

Network-Wide Inter-Domain Routing Policies: Design and Realization

Hagen Böhm	Anja Feldmann	Olaf Maennel	Christian Reiser	Rüdiger Volk
Deutsche Telekom	TU München	TU München	TU München	Deutsche Telekom
Münster, Germany	München, Germany	München, Germany	München, Germany	Münster, Germany

Abstract—Currently the inter-domain routing policy of an autonomous system (AS) is often ill-specified, undergoes constant adjustments for reasons of traffic engineering and/or to address-specific customer wishes, and is often realized by manually configuring each router individually, an error prone approach. This paper presents a system that raises the abstraction level at which routing policies are specified from individual BGP statements to a network-wide routing policy.

Our system enables an autonomous system (i) to explicitly specify its inter-domain network-wide routing policy as first class entities, an extensible collection of individual policies and services such as a peering-policy, a filter-martians-policy, a signaled black-hole service, etc.; (ii) to specify its routing policy independently of the current state of the network; (iii) to automatically generate the appropriate pieces of the router configurations for all routers in the network from appropriate databases.

We are now able to manage the overall routing architecture rather than each individual router. Initial deployment of the system to manage the network-wide routing policy of a large AS affirm the above advantages in an operational setting.

I. INTRODUCTION

A network's *routing policy* is embodied in the configuration of the routing protocols of each individual router and link in the network. These configurations are the basis on which the Internet routing protocols create the “network-wide intelligence” that transforms individual network components into an operational IP network [1]. Initially the goal of a routing policy was to just ensure connectivity. But nowadays the requirements and therefore the complexity has grown substantially and includes maintaining contractual or business relationships between different domains, ensuring resilience requirements even under overload, achieving stability, guaranteeing efficient internal operations, and supporting growing customer demands to control their traffic flows.

The de-facto standard policy routing protocol for realizing the routing policy of an autonomous system (AS) is the Border Gateway Protocol (BGP) [2], [3]. In BGP routes carry attributes, some of which have a well specified semantic. In contrast, the community attribute [4] provides a large code space that can be freely used by a routing policy designer to define signaling within the domain and/or across domains.

Sophisticated customers of Internet transit services demand more control over how their routes are propagated past their immediate transit providers (with the goal to control the paths of the traffic attracted by the routes [5]). Community signaled services address this problem, and more recently they have become an essential part of transit providers service offerings. The NOPEER community [6] gives a simple, and useful example, that has additional appeal by virtue of defining Internet wide semantics. More use cases have been collected in [7].

Even though the demands on the routing policy have grown,

the way how a routing policy is expressed and realized has not changed. It is still embedded in the low level router configuration commands of the individual network components of the network. This means that a network-wide entity, the routing policy, is not managed network-wide but rather component-by-component. Therefore artifacts within the routing policy can lead to routing stability problems, e.g., [8], [9]. Even worse, component-by-component realization often involves manual configuration which is error-prone and expensive [10], [11], [12], [13], [14]. In the worst case a simple omission on one router can invalidate the overall policy [15].

Already the problem of formulating the network-wide routing policy of an Internet service provider (ISP) is non-trivial. The Internet Routing Registry (IRR) [16], which facilitates universal access to routing policies and is part of an architecture for coordinating Internet routing policies [17], has not yet fulfilled its potential [18]. Either the ISP does not even try to formulate its policy in the proposed language, RPSL [19], [20], or the policy is not consistent with the routing policy actually realized. Yet, the growing complexity of policies as well as the growing customer demands in addition to the intrinsic complexities of BGP itself, e.g., [21], [22], [23], [24], [25], accentuate the need for a system that allows an abstract formulation of the network-wide routing policy of an ISP as well as the automatic configuration of the network components.

In this paper we propose a system for formulating and managing the routing policy as a first class entity. Using the routing policy as its basis, our system generates the necessary configuration *configlets* for all routers in a specific network to realize the routing policy as well as a documentation of the routing policy. In effect we raise the management level for the routing policy from the management of individual network components to managing the routing policy itself. A version of our system is in production use at a large ISP with more than thousand routers.

The remainder of this paper is organized as follows: in Section II we explore the concept of a routing policy and then explain, in Section III, our system design in more detail. Section IV describes the data models. Section V discusses how to construct a database-driven provisioning system, the *configurator*, based on these data models. Section VI shows some generated *configlets* for the examples presented in Section IV and discusses our experiences with the system. After reviewing related work, in Section VII, the paper concludes in Section VIII with a summary of our contributions and future research directions. Some background material on router configuration and RPSL can be found in the Appendix.

II. NETWORK-WIDE ROUTING POLICY

Before we can present our system for defining and realizing a *network-wide routing policy* we need a better understanding of the underlying principles of a routing policy and how routing policies are realized today in operational networks.

A. Principles underlying a network-wide routing policy

A routing policy should capture the essences of the many rules according to which ASes interact with each other. The multitude of such rules include rules for realizing best common practice (such as martians filter, prefix aggregation), rules for maintaining business relationships (such as a peering policy), and rules for offering value-added services to customers (such as community-signaled services). As a result, a BGP session is governed by some combination of these rules, which together determine which routes are accepted on ingress or propagated on egress. The building blocks for a network-wide routing policy are policies and services. Each policy or service corresponds to a rule like the ones discussed above. The difference is that a service is an optional value-added feature, that can be booked for any neighbor. Policies are applicable network-wide (within the AS). One aspect to consider is that rules change over time, that new rules are added while others are removed, and that even policies may become services or vice versa.

Almost none of the above rules refer to specific BGP neighbors or specific network components. Indeed, the rules can be formulated independently of the specifics of any network. Just the selection of BGP sessions, to which a rule has to be applied, depends on certain parameters of the BGP session, the BGP neighbor, or the services that are booked for the BGP session or the BGP neighbor. In fact, certain rules cannot be specified on a session-by-session basis. Rather they require that some actions are taken for all routes entering a network and for all routes leaving the network. But depending on the session over which they enter the AS and the session over which they leave the AS the actions have to differ. For example, a peering policy [26] may mark all routes on ingress with a community that reflects the peering type of the BGP session and on egress it will filter the routes based on the community tags and the peering type of the session. Here the peering type, a parameter of the session, determines the actions on the routes. Other rules, e.g., a community-signaled black-hole service [27], require different actions to be taken for different sets of sessions. For those sessions that have subscribed to the service, the rule checks if an appropriate black-hole community is set for a route. If it is, traffic to that prefix is redirected to a discard interface. If a session has not subscribed to the service the rule does not allow the BGP neighbor to send routes tagged with the black-hole community. These are rejected.

Relying on the above example we identify the following *principles* for defining a network-wide routing policy:

- It consists of a collection of *policies* and *services*.
- Policies apply network-wide.
- Services are bookable for each session or each neighbor.
- Policies and services perform *route manipulations* for *subsets* of BGP sessions. (A route manipulation consists of applying a filter to the incoming or outgoing routes and manipulating their attributes.)

- Subsets of BGP sessions are identified via *conditions* on BGP session parameters or their services.
- A policy/service may perform route manipulations on *multiple* subsets of BGP sessions.

Accordingly, any system that supports the definition of a network-wide routing policy needs primitives for defining policies and services as well as specifying sets of session and route manipulations. There is no need for these primitives to be network-specific. Indeed, in principle it should be possible to use a policy or a service defined by one ISP within another one.

B. Today's realization of a routing policy

Now that we understand the principles behind routing policies we examine how routing policies are actually realized in operational networks. This points out the discrepancy between the above abstract notion and the actual reality.

An ISP interacts with other domains, their neighbors, by setting up connectivity between their networks with the purpose of exchanging traffic. As traffic flows are the result of routing information exchange this exchange has to adhere to the routing policy of the ISP. Currently, an ISP realizes his inter-domain routing policy via BGP. This includes an appropriate setup of IBGP within his AS to control the flow of routing information inside his network, and EBGP across external links to control the flow of routing information entering and leaving his network. In EBGP routing information is exchanged over BGP sessions, one session per link. As such BGP operation is specified in a distributed manner. Even if the ISP has multiple connections to the same neighbor using the same policy, it is necessary to specify multiple BGP sessions on different routers that are for the most parts duplicates of each other.

A router exchanges routing information about particular network addresses (prefixes) via BGP sessions in the following way. First, the input filter policies of the session over which a route is received is applied. This can rewrite the BGP attributes. Then a BGP decision process considers a priority-list of attributes to select the preferred route. If the “best” route changes, then the routing table is updated, and the new best route passes through the output filter policies of all sessions. These can again rewrite the BGP attributes. Finally, if it passes the filter the route is propagated. This process is referred to as route manipulation. While most attributes have a well specified semantics it is up to the ISP to provide the semantics for the values of the community attribute [4]. Commonly communities are used for grouping destination prefixes into sets, tagging them, and defining routing decisions based on the identity of the group.

Note, that BGP operation has to be specified device-by-device in a distributed manner. The input and output filter policies are the tools of the ISP for realizing his routing policy. Unfortunately, these filter policies provide a very low-level of abstraction: route filters and attribute manipulations.

Each language for instructing routers, the router-specific configuration language, e.g., Cisco IOS [28], JunOS CLI [29], [30], includes primitives to express which BGP operations are to be performed for which session. These support the low level abstractions of BGP operations in a sometimes awkward, non-intuitive language. For example in IOS “router bgp” is used

for specifying the sessions themselves and “route-map” for the route manipulations.

As a result, network’s routing policy is embodied in the low-level configuration of each BGP session which is split across the network components. This creates a situation, where the concept of an abstract routing policy is easily lost.

III. SYSTEM DESIGN

In this paper we propose a simple and efficient way for defining network-wide routing policies and realizing them. In effect we are bridging the gap between the principles underlying network-wide routing policies and how routing policies have to be configured on a session-by-session basis. In order to realize a routing policy, we need additional information besides a policy specification: a description of the network and a library with BGP operations to be used as building blocks. All three entities are realized as databases. These databases are the input to a *configurator* that generates EBGp configuration *configlets* for all routers (all BGP sessions) in the network. In addition, the *configurator* generates a policy description in RPSL, which, if desired, can be published at the IRR and may in the future be useful for generating *configlets*. An overview of the system is shown in Figure 1.

In this section we briefly sketch the abstractions we have chosen for the inputs to the *configurator* before discussing why these are sufficient for satisfying the principles of a network-wide routing policy. Then we explain how the *configurator* generates the desired output, *configlets* for all EBGp sessions. We end this section with a discussion of the advantages that our system offers.

A. Concepts underlying the configurator input

The decomposition of the inputs to the *configurator* into three logical entities, routing policy specification, network specification, and library with BGP operations, imposes a clear separation of tasks and is aligned with the organizational boundaries within an ISP. The policy designer should focus his attention on his set of policies and services. The network administrator, that adds a new BGP session, should only have to worry about the parameters of the specific session and which services are to be booked. He does not have to worry about ensuring that the configuration of this session respects all policies. The person realizing and testing some BGP operation has no need to know which BGP sessions use this specific operation.

The overall network-wide routing policy is stored in the database of the *policy module* and consists of a set of policies and services. The main difference between a service and a policy is that services are bookable and therefore require interactions with the *network module*, while policies apply network-wide. Accordingly, we refer to services as *available services* and to policies as *enforced policies*. As they rely on the same concepts they are both specified as *service*. Each service is expressed as one or more steps of first selecting a set of BGP sessions based on some *conditions* on the *network module* database and then applying one or more BGP operations from the *back-end module* database to these sessions. Each of these combinations is referred to as a *sessionset*. Note, that this process results in two selection steps. The first step, the condition

in the *sessionset*, selects a set of BGP sessions. The second step, the BGP operation, selects a set of routes (via a filter) for that session. Both levels of selection are crucial.

The network specification is stored in a database within the *network module*. It contains a description of the network including routers, BGP neighbors, and BGP sessions with their parameters and list of booked services. We include BGP neighbors (another AS), as most relationships are realized via multiple BGP sessions between the ASes, that either use the same or similar services. To simplify the specification of the network inheritance of parameters is supported. The BGP session inherits (unless overwritten) its router-specific elements from the router on which it is configured and its other elements from the BGP neighbor. Services are parameters of BGP neighbors and sessions. The only difference is that parameters are singular, while services are not. A network operator might want to use different service parameter values for different subsets of routes in order to, e.g., tune traffic flows through its network. We support this via the concept of *cases*.

The *back-end module* database contains a library of *fragments*. Each *fragment* pairs a name with a specific BGP operation. A BGP operation is specific to a single BGP session. A BGP operation consists of selecting a group of routes, which are to be accepted or denied, and performing specific BGP attribute manipulations on the selected routes. Route selection is done via *filters* based on lists of prefixes, lists of communities, regular expressions over AS paths, etc.. Attribute manipulations involve some *actions* on route attributes, e.g., setting the local preference. These are very low-level operations, whose syntax depends on the configuration language supported by the router on which the session is realized. Since, for the purpose of specifying a routing policy, this syntax is irrelevant, the functionality is encapsulated in a library object which can be referred to by a name. *Fragments* are the building blocks for our assembly system, the services and policies.

Besides encapsulating vendor-specific realizations each *fragment* also includes a non-router vendor-specific one to enable us to generate a realization of the routing policy via an intermediate language, a necessity if the number of router vendors that have to be supported increases. RPSL, an object-oriented language designed for publishing routing policies, is the prime candidate for such an intermediate language. RPSL supports most of the features provided by the vendor-specific languages, has excellent support for filter handling, delegation of responsibility, and the ability to operate on sets of routes, sets of BGP sessions, and sets of ASes. As filter handling and delegation are hard problems that are well handled by RPSL, we allow the *fragment* writers to include RPSL filter expressions inside the vendor-specific parts of the *fragments*. These are resolved with the help of RtConfig [31], which is part of the IRRToolSet [32]. RtConfig converts RPSL expressions to vendor-specific router configurations. Since RtConfig can interact directly with the IRR [16], it is possible to retrieve information about neighbors and their sets of routes. This allows us to delegate the responsibility of specifying, e.g., a customer routeset, to the customer itself. This has the benefit that, if the customer changes its routeset, he can handle the change himself [33]. No human interaction with the ISP is necessary, a rather convenient feature.

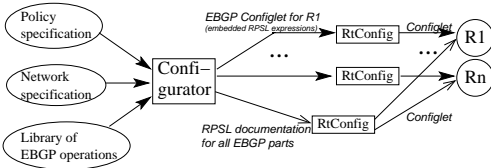


Fig. 1. Overview of proposed system.

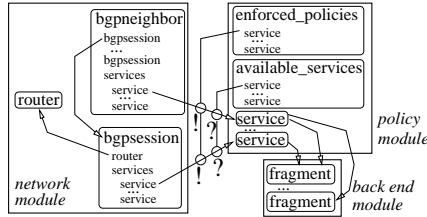


Fig. 2. Main database objects and their relationships.

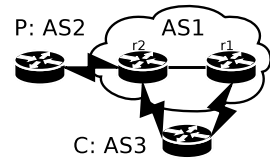


Fig. 3. Example network

An overview of the main objects in the three databases is shown in Fig. 2. This figure also includes the various relationships between the modules:

- The *fragments* of the *back-end module* need information about the specific router and the specific BGP session in order to be converted into a *configlet*. We solve this by writing *fragments* as templates. They can use variables to refer to network-specific elements and a functional interface to access other data sources.
- As soon as a service is specified as an *available service*, it is made bookable as a BGP session parameter. This enables the network operator to book the service for any session. Note, that each service specification defines an appropriate language for specifying the now feasible network configuration. *Enforced services* are not available as BGP session parameters!
- Conditions within the *policy module* are based on the session parameters from the *network module*. This is again solved via variables for referring to network-specific elements.
- Services are defined by applying *fragments* to subsets of BGP sessions. Such references are possible by simply using the name of the *fragment*.

B. Enabling abstract routing policy specification

Next we discuss how our system is able to address the principles identified for defining a routing policy.

“Policies and services perform route manipulations”: For us a route manipulation corresponds to selecting a group of routes, which are to be accepted or denied, and performing specific BGP attribute manipulations on the selected routes. Accordingly, each route manipulation is a BGP operation. The policy designer can use *fragments* to specify what kind of route manipulations to perform as long as the operation of the *fragment* is well-specified and documented.

“Subsets of BGP sessions are identified via conditions on BGP session parameters”: While policies can be specified in an abstract manner, they eventually have to be combined with the network in which the routing policy is realized. At that point the policies have to be applied to specific sessions. The choice of sessions depends on the session parameters and the booked services. This network-specific information is accessible to the policy designer via variables referencing the network elements. Identification or rather selection of BGP sessions is therefore possible via *conditions* on *any* of the BGP session parameters.

“Subsets of BGP sessions are identified via conditions on their services”: Once an *service* is specified as *available service*, it is accessible as BGP session parameter. There is no fundamental difference in identifying or selecting sessions based on booked services or based on other session parameters. The condition can include both.

“Policies and services perform route manipulations on subsets of BGP sessions”: Given that *fragments* are organized in the

back-end module database and network elements are organized in the *network module* database, policies or services can be defined by multiple selections and one join operation. Identifying BGP sessions via conditions is a “select operation” on the network database, determining the appropriate route manipulations is a “select operation” on the *back-end module* database. Combining them is a “join”. This is expressible via *sessionset*.

“A policy/service may perform route manipulations on multiple subsets of BGP sessions”: This corresponds to multiple select and join operations, meaning multiple *sessionsets*.

“Policies apply network-wide; services are bookable for each session or each neighbor”: Available services are bookable by the network administrator via the *network module*, *enforced policies* are not selectable. Their route manipulations just depend on the *conditions* that they use for selecting BGP sessions.

This completes our discussion about how our proposed system allows us to address the principles for defining routing policies. Lets consider how to realize the *configurator*.

C. Design alternatives for the configurator

The next problem to tackle is how to realize a routing policy. Realizing a routing policy corresponds to generating appropriate vendor-specific *configlets* for all routers in the network using the data of the three modules: *policy*, *network*, and *back-end* as input. This is the task of the *configurator*. We choose to generate *configlets* with embedded RPSL filter statements as well as pure RPSL. The latter currently serves documentational purposes. But as the capabilities of RPSL, RtConfig, and the vendor-specific router configuration languages evolve, and our experience with the system grows, this might become the basis for automatically generating the *configlets*.

The motivation for using an hybrid approach is that it lets us benefit from the advantages of generating both, while avoiding some of their disadvantages. The benefits of generating the *configlets* directly are precise control over how policies are realized as well as the chance to optimize the generated code. In addition, the tool can ensure that any filter IDs are used consistently across different instantiations as well as across different routers (very useful for debugging). This eases the transition from a manual configuration or template-based process to our system in the sense that it is often possible to write policies that resemble the existing configurations. Once all configurations are based on the *configlets* generated by the system, it is possible to add new features by enhancing the routing policy. If there is a problem, it is clearly within the *configurator* or any of its input databases. There are no dependencies on other tools or developers. Indeed, since each *fragment* has a very precise and simple functionality, they are easy to write and easy to test.

Currently our tool supports two router vendors. If the number of router vendors increases, an intermediate language such as

RPSL is not only useful but a necessity. Still at this point RPSL does not suffice. In addition, RtConfig is a complex tool grown over time that is fairly intolerant to mistakes in its input data, which is aggravated by a somewhat sparse documentation. Another issue with RPSL and RtConfig is that they do not yet support all features. Also adding support for new vendor features is not trivial and usually comes with some time delay. These problems are circumvented by our hybrid approach.

The *configurator* proceeds by processing the services one by one. For each service, listed under available service or enforced policy, it considers all *sessionsets* within their service specifications. For each *sessionset* the sessions for which the condition evaluates to true are selected. For each of these relevant sessions and all *fragments* within the *sessionset* it adds the *fragment* together with the necessary parameters to a *fragment_list* at the session. Once all services have been processed, the *configurator* generates the *configlets* from the *fragment_lists*, one session at a time, by merging the appropriate vendor-specific realizations of the *fragment* functionality into one *configlet*. The generation of the RPSL documentation proceeds in a different order to take advantage of RPSL's capability of grouping sessions.

The functionality within our system goes beyond RPSL's concept of a policy. For RPSL a policy is a BGP operation on a set of routes. For us a routing policy is collection of policies and services that are each specified as first class entities. As a result if one adds a session to the network in our system one has an explicit choice of which services to enable while all policies are automatically applied. With RPSL one has to explicitly add the session to all sets for all policies that need to be applied to this session. Still, our system is not a superset of RPSL, rather it can be realized via RPSL.

D. Advantages of the approach

Lets consider the properties that our proposed system offers for defining a network-wide routing policy.

Abstraction: Our system allows us to express a routing policy as a set of enforced policies and available services using high-level language primitives rather than forcing one to use low level BGP filters and attribute manipulations. The specifications are stored in a database which eases their maintenance.

Separability: It is possible to independently express how each policy is realized.

Extensibility: Extensibility goes two ways: First it is possible to add new policies or services and/or to change existing ones. Changing existing or introducing new policies is possible in stages. Second it is possible to take advantage of new router features without having to change the routing policy. One just has to rewrite the appropriate *fragments*.

Customizable: Services rely on *fragments* that can access network-specific parameters and filters. As such the services are easily customizable. In addition, our decision to enable the use of RPSL filters allows us to use an existing database.

Modularity: Our system encourages the policy designer to realize each of the many rules of a routing policy as a service. In addition, the services are in principle independent of each other. Of course if two policies manipulate the same BGP attribute, possible conflicts in the realizations have to be addressed and resolved by the overall system.

Debugability: By adding comments to the *configlets* the origin of each line in the generated *configlet* it is easily traceable. Each *fragment* can be debugged individually and the system is able to recognize dependencies and force the designer to disambiguate them. How optimized the generated *configlets* are depends on how optimized the *fragments* are. To improve readability and consistency across routers some redundancy can easily be justified as it improves debugability.

Testability: It is possible to automatically generate *configlets* for all common service combinations. These can then be used as test cases in labs or for manual inspection by the routing policy designer.

IV. DATA MODEL

Our proposed system relies on three databases, the *network module*, which captures the state of the network, the *policy module* for specifying the routing policy, and the *back-end module* for providing a library of base elements for formulating services. We choose to represent each database as a set of XML documents. Among the advantages of choosing XML are: the set of tags is flexible and extensible; XML is designed for representing data and is capable of capturing structure even in semi-structured data; a wide range of editors and libraries are available to ease the generation, maintenance, verification, and processing of the modules.

To illustrate the capabilities of our system we use the example shown in Figure 3. AS 1 uses our system and operates two routers: *r1* and *r2*. It has two BGP neighbors: *P* and *C*. While *C* (AS 2) is a customer with two BGP sessions (*c1* to router *r1* and *c2* to *r2*), *P* (AS 3) is a peer with one BGP session (*p* to *r2*). The routing policy consists of the policies: peering, martians filter, and community filter and the services: black-hole and traffic engineering via MED. The black-hole service is booked for all of *C* sessions and restricted to an authorized routeset. In addition, *C* has the ability to use MED based traffic engineering on session *c2*.

A. Network Module

The task of the *network module* is the description of the components of the network that are necessary for realizing a routing policy. On first sight one would say that these are routers and BGP sessions. We choose to explicitly represent BGP neighbors as their BGP sessions often share parameters and use the same policies. This allows BGP sessions to inherit properties from BGP neighbors. In addition, there are some AS-wide parameters, such as the AS number. Examples of how to describe network elements of the network of Fig. 3 are shown in Fig. 4. (The * after an element indicates that it can be used multiple times.) The relevant XML tags are explained in Fig. 5.

In general, each `<router>` consists of a set of interfaces, is of a certain hardware type, runs a specific OS version, is at a specific location, meaning in a country and geographic region, etc.. Yet, not all of these elements are relevant for expressing policies, e.g., so far there has been no need to access the set of interfaces for realizing any routing policy. Still, location matters, e.g., for supporting policies that tag routes with communities marking the location where the route entered the network. The OS version is needed by the *configurator* to ensure that *configlets* that

<pre><router> <name> r1 </name> <loopback>1.0.0.1</loopback> <location> <city> Munich </city> <country> DE </country> <region> Europe </region> </location> <system> <hw> GSR </hw> <sw> IOS 42.0(12)ST3 </sw> </system> </router></pre>	<pre><bgpneighbor> <name> Neighbor P </name> <neighborAS> 2 </neighborAS> <neighbortype> peer </neighbortype> <session> p </session> <filter_import>AS2:RS-I</filter_import> <filter_export>AS2:RS-E</filter_export> </bgpneighbor> <bgpsession> <name> p </name> <myrouter> r2 </myrouter> <remoteIPAddr>2.0.0.1</remoteIPAddr> </bgpsession></pre>	<pre><bgpsession><name> c2 </name> ... <services> <egress_med> <case><filter>SET-2000</filter> <med_value>2000</med_value></case> <case><filter>SET-3500</filter> <med_value>3500</med_value></case> <default><med_value>100 </med_value></default> </egress_med> </services></bgpsession></pre>
(a) Router <i>r1</i>	(b) BGP neighbor <i>P</i> and session <i>p</i>	(c) BGP session <i>c2</i> for neighbor <i>C</i>

Fig. 4. Examples for *network module* entries

Router name	<name></name>
Router IP address (e.g., loopback IP)	<loopback></loopback>
Router location	<location>
e.g., Region	<region></region>
e.g., Country	<country></country>
	</location>
Router information	<system>
e.g., Hardware type	<hw></hw>
e.g., Software version	<sw></sw>
	</system>

(a) Subelements of <router>

Neighbor name	<name></name>
Neighbor type (e.g., peer)	<neighbortype></neighbortype>
AS number of the neighbor	<neighborAS></neighborAS>
Import filter handle	<filter_import></filter_import>
Export filter handle	<filter_export></filter_export>
BGP session list	<session></session>*
Subscribed services	<services></services>

(b) Basic subelements of <bgpneighbor>

Session name	<name></name>
Router name (within this AS)	<myrouter></myrouter>
AS number of remote endpoint	<neighborAS></neighborAS>
IP address of remote endpoint	<remoteIPAddr></remoteIPAddr>
Port number of remote endpoint	<remoteport></remoteport>
Subscribed services	<services></services>

(c) Basic subelements of <bgpsession>

Service name	<servicename>
Case differentiation	<case>
Filter specification	<filter></filter>
Parameter name	<parametername></parametername>*
	</case>
	<default>
Parameter name	<parametername></parametername>*
	</default>
	</servicename>

(d) Subelements of <services>

Fig. 5. *Network module* XML elements

have been tested for the specific OS versions are generated.

Each <bgpneighbor> captures a relationship with a specific BGP neighbor, which is realized by a number of associated BGP sessions. Each BGP neighbor description includes their AS number and some set of associated basic BGP configuration options, e.g., the maximum number of prefixes that it is allowed to announce. In terms of routing policy it is mandatory/desirable to specify filters to be able to apply incoming and/or outgoing filters. Furthermore, the routing policy has to include a service that ensures that the business relationship with the BGP neighbor is respected by enforcing the peering status: upstream, peer, customer, etc.. We propose to enforce this via a policy rather than offering it as a service. Accordingly, the neighbor type

is not a service parameter but an subelement of BGP neighbor. This way it can be used in the condition of the peering policy. Observe, that there is no need to specify on which routers the BGP sessions are realized.

Each <bgpsession> has to contain sufficient details to realize the session. This includes information about the endpoints, such as the IP addresses, port numbers, and AS numbers. As one does not want to duplicate information from the router or the BGP neighbor we rely on inheritance. This suffices to access the local endpoint information via a cross-reference to the router. Furthermore, the BGP neighbor contains a cross-reference to all its BGP sessions. Accordingly, access to the remote endpoint information is possible. Cross-relationships are realized by using the value of the <name> tag. This enables a BGP session to inherit properties from the router as well as the BGP neighbor, including the service selection. To disable inheritance or to overwrite inherited values the subelements of <router> and <bgpneighbor> are valid within <bgpsession>.

While services are defined within the *policy module* they are booked via the the *network module*. Accordingly, each <bgpneighbor> and <bgpsession> contains a <services> section. The list of available services together with their parameters are extracted (indicated by the italic font in Fig. 5 (d)) from the list of services listed under <available_services>.

Still this is not quite sufficient to enable an appropriate specification of the services in the *network module*. For example an ISP might want to use the MED value 100 as its default. But for some subset of its routes it prefers the value 2000 and for yet another subset the value 3500. This example highlights that a case differentiation is needed within the service specification. Each case has the ability to select routes according to some filter and specify a different parameter value. For the user of the system it can be useful to specify a default case which is applicable if none of the other filters match. A default case is just a case entry without filter. Such a service definition can lead to conflicts, e.g., if route A matches the filters in the first and the second case entry. Which parameter value is appropriate? We decided to resolve such conflicts according to the sequence in which the cases are specified. In this case the parameter value from the first case entry is chosen for route A. We stress that inheritance also applies to services: if a BGP neighbor uses a consistent service set with consistent parameters for all BGP sessions it is sufficient to specify them once (at the BGP neighbor). Otherwise they can be redefined, removed, or/and changed individually at each session.

Network-wide policies	<enforced_policies>
Service name	<name></name>*
	</enforced_policies>
Available routing services	<available_services>
Service name	<name></name>*
	</available_services>
Service definition	<service>
Name	<name></name>
Parameter list	<parameter></parameter>*
Session selection criteria	<sessionset>
Ingress or egress	<direction></direction>
Session selection criteria	<condition></condition>
Group of manipulations	<task>
Manipulations to perform	<fragment></fragment>*
	</task>
Alternative manipulation group	<default>
Manipulations to perform	<fragment></fragment>*
	</default>
	</sessionset>*
	</service>*

Fig. 6. Policy module XML base elements

B. Policies

The **policy module** is at the core of the routing policy description. It allows a policy designer to define a network-wide routing policy using the principles identified in Section II. It enables him to distinguish services that are selectable on a per session basis from policies, that have to be applied network-wide. Accordingly, the top level tags, see Fig. 6, are <service> for specification, <enforced_policy> and <available_service> to distinguish between services and policies.

Recall that a service consists of multiple *sessionsets* each of which uses a condition for selecting sets of BGP session and then applies a set of *fragments* to realize some route manipulations. This structure is reflected in the data model, under <sessionset>. It contains, with <task>, references to the appropriate *fragments* of the *back-end module* for realizing the route manipulation functionality, and with <condition> the ability to specify the desired condition.

Lets review how to select sets of BGP sessions. For a policy the selection depends on some parameters of the session. For a service it matters if the service is booked. In addition, the realization of the service might depend on some parameters of the session. Accordingly, useful selection criteria include, besides the direction of the session, any of the session parameters specified in the *network module*. These are accessible via cross-references using the following (Perl like) syntax: \$element_name.\$subelement_name. All elements of the network are available as such variables for defining services. As the direction of the BGP session, ingress or egress, in which some actions should be applied is orthogonal to the condition for selecting set of session we choose to explicitly separate it. Accordingly, <sessionset> contains the subelements: <direction> and <condition> for selection purposes.

The <condition> allows the policy designer to select BGP sessions based on equality, on numerical comparisons (LOWER, GREATER, EQUAL), or/and on existence tests (IF), etc., on the session parameters, the booked services, and/or the parameters of the booked services. In addition, it is possible to create more complex conditions using the logical operations AND, OR, NOT. It is easily possible to expand this set of operators.

Sometimes it is useful to bundle the specification of which

route manipulations should be performed, if the condition evaluates to true, with those that should be performed, if the condition evaluates to false. For example, the latter is useful to specify the behavior, if a service is not booked, while the former is useful to specify the behavior, if the service is booked. We provide this ability via <task> and <default>. <default> allows the policy designer to specify which *fragments* to apply to those sessions not selected by the condition tag in the same direction. To finish the specification of the services we need the ability to specify the names of the service parameters via parameter. Note, that parameters are only sensible for available services but not for enforced policies.

Now, that we have discussed how to define a routing policy, we show how to specify a routing policy consisting of a black-hole service and a peering policy. We structure our discussion by first specifying the service informally. Next we analyze how it can be represented using our schema. Finally, we explain how the chosen realization, see Fig. 7, fullfils the informal specification.

Black-hole service:

Specification: A possible black-hole service offering might be: If a customer sends a route for a prefix tagged with a specific black-hole community then the AS rewrites the next hop of the route such that the traffic is discarded or could be analyzed. To avoid abuse this is only possible for some well-specified set of prefixes, P_s . (Typically consisting of some of the more specifics of the prefixes, P , that a customer is allowed to announce.)

Analysis: Such a service only requires a *sessionset* for the ingress direction. If the customer has booked the service and the black-hole community is present, then the next hop has to be rewritten. Furthermore, care has to be taken that the prefixes in P_s , which are not part of P , are not rejected. If the customer has not booked the service then routes, tagged with the black-hole community, should be rejected.

Realization: One *sessionset* suffices. Its direction is ingress and the condition is used to select sessions that have booked the service. For these sessions the *fragments* `ingress_black_hole_community` and `ingress_blackhole_accept` ensure, that if the community is present the appropriate route manipulations happen. If the session has not booked the service, `ingress_blackhole_community_deny` is used to reject invalid routes: routes tagged with the black-hole community. Note, that multiple *fragments* are used to realize the service: `ingress_blackhole_community` ensures that, if the community is present, the next hop is rewritten, while `ingress_blackhole_accept` ensures that the prefixes for which the black-hole service is enabled, specified via the parameter `blackhole_set`, are accepted.

Peering policy:

Specification: An AS may partition its set of neighbors into several classes such as, customers, peers, upstream, etc., and use the following peering policy: announce all routes to its customers; announce all routes learned from customers to its peers and its upstreams.

Analysis: This service requires *sessionsets* for ingress as well as for egress. On ingress the AS marks all routes with the neighbor type of the session over which it receives the route. On egress side it filters routes based on these neighbor type marks.

```

<enforced_policies>
  <name>peering</name>
</enforced_policies>
<available_services>
  <name>blackhole</name>
</available_services>
<service><name>blackhole</name>
  <parameter><blackhole_set/></parameter>
  <sessionset><direction>ingress</direction>
  <condition>IF($service.blackhole)</condition>
  <task><fragment>ingress_blackhole_community</fragment>
  <fragment>ingress_blackhole_accept</fragment></task>
  <default><fragment>
    ingress_blackhole_community_deny</fragment></default>
  </sessionset>
</service>

```

```

<service><name>peering</name>
  <parameter>$session.neighbortype</parameter>
  <sessionset><direction>ingress</direction>
  <task><fragment>ingress_mark_neighbortype</fragment></task></sessionset>
  <sessionset><direction>egress</direction>
  <condition>EQUAL($session.neighbortype, "peer")</condition>
  <task><fragment>egress_allow_customer</fragment>
  <fragment>egress_allow_self</fragment></task></sessionset>
  <sessionset><direction>egress</direction>
  <condition>EQUAL($session.neighbortype, "upstream")</condition>
  <task><fragment>egress_allow_customer</fragment>
  <fragment>egress_allow_self</fragment></task></sessionset>
  <sessionset><direction>egress</direction>
  <condition>EQUAL($session.neighbortype, "customer")</condition>
  <task><fragment>egress_allow_all</fragment></task></sessionset>
</service>

```

Fig. 7. Examples for *policy module* entries

Realization: One *sessionset* is not sufficient, since different tasks have to be taken for ingress and egress and on egress for customers and peers respectively upstreams. On ingress no distinction between the sessions is needed, as all routes have to be tagged. Accordingly, no condition is needed and the tagging is performed with the *fragment* `ingress_mark_neighbortype`. On egress the distinction is based on the `neighbortype` of the session. The variable that enables access to the value of this session element is `$session.neighbortype`. The condition is based on the value of the variable. More specifically the value is tested against a fixed string: `peer`, `customer`, or `upstream`. If the `neighbortype` is `peer` or `upstream` the two *fragments* `egress_allow_customer` and `egress_allow_self` are used to ensure the appropriate action. If the `neighbortype` is `customer` no specific filtering is necessary.

C. Back-end module

The **back-end module** provides a library of *fragments* for performing route manipulations and their vendor-specific realizations for use by the *policy module*. Each *fragment* is a capsule for selecting prefix groups and then performing specific BGP operations. The task of realizing a *fragment* can be delegated to an expert for that vendor. For some background material on router configuration see the Appendix.

Fragments are the building blocks for our assembly system, the services. Each *fragment* is a configuration template that should consist of a *filter* for selecting routes and a fairly simple *action* that manipulates some of the attributes of the selected routes. This suffices as the service specification enables one to use multiple *fragments* in the same *sessionset*. Therefore, should a *fragment* become too complicated, it can/should be split into two *fragments*. Accordingly, *fragments* are easy to write and simple to verify.

A particularity that has to be addressed is the need of a uniform way to access information from the network and the *policy module* while dealing with the specifics of the router OSES. This is solved in the *back-end module* in the same way as the *policy module* via variables.

We further simplify the task of the *fragment* designer by delegating the handling of some naming conventions and some syntax particulars to the *configurator*. One example is the space of community values. A community value consists of two 16-bit values X and Y written as X:Y. Currently the convention is that communities X:Y are reserved for AS X. Given that an AS may want to support a whole range of community sig-

Fragment	<fragment>
Name	<name></name>
RPSL realization	<rpsl>
Filter specification	<filter></filter>
Action specification	<action></action>
	</rpsl>
IOS realization	<ios>
Session selection/parameter	<bgp></bgp>
Filter/action spec.	<route-map></route-map>*
Filter list spec.	<filter-list></filter-list>*
	</ios>
JUNOS realization	<junos>
Session selection/parameter	<protocols></protocols>*
Filter/action spec.	<policy-options>
	</policy-options>*
	</junos>
	</fragment>*

Fig. 8. Back-end module XML base elements

naled services one needs an easily adaptable mapping between the service specifics and community values that can be used for *fragment* specification. The first approach would be to use variables. Yet, since the result can depend on another variable, we rather use a functional notation: e.g., `community_value(...)`. The parameters to the "function" can be variables, that access parameters from the *network module*. This is useful, e.g., when marking routes by the region in which they entered the AS via `community_value($session.region)`. Note, that the notation as function does not imply that it has to be realized as a function! Even though the specific value, e.g., for community-pattern-matching-expression, might depend on the vendor OS, we do not need to pass the OS as a parameter, since this information is available to the *configurator* via the *network module*. Another problem that can be circumvented using the functional notation is that for some vendors and some purposes, e.g., community filter list, one needs consistent names. Those can be generated in the same fashion, just using a different function name, e.g., `community_filter_name("black-hole")`. Another particularity is that, in writing policies, one does not want to distinguish, if a parameter used as a filter imposes a restriction to a specific prefix set, or an AS set, or is an expression on the AS path, etc.. Yet, for most vendors each of the above filters requires a different syntax. We solve this problem in the same manner as the naming problem, by using a functional notation of the following form: `filter($blackhole_set)`.

Given that each *fragment* has to contain the templates for each vendor, the top level XML elements currently include `<rpsl>`, `<ios>`, and `<junos>`. RPSL is treated as just another vendor. Once the system supports additional vendor OSES this

```

<fragment>
<name>ingress_blackhole_community<\name>
<rpsl><action>next-hop=172.24.42.172;
community.append(community_value("no-export"))</action>
<filter>community.contains(
community_value("black-hole"))</filter>
</rpsl>
<ios><bgp>
neighbor $remoteIPAddr route-map $routemapname_in in
</bgp><routemap><map>
route-map $routemapname_in permit $priority
filter($blackhole_set)
match community community_filter_name("black-hole")
set ip next-hop 172.24.42.172
set community community_value("no-export") additive
</map><routemapaction>continue</routemapaction>
</routemap>
<filterlist>
ip community-list expanded community_filter_name(
"black-hole") permit community_value("black-hole")
</filterlist></ios>
...
</fragment>

```

Fig. 9. Examples for *back-end module* entries

list has to be expanded. An overview of the elements of the *back-end module* is shown in Fig. 8. An example *fragment* is shown in Fig. 9.

For RPSL it is easy to follow the idea that each *fragment* consists of filters and actions. Indeed, `<filter>` and `<action>` are the only subelements. Each one contains the RPSL code that realizes the filter or the action. Both filters and actions can embed variable references and function calls that are resolved by the *configurator*.

In Cisco IOS the basic setup of a BGP session, including overall session parameters, is realized within the “router bgp” block of the configuration tree. “Route-map” is the syntax element for realizing filters and actions. Each routemap consists of a set of routemap entries. Each individual routemap entry allows the user to combine several kinds of different filters into one filter and to manipulate the attributes of those routes that pass the filter. A filter can, e.g., consist of a prefix list filter and a community list filter. An attribute manipulation can consist of modifying the next-hop and adding a community tag. Each routemap entry can either accept routes that pass its filters, reject routes that pass its filters, or continue the processing of the route at a routemap entry specified via the “continue” keyword. Accordingly, `<bgp>`, `<routemap>`, and `<filterlist>` are the subelements of `<ios>`. Since the “continue” feature is extremely helpful for combining routemap entries from various *fragments* into a combined routemap, `<routemap>` explicitly separates `<map>` from the `<routemapaction>`. Each entry can embed variable references and function calls that are resolved by the *configurator*.

In Juniper JunOS the high-level elements for realizing policies are “protocols” and “policy-options”. This is reflected in the subelement tags. The actual filters and actions are specified within the policy-options. The session selection and session parameter specification in addition to the specification of the order in which the policy-options are applied is done within the protocol section. Due to space constraints we do not further discuss JunOS or show *configlets* for Juniper routers.

Since there are a large number of possible kinds of *fragments* we propose to use a naming schema. One possible convention is: `<direction>_(<attribute>|<service>)_<operation>_<parameter>_<accept|deny>`. Here *direction* stands for ingress

or egress. Attributes can be AS path or MED, while “services” may be black-hole, martians, or neighbor type. Operation can be set, add, filter, remove, mark, etc.. Parameter captures if a parameter is necessary or if the action is triggered by a community. Accept is the expected default behavior while deny is explicitly stated. This results in names such as `ingress_blackhole_community`, a *fragment* which is used by the black-hole service in the ingress direction. As the route manipulation of the *fragment* is triggered by a community the parameter is community.

Since *fragments* may manipulate the same attributes it is necessary to resolve conflicts. Should one *fragment* set the value of a route attribute to X and another set it to Y then one needs some method to resolve such conflicts. We propose to solve this problem in two steps. First we partition the *fragments* along the dimensions in which route manipulation occurs. The important aspect is that if two *fragments* belong to two different dimensions than no conflict is possible. Typically a dimension corresponds to one specific attribute. Additional dimensions arise due to certain prefix filters, e.g., a martians filter and a community filter.

The second step is to capture for each dimension the relationships between the *fragments* in a partial order. If two *fragments* manipulate the same object, e.g., *fragment A* via $X = a$ and *fragment B* via $X = b$, then the result of applying *fragment A* and *fragment B* to a session will be a if *fragment A* is less than *fragment B* for attribute X and b if *fragment B* is higher in the partial order. This implies that it is not possible for two *fragments* F_1 and F_2 to both manipulate attributes A and B with partial orders: $F_1 <_A F_2$ and $F_2 <_B F_1$. This is an illegal specification and will be rejected by the *configurator*. In principle this should never occur, since it indicates that the two services, using these *fragments*, are overwriting each others settings. If this is a desired behavior, the check can be circumvented by using four *fragments* instead of two. Two *fragments* for manipulating each of the two attributes.

D. Lessons learned

The ability to delegate everything BGP-specific to the *back-end module* and everything network-specific to the *network module* enables the formulation of a routing policy at a level of abstraction previously unavailable. This is accentuated by the specifications of the sample policies, see Fig. 9.

Our approach differs from other XML based languages such as for example NETCONF [29] and NetML [34]. NETCONF uses XML requests and queries to provide a simple mechanism for managing configuration data and system state of network devices. NetML uses XML to describe computer networks in a vendor-independent manner. While both provide nice abstractions and highlight the differences in the approaches of the vendors, they do not have the ability or the goal of expressing abstract routing policy.

V. CONFIGURATOR

In collaboration with an ISP, we have developed a prototype, the *configurator*, written in Perl, that generates router *configlets* for the EBGp parts of each router in the network based on the

information provided in the XML databases. In this section, we describe its design and operation.

Syntax and semantic checks: The tool starts by parsing the XML databases and using XML schemas, using the xerces library [35]. Using XML schemas ensures that only appropriate tags are used. Furthermore, they allow us to ensure that elements that have to be present are indeed present (yet not too often), have the appropriate types and appropriate values or value ranges. In addition, we use the capability of XSD schemas to perform integrity checks of the cross-references in the *network module* database: are the router names used in the *bgpsession* elements defined; are the *bgpsession* who's names are used in the *bgpneighbor* elements valid? Other consistency checks within the modules include checks for collisions in the name space, e.g., is the same public AS number used for two different neighbors, is the same IP address used as loopback IP address for two different routers, ... The XML schemas are also the basis for ensuring consistency between the *network module* and the *policy module*. The policy database is used to generate XSD file components to check the syntax of the services elements. Next the tool performs additional consistency checks between the modules. It checks that the *fragment* and *service* definitions only reference existing variables, meaning elements in the network database, call functions that are realized in the current version of the *configurator*, and if all *fragments* exist that are used in the services, etc..

Inheritance: Inheritance is an excellent method since it simplifies the specification of the network database and it ensures that it is possible to highlight where exceptions for specific BGP sessions are necessary. The amount of overhead for the *configurator* is limited since the inheritance is static and therefore can be resolved just after performing the basic checks. The *configurator* deals with inheritance by determining for each BGP session which services with which parameters are indeed booked. By storing this information separately the *configurator* still knows which service is session-specific and which service is neighbor-specific.

Configlet generation for RPSL: In principle the generation of the RPSL documentation could be done on a per session basis. Yet, the drawback of this approach is that one cannot take advantage of RPSL's capabilities of abstraction by grouping objects using session sets or AS sets. Our goal is to use this ability to document which BGP sessions to which ASes have common parts in their policies.

RPSL offers the ability to separate route manipulations that are independent of each others using the "refine" construct. Accordingly, each "dimension" of route manipulation is realized via a refine block. (Note, that in order to accept a route there has to be a matching RPSL statement in each refine block.) As each RPSL statement has to be added to some refine block we proceed by handling dimensions within the outer loop. The next loop addresses everything that needs to be done for this dimension. We start with all *fragments* in this dimension and determine the *sessionsets* and with the *sessionsets* the services that use these *fragments*. To resolve the conflicts within each dimension the *fragments* are processed in accordance with the partial order for the dimension. Once the appropriate *sessionsets* are determined it is possible to evaluate the *condition* within

the *sessionset* to select the appropriate group of BGP sessions, construct the RPSL statements from the *fragments*, and append these to the refine block.

At this point we explore, if there is a way to partition the session group, e.g., by using RPSL session sets or by using one or more AS sets. One can use a RPSL session set if the RPSL statement for the *fragment* is the same for all sessions in the group. This is possible if the RPSL *fragment* elements do not contain any session specific variables. Otherwise we explore if it is possible to split the session group into subgroups, e.g., into AS sets. This is possible if the RPSL elements do not contain any neighbor specific variables and if the *condition* is not session specific. Otherwise even finer partitions are considered. In the worst case no subgroups exist and an RPSL statement for each session is generated. Once the subgroups, RPSL session sets, AS sets, etc., have been identified, the RPSL statement is added to the refine block after restricting it to the appropriate level.

Vendor-specific configlet generation: In contrast to the generation of the RPSL documentation the *configlet* generation for Juniper and Cisco routers is done on a per session basis. The *configlet* generation for Juniper routers proceeds in a similar fashion as the one for Cisco routers. We therefore only discuss the *configlet* generation for Cisco IOS. Recall, for Cisco routers the basic element for realizing BGP filters and actions are routemaps. Also *fragments* can be written under the assumption that the continue feature is available. The *configlet* generation for IOS proceeds in three steps: we first determine for each session which *fragments* have to be combined; next we generate a preliminary *configlet* for each session; finally, we deal with specifics of Cisco IOS and optimize the generated *configlets* with embedded RPSL expressions for filter generation.

The first step involves selecting for each `<enforced_policy>`, each `<available_service>`, and each *sessionset* within the corresponding services, which *bgpsessions* fulfill the *condition* of the *sessionset*. Next we evaluate the service parameters for each session that meets the *condition*, (in the context of this *bgpsession*) and store the result together with the *fragments* listed under `<task>` in a list of *fragments* at the session. For all sessions that do not meet the *condition*, the evaluated parameters and the *fragments* under `<default>` are stored. In effect this list gathers for each session the names of all *fragments* with their parameters that are applied to this session.

In the second step we proceed session by session and direction by direction to generate preliminary *configlets* based on the list of *fragments*. In this step the *configurator* combines the *fragments* in the appropriate order and ensures that all variables, all function calls, and all parameter values of all subelements of `<ios>` are replaced with the appropriate values. The *fragments* are processed in an order that respects the partial order of the *fragments*. For each subelement the results are joined. This is a simple merge for `<bgp>` and `<list>`. Any conflict indicates that the *fragment* designer made a mistake and the *configurator* returns an error. For "routemaps" the join is not that simple and postponed to the final processing step. At this point the new piece is just added to the pieces from the other *fragments*.

Once all *fragments* have been processed the join of the routemap entries can be completed. Just chaining the individual routemap entries according to the partial order is not sufficient.

For example, if *fragment A* and *fragment B* manipulate attribute x with $A <_x B$, one needs to ensure that attribute x will have the value of *fragment A* even if the filters in both *fragments* match. Even though the serialization ensures that the partial order is respected, it does not ensure that route filtering proceeds at the appropriate place: e.g., the first routemap entry from a *fragment* of the next dimension. This is accomplished by rewriting the continue entries of the joined routemap entry.

Unfortunately for Cisco routers the continue feature is not available for all software versions currently deployed in the Internet. Luckily, the continue functionality can be realized with routemaps without continue by computing a *convolution* of all routemap entries. A convolution explores all possible combinations of routemap entries and arranges them in an order that has the most restrictive filter at the beginning and the least restrictive one at end of the new routemap. A combination of two routemap entries is computed by joining their filters with a logical AND and their attribute manipulations with a logical OR operation. The latter is the case as long as the manipulations are orthogonal. If the same attribute is manipulated the manipulation from the routemap entry with the smaller value is chosen to ensure consistency. Unfortunately routemap entries do not support arbitrary filter combinations using logical AND operation. For example it is not possible to combine two prefix filter lists A and B using a logical AND in the same routemap entry. Instead this can be realized by generating a third prefix filter list, $C = A \text{ AND } B$, for use in the routemap entry. We solve this complication by using RtConfig to generate the appropriate filter lists, specified via RPSL statements, and adjust the routemap entries to use the new filter.

Finally, duplicate entries, filters that are no longer used, and routemap entries that are never reached are removed. For example, if routemap entry 1 uses filter A and routemap entry 2 uses filter A then the combined entry from 1 and 2 suffices. The entry for only 1 and the entry for only 2 can be removed as it will never be reached. If entry 2 uses a different filter all three routemap entries have to be present.

RtConfig: At this point the output of the *configurator* is a documentation of the routing policy in RPSL and a set of *configlets* with embedded RPSL expressions for every bgp session in the network. These embedded RPSL expressions need to be replaced with actual filter statements before the *configlet* can be uploaded to the router. For this purpose we rely on RtConfig. RtConfig is able to resolve all RPSL filter statements by using a local cache of RPSL objects for AS internal filters together with the IRR database. This implies that if a customer changes his entry in the IRR database, this change is automatically propagated to the *configlet* the next time the *configlet* is regenerated.

Upload: If there has been a change in the *configlet* it needs to be uploaded to the appropriate router. For this purpose it is combined with some base configuration code which ensures redistribution of routes, deals with static routes, etc.. The upload for Juniper router is nicely supported by their release management. The upload to Cisco routers is more troublesome and involves using a set of AS internal tools for uploading the *configlet* and then soft-resetting the appropriate BGP sessions.

```

aut-num: AS1
import: from AS-ANY accept ANY;
refine { # from ingress_route_filter:
  from AS2 accept AS2:RS-I;
  from AS3 accept RS-AS3-IMPORT;
  from AS2 accept RS-AS3-BLACKHOLE AND community.contains(65000:0);
} refine { # from blackhole
  from AS3 2.1.1.2 at 1.0.0.2 action next-hop=172.24.42.172;
  community.append(no_export);
  accept community.contains(65000:0) AND RS-AS3-BLACKHOLE;
  from AS-ANY accept not community.contains(65000:0);
} refine { # from ingress_peer
  from AS-CUST action community.append(1:1) accept ANY;
  from AS-PEER action community.append(1:2) accept ANY; }
export: to AS-ANY announce ANY;
refine { # from egress_route_filter:
  to AS2 announce AS2:RS-E;
  to AS3 announce RS-AS3-EXPORT;
} refine { # from egress_wellknowncommunities_filter
  to AS-ANY announce not community.contains(no_export, no_advertise);
} refine { # from peering
  to AS-CUST announce community.contains(1:0) OR
  community.contains(1:1) OR community.contains(1:2)
  OR community.contains(1:3);
  to AS-PEER announce community.contains(1:0)
  OR community.contains(1:1);
} refine { # from egress_med
  to AS2 2.1.1.2 at 1.0.0.2 action med=2000; announce RS-2000;
  to AS2 2.1.1.2 at 1.0.0.2 action med=3500; announce RS-3500;
  to AS2 2.1.1.2 at 1.0.0.2 action med=100; announce ANY;
  to AS-ANY action med=500; announce ANY; }

```

Fig. 10. Generated documentation in RPSL.

VI. OPERATIONAL CONSIDERATIONS

In this section we show examples of generated *configlets* and discuss some challenges that an operator faces.

A. Generated configlets: Examples

For the example of Fig. 3 the generated RPSL documentation is shown in Figure 10. The various dimensions are apparent in the different refine blocks for both directions, e.g., for ingress the route filtering is handled in a different dimension than the black-hole service. The differences between the default action and those based on the condition are easily seen for the black-hole service. If the black-hole service is not booked or used for a route that is not in the appropriate routeset (RS-AS3-BLACKHOLE), then the route is rejected. Otherwise the appropriate action is taken. The realization of different cases is nicely highlighted by the handling of med for AS 2 on session c2. Here, the value is set to 100 per default. If the route is part of the RS-2000 the value is 2000, if the route is part of RS-3500 the value is 3500. For all other sessions the med is set to 500. As the example is constructed to enable different services for different sessions, the *configurator* has only limited success in using session sets, e.g., AS-CUST and AS-PEER (from the peering policy) each consisting of a single AS.

The main difference between the generated RPSL and the vendor-specific code is, that in RPSL the full EBGP policy of the whole network can be specified within one statement. Of course it is also possible to create an import and an export statement for each session, but this would not take advantage of the features of RPSL. At the moment RPSL has the drawback that it is not capable of matching or deleting communities specified by patterns. All communities have to be listed explicitly. In some cases this is not feasible, e.g., the cleaning of internal communities at ingress. Therefore we currently lose some functionality.

The generated code for a Cisco router for BGP session c1

```

router bgp 1
 neighbor 2.1.1.2 remote-as 2
 neighbor 2.1.1.2 next-hop-self
 neighbor 2.1.1.2 route-map c1_routemap_in in
 neighbor 2.1.1.2 route-map c1_routemap_out out
!
route-map c1_routemap_out deny 100
 match ip address prefix-list martians
route-map c1_routemap_out permit 200
 set comm-list out_fltr_communities delete
 continue 300
route-map c1_routemap_out permit 300
 match community export_all
!
route-map c1_routemap_in deny 500
 match ip address prefix-list martians
route-map c1_routemap_in permit 600
 set comm-list in_fltr_communities delete
 continue 700
route-map c1_routemap_in permit 700
 set community 1:1 additive
 continue 800
route-map c1_routemap_in permit 800
 match ip address prefix-list c1-blackhole
 match community blackhole
 set ip next-hop 172.24.42.172
 set community no-export additive
 continue 900
route-map c1_routemap_in permit 900
 match ip address prefix-list c1-import
route-map c1_routemap_in permit 910
 match ip address prefix-list c1-blackhole
 match community blackhole
!
ip community-list expanded in_fltr_communities permit _1:.*_
ip community-list expanded in_fltr_communities permit 64900:.*
ip community-list expanded out_fltr_communities permit 64900:.*
ip community-list expanded blackhole permit 65000:0_
ip community-list expanded export_all permit _1:[0123]_
!
ip prefix-list c1-import permit 2.1.1.0/22 ge 24 le 24
ip prefix-list c1-blackhole permit 2.1.1.0/24 ge 32 le 32
ip prefix-list c1-blackhole permit 2.1.1.0/24
ip prefix-list martians permit ...

```

Fig. 11. Generated *confi glet* for c1 using routemap feature continue.

on r1 is shown in Fig. 11. Each of the different *fragments* contributes some pieces to the “router bgp” and “route-map” subsection of the *confi glet*. The first entry in the routemap_out part is the result of martians filter. The second one is the result of clearing internal communities. The final part is realizing the appropriate part of the peering policy from *fragment*: egress_allow_all. Most of the match statement need some filter which can be found in the appropriate sections. The routemap for ingress filtering has a few additional pieces: entry 700 marks the received routes as coming from a customer (community 1:1), entry 800 realizes black-holing. First the next hop is rewritten, then the black-hole community is added to the route (as routemap entry 600 deletes all communities). Note, that a route can be accepted if it is in the regular routeset or if it is in the black-hole routeset.

To highlight the differences between the generated code for an IOS version that does not support the continue feature Fig. 12 shows the route-maps of the generated code for the same session. Now each route-map entry consists of a self-contained set of route manipulations that are appropriate for the filters. In this case, the order of the route-map entries is crucial. It goes from the most restrictive filter to the least restrictive one. One needs an additional filter with c1-import-c1-blackhole which is the result of c1-import AND c1-blackhole.

```

route-map c1_routemap_in deny 100
 match ip address prefix-list martians
route-map c1_routemap_in permit 76
 match ip address prefix-list c1-import-c1-blackhole
 match community blackhole
 set comm-list in_fltr_communities delete
 set ip next-hop 172.24.42.172
 set community 1:1 no-export additive
route-map c1_routemap_in permit 77
 match ip address prefix-list c1-blackhole
 match community blackhole
 set comm-list in_fltr_communities delete
 set ip next-hop 172.24.42.172
 set community 1:1 no-export additive
route-map c1_routemap_in permit 78
 match ip address prefix-list c1-import
 set comm-list in_fltr_communities delete
 set community 1:1 additive
!
route-map c1_routemap_out deny 100
 match ip address prefix-list martians
route-map c1_routemap_out permit 15
 match community export_all
 set comm-list out_fltr_communities delete
!
ip prefix-list c1-import-c1-blackhole permit 2.1.1.0/24

```

Fig. 12. Generated *confi glet* for c1 after convolution (without continue).

B. Experiences

Our system is in production use at a large ISP. It is used for generating the *confi glets* for the EBGp sections of several 100 routers and it has proven to be beneficial to consider the routing policy as a first class entity. Each service and each policy is now well specified, which increases the degree of transparency for the ISP. In addition, it is easy to add, expand, and change services and policies.

The flexibility of the system proved to be crucial during the migration phase: moving from a “legacy” state to an automatically generated state. All of the network specific information had to be gathered for the *network module*. In addition, the routing policy in use has to be expressed using our abstraction. Here, the capability provided by the *fragments* of the *back-end module* proved to be extremely useful. The ability of using almost arbitrary router configuration commands simplified reproduction of the legacy state. Once the legacy state could be regenerated and uploaded to the routers the next phase of transitioning to the desired routing policy was possible. This involves, as a first step, the definition of the desired routing policy. After definition each of the policies and services has to be introduced into the network. This can involve multiple stages. After all it is impossible to change the configuration of all routers at exactly the same time. If the policy or service depends on some precondition, e.g., that all routes are marked with a community on ingress, then this precondition has to be introduced on all sessions before, in a second stage, the postcondition can be introduced, e.g., that routes are filtered based the community.

Specifying the desired routing policy is still not trivial as the system does not restrict the routing policy designer in realizing various variants. Among the choices to consider are: how to realize filters, e.g., prefix and community filters. Does the policy include the acceptance of all routes that are not filtered due to some reasons or another? Are all inappropriate communities removed in one filter or should each service, that is signaled via a community, implement its part of the community filter? It is possible to realize any of these options within our system. It is up to the policy designer to make a choice. Each option

has certain implications. For example filtering communities in one policy reduces the number of routemaps needed for the IOS *configlet*. But with JunOS realizing such a feature is a bit cumbersome.

We found that using vendor-specific code inside the *fragments* is useful as the designer has complete control over the generated *configlets*. *Fragments* can be extended, rewritten, repartitioned, and rearranged as needed. One aspect to be aware of is the choice of embedding more functionality inside a single *fragment* vs. using multiple *fragments*. The first enables more control over the generated *configlet* and therefore its optimality, while the second improves debuggability and reusability. Most policies and services can be realized using just one, two, or at most three *fragments*, indicating that this is an appropriate level of abstraction for route manipulations. Using the suggested naming schema is helpful for associating *fragments* with dimensions and expressing the conflicts between the *fragments*, currently the most awkward part of the system.

Separating the network-specific parts into its own database has proven to be essential as it separates the task of the network administrator from those of the routing policy designer. In addition, the documentation of the current state of the network as well as the routing policy (in the *policy module* as well as in RPSL) is extremely helpful.

VII. RELATED WORK

Work towards systems that help ISPs manage and structure their policies exists in a manifold diversity. Let us start with tools that are able to manage routers or switches in an automated or semi-automated fashion. Such tools, range from free software scripts, like RANCID [36], to commercial products from companies such as AlterPoint, Cariden, CPlane, Gold Wire Technology, Intelliden, Network-Clarity, OpNet, Rendition Networks, Tripwire, Voyence, Wandl, etc. This set includes products from router vendors themselves, e.g., [37]. The goals of such utilities are to help administrators to reduce manual configuration and provisioning tasks, and this is claimed to lead to a minimization of human errors, which in turn reduces potential downtime. Such tools can help with keeping track of router updates, maintain consistency across similar devices, and document critical changes automatically. Furthermore, they provide more security through tight access controls and configuration audits. Some even include mechanisms to optimize packet traffic flows. While most products have the capability to manage multiple devices in the network at the same time, most of them focus on direct device management, e.g., the routers. They allow, e.g., setting up MPLS/VPNs, or downloading previously stored configurations. While most of these products are targeted towards enterprise solutions, some of them are developed for the needs of Internet service providers. The reason that only very few of the commercial tools are designed for the needs of an ISP is that those policies are often complex and do not fit any pre-defined specifications. Therefore most ISP rely on engineers for maintaining and realizing their routing policy. To the best of our knowledge, this task is often accomplished by using a combination of small utilities, including homegrown scripts, templates, and human experience.

Recent work by Caldwell et al. [11] shows that automated

network-wide policy configuration should be viewed within a larger framework. Gottlieb et al. [13] describe an automated provisioning system for BGP. How to migrate, store and analyze router configuration is presented in [10], [11].

A good overview about how network operators are using BGP to realize various policies in their network (such as traffic engineering) can be found by Uhlig et al. [38]. On the other hand the implications of certain routing policies on the control and data plane can be difficult to predict [39], [40]. To gain a better understanding about the control plane interactions within the Internet, some projects monitor the routing system, e.g., [41], [18], and try to validate the routing policies in use, e.g., [12], [42], while others try to understand their interactions via simulation, e.g., [43].

VIII. SUMMARY

In the past networking research has tended to focus on the individual parts of the network rather than on the network as a whole. In this paper we propose a system that can help raise the level of abstraction for a small but crucial piece of the overall Internet: the routing policy of one AS.

For this purpose we tackle the problem of identifying the principles underlying a network-wide routing policy. Based on this understanding we propose a system for managing network-wide routing policies as first class entities. This includes providing a flexible, extensible, and scalable framework for formulating the routing policy, eliminating the need of manual configuration of IP routers for the purpose of realizing the policy, and respecting the division of responsibilities within the ISP.

As such we have presented a data model that enables the definition of an abstract routing policy but also allows its concrete realization on the routers in the network. It is now possible to express the enforced policies and available services of a routing policy precisely as well as in a compact format. The experiences gathered from the production use at a large ISP have shown that transitioning from a network configuration grown bottom up to one with a well documented routing policy that is specified top down is possible.

Clearly, the work reported herein has not exhausted the problem area, and there is much more that can be done. One limitation of the current system is that it does not have support for IBGP. Other avenues for expanding the capability of the system are support for IPv6 and VPNs as well as support for additional router vendors.

Yet, the network-wide routing policy is only a small piece of the management of an ISP and one question that arises if a similar approach to ours can be applied to other parts of the configuration management. Moreover it will be interesting to see, if our system can help in improving the management of the routing policy not just for individual ASes but overall. A first step is to improve the documentation of the routing policies currently used in the Internet, e.g., via the IRR [16]. Once available such a documentation could be the basis of some logic, e.g., [25], for localizing possible routing instabilities, e.g., [8], [9].

APPENDIX

The policy information is provided to the router via a router configuration language. It can be altered by applying commands

to a router via its operating system, e.g., Cisco IOS [28]. In this paper we are in particular interested in the EBGp parts of the configuration (e.g., see Fig. 11, 12). The “router bgp” entry identifies the AS number and may include a number of commands that enable/disable certain features. In addition, the “neighbor” commands are used to configure EBGp sessions with a router in another AS. Each BGP session is associated with import and export statements, specified via route-maps that appear later in the configuration. A route-map consists of a set of route-map entries, such as filters or attribute manipulations. If a router receives a route over a BGP session with an associated import route-map it processes the route-map entries one by one until one of the filters matches the current route. As long as the route-map statement does not contain the keyword “continue” the attribute manipulations are applied to the route and the route is either accepted via the keyword permit or denied via the keyword deny. If the route-map entry contains the continue keyword with value 300 attribute manipulation are stored in a todo list and processing continues with the route-map entry 300.

Another way of specifying policies is via RPSL [19]. The advantage of expressing policies in RPSL is that the language is vendor-independent while allowing similar expressiveness in terms of attribute modifications and filtering. Tools like RtConfig [31] can convert RPSL expressions to vendor-specific configuration code. Besides policies, various kinds of network administrative objects can be handled with RPSL. There is a way to look up all objects related to an AS. For example a route-object contains the origin AS. Via this mechanism it is possible to find all prefixes that are potentially announced by an AS. There are several ways to group objects into sets, in the case of route objects this is called a route-set. Routes and route-sets can be referenced when creating filters. Additionally filters can be based on any available BGP attribute, including communities and as-path, and combined via AND, OR and NOT. They can be specified inline or stored in filter-sets, which combine one or more filters. Such filters can be referenced when specifying the routing policy.

The specification of import and export policies is done within the aut-num class, which represents one AS. In RPSL the components of policies are called factors and are of the form:

```
import: from <peering> [action <action>];
      accept <filter>;
```

<peering> specifies for which sessions this factor should be applied. <action> (optionally) specify the actions, which should be taken, if the factor matches. Only updates which match <filter> are accepted. If there are more than one import/export factors, they are searched according to their order for a match. All latter and all not matching leading factors are ignored. There is no explicit deny within RPSL: updates that match a rule are accepted, all others are dropped.

As there are various policies that manipulate various attributes, it is necessary to match multiple factors at the same time, each responsible for handling a different attribute. In RPSL this is addressed via the refine operator, used for building structured policies. Multiple factors are combined by merging them into a block. The resulting policy is evaluated as follows: For each block the first matching factor is located. If there is no match within one block, the whole update is rejected. Oth-

erwise, all blocks match, and the update is accepted and the actions are applied according to their specified order.

REFERENCES

- [1] D. A. Maltz, G. Xie, J. Zhan, H. Zhang, G. Hjalmytsson, and A. Greenberg, “Routing design in operational networks: A look from the inside,” in *Proc. ACM SIGCOMM*, 2004.
- [2] Y. Rekhter and T. Li, “A Border Gateway Protocol 4,” 1995. RFC 1771.
- [3] J. W. Stewart, *BGP4: Inter-Domain Routing in the Internet*. 1999.
- [4] R. Chandra, P. Traina, and T. Li, “BGP Communities Attribute,” 1996. RFC 1997.
- [5] B. Raghavan and A. C. Snoeren, “A system for authenticated policy-compliant routing,” in *Proc. ACM SIGCOMM*, 2004.
- [6] G. Huston, “NOPEER Community for Border Gateway Protocol (BGP) Route Scope Control,” 2004. RFC 3765.
- [7] O. Bonaventure and B. Quoitin, “Common utilizations of the BGP community attribute,” June 2003. Internet Draft.
- [8] T. Griffin and G. Huston, “BGP Wedgies,” 2004. Internet Draft.
- [9] D. McPerson, V. Gill, D. Walton, and A. Retana, “Border Gateway Protocol (BGP) Persistent Route Oscillation Condition,” 2002. RFC 3345.
- [10] A. Feldmann and J. Rexford, “IP network configuration for interdomain traffic engineering,” *IEEE Network Magazine*, 2001.
- [11] D. Caldwell, A. Gilbert, J. Gottlieb, A. Greenberg, G. Hjalmytsson, and J. Rexford, “The cutting EDGE of IP router configuration,” in *ACM SIGCOMM HotNets Workshop*, 2003.
- [12] N. Feamster and H. Balakrishnan, “Detecting BGP Configuration Faults with Static Analysis,” in *Proc. USENIX/ACM NSDI*, 2005.
- [13] J. Gottlieb, A. Greenberg, J. Rexford, and J. Wang, “Automated provisioning of BGP customers,” *IEEE Network Magazine*, 2003.
- [14] D. Wetherall, R. Mahajan, and T. Anderson, “Understanding BGP misconfigurations,” in *Proc. ACM SIGCOMM*, 2002.
- [15] North American Network Operators Group. <http://www.nanog.org/>.
- [16] Internet Routing Registry. <http://www.irr.net/>.
- [17] R. Govindan, C. Alaettinoglu, G. Eddy, D. Kessens, S. Kumar, and W.-S. Lee, “An architecture for stable, analyzable Internet routing,” *IEEE Network Magazine*, 1999.
- [18] G. Siganos and M. Faloutsos, “Analyzing BGP Policies: Methodology and Tool,” in *Proc. IEEE INFOCOM*, 2004.
- [19] C. Alaettinoglu, C. Villamizar, E. Gerich, D. Kessens, D. Meyer, T. Bates, D. Karrenberg, and M. Terpstra, “Routing Policy Specification Language (RPSL),” 1999. RFC 2622.
- [20] D. Meyer, J. Schmitz, C. Orange, M. Prior, and C. Alaettinoglu, “Using RPSL in practice,” RFC 2650.
- [21] C. Labovitz, R. Malan, and F. Jahanian, “Origins of Internet routing instability,” in *Proc. IEEE INFOCOM*, 1999.
- [22] Z. M. Mao, R. Bush, T. G. Griffin, and M. Roughan, “BGP Beacons,” in *Proc. ACM IMC*, 2003.
- [23] A. Feldmann, O. Maennel, M. Mao, A. Berger, and B. Maggs, “Locating Internet Routing Instabilities,” in *Proc. ACM SIGCOMM*, 2004.
- [24] Tim Griffin’s Interdomain routing links. <http://www.cl.cam.ac.uk/users/tgg22/interdomain/>.
- [25] N. Feamster and H. Balakrishnan, “Towards a Logic for Wide-Area Internet Routing,” in *Proc. ACM SIGCOMM FDNA Workshop*, 2003.
- [26] G. Huston, “Interconnection, Peering, and Settlements,” in *Internet Protocol Journal*, 1999.
- [27] D. Turk, “Configuring BGP to Block Denial-of-Service Attacks,” 2004. RFC 3882.
- [28] Cisco, *Cisco IOS Configuration Fundamentals*. Cisco Press, New Riders Publishing, 1998. Documentation from the Cisco IOS reference Library.
- [29] R. Enns, “NETCONF Configuration Protocol,” 2004. Internet Draft.
- [30] “Introduction to Configuring JUNOS Internet Software.” http://www.juniper.net/training/elearning/junos_cli/.
- [31] C. Alaettinoglu, “Rtconfig: Router configuration generator,” February 1996. NANOG 6.
- [32] IRRToolSet. <http://www.isc.org/sw/IRRToolSet/>.
- [33] C. Villamizar, C. Alaettinoglu, D. Meyer, and S. Murphy, “Routing Policy System Security,” 1999. RFC 2725.
- [34] NetML (an XML specification language to configure and express network devices). <http://giga.dia.uniroma3.it/~ivan/NetML/>.
- [35] Xerces. <http://xml.apache.org/xerces-p/>.
- [36] RANCID - Really Awesome New Cisco config Differ. <http://www.shrubbery.net/rancid/>.
- [37] CISCO IP Solution Center. <http://www.cisco.com/en/US/products/sw/netmgtsw/ps4748/index.html>.
- [38] S. Uhlig and O. Bonaventure and B. Quoitin, “Interdomain Traffic Engineering with minimal BGP configurations,” in *Proc. of the 18th International Teletraffic Congress, Berlin*, 2003.

- [39] S. Agarwal, C.-N. Chuah, S. Bhattacharyya, and C. Diot, "The Impact of BGP Dynamics on Intra-Domain Traffic," in *Proc. ACM SIGMETRICS*, 2004.
- [40] S. Uhlig and O. Bonaventure, "Implications of interdomain traffic characteristics on traffic engineering," *European Transactions on Telecommunications*, January 2002.
- [41] University of Oregon RouteViews project. <http://www.routeviews.org/>.
- [42] N. Feamster, "Practical verification techniques for wide-area routing," in *Proc. ACM Workshop on Hot Topics in Networks*, 2003.
- [43] S. Uhlig and B. Quoitin, "Tweak-it: BGP-based Interdomain Traffic Engineering for Transit ASes," in *Proc. of the first EuroNGI Conference on Next Generation Internet Networks*, 2005.