JNSM manuscript No.
(will be inserted by the editor)

A Declarative Approach to Network Device Configuration Correctness

Éric Lunaud Ngoupé · Clément Parisot · Sylvan Stoesel · Petko Valtchev · Roger Villemaire · Omar Cherkaoui · Pierre Boucher · Sylvain Hallé

Received: 24 July 2014 / Accepted: date

Abstract Configuration Logic (CL) is a formal language that allows a network engineer to express constraints in terms of the actual parameters found in the configuration of network devices. We present an efficient algorithm that can automatically check a pool of devices for conformance to a set of CL constraints; moreover, this algorithm can point to the part of the configuration responsible for the error when a constraint is violated. Contrary to other validation approaches that require dumping the configuration of the whole network to a central location in order to be verified, we also present an algorithm that analyzes the correct formulas and greatly helps reduce the amount of data that need to be transferred to that central location, pushing as much of the evaluation of the formula locally on each device. The procedure is also backwards-compatible, in such a way that a device that does not (or only partially) supports a local evaluation may simply return a subset or all of its configuration. These capabilities have been integrated into a network management tool called ValidMaker.

Keywords management · fault detection · logical constraints

E.L. Ngoupé, S. Hallé
Laboratoire d'informatique formelle
Université du Québec à Chicoutimi, Canada
E-mail: shalle@acm.org

C. Parisot, S. Stoesel
Télécom Nancy, France

P. Valtchev, R. Villemaire, O. Cherkaoui
Laboratoire de téléinformatique et réseaux
Université du Québec à Montréal, Canada
E-mail: valtchev.petko@uqam.ca, E-mail: villemaire.roger@uqam.ca, E-mail: cherkaoui.omar@uqam.ca

P. Boucher
Ericsson Canada

1 Introduction

The management of computer networks is an increasingly complex and error-prone task. On the one hand, the devices that form a network must behave as a group; however, on the other hand, each of these devices is managed and configured individually. The fundamental issue has remained mostly unchanged for many years. A network engineer is given the responsibility of a pool of devices whose individual configurations are managed mostly by hand. Every time a new service needs to be added to the network, he must ensure that the configuration parameters of these devices are set to appropriate values. This delicate operation must fulfil two goals: implementing the desired functionality, while preserving proper operation of existing services. This entails, in particular, that the new configuration parameters must not conflict with already configured parameters of these or other devices.

Research in the past has shown that between 40% and 70% of changes made to the configuration of a network fail at their first attempt, and half of these changes are motivated by a problem located elsewhere in the network [1]. It is reasonable to think that these figures have not significantly changed in the past couple of years: Feamster and Balakrishnan revealed more than 1,000 errors in the Border Gateway Protocol (BGP) configuration of 17 networks [2]; Wool studied firewalls from a quantitative aspect and reported that all of them were misconfigured in some way or another [3].

How can one be assured that a service installed on a network works correctly? In a paper about next generation configuration management tools, Burgess and Couch [4] put forward the concept of *aspects*, similar in nature to Aspect-Oriented Programming (AOP). An aspect is a set of configuration parameters $p_1, \ldots, p_n$ with domains $D_1, \ldots, D_n$ and a set $S \subseteq D_1 \times \cdots \times D_n$ of admissible values for these parameters that can be interdependent. Possible values can be restricted for technical reasons, policies, QoS requirements or the semantics of the parameters. Any formal language (e.g., logic, set theory) can be used to compactly represent S.

This task, already non-trivial at the onset, is becoming increasingly hard because of the fast evolution of the number of devices, the complexity of the configuration, the specific needs of each service and the sheer number of services a network must be able to support. When one adds to this portrait the fact that data generally traverses heterogeneous networks owned by multiple operators, one realizes why the advent of novel approaches to the problem of network configuration management is vital. In particular, the issue of the reliance on a central location for managing or examining the network's configuration requires to be addressed, for both simplicity and bandwidth consumption considerations.

In Section 2, we shall see that configuration constraints can arise for various technical reasons, ranging from syntactical restrictions to the higher-level semantics of the network services implemented in a network. Moreover, such constraints not only restrict parameter values on a single device, but they also impose correlations of multiple parameters in multiple devices. This occurs in such a way that whether some parameter has a valid value on one device may depend on the value of one or more parameters located on *other* devices in the network.

For a reasonably small number of devices, configuration management can still be handled by hand; assessing configuration correctness can easily be done by manually

querying relevant parameters on each device (through the use of so-called "show commands") and making sure that their values follow the policies and constraints to enforce. This manual inspection is neither desirable nor possible when managing tens or hundreds of devices, where some form of automation is required. In turn, the high costs of automation might pay off for very large networks, but not turn out to be viable for medium-sized networks, which become too large to be handled by hand, and too small to benefit from full-scale automation.

To this end, various protocols and tools have been developed, which are surveyed in Section 3. Alas, we shall see that, while such tools are efficient at managing modifications to configurations, they are much less powerful in their capabilities to automatically assess their correctness. Simple verification scripts running independently on each device are not sufficient, since correlations exist between parameters across multiple devices.

In Section 4, we present a network configuration validation tool called ValidMaker, using a declarative approach to configuration management. Through a formal language called Configuration Logic (CL), a network engineer can express constraints in terms of the actual parameters in the configuration of the devices; each constraint can be seen as a specific aspect. The integration of a CL validation engine within ValidMaker allows us to to automatically check a pool of devices for conformance to a set of CL formulas. As has been shown in previous works, the use of a logic-based formalism for configuration management provides unique benefits. First, the verification of configurations on multiple devices can be done very efficiently. Second, the approach enforces a clean separation between the specification of constraints and its actual validation. Finally, in the event one of the constraints is violated, we describe an algorithm that returns the parts of the configuration that are incorrect as evidence to the user. The validation can then switch to an interactive mode where the user can explore this evidence, backtrack, and resume validation to locate the exact source of the error. This functionality is missing from the existing tools that provide a mere yes/no answer.

The solution adopted by the few tools that provide configuration checking capabilities amounts to dumping the configuration of each device to a centralized location, and to perform whatever verification is needed by providing local, random access to the complete configuration of the network. One can easily see why this mode of operation is neither efficient nor desirable for a variety of reasons. This is why this paper presents a solution for assessing configuration correctness that attempts to alleviate this problem by borrowing on the concept of *lazy evaluation*.[1] In Section 5, we first show how configuration constraints can be expressed as first-order logical expressions on formal representations of configurations as hierarchical data structures. Assessing configuration correctness then amounts to computing whether a set of such formulas evaluates as true. We then present an algorithm that automatically performs this evaluation; while the algorithm still requires the need for a centralized point of verification, the analysis of a formula greatly helps reduce the amount of data that needs to be transferred to that central location, pushing as much of the evaluation of

[1] In programming, lazy evaluation is the process of fetching only the parts of a data structure or object passed as an argument to a function that are actually accessed by that function.

the formula locally on each device. The procedure is also backwards-compatible, in such a way that a device that does not, or only partially supports local evaluation, may simply return a subset or all of its configuration.

An implementation of this algorithm is presented in Section 6, and its evaluation on synthetically generated configurations is then discussed. Preliminary results indicate that, for some configuration constraints, almost all the verification can be performed locally. Even for constraints that impose correlations on parameters of multiple devices, only the minimal information required for verifying these correlations is transferred to the centralized engine, resulting in considerable savings in terms of consumed bandwidth.

2 Network Device Configuration Correctness

In this section, we first formally define the concept of network device configuration, and then introduce the notion of configuration correctness by providing examples of constraints over configuration parameters taken from a real-world network service.

2.1 Devices and Configurations

What we define as a device can be a hardware component such as a router, a switch, or a wireless access point or gateway. This device has its own internal features and communication interfaces and usually varies depending on its manufacturer (for physical devices) or its version (for virtual devices). The main characteristic of a device is that its default behaviour can be modified dynamically through configuration.

The configuration of a device corresponds to a set of parameters and associated values, recorded by the device and used for its operations. These settings allow one to customize the device, for example by setting its physical interfaces, by setting rules for data redirection, by adapting the generic behaviour of the device to the network, or by modifying its behaviour according to events received in the past.

Editable elements in a configuration are called *parameters*. These parameters are the elements used to customize the behaviour of a device for a given network. Parameter values can be numeric (e.g., an IP address or subnet mask number), strings (e.g., the name of an Ethernet interface) or be complex data structures (e.g., the state of a port).

The precise form of a configuration within a device depends on its manufacturer. For instance, the manufacturer Cisco uses a batch file containing Command Line Interface (CLI) commands and associated values. This file is hierarchized, as some commands make the device enter or exit from a *mode* (a particular context, such as the configuration of a specific interface) or a submode. More generally, we shall consider (without loss of generality) that we can reduce the configuration of each device, regardless of its vendor (Cisco, Juniper, etc.) to a generic configuration in an XML format (or so-called “Meta-CLI”) as is shown in Figure 1. This uniformization of configurations is out of the scope of the present paper and has already studied in previous work [5].

```
!
version 12.0
!
hostname Pomerol
!
interface Loopback0
ip address 10.10.10.3 255.255.255.255
ip router isis
```

Fig. 1: A simple example of a Cisco device configuration

2.2 Configuration Correctness

The parameters of a configuration cannot take any arbitrary value. For a configuration to be considered correct, a number of rules must be applied depending on the *services* that the device is required to support. Constraints usually describe the necessary conditions that must be met for a configuration to be consistent.

We illustrate this using a simple example based on the configuration of a Virtual Local Area Network (VLAN), generally handled by network switches, and which allows a network to be partitioned into logical segments. Each port of a switch can be assigned to a particular VLAN. Ports that are assigned to the same VLAN are able to communicate at OSI Layer 2 while ports not assigned to the same VLAN require Layer 3 communication. All the switches that need to share Layer 2 intra-VLAN communication need to be connected by a link called a *trunk* that joins two interfaces (one on each switch) and these interfaces should be encapsulated in the same *mode*. IEEE 802.1Q [6] and the Virtual Trunking Protocol (VTP) [7] are two popular protocols involved in the management of VLANs.

The correct configuration of a VLAN imposes a number of constraints on the parameter values that can be found on each switch. We give two examples of such constraints:

1. *No two devices on the same VLAN can have the same address for more than one interface.* Since IP addresses can be manually configured for each interface on a device, conflicts between addresses can occur. This can make one of the interfaces, which shares the same address, to appear to be defective and cause interruptions in the communications.
2. *Every device has to be a VTP client or a VTP server, and only one server can exist.* This means that for each switch where VTP is activated, a parameter must define whether the switch acts as a client or a server (the so-called “VTP mode”). Moreover, a conflict can arise if more than one switch is acting as a server, again potentially causing interruptions in the network service.

This is just a small sample of configuration constraints that can be elicited. A key observation is that most constraints impose restrictions on the values of parameters, and that many times, values inside a device are correlated to values occurring for other parameters in other devices. The reader can find a more detailed presentation of VLANs in [5].

3 State of the Art in Configuration Management

The management of data networks is still a tedious and error-prone task, whose complexity is constantly increasing due to the evolution of the technologies and of the equipment used [8, 9]. We now present some protocols and tools for configuration management.

3.1 Management Protocols

Several protocols exist that are related to network management. We briefly present them in increasing order of complexity.

3.1.1 FTP

One of the simplest ways of managing configurations is to transfer configurations as files through a connection using the File Transfer Protocol (FTP). In such a scenario, each device acts as an FTP server to which configurations are pulled or pushed as simple files, and any changes made to the configurations are applied immediately. Such a mode of operation provides almost no error checking; a user can easily overwrite a working configuration with a file containing syntax errors or erroneous parameter values, rendering the device incapable of operating. However, it often provides a fallback method for acting upon a device, in the event higher-level methods fail to work.

3.1.2 Command-oriented Protocols

Network devices rapidly increase in complexity, and most of them are now running small-scale operating systems providing a set of commands for modifying their internal configuration. A popular means of acting upon the configuration of a device emerged in the form of a Command-Line Interface (CLI), in which communication between the user and the device is done in text mode in a terminal. Nowadays, most network devices have an embedded CLI for purposes of troubleshooting and configuration. Cisco is one of the most prominent users of CLI, with its device operating system providing a very rich set of thousands of commands. This has led some to redefine the acronym CLI as “Cisco-Like Interface”.

Although still relatively low-level, the CLI has several advantages over FTP. One notable improvement is that the network manager can perform modifications on a copy of the configuration currently in use by the device, called the *candidate* configuration. Basic verifications on the configuration commands (such as syntax error or invalid parameter ranges) are performed before the configuration can be put into production (i.e., set as the *running* configuration). Moreover, the CLI also provides ways for a user to display various data about the current state of the device, using so-called *show commands*, called this way in Cisco-talk because many write operations on a configuration can be turned into read operations by prefixing them by the keyword `show`.

3.1.3 Variable-oriented Protocols

The previous two protocols were mainly used for remote connections to equipment, but make it difficult to get an overview of the infrastructure as each device is accessed individually. Moreover, browsing or searching through the configuration is a tedious task that can only be done by issuing multiple show commands to various devices and in various modes. To alleviate these issues, the Simple Network Management Protocol (SNMP) was standardized by the Internet Engineering Task Force (IETF) [10]. The initialism SNMP is typically used to refer to a set of specifications including the protocol itself, the definition of an information model, a corresponding database and finally, related concepts. One of the objectives is to enable independence between architecture and mechanisms of particular hosts or particular gateways as much as possible.

The Management Information Base (MIB) is probably the most important element of SNMP that allows one to describe a large number of components (networks, routers, servers, etc.) in a standard way. The structure of the MIB is hierarchical; data elements are grouped into a tree. Each parameter of every device from every vendor is given a unique sequence of dot-separated digits that represents it into a global namespace. Managing parameters using SNMP then amounts to querying or writing to parameters by giving their unique numerical identifier.

As such, SNMP can be dubbed a "variable-oriented" protocol, as each individual element of the MIB can be accessed and modified on an individual basis. This structure can be seen as an improvement over FTP and CLI, in that SNMP somehow standardizes the mechanism for interacting with configurations. SNMP agents, coupled with SNMP client software, makes it possible to browse through the configuration of a single device or even of multiple devices, and perform queries and modifications via a more user-friendly interface.

3.1.4 Document-oriented Protocols

The structure of the MIB, however, still imposes some rigidity to the way configuration information must be structured and can be accessed. The Network Configuration Protocol, called NETCONF, is an IETF network management protocol [11] that is being adopted by major network equipment providers. It is an XML-based protocol used to manage the configuration of networking equipment (typically routers or switches) and is intended to provide extended features over SNMP. NETCONF provides mechanisms to install, manipulate, and delete the configuration of network devices. It specifies a protocol that allows a manager to change the configuration of the equipment (device) by sending and receiving XML data. Because of its XML nature, NETCONF can be called a "document-oriented" management protocol.

To this end, NETCONF provides a small set of low-level operations to manage device configurations and retrieve device state information. The base protocol provides operations to retrieve, edit, copy, and delete configuration datastores. Additional operations are provided, based on the capabilities advertised by the device [11].

3.1.5 YANG

Complementary to NETCONF is the YANG[2] language [12] that allows the flexible description of the data structures used to represent the configuration of an equipment managed through NETCONF. In a way similar to what schemas are to XML documents, YANG declarations can specify the hierarchical structure of configuration parameters (such as cardinality constraints on child elements), their types, and optional arguments.

One notable feature of YANG is its capability to express *constraints* on the possible structure of a configuration. For example, the "when" statement makes its parent data definition statement conditional. The node defined by the parent data definition statement is only valid when the condition specified by the "when" statement is satisfied. This condition itself is an XPath expression that can be used to query other parts of the tree. Similarly, the "must" statement takes as an argument a string that contains an XPath expression, and is used to formally declare a constraint on valid data.

3.2 Management Tools

The aforementioned protocols provide means of accessing configurations, but do not provide any management facilities. These functionalities are assumed by tools operating over one of these protocols. We mention some of the most prominent tools below.

3.2.1 Cfengine

Cfengine [13] is a tool for automatic configuration management. It is usually run periodically on hosts. When started, it retrieves an instance of policy definitions or established constraints from the main server (the one hosting the policy set) and tries to adapt its local system to this policy set. This makes it possible to declare constraints or requirements on an infrastructure level beforehand. During its execution, Cfengine compares the desired configuration and the one on the equipment [14] and applies the required modifications, if any. Cfengine allows the deployment of configurations on a set of computers, the synchronization of files on heterogeneous servers, and the sending of commands to these servers.

3.2.2 Kiwi CatTools

CatTools is an application that allows automated configuration management of devices such as routers, switches and firewalls [15]. CatTools uses Telnet or SSH for connection, and an embedded TFTP server to push its configurations onto equipment. After an interval of time defined by the administrator, CatTools will retrieve the configuration of each device and compare it to the version on the server; if different, CatTools will replace the one on the device with the one on the server.

[2] "Yet Another Network Language"

3.2.3 Puppet

Puppet is a management tool for automatic configuration dedicated to server operating systems [16]. Puppet allows remote deployment of configurations on a set of servers or network devices in a very short time. Like the other tools, Puppet works in a client-server mode. The server part, called "Puppetmaster", is installed on the main server. The server allows one to create and edit configuration files in the form of a manifesto. The server listens for client connections, then indicates to them which manifesto they should apply.

3.2.4 Chef

Chef is another tool for automatic management of configurations [17]. The operation of Chef is based on abstract definitions known as "cookbooks". Cookbooks contain "recipes" that are scripts written in Ruby and managed as source code. Each definition describes how a specific part of the infrastructure should be built and managed. Chef then applies these definitions automatically on the equipment as specified, resulting in a fully automated infrastructure. When a new server is online, the only thing that Chef needs to know is what cookbooks must be applied to the new equipment. The Chef server acts as a data concentrator for configurations; it stores cookbooks and policies and contains an inventory of all available nodes.

3.2.5 SmartFrog

SmartFrog [18] is another framework for creating configuration-driven systems. It has been designed with the express purpose of making the design, deployment and management of distributed component-based systems simpler and more robust. It provides its own declarative data description notation that is defined along with supporting tools and programmatic data structures. Configuration descriptions, especially those that are to be used as templates and so modified through extension, can also be validated against component-specific assumptions that may not be present in the structure alone. To this end, SmartFrog allows one to express simple Boolean constraints on parameter values. However, it lacks advanced features allowing one to, e.g., correlate parameter values in the configuration of multiple devices.

3.3 A Need for Declarative Configuration Management

Very few of the solutions described in the previous section address the issue of configuration correctness. Management protocols allow modifications to configurations of one device at a time, and can (at the most) perform some form of syntactical or structural checking of the modifications to the parameters to be applied. The only exception is YANG that makes it possible to define constraints over a configuration, and in some cases, the possible values it can contain. Unfortunately, YANG only accepts XPath 1.0 expressions as conditions. They lack the variables and first-order quantifiers that are required to model some of the consistency rules that we will describe in Section 4.

Management tools, on their side, allow the automation of various tasks on multiple devices, but offer very few in the way of expressing and verifying correctness of a configuration according to rules. Moreover, it might be desirable, when one of the configuration rules is violated in a given network, to be shown evidence for the failure. For example, if all devices must agree on the value of a parameter, a pointer to the device having a different value could be given. Similarly, if all neighbors of a device must be declared in some section of its configuration, the address of the missing neighbors could be provided as a guide to the administrator. In other words, counter-examples, or "witnesses" to the effect that a constraint is violated, can be powerful tools to help correct an error (instead of a simple true/false verdict). Again, none of the existing management tools provide advanced counter-example generation features besides a few hard-coded capabilities restricted to very simple cases (e.g., when a parameter is outside the set of its valid values).

The network management community has proposed other approaches. Some frameworks that are under development consist in enriching a UML model with a set of constraints that can be resolved using policies. The Policy Description Language (PDL), developed by Lobo et al. [19], is designed for representing event and action oriented generic policy; it has been extended into the CIMSPL policy language for the distributed management of infrastructures [20] that complies with the CIM (Common Information Model) Policy Model and fully incorporates CIM constructs. Similarly, the Ponder language [21] is an example of a policy-based system for service management describing constraints in the Object Constraint Language (OCL) [22] 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. For example, it is far from clear how the verification of an arbitrary OCL constraint could be carried out in a distributed fashion, as will be described in Section 5 for the formal language introduced in this paper.

4 Logic-Based Configuration Management

Following the requirements for future network configuration tools suggested in [4], the network configuration management tool, ValidMaker, has been developed by the team of Lab Téléinfo at Université du Québec à Montréal. The tool serves two main purposes. First, it levels the heterogeneity of devices by providing a common representation of configuration information called Meta-CLI. Second, ValidMaker allows formal constraints to be expressed on Meta-CLI structures. To this end, it provides a language called Configuration Logic (CL) that allows a network engineer to input custom constraints, and a CL validation engine to automatically check a given configuration for conformance. The constraints can impose dependencies between many parameters of the configuration and correspond to the definition of an aspect in [4].

To fulfil these two goals, the ValidMaker tool is composed of two modules, as shown in Figure 2. We briefly describe these two modules below.

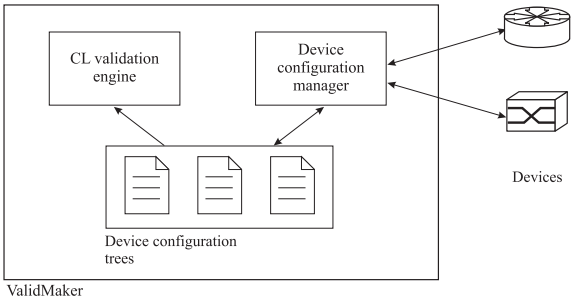

Fig. 2: The architecture of the ValidMaker configuration tool

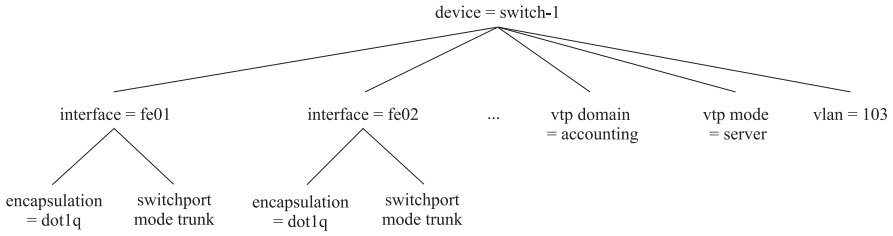

Fig. 3: A portion of a Meta-CLI configuration tree

4.1 Device Configuration Manager

The device configuration manager is the part of the system responsible for communicating with the devices, retrieving their configuration and transforming them into Meta-CLI structures. In reverse, Meta-CLI configurations inside ValidMaker can be translated back into runnable configurations sent to the devices in the proper format, according to their version number.

As explained in [23], the configuration of network devices, such as routers and switches, can be represented as a tree where each node is a pair composed of a name and a value. This tree represents the hierarchy of parameters inherent in the configuration of such devices. The Meta-CLI structures used in the internal representation of configurations in ValidMaker use this tree form. As an example, Figure 3 shows the representation of the configuration of a switch. Configurations are currently retrieved through a shared directory, where device configurations are dumped to text files, imported into ValidMaker and converted (on-the-fly) into Meta-CLI structures. A number of Cisco devices are supported; the actual process by which the configurations are converted is beyond the scope of the present paper.

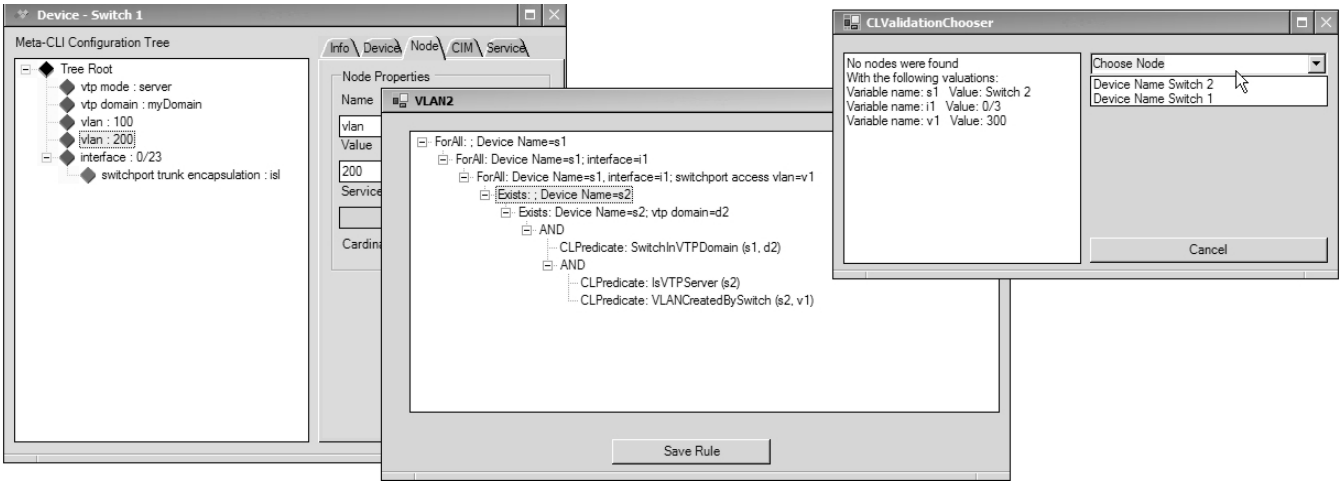

Fig. 4: Illustrates a screenshot from ValidMaker's configuration view. The configuration of a device is abstracted as a tree of name-value pairs (left window). When a CL constraint is violated on a given configuration, ValidMaker highlights the part of the formula that is false (center window). The user is presented a list of tree nodes that violate that part of the formula (right window). The validation can then be resumed on one of these nodes to further explore the cause of the violation.

4.2 Configuration Logic Validation Engine

The configuration logic Validation Engine is the part of ValidMaker responsible for the verification and automatic validation of the configuration file. Once the device configurations are abstracted into Meta-CLI trees of name-value pairs, ValidMaker allows the network engineer to express formal constraints on these trees with the means of Configuration Logic (CL) [8].

4.2.1 Formalism, Syntax and Semantics of CL

CL formulas use the traditional Boolean connectives of classical propositional logic: $\wedge$ ("and"), $\vee$ ("or"), $\neg$ ("not"), $\rightarrow$ ("implies"). The notion of *path* is central to CL. A path is a sequence of name-value pairs; for example, the following is an existing path in the tree from Figure 3:

device=switch-1, interface=fe02, encapsulation=dot1q

For the sake of simplicity, we shall represent paths in the shorthand form $\overline{p} = \overline{x}$, where $\overline{p}$ is a list of names and $\overline{x}$ is a list of values or variables standing for actual values; hence, the notation $\overline{p} = \overline{x}$ is simply a shorthand way of writing $p_1 = x_1$, $p_2 = x_2$, ..., $p_n = x_n$. The *domain function* ν is used to query the contents of a tree T according to some path $\overline{p} = \overline{x}$. More precisely, $\nu(T; \overline{p} = \overline{x}, p)$ returns the set of all values for parameter p at the end of path $\overline{p} = \overline{x}$ in tree T.

$$
\begin{aligned}
T \models \neg\varphi &\equiv T \not\models \varphi \\
T \models \varphi \vee \psi &\equiv T \models \varphi \text{ or } T \models \psi \\
T \models \varphi \wedge \psi &\equiv T \models \varphi \text{ and } T \models \psi \\
T \models \varphi \rightarrow \psi &\equiv T \not\models \varphi \text{ or } T \models \psi \\
T \models \langle \overline{p} = \overline{x};\, p = x \rangle\, \varphi(x) &\equiv T \models \varphi(k) \text{ for some } k \in \nu(T; \overline{p} = \overline{x}; p) \\
T \models [\overline{p} = \overline{x};\, p = x]\, \varphi(x) &\equiv T \models \varphi(k) \text{ for each } k \in \nu(T; \overline{p} = \overline{x}; p) \\
T \models k_1 = k_2 &\equiv k_1 \text{ and } k_2 \text{ have the same value}
\end{aligned}
$$

Table 1: The recursive semantics of Configuration Logic

The universal quantifier, identified by [ ], indicates a path in the tree and imposes that a formula be true for all nodes at the end of that path. For example, a formula of the form $[\text{device} = s_1]\ s_1 \neq \text{abc}$ asserts that for every root node with name "device" and value s_1, then s_1 does not equal "abc". In other words, no device has "abc" as the value of its top-level node. Likewise, the existential quantifier, identified by $\langle\ \rangle$, indicates a path in the tree and imposes that a formula be true for some node at the end of that path.

Quantification in CL is *location-based*, as there can be multiple nodes with the same name (e.g., p). The set of possible values for p is relative to the particular subtree of the global configuration one is talking about. This is why a quantifier in CL always carries the path $\overline{p} = \overline{x}$ leading to a subtree (if any) that may refer to previously quantified variables.

A tree T is said to satisfy some CL formula φ, and is noted $T \models \varphi$, when the recursive evaluation of φ on T returns true. The complete semantics of CL is summarized in Table 1; the recursive application of these rules provides a *bona fide* algorithm for validating any CL formula on any configuration tree. It shall be noted that the evaluation of a quantifier successively replaces the occurrences of its variable by a set of values determined by ν; hence, the base case for the recursion always amounts to the comparison of two hard values.

For the sake of simplicity, it shall also be noted that ground terms are considered to be equality comparisons between two values. The semantics of CL can easily be extended to allow arbitrary predicates or arity greater than 2, of which classical equality is but a particular case.

For example, consider the following CL formula:

CL Constraint 1

$$
\begin{aligned}
[\text{device} = s_1]\ \langle \text{device} = s_1\,;\, \text{vtp mode} = x \rangle\, x = \text{client} \\
\vee\, \langle \text{device} = s_1\,;\, \text{vtp mode} = x \rangle\, x = \text{server}
\end{aligned}
$$

This formula reads as follows: for every root node with name "device" and value s_1, there exists a node under "device $= s_1$" with name "vtp mode" and value x, such

that x is equal to "client", or that there exists a node under "device $= s_1$" with name "vtp mode" and value x, such that x is equal to "server". In the example configuration shown in Figure 4, we see that a node exists with the name `vtp mode` and that its value is `server`. This constraint is therefore true for that particular device.

4.2.2 Network constraints in CL

We will use the examples presented in Section 2 to write those constraints into configuration logic formulas.

For instance, the constraint *"No router can have the same address for more than one interface"* can be written as the following CL formula:

$$[\text{router} = r_1]\,[\text{router} = r_1\,;\text{interface} = i_1]\,[\text{router} = r_1\,;\text{interface} = i_2]$$
$$[\text{router} = r_1, \text{interface} = i_1\,;\text{ip-address} = ip_1]$$
$$[\text{router} = r_1, \text{interface} = i_2\,;\text{ip-address} = ip_2]\;(i_1 \neq i_2 \rightarrow ip_1 \neq ip_2)$$

This means that for a router r_1 with two interfaces i_1 and i_2 having IP address as ip_1 and ip_2, then ip_1 and ip_2 can never be equal.

Similarly, we can express the constraint *"Every switch has to be a VTP client or a VTP server"* can be written as:

$$[\text{device} = s_1]\;(\langle \text{device} = s_1\,;\text{vtp mode} = x\rangle\, x = \text{client}$$
$$\vee \langle \text{device} = s_1\,;\text{vtp mode} = x\rangle\, x = \text{server})$$

This means that for every root node with name "device" and value s_1, there exists a node under "device = s_1" with the name "vtp mode" and value x, so that x is equal to "client", or that a node exists under "device = s_1" with the name "vtp mode" and value x, so that x is equal to "server".

4.2.3 Predicates

To improve the readability of CL rules, ValidMaker introduces the concept of *predicates*. The use of predicates follows the same goal as the decomposition of a computer program into functions: they are blocks of CL code that can be defined as Boolean functions, and then called and reused in many CL formulas. Predicates are expressed in the same way as formulas but can contain arguments. For example, consider the following predicate:

$$\text{IsVTPClient}(S) \text{ :- } \langle S\,;\text{vtp mode} = x\rangle\, x = \text{client}$$

This predicate states that under the node S passed as an argument, there exists a node whose name is "vtp mode" and whose value is x, and where x is equal to "client". In other words, this predicate returns true whenever S has a child labelled "vtp mode = client". The predicate IsVTPServer is defined in a similar way.

$$\text{IsVTPServer}(S) \text{ :- } \langle S\,;\text{vtp mode} = x\rangle\, x = \text{server}$$

Equipped with these predicates, it is possible to simplify CL Constraint 1 and rewrite it in an alternate way:

CL Constraint 1 (Alternate)

$$[\text{device} = s_1]\ (\text{IsVTPServer}(s_1) \vee \text{IsVTPClient}(s_1))$$

Starting from basic, low-level predicates that refer directly to configuration parameters, one defines increasingly higher-level predicates that progressively abstract these configuration details to capture important functions. The alternate version of CL Constraint 1 shows it. At the top level, network constraints can be expressed as a set of broad policies that the network engineer can easily manage. Therefore, the use of predicates in ValidMaker is an easy and straightforward way to encapsulate relationships and *roles* between parameters. This feature is in line with the suggestions of [4].

ValidMaker provides an interface that allows users to input their own constraints and predicates. It can also import a set of predefined CL constraints associated to a particular network service or policy. The CL validation functionality is exposed to the user as a simple menu entry.

4.3 A ValidMaker Use Case Scenario

In this section, we develop a simple configuration example based on the Virtual Trunking Protocol for Virtual Local Area Networks (VLANs) and show how our methodology provides a general environment for formalizing and automatically validating constraints on actual device configurations. VLANs are indeed recognized as a specific area of pain that requires the support of configuration management tools, yet is often neglected in real-world tools. The reader should keep in mind, however, that our methodology is not tied to a particular device or protocol; earlier works formalize constraints on Virtual Private Networks [23] in a similar way.

4.3.1 Virtual Local Area Networks and the Virtual Trunking Protocol

Switches allow a network to be partitioned into logical segments through the use of VLANs. This segmentation is independent of the physical location of the users in the network. The ports of a switch can be assigned to a particular VLAN. Ports that are assigned to the same VLAN are able to communicate at Layer 2 while ports not assigned to the same VLAN require Layer 3 communication. There can be numerous VLANs on a single switch and all the stations of a VLAN can be distributed on many switches. All the switches that need to share Layer 2 intra-VLAN communication need to be connected by a link called a *trunk*. The trunk joins two interfaces, one on each switch, and these interfaces should be encapsulated in the same *mode*. IEEE 802.1Q [6] and VTP [7] are two popular protocols for VLAN trunks.

The VLAN configuration must be entered on each switch where this VLAN is required. Otherwise, if a port is assigned to a non-existing VLAN then the port is disabled. The Virtual Trunking Protocol (VTP) [7] has been developed on Cisco devices to centralize the creation and deletion of VLANs in a network into a VTP *server*. This server takes care of creating, deleting, and updating the status of existing VLANs to the other switches sharing the same VTP *domain*. The clients that are in the

same VTP domain of the server will update their VLAN list according to the update. The switches that are in transparent mode will simply ignore the transmission but will nevertheless broadcast it so that other switches might get it.

Consider a network of switches where several VLANs are available. In order to have a working VTP configuration, the network needs a unique VTP server; all other switches must be VTP clients. This can be enforced by a first set of two constraints:

Configuration Constraint 1 *VTP must be activated on all switches.*

Configuration Constraint 2 *There is a unique VTP server.*

Using Configuration Logic, these requirements can be expressed in terms of predicates and configuration parameters. Configuration Constraint 1 requires that every switch be either a VTP client or a VTP server; CL Constraint 1 shown in Section 4.2 asserts exactly that. The second constraint makes sure that there is one, and only one, server in the network. It first states that there exists a device s_1, which is a VTP server, and then that every device s_2 different from s_1 is a VTP client.

CL Constraint 2 (UniqueServer)

$$\langle \text{device} = s_1 \rangle \, (\text{IsVTPServer}(s_1) \wedge [\text{device} = s_2] \, s_1 \neq s_2 \rightarrow \text{IsVTPClient}(s_2))$$

For the needs of the example, we impose that all switches be in the same VTP domain.

Configuration Constraint 3 *All switches must be in the same VTP domain.*

This constraint becomes the following CL formula. It states that for every pair of devices s_1 and s_2, the predicate "SwitchesInSameVTPDomain" (Table 2) is true. This predicate asserts that two switches are in the same VTP domain; this is done by checking that for the two nodes S and T that represent the root of the configuration tree of two devices, every VTP domain listed under S also appears under T.

CL Constraint 3 (SameVTPDomain)

$$[\text{device} = s_1] \, [\text{device} = s_2] \, \text{SwitchesInSameVTPDomain}(s_1, s_2)$$

Here we used a predicate called SwitchesInSameVTPDomain; this predicate asserts that two switches are in the same VTP domain. This is done by checking that, for the two nodes S and T, representing the root of the configuration tree of two devices, every VTP domain listed under S also appears under T. This predicate can be detailed as follows:

$$\text{SwitchesInSameVTPDomain}(S;T) : [S; \text{vtp domain} = x] \, \langle T; \text{vtp domain} = y \rangle \, x = y$$

Finally, we must impose a technical constraint on VLAN trunks and give its corresponding CL formula.

Configuration Constraint 4 *The interfaces at both ends of a trunk should be defined as such and encapsulated in the same mode.*

$$\text{IsTrunk}(I) \text{ :- } \langle I\,;\, \text{switchport mode} = x\rangle\, x = \text{trunk}$$

$$\text{SwitchesInSameVTPDomain}(S,T) \text{ :- } [S\,;\, \text{vtp domain} = x]\ \langle T\,;\, \text{vtp domain} = y\rangle\, x = y$$

$$\begin{aligned}\text{SameEncapsulation}(I_1,I_2) \text{ :- } & [I_1\,;\, \text{switchport encapsulation} = x_1] \\ & \langle I_2\,;\, \text{switchport encapsulation} = x_2\rangle\, (x_1 = \text{dot1q} \wedge x_2 = \text{dot1q}) \vee (x_1 = \text{isl} \wedge x_2 = \text{isl})\end{aligned}$$

Table 2: CL predicates required for the VLAN example.

In CL, this constraint becomes:

CL Constraint 4 (TrunkActive)

$$\begin{aligned}&[\text{device} = s_1]\ [\text{device} = s_2] \\ &\qquad [s_1\,;\, \text{interface} = i_1]\ \langle s_2\,;\, \text{interface} = i_2\rangle \\ &\qquad (\text{InterfacesConnected}(i_1,i_2) \rightarrow (\text{IsTrunk}(i_1) \\ &\qquad\qquad \wedge \text{IsTrunk}(i_2) \wedge \text{SameEncapsulation}(i_1,i_2)))\end{aligned}$$

The predicate IsTrunk (Table 2) indicates that the device is a VTP server and that a specific interface is connected to a trunk. Finally, the predicate SameEncapsulation verifies that the encapsulation on a VLAN trunk is either IEEE 802.11Q or ISL, and that both ends use matching protocols. The predicate "InterfacesConnected" is not a predicate defined in CL, but rather a system primitive that the network can tell us about. It returns true if the two interfaces are connected by a link.

4.3.2 Interactive Validation of CL Constraints

Once these constraints are defined (or imported), ValidMaker offers the possibility to automatically validate them on a set of device configurations forming a network. This validation does not merely return *true* or *false*. Rather, the CL validation algorithm extracts valuable information from the configuration to explain to the user where and why a given configuration violates a constraint. This interactive counter-example exploration is unique to ValidMaker and distinguishes it from related work described earlier.

In order to do so, ValidMaker returns to the user a part P of a configuration and a sub-formula φ not satisfied by P that is the cause of the violation. Such a pair (P,ψ) is called an *evidence*. The construction of the evidence for a falsified CL formula follows a recursive algorithm that depends on the structure of the formula that is false. Let T be a configuration tree, μ be a function that associates variables in a CL formula with the tree node it takes its value from, and ν be the evaluation function described earlier. An evidence is a set of tuples $\{(\varphi,S_1),\ldots,(\varphi,S_n)\}$, where φ is a formula and the S_i are themselves evidence for the subformulas of φ. Intuitively, an evidence is meant to be read top-down, with each of the S_i interpreted as alternate explanations for the falsity of φ, expressed in terms of φ's subformulas. Alternately, each evidence can be seen as a set of configuration elements to correct in order to restore the validity of φ.

$$
\begin{aligned}
\Xi_{T,\mu}(\varphi \wedge \psi) &= \{(\varphi \wedge \psi, \{\Xi_{T,\mu}(\varphi) \cup \Xi_{T,\mu}(\psi)\}\} \\
\Xi_{T,\mu}(\varphi \vee \psi) &= \{(\varphi \vee \psi, \Xi_{T,\mu}(\varphi)), (\varphi \vee \psi, \Xi_{T,\mu}(\psi))\} \\
\Xi_{T,\mu}(\langle \overline{p} = \overline{x};\, p = x \rangle\, \varphi(x)) &= \bigcup_{n \in \nu(T; \overline{p} = \overline{x}, p)} \{(\langle \overline{p} = \overline{x};\, p = x \rangle\, \varphi(x), \Xi_{T,\mu[x \to \overline{p} = \overline{x}, p = n]}(\varphi))\} \\
\Xi_{T,\mu}([\overline{p} = \overline{x};\, p = x]\, \varphi(x)) &= \left\{ (\langle \overline{p} = \overline{x};\, p = x \rangle\, \varphi(x), \bigcup_{n \in \nu(T; \overline{p} = \overline{x}, p)} \{\Xi_{T,\mu[x \to \overline{p} = \overline{x}, p = n]}(\varphi))\} \right\} \\
\Xi_{T,\mu}(x = y) &= \begin{cases} \emptyset & \text{if } \nu(T; \mu(x)) = \nu(T; \mu(y)) \\ \{(x = y, \{\mu(x), \mu(y)\})\} & \text{otherwise} \end{cases} \\
\Xi_{T,\mu}(x \neq y) &= \begin{cases} \emptyset & \text{if } \nu(T; \mu(x)) \neq \nu(T; \mu(y)) \\ \{(x = y, \{\mu(x), \mu(y)\})\} & \text{otherwise} \end{cases}
\end{aligned}
$$

Table 3: The recursive counter-example generation procedure

At the lowest level, the evidence is made of references to nodes $n_1, n_2, \ldots$ inside T, and a sub-formula giving the statement between these nodes that is violated. If the evidence contains a predicate, it is treated as an atomic unit (black box). However, at the choice of the user, it is possible to step into a predicate and refine the evidence; ultimately, predicates can be completely eliminated.

To this end, we define a procedure called $\Xi_{T,\mu}$ that recursively creates the evidence for a rule given a global configuration tree T, and a mapping μ (initially empty) between variables and tree nodes. This procedure is detailed in Table 3. It assumes that implications $\varphi \to \psi$ have been transformed into their equivalent form $\neg\varphi \vee \psi$, and that all negations in a CL formula have been pushed down to the lowest level using DeMorgan's identities.[3]

The case of the conjunction is straightforward. For a formula $\varphi \wedge \psi$ to be true, both φ and ψ must be true. It follows that the evidence for the falsity of $\varphi \wedge \psi$ is the combination of the evidence for the falsity of φ and ψ, noted $\Xi_{T,\mu}(\varphi) \cup \Xi_{T,\mu}(\psi)$. Similarly, for a formula $\varphi \vee \psi$ to be true, it suffices that either φ or ψ be true. The evidence for the falsity of $\varphi \vee \psi$ hence creates two nodes that represent the two possible ways by which its validity can be restored. The remaining cases are handled using a similar intuition.

In some cases, the source of the error can be unequivocally associated to the value of some parameter that does not fulfil the constraint. This is the case, for example, for a formula of the form $\varphi \wedge \psi$. For such a formula to be false, then either φ or ψ is false. The evidence returned by the algorithm is therefore the evidence of the first of φ or ψ that evaluates to false.

There are cases, however, where this evidence does not return anything. For example, a formula of the form $\langle \text{device} = x \rangle\, \varphi(x)$ asserts that there exists a node device $= x$ in the configuration, for some value x, such that $\varphi(x)$ is true. If the formula

[3] Namely: $\neg(\varphi \vee \psi) = \neg\varphi \wedge \neg\psi$, $\neg(\varphi \wedge \psi) = \neg\varphi \vee \neg\psi$, $\neg[\overline{p} = \overline{x};\, p = x]\, \varphi(x) = \langle \overline{p} = \overline{x};\, p = x \rangle\, \neg\varphi(x)$, $\neg\langle \overline{p} = \overline{x};\, p = x \rangle\, \varphi(x) = [\overline{p} = \overline{x};\, p = x]\, \neg\varphi(x)$.

is false, then no such node exists: $\varphi(x)$ is false for all possible values of x that occur in the tree.

ValidMaker uses the structure produced by the function Ξ internally to display its counter-examples to the user in an interactive mode. The validation process is halted, and the user is presented with a menu showing all the possible values of x, as is shown in Figure 4. He can then choose one of these values and resume the validation process on that specific branch, and find out why the branch falsifies the formula.

5 Lazy Evaluation of Configuration Constraints

The analysis of the protocols and management tools described in Section 3 raises a number of issues. Most importantly, all of these tools expect configurations to be handled in a centralized location. In the case of ValidMaker, the verification of configuration correctness is done locally on a mirror image of the complete network. Unless modifications to the whole network are managed solely through ValidMaker, the complete configuration of each device must be retrieved every time a verification is performed to avoid synchronization problems.

The assumption of a central location for the configuration of the whole network goes against the inherently decentralized nature of networks themselves. For example, routing protocols do not require a centralized and complete view of the network to properly assign routes to packets. Similarly, a wide body of work has concentrated over the years on an ever-increasing distribution in the management of various other aspects of a network: resource allocation [24], configuration access control [25], and configurations themselves [26, 27, 28]. The hypothesis of a single configuration repository, always in sync with each device's runtime state, may be unrealistic in many cases. For example, Cisco devices can autonomously negotiate VTP configuration parameters with their neighbors, and hence, change their configuration parameters by themselves. In other scenarios, it is flatly impossible, as in the case of autonomic networks [29, 30].

On a more technical side, keeping a central configuration state can also incur undesirable bandwidth consumption, to the point where control data occupies an unacceptable fraction of total bandwidth. The bandwidth consumed by such a procedure can be estimated through a back-of-the-envelope calculation; networks of 1,000 devices are not uncommon, and uncompressed configurations can exceed the size of the device's NVRAM (128 kB is a typical value) up to a factor of 3; this indicates that verifying the configuration amounts to retrieving almost 400 MB of data every time.[4] Unless modifications to the whole network are managed through the same centralized tool, the complete configuration of each device must be retrieved every time a verification is performed to avoid synchronization problems. This issue has long been recognized; discussions on the configuration retrieval capabilities of the NETCONF protocol from a decade ago already mention, "A less powerful filtering mechanism would cause more network traffic and thus increase network and CPU

[4] `http://www.cisco.com/c/en/us/td/docs/ios/12_2/configfun/configuration/guide/ffun_c/fcf007.html`

load since managers would have to retrieve more data than required if the required filtering cannot be done on the agent side."[5]

In this section, we present a strategy that follows that trend, and show how the verification of configuration constraints can be verified using only minimal centralized information. This strategy takes advantage of the fact that correctness rules are expressed in a logical formalism that allows them to be analyzed and their evaluation be optimized through systematic algorithmic processes. We call this set of techniques *lazy evaluation* of configurations, as they are similar in spirit to the concept of the same name in programming languages [31, 32].

5.1 Picking Configuration Subsets with NETCONF

The recursive application of the semantic rules given in Table 1 provides a *bona fide* algorithm for the evaluation of any CL formula on any configuration. Every time a quantifier needs to be evaluated, relevant values given the currently defined variables are queried on the configuration through the valuation function ν, and each such value spawns a new subformula to be evaluated recursively.

Unfortunately, as was already highlighted earlier, such an algorithm assumes random access to any part of the configuration at any time. Consider, for example, the following expression:

$$[a = x_1]\,[b = x_2]\,[a = x_1\,;\, c = x_3]\,[b = x_2\,;\, d = x_4]\,x_1 = x_2 \rightarrow x_3 = x_4$$

Applying the semantics of Table 1 will require first querying all values of parameter a; for each such value, one will then query all values of parameter b, followed by all values of parameter c under the a previously fetched. Finally, all values of parameter d under b will be queried, and the values retrieved will be compared. Assuming they are equal, the algorithm will backtrack to fetch a new value of parameter d under b, and eventually a new value of parameter c under a, and so on. Clearly, this is only feasible if these various parts of the configuration can be fetched very quickly, which rules out opening and closing a connection to a device every time a new value must be obtained. Hence, local access to the configuration seems the only reasonable option.

However, knowledge of the formal property to evaluate can be put to good use to reduce the amount of information that actually needs to be downloaded from each device. A first step in the lazy evaluation of a configuration for verification purposes consists in filtering out parts of the global configuration that are not relevant for the CL formula to verify. In the previous example, clearly, any path in a device's tree other than a, b, a/c and b/d has no impact on the outcome of the evaluation algorithm. Hence, one can only download portions of the configuration corresponding to the relevant paths. In such a setting, the algorithm still operates in a centralized fashion but operates on a pruned out version of the global configuration where only relevant parts of the global tree have been kept.

It turns out that network management protocols provide exactly that kind of functionality. In particular, NETCONF, following RFC 6241, provides so-called *XML*

[5] `http://psg.com/lists/netconf/netconf.2004/msg00448.html`

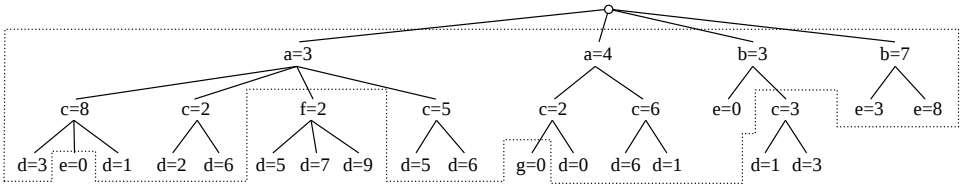

Fig. 5: A portion of a Meta-CLI configuration tree is illustrated.. Parameter names and values are abstract.

subtree filtering, which is a mechanism that allows an application to select particular XML subtrees to include in `get` and `get-config` RPC message replies. A small set of filters for inclusion, simple content exact-match, and selection is provided, which allows some useful (but also very limited) selection mechanisms.

Conceptually, a subtree filter is comprised of zero or more elements subtrees that represent the filter selection criteria [11]. At each containment level within a subtree, the set of sibling nodes is logically processed by the server to determine if its subtree and path of elements to the root are included in the filter output.

A first possibility is to select nodes according to the namespace they belong to. If assignment of namespaces to nodes is done carefully, nodes can be selected in a very flexible way, as was suggested in [33]. Otherwise, nodes can be selected according to specific criteria, as in the following snippet:

```
<filter type="subtree">
 <top xmlns="http://example.com/schema/1.2/config">
   <config_file>
     <parameter>
       <name_para>ipaddress</name_para>
     </parameter>
   </config_file>
 </top>
</filter>
```

In this example, the <config_file> and <parameter> nodes are called containment nodes, and <name_para> is a content match node. Since no sibling nodes of <name> are specified (and therefore no containment or selection nodes), all of the sibling nodes of <name_para> are returned in the filter output. Only “parameter” nodes in the `"http://example.com/schema/1.2/config"` namespace that match the element hierarchy and for which the <name> element is equal to “ipaddress” will be included in the filter output.

$$
\begin{array}{rcl}
([\overline{p} = \overline{x}; p = x] : \varphi) \wedge \psi & \Leftrightarrow & [\overline{p} = \overline{x}; p = x] : (\varphi \wedge \psi) \\
([\overline{p} = \overline{x}; p = x] : \varphi) \vee \psi & \Leftrightarrow & [\overline{p} = \overline{x}; p = x] : (\varphi \vee \psi) \\
(\langle \overline{p} = \overline{x}; p = x \rangle : \varphi) \wedge \psi & \Leftrightarrow & \langle \overline{p} = \overline{x}; p = x \rangle : (\varphi \wedge \psi) \\
(\langle \overline{p} = \overline{x}; p = x \rangle : \varphi) \vee \psi & \Leftrightarrow & \langle \overline{p} = \overline{x}; p = x \rangle : (\varphi \vee \psi) \\
\neg \langle \overline{p} = \overline{x}; p = x \rangle : \varphi & \Leftrightarrow & [\overline{p} = \overline{x}; p = x] : \neg \varphi \\
\neg [\overline{p} = \overline{x}; p = x] : \varphi & \Leftrightarrow & \langle \overline{p} = \overline{x}; p = x \rangle : \neg \varphi \\
([\overline{p} = \overline{x}; p = x]\ \varphi) \rightarrow \psi & \Leftrightarrow & \langle \overline{p} = \overline{x}; p = x \rangle\, (\varphi \rightarrow \psi) \\
(\langle \overline{p} = \overline{x}; p = x \rangle\ \varphi) \rightarrow \psi & \Leftrightarrow & [\overline{p} = \overline{x}; p = x]\, (\varphi \rightarrow \psi) \\
\varphi \rightarrow ([\overline{p} = \overline{x}; p = x]\ \psi) & \Leftrightarrow & [\overline{p} = \overline{x}; p = x]\, (\varphi \rightarrow \psi) \\
\varphi \rightarrow (\langle \overline{p} = \overline{x}; p = x \rangle\ \psi) & \Leftrightarrow & \langle \overline{p} = \overline{x}; p = x \rangle\, (\varphi \rightarrow \psi)
\end{array}
$$

Table 4: Rewriting rules for prenex normal form in CL, where φ and ψ are arbitrary expressions.

5.2 Further Filtering Through Local Evaluation

Consider the following formula:

$$
\begin{array}{l}
\langle \mathrm{a} = x_1 \rangle\ [\mathrm{b} = x_2]\ \langle \mathrm{a} = x_1 \,;\, \mathrm{c} = x_3 \rangle \\
\qquad\qquad [\mathrm{a} = x_1, \mathrm{c} = x_3 \,;\, \mathrm{d} = x_4]\ \langle \mathrm{b} = x_2 \,;\, \mathrm{e} = x_5 \rangle \\
\qquad\qquad\qquad\qquad x_1 = x_2 \wedge x_3 > x_1 \wedge x_4 = x_3 \wedge x_5 > x_2
\end{array}
$$

For this formula, basic subtree filtering corresponds in Figure 5 to sending only the nodes included in the dotted area of the tree. It is possible, however, to further filter the amount of data that needs to be sent to the centralized validator by performing a finer analysis of the actual values that may require to be compared with others. This algorithm is presented in the following.

5.2.1 PNF and DNF Rewriting

The first step is straightforward and consists of rewriting the CL expression in *prenex normal form* [34]. This has the effect of obtaining a logically equivalent expression, but where all quantifiers appear at the beginning of the formula. Table 4 lists the rewriting rules that are repeatedly applied until the prenex form is obtained.

The second part of this step consists of rewriting the non-quantifier part of the expression in Disjunctive Normal Form (DNF). By the result of the previous manipulations, the non-quantifier part of the formula is concentrated at the very end of the expression and can be transformed into DNF by repeatedly applying a number of transformation rules, as described in Table 5.

The result of this step is an expression of the following form:

$$Q_1 Q_2 \ldots Q_q (e_{1,1} \wedge \cdots \wedge e_{m_1,1}) \vee \cdots \vee (e_{n,1} \wedge \cdots \wedge e_{m_n,n})$$

$$
\begin{aligned}
\varphi \wedge (\psi \vee \psi') &\Leftrightarrow (\varphi \wedge \psi) \vee (\varphi \wedge \psi') \\
\varphi \vee (\psi \wedge \psi') &\Leftrightarrow (\varphi \vee \psi) \wedge (\varphi \vee \psi') \\
\neg(\varphi \vee \psi) &\Leftrightarrow \neg\varphi \wedge \neg\psi \\
\neg(\varphi \wedge \psi) &\Leftrightarrow \neg\varphi \vee \neg\psi \\
\neg\neg\varphi &\Leftrightarrow \varphi
\end{aligned}
$$

Table 5: Rewriting rules for disjunctive normal form in CL, where φ, ψ and ψ' are arbitrary expressions.

where all quantifiers occur at the beginning of the formula, followed by an expression of the form $\varphi_1 \vee \varphi_2 \vee \cdots \vee \varphi_n$. Each φ_i is itself a term of the form $e_1 \wedge e_2 \wedge \cdots \wedge e_m$, where e_i is an expression of the form $x_i \star i_j$ for some binary operator $\star$. By the rules of Table 4, this expression can then be rewritten as follows:

$$(Q_1 Q_2 \ldots Q_q(e_{1,1} \wedge \cdots \wedge e_{m_1,1})) \vee \cdots \vee (Q_1 Q_2 \ldots Q_q(e_{n,1} \wedge \cdots \wedge e_{m_n,n}))$$

We have now reached a point where the original constraint has been decomposed into a number of "alternatives", each of which is a quantified formula in PNF and composed only of conjunctions of Boolean comparisons between variables. The value of the global constraint can be obtained from the Boolean disjunction of each alternative. Since each alternative can be handled independently from the others, we describe the processing of a single such alternative in the following.

5.2.2 Identify Chains of Variable Dependencies

The third step is to identify dependencies between variables of the expression that can be evaluated locally. This is done by first analyzing the quantified part of the expression and extracting all maximal chains of quantifiers $Q_1 Q_2 \ldots Q_n$ so that Q_{i+1} extends the path defined in Q_i by one segment. These quantifiers need not follow each other directly in the original formula; however, they must occur in the proper order. For each chain, all constraints involving variables occurring in the chain are then appended, yielding a set of partial CL formulas containing all constraints local to each chain.

In our example expressions, two chains can be extracted, along with their respective constraints, namely:

$$\langle \mathrm{a} = x_1 \rangle \, \langle \mathrm{a} = x_1 \,;\, \mathrm{c} = x_3 \rangle \, [\mathrm{a} = x_1, \mathrm{c} = x_3 \,;\, \mathrm{d} = x_4] \, x_3 > x_1 \wedge x_4 = x_3$$

and

$$[\mathrm{b} = x_2] \, \langle \mathrm{b} = x_2 \,;\, \mathrm{e} = x_5 \rangle \, x_5 > x_2$$

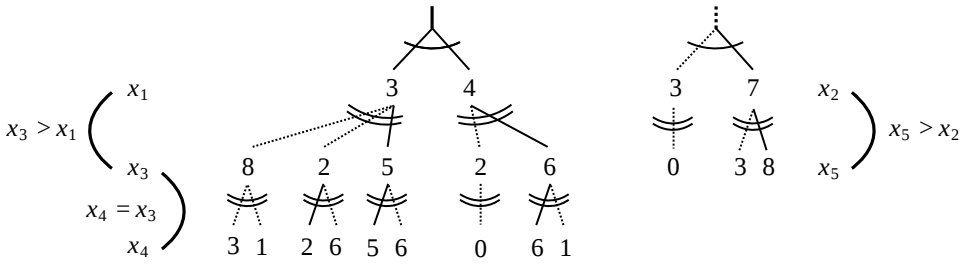

Fig. 6: The and/or tree for the configuration of Figure 5, for the two dependency chains identified earlier. Single arcs indicate conjunctions, and double arcs indicate disjunctions.

5.2.3 Identify Cross-Chain Dependencies

At the end of Step 3, it is possible that some conditions between variables have not been included into any chain expression. In our example, this is the case for $x_1 = x_2$, since it compares the values of variables between two distinct chains. To prepare each chain expression for local processing, we annotate each expression by identifying the variables over which a cross-chain dependency exists. In our present example, variables x_1 and x_2 would be marked as such.

5.2.4 Evaluate Locally

The complete set of chain expressions is then sent to every device, which is then required to evaluate them locally. This evaluation simply amounts to building one and/or tree for each chain, as shown in Figure 6. This and/or tree is then trimmed of any nodes and branches that do not satisfy one of the Boolean conditions attached to this chain. In the previous tree, branches for which a condition evaluates as false are dotted. For example, in the leftmost tree, node "8" is eliminated because none of its children satisfy the condition $x_4 = x_3$, while its sibling "4" is eliminated because it violates the condition $x_3 > x_1$.

This step ensures backwards compatibility for devices that do not support local evaluation of constraints. Instead of performing a complete filtering of its configuration, a device may elect to simply filter the configuration by keeping only the paths occurring in the quantifiers, without evaluating the Boolean conditions. In an extreme case, the device may simply ignore the chain expressions and return its whole configuration. In either case, performing a coarser filtering of the configuration has no impact on the validity of the final outcome of the algorithm.

5.2.5 Transmit Resulting Tree and Evaluate

In the last step, every device returns the filtered configuration resulting from its local processing of the chain expressions to the centralized validator. In the trees of Figure 6, this corresponds only to the paths that start from the root and are not dotted. One

can see that the fraction of the tree that actually needs to be sent is much smaller even than a mere filtering of the tree based on the paths occurring in quantifiers; in the first chain, only the subtree 3–5–5 and 4–6–6 is required, while in the second chain, only the subtree 7–8 is sent.

The resulting trees are then merged, and the centralized validator then proceeds to evaluate the *original* CL formula on this merged tree structure. The end result is that the centralized validator performs a normal evaluation of the property to verify, but on an expurgated copy of the global configuration containing only the parameters on which the global outcome is dependent.

The proof of the correctness of the algorithm is omitted, due to lack of space. However, intuitively, one can see that the parameter values trimmed locally on each device have no impact on the global result, as they make one of the conditions of the expression evaluate to false. Therefore, even if such values have dependencies with other variables in other chains (and hence possibly in other devices), they cannot make the global expression true and can then safely be ignored. Note that this is only possible if the original expression is composed only of conjunctions of atomic conditions, an hypothesis we can assume because of the transformation into PNF and DNF in Step 1.

As an additional optimization, if one assumes that all devices have evaluated the intra-chain conditions, then these conditions can be removed from the expression (more precisely, replaced by the constant *true*) upon evaluation at the centralized validator. In such a case, only the subset of the and/or tree containing variables subject to cross-chain dependencies needs to be sent. In our example, this corresponds only to nodes $a = 3$, $a = 4$ and $b = 7$.

One can see the potential for the reduction of data to be transfered to a centralized location. In our example, sending the complete configuration would require transmitting 30 nodes; applying simple subtree filtering reduces this number to 21. Computing the and/or tree and evaluating the conditions reduces it further down to 8, and sending only values subject to cross-chain dependencies takes it down to 3.

6 Experiments

To assess the feasibility of this approach and measure its impact on the amount of configuration data that needs to be transferred, we implemented the algorithm described in Section 5.2 and tested it on sample configuration trees. The results of these experiments are detailed in this section.

6.1 Implementation Details

The algorithm is implemented in Python, and is publicly available.[6] The UML class diagram in Figure 7 describes the data structure used to represent dependency chains. The main component is the CentralValidation that manages the whole process of

[6] `https://github.com/sylvainhalle/MetaConfig`. Note that the algorithm currently assumes that the input formula is already in PNF/DNF.

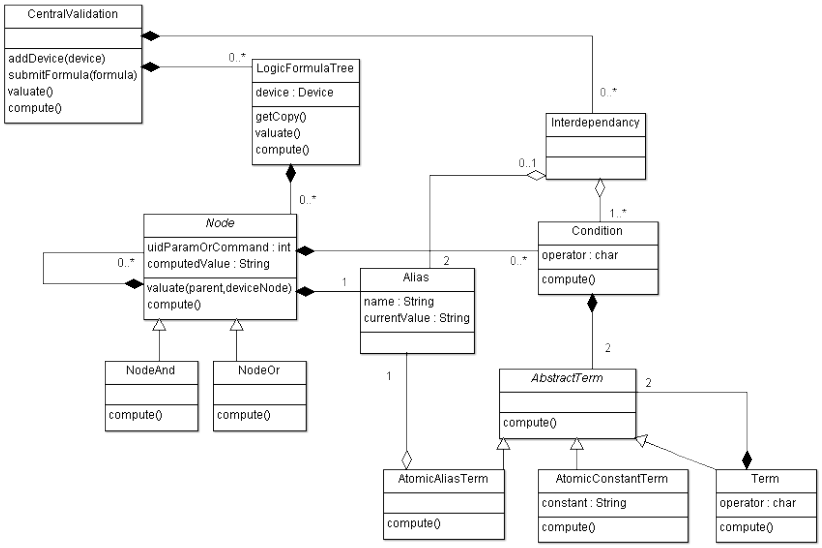

Fig. 7: Class diagram from lazy evaluation implementation

verifying a logical formula for devices of a given network. The abstract formula to compute is formatted as a tree (LogicFormulaTree) and stored in the CentralValidation. Each device receives a copy of this formula.

The valuate() method of the formula is used to valuate the nodes of the tree recursively, by browsing the corresponding device. It is important to say that only the values strictly needed to verify the formula are retrieved from the device. If there are nodes that remain without value and children, they are removed from the formula tree, in order to minimize space and time requirements. The necessary nodes are only those whose name is present in the formula to validate (see comprehensive example below).

The compute() method is called to evaluate the nodes according to their nature: NodeAnd (each child must return true), NodeOr (one child returning true is enough), Node (neutral, used for the leaves of the tree; only this node's condition must return true); the conditions attached to these nodes are also computed. Nodes whose conditions are false are removed. Thus, this function is recursive and returns a Boolean result. If interdependencies exist between remote nodes or between devices, the result is a tree that contains all the existing values of the interdependent parameters. In this case, the Central Validation retrieves a copy of all the LogicalFormulaTree produced by the devices and analyzes the interdependencies, if any.

The Alias implementation is used to assign a reference to a Node. The Condition is represented as a link between 2 AbstractTerms and an operator. These can either be AtomicAliasTerm (value assigned to a Node via an Alias), AtomicConstantTerm (a constant value) or another simple condition (this structure is recursive).

The input of our program is a CL formula and multiple device configurations expressed as Meta-CLI. The output is a Boolean value representing the outcome of the evaluation of the constraint on the multiple devices.

6.2 Results

We conducted several tests of the algorithm as described below. The device used for testing was a fictitious configuration tree with simple values, similar to the one shown in Figure 5.

The first CL formula tested on this configuration was:

$$[d = x]\ [d = x; a = y]\ y \leq 0$$

The computation of this formula must have returned true for the given device. This formula was stored as a tree in memory and sent to the device. It contains a reference to a command d and to the parameter a in the Meta-CLI file of a given device. The logical condition $y \leq 0$ is attached to the node parameter a.

Each device executed the computation locally. To check the formula, it was only required to retrieve one command name and one parameter value. In this example, it compared only the values 0 and -1 of parameter a under command d of its configuration. Hence, it required very little data compared to all the parameters contained in the complete configuration files. The amount of data to be retrieved for the validation of the formula on a device containing n nodes was the constant 4: two values for a and two for d.

We also tested 3 other formulas, as follows:

$$[d = x]\ [d = x; a = y]\ [d = x; b = z]\ y < z$$

$$\langle d = x\rangle\ \langle d = x; a = y\rangle\ \langle d = x; b = z\rangle\ z = 3 \wedge y = 0$$

$$\langle d = x\rangle\ \langle d = x; a = y\rangle\ [e = w]\ [e = w; c = z]\ y = -1 \wedge z = 3$$

We created multiple device configuration files as detailed in Table 6. Filesets contained from 1 to 10 files with 5 to 500 randomly generated nodes. For each CL formula and each configuration, we performed the evaluation of the configuration using both the lazy evaluation approach described in the previous section, and the "total" approach that required downloading the complete configuration of each device before starting the computation. In each case, we measured the amount of data that was required to be sent to the centralized validator in order to compute the result.

	C1	C2	C3
Number of devices	2	5	10
Number of nodes per device	5	100	500

Table 6: Sample configurations generated in the experiments.

Approach	Selective				Total			
Formula	1	2	3	4	1	2	3	4
C1	8	12	12	12	10	10	10	10
C2	20	30	30	30	500	500	500	500
C3	40	60	60	60	5000	5000	5000	5000

Table 7: Data exchanged in the devices

The results we obtained are listed in Table 7. In both scenarios, we counted as one unit of bandwidth each tree node that was transmitted by a device to the centralized validator. We can see that the amount of data exchanged in the lazy approach is much lower than in the total approach, some times by a factor of 100. One exception was configuration C1, whose small size, combined with the overhead of marshaling dependency chains to and from devices, resulted in a slightly higher bandwidth consumption for the lazy approach. However, one can see that the lazy approach was very effective as soon as the number of devices and the size of configurations reached a minimum threshold.

7 Conclusion

In this paper, we tackled the issue of network configuration correctness by showing with a sample of real-world scenarios how configuration parameters in network devices are subject to various dependencies. A study of available management protocols and tools showed that very few solutions offer capabilities for checking configurations according to user-defined rules, and that those that assume arbitrary access to a local copy of the complete configuration of the network. This, in turn, incurred high bandwidth consumption, as the configuration of the entire network had to be dumped to a central location every time a check for correctness had to be executed.

We then presented a processing strategy, based on the concept of lazy evaluation that attempted to retrieve as little information as possible from each device for centralized processing. This strategy is based on the fact that configuration constraints can be expressed formally in a language called Configuration Logic, and that CL expressions can be analyzed to devise filtering rules to be applied on each device. First, parts of the expression that can be evaluated locally were identified. These parts were sent to the device, which applied the required filtering and returned to the centralized validator a simplified version of its complete configuration. The validator then proceeded to its evaluation as usual, except on a trimmed down version of the configuration where parameters and values that did not matter in the final outcome were already removed. Early empirical results show that this technique presents the potential to greatly reduce the amount of data to be retrieved from each device, while still preserving the same guarantees on the correctness of the configuration.

This project lends itself to several extensions, optimizations and improvements. First, it was assumed that all snapshots of configurations (either managed centrally or in a distributed way) were consistent –that is, no external modification to the configuration could occur between the start and end of the validation operation. A mechanism for ensuring the atomicity of the distributed validation process should be considered.

Moreover, further trimming of the and-or trees produced by each device could be computed locally, by intersecting multiple and-or trees that share quantified variables, resulting in increased bandwidth savings. Identifying configuration parameters that are relevant for the Boolean outcome of a rule could also be used to create *alarms*, which would be triggered whenever some parameter present in a pre-computed and/or tree changes its value.

Acknowledgements The authors would like to acknowledge the work of Éric Wenaas on an early version of ValidMaker. The authors acknowledge the financial support of the Natural Sciences and Engineering Research Council of Canada and Ericsson Canada.

References

1. J. Strassner, "Bridge to IP profitability," 2002.
2. N. Feamster and H. Balakrishnan, "Detecting BGP configuration faults with static analysis," in *2nd Symp. on Networked Systems Design and Implementation (NSDI)*, (Boston, MA), pp. 43–56, May 2005.
3. A. Wool, "A quantitative study of firewall configuration errors," *IEEE Computer*, no. 6, pp. 62–67, 2004.
4. M. Burgess and A. Couch, "Modeling next generation configuration management tools," in *LISA*, pp. 131–147, USENIX, 2006.
5. S. Hallé, E. L. Ngoupe, G. Nijdam, O. Cherkaoui, P. Valtchev, and R. Villemaire, "Validmaker: A tool for managing device configurations using logical constraints," in *NOMS* [35], pp. 1111–1118.
6. "802.11Q: Virtual bridged local area networks standard," 2003. `http://standards.ieee.org/getieee802/download/802.1Q-2003.pdf`.
7. "Configuring VTP," 2007.
8. R. Villemaire, S. Hallé, and O. Cherkaoui, "Configuration logic: A multi-site modal logic," in *TIME*, pp. 131–137, IEEE Computer Society, 2005.
9. T. Delaet and W. Joosen, "A language for high-level configuration management," in *21st Large Installation System Administration Conference* (, ed.), pp. 131–137, Usenix Association, 2007.
10. M. Fedor, M. L. Schoffstall, and J. Davin, "An architecture for describing SNMP management frameworks (RFC 1157)," 1990.
11. R. Enns, M. Bjorklund, J. Schoenwaelder, and A. Bierman, "Network configuration protocol (NETCONF) (RFC 6241)," 2011.
12. M. Bjorklund, "A data modeling language for the network configuration protocol (NETCONF). (RFC 6020)," 2010.
13. M. Burgess, "A site configuration engine," *USENIX*, pp. 309–337, 1995.
14. J. Schönwälder, V. Marinov, and M. Burgess, "Integrating cfengine and scli: Managing network devices like host systems," in *IEEE/IFIP Network Operations and Management Symposium: Pervasive Management for Ubioquitous Networks and Services, NOMS 2008, 7-11 April 2008, Salvador, Bahia, Brazil*, pp. 1067–1070, IEEE, 2008.

15. SolarWinds CatTools, "Cattools help," 2012. http://www.kiwisyslog.com/help/cattools/mnu_filedbimportdevicefrmtab.htm.
16. Puppet Labs investors, 2013. http://puppetlabs.com/solutions/juniper-networks.
17. M. Taylor and S. Vargo, *Learning Chef*. O'Reilly, 2014.
18. P. Goldsack, J. Guijarro, S. Loughran, A. N. Coles, A. Farrell, A. Lain, P. Murray, and P. Toft, "The SmartFrog configuration management framework," *Operating Systems Review*, vol. 43, no. 1, pp. 16–25, 2009.
19. J. Lobo, R. Bhatia, and S. A. Naqvi, "A policy description language," in *Proceedings of the Sixteenth National Conference on Artificial Intelligence and Eleventh Conference on Innovative Applications of Artificial Intelligence, July 18-22, 1999, Orlando, Florida, USA*. (J. Hendler and D. Subramanian, eds.), pp. 291–298, AAAI Press / The MIT Press, 1999.
20. D. Agrawal, S. B. Calo, K.-W. Lee, and J. Lobo, "Issues in designing a policy language for distributed management of IT infrastructures," in *Integrated Network Management*, pp. 30–39, IEEE, 2007.
21. N. Damianou, N. Dulay, E. Lupu, and M. Sloman, "The Ponder policy specification language," in *POLICY* (M. Sloman, J. Lobo, and E. Lupu, eds.), vol. 1995 of *Lecture Notes in Computer Science*, pp. 18–38, Springer, 2001.
22. "Object constraint language version 2.2," tech. rep., February 2010. http://www.omg.org/spec/OCL/2.2.
23. S. Hallé, R. Deca, O. Cherkaoui, and R. Villemaire, "Automated validation of service configuration on network devices," in *MMNS* (J. B. Vicente and D. Hutchison, eds.), vol. 3271 of *Lecture Notes in Computer Science*, pp. 176–188, Springer, 2004.
24. D. Tuncer, M. Charalambides, G. Pavlou, and N. Wang, "Dacorm: A coordinated, decentralized and adaptive network resource management scheme," in *2012 IEEE Network Operations and Management Symposium, Maui, HI, USA, April 16-20, 2012* [35], pp. 417–425.
25. L. Seitz, G. Selander, E. Rissanen, C. Ling, and B. Sadighi, "Decentralized access control management for network configuration," *J. Network Syst. Manage.*, vol. 16, no. 3, pp. 303–316, 2008.
26. M. Burgess, "Theory and practice of configuration management in decentralized systems," in *Management of Integrated End-to-End Communications and Services, 10th IEEE/IFIP Network Operations and Management Symposium, NOMS 2006, Vancouver, Canada, April 3-7, 2006. Proceedings* (J. L. Hellerstein and B. Stiller, eds.), p. 583, IEEE, 2006.
27. F. L. Koch and C. B. Westphall, "Decentralized network management using distributed artificial intelligence," *J. Network Syst. Manage.*, vol. 9, no. 4, pp. 375–388, 2001.
28. M. Kahani and P. Beadle, "Decentralized approaches for network management," *Computer Communications Review*, vol. 27, no. 3, pp. 36–47, 1997.
29. S. Hallé, É. Wenaas, R. Villemaire, and O. Cherkaoui, "Self-configuration of network devices with configuration logic," in *Autonomic Networking* (D. Gaïti, G. Pujolle, E. S. Al-Shaer, K. L. Calvert, S. A. Dobson, G. Leduc, and O. Martikainen, eds.), vol. 4195 of *Lecture Notes in Computer Science*, pp. 36–49,

Springer, 2006.
30. N. Agoulmine, *Autonomic Network Management Principles*. Academic Press, 2011.
31. P. Henderson and J. M. Jr., "A lazy evaluator," in *Proceedings of the 3rd ACM SIGACT-SIGPLAN Symposium on Principles on Programming Languages*, pp. 95–103, ACM, 1976.
32. D. P. Friedman and D. S. Wise, "CONS should not evaluate its arguments," in *ICALP*, pp. 257–284, 1976.
33. S. Hallé, O. Cherkaoui, and P. Valtchev, "Towards a semantic virtualization of configurations," in *NOMS* [35], pp. 1268–1271.
34. E. Mendelson, *Introduction to Mathematical Logic, Fourth Edition*. Springer, 1997.
35. *2012 IEEE Network Operations and Management Symposium, Maui, HI, USA, April 16-20, 2012*, IEEE, 2012.

Author Biographies

Éric Lunaud Ngoupé is currently an Analyst as CGI. Prior to that, he completed a PhD in Computer Science at Université du Québec à Chicoutimi under the supervision of Pr. Sylvain Hallé.

Clément Parisot is currently a graduate student at Télécom Nancy, France. Prior to that, he completed an internship at Université du Québec à Chicoutimi in the *Laboratoire d'informatique formelle* (LIF).

Sylvain Stoesel is currently a graduate student at Télécom Nancy, France. Prior to that, he completed an internship at Université du Québec à Chicoutimi in the *Laboratoire d'informatique formelle* (LIF).

Petko Valtchev is a Full Professor in the Department of Computer Science at Université du Québec à Montréal, Canada. His research interests include methods for efficient mining of patterns and association rules; condensed representations of patterns and rules; on-line mining of patterns and rules; pattern/rule mining on top of complex data records; design and exploitation of ontologies for deductive and inductive reasoning; mining on the Semantic Web.

Roger Villemaire is a Full Professor in the Department of Computer Science at Université du Québec à Montréal, Canada. His research interests include model checking, formal methods, satisfiability solving, and their applications to web services and computer networks.

Omar Cherkaoui is a Full Professor in the Department of Computer Science at Université du Québec à Montréal, Canada. His research interests include network management (standardization, protocols, configuration, validation, modeling, testing), and optical networks. He has supervised more than ten projects in the domain of high-speed network management, web services platforms, and new multimedia software.

Pierre Boucher is Director of Research at Ericsson Canada. He participated in the design and implementation of the configuration validation solutions described in this paper.

Sylvain Hallé is an Associate Professor in the Department of Computer Science and Mathematics at Université du Québec à Chicoutimi, Canada, and a Senior Member of both the IEEE and ACM. He completed a PhD in Computer Science at Université du Québec à Montréal in 2008. His research interests include automated verification, formal methods and their application to computer networks.