A Formal Validation Model for the Netconf Protocol

Sylvain Hallé[1], Rudy Deca[1], Omar Cherkaoui[1], Roger Villemaire[1], and Daniel Puche[2]

[1] Université du Québec à Montréal
{halle,deca,cherkaoui.omar,villemaire.roger}@info.uqam.ca
[2] Cisco Systems, Inc.
dpuche@cisco.com

Abstract. Netconf is a protocol proposed by the IETF that defines a set of operations for network configuration. One of the main issues of Netconf is to define operations such as `validate` and `commit`, which currently lack a clear description and an information model. We propose in this paper a model for validation based on XML schema trees. By using an existing logical formalism called TQL, we express important dependencies between parameters that appear in those information models, and automatically check these dependencies on sample XML trees in reasonable time. We illustrate our claim by showing different rules and an example of validation on a Virtual Private Network.

1 Introduction

The area of network services has significantly developed over the past few years. New and more complex services are deployed into the networks and strain the resources. Network management capabilities have been pushed to their limits and have consequently become more complex and error-prone. The lack of a centralised information base, heterogeneity of all kinds (management tools, configuration modes, services, networks and devices) dependencies among service components, increase of service complexity and undesired services interaction are all possible causes of eventual configuration inconsistency.

Network management must constantly ensure the consistency of the network configuration and of the deployed services. This task is difficult, since there is no formal approach for ensuring the consistency of the network services, and no adequate information model adapted to network configuration. Therefore, adequate formalisms, information models and verification methods are required that must capture the constraints and properties and ensure the integrity of the network services.

The Netconf protocol [6] provides a framework for the network configuration operations. Its `validate` operation checks syntactically and semantically the

We gratefully acknowledge the support of the National Sciences and Engineering Research Council of Canada as well as Cisco Systems for their participation on the Meta-CLI project.

configurations. However, since the work is in progress, this operation is still too generic and not fully defined.

In this paper, we present an implementation of the Netconf `validate` capability that extends beyond simple syntax checking. From an XML Schema representing a given device configuration, we extract a tree structure and express validation rules in terms of these tree elements. By using an existing logical formalism called TQL [3], we express important, semantic dependencies between parameters that appear in those information models, and automatically check these dependencies against sample XML trees within reasonable delays. We illustrate our claim by showing different rules and and validating some of them on a sample Virtual Private Network configuration.

The network management community has proposed other approaches. Some frameworks under development consist in enriching an UML model with a set of constraints that can be resolved using policies. The *Ponder* language [9] is an example of a policy-based system for service management describing OCL constraints on a CIM model. The DMTF community as a whole is working on using OCL in conjunction with CIM. However, object-oriented concepts like class relationships are not sufficient for modelling dependencies between configuration parameters in heterogeneous topologies, technologies and device types.

On a different level, [10] defines a meta-model for management information that takes into account some basic semantic properties. [1] has also developed a formal model for studying the integrity of Virtual Private Networks. However, these approaches can be considered high-level, and ultimately need to be translated into concrete rules using device commands and parameters, in order to be effectively applied on real networks.

In section 2, we give a brief overview of the Netconf protocol and the modelling of XML configuration data in tree structures. Section 3 provides examples of different syntactical and semantic constraints of typical network services, while section 4 introduces the TQL tree logic and shows how these constraints become logical validation rules. Section 5 presents the results of the validation of several configuration rules referring to the *Virtual Private Network* service, and section 6 concludes and indicates further directions of research.

2 The Netconf Protocol

Netconf is a protocol currently under development aimed at defining a simple mechanism through which a network device can be managed [6]. It originates from the need for standardised mechanisms to manipulate the configuration of a network device. In a typical Netconf session, XML-encoded remote procedure calls (RPC) are sent by an application to a device, which in turn sends an RPC-reply giving or acknowledging reception of a full or partial XML configuration data set.

2.1 Netconf Capabilities

In order to achieve such standardised communication, the current Netconf draft defines a set of basic operations that must be supported by devices:

- `get-config`: Retrieves all or part of a specified configuration from a source in a given format
- `edit-config`: Loads all or part of a specified configuration to the specified target configuration
- `copy-config`: Creates or replaces an entire configuration with the contents of another configuration
- `delete-config`: Deletes a configuration datastore
- `lock`: Locks a configuration source
- `unlock`: Unlocks a configuration source
- `get-all`: Retrieves both configuration *and* device state information
- `kill-session`: Forces the termination of a Netconf session

Among other things, these base operations define a generic method enabling an application to retrieve an XML-encoded configuration of a Netconf-enabled device, apply modifications to it, send the updated configuration back to the device and close its session. Alternate configuration data sets can also be copied and protected from modifications. Figure 1 shows a typical RPC, and its reply by the device.

This set of basic operations can be further extended by custom, user-defined capabilities that may or may not be supported by a device. For example, version 2 of the Netconf draft proposes a command called `validate`, which consists in checking a candidate configuration for syntactical and semantic errors before effectively applying the configuration to the device.

The Netconf draft leaves a large margin in the definition of what `validate` must do. A device advertising this capability must be at least able to make simple syntax checking on the candidate configuration to be validated, thus preventing the most trivial errors to pass undetected. However, semantic validation of the configuration is left optional, but is equally important. For example, a simple syntax parser will not complain in the case of a breach of the VPNs isolation caused by address overlapping.

Moreover, although the draft currently defines the behaviour of the validation capability, it leaves open the question of the actual implementation of this capability on a network device. At the moment, there exists no systematic procedure for achieving such validation.

2.2 Modelling Configuration Data

One can remark from the example in figure 1 that the actual XML schema encoding the configuration data might depend on the device. Its format is specified by the XML namespace of the `config` tag in both the RPC and its reply.

```
<rpc message-id="105" xmlns="http://ietf.org/netconf/base/1.0">
  <get-config>
    <source>
      <running/>
    </source>
    <config xmlns="http://info.uqam.ca/schema/node-model"/>
    <format>xml</format>
  </get-config>
</rpc>


<rpc-reply message-id="105" xmlns="http://ietf.org/netconf/base/1.0">
  <config xmlns="http://info.uqam.ca/schema/node-model">
    <node>
      <name>ip_address</name>
      <value>10.0.0.0</value>
      <child></child>
    </node>
     ...
  </config>
</rpc-reply>
```

Fig. 1. Typical Netconf RPC and reply by the device

We briefly describe here the generic XML schema we use in our approach. All properties of a given configuration are described by hierarchically nested attribute-value pairs.

The basic element of our schema is the *configuration node*, which implements the concept of attribute-value pairs. A configuration node is in itself a small tree having a fixed shape. Its main tag is named `node`, and it must contain three children tags:

- `name`, that contains the character string of the name of the attribute
- `value`, that contains is the character string of the value of the attribute
- `child`, inside which can be nested as many other `node` structures as desired

For example, in figure 1, the boldface snippet of XML code inside the `config` tag of the `rpc-reply` shows a sample encoding of an IP address using this schema.

There is a direct correspondence between XML data and labelled trees. By viewing each XML tag as a tree node, and each nested XML tag as a descendent of the current node, we can infer a tree structure from any XML snippet. Figure 2 depicts the tree equivalent of the sample XML configuration code of figure 1.

The tree representation is a natural choice, since it reflects dependencies among components, such as the parameters, statements and features. For more information on the specific schema used in this work, we refer the reader to [8].

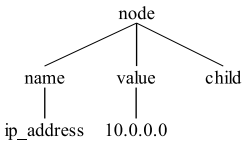

Fig. 2. A simple configuration node

3 Service Configuration Integrity

In this section, we examine the possible configuration inconsistencies that the `validate` capability could encounter and identify when performing verification on a device's candidate configuration. Our study is principally aimed at constraints arising from installation and management of network services.

A network service has a life cycle that starts from a customer's demand, and is followed by negotiation, provisioning and actual utilisation by the customer. Many steps of this life cycle demand that configuration information on one or more devices be manipulated. Configuration parameters can be created or removed, and their values can be changed according to a goal.

However, these manipulations must ensure that the global conditions ruling correct service operation and network integrity are fulfilled. Thus, the parameters and commands of the configuration affected by a service are in specific and precise dependencies. We present here two examples of dependencies, and deduce from each a configuration rule that formalises them.

3.1 Acces List Example

The existence or the possible state of a parameter may depend on another such parameter somewhere else in the configuration. As a simple example of this situation, consider extended IP access lists. Some network devices use these lists to match the packets that pass through an interface and decide whether to block or let them pass, according to packet information. The configuration of these extended IP access lists is variable. If the protocol used for packet matching is TCP or UDP, the port information is mandatory. If the protocol used is different, no port information is required.

Figure 3 shows two examples of access list entries, both of which are valid, although the trees that represent them do not have the same structure.

This example leads us to the formulation of a rule related to the proper use of access list entries:

> **Configuration Rule 1** *If the protocol used in an access list is TCP or UDP, then this access list must provide port information.*

3.2 Virtual Private Network Example

The previous example is nearest to mere syntax checking. On the other end of the scope, there are more complex situations that can be encountered, where the parameters of several devices supporting the same service are interdependent. An example is provided by the configuration of a *Virtual Private Network* (VPN) service [11], [12], [13].

VPNs must ensure the connectivity, reachability, isolation and security of customer sites over some shared public network. A VPN is a complex service that consists of multiple sub-services and its implementation depends on the network technology and topology. For instance, it can be provided at Layer 2 through virtual circuits (Frame Relay or ATM) or at Layer 3 using the Internet (tunnelling, IPsec, VLAN, encryption). The MPLS VPN uses MPLS for tunnelling, an IGP protocol (OSPF, RIP, etc.) for connectivity between the sites and the provider backbone, and BGP for route advertisement within the backbone. The BGP process can be configured using the *direct neighbour* configuration method, which

```
<node>
  <name>protocol</name>
  <value>tcp</value>
  <child>
    <node>
      <name>source</name>
      <value>10.0.0.1</value>
      <child>
        <node>
          <name>wildcard</name>
          <value>0.0.255.255</value>
          <child/>
        </node>
      </child>
    </node>
    <node>
      <name>operator</name>
      <value>eq</value>
      <child>
        <node>
          <name>port</name>
          <value>80</value>
          <child/>
        </node>
      </child>
    </node>
  </child>
</node>
```

```
<node>
  <name>protocol</name>
  <value>icmp</value>
  <child>
    <node>
      <name>source</name>
      <value>10.0.0.1</value>
      <child>
        <node>
          <name>wildcard</name>
          <value>0.0.255.255</value>
          <child/>
        </node>
      </child>
    </node>
  </child>
</node>
```

Fig. 3. Excerpts of XML code for two access list entries

adds routing information necessary for the inter-connection on each provider edge router (PE-router).

Among other requirements of this method, an interface on each PE-router (for example, `Loopback0`), must have its IP address publicised into the BGP processes of all the other PE-routers' configurations using a `neighbor` command [11]. If one of these IP addresses changes the connectivity is lost and the VPN service functioning is jeopardised. Thus,

> **Configuration Rule 2** *In a VPN, the IP address of the* `Loopback0` *interface of every PE-router must be publicised as a neighbour in every other PE-router.*

4 Validating Network Service Integrity

As we have shown in section 2, each XML snippet can be put in correspondence with a an equivalent labelled tree. Thus, configuration rules like those previously described can be translated into constraints on trees.

For example, Configuration Rule 2 becomes the following Tree Rule:

> **Tree Rule 2** *The* `value` *of the IP address of the interface* `Loopback0` *in the PE* `router_i` *is equal to the IP address value of a* `neighbor` *component configured under the BGP process of any other PE* `router_j`*.*

This conversion has the advantage that many formalisms have been developed in recent years [2], [7] that allow such description. Among them, the Tree Query Logic (TQL) [3] is particularly noteworthy, as it supports both property and query descriptions. Hence one can not only check if a property is true or false, but also extract a subtree that makes that property true or false.

In the next section, we demonstrate this claim by showing how TQL can be used to perform validation on tree structures.

4.1 Expressing Configuration Rules

One can loosely define TQL as a description language for trees. Following logical conventions, we say that a tree t matches a given TQL expression e, which we write $t \models e$, when e is true when it refers to t. We also say that e *describes* t.

TQL is an extension of the traditional first-order logic suitable for description of tree-like structures. To allow branching, two operators are added: the edge ([ ]) and the composition (|).

First, the edge construct allows expression of properties in a descendent node of the current node. Thus, any TQL expression enclosed within square brackets is meant to describe the subtree of a given node. For example, the expression `root[child]` indicates that the root of the current tree is labelled "root", and that this root has only one child, labelled "child".

Second, the composition operator juxtaposes two tree roots; hence, the expression node[name | value] describes a tree whose root is "node", and whose

two children are the nodes “name” and “value”. Like other TQL operators, edge and composition are supported at any level of recursion.

The tree depicted in figure 2 is described by the following TQL expression:

```
node[
   name[ip_address] |
   value[10.0.0.0] |
   child    ]
```

Remark the similarity between this TQL description and the XML code that actually encodes this structure in figure 1. This similarity is not fortuitous: it is in fact easy to see that edge and composition alone can describe any single XML tree.

However, interesting properties do not apply on a single tree, but rather to whole classes of trees. It is hence desirable to add the common logical operators to the syntax, whose intuitive meaning is given in table 1.

- T: matches any tree.
- $\neg A$ (negation): if a tree does not match A, then it matches $\neg\ A$
- $A \vee B$ (disjunction): if a tree matches $A \vee B$, then either it matches A or it matches B (or both)
- $A \wedge B$ (conjunction): if a tree matches $A \wedge B$, then it must match both A and B
- % (label wildcard): % matches any label
- . (existence of a child): $.x$ matches any tree whose root has a child labelled x
- !: (all children) : $!x[P]$ matches a tree if and only if all children labelled x verify property P

Table 1. Some of the most common TQL operators

These operators allow us to express, for example, the fact that a given access list entry has a `port` node if its protocol is TCP or UDP:

```
<rule xmlns="http://info.uqam.ca/config-rules/access-list">
   node[
      name[protocol] |
      value[TCP ∨ UDP] |
      child[
          .node.child.node[.name[port]]]]]
   ∨
   node[
      name[protocol] |
      value[¬ (TCP ∨ UDP)] |
      child]
</rule>
```

This rule stipulates that if the `node` defines a `protocol` whose `value` tag contains either TCP or UDP, then there must be a `child` tag containing a `port` node. On the contrary, if `protocol` is different from TCP and UDP, then the node has an empty `child` tag. We can check that both XML trees in table 1 verifiy the property. In the previous and all the following examples, the actual logical connectors (`and`, `or`, and the like) recognised by TQL have been replaced by their common symbols for improved clarity.

Notice that this rule is encapsulated inside an XML tag and is referenced in a global namespace, allowing for a uniform hierarchical classification of possible syntactical and semantic dependencies, and a better integration in Netconf's model. At the moment, we simply ignore this tag and submit the inside query to the standard TQL tool.

It is even possible to extract the protocol name using the query:

node[.value[$P]]

which places into the variable $P the text inside the `value` tag for a given tree.

There are many other operators which further extend TQL's rich semantics [2], [3]. However, all of the interesting constraints we encountered in our work are expressible by means of those mentioned in this section. For more information related to TQL and its syntax, the reader is referred to [2] and [3].

4.2 The `validate` Operation

As there is a correspondence between XML and labelled trees, there is also a correspondence between tree rules and TQL queries. For example, Tree Rule 2 becomes the TQL query shown in figure 4.

The first part of the query retrieves all tuples of values of `device_name` and `ip_address` for the interface called `Loopback0`. These tuples are bound to the variables $N and $A. The second part of the query makes a further selection among these tuples, by keeping only those for which there exists a device whose name is not $N where $A is not listed as a neighbour. If the remaining set is empty, then all addresses are advertised as neighbours in all other devices, and the property is verified.

As one can see from the previous example, TQL queries can quickly become tedious to write and to manage. Fortunately, these queries can be automatically verified on any XML file by a software tool downloadable from TQL's site [15]. The tool loads an XML file and a set of TQL properties to be verified on that structure. It then makes the required validations and outputs the results of each query.

5 Results and Conclusions

As a concrete example of this method, we processed sample RPC-reply tags for multiple devices with constraints taken from the MPLS VPN service. These constraints have been checked in a different context in [8].

```
<rule xmlns="http://info.uqam.ca/config-rules/vpn/neighbours-declaration">
  network[
    .node[
      .name[device_name] |
       .value[$N] |
       .child.node[
          .name[interface_type] |
          .value[loopback] |
          .child.node[
            .name[interface_number] |
            .value[0] |
            .child.node[
              .name[ip_address] |
              .value[$A]]]]]
    ∧
    .node[
      .name[device_name] |
      .value[¬ $N]
      ∧
      ¬ .child.node[
        .name[bgp] |
        .value[%] |
        .child.node[
          .name[neighbor] |
          .child.node[
            .name[ip_address] |
            .value[$A]]]]]
</rule>
```

Fig. 4. TQL query for Tree Rule 2

P1 If two sites belong to the same VPN, they must have similar route distinguisher and their mutually imported and exported route-targets must have corresponding numbers.

P2 The VRF name specified for the PE-CE connectivity and the VRF name configured on the PE interface for the CE link must be consistent.

P3 The VRF name used for the VPN connection to the customer site must be configured on the PE router.

P4 The interface of a PE router that is used by the BGP process for PE connectivity, must be defined as BGP process `neighbor` in all of the other PE routers of the provider.

P5 The address family vpnv4 must activate and configure all of the BGP neighbours for carrying only VPN IPv4 prefixes and advertising the extended community attribute.

All these properties were translated into TQL queries, and then verified against sample XML schema trees of sizes varying from about 400 to 40000

XML nodes. One query verified only P1 and had a size of 10 XML nodes; the second query incorporated all the previous five rules and was 81 XML nodes long. Table 2 shows validation time for these different settings.

Configuration size	Query size	Validation time (s)
413	10	0,04
413	81	0,24
1639	10	0,06
1639	81	0,47
3681	10	0,09
3681	81	0,84
6539	10	0,12
6539	81	1,29
10213	10	0,15
10213	81	1,87
40823	10	0,52
40823	81	7,03

Table 2. Validation time for different configuration and rule sizes

All queries have been validated on an AMD Athlon 1400+ system running on Red Hat Linux 9. Validation time for even the complete set of constraints is quite reasonable and does not exceed 8 seconds for a configuration of more than 40000 nodes. As an indication, a device transmitting a configuration of this size via an SSH connection in a Netconf `rpc-reply` tag would send more than 700 kilobytes of text.

For all these sets, TQL correctly validated the rules that were actually true, and identified the different parts of the configurations that made some rules false, if any.

6 Conclusions

We have shown in this paper a model for the `validate` capability proposed by the current Netconf draft. Based on an existing logical formalism called TQL that closely suits the XML nature of the protocol, this model extends beyond simple syntax checking.

We stress the fact that the validation concept must not be limited simply to mere syntax checking and should encompass semantic dependencies that express network functions and rules. Formalisms such as the Common Information Model (CIM) [5] and Directory Enabled Networking (DEN) [14] could be further exploited to this end.

The VPN case illustrated in the previous sections indicates that using a subset of a query language like TQL is sufficient to handle complex semantic dependencies between parameters on interdependent devices.

The results obtained suggest that this framework could be extended to model most, if not all, such dependencies in device configurations. It is therefore a good candidate as a template for a formal Netconf model of the `validate` capability.

References

1. Bush, R., Griffin, T.: Integrity for Virtual Private Routed Networks. Proc. IEEE INFOCOM (2003)
2. Cardelli, L.: Describing semistructured data. SIGMOD Record, 30(4) (2001) 80–85
3. Cardelli, L., Ghelli, G.: TQL: A query language for semistructured data based on the ambient logic. Mathematical Structures in Computer Science (to appear).
4. Deca, R., Cherkaoui, O., Puche, D.: A Validation Solution for Network Configuration. Communications Networks and Services Research Conference (CNSR 2004), Fredericton, N.B. (2004)
5. DSP111, DMTF white paper, Common Information Model core model, version 2.4, August 30, 2000.
6. Enns, R.: NETCONF Configuration Protocol. Internet draft, Feb. 2004. http://www.ietf.org/internet-drafts/draft-ietf-netconf-prot-02.txt
7. Gottlob G., Koch, C.: Monadic queries over tree-structured data. LICS02 (2002) 189–202
8. Hallé, S., Deca, R., Cherkaoui, O., Villemaire, R.: Automated Validation of Service Configuration on Network Devices. Proc. MMNS 2004 (2004) (to appear).
9. Lymberopoulos, L., Lupu, E., Sloman, M.: Ponder Policy Implementation and Validation in a CIM and Differentiated Services Framework. NOMS 2004 (2004)
10. López de Vergara, J.E., Villagrá, V.A., Berrocal, J.: Semantic Management: advantages of using an ontology-based management information meta-model. HP-OVUA 2002 (2002)
11. Pepelnjak, I., Guichard, J.: MPLS VPN Architectures, Cisco Press (2001)
12. Rosen, E., Rekhter, Y.: BGP/MPLS VPNs. RFC 2547 (1999)
13. Scott, C., Wolfe, P. Erwin, M.: Virtual Private Networks, O'Reilly (1998)
14. Strassner J., Baker F.: Directory Enabled Networks, Macmillan Technical Publishing (1999)
15. TQL web site, Università di Pisa. http://tql.di.unipi.it/tql/