ValidMaker: A Tool for Managing Device Configurations Using Logical Constraints

Sylvain Hallé, Éric Lunaud Ngoupé, Gaëtan Nijdam
Département d'informatique et de mathématique
Université du Québec à Chicoutimi, Canada

Omar Cherkaoui, Petko Valtchev, Roger Villemaire
Département d'informatique
Université du Québec à Montréal, Canada

***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. There exists 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. A CL validation engine has been integrated into a network management tool called ValidMaker. We show on a simple use case scenario based on Virtual Local Area Networks how representative formal constraints can be expressed with CL and efficiently validated with ValidMaker.**

I. 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 that 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: [2] revealed more than 1,000 errors in the BGP configuration of 17 networks; [3] studied firewalls from a quantitative aspect and reported that all of them were misconfigured in some way or another.

How can one be assured that a service installed on a network works correctly? In a prospective paper on 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 $p_1, \dots, p_n$ with domains $D_1, \dots, D_n$ 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 work was supported by the Natural Sciences and Engineering Council of Canada in partnership with Cisco Systems.

This task, already non-trivial at the onset, is becoming increasingly hard because of the fulgurating 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 this paper, we present a network configuration validation tool called ValidMaker, following this principle. Using 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 formulæ.

The use of a logic-based formalism for configuration management provides unique benefits that are highlighted in ValidMaker. First, the verification of configurations on multiple devices can be done very efficiently, surpassing by orders of magnitude figures available from past works and presenting quadratic complexity in the size of the configurations. Second, the approach enforces a clean separation between the specification of configuration 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 an 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 existing tools, which provide a mere yes/no answer. By a simple and representative use case based on the implementation of a Virtual Local Area Network (VLAN), we show how ValidMaker can be used to formalize network constraints and detect errors in a problematic VLAN configuration.

The paper is structured as follows. In Section II, we present related work and motivate the need for a validation methodology using logical constraints. Section III presents the general architecture of ValidMaker and introduces both the data model and syntax of Configuration Logic. Section IV is the core of the paper and shows ValidMaker's validation capabilities on the VLAN example. In Section V, we provide empirical evidence that the CL engine consumes negligible resources. Finally, Section VI concludes and announces future work.

II. Current Approaches to Network Configuration Management

The existing works that address the management and integrity of network configurations fall into five broad categories.

A. Network Monitoring

A large number of network management solutions rely on monitoring external parameters such as connectivity and throughput to ensure a correct operating of the network. Some of them, such as the Minerals project [5], use statistical or artificial intelligence reasoning techniques to infer device misconfigurations from these observations. Network management tools like ManageEngine[1] perform a similar kind of monitoring on the network.

Although these solutions allow a relatively efficient detection of anomalous behaviour, they provide limited support to trace the cause of this behaviour back to the actual configuration parameters. Another drawback is that errors are detected *a posteriori*: unless realistic simulations are conducted before every change made to the network, one has to wait that the erroneous configurations be committed before discovering that something does not work as expected.

B. Configuration Change Management

A second category of solutions includes tools that manage changes in the configuration of devices. For example, the Really Awesome New Cisco confIg Differ (RANCID) system [6] periodically logs into each router of a network, retrieves its configuration and commits it to a Concurrent Version System (CVS) server where changes can be detected and tracked. Many other tools, such as the open source ZipTie [7] and the commercial Voyence [8], offer similar functionalities for comparing configurations and maintaining the history of configuration changes.

However, even if detecting changes in a configuration represents a good principle, not all changes result in misconfigurations: because of this, numerous false alarms can be triggered. Moreover, even if the source of a misconfiguration is pinned down to one specific parameter that has changed, the reason of the error still has to be figured out manually by the network engineer.

[1]http://manageengine.com

C. Decision Trees and Expert Systems

Another possibility for the diagnostic of configuration problems is to pragmatically describe and standardize network troubleshooting procedures as "decision trees" that describe tests to make and corresponding actions to take on a system. Each node in such a tree represents a test or an action to perform on the system; different edges in the tree are taken depending on the result of each test, until a working solution is found. The use of decision trees originally required human intervention but for the simplest tasks; recent tools like Babble [9] and Snitch [10] can now generate scripts that can interact with command-line tools to automate repetitive administration tasks.

Bayesian networks extend this principle by adding probabilities to the decision tree structure. This approach has been taken by numerous real-world projects, such as BATS (Bayesian Automated Troubleshooting System) [11], a system used by Hewlett-Packard to troubleshoot printer problems using the SACSO methodology [12]. ATSIG[2] is an EU project developing a concept for automation of troubleshooting processes, that has been turned into a commercial product called 2solve; a commercial troubleshooting tools like Knowledge Automation System from Vanguard uses a similar approach.[3]

D. Reactive Rules

Several systems extend on this idea and are based on rules of the form "if *condition* then *action*", which allows automated management of complex computer systems by triggering user-defined scripts when specific conditions are met in the network. Notable proponents of this approach include cfengine [13], LCFG [14], PIKT [15], Bcfg2 [16] and Prodog [17]. These works concentrate on the configuration of *computers* forming a network. However, it is reasonable to think that the approach could be extended to the configuration of network devices such as routers and switches.

This approach is "reactive" in that the action part of a particular rule is executed only when specific conditions described in its *if* clause match the configuration. Therefore every detected misconfiguration must be matched with a corrective action. A central —and still open— question is to ensure that the execution of a script does not fire an unpredictable cascade of events that never settles to a stable point. A mathematical study of reactive rules is made in [18], where conditions are given for actions to be *convergent*. Moreover, [19] argued that to be truly used as a misconfiguration detection engine, reactive systems must be provided with a base of rules that would amount to a procedural encoding of the whole engine.

E. Declarative Rules

A final approach is to specify constraints that a configuration must respect in order to be valid. The rules are assertions about the configuration of the devices, and these assertions are automatically checked before committing any changes.

[2]http://www.2operate.com/

[3]http://www.vanguardsw.com/products/knowledge-automation-system/cms/

For instance, [20] uses proof plans methods and Prolog to check orders for large computer systems parts; [21] applies a logic based on features/values attributions pairs to versioning in software configuration; [22] uses a logic on parse trees to express constraints on programming language syntax for design constraint of software components.

More recently, a prototype tool for verifying the configuration of network routers has been presented in [23]. The *rcc* tool [2] has been introduced to formalize rules about BGP configuration; the authors concentrate on BGP routes and do not extend this principle to the general case of formalizing misconfigurations. In [24], a formal approach to modelling constraints for Virtual Private Networks using first-order logic has been employed to demonstrate traffic isolation properties. In [25], Delta-X, a formal language for data integrity constraints is presented for the construction of *integrity guards*: an integrity guard is a piece of code executed before a data update is performed. The guard returns true if the update will preserve data integrity. [19], [26] uses the Alloy modelling language [27] to formalize a set of constraints for a VPN to properly work; this set of constraints is then converted into a Boolean formula and sent to a satisfiability solver. The solution returned by the solver can be converted back into the original Alloy model and constitutes a configuration that respects the original constraints. More recent tools, such as COOLAID [28], use a similar approach of declarative configuration management.

The approach followed by ValidMaker falls into this category of solutions, which is the one closest to the concept of aspects presented in Section I. However, ValidMaker distinguishes itself from the above cited works on a number of points. Contrarily to [23], ValidMaker provides a formal language to input rules of a more complex nature; as a consequence, it can provide more detailed error messages and interactive validation of the configuration (cf. Section IV-B). Moreover, existing tools lack the counter-example exploration feature implemented in ValidMaker and described in Section IV-B; they merely provide yes/no answers regarding the validity of some configuration.

The formal language it provides to express constraints, called Configuration Logic [29], is richer than Delta-X; it can be used to model general dependencies between configuration parameters and is not tied to a particular protocol or type of device as in *rcc*. Contrarily to the Alloy approach, we do not attempt to solve the broad problem of generating a configuration fulfilling a set of constraints; we rather concentrate on the narrower problem of checking that a *given* configuration actually respects the constraints.

Although the authors in [19], [26] suggest that validation can be indirectly performed as a by-product of their tool, we will see in Section V that a dedicated validation algorithm is considerably more efficient than a requirement solver, with processing times in the order of milliseconds rather than several minutes.

III. The ValidMaker Network Management Tool

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.[4]

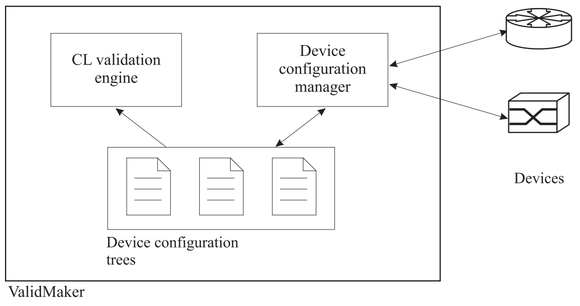


Figure 1. The architecture of the ValidMaker configuration tool.

The tool serves two main purposes. First, it levels the heterogeneity of devices across multiple platforms 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, ValidMaker tool is composed of two modules, as shown in Figure 1. We briefly describe these two modules.

A. 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 vendor operating system and version number.

As explained in [30], 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 is supported; the actual process by which the configurations are converted is beyond the scope of the present paper.

B. Configuration Logic Validation Engine

Once the device configurations are abstracted into Meta-CLI trees of name-value pairs, ValidMaker allows the network

[4]A limited version of ValidMaker is freely available for academic use at http://www.leduotang.com/sylvain/publications/2012/manfi.

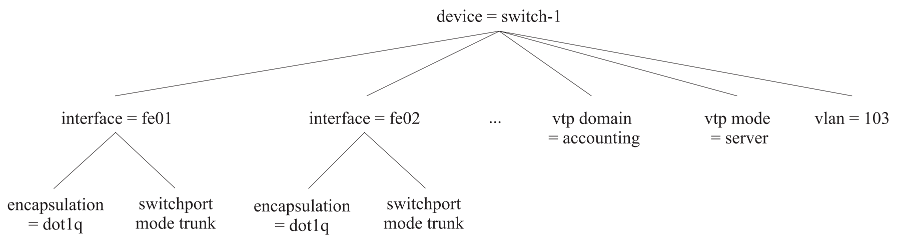
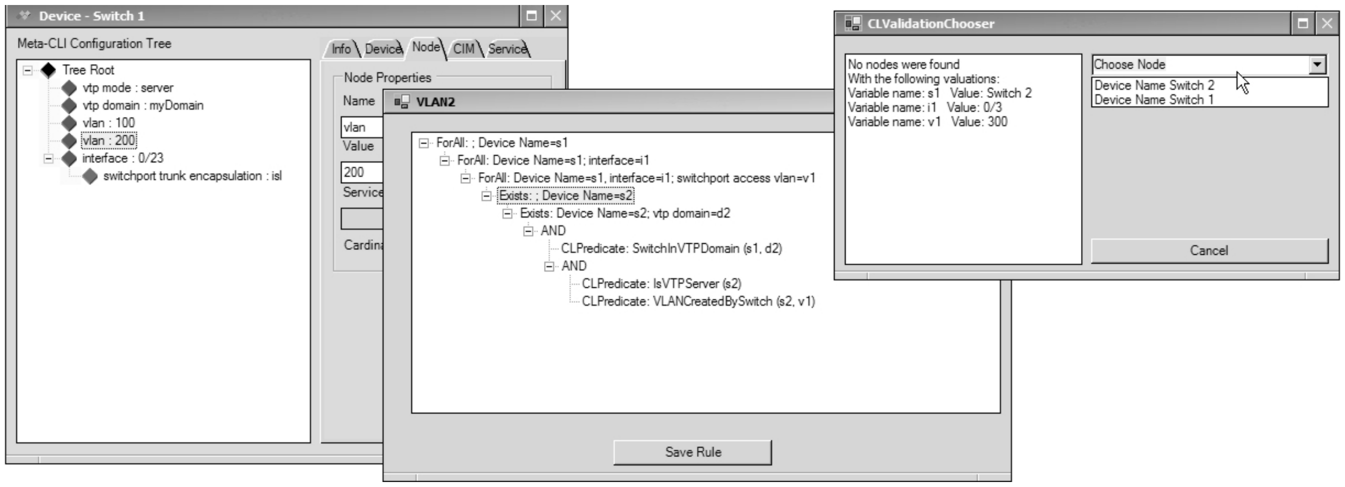


Figure 3. A portion of a Meta-CLI configuration tree.



Meta-CLI Configuration Tree

```
Tree Root
  vtp mode : server
  vtp domain : myDomain
  vlan : 100
  vlan : 200
  interface : 0/23
    switchport trunk encapsulation : isl
```

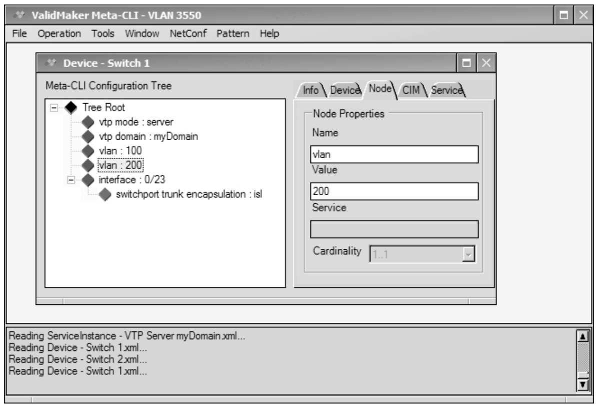
Name: vlan
Value: 200

```
ForAll: ; Device Name=s1
  ForAll: Device Name=s1; interface=i1
    ForAll: Device Name=s1, interface=i1; switchport access vlan=v1
      Exists: ; Device Name=s2
        Exists: Device Name=s2; vtp domain=d2
          AND
            CLPredicate: SwitchInVTPDomain (s1, d2)
            AND
              CLPredicate: IsVTPServer (s2)
              CLPredicate: VLANCreatedBySwitch (s2, v1)
```

```
No nodes were found
With the following valuations:
Variable name: s1   Value: Switch 2
Variable name: i1   Value: 0/3
Variable name: v1   Value: 300
```

```
Device Name Switch 2
Device Name Switch 1
```

Figure 4. 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.



Meta-CLI Configuration Tree

```
Tree Root
  vtp mode : server
  vtp domain : myDomain
  vlan : 100
  vlan : 200
  interface : 0/23
    switchport trunk encapsulation : isl
```

Name: vlan
Value: 200

```
Reading ServiceInstance - VTP Server myDomain.xml...
Reading Device - Switch 1.xml...
Reading Device - Switch 2.xml...
Reading Device - Switch 1.xml...
```

Figure 2. A screenshot from ValidMaker's configuration view. The configuration of a switch is abstracted as a tree of name-value pairs.

engineer to express formal constraints on these trees with the means of Configuration Logic (CL) [29].

1) Syntax and Semantics of CL: CL formulæ 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. 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.

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. Quantifiers are an important and distinctive part of CL; as Narain observed [31], none of the constraints described in our example below could be expressed in Prolog or any tool based upon it, since Prolog lacks the equivalent of the universal quantifier.

A tree T is said to satisfy some CL formulæ 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 I;

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

Table I
THE RECURSIVE SEMANTICS OF CONFIGURATION LOGIC

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 example, consider the following CL formula:

CL Constraint 1.

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

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 there exists a node with name `vtp mode` and that its value is `server`. This constraint is therefore true for that particular device.

2) Predicates: To improve 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 formulæ. Predicates are expressed in the same way as formulæ 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.

IV. 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 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 [30] in a similar way.

A. Virtual Local Area Networks and the Virtual Trunking Protocol

Switches allow a network to be partitioned into logical segments through the use of Virtual Local Area Networks (VLAN). 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 [32] and VTP [33] 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) [33] has been developed on Cisco devices to centralize the creation and deletion of VLANs in a

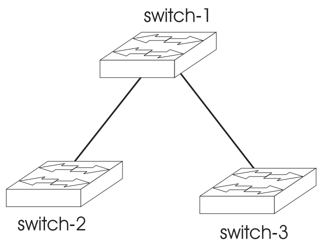


Figure 5. A simple cluster of switches in the same VLAN. The links are VLAN trunks.

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 such as the one shown in Figure 5 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 III-B 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 II) is true. This predicate asserts that two switches are in the same VTP domain; this is done by checking that for 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.

CL Constraint 3 (SameVTPDomain)**.**

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

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

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

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

Table II
CL PREDICATES REQUIRED FOR THE VLAN EXAMPLE.

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.*

This constraint becomes in CL:

CL Constraint 4 (TrunkActive)**.**

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

The predicate IsTrunk (Table II) 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.

B. 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 which is the cause of the violation. 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 evidences for the subformulæ 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 subformulæ. Alternately, each evidence can be seen as a set of configuration elements to correct in order to restore the validity of φ.

At the lowest level, the evidence is made of references to nodes $n_1,n_2,\ldots$ inside T, and a sub-formula stating the relationship 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 III. It assumes that implications $\varphi \rightarrow \psi$ 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.[5]

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, which represent the two possible ways by which its validity can be restored. The remaining cases are handled using a similar intuition. For example, a formula of the form $\langle \text{device} = x \rangle\, \varphi(x)$ asserts that there exists a node $\text{device} = x$ in the configuration, for some value x, such that $\varphi(x)$ is true. If the formula 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.

V. Experimental Results

It is known that the worst-case complexity of CL model checking is $O(|\varphi| \times |T|^k)$, where $|\varphi|$ is the length of the formula, $|T|$ is the size of the configuration tree and k is the maximal number of nested quantifiers in the formula [34]; this also applies to the computation of Ξ. For a fixed set

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

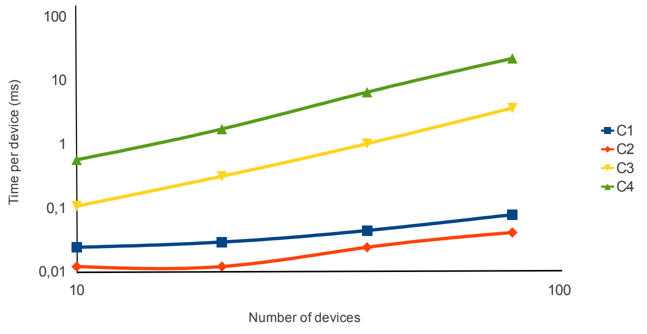


Figure 6. Validation time of each VTP constraint in a network formed of a varying number of devices. The graph uses a log-log scale.

of CL formulæ, the validation hence scales with respect to the number of managed devices (and configuration size) as a polynomial of degree at most k. This result indicates that, from a theoretical point of view, CL validation and interactive counter-example exploration is tractable and that its complexity remains manageable as the configurations grow in size. However, it remains to be shown that validation times are reasonable for realistic configurations.

To this end, we generated sample configurations for networks composed of a variable number of switches implementing a VLAN. We then launched ValidMaker's CL engine on these networks for each the four Configuration Constraints shown in Section IV. We give in Figure 6 the validation times for a network composed of 10 to 80 switches. All results have been obtained on a Pentium IV of 2.4 GHz with 512 Mb of RAM running Windows XP Professional.

As one can see from these results, the validation times are reasonable and do not exceed 25 milliseconds per device for the largest network of 80 switches. More importantly, although the theoretical upper bound predicted a growth of $O(|T|^5)$ for Configuration Constraint 4 (which contains five nested quantifiers once its predicates are expanded), in actuality we observe a sub-quadratic complexity in the order of $|T|^{1.8}$. Moreover, these figures should be compared with those mentioned in [19]. As has been explained in Section II, Alloy tries to *build* from scratch a configuration that fulfils all the constraints, instead of validating an existing configuration against a set of rules. Using first-order logic formulæ with a number of quantifiers similar to the CL constraints used in this paper and a configuration of under 50 devices, the total processing time ranges from 2 to 8 *minutes*. The considerable difference in CPU time shows that validation is a problem much more tractable than model building.

VI. Conclusion

The introduction of aspect-oriented configuration management in [4] shows that formalisms are essential in order to validate network configurations before deployment. The integration of CL into the ValidMaker configuration tool shows that a logic following the hierarchical structure familiar to network engineers gives a natural and effective framework to express and verify network configuration properties. Experimental results

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

Table III
THE RECURSIVE COUNTER-EXAMPLE GENERATION PROCEDURE

show that this approach is tractable in practice. The interactive counter-example exploration feature allows network engineers to effectively use the logic and troubleshoot configurations before deployment to the network. Based on the promising results obtained on initial use cases, extensions to the tool are under way, which will take into account the decentralized nature of configuration information and the validation of incomplete configuration trees.

REFERENCES

[1] J. Strassner, "Bridge to IP profitability," 2002. [Online]. Available: http://www.intelliden.com/library/GlobalOSS_BridgetoIP45.pdf

[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, May 2005, pp. 43–56. [Online]. Available: http://nms.csail.mit.edu/rcc

[3] A. Wool, "A quantitative study of firewall configuration errors," *IEEE Computer*, pp. 62–67, June 2004.

[4] M. Burgess and A. Couch, "Modeling next generation configuration management tools," in *LISA*. USENIX, 2006, pp. 131–147.

[5] F. Le, S. Lee, T. Wong, H. S. Kim, and D. Newcomb, "Minerals: Using data mining to detect router misconfigurations," Carnegie Mellon University, Tech. Rep. CMU-CyLab-06-008, May 2006.

[6] "RANCID - really awesome new cisco config differ." [Online]. Available: http://www.shrubbery.net/rancid/

[7] R. Castillo, "ZipTie network inventory framework: Enabling the next era of network management tools," p. 10, December 2006. [Online]. Available: http://www.alterpoint.com/index.php?s=file_download&id=7

[8] "Voyence." [Online]. Available: http://www.voyence.com/

[9] A. L. Couch, "An expectant chat about script maturity," in *LISA*. USENIX, 2000, pp. 15–28.

[10] J. Mickens, M. Szummer, and D. Narayanan, "Snitch: Interactive decision trees for troubleshooting misconfigurations," in *SysML*. USENIX, 2007, pp. 1–6.

[11] H. Langseth and F. V. Jensen, "Decision theoretic troubleshooting of coherent systems," *Reliability Engineering and System Safety*, vol. 80, pp. 49–62, 2003.

[12] F. V. Jensen, U. Kjærulff, B. Kristiansen, H. Langseth, C. Skaanning, J. Vomlel, and M. Vomlelová, "The SACSO methodology for troubleshooting complex systems," *AI EDAM*, vol. 15, no. 4, pp. 321–333, 2001.

[13] M. Burgess, "cfengine: A site configuration engine," *Computing Systems*, vol. 8, no. 2, pp. 309–337, 1995.

[14] P. Anderson and A. Scobie, "LCFG: The next generation," pp. 1–9, January 2002. [Online]. Available: http://www.lcfg.org/doc/ukuug2002.pdf

[15] "PIKT," http://www.pikt.org/. [Online]. Available: http://www.pikt.org/

[16] N. Desai, R. Bradshaw, and C. Lueninghoener, "Directing change using Bcfg2," in *LISA*. USENIX, 2006, pp. 215–220.

[17] A. L. Couch and M. Gilfix, "It's elementary, dear Watson: Applying logic programming to convergent system management processes," in *LISA*. USENIX, 1999, pp. 123–138. [Online]. Available: http://www.usenix.org/publications/library/proceedings/lisa99/couch.html

[18] A. L. Couch and Y. Sun, "On the algebraic structure of convergence," in *DSOM*, ser. Lecture Notes in Computer Science, M. Brunner and A. Keller, Eds., vol. 2867. Springer, 2003, pp. 28–40.

[19] S. Narain, "Network configuration management via model finding," in *LISA*. USENIX, 2005, pp. 155–168.

[20] H. Lowe, "Extending the proof plan methodology to computer configuration problems," *Applied Artificial Intelligence*, vol. 5, no. 3, pp. 227–252, 1991.

[21] A. Zeller and G. Snelting, "Unified versioning through feature logic," *ACM Trans. Softw. Eng. Methodol.*, vol. 6, no. 4, pp. 398–441, 1997.

[22] N. Klarlund, J. Koistinen, and M. I. Schwartzbach, "Formal design constraints," in *OOPSLA*, 1996, pp. 370–383.

[23] A. Feldmann and J. Rexford, "IP network configuration for intradomain traffic engineering," *IEEE Network*, vol. 15, no. 5, pp. 46–57, 2001.

[24] R. Bush and T. Griffin, "Integrity for virtual private routed networks," in *INFOCOM*, 2003.

[25] M. Benedikt and G. Bruns, "On guard: Producing run-time checks from integrity constraints," in *AMAST*, ser. Lecture Notes in Computer Science, C. Rattray, S. Maharaj, and C. Shankland, Eds., vol. 3116. Springer, 2004, pp. 27–41.

[26] S. Narain, G. Levin, S. Malik, and V. Kaul, "Declarative infrastructure configuration synthesis and debugging," *J. Network Syst. Manage.*, vol. 16, no. 3, pp. 235–258, 2008.

[27] D. Jackson, "Alloy: a lightweight object modelling notation," *ACM Trans. Softw. Eng. Methodol.*, vol. 11, no. 2, pp. 256–290, 2002.

[28] X. Chen, Y. Mao, Z. M. Mao, and J. E. van der Merwe, "Declarative configuration management for complex and dynamic networks," in *CoNEXT*, J. C. de Oliveira, M. Ott, T. G. Griffin, and M. Médard, Eds. ACM, 2010, p. 6.

[29] R. Villemaire, S. Hallé, and O. Cherkaoui, "Configuration logic: A multi-site modal logic," in *TIME*. IEEE Computer Society, 2005, pp. 131–137.

[30] S. Hallé, R. Deca, O. Cherkaoui, and R. Villemaire, "Automated validation of service configuration on network devices," in *MMNS*, ser. Lecture Notes in Computer Science, J. B. Vicente and D. Hutchison, Eds., vol. 3271. Springer, 2004, pp. 176–188.

[31] *Proceedings of the 19th Conference on Systems Administration (LISA 2005), San Diego, USA, December 4-9, 2005*. USENIX, 2005.

[32] "802.11Q: Virtual bridged local area networks standard," p. 327, 2003, http://standards.ieee.org/getieee802/download/802.1Q-2003.pdf.

[33] "Configuring VTP." [Online]. Available: http://www.cisco.com/en/US/products/hw/switches/ps708/products_configuration_guide_chapter09186a008019f048.html

[34] S. Hallé, R. Villemaire, and O. Cherkaoui, "CTL model checking for labelled tree queries," in *TIME*. IEEE Computer Society, 2006, pp. 27–35.

[35] *Proceedings of the 20th Conference on Systems Administration (LISA 2006), Washington, D.C., USA, December 3-8, 2006*. USENIX, 2006.