Design et implémentation d’un logiciel de validation et de génération de configurations réseaux

Grégory PARDOEN et Laurent VANBEVER

Le 28 août 2008

Promoteur : Professeur Olivier BONAVENTURE

INL: IP Networking Lab
UCL, Louvain-la-Neuve, Belgium
Table of contents

• Introduction
• Some facts
• Main objectives
• Principles
• VNG architecture
• High-level view
• Checking correctness
• Generation
• Demonstration
• Conclusion
Introduction
Some facts

• Today, most networks are still configured on a “router-by-router” basis (telnet approach)

• This is error-prone and leads to misconfigurations (e.g., AS7007 incident in ‘97, AS3561 in ‘01, YouTube in ’08...)

• Network manufacturers encourage engineers to manually apply configurations changes

• Management costs keeps growing due to the increasing complexity of network architectures

A new vision of network configuration is needed!
Main objectives

• A **extensible** and **easy** way of designing and configuring **correct** networks

• Extensibility to be able to add and check new network features

• Easy because network configuration tends to be harder and time consuming

• Correct according to given specifications

• Similar to **software engineering** techniques applied to network configurations
Principles

- **Rules** allow a network architect to specify formally his objectives
  
  - High-level objectives are design decisions (e.g., enforce route reflectors redundancy)
  
  - Low-level objectives are related to routers configurations details (e.g., same MTU on both ends of a link)

- **High-level language** allows the writing of an entire network configuration in a single entity

- Implemented in a software: Validated Network Generator or **VNG**
VNG architecture
Design

Legend:

- INPUT
- PROCESS
- OUTPUT
Checking correctness
Rule based approach

• A rule represents a condition that must be met by the network (like in software engineering)

• Rules check the correctness of the high level representation

• Rules are applied on configuration nodes

• Rules are defined in a XML document
High-level representation

- A single entity represents the whole network
- **Avoid** as much as possible redundancy (e.g. link parameters, protocols configurations)
  - Eliminate duplication errors and reduce typing faults
- Represented using a flexible and hierarchical language: XML
- Structural constraints are defined in a XML Schema
Checking correctness
Rule based approach

• Four **types** of rules were identified:

1. *Presence*
2. *Non-presence*
3. *Uniqueness*
4. *Symmetry*

• If a rule cannot be expressed as one of them:
  • *Custom*
Rules can be checked by using three techniques:

2. Queries on the representation (XQuery): Query rules
3. A programming language (Java): Language rules

<table>
<thead>
<tr>
<th></th>
<th>PRESENCE</th>
<th>NON-PRESENCE</th>
<th>UNIQUENESS</th>
<th>SYMMETRY</th>
<th>CUSTOM</th>
</tr>
</thead>
<tbody>
<tr>
<td>STRUCTURAL RULES</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>QUERY RULES</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>LANGUAGE RULES</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 1. Type of rule in function of the advised technique
The rules are expressed formally by using the notions of **scope** and its **descendants**

- **A scope** is a set of configuration nodes
- **descendants(x)** is a set of selected descendants of the scope’s element *x*
Checking correctness

presence rules

Check if certain configuration nodes are in the representation

Example: each router must have a loopback interface

\( \forall x \in \text{ROUTERS} \ \exists y \in \text{interfaces}(x) : y.id = \text{loopback} \)
Checking correctness

*presence* rules

Check if there is at least one configuration node respecting a given condition in each *descendants* set.

\[ \forall x \in \text{SCOPE}, \ \exists y \in \text{descendants}(x) : C_{\text{presence}}(T, y) \]

Example: each router must have a loopback interface

\[ \forall x \in \text{ROUTERS}, \ \exists y \in \text{interfaces}(x) : y.id = \text{loopback} \]

Query Rules are defined in a XML document

```
<rule id="LOOPBACK_INTERFACE_ON_EACH_NODE" type="presence">
  <presence>
    <scope>ALL_NODES</scope>
    <descendants>interfaces/interface</descendants>
    <condition>@id='loopback'</condition>
  </presence>
</rule>
```
Checking correctness

*uniqueness rules*

Check the uniqueness of a field value in a set of configuration nodes

Example: uniqueness of routers interfaces’ identifiers

- Ids of R1’s interfaces are unique.
- Ids of R2’s interfaces are not unique. The rule will failed.
Checking correctness

*uniqueness* rules

Check if there is no two configuration nodes with an identical value of *field*

\[ \forall x \in \text{SCOPE} \ \forall y \in \text{descendants}(x) : \neg (\exists z \neq y \in \text{descendants}(x) : y.field = z.field) \]

Example: uniqueness of interfaces’ identifiers

\[ \forall x \in \text{ROUTERS}, \forall y \in \text{interfaces}(x) : \neg (\exists z \neq y \in \text{interfaces}(x) : y.id = z.id) \]

```xml
<rule id="UNIQUENESS_INTERFACE_ID" type="uniqueness">
  <uniqueness>
    <scope>ALL_NODE</scope>
    <descendants>interfaces/interface</descendants>
    <field>@id</field>
  </uniqueness>
</rule>
```
Checking correctness

Symmetry rules

- Check the equality of fields of configuration nodes
- Such rules can be checked *implicitly* by the high level representation (i.e., using structural rules)
- Example: MTU must be equal on both ends of a link
  - It can be checked by representing the MTU once on the link level instead of twice at the interfaces level
  - Hypothesis: the duplication phase is correct
Custom rules are needed because some expressions are complicated and cannot be written easily.

Example: All OSPFs areas must be connected to the backbone.

```xml
<rule id="ALL_AREAS_CONNECTED_TO_BACKBONE_AREA" type="custom">
  <custom>
    <xquery>
      for $area in /domain/ospf/areas/area[@id!="0.0.0.0"]
      let $nodes := $area/nodes/node
      where count(/domain/ospf/areas/area) > 1
      and not(some $y in $nodes satisfies /domain/ospf/areas/area[@id="0.0.0.0"]/nodes/node[@id=$y/@id])
      return <result><area id="{$area/@id}"/></result>
    </xquery>
  </custom>
</rule>
```
A rule can be written in few lines

**Simple** XML syntax

Complex rules can also be expressed

An operator can write as many rules as he want
Generation

- High level representation is not intended to be translated easily into configuration files
- **Intermediate** representations are needed
  - It could be seen as the result of a *preprocessing* phase
- **Templates** allow the translation of intermediates representations in configuration files
  - Templates of any configuration or modeling language can be written (e.g., Cisco IOS, Juniper JunOS, etc.)
  - Written in XSLT
Generation

<node id="SALT">
  <interfaces>
    <interface id="lo0">
      <unit number="0">
        <ip type="ipv4" mask="32">198.32.8.200</ip>
        <ip type="ipv6" mask="128">2001:468:16::1</ip>
      </unit>
    </interface>
  </interfaces>
</node>

interfaces {
  lo0 {
    unit 0 {
      family inet {
        address 198.32.8.200/32;
      }
      family inet6 {
        address 2001:468:16::1/128;
      }
    }
  }
}
Demonstration
Conclusion
Conclusion

• Our tool is a **first step** towards a extensible and easy way of designing and configuring **correct** networks

  • **Easy** to:
    • Add new protocols, equipments, parameters...
    • Add rules to check specific needs or new features
    • Add new constructors to generate appropriate configlets

• **Further works**
  • Automatically produce high level representation of a network
  • Extend the prototype to a broader range of cases
  • Allow VNG to interact directly with the routers
Any questions?