UNIVERSITÉ DU QUÉBEC À MONTRÉAL

IMPLANTATION D'UNE LOGIQUE DE CONFIGURATION POUR LA VÉRIFICATION AUTOMATIQUE DE CONFIGURATIONS D'ÉQUIPEMENTS DE RÉSEAUX

MÉMOIRE

PRÉSENTÉ

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN INFORMATIQUE

PAR

ÉRIC WENAAS

DÉCEMBRE 2006

REMERCIEMENTS

Ce travail de recherche n'aurait pu être possible sans le concours de nombreuses personnes. Je tiens d'abord à remercier mon directeur de recherche, monsieur Roger Villemaire, pour ses pertinents conseils et la rigueur de son analyse.

Je remercie également monsieur Omar Cherkaoui, directeur du Laboratoire de téléinformatique de l'Université du Québec à Montréeal, pour m'avoir accueilli dans son équipe et m'avoir guidé tout au long de ce projet. Son expertise en réseautique et ses commentaires furent des plus précieux.

J'adresse également mes remerciements à messieurs Sylvain Hallé et Rudy Deca, respectivement étudiants au doctorat en mathématique à l'Université du Québec à Montréal et à l'École polytechnique de Montréal. Leurs conseils et leur aide furent très appréciés.

Enfin, je tiens à reconnaître le support financier du Fonds québécois de la recherche sur la nature et les technologies et du Laboratoire de téléinformatique de l'Université du Québec à Montréal.

Table des matières

Liste des tableaux

Table des figures

Résumé

Ce travail montre comment un formalisme logique, la logique de configuration, est intégré au sein d'un outil de gestion de configuration de réseaux, *ValidMaker*. Le principal objectif de ce travail est de démontrer que la logique de configuration est particulièrement bien adaptée à la vérification automatique de configuration de réseaux. À cette fin, nous développons un exemple réel de configuration de réseaux et nous trouvons des règles qui doivent être vérifiées pour qu'une telle configuration soit fonctionnelle. Ensuite, nous expliquons comment nous avons implanté la logique de configuration dans *ValidMaker* et nous illustrons le fonctionnement de notre algorithme de vérification.

Chapitre I

INTRODUCTION

1.1 Objectifs et contribution du travail

La complexité croissante des réseaux posent divers problèmes de configuration aux administrateurs. Le déploiement de services réseaux devient de plus en plus compliqués et le fait de déployer un nouveau service dans un réseau peut avoir comme effet de causer des problèmes d'incompatibilité avec d'autres services. Des outils existent afin d'aider un administrateur réseau dans sa tâche. Notre travail s'intègre dans le cadre de l'un de ces outils : *ValidMaker*. Nous y implantons un formalisme logique permettant de faire des vérifications dans des structures arborescentes. Ce formalisme est la logique de configuration.

Le premier objectif de notre travail est de démontrer de façon empirique que la logique de configuration est particulièrement bien adaptée à la vérification automatique de configuration de réseaux. Nous implantons donc un moteur de validation au sein de *ValidMaker*. Notre moteur de validation utilise la logique de configuration. Le deuxième objectif de notre travail est d'ajouter à *ValidMaker* une plus grande flexibilité quant à l'édition de règles pouvant être vérifiées dans un réseau.

Notre approche sera tout d'abord de présenter les différents éléments que nous allons utiliser soit *ValidMaker* et la logique de configuration. Ensuite, nous développerons un exemple réel de configuration réseaux en utilisant les réseaux locaux virtuels. De cet exemple réel, nous dégagerons des règles de configuration qui devront être vérifiées afin d'obtenir une configuration fonctionnelle.

La contribution de notre de travail consiste en l'implantation de la logique dans *ValidMaker*. Nous expliquerons quelles sont nos structures de données pour représenter une règle de logique de configuration. Nous expliquerons également de façon détaillée le fonctionnement de notre algorithme de vérification. Une trace d'exécution sera également présentée. Nous verrons également comment notre algorithme offre une valeur ajoutée à l'outil *ValidMaker* en permettant à un administrateur réseau d'analyser une vérification en reprenant la vérification à partir d'une partie d'une règle de validation.

1.2 Division du document

Le chapitre 2 présente les arbres de configuration. Les arbres de configuration sont des arbres dont les noeuds correspondent à des couples paramètre-valeur. Ces arbres seront utilisées dans l'outil *ValidMaker* et dans la logique de configuration.

Le chapitre 3 présente l'outil *Validmaker*. Nous y présentons différentes fonctionnalités nécessaires à la compréhension du déploiement de services à l'aide de *ValidMaker*. Nous y présentons également comment fonctionne la vérification du format de règles antérieur à notre travail.

Le chapitre 4 présente la logique de configuration. Nous y présentons la syntaxe de cette logique et montrons comment la notion de prédicat peut y être introduite.

Le chapitre 5 présente les *Virtual Local Area Networks*(VLAN). Notre exemple de configuration est basé sur ce service réseau. Dans ce chapitre, nous expliquerons comment fonctionnent les VLANs. Nous expliquerons également la démarche que nous avons prise afin de représenter les configurations de VLANs dans *ValidMaker* et de trouver des règles de configuration qui doivent être respectées afin de s'assurer que les services de VLANs soient fonctionnels dans un réseau.

Le chapitre 6 présente les structures de données et les algorithmes que nous avons utilisées afin d'implanter la logique de configuration dans *ValidMaker*. Nous présentons également une trace d'exécution afin d'illustrer le fonctionnement de la vérification

Le chapitre 7 est la conclusion, un rappel des principaux points de ce document sera fait et

nous y présenterons les travaux futurs qui pourraient être envisagés suite à notre travail.

Chapitre II

ARBRES DE CONFIGURATION

Nous présentons dans ce chapitre comment la configuration d'un appareil peut être représentée sous forme d'un arbre de configuration. À cette fin, nous commencerons par traiter de la façon que la configuration des appareils réseaux est représentée. Ensuite, nous expliquerons en quoi consiste un arbre de configuration et nous donnerons un exemple d'arbre de configuration.

2.1 Représentation de la configurations des appareils

La configuration d'appareils réseaux se fait de façon hiérarchique. La hérarchie est représentée par différents modes et sous-modes. Par exemple, pour configurer une interface particulière, il faut d'abord entrer dans le mode de configuration de cette interface. La configuration d'un appareil est généralement stockée sous la forme d'un fichier texte[1]. Ce fichier suit une structure hiérarchique similaire à la structure hiérarchique des modes et des sous-modes. On se retrouve donc avec un arbre de couples paramètre-valeur. Il est possible à partir du fichier de configuration de reconstituer la hiérarchie des couples paramètre-valeur correspondant aux différents paramètres de configuration. Lors du démarrage de l'appareil, la configuration de départ est chargée et le fichier est interprété par l'appareil. L'appareil ajustera ses paramètres en fonction de ce qu'il trouve dans son fichier de configuration.

---

[1] Par exemple, sur un commutateur ou un routeur Cisco ce fichier se nomme *config.txt*.

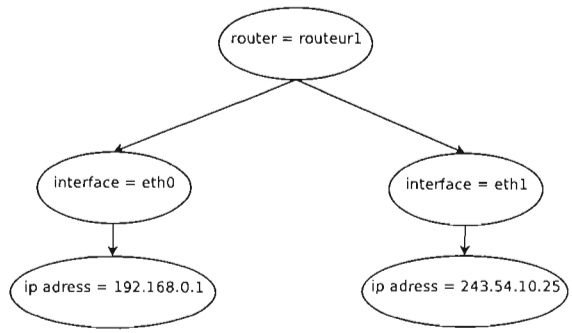

Fig. 2.1 Arbre de configuration

2.2 Arbre de configuration

Puisque les paramètres d'un appareil sont représentés d'une façon hiérarchique, il est possible de représenter la configuration d'un appareil réseau sous la forme d'un arbre où les noeuds correspondent à des couples paramètre-valeur. Nous appelons ce type d'arbre, un arbre de configuration. La configuration globale du réseau est alors une forêt d'arbres de configuration. La racine d'un arbre identifie un appareil. Les descendants de la racine correspondent aux paramètres directement reliés à cet appareil.

La figure 2.1 représente un arbre de configuration simple. Dans cet arbre, on peut voir que les valeurs pour l'adresse IP des interfaces se situent sous les noeuds des interfaces correspondantes. Une caractéristique importante des arbres de configuration est que, étant donné un type de matériel et une version du système d'exploitation donnés, la structure de l'arbre est connue d'avance. On connaît le nombre de paramètres possibles ainsi que leurs positions relatives dans l'arbre. Ce que l'on ne connaît pas sont les paramètres présents ainsi que la valeur associée à chacun des paramètres. L'exercice consistant en la vérification de configuration consistera donc en la vérification de valeurs de certains paramètres dont les positions dans l'arbre de configuration sont prédéterminées.

Chapitre III

VALIDMAKER

ValidMaker est un outil de gestion de configuration de réseaux informatiques qui a été développé au laboratoire de téléinformatique de l'université du Québec à Montréal (5). *ValidMaker* représente la configuration d'un appareil réseau en utilisant la notation Meta-CLI (6). Les configurations d'appareils réseaux sont représentés sous forme d'arbres de configuration. Chaque appareil d'un réseau correspond à un arbre différent.

Puisque nous avons introduit la logique de configuration au sein de l'outil *ValidMaker*, il est impératif de faire une présentation de cet outil avant d'aborder comment nous y avons introduit la logique. Afin de présenter *ValidMaker*, nous traiterons des points qu'il faut comprendre afin d'être capable de bien saisir la démarche d'intégration de la logique de configuration au sein de cet outil. Notre présentation de *ValidMaker* comporte trois parties.

En premier lieu, nous présenterons certaines des fonctionnalités de *ValidMaker*. Ensuite, nous traiterons de la façon dont la création et le déploiement de services réseaux se font dans *ValidMaker*. En dernier lieu, nous présenterons comment fonctionnait l'édition et la vérification de règle dans *ValidMaker* préalablement à notre travail d'intégration de la logique de configuration.

3.1 Principales fonctions de *ValidMaker*

Dans cette section nous traitons des principales fonctions de *ValidMaker*. Nous présenterons comment ValidMaker représente la configuration d'un matériel et comment il peut générer

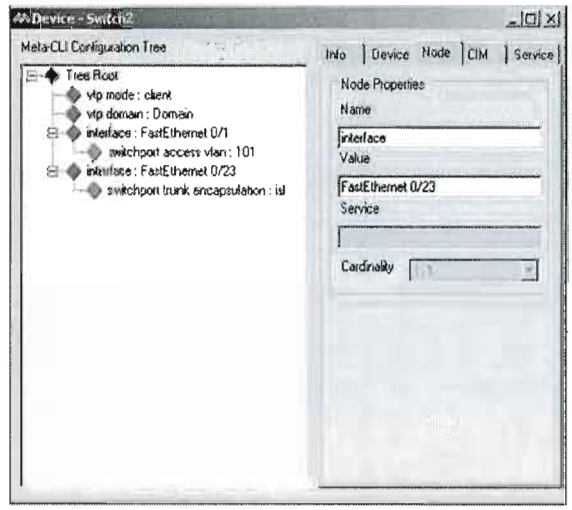

Fig. 3.1 Représentation d'un appareil dans ValidMaker

une configuration de matériel. Certes, *ValidMaker* comprend d'autres fonctionnalités[1] mais la compréhension de celles que nous présentons ici sont suffisantes à la compréhension de notre démarche.

3.1.1 Représentation graphique de la configuration d'un matériel

Une importante faculté de *ValidMaker* est d'être capable de construire un arbre de configuration à partir d'un fichier de configuration et ainsi d'obtenir une représentation graphique de la configuration d'un appareil. À cet égard, *ValidMaker* est un outil flexible car il permet de représenter la configuration de plusieurs types d'appareil. Pour ce faire, il suffit d'ajouter les paramètres de configuration du matériel en question. C'est ainsi par exemple, qu'en un tour de main nous avons pu y ajouter la reconnaissance des paramètres de configuration des VLANs. *ValidMaker* n'est donc pas dépendant d'un type de matériel ou encore d'un système d'exploitation particuliers. La figure 3.1 montre un arbre de configuration tel que représenté au sein de *ValidMaker*.

Cette fonctionnalité permet de voir facilement les configurations de tous les appareils d'un

[1] voir à ce titre (5)

réseau. Les configurations peuvent être obtenues avec un protocole comme NetConf (8) par exemple.

3.1.2 Génération de la configuration

S'il est possible de représenter graphiquement une configuration au sein de *ValidMaker*, il est également possible de générer une configuration fonctionnelle à partir de la représentation graphique d'un arbre. Puisque *ValidMaker* permet de modifier la représentation graphique d'une configuration, les modifications peuvent par la suite se refléter dans un fichier de configuration fonctionnel. Ceci permet à un ingénieur réseau d'utiliser *ValidMaker* pour charger les configurations des appareils du réseau, modifier ces configurations à l'intérieur de *ValidMaker* et ensuite générer de nouvelles configurations et les remettre sur les équipements du réseau.

3.2 Organisation conceptuelle des services réseaux dans *ValidMaker*

Un réseau dans *ValidMaker* comprend les appareils du réseau, les services rattachés au réseau et les règles devant être appliquées dans le réseau. Les configurations d'appareils du réseau sont modifiées par le déploiement et le retrait de services sur un appareil. Nous expliquerons dans cette section comment créer et déployer des services réseaux dans *ValidMaker*.

3.2.1 Étapes de création et déploiement de services réseaux dans *ValidMaker*

Afin de déployer un service sur un appareil dans *ValidMaker*, trois étapes sont nécessaires. Il faut tout d'abord définir un service générique. Ensuite, ce service générique doit être instantié. Et finalement, le service instantié sera déployé sur un appareil.

Une fois que le service est déployé, l'ingénieur réseau peut faire générer les fichiers configuration et modifier les configurations des appareils concernés. Nous expliquons dans cette section les distinctions entre services génériques, services instantiés et déploiements de services.

3.2.2 Services génériques

Tous les services que l'on veut déployer dans *ValidMaker* doivent tout d'abord être des instantiations d'un service générique. Un service générique est une représentation abstraite d'un service. Il contient les paramètres qui doivent être ajoutés à la configuration mais sans contenir de valeur pour ces paramètres. Le service est donc décrit mais sans tenir compte du cadre dans lequel il sera déployé. La représentation sous forme d'arbre de configuration d'un service générique va comprendre les noeuds que l'on doit ajouter dans la configuration d'un matériel où le service va être déployé. Toutefois, ces noeuds ne contiendront que des paramètres et n'auront pas encore de valeur. On retrouve donc dans la description du service générique le squelette complet des noeuds qui devront apparaître dans la configuration d'un appareil pour que le service soit déployé correctement.

Pour des fins de regroupement logique, il est possible de regrouper les services en familles de services. Une famille de service regroupera tous les services génériques qui ont un lien entre eux.

3.2.3 Instantiation d'un services génériques

Un service générique n'est pas fonctionnel. L'instantiation d'un service générique consiste en la création d'un service plus spécifique en spécifiant certaines valeurs du service générique. Les valeurs spécifiées sont celles qui ne seront pas dépendantes de l'appareil sur lequel le service sera déployé. Il est donc possible qu'une instantiation d'un service ne permette pas de donner des valeurs à tous les paramètres du service. On laissera en blanc les noeuds qui sont spécifiques au matériel sur lequel le service sera déployé. Lors de l'instantiation du service générique, on spécifie les valeurs des paramètres qui qualifieront le service mais sans pour autant spécifier des valeurs qui varieront selon les décisions qui seront prises lors du choix de l'équipement sur lequel le service sera déployé.

3.2.4 Déploiement d'un service

Lorsqu'un service aura été instantié, il sera possible de le déployer. Un service doit toujours être déployé sur un appareil et ce n'est qu'avec le déploiement d'un service que la configu-

ration d'un appareil pourra être modifiée. Il est possible que des valeurs supplémentaires doivent être spécifiées lors du déploiement du service. Ce qui est le cas des paramètres se rattachant directement à l'appareil. Un bon exemple se présente lorsque le service doit être rattaché à une interface de l'appareil. Il est alors évident que la valeur affectée au paramètre interface du service ne peut être déterminée que lors du déploiement du service.

3.3 Format antérieur des règles de *ValidMaker*

Une fois que des services ont été déployés et avant de faire générer des nouvelles configurations, l'ingénieur réseau peut vérifier que l'ajout des nouveaux services n'a pas introduit de problème dans le réseau. Pour ce faire, des règles peuvent être appliquées dans le réseau. Les règles permettront d'éviter certaines erreurs qui auraient pu être commises dans le cadre du déploiement des services. Les règles peuvent aussi être utlisées afin de détecter des erreurs dans une configuration de réseau existante. Les règles doivent être définies par l'ingénieur réseau.

Préalablement à notre travail, il existait déjà un format pour définir des règles et un moteur pour vérifier ces règles dans *ValidMaker*. Nous faisons ici un survol du fonctionnement de la création et de la vérification de ce format de règles. Nous en donnerons également quelques avantages et inconvénients. Nous conclurons sur ce sujet en présentant les objectifs que nous poursuivons en introduisant un nouveau format de règles dans *ValidMaker*.

3.3.1 Description de l'ancien format de règles

Les règles de l'ancien format sont principalement basées sur la logique propositionnelle. Les seules opérations logiques possibles sont l'implication logique, la conjonction et la disjonction. Les propositions de base sont des opérations de haut niveau permettant de faire diverses comparaisons entre des noeuds ou entre des branches d'arbres. Ces opérations sont alors combinées avec des conjonctions ou des disjonctions à l'intérieur d'une implication logique afin de produire une règle. L'utilisateur ne peut créer de nouvelles opérations, elles sont intégrées dans le logiciel.

Par exemple, si on a quatre noeuds (n1, n2, n3 et n4) issus d'arbres de configurations, il est

possible d'écrire une règle qui vérifie que si les noeuds n1 et n3 ont la même valeur alors n2 et n4 ont également la même valeur, sinon, il y aura égalité des sous-arbres n2 et n4. Cette règle s'exprimerait de la façon suivante :

```
IF
   NodeValueEqual(n1,n3)
THEN
   NodeValueEqual(n2, n4)
ELSE
   BranchDataEqual(n2, n4)
```

Les noeuds font références aux noeuds des services génériques, ce qui fait que les règles sont rattachées aux services génériques. Il est de plus possible de spécifier que des noeuds doivent provenir d'arbres de configurations différents et donc d'appareils différents.

Ce format de règles a comme principal avantage d'être relativement simple. Toutefois, cette simplicité sacrifie l'expressivité des règles. Nous voyons comme principales limitations les éléments suivants :

- impossibilité d'imbriquer deux implications logiques

- nécessité d'utiliser les relations prédéfinies au sein de *ValidMaker*

- obligation d'intégrer les règles dans le squelette prédéfini d'implication logique

- absence de variables pour conserver des valeurs de noeuds

3.3.2 Objectifs recherchés par l'introduction de la logique de configuration

L'objectif de l'intégration de la logique de configuration est principalement d'offrir au sein de *ValidMaker* une sémantique de règles plus flexible permettant d'exprimer des relations plus complexes entre les différents paramètres. Cet ajout devrait se faire sans avoir à modifier les structures internes de *ValidMaker*. Bien que la composition de règles dans le format de

la logique de configuration soit plus complexe que dans l'ancien format, l'utilisation que nous ferons des prédicats permettra de simplifier grandement la tâche d'écrire des règles complexes.

Chapitre IV

PRÉSENTATION DE LA LOGIQUE DE CONFIGURATION

Dans ce chapitre, nous présentons les fondements de la logique de configuration. Ce formalisme logique a été développé dans (2). Afin de faire cette présentation, nous nous baserons sur un exemple simple pouvant s'appliquer sur l'arbre d'expression que l'on retrouve à la figure 2.1. La présentation de la logique se fera en trois temps. Nous commencerons par présenter les éléments de syntaxe particuliers de la logique. Nous présenterons ensuite le fonctionnement des quantificateurs et en dernier lieu, nous présenterons les prédicats.

4.1 Exemple où la logique peut être appliquée

Si nous revenons à la figure 2.1, on peut constater que les deux interfaces ont des adresses différentes. Une règle simple pourrait être de vérifier que deux interfaces sur un même matériel ne partagent pas la même adresse IP. Cette règle pourrait être vérifiée pour tous les appareils dans le réseau. La vérification visuelle de cette règle pour notre exemple se fait simplement. Il suffit de vérifier que les valeurs associées aux paramètres *ip address* ne correspondent pas. Toutefois, comment pourrait-on exprimer cette vérification dans un formalisme logique ?

Il faudrait arriver à dire qu'il n'y a aucun appareil qui partage une adresse IP sur deux interfaces différentes. La logique de configuration nous permet d'exprimer une telle chose. Cette logique est conçue pour exprimer des propriétés dans des arbres de configuration et est donc particulièrement bien adaptée à la vérification de configuration d'appareils réseaux.

4.2 Syntaxe de la logique de configuration

L'unité de base de la logique de configuration est la formule. Une formule peut être soit vraie soit fausse. Une formule peut consister en un quantificateur ou un opérateur binaire retournant un résultat booléen. Les quantificateurs supportés sont le quantificateur universel noté par [ ], ou encore le quantificateur existentiel noté par < >. Les opérateurs booléens sont ceux que l'on retrouve dans la logique traditionnelle soit la conjonction, la disjonction et l'implication logique. Des valeurs peuvent également être comparées avec des opérateurs de comparaisons.

Une formule peut être composée d'autres formules. Il n'y a pas de limites quant au nombre de niveaux d'imbrication des formules. Ceci représente une différence marquée entre la logique de configuration et le format antérieur des règles de *ValidMaker*. Dans ce dernier format, il était impossible d'imbriquer deux implications logiques. Dans la logique de configuration, il n'y aucune limite à cet effet.

4.3 Quantificateurs

Si le traitement des opérateurs binaires est simple, la particularité de la logique de configuration vient de son traitement des quantificateurs. Cette section décrit donc le fonctionnement des quantificateurs. Les quantificateurs, qu'ils soient existentiels ou universels contiennent deux éléments, un chemin et une formule. Le quantificateur universel vérifie que la formule est vraie pour tous les noeuds qui peuvent quantifier le chemin alors que le quantificateur existentiel retourne vrai dès qu'il existe un noeud quantifiant le chemin qui vérifie la formule.

4.3.1 Description du chemin et quantification des variables

Le chemin est composé d'une ou plusieurs étapes. Il représente une suite de noeuds consécutifs dans un arbre de configuration où, à chaque nouvelle étape, on descend dans la hiérarchie de l'arbre. Deux étapes voisines représentent donc deux noeuds voisins dans la même branche. Puisque la structure des arbres de configuration est connue d'avance, on peut donc avec un chemin identifier directement un type de noeud dans un arbre de configuration.

Chaque étape du chemin est représentée comme un couple paramètre-valeur. Le paramètre correspond au paramètre que l'on recherche et la valeur peut être soit une variable quantifiée ou une variable non quantifiée. Chaque étape du chemin aura sa propre variable. S'il existe des variables quantifiées dans le chemin alors celles-ci seront avant le point-virgule. Si une variable n'est pas quantifiée (donc après le point-virgule) alors le quantificateur s'applique à cette variable. Dans un chemin qui ne contient qu'une racine qui doit être quantifiée le point-virgule est omis.

Par exemple, si nous avons la formule suivante : < p=x > x = 0. Cette formule signifie qu'il existe une racine contenant lc paramètre p dans l'ensemble des configurations et que la valeur de ce paramètre est égal à 0. La formule [p=x] x = 0 signifie que toutes les racines qui ont comme paramètre p ont 0 comme valeur.

Il est possible d'imbriquer des quantificateurs afin de décrire des chemins de plus grandes longueurs. Par exemple, cette formule [p=x] < p=x ; q=y > x = y signifie que pour toute racine ayant pour paramètre p, il existe un noeud sous cette racine ayant comme paramètre q et la valeur associée au paramètre p est égale à la valeur associée au paramètre q.

De plus, la logique de configuration permet d'avoir deux quantificateurs successifs au même niveau. Par exemple, la formule [p=x] < q=y > x=y exprime que pour toute racine ayant pour paramètre p, il existe une racine ayant comme paramètre q et que les valeurs associées à p et q sont égales. Les deux valeurs sont donc égales mais les paramètres sont différents. Une telle chose ne peut être exprimée avec une logique comme CTL ou LTL (11).

À titre d'autre exemple cette fois relié à la configuration de réseau, regardons la formule suivante :

$< router = r1, interface = i1; ipaddress = a1 > \psi$

Cette formule ne peut apparaître que dans un contexte où *r1* et *i1* ont été quantifiées. Il ne restera donc qu'à quantifier la variable *a1*. Le quantificateur <router = r1, interface = i1 ; ip address = a1> ψ exprime alors qu'il existe un noeud dont le paramètre est *ip address* et ce sous le noeud correspondant à la valeur de la variable *i1* qui est lui-même sous le noeud de la variable *r1*. Dans notre exemple de la figure 2.1, en supposant que *r1* = routeur1 et

que $i1$ = eth0 alors la variable $a1$ prendra la valeur 192.168.0.1.

Les variables d'un quantificateur restent visibles et quantifiées à l'intérieur de la formule associée à ce quantificateur. La variable existera donc et contiendra la même valeur pour toutes les sous-formules de ce quantificateur. Voici par exemple une règle qui vérifie qu'aucun routeur ne possède la même adresse pour plus d'une interface :

[router = r1] [router=r1 ; interface = i1]
[router=r1, interface = i1 ; ip address = a1]
[router=r1 ; interface = i2]
[router=r1, interface = i2 ; ip address = a2] ($i1 \neq i2 \rightarrow a1 \neq a2$)

4.3.2 Sémantique récursive

Dans cette section nous présentons la sémantique récursive des formules de la logique de configuration.

Une valuation comprend des valeurs pour les variables d'une formule. Une valuation est notée par le symbole ρ. Une configuration est notée par le symbole $\mathcal{C}$. Dans un chemin l'ensemble des couples déjà quantifiés sera noté par $\bar{p} = \bar{x}$. Une valuation notée $\rho[x/v]$ représente une valuation qui correspond en tout points à ρ sauf pour la variable x qui aura alors la valeur v. Cette notation permet de représenter le cas où on ajoute une variable à la valuation ρ.

On notera donc qu'une configuration $\mathcal{C}$ et une valuation ρ satisfont une formule de logique de configuration φ de la façon suivante : $\mathcal{C}, \rho \models \varphi$. Ceci est vrai si récursivement on a que :

1) $\mathcal{C}, \rho \models \mathcal{R}(\rho)$ si la relation $\mathcal{R}$ est vraie avec la valuation ρ.
2) $\mathcal{C}, \rho \models \varphi \wedge \psi$ si on a $\mathcal{C}, \rho \models \varphi$ et $\mathcal{C}, \rho \models \psi$
3) $\mathcal{C}, \rho \models \varphi \vee \psi$ si on a $\mathcal{C}, \rho \models \varphi$ ou $\mathcal{C}, \rho \models \psi$
4) $\mathcal{C}, \rho \models \neg\varphi$ si on a $\mathcal{C}, \rho \not\models \varphi$ C'est à dire que $\mathcal{C}, \rho \models \varphi$ est faux
5) $\mathcal{C}, \rho \models < \bar{p} = \bar{x}; p = x > \varphi$ s'il existe une valeur v pour le paramètre p sous le chemin représenté par $\bar{p} = \bar{x}$ tel que $\mathcal{C}, \rho[x/v] \models \varphi$.
6) $\mathcal{C}, \rho \models [\bar{p} = \bar{x}; p = x]\varphi$ si pour toutes les valeurs que v peut prendre pour le paramètre p sous le chemin représenté par $\bar{p} = \bar{x}$ on a que $\mathcal{C}, \rho[x/v] \models \varphi$.

4.4 Prédicats

Un prédicat représente une sous-formule et est une formule en lui-même. Un prédicat devrait porter un nom significatif qui permet d'identifier sa signification. Les prédicats ont été introduits pour simplifier la compréhension et la conception des règles exprimées en logique de configuration. L'utilisation de prédicats en logique de configuration s'apparente à la décomposition fonctionnelle en programmation. Les prédicats ne permettent pas d'exprimer des choses que l'on ne pouvait pas déjà exprimer en logique de configuration mais ils permettent de simplifier la lisibilité des règles en donnant des noms représentatifs à des sous-formules.

Un prédicat comprend des paramètres. Ces paramètres représentent des noeuds d'un arbre de configuration. On peut par exemple, définir un prédicat qui retourne vrai si un routeur ne possède aucune interface qui partage la même adresse IP. Ce prédicat prend en paramètre un noeud représentant un appareil. Voici ce prédicat et un exemple de son utilisation qui permet de simplifier la règle que nous avons précédemment énoncée.

Définition d'un prédicat :

DifferentAddressesForEachInterface(R) :-
[R ; interface = i1]
[R, interface = i1 ; ip address = a1]
[R ; interface = i2]
[R, interface = i2 ; ip address = a2] ($i1 \neq i2 \rightarrow a1 \neq a2$)

Utilisation du prédicat afin de simplifier l'expression d'une règle :

```
[router = r1]
DifferentAddressesForEachInterface(r1)
```

Les paramètres d'un prédicat doivent être quantifiés préalablement à l'appel. Nous voyons dans la description du prédicat comment un noeud passé en paramètre peut être utilisé pour

quantifier des descendants éventuels de ce noeud. Bien que le formalisme de la logique de configuration soit plus complexe que le format antérieur des règles de *ValidMaker*, l'introduction des prédicats simplifie beaucoup la définition des règles en logique de configuration.

Chapitre V

RÈGLES DE CONFIGURATION ASSOCIÉES AUX RÉSEAUX LOCAUX VIRTUELS STATIQUES ET AU PROTOCOLE VTP

L'objectif de ce chapitre est de présenter la démarche que nous avons suivie afin de développer un exemple réel de configuration de réseau nous permettant d'illustrer notre mise en oeuvre de la logique de configuration. Nous avons choisi de travailler sur les VLANs configurés à l'aide du protocole VTP[1]. Ce chapitre commencera donc par expliquer en quoi consiste et comment sont configurés les VLANs et VTP. Ensuite, nous ferons un exposé des expérimentations que nous avons faites et des règles de configuration que nous en avons dégagées pour qu'une configuration soit fonctionnelle. En dernier lieu, nous expliquons comment nous avons généré les configurations à l'aide de l'outil *ValidMaker*.

5.1 Présentation des VLANs et du protocole VTP

5.1.1 Définition et fonction des VLANs

Avant de présenter les VLANs, il faut tout d'abord présenter brièvement en quoi consiste un LAN[2]. C'est un réseau regroupant un ensemble de stations reliées par des concentrateurs et/ou des commutateurs. La communication entre les stations d'un même LAN se fait au niveau de la couche liaison de données (le deuxième niveau du modèle OSI (13)). Pour

---

[1] *Virtual Trunking Protocol*

[2] Réseau local ou *Local Area Network* en anglais

communiquer à ce niveau, l'émetteur diffuse un message dans le LAN. Ce message contient l'adresse physique du récepteur. C'est ainsi que ce dernier identifie que le message lui est destiné.

Avec les LANs, il est possible de regrouper les postes d'une organisation en divers sous-réseaux reliés entre eux par des routeurs. Toutefois, une limitation importante des LANs est que le regroupement logique des stations doit correspondre au regroupement physique. Ainsi, par exemple, si une organisation désire regrouper des postes ensemble, elle devra s'assurer qu'ils soient tous reliés aux mêmes appareils. De plus, si plusieurs stations sont reliées à un commutateur, elles doivent nécessairement faire partie du même LAN puisqu'un commutateur ne déchiffrera pas l'adresse réseau de la station émettrice mais se contentera de diffuser le message sur chaque interface active.

Les VLANs permettent de contourner cette limitation. Ils font appel à l'habileté des commutateurs et des routeurs à configurer des topologies logiques au-dessus de l'architecture physique du réseau. Il est ainsi possible de réunir dans le même réseau local des stations qui auraient fait partie de deux réseaux locaux différents car elles ne pourraient être connectées sur le même appareil. Les VLANs permettent également de créer plus d'un sous réseau sur un même commutateur.

Les VLANs permettent donc de garder l'avantage des LANs tout en ne se limitant pas aux contraintes physiques que ces derniers imposent. L'association entre un poste et le VLAN auquel ce poste appartient peut se faire de diverses façons. La première de ces façons est celle que nous avons développée dans le cadre de notre projet. Il s'agit d'associer une interface d'un commutateur avec un VLAN particulier. Ainsi, lorsqu'un poste est connecté à une interface, il fera partie du VLAN associé à cette interface. Il s'agit alors d'un VLAN statique. Il existe d'autres types de VLANs. Par exemple, l'appartenance à un VLAN peut être déterminée par l'adresse physique du poste ou encore par son adresse réseau.

5.1.2 Configuration des VLANs

Puisque l'outil *ValidMaker* que nous utilisons est un outil travaillant directement sur les

configurations d'appareils et non sur les commandes, nous traiterons ici de la façon dont les VLANs sont représentés dans la configuration des commutateurs. Puisque nous avons utilisé des routeurs Cisco, notre exemple est construit sur ce type de routeur. La flexibilité de *ValidMaker* permettrait facilement de construire un nouvel exemple sur un autre type de matériel.

Sur un commutateur Cisco, la configuration des VLANs est comprise dans deux fichiers. Le premier de ces fichiers est le fichier de configuration normal, soit *config.txt* qui est un fichier texte. L'autre fichier est un fichier binaire, *vlan.dat*. Le contenu du fichier *vlan.dat* devrait éventuellement être intégré dans le fichier *config.txt*[3]. Toutefois, tout dépendant du type de matériel utilisé, il est possible que le fichier de configuration contienne également les informations de *vlan.dat*. Pour développer nos exemples, nous considérons que l'information est disponible dans le fichier de configuration[4].

Lorsqu'un port est associé à un VLAN et qu'un poste y est connecté, le poste fait partie du VLAN. Le VLAN est identifié par un nombre de 2 à 1000[5] et de façon optionnelle par un nom. Le VLAN doit également exister dans la liste des VLANs créés sur le commutateur. Dans la configuration, l'association d'une interface à un VLAN est notée de la façon suivante :

```
interface [identification de l'interface]
        switchport mode access
        switchport access vlan [identification du vlan]
```

La création d'un VLAN sur un commutateur se reflète dans la configuration de la façon suivante :

---

[3]Voir à ce titre (1) à la page 325.

[4]Veuillez noter que le matériel qui a été utilisé ne comprenait pas toutes les informations sur les VLANs dans le fichier *config.txt*. Nous avons donc écrit un script permettant de générer la liste de commandes à partir de la configuration générée par *ValidMaker*. L'exécution de ce script nous a donné les résultats escomptés.

[5]Il existe d'autres identificateurs pour les VLANs. Le VLAN 1 est le VLAN par défaut et est réservé. Les VLAN 1001 à 1005 sont également réservés. Il existe également des VLANs étendus.

```
vlan [identification du vlan]
```

Lorsque qu'un poste associé à un VLAN désire communiquer avec un autre poste associé au même VLAN, il doit diffuser son message. C'est ce qui est appelé la communication intra-VLAN. Le message n'a pas à être routé, puisque c'est une communication de niveau deux. Il aura besoin d'être routé seulement si un poste désire communiquer avec un autre poste qui n'appartient pas au même VLAN. Dans le cadre de la communication intra-VLAN, les commutateurs ont la responsabilité de diffuser le message sur toutes les interfaces associées au VLAN.

Lorsque deux commutateurs partageant des VLANs sont connectés ensemble, alors ils doivent être connectés par un faisceau de jonction[6]. Un faisceau de jonction est destiné à faire circuler le trafic des VLANs partagés par les commutateurs. C'est grâce aux faisceaux de jonction que la communication intra-VLAN fonctionne entre deux commutateurs. Lorsqu'un message est diffusé sur un VLAN, le commutateur le diffuse donc non seulement sur toutes ses interfaces qui sont associées à ce VLAN mais également sur tous les faisceaux de jonction chargés de faire circuler le trafic de ce VLAN. Les faisceaux de jonction doivent également être encapsulés dans un protocole. Les principaux protocoles d'encapsulation sont *dot1q* et *isl*.

Par défaut, les faisceaux de jonction peuvent transporter les informations de tous les VLANs créés sur le commutateur. Ce paramètre peut également être modifié. Lorsqu'une interface est configurée comme faisceau de jonction, on retrouve les éléments suivants dans la configuration de l'appareil :

```
interface [identification de l'interface]
        switchport mode trunk
        switchport trunk encapsulation [protocole d'encapsulation]
```

5.1.3 Fonction de VTP

Le fait que les VLANs doivent être créés sur chaque commutateur pour que la communication

[6] en anglais : *trunk*

intra-VLAN fonctionne et que chaque commutateur ait sa propre liste de VLANs (qui bien souvent doit être la même pour chaque commutateur) a amené le développement de VTP. Ce protocole centralise la création des VLANs dans un seul commutateur. L'information est alors diffusée vers les autres commutateurs qui mettent à jour leurs informations sur les VLANs existant lorsqu'il y a lieu.

Dans VTP, il y a trois types de mode : *server*, *client* et *transparent*. Nous ne nous intéressons qu'aux deux premiers modes[7]. Le principe de VTP est que les VLANs seront créés et détruits sur le serveur exclusivement. Celui-ci enverra des mises à jour périodiques de la liste des VLANs existants. Les clients mettront leur liste de VLANs à jour selon ces informations.

Il existe un autre paramètre dans VTP, soit le domaine. Lorsque VTP est activé sur un appareil, il faut également spécifier un domaine. Les clients ne tiendront compte que des mises à jour envoyées par le serveur de leur domaine. Lorsque VTP est activé sur un appareil, on retrouve les lignes suivantes dans la configuration :

```
vtp mode [un mode]
vtp domain [un nom de domaine]
```

5.2 Définition des services de VLAN dans le cadre de ValidMaker

Tel que nous l'avons vu, pour intégrer un service dans *ValidMaker*, il faut tout d'abord définir un service générique. Par la suite, le service sera instantié. L'instantiation d'un service permet de spécifier certains paramètres qui ne seront pas partagés par toutes les instances du service générique. Un service doit ensuite être déployé sur un matériel. Le déploiement ajoute les paramètres de configuration du service à la configuration du matériel. Lors du déploiement, des paramètres additionnels propres à l'appareil peuvent être spécifiés. Afin de déployer des services de VLANs, nous avons donc créé des services génériques et instantié ces services avant de les déployer. Cette section présente la façon que nous avons retenue pour représenter les VLANs dans ValidMaker.

---

[7]Le mode *transparent* se contente d'ignorer les mises-à-jour envoyées par le serveur sans néanmoins les bloquer. Il ne représente pas d'intérêt pour la dynamique client-serveur du protocole VTP.

Quatre services génériques ont été créés. Un service pour l'activation de VTP, un service pour l'association d'une interface à un VLAN, un service pour la création de VLAN et finalement un service pour la création d'un faisceau de jonction. Pour chacun de ces services, nous expliquerons comment il doit être instantié et déployé.

5.2.1 Activer VTP

Ce service générique comprend les lignes suivantes :

```
vtp mode [valeur]
vtp domain [valeur]
```

L'ajout de ces lignes à une configuration a pour effet d'activer VTP. Au moment de l'instantiation de ce service, les deux valeurs doivent être spécifiés. Il faut donc créer une instance pour le client et une pour le serveur et ce dans chaque domaine. Au moment du déploiement sur un appareil, il n'y a rien à faire.

5.2.2 Associer une interface à un VLAN

Ce service générique comprend les lignes suivantes :

```
interface  [valeur]
    switchport mode access
    switchport access vlan [valeur]
```

Ces lignes vont ajouter à la configuration les paramètres nécessaires pour associer une interface à un VLAN. Nous avons choisi, pour l'instantiation du service de spécifier la valeur pour le VLAN. Ceci nécessite donc la création d'une instance de service pour chaque VLAN. Toutefois, le choix de l'interface étant propre à l'appareil, cette valeur ne sera spécifiée qu'au moment du déploiement sur cet appareil.

5.2.3 Créer un VLAN

Ce service générique ne comprend que la ligne suivante :

```
vlan [valeur]
```

Cette ligne va simplement ajouter à la configuration l'information permettant de créer un VLAN sur le matériel. Au moment de l'instantiation du service, il faut spécifier le VLAN. Au moment du déploiement, il ne reste rien à spécifier.

5.2.4 Créer un faisceau de jonction

Ce service générique comprend les lignes suivantes :

```
interface  [valeur]
    switchport mode trunk
    switchport trunk encapsulation [valeur]
```

Ces lignes ajoutent à la configuration l'information permettant de créer un faisceau de jonction sur une interface. Le fichier de configuration ne permet pas de déterminer à quel appareil cette interface est reliée. Au moment de l'instantiation de ce service, nous avons décidé de spécifier le mode d'encapsulation. La sélection de l'interface est donc reportée au moment du déploiement, comme pour le service d'affectation à un VLAN.

5.3 Expérimentations faites sur les VLANs et règles de configuration en découlant

Dans cette section, nous traiterons des règles de configuration que nous avons dégagées de nos expérimentations. Pour chacune de ces règles, nous expliquerons la nature des expérimentations faites, nous exprimerons ensuite la règle de façon textuelle et finalement nous proposerons une formule en logique de configuration. À titre de référence future, les règles sont simplement numérotées. Pour chacune des expérimentations, les appareils sont tout d'abord

remis dans leur configuration initiale. Le but de ces expérimentations est de montrer que ces problèmes peuvent survenir dans la réalité. Nous générons ici des exemples de problèmes de configuration qu'on ne peut découvrir à partir de l'information venant d'un seul appareil. Les problèmes que nous générons viennent du fait que plusieurs appareils ont des configurations incompatibles. La vérification des règles permettra de détecter ces incompatibilités.

5.3.1 Première règle

Nature de l'expérimentation

Le but de cette première expérience est de voir s'il existe des conditions pour la création d'un VLAN lorsque VTP est activé. Pour ce faire, nous avons testé ce qui arrivait si VTP était activé en mode client sur un commutateur alors qu'il n'y a pas de serveur dans le même domaine VTP. Nous avons activé VTP en mode client sur un commutateur. Deux postes ont été connectés à deux interfaces du commutateur et les postes ont été configurés avec des adresses du même sous-réseau. Le VLAN 100 a été créé sur le commutateur et chaque interface a été affectée au VLAN 100. Il faut noter que lors de la création du VLAN, un avertissement était visible sur la console du commutateur mentionnant que l'information n'avait pas été sauvegardée. Sur un des postes nous avons ensuite tenté de communiquer avec l'autre poste. La communication était impossible.

L'expérimentation a ensuite été reprise avec comme modification que le commutateur a été activé en mode serveur. Maintenant la communication est possible entre les deux postes et nous pouvons consulter l'adresse de l'autre poste avec ARP.

Nous avons donc ajouté un commutateur au réseau et quatre postes ont été utilisés. L'ajout d'un commutateur implique la nécessité de créer un faisceau de jonction entre les deux commutateurs. À partir de cette configuration, trois tests ont été faits. Dans le premier test, les deux commutateurs ont été activés en mode client. La communication est alors impossible entre deux postes du même VLAN. Dans le deuxième test, VTP est activé en mode serveur sur un des commutateurs mais le domaine VTP des deux commutateurs est différent. La communication entre deux postes connectés au serveur est fonctionnelle au

même titre que pour notre première expérimentation mais puisque que le commutateur client n'est pas dans le même domaine que le serveur, il a ignoré les mises à jour du serveur et le VLAN n'existe donc pas sur ce commutateur. Il est donc impossible de communiquer avec des postes connectés sur le commutateur client.

Le troisième test est de mettre les deux commutateurs dans le même domaine. Dans ce cas, tout fonctionne tel que prévu. Nous avons pu communiquer avec tous les postes et ce à partir de n'importe quel poste. Nous pouvons donc en déduire que si VTP est activé, alors il doit y avoir un serveur dans chaque domaine.

Expression de la règle

Nous avons déduit de l'expérience précédente qu'il est impératif pour un commutateur où VTP est activé en mode client qu'il y ait dans le même domaine un commutateur pour lequel VTP est activé en mode serveur. Ceci revient à dire que pour chaque commutateur où VTP y est activé en mode client alors il doit exister un commutateur où VTP est activé en mode serveur. De plus les deux commutateurs doivent être dans le même domaine. En logique de configuration, on peut exprimer la chose de la façon suivante (les prédicats `SwitchInVTPDomain` et `IsVTPServer` seront définis à la page 38) :

```
[device = s1]
(
        IsVTPClient(s1)
->
        <device = s2>
        <device = s1; vtp domain = d1>
        (
                SwitchInVTPDomain(s2, d1)
        AND
                IsVTPServer(s2)
        )
)
```

5.3.2 Deuxième règle

Nature de l'expérimentation

Le but de cette deuxième expérience est de vérifier ce qu'il advient si une interface est affectée à un VLAN inexistant. Cette expérience fait suite à la première expérience puisque nous venons de voir que seul le serveur peut créer un VLAN. Or, le fait qu'il existe un serveur dans le domaine ne garantie pas que le VLAN ait été créé. Ceci laisse donc la possibilité qu'une interface soit associée à un VLAN inexistant, ce qui peut arriver si l'ingénieur réseau commet une erreur sur l'identification d'un VLAN, du domaine VTP ou encore omet de créer le VLAN. Cette erreur peut également se produire si le serveur VTP tombe en panne avant d'envoyer une mise à jour des VLANs.

Nous avons donc tenté d'affecter une interface à un VLAN inexistant. Il en résulte que l'interface est désactivée. Il est donc nécessaire que le VLAN soit créé.

Expression de la règle

Nous pouvons déduire de cette expérience que pour qu'une affectation d'une interface à un VLAN soit effective, il faut que le VLAN existe. Ce qui revient à dire qu'il ait été créé sur un serveur dans le même domaine que le commutateur. Nous allons donc vérifier que pour chaque affectation d'une interface à un VLAN, il existe un serveur VTP dans le même domaine et que ce serveur a créé le VLAN. En logique de configuration on obtient la formule suivante :

```
[device = s1]
[device = s1; interface = i1]
[device = s1, interface = i1; switchport access vlan = v1]
<device = s2>
<device = s2; vtp domain = d2>
(
```

```
        SwitchInVTPDomain(s1, d2)
AND
        VLANCreatedBySwitch(s2, v1)
)
```

Définition des prédicats utilisés

Les précédentes règles utilisent des prédicats. Nous présentons ici la définition de ces prédicats dans la logique de configuration. Nous mentionnons également le but de chaque prédicat.

Le prédicat *SwitchInVTPDomain* est vrai si le commutateur S est dans le domaine VTP D.

```
SwitchInVTPDomain(S, D) :-
   <device = S; vtp domain = d1> D = d1
```

Le prédicat *VLANCreatedBySwitch* est vrai si le VLAN V est créé sur le commutateur S.

```
VLANCreatedBySwitch(S, V) :-
    <device = S; vlan = id> V = id
```

Le prédicat *IsVTPClient* est vrai si le commutateur S est un client VTP.

```
IsVTPClient(S) :-
  <device = S; vtp mode = x> x = client
```

Le prédicat *IsVTPServer* est vrai si le commutateur S est un serveur VTP.

```
IsVTPServer(S) :-
  <device = S; vtp mode = x> x = server
```

5.4 Exemple de configuration

Dans cette section, nous présentons un exemple de configuration que nous avons créé dans ValidMaker. Cette configuration comprend des erreurs et nous verrons que les deux règles que nous venons d'énoncer ne sont pas respectées.

Dans notre exemple, nous avons deux commutateurs, *Switch1* et *Switch2*. Le commutateur *Switch1* sera le serveur VTP alors que *Switch2* le client. Initialement, nous allons mettre les deux commutateurs dans deux domaines VTP différents. Ce qui provoquera une violation de la règle 1. Nous allons également associer une interface du commutateur *Switch2* à un VLAN inexistant (puisqu'il n'aura pas été créé sur le serveur *Switch1*). Ceci fera en sorte qu'une fois que la première erreur sera corrigée, il y aura violation de la deuxième règle. Pour générer les configurations correspondantes nous avons tout d'abord défini différentes instances de services :

VTP Server MyDomain : Ce service permet d'assigner à un commutateur le rôle de serveur dans le domaine *MyDomain*

VTP Client Domain : Ce service permet d'assigner à un commutateur le rôle de client dans le domaine *Domain*

VLAN Creation 100 : Ce service permet de créer le VLAN 100 sur un commutateur (il doit être un serveur)

VLAN Creation 200 : Ce service permet de créer le VLAN 200 sur un commutateur (il doit être un serveur)

VLAN Assignation 101 : Ce service permet d'assigner une interface d'un commutateur au VLAN 101

VLAN Assignation 200 : Ce service permet d'assigner une interface d'un commutateur au VLAN 200

TRUNK Encapsulation isl : Ce service permet de définir une interface en tant que faisceau de jonction et de l'encapsuler dans le mode isl

Par la suite, les services ont été déployés sur chaque commutateur. Nous avons déployés sur le commutateur *Switch1* les services suivants :

VTP Server MyDomain, VLAN Creation 100, VLAN Creation 200, TRUNK Encapsulation isl

Et sur le commutateur *Switch2* nous avons déployé :

VTP Client Domain, VLAN Assignation 101, TRUNK Encapsulation isl

Ce qui donne les configurations suivantes :

```
Pour le commutateur Switch1:

vtp mode server
vtp domain myDomain
vlan 100
vlan 200
interface FastEthernet 0/23
  switchport trunk encapsulation isl


Pour le commutateur Switch2:

vtp mode client
vtp domain Domain
interface FastEthernet 0/1
  switchport access vlan 101
interface FastEthernet 0/23
  switchport trunk encapsulation isl
```

Deux erreurs de configuration ont donc été commises. La première de ces erreurs est que les deux commutateurs sont dans des domaines différents. Pour la réparer, il suffirait de définir un nouveau service *VTP Client MyDomain* qui devrait alors remplacer le service *VTP Client Domain* sur le commutateur *Switch2*. Pour ce faire, le service *VTP Client Domain* serait tout d'abord retiré et ensite on déploierait *VTP Client MyDomain*. Après cette première modification, la règle 1 est vérifiée.

Toutefois, c'est maintenant la règle 2 qui cause un problème car le VLAN 101 n'existe pas. Il faut donc soit créer un VLAN 101 par la création et le déploiement d'un service *VLAN Creation 101* soit créer un service *VLAN Assignation 100* et s'en servir pour remplacer le service *VLAN Assignation 101* qui avait été déployé sur le commutateur *Switch2* et ce sur la même interface.

Après ces deux modifications, les deux règles sont vérifiées. La vérification de la règle 1 avec cet exemple sera reprise dans le prochain chapitre afin d'illustrer le fonctionnement des algorithmes de vérification.

Chapitre VI

STRUCTURES DE DONNÉES ET ALGORITHMES UTILISÉS

L'objectif de ce chapitre est de présenter comment nous avons implanté la logique de configuration au sein de *ValidMaker*. Nous ferons donc, dans un premier temps, une présentation des structures de données que nous avons employées et dans un deuxième temps, nous illustrerons comment fonctionne notre algorithme de validation en relation avec nos structures de données.

6.1 Structures de données

Cette section fait état de la façon dont les règles sont représentées dans *ValidMaker*. Nous faisons un survol des principales classes utilisées dans la construction et dans la résolution des règles. Nous verrons comment sont constituées les règles et les prédicats.

6.1.1 Règles

La classe de base pour les règles est simplement nommée *Rule*. Elle comporte l'interface publique qui nous permet d'intégrer la logique de configuration dans *ValidMaker* de façon relativement transparente. La représentation de la formule de la règle est un arbre d'expression. Nous utilisons une conception orientée objet. Les différents types de noeuds de l'arbre d'expression héritent de la même classe abstraite nommée *BooleanNode*. Chaque type de noeud héritant de *BooleanNode* doit surcharger une fonction nommée *evaluate*. Cette fonction retourne un booléen indiquant si la formule est vérifiée. L'évaluation d'une formule se fait par une succession d'appels récursifs à la fonction *evaluate*. La classe *Rule* contient un

pointeur sur la racine de l'arbre d'expression. Au chargement de la règle, la construction de l'arbre se fait en parcourant de façon récursive les noeuds du fichier XML qui suivent la structure syntaxique de la formule.

6.1.2 Prédicats

Les prédicats peuvent être vus sous deux angles différents. Au niveau des formules, le prédicat est simplement considéré comme un noeud. Toutefois, les prédicats constituent des formules en soi et comportent donc également une racine et une interface permettant de les évaluer. La classe *Predicate* est une classe représentant le prédicat en tant qu'entité autonome.

Le prédicat est construit de la même façon que la règle, soit par le parcours récursif des noeuds du fichier XML qui le décrit. Le comportement d'un prédicat est donc similaire à celui de la formule. La distinction entre les deux vient de la façon qu'ils sont utilisés. Lorsqu'une règle est appelée, une vérification globale est effectuée. Lorsqu'un prédicat est appelé, une vérification se fait simplement dans le cadre des paramètres du prédicat.

Dans les sous-sections suivantes, lorsqu'il sera question des règles il faudra noter que tout ce que nous mentionnerons sera également applicable aux prédicats (sauf si le contraire est expressément mentionné).

6.1.3 Autres classes

Les figures 6.1 et 6.2 représentent la hiérarchie des classes héritant de *BooleanNode*. Nous avons divisé le diagramme de classes en deux parties pour qu'il soit plus facile à visualiser. On peut y voir que les opérations de la logique de configuration sont représentées par une classe héritant de la classe *BooleanNode*. On y retrouve les deux types de quantificateurs, des opérateurs booléens et de comparaison de valeurs ainsi que des constantes représentant les valeurs vrai et faux. Dans cette hiérarchie, la classe *PredicateReference* nécessite quelques explications.

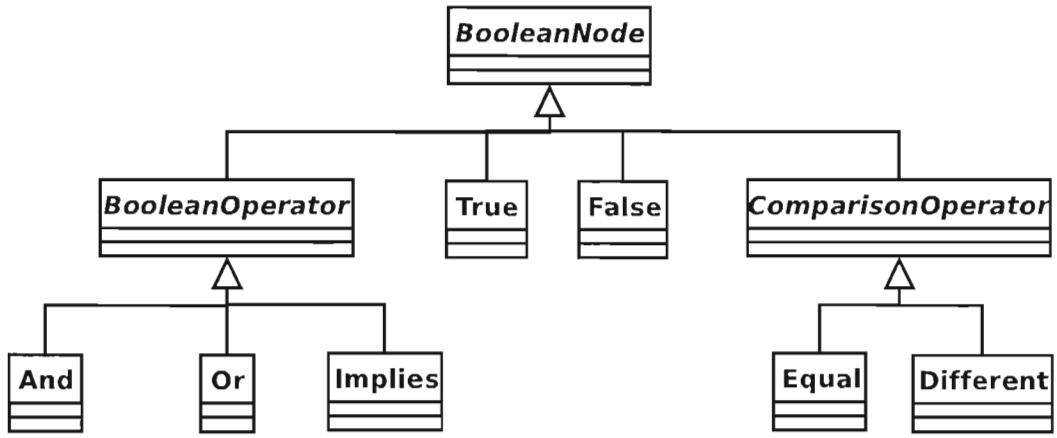

Fig. 6.1 Diagramme de classes arbre expression

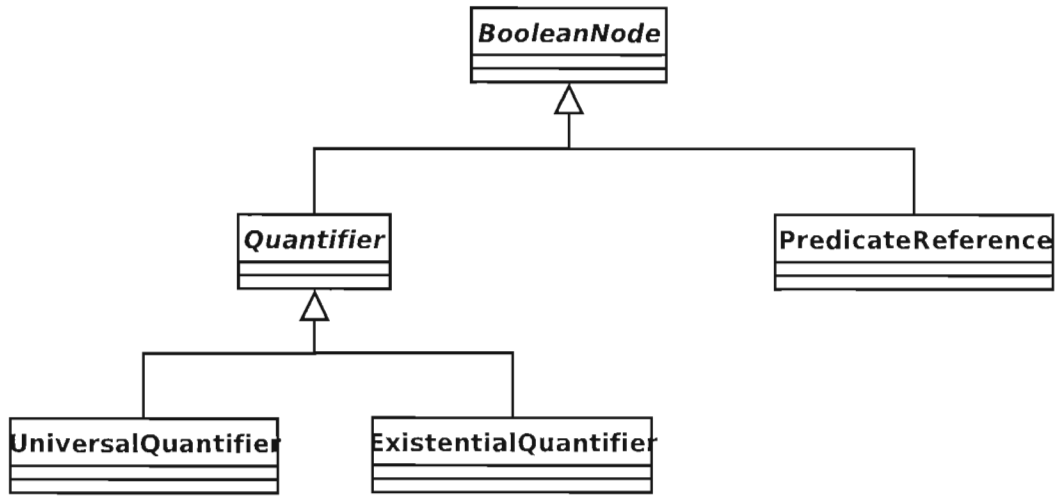

Fig. 6.2 Diagramnme de classe arbre expression(suite)

Référence aux Prédicats

La classe *PredicateReference* est utilisée pour représenter un prédicat. Tel que nous l'avons mentionné, le prédicat se comporte comme une formule en soi. Toutefois, au niveau de la règle (ou d'un autre prédicat si les prédicats sont imbriqués) il est simplement vu comme un noeud. Afin de représenter cette réalité, nous avons décidé de créer une nouvelle classe qui peut-être vue comme une boîte noire encapsulant le prédicat. Ainsi, une formule qui contient une référence à un prédicat ne peut pas voir le contenu du prédicat mais doit se contenter d'appeler la fonction *evaluate* de la référence au prédicat. Cette fonction se chargera alors de démarrer l'évaluation du prédicat.

Les informations que l'on doit donc conserver dans une référence à un prédicat sont le nom du prédicat ainsi que la liste des paramètres effectifs avec lesquels le prédicat doit être appelé. La fonction *evaluate* procédera à la recherche de l'instance de la classe *Predicate* correspondante. Elle procédera également à la recherche des valeurs pour les paramètres du prédicat. Elle retournera ensuite le résultat de l'évaluation du prédicat.

6.2 Trace de déroulement de la vérification

Dans cette section nous présenterons une trace qui nous permettra de montrer comment fonctionne notre algorithme sur la configuration que nous avons présentée en exemple dans le chapitre précédent. Nous présenterons également comment les résultats de cette trace sont montrées dans ValidMaker. Ceci nous permettra d'expliquer comment nous faisons pour montrer à l'utilisateur quelle partie de la règle est fausse ainsi que pour lui permettre de redémarrer la vérification à partir d'un noeud particulier lorsque la vérification a échouée.

L'algorithme de base est une vérification récursive de la formule. Pour un quantificateur existentiel nous retournons vrai dès que la formule associée au quantificateur retourne vrai. Pour un quantificateur universel, nous retournons vrai si la formule retourne vrai pour chaque noeud qui quantifie le chemin. Voici le pseudo-code de chacun des quantificateurs :

```
Quantificateur Existentiel
```

```
début
  réponse = faux

  Tant que réponse est faux et qu'il reste des noeuds à vérifier
    choisir le prochain noeud qui quantifie le chemin
    réponse = évaluation récursive de la formule en considérant ce noeud
  fin tant que

  retourner réponse
fin
```

```
Quantificateur Universel

début
  réponse = vrai

  Tant que réponse est vrai et qu'il reste des noeuds à vérifier
    choisir le prochain noeud qui quantifie le chemin
    réponse = évaluation récursive de la formule en considérant ce noeud
  fin tant que

  retourner réponse
fin
```

Les opérateurs logiques effectuent également des vérifications récursives. Comme nous le démontrerons, dans le cadre de l'exécution de cet algorithme de base, nous retenons diverses informations afin de permettre la possibilité de fournir une justification dans l'éventualité où une règle est fausse ainsi que pour permettre la reprise de la vérification à partir d'un noeud particulier. Beaucoup de choses sont donc greffés à cet algorithme de base, ce qui en

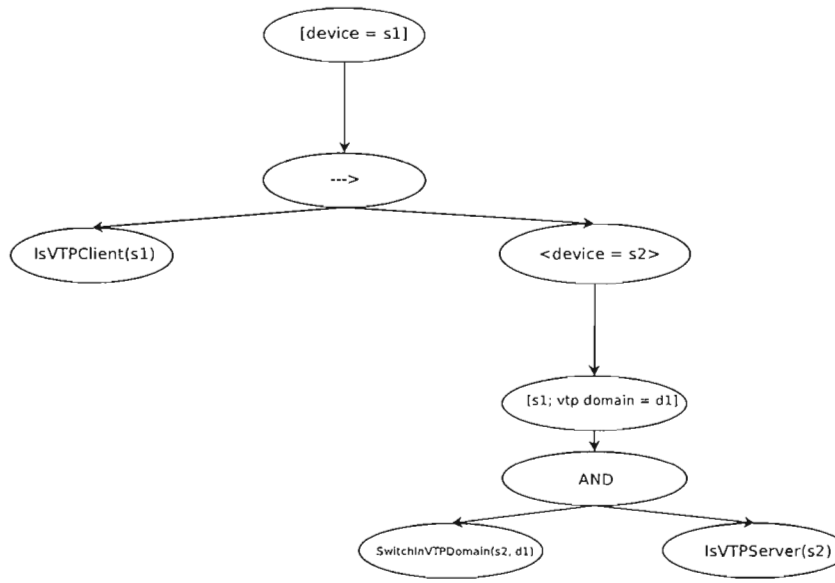

Fig. 6.3 Arbre expression de la règle 1

augmente la complexité.

Comme exemple de vérification nous prenons la règle 1. L'arbre d'expression de cette règle est représenté par la figure 6.2.

Si nous tentons de vérifier cette règle avec notre exemple de réseau, nous obtiendrons un échec. La règle n'est pas vérifiée puisque le commutateur *Switch2* a VTP activé en mode client et qu'il n'y a aucun serveur dans le domaine. L'algorithme de vérification se déroule de la façon suivante :

Premièrement, on retrouve à la racine de l'arbre de la figure 6.2 un quantificateur universel pour la variable *s1*. Le fonctionnement du quantificateur fera donc en sorte que cette variable doit prendre les valeurs de chaque noeud pouvant la quantifier. Puisque la variable à quantifier st au sommet du chemin, il s'agit d'une racine.

Dans notre exemple, il y a deux racines *Switch1* et *Switch2*. La variable *s1* sera donc initialement associée au premier commutateur soit le commutateur *Switch1*. Une fois cette première quantification effectuée, la formule associée au commutateur est évaluée récursivement. La

formule associée au commutateur est une implication logique. On commence donc par vérifier la partie gauche de l'implication. Dans l'éventualité où cette partie est fausse, l'implication retourne vrai, sinon, elle retourne le résultat obtenu par l'évaluation de la partie droite de l'implication.

Du côté gauche on retrouve le prédicat *IsVTPClient*. Ce prédicat est appelé avec comme paramètre effectif le commutateur *Switch1*. Puisque ce commutateur n'est pas un client VTP, le prédicat retourne faux et l'implication logique retourne vrai. La formule du quantificateur universel pour la variable *s1* a donc retourné vrai pour la première quantification de la variable *s1*.

L'algorithme affecte maintenant *s1* à *Switch2*. Comme pour le commutateur *s1*, on se retrouve sur le côté gauche de l'implication. Cette fois, par contre, le prédicat retourne vrai puisque *Switch2* est un client VTP. La vérification se poursuit donc du côté droit de l'implication.

Le côté droit de l'implication vérifie si il y a un serveur VTP dans le même domaine que le commutateur *Switch2*. Pour ce faire, nous utilisons deux nouvelles variables, *s2* et *d1*. La variable *s2* doit donc parcourir tous ls commutateurs. On débute par l'association *s2* = *Switch1*.

La formule associée au commutateur de *s2* est un autre quantificateur. Il quantifie la variable *d1*. La valeur choisie pour cette variable doit être sous le noeud de la variable *s1*. La variable *d1* sera donc associée à la valeur *Domain* qui est le domaine VTP de *Switch2*. L'algorithme traite finalement la conjonction en bas à droite de la figure 6.2. Pour que la formule celle-ci soit vraie, il faut que les deux côtés de la conjonction soient vrais. On commence par évaluer le côté gauche. Ce côté retourne faux puisque le commutateur représenté par s2 (*Switch1*) n'est pas dans le domaine représenté par d1 (*Domain*).

L'algorithme revient donc au quantificateur pour *s2* puisque le quantificateur pour *d1* n'a plus de noeuds candidats (il n'y a qu'un seul noeud *vtp domain* par appareil). La variable *s2* va maintenant être associée à *Switch2*. La variable *d1* va être quantifiée à nouveau mais puisque la valeur de *s1* n'a pas été changée, elle va simplement reprendre la même valeur que lors de la précédente itération soit *Domain*.

On se retrouve à nouveau sur la conjonction et on va à gauche. Puisque *s2* est maintenant dans le domaine *d1* (en ce moment *s1* et *s2* sont toutes deux égales à *Switch2*) le côté gauche retourne vrai. Toutefois, le côté droit retourne faux puisque *Switch2* n'est pas un serveur VTP.

Les quantificateurs existentiels pour *d1* et *s2* retournent tous les deux faux puisqu'aucun noeud ne vérifient la relation qui leur est associée. L'implication (le noeud immédiatement en bas de la racine) retourne faux puisque son côté gauche est vrai et son côté droit est faux. Le quantificateur universel retourne faux puisqu'il a trouvé un appareil pour lequel la formule est fausse. Cet appareil étant *Switch2*.

6.2.1 Justification en cas de réponse négative

Dans l'éventualité où une règle n'est pas vérifiée, il faut être capable de fournir une justificaton à l'ingénieur réseau. Une simple réponse négative n'aidera pas ce dernier à corriger le problème. Toutefois, on ne peut pas toujours justifier complètement le fait qu'une formule ou une partie d'une formule soit fausse.

La justification est construite de façon récursive au cours de la vérification. Le principe est que lorsqu'une partie de formule retourne faux, l'algorithme conserve des informations sur les valeurs des variables quantifiés qui ont rendu la formule fausse. Lorsque viendra le temps de justifier pourquoi une partie de formule est fausse, l'algorithme indiquera ces valeurs.

Dans notre exemple, le noeud à la racine de l'arbre de la figure 6.2 est un quantificateur universel. Si ce quantificateur est faux, alors c'est qu'il y a une valeur qui n'a pas vérifié la formule. Pour ce noeud, la justification est simplement d'indiquer la valeur en question. Nous indiquerons donc que l'affectation $s1 = Switch2$ ne vérifie pas la formule. Ceci permet à l'ingénieur réseau d'identifier l'appareil reponsable. Néanmoins, ceci ne lui indique pas pourquoi cet appareil a rendu la formule du quantificateur fausse.

Pour aller plus loin, il faut regarder comment il se fait que la formule a retourné faux. L'algorithme tentera donc de justifier pourquoi cette affectation a rendu la formule fausse. Dans notre exemple, cette formule est une implication logique. Si une implication logique

est fausse, on a nécessairement que la partie gauche est vraie et la partie droite fausse. La justification doit donc se retrouver à droite de l'implication logique. Cette partie de la formule est un quantificateur existentiel.

Le quantificateur existentiel offre une justification par quantification possible de la variable. Dans notre exemple, nous pourrions justifier que la règle est fausse de deux façons. La première serait de dire que *Switch1* n'est pas dans le domaine *Domain*. La deuxième serait de mentionner que la règle est fausse car *Switch2* n'est pas un serveur VTP. Lorsqu'une telle situation se présente, nous permettons à l'ingénieur réseau de reprendre la vérification à partir du noeud pour lequel la justification est multiple. Avant de reprendre la vérification, l'ingénieur réseau devra choisir un noeud pour quantifier la variable du quantificateur. Ceci permettra à l'algorithme de vérification de fournir une raison spécifique pour la valeur choisie.

La même stuation se présente dans le cas d'une disjonction. Dans ce cas, nous offrons la possibilité d'explorer la partie gauche ou la partie droite de la disjonction.

La figure 6.4 montre comment ce résultat est présenté dans ValidMaker. On y voit que la partie qui est fausse de la formule est surlignée. La figure 6.5 montre le choix présenté à l'utilisateur.

6.2.2 Construction récursive de la justification

Lorsque qu'une partie de formule est vraie, il n'y a aucune justification à fournir. Ce n'est que dans le cas où une partie de formule est fausse qu'il est nécessaire de conserver des informations sur l'état des variables. Afin de justifier une réponse négative, notre algorithme conserve une instance d'une classe nommée *Reason*.

Cette classe contient les valeurs de toutes les variables qui ont été assignées, une référence sur la partie de la formule qui a retourné faux et dans l'éventualité où c'est un quantificateur existentiel, une liste de valeurs candidates pour la reprise de la vérification. Le principe appliqué est que lorsqu'un noeud de la formule retourne faux et que l'instance de la classe *Reason* ne contient rien, il met à jour l'instance de cette classe avec les informations dis-

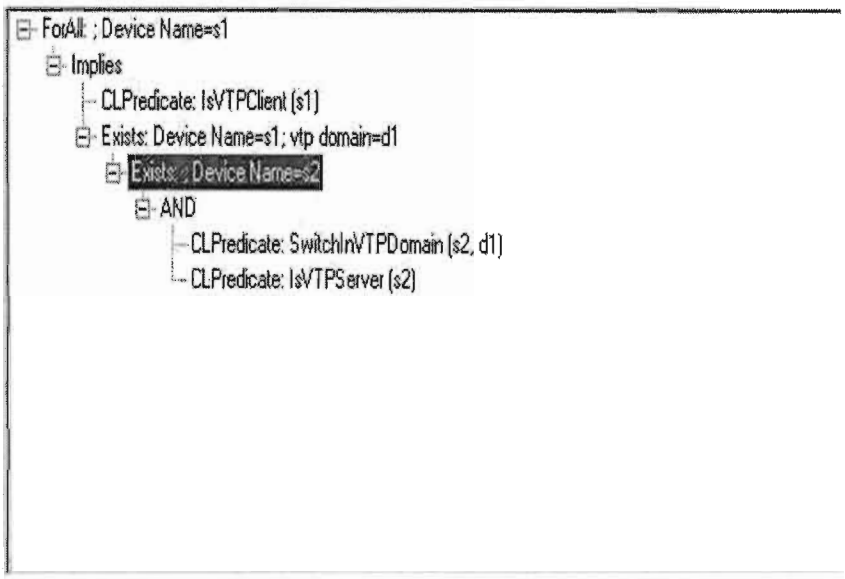

Fig. 6.4 La partie fausse d'une règle est surlignée

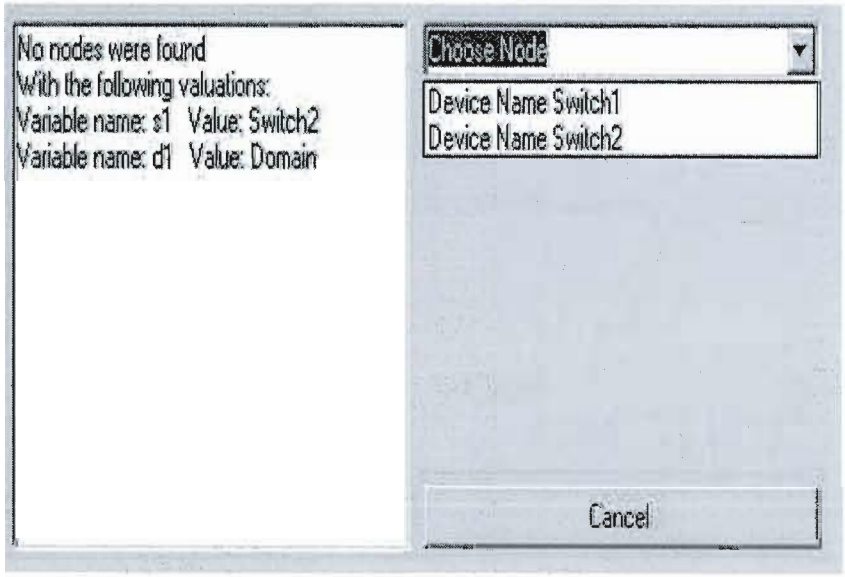

Fig. 6.5 Choix de noeuds présenté à l'utilisateur

ponibles à son niveau. Le quantificateur existentiel[1] et la disjonction sont des exceptions à cette règle générale. Ils écrasent toujours l'instance de la classe *Reason* puisque la vérification devra absolument reprendre à partir de ce type de noeuds.

Les différents comportement, eu égard à l'instance de la classe *Reason* que l'on retourne, selon le type de noeud sont les suivants :

Quantificateurs

UniversalQuantifier : Il génère une instance si elle n'existe pas déjà

ExistentialQuantifier : Il génère une instance

Opérateurs logiques

And : Il retourne l'instance du côté qui a retourné faux.

Or : Il génère une instance

Implies : Il retourne l'instance du côté droit de l'implication.

Opérateurs de comparaison

Equal : Il génère une instance

Different : Il génère une instance

Prédicat

PredicateReference : Il génère une instance

6.2.3 Reprise à partir d'un noeud particulier

Revenons à notre exemple. L'instance de la classe *Reason* qui est associée à l'échec de la

[1]Sauf s'il n'y a qu'une seule façon de quantifier la variable.

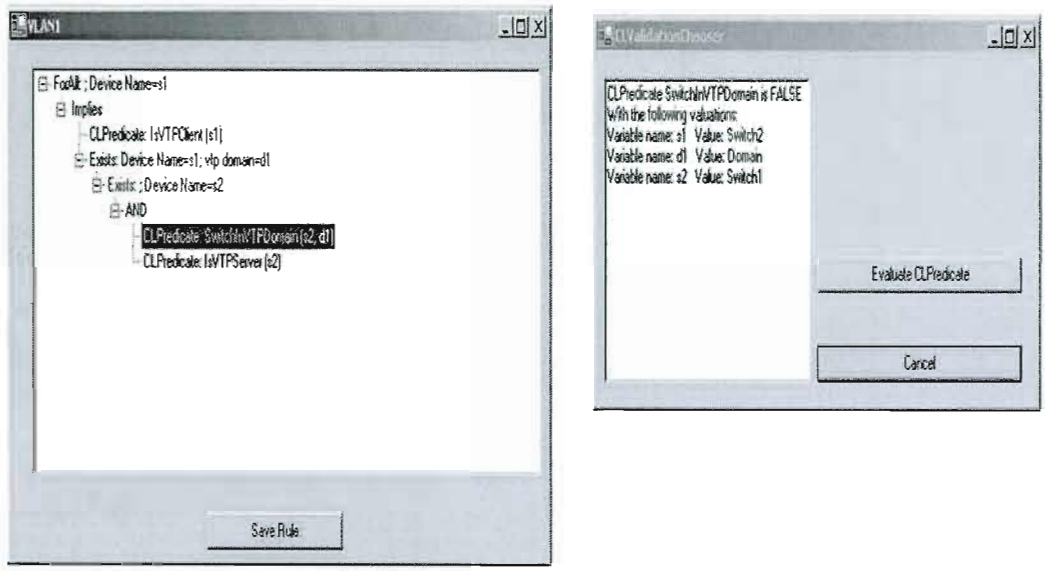

Fig. 6.6 Résultat de la reprise de la vérification

vérification contient donc la liste des valeurs et une liste de noeud candidats. La seule variable quantifiée est *s1*, qui a la valeur *Switch2*. Le noeud de l'arbre d'expression à partir duquel la vérification sera reprise est le quantificateur de la variable *s2*. La sous-formule évaluée est donc ce quantificateur.

L'ingénieur réseau devra choisir quel appareil quantifiera la variable. C'est ici que l'ingénieur réseau pourra, en choisissant un noeud, aller explorer différentes possibilités qui lui permettront de détecter le problème et de le corriger.

Ici, l'ingénieur réseau devra donc choisir entre *Switch1* et *Switch2*. Supposons qu'il choisit *Switch1*, la vérification reprendra à partir du quantificateur de *s2* mais la variable ne sera pas quantifiée avec chaque appareil du réseau mais seulement avec *Switch2*. La vérification retournera encore faux mais cette fois la raison contiendra des informations supplémentaires. Des valeurs supplémentaires seront quantifiés. Le noeud à partir duquel il sera possible de reprendre sera le prédicat *SwitchInVTPDomain*. La figure 6.2.3 montre le résultat de cette reprise de la vérification.

L'ingénieur réseau réalisera donc que la règle est présentement fausse car le commutateur

Switch1 n'est pas dans le domaine *Domain*. Cette information pourrait être suffisante pour corriger le problème. Néanmoins, il est possible d'aller explorer pourquoi un prédicat retourne faux. Si l'ingénieur réseau décide d'évaluer le prédicat la vérification reprendra à partir de ce dernier.

6.2.4 Vérification à partir d'un prédicat

Si le noeud à partir duquel il faut reprendre est un *PredicateReference* il n'y aura pas de choix de noeud à faire mais simplement une indication que le prédicat en question a retourné faux. Le prédicat étant similaire à une règle en plusieurs points, il est alors possible de reprendre l'analyse a partir de ce prédicat. Cette reprise de l'analyse est similaire au départ d'une validation d'une règle. On appelle la fonction d'évaluation de la racine du prédicat.

Toutefois, si dans un déclenchement d'une vérification de règle, la liste de valeurs pour les variables est initialement vide, tel n'est pas le cas du prédicat. Il faut tout d'abord trouver les valeurs pour lesquels le prédicat a retourné faux. Ces valeurs correspondent aux paramètres du prédicat. Une recherche des valeurs et un changement de variables sont donc nécessaires. Nous reconstruisons une nouvelle liste de valeurs qui correpondent aux paramètres effectifs du prédicats.

Dans notre exemple, le prédicat *SwitchInVTPDomain* a retourné faux. Si l'utilisateur demande l'évaluation, on ne retiendra que les valeurs correspondant aux paramètres du prédicat soit *Switch1* et *Domain*. L'arbre d'expression du prédicat *SwitchInVTPDomain* est représenté à la figure 6.7.

Le seul quantificateur du prédicat est pour la variable d qui s'associera au domaine du commutateur avec lequel le prédicat est appelé. Puisque ce commutateur est *Switch1* la variable prendra la valeur *MyDomain*. La raison qui sera fournie sera un message mentionnant que *MyDomain* est différent de *Domain*. Ce message indiquera à l'ingénieur réseau la nature du problème. Il pourra corriger en ajustant le paramètre *vtp domain* d'un des deux commutateurs pour qu'ils correspondent. La figure 6.2.4 représente le résultat final de notre trace. on peut y voir qu'une substitution de variables a été effectuée pour l'évaluation du prédicat ainsi que la raison pourquoi la règle est fausse.

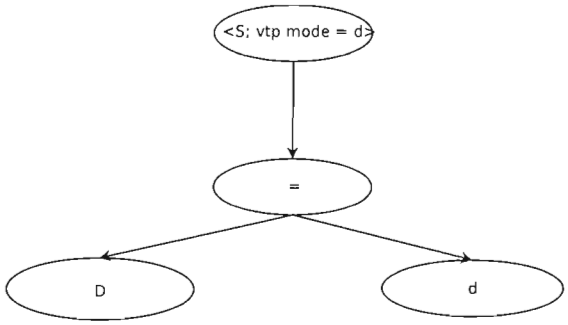

Fig. 6.7 Arbre Expression SwitchInVTPDomain

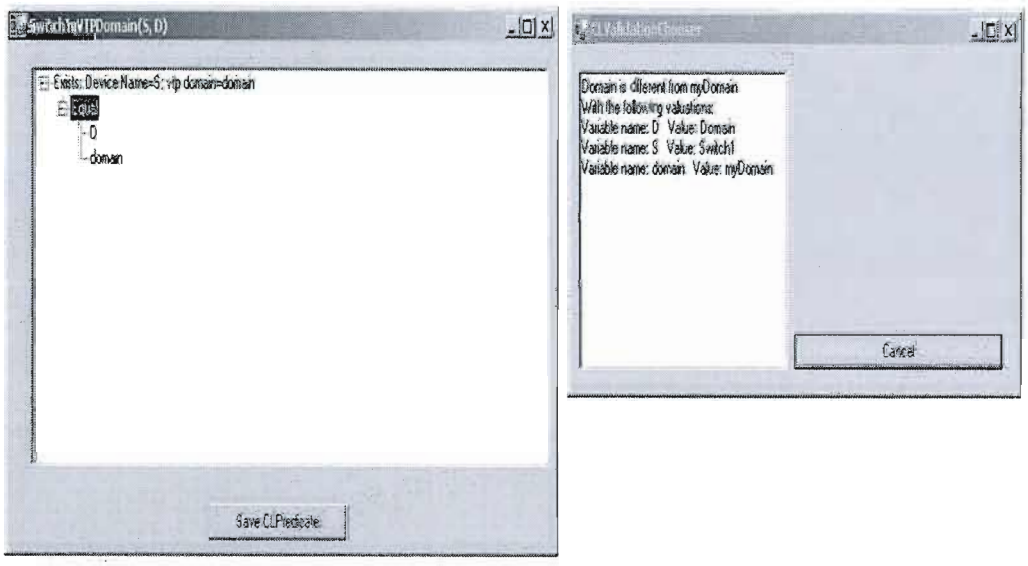

Fig. 6.8 Changement de variables lorsqu'un prédicat est évalué

Cette trace démontre que l'implantation de la logique de configuration a été faite en permettant de retenir des informations sur la vérification effectuée qui nous permet de reprendre la vérification à partir d'un noeud particulier. Ce processus permet à l'utilisateur d'aller en profondeur dans l'analyse du réseau et ainsi de cibler précisément la nature du problème ce qui accélérera la découverte de l'erreur et donc le correctif nécessaire. L'introduction de cette possibilité est notre principal contribution au développement de *ValidMaker*. La prochaine section présente les performances de notre algorithme.

6.3 Résultats obtenus

Afin de tester comment se comporte notre algorithme eu égard à l'augmentation de la taille du réseau, nous avons créé des scripts qui génèrent des fichiers de configurations de commutateurs. Sur ces commutateurs, nous avons fait varier le nombre de VLANs et d'interfaces assignées à des VLANs. Nous avons également fait varier le nombre de commutateurs. La variation du nombre de VLANs n'a eu aucun impact sur les temps d'exécution. Toutefois, la variation du nombre de commutateur et d'interfaces a eu un impact.

Sur chacun des exemples de réseaux que nous avons ainsi générés, nous avons mesuré le temps de vérification de chacune de nos règles. Les résultats sont présentés dans les tableaux suivants :

Nombre de commutateurs	règle 1	règle 2
10	0.0005	0.0007
20	0.0011	0.0023
40	0.0041	0.0078
80	0.0152	0.0297

Tab. 6.1 Variation du nombre de commutateurs

Puisque la règle 1 ne fait aucunement référence aux interfaces des commutateurs, il n'est pas surprenant que les temps soient constants lorsque le nombre d'interfaces changent. Toutefois, ceci n'est pas le cas dans le cadre où c'est le nombre de commutateurs qui change.

Nombre d'interfaces	règle 1	règle 2
4	0.031	0.30
8	0.031	0.63
12	0.031	1.30
23	0.031	1.97

Tab. 6.2 Variation du nombre d'interfaces

Pour la deuxième règle, l'augmentation du nombre de commutateurs ainsi que l'augmentation du nombre d'interfaces ont tout deux un impact. Ces temps démontrent que notre algorithme performe de façon acceptable lorsque la complexité présentée par le réseau augmente.

CONCLUSION

L'intérêt de la vérification automatique de la configuration de réseaux tient au fait que les réseaux deviennent de plus en plus complexes et que les interdépendances entre les différents matériels doivent être vérifiées. Puisque la configuration d'un matériel réseau peut être exprimée sous la forme d'un arbre de configuration où les noeuds sont des couples paramètres-valeurs, la logique de configuration est particulièrement bien adaptée à l'expression des règles applicables sur ces configurations.

Notre travail a consisté en l'implantation de la logique de configuration au sein d'un outil destiné à la gestion de la configuration de réseau nommé ValidMaker. Nous avions deux principaux objectifs. Premièrement, nous voulions prouver par l'exemple que la logique de configuration est appropriée à la vérification de réseaux. Ensuite, nous voulions donner à ValidMaker une plus grande expressivité et flexibilité quant à l'édition et à la vérification de règles de configuration de réseaux. Nous pouvons affirmer que ces deux objectifs ont été atteints.

Afin de prouver que la logique de configuration est appropriée et donne des résultats acceptables, un exemple réel a été développé. Nous avons ainsi développé un exemple de configuration de réseau afin d'en dégager certaines propriétés qui doivent être vérifiées pour que la configuration soit fonctionnelle. Pour atteindre cet objectif, nous avons basé nos exemples sur les VLANs statiques et le protocole VTP. Deux principales propriétés qui doivent être vérifiées pour assurer le bon fonctionnement des VLANs ont été trouvées.

Les algorithmes que nous avons développés nous permettent non seulement de vérifier la validité d'une règle mais également de retourner une raison si celle-ci est fausse. Ainsi, l'utilisateur de ValidMaker peut voir quelle partie de la règle entraîne l'échec de la vérification et ce en relation avec une partie de la configuration. Il est également possible de reprendre une vérification afin d'explorer les différentes possibilités.

Les algorithmes ont donc été implantés pour offrir la possibilité de reprendre la vérification à partir d'un noeud spécifique sans avoir besoin de réévaluer la règle à partir de la racine de l'arbre d'expression. L'utilisateur pourra donc choisir un noeud de façon interactive pour reprendre la vérification.

Une interface de création de règle a été développée pour permettre la création de règles au sein de ValidMaker. Toutefois, nous avons réalisé que la création de ces règles peut être lourde et non familière pour un ingénieur réseau. L'utilisation des prédicats vient simplifier cette tâche car la création de prédicats est analogue à la création de fonctions et ceci est familier aux ingénieurs réseaux. Les prédicats sont utilisés afin de regrouper certaines parties de règles dans un bloc. Ceci permet de donner un nom représentatif à certaines parties de règles et permet donc de simplifier la création et la lecture des règles. L'utilisation de prédicats favorise aussi la réutilisation de ces derniers puisque rien n'interdit qu'un prédicat soit utilisé dans plusieurs règles.

Le travail d'intégration dans ValidMaker de la logique de configuration a donc été mené à bien. La structure de ValidMaker n'a pas été modifiée de façon majeure de sorte que l'intégration de cette logique est relativement transparente. Les résultats obtenus nous permettent de constater que la vérification sur des réseaux de taille réelle prend un temps tout à fait acceptable. Nous croyons donc avoir démontré de façon empirique que la logique de configuration est particulièrement bien adaptée à la configuration de réseau et qu'il est possible de l'utiliser afin de vérifier des règles dans un réseau.

À titre de travaux futurs, il serait possible d'étendre notre processus de vérification de règles à un système qui pourrait s'auto-configurer ou s'auto-réparer. Au lieu de solliciter directement l'utilisateur quand une règle est fausse, l'algorithme de vérification serait modifié de façon à trouver une configuration qui permet de rendre une règle vraie. Ce ne serait qu'en cas de contradictions ne pouvant être résolues que l'utilisateur se verrait présenter divers correctifs potentiels et qu'il pourrait choisir entre ces divers correctifs.

Pour réaliser une telle chose, il faudrait trouver une façon de rattacher aux règles des façons de corriger une configuration lorsque celle-ci est incorrecte. Il pourrait également être envisagé de formuler des règles qui permettraient de configurer un appareil lors du branchement

de celui ci dans un réseau. Un tel système pourrait être suggéré afin d'être utilisé dans le cadre des réseaux autonomiques(14) pour réaliser une partie de l'auto-configuration et l'auto-réparation d'un réseau.

Bibliographie

(1) James Boney, *Cisco IOS in a nutshell*, O'Reilly, 2006.

(2) Villemaire R., Hallé S., Cherkaoui O., *Configuration Logic : A Multi-Site Modal Logic.* Proceddeings of the the 12th Interbational Symposium on Temporal Representation and Reasoning, TIME2005, June 23-25, Burlington, Vermont USA, 2005.

(3) Cisco Systems Inc., *Catalyst 2950 Desktop Switch Software Configuration Guide*, Cisco IOS Release 12.1(11) EA1 and 12.1(11) YJ, 2002.

(4) Cisco Systems Inc., *Catalyst 3550 Multilayer Switch Software Configuration Guide*, Cisco IOS Release 12.2(25) SE, 2004.

(5) Deac D., *Unified Network Management using Meta-CLI*, Thesis report, Université du Québec à Montréal, 2004.

(6) Deca R., Cherkaoui O., Puche D. *A Validation Solution for Network Configuration*, Communication Networks and Services Reasearch Conference (CSNR2004), Fredericton, N.B. (2004).

(7) Laboratoire Recherche Téléinformatique de l'Université du Québec à Montréal, *ValidMaker User Guide*, 2004.

(8) Enns R., NETCONF Configuration Protocol. Internet draft, Feb 2004. http ://bgp.potaroo.net/ietf/ids/draft-ietf-netconf-prot-12.txt

(9) Hallé S., Deca R., Cherkaoui R., Villemaire R., *Automated Verification of Service Configuration on Network Devices*, Conference on Management of Multimedia Networks and Services(MMNS2004), Sand Diego, USA, 2004.

(10) Hallé S. *Formalismes logiques pour la gestion des configurations dans les réseaux informatiques*, Mémoire de maîtrise, Université du Québec à Montréal, 2005.

(11) Clarke E.M., Grumberg O., Peled D.A., *Model Checking*, MIT Press. Cambridge, MA, 2000.

(12) Cardelli L., *Describing semistructured data*, SIGMOD Record, 30(4), 80-85, 2001.

(13) J.D. Day and H. Zimmermann. The OSI Reference Model. Proc. of the IEEE, vol 71, 1983.

(14) Parashar M., Hariri S., *Autonomic Computing : An Overview*, J.P. Banâtre et al. : UPP 2004, LNCS 3566, pp.247-259, 2005.