A High-level Network-wide Router Configuration Language

Miroslav Sveda Michal Sekletar Tomas Fidler Ondrej Rysavy
Faculty of Information Technology
Brno University of Technology,
612 66 Brno, Czech Republic
e-mail:{sveda, xsekle00, xfidle01, rysavy}@fit.vutbr.cz

***Abstract*—In this short paper, we discuss the design of a high-level network-wide router configuration language. At its current stage of development, the language enables us to specify basic routing and security configurations. A declarative nature of the language is supposed to be intuitive to network administrators. We have developed an experimental compiler that produces configuration files for Cisco routers. The contribution of the paper consists of the description a language for configuration programming and the demonstration of its capabilities on several examples.**

***Index Terms*—network configuration management; routing configuration; access control lists**

I. Introduction

Configuration languages for network devices enable to define every aspect of their functionality. Network administrators can thus write a network configuration that meets the required functionality for different and often very specific requirements. These languages have a simple declarative form. A network configuration consists of configuration files of all devices in a network. The difficulty in implementation of a correct network configuration stems from the necessity to create several separate configuration files that need to be consistent.

To overcome the difficulty in delivering consistent set of configuration files, device vendors provide tools implementing configuration wizards, web configuration interfaces or configuration generators. These tools may simplify basic configuration tasks but usually do not provide any additional mechanisms to guarantee configuration consistency and correctness. An alternative approach is to use high-level configuration languages.

This idea is behind the design of Nettle language [1], which is a domain-specific high-level language for BGP configurations. The other example is the Flow-based Management Language by Hinrichs et al. [2], which is a declarative policy language supposed for developing configurations for enterprise networks. It covers ACL, VLAN, NAT, policy routing and admission control features. However, the specified configuration is compiled only for the NOX platform. Our motivation is to define a high-level router configuration language that can be compiled to common router configuration languages of devices deployed in present enterprise networks.

In this short paper, we present the design of a high-level network configuration language. Currently, the language allows us to specify a limited set of network configurations, which includes network address assignment, static and dynamic routing and basic security. We also implemented an experimental compiler that produces IOS configuration files for Cisco routers. To be practically usable, the language needs to support other configuration features and to generate configuration for other network devices.

The structure of the paper is as follows. The next section describes a syntax of the language and illustrates its usage on simple examples. Section III briefly describes an implementation of the language compiler and the IOS configuration generator. Finally, section IV concludes the paper by summarizing the current state and discussing the future development.

II. The Language

The language consists of a set of simple declarative statements and an embedded expression language. The statements are grouped in different configuration sections depending on their purposes. Currently, we have defined and implemented rather a small subset of such language, which we present in this paper on a series of examples specifying network devices, routing areas, network areas, network connections, routing options and security policies.

The expressions can be embedded in declarations. In expressions we can refer to declared elements and predefined methods. In the future we plan to extend the language with possibility of defining user methods. It is important to note that these expressions represent the side effect free computations. The proposed expression syntax reassembles the syntax of object-oriented languages. We use dot notation to access fields of objects and call their methods. Usually, an expression is to be evaluated to a collection, a simple value or a structure, which can be inferred by type checking. Types of results have to conform to expected types of surrounding contexts. Declarations define attributes that can be accessed from expressions as shown in several examples in the rest of this section.

Each declaration block specifies a certain part of a network configuration, e.g., routing, address assginement, etc. To be treat as a programming language statement, it can be viewed as a macro definition, which evaluates to a corresponding program block. For instance, a device list from the next subsection can be viewed as the following list definition[1]:

```
var Devices = new[] {
  new Router("Austin", "A", "cisco_2811"),
  new Router("Dallas", "D", "cisco_2811"),
  new Router("Houston", "H", "cisco_2811")
};
```

Another characteristic of the language is that for specifying packet forwarding and filtering, a flow-based description [2] is employed.

[1]We use C# syntax in this example representation.

 ISBN: 978-1-61208-183-0

A. Device List

A device configuration group enumerates all devices in the configured network. A device declaration assigns a specific device type to each router, which tells the compiler what generator should be used for generating a configuration file. The compiler can be extended by custom generators for new platforms and models.

```
Devices {
    Router Austin[A] cisco_2811;
    Router Dallas[D] cisco_2811;
    Router Houston[H] cisco_2811;
}
```

The presented configuration snippet declares three routers appearing in Texas area and specifies their hardware platforms. Together with the full router name we may provide its short name that can be used for referring to the router from other places of the configuration file. For the language of expressions, this declaration defines a collection called `Devices`, which consists of three objects of type router. `Router` class is one of the classes derived from `Device` abstract class. Another derived classes could be `Switch`, `Gateway`, etc.

Currently, a device type specification consists of an enumeration of all device's interfaces, as shown in the following example:

```
Device cisco_2811 {
    PORT    Serial0/0/0 s0/0/0;
    PORT    Serial0/0/1 s0/0/1;
    PORT    FastEthernet0/0 fa0/0;
    PORT    FastEthernet0/1 fa0/1;
}
```

A device type specification is compiled into a plug-in module that is used by the language compiler for generating a device configuration for the specified device type.

B. Area List

The purpose of an area list is to define routing areas. Each routing area consists of routers which run the same instance of a routing protocol. The following is a definition of three different areas:

```
Areas {
    AREA {A, D, H} {A} RIP Texas;
    AREA {A,Tampa,M,T} {A,T} EIGRP Florida;
    AREA {R,S,T} {T} OSPF Washington;
}
```

Each area declaration consists of a list of area routers, a list of border routers, a definition of a routing protocol and a name of the area. A non-empty intersection of sets of border routers denotes routers where the redistribution between routing protocols can be configured. The redistribution options are stipulated in a routing configuration section.

C. Network List

A network list enumerates all destination networks. Each network declaration defines a network address and a network name, as shown in the following example:

```
Networks {
    Intermediate 192.168.1.0/30 Dallas-Austin;
    EndUser 192.168.2.0/24 management;
    EndUser 192.168.3.0/24 servers;
}
```

The network list does not include interconnecting networks except if these networks are significant from the viewpoint of routing or security configurations. The unspecified interconnecting networks are listed in a connection configuration block.

D. Connections

Connections among routers and destination networks are specified in a connection list. Point-to-point and point-to-multipoint connections can be distinguished. Following example contains several kinds of connections:

```
Links {
    A.s0/0/0 -> D.s0/0/0  Austin-Dallas;
    A.fa0/0 -> TERM       management;
    D.fa0/1 -> DEFAULT    ISP;
    Tampa.fa0/0 -> SWITCH Florida-Net;
    Miami.fa0/0 -> SWITCH Florada-Net;
}
```

A connection is specified by its endpoints and its name. An endpoint is either a router interface or one of the following keywords:

- `DEFAULT` denotes that the connection represents a default gateway for the network,
- `TERM` denotes that a router interface is connected to a destination network and there are no other routers in this network, and
- `SWITCH` denotes a router interface connected to a port of a switch.

Note that addresses of interfaces are generated automatically by the compiler. For interconnecting networks these addresses are taken from the pool of addresses that can be defined by the user.

E. Routing

A dynamic routing configuration is implicitly defined by specifying routing areas. In a routing configuration block, static routing, redistribution and other routing related options can be defined to customize the network routing. A following example shows redistribution of routing information from Florida routing instance to Texas routing instance. The redistribution is performed at Austin router, which runs both routing protocols. All routing information on end user networks maintained by the EIGRP in Texas routing domain will be copied to the RIP with a specified metric.

```
Routing {
    REDISTRIBUTE  Florida -> Texas
      END_USER_NETWORK METRIC 5;
}
```

Keyword `END_USER_NETWORK` selects what information is to be redistributed. In the presented example, all destination networks will be redistributed. At this position an arbitrary predicate that selects a set of redistributed networks can be used. For instance, we may write the following configuration:

 ISBN: 978-1-61208-183-0

```
Routing {
  REDISTRIBUTE  Florida -> Texas
    {Networks.Select(n => n.Name.StartsWith("D"))}
    METRIC this.Network.Name.Length;
}
```

The redistribution predicate selects all networks which names begin with letter 'D'. It means that when compiling and generating output for this statement, the expression is evaluated and replaced with a set of networks satisfying this condition. This configuration uses a weird policy for setting metrics. For each network, a metric is set to a value which equals to the length of its name. While this particular example is not very useful in practice, it demonstrates the use of `this` statement that refers to an object in which scope this expression occurs.

A purpose of a static routing configuration is to define preferred paths for network traffic. This is defined separately for each destination network. An example of static configuration is presented for `HoustonNet`. The static routing configuration consists of a subset of network links.

```
Routing {
  STATIC HoustonNet
    { Austin -> Houston,
      Tampa -> Austin,
      Miami -> Tampa }
}
```

It is also possible to apply a predefined algorithm to compute the best paths with respect to given criteria. The following configuration snippet demonstrates this approach:

```
Routing {
  STATIC HoustonNet
    SpanningTree(HoustonNet).Edges.Select
    (e => e.Contains(Austin|Houston|Tampa|Miami))
}
```

For computing the set of links we use `SpanningTree` algorithm, which computes a minimum spanning tree for `HoustonNet`. Then resulting set of links are filtered and only links which begin or end in one of four specified routers are kept in the configuration.

F. Security

Routers implement security policy by filtering traffic according to filtering rules maintained in access control lists (ACL). The configuration language is able to specify a security policy and the compiler generates ACLs and assigns them to appropriate interfaces.

First, a set of interesting flows is enumerated in a flow declaration block:

```
Flows {
  Web tcp any:any -> public:80;
  Mail (s => tcp any:any -> s:25);
}
```

Currently, flows are represented as tuples consisting of five components, namely, protocol type, source address, source port, destination address and destination port. Flows can be parametrized as can be seen in the case of `Mail` flow.

In filtering section, it is specified, which flows are permitted or denied on a particular link. In the following example, only web and mail traffic is permitted. Mail flow is instantiated with `TexasMail` server.

```
Filtering {
  Austin-Dallas {
    allow Web,
    allow Mail(TexasMail),
    implicit deny
  }
}
```

While flow-based security management brings a benefit of simplifying the implementation of security policy, it does not guarantee the correctness and consistency, because one still needs to pair filters with network locations. Along the line of proposals described in [3], [4], [5], [6], we would like to research the possibility to infer security implementations from high-level security policy specifications. Currently, we have attempted to apply techniques for filter consistency verification [7], [8], [9]. We implemented a simple tool, which reports conflicts for the given set of ACL rules. The employed method is based on work reported in [10].

III. Implementation Notes

We have implemented an experimental compiler and a configuration generator in the C++ language. Except the STL library, the compiler depends on the BOOST library, which provides data types and methods for manipulating advanced data structures. The configuration processing consists of the following steps:

1) Parsing an input configuration and generating a *network configuration object model*. This model is a structured description of parsed configuration amenable to further analysis.
2) Evaluation of expressions in the object model. The expressions are replaced with results yielded from their evaluation. After evaluating all expressions we obtain a concrete model.
3) Optional static analysis of the model. For instance, we may run an ACL conflict detection algorithm.
4) Generation of device configurations using plugins for registered device types. Based on the model, the tool generates for every known device its partial configuration by using the corresponding plugin.

Currently, the tool contains plugins only for a few devices and the expression language has a very limited form. In the future, we plan to extend the tool in both directions.

IV. Summary

In this short paper, we presented work in progress, which aims at the definition of a high-level network configuration language and the implementation of its compiler. The compiler produces device configuration files and it is extensible for different vendors and different router models. As it can be seen from the brief language description, the current state provides the basic functionality. The language is able to describe an enterprise network as a collection of devices and routing areas, to generate address assignment and to define basic security policies. The future work is focused on extending

 ISBN: 978-1-61208-183-0

the language with other features, e.g. NAT configuration, VLAN definitions, VPN configuration, policy routing, etc. For a first experimental implementation we decided to implement configuration generator only for CISCO devices. Currently, we are working on the support for other platforms.

The presented approach is directly comparable to Nettle language [1] and the FML [2]. These languages attempt to define a network configuration by specifying which services should be available rather than encoding the network behavior by using low level configuration commands. Nevertheless, there are other methods that simplify the network configuration. From industrial perspective, the major achievement in this area has been made by XML-based network configuration methods and protocols [11]. For instance, Juniper Networks introduced a Network Configuration Protocol called NETCONF, which was standardized by IETF as RFC4741. The protocol provides mechanisms to install, manipulate, and delete the configuration of network devices. The aim of Network Description Language (NDL) [12] is to simplify a description of networks and configurations by creating the ontology for computer networks based on the Resource Description Framework (RDF).

Our approach goes beyond merely introducing a new language for describing network configurations. Rather, we would like to construct network configurations by using a high-level configuration programming language. For this we set foundations in the presented language, which supports declarative statements with embedded expressions increasing the expressiveness and minimizing the need to repeatedly write routine configuration statements. There is a similarity to TCL scripting in IOS configuration, which can be employed to automatize certain tasks. Nevertheless, this scripting is rather limited to a single device.

The proposed language contains also concepts known from network configuration management tools. The NetScope toolkit [13], for instance, integrates topology model, traffic, and routing based on flows. It visualizes traffic and enables users to determine effects of configuration changes before they are applied to a real network. For security specifications, our language employs ideas described by Guttman in [14]. In particular, we attempt to generate access control lists from a security policy specifications.

The other line of research has been focused on configuration synthesis. Tools such as ConfigAssure [15] are able to refine or generate configurations for network devices based on a predefined configuration database and given constraints. This approach requires the implementation of advanced reasoning methods that perform model-finding. The goal of our tool is similar, but we employ less sophisticated techniques requiring that a user will provide the intended configuration by programming it in the proposed configuration language.

The presented paper briefly reported the first attempt to tackle the specified goal. We plan to extend the language with more advanced constructs, which would allow us to define a network specification in the modular manner that is typical for programming languages. This means that a network configuration would be split in modules logically representing network areas. These modules would have defined public interfaces through which interconnections are only possible. These are also points where security enforcement on the highest level is to be implemented. Thus it may be possible to hide internal structures of individual modules to simplify the configuration management.

To evaluate the presented approach we need to i) extend our language beyond the currently supported set of rather basic configuration blocks and ii) support more than a single target platform for which the configuration can be generated. Both are topics for the further work.

ACKNOWLEDGMENT

This work was partially supported by research programs MSM 0021630528 and CZ 1.05/1.1.00/02.0070, and the BUT grant FIT-S-11-1.

REFERENCES

[1] A. Voellmy and P. Hudak, “Nettle: A language for configuring routing networks,” in *Domain-Specific Languages*. Springer, 2009, pp. 211–235.

[2] T. L. Hinrichs, N. S. Gude, M. Casado, J. C. Mitchell, and S. Shenker, “Practical declarative network management,” *Proceedings of the 1st ACM workshop on Research on enterprise networking - WREN ’09*, p. 1, 2009.

[3] J. Guttman, “Filtering postures: Local enforcement for global policies,” in *IEEE Symposium on Security and Privacy*. IEEE Comput. Soc. Press, 1997, pp. 120–129.

[4] G. Stone, B. Lundy, and G. Xie, “Network policy languages: a survey and a new approach,” *IEEE Network*, vol. 15, no. 1, pp. 10–21, 2001.

[5] S. Narain, “Network configuration management via model finding,” in *Proceedings of the 19th conference on Large Installation System Administration Conference-Volume 19*. USENIX Association, 2005, p. 15.

[6] X. Ou, S. Govindavajhala, and A. Appel, “MulVAL: A logic-based network security analyzer,” in *Proceedings of the 14th conference on USENIX Security Symposium-Volume 14*. USENIX Association, 2005, pp. 8–8.

[7] E. Lupu and M. Sloman, “Conflict analysis for management policies,” in *Proceedings of IFIP/IEEE International Symposium on Integrated Network Management (IM1997)*, vol. 97, no. May. Citeseer, 1997, pp. 1–14.

[8] A. Couch and M. Gilfix, “Its elementary, dear Watson: applying logic programming to convergent system management processes,,” in *Proc. Lisa XIII*, 1999.

[9] A. X. Liu and M. G. Gouda, “Firewall Policy Queries,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 20, no. 6, pp. 766–777, Jun. 2009.

[10] F. Baboescu and G. Varghese, “Fast and scalable conflict detection for packet classifiers,” *Computer Networks*, vol. 42, no. 6, pp. 717–735, Aug. 2003.

[11] J. Hong, “XML-based configuration management for IP network devices,” *IEEE Communications Magazine*, vol. 42, no. 7, pp. 84–91, Jul. 2004.

[12] J. van der Ham, P. Grosso, R. van der Pol, A. Toonk, and C. de Laat, “Using the network description language in optical networks,” in *Integrated Network Management, 2007. IM’07. 10th IFIP/IEEE International Symposium on*. IEEE, 2007, pp. 199–205.

[13] A. Feldmann and A. Greenbreg, “NetScope: traffic engineering for IP networks,” *IEEE Network,March/April, 200o. 1l* , 2000.

[14] J. D. Guttman and A. L. Herzog, “Rigorous automated network security management,” *International Journal of Information Security*, vol. 4, no. 1-2, pp. 29–48, Dec. 2004.

[15] S. Narain, G. Levin, and S. Malik, “Declarative Infrastructure Configuration Synthesis and Debugging,” *Journal of Network and Systems*, pp. 1–26, 2008.

 ISBN: 978-1-61208-183-0