Towards validated network configurations with NCGuard

Laurent Vanbever, Grégory Pardoen, Olivier Bonaventure
Université catholique de Louvain, Belgium

***Abstract*— Today, most IP networks are still configured manually on a router-by-router basis. This is error-prone and often leads to misconfiguration. In this paper, we describe the Network Configuration Safeguard (`NCGuard`), a tool that allows the network architect to apply a safer methodology. The first step is to define his design rules. Based on a survey of the networking literature, we classify the most common types of rules in three main patterns: presence, uniqueness and symmetry and provide several examples. The second step is to write a high-level representation of his network. The third step is to validate the network representation and generate the configuration of each router. This last step is performed automatically by our prototype. Finally, we describe our prototype and apply it to the Abilene network.**

I. Introduction

Managing and configuring the devices that compose an IP network is a complex, costly and error-prone task [1]. Network operators must ensure consistency among neighboring routers while facing complex configuration languages and frequent changes in the network [2], [3]. Furthermore, a typical network may contain hundreds of devices from different vendors running different operating systems with their own configuration language.

In many, if not most, networks, the state of the art methodology used by network engineers to configure their routers and switches is very simple [4]. A wiki or a set of text files contain the network documentation, policies and small configuration templates for the most common tasks [2]. When a router's configuration must be changed, the network engineers often update the configuration by using cut-and-paste from the wiki template, after having changed parameters such as IP addresses. If the same operation must be performed frequently, a small script is written and stored on the wiki. Several studies have shown that routers' configuration often contain faults [5], [6], sometimes with consequences on end-to-end Internet connectivity [7].

During the last fifty years, several methods have been proposed to improve software quality. Most of these methods share several common steps. First, the requirements of each application must be analyzed and documented in details. Second, the software will be divided in modules and written in a high-level language. Third, each module can be thoroughly tested. Fourth, formal methods can be used to validate key modules. Compared to software engineering practices, and although more and more IP networks support mission critical services, network configuration remains too often an art. Some authors have compared the current way of configuring networks to *writing a distributed program in assembly language* [8].

In this paper, we describe the Network Configuration Safeguard (`NCGuard`). `NCGuard` is a first, but important, step towards the utilization of software engineering techniques to produce network configurations that can be validated. `NCGuard` encourages the network architect to first specify formally the objectives of his network. Such a formalization is used by several software engineering techniques such as *design by contract* [9] for example. These objectives are defined as a set of rules that must be met by the configuration. Then, the network architect writes a high-level representation of his network. `NCGuard` then validates the configuration automatically based on the rules defined by the architect and generates the routers' configurations in their respective configuration languages.

This paper is organized as follows. We first describe in Section II how a network architect can specify his objectives. Section III details the design of `NCGuard`. Section IV uses `NCGuard` to produce the configuration of the Abilene network and Section V discusses related work.

II. Configuration objectives

Network architects usually have two different types of configuration objectives. High-level objectives are design decisions made by the architect about the organization of his network. Some examples of high-level objectives are: ensure that the iBGP sessions distribute interdomain routes to all BGP routers, ensure that all intradomain routes are distributed by the IGP protocol, enable MPLS, use MPLS fast-reroute to protect links, ... High-level objectives can depend on and be realized through other lower-level objectives. For example, the objective of ensuring that interdomain routes are distributed to all BGP routers can be realized by using a full-mesh of iBGP sessions, redundant Route Reflectors or confederations. Each of these objectives can also depend on lower level objectives. For example, the full-mesh of iBGP sessions requires that the IGP protocol advertises to all routers the reachability of the endpoints of all BGP sessions. This IGP objective can be realized by using OSPF or ISIS. If OSPF is used its correct configuration depends on other lower-level rules such as: the areas must be connected to the backbone area, the same MTU must be configured on both ends of each link, OSPF should run in passive mode on peering links ... `NCGuard` allows the network architect to define a hierarchy of configuration objectives where a high-level objective is composed of several

low-level objectives and allows the network architect to define them as configuration rules.

In this section, we first survey the networking literature to describe the most common configuration objectives and then we show how to formalize them.

A. Patterns of configuration objectives

To build an efficient representation of configuration objectives, we studied the entire configuration of the Belgian Research Network and of the Abilene network. We also analyzed the recommendations from routers vendors such as [10], [11], [12], [13] as well as configuration problems found by tools such as rcc [7], minerals [6] and others [5], [14], [15], [16]. Based on this survey, we found that most of the objectives, and in particular the low-level ones, could be expressed by using three main different patterns : **presence** (and **non-presence**), **uniqueness** and **symmetry**. The last two were already mentioned in [15]. Each pattern is applied on sets of configuration elements that we will call SCOPE. Commonly used scopes include the set of all routers, the set of all border routers, the set of all loopback interfaces ...

The **presence** pattern is used when a given configuration rule must be defined on a set of devices. For instance, [10] recommends to define a `router id` on each router to avoid letting the router select a different one after a reboot. Another example is that the `passive` keyword should appear on the definition of the peering links that connect a router to a different network to avoid creating OSPF adjacencies with a peer [10]. The **non-presence** pattern is the dual of the presence pattern. For instance, it can check that stub areas and not so stubby areas in OSPF do not contain virtual links [11].

The **uniqueness** pattern is used to represent objectives where several network elements must use unique values for a given parameter. The most common uniqueness pattern is to check that the IP addresses assigned to physical interfaces are different [10]. Another example is that all routers should have a different name. A third example could be that for security reasons, the network architect forces all eBGP sessions to use a different MD5 password.

Two **symmetry** patterns that cover most cases have been identified. They are used when two configurations contain related parameters. The *simple equality* pattern is used to express that some parameters in two different routers must be equal. For example, the two routers attached to a given physical link must use the same layer-2 encapsulation. Another example is that the same OSPF timers must be configured on the two routers attached to one link. A third example is that that same OSPF weight is used on both directions of each link. The second kind of symmetry pattern is the *cross equality* **symmetry** pattern. For instance, in order to establish an iBGP session between routers A and B, router A has to configure B as its neighbor, and B has to configure A as its neighbor [11], [10].

Finally, there are some more complex objectives that do not fit in those patterns. These will be expressed as custom rules that are written in a programming or query language. For instance, a network designer could want his network to remain connected even after single link failures while another designer could want his network to remain connected even after the failure of any pair of routers. A second example is a network designer who needs to ensure that at least two disjoint equal cost paths exist between his data centers.

B. Formalization of the configuration objectives

The configuration objectives can be expressed as rules that will be verified by `NCGuard`. A **rule** can represent a design choice or any requirement of an operator. Rules are applied to configuration elements of the network. A configuration element represents any relevant network information such as a router, a router's interface or a routing protocol... All the configuration elements together describe the entire network. They can be represented as a tree noted $T = (V, E)$ with V being the set of vertexes and E the set of edges. A vertex p of the tree represents a configuration element and it is referenced as a Configuration Node (*CNode*). *CNode* p is a child of *CNode* q (i.e., $(p, q) \in E$), if the configuration element represented by p is a part of the configuration element represented by q. Figure 1 depicts an excerpt of the tree T representing a simple network composed of two routers shown on the upper part of the figure.

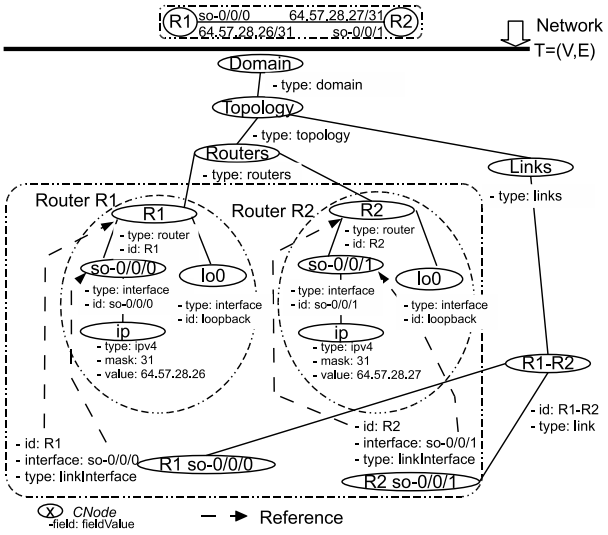

Fig. 1. A simple network containing two routers

Our survey of high and low-level objectives showed us that many rules need to check features on sets of configuration elements. For instance, checking the presence of a `router id` on each router can be done by examining the set of *CNodes* containing all routers, i.e., the children of the `Routers` *CNode* in Figure 1. We call this set the SCOPE of the rule. The SCOPE of a rule is thus the set of *CNodes* on which the rule will be applied.

To formalize the scope, let $C(T, p)$ be a function returning a boolean value indicating whether *CNode* p satisfies a particular condition in tree T. More precisely, a SCOPE is the subset of

CNodes of T respecting condition C_{scope}, thus

$$\text{SCOPE} = \{p | p \in V \ : \ C_{\text{scope}}(T, p)\} \quad (1)$$

For example, to select all the routers shown on Figure 1, the function C_{scope} returns true for every *CNode* p having $p.type = router$ (i.e., R1 and R2) and returns false otherwise. This scope can be written as follows:

$$\text{ROUTERS} = \{p | p \in V \ : \ p.type = router\} \quad (2)$$

In practice, several SCOPES are pre-defined and the network architect can easily add new ones. In addition to ROUTERS, we have defined several scopes [17] corresponding to all BGP routers, all loopback interfaces, ... However, some rules cannot be validated by using only a SCOPE. For instance, a rule that checks that all routers have at least one loopback interface [11], [10] must extract from the network representation a set containing all routers. Then, for each router, the rule must extract its set of interfaces and verify that it contains at least one loopback interface. A router represented by *CNode* p leads to the set of its interfaces noted $\texttt{interfaces}(p)$. In general terms, a set computed from *CNode* p is noted $\texttt{descendants}(p)$ or in short $\texttt{d}(p)$.

For instance, obtaining the *CNodes* representing the interfaces of router $R1$ in Figure 1 requires to select the descendants of $R1$ whose field *type* is equal to `interface`. Let p be the *CNode* that represents $R1$.

$$\texttt{interfaces}(p) = \{q | q \in descendants(T, p) \ : \ q.type = interface\} \quad (3)$$

Now, we describe how the three main patterns of rules can be expressed. A **presence** rule checks whether *CNodes* in T meet a particular condition. More precisely, a presence rule checks for each *CNode* x of the SCOPE, whether there is at least one *CNode* respecting condition C_{presence} in $\texttt{d}(x)$. This rule is expressed formally by:

$$\forall x \in \text{SCOPE} \, \exists y \in \texttt{d}(x) \ : \ C_{\text{presence}}(T, y) \quad (4)$$

For example, Equation 5 shows the presence rule particularized to check that all routers have at least one loopback interface.

$$\forall x \in \text{ROUTERS} \, \exists y \in \texttt{interfaces}(x) \ : y.id = loopback \quad (5)$$

The **non-presence** rule is the dual of the presence rule.

A **uniqueness** rule checks the uniqueness of the value of a field among a set of *CNodes*. More precisely, it checks for each *CNode* x of the SCOPE, whether all *CNodes* in $\texttt{d}(x)$ have different values of *field*. This can be expressed by Equation 6.

$$\forall x \in \text{SCOPE} \, \forall y \in \texttt{d}(x) \ : \ \neg(\exists z_{\neq y} \in \texttt{d}(x) : y.field = z.field) \quad (6)$$

Symmetry rules cover two kinds of symmetries. The first type checks the equality of a value for all members of a set. It is called a symmetry rule with *simple equality*. A symmetry rule can be expressed by Equation 7.

$$\forall x \in \text{SCOPE} \, \forall y, z \in \texttt{d}(x) : y.field = z.field \quad (7)$$

For instance, the MTU must be equal on all connected interfaces [10], otherwise OSPF will not establish an adjacency. This is expressed by Equation 8.

$$\forall x \in \text{LINKS} \, \forall y, z \in \texttt{linkInterfaces}(x) : \ y.MTU = z.MTU \quad (8)$$

The second kind of symmetry rule checks equality between different parameters. We call it a symmetry rule with *cross equality*. The general symmetry rule with *cross equality* is expressed by Equation 9.

$$\forall x \in \text{SCOPE} \ \forall y \in \texttt{d}(x) \ : \ \exists z \in \text{SCOPE} \ \exists w \in \texttt{d}(z) \ : \ y.field1 = z.field2 \text{ and } w.field1 = x.field2 \quad (9)$$

In this section, we have discussed the three main patterns of low-level rules. We explain how `NCGuard` supports them in the next section.

III. THE DESIGN OF NCGUARD

`NCGuard`'s design relies on two main concepts. First, the network configuration is represented at a high-level. Second, the design objectives are expressed as rules that can be automatically tested. Figure 2 illustrates `NCGuard`'s architecture. `NCGuard` is divided in two processes, a *validation engine* and a *generation engine*. The first validates the network representation against the given design rules. The latter generates concrete devices' configurations based on the high-level representation and vendor templates. A vendor template describes the way the high-level representation must be transformed in order to obtain devices' configuration expressed in the vendor configuration language (e.g. Juniper JunOS or Cisco IOS).

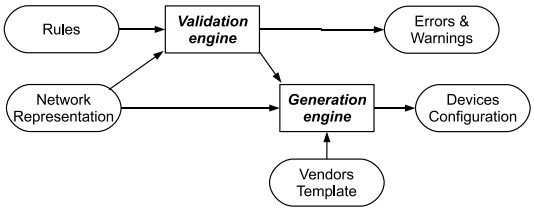

Fig. 2. Overview of `NCGuard`

A. Network representation

A network configuration can be seen as the sum of its component's configuration. Instead of dealing with a set of configurations, `NCGuard` uses one high-level representation encoded as a XML document. We have chosen XML because it is a flexible and self-describing metalanguage that can easily represent hierarchical structures like network configurations. This approach brings several benefits. First, our representation avoids redundancy. When configuring a large network, the

same value for a given parameter must be encoded multiple times. By using a high-level network representation, those parameters are encoded only once. For instance, instead of configuring each router with a logging server, that value can be represented once in the high-level representation and then replicated on each device in an automated way. Thus, the probability of making replication errors vanishes. Second, we adopt a platform-independent representation that can easily represent multi-vendors networks. This is key since each vendor has its own specific configuration language (e.g., Cisco IOS, Juniper JunOS, etc.). Third, we can leverage the existing XML libraries to build the validation and the generation engines.

Due to space limitations, we cannot describe in details how the XML representation is written. A detailed example is available in [17]. In brief, a router (<**node**> element) contains some characteristics such as vendor, a model, and a specific OS version. It contains several elements. The <**rid**> element represents its *router-id*. The <**interfaces**> element contains the description of each interface. A link in our representation connects either two <**node**> elements together, or a <**node**> and a <**ext-node**> element (i.e., a node that doesn't belong to the AS). Parameters that must have the same value on both sides of the link, such as the MTU or the encapsulation are described under the <**attributes**> element. During the generation phase (see Section III-C), these values will be duplicated on each node referenced in the link. We have found that almost all errors due to symmetry incoherence can be avoided by following this simple principle. This abstraction also offers a nice way of dealing with protocol configurations. For example, instead of configuring OSPF on each internal link belonging to the domain, an operator configures OSPF under a single entity. A snapshot of a simple OSPF configuration may be found in [17].

B. Validating rules

High-level configuration objectives are usually expressed as the conjunction or the disjunction of lower-level rules. For example, a top-level configuration objective for OSPF would be encoded as follows :

```
<rule id="CORRECT_OSPF">
        <dependencies type="and">
                <depends>CORRECT_OSPF_INTERFACES</depends>
                <depends>CORRECT_OSPF_AREAS</depends>
        </dependencies>
</rule>
```

NCGuard uses several techniques to check rules. Each technique has its pro's and con's. As a result, a particular technique can be more appropriate than another to check a particular type of rules. Firstly, rules are checked by the **XML Schema** associated to the network representation [18]. We call these rules the *Structural rules*. Secondly, rules can be checked by a **query** on the network representation. We call these rules the *Query rules*. The queries on the XML file representing the network are realized by using XQuery [19], [20]. Finally, the more complex rules are checked by using a **programming language** (Java in our prototype). These rules are called *Language rules*.

A presence, non-presence, uniqueness or symmetry rule fits well with the two first techniques (*Structural rule* and *Query rule*), while a custom rule fits well with the last two techniques (*Query rule* or *Language rule*). A non-custom rule can often be verified by using a query if a SCOPE, or `descendants` obtained from the SCOPE or evaluating a *condition* is required, otherwise a *Structural rule* can be used.

Symmetry rules are often checked directly by the XML Schema of the network representation because redundant informations are stored at a high-level (see III-A) and this implicitly checks the rule. For example, directly connected interfaces must have the same MTU in order to work properly. In NCGuard's network representation, the MTU is defined as a link parameter. As a result, the corresponding rule is already checked by writing the network representation and by ensuring that the correct value will be generated in the final configuration files.

The *Query rules* and their different parameters are written as a XML file. The following examples will illustrate the easiness of writing rules.

*a) **All the routers have a unique router id**:* This presence rule is part of the XML schema.

*b) **A loopback interface is present on all nodes**:* This presence rule requires a *condition* to identify a loopback interface and the notion of `descendants` in order to check the presence of a loopback interface among all interfaces of a node.

```
<rule id="LOOPBACK_INTERFACE_ON_EACH_NODE"
type="presence">
    <presence>
        <scope>ALL_NODES</scope>
        <descendants>interfaces/interface</descendants>
        <condition>substring(@id,1,2)='lo'</condition>
    </presence>
</rule>
```

*c) **Loopback addresses are advertised in OSPF**:* This presence rule needs a SCOPE to check the condition only on OSPF nodes. It can be written by replacing the ALL_NODES SCOPE in the previous rule by a SCOPE that selects all OSPF routers.

*d) **OSPF areas must be directly connected to the backbone area**:* This custom rule checks the OSPF topology. As it does not require running an algorithm, it can be expressed as a XQuery [20] query on the XML file representing the network. To report useful errors and warnings, our queries on the XML file always search for the elements that do not respect the rule. In this example, the query searches for areas without an area border router connected to the backbone area:

```
<rule id="AREAS_CONNECTED_TO_BACKBONE_AREA" type="custom">
    <custom>
        <xquery><![CDATA[
for $area in /domain/ospf/areas/area[@id!="0.0.0.0"]
let $backbone_nodes :=
/domain/ospf/areas/area[@id="0.0.0.0"]/nodes/node
where not(exists($backbone_nodes[@id=$area/nodes/node/@id]))
return <result><area id="{$area/@id}"/></result>]]>
        </xquery>...
```

NCGuard's validation engine is composed of two parts: the engine and the rules. The validation engine was written in about 1500 lines of Java by leveraging the saxon XML libraries. The rules are either part of the schemas of expressed as XQuery rules or by using Java classes. The XML schemas used by NCGuard's current prototype contain about 1000 lines. Additionally, the *Language rules* that check more advanced OSPF or BGP features contain about 950 lines.

C. Generation of the router's configurations

The last step of NCGuard is the generation of network devices' configurations. This is performed in two steps. Firstly, the high-level network representation is transformed into a lower level XML-based representation to facilitate the generation. This representation contains redundancies and is closer to the configurations of real routers. For example, while in the high-level representation the MTU was specified on each link, in the lower-level representation this parameter is specified on each interface. Furthermore, this representation contains the characteristics of each router (vendor, operating system release, memory, types of interfaces, ...). This is very important as some configuration parameters are only valid on some specific platforms.

Secondly, this low-level representation is transformed to concrete vendor configurations (IOS, JunOS, etc.) by using XSLT templates [21]. This allows our validation engine to remain flexible. We can easily add support for new operating systems or generate configurations that depend on the characteristics of each router. Furthermore, it is also possible generate models of the network for simulators such as C-BGP [22] or traffic engineering tools [16], [15]. This can be used to write more complex validation rules. Below you can find a snapshot of an interface's representation and the corresponding generated Juniper configuration.

```
<interface id="so-0/0/0">
    <unit number="0">
        <ip mask="31" type="ipv4">64.57.28.11</ip>
    </unit>...
```

```
interfaces {
    so-0/0/0 {
        unit 0 {
            family inet {
                address 64.57.28.11/31;
...
```

NCGuard's generation engine was written in about 800 lines of Java. The XSLT style sheets used to generate the BGP and OSPF JunOS configurations contain about 1200 lines.

IV. Case study

To illustrate the operation of NCGuard with publicly available configurations, we used it to develop a high-level representation of the Abilene network. We chose Abilene because this is the only network whose configuration is publicly available. This representation was written by reverse-engineering the Abilene router configurations written in JUNOS. The XML representation of the IGP and BGP configurations of Abilene is composed of 1170 lines. We focused on IGP and BGP and inferred their configuration objectives based on the actual configurations. For example, the top-level OSPF design objective is that the OSPF interfaces are correctly configured and that the OSPF areas are correctly used. Since Abilene only uses a single area this objective is easily verified. However, the validation of the OSPF interfaces depends on several lower-level objectives. For example, loopback addresses and addresses of backbone interfaces must be advertised by OSPF. Another example is that OSPF should only be run in passive mode (i.e., without creating adjacencies) on peering links ...

Table I summarizes the rules that are checked against the corresponding high-level representation in function of their type and the technique used to check them. Most of them (78%) are expressed by using *Structural rules*. Those rules are mainly used to perform low-level checks such as verifying the presence or the type of an element/attribute. Most of these rules could be provided directly by the router vendors as input to NCGuard. *Query rules* are typically rules that an operator would write himself to ensure that his design decisions are followed everywhere in the network. For Abilene, we wrote such rules to verify that the IGP is enabled on all backbone interfaces or that the BGP nexthops are reachable. Finally, *Language rules* are used to express more complex rules, generally the ones that require the utilization of an algorithm. For example, in Abilene, one of our custom rules checks that a single link failure does not partition the network.

	Structural	*Query*	*Language*	**Total**
Uniqueness	14	6	-	20
Symmetry	10	-	-	10
Presence	82	15	-	97
Custom	-	6	3	9
Total	106	27	3	**136**

TABLE I
NUMBER AND TYPES OF RULES FOR THE ABILENE NETWORK

To illustrate the usefulness of NCGuard's validation engine, we introduced deliberate errors in the network representation. For example, we deactivated OSPF on one side of an internal link. NCGuard immediately detected it and warned the user with an error message. As noted by [3], comprehensive error messages are important for the network operators. Other examples may be found in [17]. Finally, once the Abilene representation has been validated, NCGuard generate the configuration of each router. The original and generated configuration files are compared in [17]. Globally, the corresponding parts of those files are almost identical. The main differences come from the fact that we don't use exactly the same configuration schemes as the Abilene operators. For example, our configurations follow the Team Cymru's recommendations [23].

V. Related Work

Several researchers have proposed solutions to allow network operators to better configure their routers. Metaconfiguration, proposed by Matuska in [24] also uses a high-level

XML-based representation of the network. Enck et. al describe in [3] the PRESTO software that they use to generate parts of the configurations of routers. Metaconfiguration and PRESTO perform some checks on the generated configuration. However, their validation is not as detailed and as extensible as with `NCGuard`.

Several authors have proposed tools to improve the configuration of the routing policies and the BGP filters. The Routing Policy Specification Language [25] was designed to allow network operators to document their routing policies in the `whois` databases. Some network operators, notably in Europe, rely on these RPSL databases to automatically configure their route filters by using the IRR toolset. Gottlieb et al. in [1] and Maennel et al. in [26] also propose tools to automate the configuration of BGP sessions on routers. Feamster et al. [7] use static analysis to detect BGP configuration errors. Other authors have analyzed router configurations either based on data mining techniques or by using special purpose tools. Caldwell et al. [2], El-Arini et al. [27] and Le et al. [6] use different data mining techniques to detect configuration errors. Some commercial tools from vendors such as WANDL or OPNET can perform various checks on router configurations, but operators are often reluctant to correct all the warnings reported by these tools. `NCGuard` goes one step further by generating the entire configuration and by ensuring that the generated configuration meets the design objectives of the network architect. While finalizing this work, we learned about a similar approach to use programs to generate router's configurations [28].

VI. Conclusion

Most IP networks are still configured manually, which is both error-prone and costly. In this paper, we have proposed the Network Configuration Safeguard (`NCGuard`) that supports a different approach. Instead of manually configuring each router, the network architect first formally specifies the objectives of his network. These objectives are defined as a set of rules that must be met by the configuration. `NCGuard` allows the network architect to easily define his own rules. Then, he writes the configuration of his network as a high-level XML-based representation. Finally, `NCGuard` validates the configuration and generates the router configurations in their respective configuration languages.

Our further work is to improve `NCGuard` to support more protocols and other router vendors. We also intend to enhance `NCGuard` to allow it to interact directly with the routers.

Acknowledgments

We would like to thank Bruno Quoitin, Pierre Francois, Cristel Pelsser and Olaf Maennel for their suggestions and comments. We would also like to thank Alain Fontaine, Jan Torrele and Abilene for having allowed us to study their configuration files.

References

[1] J. Gottlieb, A. Greenberg, J. Rexford, and J. Wang, "Automated provisioning of BGP customers," *IEEE Network*, Nov/Dec 2003.

[2] D. Caldwell, A. Gilbert, J. Gottlieb, A. Greenberg, G. Hjalmtysson, and J. Rexford, "The cutting edge of ip router configuration," *SIGCOMM Comput. Commun. Rev.*, vol. 34, no. 1, pp. 21–26, 2004.

[3] W. Enck, P. D. McDaniel, S. Sen, P. Sebos, S. Spoerel, A. G. Greenberg, S. G. Rao, and W. Aiello, "Configuration management at massive scale: System design and experience." in *USENIX Annual Technical Conference*. USENIX, 2007, pp. 73–86.

[4] R. Bush, "Private communication," 2008.

[5] R. Mahajan, D. Wetherall, and T. Anderson, "Understanding bgp misconfiguration," in *SIGCOMM '02*, 2002, pp. 3–16.

[6] F. Le, S. Lee, T. Wong, H. S. Kim, and D. Newcomb, "Minerals: using data mining to detect router misconfigurations," in *MineNet '06*, 2006, pp. 293–298.

[7] N. Feamster and H. Balakrishnan, "Detecting bgp configuration faults with static analysis," in *NSDI'05*. Berkeley, CA, USA: USENIX Association, 2005, pp. 43–56.

[8] S. Lee, T. Wong, and H. Kim, "To automate or not to automate : On the complexity of network configuration," in *IEEE ICC 2008*, Beijing, China, May 2008.

[9] B. Meyer, "Applying "design by contract"," *Computer*, vol. 25, no. 10, pp. 40–51, 1992.

[10] K. Dooley and I. Brown, *Cisco IOS Cookbook*. O'Reilly Media, Inc., 2006.

[11] J. Doyle, *CCIE Professional Development, Routing TCP/IP, Volume 1*. Cisco Press, 1998.

[12] M. Kolon and J. Doyle, Eds., *Juniper Networks(r) Routers: The Complete Reference*. Osborne/McGraw-Hill, Feb. 2002.

[13] A. Garrett, *JUNOS Cookbook*, 1st ed. O'Reilly Media, Inc., 4 2006.

[14] H. Peine and R. Schwarz, "A multi-view tool for checking the security semantics of router configurations," in *ACSAC '03: Proceedings of the 19th Annual Computer Security Applications Conference*. Washington, DC, USA: IEEE Computer Society, 2003, p. 56.

[15] A. Feldmann and J. Rexford, "IP Network Configuration for Intradomain Traffic Engineering," pp. 46–57, Sept. 2001.

[16] S. Balon, J. Lepropre, O. Delcourt, F. Skivée, and G. Leduc, "Traffic Engineering an Operational Network with the TOTEM Toolbox," *IEEE Transactions on Network and Service Management*, vol. 4, no. 1, pp. 51–61, June 2007.

[17] L. Vanbever and G. Pardoen, "NCGuard," 2008, http://inl.info.ucl.ac.be/softwares/NCGuard.

[18] D. C. Fallside and P. Walmsley, "XML Schema Part 0: Primer Second Edition," W3C," W3C Recommendation, Oct. 2004.

[19] P. Walmsley, *XQuery*. O'Reilly Media, Inc., 2007.

[20] J. Siméon, J. Robie, D. Florescu, D. Chamberlin, M. F. Fernández, and S. Boag, "XQuery 1.0: An XML query language," W3C," W3C Recommendation, Jan. 2007.

[21] J. Clark, "XSL transformations (XSLT) version 1.0," W3C," W3C Recommendation, Nov. 1999, http://www.w3.org/TR/1999/REC-xslt-19991116.

[22] B. Quoitin, "BGP-based interdomain traffic engineering," Ph.D. dissertation, Université catholique de Louvain, August 2006.

[23] Team Cymru's recommendations, `http://www.team-cymru.org`.

[24] M. Matuska, "Metaconfiguration of the computer network," CESNET, Tech. Rep. 27/2004, 2004.

[25] C. Alaettinoglu, T. Bates, E. Gerich, D. Karrenberg, D. Meyer, M. Terpstra, and C. Villamizar, "Routing Policy Specification Language (RPSL)," RFC 2280, Jan. 1998.

[26] O. Maennel, A. Feldmann, C. Reiser, R. Volk, and H. Böhm, "As-wide inter-domain routing policies: Design and realization," 2005, NANOG 34.

[27] K. El-Arini and K. Killourhy, "Bayesian detection of router configuration anomalies," in *MineNet '05: Proceedings of the 2005 ACM SIGCOMM workshop on Mining network data*. New York, NY, USA: ACM, 2005, pp. 221–222.

[28] V. Gill, "Automatic configuration generation and auditing of network," North American Network Operators' Group (NANOG) presentation, October 2008.