Self-configuration of Network Devices with Configuration Logic

Sylvain Hallé, Éric Wenaas, Roger Villemaire, and Omar Cherkaoui

Université du Québec à Montréal
C.P. 8888, Succ. Centre-ville
Montréal (Canada) H3C 3P8
halle@info.uqam.ca

Abstract. Autonomic networking is an emerging approach to the management of computer networks that aims at developing self-governed devices. Among the main issues of autonomic systems is the question of self-configuration. In this paper, we describe a method for discovering and self-generating the configuration of a network device in order to dynamically push a new service into a network. On each configuration, several rules representing the semantics of the services are expressed in a logical formalism called Configuration Logic. From these rules, we show how to use traditional satisfiability methods to automatically generate or modify the configuration of a device with respect to the configuration of its neighbours. We illustrate our case with an example of a switch that automatically discovers its VLAN configuration when connected to an existing network. The results presented here have been implemented into the configuration management tool ValidMaker.

1 Introduction

Despite the tremendous development of network services and functionalities over the years, the configuration and deployment of network elements such as routers and switches remains a mostly manual task. An intricate knowledge of each devices' and services' inner workings and dependencies between configuration parameters is required from the network engineer in order to successfully run even basic use cases. The addition of a new device or the deployment of a new service to an existing infrastructure requires repetitive but careful manipulation of multiple configuration parameters on many elements, and even such a cautious work can spawn unpredicted side effects that are discovered by trial and error.

The application of the autonomic systems paradigm [19] to computer networks offers a promising means to release the burden of knowledge and tedious manipulations currently needed from engineers. By definition, self-governed devices can automatically modulate their behaviour in reaction to configuration inconsistencies or changes in the topology or management policies of the network.

We gratefully acknowledge the financial support of the Natural Sciences and Engineering Research Council of Canada on this research.

In this paper, we describe a method of automatically discovering and generating the configuration of a network device. Each configuration is represented in the form of a hierarchy of parameter-value pairs that is assimilated to a labelled tree. The dependencies between these parameters are then expressed as self-rules in a special formalism called Configuration Logic (CL) [22].

From the CL formula defining each rule, we show how an action can be automatically associated; this action consists in a number of configuration operations whose end result ensure that the rule is fulfilled. A CL validation engine can automatically check whether a given set of self-rules are respected in a network and trigger the appropriate actions associated with the rules that are violated.

The system then uses a standard Boolean satisfiability test to generate a plan that properly instantiates parameter values and chooses between conflicting actions; in the case where more than a single plan is possible, the final choice can then be passed on to a higher-level policy manager. This method can be used for integrating self-configuring and self-healing functionalities in any autonomic network where devices' configurations are represented by trees. We illustrate it by a simple example on Virtual LANs in which a new switch is connected and discovers its configuration in a plug-and-play manner based on data obtained from other devices in the network.

The rest of the paper is structured as follows. Section 2 presents related work. Section 3 presents the VLAN use case that illustrates our method and elicits a number of VLAN configuration self-rules. Section 4 presents the tree structure used to describe configurations and formalises the configuration rules using Configuration Logic. Section 5 extends the original VLAN scenario to allow for self-configuration and shows how violated VLAN rules trigger configuration operations. Finally, section 6 concludes and indicates possible future work.

2 Related Work

The autonomic approach to network management has been the source of numerous related work in the recent years. Companies like Cisco and Motorola are developing similar concepts respectively called *programmable networks* [2] and *cognitive networks*; the idea has also been developed in [16].

The SELFCON [5] project developed a self-configuring environment based on the Directory-enabled Networking (DEN) [21] principles, where devices register at a centralised directory server that notifies them of changes in configuration policies or network state. Similarly, the AUTONOMIA [15] environment provides an autonomic architecture for automated control and management of networked applications. In [20], a suite of protocols compatible with TCP/IP is described that could be implemented into autonomic, agent-based "smart routers" that take decisions about which protocol in the suite should be used to optimise some user-defined goals. All these projects describe an infrastructure in terms of high level concepts and do not concentrate on the representation, validation and actual generation of configurations and rules, but rather provide an environment in which these operations can take place. The agent approach is extended in [11] from a quality of service perspective and is currently under development.

In [12], the parameters of an optical network are automatically reconfigured based on link traffic using regression techniques. However, the range of legal values of these parameters is fixed and known in advance and the reconfiguration only aims at finding an optimal adjustment of existing values: the network itself is supposed to be properly working at any moment. Our work rather attempts to structurally modify a configuration by adding and removing parameters. Moreover, in our situation, the legal range of values changes from time to time and our method attempts to discover that range from the configuration rules.

The GulfStream software system [10] provides a dynamic topology discovery and failure detection for clusters of switches in multiple VLANs; the approach is not based on the examination of the configuration of other switches, but rather on the broadcasting of Beacon and heartbeat messages between VLAN peers and is somewhat restrained to this particular situation.

Our approach also differs from well-established configuration systems like Cfengine [7] in that only the desired properties of the configuration are expressed in a declarative way, but no action or script must be specified by the user. The method we present automatically determines the proper actions to take on the configuration in order to fulfill the desired rules.

Finally, a lot of work has been published about self-configuration applied to wireless sensor networks. In this context, the word “configuration” generally refers to the topological arrangment of the different elements forming the sensor mesh, and not the logical parameters that regulate the behaviour of a device in itself. It is therefore only faintly related to the present work.

3 Constraints and Rules in Network Services

In this section, we develop a configuration example based on Virtual Local Area Nertworks and the Virtual Trunking Protocol. We express configuration self-rules for a configuration of this type to be functional. This example will later be used to show how we can find the appropriate configuration of a device in a self-configuring and self-healing context.

3.1 Virtual Local Area Networks and the Virtual Trunking Protocol

Switches allow a network to be partitionned 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 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. All the switches that need to share Layer 2 intra-VLAN communication need to be connected by a *trunk*. IEEE 802.1Q [3] and VTP [1] are two popular protocols for VLAN trunks.

In principle, for a VLAN to exist on a switch, it has to be manually created by the network enginneer on the said switch. The Virtual Trunking Protocol (VTP) [1] has been developped on Cisco devices to centralise 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*.

3.2 Constraints on VLAN Configurations

Our configuration example involves VTP. Consider a network of switches such as the one shown in Figure 1 where several VLANs are available. We express a number of VTP self-rules that must be true across this network.

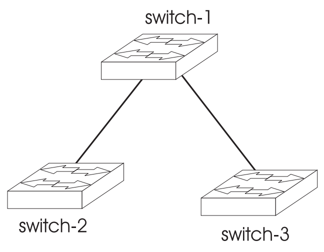

Fig. 1. A simple cluster of switches in the same VLAN. The links are VLAN trunks.

First, in order to have a working VTP configuration, the network needs a unique VTP server; all other switches must be VTP clients. This calls for a first set of two self-rules:

VTP Self-Rule 1 *The VTP must be activated on all switches.*

VTP Self-Rule 2 *There is a unique VTP server.*

We impose that all switches be in the same VTP domain, and that if two switches are connected by a trunk, then this trunk must be encapsulated in the same mode on both interfaces. This gives us two more constraints that should be true in all times:

VTP Self-Rule 3 *All switches must be in the same VTP domain.*

VTP Self-Rule 4 *The interfaces at both ends of a trunk should be defined as such and encapsulated in the same mode.*

4 Configurations and Rules

In this section, we give a brief overview of Configuration Logic (CL), a logical formalism over tree structures developed for expressing properties of configurations. We show how the VTP Self-Rules described in the previous section can be rewritten in CL to become Formal VTP Self-Rules.

4.1 Representing Configurations

As explained in [13], 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 to the configuration of such devices. As an example, Figure 2 shows a tree representation of the configuration of `switch-1` in the network of Figure 1.

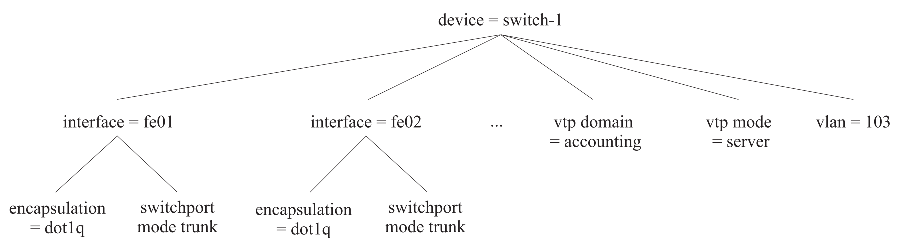

Fig. 2. A portion of the configuration of the `switch-1` in the network of Figure 1. The configuration of `switch-2` and `switch-3` differs in the VTP mode and trunk information.

Building a tree from an XML document is trivial; therefore, the representation of configurations as trees closely matches the XML nature of a protocol such as Netconf [9] that uses this format to fetch and modify the configuration of a device.

4.2 A Logic for Configurations

Configuration Logic [22] was developped in order to express properties on configuration trees. CL formulas use the traditional Boolean connectives of predicate logic: $\wedge$ ("and"), $\vee$ ("or"), $\neg$ ("not"), $\rightarrow$ ("implies"), to which two special quantifiers are added. 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. 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.

To improve readability of CL rules, we can use predicates. Predicates are expressed in the same way as rules, with the exception that arguments which correspond to already quantified nodes can be specified. For example, to create a predicate that returns true if a switch is a VTP Client, one can do so by writing:

$$\begin{aligned}&\mathrm{IsVTPClient}(S)\ :-\\&\quad\langle S\,;\,\text{vtp mode} = x\rangle\, x = \text{client}\end{aligned}$$

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 is true whenever S has a child labelled "vtp mode = client", which means that the device is indeed a VTP client.

Similarly, the predicates IsVTPServer and IsTrunk respectively indicate that the device is a VTP server and that a specific interface is connected to a trunk. They are defined as the following:

$$\begin{aligned}&\mathrm{IsVTPServer}(S)\ :-\\&\quad\langle S\,;\,\text{vtp mode} = x\rangle\, x = \text{server}\end{aligned}$$

$$\begin{aligned}&\mathrm{IsTrunk}(I)\ :-\\&\quad\langle I\,;\,\text{switchport mode} = x\rangle\, x = \text{trunk}\end{aligned}$$

The next 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.

$$\begin{aligned}&\mathrm{SwitchesInSameVTPDomain}(S,T)\ :-\\&\quad[S\,;\,\text{vtp domain} = x]\\&\qquad\langle T\,;\,\text{vtp domain} = y\rangle\, x = y\end{aligned}$$

Finally, this last predicate verifies that the encapsulation on a VLAN trunk is either IEEE 802.11Q or ISL, and that both ends use matching protocols:

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

For more details about CL, the reader is directed to [14, 22, 23].

4.3 Rules Expressed in Configuration Logic

Equipped with the previous predicates, the VTP self-rules mentioned in section 3 can now be expressed in Configuration Logic. The first rule checks whether VTP is activated on all switches.

Formal VTP Self-Rule 1 (VTPActivated)

$$\begin{aligned}&[\text{device} = s_1]\\&\quad\mathrm{IsVTPServer}(s_1) \vee \mathrm{ISVTPClient}(s_1)\end{aligned}$$

This is done by stating that for every device s_1, either s_1 is a VTP server, or s_1 is a VTP client. Note that this rule uses the predicates defined above to simplify its expression; however, the predicates are not necessary, and their definition could have simply been copied in the rule instead.

The second rule 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. Remark that this rule subsumes the previous one, which has still been included to illustrate later concepts.

Formal VTP Self-Rule 2 (UniqueServer)

$$\begin{aligned}
&\langle \text{device} = s_1 \rangle \\
&\quad (\text{IsVTPServer}(s_1) \wedge \\
&\qquad [\text{device} = s_2] \\
&\qquad\quad s_1 \neq s_2 \rightarrow \text{IsVTPClient}(s_2))
\end{aligned}$$

The third rule checks that all switches in the network have the same VTP Domain.

Formal VTP Self-Rule 3 (SameVTPDomain)

$$\begin{aligned}
&[\text{device} = s_1] \\
&\quad [\text{device} = s_2] \\
&\qquad \text{SwitchesInSameVTPDomain}(s_1, s_2)
\end{aligned}$$

It does so by stating that for every pair of devices s_1 and s_2, the predicate "SwitchesInSameVTPDomain" defined previously is true.

Finally, the fourth rule checks that if two interfaces are connected, then these two interfaces must be defined as trunks and have the same encapsulation protocol.

Formal VTP Self-Rule 4 (TrunkActive)

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

The relation "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. This requires knowledge of the topology of the network and depends on the architecture of the autonomic component, as will be discussed in the next section.

5 A Self-Configuration Scenario

In this section, we show how to use Configuration Logic to trigger configuration changes in devices. Figure 3 presents a possible architecture using this method. It shows how an existing tool called ValidMaker [8] that downloads and uploads configuration trees from the configuration file stored on routers and switches, can be used off-the-shelf to provide autonomic behaviour to the devices. A CL validation engine is implemented within ValidMaker. Therefore, one can load configurations, define self-rules on these configurations and automatically verify them. If one or many rules happen to be false, an additional configuration generator can adjust the configuration trees and generate the new configurations which can then be put back on the devices.

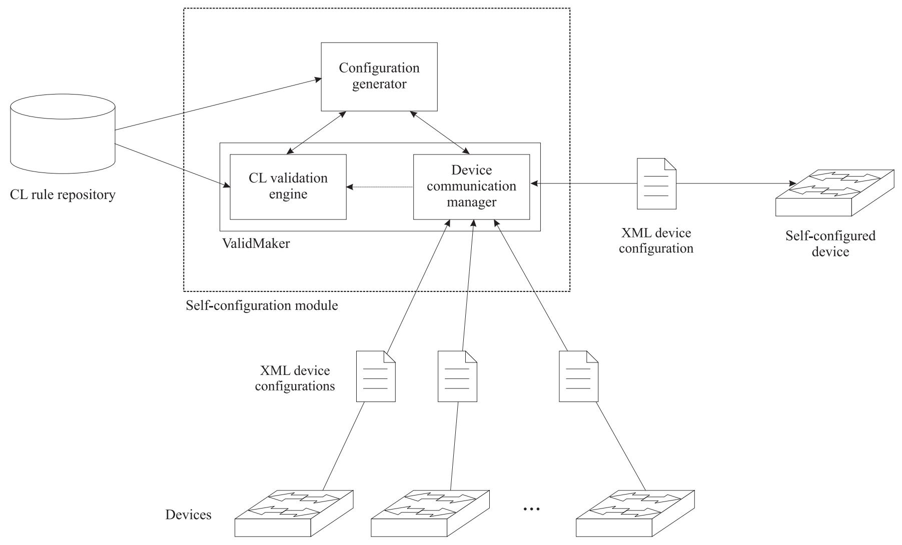

Fig. 3. A possible self-configuring architecture using off-the-shelf components.

To this end, we revisit the original VLAN scenario described in section 3 from a network autonomic viewpoint. Instead of validating configuration rules on a static, fully formed cluster of switches, we suppose one switch is connected to the cluster with a blank configuration, as shown in Figure 4. More precisely, the interface fe04 of `switch-4` is connected to the interface fe06 of `switch-3`. The initial configuration tree of `switch-4` is is a tree with only one root node labelled “device=switch-4”.

Based on this use case, we describe a method that enables the switch to automatically discover its configuration.

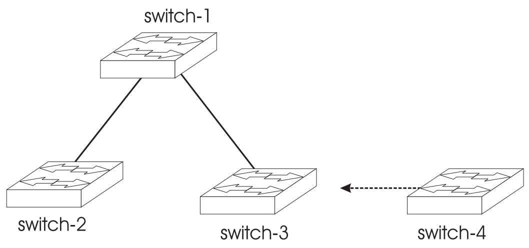

Fig. 4. Autonomic use case. Switch 4 is connected to Switch 3 and attempts to discover its configuration.

5.1 Active Constraints

The network rules shown in section 3.2 have been up to now interpreted in a static way: their validation allowed us to determine whether a configuration satisfied them or not. However, it is possible to go further and automatically associate with each rule an action, thereby making it a "self-rule". This action, in line with the semantics of the operators used in the rule, is generated such that executing it changes the configuration in a way that fulfills the rule that was originally proved false.

As an example, consider a rule of the form $\langle \mathrm{p} = x\rangle\, \varphi(x)$. Such a rule imposes the existence of a node $\mathrm{p} = x$ in the configuration of the device. If it is violated, this means that no such tuple $\mathrm{p} = x$ exists; to make this rule true, the node must be added to the configuration. This addition of a new node corresponds to a configuration operation equivalent to typing a command (or a sequence of commands) to the CLI of the device. Moreover, the value x of this parameter must be such that $\varphi(x)$ is true.

For such a procedure to be possible, the formulas it attempts to fulfill must be expressed in a logic where the existence of a model is a *decidable* problem. Otherwise, it is not always possible to find and construct a configuration satisfying a given formula. Therefore, the choice of CL as the logical formalism of formal rules is not arbitrary: it has been shown in [23] that under reasonable conditions, CL is a decidable logic. Using a richer logic such as XPath or TQL would lead to undecidability, as has been shown in [6].

Hence, by recursively descending through the nested CL subformulas, it is possible to come up with a series of configuration operations and constraints on their parameters that make every self-rule true. To this end, we define a procedure called $\textsc{Plan}_T$ that recursively creates the plans for each rule given a global configuration tree T. The special notation $\oplus(\overline{p} = \overline{x}\,;\, p = x)$ indicates that a node of parameter p and value x must be added to the tree at the tip of the branch $\overline{p} = \overline{x}$. This procedure is defined in Table 1.

As an example, consider the initial configuration of `switch-4` when added to the configuration of the original cluster of switches shown in Figure 2. It is clear that Formal VTP Self-Rule 1, which imposes the presence of either

$$\begin{aligned}
\textsc{Plan}_T(\varphi) &= 1 \text{ if } T \models \varphi \\
\textsc{Plan}_T(\neg\varphi) &= \neg\textsc{Plan}_T(\varphi) \\
\textsc{Plan}_T(\varphi \wedge \psi) &= \textsc{Plan}_T(\varphi) \wedge \textsc{Plan}_T(\psi) \\
\textsc{Plan}_T(\varphi \vee \psi) &= \textsc{Plan}_T(\varphi) \vee \textsc{Plan}_T(\psi) \\
\textsc{Plan}_T(\langle \overline{p} = \overline{x}\,;\, p = x\rangle\, \varphi(x)) &= \oplus(\overline{p} = \overline{x}\,;\, p = x) \wedge \textsc{Plan}_T(\varphi) \\
\textsc{Plan}_T([\overline{p} = \overline{x}\,;\, p = x]\, \varphi(x)) &= \bigwedge_x \textsc{Plan}_T(\varphi)
\end{aligned}$$

Table 1. The recursive plan generation procedure

server or client definition on every switch, is violated for `switch-4`. The recursive application of Plan on this formula produces the following action:

$$\begin{aligned}&\oplus(\text{device} = \text{switch-4}\,;\, \text{vtp mode} = \text{server}) \vee \\ &\oplus(\text{device} = \text{switch-4}\,;\, \text{vtp mode} = \text{client})\end{aligned} \tag{1}$$

which commands that the VTP role of the switch be determined. This role is specified by the addition of either node vtp mode = server or vtp mode = client under the root of `switch-4`'s configuration tree.

Similarly, Formal VTP Self-Rule 2 is violated. It imposes that there is only one server in the network and that all other be clients, and produces the following action:

$$\oplus(\text{device} = \text{switch-4}\,;\, \text{vtp mode} = \text{client}) \tag{2}$$

In other words, the only possible action is the definition of `switch-4` as a client.

Formal VTP Self-Rule 3 is also violated. This rule imposes that all switches be in the same domain. The application of Plan on this rule produces this action:

$$\oplus(\text{device} = \text{switch-4}\,;\, \text{vtp domain} = \text{accounting}) \tag{3}$$

Finally, the actions produced by Formal VTP Self-Rule 4 are the following:

$$\begin{aligned}
&\oplus(\text{device} = \text{switch-3}\,;\, \text{interface} = \text{fe06}) \\
&\ \wedge \oplus(\text{device} = \text{switch-4}\,;\, \text{interface} = \text{fe04}) \\
&\ \wedge \oplus(\text{device} = \text{switch-3}\,,\, \text{interface} = \text{fe06}\,;\, \text{switchport mode} = \text{trunk}) \\
&\ \wedge \oplus(\text{device} = \text{switch-4}\,,\, \text{interface} = \text{fe04}\,;\, \text{switchport mode} = \text{trunk}) \\
&\ \wedge (\\
&\ (\oplus(\text{device} = \text{switch-3}\,,\, \text{interface} = \text{fe06}\,;\, \text{switchport encapsulation} = \text{dot1q}) \wedge \\
&\quad \oplus(\text{device} = \text{switch-4}\,,\, \text{interface} = \text{fe04}\,;\, \text{switchport encapsulation} = \text{dot1q})) \vee \\
&\ (\oplus(\text{device} = \text{switch-3}\,,\, \text{interface} = \text{fe06}\,;\, \text{switchport encapsulation} = \text{isl}) \wedge \\
&\quad \oplus(\text{device} = \text{switch-4}\,,\, \text{interface} = \text{fe04}\,;\, \text{switchport encapsulation} = \text{isl})) \\
&\ )
\end{aligned} \tag{4}$$

This last set of actions is interesting, since it imposes addition of nodes not only under the newly connected `switch-4`, but also under `switch-3` which is already part of the working VLAN. The reason is that connecting `switch-3` to `switch-4` involves setting up a trunk, an operation that requires configuration modifications on both ends of the wire.

Therefore, the actual course of actions will depend on the architecture chosen: if the self-configuration module depicted in Figure 3 is decentralised in every switch, then each switch must ignore the actions that modify other devices, assuming that the concerned devices will take proper measures to correct the situation on their side. If the self-configuration module is rather centralised for some number of devices, the changes can be sent by the device communication manager to multiple switches at once.

The global plan that must be executed to make the rules true is the conjunection of equations (1)-(4).

5.2 Generating the Configuration

It can be seen that the previous plan is nondeterministic. For example, if a formula $\langle \mathrm{p} = x \rangle\, \varphi \vee \psi$ is false, there are two different ways of making it true: either by making φ true, or by making ψ true. In such a case, both solutions are explored and propagated up the recursion stack. This happens with the actions produced by the Formal VTP Self-Rule 1 that either define the switch as client or as server. In the same way, the actions produced by the Formal VTP Self-Rule 4 rule impose that the trunk encapsulation in `switch-3` and `switch-4` be either both dot1q or both isl, but since none of them is already configured and no further constraint applies, the choice is free.

Moreover, a separate plan has been generated for each violated self-rule; however, it is well possible that these plans are mutually contradictory and that no global solution exists. Finally, one must make sure that executing the plans not only makes the violated rules true, but in turn does not produce side effects that could falsify other rules that were previously true.

To this end, two further steps are taken before committing any configuration to the device. First, a satisfying assignment of the global plan must be found. This can be easily obtained by applying a standard Boolean satisfiability (SAT) solver such as zChaff [17] or SATO [24]. This assignment designates the actions that are actually chosen from the many alternatives suggested by the plan to make the formulas true.

Once this assignment has been found, the plan is tentatively executed on the configuration, and the whole set of self-rules is then revalidated against this modified configuration. If one of the rules is violated, the satisfying assignment is discarded and the procedure backtracks to find a new assignment. This is the case if the following operations are chosen as the action to execute on the configuration (for clarity, the actions on `switch-3` have been omitted):

$$\oplus(\text{device} = \text{switch-4}\,;\ \text{vtp mode} = \text{server})$$

$$\oplus(\text{device} = \text{switch-4}\,;\ \text{vtp mode} = \text{client})$$
$$\oplus(\text{device} = \text{switch-4}\,;\ \text{vtp domain} = \text{accounting})$$
$$\oplus(\text{device} = \text{switch-4}\,;\ \text{interface} = \text{fe04})$$
$$\oplus(\text{device} = \text{switch-4}\,,\ \text{interface} = \text{fe04}\,;\ \text{switchport mode} = \text{trunk})$$
$$\oplus(\text{device} = \text{switch-4}\,,\ \text{interface} = \text{fe04}\,;\ \text{switchport encapsulation} = \text{dot1q})$$

The configuration resulting of the previous operations violates Formal VTP Self-Rule 2, since it assigns `switch-4` both to the role of client and server at the same time. It violates Formal VTP Self-Rule 2 for another reason, as a server already exists in the network.

However, a second possible truth assignment to the global plan is the following:

$$\oplus(\text{device} = \text{switch-4}\,;\ \text{vtp mode} = \text{client})$$
$$\oplus(\text{device} = \text{switch-4}\,;\ \text{vtp domain} = \text{accounting})$$
$$\oplus(\text{device} = \text{switch-4}\,;\ \text{interface} = \text{fe04})$$
$$\oplus(\text{device} = \text{switch-4}\,,\ \text{interface} = \text{fe04}\,;\ \text{switchport mode} = \text{trunk})$$
$$\oplus(\text{device} = \text{switch-4}\,,\ \text{interface} = \text{fe04}\,;\ \text{switchport encapsulation} = \text{dot1q})$$

The configuration resulting of these operations validates all formal VTP self-rules and is therefore admissible. In such a case, the tentative configuration is committed as the running configuration of the switch. The resulting configuration tree obtained from these autonomic configuration steps is shown in Figure 5.

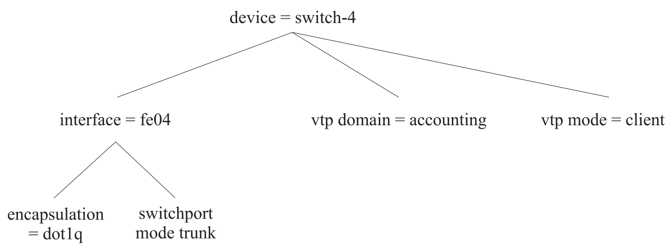

Fig. 5. The configuration of `switch-4` after automatic configuration generation.

One can remark that this method can also work in a self-healing context where a working network is disturbed; the recursive plan generation method provided here ensures that the configurations of the devices can be but back to a state where they fulfill all the rules that must apply on the network.

	Validation time per device (ms)			
Switches	Rule 1	Rule 2	Rule 3	Rule 4
10	0.024	0.012	0.106	0.564
20	0.029	0.012	0.312	1.715
40	0.044	0.024	1.011	6.482
80	0.078	0.041	3.655	21.948

Table 2. Validation time of each formal VTP self-rule in a network formed of a varying number of switches

5.3 Complexity of the Method and Experimental Results

We now evaluate the global complexity of the method. Although Boolean satisfiability is an NP-complete problem, it is not the bottleneck of the procedure, as the global Boolean formula that is generated never has more than a few dozens, possibly a hundred, configuration operations. Therefore, the size of the Boolean instance to be processed is negligible by SAT standards, and can be processed in fractions of a second by SAT solvers accustomed to problems topping the million of variables.

Each configuration operation in itself consists solely in the addition of a single node in the configuration tree of a device and can also be considered instantaneous. Hence, the core of the complexity of this method resides in the validation of CL formulas on multiple incarnations of configurations for many devices.

In order for this method to be tractable, the validation of CL formulas must be quick and simple. It has been shown in [14] that CL model checking of a formula is polynomial in the size of the tree.

We give in Table 2 the validation times for a network composed of 10 to 80 switches. These configurations were generated by a parameterisable script and then loaded into ValidMaker. All results have been obtained on a Pentium IV of 2.4 GHz with 512 Mb of RAM running Windows XP Professional; they are consistent with the polynomiality result already demonstrated.

As one can see from these results, the validation times are reasonable and do not excess 25 milliseconds per device for the largest network of 80 switches. One should compare these figures with the time required for actually injecting a new configuration in a switch and restarting it to make it effective, which is at least a few seconds. Moreover, it should be noted that the validation experiments shown here involve the validation of the whole network, and not only of the few devices that might be concerned when a new switch is connected; this is especially true for VTP Self-Rule 4.

6 Conclusion and Future Work

In this paper, we have shown how to self-configure a network device by validating constraints on the configuration of other devices expressed as trees of parameters and values. We showed how a new switch connected to an VLAN can discover its configuration by selecting a suitable plan that tries to fulfill the existing configuration constraints. We also showed by experimental results with the ValidMaker configuration tool that the validation of self-rules is a tractable operation that imposes negligible time to execute.

At the moment, the SAT instances forming the possible plans are treated separately from the CL validation in an independent solver embedded into the Configuration generator. A natural extension of this work is to enrich this active model by dealing with side effects of the application of a self-rule and support node deletion and modification, for example by integrating CL components into an existing SAT solver in the context of SAT modulo theories [18].

References

1. Configuring VTP. http://www.cisco.com/en/US/products/hw/switches/ps708/products_configuration_guide_chapter09186a008019f048.html.
2. An OSS for packet networks. *Cisco Systems Packet Magazine*, 14(1):25–27, January 2002.
3. 802.11Q: Virtual bridged local area networks standard, 2003. http://standards.ieee.org/getieee802/download/802.1Q-2003.pdf.
4. F. A. Aagesen, C. Anutariya, and V. Wuwongse, editors. *Intelligence in Communication Systems, IFIP International Conference, INTELLCOMM 2004, Bangkok, Thailand, November 23-26, 2004, Proceedings*, volume 3283 of *Lecture Notes in Computer Science*. Springer, 2004.
5. R. Boutaba, S. Omari, and A. P. S. Virk. SELFCON: An architecture for self-configuration of networks. *Journal of Communications and Networks*, 3(4):317–323, December 2001.
6. C. Calcagno, L. Cardelli, and A. D. Gordon. Deciding validity in a spatial logic for trees. In *TLDI*, pages 62–73, 2003.
7. A. L. Couch and M. Gilfix. It's elementary, dear Watson: Applying logic programming to convergent system management processes. In *LISA*, pages 123–138. USENIX, 1999.
8. R. Deca, O. Cherkaoui, and D. Puche. A validation solution for network configuration. In *CNSR*, 2004.
9. R. Enns. Netconf configuration protocol, IETF internet draft, February 2006.
10. S. A. Fakhouri, G. S. Goldszmidt, M. H. Kalantar, J. A. Pershing, and I. Gupta. Gulfstream - a system for dynamic topology management in multi-domain server farms. In *CLUSTER*, pages 55–62. IEEE Computer Society, 2001.
11. D. Gaïti, G. Pujolle, M. Salaün, and H. Zimmermann. Autonomous network equipments. In I. Stavrakakis and M. Smirnov, editors, *WAC*, volume 3854 of *Lecture Notes in Computer Science*, pages 177–185. Springer, 2005.
12. W. Golab and R. Boutaba. Optical network reconfiguration using automated regression-based parameter value selection. In *ICN*, 2004.
13. S. Hallé, R. Deca, O. Cherkaoui, and R. Villemaire. Automated validation of service configuration on network devices. In J. B. Vicente and D. Hutchison, editors, *MMNS*, volume 3271 of *Lecture Notes in Computer Science*, pages 176–188. Springer, 2004.
14. S. Hallé, R. Villemaire, and O. Cherkaoui. CTL model checking for labelled tree queries. In *TIME*, pages 27–35. IEEE Computer Society, 2006.

15. S. Hariri, L. Xue, H. Chen, M. Zhang, S. Pavuluri, and S. Rao. Autonomia: An autonomic computing environment. In *IPCCC*, April 2003.
16. R. M. Keller. *Self-Configuring Services for Extensible Networks – A Routing-Integrated Approach*. PhD thesis, Swiss Federal Institute of Technology, 2004.
17. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering and efficient SAT solver. In *Proceedings of the 39th Design Automation Conference*, June 2001.
18. R. Nieuwenhuis and A. Oliveras. Decision procedures for SAT, SAT modulo theories and beyond. the BarcelogicTools. In G. Sutcliffe and A. Voronkov, editors, *LPAR*, volume 3835 of *Lecture Notes in Computer Science*, pages 23–46. Springer, 2005.
19. M. Parashar and S. Hariri. Autonomic computing: An overview. In J.-P. Banâtre, P. Fradet, J.-L. Giavitto, and O. Michel, editors, *UPP*, volume 3566 of *Lecture Notes in Computer Science*, pages 257–269. Springer, 2004.
20. G. Pujolle and D. Gaïti. Intelligent routers and smart protocols. In Aagesen et al. [4], pages 16–27.
21. J. Strassner. *Directory Enabled Networks*. Macmillan Technical Publishing, 1999.
22. R. Villemaire, S. Hallé, and O. Cherkaoui. Configuration logic: A multi-site modal logic. In *TIME*, pages 131–137. IEEE Computer Society, 2005.
23. R. Villemaire, S. Hallé, and O. Cherkaoui. A hierarchical logic for network configuration. In *LPAR*, 2005. Short paper proceedings.
24. H. Zhang and M. E. Stickel. Implementing the Davis-Putnam method. *J. Autom. Reasoning*, 24(1/2):277–296, 2000.