will be forwarded to this email address whenever an object maintained by this maintainer is added, changed or deleted.

The descr attribute is a short, free-form textual description of the object. The tech-c attribute is a technical contact NIC handle. This is someone to be contacted for technical problems such as misconfiguration. The admin-c attribute is an administrative contact NIC handle. The remarks attribute is a free text explanation or clarification. The notify attribute is an email address to which notifications of changes to this object should be sent. The mnt-by attribute is a list of mntner object names. The authorization for changes to this object is governed by any of the maintainer objects referenced. The changed attribute documents who last changed this object, and when this change was made. Its syntax has the following form:

```
changed: <email-address> <YYYYMMDD>

E.g.
changed: johndoe@terabit-labs.nn 19900401
```

The <email-address> identifies the person who made the last change. <YYYYMMDD> is the date of the change. The source attribute specifies the registry where the object is registered. Figure 2 shows an example mntner object. In the example, UNIX crypt format password authentication is used.

```
mntner:      RIPE-NCC-MNT
descr:       RIPE-NCC Maintainer
admin-c:     DK58
tech-c:      OPS4-RIPE
upd-to:      ops@ripe.net
mnt-nfy:     ops-fyi@ripe.net
auth:        CRYPT-PW lz1A7/JnfkTtI
mnt-by:      RIPE-NCC-MNT
changed:     ripe-dbm@ripe.net 19970820
source:      RIPE
```

Figure 2: An example mntner object.

The descr, tech-c, admin-c, remarks, notify, mnt-by, changed and source attributes are attributes of all RPSL classes. Their syntax, semantics, and mandatory, optional, multi-valued, or single-valued status are the same for for all RPSL classes. Only exception to this is the admin-c attribute which is mandatory for the aut-num class. We do not further discuss them in other sections.

3.2 person Class

A person class is used to describe information about people. Even though it does not describe routing policy, we still describe it here briefly since many policy objects make reference to person objects. The person class was first described in [15].

Attribute	Value	Type
`person`	⟨*free-form*⟩	mandatory, single-valued
`nic-hdl`	⟨*nic-handle*⟩	mandatory, single-valued, class key
`address`	⟨*free-form*⟩	mandatory, multi-valued
`phone`	see description in text	mandatory, multi-valued
`fax-no`	same as phone	optional, multi-valued
`e-mail`	⟨*email-address*⟩	mandatory, multi-valued

Figure 3: `person` Class Attributes

The attributes of the `person` class are shown in Figure 3. The `person` attribute is the full name of the person. The `phone` and the `fax-no` attributes have the following syntax:

```
   phone: +<country-code> <city> <subscriber> [ext. <extension>]

E.g.:
   phone: +31 20 12334676
   phone: +44 123 987654 ext. 4711
```

Figure 4 shows an example `person` object.

```
person:      Daniel Karrenberg
address:     RIPE Network Coordination Centre (NCC)
address:     Singel 258
address:     NL-1016 AB  Amsterdam
address:     Netherlands
phone:       +31 20 535 4444
fax-no:      +31 20 535 4445
e-mail:      Daniel.Karrenberg@ripe.net
nic-hdl:     DK58
changed:     Daniel.Karrenberg@ripe.net 19970616
source:      RIPE
```

Figure 4: An example `person` object.

3.3 `role` Class

The `role` class is similar to the person object. However, instead of describing a human being, it describes a role performed by one or more human beings. Examples include help desks, network monitoring centers, system administrators, etc. Role object is particularly useful since often a person performing a role may change, however the role itself remains.

Attribute	Value	Type
`role`	⟨*free-form*⟩	mandatory, single-valued
`nic-hdl`	⟨*nic-handle*⟩	mandatory, single-valued, class key
`trouble`	⟨*free-form*⟩	optional, multi-valued
`address`	⟨*free-form*⟩	mandatory, multi-valued
`phone`	see description in text	mandatory, multi-valued
`fax-no`	same as phone	optional, multi-valued
`e-mail`	⟨*email-address*⟩	mandatory, multi-valued

Figure 5: `role` Class Attributes

The attributes of the `role` class are shown in Figure 5. The `nic-hdl` attributes of the `person` and `role` classes share the same name space. The `trouble` attribute of `role` object may contain additional contact information to be used when a problem arises in any object that references this `role` object. Figure 6 shows an example `role` object.

```
role:        RIPE NCC Operations
trouble:
address:     Singel 258
address:     1016 AB Amsterdam
address:     The Netherlands
phone:       +31 20 535 4444
fax-no:      +31 20 545 4445
e-mail:      ops@ripe.net
admin-c:     CO19-RIPE
tech-c:      RW488-RIPE
tech-c:      JLSD1-RIPE
nic-hdl:     OPS4-RIPE
notify:      ops@ripe.net
changed:     roderik@ripe.net 19970926
source:      RIPE
```

Figure 6: An example `role` object.

4 route Class

Each interAS route (also referred to as an interdomain route) originated by an AS is specified using a `route` object. The attributes of the `route` class are shown in Figure 7. The `route` attribute is the address prefix of the route and the `origin` attribute is the AS number of the AS that originates the route into the interAS routing system. The `route` and `origin` attribute pair is the class key.

Attribute	Value	Type
`route`	⟨*address-prefix*⟩	mandatory, single-valued, class key
`origin`	⟨*as-number*⟩	mandatory, single-valued, class key
`withdrawn`	⟨*date*⟩	optional, single-valued
`member-of`	list of ⟨`route-set`-*names*⟩ see Section 5	optional, multi-valued
`inject`	see Section 8	optional, multi-valued
`components`	see Section 8	optional, single-valued
`aggr-bndry`	see Section 8	optional, single-valued
`aggr-mtd`	see Section 8	optional, single-valued
`export-comps`	see Section 8	optional, multi-valued
`holes`	see Section 8	optional, multi-valued

Figure 7: `route` Class Attributes

Figure 8 shows examples of four `route` objects (we do not include contact attributes such as `admin-c`, `tech-c` for brevity). Note that the last two `route` objects have the same address prefix, namely `128.8.0.0/16`. However, they are different route objects since they are originated by different ASes (i.e. they have different keys).

```
route: 128.9.0.0/16
origin: AS226

route: 128.99.0.0/16
origin: AS226

route: 128.8.0.0/16
origin: AS1

route: 128.8.0.0/16
origin: AS2
withdrawn: 19960624
```

Figure 8: Route Objects

The `withdrawn` attribute, if present, signifies that the originator AS no longer originates this address prefix in the Internet. Its value is a date indicating the date of withdrawal. In Figure 8, the last route object is withdrawn (i.e. no longer originated by `AS2`) on June 24, 1996.

5 Set Classes

To specify policies, it is often useful to define sets of objects. For this purpose we define two classes: `route-set` and `as-set`. These classes define a *named* set. The members of these sets can be specified by either explicitly listing them in the set object's definition, or implicitly by having `route` and `aut-num` objects refer to their names, or a combination of both methods.

5.1 `route-set` Class

The attributes of the `route-set` class are shown in Figure 9. The `route-set` attribute defines the name of the set. It is an RPSL name that starts with "`rs-`". The `members` attribute lists the members of the set. The `members` attribute is a list of address prefixes or other route-set names. Note that, the `route-set` class is a set of route prefixes, not of RPSL `route` objects.

Attribute	Value	Type
`route-set`	⟨*object-name*⟩	mandatory, single-valued, class key
`members`	list of ⟨*address-prefix-range*⟩ or ⟨`route-set`-*name*⟩ or ⟨`route-set`-*name*⟩⟨*range-operator*⟩	optional, multi-valued
`mbrs-by-ref`	list of ⟨`mntner`-*names*⟩	optional, multi-valued

Figure 9: `route-set` Class Attributes

Figure 10 presents some example `route-set` objects. The set `rs-foo` contains two address prefixes, namely `128.9.0.0/16` and `128.9.0.0/24`. The set `rs-bar` contains the members of the set `rs-foo` and the address prefix `128.7.0.0/16`. The set `rs-empty` contains no members.

An address prefix or a `route-set` name in a `members` attribute can be optionally followed by a range operator. For example, the following set

```
route-set: rs-bar
members: 5.0.0.0/8^+, 30.0.0.0/8^24-32, rs-foo^+
```

contains all the more specifics of `5.0.0.0/8` including `5.0.0.0/8`, all the more specifics of `30.0.0.0/8`

```
route-set: rs-foo
members: 128.9.0.0/16, 128.9.0.0/24

route-set: rs-bar
members: 128.7.0.0/16, rs-foo

route-set: rs-empty
```

Figure 10: `route-set` Objects

which are of length 24 to 32 such as `30.9.9.96/28`, and all the more specifics of address prefixes in route set `rs-foo`.

The `mbrs-by-ref` attribute is a list of maintainer names or the keyword `ANY`. If this attribute is used, the route set also includes address prefixes whose `route` objects are registered by one of these maintainers and whose `member-of` attribute refers to the name of this route set. If the value of a `mbrs-by-ref` attribute is `ANY`, any `route` object referring to the route set name is a member. If the `mbrs-by-ref` attribute is missing, only the address prefixes listed in the `members` attribute are members of the set.

```
route-set: rs-foo
mbrs-by-ref: MNTR-ME, MNTR-YOU

route-set: rs-bar
members: 128.7.0.0/16
mbrs-by-ref: MNTR-YOU

route: 128.9.0.0/16
origin: AS1
member-of: rs-foo
mnt-by: MNTR-ME

route: 128.8.0.0/16
origin: AS2
member-of: rs-foo, rs-bar
mnt-by: MNTR-YOU
```

Figure 11: `route-set` objects.

Figure 11 presents example `route-set` objects that use the `mbrs-by-ref` attribute. The set `rs-foo` contains two address prefixes, namely `128.8.0.0/16` and `128.9.0.0/16` since the `route` objects for `128.8.0.0/16` and `128.9.0.0/16` refer to the set name `rs-foo` in their `member-of` attribute. The set `rs-bar` contains the address prefixes `128.7.0.0/16` and `128.8.0.0/16`. The route `128.7.0.0/16` is explicitly listed in the `members` attribute of `rs-bar`, and the `route` object for `128.8.0.0/16` refer to the set name `rs-bar` in its `member-of` attribute.

Note that, if an address prefix is listed in a `members` attribute of a route set, it is a member of that route set. The `route` object corresponding to this address prefix does not need to contain a `member-of` attribute referring to this set name. The `member-of` attribute of the `route` class is an additional mechanism for specifying the members indirectly.

5.2 `as-set` Class

The attributes of the `as-set` class are shown in Figure 12. The `as-set` attribute defines the name of the set. It is an RPSL name that starts with "`as-`". The `members` attribute lists the members of the set. The `members` attribute is a list of AS numbers, or other `as-set` names.

Attribute	Value	Type
`as-set`	⟨*object-name*⟩	mandatory, single-valued, class key
`members`	list of ⟨*as-numbers*⟩ or ⟨`as-set`-*names*⟩	optional, multi-valued
`mbrs-by-ref`	list of ⟨`mntner`-*names*⟩	optional, multi-valued

Figure 12: `as-set` Class Attributes

Figure 13 presents two `as-set` objects. The set `as-foo` contains two ASes, namely `AS1` and `AS2`. The set `as-bar` contains the members of the set `as-foo` and `AS3`, that is it contains `AS1, AS2, AS3`.

```
as-set: as-foo
members: AS1, AS2
```

```
as-set: as-bar
members: AS3, as-foo
```

Figure 13: `as-set` objects.

The `mbrs-by-ref` attribute is a list of maintainer names or the keyword `ANY`. If this attribute is used, the AS set also includes ASes whose `aut-num` objects are registered by one of these maintainers and whose `member-of` attribute refers to the name of this AS set. If the value of a `mbrs-by-ref` attribute is `ANY`, any AS object referring to the AS set is a member of the set. If the `mbrs-by-ref` attribute is missing, only the ASes listed in the `members` attribute are members of the set.

Figure 14 presents an example `as-set` object that uses the `mbrs-by-ref` attribute. The set `as-foo` contains `AS1`, `AS2` and `AS3`. `AS4` is not a member of the set `as-foo` even though the `aut-num` object references `as-foo`. This is because `MNTR-OTHER` is not listed in the `as-foo`'s `mbrs-by-ref` attribute.

```
as-set: as-foo
members: AS1, AS2
mbrs-by-ref: MNTR-ME

aut-num: AS3                    aut-num: AS4
member-of: as-foo               member-of: as-foo
mnt-by: MNTR-ME                 mnt-by: MNTR-OTHER
```

Figure 14: `as-set` objects.

5.3 Predefined Set Objects

In a context that expects a route set (e.g. `members` attribute of the `route-set` class), an AS number ASx defines the set of routes that are originated by ASx; and an `as-set` AS-X defines the set of routes that are originated by the ASes in AS-X. A route p is said to be originated by ASx if there is a `route` object for p with ASx as the value of the `origin` attribute. For example, in Figure 15, the route set `rs-special` contains `128.9.0.0/16`, routes of `AS1` and `AS2`, and routes of the ASes in AS set `AS-FOO`.

```
route-set: rs-special
members: 128.9.0.0/16, AS1, AS2, AS-FOO
```

Figure 15: Use of AS numbers and AS sets in route sets.

The set `rs-any` contains all routes registered in IRR. The set `as-any` contains all ASes registered in IRR.

5.4 Hierarchical Set Names

Set names can be hierarchical. A hierarchical set name is a sequence of set names and AS numbers separated by colons ":". For example, the following names are valid: `AS1:AS-CUSTOMERS`, `AS1:RS-EXCEPTIONS`, `AS1:RS-EXPORT:AS2`, `RS-EXCEPTIONS:RS-BOGUS`. All components of an hierarchical set name which are not AS numbers should start with "`as-`" or "`rs-`" for as sets and route sets respectively. And at least one component must be a set name.

A set object with name `X1:...:Xn-1:Xn` can only be created by the maintainer of the object with name `X1:...:Xn-1`. That is, only the maintainer of `AS1` can create a set with name `AS1:AS-FOO`; and only the maintainer of `AS1:AS-FOO` can create a set with name `AS1:AS-FOO:AS-BAR`. Please see RPS Security Document [20] for details.

The purpose of an hierarchical set name is to partition the set name space so that the controllers of the set name X1 controls the whole set name space under X1, i.e. X1:...:Xn-1. This is important since anyone can create a set named AS-MCI-CUSTOMERS but only the people created AS3561 can create AS3561:AS-CUSTOMERS. In the former, it is not clear if the set AS-MCI-CUSTOMERS has any relationship with MCI. In the latter, we can guarantee that AS3561:AS-CUSTOMERS and AS3561 are created by the same entity.

6 aut-num Class

Routing policies are specified using the aut-num class. The attributes of the aut-num class are shown in Figure 16. The value of the aut-num attribute is the AS number of the AS described by this object. The as-name attribute is a symbolic name (in RPSL name syntax) of the AS. The import, export and default routing policies of the AS are specified using import, export and default attributes respectively.

Attribute	Value	Type
aut-num	⟨*as-number*⟩	mandatory, single-valued, class key
as-name	⟨*object-name*⟩	mandatory, single-valued
member-of	list of ⟨as-set-*names*⟩	optional, multi-valued
import	see Section 6.1	optional, multi valued
export	see Section 6.2	optional, multi valued
default	see Section 6.5	optional, multi valued

Figure 16: aut-num Class Attributes

6.1 import Attribute: Import Policy Specification

Figure 17 shows a typical interconnection of ASes that we will be using in our examples throughout this section. In this example topology, there are three ASes, AS1, AS2, and AS3; two exchange points, EX1 and EX2; and six routers. Routers connected to the same exchange point peer with each other and exchange routing information. Each router would export a subset of the routes it has to its peer routers. Peer routers would import a subset of these routes. A router while importing routes would set some route attributes. For example, AS1 can assign higher preference values to the routes it imports from AS2 so that it prefers AS2 over AS3. While exporting routes, a router may also set some route attributes in order to affect route selection of other ASes. For example, AS2 may set the MULTI-EXIT-DISCRIMINATOR BGP attribute so that AS1 prefers the routes through router 9.9.9.2. Most interAS policies are specified by specifying what route subsets can be imported or exported, and how the various BGP route attributes are set and used.

In RPSL, an import policy is divided into import policy expressions. Each import policy expression is specified using an import attribute. The import attribute has the following syntax (we will extend this syntax later in Sections 6.3 and 6.6):

```
----------------------                   ----------------------
|            7.7.7.1 |-------|   |-------| 7.7.7.2            |
|                    |     ========      |                    |
|   AS1              |      EX1  |-------| 7.7.7.3     AS2    |
|                    |                   |                    |
|            9.9.9.1 |------       ------| 9.9.9.2            |
----------------------     |       |     ----------------------
                          ===========
                              |    EX2
----------------------        |
|            9.9.9.3 |---------
|                    |
|   AS3              |
----------------------
```

Figure 17: Example topology consisting of three ASes, AS1, AS2, and AS3; two exchange points, EX1 and EX2; and six routers.

```
    import: from <peering-1> [action <action-1>]
            . . .
            from <peering-N> [action <action-N>]
            accept <filter>
```

The action specification is optional. The semantics of an import attribute is as follows: the set of routes that are matched by <filter> are imported from all the peers in <peerings>; while importing routes at <peering-M>, <action-M> is executed.

```
  E.g.
    aut-num: AS1
    import: from AS2 action pref = 1; accept { 128.9.0.0/16 }
```

This example states that the route 128.9.0.0/16 is accepted from AS2 with preference 1. In the next few subsections, we will describe how peerings, actions and filters are specified.

6.1.1 Peering Specification

Our example above used an AS number to specify peerings. The peerings can be specified at different granularities. The syntax of a peering specification has two forms. The first one is as follows:

```
<peer-as> [<peer-router>] [at <local-router>]
```

where `<local-router>` and `<peer-router>` are IP addresses of routers, `<peer-as>` is an AS number. `<peer-as>` must be the AS number of `<peer-router>`. Both `<local-router>` and `<peer-router>` are optional. If neither router is specified, this peering specification identifies all the peerings between the local AS and the `<peer-as>`. If both `<local-router>` and `<peer-router>` are specified, only that particular peering between these two routers are identified. If only the `<local-router>` is specified, all peerings between the `<local-router>` and any router in `<peer-as>` are identified. If only the `<peer-router>` is specified, all peerings from the routers in the local AS to the `<peer-router>` are identified.

We next give examples. Consider the topology of Figure 17 where `7.7.7.1`, `7.7.7.2` and `7.7.7.3` peer with each other; `9.9.9.1`, `9.9.9.2` and `9.9.9.3` peer with each other. In the following example `7.7.7.1` imports `128.9.0.0/16` from `7.7.7.2`.

```
(1) aut-num: AS1
    import: from AS2 7.7.7.2 at 7.7.7.1 accept { 128.9.0.0/16 }
```

In the following example `7.7.7.1` imports `128.9.0.0/16` from `7.7.7.2` and `7.7.7.3`.

```
(2) aut-num: AS1
    import: from AS2 at 7.7.7.1 accept { 128.9.0.0/16 }
```

In the following example `7.7.7.1` imports `128.9.0.0/16` from `7.7.7.2` and `7.7.7.3`, and `9.9.9.1` imports `128.9.0.0/16` from `9.9.9.2`.

```
(3) aut-num: AS1
    import: from AS2 accept { 128.9.0.0/16 }
```

The second form of `<peering>` specification has the following syntax:

```
<as-expression> [at <router-expression>]
```

where `<as-expression>` is an expression over AS numbers and sets using operators `AND`, `OR`, and `NOT`, and `<router-expression>` is an expression over router IP addresses and DNS names using operators `AND`, `OR`, and `NOT`. The DNS name can only be used if there is an `inet-rtr` object for that name that binds the name to IP addresses. This form identifies all the peerings between any local router in `<router-expression>` to any of their peer routers in the ASes in `<as-expression>`. If `<router-expression>` is not specified, it defaults to all routers of the local AS that peer with ASes in `<as-expression>`.

In the following example `9.9.9.1` imports `128.9.0.0/16` from `9.9.9.2` and `9.9.9.3`.

```
(4) as-set: AS-FOO
    members: AS2, AS3

    aut-num: AS1
    import: from AS-FOO      at 9.9.9.1 accept { 128.9.0.0/16 }
```

In the following example `9.9.9.1` imports `128.9.0.0/16` from `9.9.9.2` and `9.9.9.3`, and `7.7.7.1` imports `128.9.0.0/16` from `7.7.7.2` and `7.7.7.3`.

```
(5) aut-num: AS1
    import: from AS-FOO                    accept { 128.9.0.0/16 }
```

In the following example `AS1` imports `128.9.0.0/16` from `AS3` at router `9.9.9.1`

```
(6) aut-num: AS1
    import: from AS-FOO and not AS2
            at not 7.7.7.1
            accept { 128.9.0.0/16 }
```

This is because "`AS-FOO and not AS2`" equals `AS3` and "`not 7.7.7.1`" equals `9.9.9.1`.

6.1.2 Action Specification

Policy actions in RPSL either set or modify route attributes, such as assigning a preference to a route, adding a BGP community to the BGP community path attribute, or setting the `MULTI-EXIT-DISCRIMINATOR` attribute. Policy actions can also instruct routers to perform special operations, such as route flap damping.

The *routing policy attributes* whose values can be modified in policy actions are specified in the RPSL dictionary. Please refer to Section 7 for a list of these attributes. Each action in RPSL is terminated by the semicolon character (';'). It is possible to form composite policy actions by listing them one after the other. In a composite policy action, the actions are executed left to right. For example,

```
aut-num: AS1
import: from AS2
        action pref = 10; med = 0; community.append(10250, 3561:10);
        accept { 128.9.0.0/16 }
```

sets pref to 10, med to 0, and then appends 10250 and 3561:10 to the BGP community path attribute. The `pref` attribute is the inverse of the `local-pref` attribute (i.e. `local-pref == 65535 - pref`). A route with a `local-pref` attribute is always preferred over a route without one.

6.1.3 Filter Specification

A policy filter is a logical expression which when applied to a set of routes returns a subset of these routes. We say that the policy filter matches the subset returned. The policy filter can match routes using any path attribute, such as the destination address prefix (or NLRI), AS-path, or community attributes.

The policy filters can be composite by using the operators `AND`, `OR`, and `NOT`. The following policy filters can be used to select a subset of routes:

`ANY` The filter-keyword `ANY` matches all routes.

Address-Prefix Set This is an explicit list of address prefixes enclosed in braces '{' and '}'. The policy filter matches the set of routes whose destination address-prefix is in the set. For example:

```
{ 0.0.0.0/0 }
{ 128.9.0.0/16, 128.8.0.0/16, 128.7.128.0/17, 5.0.0.0/8 }
{ }
```

An address prefix can be optionally followed by a range operator (i.e. '`^-`', '`^+`', '`^n`', or '`^n-m`'). For example, the set

```
{ 5.0.0.0/8^+, 128.9.0.0/16^-, 30.0.0.0/8^16, 30.0.0.0/8^24-32 }
```

contains all the more specifics of `5.0.0.0/8` including `5.0.0.0/8`, all the more specifics of `128.9.0.0/16` excluding `128.9.0.0/16`, all the more specifics of `30.0.0.0/8` which are of length 16 such as `30.9.0.0/16`, and all the more specifics of `30.0.0.0/8` which are of length 24 to 32 such as `30.9.9.96/28`.

Route Set Name A route set name matches the set of routes that are members of the set. A route set name may be a name of a `route-set` object, an AS number, or a name of an `as-set` object (AS numbers and `as-set` names implicitly define route sets; please see Section 5.3). For example:

```
aut-num: AS1
import: from AS2 action pref = 1; accept AS2
import: from AS2 action pref = 1; accept AS-FOO
import: from AS2 action pref = 1; accept RS-FOO
```

The keyword `PeerAS` can be used instead of the AS number of the peer AS. `PeerAS` is particularly useful when the peering is specified using an AS expression. For example:

```
as-set: AS-FOO
members: AS2, AS3

aut-num: AS1
import: from AS-FOO action pref = 1; accept PeerAS
```

is same as:

```
aut-num: AS1
import: from AS2 action pref = 1; accept AS2
import: from AS3 action pref = 1; accept AS3
```

A route set name can also be followed by one of the operators '`^-`', '`^+`', '`^n`' or '`^n-m`'. These operators are distributive over the route sets. For example, { `5.0.0.0/8, 6.0.0.0/8` }`^+` equals { `5.0.0.0/8^+, 6.0.0.0/8^+` }, and `AS1^-` equals all the exclusive more specifics of routes originated by `AS1`.

AS Path Regular Expressions An AS-path regular expression can be used as a policy filter by enclosing the expression in '`<`' and '`>`'. An AS-path policy filter matches the set of routes which traverses a sequence of ASes matched by the AS-path regular expression. A router can check this using the `AS_PATH` attribute in the Border Gateway Protocol [19], or the `RD_PATH` attribute in the Inter-Domain Routing Protocol[18].

AS-path Regular Expressions are POSIX compliant regular expressions over the alphabet of AS numbers. The regular expression constructs are as follows:

`ASN` where `ASN` is an AS number. `ASN` matches the AS-path that is of length 1 and contains the corresponding AS number (e.g. AS-path regular expression AS1 matches the AS-path "1").

The keyword `PeerAS` can be used instead of the AS number of the peer AS.

`AS-set` where `AS-set` is an AS set name. `AS-set` matches the AS-paths that is matched by one of the ASes in the `AS-set`.

`.` matches the AS-paths matched by any AS number.

`[...]` is an AS number set. It matches the AS-paths matched by the AS numbers listed between the brackets. The AS numbers in the set are separated by white space characters. If a '-' is used between two AS numbers in this set, all AS numbers between the two AS numbers are included in the set. If an `as-set` name is listed, all AS numbers in the `as-set` are included.

`[^...]` is a complemented AS number set. It matches any AS-path which is not matched by the AS numbers in the set.

`^` Matches the empty string at the beginning of an AS-path.

`$` Matches the empty string at the end of an AS-path.

We next list the regular expression operators in the decreasing order of evaluation. These operators are left associative, i.e. performed left to right.

Unary postfix operators `* + ? {m} {m,n} {m,}`
For a regular expression `A`, `A*` matches zero or more occurrences of `A`; `A+` matches one or more occurrences of `A`; `A?` matches zero or one occurrence of `A`; `A{m}` matches `m` occurrence of `A`; `A{m,n}` matches `m` to `n` occurrence of `A`; `A{m,}` matches `m` or more occurrence of `A`. For example, `[AS1 AS2]{2}` matches `AS1 AS1`, `AS1 AS2`, `AS2 AS1`, and `AS2 AS2`.

Unary postfix operators `~* ~+ ~{m} ~{m,n} ~{m,}`
These operators have similar functionality as the corresponding operators listed above, but all occurrences of the regular expression has to match the same pattern. For example, `[AS1 AS2]~{2}` matches `AS1 AS1` and `AS2 AS2`, but it does not match `AS1 AS2` and `AS2 AS1`.

Binary catenation operator
This is an implicit operator and exists between two regular expressions `A` and `B` when no other explicit operator is specified. The resulting expression `A B` matches an AS-path if `A` matches some prefix of the AS-path and `B` matches the rest of the AS-path.

Binary alternative (or) operator `|`
For a regular expressions `A` and `B`, `A | B` matches any AS-path that is matched by `A` or `B`.

Parenthesis can be used to override the default order of evaluation. White spaces can be used to increase readability.

The following are examples of AS-path filters:

```
<AS3>
<^AS1>
<AS2$>
<^AS1 AS2 AS3$>
<^AS1 .* AS2$>.
```

The first example matches any route whose AS-path contains `AS3`, the second matches routes whose AS-path starts with `AS1`, the third matches routes whose AS-path ends with `AS2`, the fourth matches routes whose AS-path is exactly "`1 2 3`", and the fifth matches routes whose AS-path starts with `AS1` and ends in `AS2` with any number of AS numbers in between.

Composite Policy Filters The following operators (in decreasing order of evaluation) can be used to form composite policy filters:

`NOT` Given a policy filter `x`, `NOT x` matches the set of routes that are not matched by `x`. That is it is the negation of policy filter `x`.

`AND` Given two policy filters `x` and `y`, `x AND y` matches the *intersection* of the routes that are matched by `x` and that are matched by `y`.

`OR` Given two policy filters `x` and `y`, `x OR y` matches the *union* of the routes that are matched by `x` and that are matched by `y`.

Note that an `OR` operator can be implicit, that is '`x y`' is equivalent to '`x OR y`'.

```
E.g.
  NOT {128.9.0.0/16, 128.8.0.0/16}
  AS226 AS227 OR AS228
  AS226 AND NOT {128.9.0.0/16}
  AS226 AND {0.0.0.0/0^0-18}
```

The first example matches any route except `128.9.0.0/16` and `128.8.0.0/16`. The second example matches the routes of `AS226`, `AS227` and `AS228`. The third example matches the routes of `AS226` except `128.9.0.0/16`. The fourth example matches the routes of `AS226` whose length are not longer than 18.

Routing Policy Attributes Policy filters can also use the values of other attributes for comparison. The attributes whose values can be used in policy filters are specified in the RPSL dictionary. Please refer to Section 7 for details. An example using the the BGP community attribute is shown below:

```
aut-num: AS1
export: to AS2 announce AS1 AND NOT community.contains(NO_EXPORT)
```

Filters using the routing policy attributes defined in the dictionary are evaluated before evaluating the operators `AND`, `OR` and `NOT`.

6.1.4 Example Policy Expressions

```
aut-num: AS1
import: from AS2 action pref = 1;
        from AS3 action pref = 2;
        accept AS4
```

The above example states that AS4's routes are accepted from AS2 with preference 1, and from AS3 with preference 2 (routes with lower integer preference values are preferred over routes with higher integer preference values).

```
aut-num: AS1
import: from AS2 7.7.7.2 at 7.7.7.1 action pref = 1;
        from AS2                    action pref = 2;
        accept AS4
```

The above example states that AS4's routes are accepted from AS2 on peering 7.7.7.1-7.7.7.2 with preference 1, and on any other peering with AS2 with preference 2.

6.2 export Attribute: Export Policy Specification

Similarly, an export policy expression is specified using an export attribute. The export attribute has the following syntax:

```
export: to <peering-1> [action <action-1>]
        . . .
        to <peering-N> [action <action-N>]
        announce <filter>
```

The action specification is optional. The semantics of an export attribute is as follows: the set of routes that are matched by <filter> are exported to all the peers specified in <peerings>; while exporting routes at <peering-M>, <action-M> is executed.

```
E.g.
  aut-num: AS1
  export: to AS2 action med = 5; community .= { 70 };
          announce AS4
```

In this example, AS4's routes are announced to AS2 with the med attribute's value set to 5 and community 70 added to the community list.

Example:

```
aut-num: AS1
export: to AS-FOO announce ANY
```

In this example, AS1 announces all of its routes to the ASes in the set AS-FOO.

6.3 Other Routing Protocols, Multi-Protocol Routing Protocols, and Injecting Routes Between Protocols

The more complete syntax of the import and export attributes are as follows:

```
import: [protocol <protocol-1>] [into <protocol-2>]
        from <peering-1> [action <action-1>]
        . . .
        from <peering-N> [action <action-N>]
        accept <filter>
export: [protocol <protocol-1>] [into <protocol-2>]
        to <peering-1> [action <action-1>]
        . . .
        to <peering-N> [action <action-N>]
        announce <filter>
```

Where the optional protocol specifications can be used for specifying policies for other routing protocols, or for injecting routes of one protocol into another protocol, or for multi-protocol routing policies. The valid protocol names are defined in the dictionary. The <protocol-1> is the name of the protocol whose routes are being exchanged. The <protocol-2> is the name of the protocol which is receiving these routes. Both <protocol-1> and <protocol-2> default to the Internet Exterior Gateway Protocol, currently BGP.

In the following example, all interAS routes are injected into RIP.

```
aut-num: AS1
import: from AS2 accept AS2
export: protocol BGP4 into RIP
        to AS1 announce ANY
```

In the following example, AS1 accepts AS2's routes including any more specifics of AS2's routes, but does not inject these extra more specific routes into OSPF.

```
aut-num: AS1
```

```
import: from AS2 accept AS2^+
export: protocol BGP4 into OSPF
        to AS1 announce AS2
```

In the following example, AS1 injects its static routes (routes which are members of the set AS1:RS-STATIC-ROUTES) to the interAS routing protocol and appends AS1 twice to their AS paths.

```
aut-num: AS1
import: protocol STATIC into BGP4
        from AS1 action aspath.prepend(AS1, AS1);
        accept AS1:RS-STATIC-ROUTES
```

In the following example, AS1 imports different set of unicast routes for multicast reverse path forwarding from AS2:

```
aut-num: AS1
import: from AS2 accept AS2
import: protocol IDMR
        from AS2 accept AS2:RS-RPF-ROUTES
```

6.4 Ambiguity Resolution

It is possible that the same peering can be covered by more that one peering specification in a policy expression. For example:

```
aut-num: AS1
import: from AS2 7.7.7.2 at 7.7.7.1 action pref = 2;
        from AS2 7.7.7.2 at 7.7.7.1 action pref = 1;
        accept AS4
```

This is not an error, though definitely not desirable. To break the ambiguity, the action corresponding to the first peering specification is used. That is the routes are accepted with preference 2. We call this rule as the specification-order rule.

Consider the example:

```
aut-num: AS1
import: from AS2                      action pref = 2;
        from AS2 7.7.7.2 at 7.7.7.1 action pref = 1; dpa = 5;
        accept AS4
```

where both peering specifications cover the peering 7.7.7.1-7.7.7.2, though the second one covers it more specifically. The specification order rule still applies, and only the action "pref = 2" is executed. In fact, the second peering-action pair has no use since the first peering-action pair always covers it. If the intended policy was to accept these routes with preference 1 on this particular peering and with preference 2 in all other peerings, the user should have specified:

```
aut-num: AS1
import: from AS2 7.7.7.2 at 7.7.7.1 action pref = 1; dpa = 5;
        from AS2                    action pref = 2;
        accept AS4
```

It is also possible that more than one policy expression can cover the same set of routes for the same peering. For example:

```
aut-num: AS1
import: from AS2 action pref = 2; accept AS4
import: from AS2 action pref = 1; accept AS4
```

In this case, the specification-order rule is still used. That is, AS4's routes are accepted from AS2 with preference 2. If the filters were overlapping but not exactly the same:

```
aut-num: AS1
import: from AS2 action pref = 2; accept AS4
import: from AS2 action pref = 1; accept AS4 OR AS5
```

the AS4's routes are accepted from AS2 with preference 2 and however AS5's routes are also accepted, but with preference 1.

We next give the general specification order rule for the benefit of the RPSL implementors. Consider two policy expressions:

```
aut-num: AS1
import: from peerings-1 action action-1 accept filter-1
import: from peerings-2 action action-2 accept filter-2
```

The above policy expressions are equivalent to the following three expressions where there is no ambiguity:

```
aut-num: AS1
import: from peerings-1 action action-1 accept filter-1
import: from peerings-3 action action-2 accept filter-2 AND NOT filter-1
import: from peerings-4 action action-2 accept filter-2
```

where `peerings-3` are those that are covered by both `peerings-1` and `peerings-2`, and `peerings-4` are those that are covered by `peerings-2` but not by `peerings-1` ("`filter-2 AND NOT filter-1`" matches the routes that are matched by `filter-2` but not by `filter-1`).

Example:

```
aut-num: AS1
import: from AS2 7.7.7.2 at 7.7.7.1
        action pref = 2;
        accept {128.9.0.0/16}
import: from AS2
        action pref = 1;
        accept {128.9.0.0/16, 75.0.0.0/8}
```

Lets consider two peerings with `AS2`, `7.7.7.1-7.7.7.2` and `9.9.9.1-9.9.9.2`. Both policy expressions cover `7.7.7.1-7.7.7.2`. On this peering, the route `128.9.0.0/16` is accepted with preference 2, and the route `75.0.0.0/8` is accepted with preference 1. The peering `9.9.9.1-9.9.9.2` is only covered by the second policy expressions. Hence, both the route `128.9.0.0/16` and the route `75.0.0.0/8` are accepted with preference 1 on peering `9.9.9.1-9.9.9.2`.

Note that the same ambiguity resolution rules also apply to `export` and `default` policy expressions.

6.5 `default` Attribute: Default Policy Specification

Default routing policies are specified using the `default` attribute. The `default` attribute has the following syntax:

```
default: to <peering> [action <action>] [networks <filter>]
```

The `<action>` and `<filter>` specifications are optional. The semantics are as follows: The `<peering>` specification indicates the AS (and the router if present) is being defaulted to; the `<action>` specification, if present, indicates various attributes of defaulting, for example a relative preference if multiple defaults are specified; and the `<filter>` specifications, if present, is a policy filter. A router chooses a default router from the routes in its routing table that matches this `<filter>`.

In the following example, `AS1` defaults to `AS2` for routing.

```
aut-num: AS1
default: to AS2
```

In the following example, router `7.7.7.1` in `AS1` defaults to router `7.7.7.2` in `AS2`.

```
aut-num: AS1
default: to AS2 7.7.7.2 at 7.7.7.1
```

In the following example, `AS1` defaults to `AS2` and `AS3`, but prefers `AS2` over `AS3`.

```
aut-num: AS1
default: to AS2 action pref = 1;
default: to AS3 action pref = 2;
```

In the following example, `AS1` defaults to `AS2` and uses `128.9.0.0/16` as the default network.

```
aut-num: AS1
default: to AS2 networks { 128.9.0.0/16 }
```

6.6 Structured Policy Specification

The `import` and `export` policies can be structured. We only reccomend structured policies to advanced RPSL users. Please feel free to skip this section.

The syntax for a structured policy specification is the following:

```
<import-factor> ::= from <peering-1> [action <action-1>]
                    . . .
                    from <peering-N> [action <action-N>]
                    accept <filter>;

<import-term> ::=  <import-factor> |
                   LEFT-BRACE
                   <import-factor>
                   . . .
                   <import-factor>
                   RIGHT-BRACE

<import-expression> ::= <import-term>                              |
                        <import-term> EXCEPT <import-expression> |
                        <import-term> REFINE <import-expression>

import: [protocol <protocol1>] [into <protocol2>]
        <import-expression>
```

Please note the semicolon at the end of an `<import-factor>`. If the policy specification is not structured (as in all the examples in other sections), this semicolon is optional. The syntax and semantics for an `<import-factor>` is already defined in Section 6.1.

An <import-term> is either a sequence of <import-factor>'s enclosed within matching braces (i.e. '{' and '}') or just a single <import-factor>. The semantics of an <import-term> is the union of <import-factor>'s using the specification order rule. An <import-expression> is either a single <import-term> or an <import-term> followed by one of the keywords "except" and "refine", followed by another <import-expression>. Note that our definition allows nested expressions. Hence there can be exceptions to exceptions, refinements to refinements, or even refinements to exceptions, and so on.

The semantics for the except operator is as follows: The result of an except operation is another <import-term>. The resulting policy set contains the policies of the right hand side but their filters are modified to only include the routes also matched by the left hand side. The policies of the left hand side are included afterwards and their filters are modified to exclude the routes matched by the right hand side. Please note that the filters are modified during this process but the actions are copied verbatim. When there are multiple levels of nesting, the operations (both except and refine) are performed right to left.

Consider the following example:

```
import: from AS1 action pref = 1; accept as-foo;
        except {
           from AS2 action pref = 2; accept AS226;
           except {
              from AS3 action pref = 3; accept {128.9.0.0/16};
           }
        }
```

where the route 128.9.0.0/16 is originated by AS226, and AS226 is a member of the as set as-foo. In this example, the route 128.9.0.0/16 is accepted from AS3, any other route (not 128.9.0.0/16) originated by AS226 is accepted from AS2, and any other ASes' routes in as-foo is accepted from AS1.

We can come to the same conclusion using the algebra defined above. Consider the inner exception specification:

```
from AS2 action pref = 2; accept AS226;
except {
   from AS3 action pref = 3; accept {128.9.0.0/16};
}
```

is equivalent to

```
{
 from AS3 action pref = 3; accept AS226 AND {128.9.0.0/16};
```

```
   from AS2 action pref = 2; accept AS226 AND NOT {128.9.0.0/16};
  }
```

Hence, the original expression is equivalent to:

```
 import: from AS1 action pref = 1; accept as-foo;
         except {
            from AS3 action pref = 3; accept AS226 AND {128.9.0.0/16};
            from AS2 action pref = 2; accept AS226 AND NOT {128.9.0.0/16};
         }
```

which is equivalent to

```
 import: {
    from AS3 action pref = 3;
             accept as-foo AND AS226 AND {128.9.0.0/16};
    from AS2 action pref = 2;
             accept as-foo AND AS226 AND NOT {128.9.0.0/16};
    from AS1 action pref = 1;
             accept as-foo AND NOT
               (AS226 AND NOT {128.9.0.0/16} OR AS226 AND {128.9.0.0/16});
    }
```

Since `AS226` is in `as-foo` and `128.9.0.0/16` is in `AS226`, it simplifies to:

```
 import: {
           from AS3 action pref = 3; accept {128.9.0.0/16};
           from AS2 action pref = 2; accept AS226 AND NOT {128.9.0.0/16};
           from AS1 action pref = 1; accept as-foo AND NOT AS226;
         }
```

In the case of the `refine` operator, the resulting set is constructed by taking the cartasian product of the two sides as follows: for each policy l in the left hand side and for each policy r in the right hand side, the peerings of the resulting policy are the peerings common to both r and l; the filter of the resulting policy is the intersection of l's filter and r's filter; and action of the resulting policy is l's action followed by r's action. If there are no common peerings, or if the intersection of filters is empty, a resulting policy is not generated.

Consider the following example:

```
 import: { from AS-ANY action pref = 1; accept community.contains({3560,10});
```

```
      from AS-ANY action pref = 2; accept community.contains({3560,20});
    } refine {
       from AS1 accept AS1;
       from AS2 accept AS2;
       from AS3 accept AS3;
    }
```

Here, any route with community {3560,10} is assigned a preference of 1 and any route with community {3560,20} is assigned a preference of 2 regardless of whom they are imported from. However, only AS1's routes are imported from AS1, and only AS2's routes are imported from AS2, and only AS3's routes are imported form AS3, and no routes are imported from any other AS. We can reach the same conclusion using the above algebra. That is, our example is equivalent to:

```
import: {
  from AS1 action pref = 1; accept community.contains({3560,10}) AND AS1;
  from AS1 action pref = 2; accept community.contains({3560,20}) AND AS1;
  from AS2 action pref = 1; accept community.contains({3560,10}) AND AS2;
  from AS2 action pref = 2; accept community.contains({3560,20}) AND AS2;
  from AS3 action pref = 1; accept community.contains({3560,10}) AND AS3;
  from AS3 action pref = 2; accept community.contains({3560,20}) AND AS3;
}
```

Note that the common peerings between "`from AS1`" and "`from AS-ANY`" are those peerings in "`from AS1`". Even though we do not formally define "common peerings", it is straight forward to deduce the definition from the definitions of peerings (please see Section 6.1.1).

Consider the following example:

```
import: {
  from AS-ANY action med = 0; accept {0.0.0.0/0^0-18};
  } refine {
       from AS1 at 7.7.7.1 action pref = 1; accept AS1;
       from AS1            action pref = 2; accept AS1;
    }
```

where only routes of length 0 to 18 are accepted and med's value is set to 0 to disable med's effect for all peerings; In addition, from AS1 only AS1's routes are imported, and AS1's routes imported at 7.7.7.1 are preferred over other peerings. This is equivalent to:

```
import: {
  from AS1 at 7.7.7.1 action med=0; pref=1; accept {0.0.0.0/0^0-18} AND AS1;
  from AS1            action med=0; pref=2; accept {0.0.0.0/0^0-18} AND AS1;
}
```

The above syntax and semantics also apply equally to structured `export` policies with "`from`" replaced with "`to`" and "`accept`" is replaced with "`announce`".

7 `dictionary` Class

The dictionary class provides extensibility to RPSL. Dictionary objects define *routing policy attributes*, *types*, and *routing protocols*. Routing policy attributes, henceforth called rp-attributes, may correspond to actual protocol attributes, such as the BGP path attributes (e.g. `community`, `dpa`, and `AS-path`), or they may correspond to router features (e.g. BGP route flap damping). As new protocols, new protocol attributes, or new router features are introduced, the dictionary object is updated to include appropriate rp-attribute and protocol definitions.

An rp-attribute is an abstract class; that is a data representation is not available. Instead, they are accessed through access methods. For example, the rp-attribute for the `BGP AS-path` attribute is called `aspath`; and it has an access method called `prepend` which stuffs extra AS numbers to the `AS-path` attributes. Access methods can take arguments. Arguments are strongly typed. For example, the method `prepend` above takes AS numbers as arguments.

Once an rp-attribute is defined in the dictionary, it can be used to describe policy filters and actions. Policy analysis tools are required to fetch the dictionary object and recognize newly defined rp-attributes, types, and protocols. The analysis tools may approximate policy analyses on rp-attributes that they do not understand: a filter method may always match, and an action method may always perform no-operation. Analysis tools may even download code to perform appropriate operations using mechanisms outside the scope of RPSL.

We next describe the syntax and semantics of the dictionary class. This description is not essential for understanding dictionary objects (but it is essential for creating one). Please feel free to skip to the RPSL Initial Dictionary subsection (Section 7.1).

The attributes of the dictionary class are shown in Figure 18. The `dictionary` attribute is the name of the dictionary object, obeying the RPSL naming rules. There can be many dictionary objects, however there is always one well-known dictionary object "`RPSL`". All tools use this dictionary by default.

Attribute	Value	Type
`dictionary`	⟨*object-name*⟩	mandatory, single-valued, class key
`rp-attribute`	see description in text	optional, multi valued
`typedef`	see description in text	optional, multi valued
`protocol`	see description in text	optional, multi valued

Figure 18: `dictionary` Class Attributes

The `rp-attribute` attribute has the following syntax:

```
rp-attribute: <name>
   <method-1>(<type-1-1>, ..., <type-1-N1> [, "..."])
   ...
   <method-M>(<type-M-1>, ..., <type-M-NM> [, "..."])
```

where `<name>` is the name of the rp-attribute; and `<method-i>` is the name of an access method for the rp-attribute, taking `Ni` arguments where the `j-th` argument is of type `<type-i-j>`. A `method` name is either an RPSL name or one of the operators defined in Figure 19. The operator methods with the exception of `operator()` and `operator[]` can take only one argument.

```
operator=           operator==
operator<<=         operator<
operator>>=         operator>
operator+=          operator>=
operator-=          operator<=
operator*=          operator!=
operator/=          operator()
operator.=          operator[]
```

Figure 19: Operators

An rp-attribute can have many methods defined for it. Some of the methods may even have the same name, in which case their arguments are of different types. If the argument list is followed by "`...`", the method takes a variable number of arguments. In this case, the actual arguments after the Nth argument are of type `<type-N>`.

Arguments are strongly typed. A `<type>` in RPSL is either a *predefined type*, a *union type*, a *list type*, or a *dictionary defined type*. The predefined types are listed in Figure 20.

```
integer[lower, upper]              as_number
real[lower, upper]                 ipv4_address
enum[name, name, ...]              address_prefix
string                             address_prefix_range
boolean                            dns_name
rpsl_word                          filter
free_text                          as_set_name
email                              route_set_name
```

Figure 20: Predefined Types

The `integer` and the `real` predefined types can be followed by a lower and an upper bound to specify the set of valid values of the argument. The range specification is optional. We use the ANSI C language conventions for representing `integer`, `real` and `string` values. The `enum` type is followed by a list of RPSL names which are the valid values of the type. The `boolean` type can take the values `true` or `false`. `as_number`, `ipv4_address`, `address_prefix` and `dns_name` types are as in Section 2. `filter` type is a policy filter as in Section 6.

The syntax of a union type is as follows:

```
union <type-1>, ... , <type-N>
```

where `<type-i>` is an RPSL type. The union type is either of the types `<type-1>` through `<type-N>` (analogous to unions in C[13]).

The syntax of a list type is as follows:

```
list [<min_elems>:<max_elems>] of <type>
```

In this case, the list elements are of `<type>` and the list contains at least `<min_elems>` and at most `<max_elems>` elements. The size specification is optional. If it is not specified, there is no restriction in the number of list elements. A value of a list type is represented as a sequence of elements separated by the character "," and enclosed by the characters "{" and "}".

The `typedef` attribute in the dictionary defines named types as follows:

```
typedef: <name> <type>
```

where `<name>` is a name for type `<type>`. `typedef` attribute is paticularly useful when the type defined is not a predefined type (e.g. list of unions, list of lists, etc.).

A `protocol` attribute of the `dictionary` class defines a protocol and a set of peering parameters for that protocol (which are used in `inet-rtr` class in Section 9). Its syntax is as follows:

```
protocol: <name>
   MANDATORY | OPTIONAL <parameter-1>(<type-1-1>, ..., <type-1-N1> [, "..."])
   ...
   MANDATORY | OPTIONAL <parameter-M>(<type-M-1>, ..., <type-M-NM> [, "..."])
```

where `<name>` is the name of the protocol; `MANDATORY` and `OPTIONAL` are keywords; and `<parameter-i>` is a peering parameter for this protocol, taking `Ni` many arguments. The syntax and semantics of the arguments are as in the `rp-attribute`. If the keyword `MANDATORY` is used, the parameter is mandatory and needs to be specified for each peering of this protocol. If the keyword `OPTIONAL` is used, the parameter can be skipped.

7.1 Initial RPSL Dictionary and Example Policy Actions and Filters

```
dictionary:    RPSL
rp-attribute: # preference, smaller values represent higher preferences
               pref
               operator=(integer[0, 65535])
rp-attribute: # BGP multi_exit_discriminator attribute
               med
               # to set med to 10: med = 10;
               # to set med to the IGP metric: med = igp_cost;
               operator=(union integer[0, 65535], enum[igp_cost])
rp-attribute: # BGP destination preference attribute (dpa)
               dpa
               operator=(integer[0, 65535])
rp-attribute: # BGP aspath attribute
               aspath
               # prepends AS numbers from last to first order
               prepend(as_number, ...)
typedef:       # a community value in RPSL is either
               #  - a 4 byte integer (ok to use 3561:70 notation)
               #  - internet, no_export, no_advertise (see RFC-1997)
               community_elm union
                   integer[1, 4294967200],
                   enum[internet, no_export, no_advertise],
typedef:       # list of community values { 40, no_export, 3561:70 }
               community_list list of community_elm
rp-attribute: # BGP community attribute
               community
               # set to a list of communities
               operator=(community_list)
               # append community values
               operator.=(community_list)
               append(community_elm, ...)
               # delete community values
               delete(community_elm, ...)
               # a filter: true if one of community values is contained
               contains(community_elm, ...)
               # shortcut to contains: community(no_export, 3561:70)
               operator()(community_elm, ...)
               # order independent equality comparison
               operator==(community_list)
rp-attribute: # next hop router in a static route
               next-hop
               # to set to 7.7.7.7: next-hop = 7.7.7.7;
```

```
              # to set to router's own address: next-hop = self;
              operator=(union ipv4_address, enum[self])
rp-attribute: # cost of a static route
              cost
              operator=(integer[0, 65535])
protocol: BGP4
          # as number of the peer router
          MANDATORY asno(as_number)
          # enable flap damping
          OPTIONAL flap_damp()
          OPTIONAL flap_damp(integer[0,65535],# penalty per flap
                             integer[0,65535],# penalty value for supression
                             integer[0,65535],# penalty value for reuse
                             integer[0,65535],# halflife in secs when up
                             integer[0,65535],# halflife in secs when down
                             integer[0,65535])# maximum penalty
protocol: OSPF
protocol: RIP
protocol: IGRP
protocol: IS-IS
protocol: STATIC
protocol: RIPng
protocol: DVMRP
protocol: PIM-DM
protocol: PIM-SM
protocol: CBT
protocol: MOSPF
```

Figure 21: RPSL Dictionary

Figure 21 shows the initial `RPSL` dictionary. It has seven rp-attributes: `pref` to assign local preference to the routes accepted; `med` to assign a value to the `MULTI_EXIT_DISCRIMINATOR` BGP attribute; `dpa` to assign a value to the `DPA` BGP attribute; `aspath` to prepend a value to the `AS_PATH` BGP attribute; `community` to assign a value to or to check the value of the `community` BGP attribute; `next-hop` to assign next hop routers to static routes; and `cost` to assign a cost to static routes. The dictionary defines two types: `community_elm` and `community_list`. `community_elm` type is either a 4-byte unsigned integer, or one of the keywords `internet`, `no_export` or `no_advertise` (defined in [8]). An integer can be specified using two 2-byte integers seperated by ":" to partition the community number space so that a provider can use its AS number as the first two bytes, and assigns a semantics of its choice to the last two bytes.

The initial dictionary (Figure 21) defines only options for the Border Gateway Protocol: `asno` and `flap_damp`. The mandatory `asno` option is the AS number of the peer router. The optional `flap_damp` option instructs the router to damp route flaps[21] when importing routes from the peer router. It can be specified with or without parameters. If parameters are missing, they default to:

```
flap_damp(1000, 2000, 750, 900, 900, 20000)
```

That is, a penalty of 1000 is assigned at each route flap, the route is suppressed when penalty reaches 2000. The penalty is reduced in half after 15 minutes (900 seconds) of stability regardless of whether the route is up or down. A supressed route is reused when the penalty falls below 750. The maximum penalty a route can be assigned is 20,000 (i.e. the maximum suppress time after a route becomes stable is about 75 minutes). These parameters are consistent with the default flap damping parameters in several routers.

Policy Actions and Filters Using RP-Attributes

The syntax of a policy action or a filter using an rp-attribute `x` is as follows:

```
x.method(arguments)
x ''op'' argument
```

where `method` is a method and `''op''` is an operator method of the rp-attribute `x`. If an operator method is used in specifying a composite policy filter, it evaluates earlier than the composite policy filter operators (i.e. `AND`, `OR`, `NOT`, and implicit or operator).

The `pref` rp-attribute can be assigned a positive integer as follows:

```
pref = 10;
```

The `med` rp-attribute can be assigned either a positive integer or the word "`igp_cost`" as follows:

```
med = 0;
med = igp_cost;
```

The `dpa` rp-attribute can be assigned a positive integer as follows:

```
dpa = 100;
```

The BGP `community` attribute is list-valued, that is it is a list of 4-byte integers each representing a "community". The following examples demonstrate how to add communities to this rp-attribute:

```
community .= { 100 };
community .= { NO_EXPORT };
community .= { 3561:10 };
```

In the last case, a 4-byte integer is constructed where the more significant two bytes equal 3561 and the less significant two bytes equal 10. The following examples demonstrate how to delete communities from the `community` rp-attribute:

```
community.delete(100, NO_EXPORT, 3561:10);
```

Filters that use the `community` rp-attribute can be defined as demonstrated by the following examples:

```
community.contains(100, NO_EXPORT, 3561:10);
community(100, NO_EXPORT, 3561:10);              # shortcut
```

The community rp-attribute can be set to a list of communities as follows:

```
community = {100, NO_EXPORT, 3561:10, 200};
community = {};
```

In this first case, the community rp-attribute contains the communities 100, `NO_EXPORT`, 3561:10, and 200. In the latter case, the community rp-attribute is cleared. The community rp-attribute can be compared against a list of communities as follows:

```
community == {100, NO_EXPORT, 3561:10, 200};   # exact match
```

To influence the route selection, the BGP `as_path` rp-attribute can be made longer by prepending AS numbers to it as follows:

```
aspath.prepend(AS1);
aspath.prepend(AS1, AS1, AS1);
```

The following examples are invalid:

```
med = -50;                     # -50 is not in the range
med = igp;                     # igp is not one of the enum values
med.assign(10);                # method assign is not defined
community.append(AS3561:20);   # the first argument should be 3561
```

Figure 22 shows a more advanced example using the rp-attribute `community`. In this example, `AS3561` bases its route selection preference on the community attribute. Other ASes may indirectly affect `AS3561`'s route selection by including the appropriate communities in their route announcements.

```
aut-num: AS1
export: to AS2 action community.={3561:90};
        to AS3 action community.={3561:80};
        announce AS1

as-set: AS3561:AS-PEERS
members: AS2, AS3

aut-num: AS3561
import: from AS3561:AS-PEERS
        action pref = 10;
        accept community.contains(3561:90)
import: from AS3561:AS-PEERS
        action pref = 20;
        accept community.contains(3561:80)
import: from AS3561:AS-PEERS
        action pref = 20;
        accept community.contains(3561:70)
import: from AS3561:AS-PEERS
        action pref = 0;
        accept ANY
```

Figure 22: Policy example using the `community` rp-attribute.

8 Advanced `route` Class

8.1 Specifying Aggregate Routes

The `components`, `aggr-bndry`, `aggr-mtd`, `export-comps`, `inject`, and `holes` attributes are used for specifying aggregate routes [10]. A route object specifies an aggregate route if any of these attributes, with the exception of `inject`, is specified. The `origin` attribute for an aggregate route is the AS performing the aggregation, i.e. the aggregator AS. In this section, we used the term "*aggregate*" to refer to the route generated, the term "*component*" to refer to the routes used to generate the path attributes of the aggregate, and the term "*more specifics*" to refer to any route which is a more specific of the aggregate regardless of whether it was used to form the path attributes.

The `components` attribute defines what component routes are used to form the aggregate. Its syntax is as follows:

```
components: [ATOMIC] [[protocol <protocol>] <filter>
                      [protocol <protocol> <filter> ...]]
```

where `<protocol>` is a routing protocol name such as `BGP`, `OSPF` or `RIP` (valid names are defined in the dictionary) and `<filter>` is a policy expression. The routes that match one of these filters and are learned from the corresponding protocol are used to form the aggregate. If `<protocol>` is omitted, it defaults to any protocol. `<filter>` implicitly contains an "`AND`" term with the more specifics of the aggregate so that only the component routes are selected. If the keyword `ATOMIC` is used, the aggregation is done atomically [10]. If a `<filter>` is not specified it defaults to more specifics. If the `components` attribute is missing, all more specifics without the `ATOMIC` keyword is used.

```
route: 128.8.0.0/15
origin: AS1
components: <^AS2>

route: 128.8.0.0/15
origin: AS1
components: protocol BGP  {128.8.0.0/16^+}
            protocol OSPF {128.9.0.0/16^+}
```

Figure 23: Two aggregate `route` objects.

Figure 23 shows two route objects. In the first example, more specifics of `128.8.0.0/15` with AS paths starting with `AS2` are aggregated. In the second example, some routes learned from BGP and some routes learned form OSPF are aggregated.

The `aggr-bndry` attribute is an expression over AS numbers and sets using operators `AND`, `OR`, and `NOT`. The result defines the set of ASes which form the aggregation boundary. If the `aggr-bndry` attribute is missing, the origin AS is the sole aggregation boundary. Outside the aggregation boundary, only the aggregate is exported and more specifics are suppressed. However, within the boundary, the more specifics are also exchanged.

The `aggr-mtd` attribute specifies how the aggregate is generated. Its syntax is as follow:

```
aggr-mtd: inbound
        | outbound [<as-expression>]
```

where `<as-expression>` is an expression over AS numbers and sets using operators `AND`, `OR`, and `NOT`. If `<as-expression>` is missing, it defaults to `AS-ANY`. If outbound aggregation is specified, the more specifics of the aggregate will be present within the AS and the aggregate will be formed at all inter-AS boundaries with ASes in `<as-expression>` before export, except for ASes that are within the aggregating boundary (i.e. `aggr-bndry` is enforced regardless of `<as-expression>`). If inbound aggregation is specified, the aggregate is formed at all inter-AS boundaries prior to importing routes into the aggregator AS. Note that `<as-expression>` can not be specified with inbound aggregation. If `aggr-mtd` attribute is missing, it defaults to "`outbound AS-ANY`".

```
route:      128.8.0.0/15              route:      128.8.0.0/15
origin:     AS1                       origin:     AS2
components: {128.8.0.0/15^-}          components: {128.8.0.0/15^-}
aggr-bndry: AS1 OR AS2                aggr-bndry: AS1 OR AS2
aggr-mtd:   outbound AS-ANY           aggr-mtd:   outbound AS-ANY
```

Figure 24: Outbound multi-AS aggregation example.

Figure 24 shows an example of an outbound aggregation. In this example, AS1 and AS2 are coordinating aggregation and announcing only the less specific `128.8.0.0/15` to outside world, but exchanging more specifics between each other. This form of aggregation is useful when some of the components are within `AS1` and some are within `AS2`.

When a set of routes are aggregated, the intent is to export only the aggregate route and suppress exporting of the more specifics outside the aggregation boundary. However, to satisfy certain policy and topology constraints (e.g. a multi-homed component), it is often required to export some of the components. The `export-comps` attribute serves this purpose. It equals an RPSL export policy expression as follows (see Section6.2):

```
export-comps: to <peering-1> [action <action-1>]
              . . .
              to <peering-N> [action <action-N>]
              announce <filter>
```

where `<filter>` matches the more specifics that need to be exported to the peerings specified. While exporting to `<peering-i>`, the `<action-i>` is executed to set the path attributes. If this attribute is missing, more specifics are not exported outside the aggregation boundary. Note that, the `<filter>` contains an implicit "`AND`" term with the more specifics of the aggregate.

```
route:      128.8.0.0/15
origin:     AS1
components: {128.8.0.0/15^-}
aggr-mtd:   outbound AS-ANY
export-comps: to as-any announce { 128.8.8.0/24 }
```

Figure 25: Outbound aggregation with export exception.

Figure 25 shows an example of an outbound aggregation. In this example, the more specific `128.8.8.0/24` is exported outside AS1 in addition to the aggregate. This is useful, when `128.8.8.0/24` is multi-homed site to AS1 with some other AS.

The `inject` attribute specifies which routers perform the aggregation and when they perform it.

Its syntax is as follow:

```
inject: [at <router-expression>] ...
        [action <action>]
        [upon <condition>]
```

where `<action>` is an action specification (see Section 6.1.2), `<condition>` is a boolean expression described below, and`<router-expression>` is an expression over router IP addresses and DNS names using operators `AND`, `OR`, and `NOT`. The DNS name can only be used if there is an `inet-rtr` object for that name that binds the name to IP addresses.

All routers in `<router-expression>` and in the aggregator AS perform the aggregation. If a `<router-expression>` is not specified, all routers inside the aggregator AS perform the aggregation. The `<action>` specification may set path attributes of the aggregate, such as assign a preferences to the aggregate.

The upon clause is a boolean condition. The aggregate is generated if and only if this condition is true. `<condition>` is a boolean expression using the logical operators `AND` and `OR` (i.e. operator `NOT` is not allowed) over:

```
HAVE-COMPONENTS { list of prefixes }
EXCLUDE { list of prefixes }
STATIC
```

The list of prefixes in `HAVE-COMPONENTS` can only be more specifics of the aggregate. It evaluates to true when *all* the prefixes listed are present in the routing table of the aggregating router. The list can also include prefix ranges (i.e. using operators `^-`, `^+`, `^n`, and `^n-m`). In this case, at least one prefix from each prefix range needs to be present in the routing table for the condition to be true. The list of prefixes in `EXCLUDE` can be arbitrary. It evaluates to true when *none* of the prefixes listed is present in the routing table. The list can also include prefix ranges, and no prefix in that range should be present in the routing table. The keyword `static` always evaluates to true. If no `upon` clause is specified the aggregate is generated if an only if there is a component in the routing table (i.e. a more specific that matches the filter in the `components` attribute).

Figure 26 shows two examples. In the first case, the aggregate is injected at two routers each one setting the `dpa` path attribute differently. In the second case, the aggregate is generated only if *both* `128.8.0.0/16` and `128.9.0.0/16` are present in the routing table, as opposed to the first case where the presence of just *one* of them is sufficient for injection.

The `holes` attribute lists the component address prefixes which are not reachable through the aggregate route (perhaps that part of the address space is unallocated). The `holes` attribute is useful for diagnosis purposes. In Figure 26, the second example has a hole, namely `128.8.8.0/24`. This may be due to a customer changing providers and taking this part of the address space with it.

```
route:      128.8.0.0/15
origin:     AS1
components: {128.8.0.0/15^-}
aggr-mtd:   outbound AS-ANY
inject:     at 1.1.1.1 action dpa = 100;
inject:     at 1.1.1.2 action dpa = 110;

route:      128.8.0.0/15
origin:     AS1
components: {128.8.0.0/15^-}
aggr-mtd:   outbound AS-ANY
inject:     upon HAVE-COMPONENTS {128.8.0.0/16, 128.9.0.0/16}
holes:      128.8.8.0/24
```

Figure 26: Examples of inject.

8.1.1 Interaction with policies in `aut-num` class

An aggregate formed is announced to other ASes only if the export policies of the AS allows exporting the aggregate. When the aggregate is formed, the more specifics are suppressed from being exported except to the ASes in `aggr-bndry` and except the components in `export-comps`. For such exceptions to happen, the export policies of the AS should explicitly allow exporting of these exceptions.

If an aggregate is not formed (due to the `upon` clause), then the more specifics of the aggregate can be exported to other ASes, but only if the export policies of the AS allows it. In other words, before a route (aggregate or more specific) is exported it is filtered twice, once based on the route objects, and once based on the export policies of the AS.

In Figure 27 shows an interaction example. By examining the route objects, the more specifics `128.8.0.0/16` and `128.9.0.0/16` should be exchanged between `AS1`, `AS2` and `AS3` (i.e. the aggregation boundary). Outbound aggregation is done to `AS4` and `AS5` and not to `AS3`, since `AS3` is in the aggregation boundary. The `aut-num` object allows exporting both components to `AS2`, but only the component `128.8.0.0/16` to `AS3`. The aggregate can only be formed if both components are available. In this case, only the aggregate is announced to `AS4` and `AS5`. However, if one of the components is not available the aggregate will not be formed, and any available component or more specific will be exported to `AS4` and `AS5`. Regardless of aggregation is performed or not, only the more specifics will be exported to `AS6` (it is not listed in the `aggr-mtd` attribute).

When doing an inbound aggregation, configuration generators may eliminating the aggregation statements on routers where import policy of the AS prohibits importing of any more specifics.

```
route:        128.8.0.0/16
origin:       AS1

route:        128.9.0.0/16
origin:       AS1

route:        128.8.0.0/15
origin:       AS1
aggr-bndry:   AS1 or AS2 or AS3
aggr-mtd:     outbound AS3 or AS4 or AS5
components:   {128.8.0.0/16, 128.9.0.0/16}
inject:       upon HAVE-COMPONENTS {128.9.0.0/16, 128.8.0.0/16}

aut-num: AS1
export:  to AS2 announce AS1
export:  to AS3 announce AS1 and not {128.9.0.0/16}
export:  to AS4 announce AS1
export:  to AS5 announce AS1
export:  to AS6 announce AS1
```

Figure 27: Interaction with policies in `aut-num` class.

8.1.2 Ambiguity resolution with overlapping aggregates

When several aggregate routes are specified and they overlap, i.e. one is less specific of the other, they must be evaluated more specific to less specific order. When an *outbound* aggregation is performed for a peer, the aggregate and the components listed in the `export-comps` attribute for that peer are available for generating the next less specific aggregate. The components that are not specified in the `export-comps` attribute are not available. A route is exportable to an AS if it is the least specific aggregate exportable to that AS or it is listed in the `export-comps` attribute of an exportable route. Note that this is a recursive definition.

In Figure 28, `AS1` together with `AS2` aggregates `128.8.0.0/16` and `128.9.0.0/16` into `128.8.0.0/15`. Together with `AS3`, `AS1` aggregates `128.10.0.0/16` and `128.11.0.0/16` into `128.10.0.0/15`. But altogether they aggregate these four routes into `128.8.0.0/14`. Assuming all four components are available, a router in `AS1` for an outside AS, say `AS4`, will first generate `128.8.0.0/15` and `128.10.0.0/15`. This will make `128.8.0.0/15`, `128.10.0.0/15` and its exception `128.11.0.0/16` available for generating `128.8.0.0/14`. The router will then generate `128.8.0.0/14` from these three routes. Hence for `AS4`, `128.8.0.0/14` and its exception `128.10.0.0/15` and its exception `128.11.0.0/16` will be exportable.

For `AS2`, a router in `AS1` will only generate `128.10.0.0/15`. Hence, `128.10.0.0/15` and its exception `128.11.0.0/16` will be exportable. Note that `128.8.0.0/16` and `128.9.0.0/16` are also

```
route:        128.8.0.0/15
origin:       AS1
aggr-bndry:   AS1 or AS2
aggr-mtd:     outbound
inject:       upon HAVE-COMPONENTS {128.8.0.0/16, 128.9.0.0/16}

route:        128.10.0.0/15
origin:       AS1
aggr-bndry:   AS1 or AS3
aggr-mtd:     outbound
inject:       upon HAVE-COMPONENTS {128.10.0.0/16, 128.11.0.0/16}
export-comps: to as-any announce {128.11.0.0/16}

route:        128.8.0.0/14
origin:       AS1
aggr-bndry:   AS1 or AS2 or AS3
aggr-mtd:     outbound
inject:       upon HAVE-COMPONENTS {128.8.0.0/15, 128.10.0.0/15}
export-comps: to as-any announce {128.10.0.0/15}
```

Figure 28: Overlapping aggregations.

exportable since they did not participate in an aggregate exportable to AS2.

Similarly, for AS3, a router in AS1 will only generate 128.8.0.0/15. In this case 128.8.0.0/15, 128.10.0.0/16, 128.11.0.0/16 are exportable.

8.2 Specifying Static Routes

The inject attribute can be used to specify static routes by using "upon static" as the condition:

```
inject: [at <router>] ...
        [action <action>]
        upon static
```

In this case, the <router> executes the <action> and injects the route to the interAS routing system statically. <action> may set certain route attributes such as a next-hop router or a cost.

In the following example, the router 7.7.7.1 injects the route 128.7.0.0/16. The next-hop routers (in this example, there are two next-hop routers) for this route are 7.7.7.2 and 7.7.7.3 and the route has a cost of 10 over 7.7.7.2 and 20 over 7.7.7.3.

```
route:  128.7.0.0/16
origin: AS1
inject: at 7.7.7.1 action next-hop = 7.7.7.2; cost = 10; upon static
inject: at 7.7.7.1 action next-hop = 7.7.7.3; cost = 20; upon static
```

9 inet-rtr Class

Routers are specified using the `inet-rtr` class. The attributes of the `inet-rtr` class are shown in Figure 29. The `inet-rtr` attribute is a valid DNS name of the router described. Each `alias` attribute, if present, is a canonical DNS name for the router. The `local-as` attribute specifies the AS number of the AS which owns/operates this router.

Attribute	Value	Type
`inet-rtr`	⟨*dns-name*⟩	mandatory, single-valued, class key
`alias`	⟨*dns-name*⟩	optional, multi-valued
`local-as`	⟨*as-number*⟩	mandatory, single-valued
`ifaddr`	see description in text	mandatory, multi-valued
`peer`	see description in text	optional, multi-valued

Figure 29: `inet-rtr` Class Attributes

The value of an `ifaddr` attribute has the following syntax:

```
<ipv4-address> masklen <integer> [action <action>]
```

The IP address and the mask length are mandatory for each interface. Optionally an action can be specified to set other parameters of this interface.

Figure 30 presents an example `inet-rtr` object. The name of the router is "`amsterdam.ripe.net`". "`amsterdam1.ripe.net`" is a canonical name for the router. The router is connected to 4 networks. Its IP addresses and mask lengths in those networks are specified in the `ifaddr` attributes.

Each `peer` attribute, if present, specifies a protocol peering with another router. The value of a `peer` attribute has the following syntax:

```
<protocol> <ipv4-address> <options>
```

where `<protocol>` is a protocol name, `<ipv4-address>` is the IP address of the peer router, and `<options>` is a comma separated list of peering options for `<protocol>`. Possible protocol names

```
inet-rtr: Amsterdam.ripe.net
alias:    amsterdam1.ripe.net
local-as: AS3333
ifaddr:   192.87.45.190 masklen 24
ifaddr:   192.87.4.28   masklen 24
ifaddr:   193.0.0.222   masklen 27
ifaddr:   193.0.0.158   masklen 27
peer:     BGP4 192.87.45.195 asno(AS3334), flap_damp()
```

Figure 30: `inet-rtr` Objects

and attributes are defined in the dictionary (please see Section 7). In the above example, the router has a BGP peering with the router `192.87.45.195` in `AS3334` and turns the flap damping on when importing routes from this router.

10 Extending RPSL

Our experience with earlier routing policy languages and data formats (PRDB [2], RIPE-81 [7], and RIPE-181 [6]) taught us that RPSL had to be extensible. As a result, extensibility was a primary design goal for RPSL. New routing protocols or new features to existing routing protocols can be easily handled using RPSL's dictionary class. New classes or new attributes to the existing classes can also be added.

This section provides guidelines for extending RPSL. These guidelines are designed with an eye toward maintaining backward compatibility with existing tools and databases. We next list the available options for extending RPSL from the most preferred to the least preferred order.

10.1 Extensions by changing the dictionary class

The `dictionary` class is the primary mechanism provided to extend RPSL. Dictionary objects define *routing policy attributes*, *types*, and *routing protocols*.

We recommend updating the RPSL dictionary to include appropriate rp-attribute and protocol definitions as new path attributes or router features are introduced. For example, in an earlier version of the RPSL document, it was only possible to specify that a router performs route flap damping on a peer, but it was not possible to specify the parameters of route flap damping. Later the parameters were added by changing the dictionary.

When changing the dictionary, full compatibility should be maintained. For example, in our flap damping case, we made the parameter specification optional in case this level of detail was not desired by some ISPs. This also achieved compatibility. Any object registered without the parameters will continue to be valid. Any tool based on RPSL is expected to do a default action on routing policy attributes that they do not understand (e.g. issue a warning and otherwise ignore). Hence, old tools upon encountering a flap damping specification with parameters will ignore the parameters.

10.2 Extensions by adding new attributes to existing classes

New attributes can be added to any class. To ensure full compatibility, new attributes should not contradict the semantics of the objects they are attached to. Any tool that uses the IRR should be designed so that it ignores attributes that it doesn't understand. Most existing tools adhere to this design principle.

We recommend adding new attributes to existing classes when a new aspect of a class is discovered. For example, RPSL `route` class extends its RIPE-181 predecessor by including several new attributes that enable aggregate and static route specification.

10.3 Extensions by adding new classes

New classes can be added to RPSL to store new types of policy data. Providing full compatibility is straight forward as long as existing classes are still understood. Since a tool should only query the IRR for the classes that it understand, full compatibility should not be a problem in this case.

Before adding a new class, one should question if the information contained in the objects of the new class could have better belonged to some other class. For example, if the geographic location of a router needs to be stored in IRR, it may be tempting to add a new class called, say `router-location` class. However, the information better belongs to the `inet-rtr` class, perhaps in a new attribute called `location`.

10.4 Extensions by changing the syntax of existing RPSL attributes

If all of the methods described above fail to provide the desired extension, it may be necessary to change the syntax of RPSL. Any change in RPSL syntax must provide backwards compatibility, and should be considered only as a last resort since full compatibility may not be achievable. However, we require that the old syntax to be still valid.

11 Security Consideration

This document describes RPSL, a language for expressing routing policies. The language defines a maintainer (`mntner` class) object which is the entity which controls or "maintains" the objects stored in a database expressed by RPSL. Requests from maintainers can be authenticated with various techniques as defined by the "`auth`" attribute of the maintainer object.

The exact protocols used by IRR's to communicate RPSL objects is beyond the scope of this document, but it is envisioned that several techniques may be used, ranging from interactive query/update protocols to store and forward protocols similar to or based on electronic mail (or even voice telephone calls). Regardless of which protocols are used in a given situation, it is expected that appropriate security techniques such as IPSEC, TLS or PGP/MIME will be utilized.

12 Acknowledgements

We would like to thank Jessica Yu, Randy Bush, Alan Barrett, David Kessens, Bill Manning, Sue Hares, Ramesh Govindan, Kannan Varadhan, Satish Kumar, Craig Labovitz, Rusty Eddy, David J. LeRoy, David Whipple, Jon Postel, Deborah Estrin, Elliot Schwartz, Joachim Schmitz, Mark Prior, Tony Przygienda, David Woodgate, Rob Coltun, Sanjay Wadhwa, Ardas Cilingiroglu, and the participants of the IETF RPS Working Group for various comments and suggestions.

References

[1] Internet routing registry. procedures. http://www.ra.net/RADB.tools.docs/, http://www.ripe.net/db/doc.html.

[2] Nsfnet policy routing database (prdb). Maintained by MERIT Network Inc., Ann Arbor, Michigan.
Contents available from `nic.merit.edu.:/nsfnet/announced.networks/nets.tag.now` by anonymous ftp.

[3] C. Alaettinoğlu, D. Meyer, and J. Schmitz. Application of routing policy specification language (rpsl) on the internet. Internet Draft draft-ietf-rps-appl-rpsl-01, July 1997. Work in progress.

[4] T. Bates. Specifying an 'internet router' in the routing registry. Technical Report RIPE-122, RIPE, RIPE NCC, Amsterdam, Netherlands, October 1994.

[5] T. Bates, E. Gerich, L. Joncheray, J-M. Jouanigot, D. Karrenberg, M. Terpstra, and J. Yu. Representation of ip routing policies in a routing registry. Technical Report ripe-181, RIPE, RIPE NCC, Amsterdam, Netherlands, October 1994.

[6] T. Bates, E. Gerich, L. Joncheray, J-M. Jouanigot, D. Karrenberg, M. Terpstra, and J. Yu. Representation of ip routing policies in a routing registry. Technical Report RFC-1786, Network Information Center, March 1995.

[7] T. Bates, J-M. Jouanigot, D. Karrenberg, P. Lothberg, and M. Terpstra. Representation of ip routing policies in the ripe database. Technical Report ripe-81, RIPE, RIPE NCC, Amsterdam, Netherlands, February 1993.

[8] R. Chandra, P. Traina, and T. Li. Bgp communities attribute. Request for Comment RFC-1997, Network Information Center, August 1996.

[9] D. Crocker. Standard for the format of arpa internet text messages. Request for Comment RFC-822, Network Information Center, August 1982.

[10] V. Fuller, T. Li, J. Yu, and K. Varadhan. *Classless Inter-Domain Routing (CIDR): an Address Assignment and Aggregation Strategy*, 1993.

[11] D. Karrenberg and T. Bates. Description of inter-as networks in the ripe routing registry. Technical Report RIPE-104, RIPE, RIPE NCC, Amsterdam, Netherlands, December 1993.

[12] D. Karrenberg and M. Terpstra. Authorisation and notification of changes in the ripe database. Technical Report ripe-120, RIPE, RIPE NCC, Amsterdam, Netherlands, October 1994.

[13] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall, 1978.

[14] D. Kessens, W. Woeber, and D. Conrad. Ride referencing. Internet Draft draft-kessens-ride-referencing-00.txt, Network Information Center, August 1997.

[15] A. Lord and M. Terpstra. Ripe database template for networks and persons. Technical Report ripe-119, RIPE, RIPE NCC, Amsterdam, Netherlands, October 1994.

[16] A. M. R. Magee. Ripe ncc database documentation. Technical Report RIPE-157, RIPE, RIPE NCC, Amsterdam, Netherlands, May 1997.

[17] P. V. Mockapetris. Domain names - concepts and facilities. Request for Comment RFC-1034, Network Information Center, November 1987.

[18] Y. Rekhter. Inter-domain routing protocol (idrp). *Journal of Internetworking Research and Experience*, 4:61–80, 1993.

[19] Y. Rekhter and T. Li. A border gateway protocol 4 (bgp-4). Request for Comment RFC-1771, Network Information Center, March 1995.

[20] C. Villamizar, C. Alaettinoğlu, D. Meyer, S. Murphy, and C. Orange. Routing policy system security. Internet Draft draft-ietf-rps-auth-01, Network Information Center, May 1998.

[21] C. Villamizar, R. Chandra, and R. Govindan. Bgp route flap damping. Internet Draft draft-ietf-idr-route-damp-00, Network Information Center, October 1997.

[22] J. Zsako. Pgp authentication for ripe database updates. Internet Draft draft-zsako-ripe-dbsec-pgp-authent-00, Network Information Center, July 1998.

A Routing Registry Sites

The set of routing registries as of November 1996 are `RIPE`, `RADB`, `CANet`, `MCI` and `ANS`. You may contact one of these registries to find out the current list of registries.

B Authors' Addresses

Cengiz Alaettinoglu
USC Information Sciences Institute
4676 Admiralty Way, Suite 1001
Marina del Rey, CA 90292
email: cengiz@isi.edu

Tony Bates
Cisco Systems, Inc.
170 West Tasman Drive
San Jose, CA 95134
email: tbates@cisco.com

Elise Gerich
At Home Network
385 Ravendale Drive
Mountain View, CA 94043
email: epg@home.net

Daniel Karrenberg
RIPE Network Coordination Centre (NCC)
Kruislaan 409
NL-1098 SJ Amsterdam
Netherlands
email: dfk@ripe.net

David Meyer
University of Oregon
Eugene, OR 97403
email: meyer@antc.uoregon.edu

Marten Terpstra
c/o Bay Networks, Inc.
2 Federal St
Billerica MA 01821
email: marten@BayNetworks.com

Curtis Villamizar
ANS
email: curtis@ans.net