ÉCOLE POLYTECHNIQUE DE LOUVAIN
Département d'Ingénierie Informatique

Université catholique de Louvain
UCL
Membre de l'Académie Louvain et du Pôle Louvain

Design et implémentation d'un logiciel de validation et de génération de configurations réseaux

Laurent VANBEVER Grégory PARDOEN

Mémoire présenté sous la direction du **Professeur Olivier BONAVENTURE**
en vue de l'obtention du grade d'Ingénieur Civil en Informatique
Année académique 2007–2008

Remerciements

Nous tenons tout particulièrement à remercier notre promoteur, le professeur Olivier Bonaventure, pour nous avoir permis d'étudier cette problématique. Son suivi actif, sa disponibilité ainsi que ses conseils avisés nous ont permis de mener ce travail à bien.

Nous tenons également à remercier Bruno Quoitin pour ses suggestions ainsi que ses idées qui, pour la plupart, se retrouvent d'une façon ou d'une autre dans ce travail. De même, nous remercions Pierre François pour ses remarques et ses conseils.

Nous remercions BELNET, le réseau national belge de la recherche, qui nous a fourni les configurations qui ont servi de base à la réflexion de ce mémoire. Nous remercions tout spécialement Jan Torreele pour le temps consacré à la lecture de notre travail.

Enfin, nous adressons nos remerciement à nos parents, Cédric, Carine et Natasha pour leurs relectures ainsi qu'à Pierre pour la réalisation de notre page de garde.

Table des matières

Table des figures

Table des listings

Chapitre 1

Introduction

Configurer un réseau s'avère être une tâche complexe et sujette aux erreurs. En effet, les opérateurs doivent assurer continuellement le bon fonctionnement de leur réseau tout en y apportant des changements fréquents (modification de topologie, ajout de nouveaux services, etc.). La difficulté de cette tâche réside principalement dans le fait qu'une majorité d'opérateurs ne disposent pas d'outils de gestion adaptés. Force est de constater que, de nos jours, la configuration manuelle du réseau équipement par équipement via une interface en ligne de commande – ou CLI, *Command Line Interface* – reste effectivement le mode opérationnel privilégié [CGG+04].

Or, il a été montré à maintes reprises que l'utilisation de ces mécanismes de bas niveau constitue la cause de nombreuses erreurs dans les configurations [FR01, CGG+04, Sch03, MWA02]. En effet, les CLI présentent de nombreux désavantages. Tout d'abord, elles ne disposent pas, en général, d'un modèle commun de données, et ce, parfois même pour un seul et unique constructeur. Dès lors, certaines commandes, bien qu'identiques, n'ont pas les mêmes significations en fonction des équipements et des versions. Ensuite, les CLI sont principalement destinées aux opérateurs qui sont capables de s'adapter facilement à des changements mineurs de syntaxe. Les CLI sont donc rarement utilisées par des logiciels en raison des difficultés d'analyse syntaxique. Enfin, étant donné que les CLI sont des systèmes propriétaires, il est difficile de les utiliser afin d'automatiser des tâches dans un environnement hétérogène au vu des disparités existant entre ces systèmes.

Plusieurs événements célèbres illustrent ces erreurs de configuration. En 1997, à la suite d'une erreur de configuration dans les politiques BGP (*Border Gateway Protocol*), l'AS7007[1] a annoncé à lui seul la majorité des routes de l'Internet. Il a ainsi attiré vers lui une partie du trafic Internet agissant comme un véritable « trou noir ». Cet événement a provoqué des troubles de connectivité dans le monde entier pendant plusieurs heures [5]. Plus récemment, en 2008, le site de vidéos en ligne *YouTube* a été victime d'une erreur d'un réseau pakistanais. Celle-ci a empêché l'accès au site à un grand nombre d'utilisateurs [14, 15]. Ce second événement constitue la preuve que ces problèmes sont plus que jamais d'actualité.

[1] Un AS ou *Autonomous System* est un réseau géré par une seule entité administrative, par exemple, une entreprise ou une université.

Au vu de ces faits, il est clair que de nouveaux paradigmes de configuration sont nécessaires. En 2001, plusieurs réunions ont été organisées lors de divers événements (NANOG-22, RIPE-40, IETF-52, etc.) afin de débuter un dialogue entre les opérateurs et les développeurs. En 2002, l'*Internet Architecture Board* a organisé un *workshop* dont le thème principal a été la gestion de réseaux. Lors de ces quelques jours, les opérateurs ont identifié leurs besoins dans ce domaine. Un grand nombre de points ont été discutés. Tout d'abord, il est apparu que la facilité d'utilisation des outils de gestion était une caractéristique indispensable pour les opérateurs. Ils ont également exprimé le désir de pouvoir se concentrer sur la configuration du réseau dans sa globalité et non plus sur la configuration de chaque équipement. De plus, ils ont évoqué le souhait de pouvoir effectuer aisément des vérifications de la consistance des configurations. Enfin, les opérateurs ont demandé un modèle standard de représentation d'un réseau au sein d'une base de données. Nous renvoyons le lecteur à [Sch03] s'il désire obtenir la liste exhaustive des points discutés. La conclusion de ce *workshop* [SPMF03] a résulté en l'approbation unanime d'une approche de gestion de réseaux basée sur le langage de balisage XML [PSMY+06].

Feamster [Fea04, FB05] se base également sur le constat de l'inadaptation des outils de configuration et souligne l'importance des techniques de vérification. Son travail s'est focalisé sur le protocole de routage interdomaine BGP. Il avance qu'à partir du moment où la configuration d'un protocole affecte son comportement, sa vérification relève d'un problème d'*analyse de programme*. Dans ce cas, une configuration est dite correcte si elle répond à un ensemble de spécifications. Ces dernières sont représentées sous la forme de règles. Selon lui, les opérateurs doivent en outre bénéficier d'une vue abstraite des protocoles afin de raisonner sur des considérations de haut niveau et non plus sur des détails de configuration.

Cette nouvelle façon de configurer un réseau peut être reliée aux méthodes de spécification et de vérification utilisées dans le domaine de l'ingénierie logicielle [Mey92]. La philosophie poursuivie ici est de permettre une vérification *a priori* du réseau. Elle contraste avec l'approche par essais-erreurs utilisée jusqu'à présent par les opérateurs [Nic04].

Dans ce mémoire, nous présenterons une solution logicielle qui tente de rencontrer l'ensemble des besoins formulés par les opérateurs en se focalisant sur l'aspect *validation* de la configuration d'un réseau. Le logiciel est intitulé *vng*, l'acronyme de *Validated Network Generator*. Il se veut être une solution *flexible* capable de *générer* des configurations de réseaux *validées*. La flexibilité du logiciel est motivée par le fait qu'il est extrêmement difficile de modéliser chaque concept réseau existant ou ayant existé au sein d'une seule et même représentation. Dès lors, nous désirons un logiciel dans lequel l'ajout d'une fonctionnalité soit la plus simple possible. Ainsi, nous permettrons à chaque opérateur de représenter facilement les fonctionnalités dont il a besoin pour configurer son propre réseau.

Avec *vng*, nous nous baserons essentiellement sur deux grands principes. En premier lieu, nous utiliserons une représentation de haut niveau afin de modéliser l'entièreté d'un réseau sous une seule entité. Cette dernière sera décrite sous la forme d'un fichier XML. Une telle représentation permet, entre autres, d'éviter la redondance et d'être totalement indépendant d'un vendeur particulier. De plus, l'introduction de certaines abstractions permet de simplifier la tâche de configuration. En second lieu, le logiciel se basera sur un ensemble de règles qui valident cette représentation. Ces règles doivent être vues comme des conditions que la représentation doit respecter. Après l'analyse de plusieurs réseaux, il s'est avéré que différents types de règles sont

capables de vérifier la plupart des conditions relatives à un réseau. Dans *vng*, cinq types de règles sont disponibles : des règles de *présence*, de *non-présence*, d'*unicité*, de *symétrie* et des règles *personnalisées*. Ces dernières permettent des vérifications plus avancées et non supportées par les quatre premières. Au même titre que la représentation, ces règles seront décrites dans un fichier XML. Elles seront implémentées grâce à trois techniques : une spécification qui contraint la structure de la représentation, un langage de requêtes qui permet de vérifier des conditions sur certains éléments de la représentation et un langage de programmation de haut niveau.

Enfin, *vng* permet de *générer* une configuration validée pour chaque équipement décrit dans la représentation et ce, dans n'importe quel langage de configuration ou de modélisation comme par exemple Cisco IOS [Cis98], Juniper JunOS [9] ou C-BGP [QU05]. Pour ce faire, nous utiliserons des modèles définissant ces langages.

L'ensemble du code source et des fichiers utilisés est repris sur le CD accompagnant ce mémoire. Notons également que les travaux présentés, dans ce mémoire, ont donné lieu à la rédaction d'un article intitulé « *Towards Validated Router Configurations* ».

Dans ce qui suit, nous présenterons la structure globale du document. Nous mentionnerons également le contenu de chaque chapitre et de chaque section.

vng se base en grande partie sur le langage de balisage XML. Dès lors, le premier chapitre (chapitre 2) sera consacré, dans un premier temps, à une présentation de ce langage (section 2.1) afin de familiariser le lecteur avec certains des concepts clés. La suite du chapitre exposera les technologies connexes utilisées. En premier lieu, nous décrirons XML Schema, un langage permettant de spécifier la structure d'un document XML (section 2.2). En second lieu, nous aborderons XQuery, un langage de requêtes qui permet l'interrogation d'un document XML (section 2.4). Enfin, nous exposerons XSLT (section 2.5), un langage de transformation d'un document XML.

Le chapitre 3 sera consacré à la formalisation des concepts abordés dans ce mémoire. Tout d'abord, nous présenterons la structure arborescente utilisée (section 3.1) afin de représenter la configuration d'un réseau. Ensuite, nous détaillerons les différents constituants des règles (section 3.2) ainsi que la façon dont celles-ci permettent de valider une représentation. À la section suivante (section 3.3), nous aborderons les cinq types de règles identifiés : les règles de *présence*, de *non-présence*, les règles d'*unicité*, les règles de *symétrie* et les règles *personnalisées*. Enfin, nous terminerons le chapitre par une description des dépendances entre les règles (section 3.4) avec la notion de *graphe de règles*.

Dans le chapitre suivant (chapitre 4), nous aborderons notre logiciel *vng* ainsi que sa structure. Pour commencer, nous justifierons nos différents choix technologiques (section 4.2). Ensuite, nous détaillerons concrètement la façon dont nous représentons un réseau au sein de *vng* (section 4.3). Enfin, la dernière section expliquera la manière dont nous représentons les règles et la façon dont nous nous assurons que la représentation de haut niveau les respecte.

Au chapitre 5, nous détaillerons notre représentation de la configuration écrite en XML. Nous débuterons par une description des principes utilisés ainsi que les conséquences liées à l'utilisation d'une représentation de haut niveau (section 5.1). Dans la section suivante (section 5.2), nous

détaillerons la structure de la représentation, c'est-à-dire l'ensemble des conventions à respecter afin de construire une représentation interprétable par notre système.

Le chapitre 6 sera consacré aux trois techniques d'implémentation des règles utilisées. Premièrement, nous aborderons les règles structurelles (section 6.1). Ces règles expriment des contraintes sur la structure de la représentation. Par la suite, nous présenterons les règles implémentées par l'intermédiaire du langage de requête XQuery (section 6.2). Enfin, nous étudierons à la section 6.3, les règles implémentées dans un langage de programmation de haut niveau.

Dans le chapitre 7, nous expliquerons comment des configurations d'équipements (IOS, JunOS, etc.) peuvent être générées à partir de notre représentation de haut niveau. Pour ce faire, nous détaillerons l'utilisation d'une représentation intermédiaire (section 7.1). Cette dernière permet de faciliter le processus de traduction en se rapprochant des représentations de bas niveau utilisées par les équipements. À la section suivante (section 7.2), nous expliquerons le processus de traduction en tant que tel. À cette fin, nous utiliserons un langage de transformation (XSLT) qui permet de transformer facilement un document XML en un autre document. Enfin, la section (section 7.2.1) traitera des différentes règles de « bonnes pratiques » présentes dans les configurations générées.

Le chapitre suivant (chapitre 8) sera consacré à une étude de l'utilisation de notre logiciel basée sur le réseau américain de la recherche *Abilene*. La première section (section 8.1) décrira sa topologie et sa représentation dans *vng*. Ensuite, nous présenterons la validation de ce réseau par notre logiciel (section 8.2). Cette section sera également l'occasion de donner quelques exemples d'erreurs détectées. La section 8.3 donnera un aperçu de l'utilisation de deux abstractions proposées afin de faciliter la mise en place de politiques interdomaines : la première décrivant les politiques de filtrage d'un domaine (section 8.3.1) et la seconde permettant de modifier le comportement d'exportation par défaut des routes (section 8.3.2).

Le dernier chapitre (chapitre 9) livrera un résumé de l'état de l'art, c'est-à-dire des principales solutions existantes dans le domaine de la validation et de la gestion de configurations de réseaux. La première section (section 9.1) évoquera quelques solutions standardisées. La deuxième (section 9.2) abordera plusieurs solutions de recherches en suivant deux approches : la première orientée « règles » (section 9.2.1), la seconde orientée « *data-mining* » (section 9.2.2).

Chapitre 2

Technologies XML

Le langage de balisage XML est devenu en quelques années une technologie incontournable dans de nombreux domaines [HM02]. Ce langage vise à permettre une structuration hiérarchique de l'information à l'aide de balises. Une balise peut être vue comme une étiquette qui permet d'identifier facilement des éléments d'information. XML tire sa force de sa grande simplicité car un document XML est un simple fichier textuel. Dès lors, il est possible de le *parser*, générer et manipuler à l'aide d'outils appropriés. En outre, cette caractéristique lui assure une grande portabilité et l'interopérabilité des systèmes qui l'utilisent. De plus, un fichier XML est relativement aisé à comprendre, et ce, même pour un être humain. En effet, la plupart du temps, les balises portent des noms explicites.

Cependant, le langage XML ne constitue qu'un moyen d'encoder de l'information. Par conséquent, les applications utilisent d'autres technologies afin d'effectuer des opérations sur cette information, par exemple, rechercher efficacement un élément au sein d'un document ou transformer un document en un autre.

Ce chapitre débutera par une une brève introduction à XML. Ensuite, nous présenterons successivement trois technologies associées et utilisées dans ce travail. En premier lieu, nous parlerons de W3C XML Schema qui permet de spécifier la structure d'un document XML. Ensuite, nous aborderons XPath, un langage permettant de parcourir facilement ce type de document. Nous introduirons également XQuery, un langage de requêtes qui permet d'effectuer des recherches au sein d'un document. Enfin, nous terminerons avec XSLT, un langage de transformation d'un document XML.

2.1 XML

Le langage XML ou *eXtensible Markup Language* est un langage de balisage extensible. Le standard est proposé et défini par le *World Wide Web Consortium* ou W3C [PSMY+06].

Le but premier de XML est de faciliter l'échange d'informations entre différents systèmes, principalement sur Internet. XML est basé sur deux idées très simples : représenter le document

au même titre que les données par des arbres et représenter le type de chacun d'entre eux comme des grammaires appliquées à ces mêmes arbres [BFRW01].

Un document XML se doit de respecter la syntaxe du langage XML, on dit alors qu'il est « bien formé ». Celui-ci est, en outre, « valide » s'il respecte des contraintes sémantiques particulières (section 2.2).

Comme son nom l'indique, XML est un langage extensible car il autorise les utilisateurs à définir leurs propres éléments, ce que d'autres langages de balisage ne permettent pas. Par exemple, HTML ou *HyperText Markup Language* n'autorise l'utilisation que d'une centaine de balises prédéfinies.

Un exemple de document XML est repris dans le listing 2.1. Celui-ci décrit une personne de manière très succincte [HM02]. Le document est composé d'un seul élément nommé `personne`. Cet élément correspond à la racine du document et est délimité par une balise ouvrante `<personne>` et une balise fermante `</personne>`. Tout ce qui est compris entre ces deux balises correspond au contenu de l'élément.

Listing 2.1 – Exemple de fichier XML décrivant une personne

```
<?xml version="1.0" encoding="ISO-8859-15" standalone="yes"?>
<personne>
  <prenom>Alan</prenom>
  <nom>Turing</nom>
  <age valeur="42"/>
  <professions>
    <profession>Informaticien</profession>
    <profession>Mathematicien</profession>
    <profession>Cryptographe</profession>
  </professions>
</personne>
```

L'élément `personne` contient quatre éléments enfants: `prenom`, `nom`, `age` et `professions`. L'élément `professions` dispose lui-même de trois éléments enfants `profession`. Un document XML décrit une structure hiérarchique. L'élément `professions` est aussi appelé parent des éléments `profession` et, de la même manière, l'élément `personne` est le parent de l'élément `professions`. Le contenu de l'élément `prenom` est la chaîne de caractères « `Alan` ».

Dans un document XML, il est obligatoire que tout élément possède un et un seul parent, exception faite de la racine qui n'a pas de parent. Concrètement, un élément doit être entièrement inclus dans un autre, c'est-à-dire que, si la balise ouvrante d'un élément est située au sein d'un élément, la balise fermante correspondante doit se trouver au sein de ce même élément. Par exemple, `<personne><prenom>Alan</personne></prenom>` n'est pas un document XML valide tandis que `<personne><prenom>Alan</prenom></personne>` l'est.

Des éléments XML peuvent avoir des attributs. Un attribut est une paire nom-valeur rattachée à la balise ouvrante d'un élément. Par exemple, dans le listing 2.1, l'élément `age` dispose d'un attribut nommé `valeur` et dont la valeur est égale à 42.

Il est aisé de voir qu'une balise ouvrante doit commencer par le caractère `<` tandis qu'une balise fermante par `</`. En ce qui concerne les éléments vides (ceux qui n'ont pas de contenu), il existe une syntaxe particulière. En effet, de tels éléments peuvent être représentés par une balise qui commence par `<` et se termine par `/>`. Un exemple est l'élément `<age valeur="42"/>`. Notons que cette notation est en tout point équivalente à celle-ci : `<age valeur="42"></age>`.

Des documents XML peuvent être commentés. Un commentaire commence par `<!--` et se termine à la première occurrence de `-->`.

Il n'est pas autoriser d'utiliser des caractères tels que `<, >, &` dans le contenu d'un élément ou d'un attribut XML. Ces caractères doivent être « encodés ». Par exemple, `<` deviendra `&lt`. De tels encodages ne sont pas pratiques si, par exemple, le document XML est utilisé pour contenir un autre document XML. Néanmoins, le langage XML permet de définir des sections `CDATA`. Celles-ci débutent par [CDATA[! et se terminent par `]]`. Tout ce qui est compris entre ces deux balises n'est pas interprété par l'analyseur syntaxique.

Le document peut éventuellement commencer par une déclaration XML avec trois attributs : `version`, `standalone` et `encoding`. Notons que si c'est le cas, la déclaration doit être la première instruction présente dans le document. En outre, un document XML doit contenir un et un seul élément racine.

2.2 XML Schema

Un schéma est une description formelle de la validité d'un document XML. Comme mentionné précédemment, il y a deux niveaux de vérification d'un document XML : le premier niveau stipule simplement qu'un document doit être « bien formé », c'est-à-dire qu'il doit respecter la syntaxe XML ; le second niveau, quant à lui, définit la validité d'un document XML, c'est-à-dire s'il respecte les règles sémantiques qui lui sont associées.

Un schéma XML est donc responsable de définir ces règles sémantiques ou « grammaire » associées au document XML. Dès lors, il peut être considéré comme un langage de description d'un document XML. Il existe beaucoup de langages de schémas, par exemple, *Document Type Definition* (DTD) [PSMY+06] ou W3C XML Schemas [FW04, BMTM04, BM04].

W3C XML Schema est plus puissant que DTD [HM02]. Tout d'abord, un schéma XML utilise une syntaxe XML et correspond donc à un document XML. Ensuite, XML Schema permet d'hériter des définitions d'autres documents. En outre, il supporte de nombreux types de données (entiers, flottants, etc.), là où DTD n'est capable que de traiter des données textuelles. De plus, un schéma XML permet de définir de nouveaux types de données. Enfin, il permet également d'exprimer des contraintes explicites sur le nombre d'enfants d'un élément, leur ordre, etc.

Notons néanmoins que W3C XML Schema est beaucoup plus verbeux que DTD, et ce, en raison de sa syntaxe XML. Malgré cela, nous l'avons retenu car il correspondait mieux à nos besoins. La principale raison qui a motivé notre choix est la possibilité de définir de nouveaux types de données, ce qui est capital pour le reste de ce travail. Dès lors, dans la suite de ce docu-

ment, à chaque fois que nous utiliserons le terme schéma, il s'agira de la version recommandée par le W3C.

Il est évident que nous ne détaillerons dans cette section qu'un sous-ensemble des possibilités offertes. Nous renvoyons le lecteur désireux d'en savoir plus aux spécifications afin d'obtenir une liste exhaustive des fonctionnalités [FW04, BMTM04, BM04].

Dans le listing 2.2, nous reprenons le schéma correspondant au document XML décrit dans le listing 2.1. Un schéma est un document XML dont la racine est obligatoirement l'élément `xs:schema`. Cet exemple reprend la définition de l'élément `personne` qui utilise plusieurs types complexes.

Listing 2.2 – Exemple de schéma XML décrivant une personne

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="personne">
        <xs:complexType>
            <xs:sequence>
                <!-- Elements obligatoires -->
                <xs:element name="prenom" type="xs:string" minOccurs="1" maxOccurs="1"/>
                <xs:element name="nom" type="xs:string" minOccurs="1" maxOccurs="1"/>
                <!-- Elements optionnels -->
                <xs:element name="age" type="ageType" minOccurs="0" maxOccurs="1"/>
                <xs:element name="professions" type="profType"
                    minOccurs="0" maxOccurs="1"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:complexType name="ageType">
        <xs:attribute name="valeur" type="positiveInteger" use="required"/>
    </xs:complexType>
    <xs:complexType name="profType">
        <xs:sequence>
            <xs:element name="profession" type="xs:string"
                minOccurs="1" maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>
    <xs:simpleType name="positiveInteger">
        <xs:restriction base="xs:integer">
            <xs:minInclusive value="0"/>
        </xs:restriction>
    </xs:simpleType>
</xs:schema>
```

L'élément `xs:element` est une des composantes les plus utilisées dans un schéma ; il décrit explicitement le contenu d'un élément d'un document XML. Il dispose d'un attribut obligatoire `name` qui représente le nom que doit prendre l'élément correspondant dans le document XML. Il peut comporter également certains attributs optionnels qui définissent, par exemple, le type de l'élément (`type`), sa cardinalité (`minOccurs`, `maxOccurs`), une valeur par défaut (`default`), etc.

À l'aide d'un schéma, il est possible de définir deux *types* de contenu : un simple (*simpleType*) et un complexe (*complexType*). Les types simples correspondent aux types de données de base que l'on retrouve dans les langages de programmations (chaînes de caractères, entiers, dates,

temps, etc.). Ceux-ci ne peuvent avoir d'attributs ou d'éléments imbriqués. Les types complexes peuvent, quant à eux, disposer d'éléments imbriqués et posséder des attributs[1].

Dans l'exemple repris dans le listing 2.2, nous définissons un type complexe à l'intérieur de l'élément nommé `personne`. Celui-ci contient l'élément `xs:sequence`. Cet élément implique que la liste de ses sous-éléments doit apparaître dans le document cible dans le même ordre que celui précisé, c'est-à-dire `prenom`, `nom`, `age` et enfin `professions`. Tout autre ordre amènera à une erreur lors de la validation du document. D'autres éléments permettent de préciser des contraintes sur les éléments enfants. Par exemple, l'élément `xs:choice` implique un choix parmi un ensemble d'enfants ou encore l'élément `xs:all` implique que tous les enfants doivent apparaître dans le document cible, et ce, dans n'importe quel ordre.

Un autre concept relativement important des schémas est celui des *facets*. Une *facet* est une contrainte sur l'« aspect » d'une valeur possible pour un type de données simple. Par exemple, un type numérique peut être restreint à des valeurs minimales et maximales. Dans le listing 2.2, nous définissons un nouveau type *positiveInteger* comme une restriction du type `xs:integer` avec comme contrainte que la valeur minimale autorisée est zéro (`xs:minInclusive`). Deux autres restrictions sont souvent utilisées dans le cadre de ce travail : d'une part, `xs:pattern` qui permet de définir des contraintes à l'aide d'expressions régulières et, d'autre part, `xs:enumeration` qui permet de définir une liste de valeurs possibles pour un type donné.

Les derniers concepts que nous désirons introduire concernent l'unicité ainsi que les clés définies entre les valeurs des éléments et les attributs d'un document. Ces concepts sont modélisés respectivement par les éléments `xs:unique` et `xs:key`. Ces deux derniers permettent d'assurer l'unicité de certains éléments dans l'entièreté d'un document XML.

Les deux constructions sont similaires et se font en deux temps. Tout d'abord, l'ensemble des éléments sur lesquels la contrainte d'unicité doit être évaluée est sélectionné grâce à une expression XPath (section 2.3). Ensuite, les attributs sur lesquels portent la contrainte sont également sélectionnés à l'aide d'une telle expression. Un exemple simple dans lequel ce genre de contraintes est important est repris dans le listing 2.3.

Cet exemple représente de manière très schématique le catalogue d'un magasin. Celui-ci définit d'abord l'ensemble des catégories qui sont présentes (élément `categorie`), chacune de ces catégories dispose d'un attribut `id`. Le catalogue contient également un ensemble d'éléments `produit`. Chacun de ceux-ci décrit un produit particulier et dispose d'un attribut `cat`.

Le schéma XML correspondant à l'exemple est repris au sein du listing 2.4. L'élément intéressant est `xs:key` nommé `catId` composé d'un sélecteur (`xs:selector`) qui sélectionne tous les éléments `categorie` et d'un champ sur lequel s'applique la contrainte, ici `id`. Cet élément s'assure donc qu'au sein de l'élément `catalogue`, il ne puisse exister deux éléments `categorie` tels que ces éléments ont la même valeur pour leur attribut `id`. Si c'est le cas, le document XML cible n'est pas valide.

Enfin, un schéma permet également de définir des références à des clés (*i.e.* à des éléments `xs:key`) grâce à l'élément `xs:keyref` et son attribut `refer`. Ces références permettent de s'assurer

[1]Le type d'un attribut doit toujours être un *simpleType* jamais un *complexType*

Listing 2.3 – Exemple de fichier XML décrivant un catalogue

```xml
<catalogue>
    <categorie id="COMMUTATEUR"/>
    <categorie id="ROUTEUR"/>
    <produit cat="COMMUTATEUR">
        <nom>Cisco Catalyst 2950</nom>
        <prix>1000</prix>
    </produit>
    <produit cat="COMMUTATEUR">
        <nom>Cisco Catalyst 6500</nom>
        <prix>2500</prix>
    </produit>
    <produit cat="ROUTEUR">
        <nom>Juniper M40</nom>
        <prix>10000</prix>
    </produit>
</catalogue>
```

Listing 2.4 – Exemple de schéma XML décrivant un catalogue

```xml
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="catalogue">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="categorie" minOccurs="1" maxOccurs="unbounded">
                    <xs:complexType>
                        <xs:attribute name="id" type="xs:string" use="required"/>
                    </xs:complexType>
                </xs:element>
                <xs:element name="produit" minOccurs="1" maxOccurs="unbounded">
                    <xs:complexType>
                        <xs:sequence>
                            <xs:element name="nom" type="xs:string" minOccurs="1" maxOccurs="1"/>
                            <xs:element name="prix" type="xs:float" minOccurs="1" maxOccurs="1"/>
                        </xs:sequence>
                        <xs:attribute name="cat" type="xs:string" use="required"/>
                    </xs:complexType>
                </xs:element>
            </xs:sequence>
        </xs:complexType>

        <xs:key name="catId">
            <xs:selector xpath="categorie"/>
            <xs:field xpath="@id"/>
        </xs:key>
        <xs:keyref name="produitCat" refer="catId">
            <xs:selector xpath="produit"/>
            <xs:field xpath="@cat"/>
        </xs:keyref>
    </xs:element>
</xs:schema>
```

que la valeur d'un élément correspond à la valeur prise par une clé. L'élément `xs:keyref` comporte également un sélecteur et un champ. Il indique que la valeur du champ de l'élément sélectionné par le sélecteur doit correspondre à une des valeurs de la clé référencée par l'attribut `refer`. Dans le listing 2.3 qui décrit un catalogue, une référence est définie entre l'attribut `cat` d'un élément

`produit` et l'attribut `id` de l'élément `categorie`. Cette référence est définie dans le schéma repris dans le listing 2.4. Dans ce schéma, l'élément `xs:keyref` nommé `produitCat` référence la clé `catId`.

2.3 XPath

Le langage XPath permet de parcourir aisément un fichier XML en tirant parti de sa structure arborescente. De manière générale, une expression XPath sélectionne des éléments ou des attributs XML. Par exemple, l'expression XPath sélectionnant l'ensemble des produits du listing 2.3 est donné par `doc("catalogue.xml")/catalogue/produit`. La fonction `doc()` permet de définir le fichier XML à ouvrir ; `catalogue` sélectionne l'élément `catalogue` qui est la racine du document ; `produit` sélectionne tous les enfants nommés de l'élément `catalogue` « produit ». Le résultat de cette expression correspond exactement aux trois éléments `produit` décrits au sein du listing 2.3.

Dans une expression XPath, les attributs XML sont sélectionnés en utilisant le symbole `@`. Par exemple, l'expression `doc("catalogue.xml")//produit/@cat` retourne les trois attributs `cat` de l'élément `produit` ; le symbole `//` permet de sélectionner un descendant quelle que soit sa position dans la hiérarchie. Par exemple `doc("catalogue.xml")/catalogue/produit` est équivalent à `doc("catalogue.xml")//produit`.

Enfin, ces expressions peuvent contenir des prédicats qui permettent de filtrer des éléments ou des attributs sur base d'une condition. Par exemple, l'expression :
`doc("catalogue.xml")//produit[@cat="ROUTEUR"]` retourne seulement les éléments `produit` dont l'attribut `cat` équivaut à `ROUTEUR`.

2.4 XQuery

XQuery est le langage de requêtes d'un document XML recommandé par le W3C [SRF+07, Wal07]. XQuery doit être vu comme l'équivalent de SQL appliqué à un document XML au lieu d'une base de données relationnelle.

Afin d'y voir plus clair sur les possibilités offertes par XQuery, voici une liste non exhaustive de ses fonctionnalités :

- sélectionner de l'information contenue dans un document XML en fonction de critères spécifiques ;
- filtrer de l'information ;
- chercher de l'information au sein d'un document ou d'un ensemble de documents ;
- joindre des données provenant de documents multiples ;
- trier, grouper et agréger des données.

Le langage XQuery utilise comme brique de base les expressions XPath décrites dans la section précédente. Les requêtes correspondent à des expressions de type *flwor*[2] qui est l'acronyme de *for, let, where, order by, return*. Ces expressions permettent de manipuler, transformer et trier les résultats, ce qui est impossible à réaliser avec des expressions XPath.

[2] *flowr* est prononcé *flower*.

Un exemple d'une expression *flwor* est repris dans le listing 2.5. Cette requête retourne les noms des produits appartenant à la catégorie « commutateur » du listing 2.3. Les résultats sont triés par prix. Par conséquent, le résultat de cette requête contient les deux éléments suivants : `<nom>Cisco Catalyst 2950</nom><nom>Cisco Catalyst 6500</nom>`.

Listing 2.5 – Exemple d'une requête XQuery

```
for $prod in doc("catalogue.xml")/catalogue/produit
where $prod/@cat = "COMMUTATEUR"
order by $prod/prix
return $prod/nom
```

Cette requête est composée de plusieurs éléments :

for : cette instruction définit une itération à travers la séquence d'éléments `produit`. À chaque itération, une variable nommée `$prod` est assignée à un des différents éléments `produit`.

where : cette instruction permet de filtrer les éléments sur base d'une condition. Dans le listing 2.5, la requête retournera les éléments `produit` appartenant à la catégorie « `COMMUTATEUR` ». La sémantique de l'instruction est identique à celle du prédicat `[@cat = "COMMUTATEUR"]` dans une expression XPath.

order by : comme son nom l'indique, cette instruction permet de trier les résultats suivant un critère donné. Dans ce cas-ci, il s'agit du prix de chaque élément.

return : cette instruction permet d'indiquer quel élément doit être retourné par la requête. Dans ce cas-ci, il s'agit des éléments enfants `nom`.

La clause *let* qui n'a pas été utilisée dans l'exemple précédent permet aussi de définir la valeur d'une variable. Cependant, contrairement à l'instruction *for*, elle ne définit pas une itération sur un ensemble d'éléments mais définit l'ensemble directement. Une requête dont le résultat est identique à la requête 2.5 est repris dans le listing 2.6.

Listing 2.6 – Exemple d'une requête XQuery utilisant une instruction *let*

```
for $prod in doc("catalogue.xml")/catalogue/produit
let $nom := $prod/nom
where $prod/@cat = "COMMUTATEUR"
order by $prod/prix
return $nom
```

2.5 XSLT

XSLT ou *Extensible Stylesheet Language for Transformations* est un langage de transformation d'un document XML. XSLT suit la spécification du W3C [Tid01, Cla99]. Le champ d'action de XSLT est énorme, car il permet de transformer un document XML en à peu près « n'importe quoi » comme par exemple, un autre document XML, un fichier HTML, un document PDF (*Portable Format Document*), du code Java ou un fichier texte. Dans cette section, nous ne

présenterons qu'une petite partie des possibilités offertes par XSLT. Nous renvoyons le lecteur à la définition du standard pour plus d'informations [Cla99].

Comme W3C XML Schema (section 2.2), une feuille de style XSLT est un document XML. XLST se base principalement sur le principe dit de *pattern matching*, c'est-à-dire qu'un document XSLT est composé d'un ensemble de règles dont la signification est : « *lorsque* tel élément est rencontré, *alors* exécuter l'action suivante ».

Un autre élément important de XSLT réside dans le fait que ce langage est fortement inspiré des langages fonctionnels. Un programme peut donc être vu comme une série de fonctions. Une caractéristique importante de ces langages est qu'ils utilisent des variables immuables (i.e. dont le contenu ne peut être changé au cours du temps). L'une des conséquences d'un tel paradigme est qu'il est impossible d'exécuter une boucle classique (p.ex. une boucle *for*) en XSLT. Pour ce faire, il faudra utiliser les principes d'itération et de récursivité.

Un document XSLT contient un ensemble d'instructions de type *pattern-template*, la partie *pattern* est utilisée afin de sélectionner un élément dans le document XML source. Cette partie est exprimée grâce à une expression XPath. La partie *template* décrit le format du fichier destination correspondant aux éléments sélectionnés. Dans le document résultat, certains éléments peuvent être filtrés, réorganisés ou encore ajoutés, et ce, par rapport au document initial.

Dans le listing 2.7, nous avons créé une feuille de style qui convertit le document XML repris dans le listing 2.3 en un fichier texte contenant une liste des noms des différents produits suivis de leur prix. Une feuille de style est un document XML, celle-ci doit posséder comme racine l'élément `xsl:stylesheet`.

L'un des éléments les plus importants de XLST est l'élément `xsl:template` ; cet élément dispose d'un attribut `match` qui contient le *pattern* permettant de sélectionner certains noeuds du document cible. Ce *pattern* est exprimé à l'aide d'une expression XPath.

L'élément `xsl:for-each` permet de définir une itération parmi une collection définie grâce à l'attribut `select`. Dans ce cas-ci, l'itération se fait sur tous les éléments enfants `produit` de l'élément `catalogue`. En XSLT, les références sont définies de manière relative à l'élément parent. Ainsi, dans l'exemple, `<xsl:for-each select="produit">` désigne les enfants `produit` de l'élément `catalogue` sélectionné par `<xsl:template match="catalogue">`.

Listing 2.7 – Exemple d'une feuille de style XSLT

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
    <xsl:template match="catalogue">
        Liste d'equipements disponibles :
        <xsl:for-each select="produit">
            <xsl:value-of select="nom"/> - (<xsl:value-of select="prix"/> EUR)
        </xsl:for-each>
    </xsl:template>
</xsl:stylesheet>
```

Chapitre 3

Validation d'un réseau

Les approches d'ingénierie logicielle (*Software Engineering*) utilisant des *pré/post conditions* [Mey92] et celles utilisant des méthodes formelles [dBBGdR06] ont prouvé leur importance dans le domaine du développement de logiciels. Néanmoins, ces pratiques sont rarement utilisées pour configurer des réseaux. En effet, la méthode actuelle de configuration suit généralement l'approche par essais-erreurs en configurant individuellement chaque équipement. Ceux-ci sont configurés manuellement à l'aide d'une interface en ligne de commande ou CLI (*i.e. Command Line Interface*). Les CLI n'ont pas été conçues pour automatiser la configuration des routeurs. En effet, la sémantique des commandes d'une CLI n'est pas stable car celles-ci peuvent avoir une signification différente selon la version utilisée. De plus, les constructeurs utilisent chacun une CLI propriétaire. Cette approche de bas niveau a déjà, par le passé, provoqué des erreurs [FR01, CGG$^+$04, Sch03, MWA02] et continue d'en provoquer aujourd'hui.

Plusieurs événements illustrent ce constat. En 1997, l'AS7007 a annoncé la majorité des routes de l'Internet suite à une erreur de configuration dans ses politiques BGP [5]. En 2001, l'AS3561 a propagé plus de 5000 routes incorrectes provenant d'un de ses clients [4]. Ces deux événements ont provoqué de graves problèmes de connectivité dans l'entièreté de l'Internet. Récemment, le site de vidéos en ligne *YouTube* a été injoignable pour de nombreux utilisateurs à la suite d'une erreur dans un réseau pakistanais [14, 15]. Ces événements mettent en lumière l'importance de valider la configuration d'un réseau avant de le déployer.

Les événements cités précédemment auraient pu être évités en utilisant une approche d'ingénierie logicielle permettant de valider la configuration d'un réseau. Dans cet objectif, nous avons étudié des réseaux réels dont deux réseaux de recherche (BELNET et *Abilene*) ainsi qu'un réseau universitaire (Université catholique de Louvain). Nous avons également étudié des recommandations issues d'ouvrages spécialisés [Hal01, DB06, Doy98, GS02, KD02, Jun04, Jun05, Gar06] ainsi que des publications scientifiques dans le domaine tel que rcc [FB05], minerals [LLW$^+$06] et bien d'autres [MWA02, PS03, FR01, 12]. Cette étude nous a montré qu'un réseau peut être validé par l'utilisation de règles. Celles-ci correspondent à des spécifications que le réseau doit respecter. Par exemple, une règle peut vérifier que toutes les sessions iBGP soient établies entre les adresses *loopback* des routeurs pour obtenir une plus grande résistance aux pannes [Hal01]. Dans cette approche, un réseau est validé s'il respecte toutes les règles définies.

Dans ce chapitre, nous présenterons la démarche que nous avons suivie afin de valider la configuration d'un réseau. Plus particulièrement, ce chapitre montrera comment les règles de validation peuvent être exprimées. Celles-ci vérifient des conditions sur des éléments de configuration. Dans un premier temps, nous présenterons comment ces éléments de configuration sont représentés sous la forme d'un arbre. Dans un deuxième temps, nous détaillerons les différents types de règles identifiés et leur formalisation.

3.1 Représentation de la configuration

La configuration d'un réseau peut être vue comme un ensemble d'éléments de configuration. Ceux-ci représentent n'importe quel élément pertinent pour la configuration d'un réseau tel qu'un routeur, une interface ou un protocole de routage. Tous ces éléments forment ensemble une description complète de la configuration du réseau. Ceux-ci sont représentés sous la forme d'un arbre $T = (V, E)$ avec V l'ensemble des noeuds de l'arbre et E l'ensemble des arêtes de l'arbre. Toutes les relations habituelles telles que enfant, parent, descendant et ancêtre sont applicables à l'arbre T. Un noeud de l'arbre représente un élément de configuration et est appelé un *CNode* (*i.e.* Configuration Node).

Un élément de configuration peut être un agrégat d'autres éléments. Une agrégation est une association qui représente une relation d'inclusion structurelle d'un élément dans un ensemble [Aud08]. Par exemple, dans le contexte des réseaux, un routeur est un agrégat de ses interfaces et la topologie est un agrégat de routeurs et de liens. La relation d'agrégation est notée, en UML, par un losange vide $\Diamond$ du côté de l'agrégat [Aud08]. La figure 3.1 montre un extrait de la représentation UML de la topologie dans laquelle un lien est un agrégat d'interfaces.

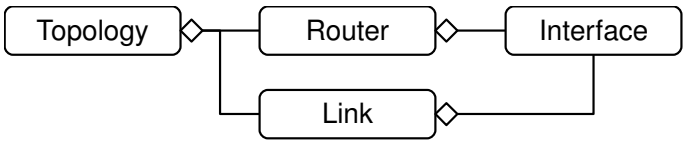

FIG. 3.1 – Extrait de la représentation UML de la topologie

Ce type d'association est modélisé dans l'arbre T par une relation parent-enfant. Un *CNode* p est un enfant du *CNode* q (*i.e.* $(p, q) \in E$), si l'élément de configuration représenté par q est un agrégat de l'élément de configuration représenté par p.

Un extrait de la représentation arborescente d'une configuration d'un réseau composé de quatre routeurs est repris dans la figure 3.2. L'arbre décrit plus particulièrement la représentation du lien entre les routeurs $R1$ et $R2$. La racine de l'arbre est un *CNode* qui représente l'entièreté du réseau et est nommé `domain`. Il possède un *CNode* enfant représentant la topologie qui, lui-même possède deux enfants : un *CNode* représentant les routeurs et un *CNode* représentant les liens physiques entre ceux-ci. Ceci correspond bien à la relation d'agrégation expliqué précédemment entre d'une part, la topologie et d'autre part, les routeurs et les liens du réseau.

Un *CNode* contient de l'information accessible par des champs. La valeur du champ `f` du *CNode* p est notée `p.f`. Un *CNode* contient toujours le champ obligatoire `type` indiquant quel

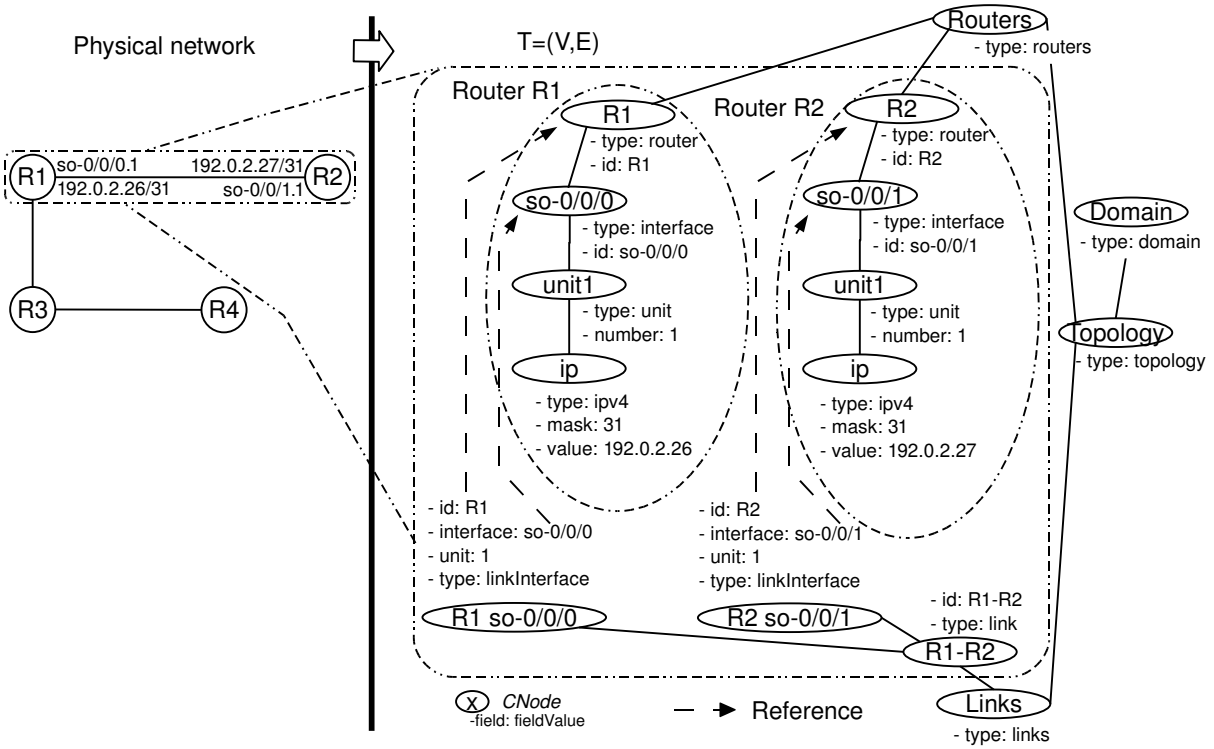

FIG. 3.2 – Arbre représentant un lien entre deux routeurs

élément de configuration ce *CNode* représente. Par convention, un *CNode* avec son champ `type` égal à `xyz` est appelé un *CNode* de type `xyz`. Par exemple, dans la figure 3.2, le routeur $R1$ est représenté par un *CNode* de `type router` dont le champ `id` vaut `R1`.

La valeur d'un champ peut être soit une valeur textuelle (p.ex. une chaîne de caractères, une valeur entière, etc.), soit une référence à un autre *CNode*. Par exemple, le lien entre le routeur $R1$ et le routeur $R2$ est représenté par un *CNode* de type `link` avec comme `id R1-R2`. Ce lien est composé de deux interfaces représentées par des *CNodes* de type `linkInterface`. Un *CNode* de ce type contient les champs `id` et `interface`. Ceux-ci contiennent des références vers respectivement un *CNode* de type `router` et un *CNode* de type `interface`. Ce dernier indique quelle interface du routeur précédemment référencé est connectée au lien.

3.2 Représentation des règles

Nos études préliminaires avaient pour objectif d'obtenir un ensemble de règles non triviales que nous avons rassemblées afin d'en obtenir une représentation efficace. La première particularité commune à toutes les règles est le fait qu'elles sont toutes appliquées à un ensemble d'éléments de configuration, c'est-à-dire, à un ensemble de *CNodes*. Nous avons décidé d'appeler cet ensemble le SCOPE d'une règle. Par exemple, vérifier la présence d'un *router id* sur tous les routeurs du réseau nécessite un ensemble de *CNodes* qui représentent tous les routeurs. Cependant, beaucoup de règles ont besoin de plusieurs ensembles de *CNodes*. Par exemple, vérifier la présence d'une interface *loopback* sur tous les routeurs [Doy98, DB06] nécessite d'identifier pour

chacun de ceux-ci l'ensemble de ses interfaces. Cette règle nécessite donc plusieurs ensembles de *CNodes*, car chaque routeur dispose de son propre ensemble d'interfaces.

Afin de formaliser ces ensembles de *CNodes*, nous définissons la fonction $C(T, p)$ qui retourne une valeur booléenne indiquant si le *CNode* p satisfait une certaine condition dans l'arbre T. De manière plus formelle avec $\mathcal{T} = (\mathcal{V}, \mathcal{E})$ l'ensemble des arbres et $\mathcal{V}$ un ensemble de *CNodes* :

$$C : \mathcal{T} \times \mathcal{V} \rightarrow Boolean \tag{3.1}$$

Par exemple, pour obtenir tous les routeurs de la figure 3.2 (*i.e.* $R1$, $R2$, $R3$ et $R4$[1]), la fonction C renvoie *true* pour chaque *CNode* p tel que $p.type = router$ et *false* pour tous les autres *CNodes*. De manière plus formelle, nous pouvons écrire :

$$C(T, p) := \begin{cases} true & \text{si } p.type = router \\ false & \text{sinon} \end{cases} \tag{3.2}$$

Cette fonction peut être écrite de manière plus concise par l'équation 3.3. Cette notation sera utilisée dans la suite de ce chapitre.

$$C(T, p) := (p.type = router) \tag{3.3}$$

Une condition C peut également vérifier si un *CNode* contient un champ particulier. Ceci est réalisé par la fonction $C(T, p) := p.field$. Cette fonction renvoit *true* lorsque le *CNode* p contient le champ $field$, sinon *false* est renvoyé.

Des informations provenant d'autres *CNodes* de T peuvent être nécessaires pour décider si un *CNode* respecte une certaine condition. Ceci explique pourquoi la fonction C prend également l'arbre T comme paramètre. Par exemple, pour sélectionner uniquement les routeurs bordures, il faut sélectionner parmi tous les routeurs du réseau ceux qui possèdent au moins une session eBGP. La partie gauche de la figure 3.3 montre un réseau composé de trois routeurs formant un *full mesh* iBGP et d'une session eBGP entre $R1$ et $R4$. La partie droite de la figure montre la façon dont la session eBGP est représentée par un arbre. La session eBGP est décrite par un *CNode* de type `externalBGPSession` avec comme `id R1-R4`. Celui-ci possède deux enfants : un *CNode* de type `routerBGP` qui représente un routeur BGP du domaine et un *CNode* de type `externalBGPPeer` qui représente un voisin BGP externe au domaine. Décider si un routeur représenté par un *CNode* p est un routeur bordure nécessite de chercher dans T un *CNode* r de type `routerBGP` tel que $r.id = p.id$ (*i.e.* le champ `id` du *CNode* r contient une référence vers le *CNode* p) et tel que le parent de r soit un *CNode* de type `externalBGPSession`. Dans notre exemple, $R1$ est bien un routeur bordure car il existe un *CNode* de type `routerBGP` avec comme `id R1` dont le parent est un *CNode* de type `externalBGPSession`.

Comme expliqué ci-dessus, une fonction $C(T, p)$ peut utiliser les informations présentes dans d'autres *CNodes* de l'arbre. Dès lors, il est utile de définir quelques fonctions permettant de parcourir facilement T. Voici les différentes fonctions pouvant être utilisées au sein d'une condition C :

[1] Pour des raisons de lisibilité, les *CNodes* représentant les routeurs $R3$ et $R4$ ne sont pas montrés.

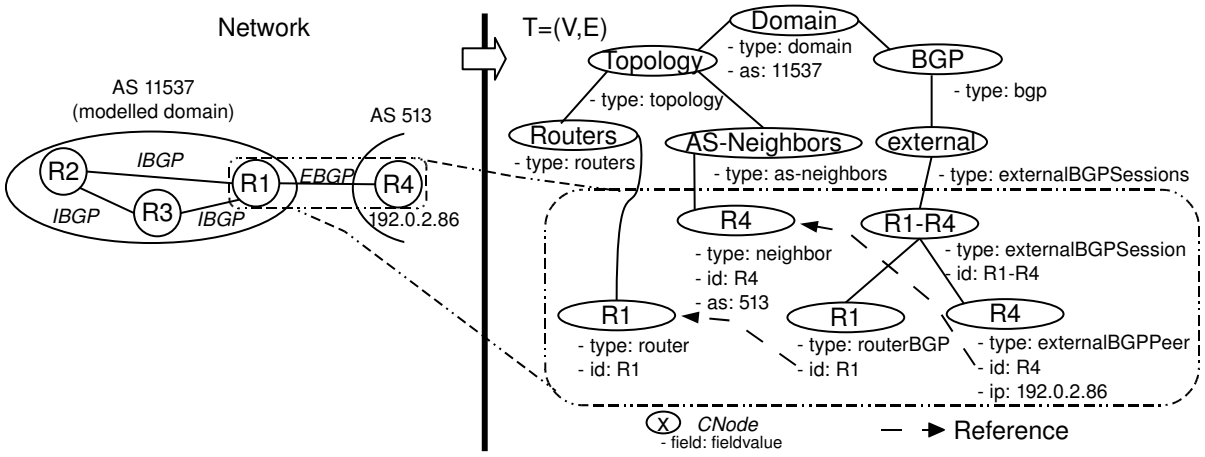

FIG. 3.3 – Arbre représentant une session eBGP

- $parent(p)$ renvoie le *CNode* parent du *CNode* p.
- $child(p)$ renvoie l'ensemble des *CNodes* enfants à p.
- $descendents(p)$ renvoie l'ensemble des *CNodes* descendants de p.
- $ancestors(p)$ renvoie l'ensemble des *CNodes* ancêtres de p.

Une fonction $C(T, p)$ peut également utiliser les quantificateurs $\forall$ et $\exists$.

La fonction permettant d'obtenir les routeurs bordures est donnée par l'équation 3.4. Pour rappel, $T = (V, E)$.

$$C(T,p) := \begin{cases} true & \text{si } p.type = router \text{ et } (\exists r \in V \ : \ r.type = routerBGP \\ & \text{et } r.id = p.id \text{ et } parent(r).type = externalBGPSession) \\ false & \text{sinon} \end{cases} \tag{3.4}$$

Afin de sélectionner des *CNodes* de l'arbre T pour former le SCOPE, nous utilisons une fonction C_{scope} respectant la définition de la fonction C. La fonction $C_{\text{scope}}(T, p)$ renvoie *true* si le *CNode* p doit faire partie du SCOPE, et renvoie *false* sinon. Un SCOPE est donc défini comme un sous-ensemble des *CNodes* de T respectant une condition C_{scope}. Cette définition est formalisée par l'équation 3.5.

$$\text{SCOPE} = \{p | p \in V \ : \ C_{\text{scope}}(T,p)\} \tag{3.5}$$

Par exemple, le SCOPE contenant les *CNodes* représentant tous les routeurs du réseau est donné par l'équation 3.6.

$$\text{ALLROUTERS} = \{p | p \in V \ : \ p.type = router\} \tag{3.6}$$

Comme cela a été introduit, certaines règles nécessitent plusieurs ensembles de *CNodes*. Notre étude des règles nous a montré que ces ensembles contiennent des *CNodes* qui sont des descendants d'autres *CNodes*. Par exemple, vérifier l'unicité des *vlan-id* définis sur une interface nécessite d'obtenir les *CNodes* représentant les *vlan-id* de chaque interface. Ces *CNodes* de type *vlan-id* sont des descendants des *CNodes* de type `interface`. Nous pouvons formaliser un ensemble

de *CNodes* de type *vlan-id* par l'équation 3.7. Cet ensemble est obtenu en sélectionnant parmi les descendants d'un *CNode p* ceux dont le type est `vlan-id`.

$$\texttt{vlanIds}(p) = \{q | q \in \mathit{descendents}(p) \ : \ q.\mathit{type} = \mathsf{vlan\text{-}id}\} \tag{3.7}$$

Cette formalisation peut être généralisée en introduisant une condition C_d qui permet de sélectionner quels sont les descendants qui doivent faire partie de l'ensemble. Cette formalisation est donnée par l'équation 3.8. Notons que la notation *descendents(p)* représente l'ensemble des descendants de p tandis que la notation $\texttt{descendents}(p)$ représente un ensemble des descendants de p qui respectent la condition C_d.

$$\texttt{descendents}(p) = \{q | q \in \mathit{descendents}(T, p) : \ C_\mathrm{d}(T, q)\} \tag{3.8}$$

Tous ces ensembles contiennent des *CNodes* qui sont des descendants d'autres *CNodes*. Nous pouvons regrouper tous les *CNodes* parents ou ancêtres dans un ensemble unique. Dans ce contexte, nous pouvons réutiliser la notion de SCOPE afin de représenter cet ensemble. En effet, un SCOPE est défini comme un ensemble de *CNodes* respectant une certaine condition. Il peut donc également contenir les *CNodes* ancêtres. Dès lors, un ensemble de descendants noté $\texttt{descendents}(p)$ est obtenu à partir d'un *CNode p* appartenant à l'ensemble SCOPE.

Nous pouvons distinguer deux types de règles : celles qui nécessitent d'utiliser un seul ensemble (*i.e.* SCOPE) et celles qui nécessitent d'obtenir des ensembles de descendants de celui-ci (*i.e.* `descendents`). Afin d'obtenir une représentation efficace, ces deux types de règles doivent pouvoir s'exprimer de manière unique. Dans cet objectif, les règles nécessitant un seul ensemble d'éléments sont considérées comme des cas particuliers de celles où la notion de `descendents` est utilisée. Ceci peut être réalisé en permettant à l'ensemble $\texttt{descendents}(p)$ de contenir uniquement p. La nouvelle définition ainsi obtenue est donnée par l'équation 3.9.

$$\texttt{descendents}(p) = \{q | q \in \mathit{descendents}(T, p) \cup p \ : \ C_\mathrm{d}(T, q)\} \tag{3.9}$$

Cette équation montre que l'ensemble des descendants de p est uni avec p lui-même. Ceci permet d'obtenir le cas particulier où $\texttt{descendents}(p) = \{p\}$. Ce cas particulier est utilisé par certains types de règles et sera détaillé dans les sections appropriées.

3.3 Types de règles

Notre étude nous a montré qu'il existe une grande variété de règles et que celles-ci n'ont pas toutes la même complexité. En effet, une règle vérifiant la présence du *router id* est intuitivement plus simple qu'une règle qui vérifie la connexité de la topologie. Malgré cette complexité variable, il est possible d'identifier des *patterns* communs à plusieurs règles. Autrement dit, il est possible d'identifier des règles qui vérifient des conditions similaires. Une fois ces *patterns* identifiés, nous pouvons les abstraire en différents types de règles. Prenons, par exemple, la vérification d'unicité

des adresses IP et la vérification d'unicité des *hostnames*. Le *pattern* mis en valeur est clairement la vérification d'unicité qui est identifiée comme un nouveau type de règles. Vérifier la présence d'un *router id* sur chaque routeur ou vérifier que les adresses *loopback* sont annoncées dans OSPF [GS02, KD02] sont des exemples de règles qui vérifient respectivement la présence du paramètre `rid` et la présence d'interfaces *loopback* dans OSPF. Ces deux règles suivent le même *pattern* qui est identifié comme un nouveau type de règles : les règles de présence.

Les cinq types de règles identifiés sont les règles de *présence*, de *non-présence*, d'*unicité*, de *symétrie* et les règles *personnalisées*. Cette section présentera ces différents types. La plupart de ceux-ci utilisent les notions de SCOPE et de `descendents` définis précédemment.

3.3.1 Règles de *présence*

Ce type de règles permet de vérifier que certains *CNodes* respectant une condition donnée sont présents dans l'arbre T. Par exemple, une règle de présence peut vérifier que toutes les interfaces possèdent une adresse IP ou encore que toutes les interfaces OSPF connectées à un voisin externe sont configurées en mode passif de manière à ne pas former d'adjacence avec celui-ci [GS02].

Ce type de règles couvre différents cas :

(i) vérifier la présence d'un champ d'un *CNode*.

(ii) vérifier la présence d'un *CNode* d'un certain type.

(iii) vérifier la présence d'un *CNode* dont un de ses champs respecte une condition donnée.

(iv) vérifier la présence d'un *CNode* respectant une certaine condition parmi un ensemble de *CNodes*.

Une règle de présence vérifie pour chaque *CNode* x du SCOPE s'il existe au moins un *CNode* dans `descendents`(x) qui respecte la condition C_{presence}. L'équation 3.10 est l'équation générale des règles de *présence*.

$$\forall x \in \text{SCOPE}\ \exists y \in \texttt{descendents}(x)\ :\ C_{\text{presence}}(T, y) \tag{3.10}$$

Par exemple, la règle qui vérifie que tous les routeurs possèdent au moins une interface *loopback* [DB06] peut être traduite plus formellement : vérifier que, pour chaque routeur, il existe une interface *loopback* parmi l'ensemble de ses interfaces. Le SCOPE de la règle est l'ensemble des routeurs. Chaque ensemble `descendents` regroupe les interfaces d'un routeur. Autrement dit, cette règle vérifie que pour chaque *CNode* x du SCOPE, il existe au moins un *CNode* y de `interfaces(x)` qui est une interface *loopback*. L'équation 3.11 montre la définition formelle de cette règle.

$$\forall x \in \text{ROUTERS}\ \exists y \in \texttt{interfaces}(x)\ : y.id = loopback \tag{3.11}$$

Tous les cas cités précédemment peuvent s'écrire sous la même forme que l'équation 3.10. Une explication détaillée est disponible dans l'annexe A.1. L'équation 3.10 est l'équation générale de la règle de présence. Celle-ci permet d'exprimer les règles de *présence* que nous trouvons les plus significatives.

3.3.2 Règles de *non-présence*

Cette règle permet de vérifier l'absence d'un *CNode* respectant une certaine condition dans T. Par exemple, une règle de *non-présence* peut être la vérification pour OSPF qu'il n'existe aucune *stub area* ou *not so stubby area* qui contient un *virtual link* [Doy98].

Cette règle est duale à la règle de *présence* et est montrée par l'équation 3.12. Nous observons que l'équation de la règle de *non-présence* est la négation de la partie comprenant `descendents(x)` et la condition de la règle de *présence* 3.10.

$$\forall x \in \text{SCOPE}\ \neg(\exists y \in \texttt{descendents}(x)\ :\ C_{\text{non-presence}}(T, y)) \tag{3.12}$$

L'équation 3.12 peut être écrite de manière équivalente par l'équation 3.13 en modifiant la portée de la négation (*i.e.* $\neg$). La règle de *non-présence* est respectée si, pour chaque *CNode* x du SCOPE, tous les *CNodes* de `descendents(x)` ne satisfont pas la condition $C_{\text{non-presence}}$.

$$\forall x \in \text{SCOPE}\ \forall y \in \texttt{descendents}(x)\ :\ \neg C_{\text{non-presence}}(T, y) \tag{3.13}$$

Par exemple, vérifier qu'il n'existe aucune *stub area* ou *not so stubby area* dans OSPF qui contient un *virtual link* est une règle utilisant un unique ensemble de *CNodes*. Elle est exprimée par l'équation 3.14.

$$\forall x \in \text{STUBAREAORNSSA}\ :\ \neg(\exists z \in \mathit{child}(x)\ :\ z.\mathit{type} = \mathit{virtualLink}) \tag{3.14}$$

Rappelons que la fonction $\mathit{child}(x)$ renvoie l'ensemble des *CNodes* enfants de x. Cette équation peut s'écrire de manière équivalente sous la forme de l'équation 3.13. La justification est donnée dans l'annexe A.2.

3.3.3 Règles d'*unicité*

Cette règle permet de vérifier l'unicité de la valeur d'un champ d'un ensemble de *CNodes*. Par exemple, vérifier l'unicité des adresses IP ou vérifier que toutes les sessions BGP utilisent des clés d'authentification différentes correspondent à des règles d'*unicité*. Notons $field$ le champ dont nous voulons garantir l'unicité de la valeur.

La règle d'*unicité* est respectée si, pour chaque *CNode* x du SCOPE, le champ `field` est unique pour tous les *CNodes* de `descendents`(x). Cette règle peut être formalisée par l'équation 3.15. Cette équation vérifie pour chaque *CNode* x du SCOPE et pour chaque *CNode* y de `descendents`(x), s'il n'existe pas un *CNode* z appartenant à `descendents`(x) qui possède la même valeur du champ $field$. Le *CNode* z doit être différent de y car choisir deux fois le même *CNode* n'a pas de sens. En effet, ils ont évidemment la même valeur du champ $field$.

$$\forall x \in \text{SCOPE}\ \forall y \in \texttt{descendents}(x)\ :\ \neg(\exists z_{\neq y} \in \texttt{descendents}(x) : y.field = z.field) \tag{3.15}$$

Par exemple, la règle vérifiant l'unicité du nom des interfaces peut être écrite par l'équatio 3.16. Le nom d'une interface est représenté par le champ `id` d'un *CNode* de type `interface`.

$$\forall x \in \text{ROUTERS}\ \forall y \in \texttt{interfaces}(x)\ :\ \neg(\exists z_{\neq y} \in \texttt{interfaces}(x) : y.id = z.id) \tag{3.16}$$

3.3.4 Règles de *symétrie*

Parmi les règles de *symétrie* identifiées, nous avons découvert deux modèles (*patterns*) qui couvrent la plupart des cas étudiés. Ceux-ci sont formalisés comme deux types de règles de *symétrie*. Le premier type consiste à vérifier l'égalité de la valeur d'un champ dans un ensemble de *CNodes*. Par exemple, la vérification de l'égalité des MTU pour des interfaces interconnectées [DB06] est un exemple d'une règle de symétrie dite d'*égalité simple*. Cet exemple de règle peut être écrit par l'équation 3.17.

$$\forall x \in \text{LINKS}\, \forall y, z \in \texttt{linkInterfaces}(x) : y.MTU = z.MTU \tag{3.17}$$

De manière générale, ce type de règle est respecté si, pour chaque *CNode* x du SCOPE, tous les *CNodes* de $\texttt{descendents}(x)$ possèdent la même valeur de **field**.

$$\forall x \in \text{SCOPE}\, \forall y, z \in \texttt{descendents}(x) : y.field = z.field \tag{3.18}$$

Le deuxième type de symétrie est relativement plus complexe car son objectif est de vérifier une symétrie entre des champs différents. Par exemple, l'établissement d'une session BGP entre le routeur A et le routeur B requiert que A possède comme *neighbor* B, et, de la même manière, que le routeur B possède comme *neighbor* le routeur A [Doy98, DB06]. Il est intéressant de bien remarquer que la condition de symétrie porte sur deux champs différents. En effet, la valeur indiquée dans le champ `neighbor` doit correspondre au champ `rid` du routeur avec lequel la session doit être établie. De plus, chaque routeur possède plusieurs *neighbors*. Cette règle vérifie donc que, pour chaque *neighbor* N du routeur A, le routeur N possède parmi l'ensemble de ses *neighbors* le routeur A. Cette règle de symétrie est dite d'*égalité croisée*.

Cet exemple peut être écrit par l'équation 3.19.

$$\forall x \in \text{BGPROUTERS}\, \forall y \in \texttt{BGPneighbors}(x) \ : \ \exists z \in \text{BGPROUTERS}\, \exists w \in \texttt{BGPneighbors}(z) \ : \\ y.neighbor = z.rid \text{ and } w.neighbor = x.rid \tag{3.19}$$

De manière générale, la règle de *symétrie* d'*égalité croisée* est vérifiée si pour chaque *CNode* x du SCOPE et, pour chaque *CNode* y de $\texttt{descendents}(x)$, il existe au moins un *CNode* z du SCOPE tel qu'il existe un *CNode* w de $\texttt{descendents}(z)$ vérifiant la symétrie entre `field1` et `field2` (*i.e.* $y.field1 = z.field2 \quad \text{and} \quad w.field1 = x.field2$).

$$\forall x \in \text{SCOPE}\ \forall y \in \texttt{descendents}(x) \ : \ \exists z \in \text{SCOPE}\ \exists w \in \texttt{descendents}(z) \ : \\ y.field1 = z.field2 \text{ and } w.field1 = x.field2 \tag{3.20}$$

3.3.5 Règles *personnalisées*

Si une règle n'est pas représentable à l'aide d'une règle de *présence*, de *non-présence*, d'*unicité* ou de *symétrie* alors celle-ci est une règle *personnalisée*. Ce type de règle est, de loin, le plus expressif et couvre les règles les plus complexes.

Vérifier la connexité du réseau ou que toutes les *area* OSPF sont directement connectées au *backbone area* [DB06] sont des règles personnalisées, car elles ne peuvent pas être exprimées à l'aide de l'un des types de règles expliqué précédemment.

3.4 Dépendances entre les règles

Les règles peuvent décrire des objectifs de haut ou de bas niveau. Par exemple, une règle de haut niveau peut représenter le fait qu'un protocole de routage fonctionne correctement ou que le réseau tout entier soit correct. La signification du mot « correct » est de la responsabilité de l'opérateur réseau dans le sens où c'est à lui de définir ses propres règles de *design*. Ainsi, un protocole de routage fonctionne correctement si toutes les règles concernant ce protocole sont respectées.

Une règle peut dépendre de plusieurs autres règles qui, elles-mêmes, peuvent dépendre d'autres règles et ainsi de suite. Une règle de plus haut niveau permet de raffiner plusieurs règles d'un niveau conceptuel inférieur. Ce raffinement peut être de type « et » ou de type « ou ». Le type « et » nécessite que toutes les règles raffinées soient vérifiées et respectées, tandis que le raffinement de type « ou » nécessite seulement qu'au moins une d'entre elles le soit.

Un graphe dirigé de règles est formé à partir de ces dépendances. Ce graphe doit être sans cycle pour garantir la terminaison de l'algorithme de vérification des règles[2].

La figure 3.4 montre un exemple de dépendances entre les règles.

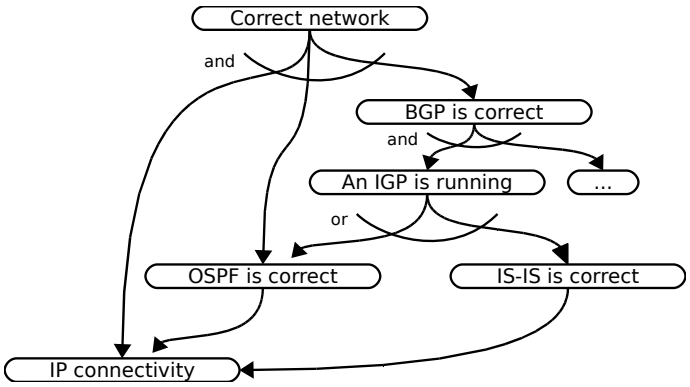

FIG. 3.4 – Exemple de dépendances entre les règles.

3.5 Conclusion

Ce chapitre a présenté une approche vérifiant la validité d'un réseau à l'aide de règles. Cette méthode de validation est similaire aux approches utilisées dans le domaine de l'ingénierie

[2]L'utilisation de raffinement « ou » permet, dans certains cas, à l'algorithme de se terminer malgré la présence de cycles !

logicielle. En effet, une règle est en tout point comparable à une condition ou pré-condition dans ce domaine.

Notre étude des règles nous a amenés à définir deux notions qui permettent d'exprimer efficacement la plupart des règles. Premièrement, la notion de SCOPE permet d'obtenir un ensemble d'éléments de configuration (*CNodes*). Deuxièmement, la notion de `descendents` permet d'obtenir d'autres ensembles composés de descendants des éléments du SCOPE respectant une certaine condition.

Dans ce chapitre, nous avons présenté cinq types de règles : les règles de *présence*, de *non-présence*, d'*unicité*, de *symétrie* et *personnalisées*. Les quatre premiers types ont été formalisés grâce aux notions de SCOPE et de `descendents`, tandis que les règles *personnalisées* n'utilisent pas ces notions et expriment tous les autres cas.

Finalement, nous avons introduit un graphe qui modélise les dépendances entre les règles. Dans ce graphe, une règle peut représenter un objectif de haut niveau composé de plusieurs règles vérifiant des objectifs de plus bas niveau.

Chapitre 4

Présentation du logiciel

Actuellement, les opérateurs réseaux ont des besoins et des exigences différents. Par exemple, un opérateur peut souhaiter la redondance des *Route Reflectors* et ce, afin d'éviter des *black holes* [DB06]. Il peut également exiger que les adresses définies sur les liens *point-to-point* possèdent un masque d'une longueur de 31 bits, tandis qu'un autre opérateur peut souhaiter un masque d'une longueur de 30 bits. Dès lors, chaque opérateur possède sa propre vision de la validité de son réseau.

Comme présenté au chapitre 3, la validité de la configuration d'un réseau est définie en termes de règles. Afin que chaque opérateur puisse aisément définir sa conception de la validité, notre logiciel doit lui permettre d'écrire facilement des règles.

Au vu du grand nombre de fonctionnalités différentes et de l'apparition fréquente de nouvelles technologies (p.ex. optiques), une représentation exhaustive de tous les éléments de configuration est très difficile à obtenir et n'est pas viable à long terme. Nous avons donc opté pour une approche flexible permettant d'ajouter facilement de nouveaux éléments de configuration.

Notre logiciel a également pour objectif de générer les configurations concrètes pouvant être déployées dans un réseau réel. Cette étape de génération ne peut être effectuée qu'après la validation de la configuration du réseau. Celle-ci doit pouvoir être traduite en configurations pour n'importe quel type d'équipement (p.ex. Cisco, Juniper, Alcatel, etc.). Les configurations générées peuvent également être destinées à d'autres outils réseaux. Par exemple, si nous voulons vérifier des aspects avancés concernant BGP, *vng* pourrait générer les configurations nécessaires à un outil tel que C-BGP [QU05]. Pour réaliser cet objectif, notre programme doit permettre de définir n'importe quel format de configuration.

La première partie de ce chapitre présentera la structure globale de *vng*. Ensuite, le choix des technologies utilisées sera discuté. Les parties suivantes présenteront comment la configuration d'un réseau et les règles sont représentées dans notre logiciel. Enfin, la dernière partie de ce chapitre présentera la méthode utilisée afin de générer des configurations.

4.1 Structure

Afin de réaliser tous les objectifs fixés, notre logiciel doit être structuré afin d'être un maximum flexible. La figure 4.1 montre sa structure.

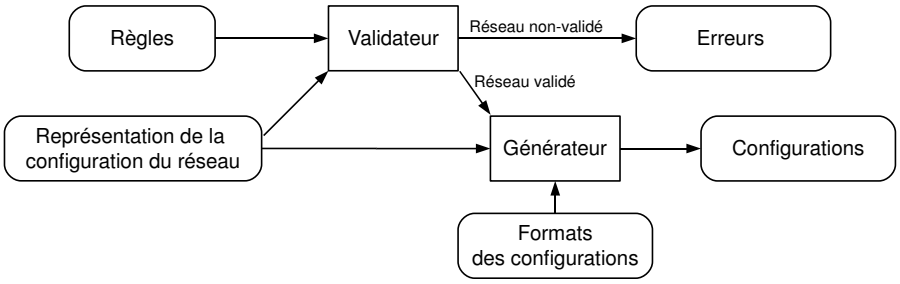

FIG. 4.1 – Structure de *vng*

vng est composé de deux processus principaux : le *validateur* et le *générateur*. Le *validateur* permet de vérifier si la représentation de la configuration du réseau satisfait aux règles définies. Ce processus a besoin de deux entrées : la définition des règles et la représentation. Il affiche des erreurs lorsque les règles ne sont pas respectées. Dans le cas contraire, la configuration du réseau est validée et le *générateur* peut commencer son travail. Celui-ci produit les configurations souhaitées à partir de la représentation du réseau et d'un fichier décrivant le langage de ces configurations.

4.2 Choix technologiques

Dans cette section, nous présenterons et justifierons le choix des différentes technologies utilisées dans chaque partie du logiciel. Premièrement, nous justifierons notre choix du langage de balisage XML afin de représenter la configuration d'un réseau. Ensuite, sur base de cette première décision, nous expliquerons deux alternatives permettant d'appliquer des règles sur cette représentation : l'utilisation d'un langage de programmation (Java) et l'utilisation d'un langage de requêtes appliqué à un fichier XML (XQuery). Enfin, la dernière partie justifiera le choix de l'adoption de XSLT afin de traduire notre représentation de haut niveau en configurations exprimées dans un langage quelconque.

4.2.1 Représentation de la configuration d'un réseau

Comme mentionné dans la section 3.1, nous travaillons avec une structure arborescente afin de représenter la configuration d'un réseau dans sa globalité. Cette structure est représentée dans notre logiciel par un document XML et sera détaillée au chapitre 5. Voici les principaux choix qui ont motivé notre décision :

1. XML constitue le standard *de facto* lorsqu'il s'agit de représenter de l'information hiérarchique ;

2. un document XML est un simple document textuel. Dès lors, il est facilement modifiable à l'aide d'une grande variété d'outils ;
3. un document XML est aisément extensible. C'est d'ailleurs l'une de ses caractéristiques principales ;
4. un document XML est facilement lisible grâce à l'utilisation de balises aux noms généralement explicites ;
5. XML est considéré par les opérateurs comme l'une des solutions à leurs problèmes de gestion de réseaux [SPMF03, Enn06].

4.2.2 Représentation et réalisation des règles

Les règles sont le coeur de ce travail, car elles permettent la validation de la représentation. Les formalisations de celles-ci sont disponibles dans le chapitre 3.

Pour les mêmes raisons qui ont dicté notre choix du langage XML dans la section précédente, nous avons décidé de représenter les règles au sein d'un fichier XML afin de faciliter l'ajout et la modification de celles-ci.

Pour leur implémentation, nous désirons utiliser une technologie qui nous permette de sélectionner le plus facilement possible des éléments de la représentation sur base d'une condition donnée (p.ex. tous les éléments représentant des liens physiques) afin d'exécuter des opérations sur ceux-ci (p.ex. des tests d'égalité, des itérations, etc.). Typiquement, pour atteindre cet objectif, deux approches existent : utiliser un langage de programmation de haut niveau tel que Java[1] et utiliser une technologie liée à XML telle que le langage de requêtes XQuery. Dans la suite de cette section, nous détaillons ces deux approches et expliquons pourquoi nous avons privilégié la seconde.

La première approche consiste à transformer automatiquement la représentation de la configuration en classes Java. Pour réaliser cet objectif, nous pouvons utiliser l'API *JDom* [7] qui permet de représenter de façon automatique chaque élément XML par une classe Java. De plus, l'API *Reflection* permet d'accéder facilement aux informations contenues dans toutes ces classes (p.ex. nom des champs, nom des méthodes, etc.). Ces informations sont utilisées dynamiquement (*i.e.* pendant l'exécution) pour accéder, par exemple, à la valeur d'un champ d'une certaine classe. Dans ce contexte particulier, les règles sont donc exprimées en Java.

La seconde approche consiste à utiliser le langage de requêtes XQuery (section 2.4) basé sur XML. Celui-ci est en tout point comparable à un langage de requêtes appliquées à une base de données relationnelle. Dans notre cas, c'est le document XML qui fait office de base de données. L'avantage de cette démarche réside dans le fait que ce langage est très proche de la formalisation de nos règles (p.ex. possibilité d'utiliser les quantificateurs $\forall$ et $\exists$). De plus, XQuery n'est pas difficile à manipuler. En effet, n'importe quel opérateur possédant des notions de SQL ou étant capable d'écrire des *scripts* simples est apte à écrire des requêtes dans ce langage.

Etant donné la grande similarité entre le langage XQuery et la formalisation des règles, nous avons choisi d'utiliser cette approche. Notons néanmoins qu'un langage de haut niveau pourrait

[1] La plupart de ces langages disposent déjà d'analyseurs syntaxiques compatibles avec le langage XML.

être défini pour permettre une écriture de règles indépendante d'un langage de programmation. Cependant, la réalisation d'un tel langage est une tâche complexe et ne constitue pas le but premier de ce travail.

4.2.3 Génération de configurations

Notre logiciel doit permettre de générer n'importe quel format de configurations à partir de la représentation de haut niveau. À nouveau, la transformation peut être réalisée en utilisant un langage de programmation tel que Java. Cependant, la technologie XSLT (section 2.5) est spécialement prévue à cet effet. Une feuille de style XSLT permet de transformer un fichier XML en n'importe quel type de fichiers comme un fichier texte ou un fichier *pdf*. Pour ces raisons, nous avons décidé de l'utiliser comme technologie de base dans le processus de génération (section 4.5).

4.3 Représentation de la configuration

Pour rappel, nous utilisons un fichier XML pour représenter la configuration du réseau. Celle-ci a été présentée dans la section 3.1 sous la forme d'un arbre. Dans cette section, nous présentons le procédé de traduction de cet arbre en un fichier XML. L'arbre représentant la configuration d'un réseau est composé de *CNodes*. Chaque *CNode* est traduit par un élément XML [PSMY+06].

Le champ `type` d'un *CNode* reçoit un traitement particulier. Sa valeur indique le nom de l'élément XML. Par exemple, un *CNode* de type `interface` est traduit par un élément XML dont le nom est `interface` (*i.e.* `<interface>...</interface>`). Dans le reste de ce chapitre, un élément XML dont le nom est `xyz` sera appelé un élément de type `xyz`. Notons également que nous utiliserons le terme élément pour indiquer un élément XML.

Les champs d'un `CNode` sont traduits par des attributs de l'élément correspondant. Par exemple, le champ `id` d'un `CNode` de type `interface` est traduit par un attribut de même nom dans l'élément correspondant (*i.e.* `<interface id="...">...</interface>`). Une différence de notation existe entre l'arbre présenté dans le chapitre 3 et la représentation en XML. Un routeur est représenté par un *CNode* de type `router` dans l'arbre tandis qu'il est représenté par un élément de type `node` dans la représentation XML.

Par exemple, le résultat de la traduction de l'arbre montré à la figure 3.2 est repris dans le listing 4.1. Cet arbre représente un extrait de la représentation d'un réseau simple composé de quelques routeurs. Il montre, plus particulièrement, le lien entre le routeur $R1$ et le routeur $R2$.

La notion de référence entre *CNodes* présentée dans la section 3.1 est facilement applicable à la représentation XML. En effet, un champ d'un *CNode* qui référence un autre *CNode* est traduit par un attribut d'un élément qui contient la même valeur que l'identificateur de l'élément référencé. Par exemple, dans le listing 4.1, un élément `node` sous `topology/nodes` est identifié de manière unique par l'attribut `id` (identificateur). L'attribut `id` d'un élément `node`

Listing 4.1 – Document XML obtenu à partir de l'arbre représentant un lien entre deux routeurs

```
<domain>
   <topology>
      <nodes>
         <node id="R1">
            <interfaces>
               <interface id="so-0/0/0">
                  <unit number="0">
                     <ip mask="31" type="ipv4">192.0.2.26</ip>
                  </unit>
               </interface>
            </interfaces>
         </node>
         <node id="R2">
                ...
         </node>
      </nodes>
      <links>
         <link id="R1-R2">
            <node id="R1" interface="so-0/0/0" unit="0">
            <node id="R2" interface="so-0/0/1" unit="0">
         </link>
      </links>
   <topology>
</domain>
```

sous `topology/links/link` doit posséder une valeur égale à cet identificateur. Dans cet exemple, `<node id="R1"interface="so-0/0/0"unit="0">` référencie `<node id="R1">`.

L'identificateur d'un élément ainsi qu'une référence à celui-ci peut être défini à l'aide de la notion de clés et de références aux clés dans un schéma XML (section 2.2).

4.4 Représentation des règles

Dans cette section, nous présenterons les différentes techniques utilisées permettant de vérifier les règles formalisées dans le chapitre 3. Ensuite, nous expliquerons comment exprimer et introduire les règles dans notre logiciel en fonction de la technique utilisée.

4.4.1 Techniques de vérification

Comme cela a été expliqué lors de la discussion des choix technologiques, XQuery a été choisi pour vérifier les règles. Cette technologie permet d'exprimer et d'exécuter n'importe quelle requête sur la représentation XML. Par conséquent, elle permet d'exprimer les formalisations des règles de *présence*, de *non-présence*, d'*unicité* et de *symétrie*. De plus, les règles *personnalisées* peuvent être écrites sous la forme de requêtes XQuery.

Comme présenté dans le chapitre 2, un schéma XML spécifie la structure d'un document XML. Nous pouvons donc l'utiliser pour vérifier la structure de notre représentation. Par conséquent, un schéma XML permet de vérifier certaines règles. Notons que toutes les règles vérifiées

par un schéma peuvent l'être par des requêtes XQuery. Cette vérification est, en réalité, une première étape vers la validation de la représentation.

Malgré que le langage XQuery soit un langage très expressif [Kep04], certaines règles nécessitant l'application d'algorithmes sont difficiles à exprimer. Afin de faciliter leur écriture, nous avons décidé d'allier la puissance de XQuery avec celle d'un langage de programmation tel que Java.

En résumé, voici les différentes techniques utilisées pour vérifier les règles :

1. règles vérifiées par le schéma XML. Elles sont appelées *Structural rule*.
2. règles vérifiées par des requêtes XQuery. Elles sont appelées *Query rule*.
3. règles vérifiées par un langage de programmation et plus particulièrement par des classes Java. Elles sont appelées *Language rule*.

Chacune de ces techniques possède ses avantages et ses inconvénients. Dès lors, certaines sont plus appropriées que d'autres pour vérifier un certain type de règles. Tous les détails concernant ces différentes techniques seront donnés dans le chapitre 6.

4.4.2 *Structural rule*

Les *Structural rule* sont représentées par un schéma XML lié au fichier XML représentant le réseau. Le fichier contenant le schéma XML est repris au sein du fichier **schema.xsd**. L'opérateur doit seulement modifier ce fichier pour ajouter une *Structural rule*. Tous les détails concernant les règles représentées dans ce fichier sont disponibles dans la section 6.1.

4.4.3 *Query rule*

Les *Query rule* sont représentées dans le fichier XML `rules.xml`. La technologie XML a été choisie pour représenter les règles afin de faciliter leur définition. De plus, sa structure arborescente ainsi que les balises utilisées facilitent sa compréhension et son utilisation. Une *Query rule* est représentée dans ce fichier par un élément `rule`. Le listing 4.2 montre un exemple d'un tel élément.

Listing 4.2 – Exemple de la structure d'une règle

```
<rule id="ID_DE_LA_REGLE" type="presence">
  <description>Description de la regle.</description>
  <presence>
    <scope>...</scope>
    <descendents>...</descendents>
    <condition>...</condition>
  </presence>
</rule>
```

L'élément `rule` possède l'attribut obligatoire `id` qui identifie une règle de manière unique. Il possède également un attribut indiquant le type de la règle. La valeur prise par cet attribut peut être, par exemple, `presence`, `non-presence`, `uniqueness` ou `custom`. La règle illustrée dans le listing 4.2 est une règle de *présence*. Les paramètres des règles doivent obligatoirement se trouver sous un élément portant le même nom que le type de règles. Dans le listing 4.2, nous voyons que les paramètres nécessaires à une règle de *présence* sont situés sous l'élément `presence`. Ces paramètres sont le SCOPE, la définition de ses `descendents` ainsi qu'une condition. Tous les détails concernant les paramètres des différentes règles sont disponibles dans la section 6.2.

Représentation des scope

Pour rappel, l'équation 4.1 montre la définition du SCOPE. Celle-ci utilise une représentation sous la forme d'un arbre $T = (V, E)$. Tous les détails concernant la définition du SCOPE sont disponibles dans la section 3.2.

$$\text{SCOPE} = \{p | p \in V \ : \ C_{\text{scope}}(T, p)\} \tag{4.1}$$

Dans cette représentation, un SCOPE est composé de *CNodes*. Cette représentation a été traduite dans un fichier XML. Dès lors, un SCOPE ainsi que ses `descendents` sont composés d'éléments XML.

Par exemple, le SCOPE contenant tous les routeurs est défini par l'équation 4.2.

$$\text{ROUTERS} = \{p | p \in V \ : \ p.type = router\} \tag{4.2}$$

Dans notre représentation XML, nous allons traduire ce SCOPE en une requête XQuery renvoyant les éléments XML appropriés. Le listing 4.3 montre la requête XQuery correspondante à l'équation 4.2 (ROUTERS).

Listing 4.3 – Requête XQuery représentant le SCOPE contenant tous les routeurs

```
for $node in /domain/topology/nodes/node
return $node
```

D'après notre étude des règles, nous avons observé que certaines règles sont appliquées sur les mêmes éléments de configuration. Autrement dit, elles utilisent le même SCOPE. Pour cette raison, nous avons autorisé la réutilisation de SCOPE préalablement définis. L'opérateur réseau peut définir autant de SCOPE qu'il le désire. Chaque SCOPE possède un identificateur et peut être utilisé par plusieurs *Query rule*. Les SCOPE sont définis dans le fichier XML `scopes.xml`. Le listing 4.4 montre la définition du SCOPE représentant tous les routeurs du réseau. L'identificateur de celui-ci est `ALL_NODES`.

Une règle doit seulement indiquer l'identificateur d'un SCOPE préalablement défini pour pouvoir l'utiliser. Le listing 4.5 montre un exemple de la représentation en XML d'une règle de *présence* utilisant le SCOPE `ALL_NODES`.

Un listing complet des SCOPE que nous avons définis est disponible en annexe B.3.

Listing 4.4 – Requête XQuery représentant un SCOPE identifié par l'attribut id

```
<scope id="ALL_NODES">
    for $node in /domain/topology/nodes/node
    return $node
</scope>
```

Listing 4.5 – Référence à un SCOPE dans la représentation d'une règle

```
<rule id="ID_DE_LA_REGLE" type="presence">
    <presence>
        <scope>ALL_NODES</scope>
        <set>...</set>
        <condition>...</condition>
    </presence>
</rule>
```

Représentation des descendants

Pour rappel, `descendents` est un ensemble de descendants d'un *CNode* du SCOPE respectant une certaine condition C. Il a été formalisé par l'équation 4.3 (section 3.2). Cette formalisation utilise un arbre $T = (V, E)$.

$$\texttt{descendents}(p) = \{q | q \in descendents(p) \cup p \ : \ C(T, q)\} \tag{4.3}$$

Par exemple, le `descendents` qui contient l'ensemble des *CNodes* représentant les interfaces d'un routeur (*CNode* p) est donné par l'équation 4.4.

$$\texttt{interfaces}(p) = \{q | q \in descendents(p) \cup p \ : \ q.type = interface\} \tag{4.4}$$

Dans la représentation XML, `descendents(p)` est un ensemble d'éléments XML hiérarchiquement inférieurs à l'élément XML p.

De cette manière dans le fichier XML, l'ensemble des interfaces d'un routeur est obtenu en prenant tous les éléments de type `interface` accessibles à partir de l'élément de type `node`. Autrement dit, nous sélectionnons, parmi les descendants de l'élément `node`, ceux dont le nom est `interface`. Ceci est équivalent à l'équation 4.4. Cette sélection des descendants peut facilement être réalisée par une expression XPath (chapitre 2). Dans notre exemple, si `$node` représente un élément `node`, alors l'ensemble de ses interfaces est obtenu par `$x//interface`.

De manière générale, un ensemble `descendents`(x) est obtenu à partir d'un élément XML `$x` en indiquant le chemin à suivre à partir de `$x` dans le fichier XML pour sélectionner ses descendants (*i.e.* `$x/path`).

Dans la formalisation des règles, un ensemble `descendents` est obtenu à partir d'un élément du SCOPE. Ceci reste évidemment valable dans la représentation XML. Dès lors, il est important de bien comprendre quels éléments sont renvoyés par une requête XQuery définissant un SCOPE de manière à exprimer correctement les `descendents` de ceux-ci.

Dépendances entre les règles

Une règle peut dépendre d'autres règles. Ceci permet de définir un graphe de règles. Par exemple, une règle vérifiant que OSPF est correct dépend des différentes règles relatives à OSPF. La règle de plus haut niveau peut, par exemple, représenter le fait que l'entièreté de la configuration du réseau est correcte. Les détails sont disponibles dans la section 3.4.

Chaque dépendance est exprimée à l'aide des balises `<depends>` et `</depends>`. Celles-ci doivent contenir l'identificateur d'une règle et doivent être incluses à l'intérieur des balises `<dependencies>` `</dependencies>`. Les dépendances sont soit de type « et », soit de type « ou ». Une règle avec des dépendances de type « et » est vérifiée si toutes ses dépendances le sont. Une règle avec des dépendances de type « ou » est vérifiée si au moins une de ses dépendances l'est. Ces différentes sortes de dépendances sont exprimées grâce à l'attribut `type` de l'élément `dependencies` qui peut prendre la valeur `and` ou `or`. La règle de plus haut niveau (règle racine) est identifiée par l'attribut `rootid` de l'élément `rule`. La procédure de vérification des règles est récursive. On vérifie d'abord la règle *racine*, ensuite ses dépendances et ainsi de suite.

Listing 4.6 – Extrait du fichier `rules.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<ruledefinition>
  <rule rootid="CORRECT_NETWORK">
    <rule id="CORRECT_NETWORK">
      <dependencies type="and">
        <depends>ID_DE_LA_REGLE</depends>
      </dependencies>
    </rule>
    <rule id="ID_DE_LA_REGLE" type="presence">
      <description>Description de la regle</description>
      <presence>
        <scope>...</scope>
        <descendents>...</descendents>
        <condition>...</condition>
      </presence>
    </rule>
  </rule>
  ...
</ruledefinition>
```

Le listing 4.6 montre un extrait du document XML représentant les règles. Nous pouvons y observer que la règle avec comme identificateur `CORRECT_NETWORK` est la règle de plus haut niveau car l'attribut `rootid` de l'élément XML `rule` vaut `CORRECT_NETWORK`. Cette règle possède des dépendances de type « et » et possède une dépendance : la règle de *présence* `ID_DE_LA_REGLE`.

Afin d'écrire et de représenter plus facilement les règles, nous conseillons de créer, dans un premier temps, des règles sans dépendances. Dans un deuxième temps, il est conseillé de créer des règles de raffinement dont le seul objectif est de regrouper les autres règles précédemment définies. Par exemple, si nous voulons vérifier que les *area* dans OSPF sont correctement configurées, nous pouvons d'abord définir toutes les règles relatives aux *area*. Ensuite, nous pouvons les rassembler sous une règle intitulée `CORRECT_OSPF_AREAS`.

Notons que seules les règles représentées par le fichier `rules.xml` peuvent avoir des dépendances. Les *Structural rule* ne possèdent pas de dépendances entre elles.

Ajouter une règle dans rules.xml

Pour ajouter une règle dans le fichier `rules.xml`, il suffit de rajouter un élément `rule`. Cet élément doit respecter le format détaillé précédemment. De plus, cette règle doit apparaître parmi les dépendances d'une autre règle, sinon la nouvelle règle ne sera jamais utilisée. Pour ajouter une telle dépendance, il suffit de rajouter un élément `depends` sous l'élément `dependencies` de la règle parente avec comme valeur l'identificateur de la nouvelle règle
(p.ex. `<depends>ID_DE_LA_NOUVELLE_REGLE</depends>`).

Si la règle l'exige, un nouveau SCOPE peut être rajouté au fichier `scopes.xml`. À nouveau, il suffit d'ajouter un nouvel élément XML `scope` avec un attribut `id` qui l'identifie. La valeur de cet identificateur doit être unique parmi tous les SCOPE disponibles. Pour rappel, un SCOPE est décrit par une requête XQuery sur la représentation du réseau. Notons qu'aucune vérification n'est appliquée sur ces requêtes. Il est donc capital de bien comprendre ce que ces requêtes renvoient pour que les règles les utilisant puissent fonctionner correctement.

Pour les règles de *présence*, de *non-présence*, d'*unicité* et *personnalisées*, il suffit d'indiquer respectivement `presence`, `non-presence`, `uniqueness` ou `custom` dans l'attribut `type` de l'élément `rule`. Ensuite, un élément portant le même nom que la valeur de l'attribut `type` doit être rajouté sous l'élément `rule`. Ce nouvel élément doit contenir tous les paramètres nécessaires pour le type de la règle correspondante. Par exemple, une règle de *présence* nécessite comme paramètres `scope`, `descendents` et `condition`.

Inclusion de fichiers XML

Pour améliorer la lisibilité du fichier `rules.xml`, celui-ci peut inclure d'autres fichiers XML. De cette manière, les SCOPE sont définis dans le fichier `scopes.xml` qui est inclus dans le fichier `rules.xml`. Par exemple, il est intéressant de définir toutes les règles relatives à OSPF dans le fichier `ospf.xml` et de l'inclure dans `rules.xml`. Le listing 4.7 montre l'utilisation de ce mécanisme d'inclusion pour le fichier `rules.xml`. Tous les détails sont disponibles dans la définition du standard XML [PSMY+06].

4.4.4 *Language rule*

Une *Language rule* est représentée par une classe Java. Celle-ci doit être référencée à partir du fichier `rules.xml`. Par conséquent, ces règles sont également représentées par des élément XML `rule`. L'attribut `type` doit identifier la classe Java qui implémente la règle. Par exemple, l'attribut `type` peut contenir `java:nom_de_ma_regle_java`[2] afin de référencer la règle implémentée par la classe `NomDeMaRegleJava`.

[2]Le choix de cette appellation est de la responsabilité de l'auteur de la règle. Aucune convention particulière ne doit être respectée.

Listing 4.7 – Exemple d'inclusion de fichiers XML

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ruledefinition [
    <!ENTITY ospf SYSTEM "ospf.xml">
    <!ENTITY scope SYSTEM "scopes.xml">
]>
<ruledefinition>
   <rule rootid="CORRECT_NETWORK">
       <rule ...>
       </rule>
       ...
       <!-- Include OSPF file -->
       &ospf;
   </rule>
   <scope>
       <!-- Include scope file -->
       &scope;
   </scope>
</ruledefinition>
```

La figure 4.2 montre le diagramme UML du *package rules* représentant les classes permettant de vérifier les différents types de règles[3]. La classe principale est la classe abstraite `Rule`. Toutes les autres classes de ce *package* étendent (*i.e. extends*) celle-ci. Cette figure montre également quelques exemples de *Language rule* : *Connectivity rule* qui vérifie la connexité du réseau et *OSPF no single link of failure* dont le but est de vérifier dans OSPF que la panne d'un seul lien ne provoquera pas la partition de l'*area* 0. Ces règles sont détaillées dans la section 6.3.

Une classe étendant la classe `Rule` doit suivre le modèle repris dans le listing 4.8. Toutes ces classes doivent définir la méthode `checkRule()` car cette méthode n'est pas implémentée dans la classe abstraite `Rule`. Cette méthode doit renvoyer une des trois constantes définies dans la classe `Rule` et montrées par le listing 4.9. `PASSED` indique que la règle est respectée par la représentation du réseau. `FAILED` indique que la règle n'est pas respectée. `UNKNOWN` indique que la décision finale n'a pas pu être prise car tous les calculs n'ont pas pu être effectués. Typiquement, ce dernier cas se présente lorsqu'une erreur est lancée.

Le constructeur d'une classe étendant la classe `Rule` possède les mêmes paramètres que le constructeur de cette dernière et appelle celui-ci (*i.e.* `super(idRule, ruleChecker)`).

Ce constructeur possède deux paramètres :
- `ruleId` : un *string* identifiant la règle (p.ex. `ID_DE_LA_REGLE`) ;
- `ruleChecker` : une instance de la classe *RulesChecker*[4]. Cette classe permet de vérifier récursivement toutes les règles présentes dans `rules.xml`. Elle contient, entre autres, comme variables d'instances `queryOnConfiguration` et `logger`. Une classe vérifiant une règle a besoin de ces variables pour pouvoir respectivement effectuer des requêtes XQuery sur le fichier XML représentant le réseau et pour *logger* les erreurs afin de les afficher à la console.

[3] L'ensemble des *package* est décrit dans l'annexe D.

[4] La classe *RulesChecker* n'est pas montrée sur la figure 4.2 car elle ne fait pas partie du package *rule*.

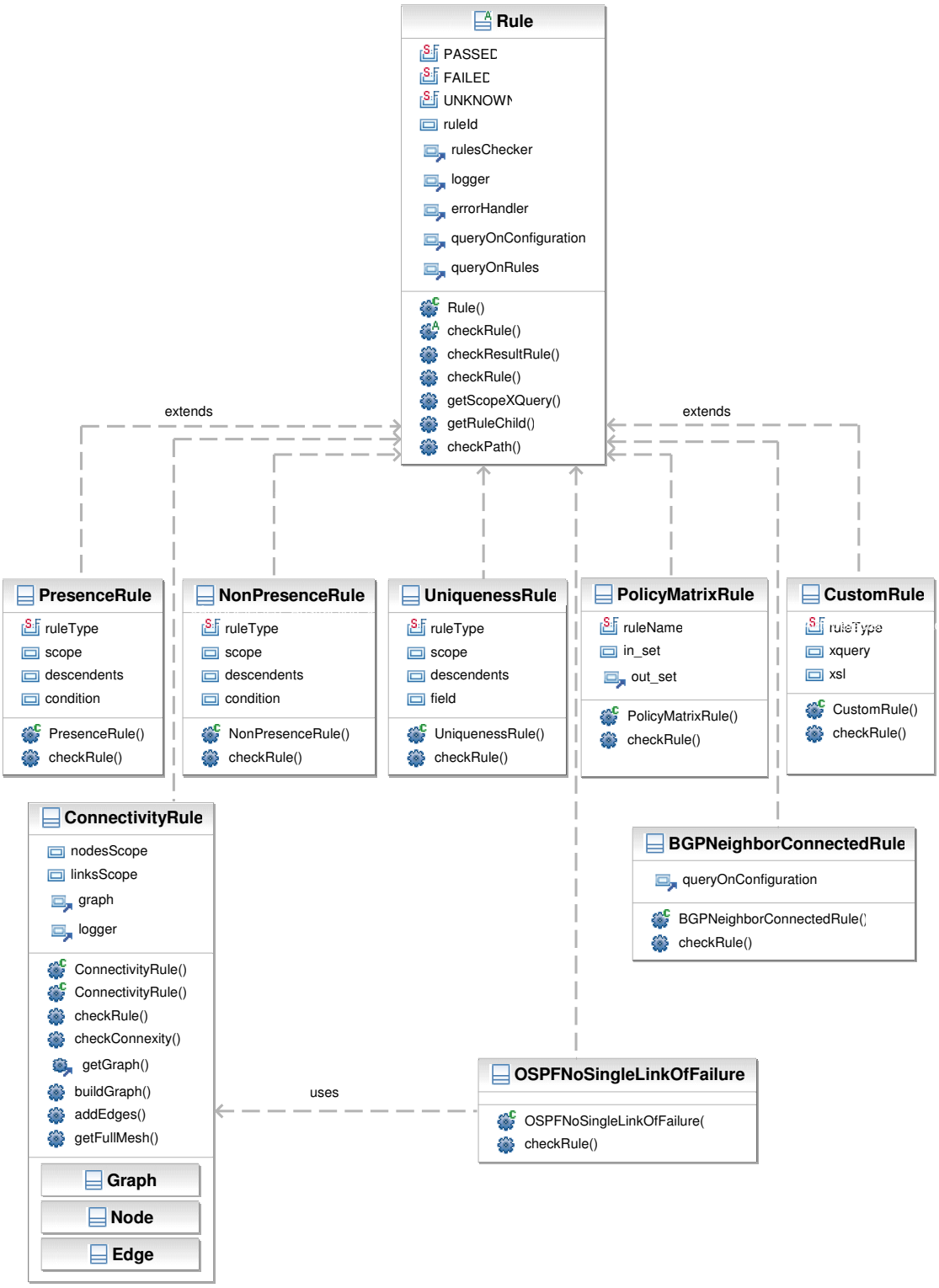

FIG. 4.2 – Diagramme UML du package *rules*

Listing 4.8 – Structure d'une classe Java implémentant une règle

```
package rule;
...
public class NomDeLaRegle extends Rule {

  public NomDeLaRegle(String idRule, RulesChecker ruleChecker) {
    super(idRule, ruleChecker);
  }

  public int checkRule() {
    ...
    return PASSED;
  }
```

Listing 4.9 – Résultats possibles de la méthode CheckRule

```
public static final int PASSED = 0;
public static final int FAILED = 1;
public static final int UNKNOWN = 2;
```

Comme expliqué au début de cette section, une *Language rule* est décrite dans le fichier `rules.xml` par un élément XML `rule` qui possède un attribut `type`. La valeur de cet attribut permet de déterminer quelle classe doit être utilisée pour vérifier la règle. La classe *RulesChecker* responsable de la vérification des règles se base sur cette valeur pour instancier la classe adéquate. Par exemple, un objet de la classe *NomDeLaRegle* doit être créé lorsque le type de la règle est `java:nom_de_la_regle`. Il est donc nécessaire d'ajouter une telle condition dans la méthode `checkRule`[5] de la classe *RulesChecker*[6].

Le listing 4.10 montre un exemple d'instanciation de la classe correspondant au type de la règle. La méthode `checkRule()` est ensuite appelée sur cette nouvelle instance.

4.5 Génération de configurations

Dès que la représentation de la configuration du réseau est validée, les configurations réelles destinées à être déployées sur les équipements du réseau (ou elles peuvent être destinées à d'autres outils réseaux) peuvent être générées. Dans cette section, nous expliquerons comment ceci est réalisé par notre logiciel.

La génération de configurations nécessite deux étapes consécutives. La première consiste à produire des représentations intermédiaires à partir de la représentation de la configuration du réseau. La deuxième consiste à générer les fichiers de configuration souhaités à partir de ces représentations intermédiaires.

[5] Attention, le nom de cette méthode est également `checkRule` mais elle n'a rien en commun avec la méthode `checkRule` des classes filles de la classe `rule`.

[6] Cette étape aurait pu être évitée en utilisant l'*API Reflection*.

Listing 4.10 – Instanciation des classes Java selon le type de la règle

```
/* Get the type of the rule */
String ruleType = queryOnRules.evaluate("for $x in //rule/rule where
$x/@id='"+ruleId+"'return data($x/@type)");

/* All the different types of rule must be checked here ! */
if (ruleType!=null && !ruleType.equals("")) {
  if (ruleType.equals("presence")) {
    PresenceRule rule = new PresenceRule(ruleId, this);
    result = rule.checkRule();
  } else if (ruleType.equals("java:nom_de_la_regle")) {
    NomDeLaRegle rule = new NomDeLaRegle(ruleId, this);
    result = rule.checkRule();
  }   ...
  } else {
    logger.log("Unrecognized rule type : "+ ruleType + ". This rule
     is not fully checked", Logger.ERROR);
    result = UNKNOWN;
  }
}
```

La représentation initiale de la configuration du réseau ne permet pas facilement d'être transformée en fichiers de configurations, car elle possède une structure destinée à avoir une vision de haut niveau du réseau. Dès lors, des représentations intermédiaires sont nécessaires. Dans l'objectif d'obtenir des fichiers de configuration distincts pour chaque équipement du réseau, la représentation de haut niveau est transformée en n représentations intermédiaires. n est le nombre d'équipements (p.ex. routeurs) dans le réseau.

La création d'une représentation intermédiaire nécessite de regrouper toutes les informations relatives à un équipement particulier. Ces informations se situent à différents endroits dans la représentation de la configuration du réseau. Cette opération est réalisée à l'aide de requêtes XQuery avec lesquelles nous sommes déjà familiers. Cette opération aurait pu être réalisée avec une feuille de style XSLT car cette technologie est aussi puissante que XQuery [Kep04]. Nous avons choisi d'effectuer cette transformation avec XQuery car rechercher des informations à différents endroits revient à effectuer des requêtes. L'utilisation de feuilles de style XSLT convient mieux pour transformer un document XML en un document avec un format différent mais respectant une structure similaire [Kep04]. Ces arguments nous ont guidés vers XQuery pour obtenir les représentations intermédiaires. Les résultats de requêtes XQuery sont obtenus sous la forme XML. Dès lors, les représentations intermédiaires sont des fichiers XML.

La deuxième étape consiste à transformer les représentations intermédiaires en configurations dans le langage souhaité. Les représentations intermédiaires sont structurellement proches des futures configurations. Par conséquent, nous pouvons utiliser facilement des feuilles de style XSLT.

La figure 4.3 montre la structure du *générateur* permettant de générer les configurations. Celui-ci est composé de deux processus : le *transformateur* qui transforme la représentation de la configuration du réseau en représentations intermédiaires et le *générateur* proprement dit qui

transforme ces représentations intermédiaires en fichiers de configuration en utilisant des feuilles de style XSLT.

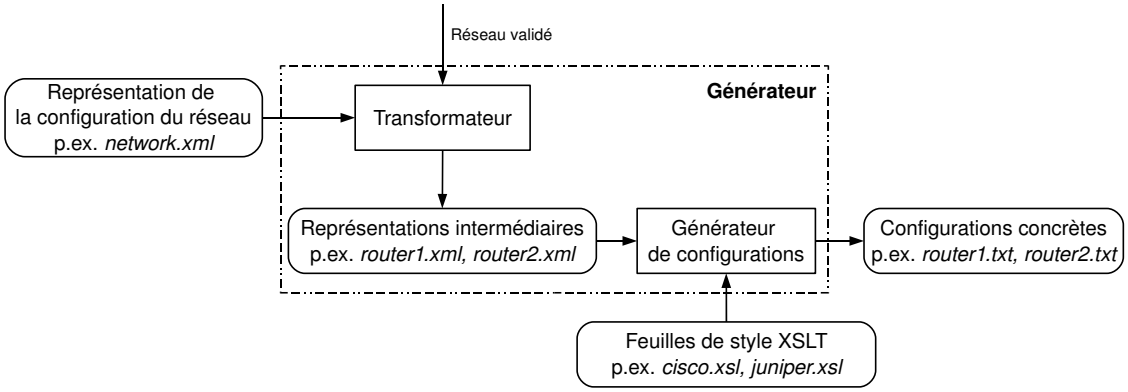

FIG. 4.3 – Structure du générateur de configurations

Tous les détails concernant la transformation en représentations intermédiaires et leur transformation en fichiers de configuration seront donnés au chapitre 7.

4.6 Conclusion

Notre logiciel permet à un opérateur de définir la validité de son réseau à l'aide de règles. Celles-ci sont appliquées sur la représentation de haut niveau de la configuration du réseau. Cette représentation est flexible et évolutive afin de faciliter l'ajout de nouveaux éléments de configuration.

Notre logiciel encourage l'opérateur à suivre une démarche de validation similaire à celle utilisée dans le domaine de l'ingénierie logicielle. En effet, celui-ci peut, dans un premier temps, définir les conditions que le réseau doit respecter en définissant des règles. Dans un deuxième temps, il peut s'atteler à modéliser la configuration de son réseau en s'assurant que celle-ci respecte bien les règles préalablement définies.

Dans ce chapitre, nous avons d'abord détaillé la structure de notre logiciel. Celui-ci est composé de deux processus. En premier lieu, le *validateur* permet de valider la représentation sur base des règles. En deuxième lieu, le *générateur* permet de produire des configurations exprimées dans différents langages sur base de modèles décrivant leur format. Ensuite, nous avons justifié les technologies utilisées par ces deux processus dont, notamment, le choix de XML et de certaines technologies connexes tel que le langage de requêtes XQuery et le langage de transformation XSLT.

Ce chapitre a également décrit la traduction de l'arbre de *CNodes* sous la forme d'un document XML ainsi que la façon dont les différents types de règles sont représentés dans notre logiciel.

Chapitre 5

Représentation d'un réseau en XML

À l'heure actuelle, la plupart des opérateurs configurent encore et toujours leur réseau de façon entièrement manuelle avec tous les désavantages que cela implique (redondance, erreurs fréquentes, etc.) [CGG+04]. Dans ce mémoire, nous proposons l'utilisation d'une représentation abstraite ou de haut niveau de la configuration d'un réseau. Dès lors, l'entièreté de la configuration est modélisée par une seule et même entité.

Pour rencontrer nos objectifs de flexibilité (chapitre 4), notre représentation est décrite sous la forme d'un document XML. Celle-ci est donc facilement modifiable et extensible à souhait (caractéristiques principales de XML). De plus, la structure arborescente d'un document XML s'avère être un atout, car la plupart des configurations réseaux sont organisées de façon hiérarchique. Dans le but de vérifier la conformité d'un document XML par rapport aux différentes spécifications présentées dans ce chapitre, nous utilisons un schéma XML (section 2.2).

Nous débuterons ce chapitre par une présentation des quelques principes utilisés lors de la conception de la représentation. Ensuite, nous détaillerons la structure de celle-ci, c'est-à-dire une description de ses différents constituants et de leur agencement.

5.1 Principes utilisés

Il est évident que l'ensemble des paramètres présents au sein de la configuration de tout équipement doit être modélisé dans notre représentation. De plus, l'ajout d'un élément quelconque doit d'être le plus aisé possible. En effet, il est difficile voire impossible d'obtenir une modélisation exhaustive capable de représenter tous les paramètres (passé, présent et futur) de n'importe quel équipement réseau. Par contre, en facilitant l'extensibilité de la structure, chaque utilisateur pourra la façonner en fonction de ses propres besoins.

Les configurations d'équipements d'un même réseau contiennent un grand nombre de paramètres redondants. Citons, entre autres, les paramètres d'authentification, ceux relatifs aux interfaces interconnectées, une grande partie des paramètres relatifs aux protocoles de routage ou à certaines politiques. Notre structure, afin d'être la plus claire et concise possible, doit

éviter au maximum cette redondance. Cette dernière peut être contournée grâce au principe d'abstraction. En effet, des paramètres identiques présents dans plusieurs configurations différentes peuvent être représentés une seule fois au sein d'une configuration globale. Par exemple, si un MTU doit être configuré sur un lien point à point, plutôt que de placer ce paramètre au niveau des deux interfaces interconnectées, nous le plaçons au niveau du lien. Par conséquent, ce paramètre apparaît une seule fois.

La conséquence principale de cette démarche est la vérification implicite d'un grand nombre d'erreurs potentielles. Ainsi, si un paramètre doit avoir la même valeur sur plusieurs équipements, le fait d'indiquer celle-ci en un seul endroit implique l'absence d'erreur lors de la réplication de la valeur sur les différents équipements, sous l'hypothèse que cette réplication est correcte (chapitre 7).

5.2 Structure de la représentation

Dans cette section, nous détaillerons la structure de la représentation définie à l'aide d'un schéma XML. Pour information, nous nous sommes inspirés de la structure des document XML et des schémas utilisés par le projet TOTEM [12].

Pour rappel, un schéma XML est un document XML, il a donc la caractéristique d'être modifiable et extensible (section 2.2). Par conséquent, la structure décrite ci-après doit être vue comme une base de travail qui peut être particularisée par chaque opérateur en fonction de ses besoins.

Dans le cadre de ce travail, deux protocoles de routages ont été modélisés : le protocole intradomaine OSPF et le protocole interdomaine BGP.

La suite de cette section suivra une approche *top-down*, c'est-à-dire que nous commencerons par décrire la racine du document (*i.e.* l'élément `domain`) pour ensuite décrire les éléments sous-jacents en suivant la hiérarchie.

5.2.1 Description de l'élément racine `domain`

La racine du document est l'élément `domain`. Celui-ci modélise un réseau comme un domaine autonome. Il dispose d'un attribut obligatoire `ASID` (un entier entre 0 et 65535[1]) et d'un optionnel `name` (une chaîne de caractères). Ils représentent respectivement : le numéro d'AS du réseau et son nom. L'élément contient deux sous-éléments obligatoires : `info` et `topology` ainsi que trois sous-éléments optionnels `ospf`, `policies` et `bgp`.

info : donne des informations générales sur le document comme par exemple, l'auteur, la date ou une description générale du fichier.

topology : contient toute l'information à propos de la structure du réseau notamment la description des équipements et des liens qui les connectent.

[1]Nous considérons des numéros d'AS représentés sur deux octets.

ospf : contient l'information relative à la configuration du protocole de routage OSPF comme par exemple, une description des différentes *area*.

policies : contient l'information relative aux politiques interdomaines utilisées comme par exemple, les filtres d'exportation BGP.

bgp : contient l'information relative à la configuration du protocole de routage BGP comme par exemple, la description des différents *peering*.

Le listing 5.1 illustre l'agencement des éléments décrits ci-dessus.

Listing 5.1 – Structure de l'élément `domain`

```
<domain name="OURNET Network" ASID="2611">
  <info>...</info>
  <topology>...</topology>
  <policies>...</policies>
  <ospf>...</ospf>
  <bgp>...</bgp>
</domain>
```

5.2.2 Description de l'élement `info`

Cet élément décrit le document dans sa globalité. Il contient quatre sous-éléments optionnels : `title`, `date`, `author` et `description`. Ceux-ci contiennent tous des chaîne de caractères. Notons que le contenu de cet élément est purement informatif et n'est donc pas utilisé lors de la génération de configuration. Un exemple de contenu est repris dans le listing 5.2.

Listing 5.2 – Structure de l'élément `info`

```
<info>
    <title>Backbone topology of the OURNET network</title>
    <date>2008-06-02</date>
    <author>
        Laurent Vanbever (laurent.vanbever@student.uclouvain.be) - UCL - EPL - INGI
        Gregory Pardoen (gregory.pardoen@student.uclouvain.be) - UCL - EPL - INGI
    </author>
    <description>
        This file summarizes the backbone topology of OURNET (four core routers).
    </description>
</info>
```

5.2.3 Description de l'élément `topology`

Cet élément décrit la structure du réseau. Il est composé de deux sous-éléments obligatoires : `nodes` et `links` et d'un élément optionnel : `as-neighbors`.

nodes : contient une séquence d'éléments `node` qui correspondent aux différents équipements du domaine.

as-neighbors : contient une séquence d'éléments `neighbor` qui représentent les voisins du domaine.

local-addresses : contient une séquence d'éléments `address` qui correspondent aux espaces d'adresses IP détenus par le domaine.

links : décrit les liens physiques du domaine. Deux types de liens sont représentés : d'une part, les liens internes (*i.e.* entre deux noeuds du domaine), d'autre part, les liens externes (*i.e.* entre un noeud du domaine et un noeud d'un domaine voisin).

Un exemple illustrant le contenu de ces éléments est repris dans le listing 5.3.

Listing 5.3 – Structure de l'élément `topology`

```
<topology>
  <nodes>
    <node id="node1">...</node>
    <node id="node2">...</node>
  </nodes>
  <as-neighbors>
    <neighbor id="geant2" as="20965"/>
  </as-neighbors>
  <local-addresses>
        <address type="ipv4" mask="16">A.B.C.D</address>
        <address type="ipv6" mask="64">2001::AAAA</address>
  </local-addresses>
  <links>
    <intra>
      <link id="node1-node2">...</link>
    </intra>
    <inter>
      <link id="node1-geant2">...</link>
    </inter>
  </links>
</topology>
```

Description du sous-élément `node`

Cet élément décrit un équipement vu comme un noeud du graphe représentant la topologie du réseau. Il contient quatre sous-éléments obligatoires : `characteristics`, `rid`, `interfaces` et `static-routes` ainsi qu'un sous-élément optionnel : `status`. Un exemple de contenu est repris dans le listing 5.4. Notons que dans ce travail un élément `node` représente un routeur[2].

Élément `characteristics` Il reprend les caractéristiques générales d'un noeud à savoir : un élément optionnel `description` (chaîne de caractères) et un élément obligatoire `reference`.

L'élément `reference` reprend les différentes propriétés du noeud. Il est composé de trois sous-éléments obligatoires : `constructor` (`cisco` ou `juniper`), `model` (chaîne de caractères) et `os-version` (chaîne de caractères). Ces informations seront utilisées notamment lors de la génération des configurations (chapitre 7).

[2]Cette notion peut être étendue par la suite pour englober, par exemple, les commutateurs ou les pare-feus.

Élément `rid` Il contient le *router-id* (adresse IP) du noeud. Ce paramètre est à la base du bon fonctionnement de certains protocoles de routage. Notons que, la plupart du temps, le *router-id* est équivalent à l'adresse *loopback* de l'équipement.

Élément `interfaces` L'élément `interfaces` contient une séquence d'éléments `interface` qui représentent les interfaces physiques de l'équipement exception faite de l'interface *loopback* qui est logique.

Un élément `interface` décrit une interface d'un noeud. Il est identifié par l'attribut obligatoire `id` (chaîne de caractères). Il peut contenir trois sous-éléments optionnels : `status` (`UP` ou `DOWN`), `description` (chaîne de caractères) et `mtu` (entier entre 256 et 9192). L'élément `interface` doit contenir au moins un sous-élément `unit`.

L'élément `unit` décrit une interface logique. Celui-ci est identifié par l'attribut obligatoire `number` (un entier entre 0 et 65535). Il peut contenir le sous-élément optionnel : `vlan-id` (un entier entre 0 et 4095) qui représente l'étiquette qui sera ajoutée à toutes les trames sortantes de cette unité logique. L'élément contient au moins un élément `ip` qui représente une adresse IPv4 ou une adresse IPv6. Notons que l'élément `unit` est obligatoire dans notre représentation. En pratique, si une unité logique n'a pas d'utilité pour une certaine interface, il suffit de définir les paramètres nécessaires au sein de l'unité par défaut, l'unité 0.

L'élément `ip` décrit une adresse IP. Il dispose de deux attributs obligatoires : `mask` (un entier entre 0 et 128) qui représente la longueur du masque et `type` (`ipv4` ou `ipv6`) qui représente le type de l'adresse IP. Le contenu de l'élément `ip` est une chaîne de caractères respectant le format d'une adresse IPv4 ou d'une adresse IPv6.

Élément `static-routes` L'élément `static-routes` décrit les routes statiques configurées sur un noeud. Il contient au moins un élément `static-route`. Ce dernier contient deux sous-éléments obligatoires : `destination` (une adresse IP accompagnée d'une longueur de masque) et `next-hop` (une adresse IP).

Description du sous-élément `as-neighbors`

Cet élément recense les domaines (AS) voisins au domaine considéré. Il contient au moins un élément `neighbor`. Chacun de ceux-ci contient un attribut obligatoire `id` et deux attributs optionnels : `as` (un entier entre 0 et 65535) et `type` (une chaîne de caractères). Ces attributs représentent respectivement : le numéro d'AS du voisin en question et le type du voisin (p.ex. un point d'interconnexion[3]). Un exemple de contenu possible décrivant deux voisins et deux connecteurs est repris dans le listing 5.5.

[3]Un point d'interconnexion est une infrastructure au travers de laquelle de nombreux réseaux s'interconnectent.

Listing 5.4 – Structure de l'élément `node`

```
<node id="m320.external">
    <characteristics>
        <description>m320 Border router</description>
        <reference>
            <constructor>juniper</constructor>
            <model>M320</model>
            <os-version>7.4R3.4</os-version>
        </reference>
    </characteristics>
    <rid>192.0.2.2</rid>
    <interfaces>
        <interface id="lo0">
            <description>Loopback</description>
            <unit number="0">
                <ip mask="32" type="ipv4">192.0.2.2</ip>
            </unit>
        </interface>
        <interface id="so-0/0/0">
            <description>core:m160.external</description>
            <mtu>9192</mtu>
            <unit number="0">
                <ip mask="30" type="ipv4">192.0.2.17</ip>
                <ip mask="64" type="ipv6">2001:06a8:0000:4000::2</ip>
            </unit>
        </interface>
        <interface id="ge-0/1/0">
            <description>external:CustomerA</description>
            <mtu>9192</mtu>
            <unit number="0">
                <vlan-id>2</vlan-id>
                <ip mask="30" type="ipv4">192.0.2.161</ip>
            </unit>
        </interface>
    </interfaces>
    <static-routes>
        <route>
            <destination mask="24">10.0.100.0</destination>
            <next-hop>192.0.2.74</next-hop>
        </route>
    </static-routes>
</node>
```

Listing 5.5 – Structure de l'élément `as-neighbors`

```
<as-neighbors>
  <neighbor id="CustomerA" as="2111"/>
  <neighbor id="CustomerB" as="2112"/>
  <neighbor id="Connector1" type="connector"/>
  <neighbor id="Connector2" type="connector"/>
</as-neighbors>
```

Description du sous-élément `local-addresses`

Cet élément décrit l'ensemble des espaces d'adresses IPv4 et IPv6 détenus par le domaine. Il contient au moins un élément de type `address` (une adresse IP avec une longueur de masque). Un exemple de contenu reprenant deux préfixes d'adresses est repris dans le listing 5.6.

Listing 5.6 – Structure de l'élément `local-addresses`

```
<local-addresses>
  <address type="ipv4" mask="24">192.0.2.0</address>
  <address type="ipv6" mask="32">2001:468::</address>
</local-addresses>
```

Description du sous-élément `links`

Cet élément décrit un ensemble de liens physiques qui relient, soit plusieurs noeuds du domaine (lien interne), soit un noeud du domaine et un noeud situé hors du domaine (lien externe). Un exemple représentant un lien interne et externe est repris dans le listing 5.7. Cet élément est composé de deux sous-éléments optionnels : `intra` et `inter`.

Listing 5.7 – Structure de l'élément `links`

```
<links>
  <intra>
      <link id="m320.external-m160.external">
            <node id="m320.external" if="so-0/0/0" unit="0" />
            <node id="m160.external" if="so-0/0/0" unit="0" />
            <attributes>
                <speed unit="mbps">100</speed>
                <mtu>1500</mtu>
                <link-mode>full-duplex</link-mode>
            </attributes>
        </link>
       ...
  </intra>
  <inter>
      <link id="m320.external-CustomerA">
        <node id="m320.external" if="ge-0/1/0" unit="0"/>
        <ext-node neighbor-id="CustomerA" />
      </link>
        ...
  </inter>
</links>
```

Élément `intra` Cet élément décrit un lien interne. Il est composé d'au moins un sous-élément `link`. Ce dernier est identifié par l'attribut obligatoire `id` (une chaîne de caractères). Il comprend au moins deux sous-éléments `node` et peut contenir deux-sous éléments optionnels : `description` et `attributes`. Notons que le fait d'avoir plus de deux éléments `node` modélise un réseau local.

Élément `inter` Cet élément décrit un lien externe. Il est composé d'au moins un sous-élément `link`. Celui-ci est identifié par l'attribut obligatoire `id` (une chaîne de caractères). Il est composé de deux sous-éléments obligatoires : `node` et `ext-node` et de deux sous-éléments optionnels : `description` et `attributes`.

Élément `node` Cet élément modélise un noeud du domaine prenant part au lien. Il est identifié par l'attribut obligatoire `id` (une chaîne de caractère). Il dispose de deux autres attributs obligatoires : `if` et `unit`. Ces derniers référencent respectivement l'interface et l'unité logique qui sont impliquées dans le lien.

Élément `ext-node` Cet élément modélise un noeud n'appartenant pas au domaine. Il dispose d'un seul attribut obligatoire : `neighbor-id` qui référence un des éléments `neighbor` (*i.e.* les enfants de l'élément `as-neighbors`).

Élément `attributes` Un élément `attributes` décrit les attributs associés au lien. Par attributs, nous entendons :

- la vitesse du lien représenté par l'élément `speed` ;
- la taille maximale des unités de transmission représentée par l'élément `mtu` ;
- l'encapsulation représentée par l'élément `encapsulation` ;
- le mode de transmission (*full-duplex* ou *half-duplex*) représenté par l'élément *link-mode*.

5.2.4 Description de l'élément `ospf`

Cet élément décrit l'ensembles des paramètres nécessaires à la configuration du protocole *OSPF* [Moy98] pour l'ensemble du domaine modélisé. Il contient quatre éléments optionnels : `reference-bandwidth`, `traffic-engineering`, `route-redistribution` et `intervals`, ainsi qu'un élément obligatoire : `area`. Un exemple reprenant les quatre premiers sous-éléments est repris au sein du listing 5.8.

Listing 5.8 – Structure de l'élément `ospf`

```
<ospf>
    <reference-bandwidth>100g</reference-bandwidth>
    <traffic-engineering />

    <route-redistribution>
        <node id="m160.core.place1">
            <from protocol="rip" metric="2"/>
        </node>
    </route-redistribution>

    <intervals>
        <link id="m320.external-m160.external">
            <hello>10</hello>
            <dead>20</dead>
        </link>
    </intervals>
...
</ospf>
```

Élément `reference-bandwidth` Cet élément décrit la bande passante de référence utilisée par les routeurs lorsque ceux-ci doivent calculer les coûts par défaut associés à leurs interfaces. Le contenu de cet élément est une chaîne de caractères.

Élément `traffic-engineering` Cet élément caractérise le fait que les extensions d'ingénierie de trafic du protocole *OSPF* sont activées ou non. Cet élément est un élément vide, c'est-à-dire un élément qui n'a pas de contenu (chapitre 2).

Élément `route-redistribution` Cet élément représente le fait qu'un routeur peut annoncer au sein d'*OSPF* des routes apprises via d'autres protocoles, ce faisant, le routeur devient un ASBR (*Autonomous System Border Router)*. Cet élement contient au moins un sous-élément `node`. Un élément `node` est identifié par l'attribut obligatoire `id` (une chaîne de caractères) et contient au moins un sous-élément `from` qui représente le protocole redistribué. L'élément `from` dispose d'un attribut obligatoire `protocol` (une chaîne de caractères) qui identifie le protocole redistribué, ainsi que d'un attribut optionnel `metric` (un entier entre 0 et 65535) qui représente la métrique associée aux routes redistribuées.

Élément `intervals` Cet élément contient une description des différents temporisateurs utilisés par le protocole *OSPF*. Il est important de savoir que deux routeurs OSPF ne peuvent former une adjacence si ces paramètres diffèrent [Moy98]. Dans les équipement, ces paramètres sont précisés au niveau des interfaces, une erreur dans un de ceux-ci implique donc l'impossibilité de former une adjacence. Toujours dans un soucis d'abstraction et pour éviter au maximum la redondance, nous avons décidé de placer ces paramètres au niveau d'un lien et non au niveau d'une interface[4].

L'élément `intervals` est constitué d'au moins un sous-élément `link`. Ces derniers sont identifiés par l'attribut obligatoire `id` lesquels référencent un des liens internes présents dans l'élément `topology`. L'élément `link` peut être composé des trois sous-éléments optionnels suivants : `hello`, `dead` et `retransmit`.

Description du sous-élément `area`

Cet élément décrit le contenu d'une *area OSPF*. Un exemple de structure est repris dans le listing 5.9.

L'élément `area` est identifié par l'attribut obligatoire `id` (une adresse IPv4). Il est composé d'un sous-élément obligatoire : `nodes` qui comprend une séquence d'élément `node` et de deux sous-éléments optionnels : `type` (`stub`, `nssa` ou `totallystubby`) qui représente le type de l'*area* et `authentication`.

[4]Placer la description de ces paramètres au niveau du réseau n'est pas un choix judicieux car un opérateur peut très bien, pour des raisons de performances, utiliser des temporisateurs plus agressifs (*i.e.* plus petits) sur certaines parties de son réseau.

Listing 5.9 – Structure de l'élément `area`

```
<area id="0.0.0.0">
  <authentication type="md5">
      <key id="1">keyusedinarea0</key>
  </authentication>
  <nodes>
        <node id="m320.external">
            <interfaces>
                <interface id="lo0" unit="0">
                   <passive />
                </interface>
                <interface id="so-0/0/0" unit="0" />
                ...
            </interfaces>
        </node>
     ...
     </nodes>
 </area>
```

Élément `authentication` L'élément `authentication` décrit l'information d'authentification relative à l'*area*. Il contient au moins un sous-élément `key`. Chacun de ceux-ci dispose de deux attributs obligatoires : `id` (un entier) qui identifie la clé et `type` (une chaîne de caractères) qui contient le type d'authentification utilisée.

Élément `node` Cet élément décrit un noeud présent dans une *area* OSPF. Il est identifié par l'attribut obligatoire `id` (une chaîne de caractères) qui référence un des noeuds présents dans l'élément `topology`. Cet élément est composé de trois sous-éléments optionnels : `default-route`, `no-summaries` et `summarization` et d'un élément obligatoire `interfaces`.

L'élément `default-route` n'a pas de contenu et modélise le fait que le noeud en question annonce une route par défaut au sein de l'*area*. Il dispose d'un attribut optionnel `metric` (un entier) qui représente la métrique associée à cette route.

L'élément `no-summaries` n'a pas de contenu et modélise le fait qu'un noeud n'annonce pas, au sein de l'*area*, des LSA (*Link State Advertisement*) de type 3 (*Summary-LSA*).

L'élément `summarization` représente le fait que le routeur peut agréger des routes. Par exemple, au lieu d'annoncer les deux routes 10.10.3.16/28 et 10.10.3.32/28, il est possible d'annoncer un seul agrégat : 10.10.3.0/24. Cet élément contient au moins un sous-élément `range` (une adresse IP avec un masque).

L'élément `interfaces` décrit les interfaces du noeud sur lesquelles OSPF est activé. Il contient au moins un élément de type `interface`.

L'élément `interface` représente une interface d'un noeud sur laquelle OSPF est activé. Il est identifié par deux attributs obligatoires `id` et `unit`. Il peut également contenir les sous-éléments optionnels suivants : `passive`, `priority` et `metric`. L'élément `passive` n'a pas de contenu modélise le fait qu'une interface fonctionne en mode passif. L'élément `priority` modélise la priorité de l'interface (un entier entre 0 et 255). Celle-ci est utilisée lors du processus d'élection du *Designated Router* et du *Backup Designated Router*, ces concepts sont relatifs au fonctionnement d'OSPF

sur un réseau local. Enfin, l'élément `metric` décrit le coût de l'interface (un entier entre 0 et 65535). Cette valeur sera utilisée lors de l'application de l'algorithme *Shortest Path First*.

5.2.5 Description de l'élément `bgp`

Cet élément contient l'ensemble des paramètres nécessaires à la configuration du protocole *BGP* [RLH06]. Un exemple de contenu possible pour cet élément est repris au sein du listing 5.10.

Notons que cet élément comprend uniquement les informations relatives à la configuration des sessions *BGP*. La configuration des politiques d'importation et d'exportation est traitée dans l'élément `policies` (section 5.2.6).

L'élément `bgp` comprend un sous-élément optionnel : `communities`, ainsi qu'un sous-élément obligatoire : `sessions`.

Listing 5.10 – Structure de l'élément `bgp`

```
<bgp>
  <communities>
    <community id="BLOCK-TO-COMMERCIAL">2002</community>
    <community id="ANNOUNCE-TO-PEER">2003</community>
  </communities>
  <sessions>
    <internal>
      <description>iBGP configurations with 2 RRs</description>
      <authentication-key>ibgpkey</authentication-key>
      <node id="m320.external" next-hop-self="true">
        <cluster id="192.0.2.2">
          <clients>
            <node id="m160.core.place1"/>
            <node id="m160.core.place2"/>
          </clients>
        </cluster>
      </node>
    </internal>
    <external>
      <session id="m320.external-CustomerA">
        <description>Connection to CutomerA</description>
        <authentication-key>keywithCustomerA</authentication-key>
        <multihop>2</multihop>
        <node id="m320.external"/>
        <peer id="CustomerA" ip="192.0.2.162"/>
      </session>
    </external>
  </sessions>
  <sessions-sets>
    <sessions-set id="CUSTOMERS">
      <session id="m320.external-CustomerA"/>
      ...
    </sessions-set>
  </sessions-sets>
</bgp>
```

Élément `communities` Cet élément définit les communautés BGP connues *a priori*[5]. Il est composé d'au moins un sous-élément `community`. Chacun de ceux-ci est identifié par un attribut obligatoire `id` (une chaîne de caractères) et contient un entier représentant la valeur de la communauté. Notons qu'une communauté BGP se présente sous la forme $X : Y$ où X est le numéro de l'AS et Y la valeur de la communauté. Dans notre cas, la valeur X est automatiquement préfixée lors de la phase de génération des configurations réelles pour les équipements du réseau (chapitre 7).

Description de l'élément `sessions`

Cet élément décrit l'ensemble des sessions (internes et externes) du domaine. Il est composé de deux sous-éléments obligatoires : l'élément `internal` qui contient les informations nécessaires à la configuration des sessions *iBGP* et l'élément `external` qui contient les informations relatives aux sessions *eBGP*. Chacune de ceux-ci représente une session *eBGP* particulière entre un routeur du domaine et un routeur qui n'appartient pas au domaine. L'élément `sessions` contient également l'élément optionnel `sessions-sets` qui permet de regrouper des sessions eBGP disposant de propriétés communes (p.ex. les sessions à coûts partagés).

Élément `internal` L'élément `internal` comprend au moins un élément `node` ainsi que les sous-éléments optionnels suivants: `description` (chaîne de caractères) et `authentication-key` (une chaîne de caractères) représentant la clé d'authentification.

L'élément `node` dispose d'un attribut obligatoire `id` qui référence un des noeuds appartenant à l'élément `topology`. Il dispose également d'un attribut optionnel `next-hop-self` (booléen) qui modélise le fait que le routeur modifie l'adresse du *next-hop* par la sienne lorsqu'il réannonce les routes apprises d'un *peer eBGP* sur une session *iBGP*. Un routeur BGP peut être utilisé comme *Route Reflector* (RR) [BCC06] afin de diminuer le nombre de sessions *iBGP* nécessaires au sein du domaine. Dès lors, l'élément `node` peut également contenir le sous-élément optionnel : `cluster` dont l'attribut obligatoire `id` identifie le *Cluster ID* utilisé par BGP. L'élément `cluster` contient au moins un élément `node`. Chacun de ceux-ci est identifié par l'attribut `id` qui référence un noeud appartenant à l'élément `topology`.

Élément `external` Cet élément contient au moins un sous-élément `session` qui représente les différentes sessions *eBGP* entre un routeur du domaine et un routeur d'un autre système autonome. Chaque élément `session` est identifié par l'attribut `id` et peut contenir les sous-éléments suivants :

- `description` (optionnel) qui peut contenir n'importe quelle chaîne de caractère ;
- `authentication-key` (optionnel) qui contient la clef d'authentification éventuelle utilisée sur la connexion ;

[5]Ces communautés sont souvent utilisées afin de permettre un traitement différentié de certaines routes. Par exemple, une politique augmentant ou diminuant la préférence d'une route en fonction de la valeur de la communauté.

- `local-pref` (optionnel) qui contient la préférence locale à associer aux routes apprises sur cette connexion ;
- `multihop` (optionnel) qui contient un entier représentant la distance à laquelle se trouve le *peer* en termes de nombre de sauts ;
- `node` (obligatoire) dont l'attribut `id` identifie le noeud impliqué dans la session *eBGP*. Le noeud référencé se doit d'appartenir à la topologie ;
- `peer` (obligatoire) qui représente le *peer* en question, il dispose de trois attributs obligatoires : `id` qui référence un des éléments présents dans l'élément `as-neighbors`, `ip` et `as` représentant respectivement l'adresse IP du peer et le numéro d'AS distant.

Élément `sessions-sets` Cet élément permet de regrouper un ensemble de sessions possédant des propriétés communes sous une seule dénomination, par exemple, l'ensembles des sessions liant un domaine à ses fournisseurs. Ces ensembles sont utiles pour appliquer des politiques similaires sur plusieurs sessions de façon simultanée. Un exemple de contenu est donné dans le listing 5.10.

L'élément se compose d'au moins un sous-élément `sessions-set`, chacun d'entre eux est identifié par l'attribut `id` et comprend un certain nombre de sous-éléments `session`. Ces derniers disposent d'un attribut obligatoire `id` qui référence un des éléments `session` défini sous l'élément `external`.

5.2.6 Description de l'élément `policies`

Cet élément décrit les politiques de routage interdomaines d'un réseau, c'est-à-dire les politiques BGP. Pour le moment, cet élément traite des politiques d'exportation, c'est-à-dire « quelles routes vont être annoncées et à qui ». Cependant, les mécanismes décrits dans cette section peuvent être étendus afin de traiter les politiques d'importation. Notons qu'au sein de cette partie, nous considérons uniquement les sessions eBGP qui relient un domaine à un autre domaine.

Feamster et Balakrishnan ont montré que la configuration de ces politiques est une tâche compliquée et sujette aux erreurs [FB05]. En effet, la moindre faute dans une politique peut avoir des conséquences importantes. Par exemple, un domaine qui devient, sans le vouloir, un domaine de transit. Ces dysfonctionnements peuvent se produire malgré une configuration correcte de l'entièreté du réseau hormis les politiques.

Afin de simplifier le processus de *design* des politiques, nous présenterons deux abstractions basées sur des représentations matricielles. La première matrice permet une définition des politiques de filtrage propres à l'opérateur. Par exemple, les politiques d'exportation relatives aux différents types de *peering* (client, fournisseur, coût partagé, ect.). La seconde matrice permet de représenter des politiques qui modifient le comportement d'exportation par défaut de l'information de routage en fonction de la communauté utilisée. Ces politiques sont notamment utilisées par les fournisseurs d'accès afin de permettre à leurs clients de modifier le traitement de certaines routes envoyées, par exemple, empêcher l'exportation de certaines de celles-ci aux fournisseurs.

L'élément `policies` représente ces politiques. Il est composé de deux sous-éléments optionnels : `filter-matrix` et `community-matrix` qui correspondent aux matrices ci-dessus.

Description du sous-élément `filter-matrix`

Cet élément représente une matrice qui définit la façon dont les informations de routage sont propagées aux voisins du réseau. Un exemple de matrice, contenant la définition des politiques de *peering* dites « classiques » [GR00], est repris dans le tableau 5.1. Dans un premier temps, nous formaliserons la matrice. Ensuite, nous détaillerons sa représentation en XML.

Les lignes et colonnes de la matrice représentent des ensembles de sessions eBGP (dans l'exemple : SHARED-COSTS, CUSTOMERS, PROVIDERS). Notons que ces ensembles doivent être disjoints pour assurer la cohérence des politiques générées, et ce, afin d'éviter que deux actions différentes soient définies en regard d'une même session.

Les lignes correspondent aux sessions sur lesquelles de l'information de routage est susceptible d'être apprise. Les colonnes correspondent aux sessions sur lesquelles de l'information est, soit annoncée, soit propagée. Concrètement, dans le tableau 5.1, le symbole ✓entre SHARED-COSTS et CUSTOMERS signifie que toutes les routes apprises sur une des sessions appartenant à l'ensemble SHARED-COSTS sont exportées sur l'ensemble des sessions appartenant à CUSTOMERS.

La signification de la matrice peut être expliquée de façon plus formelle de la manière suivante : soit F, la matrice des politiques de filtrage, soit A, B, deux ensembles de sessions eBGP disjoints. Soit F_{AB}, l'élément situé à la ligne A et colonne B. $F_{AB} = \checkmark$ signifie que toutes les routes apprises sur les sessions appartenant à A seront annoncées sur toutes les sessions appartenant à B. Une valeur non-définie pour F_{AB} signifie qu'aucune des routes apprises sur les sessions appartenant à A ne sont annoncées sur une session appartenant à B. En clair, rien n'est annoncé sauf ce qui est explicitement indiqué.

La représentation de la matrice définie dans le tableau 5.1 est reprise dans le listing 5.11. Chaque élément `in-group` représente une ligne de la matrice. Les éléments `out-group` représentent les colonnes. Chacun de ces éléments dispose d'un attribut obligatoire `id` qui se doit de faire référence à un élément de type `sessions-sets` défini dans la section 5.2.5.

	SHARED-COSTS	CUSTOMERS	PROVIDERS
SHARED-COSTS	-	✓	-
CUSTOMERS	✓	-	✓
PROVIDERS	-	✓	-

TAB. 5.1 – Matrice des politiques d'exportation propres à l'opérateur

Listing 5.11 – Structure de l'élément `filter-matrix`

```
<filter-matrix>
  <in-groups>
    <in-group id="SHARED_COSTS">
      <out-groups>
        <out-group id="CUSTOMERS"/>
      </out-groups>
    </in-group>
    <in-group id="CUSTOMERS">
      <out-groups>
        <out-group id="SHARED_COSTS"/>
        <out-group id="PROVIDERS"/>
      </out-groups>
    </in-group>
    <in-group id="PROVIDERS">
      <out-groups>
        <out-group id="CUSTOMERS"/>
      </out-groups>
    </in-group>
  </in-groups>
</filter-matrix>
```

Description du sous-élément `community-matrix`

Cet élément représente une matrice qui décrit les politiques modifiant le comportement d'exportation par défaut de l'information de routage. Nous décrivons ces politiques à l'aide de communautés BGP. La matrice permet de définir des politiques du type : « exporter ou ne pas exporter toutes les routes étiquetées avec une telle communauté sur tel ensemble de sessions ». Un exemple d'une telle matrice est repris dans le tableau 5.2.

Listing 5.12 – Structure de l'élément `community-matrix`

```
<community-matrix>
    <in-communities>
        <in-community id="BLOCK-TO-PROVIDER" type="reject">
            <out-groups>
                <out-group id="PROVIDERS"/>
            </out-groups>
        </in-community>
        <in-community id="ANNOUNCE-TO-SHARED" type="accept">
            <out-groups>
                <out-group id="SHARED_COSTS"/>
            </out-groups>
        </in-community>
    </in-communities>
</community-matrix>
```

Les lignes de la matrice correspondent à l'ensemble des routes BGP étiquetées avec une certaine communauté et transitant dans le domaine (dans l'exemple : ANNOUNCE-TO-SHARED,

BLOCK-TO-PROVIDER). Les colonnes correspondent à des groupes de sessions eBGP. À nouveau afin d'assurer la cohérence des politiques générées, ces groupes de sessions doivent être disjoints.

Intuitivement, les lignes de la matrices correspondent aux routes transitant dans le domaine, tandis que les colonnes correspondent aux ensembles de sessions sur lesquelles ces routes peuvent être annoncées. Ainsi dans le tableau 5.2, le symbole ✗ entre BLOCK-TO-PROVIDER et PROVIDERS signifie qu'aucune route étiquetée avec la communauté BLOCK-TO-PROVIDER ne sera pas annoncée sur une session appartenant à l'ensemble PROVIDERS.

Une formalisation possible est la suivante : soit C, la matrice des communautés, soit R, un ensemble de routes étiquetées avec une certaine communauté donnée et, soit B, un ensemble de sessions eBGP. $C_{RB} = \checkmark$ signifie que toutes les routes appartenant à R sont annoncées sur toutes les sessions appartenant à B. $C_{RB} = \text{✗}$ signifie qu'aucune route appartenant à R n'est annoncée sur un élément de B. Enfin, une valeur indéterminée pour C_{RB} signifie qu'aucune action particulière ne sera prise en regard des routes appartenant à R.

La représentation de la matrice décrite dans le tableau 5.2 est reprise dans le listing 5.12. Les lignes de la matrices correspondent aux éléments `in-community`. Chacun de ces éléments dispose de deux attributs obligatoires : l'attribut `id` (chaîne de caractères) dont la valeur est une référence à un des éléments `community` (section 5.2.5) et l'attribut `type` (`accept` ou `reject`) qui correspond à l'action à appliquer. Les colonnes de la matrice correspondent aux éléments `out-group` (sous-éléments de `out-groups`) dont le seul attribut obligatoire `id` fait référence à un groupe de sessions eBGP (section 5.2.5).

Remarque importante : Lors de l'utilisation simultanée des deux matrices C et F, l'ordre a de l'importance. En effet, soit R, l'ensemble de routes étiquetées avec une certaine communauté, soit A, B, deux ensembles de sessions (où $A \cap B = \emptyset$). Il est possible qu'une des routes appartenant à R soit annoncée sur une des sessions de A et que deux actions différentes soient définies en regard de l'ensemble B (p.ex. $C_{RB} = \text{✗}$ et $F_{AB} = \checkmark$). Par convention, nous avons choisi de donner une priorité plus élevée à l'action définie à l'aide de la matrice C. Dans ce cas-ci, l'action est donc ✗. La raison de ce choix est que la matrice des communautés C est souvent utilisée pour définir des politiques de traitement spécifique. Dès lors, il est logique d'appliquer l'action la plus spécifique en premier lieu.

	SHARED-COSTS	CUSTOMERS	PROVIDERS
BLOCK-TO-PROVIDER	-	-	✗
ANNOUNCE-TO-SHARED	✓	-	-

TAB. 5.2 – Matrice des politiques relatives au comportement des routes au sein du réseau

5.3 Conclusion

Dans ce chapitre, nous avons détaillé la structure de notre représentation de haut niveau. Celle-ci est utilisée afin de modéliser l'entièreté de la configuration d'un réseau au sein d'une seule entité. Cette représentation est présentée sous la forme d'un document XML contraint par un schéma XML.

Cette représentation de haut niveau apporte deux grands avantages. Tout d'abord, la redondance des paramètres peut être évitée. En effet, les paramètres répétés peuvent être représentés une seule fois dans une structure de plus haut niveau. La réplication de ceux-ci est effectuée lors du processus de génération des configurations (chapitre 7). Dès lors, les erreurs provoquées par une duplication manuelle (p.ex. fautes de frappe) sont évitées. Cette démarche correspond au principe d'abstraction bien connu dans le domaine de l'ingénierie logicielle, qui consiste à regrouper des concepts communs à plusieurs entités au sein d'une entité de plus haut niveau. Deuxièmement, cette démarche permet d'être totalement indépendant d'un vendeur particulier. En effet, il n'est pas rare pour un opérateur de devoir travailler dans des environnements hétérogènes (*i.e.* dans lesquels des équipements de plusieurs vendeurs sont présents), ce qui signifie manipuler des langages et donc des conventions de configuration différents. Notre représentation a l'avantage d'être générique. Par conséquent, l'opérateur peut se concentrer uniquement sur sa configuration en tant que telle et non plus sur des détails d'implémentation propres à chaque vendeur.

Enfin, cette représentation permet l'utilisation de structures abstraites qui facilite la tâche de configuration. Deux abstractions matricielles, relatives au protocole BGP, ont été proposées. La première décrit les politiques de transit d'un domaine, c'est-à-dire vers quels domaines un réseau exporte les routes apprises issues d'autres domaines. La seconde exprime des politiques qui modifient le comportement d'exportation par défaut des routes étiquetées. Par exemple, refuser l'exportation de tel ensemble de routes (identifiée par une communauté BGP donnée). Ces deux matrices permettent d'exprimer plus aisément des politiques d'exportation de routes BGP. Notons que des mécanismes similaires sont réalisables pour l'importation des routes BGP au sein du domaine.

Chapitre 6

Techniques de vérification utilisées

Dans le chapitre 3, nous avons identifié cinq types de règles : les règles de *présence*, de *non-présence*, d'*unicité*, de *symétrie* et les règles *personnalisées*. Dans le chapitre 4, nous avons déterminé quelles sont les technologies utilisées par notre logiciel. Ces technologies ont permis d'obtenir différentes techniques de vérification des règles.

Pour rappel, voici les différentes techniques utilisées :

1. règles vérifiées par le schéma XML. Celui-ci décrit la structure d'un document XML. Elles sont appelées *Structural rule*.
2. règles vérifiées par des requêtes XQuery. Ces requêtes sont comparables à celles utilisées dans le domaine des bases de données relationnelles. Elles sont appelées *Query rule*.
3. règles vérifiées par un langage de programmation, tel que Java. Elles sont appelées *Language rule*.

Ce chapitre présentera comment les *Structural rule*, *Query rule* et *Language rule* implémentent les règles de *présence*, de *non-présence*, d'*unicité*, de *symétrie* ainsi que les règles *personnalisées*. Nous présenterons également la technique de vérification conseillée en fonction du type des règles.

Un listing complet des règles pouvant être vérifiées par notre logiciel est disponible en annexe B.

6.1 *Structural rule*

Ces règles sont vérifiées à l'aide d'un schéma XML associé à notre représentation de la configuration d'un réseau. Un schéma XML spécifie la structure d'un fichier XML. Tous les détails concernant les schémas XML sont disponibles dans la section 2.2. Les sections suivantes montreront comment certaines règles de *présence*, *non-présence*, *unicité* et *symétrie* peuvent être exprimées par un tel schéma.

6.1.1 Règles de *présence* et de *non-présence*

Certaines règles de présence peuvent être vérifiées par le schéma XML. Ces règles doivent être assez simples dans le sens où, elles ne nécessitent ni la notion de SCOPE, ni celle de ses `descendents`. De plus, les conditions à vérifier doivent pouvoir être exprimées par le schéma.

Pour rappel, un schéma XML permet de contraindre la cardinalité des éléments XML. Ceci permet de vérifier, par exemple, qu'un lien relie au moins deux routeurs. Le listing 6.1 montre un extrait du schéma XML définissant la cardinalité minimale de l'élément XML `node` à deux (*i.e.* `minOccurs="2"`).

Listing 6.1 – *Structural rule* : Un lien relie au moins deux routeurs

```
<xs:complexType name="intra-link">
    <xs:sequence>
        <xs:element name="node" minOccurs="2" maxOccurs="unbounded">
           ...
        </xs:element>
    </xs:sequence>
</xs:complexType>
```

De plus, un schéma XML permet également de définir le type ou le format de la valeur d'un élément ou d'un attribut. Par exemple, dans le listing 6.2, le type de la valeur de l'élément `description` est `xs:string` (*i.e* une chaîne de caractères), tandis que la valeur d'un *router id* est de type `InetAddressIPv4`. Ce dernier correspond à une expression régulière représentant une adresse IPv4.

Listing 6.2 – *Structural rule* : Vérification du type d'un élément

```
<xs:element name="description" type="xs:string" minOccurs="0" maxOccurs="1"/>
<xs:element name="rid" type="InetAddressIPv4" minOccurs="1" maxOccurs="1"/>
```

Le schéma permet de définir de nouveaux types de données. Par exemple, la valeur d'un MTU doit être une valeur entière comprise entre 256 et 9192. Cette règle peut être définie en définissant un nouveau type de données appelé `mtuNumber`. Le listing 6.3 montre la définition d'un élément `mtu` dont la valeur est de type `mtuNumber` ainsi que la définition de ce nouveau type.

Listing 6.3 – *Structural rule* : MTU entre 256 et 9192

```
<xs:element name="mtu" type="mtuNumber" minOccurs="0" maxOccurs="1"/>
...
<xs:simpleType name="mtuNumber">
    <xs:restriction base="xs:integer">
        <xs:minInclusive value="256" />
        <xs:maxInclusive value="9192" />
    </xs:restriction>
</xs:simpleType>
```

Pour de plus amples renseignements concernant les schémas XML, nous renvoyons le lecteur aux spécifications des schémas [FW04, BMTM04, BM04] ainsi qu'au chapitre 2.

6.1.2 Règles d'*unicité*

Comme détaillé au sein du chapitre 2 et dans la section 4.3, un schéma XML permet de définir des clés. Une clé peut être définie sur un élément ou sur l'attribut d'un élément. Elle permet de garantir l'unicité de sa valeur à travers l'entièreté du document XML. Certaines règles d'unicité peuvent donc être vérifiées par l'utilisation de clés.

Par exemple, la vérification de l'unicité du nom des routeurs du réseau est réalisée en définissant une clé sur le nom des routeurs. Plus précisément, le nom d'un routeur est représenté par l'attribut `id` d'un élément XML de type `node`. Grâce à la définition d'une clé sur cet attribut, nous garantissons l'unicité de sa valeur. La définition de cette clé est reprise dans le listing 6.4.

Listing 6.4 – *Structural Rule* : Unicité du nom des routeurs

```
<xs:key name="nodeIdKey">
    <xs:selector xpath="topology/nodes/node"/>
    <xs:field xpath="@id"/>
</xs:key>
```

Notons cependant que, actuellement, les schémas XML ne permettent pas de définir des clés hiérarchiques. Celles-ci sont des clés garantissant l'unicité sur des parties du fichier XML. Par exemple, l'unicité du nom des interfaces des routeurs nécessite de vérifier l'unicité sur chaque ensemble d'interfaces de chaque routeur. Actuellement, une clé dans un schéma XML porte toujours sur l'entièreté du fichier XML. Dans notre exemple, une telle clé obligerait que toutes les interfaces de tous les routeurs du réseau possède un nom différent. Ce qui n'est pas le comportement attendu car il est clair que plusieurs routeurs peuvent avoir une interface *so-0/0/0* ! Cette règle ne peut donc pas être vérifiée par un schéma XML[1]. Néanmoins, celle-ci peut être vérifiée par une *Query rule* en utilisant les notions de SCOPE et de ses `descendents`.

6.1.3 Règles de *symétrie*

La majorité des règles de *symétrie* peuvent être vérifiées implicitement par le schéma XML. En effet, notre représentation de haut niveau garantit la symétrie de l'information en représentant les paramètres redondants par des entités plus abstraites.

Les règles de *symétrie* d'*égalité simple* peuvent être vérifiées de cette manière. Pour rappel, celles-ci vérifient qu'un ensemble d'éléments de configuration possède la même valeur (section 3.3.4). Par exemple, la règle qui vérifie que deux interfaces interconnectées possèdent le

[1]En réalité, il est possible de vérifier ce genre de règles à l'aide d'un schéma XML en rajoutant au niveau d'une interface l'identificateur du noeud auquel elle appartient. De cette manière nous garantissons l'unicité du couple (*id du noeud - id de l'interface*). Cette approche n'a pas été choisie car elle est trop contraignante pour la représentation.

même MTU est une règle de *symétrie* d'*égalité simple*. Celle-ci est vérifiée, d'une part, en plaçant le MTU au niveau du lien et non au niveau des interfaces, et d'autre part, en garantissant que le processus de génération des configurations duplique correctement le MTU sur les interfaces correspondantes.

Cette approche est également utilisée pour vérifier les règles de *symétrie* d'*égalité croisée*. Pour rappel, ce type de règle vérifie la symétrie entre deux éléments différents de configuration (section 3.3.4). Par exemple, la règle qui vérifie le *full mesh* des sessions iBGP entre un ensemble de routeurs est une règle de *symétrie* d'*égalité croisée*. En effet, la valeur de l'élément `neighbor` doit contenir la valeur de l'élément `rid` du routeur avec lequel la session iBGP doit être établie. Cette règle utilise donc une forme de symétrie entre deux éléments de configurations : l'élément `neighbor` et l'élément `rid`. Cette règle est vérifiée, d'une part, en précisant, au sein d'un même élément, l'ensemble des routeurs qui doivent faire partie du *full mesh* iBGP, et d'autre part, en garantissant que les configurations générées respectent ce *full mesh*.

Les règles de *symétrie* sont vérifiées en utilisant deux processus de notre logiciel. Premièrement, une *Structural rule* définit la façon dont l'information est introduite dans la représentation de haut niveau. Deuxièmement, le processus de génération des configurations utilise cette information afin de produire des configurations respectant les règles de *symétrie*. Par conséquent, nous dirons que celles-ci sont respectées par construction.

6.2 *Query rule*

Les *Query rule* sont des règles vérifiées à l'aide de requêtes XQuery sur un document XML. Plus de détails concernant le langage XQuery sont disponibles dans le chapitre 2. L'objectif de cette section sera de transformer les formalisations des différentes règles expliquées précédemment dans le langage XQuery.

Afin de corriger facilement les erreurs de configuration, nous aimerions connaître les éléments de configuration qui ne respectent pas les règles définies. Un simple message disant que le réseau ne respecte pas certaines règles est intéressant mais n'est pas suffisant pour permettre à un opérateur de corriger facilement l'erreur. Dès lors, nous avons décidé d'orienter l'écriture des règles et, plus particulièrement des XQuery, afin d'obtenir les éléments fautifs, ou les contre-exemples, ne respectant pas celles-ci. Cela signifie également que si la requête ne renvoie aucun résultat, alors la règle est vérifiée.

6.2.1 Règles de *présence*

Comme décrit dans la section 3.3.1, une règle de *présence* utilise les notions de SCOPE et de ses `descendents`. La formule générale est rappelée par l'équation 6.1.

$$\forall x \in \text{SCOPE}\ \exists y \in \texttt{descendents}(x)\ :\ C_{\text{presence}}(T, y) \tag{6.1}$$

Cette équation montre qu'une règle de *présence* nécessite trois paramètres : la définition du SCOPE, de ses `descendents` et de la condition de présence C_{presence}.

Comme expliqué ci-dessus, une *Query rule* cherche les contre-exemples de la règle. Nous voulons donc obtenir les éléments x du SCOPE pour lesquels il n'existe aucun élément de `descendents`(x) qui satisfont la condition de présence C_{presence}. Ceci est obtenu en prenant la négation autour de `descendents` et de la condition C_{presence} de l'équation 6.1 :

$$\forall x \in \text{SCOPE}\ \neg(\exists y \in \texttt{descendents}(x)\ :\ C_{\text{presence}}(T, y)) \tag{6.2}$$

$$\Leftrightarrow \forall x \in \text{SCOPE}\ \forall y \in \texttt{descendents}(x)\ :\ \neg C_{\text{presence}}(T, y) \tag{6.3}$$

Cette dernière équation permet d'obtenir la XQuery reprise dans le listing 6.5.

Listing 6.5 – Requête XQuery utilisée par la règle de *présence*

```
for $x in SCOPE
where every $y in $x/DESCENDENTS satisfies not($y[CONDITION])
return $x
```

Afin de bien comprendre les requêtes XQuery générées à partir de cette dernière, nous allons l'illustrer par un exemple. Celui-ci est basé sur le listing 6.6 montrant la représentation partielle en XML d'un routeur . Cet exemple vérifie que tous les routeurs possèdent au moins une interface *loopback* et il est formalisé par l'équation 6.4.

$$\forall x \in \text{ROUTERS}\ \exists y \in \texttt{interfaces}(x)\ : y.id = loopback \tag{6.4}$$

Listing 6.6 – Représentation d'un routeur d'*Abilene*

```
<node id="R1">
    <characteristics>
        <reference>
            <constructor>juniper</constructor>
            <model>t640</model>
            <os-version>8.4</os-version>
            <comment>JUNOS Base OS boot [8.4R3.3]</comment>
        </reference>
    </characteristics>
    <rid>198.0.2.200</rid>
    <interfaces>
        <interface id="lo0">
            <unit number="0">
                <ip type="ipv4" mask="32">198.0.2.200</ip>
                <ip type="ipv6" mask="128">2001:468:16::1</ip>
            </unit>
        </interface>
    </interfaces>
</nodes>
```

Cette règle peut s'écrire en indiquant ce que représente le SCOPE, les `descendents` et la condition C_{presence}. Le SCOPE de cette règle contient les éléments de type `node` qui représentent tous les routeurs du réseau. L'ensemble des interfaces d'un routeur est obtenu en prenant tous les éléments XML de type `interface` accessible à partir de l'élément de type `node`. Autrement

dit, nous sélectionnons, parmi les descendants de l'élément de type `node`, ceux dont le nom est `interface`. Plus précisément, les éléments de type `interface` sont regroupés sous un élément de type `interfaces`. L'ensemble des éléments représentant les interfaces d'un routeur sont obtenus en indiquant, dans la structure XML, le chemin à partir d'un élément de type `node`. En d'autres mots, si `$node` représente un élément XML de type `node`, alors l'ensemble de ses interfaces est obtenu par `$x/interfaces/interface`. Ceci est illustré dans la définition de `descendents` dans le listing 6.7.

Comme nous pouvons le voir dans le listing 6.6, une interface *loopback* est représentée par un élément XML de type `interface` dont l'attribut `id` commence avec `lo`. La condition utilise la fonction `substring`[2] permettant de récupérer une sous-chaîne d'une chaîne de caractères, et ce afin d'obtenir les deux premiers caractères de l'attribut `id`.

Listing 6.7 – *Query rule* : Tous les routeurs ont une interface *loopback*

```
<rule id="LOOPBACK_INTERFACE_ON_EACH_NODE" type="presence">
    <presence>
        <scope>ALL_NODES</scope>
        <descendents>interfaces/interface</descendents>
        <condition>substring(@id,1,2)='lo'</condition>
    </presence>
</rule>
```

La requête générée est donnée dans le listing 6.8. Nous constatons que le SCOPE défini par `ALL_NODES` a été remplacé par la requête XQuery correspondante (*i.e.* `for $node in /domain/topology/...`). De même, nous remarquons que `interfaces`(x) de l'équation 6.4 est traduit par `$x/interfaces/interface`.

Listing 6.8 – Requête XQuery : Tous les routeurs ont une interface *loopback*

```
for $x in (for $node in /domain/topology/nodes/node return $node)
where every $y in $x/interfaces/interface satisfies not($y[substring(@id,1,2)='lo'])
return $x
```

6.2.2 Règles de *non-présence*

La règle de *non-présence* a été formalisée par l'équation 6.5.

$$\forall x \in \text{SCOPE}\, \forall y \in \texttt{descendents}(x) \ : \ \neg C_{\text{non-presence}}(T, y) \tag{6.5}$$

Comme toutes les *Query rule*, la requête XQuery permettant de vérifier une règle de *non-présence* cherche ses contre-exemples. De manière sembable à la démarche suivie pour la règle de *présence*, nous prenons la négation autour de `descendents` et de la condition $C_{\text{non-presence}}$ de

[2]Cette fonction fait partie des fonctions fournies par le langage XQuery.

l'équation 6.5 :

$$\forall x \in \text{SCOPE}\ \neg(\forall y \in \texttt{descendents}(x)\ :\ \neg C_{\text{non-presence}}(T, y)) \tag{6.6}$$

$$\Leftrightarrow \forall x \in \text{SCOPE}\ \exists y \in \texttt{descendents}(x)\ :\ C_{\text{non-presence}}(T, y) \tag{6.7}$$

Cette dernière indique que la requête recherche les éléments x du SCOPE pour lesquels il existe un élément de `descendents`(x) qui respecte la condition $C_{\text{non-presence}}$. Le listing 6.9 montre la XQuery ainsi obtenue.

Listing 6.9 – Requête XQuery générée par la règle de *non-présence*

```
for $x in SCOPE
where some $y in $x/DESCENDENTS satisfies $y[CONDITION]
return $x
```

6.2.3 Règles d'*unicité*

L'équation 6.8 est un rappel de la formalisation de la règle d'*unicité*.

$$\forall x \in \text{SCOPE}\ \forall y \in \texttt{descendents}(x)\ :\ \neg(\exists z_{\neq y} \in \texttt{descendents}(x) : y.field = z.field) \tag{6.8}$$

Cette équation montre qu'une règle d'*unicité* nécessite trois paramètres pour être entièrement définie : le SCOPE, ses `descendents` et le champ à tester (*i.e.* $field$ dans l'équation).

Cette formalisation doit être transformée de manière à rechercher les contre-exemples qui ne respectent pas la règle d'*unicité*. La requête XQuery cherchant les contre-exemples peut être écrite par le listing 6.10. Celle-ci renvoie les éléments possédant la même valeur du champ $field$ pour chaque ensemble `descendents`(x). Pour information, la fonction *count* permet d'obtenir la cardinalité d'un ensemble d'éléments.

Listing 6.10 – Requête XQuery générée par la règle d'*unicité*

```
for $x in SCOPE,
  $item1 in $x/DESCENDENTS
let $item2 := $x/DESCENDENTS[FIELD=$item1/FIELD]
where count($item2) > 1
return $item1
```

Pour illustrer cette requête, prenons, par exemple, la règle vérifiant l'unicité du nom des interfaces. Celle-ci a été formalisée par l'équation 6.9.

$$\forall x \in \text{ROUTERS}\ \forall y \in \texttt{interfaces}(x)\ :\ \neg(\exists z_{\neq y} \in \texttt{interfaces}(x) : y.id = z.id) \tag{6.9}$$

Cette règle est décrite en XML par le listing 6.11. Le $field$ est `@id` car `id` est l'attribut représentant le nom de l'interface. En XML, le symbole `@nom` représente l'attribut `nom` de l'élément [PSMY+06].

Le listing 6.12 montre la requête XQuery obtenue à partir de cet exemple.

Listing 6.11 – *Query rule* : Unicité du nom des interfaces des routeurs

```
<rule id="UNIQUENESS_INTERFACE_ID" type="uniqueness">
    <uniqueness>
        <scope>ALL_NODES</scope>
        <descendents>interfaces/interface</descendents>
        <field>@id</field>
    </uniqueness>
</rule>
```

Listing 6.12 – XQuery : Unicité du nom des interfaces des routeurs

```
for $x in (for $node in /domain/topology/nodes/node return $node),
  $item1 in $x/interfaces/interface
let $item2 := $x/interfaces/interface[@id=$item1/@id]
where count($item2) > 1
return $item1
```

6.2.4 Règles *personnalisées*

Comme expliqué dans le chapitre 3, les règles *personnalisées* représentent toutes les règles qui ne peuvent pas être exprimées par les formalisations des règles de *présence*, de *non-présence*, d'*unicité* ou de *symétrie*. Une règle *personnalisée* implémentée par une *Structural rule* permet de définir sa propre requête XQuery sur la représentation de haut niveau.

Cette règle nécessite deux paramètres : le paramètre `xquery` qui permet de définir sa propre requête XQuery et le paramètre `xsl` qui permet de mettre en forme les résultats de la requête (*i.e.* pour afficher correctement les contre-exemples). Le paramètre `xsl` doit contenir une feuille de style XSLT. Tous les détails concernant les feuilles de style XSLT sont disponibles dans la section 2.5.

La requête XQuery doit être construite en gardant à l'esprit qu'elle ne doit rien retourner si la règle est respectée et qu'elle doit renvoyer les contre-exemples dans le cas contraire.

Par exemple, la règle décrite par le listing 6.13 vérifie que toutes les *area* OSPF sont directement connectées à l'*area* 0 (*backbone*). Cette requête cherche les *area* qui ne possèdent pas un routeur bordure connecté au *backbone*. Un routeur bordure est un routeur faisant partie de plusieurs *area*.

Plusieurs éléments sont importants à observer dans cet exemple. Premièrement, nous remarquons la présence de `<![CDATA[` et de `]]>` englobant la requête XQuery et situé sous l'élément `xquery`. Ces balises particulières sont nécessaires pour indiquer à l'analyseur syntaxique (*parser*) de XML de ne pas considérer les balises à l'intérieur de celles-ci. En effet, dans l'exemple montré par le listing 6.13, nous ne voulons pas que le *parser* interprète les balises `<result>` et `<area>` car celles-ci font partie de la requête XQuery. Ces dernières seront utilisées par le compilateur de XQuery pour structurer les résultats de la requête. Ceci est également valable pour le contenu du paramètre `xsl`.

Listing 6.13 – *Query rule* : Les areas OSPF sont directement connectés à l'area 0

```
<rule id="AREAS_CONNECTED_TO_BACKBONE_AREA" type="custom">
    <custom>
        <xquery><![CDATA[
            for $area in /domain/ospf/areas/area[@id!="0.0.0.0"]
            let $backbone_nodes := /domain/ospf/areas/area[@id="0.0.0.0"]/nodes/node
            where not(exists($backbone_nodes[@id=$area/nodes/node/@id]))
            return <result><area id="{$area/@id}"/></result>]]>
        </xquery>
        <xsl><![CDATA[
            <xsl:template match='result'>
                Area <xsl:value-of select='area/@id'/> is not connected to the backbone !
            </xsl:template>]]>
        </xsl>
    </custom>
</rule>
```

Deuxièmement, il est conseillé de placer les résultats d'une XQuery (*i.e.* après `return`) à l'intérieur des balises `<result>...</result>`. Ceci permet de faciliter la mise en forme des contre-exemples grâce à l'utilisation de `<xsl:template match='result'>` dans la feuille de style. L'utilisation de ce genre de balise est décrite dans la section 2.5 et dans la spécifications de XSLT [Cla99]. Il est important que l'auteur de la règle *personnalisée* effectue correctement le lien entre les résultats renvoyés par la requête XQuery et leur mise en forme. En effet, aucune vérification n'est appliquée sur ces paramètres. L'auteur de la règle est donc responsable de garantir leur cohérence. Une mauvaise requête XQuery, une mauvaise feuille de style ou une mauvaise cohérence entre les deux font apparaître inévitablement des erreurs.

Notons finalement que la feuille de style indiquée sous l'élément `xsl` ne contient pas d'en-tête spécifique aux feuilles de style. En effet, une feuille de style *XSLT* doit, pour être correctement formée, commencer par un élément `<xsl:stylesheet>` et doit se terminer par `</xsl:stylesheet>`. Ces éléments ne doivent pas être indiqués car ils sont automatiquement ajoutés par notre logiciel.

6.3 *Language rule*

Certaines règles nécessitent des calculs avancés ou l'application d'un algorithme. Celles-ci ne peuvent pas, en général, être exprimées facilement sous la forme de requêtes XQuery. Par exemple, la vérification de la connexité du réseau nécessite l'application d'un algorithme et est difficilement exprimable en XQuery. C'est ici que la puissance de Java intervient. Lorsqu'une règle est trop complexe, nous pouvons définir une classe Java qui la vérifie. Une telle règle est, dans la plupart des cas, une règle *personnalisée*. En effet, il est toujours possible d'écrire une classe Java pour vérifier une règle de présence, mais cela n'est pas très judicieux car une *Query rule* de type *présence* est mieux adaptée. Le choix de la technique la plus appropriée est discuté dans la section 6.4.

Comme expliqué dans la section 4.4.4, une règle représentée par une classe Java peut effectuer elle-même des requêtes XQuery sur le fichier XML grâce à la variable `queryOnConfiguration` de la

classe *Rule*. Cette variable contient une référence vers une instance de la classe `XQueryEvaluator` qui permet d'exécuter une requête XQuery sur un fichier XML donné. Une *Language rule* est donc au moins aussi expressive qu'une *Query rule*. La grande force d'une règle écrite en Java est de pouvoir utiliser la puissance de Java sur les résultats de requêtes XQuery.

Comme cela a été expliqué dans la section 4.4.4, une *Language rule* est écrite à l'aide d'une classe Java qui étend (*i.e. extends*) la classe *Rule*. Le constructeur de cette classe nécessite deux paramètres : l'identificateur de la règle et une référence vers l'instance de la classe *RulesChecker* qui vérifie les règles. Cette classe doit également définir une méthode `checkRule()` qui ne prend aucun paramètre et renvoie un entier. Celui-ci doit être l'une des trois constantes définies dans la classe *Rule* : `PASSED`, `FAILED` ou `UNKNOWN`. Le listing 6.14 reprend la structure d'une telle classe.

Listing 6.14 – Structure d'une classe Java implémentant une règle

```
package rule;
...
public class NomDeLaRegle extends Rule {

  public NomDeLaRegle(String idRule, RulesChecker ruleChecker) {
    super(idRule, ruleChecker);
  }

  public int checkRule() {
    ...
    return PASSED;
  }
```

La classe parente *Rule* contient différentes variables dont la visibilité est *protected*. Celle-ci permet à ses classes enfants d'hériter de ces variables sans passer par des *getters* et *setters*[3]. Le listing 6.15 montre ces différentes variables.

Listing 6.15 – Variable d'instance de la classe `Rule`

```
protected RulesChecker ruleChecker;
protected Logger logger;
protected ErrorHandler errorHandler;

protected XQueryEvaluator queryOnConfiguration;
protected XQueryEvaluator queryOnRules;
```

L'auteur de la nouvelle classe est libre de les utiliser comme bon lui semble. Si des erreurs sont lancées, elles peuvent automatiquement être gérées par *errorHandler* qui permet également de les *logger* automatiquement. Il est vivement conseillé d'utiliser le *logger* au moins une fois pour afficher les contre-exemples trouvés. Ceci est réalisé grâce à la méthode `logFailedRule`. Le listing 6.16 montre la signature d'une telle méthode.

[3]Nous supposons ici que le créateur n'est pas une personne malicieuse envers sa propre implémentation. De plus, s'il désire les modifier, c'est seulement sa propre instance de la classe `Rule` qui en portera les conséquences.

Listing 6.16 – Signature de la méthode `logFailedRule`

```
/**
* Log the counter-examples of the rule with id 'ruleId'.
* @param result : represent the counter-examples of the rule as a string
      (\textit{i.e.}, not in a XML form).
* @param ruleId : id of the rule
*/
public void logFailedRule(String result, String ruleId);
```

Cependant, il n'est pas toujours nécessaire de *logger* soi-même les résultats. En effet, si la règle écrite en Java nécessite au final une seule requête XQuery, alors nous pouvons utiliser la méthode `checkRule(String xquery, String xsl)` de la classe `Rule`[4]. Cette méthode permet de renvoyer `PASSED` si la règle est respectée ou de renvoyer `FAILED` en *loggant* les contre-exemples si la règle n'est pas respectée. Cette méthode prend en paramètres la requête XQuery ainsi qu'une feuille de style XSLT, toutes deux représentées sous la forme de chaîne de caractères. La feuille de style permet de mettre en forme les résultats de la requête. Le listing 6.17 montre la signature d'une telle méthode.

Listing 6.17 – Signature de la méthode `checkResultRule`

```
/**
* Check the rule represented by the XQuery 'xquery'.
*
* @param xquery : well formed XQuery.
* @param xsl : XSLT style sheet without a XSLT header
* @return PASSED if the result of the XQuery 'xquery' is empty
*       otherwise return FAILED and log the result of applying the style sheet 'xsl'
*     to the result of the XQuery 'xquery' (\textit{i.e.} the counter-exemples of the rule).
*/
protected int checkRule(String xquery, String xsl)
```

Les variables les plus utiles de la classe `rule` sont `queryOnConfiguration` et `queryOnRules`. Celles-ci référencent des objets de la classe `XQueryEvaluator` permettant d'effectuer n'importe quelles requêtes XQuery sur le fichier XML représentant le réseau et sur le fichier XML contenant les règles. Le listing 6.18 montre les signatures de deux méthodes disponibles sur un objet de type `XQueryEvaluator`.

La documentation complète concernant les classes et les méthodes est disponible dans la *Javadoc* de notre logiciel. Les sections suivantes montreront deux exemples de règles vérifiées par des classes Java.

[4]Cette méthode ne doit pas être confondue avec la méthode abstraite portant le même nom mais sans paramètres (*i.e.* `checkRule()`)

Listing 6.18 – Méthodes de la classe `XQueryEvaluator`

```
/**
* Evaluate the XQuery on the XML context of this object.
* @param xquery : a well formed XQuery.
* @return a String representing the result of the XQuery.
*     If an error occur then the empty sequence "" will be returned.
*/
public String evaluate(String xquery);

/**
* Evaluate the XQuery on the XML context of this object
*    and get the result as a SequenceIterator.
* @param xquery : a well formed XQuery.
* @return a SequenceIterator of the results of the XQuery 'xquery'.
*   Each XML element of the result of the XQuery 'xquery' is a item in the sequence iterator.
*     If an error occur then null will be returned
*/
public SequenceIterator evaluateToIterator(String xquery);
```

Connexité de la topologie physique

La topologie physique peut être représentée par un graphe dans lequel un routeur est représenté par un sommet et un lien est représenté par une arête. La vérification de la connexité de ce graphe nécessite un algorithme, comme celui présenté dans [WB08].

Les réseaux locaux sont représentés dans notre fichier XML par des liens composés de plus de deux noeuds. Ceci n'est pas directement représentable en termes de sommets et d'arêtes. Plusieurs solutions sont possibles : soit créer un sous-graphe connexe à partir de ces noeuds en rajoutant des arêtes entre eux (p.ex. *full mesh*, les relier consécutivement, etc.), soit créer une entité spéciale représentant un réseau local (LAN) et y connecter tous les noeuds du lien.

La figure 6.1 montre la transformation d'un réseau local en sommets et arêtes selon les deux possibilités décrites.

L'algorithme de test de connexité fonctionne aussi bien avec l'une ou l'autre solution. En effet, d'une manière assez intuitive, dans la première solution, on ne fait que rajouter des arêtes et dans la deuxième, on rajoute des arêtes et un sommet. Ces ajouts garantissent la connexité entre les noeuds du LAN. Ils sont effectués seulement entre des noeuds qui peuvent communiquer ensemble. On ne rajoute pas des arêtes entre des noeuds qui ne sont pas connectés, par conséquent, on ne modifie en rien le résultat de l'algorithme.

En prévision de la règle suivante qui vérifie la présence d'un *single link of failure*, nous avons choisi la seconde solution. Ce choix sera motivé dans la section suivante.

Algorithme de test de connexité Soit G=(V,E) un graphe (non trivial $|V| \geq 2$); $E' = E$ et $F = \emptyset$.

1. Prendre une arête $e \in E'$, ajouter e à F, enlever e de E'.

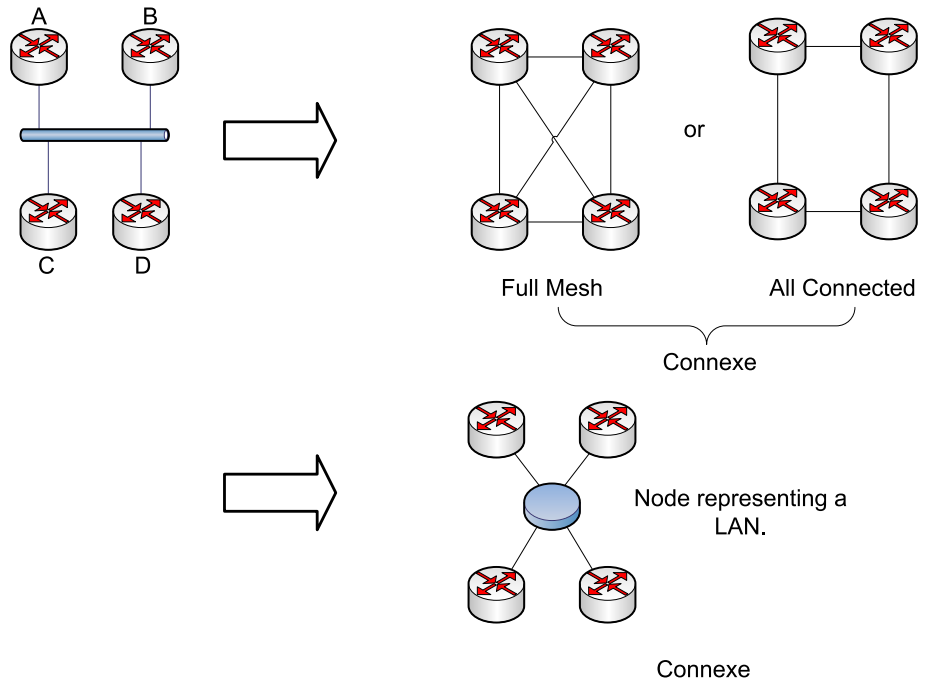

FIG. 6.1 – Transformation d'un LAN en sommets et arêtes.

2. Pour toute arête $e \in E'$ qui est incidente à au moins un noeud de $V(F)$, avec $V(F)$ l'ensemble des noeuds atteints par les arêtes dans F, ajouter e à F et enlever e de E'.
3. Répéter 2 jusqu'à ce que E' soit vide ou qu'on ne puisse plus décroître $|E'|$

Si l'algorithme se termine avec $F = E$ et $V = V(F)$, alors G est connexe.

Pas de *single link of failure* dans l'*area* 0

Dans OSPF, l'*area* 0 représente le *backbone* ou le coeur du réseau. Cette partie du réseau est cruciale pour garantir la communication entre les *area* qui y sont connectées. Nous voulons éviter que, si un seul lien tombe en panne dans cette *area*, alors elle soit divisée en deux parties qui ne peuvent plus communiquer.

Cette règle requiert un algorithme de test de connexité pour être vérifiée. Dès lors, elle est implémentée par une classe Java. De plus, cet algorithme est déjà implémenté pour la vérification de la connexité de la topologie. Par conséquent, la classe Java effectuant cette vérification est réutilisée. L'*area* 0 est représenté par un graphe dont les sommets représentent les routeurs de cette *area* et les arêtes représentent les liens OSPF entre ceux-ci.

Plus précisément, cette règle est respectée, si le graphe obtenu en retirant n'importe quelle arête du graphe représentant l'*area* 0 est connexe.

Algorithme de test de présence d'un *single link of failure* Soit $G = (V, E)$ représentant l'*area* 0, avec V l'ensemble des sommets et E l'ensemble des arêtes. Initialisons $E' = E$.

1. Prendre une arête $e \in E'$, enlever e de E' et enlever e de E.

2. Effectuer le test de connexité sur $G = (V, E)$, si le test échoue alors e est un *single link of failure* et l'algorithme s'arrête, si le test réussit alors remettre e dans E.
3. Répéter les étapes 1 et 2 jusqu'à ce que E' soit vide.
4. Le graphe $G = (V, E)$ ne contient pas de *single link of failure*.

Motivation du choix de représenter un réseau local par un noeud Pour cet algorithme, l'objectif est d'obtenir un graphe non-connexe lorsqu'on retire une arête d'un noeud faisant partie uniquement d'un réseau local. La première solution qui consiste à rajouter des arêtes entre les noeuds du réseau local ne convient pas. En effet, prenons le cas le plus évident où l'on construit un *full mesh* entre les routeurs du réseau local. Ce graphe reste connexe lorsqu'on enlève une arête du *full mesh*. Ceci n'est pas le comportement souhaité.

Nous avons donc décidé de modéliser un réseau local par un sommet supplémentaire et d'y connecter tous les noeuds de ce LAN. Ceci fonctionne avec l'algorithme. Prenons l'exemple d'un routeur possédant uniquement un lien avec le réseau local, ce routeur perd cette connexion et devient alors injoignable. Ce cas est correctement traité par l'algorithme avec cette représentation des réseaux locaux. En effet, lorsque l'arête reliant le noeud représentant le routeur au noeud représentant le LAN est retirée, alors ce noeud n'a plus d'arêtes et l'algorithme déclare que le graphe n'est plus connexe. Cette arête est détectée comme un *single link of failure*.

La figure 6.2 montre la transformation d'un LAN dans lequel un routeur n'est plus joignable en sommets et arêtes selon les deux possibilités décrites précédemment. Nous voyons qu'après avoir retiré une arête, la première solution donne un graphe qui est toujours connexe, tandis que la seconde donne un graphe qui ne l'est plus.

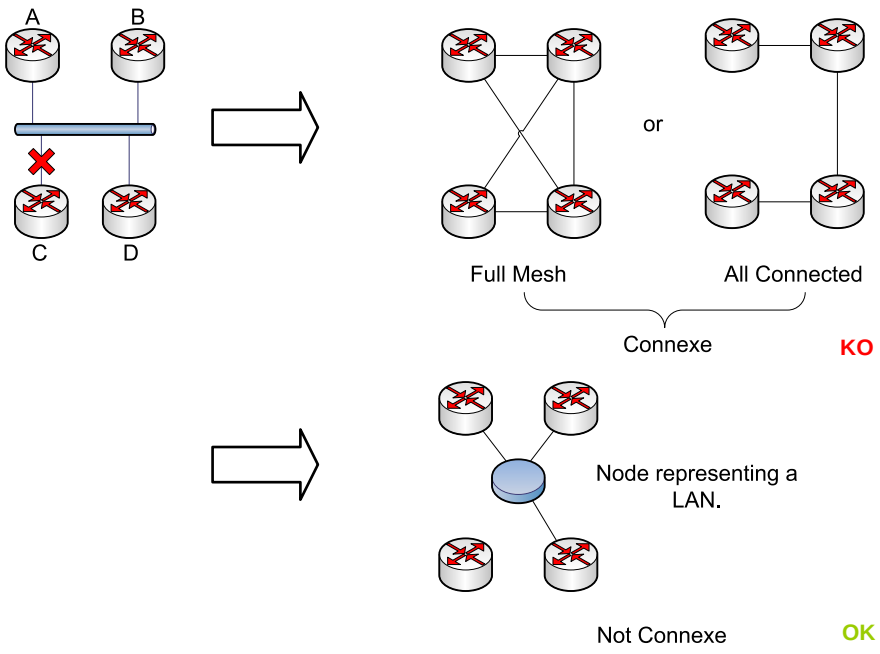

FIG. 6.2 – Représentation d'un LAN pour l'algorithme de détection des *single link of failure*

6.4 Choix de la technique

Chaque technique possède ses avantages et ses inconvénients. Ceux-ci vont influencer le choix de la technique à utiliser pour vérifier une règle. Logiquement, la technique utilisée est celle qui permet d'exprimer la règle de la façon la plus simple possible. Par exemple, si une règle peut être exprimée dans un schéma XML (*Structural rule*) alors il n'est pas nécessaire d'utiliser XQuery (*Query rule*) pour la vérifier. De même, si une règle ne peut pas être exprimée à l'aide d'un schéma XML, alors il est nécessaire d'utiliser soit une *Query rule*, soit une *Language rule*.

Le tableau 6.1 montre les techniques conseillées en fonction du type de règles. Ces conseils sont issus de notre expérience et ne constituent en rien une marche à suivre obligatoire. Cependant, toutes les règles écrites à ce jour respectent ce tableau. Lorsque plusieurs choix sont possibles, il est utile de connaître les avantages et les inconvénients de chaque technique pour choisir la bonne marche à suivre. Toutes les informations relatives à ces différentes techniques sont reprises dans les sections 6.1, 6.2 et 6.3.

	présence	non-présence	unicité	symétrie	personnalisée
Structural rule	✓	✓	✓	✓	
Query rule	✓	✓	✓	✓	✓
Language rule					✓

Tab. 6.1 – Techniques conseillées en fonction du type de règles

La figure 6.3 montre sous la forme d'un organigramme la marche à suivre pour choisir la technique appropriée selon le type des règles et selon ses besoins. Ceux-ci sont identifiés à l'aide de quelques questions types telles que « La règle s'applique-t-elle sur l'entièreté du réseau ? » ou « Cette règle nécessite-t-elle l'évaluation d'une condition ? ». Chaque choix, dans cet organigramme, est illustré par un exemple. Comme précédemment, ceci n'est pas une marche à suivre obligatoire, mais plutôt une considération empirique pour faciliter le choix de la technique à utiliser.

6.5 Ajout d'un nouveau type de règles

Dans cette section, nous expliquerons comment un opérateur peut aisément ajouter un nouveau type de règles au sein de notre logiciel. Dans cet objectif, nous partons d'un cas concret basé sur la matrice de filtrage définie dans la section 5.2.6 et, plus particulièrement, sur la matrice qui définit les politiques BGP fréquentes (*i.e.* fournisseur, client, etc.) décrite dans le tableau 5.1. Le but poursuivi est ici de s'assurer que la matrice des politiques est correctement définie et qu'elle le restera après d'éventuels changements futurs. Notons néanmoins que, si d'autres lignes de la matrice doivent être ajoutées par la suite, de nouvelles règles les vérifiant seront également nécessaires.

La première étape est d'identifier les situations dans lesquelles les politiques ne sont pas respectées. Dans ce cas simple, quatre violations sont identifiables :

1. les routes annoncées par un fournisseur sont redistribuées sur une session à coût partagé ;

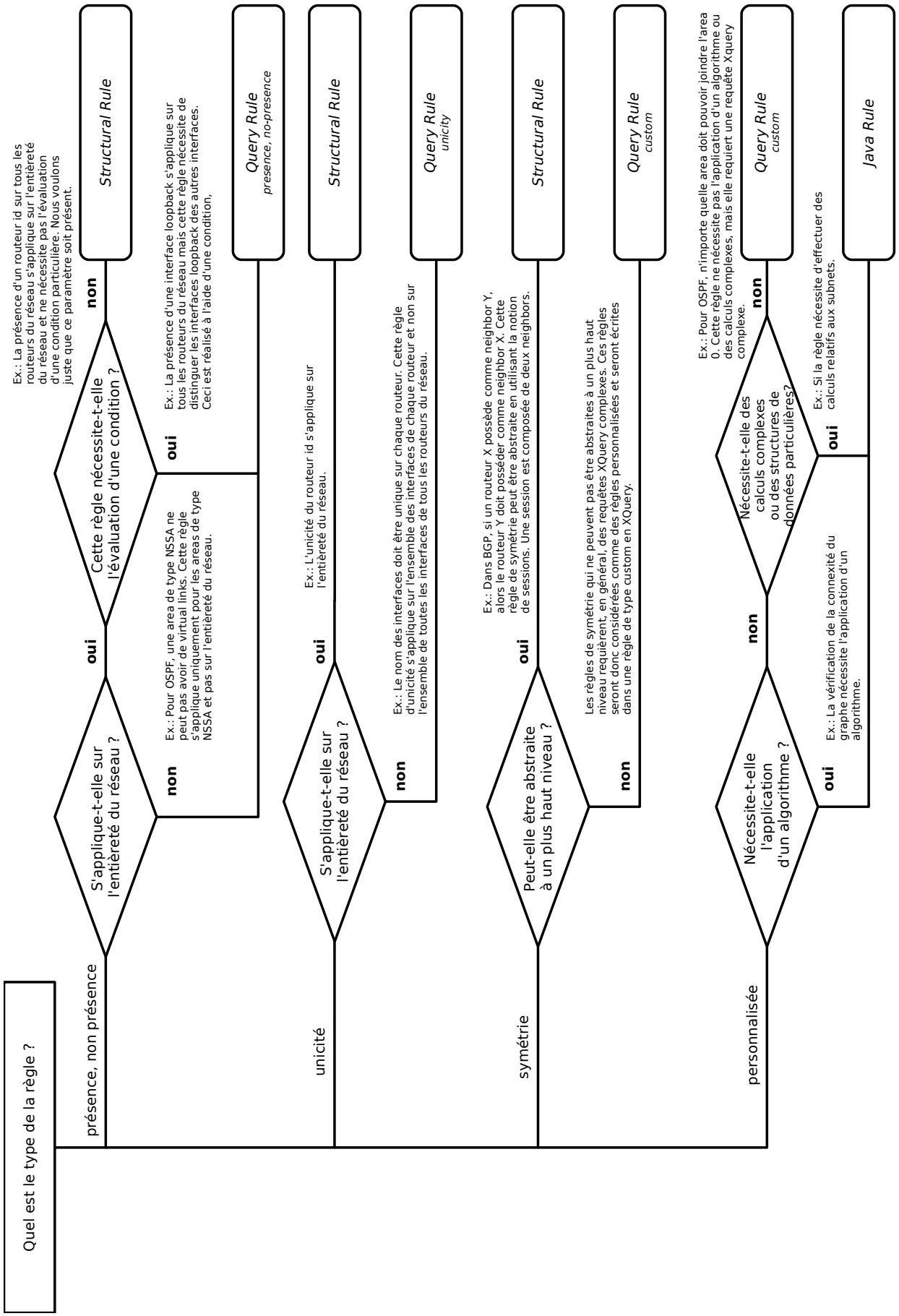

FIG. 6.3 – Choix de la technique en fonction du type de règles

2. les routes annoncées par un fournisseur sont redistribuées à un autre fournisseur ;
3. les routes annoncées sur une session à coût partagé sont redistribuées sur une autre session à coût partagé ;
4. les routes annoncées sur une session à coût partagé sont redistribuées à un fournisseur.

Ensuite, l'opérateur doit écrire les règles qui vérifient que ces différentes situations n'apparaissent pas dans la représentation de son réseau. De la même manière, il peut également souhaiter vérifier que ses politiques sont effectivement appliquées (p.ex. que les routes qui proviennent d'un client sont annoncées aux fournisseurs). Pour atteindre ses objectifs, il décide logiquement d'allier des règles de *présence* et de *non-présence* et ce, afin de vérifier la présence des ses politiques et l'absence de violation de ces dernières. La technique la plus appropriée est ici d'utiliser une *Query rule*, car la règle ne peut être vérifiée à l'aide d'un schéma XML et ne nécessite pas la puissance d'un langage de programmation de haut niveau (*i.e.* pas besoin d'un algorithme). Un exemple combinant ces deux types de règles est repris dans le listing 6.19. La première règle (`PROVIDERS_ROUTE_ANNOUNCED_TO_CUSTOMERS`) s'assure que les routes annoncées par un fournisseur sont redistribuées aux clients. La seconde règle vérifie que ces routes ne sont pas annoncées sur d'autres sessions.

Listing 6.19 – Exemples de règles vérifiant la matrice de filtrage BGP

```
<rule id="PROVIDERS_ROUTE_ANNOUNCED_TO_CUSTOMERS" type="presence">
    <description>Routes coming from providers must be announced to customers</description>
    <presence>
        <scope>PROVIDERS_ENTRIES</scope>
        <descendents>out-groups/out-group</descendents>
        <condition>@id='CUSTOMERS'</condition>
    </presence>
</rule>

<rule id="PROVIDERS_ROUTES_NOT_ANNOUNCED_TO_ANYONE_EXCEPT_CUSTOMERS" type="non-presence">
    <description>Routes coming from providers must not be announced to anyone
                 except customers</description>
    <non-presence>
        <scope>PROVIDERS_ENTRIES</scope>
        <descendents>out-groups/out-group</descendents>
        <condition>@id!='CUSTOMERS'</condition>
    </non-presence>
</rule>
```

Cependant, l'opérateur remarque assez vite que cette façon de faire n'est pas la plus efficace. En effet, pour chaque ligne de la matrice, celui-ci doit écrire deux règles (une règle de *présence* et une règle de *non-présence*), ce qui peut s'avérer assez lourd avec une matrice de grande taille. Par conséquent, l'opérateur décide de rajouter un nouveau type de règle nommé `matrix` qui permet de vérifier à l'aide d'une seule règle ces deux conditions. Cette nouvelle règle vérifie que, pour un groupe de sessions donné (p.ex. `PROVIDERS`), les routes annoncées sont redistribuées uniquement aux groupes précisés. Deux violations sont possibles : les routes ne sont pas annoncées à un de ces groupes (violation de la règle de *présence*) ou les routes sont annoncées à un groupe non-précisé (violation de la règle de *non-présence*). La représentation de ces règles est reprise dans le listing 6.20. Celle-ci suit les différentes conventions de représentation de règles (section 4.4). Notons qu'un nouveau type (`matrix`) a été défini. Concrètement, les règles de ce type vérifient que les routes de chaque `in-set` sont bien annoncées aux `out-set` correspondants et uniquement

à ceux-là. Ainsi la règle `PROVIDERS_ROUTES_EXPORT` vérifie que les routes annoncées sur l'une des sessions appartenant au groupe `PROVIDERS` sont annoncées au groupe `CUSTOMERS` et uniquement à celui-là. Un exemple d'erreur produite par cette règle est repris dans le listing 6.21.

Listing 6.20 – Exemples de règles de type `matrix`

```
<rule id="PROVIDERS_ROUTES_EXPORT" type="matrix">
    <description>Routes coming from providers must only be announced to
                customers peerings !</description>
    <matrix>
        <in-set>PROVIDERS</in-set>
        <out-set>CUSTOMERS</out-set>
    </matrix>
</rule>
<rule id="SHARED_COSTS_ROUTES_EXPORT" type="matrix">
    <description>Routes coming from shared-costs peerings must only be announced to
                customers peerings</description>
    <matrix>
        <in-set>SHARED_COSTS</in-set>
        <out-set>CUSTOMERS</out-set>
    </matrix>
</rule>
<rule id="CUSTOMERS_ROUTES_EXPORT" type="matrix">
    <description>Routes coming from customers peerings must only be announced to
                shared-costs and providers peerings</description>
    <matrix>
        <in-set>CUSTOMERS</in-set>
        <out-set>SHARED_COSTS</out-set>
        <out-set>PROVIDERS</out-set>
    </matrix>
</rule>
```

Listing 6.21 – Exemples d'une erreur détectée par une règle de type `matrix`

```
---------------------- PROVIDERS_ROUTES_EXPORT failed : Summary ----------------------
Exportation error found in the filter-matrix for PROVIDERS sessions
Please check that those routes ARE announced to  CUSTOMERS  and ONLY to them !
```

Ce nouveau type de règles est implémenté par la classe Java `PolicyMatrixRule` qui étend la classe `Rule`. La structure de cette classe est similaire aux classes `PresenceRule`, `NonPresenceRule`, `UniquenessRule`, etc. Pour rappel, la structure de ces classes est détaillée dans la section 6.3.

6.6 Conclusion

Ce chapitre a présenté de manière détaillée comment les différents types de règles sont vérifiés par notre logiciel. Trois techniques ont été utilisées : un schéma XML qui spécifie la structure de la configuration du réseau, des requêtes XQuery qui permettent d'identifier les éléments de configuration qui ne respectent pas certaines règles et enfin des classes Java qui offrent la possibilité de vérifier des règles plus complexes.

Toutes ces techniques ne sont pas équivalentes. Dès lors, certaines sont plus adaptées que d'autres pour vérifier un certain type de règles. Ce choix n'est pas dicté par des conventions strictes mais relève de l'expérience de l'utilisateur. Dans le but d'aider ce dernier, un organigramme détaillant les principales questions à se poser a été repris dans ce chapitre.

Enfin, nous avons également montré qu'il est aisé de rajouter un nouveau type de règles à l'aide d'un exemple relatif aux politiques BGP.

Chapitre 7

Génération de configurations

La génération des configurations est le dernier maillon dans notre logiciel. Ce processus permet, à l'aide de modèles, de générer des configurations dans n'importe quel langage de configuration ou de modélisation (p.ex. IOS, JunOS, C-BGP, etc.). Ces différents modèles sont composés d'instructions permettant de traduire la représentation initiale en configurations exprimées dans ces langages.

Comme présenté dans la section 4.5, ce processus est réalisé en deux étapes. La première consiste à produire, en utilisant XQuery, des représentations intermédiaires à partir de la représentation de haut niveau. Celles-ci doivent être structurellement assez proches du format des configurations souhaitées de manière à faciliter la seconde étape. Cette dernière est la génération, à proprement parler, des fichiers de configurations. Elle est effectuée en appliquant des feuilles de style XSLT sur les représentations intermédiaires.

Dans ce chapitre, nous détaillerons la façon dont notre logiciel réalise ces deux étapes et ce, dans l'objectif de permettre à un opérateur d'ajouter facilement un nouveau langage de configuration. Plus particulièrement, nous présenterons comment la représentation de haut niveau est transformée en représentations intermédiaires par l'utilisation de requêtes XQuery. Ensuite, nous présenterons l'étape finale qui consiste à transformer ces représentations intermédiaires en configurations dans le langage souhaité. De plus, nous aborderons la génération automatique de certaines règles de « bonnes pratiques » qui se trouvent dans la plupart des configurations déployées.

7.1 Représentations intermédiaires

La représentation de la configuration du réseau est transformée en représentations intermédiaires. Chacune d'entre-elles doit contenir toutes les informations nécessaires pour générer la configuration correspondante. De plus, ces informations doivent être placées aux endroits appropriés. Par exemple, dans la configuration d'un routeur, le paramètre MTU se situe au niveau des interfaces alors que celui-ci se trouve au niveau des liens dans notre représentation. Dès lors,

il est nécessaire de le dupliquer et de le placer au niveau des interfaces. Cet exemple souligne la façon dont les règles de *symétrie* sont vérifiées par construction.

D'une manière générale, les représentations intermédiaires contiennent au moins autant d'informations que la représentation de départ. De plus, elles possèdent une structure proche de celle des fichiers de configurations souhaités et ce, afin de faciliter autant que possible le processus final de génération.

La figure 7.1 montre un exemple de la transformation de la représentation d'un réseau avec deux routeurs en deux représentations intermédiaires. Cet exemple montre que le MTU présent initialement au niveau du lien R1-R2 a été dupliqué au niveau des interfaces dans les deux représentations intermédiaires.

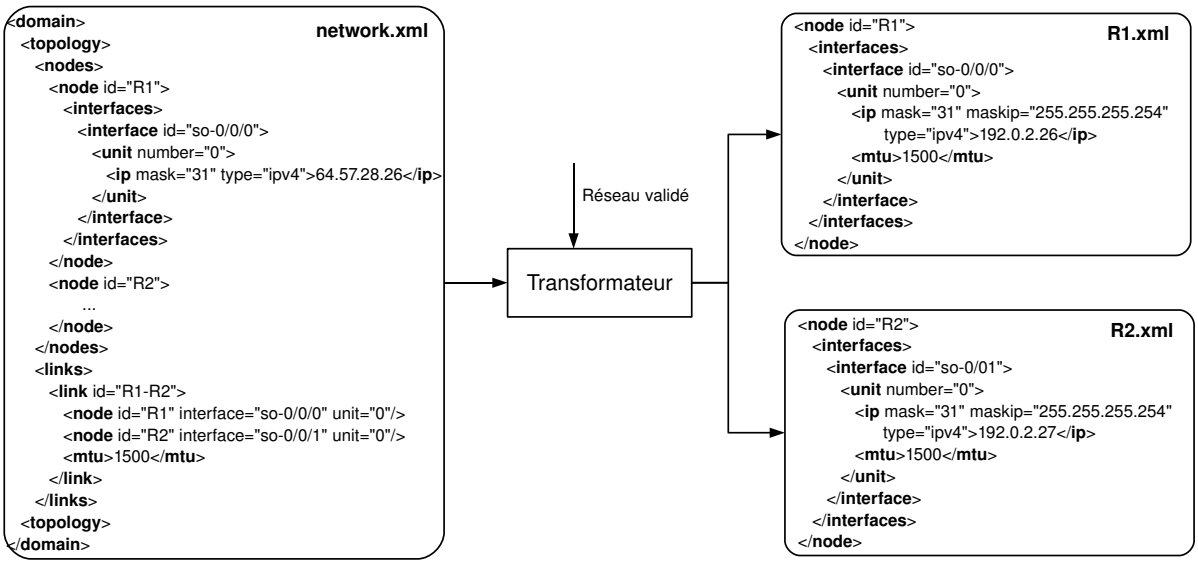

FIG. 7.1 – Exemple de génération de configurations intermédiaires

Le listing 7.1 montre un extrait des requêtes XQuery permettant de générer la représentation intermédiaire d'un routeur représenté par la variable `$node`. Ces requêtes sont imbriquées les unes dans les autres avec la propriété importante qu'une requête imbriquée peut utiliser n'importe quelle variable définie préalablement dans une des requêtes hiérachiquement supérieures. Ainsi, nous pouvons remarquer que la variable `$node` est réutilisée à la troisième ligne pour sélectionner les interfaces du routeur concerné. Cet exemple montre également comment le nom (*i.e.* id) de l'interface et sa description sont rajoutés dans la représentation intermédiaire.

7.1.1 Fonctions définies

Les informations présentes dans une représentation intermédiaire doivent permettre de générer facilement la configuration correspondante. De plus, la syntaxe et la sémantique des fichiers de configurations à générer diffèrent selon le constructeur. Par exemple, un équipement Cisco requiert d'utiliser la notation pointée (*i.e.* `xxx.xxx.xxx.xxx`) pour configurer un masque tandis que Juniper autorise l'utilisation de sa longueur. Notre représentation contient uniquement la lon-

Listing 7.1 – Extrait de la transformation en représentations intermédiaires

```
<node>{$node/@id}
  <interfaces>
  { for $interface in $node/interfaces/interface
    return
    <interface>
      {$interface/@id}
      {$interface/description}
    </interface>
  }
  </interfaces>
  ...
</node>
```

gueur du masque. Une fonction a donc été définie pour réaliser la transformation de la longueur d'un masque en un masque sous la forme pointée. Cette dernière est ajoutée aux représentations intermédiaires. Ainsi, il est fréquent de trouver des informations redondantes dans les représentations intermédiaires (p.ex. la longueur du masque et sa notation pointée). Ceci est illustré dans la figure 7.1 par l'attribut `maskip` et l'attribut `mask` d'un élément XML de type `ip`.

Le listing 7.2 montre l'en-tête de la fonction permettant de transformer un masque numérique vers sa notation pointée. Cette fonction est une fonction écrite dans le langage XQuery. Toutes les informations concernant la définition de telles fonctions sont disponibles dans [Wal07, SRF+07].

Listing 7.2 – Transformation d'un masque numérique IPv4 vers sa notation pointée

```
(: ************************************************************************
   * Transform the length of a IPv4 mask into the dotted representation.
   * of a mask.  ex.: 30 -> 255.255.255.252
   * @param mask : the length of the mask (between 0 and 32)
   * @return a well formed IP representing 'mask'.
   ************************************************************************ :)
declare function transform:transform-mask-ipv4-to-dotted-format($mask as xs:integer)
 as xs:string
```

Dans la perspective de générer n'importe quel format de configuration, un opérateur réseau peut, à sa guise, définir de nouvelles fonctions et rajouter des informations dans les représentations intermédiaires là où elles sont nécessaires.

7.1.2 Paramètres par défaut

Si aucune valeur n'est définie pour un certain élément de la configuration, alors une valeur par défaut est souvent utilisée. Cette valeur dépend du constructeur et, parfois même, de la version de l'OS utilisée. Par exemple, la valeur par défaut de la priorité d'une interface OSPF vaut 1 sur les équipements Cisco [DB06] et 128 sur les équipements Juniper [8]. Cependant, l'utilisation de différentes valeurs par défaut peut violer certaines règles de *symétrie*. Par exemple, la règle de *symétrie* d'*égalité simple* qui vérifie que le même MTU est présent sur les interfaces

interconnectées peut ne pas être respectée si la valeur du MTU n'est pas définie sur le lien correspondant. En effet, si le lien interconnecte deux routeurs de constructeurs différents et que ceux-ci ont une valeur par défaut du MTU différente alors les configurations générées ne respecteront pas cette règle de *symétrie*.

Afin d'éviter ce problème et de permettre malgré tout de définir des paramètres par défaut, nous avons adopté une stratégie qui permet à l'opérateur de définir ses propres valeurs par défaut. Une seule valeur par défaut par élément est autorisée. Par exemple, l'opérateur peut définir que la valeur par défaut du MTU est 1500 octets. Celle-ci sera utilisée lorsque le lien ne possède pas une valeur définie pour le MTU. Ceci garantit que la règle de *symétrie* est vérifiée car la même valeur du MTU est présente pour les interfaces interconnectées dans les configurations générées et ce, même si les constructeurs diffèrent. Notons que ce mécanisme est applicable si aucune règle n'exige la présence de l'élément XML représentant le paramètre en question.

Les valeurs par défaut sont stockées dans le fichier `default-parameters.xml`. La structure de ce fichier est très semblable à celle de la représentation de la configuration du réseau. Le listing 7.3 montre un exemple de ce fichier dans lequel la valeur par défaut du MTU est 1500. Les valeurs par défaut pour les différents temporisateurs d'OSPF ainsi que la bande passante de référence sont également définies.

Listing 7.3 – Exemple de la représentation des paramètres par défaut

```
<?xml version="1.0" encoding="UTF-8"?>
<domain>
  <topology>
    <links>
      <link>
        <mtu>1500</mtu>
      </link>
    </links>
  </topology>
  <ospf>
    <intervals>
      <link>
        <hello>10</hello>
        <dead>40</dead>
        <retransmit>5</retransmit>
      </link>
    </intervals>
    <reference-bandwidth>100</reference-bandwidth>
    ...
  </ospf>
  ...
</domain>
```

Une fonction particulière a été définie afin de récupérer une valeur par défaut. La signature de cette fonction est montrée au listing 7.4. Le listing 7.5 montre un exemple de l'utilisation de cette fonction sur l'élément XML représentant le MTU. Si l'élément accessible via `/attributes/mtu` n'existe pas, alors la valeur du paramètre par défaut accessible via `topology/links/link/mtu` dans

le fichier représentant les paramètres par défaut est utilisée. La variable `$default-parameters-doc` permet d'indiquer le fichier contenant les paramètres par défaut.

Listing 7.4 – Signature de la fonction `get-default-value`

```
(: ********************************************************************************
   * Get the default value of a parameter if not set
   *
   * @param default-parameters-doc : XPath representing the XML document with the default
   *        parametersex.: doc('default-parameters')/domain/
   * @param param : the value of the parameter, it can be empty.
   * @param path-param : the path to reach the paramter in the XML file with the default
   *        parameters.
   * @return 'param' if 'param' is not empty
   *    otherwise the default value of the parameter accessible via $path-param
   *    in the document $default-parameters-doc is returned.
   ********************************************************************************:)

declare function transform:get-default-value(
  $default-parameters-doc as xs:string,
  $param as element()?,
  $path-param as xs:string)
```

Listing 7.5 – Exemple d'utilisation de la fonction `get-default-value`

```
declare variable $default-parameters-doc as xs:string := "doc('default-parameters.xml')";
{transform:get-default-value($default-parameters-doc, /attributes/mtu,
    "domain/topology/links/link/mtu")}
```

7.2 Transformation en configurations

Les représentations intermédiaires obtenues sont structurellement assez proches des configurations souhaitées. Dès lors, une feuille de style XSLT peut être utilisée pour transformer les représentations intermédiaires dans un langage de configuration ou de modélisation. Chaque représentation intermédiaire représente un équipement du réseau (p.ex. un routeur). Actuellement, le choix de la feuille de style à utiliser pour effectuer la transformation est dicté par la valeur du constructeur de l'équipement.

La figure 7.2 montre un exemple pour un routeur Cisco $R1$ et pour un routeur Juniper $R2$. Des extraits des feuilles de style pour Cisco et Juniper sont également montrés dans cette figure.

Le principe de base de ces feuilles de style est d'appliquer un certain *template* selon le type de l'élément. Dans la figure 7.2, dès qu'un élément de type `<interfaces>` est trouvé, alors pour chacun de ses sous-éléments `<interface>`, nous transformons le nom de l'interface et sa description dans la syntaxe appropriée. La valeur d'un élément est obtenue grâce à la balise XSLT `<xsl:value-of select="..."/>`. Par exemple, le nom de l'interface est obtenu avec `<xsl:value-of select="@id"/>`. Tous les détails concernant XSLT sont disponibles dans [Tid01, Cla99] et dans la section 2.5.

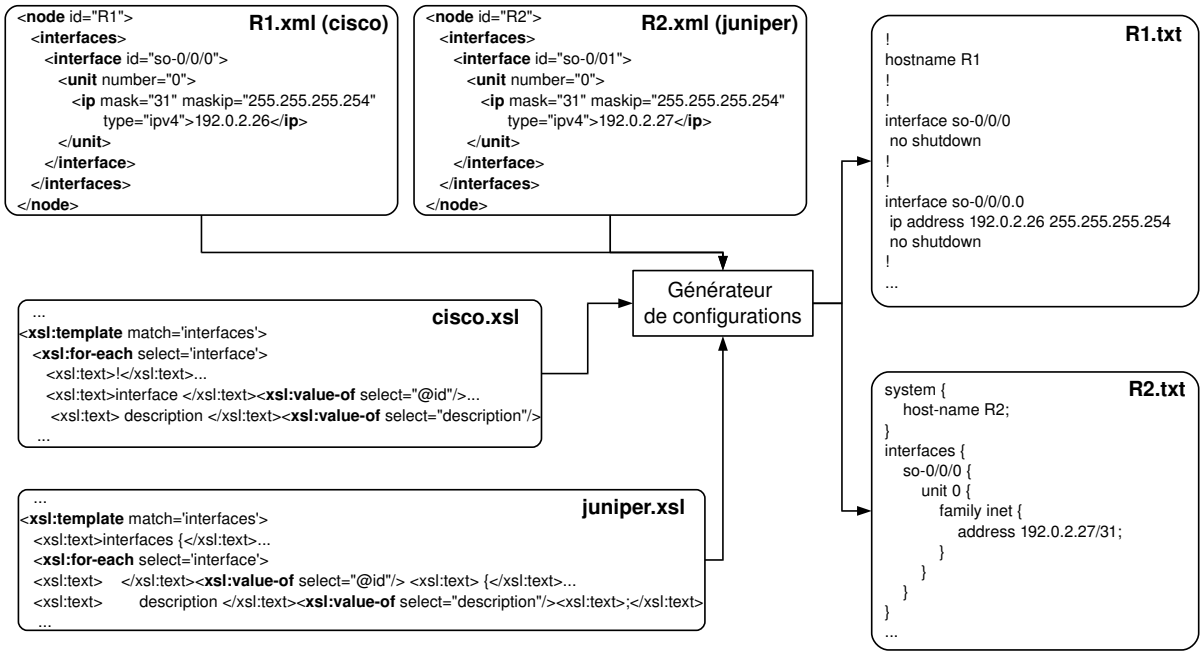

FIG. 7.2 – Exemple de génération des configurations d'un routeur Cisco et d'un routeur Juniper

7.2.1 Règles de « bonnes pratiques » générées

Comme nous l'avons déjà expliqué, configurer un réseau est une tâche complexe et sujette aux erreurs. Cependant, certaines règles de « bonnes pratiques » existent. Ces règles permettent, notamment, à un opérateur de protéger son réseau des erreurs commises par d'autres opérateurs mais également d'améliorer la sécurité de celui-ci. Par exemple, si un domaine X annonce à son voisin Y que le préfixe 192.168.0.0/16 lui appartient, il existe un risque non-nul que Y envoie à X le trafic appartenant à ce préfixe. Ceci va totalement à l'encontre du principe des adresses privées qui ne doivent, en aucun cas, sortir du domaine [RMK+96]. Une règle simple qui permet de se prémunir de ce genre d'événements est de rajouter deux filtres : le premier bloque l'importation des adresses privées ; le second bloque l'exportation de celles-ci pour éviter de causer des problèmes aux autres domaines qui ne suivraient pas encore ce principe. L'équipe *Cymru* recense sur son site [11] une partie de ces règles et ce, dans différents langages de configuration tels que IOS ou JunOS.

Ces différentes règles s'appliquent généralement à l'entièreté du réseau, c'est-à-dire sur chaque équipement qui le compose. Dès lors, nous avons choisi de générer celles-ci automatiquement dans les configurations produites par le logiciel. Ces règles de « bonnes pratiques » sont exprimées dans les différents modèles (XSLT) associés à chaque format de sortie (Cisco, Juniper, etc.). Étant donné qu'une feuille de style XSLT est un document XML (voir section 2.5), l'ajout et la modification de nouvelles règles de « bonnes pratiques » sont aisés. Pour le moment, *vng* est capable de générer automatiquement les règles suivantes :

- filtre qui empêche l'importation de routes BGP dont l'*AS-PATH* contient au moins un numéro d'AS privé ;
- filtre qui empêche l'exportation de routes BGP dont l'*AS-PATH* contient au moins un numéro d'AS privé ;

- filtre qui empêche l'importation de routes BGP dont le préfixe est trop long (supérieur à un /27) ;
- filtre qui empêche l'importation de routes BGP privées dont le préfixe est une route privée ;
- filtre qui empêche l'exportation de routes BGP privées dont le préfixe est une route privée ;
- filtre qui empêche l'importation de routes BGP dont le préfixe correspond à un préfixe local du réseau ;
- activation de la seconde version du protocole SSH ;
- désactivation du droit de se *logger* en tant que *super-utilisateur* ;
- désactivation des messages *ICMP Redirects* ;
- désactivation du *source routing*.

Le listing 7.6 contient un exemple sous JunOS de filtre d'importation appliqué automatiquement sur les différentes sessions eBGP du domaine.

Listing 7.6 – Exemple de règles de « bonnes pratiques » générées automatiquement par *vng*

```
prefix-list INTERNAL {
    64.57.16.0/20;
    198.32.8.0/22;
    198.32.12.0/22;
    198.32.154.0/24;
}
policy-statement SANITY-IN {
    term block-private-asn {
        from as-path PRIVATE;
        then reject;
    }
    term block-bogons {
        term one {
            from {
                route-filter 224.0.0.0/4 orlonger;
            }
            then reject;
        }
    }
    term no-small-prefixes {
        from route-filter 0.0.0.0/0 prefix-length-range /27-/32 reject;
    }
    term block-internal {
        term one {
            from {
                prefix-list INTERNAL;
            }
            then reject;
        }
    }
}
as-path PRIVATE ".* (64512-65535) .*";
```

7.3 Conclusion

Ce chapitre a présenté la façon dont notre logiciel génère, à partir de la représentation de haut niveau, des configurations dans le langage souhaité. Ceci est réalisé, notamment, par l'utilisation d'un processus permettant de générer des représentations intermédiaires. Celui-ci peut être vu comme un « pré-processeur » qui effectue un « pré-traitement » de l'information afin de la rendre plus intelligible par le processus suivant. Dans notre contexte, ce traitement consiste à transformer la représentation de haut niveau en plusieurs représentations de bas niveau. Ces dernières peuvent être facilement transformées en configuration dans le langage souhaité grâce à l'application de feuilles de style XSLT. De plus, notre logiciel est capable d'introduire dans ces configurations des règles de « bonnes pratiques ». Celles-ci garantissent que certaines conventions types sont bien respectées.

Chapitre 8

Étude de cas

Dans ce chapitre, nous présenterons une étude de cas basée sur le coeur du réseau *Abilene* [1], un réseau de recherche américain composé de neuf routeurs. L'étude présentera de façon synthétique un exemple d'utilisation de notre logiciel et permettra de mieux percevoir les avantages qu'il confère à la phase de conception d'un réseau.

Notre logiciel et la formalisation des règles ont été développés sur base de deux autres réseaux (un réseau de recherche et un réseau universitaire). Cependant, pour des raisons de confidentialité, nous avons choisi de baser cette étude de cas sur le réseau *Abilene* qui est totalement public et dont les configurations sont disponibles en ligne sur [3].

Le chapitre débutera par un bref aperçu de la topologie du réseau. Dans cette section, nous expliquerons la structure d'*Abilene* ainsi que les protocoles de routage utilisés.

La section suivante abordera la *validation* de la représentation. Nous tâcherons d'illustrer plus particulièrement la flexibilité du logiciel en détaillant la démarche d'ajout d'une nouvelle règle validant l'un des choix de *design* utilisé par *Abilene*. Par la suite, nous identifierons d'autres choix utilisés et validés par le logiciel. Nous montrerons notamment le comportement du logiciel lors de la présence d'erreurs dans la représentation.

Ensuite, nous montrerons la modélisation d'un sous-ensemble des politiques interdomaines d'*Abilene*, grâce à l'utilisation de nos deux matrices des politiques BGP : la matrice de filtrage et la matrice des communautés. Nous présenterons également comment quelques règles peuvent être utilisées afin de valider ces abstractions.

Nous donnerons également quelques exemples de configurations générées par notre logiciel et exprimées dans le langage de configuration JunOS [9]. Une comparaison des configurations réelles et de celles générées est disponible dans l'annexe C.

8.1 Représentation de la topologie d'*Abilene*

Abilene se compose de neuf routeurs Juniper répartis sur l'ensemble du territoire américain. Un schéma décrivant sa topologie et quelques voisins BGP est repris dans la figure 8.1.

Le protocole de routage intradomaine modélisé est OSPF[1]. Ce dernier est activé sur l'ensemble des liens internes. Notons que notre modélisation d'*Abilene* utilise une seule *area* [Moy98] bien que notre outil en supporte davantage.

Le routage interdomaine est réalisé grâce au protocole BGP. Tous les routeurs établissent entre eux des sessions iBGP de manière à former un graphe complet (*full-mesh*). Quant aux sessions eBGP, chaque routeur en établit un certain nombre en fonction de ses voisins. Notons qu'*Abilene* n'utilise pas de réflecteur de routes (*Route Reflector*).

De manière à représenter le réseau dans notre représentation de haut niveau, une phase de rétro ingénierie des configurations a été nécessaire. Dans cette phase, nous avons notamment identifié les principes importants de *design* utilisés par les opérateurs d'*Abilene*.

Par manque de place, nous ne détaillerons pas le processus de traduction. Toutes les informations nécessaires à la représentation d'une topologie dans notre logiciel sont reprises dans le chapitre 5. L'ensemble des fichiers utilisés dans cette étude est disponible sur le CD accompagnant ce travail.

Un extrait de la représentation utilisée est disponible dans le listing 8.1. Celui-ci illustre quelques éléments du routeur *NY* (notamment une interface externe), d'un lien interne, d'une session eBGP et d'un groupe de sessions eBGP.

8.2 Validation

Comme mentionné précédemment, afin de valider la représentation d'un réseau, nous utilisons un mécanisme basé sur des règles qui représentent un ensemble de conditions que la représentation se doit de satisfaire. Ces règles sont généralement écrites par un opérateur[2] sur base de ses choix de *design* et fournies *a priori* au logiciel.

Dans le cas d'*Abilene*, cette approche n'est malheureusement pas applicable, car le réseau est déjà déployé. Dès lors, nous avons adopté une démarche de « rétro ingénierie » (*reverse engineering*) afin d'identifier les principes de *design* utilisés par les opérateurs d'*Abilene* pour ensuite écrire les règles correspondantes.

Dans notre logiciel, cinq types de règles sont disponibles : des règles de *présence*, de *non-présence*, d'*unicité*, de *symétrie* et des règles *personnalisées*. Ces règles peuvent être vérifiées par l'intermédiaire de trois techniques :

[1] En réalité, Abilene utilise le protocole IS-IS. Cependant, pour des raisons pratiques liées aux autres études de cas qui utilisaient OSPF, nous avons décidé de convertir la topologie IS-IS en topologie OSPF.

[2] Un certain nombre de règles sont fournies avec le logiciel (annexe B).

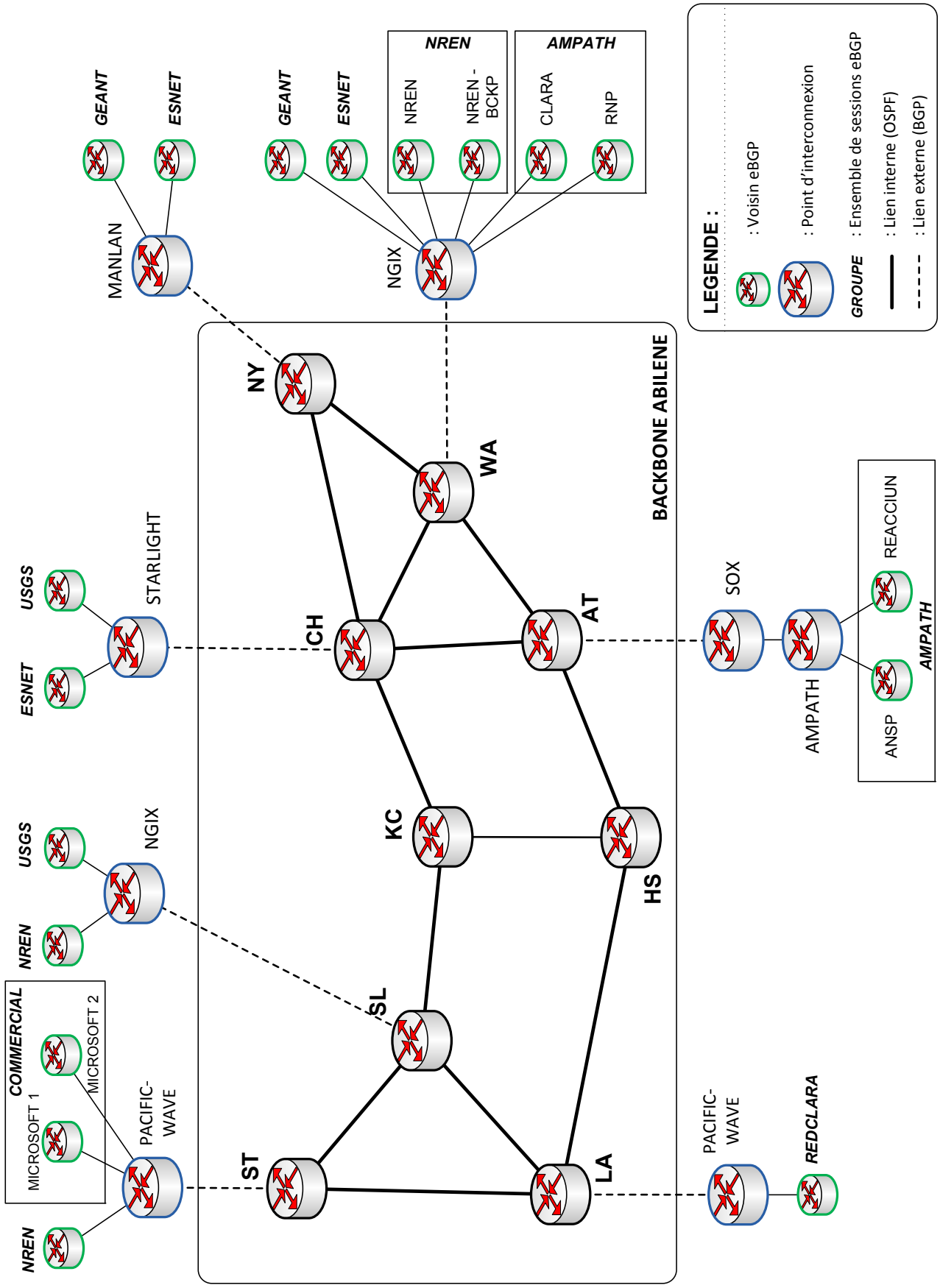

FIG. 8.1 – Extrait de la topologie d'*Abilene*

Listing 8.1 – Extrait de la représentation de haut niveau d'*Abilene*

```
<node id="NY">
    <characteristics>
        <reference>
            <constructor>juniper</constructor>
            ...
        </reference>
    </characteristics>
    <rid>64.57.28.242</rid>
    <interfaces>
        ...
        <interface id="ge-4/0/0">
            <description>sw.manlan.internet2.edu:Te11/3</description>
            <mtu>9192</mtu>
            <unit number="102">
                <vlan-id>102</vlan-id>
                <ip type="ipv4" mask="31">198.32.11.50</ip>
                <ip type="ipv6" mask="64">2001:468:ff:15c5::1</ip>
            </unit>
            <unit number="112">
                <vlan-id>112</vlan-id>
                <ip type="ipv4" mask="30">198.124.216.158</ip>
            </unit>
        ...
        </interface>
    </interfaces>
    ....
</node>
<links>
    <intra>
      <link id="NY-WA">
            <node id="NY" if="so-0/0/0" unit="0"/>
            <node id="WA" if="so-0/0/0" unit="0"/>
            <attributes>
                <mtu>9180</mtu>
                <encapsulation>cisco-hdlc</encapsulation>
            </attributes>
      </link>
...
<link>
...
<bgp>
  ...
  <sessions>
    <external>
        <session id="NY-GEANT">
            <description>GEANT M160 via MANLAN 10GigE</description>
            <node id="NY"/>
            <peer id="GEANT" ip="198.32.11.51"/>
        </session>
...
  <sessions-sets>
  ...
    <sessions-set id="GEANT">
            <session id="NY-GEANT"/>
            <session id="WA-GEANT"/>
    </sessions-set>
...
```

- des règles exprimées par un ensemble de contraintes structurelles appliquées à la représentation à l'aide d'un schéma XML. Celles-ci sont appelées *Structural rule* ;
- des règles exprimées dans un langage de requêtes : XQuery. Celles-ci sont appelées *Query rule* ;
- des règles exprimées dans un langage de programmation : Java. Celles-ci sont appelées *Language rule*.

Dans le cadre de ce travail, de nombreuses règles ont été écrites. La liste complète de celles-ci est disponible en annexe B. L'entièreté de celles-ci ne s'applique pas au réseau étudié, car certaines vérifient des éléments de configuration absents d'*Abilene* (p.ex. les règles se rapportant aux réflecteurs de routes BGP). Un compte rendu du sous-ensemble de règles appliquées à la représentation d'*Abilene* est repris dans le tableau 8.1.

Comme le montre ce tableau, la majorité des règles testées sont des *Structural rule* (section 6.1). Ceci résulte du fait que ces règles sont utilisées afin de vérifier la structure de notre représentation. Par exemple, elles vérifient la présence d'un grand nombre d'éléments de configuration (p.ex. la présence de l'élément `constructor`). De plus, elles expriment la totalité des règles de symétrie (p.ex. l'encapsulation identique aux extrémités d'un lien) à l'aide du principe d'abstraction ainsi que la plupart des règles d'unicité (p.ex. l'unicité des *router id*) grâce à l'utilisation de clés. Néanmoins, elles ne sont pas utilisées pour exprimer des règles personnalisées, car elles manquent d'expressivité nécessaire à cette tâche.

Les *Query rule* (section 6.2) sont typiquement des règles écrites par un opérateur afin de vérifier que ses choix de *design* sont respectés dans l'entièreté du réseau. Ces *Query rule* peuvent exprimer n'importe quel type de règles. Par exemple, par leur intermédiaire, nous vérifions que les *next-hop* BGP sont joignables au sein d'*Abilene* et l'activation du protocole OSPF sur toutes les interfaces internes.

Enfin, les trois *Language rule* (section 6.3) sont utilisées dans des cas très spécifiques où l'utilisation d'un algorithme ou l'application de calculs avancés (p.ex. le calcul d'adresses IP) est nécessaire. Par exemple, dans *Abilene*, nous vérifions que la topologie est connexe ou encore que la panne d'un lien ne provoque pas la partition du *backbone* en deux sous-réseaux distincts. Ces règles sont généralement très spécifiques et sont, dans la majorité des cas, des règles personnalisées.

	Structural rule	*Query rule*	*Language rule*	**Total**
Présence	82	13	-	95
Non-présence	0	2	-	2
Unicité	14	6	-	20
Symétrie	10	-	-	10
Personnalisée	-	6	3	9
Total	106	27	3	**136**

TAB. 8.1 – Nombre de règles vérifiées sur la représentation de la topologie d'*Abilene* en fonction du type de règles et de la technologie utilisés

8.2.1 Ajout d'une nouvelle règle

Comme mentionné ci-avant, la tâche d'écriture des règles incombe généralement à l'opérateur. Cette section aura pour but d'illustrer la flexibilité de l'outil en montrant à quel point il est aisé d'ajouter une règle au sein de celui-ci. Pour ce faire, nous présenterons l'ajout d'une règle vérifiant un principe de *design* découvert lors de l'analyse des configurations d'*Abilene* : une adresse IPv6 doit être définie sur chaque lien interne.

Pour commencer, nous devons identifier le type de règles ainsi que la technique à utiliser. Concrètement, nous désirons vérifier la présence d'une adresse IPv6 sur chaque interface. Par conséquent, il s'agit là d'une règle de type *présence*. Au niveau de la technique, cette règle n'est pas vérifiable directement par le schéma, car celle-ci nécessite l'évaluation d'une condition. Dès lors, étant donné que nous n'avons pas besoin d'appliquer un algorithme pour vérifier cette simple condition, notre choix se porte logiquement vers une *Query rule*.

Une fois ce choix effectué, il est nécessaire d'identifier le SCOPE ainsi que ses `descendents` relatifs à la règle. Dans la représentation, les adresses IPv6 sont définies comme des sous-éléments de l'élément `unit`. Dès lors, le SCOPE de la règle correspond à l'ensemble des éléments `unit` qui appartiennent à un lien interne. Les `descendents` du SCOPE correspondent à l'ensemble des éléments `ip`. Parmi ceux-ci, il doit y en avoir au moins un qui possède l'attribut `type` égal à 'ipv6'. La règle correspondante est donnée dans le listing 8.2.

Listing 8.2 – Représentation de la règle vérifiant la présence d'une adresse IPv6

```
<rule id="ALL_INTERNAL_UNITS_HAVE_A_IPV6" type="presence">
  <description>All internal units must have one IPv6 adress</description>
  <presence>
    <scope>ALL_INTERNAL_UNITS</scope>
    <descendents>ip</descendents>
    <condition>@type='ipv6'</condition>
  </presence>
</rule>
```

Une fois la règle exprimée et insérée dans le graphe de dépendances (section 4.4), l'opérateur est assuré que, dans l'ensemble des configurations générées par le logiciel, toutes les interfaces internes disposent d'au moins une adresse IPv6.

8.2.2 Détection d'erreurs

Dans cette section, nous introduirons de façon délibérée des erreurs dans la représentation afin d'illustrer le comportement de l'outil en présence de celles-ci.

Trois violations aux choix spécifiques d'*Abilene* seront introduites. Premièrement, nous désactiverons le protocole OSPF d'une interface *loopback* donnée. Ensuite, nous désactiverons OSPF de l'extrémité d'un des liens internes. Enfin, nous enlèverons une politique *next-hop-self* d'un des routeurs BGP.

Interface *loopback* non annoncée dans OSPF

Un bon choix de *design* utilisé dans *Abilene* est de configurer les sessions iBGP à l'aide des adresses *loopback* des routeurs. Cela est dû au fait que les interfaces *loopback* sont toujours activées. Dès lors, les sessions iBGP correspondantes resteront actives tant qu'il y a une route vers l'adresse *loopback*, et ce, malgré des changements dans la topologie [KD02]. Néanmoins, pour que ce mécanisme puisse être utilisé, il faut évidemment que les routeurs sachent comment rejoindre les adresses *loopback* des autres routeurs. Dans *Abilene*, cela est réalisé en exportant ces adresses au sein du protocole de routage intradomaine OSPF.

Notons que, si par mégarde l'adresse *loopback* d'un des routeurs n'est pas annoncée dans OSPF, celui-ci se retrouve dans l'incapacité d'établir des sessions iBGP. Dans cet exemple, nous désactivons l'exportation de l'adresse *loopback* du routeur ST.

Afin de s'assurer que ce principe est respecté, il est nécessaire d'écrire une règle qui le vérifie. Pour ce faire, nous avons ajouté la règle de *présence* : `LOOPBACK_ADDRESS_ANNOUNCED_IN_OSPF`. La désactivation de l'exportation d'une adresse *loopback* implique l'affichage du message repris dans le listing 8.3.

Listing 8.3 – Erreur retournée lorsque l'interface *loopback* n'est pas annoncée dans OSPF

```
-----LOOPBACK_ADDRESS_ANNOUNCED_IN_OSPF failed : Summary -----
821    No element of ST/interfaces/interface has substring(@id,1,2)='lo'
```

Cela correspond au comportement escompté. Dans le message d'erreur, 821 correspond à la ligne où l'interface devrait être présente dans la représentation XML. La fonction `substring()` est utilisée afin de retourner les deux premiers caractères de l'identifiant de l'interface. Dans notre représentation, si ces deux derniers sont égaux à `lo`, il s'agit d'une interface *loopback*.

Le message d'erreur est assez explicite pour qu'un opérateur comprenne immédiatement le problème et le corrige facilement.

Désactivation du protocole OSPF sur une des extrémités d'un lien

Dans cet exemple, nous considérons deux routeurs reliés par un lien interne sur lequel le protocole OSPF est activé. Afin de fonctionner, le protocole OSPF se doit d'être activé sur les deux côtés du lien, sinon une adjacence ne peut pas être formée [Moy98]. Les conséquences de cet oubli peuvent être importantes si, par exemple, ce lien est le seul qui connecte un routeur au *backbone*. En effet, le routeur peut se retrouver isolé (bien que connecté physiquement), car il ne connaîtra pas les routes nécessaires pour joindre le reste du *backbone*.

L'écriture d'une règle s'assurant de l'application de ce principe est nécessaire. Dans ce cas, il faut passer par l'écriture d'une règle *personnalisée* vérifiée à l'aide du langage de requêtes XQuery. Il s'agit donc d'une *Query rule*. Après quoi, si nous désactivons OSPF du côté du routeur NY sur le lien qui le relie (via l'interface `so-0/0/0.0`) au routeur WA (via l'interface `so-0/0/0.0`), le logiciel le détecte et l'indique via le message d'erreur repris dans le listing 8.4.

Listing 8.4 – Erreur retournée lorsque OSPF n'est pas activé sur un des deux côtés d'un lien

```
The active interface so-0/0/0 of node WA is not connected to any active OSPF interface !
It means that none of the interfaces connected to so-0/0/0 are running OSPF or if they run
OSPF, none of them is present in area: 0.0.0.0 !
```

Comme prévu, l'outil a détecté l'erreur. Remarquons également que cette dernière est également retournée lorsque les deux côtés d'un lien sur lequel tourne OSPF ne se trouvent pas dans la même *area*.

Suppression d'une politique de *next-hop-self*

Un routeur *bordure*[3] redistribue les routes apprises sur une session eBGP aux autres routeurs du domaine par l'intermédiaire des sessions iBGP. Cependant, des mesures particulières doivent être prises afin de s'assurer que les routes redistribuées soient joignables par les autres routeurs du domaine [KD02]. En effet, par défaut, l'attribut *next-hop* des routes BGP n'est pas modifié. Cet attribut correspond généralement à une adresse IP tombant dans l'espace d'adresses alloué au lien externe. Plusieurs solutions permettent de s'assurer que ces routes sont joignables dans tout le domaine, citons entre autres :

- placer des politiques dites de *next-hop-self* qui vont forcer les routeurs *bordures* à modifier l'attribut *next-hop* par leur propre adresse avant de propager la route aux autres routeurs ;
- annoncer les préfixes associés aux liens externes dans OSPF.

Au sein d'*Abilene*, la première solution est la seule utilisée. Dès lors, lorsque nous désactivons la politique *next-hop-self* sur un routeur, l'ensemble des routes redistribuées par ce dernier risque d'être injoignables pour le reste du domaine, car l'adresse du *next-hop* n'est pas connue par les autres routeurs du réseau. Le message d'erreur retourné par l'outil lorsque la politique est désactivée sur le routeur *NY* est représenté dans le listing 8.5.

Listing 8.5 – Erreur retournée lorsque l'attribut *next-hop* de BGP n'est pas joignable dans tout le domaine

```
Node NY has not 'next-hop-self' activated on iBGP sessions, thus routes announced on ebgp
session 'NY-GEANT' might be unreachable inside the as, please activate the 'next-hop-self' or
put interface 'ge-4/0/0.102' inside OSPF (as passive) !
```

À nouveau, l'erreur est correctement détectée. Deux solutions sont proposées. La première est d'insérer les interfaces correspondantes dans la topologie OSPF en prenant bien soin de les configurer dans l'état passif, et ce, afin qu'aucune adjacence ne se forme avec un routeur d'un domaine voisin. La seconde est d'instaurer une politique *next-hop-self* en exportation sur les sessions iBGP. Notons que si l'opérateur active OSPF sur les interfaces externes, mais qu'il oublie de les configurer dans l'état passif, alors une erreur est également générée.

[3] Un routeur bordure est un routeur connecté à un routeur appartenant à un autre AS.

8.3 Matrices des politiques BGP

Dans cette section, nous présenterons deux utilisations possibles des abstractions définies dans la section 5.2.6. Une première matrice est utilisée afin de définir les politiques de transit du domaine. Une seconde permet de modifier le comportement d'exportation par défaut des routes.

Chaque sous-section débutera par un bref rappel de la matrice et de son utilité. Ensuite, à l'aide de chacune d'elles, nous modéliserons un sous-ensemble des politiques interdomaines d'*Abilene*. De plus, quelques extraits de configurations générées seront inclus afin d'illustrer les résultats obtenus à partir de ces représentations de haut niveau.

Enfin, dans la dernière section, nous présenterons comment la matrice de filtrage peut être validée à l'aide d'un nouveau type de règles : `matrix`.

8.3.1 Matrice de filtrage

Au sein de la section 5.2.6, nous avons proposé une abstraction permettant à un opérateur de définir plus facilement ses politiques d'exportation. Cette abstraction se base sur une matrice afin de modéliser le transit des routes au travers du domaine considéré.

Après la phase de rétro ingénierie, un sous-ensemble des politiques utilisées par *Abilene* a été découvert. Celles-ci sont représentées dans le tableau 8.2. Pour rappel, les lignes et colonnes de cette matrice F représentent des ensembles de sessions eBGP. Lorsque deux ensembles A et B sont reliés par le symbole $\checkmark (F_{AB} = \checkmark)$, les routes apprises sur les sessions appartenant à l'ensemble A sont annoncées sur les sessions appartenant à B.

	GEANT	NREN	ESNET	AMPATH	REDCLARA	USGS
GEANT	-	✓	✓	-	-	-
NREN	✓	-	-	-	-	-
ESNET	✓	-	-	✓	-	-
AMPATH	-	-	✓	-	-	-
REDCLARA	-	-	-	-	-	✓
USGS	-	-	-	-	✓	-

Tab. 8.2 – Exemple d'utilisation de la matrice des politiques BGP

Au sein du tableau 8.2, l'ensemble GEANT correspond aux deux sessions eBGP qui relient le *backbone* d'*Abilene* au réseau européen *Geant*. Sur la figure 8.1, nous voyons que ces sessions sont définies sur le routeur NY et WA. La composition des autres groupes s'obtient de façon semblable. La représentation de la matrice au sein de notre système est donnée dans le listing 8.6. Les politiques expriment notamment le fait que les routes annoncées sur les sessions appartenant à GEANT sont redistribuées aux sessions appartenant aux groupes NREN et ESNET. L'application

de cette politique peut être vérifiée sur le routeur WA qui dispose de connexions vers ces deux groupes. La configuration générée de ces deux sessions sur le routeur WA est repris dans le listing 8.7.

Listing 8.6 – Représentation de la matrice de filtrage en XML

```
<filter-matrix>
    <in-groups>
        <in-group id="GEANT">
            <out-groups>
                <out-group id="ESNET"/>
                <out-group id="NREN"/>
            </out-groups>
        </in-group>
        <in-group id="NREN">
            <out-groups>
                <out-group id="GEANT"/>
            </out-groups>
        </in-group>
        <in-group id="ESNET">
            <out-groups>
                <out-group id="GEANT"/>
                <out-group id="AMPATH"/>
            </out-groups>
        </in-group>
        <in-group id="AMPATH">
            <out-groups>
                <out-group id="ESNET"/>
            </out-groups>
        </in-group>
        <in-group id="REDCLARA">
            <out-groups>
                <out-group id="USGS"/>
            </out-groups>
        </in-group>
        <in-group id="USGS">
            <out-groups>
                <out-group id="REDCLARA"/>
            </out-groups>
        </in-group>
    </in-groups>
</filter-matrix>
```

Comme prévu, sur chaque session vers NREN et ESNET, la politique *SEND-COMMUNITY-GEANT* est présente. Cette politique exprime simplement le fait que l'ensemble des routes étiquetées avec la communauté BGP `GEANT` sont acceptées. Placer cette politique au niveau d'un filtre d'exportation d'une session signifie donc exporter l'ensemble de ces routes sur cette session.

Notons également la génération automatique d'un certain nombre de règles de « bonnes pratiques » (section 7.2.1). Ainsi, pour chaque filtre d'importation (instruction **`import`**), la politique `SANITY-IN` est appliquée. Cette politique s'assure de la non-importation des routes suivantes :

- routes dont l'AS-PATH contient un numéro d'AS privé ;
- routes *martiennes* (*i.e.* les routes qui ne doivent jamais être annoncées sur Internet)
- routes dont la longueur du préfixe est supérieur à 27 bits ;
- routes dont le préfixe associé est un préfixe local au domaine considéré.

Quant à la politique d'exportation `ORIGINATE_LOCAL`, celle-ci s'assure que seuls les agrégats correspondant aux adresses locales seront annoncés sur une session eBGP.

Listing 8.7 – Configuration des sessions eBGP sur WA vers NREN et ESNET

```
group WASH-NREN {
    description NREN-Goddard via NGIX;
    peer-as 24;
    export [ SEND-COMMUNITY-GEANT ORIGINATE_LOCAL ];
    import [ SANITY-IN SET-COMMUNITY-NREN ];
    neighbor 198.32.11.22;
}
group WASH-ESNET {
    description ESNET via NGIX;
    peer-as 293;
    export [ SEND-COMMUNITY-GEANT SEND-COMMUNITY-AMPATH ORIGINATE_LOCAL ];
    import [ SANITY-IN SET-COMMUNITY-ESNET ];
    neighbor 198.124.194.9;
}
...
policy-statement SEND-COMMUNITY-GEANT {
    term one {
        from community GEANT;
        then accept;
    }
}
```

8.3.2 Matrice des communautés

Cette matrice (section 5.2.6) permet de définir des politiques qui modifient le comportement d'exportation par défaut des routes étiquetées avec une certaine communauté donnée transitant au travers d'un réseau. Ces types de politiques sont souvent utilisés par les fournisseurs d'accès Internet afin de permettre à leurs clients un traitement différencié de leurs routes en fonction de la communauté BGP des routes annoncées. *Abilene* ne déroge pas à la règle et utilise également ce principe. Pour information, la liste des communautés reconnues par *Abilene* est disponible sur [2].

Étant donné que notre matrice ne gère, pour le moment, que les politiques d'exportation, la seule politique de ce type modélisée est celle associée à la communauté BLOCK-TO-COMMERCIAL (valeur 11537 : 2002). Les routes étiquetées avec cette communauté transitant dans *Abilene* ne seront pas exportées aux voisins appartenant au groupe COMMERCIAL. La représentation de la matrice est reprise dans le listing 8.8.

Listing 8.8 – Représentation de la matrice des communautés en XML

```
<community-matrix>
    <in-communities>
        <in-community id="BLOCK-TO-COMMERCIAL" type="reject">
            <out-groups>
                <out-group id="COMMERCIAL"/>
            </out-groups>
        </in-community>
    </in-communities>
</community-matrix>
```

Le routeur *ST* (figure 8.1) dispose de deux sessions vers Microsoft. Ces deux sessions appartiennent à l'ensemble COMMERCIAL. Les configurations générées de ces deux sessions sont reprises dans le listing 8.9. Une comparaison entre celles-ci et la configuration réelle correspondante est donnée dans l'annexe C.

Listing 8.9 – Configuration des sessions eBGP sur *ST* vers Microsoft

```
group STTLng-MICROSOFT {
    description Microsoft via Pac Wave vlan706;
    peer-as 8075;
    export [ APPLY-BLOCK-TO-COMMERCIAL ORIGINATE_LOCAL REJECT-REST ];
    import [ SANITY-IN SET-COMMUNITY-COMMERCIAL ];
    neighbor 207.231.240.7;
}
group STTLng-MICROSOFT2 {
    description Microsoft via Pac Wave vlan776;
    peer-as 8075;
    export [ APPLY-BLOCK-TO-COMMERCIAL ORIGINATE_LOCAL REJECT-REST ];
    import [ SANITY-IN SET-COMMUNITY-COMMERCIAL ];
    neighbor 207.231.241.7;
}
...
policy-statement APPLY-BLOCK-TO-COMMERCIAL {
    term filter {
        from community BLOCK-TO-COMMERCIAL;
        then reject;
    }
}
...
community BLOCK-TO-COMMERCIAL members 11537:2002;
```

Comme prévu, la politique `APPLY-BLOCK-TO-COMMERCIAL` est appliquée en sortie sur les deux sessions vers Microsoft. Cette dernière rejette simplement l'ensemble des routes étiquetées avec la communauté `BLOCK-TO-COMMERCIAL`. Placée dans un filtre d'exportation, elle bloque l'exportation de toutes ces routes.

8.3.3 Validation des politiques interdomaines

Dans la section 6.5, nous avons montré comment un utilisateur peut ajouter un nouveau type de règles dans notre logiciel. Le nouveau type de règles introduit, `matrix`, permet à un opérateur de valider sa matrice de filtrage.

En réalité, pour utiliser ce nouveau type de règles sur le réseau *Abilene*, les seuls changements à apporter concernent le nom des ensembles des sessions eBGP. Dans le listing 8.10, nous reprenons quelques exemples de ces règles appliquées à la matrice décrite dans le listing 8.6. Ces règles vérifient que les routes annoncées sur chaque session appartenant à l'élément `in-set` sont effectivement exportées sur toutes les sessions appartenant aux éléments `out-set` et uniquement à eux.

Pour illustrer l'utilité de ces règles, nous introduisons une erreur en redistribuant les routes provenant de l'ensemble de sessions GEANT à l'ensemble REDCLARA (*i.e.* $F_{\text{GEANT,REDCLARA}} = \checkmark$). Cette politique viole la règle `GEANT_ROUTES_EXPORT` et retourne l'erreur donnée dans le listing 8.11.

Listing 8.10 – Exemple de règles validant la matrice de filtrage BGP

```
<rule id="GEANT_ROUTES_EXPORT" type="matrix">
    <description>Routes coming from GEANT peerings must only be
                announced to NREN and ESNET peerings</description>
    <matrix>
        <in-set>GEANT</in-set>
        <out-set>NREN</out-set>
        <out-set>ESNET</out-set>
    </matrix>
</rule>
<rule id="NREN_ROUTES_EXPORT" type="matrix">
    <description>Routes coming from NREN peerings must only be
                announced to GEANTlisting peerings</description>
    <matrix>
        <in-set>NREN</in-set>
        <out-set>GEANT</out-set>
    </matrix>
</rule>
....
```

Listing 8.11 – Erreur retournée lorsqu'une erreur est détectée dans la matrice de filtrage BGP

```
----- GEANT_ROUTES_EXPORT failed : Summary -----
Exportation error found in the filter-matrix for GEANT sessions
Please check that those routes ARE announced to NREN or ESNET and ONLY to them !
```

Nous voyons donc qu'avec l'aide de quelques règles (*i.e.* une par ligne de la matrice), un opérateur peut facilement valider sa matrice afin de vérifier que celle-ci décrit bien les politiques voulues. En outre, il peut se prémunir des éventuelles modifications qui iraient à l'encontre de celles-ci. Notons que lors de l'ajout d'éléments dans la matrice (p.ex. une nouvelle ligne), des règles supplémentaires vérifiant la validité de ceux-ci doivent être ajoutées.

8.4 Conclusion

Dans cette chapitre, nous avons montré qu'une approche orientée *validation*, dans laquelle un opérateur s'assure de la justesse de son réseau avant le déploiement de celui-ci, est utile et permet la découverte de problèmes difficilement détectables par un être humain. Cette approche prend tout son sens dans un réseau de grande taille.

Tout d'abord, nous avons montré que notre logiciel était capable de représenter un réseau réel, dans ce cas-ci, le réseau *Abilene*.

Ensuite, d'un point de vue validation, plus d'une centaine de règles ont été appliquées sur la représentation de haut niveau. Un opérateur peut facilement en rajouter afin, par exemple, de tester une fonctionnalité particulière ou un nouveau protocole. Aussi, des fautes ont été introduites délibérément afin d'illustrer le processus de détection des erreurs par le logiciel. Comme nous l'avions supposé, l'outil les a détectées.

De plus, au niveau de l'écriture des politiques interdomaines, deux utilisations possibles des abstractions proposées ont été décrites. Ces descriptions ont permis de modéliser certaines politiques interdomaines d'*Abilene*. Nous avons également proposé une utilisation des règles de type `matrix` pour valider ces matrices.

Enfin, afin de montrer que notre logiciel était capable de générer des configurations interprétables par des équipements réseaux (p.ex. des routeurs), des extraits des configurations générées sous JunOS ont été proposés.

Chapitre 9

État de l'Art

Dans ce chapitre, nous présenterons un résumé des principaux travaux effectués ou en cours de réalisation dans le domaine de la gestion et de la validation de configuration de réseaux. Trois grandes parties seront détaillées : quelques solutions standardisées, des travaux de recherche ainsi que deux solutions commerciales. Étant donné que les deux dernières parties traitent de problématiques similaires à la notre, nous tâcherons de dresser une comparaison entre chacun de ces travaux de celles-ci et notre approche.

La première section abordera des solutions standardisées dont certaines ont été proposées à l'*Internet Engineering Task Force* (IETF), l'organisme de standardisation mondial des protocoles Internet, à savoir :

Simple Network Management Protocol (SNMP) : un protocole simple qui permet une gestion centralisée d'un réseau ;

Netconf : un protocole de configuration de réseaux basé sur les technologies XML ;

Routing Policy Specification Language (RPSL) : un langage de spécification qui autorise la définition de politiques de routage à un haut niveau ;

Policy-based Network Management (PBNM) : une méthode de gestion de réseaux basée sur des politiques.

La deuxième section présentera quelques résultats de travaux de recherche dans le domaine de la validation de configuration. Deux approches seront décrites : une approche orientée « règles » ainsi qu'une approche orientée « *data mining* ». La première valide une configuration à l'aide d'un ensemble de règles qui spécifient la validité d'une configuration. La seconde se base sur des configurations existantes et sur des mécanismes d'intelligence artificielle afin de construire un modèle[1] du réseau. Une fois le modèle obtenu, toute instruction s'écartant trop de celui-ci peut être considérée comme une erreur potentielle.

La troisième et dernière section abordera deux solutions commerciales : OPNET Netdoctor et Wandl NPAT. Ces logiciels permettent, tout comme les travaux précédents, de détecter des erreurs au sein de configurations existantes, mais également d'aider dans le processus de *design* d'un réseau.

[1] Ce modèle doit être vu comme une « moyenne » de l'ensemble des configurations qui composent le réseau.

9.1 Solutions standardisées

Nous présenterons dans cette section des solutions plus anciennes dans le sens qu'elles ont déjà été étudiées en détails. Dès lors, les avantages et faiblesses de chacune sont connus. Quatre solutions aidant à la gestion du réseau sont résumées : *Simple Network Management Protocol* (SNMP), Netconf, *Routing Policy Specification Language* (RPSL) et *Policy Based Network Management* (PBNM).

9.1.1 *Simple Network Management Protocol*

Dès la fin des années 1970, avec l'accroissement rapide de la taille des réseaux, il est vite apparu que les solutions simples de gestion de réseaux telles que la configuration en ligne de commande[2] ou l'utilisation du protocole *Internet Control Message Protocol* (ICMP) n'étaient plus viables à long terme [Sta93]. Il était clair que des mécanismes davantage centralisés devaient être mis en place. C'est ainsi que SNMP [CFSD90] fût créé en 1988. Du fait de sa simplicité, bon nombre de vendeurs l'implémentèrent si bien qu'encore aujourd'hui SNMP est largement utilisé. Les objectifs principaux de SNMP sont :

- l'automatisation du processus de configuration et de contrôle des éléments du réseau ;
- l'automatisation du processus de supervision des performances du réseau ;
- l'automatisation de la gestion des erreurs au sein du réseau.

Les deux derniers points correspondent à l'automatisation de la tâche de surveillance ou de *monitoring* d'un réseau.

La structure de haut niveau de SNMP est composée de deux entités principales : des systèmes gestionnaires (*Network Management System*) situés sur des stations de gestion et des agents localisés sur les différents composants du réseau (routeur, commutateur, etc.). Chaque agent maintient une base d'informations appelée *Management Information Base* (MIB) qui contient toutes les informations stockées sur l'équipement auquel il est associé. Le gestionnaire questionne l'agent par l'intermédiaire de requêtes auxquelles l'agent répond avec l'information demandée. Un agent doit être capable d'envoyer des informations non sollicitées (*i.e.* de façon asynchrone) au gestionnaire lorsque quelque chose d'important s'est produit (message *trap*). SNMP fonctionne au-dessus d'UDP et se doit donc de gérer lui-même la fiabilité de la connexion (à l'aide de temporisateurs).

En raison de sa standardisation, SNMP a l'avantage d'être interopérable et d'être supporté par la grande majorité des vendeurs. Cependant, il possède certaines faiblesses inhérentes à la gestion des configurations. En effet, en raison de l'utilisation d'une abstraction de bas niveau (MIB), le développement d'applications utilisant SNMP devient vite laborieux. Par conséquent, le nombre de développeurs SNMP expérimentés est très faible, ce qui implique que la plupart des outils disponibles permettant d'implémenter des applications de gestion au-dessus de SNMP sont assez « primitifs » [SPMF03]. De plus, SNMP est conçu de manière à assurer le plus possible son indépendance envers d'autres services réseaux, par conséquent, les fonctionnalités disponibles sont des fonctionnalités de bas niveau. Le problème est que l'environnement réseau a fortement

[2] le plus souvent au-dessus d'une connexion TELNET

changé depuis la conception de SNMP et que les équipements actuels sont capables d'exécuter des opérations beaucoup plus complexes à moindre coût. Pour toutes ces raisons, SNMP est, à l'heure actuelle, beaucoup plus utilisé pour des tâches simples de *monitoring* que pour la configuration de réseau.

9.1.2 *Netconf*

À cause des problèmes de SNMP évoqués ci-dessus, l'IETF a monté, en 2003, un groupe de travail dont l'objectif était d'établir un nouveau protocole de configuration de réseaux. Le résultat de leur travail est le protocole *Netconf* [Enn06] qui utilise le langage de balisage XML [PSMY+06]. Ce protocole se base sur un modèle client-serveur et offre la possibilité à une entité « gérant » (ou *manager*) de manipuler la configuration d'équipements réseaux en leur envoyant des requêtes XML. *Netconf* utilise un modèle divisé en couches logiques. Sa caractéristique principale est qu'il est construit au-dessus d'une couche RPC (*Remote Procedure Call*) et qu'il peut utiliser plusieurs protocoles applicatifs pour fonctionner : SSH, SOAP/HTTP ou BEEP.

Netconf présente plusieurs avantages. Il a la capacité d'assurer une certaine sécurité de l'échange grâce à l'utilisation de mécanismes d'identification et également d'encryption lorsque, par exemple, le protocole SSH est utilisé. De plus, il fournit des services avancés comme le fait de pouvoir annuler des changements effectués. L'encodage XML utilisé par les messages (requêtes, réponses) facilite le traitement des données.

9.1.3 *Routing Policy Specification Language*

Internet peut être vu comme une gigantesque interconnexion de réseaux, chacun de ceux-ci est géré par une administration particulière et définit un domaine autonome ou *Autonomous System* (AS). Généralement, pour rejoindre une destination située dans un autre AS, plusieurs AS de transit doivent être utilisés. Or, il est du ressort de chaque AS de définir quels autres AS sont autorisés à utiliser ses ressources. Cette façon de procéder amène à la notion de relation entre AS et à différents types de connexion entre ceux-ci (p.ex. client-fournisseur). Ces relations particulières sont spécifiées à l'aide de politiques de routage.

Configurer les politiques de routage d'un routeur est une tâche ardue, dans [Ala96], l'auteur identifie les politiques les plus couramment utilisées sur Internet et présente un langage de spécification de haut niveau, RPSL [BDPR05]. Le but de RPSL est de fournir un moyen simple de configurer des politiques inter-domaines tout en facilitant les ajouts et modifications ultérieurs (*scalabality*). Des exemples d'utilisation concrète de RPSL sont repris dans [MSO+99].

RPSL exprime les politiques à l'aide de différents d'objets, par exemple, chaque route est spécifiée dans un objet de type *route* et chaque AS dans un objet de type *aut-num*. Il est possible de regrouper des objets au sein d'ensembles (p.ex. l'ensemble des AS clients d'un autre AS). Une politique RPSL définit les relations entre des objets.

Notons qu'il est possible de générer, à partir de politiques écrites en RPSL, des configurations de bas niveau directement utilisables par les routeurs (p.ex. Cisco). Un des outils permettant cela est Rtconfig [Ala96].

9.1.4 *Policy Based Network Management*

Une technique de gestion, qui a fait l'objet de nombreuses recherche, est la gestion d'un réseau basée sur des politiques. Le but ultime de PBNM est une gestion du réseau vu comme un système unifié. Ses principales motivations sont les suivantes :

- faciliter l'implémentation des mécanismes de qualité de service au sein du réseau ;
- simplifier la gestion des équipements, du réseau et des services fournis par ce dernier ;
- diminuer le nombre de personnes « qualifiées » requises afin de configurer un réseau ;
- définir le comportement du réseau vu comme un système distribué ;
- gérer la complexité toujours croissante des équipements ;
- guider la configuration du réseau grâce à quelques règles de haut niveau.

Les politiques utilisées par un système PBNM sont exprimées à différents niveaux d'abstraction, par exemple, autoriser tel utilisateur à utiliser le réseau (politique de haut niveau) ou régler un paramètre précis sur des interfaces particulières (politique de bas-niveau). La philosophie poursuivie est de définir le comportement global du réseau (p.ex. en termes de services) au niveau le plus élevé. Ensuite, les politiques sont traduites[3] en termes compris par le niveau inférieur et ainsi de suite jusqu'au niveau le plus bas qui correspond aux opérations à réaliser sur les équipements. Chaque niveau possède son propre langage ainsi que de sa propre modélisation (*i.e.* représentation de l'information).

L'architecture d'un système PBNM est composée de plusieurs entités distinctes, chacune jouant un rôle particulier. Ses principaux éléments sont :

Interface d'entrée : composant utilisé par l'administrateur pour lister, ajouter, modifier, supprimer et chercher des politiques ou des informations reliées ;

Dépôt de politiques : élément passif contenant les politiques ;

Serveur de politiques : ensemble de composants qui prennent les décisions et exécutent les actions correspondantes sur un ou plusieurs équipements du réseau. Il est constitué de deux composants principaux :

- **Logique de décision :** ensemble d'éléments responsables de traduire les politiques dans le niveau d'abstraction approprié, de valider celles-ci, ainsi que de vérifier si elles n'introduisent pas de conflits au sein du système. Un des éléments les plus importants de cet ensemble est le *Policy Decision Point* (PDP) qui décide quand et où il doit appliquer une politique ;
- ***Knowledge interface* :** élément qui fournit des informations décrivant les capacités de chaque élément du réseau.

***Policy Broker* :** élément contrôlant la façon dont les différents serveurs de politiques interagissent entre eux ;

[3] Ce mécanisme de traduction est appelé *Policy Continuum*.

***Policy Proxy* :** élément utilisé en tant que *proxy* lorsqu'un élément du réseau ne peut directement communiquer avec un PDP.

Le principal avantage de PBNM est qu'il permet une gestion de très haut niveau du réseau. Au travers des différentes abstractions utilisées, il permet de communiquer aisément avec n'importe quel type d'équipement, que celui-ci utilise une interface en ligne de commande ou le protocole *Netconf*. Par contre, au vu du nombre d'éléments nécessaires à sa mise en oeuvre, l'installation d'un système PBNM peut s'avérer assez contraignante.

9.2 Résultats de recherches

Dans cette section, nous détaillerons quelques résultats de recherche dans le domaine de la validation de configuration de réseaux. Deux approches principales sont pour le moment en phase d'investigation. La première est une approche orientée « règles » dans laquelle la représentation d'un réseau est validée par rapport à un ensemble de conditions. La seconde approche orientée « *data mining* » est radicalement différente et tente, grâce à des techniques issues du domaine de l'intelligence artificielle, de concevoir automatiquement un modèle d'une configuration « correcte ». Toute déviation par rapport au modèle constitue une erreur potentielle.

9.2.1 Approche orientée « règles »

Cette approche utilise un processus de validation basé sur des spécifications de haut niveau. Ces spécifications doivent être fournies *a priori* au système. Nous remarquons qu'à l'heure actuelle, il existe plus de travaux se basant sur cette approche que sur l'approche *data mining* (section 9.2.2). Cinq de ces travaux sont détaillés dans cette section. Les deux premiers valident les configurations d'un point de vue sécurité ; le troisième valide les configurations BGP des routeurs, alors que les deux dernière sont construits à partir d'une représentation de haut niveau (*i.e.* une abstraction) du réseau.

Validation de la sécurité

Dans le domaine plus spécifique de la sécurité des réseaux, un des outils les plus connus de validation de configuration est nommé *Router Audit Tool* (RAT) [Geo02]. RAT permet de valider des configurations basées sur le langage IOS de Cisco Systems. Pour arriver à ses fins, l'outil télécharge, dans un premier temps, l'ensemble des configurations du domaine. Ensuite, il teste chaque fichier par rapport à un ensemble de règles prédéfinies. Pour chaque configuration, RAT produit un rapport indiquant les règles vérifiées, leur statut respectif (passé/échoué), un score global sur 10 et une liste de commandes sous IOS permettant de régler les éventuels problèmes détectés. Les règles sont *text-based*, c'est-à-dire qu'elles se basent uniquement sur le texte contenu dans les configurations en vérifiant la présence ou l'absence de certaines chaînes de caractères.

L'outil *Cisco Router Configuration Diligent Evaluator* (Crocodile) [PS03] suit le même principe que RAT et permet, lui-aussi, de valider des configurations de routeurs, toujours d'un point

de vue sécurité. Crocodile part également des configurations existantes, vérifie si celles-ci ne contiennent pas de vulnérabilités connues (*i.e.* celles spécifiées par les règles) et génère ensuite un rapport détaillant les résultats.

Cependant, Crocodile se veut plus puissant que RAT. En effet, il permet, outre une analyse syntaxique des fichiers, l'analyse sémantique de ceux-ci (p.ex. des règles qui dépendent du contexte). L'architecture de la partie vérification est composée de plusieurs modules, chacun étant responsable d'une certaine catégorie de vérification (p.ex. authentification, SNMP, etc.). Chaque module produit comme résultat une « vue logique » qui reprend les aspects testés ainsi que leur statut. Grâce à cette modularité, l'outil est capable de traiter des configurations provenant d'autres vendeurs que Cisco Systems. Le papier mentionné ci-dessus [PS03] montre également des résultats produits par l'outil sur des fichiers « types ». De plus, les auteurs ont testé leur implémentation sur des configurations existantes avec, dans la plupart des cas, la découverte d'erreurs jusque-là non décelées.

Comparaison La philosophie suivie par ces deux outils diffère de notre approche dans le sens qu'ils valident des configurations existantes. Notre solution permet de générer des configurations validées. Un autre point important est le fait que RAT et Crocodile se concentrent sur l'aspect sécurité des configuration, là où notre travail n'est pas restreint à un domaine particulier.

Validation des configurations BGP

L'outil proposé par Feamster et Balakrishnan [FB05] est nommé *Router Configuration Checker* (RCC). Celui-ci est en mesure de découvrir des erreurs dans les configurations BGP des routeurs. RCC part des configurations existantes afin de construire une représentation intermédiaire qu'ils interrogent, à l'aide d'un langage relationnel, afin de vérifier sa validité.

RCC détecte deux grandes classes d'erreurs : des erreurs de validité, dans lesquelles un routeur peut apprendre des routes qui ne sont pas des chemins réellement utilisables, et des erreurs de visibilité, dans lesquelles un routeur n'est pas au courant de toutes les routes mises à sa disposition au sein du réseau. Même si RCC vérifie certains éléments relatifs à d'autres protocoles (quelques règles testent certains aspects des IGP), il reste principalement focalisé sur les configurations BGP. Les auteurs ont analysé, avec leur solution, les configurations de quelques 17 AS et ont trouvé de nombreuses erreurs dont certaines provoquaient des pannes passées inaperçues jusque-là.

Comparaison RCC suit une approche très similaire à la notre, à savoir, la création d'une représentation de haut niveau ainsi qu'une validation utilisant un langage relationnel. Cependant, le but premier de RCC est de vérifier les configurations BGP d'un routeur. Notre approche est donc plus générale. De plus, dans leur travaux, les auteurs ne mettent pas l'accent sur l'ajout aisé de nouvelles règles alors que c'est un de nos objectifs principaux.

Validation reposant sur une abstraction

Une autre approche intéressante est celle adoptée par NetML ou *Network Markup Language* [Iva04]. Il s'agit d'un langage basé sur XML et permettant de décrire des réseaux à un haut niveau. Un tel langage apporte une simplification dans le *design* d'un réseau, car l'accent est dorénavant mis sur les aspects conceptuels (p.ex. liens, protocoles, etc.) plutôt que sur des problèmes de configurations. L'outil basé sur NetML est capable de générer des configurations *concrètes* supportées par différents vendeurs tels que Cisco Systems, Juniper ou encore par le programme Zebra [Kun02], un logiciel de routage basé sur Linux. NetML ne valide pas la configuration représentée. Par contre, il génère des *scripts* au format Netkit [CBH+03] qui permettent de tester immédiatement le réseau dans un environnement virtuel. Le travail de vérification est donc effectué *a posteriori*.

Le travail proposé par Matuska dans *Metaconfiguration of the computer network* (metaconfig) [Mat04] part du même constat : il est crucial de configurer son réseau comme un système intégré et non comme un ensemble de systèmes autonomes. L'approche proposée permet de représenter la configuration des différentes fonctionnalités présentes dans l'entièreté du réseau plutôt que ceux des équipements particuliers. L'auteur propose une représentation de haut niveau du réseau. Cela présente plusieurs avantages comme supprimer la redondance au sein des configurations et révéler les relations entre les systèmes, jusque-là implicites. La « meta-représentation » est écrite en XML et est contrainte par un schéma (*i.e.* une grammaire qui définit la structure d'un document XML). Il est possible, à partir de la représentation de haut niveau, de générer les configurations propres à chaque constructeur.

La partie validation est réalisée par l'intermédiaire de deux types de règles : les *hard rules* (est-ce que la représentation permet de générer un réseau fonctionnel ?) et les *soft rules* (est-ce que le réseau généré est optimisé ?). Ces règles sont implémentées grâce à deux techniques : d'une part, l'utilisation de technologies XML (p.ex. XPath, Schématron, etc.) et d'autre part, l'utilisation de langages de programmation (p.ex. Java, C, C++, etc.). *metaconfig* autorise la définition de variables et de macros pouvant être utilisées dans tout le document et remplacées par les valeurs concrètes durant la phase de génération des configurations. Une interface graphique est également proposée et permet à un administrateur réseau d'effectuer un ensemble d'opérations sur la représentation.

Comparaison Ce sont ces deux travaux qui « collent » le plus à notre solution. En effet, les deux utilisent une abstraction de haut niveau (écrite en XML) afin de représenter le réseau sous une seule entité. NetML se distingue fortement de part son approche « validation par la simulation », là où *metaconfig* suit une démarche similaire à la notre. *metaconfig* utilise, tout comme nous, un processus de vérification basé sur plusieurs niveaux de règles. Cependant, l'auteur ne semble pas mettre cela en avant. Notons que, si nous avions eu vent de cette solution au début de ce travail, nous aurions certainement réutilisé certaines parties comme point de départ afin d'approfondir certains aspects non couverts par manque de temps.

9.2.2 Approche orientée « *data mining* »

D'autres techniques existent afin de découvrir des erreurs parmi des configurations. Dans cette section, nous décrivons quelques résultats utilisant des mécanismes de *machine learning*. Par rapport à l'approche orientée règle, l'avantage de celle-ci est qu'elle ne nécessite plus d'intervention humaine pour obtenir des résultats exploitables (*i.e.* plus besoin d'écrire des règles).

Le premier outil analysé est *Minerals*. Celui-ci se base sur toutes les configurations d'un domaine afin de construire un ensemble de politiques locales au réseau considéré. Celles-ci représentent les politiques les plus souvent utilisées. Toute déviation par rapport à ces politiques constitue une erreur potentielle. Les auteurs partent du principe que les approches orientées « règles » ne sont pas efficientes, car la notion de réseau « correct » dépend fortement du réseau considéré. En effet, une erreur au sein d'un réseau peut constituer une pratique courante dans un autre. Afin de pouvoir analyser des réseaux hétérogènes (*i.e.* dans lesquels plus d'un vendeur est présent), *Minerals* utilise une représentation intermédiaire. Leur outil permet de diminuer le nombre de faux positifs en permettant à l'opérateur de donner du *feeback*. Ainsi, après chaque exécution, l'opérateur peut dire, pour chaque erreur détectée, si celle-ci est correcte ou non. Dès lors, ce processus permet d'améliorer la précision des résultats. Les auteurs ont testé leur outil sur trois réseaux différents. Dans chacun de ceux-ci, les erreurs détectées par *Minerals* ont été confirmées par les opérateurs respectifs.

El-Arini et Killourhy [EAK05] proposent un algorithme de détection d'anomalies au sein de configurations. Cet algorithme se base sur des principes statistiques et plus spécialement sur un détecteur bayésien. Les auteurs l'ont testé sur un réseau universitaire présentant des erreurs. L'outil a permis de découvrir la plupart de ces erreurs avec un taux de faux positifs raisonnable.

Dans [CGG+04], les auteurs proposent une méthode originale afin d'automatiser la configuration des routeurs d'un réseau. L'originalité réside dans le fait que l'approche décrite est une approche « du bas vers le haut » partant d'une configuration existante afin d'automatiser le processus de configuration. Leur travail a débouché sur l'implémentation d'un outil nommé EDGE. Pour ce faire, la méthode requiert trois étapes :

1. un travail de *reverse-engineering* à partir des configurations existantes, et ce, afin de reconstruire la topologie du réseau et de résumer son état. Cette phase permet généralement de découvrir des erreurs corrigibles immédiatement par les administrateurs.
2. une phase de *data mining* afin d'identifier les politiques locales au réseau choisies par les administrateurs ainsi que les violations détectées par rapport à ces politiques. Cette phase permet de les alerter des inconsistances éventuellement présentes au sein de leur réseau.
3. au terme de la seconde phase, le réseau dispose d'une configuration « correcte » par rapport à un ensemble de règles. En d'autres termes, il semble avoir été configuré de façon automatique. La troisième étape consiste donc à sauvegarder l'état du réseau dans une base de données afin de faciliter les configurations ultérieures.

Leur outil est capable de réaliser les deux premières étapes. EDGE est actuellement utilisé dans une douzaine de grandes entreprises et gère plusieurs dizaines de milliers de fichiers de configuration. Néanmoins, ce papier ne propose pas de résultats concrets résultant de l'utilisation de leur outil.

Comparaison Ces solutions permettent d'obtenir des résultats probants sans pour autant requérir un passage obligé par l'écriture de règles. De plus, ces outils s'adaptent facilement à tout type de réseau. Cependant, ils partent tous du principe qu'une certaine base de connaissance (*i.e.* des configurations existantes) est disponible, ce qui n'est pas forcément le cas. Des faux positifs et négatifs peuvent également résulter du processus, ce qui implique un filtrage manuel par la suite.

9.3 Solutions commerciales

Dans cette partie, nous présentons deux des solutions payantes les plus connues : OPNET NetDoctor ainsi que Wandl IPAT. Au vu du caractère commercial de ces solutions, peu de détails sont fournis quant à leur mode de fonctionnement. Nous nous basons donc sur les quelques informations disponibles. Dès lors, nous nous concentrons davantage sur les résultats obtenus et sur les bénéfices apportés par l'utilisation de tels outils.

OPNET NetDoctor [10] est une solution qui permet d'exposer, grâce à une analyse automatique des configurations réseaux, des problèmes qui peuvent compromettre la disponibilité, les performances ou encore la sécurité du réseau considéré. L'outil se base sur une approche orientée « règles » qui permet une analyse systématique de l'entièreté du réseau. OPNET ne se contente pas d'une analyse individuelle de chaque configuration, mais utilise un modèle afin d'effectuer également une analyse basée sur la topologie dans le but de détecter des problèmes inhérents à la structure du réseau (p.ex. des problèmes de routage). Enfin, OPNET fournit aux utilisateurs une base de règles qu'ils peuvent étendre s'ils souhaitent vérifier des problèmes spécifiques à leur réseau.

La solution NPAT (*Network Planning and Analysis Tools*) vendue par la société Wandl [13] permet d'aider les opérateurs réseaux dans les phases de *design*, d'analyse et de simulation de leur réseau. NPAT permet, via un ensemble de modules, de gérer un grand nombre de vendeurs ainsi que la plupart des technologiques actuelles. NPAT permet une analyse de la topologie ayant pour but de détecter la présence d'éventuels problèmes tels que la présence de goulots d'étranglement ou encore des problèmes d'accessibilité résultant de la panne d'un noeud ou d'un lien.

Comparaison Ces approches sont relativement similaires à la notre excepté le fait qu'elles détectent des erreurs dans configurations existantes, alors que dans notre approche, nous définissons d'abord les règles avant la configuration du réseau. Le problème de ces solutions est qu'au vu de leur caractère fermé car commercial, un utilisateur se voit limité à l'utilisation des fonctionnalités présentes et contraint d'attendre que les nouvelles fonctionnalités dont il a besoin soient développées et rajoutées au logiciel. Cette relative inertie peut être lourde de conséquence dans un environnement en perpétuelle évolution comme celui d'un réseau.

Notons que ces approches ne se contentent pas d'une analyse statique des configurations. En effet, elles permettent grâce à l'utilisation d'un *solveur* de modéliser des comportements dynamiques. Notre logiciel ne permet par actuellement ce genre de vérification plus poussées.

Cependant, en le combinant avec un logiciel tel que C-BGP [QU05] qui inclus également un *solveur* et dont les sources sont ouvertes, nous pourrions également représenter des règles de validation avancées qui prennent en compte la dynamique des protocoles de routage.

9.4 Conclusion

Les domaines de recherche dans la gestion et la validation de configuration de réseaux sont extrêmement féconds au vu du nombre de publications scientifiques et des solutions proposées. Cela est notamment du au fait que la communauté scientifique prend peu à peu conscience de l'utilité de telles approches.

Dans un premier temps, nous avons présenté quatre solutions standardisées dans le domaine plus général de la gestion de configuration réseaux : SNMP, *Netconf*, RPSL et PBNM. Ces approches bien que très évoluées diffèrent de la notre, dans le sens où elles ne permettent pas de valider *a priori* la configuration d'un réseau.

Ensuite, nous avons étudié quelques solutions de recherches qui permettent de valider un réseau selon deux approches : une orientée « règles » et une orientée « *data mining* ». La première approche est celle que nous avons retenue. Actuellement, elle est de loin la plus utilisée. L'avantage de cette approche est qu'elle permet de détecter des erreurs complexes. Le problème est évidemment que le système doit avoir des règles pour vérifier chaque aspect d'une configuration. La seconde approche utilise des mécanismes d'intelligence artificielle afin de rechercher automatiquement des erreurs au sein des configurations. L'avantage de cette dernière est qu'elle ne nécessite pas d'apport externe (p.ex. des spécifications ou des règles) afin de fonctionner. Par contre, elles peuvent générer un taux important de faux positifs (*i.e.*, des erreurs détectées qui n'en sont pas réellement) que l'opérateur devra filtrer *a posteriori*.

Enfin, nous avons abordé deux solutions commerciales. Celles-ci sont notamment capables de détecter des erreurs dans des configurations existantes. Cependant, elles vont plus loin dans le sens où elles permettent de tester certaines parties relatives au comportement dynamique des protocoles là où notre logiciel ne permet qu'une analyse statique de la configuration. Néanmoins, il est possible d'atteindre le même genre de résultat avec notre logiciel en le couplant avec des outils de modélisation.

Chapitre 10

Conclusions et perspectives

Arrivés au terme de ce mémoire, nous désirons faire le point sur le chemin parcouru. Pour commencer, les objectifs présentés au début de ce travail ont été atteints : le logiciel développé (*vng*) est *flexible* et permet la *génération* de configurations de réseaux *validées*. Certes, celui-ci est encore au stade de prototype, mais il a d'ores et déjà prouvé son utilité au travers de la représentation et de la validation d'un réseau de recherche existant.

La première partie de ce travail a été consacrée à une présentation du langage de balisage XML ainsi que de ses technologies connexes (W3C XML Schema, XPath, XQuery et XSLT). En effet, celles-ci sont utilisées dans la majorité des parties du logiciel. Il nous est donc apparu essentiel de commencer par en introduire les concepts clés.

Ensuite, nous nous sommes attelés à la formalisation du concept de *validation* d'une représentation d'un réseau. Pour ce faire, nous avons décidé d'utiliser une approche basée sur des règles, c'est-à-dire sur des spécifications que la représentation doit respecter. Après une étude théorique et pratique, nous nous sommes rendus compte que quelques types de règles pouvaient valider la majorité des conditions. Cinq types de règles ont été identifiés : des règles de *présence*, de *non-présence*, d'*unicité*, de *symétrie* et des règles *personnalisées*. Afin de faciliter leur écriture, nous avons introduit les concepts de SCOPE et de `descendents`. Enfin, nous avons introduit la notion de *graphe de dépendances* qui modélise les relations entre ces règles.

Dans la suite, nous avons proposé la structure de *vng*. Tout d'abord, nous avons abordé les deux processus principaux : le *validateur* et le *générateur*. Le premier valide la représentation d'un réseau sur base d'un ensemble de règles fournies *a priori*, tandis que le second génère des configurations à partir de la représentation et de modèles décrivant les formats de sortie. Une fois la structure établie et avant de se lancer dans l'implémentation, nous avons dû convenir des différentes technologies à utiliser. Pour la représentation, notre choix s'est porté quasi immédiatement sur le langage de balisage XML qui a l'avantage d'être très flexible et d'être poussé par la communauté réseau. En ce qui concerne l'implémentation des règles, deux méthodes ont été avancées : l'utilisation d'un langage de programmation et l'utilisation de technologies liées à XML. C'est la seconde méthode que nous avons retenue. En effet, il s'est avéré qu'elle permet une expression plus simple des règles. Une fois les choix justifiés, nous avons détaillé le processus

de traduction des constructions théoriques dans une représentation utilisable par *vng*. Enfin, nous avons expliqué la façon dont est réalisé le procédé de génération des configurations.

Une fois le logiciel introduit, nous avons détaillé la structure du document XML décrivant la représentation de haut niveau d'un réseau. Nous avons alors présenté deux des avantages inhérents à l'utilisation d'une telle représentation : d'une part, la réduction de la redondance propre aux configurations réseaux et, d'autre part, la possibilité d'être totalement indépendant d'un vendeur particulier. Nous avons également proposé deux abstractions matricielles simples permettant de représenter des politiques interdomaines.

Le chapitre suivant a été consacré à l'étude de trois techniques de vérification. La première est implémentée à l'aide d'un schéma XML et permet d'exprimer des règles portant sur la structure de la représentation. Ces règles sont appelées *Structural rule*. La deuxième utilise le langage de requêtes XQuery et autorise des opérations sur la représentation XML semblables à celles applicables sur une base de données relationnelle. Les règles implémentées à l'aide de cette technique sont intitulées *Query rule*. La dernière technique, la plus évoluée, se base sur un langage de programmation, dans ce cas-ci Java, pour permettre l'application d'algorithmes ou de calculs complexes. Les règles réalisées par cette technique sont nommées *Language rule*. Nous avons montré que chacune de ces techniques est mieux appropriée dans la vérification de certains types de règles. Nous avons également donné quelques conseils permettant de choisir la technique *ad hoc* en fonction du type de règle.

L'étape suivante de notre démarche a été de présenter la génération des configurations. Pour ce faire, nous avons tout d'abord développé l'utilité des représentations intermédiaires en illustrant le fait que celles-ci peuvent être vues comme le résultat d'un prétraitement de l'information afin de faciliter l'étape de génération ultérieure. Ensuite, nous avons montré comment il est possible, à l'aide de feuilles de style XSLT, de générer des configurations exprimées dans n'importe quel langage de configuration ou de modélisation. Nous avons conclu en expliquant la manière dont des règles de « bonnes pratiques », en ce qui concerne la configuration d'un équipement, peuvent être incluses par défaut dans les configurations générées.

Afin de montrer que notre logiciel fonctionne et est capable de valider des réseaux existants, nous avons présenté une étude de cas basée sur le réseau de recherche américain *Abilene*. Dans la première étape, nous avons effectué un travail de rétro ingénierie afin d'identifier les principes de conception utilisés par les opérateurs. Ensuite, nous avons montré la façon dont ces principes ont pu être vérifiés par *vng*, et ce, à l'aide d'une centaine de règles. Cette étude a été également l'occasion de donner un exemple d'utilisation de nos abstractions décrivant les politiques interdomaines d'un réseau. Nous avons illustré chacune de ces étapes par des exemples de configurations générées et exprimées dans le langage de configuration JunOS.

Le dernier chapitre a été consacré à un tour d'horizon de l'état de l'art dans le domaine de la gestion et de la validation de configurations réseaux. Nous avons d'abord étudié quelques solutions standardisées pour ensuite passer à l'analyse des résultats de recherche. Ces derniers ont été étudiés en suivant deux approches : la première orientée « règles », la seconde orientée « *data mining* ». Nous avons également abordé deux solutions commerciales. Pour chacun de ces travaux, une brève comparaison par rapport à notre approche a été dressée.

En ce qui concerne les perspectives futures, plusieurs axes de recherches subsistent. Tout d'abord, il est assez rare pour un opérateur de concevoir un réseau de A à Z. En effet, la plupart du temps, les changements s'effectuent sur un réseau déployé. Le problème actuel est qu'un opérateur doit lui-même assurer le procédé de traduction de son réseau au sein de notre représentation, ce qui s'avère être une perte de temps et, qui plus est, est sujet aux erreurs. Une solution serait de créer un processus capable, à partir de configurations réseaux existantes, de produire la représentation de haut niveau utilisée par *vng*. Ensuite, une interface entre la représentation des données et le traitement de celles-ci serait nécessaire afin d'assurer une plus grande modularité à l'outil et son indépendance par rapport à la technologie utilisée. Enfin, comme mentionné précédemment, notre logiciel est encore au stade de prototype. Dès lors, il ne permet la validation que de quelques fonctionnalités. L'objectif serait donc de l'étendre afin de couvrir un large éventail de fonctionnalités telles que d'autres protocoles de routage et de nouvelles technologies. Un large ensemble de règles présenté sous la forme d'une librairie réutilisable pourrait également être proposé aux utilisateurs.

En guise de conclusion, nous espérons que ces approches de validation de réseau sont promises à un bel avenir. En effet, les apports sont réels et les bénéfices nombreux. *vng* répond déjà à quelques objectifs clés mais il ne constitue qu'un premier pas prometteur dans le domaine de la validation de configurations réseaux.

Annexe A

Règles

A.1 Règles de présence

Ce type de règles permet de vérifier si certains *CNodes* vérifiant une condition donnée sont présents dans l'arbre T. Par exemple, une règle de *présence* peut vérifier que tous les routeurs ont un *router id* [DB06] ou encore que toutes les interfaces OSPF connectées à un voisin externe sont configurées en mode passif de manière à ne pas former d'adjacence avec celui-ci [DB06].

Ce type de règles couvre différents cas :

(i) vérifier la présence d'un champ d'un *CNode*.

(ii) vérifier la présence d'un *CNode* d'un certain type.

(iii) vérifier la présence d'un *CNode* dont un de ses champs respecte une condition donnée.

(iv) vérifier la présence d'un *CNode* respectant une certaine condition parmi un ensemble de *CNodes*.

Les règles du cas (i) utilisent un ensemble unique de *CNodes* car nous voulons vérifier pour un ensemble de *CNodes* donné (*i.e.*, le SCOPE de la règle) que chacun d'entre eux possède un champ donné. D'une manière générale, le cas (i) peut être formulé par la formule A.1.

$$\forall x \in \text{SCOPE} : x.field \tag{A.1}$$

Pour rappel, une condition de la forme `x.field` renvoie soit *true* si le *CNode* x contient le champ `field` soit *false* dans le cas contraire.

Par exemple, vérifier que toutes les adresses IP associées à une interface possèdent toujours un masque est une règle appartenant au cas (i). Le masque d'une adresse IP est un champ du *CNode* représentant un adresse IP (*cf.* figure 3.2). Cette règle est formulée par l'équation A.2.

$$\forall x \in \text{ADDRESSESIP} : x.mask \tag{A.2}$$

Le SCOPE de la règle est l'ensemble de tous les *CNodes* représentant des adresses IP, c'est-à-dire tous les *CNodes* de type `ip`[1]. Ce SCOPE est noté ADDRESSESIP.

Le cas (ii) est une généralisation du cas (i) où la condition à vérifier utilise des informations issues d'autres *CNodes* de l'arbre T. En effet, vérifier la présence d'un *CNode* d'un certain type dans l'arbre revient à vérifier que ce *CNode* est l'enfant d'un autre *CNode*. Autrement dit, pour vérifier la présence d'un *CNode* d'un certain type, nous vérifions qu'un *CNode* possède un enfant de ce type.

De plus, nous ne voulons généralement pas vérifier la présence d'un seul *CNode* d'un certain type, mais nous voulons vérifier qu'un ensemble de *CNodes* possède un enfant d'un certain type. Par exemple, vérifier la présence d'un *router id* sur chaque routeur nécessite de vérifier que chaque *CNode* de type `router` possède un *CNode* enfant de type `rid`. Cette règle est formulée par l'équation A.3.

$$\forall x \in \text{ROUTERS} : (\exists z \in child(x) \ : \ z.type = rid) \tag{A.3}$$

L'ensemble de *CNodes* parent est représenté par le SCOPE. D'une manière générale, les règles du cas (ii) vérifient que chaque *CNode* du SCOPE est un parent d'un enfant d'un certain type. Ces règles peuvent être formalisées par l'équation générale A.4 où `desiredType` représente le type de *CNode* souhaité.

$$\forall x \in \text{SCOPE} : (\exists z \in child(x) \ : \ z.type = desiredType) \tag{A.4}$$

Le cas (iii) est très similaire au cas (ii) dans la mesure où la condition ne porte plus sur le champ `type` d'un *CNode* mais sur n'importe quel autre champ. Les cas (ii) et (iii) peuvent être formulés de manière générale par l'équation A.5.

$$\forall x \in \text{SCOPE} : C_{\text{presence}}(T, x) \tag{A.5}$$

Les règles couvertes par le cas (iv) requièrent d'obtenir des ensembles de descendants aux éléments du SCOPE. Chaque ensemble noté `descendents(p)` est obtenu à partir d'un *CNode* p du SCOPE. Par exemple, le cas (iv) peut être illustré par la règle permettant de vérifier que tous les routeurs possèdent au moins une interface *loopback* [DB06]. Cette règle vérifie que pour chaque routeur représenté par un *CNode* x du SCOPE, s'il existe au moins une interface parmi l'ensemble de ses interfaces représenté par `interfaces(x)` dont l'identificateur est `loopback`. Cette règle est formulée par l'équation A.6.

$$\forall x \in \text{ROUTERS} \ \exists y \in \texttt{interfaces}(x) \ : y.id = loopback \tag{A.6}$$

Ce genre de règles peut être écrite de manière générale en faisant apparaître la notion de SCOPE et de ses descendants (`descendents`). Comme pour les cas précédents, la condition à vérifier est représentée par C_{presence}. Une règle de présence vérifie pour chaque *CNode* x du

[1] En réalité le type peut être soit `ipv4` soit `ipv6`. De plus une vérification du parent est souvent effectuée de manière à s'assurer que cette adresse IP est une adresse IP pour une interface et pas, par exemple, pour décrire un voisin BGP.

SCOPE s'il existe au moins un *CNode* dans `descendents`(x) qui respecte la condition C_{presence}. La formalisation de cette règle est donnée par l'équation A.7.

$$\forall x \in \text{SCOPE}\ \exists y \in \texttt{descendents}(x)\ :\ C_{\text{presence}}(T, y) \tag{A.7}$$

Nous allons montrer que les équations des cas précédents peuvent être écrites sous cette forme générale.

L'équation A.7 est une généralisation du cas (iii) (eq. A.5). En effet, le cas (iii) A.5 est un cas particulier de cette équation où la notion de `descendents` n'est pas utilisée. Le cas (iii) peut être écrit de manière équivalente en faisant apparaître cette notion.

Remarquons que la vérification de l'existence d'un élément respectant une certaine condition parmi un ensemble d'éléments dont la cardinalité est un (*i.e.* l'ensemble ne contient qu'un unique élément) revient à vérifier la condition sur l'unique élément de l'ensemble. Autrement dit, l'équation A.5 peut être écrite de manière équivalente par l'équation A.8 en utilisant l'ensemble {`x`} qui contient l'unique *CNode* x.

$$\forall x \in \text{SCOPE}\ \exists y \in \{\texttt{x}\}\ :\ C_{\text{presence}}(T, y) \tag{A.8}$$

L'équation A.8 est similaire à l'équation A.7 avec `descendents`$(x) = \{x\}$. Un ensemble de descendants de x est réduit à un ensemble contenant uniquement x.

L'équation de la définition de `descendents`(p) est rappelée par l'équation A.9.

$$\texttt{descendents}(p) = \{q | q \in \mathit{descendents}(p) \cup p\ :\ C(T, q)\} \tag{A.9}$$

Dans notre cas particulier, nous avons :

$$\texttt{descendents}(x) = \{\texttt{x}\} = \{q | q \in \mathit{descendents}(x) \cup x\ :\ q = x\} \tag{A.10}$$

Remarquons l'importance de l'union entre les descendants de x et x lui-même de manière à ce que la condition $q = x$ soit vérifiée lorsque q est égal à x. Chaque *CNode* x du SCOPE permet d'obtenir l'ensemble {`x`}.

En conclusion, l'équation générale A.5 du cas (iii) est équivalente à l'équation A.8, qui est un cas particulier de l'équation générale de la règle de *présence* A.7 où `descendents`$(x) = \{x\}$. La formalisation initiale A.5 du cas (iii) couvre également le cas (ii) et le cas (i). Par conséquent, l'équation générale de la règle de *présence* A.7 couvre également ces cas.

L'équation A.7 est donc l'équation générale des règles de *présence*. La majorité des règles de *présence* découvertes peuvent être exprimées par cette équation.

A.2 Exemple d'une règle de non-présence

Par exemple, vérifier qu'il n'existe pas de *stub areas* ou de *not so stuby areas* dans OSPF qui contiennent un *virtual link* est une règle utilisant un unique ensemble de *CNodes*. Elle est exprimée par l'équation A.11.

$$\forall x \in \textsc{stubAreaOrNssa} : \neg(\exists z \in child(x) \ : \ z.type = virtualLink) \tag{A.11}$$

Rappelons que la fonction $child(y)$ renvoie l'ensemble des *CNodes* enfants de y.

Cette équation peut être écrite de manière équivalente par l'équation A.12 en faisant apparaître $\texttt{descendents}(x) = \{\texttt{x}\}$. En effet, vérifier que tous les éléments d'un ensemble de cardinalité un (i.e. $\{\texttt{x}\}$) respectent une certaine condition revient à vérifier la condition sur les éléments directement. Ceci est illustré par l'équation A.11 où la condition $\neg(\exists z \in child(x) \ : \ z.type = virtualLink)$ est appliquée directement sur les éléments x de l'ensemble stubAreaOrNssa. Cette notation est similaire à la forme de l'équation générale de la règle de *non-présence* 3.10.

$$\forall x \in \textsc{stubAreaOrNssa}\, \forall y \in \{\texttt{x}\} \ : \ \neg(\exists z \in child(y) \ : \ z.type = virtualLink) \tag{A.12}$$

Annexe B

Index des règles

Toutes les règles pouvant être vérifiées par notre logiciel sont reprises ci-dessous. Notons que celles-ci peuvent être désactivées ou activées selon les besoins des opérateurs. La section B.3 présentera les SCOPES définis et utilisés par les règles. Le tableau B.1 reprend un compte rendu du nombre de règles vérifiées en fonction de leur type.

	Structural rule	*Query rule*	*Language rule*	**Total**
Présence	101	13	-	114
Non-présence	0	3	-	3
Unicité	14	7	-	21
Symétrie	10	-	-	10
Personnalisée	-	12	4	16
Matrix	-	3	-	3
Total	125	38	4	**167**

TAB. B.1 – Nombre de règles vérifiées

B.1 Règles vérifiant la structure de la représentation du réseau

Description de la règle	Type de règle	Technique utilisée
Le domaine peut contenir un nom.	*présence*	*Structural rule*
Un domaine contient la configuration de sa topologie.	*présence*	*Structural rule*
Un domaine peut contenir une configuration pour OSPF et/ou pour BGP.	*présence*	*Structural rule*
Topologie		
La topologie est composée de routeurs, de liens physiques et de préfixes d'adresses locaux. Les préfixes d'adresses locaux sont les adresses IP allouées au domaine modélisé.	*présence*	*Structural rule*
La topologie peut décrire les AS voisins.	*présence*	*Structural rule*
Le réseau possède au moins un routeur.	*présence*	*Structural rule*
Le réseau possède au moins un préfixe d'adresse local.	*présence*	*Structural rule*
Un nombre quelconque de préfixes d'adresses locaux peuvent être indiqués.	*présence*	*Structural rule*
Un préfixe d'adresse local est une adresse IPv4 ou IPv6 avec un masque.	*présence*	*Structural rule*
Routeurs		
Chaque routeur possède un nom (*hostname*).	*présence*	*Structural rule*
Chaque routeur possède un *router id*.	*présence*	*Structural rule*
Le *router id* est une adresse IPv4.	*présence*	*Structural rule*
Chaque routeur possède une indication sur son constructeur, sur son modèle et sur la version de l'OS utilisée.	*présence*	*Structural rule*
Chaque routeur peut posséder un statut (*up* ou *down*).	*présence*	*Structural rule*
Le constructeur d'un routeur peut être soit Cisco, soit Juniper.	*présence*	*Structural rule*
Chaque routeur peut posséder des routes statiques.	*présence*	*Structural rule*
Une route statique est composée d'une adresse destination (avec un masque) ainsi que d'une adresse *next-hop* (sans masque).	*présence*	*Structural rule*

AS voisins		
Un AS voisin est identifié par un nom.	*présence*	*Structural rule*
Le nom des AS voisins sont uniques.	*unicité*	*Structural rule*
Un AS voisin représente soit un autre AS, soit un point d'interconnexion (*connector*).	*présence*	*Structural rule*
Un autre AS est représenté en indiquant son numéro d'AS.	*présence*	*Structural rule*
Un point d'interconnexion est représenté en indiquant dans l'attribut `type` la valeur `connector`.	*présence*	*Structural rule*
Un AS voisin ne peut pas être configuré comme *connector*.	*non-présence*	*Query rule*
Interfaces		
Chaque interface possède un nom.	*présence*	*Structural rule*
Les noms des interfaces des routeurs sont uniques.	*unicité*	*Query rule*
Une interface peut posséder un statut (*up* ou *down*), une description (chaîne de caractères), un MTU.	*présence*	*Structural rule*
La valeur d'un MTU doit être une valeur entière comprise entre 256 et 9192 [KD02].	*présence*	*Structural rule*
Une interface possède au moins une unité logique.	*présence*	*Structural rule*
Unités logiques		
Une unité logique posssède un numéro.	*présence*	*Structural rule*
Les numéros des unités logiques d'une interface sont uniques.	*unicité*	*Query rule*
Le numéro d'une unité logique est une valeur entière comprise entre 0 et 65535 [KD02].	*présence*	*Structural rule*
Une unité logique possède au moins une adresse IP avec un masque.	*présence*	*Structural rule*
Une unité logique peut posséder une description (chaîne de caractères) ou un vlan-id.	*présence*	*Structural rule*
Le *vlan-id* est une valeur entière comprise entre 0 et 4095 [6].	*unicité*	*Query rule*

Liens		
Un lien dans le réseau peut être soit un lien interne, soit un lien externe	*présence*	*Structural rule*
Un lien possède un identificateur.	*présence*	*Structural rule*
Les identificateurs des liens sont uniques.	*unicité*	*Structural rule*
Un lien interne est établi entre au moins 2 routeurs du domaine.	*présence*	*Structural rule*
Un lien interne interconnecte des interfaces logiques.	*présence*	*Structural rule*
Un lien externe est toujours établi entre un routeur du domaine et un routeur voisin.	*présence*	*Structural rule*
Un lien externe utilise une interface logique d'un routeur du domaine.	*présence*	*Structural rule*
Le routeur externe est représenté par un AS voisin défini dans la topologie.	*présence*	*Structural rule*
Les liens externes et internes utilisent des routeurs existants.	*présence*	*Structural rule*
Un lien peut avoir une description.	*présence*	*Structural rule*
Un lien peut posséder les attributs suivants : la vitesse, l'encapsulation, le mode et le MTU.	*présence*	*Structural rule*
L'encapsulation d'un lien peut être `ethernet-ccc`, `ethernet-tcc`, `ethernet-vpls`, `extended-vlan-ccc`, `extended-vlan-tcc`, `extended-vlan-vpls`, `ppp`, `vlan-ccc`, `vlan-vpls` ou `cisco-hdlc`.	*présence*	*Structural rule*
Le mode d'un lien peut être soit `full-duplex`, soit `half-duplex`.	*présence*	*Structural rule*
Adresses IP		
Le type d'une adresse IP peut être `ipv4` et `ipv6`.	*présence*	*Structural rule*
Le masque d'une adresse IPv4 est une valeur entière comprise entre 0 et 32 [Pos81].	*présence*	*Query rule*
Le masque d'une adresse IPv6 est une valeur entière comprise entre 0 et 128 [DH98].	*présence*	*Structural rule*
Le format d'une adresse IPv4 respecte la notation « pointée » [Pos81].	*présence*	*Structural rule*
Le format d'une adresse IPv6 respecte le standard hexadécimal défini [DH98].	*présence*	*Structural rule*

OSPF		
Un routeur OSPF peut agréger des routes.	*présence*	*Structural rule*
Un agrégat est une adresse IP (IPv4 ou IPv6) avec masque.	*présence*	*Structural rule*
OSPF - Areas		
L'identifiant d'une *area* est un entier de 32 bits représenté sous la forme d'une adresse IPv4 [Doy98].	*présence*	*Structural rule*
Les numéros des *area* sont uniques.	*unicité*	*Structural rule*
Au moins une *area* doit être définie.	*présence*	*Structural rule*
Une *area* est composée de routeurs du domaine.	*présence*	*Structural rule*
Une *area* possède au moins un routeur du domaine.	*présence*	*Structural rule*
Une *area* peut posséder des liens virtuels.	*présence*	*Structural rule*
OSPF - Routeurs		
Un routeur OSPF peut annoncer une route par défaut.	*présence*	*Structural rule*
Un routeur OSPF peut définir la métrique de la route par défaut annoncée.	*présence*	*Structural rule*
La métrique associée à une route par défaut annoncée par OSPF est une valeur entière comprise entre 0 et 65535 [KD02].	*présence*	*Structural rule*
Un routeur OSPF peut désactiver l'envoi de LSA de type 3 (*i.e. Summary-LSA*) [Doy98]	*présence*	*Structural rule*
Un routeur OSPF doit contenir au moins une interface sur laquelle OSPF est activé.	*présence*	*Structural rule*
OSPF - Interfaces		
Une interface OSPF peut être configurée en mode passif.	*présence*	*Structural rule*
Une interface OSPF peut posséder une priorité, un statut (*UP* ou *DOWN*) et une métrique.	*présence*	*Structural rule*
Une priorité définie sur une interface est une valeur entière comprise entre 0 et 255 [9].	*présence*	*Structural rule*
Le coût (ou la métrique) d'une interface OSPF est une valeur entière comprise entre 0 et 65535 [8].	*présence*	*Structural rule*
Le coût (ou la métrique) d'une interface OSPF est une valeur entière comprise entre 1 et 65535 pour les routeurs Cisco [Doy98].	*présence*	*Query rule*
Une interface OSPF doit être une interface d'un routeur existant.	*personnalisée*	*Query rule*

OSPF - Redistributions de routes		
Des routes apprises par d'autres protocoles peuvent être redistribuées dans OSPF.	*présence*	*Structural rule*
Les redistributions de routes dans OSPF sont définies sur des routeurs existants.	*présence*	*Structural rule*
Les redistributions de routes dans OSPF identifie le protocole redistribué [6].	*présence*	*Structural rule*
Une métrique peut être définie sur les routes redistribuées d'un autre protocole.	*présence*	*Structural rule*
Une métrique appliquée sur les routes annoncées dans OSPF est une valeur entière comprise entre 0 et 65535 [Doy98].	*présence*	*Structural rule*
OSPF - Temporisateurs		
Les interfaces OSPF interconnectées possèdent des temporisateurs identiques (*hello, dead* et *retransmit interval*).	*symétrie*	*Structural rule*
Les temporisateurs sont des valeurs entières.	*présence*	*Structural rule*
OSPF - Authentification		
Une *area* peut posséder des paramètres relatifs aux mécanismes d'authentification.	*présence*	*Structural rule*
Des clés d'authentification peuvent être définies par *area*.	*présence*	*Structural rule*
Le type d'une clé d'authentification doit être défini (chaîne de caractères, p.ex. `md5`).	*présence*	*Structural rule*
Chaque clé d'authentification possède un identificateur.	*présence*	*Structural rule*
La valeur de cet identificateur est unique.	*unicité*	*Structural rule*
OSPF - Liens virtuels		
Les liens virtuels sont établis entre deux routeurs du domaine.	*présence*	*Structural rule*
Un lien virtuel est défini au sein d'une *area*.	*présence*	*Structural rule*

BGP		
La description du domaine contient le numéro d'AS.	*présence*	*Structural rule*
Le numéro d'AS est une valeur entière (entre 0 et 65535) [RLH06].	*présence*	*Structural rule*
La description de BGP peut contenir des communautés, la matrice de filtrage et la matrice des communautés.	*présence*	*Structural rule*
La configuration de BGP possède au moins une session iBGP et au moins une session eBGP.	*présence*	*Structural rule*
Un routeur BGP du domaine peut utiliser la politique *next-hop-self*.	*présence*	*Structural rule*
BGP - Matrices de filtrage et des communautés		
Une communauté possède un identificateur.	*unicité*	*Structural rule*
Les identificateurs des communautés sont uniques.	*unicité*	*Structural rule*
Les matrices de filtrage référencent des groupes de sessions eBGP existants.	*présence*	*Structural rule*
Les matrices des communautés référencent des communautés existantes.	*présence*	*Structural rule*
Les matrices des communautés référencent des groupes de sessions eBGP existants.	*présence*	*Structural rule*
Le type d'une politique dans la matrice des communautés peut être soit `accept`, soit `reject`.	*présence*	*Structural rule*
BGP - Sessions iBGP et eBGP		
Une session BGP (eBGP et iBGP) peut posséder une clé d'authentification et une description.	*présence*	*Structural rule*
Les sessions iBGP sont définies sur des routeurs du domaine.	*présence*	*Structural rule*
Une session eBGP possède un identificateur.	*unicité*	*Structural rule*
Les identificateurs des sessions eBGP sont uniques.	*unicité*	*Structural rule*
Une session eBGP peut posséder une indication sur le type de *peering* (chaîne de caractères).	*présence*	*Structural rule*
Une session eBGP est définie entre un routeur du domaine et un routeur voisin.	*présence*	*Structural rule*
Un routeur voisin référence un AS voisin défini préalablement.	*présence*	*Structural rule*
Un routeur voisin possède une adresse IP (IPv4 ou IPv6).	*présence*	*Structural rule*
Une session eBGP peut être établie entre un routeur du domaine et un routeur voisin accessible via un routeur intermédiaire appelé *connector* (chaîne de caractères).	*présence*	*Structural rule*
Un connecteur (*connector*) utilisé pour décrire une session eBGP doit être défini dans la topologie.	*présence*	*Structural rule*
Une session eBGP peut modifier l'attribut *local-pref* des routes apprises.	*présence*	*Structural rule*
Une session eBGP peut définir l'attribut *multi-hop*.	*présence*	*Structural rule*

Des ensembles de groupes de sessions (i.e. *sessions-sets*) peuvent être définis. Chaque groupe est identifié par un identificateur.	*présence*	*Structural rule*
Les sessions eBGP définies dans un groupe de sessions sont des sessions eBGP existantes.	*présence*	*Structural rule*
Les identificateurs des groupes de sessions sont uniques.	*unicité*	*Structural rule*
BGP - Réflecteurs de routes		
Les réflecteurs de routes (RR) sont des routeurs du domaine.	*présence*	*Structural rule*
Un routeur BGP peut être configuré comme réflecteurs de routes pour plusieurs *clusters* [KD02].	*présence*	*Structural rule*
Un *cluster* possède un identificateur qui doit être une adresse IPv4 (32 bits).	*présence*	*Structural rule*
Les identificateurs des *clusters* définis sur un routeur BGP sont uniques.	*unicité*	*Query rule*
Les routeurs BGP définis comme client d'un RR sont des routeurs du domaine.	*présence*	*Structural rule*

B.2 Règles avancées

Description de la règle	Type de règle	Technique utilisée
Le graphe représentant la topologie du réseau est connexe.	*personnalisée*	*Language rule*
Les adresses IP définies sur les interfaces des routeurs du domaine sont uniques.	*unicité*	*Query rule*
Routeurs		
Les *router id* des routeurs sont uniques.	*unicité*	*Structural rule*
Les noms des routeurs sont uniques.	*unicité*	*Structural rule*
Le *router id* doit être une adresse définie sur une interface *loopback* [DB06].	*présence*	*Query rule*
Les interfaces d'un routeur appartiennent à des *subnet* différents.	*personnalisée*	*Language rule*
Au moins une adresse IPv6 est configurée sur les interfaces des liens internes. Cette règle est propre au réseau *Abilene*.	*présence*	*Query rule*
Interfaces *loopback*		
La longueur du masque des adresses IPv4 défini sur une interface *loopback* doit être 32 [KD02].	*présence*	*Query rule*
La longueur du masque des adresses IPv6 défini sur une interface *loopback* doit être 128 [KD02].	*présence*	*Query rule*
Une interface *loopback* est obligatoire sur chaque routeur [GS02].	*présence*	*Query rule*
Les adresses définies sur toutes les interfaces *loopback* du réseau sont uniques [Doy98].	*unicité*	*Structural rule*
Les adresses définies sont uniques.	*unicité*	*Structural rule*
Liens		
L'encapsulation est identique sur les interfaces interconnectées.	*symétrie*	*Structural rule*
Le mode est identique sur les interfaces interconnectées.	*symétrie*	*Structural rule*
La vitesse est identique sur les interfaces interconnectées.	*symétrie*	*Structural rule*
Le MTU est identique sur les interfaces interconnectées.	*symétrie*	*Structural rule*
Chaque lien interne interconnecte au moins deux interfaces du domaine. Plusieurs interfaces peuvent être interconnectées pour représenter un LAN.	*présence*	*Structural rule*
Tous les routeurs du réseau possède au moins une interface *loopback*.	*présence*	*Query rule*
Toutes les interfaces actives sont connectées à un lien.	*personnalisée*	*Query rule*

VLans		
Les numéros de *vlan-id* définis sur une interface sont uniques.	*unicité*	*Query rule*
Aucun *vlan-id* n'est défini sur une interface *loopback*.	*non-présence*	*Query rule*
OSPF		
La bande passante de référence utilisée est identique sur tous les routeurs OSPF du domaine.	*symétrie*	*Structural rule*
Si l'option *trafic-engineering* est activée sur un routeur alors elle doit l'être sur tous les routeurs OSPF du domaine.	*symétrie*	*Structural rule*
OSPF - Areas		
Une *area* peut être sans type, de type `stub`, `nssa` ou `totallystubby` [Doy98].	*présence*	*Structural rule*
L'*area* 0 doit être obligatoirement être définie [GS02, DB06].	*présence*	*Query rule*
Toutes les *area* doivent être directement connectées à l'*area* 0 [GS02, DB06].	*personnalisée*	*Query rule*
Les numéros des *area* sont uniques.	*unicité*	*Structural rule*
Les *area* OSPF de type *stub* ou *not so stuby* ne peuvent pas contenir de liens virtuels [Doy98].	*non-présence*	*Query rule*
Une *area* OSPF de type *stub* ne peut pas contenir de ASBR [Doy98].	*personnalisée*	*Query rule*
L'*area* 0 ne peut pas contenir de *single link of failure* [Doy98].	*personnalisée*	*Language rule*
Les paramètres d'authentification sont identiques pour tous les routeurs d'une *area*.	*symétrie*	*Structural rule*
OSPF - Liens virtuels		
Un lien virtuel ne peut être établi qu'entre routeurs ABR [Doy98].	*personnalisée*	*Query rule*
Deux ABRs interconnectés par un lien virtuel doivent faire partie de la même *area* [Doy98].	*personnalisée*	*Query rule*
Un lien virtuel est établi entre deux routeurs du domaine.	*présence*	*Structural rule*
OSPF - Interfaces		
Les interfaces OSPF connectées à un routeur voisin doivent être configurées en mode passif afin de ne pas former d'adjacence OSPF avec celui-ci [GS02].	*présence*	*Query rule*
Les adresses *loopback* doivent être annoncées dans OSPF [GS02].	*présence*	*Query rule*
Une interface et plus précisément une unité logique peut au maximum exécuter une instance du processus OSPF.	*unicité*	*Query rule*
Une interface sur laquelle OSPF est activé doit être connectée à une autre interface où OSPF est également activé.	*personnalisée*	*Query rule*

BGP		
Tous les routeurs du réseau doivent être présents sous l'élément `internal`. Cet élément décrit tous les routeurs qui feront partie du *full mesh* iBGP.	*présence*	*Query rule*
Un *full mesh* iBGP doit être garanti entre tous les routeurs indiqués dans *internal*.	*symétrie*	*Structural rule*
Le *next-hop* annoncé dans les routes BGP est joignable par tous les routeurs du réseau [KD02].	*personnalisée*	*Query rule*
L'adresse IP d'un voisin eBGP appartient au préfixe de l'interface connectée.	*personnalisée*	*Language rule*
Un voisin eBGP doit être directement connecté à un routeur du domaine si *multi-hop* n'est pas utilisé [KD02].	*personnalisée*	*Query rule*
Si un point d'interconnexion est utilisé dans un session eBGP alors l'attribut *multi-hop* doit être utilisé.	*présence*	*Query rule*
Local Pref est une valeur entière encodée sur 32 bits.	*présence*	*Structural rule*
Multi-hop est une valeur entière comprise entre 1 et 255.	*présence*	*Structural rule*
La clé d'authentification doit être la même sur les deux routeurs d'une session BGP.	*symétrie*	*Structural rule*
BGP - Réflecteurs de routes		
Un routeur BGP configuré comme *Route Reflector* ne peut pas être son propre client.	*personnalisée*	*Query rule*
Les *Route Reflector* doivent être redondants [KD02].	*personnalisée*	*Query rule*
Matrices		
Les routes annoncées par les fournisseurs d'un domaine sont annoncées seulement à ses clients.	*matrix*	*Query rule*
Les routes annoncées par les *shared-costs* d'un domaine sont annoncées seulement à ses clients.	*matrix*	*Query rule*
Les routes annoncées par les clients d'un domaine sont annoncées seulement à ses *shared-costs* et à ses fournisseurs.	*matrix*	*Query rule*
Les groupes de sessions (*sessions-sets*) utilisées dans les matrices doivent être disjoints.	*personnalisée*	*Query rule*

B.3 Scopes

Id	Description
ALL_NODES	Tous les routeurs du réseau.
ALL_INTERFACES	Toutes les interfaces de tous les routeurs du réseau.
ALL_PHYSICAL_INTERFACES	Toutes les interfaces physiques de tous les routeurs du réseau, c'est-à-dire toutes les interfaces qui ne sont pas des interfaces *loopback*.
ALL_INTERNAL_UNITS	Toutes les unités logiques du réseau qui sont utilisées dans des liens internes.
ALL_UNIT_ON_LOOPBACK_INTERFACES	Toutes les unités logiques des interfaces *loopback*.
ALL_IPV4_ADDRESSES_ON_UNITS	Toutes les adresses IPv4 de toutes les unités logiques de tous les routeurs du réseau.
ALL_IPV4_UNIT_ON_LOOPBACK_INTERFACES	Toutes les adresses IPv4 des interfaces *loopback*.
ALL_IPV6_UNIT_ON_LOOPBACK_INTERFACES	Toutes les adresses IPv6 des interfaces *loopback*.
TOPOLOGY	L'élément représentant la topologie du réseau.
OSPF	
OSPF	L'élément représentant la configuration de OSPF.
ALL_OSPF_NODES	Tous les routeurs OSPF du réseau. Un routeur OSPF dispose au moins d'une interface sur laquelle OSPF est activé.
ALL_OSPF_AREAS	Toutes les *area* présentes dans OSPF.
OSPF_STUB_AREAS	Toutes les *area* OSPF de type `stub`.
OSPF_NSSA_AREAS	Toutes les *area* OSPF de type `nssa`.
OSPF_ROUTERS_PRESENT_ONLY_IN_STUB_AREA	Tous les routeurs présents dans des *stub area*.
ALL_OSPF_INTERFACES	Toutes les interfaces OSPF.
ALL_OSPF_CISCO_NODES	Tous les routeurs OSPF dont le constructeur est Cisco Systems.
OSPF_INTERFACES_ON_EXTERNAL_LINKS	Toutes les interfaces OSPF qui appartiennent à des liens externes.
STUB_AND_NSSA_AREAS	Toutes les *area* OSPF de type *stub* et *nssa*.
TOTALLY_STUBBY_AREAS	Toutes les *area* OSPF de type *totally stubby*.
BGP	
BGP_INTERNAL_NODES	Tous les routeurs BGP du domaine.
eBGP_SESSIONS_USING_ CONNECTOR	Toutes les sessions eBGP qui utilisent un point d'interconnexion.

Annexe C

Comparaison entre les configurations d'*Abilene* et celles générées

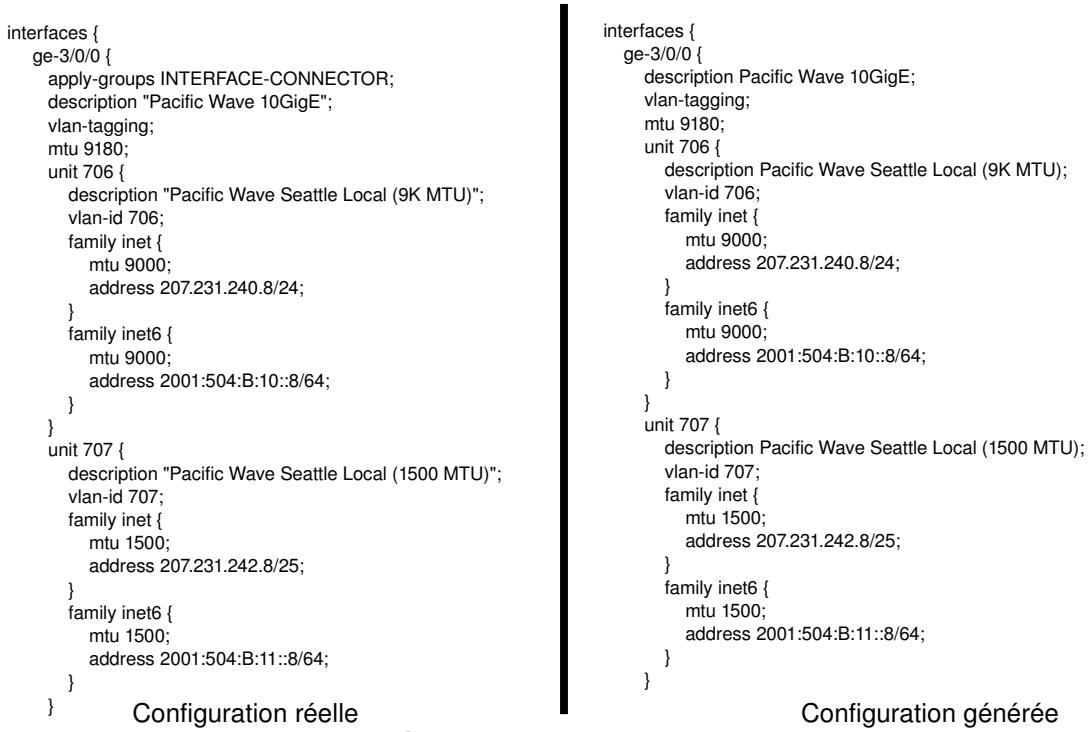

FIG. C.1 – Comparaison de la configuration de l'interface *ge-3/0/0* du routeur *ST*

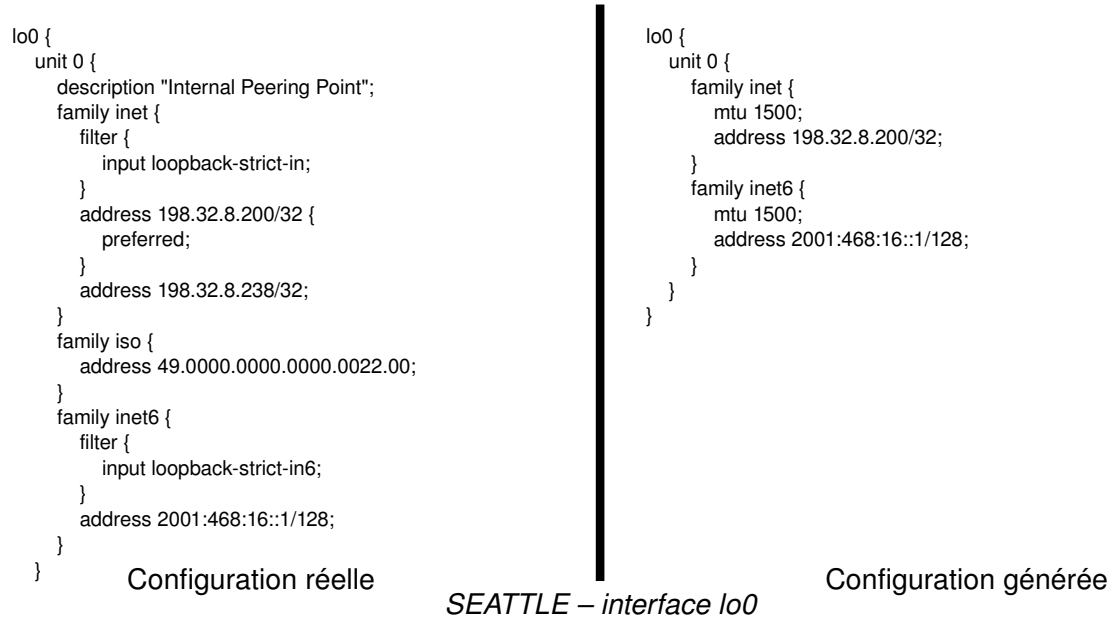

FIG. C.2 – Comparaison de la configuration de l'interface *loopback* du routeur *ST*

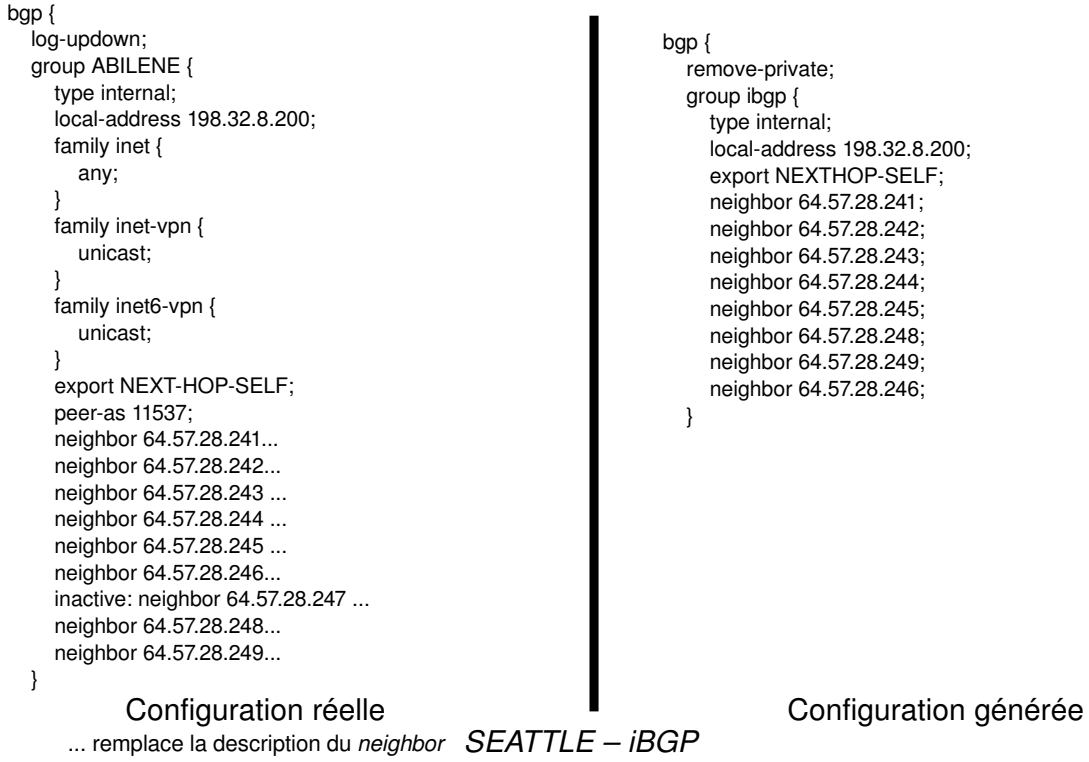

FIG. C.3 – Comparaison de la configuration des sessions iBGP du routeur *ST*

Configuration réelle

```
neighbor 207.231.240.7 {
    description "Microsoft via Pac Wave vlan706";
    import [ SANITY-IN SET-PREF FROM-MICROSOFT ];
    export [ SANITY-OUT REMOVE-COMMS-OUT ORIGINATE4 MICROSOFT-OUT ];
    peer-as 8075;
}
neighbor 207.231.241.7 {
    description "Microsoft via Pac Wave vlan776";
    import [ SANITY-IN SET-PREF FROM-MICROSOFT ];
    export [ SANITY-OUT REMOVE-COMMS-OUT ORIGINATE4 MICROSOFT-OUT ];
    peer-as 8075;
}
```

SEATTLE – eBGP Microsoft

Configuration générée

```
group ST-MICROSOFT {
   description Microsoft via Pac Wave vlan706;
   peer-as 8075;
   export [ APPLY-BLOCK-TO-COMMERCIAL ORIGINATE_LOCAL ];
   import [ SANITY-IN SET-COMMUNITY-COMMERCIAL ];
   neighbor 207.231.240.7;
}
group ST-MICROSOFT2 {
   description Microsoft via Pac Wave vlan776;
   peer-as 8075;
   export [ APPLY-BLOCK-TO-COMMERCIAL ORIGINATE_LOCAL ];
   import [ SANITY-IN SET-COMMUNITY-COMMERCIAL ];
   neighbor 207.231.241.7;
}
```

FIG. C.4 – Comparaison de la configuration des sessions eBGP avec Microsoft du routeur *ST*

Annexe D

Diagrammes UML

vng est composé de cinq *package* Java : *checker*, *rules*, *configurationGenerator*, *tools* et *xmlRelatedTools*. Cette annexe montre les diagrammes UML de ceux-ci.

D.1 Package *checker*

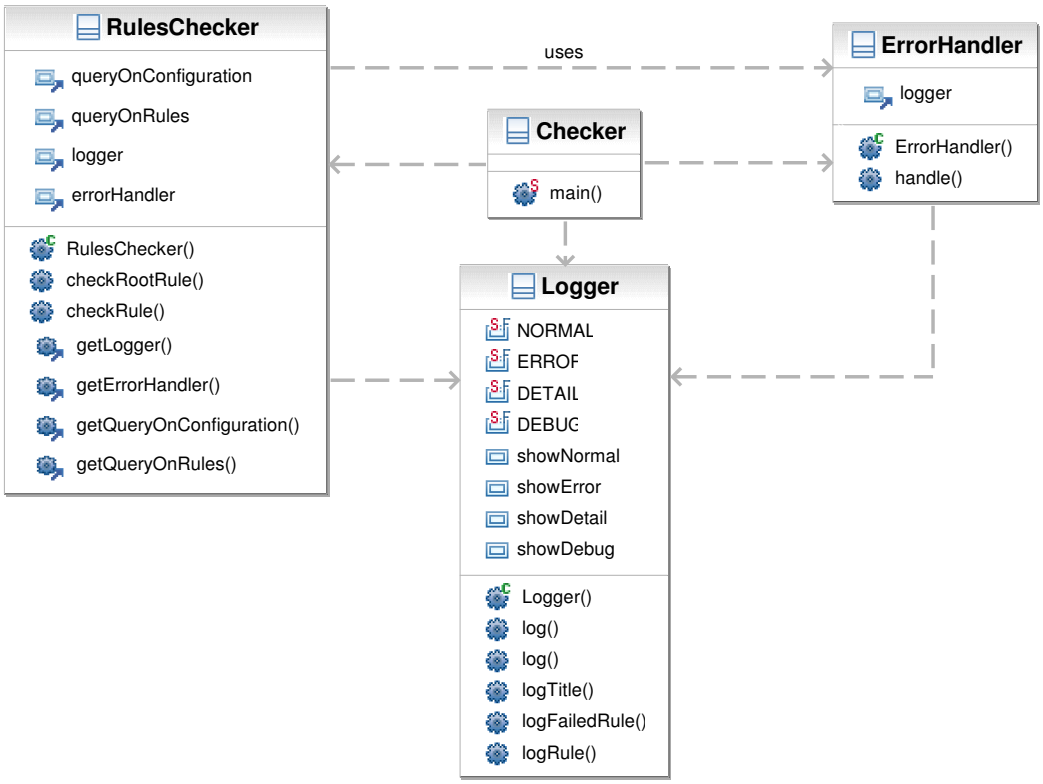

FIG. D.1 – Package *checker*

D.2 Package *rules*

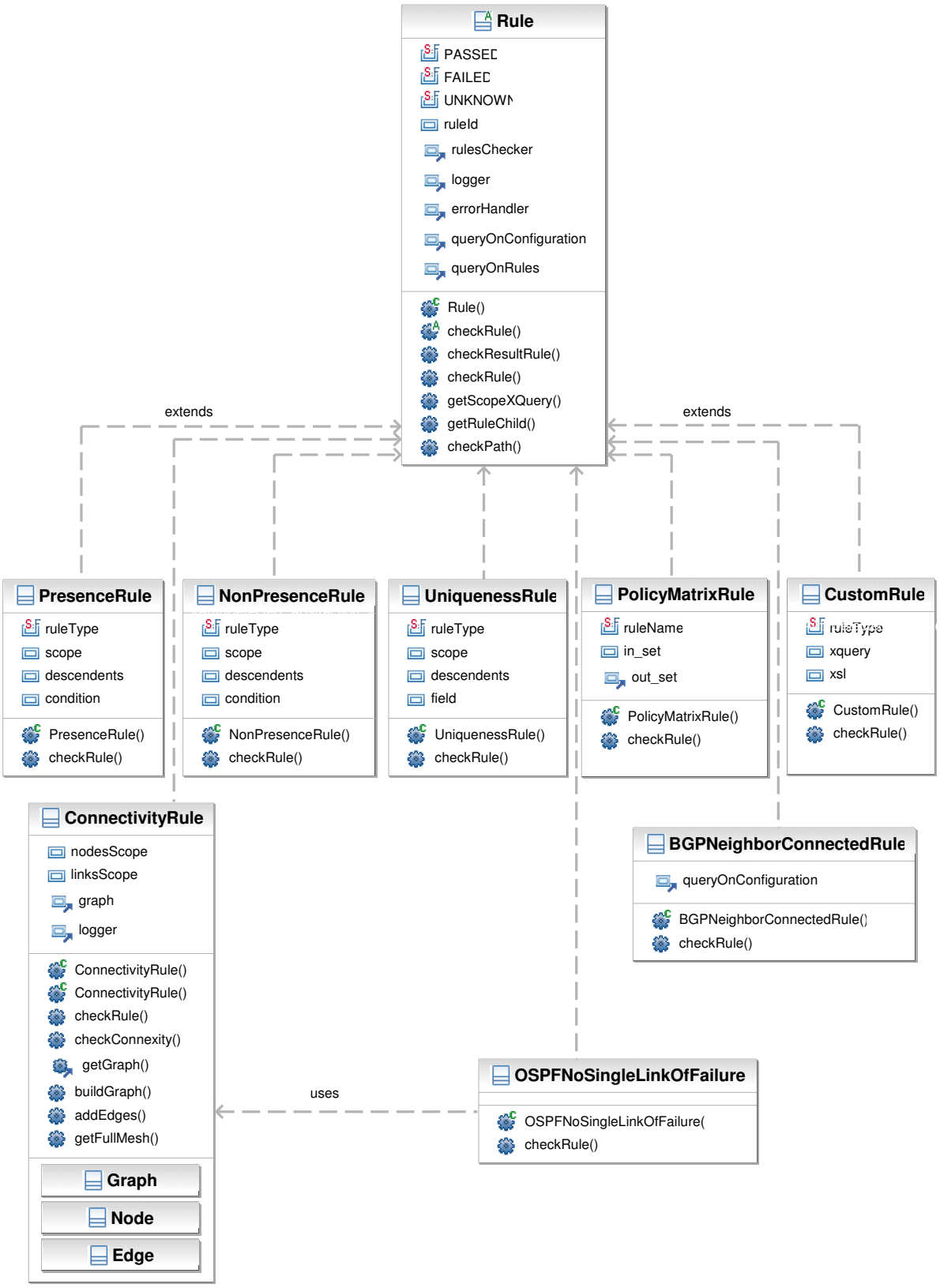

FIG. D.2 – Package *rules*

D.3 Package *configurationGenerator*

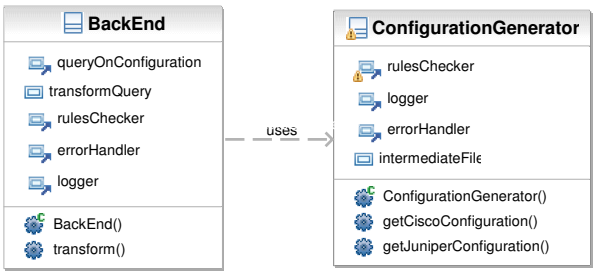

FIG. D.3 – Package *configurationGenerator*

D.4 Package *tools*

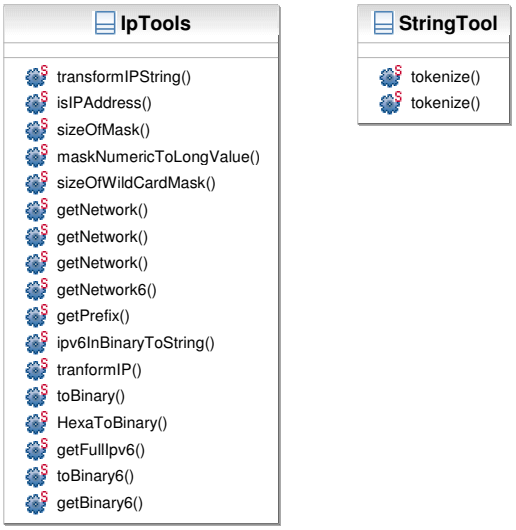

FIG. D.4 – Package *tools*

D.5 Package *xmlRelatedTools*

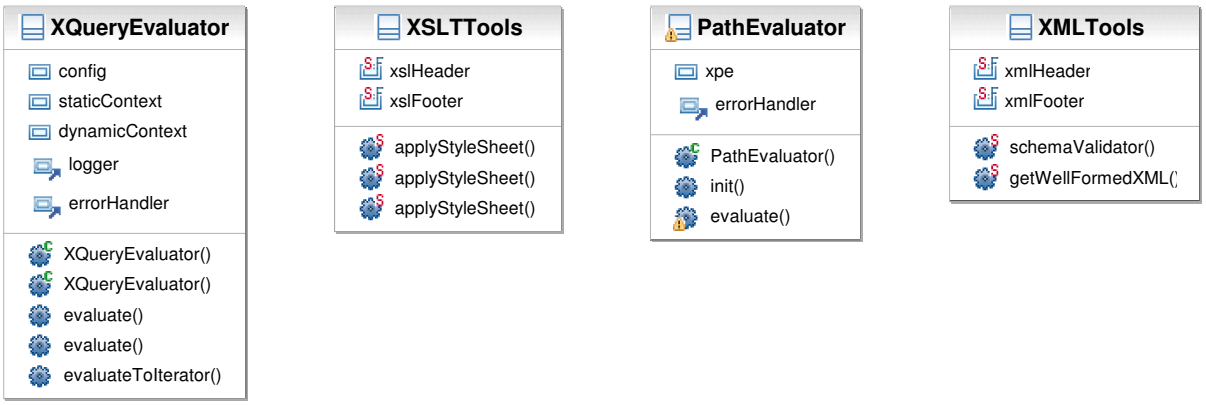

FIG. D.5 – Package *xmlRelatedTools*

Bibliographie

[Ala96] Cengiz Alaettinoglu : Scalable Router Configuration for the Internet, 1996.

[Aud08] Laurent Audibert : UML 2, 2007-2008.

[BCC06] Tony Bates, Enke Chen et Ravi Chandra : BGP Route Reflection: An Alternative to Full Mesh Internal BGP (IBGP). RFC 4456 (Draft Standard), avril 2006.

[BDPR05] Larry Blunk, Joao Damas, Florent Parent et Andrei Robachevsky : Routing Policy Specification Language next generation (RPSLng). RFC 4012 (Proposed Standard), mars 2005.

[BFRW01] Allen Brown, Matthew Fuchs, Jonathan Robie et Philip Wadler : MSL A model for W3C XML Schema. *In WWW '01: Proceedings of the 10th international conference on World Wide Web*, pages 191–200, New York, NY, USA, 2001. ACM.

[BM04] Paul V. Biron et Ashok Malhotra : XML Schema Part 2: Datatypes Second Edition. W3C recommendation, W3C, octobre 2004.

[BMTM04] David Beech, Noah Mendelsohn, Henry S. Thompson et Murray Maloney : XML Schema Part 1: Structures Second Edition. W3C recommendation, W3C, octobre 2004.

[CBH+03] Geoff Coulson, Gordon S. Blair, David Hutchison, Ackbar Joolia, Kevin Lee, Jo Ueyama, Antônio Tadeu A. Gomes et Yimin Ye : NETKIT: a software component-based approach to programmable networking. *Computer Communication Review*, 33(5):55–66, 2003.

[CFSD90] Jeffrey D. Case, Mark Fedor, Martin Lee Schoffstall et James R. Davin : Simple Network Management Protocol (SNMP). RFC 1157 (Historic), mai 1990.

[CGG+04] Donald F. Caldwell, Anna Gilbert, Joel Gottlieb, Albert G. Greenberg, Gísli Hjálmtýsson et Jennifer Rexford : The cutting EDGE of IP router configuration. *Computer Communication Review*, 34(1):21–26, 2004.

[Cis98] Cisco : Cisco IOS Configuration Fundamentals, 1998. Documentation from the Cisco IOS reference Library.

[Cla99] James Clark : XSL Transformations (XSLT) Version 1.0. W3C recommendation, W3C, novembre 1999.

[DB06] Kevin Dooley et Ian Brown : *Cisco IOS Cookbook*. O'Reilly Media, Inc., 2006.

[dBBGdR06] Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf et Willem P. de Roever, éditeurs. *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*, volume 4111 de *Lecture Notes in Computer Science*. Springer, 2006.

[DH98] Stephen E. Deering et Robert M. Hinden : Internet Protocol, Version 6 (IPv6) Specification. RFC 2460 (Draft Standard), décembre 1998.

[Doy98] Jeff Doyle : *CCIE Professional Development, Routing TCP/IP, Volume 1*. Cisco Press, 1998.

[EAK05] Khalid El-Arini et Kevin Killourhy : Bayesian Detection of Router Configuration Anomalies. *In MineNet '05: Proceedings of the 2005 ACM SIGCOMM workshop on Mining network data*, pages 221–222, New York, NY, USA, 2005. ACM.

[Enn06] Rob Enns : NETCONF Configuration Protocol. RFC 4741 (Proposed Standard), décembre 2006.

[FB05] Nick Feamster et Hari Balakrishnan : Detecting BGP Configuration Faults with Static Analysis (Awarded Best Paper). *In NSDI*, 2005.

[Fea04] Nick Feamster : Practical verification techniques for wide-area routing. *Computer Communication Review*, 34(1):87–92, 2004.

[FR01] Anja Feldmann et Jennifer Rexford : IP Network Configuration for Intradomain Traffic Engineering, 2001.

[FW04] David C. Fallside et Priscilla Walmsley : XML Schema Part 0: Primer Second Edition. W3C recommendation, W3C, octobre 2004.

[Gar06] Aviva Garrett : *JUNOS Cookbook*. O'Reilly Media, Inc., 1 édition, 4 2006.

[Geo02] George M. Jones : RAT, The Router Audit Tool, février 2002.

[GR00] Lixin Gao et Jennifer Rexford : Stable Internet Routing without Global Coordination. *In SIGMETRICS '00: Proceedings of the 2000 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 307–317, New York, NY, USA, 2000. ACM.

[GS02] Barry Raveendran Greene et Philip Smith : *Cisco ISP Essentials*. Cisco Press, 2002.

[Hal01] Sam Halabi : *Internet Routing Architectures - Second Edition*. Cisco Press, 2001.

[HM02] Elliote Rusty Harold et W. Scott Means : *XML in a Nutshell*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2002.

[Iva04] Ivan Santarelli and Alexandra Bellogini : NetML, A Network Markup Language, janvier 2004.

[Jun04] Juniper Networks. *Advanced Juniper Networks Routing. Student Guide*, 6.c édition, septembre 2004.

[Jun05] Juniper Networks. *Configuring Juniper Networks Routers, Volume 2. Student Guide*, 6.b édition, avril 2005.

[KD02] Matt Kolon et Jeff Doyle, éditeurs. *Juniper Networks(r) Routers: The Complete Reference*. Osborne/McGraw-Hill, février 2002.

[Kep04] Stephan Kepser : A Simple Proof for the Turing-Completeness of XSLT and XQuery. *In Extreme Markup Languages*, 2004.

[Kun02] Kunihiro Ishiguro : GNU Zebra, The Free Routing Software, février 2002.

[LLW+06] Franck Le, Sihyung Lee, Tina Wong, Hyong S. Kim et Darrell Newcomb : Minerals: using data mining to detect router misconfigurations. *In MineNet*, pages 293–298, 2006.

[Mat04] Miroslav Matuska : Metaconfiguration of the Computer Network, 2004.

[Mey92] Bertrand Meyer : Applying "Design by Contract". *IEEE Computer*, 25(10):40–51, 1992.

[Moy98] John Moy : OSPF Version 2. RFC 2328 (Standard), avril 1998.

[MSO+99] David Meyer, Joachim Schmitz, Carol Orange, Mark Prior et Cengiz Alaettinoglu : Using RPSL in Practice. RFC 2650 (Informational), août 1999.

[MWA02] Ratul Mahajan, David Wetherall et Thomas E. Anderson : Understanding BGP misconfiguration. *In SIGCOMM*, pages 3–16, 2002.

[Nic04] Nick Feamster and Hari Balakrishnan : Verifying the Correctness of Wide-area Internet Routing, 2004.

[Pos81] J. Postel : Internet Protocol. RFC 791 (Standard), septembre 1981. Updated by RFC 1349.

[PS03] Holger Peine et Reinhard Schwarz : A Multi-View Tool for Checking the Security Semantics of Router Configurations. *ACSAC*, 0:56, 2003.

[PSMY+06] Jean Paoli, C. M. Sperberg-McQueen, François Yergeau, Eve Maler et Tim Bray : Extensible Markup Language (XML) 1.0 (Fourth Edition). W3C Recommendation, W3C, août 2006.

[QU05] Bruno Quoitin et Steve Uhlig : Modeling the routing of an Autonomous System with C-BGP. *IEEE Network*, 19(6), November 2005.

[RLH06] Yakov Rekhter, T. Li et Susan Hares : A Border Gateway Protocol 4 (BGP-4). RFC 4271 (Draft Standard), janvier 2006.

[RMK+96] Yakov Rekhter, Robert G Moskowitz, Daniel Karrenberg, Geert Jan de Groot et Eliot Lear : Address Allocation for Private Internets. RFC 1918 (Best Current Practice), février 1996.

[Sch03] Jürgen Schoenwaelder : Overview of the 2002 IAB Network Management Workshop. RFC 3535 (Informational), mai 2003.

[SPMF03] Jürgen Schoenwaelder, Aiko Pras et Jean-Philippe Martin-Flatin : On the future of Internet management technologies. *Communications Magazine, IEEE*, 41(10):90–97, Oct 2003.

[SRF+07] Jérôme Siméon, Jonathan Robie, Daniela Florescu, Don Chamberlin, Mary F. Fernández et Scott Boag : XQuery 1.0: An XML Query Language. W3C recommendation, W3C, janvier 2007.

[Sta93] William Stallings : *SNMP, SNMPv2, and CMIP: The Practical Guide to Network Management*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1993.

[Tid01] Doug Tidwell : *XSLT*. O'Reilly Media, Inc., 2001.

[Wal07] Priscilla Walmsley : *XQuery*. O'Reilly Media, Inc., 2007.

[WB08] Laurence Wolsey et Vincent Blondel : INMA1691: Graphes et Algorithmique, 2008.

Webographie

[1] Internet2/Abilene Network home page. http://www.internet2.edu/network/.

[2] Internet2 BGP Communities. http://www.abilene.iu.edu/i2network/bgp-communities.html.

[3] Abilene Router's Configuration. http://pea.grnoc.iu.edu/Abilene/configs/configs.html.

[4] J. Farrar. C&W Routing Instability. NANOG mail archives. http://www.merit.edu/mail.archives/nanog/2001-04/msg00209.html.

[5] S. A. Misel. Wow, AS7007! NANOG mail archives. http://www.merit.edu/mail.archives/nanog/1997-04/msg00340.html.

[6] Cisco systems. http://www.cisco.com.

[7] JDom. http://www.jdom.org/.

[8] Juniper Techpubs home page. http://www.juniper.net/techpubs/.

[9] Juniper Networks. JunOS Software. http://www.juniper.net/products/junos/.

[10] OPNET home page. http://www.opnet.com/.

[11] Team Cymru. Reading Room. Documents. http://www.team-cymru.org/ReadingRoom/Documents/.

[12] TOTEM, a TOolbox for Traffic Engineering Methods home page. http://totem.info.ucl.ac.be.

[13] WANDL home page. http://www.wandl.com/.

[14] S. Dhillon. YouTube IP Hijacking. NANOG mail archives. http://www.merit.edu/mail.archives/nanog/msg06299.html.

[15] YouTube Hijacking: A RIPE NCC RIS case study. http://www.merit.edu/mail.archives/nanog/msg06299.html.