tel-00580926, version 1 - 29 Mar 2011

Panthéon-Sorbonne

N° attribué par la bibliothèque

THÈSE

pour obtenir le grade de

Docteur de l'Université Paris 1 Panthéon-Sorbonne

Spécialité : **Informatique**

préparée au laboratoire **Centre de Recherche en Informatique**

dans le cadre de l'École Doctorale **Ecole Doctorale Sciences du Management/GODI**

présentée et soutenue publiquement

par

Hicham Idabal

le 25 novembre 2010

Titre:

Contribution à l'analyse de l'impact de Mises à jour sur des Vues et sur des Contraintes d'Intégrité XML

Directeur de thèse: **Françoise Gire**

Jury

Professeur. Véronique Benzaken,	Rapporteur
Professeur. Mírian Halfeld Ferrari Alves,	Rapporteur
Professeur. Colette Rolland,	Examinateur
Professeur. Sophie Tison,	Examinateur
Professeur. Dominique Laurent,	Examinateur
Professeur. Françoise Gire,	Directeur de thèse

A mes parents qui m'ont tant donné pour faire de moi ce que je suis,
A ma femme Samia ma complice de toujours,
A Youssef et Adam qui d'une flamme nouvelle ont éclairé ma vie,
et à tous pour leur soutien, leur affection et leur amour.

tel-00580926, version 1 - 29 Mar 2011

tel-00580926, version 1 - 29 Mar 2011

Remerciements

La thèse est une école, une découverte, une aventure, un tremplin aussi. Ce mémoire en est la consécration, et cette page, malgré sa position, le point final.

En premier lieu, j'aimerais remercier vivement Madame Françoise Gire, Professeur à l'université Paris 1 Panthéon - Sorbonne et directeur de ma thèse, pour avoir su me motiver, supporter mes égarements durant ces années, mais également la remercier de sa patience et de sa pédagogie et surtout pour ces moments de recherche excitants et agréables. Merci surtout pour cet encadrement génial et tout le temps que tu m'as donné.

Un grand merci également à Madame Colette Rolland, Professeur à l'université Paris 1 Panthéon - Sorbonne pour m'avoir accueilli dans son équipe.

Je remercie sincèrement Madame Véronique Benzaken, Professeur à l'Université d'Orsay Paris 11, et Madame Mírian Halfeld Ferrari Alves, Professeur à l'Université d'Orléans, qui ont accepté de juger ce travail et d'en être les rapporteurs. Je les remercie pour les remarques (toujours pertinentes), les conseils (toujours bienvenus) et la gentillesse.

J'adresse mes plus sincères remerciements aux personnes qui ont accepté de faire partie de mon jury de thèse.

Je ne remercierai jamais assez ma femme Samia. Il faut savoir être bref, et je me contenterai de dire que sans elle, sans son soutien et son amour, ce mémoire n'aurait pas abouti.

La thèse n'est pas qu'une aventure scientifique. Elle fut également l'occasion de rencontrer des gents différents, de tous horizons et toutes nationalités. Citer des noms ne servirait qu'à en oublier, et j'espère qu'ils savent, de toute façon.

Mes remerciements vont aussi à tous mes collègues du Centre de Recherche en Informatique de Paris 1 qui ont contribué par leur aide et leur sympathie durant

toutes ces années.

Je tiens à exprimer mes vifs remerciements à tous ceux qui, par leurs travaux, leurs idées, leurs présentations, leurs collaborations ou leurs relectures, ont participé de prés ou de loin à la réalisation de ce travail, en particulier l'équipe codex : Mírian Halfeld Ferrari Alves, Béatrice Bouchou, Martin Musicante, Pierre Réty et les autres.

Un grand merci à tous ceux qui m'ont épaulé durant cette épopée. Et tous les autres à qui j'adresse toute ma gratitude.

tel-00580926, version 1 - 29 Mar 2011

tel-00580926, version 1 - 29 Mar 2011

Table des matières

tel-00580926, version 1 - 29 Mar 2011

tel-00580926, version 1 - 29 Mar 2011

tel-00580926, version 1 - 29 Mar 2011

tel-00580926, version 1 - 29 Mar 2011

«Les gens heureux sont ceux qui privilégient l'essentiel par rapport à l'accessoire, l'être à l'avoir, l'utile à l'agréable, le durable à l'éphémère, le suffisant au trop, le nécessaire au superflu, en fait les besoins aux désirs.»

Robert Blondin (Le Bonheur possible, Éditions de l'Homme, p. 78)

tel-00580926, version 1 - 29 Mar 2011

tel-00580926, version 1 - 29 Mar 2011

Introduction

1 Contexte et motivation

Le langage XML, eXtensible Markup Langage, introduit il y a un peu plus de dix ans s'est imposé comme le standard pour structurer les documents textes sous forme d'arbres au moyen de balises en vue : de les stocker, de les transporter et de les échanger. La syntaxe d'un document XML est une suite de balises bien imbriquées, dont certaines contiennent des données textuelles. Ceci diffère des bases de données relationnelles, où les données sont stockées dans des tables.
Cette syntaxe, à la fois simple et flexible, permet de représenter linéairement tout type de données structurées hiérarchiquement. Son succès dans la représentation de données échangées sur internet est à l'origine d'un champ de recherche, actuellement en plein essor, sur les données XML.

Comme dans le domaine relationnel, les travaux de recherche sur les données XML se sont inscrits dans deux problématiques majeures :

1. d'une part, interroger des documents XML et extraire les données pertinentes pour telle ou telle application. Pour cela des langages d'interrogation ou de requêtes ont été définis, comme les standards du W3C XPath, XSLT, XQuery ou des langages plus formels à base d'automates d'arbres, d'expressions de chemins ou de patterns arborescents [NS00, NS02, Deb05, CDG+07, BCCM08, HP01]. Leurs pouvoirs d'expression ont été étudiés et comparés les uns aux autres ;
2. d'autre part, contraindre les données contenues dans un document XML à respecter un certain nombre de contraintes, par le biais de schémas de structure ou de contraintes d'intégrité. Ici, divers formalismes d'expression de schéma ont été proposés comme les DTDs, les XMLSchéma, ou de manière plus formelle les automates d'arbres. De nombreux travaux ont également proposé des formalismes d'expression de contraintes pour la définition de dépendances fonctionnelles ou de clés [FS00, FL02, DT05] par exemple.

Ces deux problématiques majeures ont été étudiées, comme dans le cas relationnel à la fois dans un contexte statique, où les données n'évoluent pas et plus récemment dans un contexte dynamique prenant en compte les différentes mises à

jour pouvant affecter les données d'un document.

Une spécificité du contexte XML, par rapport au cadre relationnel, est que chacune de ces problématiques (expression de requêtes, expression de contraintes, définition de mises à jour) nécessite du fait de la structure arborescente des données XML de s'appuyer sur un mécanisme de base permettant de sélectionner un ensemble de nœuds dans l'arborescence représentant le document XML. En effet, ce mécanisme est la base même d'un langage de requêtes qui doit extraire un ensemble de nœuds d'un document ; il est également indispensable pour un langage de mises à jour qui doit extraire l'ensemble des nœuds qui doivent être mis à jour ; enfin, il est nécessaire pour désigner les nœuds du document XML dont les valeurs sont impliquées par les contraintes d'intégrité que l'on souhaite définir. Le standard, proposé par le W3C, pour un tel mécanisme est le langage XPath, que l'on retrouve effectivement comme noyau des langages de transformation/interrogation comme XSLT ou XQuery et dont certains fragments sont également utilisés dans les formalismes d'expression de contraintes d'intégrité pour XML.

Dans cette thèse, nous nous positionnons dans un contexte dynamique, où les données XML évoluent au cours du temps et sont soumises à un ensemble de mises à jour, et nous nous intéressons à deux problèmes majeurs, largement étudiés dans le cadre relationnel :

1. La détection de l'impact d'un ensemble de mises à jour sur une vue, définie par une requête.
2. La détection de l'impact d'un ensemble de mises à jour sur la satisfaction d'une dépendance fonctionnelle.

Dans le premier cas, l'objectif est de détecter si l'ensemble de mises à jour a modifié la vue : si ce n'est pas le cas, on dit qu'il y a indépendance de la vue par rapport aux mises à jour, et il est alors inutile de recalculer la vue.

Dans le deuxième cas, l'objectif est de détecter si l'ensemble de mises à jour a produit un nouveau document qui ne satisfait plus les contraintes qu'il satisfaisait avant les mises à jour, si ce n'est pas le cas, on dit qu'il y a indépendance des mises à jour par rapport aux contraintes et il est inutile de revalider le document mis à jour.

Certains travaux ont déjà étudié ces deux problèmes, dans le cadre de données XML [RS06,BC09,BHL09,Hal07]. Nous nous différencions d'eux principalement sur deux points :

– nous supposons que les données sources sont à priori indisponibles et nous étudions les deux problèmes en effectuant une analyse statique des définitions des mises à jour, et de la vue (cas1) ou de la dépendance fonctionnelle (cas 2). De plus, dans le cas où un schéma XML contraignant la structure des données est disponible, nous utilisons cette information supplémentaire dans notre analyse afin de détecter éventuellement plus de cas d'indépendance entre

tel-00580926, version 1 - 29 Mar 2011

les mises à jour et la vue ou la dépendance fonctionnelle selon le cas. La figure 1 schématise le cadre dans lequel nous nous plaçons ;

– nous prenons l'option d'utiliser, comme mécanisme de sélection de nœuds, dans la définition des mises à jour, des vues et des dépendances fonctionnelles, un langage formel, basé sur l'utilisation de motifs réguliers arborescents, permettant de définir des requêtes appelées 'requêtes arbres régulières' (en abrégé *RARs*). L'intérêt de cette approche est d'utiliser un formalisme indépendant de tout standard, que l'on comparera, du point de vue de son expressivité, aux principaux autres standards de la littérature.

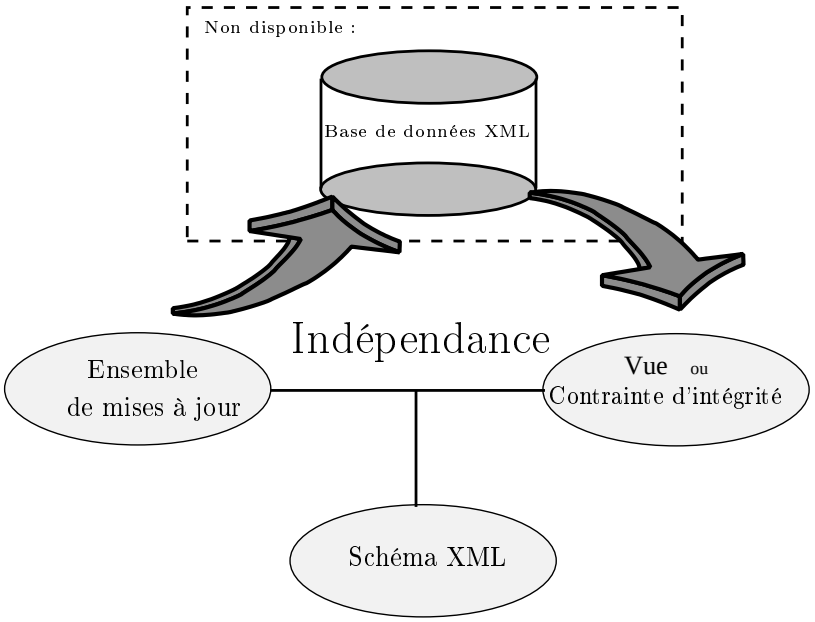


FIGURE 1 – Contexte de la thèse

Dans ce travail de thèse, nous nous somme fixé trois objectifs :

1. La définition d'un langage formel pour la sélection de nœuds sur des documents XML, puis le positionnement de ce langage par rapport aux langages existants.
2. L'utilisation de ce langage pour une analyse statique d'indépendance entre une vue et un ensemble de mises à jour.
3. La spécification des contraintes d'intégrité d'une part, et l'analyse statique d'indépendance entre une contrainte d'intégrité et un ensemble de mises à jour d'autre part.

tel-00580926, version 1 - 29 Mar 2011

2 Contribution de la thèse

Nous présentons à présent nos contributions. Nous commençons par l'étude des langages de sélection de nœuds, par la suite nous donnons la définition du langage des requêtes arbres régulières qui va être utilisé dans les deux scénarios pour l'analyse statique d'indépendance par rapport à un ensemble de mises à jour.

2.1 Étude des langages de sélection de nœuds

Nous nous intéressons aux requêtes n-aires. Celles-ci sélectionnent des n-uplets de nœuds dans les arbres XML.
XPath [XPa99] est le standard W3C pour la sélection de nœuds dans les documents XML. XPath est basé sur la description de chemins, constitués de suites d'étapes à suivre pour atteindre les nœuds sélectionnés. XPath permet également d'ajouter des filtres à chaque étape. Un filtre est une combinaison booléenne d'expressions de chemins, et est satisfait si un nœud satisfait cette combinaison. Il est également possible de tester le contenu textuel des nœuds. XPath est un langage de requête pivot, utilisé comme mécanisme de sélection de nœuds dans de nombreux autres langages.

Dans presque tous les travaux concernant XPath, les auteurs se restreignent à des fragments de XPath pour simplifier le modèle complet de XPath 1.0. Parmi ces fragments on trouve $CoreXPath$ et le fragment simple $XP^{\{[],*,//\}}$, ce dernier étant équivalent à la notion de "tree pattern" introduite en [MS04]. Beaucoup d'extensions de XPath 1.0 ont été proposées notamment le standard W3C XPath 2.0 qui introduit l'utilisation de variables [XPa07] qui permettent à XPath de définir des requêtes n-aire.

Nous optons ici pour un langage de sélection plus abstrait basé sur la notion de requête arbre régulière $\mathcal{RAR}$; bien qu'incomparable avec XPath, ce langage permet cependant d'exprimer toutes les requêtes positives de $CoreXPath$ (le fragment navigationnel de XPath) , ainsi que des requêtes non exprimables en XPath comme par exemple "Sélectionner les nœuds qui sont à une distance paire de la racine". Plus précisément, nous montrons que toute requête positive de $CoreXPath$ ($CoreXPath^{+}$) est exprimable par une union de requêtes $\mathcal{RAR}$s.
Une requête arbre régulière est formellement représentée par un arbre dont les arêtes sont étiquetées par des expressions régulières, et dont certains nœuds sont grisés pour identifier le n-uplet de nœuds de sélection. Nous montrons que les requêtes $\mathcal{RAR}$s ne sont pas comparables avec le langage XPath conditionnel introduit par Marx [Mar04a, Mar05b], qui étend $CoreXPath$ par l'utilisation de l'étoile de Kleene sur des chemins simples pour exprimer des conditions plus complexes d'accessibilité : le langage XPath conditionnel, noté CXPath est FO complet (c'est à dire qu'il permet d'extraire tout ensemble de nœuds caractérisé par une propriété exprimable par une formule de la logique du premier ordre). Par contre nous montrons que les

tel-00580926, version 1 - 29 Mar 2011

ensembles de nœuds extraits par une requête $\mathcal{RAR}$ sont caractérisés par une propriété exprimable par une formule de FOREG ; FOREG est une extension de la logique du premier ordre par les expressions régulières, introduite dans [NS00].
Le schéma ci-dessous récapitule le positionnement du pouvoir d'expression des requêtes $\mathcal{RARs}$ par rapport aux différents langages cités.

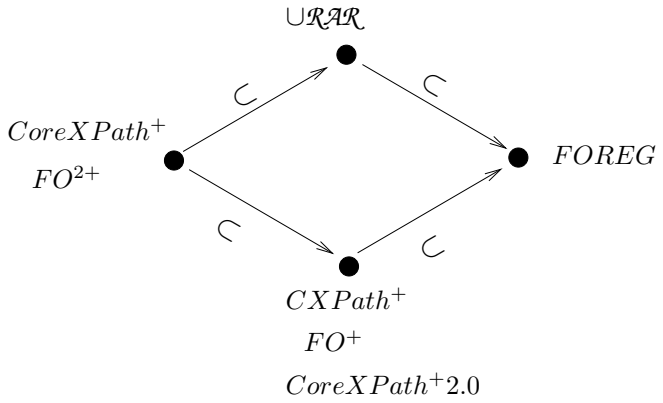


FIGURE 2 – Positionnement des $\mathcal{RARs}$

2.2 Utilisation des $\mathcal{RARs}$ pour l'analyse d'indépendance

Indépendance entre vues et classe de mises à jour

Nous modélisons la vue et la mise à jour sous la forme de deux requêtes $\mathcal{RARs}$ $\mathcal{V}$ et $\mathcal{C}$. Dans la mesure où nous ne précisons pas la façon dont les nœuds sélectionnés par $\mathcal{C}$ seront modifiés, $\mathcal{C}$ représente en fait une classe de mises à jour possibles, celle des mises à jour ne modifiant que les nœuds sélectionnés par $\mathcal{C}$. L'énoncé du problème d'indépendance que nous étudions ici est alors le suivant : "Étant données une vue $\mathcal{V}$ et une classe $\mathcal{C}$ de mises à jour, déterminer si la vue $\mathcal{V}$ est indépendante de $\mathcal{C}$, c'est-à-dire si toute mise à jour q de $\mathcal{C}$ n'a aucun impact sur l'évaluation de $\mathcal{V}$, quel que soit le document source". Dans le cas où un schéma $\mathcal{S}c$ contraignant les documents sources est connu, on peut espérer que la connaissance de $\mathcal{S}c$ permette de détecter un nombre plus élevé de cas d'indépendance qu'en l'absence de schéma. Nous étudions donc également la variante suivante du problème précédent : "Étant donnés une vue $\mathcal{V}$, une classe $\mathcal{C}$ de mises à jour et un schéma $\mathcal{S}c$, déterminer si la vue $\mathcal{V}$ est indépendante de $\mathcal{C}$ dans le contexte $\mathcal{S}c$, c'est-à-dire si toute mise à jour q de $\mathcal{C}$ n'a aucun impact sur l'évaluation de $\mathcal{V}$, quel que soit le document source $\mathcal{D}$, valide par rapport à $\mathcal{S}c$, sur lequel $\mathcal{V}$ est évaluée".
Nous montrons que le problème de l'indépendance de $\mathcal{V}$ par rapport à $\mathcal{C}$ est en général PSPACE-difficile. Après une analyse statique, nous construisons un langage régulier d'arbre $\mathcal{L}$ qui contient des arbres XML valides par rapport au schéma $\mathcal{S}c$ et

tel-00580926, version 1 - 29 Mar 2011

vérifiant certaines conditions relatives à l'existence des deux structures de $\mathcal{V}$ et $\mathcal{C}$. Nous prouvons que la vacuité de ce langage $\mathcal{L}$ constitue une condition suffisante pour qu'une vue $\mathcal{V}$ soit indépendante d'une classe de mises à jour $\mathcal{C}$ dans le contexte $\mathcal{S}c$. Cette condition est évaluable en temps polynomial par rapport à la taille des entrées $\mathcal{V}$, $\mathcal{C}$ et $\mathcal{S}c$. Dans le cas particulier des requêtes de vues $\mathcal{V}$ vérifiant la condition que chaque nœud de $\mathcal{V}$ a au moins un nœud ancêtre ou descendant qui est un nœud de sélection, et où les nœuds de mises à jour sont des feuilles dans la classe de mises à jour $\mathcal{C}$, nous montrons que le problème de l'indépendance est décidable en temps polynomial : en effet la condition suffisante d'indépendance exhibée dans le cas général, devient alors également nécessaire.

Indépendance entre dépendances fonctionnelles et classe de mises à jour

La spécification des contraintes d'intégrité représente un défi majeur dans le traitement des contraintes d'intégrité dans le cadre XML. Plusieurs types de contraintes d'intégrité ont été proposées et étudiées dans le cadre XML [FS00,BDF+01, FL02,BDF+03]. Parmi les contraintes d'intégrités les plus étudiées, on distingue la contrainte de clé et les dépendances fonctionnelles.

Nous proposons une spécification des dépendances fonctionnelles pour documents XML avec le formalisme des requêtes arbres régulières $\mathcal{RAR}\mathcal{s}$. Ce formalisme peut fédérer la plupart des approches proposées pour la modélisation des dépendances fonctionnelles en XML, tout en permettant d'exprimer de nouveaux types de contraintes jusqu'alors non exprimables par les approches antérieures. En effet la plupart des approches utilisent des chemins linéaires simples alors qu'une structure arborescente est plus adaptée pour représenter des contraintes sur des arbres XML.
En modélisant conjointement les classes de mises à jour par les requêtes arbres régulières, nous montrons qu'il est possible de définir un langage régulier d'arbre dont la vacuité forme une condition suffisante testable en temps polynomial pour assurer l'indépendance entre une dépendance fonctionnelle et une classe de mises à jour dans le contexte d'un schéma (si ce dernier est disponible). Cependant, le problème de l'indépendance d'une dépendance fonctionnelle par rapport à une classe de mises à jour reste en général PSPACE-difficile.

3 Organisation de la thèse

Ce document est constitué de quatre chapitres. Le premier chapitre introduit les concepts théoriques nécessaires pour l'étude des langages de sélection de nœuds sur des documents XML. Dans le deuxième chapitre, nous introduisons le langage des requêtes arbres régulières $\mathcal{RAR}\mathcal{s}$: on établira une comparaison de ces requêtes $\mathcal{RAR}\mathcal{s}$ avec le fragment navigationnel de XPath, et aussi avec d'autres extensions de XPath qui ont plus de puissance d'expressivité. Dans le troisième chapitre nous

tel-00580926, version 1 - 29 Mar 2011

étudions le problème d'indépendance entre une vue et une classe de mises à jour dans le cas où vue et mises à jour sont exprimées par des requêtes *RARs*. Dans le dernier chapitre, nous proposons le formalisme des *RARs* pour la spécification de dépendances fonctionnelles, puis nous étudions le problème de l'indépendance entre des dépendances fonctionnelles définies sur un document XML et une classe de mises à jour.

4 Publications associées

Les résultats contenus dans cette thèse ont conduit à diverses publications dont la liste est donnée ci-dessous.

- Françoise Gire, Hicham Idabal : "Regular tree patterns : a uniform formalism for update queries and functional dependencies in XML", EDBT/ICDT Workshops March 22, 2010 -Lausanne, Switzerland.

- Hicham Idabal, Françoise Gire :"Requêtes arbres régulières pour l'analyse de dépendances entre vues et mises à jour de documents XML", Revue des sciences et techniques de l'information (RSTI), March 2010.

- Hicham Idabal and Françoise Gire. "Dépendence entre Vues et Mise à jour de Données Semi-Structurées ", Journées Bases de Données Avancées (BDA 2008), Guilherand-Granges (Ardèche, France).

- Françoise Gire, Hicham Idabal : Updates and Views dependencies in Semi-structured Databases", International Database Engineering and Applications Symposium (IDEAS 2008), Coimbra (Portugal).

- Hicham Idabal. "Vues et Mises à jour de Données Semi-Structurées", Conférence en Recherche d'Information et Applications (CORIA), Trégastel, France, March 2008.

tel-00580926, version 1 - 29 Mar 2011

« Lorsque vous avez éliminé l'impossible, ce qui reste, même si c'est improbable, doit être la vérité.»

(A. Conan Doyle, Le signe des quatre)

tel-00580926, version 1 - 29 Mar 2011

tel-00580926, version 1 - 29 Mar 2011

Chapitre 1

Préliminaires

Durant ces dernières années, le développement et le succès du standard XML, ont conduit à l'émergence d'un nouveau type de données semi-structurées, naturellement modélisées par des structures arborescentes.
Dans ce chapitre nous donnons quelques rappels sur le langage XML, puis nous introduisons le modèle arbre des documents XML qui va être utilisé tout au long de ce travail. La dernière section introduit quelques notions sur les automates d'arbre et leur utilisation dans le cadre d'XML.

1 XML, Données Semi-structurées et Arbres

1.1 Le langage XML

XML (Extensible Markup Language [BPSM+08]) est un standard de représentation et d'échange d'information sur le web, mis en place par le World Wide Web Consortium (W3C [W3C]). Il définit une syntaxe générique utilisée pour marquer les données avec un balisage simple et lisible par les humains. Ce langage est représenté comme un bon compromis entre la richesse de SGML (Standard Generalized Markup Language [Gol90]) et la simplicité de HTML et s'est actuellement imposé comme un format standard et générique pour les bases de données semi-structurées.
XML est une norme du W3C depuis février 1998. Ce langage présente quelques avantages : il est auto-descriptif car les balises décrivent la structure et le type des noms des données, il est indépendant des plates-formes et il permet de structurer les documents sous forme arborescente. XML est un métalangage de balises, cela signifie qu'il n'a pas de jeux de balises prédéfinies. Au contraire, XML permet aux développeurs et aux rédacteurs de définir les éléments dont ils ont besoin. Bien que XML soit flexible dans sa définition des éléments, il est strict sur de nombreux autres aspects, il fournit une grammaire aux documents XML qui réglemente la position et l'ordre des balises, les noms des éléments autorisés, la façon dont les attributs sont

```
<? xml version="1.0" encoding="iso-8859-1" ?>
  <Resources>
    <UFR ID="27" >
        <Nom>Info</Nom>
        <Tel>01245857554</Tel>
        <Publications>
           <Article>
              <Titre>automata theory for XML researchers</Titre>
              <Auteur>F. Neven</Auteur>
              <Type>Conf</Type>
           </Article>
           <Article>
              <Titre>Counter-free automata</Titre>
              <Auteur>S. Papert</Auteur>
              <Auteur>R. McNaughton</Auteur>
              <Type>Workshop</Type>
           </Article>
        </Publications>
   </UFR>
  </Resources>
```

FIGURE 1.1 – Un exemple de document XML

rattachés aux éléments et ainsi de suite. Cette grammaire permet le développement de parseurs XML qui peuvent lire et vérifier la correction de n'importe quel document XML. Les documents qui respectent cette grammaire sont dits "documents bien formés". Pour imposer aux documents XML des contraintes de structure plus spécifiques, on utilise des schémas (DTD, XMLSchema, RelaxNG, ...), qui permettent de décrire des classes de documents partageant les mêmes spécifications. Les documents XML associés à un schéma sont dits valides quand ils respectent toutes les contraintes imposées par ce schéma. La figure 1.1 illustre un document XML bien formé représentant des publications.

1.2 Représentation arborescente des documents XML

Le modèle des données semi-structurées [Suc98, ABS00] peut être vu comme un relâchement du modèle relationnel classique, un des fondements des bases de données traditionnelles, dans lequel on autorise une structure moins rigide et homogène des "champs de données". Ce modèle de données s'est révélé très utile dans la représentation de familles de documents variés : multimédia, hypertexte, données scientifiques et autres.
Dans ce travail les données semi-structurées (DSS) considérées sont des documents

tel-00580926, version 1 - 29 Mar 2011

XML. La structure d'un document XML, si on s'abstrait de détails de moindre importance, est un arbre dans lequel les nœuds sont étiquetés, et l'ordre entre les nœuds enfants est significatif. La figure 1.2 montre l'arbre du document de la figure 1.1.

Ainsi nous modélisons un document XML par un arbre ordonné étiqueté sur un

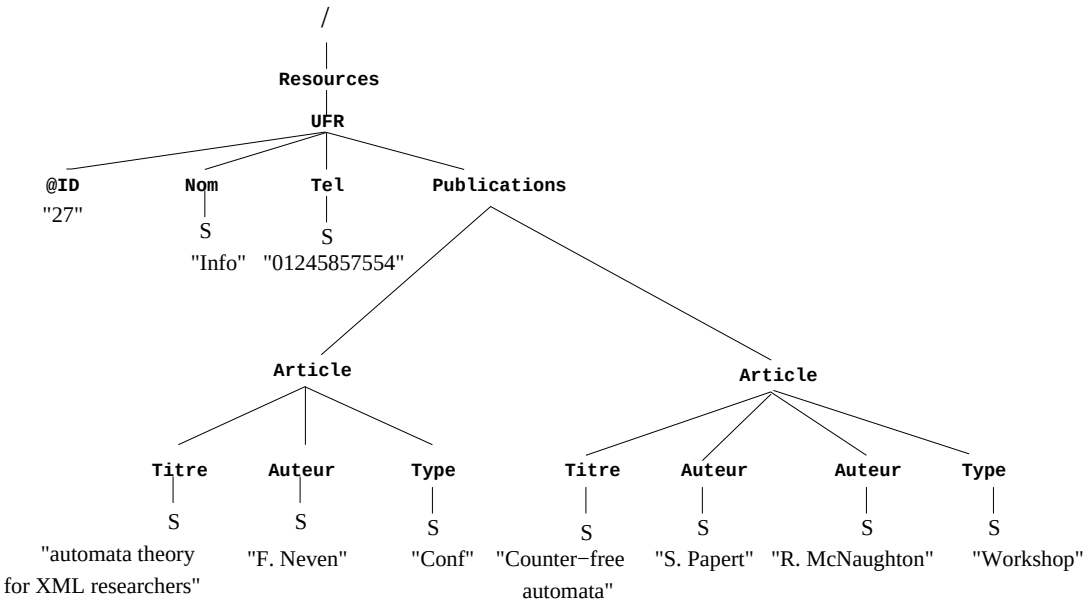


FIGURE 1.2 – Une représentation en arbre du document XML figure 1.1

alphabet à arité variable Σ i.e le nombre de fils d'un nœud n'est pas déterminé par son étiquette. Comme le montre la figure 1.2, nous choisissons de représenter les attributs d'un élément e par des nœuds feuilles de l'arbre XML, fils du nœud qui représente e et les contenus textuels du document XML par des nœuds feuilles étiquetés par le symbole S. Les étiquettes des nœuds feuilles qui représentent les attributs seront préfixées par @. Les valeurs textuelles des attributs et ceux des nœuds S sont des chaînes de caractères définies sur un alphabet auxiliaire I. Elles sont indiquées sous forme d'annotations sur la figure 1.2. Ainsi chaque nœud du document a un type : $''e''$ pour les nœuds représentant un élément XML, $''a''$ pour les nœuds feuilles représentant un attribut et $''t''$ pour les nœuds feuilles étiquetés S et représentant un contenu textuel. Nous supposons que l'alphabet Σ est partitionné en trois ensembles disjoints, $\Sigma = El \cup A \cup \{S\}$, ou les étiquettes de El étiquettent les nœuds éléments, ceux de A étiquettent les nœuds attributs et S étiquette les nœuds textuels. Formellement,

tel-00580926, version 1 - 29 Mar 2011

Définition 1. *Un document $\mathcal{D}$ est un triplet $\mathcal{D}=(d, \lambda, val)$ où*

- *d est un domaine d'arbre, i.e. d est un sous-ensemble de $\mathbb{N}^*$ contenant le mot vide ε et vérifiant $\forall i \in \mathbb{N}$, $wi \in d \Rightarrow (w \in d$ et $wj \in d, \forall j < i)$). Les nœuds internes sont de type "e", et chaque nœud feuille est soit de type "a" soit de type "t". Dans la suite, le domaine d de l'arbre $\mathcal{D}$ sera noté $\mathcal{N}(\mathcal{D})$.*
- *λ associe à chaque nœud de d une étiquette $l \in \Sigma$ conforme à son type. Ainsi $\lambda(w) \in A \cup \{S\}$ si et seulement si w est un nœud feuille de d.*
- *val est la fonction d'évaluation de d dans $d \cup I^*$. Cette fonction est l'identité sur les nœuds éléments i.e $\forall w \in d$, $val(w) = w$ si et seulement si $\lambda(w) \in El$, et associe à chaque nœud feuille une chaîne de caractère de I^* i.e $val(w) \in I^*$ si et seulement si $\lambda(w) \in A \cup \{S\}$*

Pour chaque nœud w de $\mathcal{N}(\mathcal{D})$, nous notons $\mathcal{D}(w)$, le sous-arbre issu de w dans $\mathcal{D}$ défini par $\mathcal{D}(w) = (d_w, \lambda_w)$ avec $d_w = \{wv / v \in \mathbb{N}^*\} \cap d$ et λ_w est la restriction de λ à d_w. Pour des raisons techniques, nous supposerons d'autre part, que le nœud racine est toujours étiqueté par le symbole '/' de El dans chaque document $\mathcal{D}$.
Un chemin p du nœud w au nœud w' est une suite $p=(w_0 = w, \ldots, w_n = w')$ vérifiant : $\forall\ i\ 0 \leq i < n$, $\exists\ k$ un entier tel que $w_{i+1} = w_i k$. On note $\lambda(p)$ le mot de Σ^* défini par $\lambda(p)=\lambda(w_0)\lambda(w_1)...\lambda(w_n)$.

2 Automates et XML

Les arbres et les automates d'arbres sont étudiés depuis longtemps, cependant Murata [Mur98] est le premier à utiliser les automates d'arbres comme langage de définition de schémas pour XML. Par la suite les automates d'arbres seront utilisés dans plusieurs travaux pour modéliser les types de données XML [Nev02]. Ainsi le langage de schéma RELAX NG [CM01] est inspiré directement des automates d'arbres finis. Une comparaison détaillée de quelques langages de schémas est donnée dans [MLMK05].
Nous commençons par rappeler les notions de régularité et d'automate fini dans le cadre des langages de mots.

2.1 Langages réguliers et automates de mots

Un langage régulier ou langage rationnel est un langage formel que l'on peut définir au moyen d'une expression régulière sur un alphabet.

tel-00580926, version 1 - 29 Mar 2011

tel-00580926, version 1 - 29 Mar 2011

Expressions régulières

Les expressions régulières ou rationnelles, qu'on appelle parfois motifs réguliers, sont issues de la théorie des langages formels. Leur capacité à décrire des ensembles réguliers explique qu'elles soient utilisées dans plusieurs domaines scientifiques, notamment dans le cadre du pattern matching (recherche d'une sous-chaîne dans un texte).
On se donne un ensemble Σ de caractères (ou symboles), dénommé alphabet. Un mot sur l'alphabet Σ est une suite finie de caractères. L'ensemble des mots sur Σ est noté par Σ^*. Un langage est un sous-ensemble de Σ^*, c'est à dire un ensemble particulier de mots. Le mot vide, noté ϵ, est l'unique mot de longueur zéro de Σ^*.

Définition 2. -***Expression régulière*** *: Une expression régulière E sur un alphabet Σ et son langage associé $L(E)$ sont définis de façon inductive comme suit :*

- *Le mot vide ϵ est une expression régulière qui dénote le langage $L(\epsilon)=\{\epsilon\}$.*
- *Un caractère c (un élément de l'alphabet Σ) est une expression régulière qui dénote le langage $L(c)=\{c\}$.*
- *Si E_1 et E_2 sont des expressions régulières, alors l'alternative $E_1 \mid E_2$ est une expression régulière qui dénote l'union $L(E_1) \cup L(E_2)$.*
- *Si E_1 et E_2 sont des expressions régulières, alors la concaténation $E_1.E_2$ est une expression régulière qui dénote la concaténation $L(E_1).L(E_2) = \{w_1w_2$ / $w_1 \in L(E_1)$ et $w_2 \in L(E_2)\}$.*
- *Si E est une expression régulière, alors la répétition (étoile de Kleene) E^* est une expression régulière qui dénote la fermeture $(L(E))^* = \bigcup_{j\in\mathbb{N}} (L(E))^j$ où $L(E)^0 = \{\epsilon\}$ et $L(E)^j = L(E)^{j-1}.L(E)$ $\forall j \geq 1$.*

Syntaxiquement une expression régulière E est donc engendrée par la grammaire :

$$E ::= E|E \parallel E.E \parallel E^* \parallel \epsilon \parallel c$$

Notations :

- Quotient d'une expression régulière E par un mot w : $w^{-1}E$
  Soit w un mot de Σ^* et E une expression régulière, on montre que le langage $\{ w' \in \Sigma^*$ tel que $w.w' \in L(E) \}$, noté $w^{-1}L(E)$, est un langage régulier. L'expression régulière associée est notée $w^{-1}E$.
- On note E^+ l'expression régulière $E.E^*$.

Définition 3. -***Langage régulier*** *: Un langage $L \subset \Sigma^*$ est dit régulier si et seulement s'il est associé à une expression régulière, c'est à dire il existe une expression régulière E tel que $L=L(E)$.*

Le problème de savoir comment déterminer si un mot w est dans un langage régulier L a été résolu par le théorème de Kleene ([Kle56]) qui établit que les automates d'états finis et les expressions régulières définissent la même classe de langages.

Automates de mots

Les automates sont des objets mathématiques, très utilisés en informatique [CDG+07], qui permettent de modéliser un grand nombre de systèmes (informatiques). L'étude des automates a commencé vers la fin des années cinquante. De façon très formelle, un automate est un ensemble "d'états du système", reliés entre eux par des "transitions" qui sont marquées par des symboles. Étant donné un "mot" fourni en entrée, l'automate lit les symboles du mot un par un et va d'état en état selon les transitions. Le mot lu est soit accepté par l'automate soit rejeté.

Définition 4. -***Automates finis déterministes*** *: Un automate fini déterministe est un quintuplé $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ constitué des éléments suivants :*

- *Un ensemble non vide fini d'états (Q)*
- *Un alphabet (Σ)*
- *Une fonction de transition ($\delta : Q \times \Sigma \to Q$)*
- *Un état de départ ou initial ($q_0 \in Q$)*
- *Un ensemble d'états finaux ou acceptants ($F \subseteq Q$)*

Fonctionnement : L'automate prend en entrée un mot et l'accepte ou le rejette. On dit aussi qu'il le reconnaît ou ne le reconnaît pas. A la lecture de chaque symbole, on emploie la fonction de transition δ pour se déplacer vers le prochain état (en utilisant l'état actuel et le caractère qui vient d'être lu, l'état de départ est q_0.), le mot est reconnu si et seulement si le dernier état (i.e. l'état correspondant à la lecture du dernier caractère du mot) est un état final de F.
Le langage associé à un automate est constitué de l'ensemble des mots qu'il reconnaît.
On représente souvent les automates de mots par un graphe enrichi d'étiquettes sur les arêtes et d'états initiaux et finaux. Une transition $q = \delta(p, a)$ est notée $p \xrightarrow{a} q$ à la manière d'une arête d'un graphe.

Exemple 1. *Soit $\mathcal{A}$ = ({1,2}, {a,b}, {δ(1,b)=1,δ(1,a)=2,δ(2,b)=2,δ(2,a)=1}, {1}, {1}) l'automate représenté par la figure 1.3. Cet automate a deux états 1 et 2. L'état 1 est à la fois initial (marqué d'une petite flèche entrante) et final (marqué d'une petite flèche sortante). Il possède quatre transitions représentées comme des arêtes d'un graphe.*

Notation : Soit $\mathcal{A}$=$(\Sigma, Q, q_0, \delta, F)$ un automate déterministe de mots. Quitte à supprimer des états dans Q sans changer le langage reconnu par $\mathcal{A}$, on peut toujours supposer que tout état q est accessible à partir de q_0 et permet d'atteindre au

tel-00580926, version 1 - 29 Mar 2011

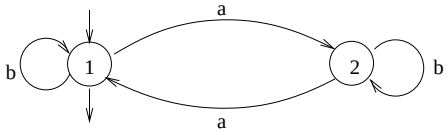


FIGURE 1.3 – Un automate avec quatre transitions

moins un état final de F. On utilisera dans la suite la notation $L_q = \mathrm{L}(\mathcal{A},\{q_0\},\{q\})$ (respectivement $\bar{L}_q = \mathrm{L}(\mathcal{A},\{q\},\mathrm{F})$) pour désigner l'ensemble des mots reconnus par $\mathcal{A}$ en utilisant q_0 (respectivement q) comme état initial et $\{q\}$ (respectivement F) comme ensemble d'état finaux.

Automates finis non-déterministes

Un automate fini non-déterministe est un automate tel que dans un état donné, il peut y avoir plusieurs transitions avec le même symbole. C'est la fonction de transition δ qui diffère ici de celle utilisée par les automates finis déterministes. Le fonctionnement d'un tel automate n'est donc pas totalement "déterminé", car on ne sait pas quel état l'automate va choisir. Les automates non-déterministes permettent de modéliser facilement des problèmes complexes. On démontre que, pour tout automate fini non-déterministe, il existe un automate fini déterministe reconnaissant le même langage. Celui-ci peut être exponentiellement plus grand que l'automate non déterministe de départ.
L'intérêt des automates par rapport aux autres méthodes pour décrire des langages est qu'ils rendent effectives de nombreuses opérations sur ces ensembles. Par exemple, étant donné deux automates, il est très facile de construire un automate reconnaissant l'union des langages qu'ils définissent et un automate qui reconnaît leur intersection. On sait également complémenter un automate donné $\mathcal{A}$ c'est à dire construire un automate qui reconnaît le complément du langage défini par $\mathcal{A}$.

Les automates finis et les expressions régulières ont la même expressivité, ce résultat bien connu est du à Kleene [Kle56].

Théorème 1. ***(Kleene)*** *: Un langage L est régulier si et seulement si il existe un automate fini $\mathcal{A}$ tel que L=$L(\mathcal{A})$.*

2.2 Automates d'arbres

Les automates d'arbres traitent des arbres, au lieu de chaînes de caractères. D'abord étudiés pour les arbres à arités fixes, et particulièrement les arbres binaires, les automates d'arbres à arité variables ont fait l'objet plus récemment de nombreux travaux, suite au développement du langage XML.

tel-00580926, version 1 - 29 Mar 2011

tel-00580926, version 1 - 29 Mar 2011

Comme pour les automates de mots, les automates d'arbres peuvent être déterministes ou non. Suivant la façon dont les automates se "déplacent" sur l'arbre qu'ils traitent, les automates d'arbres peuvent être de deux types : ascendants ou descendants. La distinction est importante, car si les automates non-déterministes (ND) ascendants et descendants sont équivalents, les automates déterministes descendants sont strictement moins puissants que leurs homologues déterministes ascendants. Nous présentons dans la suite les automates d'arbres ascendants (bottom-up) que nous utiliserons dans le cadre de cette thèse.

Les automates d'arbres ascendants ont été introduits par Doner [Don70] et Thatcher et Wright [TW68] dans le cadre des arbres à arités fixes afin de prouver l'équivalence entre les langages d'arbres MSO définissables et les langages rationnels d'arbres. Un automate d'arbres ascendant a un contrôle parallèle : il évalue un arbre, des feuilles à la racine, en affectant à chaque nœud un état en fonction de son étiquette et des états affectés à ses fils.
Les automates d'arbres ascendants constituent le modèle d'automates d'arbres le plus utilisé car ils possèdent de nombreuses propriétés que ce soit dans le cadre des arbres à arités fixes ou variables : un automate d'arbres ascendant peut être déterminisé, complémenté, et les problèmes de vacuité (le langage reconnu par $\mathcal{A}$ est-il vide ?) et d'appartenance (l'arbre t est-il reconnu par $\mathcal{A}$) sont décidables en temps polynomial.
Nous donnons çi-dessous la définition formelle d'un automate d'arbres ascendant dans le cadre des arbres à arités variables.

Définition 5. *-**Automates d'arbres ascendants** (bottom-up automata) : Un automate d'arbres ascendant est un quadruplet (Q, Σ, F, δ) où Q est un ensemble fini d'états, Σ est un alphabet, $F \subseteq Q$ est un sous-ensemble de Q appelé ensemble des états acceptants et δ est un sous-ensemble de $\Sigma \times Q \times Q^*$ appelé ensemble des transitions. L'ensemble des transitions δ vérifie de plus la condition suivante :*
$\forall\, q \in Q\ \forall\, a \in \Sigma$, le langage $L(a,q)=\{w \in Q^* / (a,q,w) \in \delta\}$ est un langage régulier de mots de Q^*.

Soient t un arbre étiqueté sur Σ et $\mathcal{A}=(Q, \Sigma, F, \delta)$ un automate d'arbres ascendant. Un calcul de $\mathcal{A}$ sur t est une fonction de l'ensemble des nœuds de t à valeur dans Q qui vérifie :
- la feuille de l'arbre étiqueté par a reçoit l'état q si (a, q, ϵ) est dans δ .
- un nœud interne de l'arbre, étiqueté par a, reçoit l'état q sachant que ses nœuds fils ont reçu respectivement les états $q_1, q_2, ..., q_n$, si (a,q,$q_1q_2...q_n$) est dans δ .
Un calcul de $\mathcal{A}$ sur t est acceptant si l'état affecté à la racine est acceptant, on dit alors que l'arbre t est accepté par $\mathcal{A}$. Le langage reconnu par $\mathcal{A}$ est l'ensemble des arbres pour lesquels il existe un calcul acceptant de $\mathcal{A}$.
La famille des langages d'arbres reconnus par des automates d'arbres ascendants est appelée la classe des langages réguliers d'arbres.

Exemple 2. *Considérons l'ensemble $\mathcal{L}$ des arbres, définis sur $\Sigma = \{a, b, c, d\}$, dont tout nœud étiqueté par 'a' ne possède aucun descendant étiqueté par 'a'. Le langage $\mathcal{L}$ est reconnu par l'automate $\mathcal{A}$=(Q, Σ, F, δ) suivant :*
$Q = \{1, 2\}, \Sigma = \{a, b, c\}, F = \{2\}$ et δ définie par :
$L(x, 1)$=1^* pour x=b ,c ou d
$L(a, 2)$=1^*
$L(x, 2)$=Q^*2Q^* pour x=b ,c ou d
L'arbre de la figure 1.4 est accepté par $\mathcal{A}$ car il existe un calcul acceptant de $\mathcal{A}$.

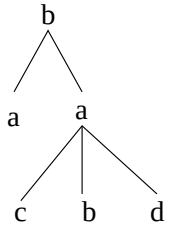


FIGURE 1.4 – Un arbre de L(A)

3 Logique et XML

Plusieurs logiques ont été proposées comme base d'expression de propriétés de documents XML. Dans ce qui suit nous présentons un trés bref rappel sur la logique du premier ordre ainsi que la logique monadique du second ordre.

3.1 Logique du premier ordre : FO

La logique du premier ordre, aussi appelée calcul des prédicats, est la logique des formules mathématiques usuelles, avec la contrainte que les variables représentent toutes des objets du même type.

Alphabet (ou signature) : La signature σ du calcul des prédicats est formé des symboles suivants :
- un ensemble de symboles de constante : a, b, c ...
- un ensemble de symboles de prédicat (ou de relation) d'arité fixée : A, B, P ...
- un ensemble de variables : x, y, z ...
- un ensemble de symboles de fonction d'arité fixée : f, g, h, ...

Termes : Les termes représentent les objets associés au langage, ils sont formés à partir des règles suivantes :
- Les constantes et les variables sont des termes.

tel-00580926, version 1 - 29 Mar 2011

- Si f est un symbole de fonction d'arité n et si $t_1...t_n$ sont des termes, alors $f(t_1,...,t_n)$ est un terme.

Formules : Les formules sont construites à partir des formules dites atomiques ($P(t_1, ..., t_n)$) en utilisant des connecteurs et des quantificateurs. On peut définir une formule FO ϕ sur σ de façon inductive par :

$$\phi ::= P(t_1, ..., t_n) \mid \phi \vee \phi \mid \neg\phi \mid \exists x\phi$$

où P est un symbole de prédicat d'arité n de σ et $t_1, ..., t_n$ des termes. $\vee$ (respectivement $\neg$) est le connecteur de disjonction (respectivement de négation), $\exists$ est le quantificateur existentiel. L'ensemble des formules FO sur σ est noté FO_σ.
Dans une formule logique, une variable qui n'est pas liée à un quantificateur existentiel ($\exists$) est dite libre. $\forall x\phi$ est une notation pour la formule $\neg(\exists x\neg\phi)$. La notation $\phi(x_1, x_2, ..., x_n)$ est utilisée pour préciser que $x_1, x_2, ..., x_n$ sont des variables libres de ϕ.

Modèle : Pour donner un sens vrai ou faux aux formules, il faut les interpréter. Pour cela il faut donner un sens aux fonctions, puis aux prédicats, et choisir un domaine dans lequel les variables seront instanciées. Une structure <D,I> est la donnée d'un domaine D et d'une interprétation I qui interprète chaque symbole de fonction et de prédicat.
Soit $\mathcal{M}$=<D,I> une structure. Une valuation ρ est une application des variables dans D. Si ϕ est une formule FO, on écrit $\mathcal{M} \models_\rho \phi$ si ϕ est vrai lorsque les symboles de fonctions et de prédicats sont interprétés selon I, et les variables libres de ϕ sont interprétées selon ρ. Formellement :

- ρ est étendu en une application sur l'ensemble des termes FO par les égalités :
  (1) $\rho(a)$=I(a) pour tout symbole de constante a,
  (2) $\rho(f(t_1, ..., t_n))$=I(f)($\rho(t_1), ..., \rho(t_n)$) ;
- $\mathcal{M} \models_\rho P(t_1, ..., t_n) \Leftrightarrow$ I(P)($\rho(t_1), ..., \rho(t_n)$) est vrai ;
- $\mathcal{M} \models_\rho \phi_1 \vee \phi_2 \Leftrightarrow \mathcal{M} \models_\rho \phi_1$ ou $\mathcal{M} \models_\rho \phi_2$;
- $\mathcal{M} \models_\rho \exists x\phi \Leftrightarrow$ il existe $d \in$ D tel que $\mathcal{M} \models_{\rho_d} \phi$ où ρ_d est la valuation qui coïncide avec ρ sur toute variable libre de ϕ distincte de x et qui satisfait $\rho_d(x) = d$.

Si $x_1, x_2, ..., x_n$ sont n variables libres de ϕ (ϕ pouvant éventuellement avoir d'autres variables libres), on utilise la notation $\mathcal{M} \models \phi(x_1, x_2, ..., x_n)[d_1, d_2, ..., d_n]$ pour exprimer le fait que $\mathcal{M} \models_\rho \phi(x_1, x_2, ..., x_n)$ dans toute valuation ρ vérifiant $\rho(x_i) = d_i$ pour $i = 1, ...n$.
On dit que $\mathcal{M}$ est un modèle de ϕ ($\mathcal{M} \models \phi$) si et seulement si $\mathcal{M} \models_\rho \phi$ pour toute valuation ρ.
ϕ est un théorème de FO si et seulement si $\mathcal{M} \models \phi$ pour tout modèle $\mathcal{M}$.

tel-00580926, version 1 - 29 Mar 2011

Décidabilité : Le calcul des prédicats est en général indécidable [Tur37] dans le sens où, il n'existe pas d'algorithme, qui, si on se donne une formule ϕ de FO, permette de décider si ϕ est un théorème ou non. Cela dépend cependant du langage utilisé, en particulier du choix des symboles primitifs (l'alphabet). Le calcul des prédicats égalitaire monadique (n'ayant que des symboles de prédicats unaires en dehors du prédicat binaire d'égalité et pas de symbole de fonction) est décidable. Le calcul des prédicats égalitaire pour un langage comportant au moins un symbole de prédicat binaire est indécidable. Le fragment FO^2 de FO, formé des formules de FO qui utilisent seulement deux variables x et y (libres ou non), est décidable [Mor75,GO99].

Logique du premier ordre pour les arbres ordonnés :

Les arbres étiquetés sur un alphabet Σ, ordonnés et à arités variables (on note Tree(Σ) l'ensemble de ces arbres) peuvent être vus comme des structures logiques [EF95] sur un alphabet composé de prédicats unaires correspondant aux étiquettes des nœuds, et un ensemble de prédicats binaires correspondant aux déplacements dans l'arbre. L'alphabet le plus utilisé dans la littérature [Nev02, Bar05, BDM$^+$06] est σ_{tree}=$\{R_{\Downarrow}$, $R_{\Rightarrow}$, $(O_a$, a $\in \Sigma$ $)\}$. Tout arbre T de Tree(Σ), étiqueté, ordonné, à arités variables est alors une structure $\mathcal{M}_T$=<D,I> où :

- D est l'ensemble des nœuds de T ;
- I(O_a) est l'ensemble des nœuds étiquetés par a ;
- I($R_{\Downarrow}$) (respectivement I($R_{\Rightarrow}$)) est la relation descendant (respectivement following_sibling) dans T.

On notera la logique du premier ordre sur cet alphabet interprétée sur des arbres étiquetés, ordonnés et à arités variables par FO_{tree}.
Les deux relations fils, noté $R_{\downarrow}$, et frère droit, noté $R_{\rightarrow}$, peuvent être exprimées dans FO_{tree} par :

$$\begin{aligned} xR_{\downarrow}y &\Leftrightarrow xR_{\Downarrow}y \wedge \neg\exists z(xR_{\Downarrow}z \wedge zR_{\Downarrow}y) \\ xR_{\rightarrow}y &\Leftrightarrow xR_{\Rightarrow}y \wedge \neg\exists z(xR_{\Rightarrow}z \wedge zR_{\Rightarrow}y) \end{aligned}$$

Cependant elles ne peuvent être exprimées avec des formules de FO_{tree} avec seulement deux variables.
On notera la logique du premier ordre sur l'alphabet $\{R_{\downarrow}$, $R_{\rightarrow}$, $R_{\Downarrow}$,$R_{\Rightarrow}$ $(O_a$, a $\in \Sigma$ $)\}$ et qui se restreint aux formules à deux variables (libres ou non) par FO^2_{tree}.

3.2 Extension de FO_{tree} par fermeture transitive

La fermeture transitive d'une relation ne pouvant être exprimée en FO_{tree}, plusieurs extensions de FO_{tree} par la fermeture transitive ont été proposées [NSsuJ01, Kel04, Leo06]. Nous définissons d'abord la fermeture transitive d'une formule ϕ puis quelques extensions de FO_{tree} par la fermeture transitive.

tel-00580926, version 1 - 29 Mar 2011

Définition 6. *(Fermeture transitive [EH07, ten06]) Pour toute formule $\phi(x,y)$, $TC_{x,y}\phi(x,y)$ est appelée fermeture transitive de ϕ et exprime que x et y restent dans la fermeture transitive de la relation définie par $\phi(x,y)$. Formellement,*
$M \models TC_{x,y}\phi(x,y)[d,e]$ si et seulement si il existe $d_1,...d_n$ du domaine de M tel que $d{=}d_1$, $e{=}d_n$, et $M \models \phi(x,y)[d_i,d_{i+1}]$ pour tout $i{<}n$.

$FO{+}TC1$ [EH07] : Le langage $FO{+}TC1$ étend FO_{tree} par l'opérateur de fermeture transitive sur les formules $\phi(x,y)$ de FO_{tree}, les autres variables libres de ϕ autres que x et y s'appellent des paramètres. Formellement, une formule ϕ de $FO+TC1$ est définie de manière inductive par :

$$\phi ::= P(t_1,...,t_n) \mid \phi \vee \phi \mid \neg\phi \mid \exists x\phi \mid TC_{x,y}\phi(x,y)$$

FO^* [ten06]) : Le langage FO^* étend FO_{tree} par l'opérateur de fermeture transitive sur les formules $\phi(x,y)$ de FO avec exactement deux variables libres qui sont x, et y. Ainsi FO^* est le fragment de $FO+TC1$ sans paramètres.

$FOREG$ [NS00]) : $FOREG$ est l'extension de FO_{tree} par des expressions régulières de chemins verticaux de la forme $\varphi = [E]^{\downarrow}_{s,t}(x,y)$ et horizontaux de la forme $\psi = [E]^{\rightarrow}_{s}(x)$.

Syntaxiquement, si E est une expression régulière sur un alphabet de symboles représentant des formules ayant au moins deux variables libres s,t alors $[E]^{\downarrow}_{s,t}(x,y)$ est une formule (chemin vertical) dont les variables libres sont x, y et celles apparaissant dans les formules de E.
Si E est une expression régulière sur un alphabet de symboles représentant des formules ayant au moins une variable libre s alors $[E]^{\rightarrow}_{s}(x)$ est une formule (chemin horizontal) dont les variables libres sont x et celles apparaissant dans les formules de E.

Sémantiquement, soit E une expression régulière sur un alphabet de symboles représentant des formules qui ont au plus deux variables libres, T un arbre et v, w deux nœuds de T. Le chemin vertical $\varphi = [E]^{\downarrow}_{s,t}(x,y)$ a pour sémantique : $T \models \varphi[v,w]$ si et seulement si w est un descendant de v, et il existe un étiquetage des arêtes sur le chemin de v à w par des formules utilisées dans E tel que (1) chaque arête (u,u') est étiquetée par une formule $\theta(s,t)$ tel que $T \models \theta[u,u']$, et (2) la séquence des étiquettes sur le chemin de v à w vérifie la contrainte régulière définie par E.

Exemple :
La formule $\phi(x,y){=}\ [(\theta_{ab}(s,t)\theta_{ba}(s,t))^*\theta_{ab}(s,t)]^{\downarrow}_{s,t}(x,y)$ avec $\theta_{ab}(s,t){=}O_a(s) \wedge O_b(t)$ et $\theta_{ba}(s,t){=}O_b(s) \wedge O_a(t)$, exprime la propriété qu'il existe un nœud y descendant de x tel que le mot formé à partir de la suite des étiquettes entre x et y est dans $\mathrm{L}((ab)^*)$.

tel-00580926, version 1 - 29 Mar 2011

Le chemin horizontal $\psi = [E]_s^{\rightarrow}(x)$ a pour sémantique : $T \models \psi[v]$ si et seulement si il existe un étiquetage des fils de v par des formules qui sont utilisées dans E tel que (1) chaque fils w de v est étiqueté par une formule $\theta(s)$ tel que $T \models \theta[w]$, et (2) la séquence des étiquettes vérifie la contrainte définie par E.

Exemple :
La formule ϕ=$\exists y \exists z\ [true^*(s = y)(O_a(s)^* O_b(s))(z = s)true^*]_s^{\rightarrow}(x)$ exprime la propriété qu'il existe deux nœuds y et z fils de x tel que le mot formé à partir de la suite des étiquettes entre y et z est dans L(a^*b).

$FOREG$ est l'extension de FO_{tree} par les formules de la forme $[E]_{s,t}^{\downarrow}(x, y)$ et $[E]_s^{\rightarrow}(x)$. Formellement, une formule ϕ de $FOREG$ est définie de manière inductive par :

$$\begin{array}{rcl} \phi & ::= & P(t_1, ..., t_n) \parallel \phi \vee \phi \parallel \neg\phi \parallel \exists x\phi \parallel [E]_{s,t}^{\downarrow}(x, y) \parallel [E]_s^{\rightarrow}(x) \\ E & ::= & E|E \parallel E.E \parallel E^* \parallel \epsilon \parallel \theta \end{array}$$

L'inclusion $FOREG \subsetneq FO^*$ s'établit facilement, car intuitivement $FOREG$ n'ajoute à FO que des fermetures transitives de formules qui ne sont satisfaites que sur des chemins verticaux ou horizontaux. Dans [NS00, ten06], les auteurs montrent que FO^* est une extension stricte de $FOREG$ en montrant que les circuits booléens peuvent être exprimés dans FO^* mais pas dans $FOREG$. Par contre FO^* est inclus dans $FO+TC1$. On a donc les inclusions suivantes : $FOREG \subsetneq FO^* \subseteq FO+TC1$

3.3 Logique du second ordre : SO

La logique du second ordre étend celle du premier ordre par l'ajout de variables relationnelles, qui peuvent donc être quantifiées. La logique monadique du second ordre (MSO) [TW68] se restreint aux variables relationnelles unaires, qui représentent les ensembles. Elle permet d'avoir des résultats de décidabilité plus intéressants qu'au premier ordre.

Formules MSO : Une formule MSO ϕ sur σ est définie de manière inductive par :

$$\phi ::= P(t_1, ..., t_n) \mid \phi \vee \phi \mid \neg\phi \mid \exists x\phi \mid \exists X\phi \mid x \in X$$

où P est un symbole de prédicat d'arité n de σ, $t_1, ..., t_n$ des termes, x une variable du premier ordre et X une variable du second ordre.

Équivalence MSO-Automates : La première correspondance entre MSO et les langages rationnels a été établie par Büchi sur les mots (chaines de caractère) [Büc60], puis a été étendu aux arbres binaires par Thatcher et Wright [TW68].

tel-00580926, version 1 - 29 Mar 2011

Ce résultat reste vrai sur les arbres à arité variables [CDG+07].

Soit Σ un alphabet. Comme dans le cas des arbres étiquetés sur Σ, on peut définir une signature logique σ_{word} interprétable sur les mots de Σ^* :
$\sigma_{word} = \{O_a, a \in \sigma\} \cup \{succ\}$.
Tout mot w de Σ^* peut alors être vu comme une σ_{word}-structure $\mathcal{M}_w$=<D,I> où :

- D est l'ensemble des positions des lettres de w ;
- I(succ) est la relation successeur ;
- I(O_a) est l'ensemble des positions de w associées à la lettre a.

Ainsi, un langage de mots (respectivement d'arbre) sur Σ (respectivement étiqueté sur Σ) peut être vu comme un ensemble de σ_{word}-structures (respectivement σ_{tree}-structures). Ceci conduit à la notion de $\mathcal{L}$-définissabilité d'un langage de mots ou d'arbres, où $\mathcal{L}$ est une logique.

Définition 7. *Soit ϕ un énoncé d'une logique $\mathcal{L}$ de signature σ_{word} (respectivement σ_{tree}), on note L(ϕ)={$w \in \Sigma^* / \mathcal{M}_w \models \phi$} (respectivement L($\phi$)={$T \in Tree(\Sigma) / \mathcal{M}_T \models \phi$}).*
Un langage L est définissable dans $\mathcal{L}$ s'il existe un $\mathcal{L}$-énoncé ϕ tel que L=L(ϕ).

Théorème 2. *- Un langage de mots finis est reconnaissable par un automate de mots fini si et seulement si il est MSO-définissable [Büc60].*
- Un langage d'arbres ordonnés d'arité variable est reconnaissable par un automate d'arbre ascendant si et seulement si il est MSO-définissable [NS02].

Remarque : Toutes les logiques avec clôtures transitives ont un pouvoir d'expression inférieur ou égal à celui de la logique MSO. Il est toujours possible d'exprimer un opérateur de clôture transitive avec une formule MSO.

3.4 Langage de mots sans étoile

Définition 8. *On dit que L est un langage de mots sans étoile s'il peut s'obtenir à partir de $\emptyset$, {ϵ}, {a_1},..., {a_1} ($a_i \in \Sigma$) et en utilisant les opérations de concaténation '.', l'union '$\cup$' et l'opération de complémentation '$\bar{}$ '.*

Exemple :

1. Σ^* est sans étoile : $\Sigma^* = \bar{\emptyset}$
2. $\Sigma^* a$ est sans étoile

Remarque : Les langages de mots sans étoile sont rationnels.

Théorème 3. *(Mc Naughton-Papert) Un langage est FO-définissable si et seulement si il est sans étoile.*

tel-00580926, version 1 - 29 Mar 2011

tel-00580926, version 1 - 29 Mar 2011

«L'ennemi de la vérité, ce n'est pas le mensonge, ce sont les convictions.»

Friedrich Wilhelm Nietzsche

tel-00580926, version 1 - 29 Mar 2011

tel-00580926, version 1 - 29 Mar 2011

Chapitre 2

Langages de sélection de nœuds

La modélisation arborescente des documents XML oblige tous les langages de requêtes sur ce type de documents à disposer d'outils ou de primitives pour parcourir des chemins dans ces arbres et sélectionner des nœuds. XPath est le langage le plus utilisé pour sélectionner des nœuds. Nous donnons en section 1 un bref aperçu des principaux langages de la littérature issus de XPath. Dans cette thèse cependant nous faisons le choix d'utiliser un nouveau langage formel de requêtes, nommées Requêtes Arbres régulières ($\mathcal{RAR}$s), permettant la sélection de nœuds dans un document XML. Ce langage est présenté en section 2 : il s'appuie sur des motifs arborescents et utilise des expressions régulières pour l'expression de chemins. On positionnera les requêtes $\mathcal{RAR}$s par rapport aux langages issus de XPath dans la section 3.

1 Les langages issus de XPath

1.1 XPath complet : Full XPath

XPath a été introduit par le W3C [XPa99] comme un langage de requêtes standard permettant de sélectionner un ensemble d'éléments d'un document XML répondant à certaines contraintes structurelles ou textuelles. Il est possible notamment de caractériser ces éléments par leur position (absolue ou relative), leur type ou leur contenu. Ce langage est utilisé dans de nombreux outils de la galaxie XML et notamment XSLT et XQuery.
XPath associe un arbre de nœuds à tout document XML. Les nœuds de cet arbre peuvent être de plusieurs types prédéfinis et sont ordonnés selon l'ordre de lecture des constituants du document XML. La syntaxe de base du langage XPath est fondée sur l'utilisation d'expressions. Une expression XPath s'évalue en fonction d'un nœud contexte. Le résultat de l'évaluation de cette expression renverra alors soit un ensemble de nœuds, soit une valeur numérique, booléenne ou alphanumérique.
Une expression XPath utilise des chemins de localisation (location path) qui sont

une succession d'étapes (step), une étape s'écrivant sous la forme **axe::type[filtre]**. Un chemin de localisation peut être absolu ou relatif. S'il est absolu, le nœud contexte est la racine du document XML : le chemin commence alors par le signe "/" qui représente la racine. S'il est relatif, il est évalué par rapport au nœud contextuel courant.

Axe. Un axe permet de définir le sens de la relation entre le nœud courant et l'ensemble de nœuds à localiser. La navigation dans l'arborescence du document XML se fait selon les différents axes suivants : `self`, `child`, `parent`, `descendant`, `descendant_or_self`, `ancestor`, `ancestor_or_self`, `following_sibling`, `following`, `preceding_sibling`, `preceding`, `attribute` (qui représente les attributs du nœud contextuel), et `namespace`.

Type. Il existe deux manières pour sélectionner les nœuds : par leur nom ou par leur type. La sélection sur le nom s'effectue soit simplement en indiquant ce nom soit en utilisant le caractère joker '*' qui les sélectionne tous. La sélection sur le type de nœud permet de sélectionner tous les nœuds (indépendamment de leur nom) qui correspondent à un certain type. Il existe quatre types de nœud possibles :

- text() : sélectionne seulement les nœuds de type text.
- comment() : sélectionne les nœuds de type comment.
- processing-instruction() : sélectionne les nœuds de type Processing Instruction.
- node() : sélectionne tous les nœuds.

Filtre. Les filtres (prédicats) sont des expressions booléennes construites à partir d'expressions de chemin et de fonctions prédéfinies. Le résultat de l'évaluation d'un prédicat est nécessairement booléen. On peut les classifier en plusieurs types :

- Filtre logique qui utilise des fonctions et des opérateurs de comparaison.
- Filtre d'existence de certains nœuds.
- Filtre de position.

Dans presque tous les travaux concernant XPath, les auteurs se restreignent à des fragments de XPath pour simplifier le modèle complet de XPath 1.0. Parmi ces fragments on trouve $CoreXPath$ et le fragment simple $XP^{\{[],*,//\}}$, ce dernier étant équivalent à la notion de "tree pattern" introduite en [MS04]. Beaucoup d'extensions de XPath 1.0 ont été proposées notamment le standard W3C XPath 2.0 qui introduit l'utilisation de variables [XPa07]. Dans la suite, nous positionnons les requêtes $\mathcal{RAR}$s par rapport à deux extensions particulières du fragment $CoreXPath$: $CXPath$ et RegularXPath.

tel-00580926, version 1 - 29 Mar 2011

tel-00580926, version 1 - 29 Mar 2011

1.2 XPath navigationnel : $CoreXPath$

Le fragment $CoreXPath$ [GKP02] utilise les principales caractéristiques de XPath, mais exclut toutes les fonctions sauf les deux connecteurs logiques 'and' et 'or'. Il exclut aussi l'utilisation des axes `attribute` et `namespaces` et la sélection par type de nœuds.
Formellement un chemin de $CoreXPath$ s'écrit sous la forme : $P_a{=}/P$ où P est un chemin relatif de $CoreXPath$ défini récursivement par la grammaire :

$$\begin{aligned} P_a &\equiv /P \\ P &\equiv axis :: \lambda \ \| \ P/axis :: \lambda \ \| \ P[F] \ \| \ P|P \\ F &\equiv F_{un} \| \ F \ and \ F \| \ not(F) \\ F_{un} &\equiv axis :: \lambda \| \ axis :: \lambda[F] \| \ /F_{un} \end{aligned}$$

où λ appartient à $\Sigma \cup \{*\}$ et $axis$ est l'un des axes navigationnels de $XPath$ i.e $axis \in$ Axes avec Axes $=\{$ `self, child, parent, descendant, descendant_or_self, ancestor, ancestor_or_self, following_sibling, following, preceding_sibling, preceding` $\}$.
D'une manière générale, on appelle expression de $CoreXPath$ toute expression engendrée par les non terminaux P_a, P ou F. Toute expression engendrée par le non terminal P_a (respectivement P, F, F_{un}) est appelée un *chemin absolu* (respectivement un *chemin relatif*, un *filtre*, un *filtre unaire*).

On note par $CoreXPath^+$ le fragment positif de $CoreXPath$ qui exclut l'utilisation de l'opérateur not. On le définit formellement par :

$$\begin{aligned} P_a &\equiv /P \\ P &\equiv axis :: \lambda \ \| \ P/axis :: \lambda \ \| \ P[F] \ \| \ P|P \\ F &\equiv axis :: \lambda \| \ axis :: \lambda[F] \| \ /F \end{aligned}$$

La sémantique de $CoreXPath$

L'évaluation, à partir d'un nœud contexte x notée $[\![e]\!]x$, d'une expression $CoreXPath$ e sur un document XML $\mathcal{D}$ retourne un ensemble de nœuds de $\mathcal{N}(\mathcal{D})$ atteignables à partir de x. Nous donnons ci-dessous une définition inductive de $[\![e]\!]x$.

Pour chaque axe de Axes, on définit l'ensemble de nœuds $[\![axis]\!]x$ comme suit :

$$
\begin{array}{ll}
[\![self]\!]x & = \{x\} \\
[\![child]\!]x & = fils(x) \\
[\![parent]\!]x & = parent(x) \\
[\![descendant]\!]x & = fils^{+}(x) \\
[\![ancestor]\!]x & = parent^{+}(x) \\
[\![descendant\_or\_self]\!]x & = [\![descendant]\!]x \cup [\![self]\!]x \\
[\![ancestor\_or\_self]\!]x & = [\![ancestor]\!]x \cup [\![self]\!]x \\
[\![preceding]\!]x & = \{y \ / y \ll x\} \diagdown [\![ancestor]\!]x \\
[\![following]\!]x & = \{y \ / x \ll y\} \diagdown [\![descendant]\!]x \\
[\![following\_sibling]\!]x & = \{y \ / y \in child(parent(x)) \wedge x \ll y\} \\
[\![preceding\_sibling]\!]x & = \{y \ / y \in child(parent(x)) \wedge y \ll x\}
\end{array}
$$

où $fils(x)$ est l'ensemble des nœuds fils du nœud x, $parent(x)$ est le nœud parent du nœud x.
$\ll$ définit l'ordre naturel des nœuds dans le document XML (ordre d'apparition des balises ouvrantes).
$fils^{+}(x)$=$\{$y tel que $\exists\ x_1, x_2, ..., x_n$ avec $x_1 = x$, $x_n = y$ et $x_{i+1} \in fils(x_i)$ $\}$
$parent^{+}(x)$=$\{$y tel que $\exists\ x_1, x_2, ..., x_n$ avec $x_1 = x$, $x_n = y$ et $x_{i+1} = parent(x_i)$ $\}$

D'autre part, on utilise les notations suivantes : $root(\mathcal{D})$ représente la racine du document XML et $name(x)$ représente l'étiquette du nœud x.

Définition 9. *Soient e une expression CoreXPath, P un chemin relatif de CoreXPath, F un filtre de CoreXPath et x un nœud de* $\mathcal{N}(\mathcal{D})$.

$$
\begin{array}{ll}
[\![/e]\!]x & = [\![e]\!]root(\mathcal{D}) \\
[\![axis :: \lambda]\!]x & = \{x_1 \in \mathcal{N}(\mathcal{D}) / x_1 \in [\![axis]\!]x \wedge name(x_1) = \lambda\} \\
[\![p/axis :: \lambda]\!]x & = \{x_2 \in \mathcal{N}(\mathcal{D}) / \exists x_1, x_1 \in [\![p]\!]x \wedge x_2 \in [\![axis :: \lambda]\!]x_1\} \\
[\![e[F]]\!]x & = \{x_1 \in \mathcal{N}(\mathcal{D}) / x_1 \in [\![e]\!]x \wedge \ x_1 \vDash_{\mathcal{D}} F\} \\
[\![p_1|p_2]\!]x & = [\![p_1]\!]x \cup [\![p_2]\!]x
\end{array}
$$

où la notation $x \vDash_{\mathcal{D}} F$ est définie par :

$$
\begin{array}{lcl}
x \vDash_{\mathcal{D}} F & \Leftrightarrow & [\![F]\!]x \neq \varnothing \text{ si } F \text{ est unitaire} \\
x \vDash_{\mathcal{D}} F & \Leftrightarrow & [\![F_1]\!]x \neq \varnothing \text{ et } [\![F_2]\!]x \neq \varnothing \text{ si } F = F_1 \ and \ F_2 \\
x \vDash_{\mathcal{D}} F & \Leftrightarrow & [\![F_1]\!]x = \varnothing \text{ si } F = not(F_1)
\end{array}
$$

tel-00580926, version 1 - 29 Mar 2011

La puissance d'expressivité de $CoreXPath$ en terme de logique ainsi que ses principales propriétés sont données par le théorème suivant :

Théorème 4. *(Marx et Rijke [Md05], Gottlob et al [GKP02])*

- *Pour toute formule $\phi(x,y)$ de $FO^2{}_{tree}$, il existe un chemin e de $CoreXPath$ tel que pour tout document $\mathcal{D}$, $[\![e]\!]x = \{y \mid \mathcal{D} \models \phi(x,y)\}$ et inversement.*
- *Pour toute formule $\phi(x)$ de $FO^2{}_{tree}$, il existe un filtre F de $CoreXPath$ tel que pour tout document $\mathcal{D}$, $x \models_{\mathcal{D}} F \Leftrightarrow \mathcal{D} \models \phi(x)$ et inversement.*
- *$CoreXPath$ est fermé par intersection.*
- *$CoreXPath$ n'est pas fermé par complément.*
- *$CoreXPath$ est évaluable en temps polynomial.*

1.3 XPath conditionnel : $CXPath$

Bien que très utilisé dans la pratique, XPath 1.0 ne permet cependant pas d'exprimer des conditions plus complexes d'accessibilité des nœuds sélectionnés, s'appuyant par exemple sur l'existence de chemins dit itératifs. Pour pallier à cette limite du pouvoir d'expression de XPath 1.0 *M. Marx* a proposé une extension appelée XPath conditionnel ($CXPath$) plus expressive, en autorisant la fermeture transitive sur des chemins simple ($axis :: \lambda[F]$). Ainsi dans [Mar04a], M. Marx a montré que $CXPath$ est complet par rapport à la logique du premier ordre sur des arbres ordonnés, et plus généralement que toute extension de $CoreXPath$ fermée par complémentation est FO-complet pour les ensembles de chemin [Mar05b].

$CXPath$: Définition [Mar05b]

Le langage $CXPath$ est défini par la grammaire suivante :

$$\begin{aligned} prim\_axis &::= child \parallel parent \parallel right \parallel left. \\ P &::= prim\_axis \parallel P/P \parallel P \cup P \parallel (prim\_axis/?F)^* \parallel ?F \\ F &::= \lambda \parallel \top \parallel <P> \parallel not\ F \parallel F\ and\ F \parallel F\ or\ F \end{aligned}$$

P (respectivement F) est appelé expression de chemin (respectivement expression de nœud). On note par $CXPath^+$ le fragment positif de $CXPath$ qui exclut l'utilisation de l'opérateur not.

tel-00580926, version 1 - 29 Mar 2011

$CXPath$: Sémantique

La sémantique de $CXPath$ est donnée par :

$$
\begin{array}{ll}
[\![\lambda]\!] & =\{x \in \mathcal{N}(\mathcal{D})/\ name(x) = \lambda\} \\
[\![< P >]\!] & =\{x \in \mathcal{N}(\mathcal{D})/\ \exists\ x', (x, x') \in\ [\![P]\!]\} \\
[\![\top]\!] & =\{x\} \\
[\![not\ F]\!] & =\{x \in \mathcal{N}(\mathcal{D})/\ x \notin\ [\![F]\!]\} \\
[\![F_1\ and\ F_2]\!] & =[\![F_1]\!] \cap [\![F_2]\!] \\
[\![F_1\ or\ F_2]\!] & =[\![F_1]\!] \cup [\![F_2]\!] \\
[\![child]\!] & =\{(x, x') \in \mathcal{N}(\mathcal{D})^2/\ x' = fils(x)\} \\
[\![parent]\!] & =\{(x, x') \in \mathcal{N}(\mathcal{D})^2/\ x' = parent(x)\} \\
[\![right]\!] & =\{(x, x') \in \mathcal{N}(\mathcal{D})^2/\ x = sk\ et\ x' = s(k+1)\} \\
[\![left]\!] & =\{(x, x') \in \mathcal{N}(\mathcal{D})^2/\ x = sk\ avec\ k \geq 1\ et\ x' = s(k-1)\} \\
[\![?F]\!] & =\{(x, x) \in \mathcal{N}(\mathcal{D})^2/\ x \in\ [\![F]\!]\} \\
[\![P_1 \cup P_2]\!] & =[\![P_1]\!] \cup [\![P_2]\!] \\
[\![P_1/P_2]\!] & =[\![P_1]\!] \circ [\![P_2]\!] = \{(x, x') \in \mathcal{N}(\mathcal{D})^2/\ \exists x'',\ (x, x'') \in [\![P_1]\!]\ et\ (x'', x) \in [\![P_2]\!]\ \} \\
[\![P^+]\!] & =[\![P]\!] \cup (\cup_{i \geq 2} [\![P]\!]^i) \text{ où } [\![P]\!]^i = [\![P]\!] \circ [\![P]\!]^{i-1}\ \forall i \geq 2 \\
[\![P^*]\!] & =\{(x, x)\ /\ x \in \mathcal{N}(\mathcal{D})\} \cup [\![P^+]\!]
\end{array}
$$

La puissance d'expressivité de $CXPath$ en terme de logique ainsi que ses principales propriétés sont données par le théorème suivant :

Théorème 5. *(Marx [Mar04a], [Mar05a])*

- *Pour toute formule $\phi(x, y)$ de FO_{tree}, il existe une expression de chemin e de CXPath tel que pour tout document $\mathcal{D}$, $[\![e]\!]$ = {(x,y) / $\mathcal{D} \models \phi(x, y)$} et inversement.*
- *Pour toute formule $\phi(x)$ de $FO_{tree}[R_{\Downarrow}, R_{\Rightarrow}]$, il existe une expression de nœud F de CXPath tel que pour tout document $\mathcal{D}$, $[\![F]\!]$ = {x / $\mathcal{D} \models \phi(x)$} et inversement.*
- *$CXPath$ est fermé par intersection.*
- *$CXPath$ est fermé par complément.*
- *$CXPath$ est évaluable en temps polynomial.*

Exemple : Soit la requête qui sélectionne les nœuds origines d'un chemin dont tous les nœuds sont étiquetés par B sauf le dernier qui est étiqueté par A. Cette requête ne peut être exprimée dans XPath, par contre, on peut l'exprimer dans $CXPath$ par l'expression de nœud suivante : $< (child?B)^*/child?A >$, la formule de FO_{tree} équivalente à cette requête est : $\exists y(xR_{\Downarrow}y \wedge O_A(y) \wedge \forall z((xR_{\Downarrow}z \wedge zR_{\Downarrow}y) \rightarrow O_B(z)))$.

tel-00580926, version 1 - 29 Mar 2011

1.4 XPath régulier : RegularXPath

Bien que permettant déjà la complétude par rapport à FO_{tree}, l'utilisation de la fermeture transitive dans $CXPath$ est limitée aux étapes simples. Dans [Mar04b], M. Marx propose une extension encore plus expressive de CoreXPath, appelée XPath régulier (RegularXPath) et levant cette limitation. Ainsi RegularXPath étend $CXPath$ par l'utilisation de la fermeture transitive sur tout type de chemin de CoreXPath. Formellement, le langage RegularXPath est défini par la grammaire :

$$\begin{aligned} prim\_axis &::= child \parallel parent \parallel right \parallel left. \\ P &::= prim\_axis \parallel P/P \parallel P \cup P \parallel P^* \parallel ?F \\ F &::= \lambda \parallel <P> \parallel not\ F \parallel F\ and\ F \parallel F\ or\ F \end{aligned}$$

La sémantique de RegularXPath est la même que celle donnée pour CXPath.

Exemple : Soit la requête qui sélectionne les nœuds qui sont à une distance paire de la racine, l'expression de RegularXPath équivalente est : $/(child/child)^*$.

La puissance expressive de RegularXPath est strictement comprise entre celle des logiques FO_{tree} et MSO [tS08]. Dans [ten06] Balder étend $CoreXPath$ avec la fermeture transitive et l'égalité de chemins et montre que le langage obtenu, appelé $RegularXPath^{\approx}$ est complet par rapport à FO^*. Une autre proposition d'extension est proposé par Balder & al dans [tS08], où les auteurs ajoutent en plus de la fermeture transitive un opérateur 'W' de relativisation de sous-arbre permettant d'évaluer une expression sur une partie restreinte (sous-arbre) du document global. Le langage obtenu, appelé $RegularXPath^W$ est complet par rapport à $FO + TC1$.

1.5 Fragment simple de $CoreXPath$ et Tree patterns

Le fragment simple, noté $XP^{\{[],*,//\}}$, de $CoreXPath$ est composé des expressions de $CoreXPath$ ne contenant que les axes child et descendant, autorisant l'utilisation de filtres, et du symbole '$*$'. Formellement une requête de $XP^{\{[],*,//\}}$ est définie par :

$P \equiv axis :: \lambda \parallel P[F] \parallel P/axis :: \lambda$

F $\equiv$ axis:: λ $\parallel$ axis:: λ[F] $\parallel$ /F

Avec $\lambda \in \Sigma \cup \{*\}$ et axis $\in$ { `child, descendant` }

Toute expression de $XP^{\{[],*,//\}}$ peut être exprimée par un tree pattern unaire introduit en [MS04]. Un tree pattern k-aire est un arbre P=(N, $A \subseteq N \times N$, l) non ordonné étiqueté sur l'alphabet $\Sigma \cup \{*\}$ dont les arêtes sont représentées soit par une seule ligne pour l'axe child, soit par une double ligne pour l'axe descendant, et un k-tuple de nœuds spéciaux appelé le k-tuple résultat. L'entier k est l'arité du pattern. Si k=0 (k=1), on parle de pattern booléen (respectivement pattern unaire).

tel-00580926, version 1 - 29 Mar 2011

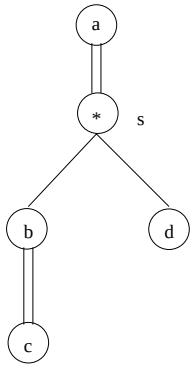


FIGURE 2.1 – Un tree pattern

Pour évaluer un tree pattern p dans un document $\mathcal{D} = (d, \lambda)$, on utilise la notion de plongement. Un plongement m de p dans $\mathcal{D}$ est une fonction de N dans $\mathcal{N}(\mathcal{D})$ vérifiant :

- m(root(p))=root($\mathcal{D}$).
- $\forall$ x $\in$ N, l(x)=$*$ ou l(x)=λ(m(x)).
- $\forall$ x,y $\in$ N, si (x,y) est une arête fils dans p alors (m(x),m(y)) est une arête de $\mathcal{D}$.
- $\forall$ x,y $\in$ N, si (x,y) est une arête descendant dans p alors m(y) est un descendant de m(x) dans $\mathcal{D}$.

Remarque :
- Le plongement m n'est pas obligatoirement injectif, c'est à dire que deux nœuds de p peuvent avoir la même image.
- Le plongement m ne préserve pas nécessairement l'ordre.

Dans [MS04] les auteurs montrent que toute requête q de $XP^{\{[],*,//\}}$ peut être représentée par un tree pattern unaire. et vice-versa.

Exemple :
Le tree pattern de la figure 2.1 est équivalent à l'expression XPath suivante :
$a//*[b//c][d]$.

1.6 Récapitulatif sur $CoreXPath$ et ses extensions

CoreXPath n'est pas FO_{tree}-complet. Plusieurs extensions de $CoreXPath$ ont été proposées pour obtenir un langage FO_{tree}-complet, comme $CXPath$ qui est FO-complet et évaluable en temps polynomial. Une autre extension de $CoreXPath$ qui est FO_{tree}-complet est CoreXPath2.0, Ce dernier n'est pas évaluable en temps polynomial. Dans [FNTT07] les auteurs proposent un fragment de CoreXPath2.0 (PPL) évaluable en temps polynomial et aussi FO_{tree}-complet, ce fragment inter-

tel-00580926, version 1 - 29 Mar 2011

dit : (1)l'utilisation des quantificateurs, (2)l'utilisation des variables partagées dans les compositions de chemin et (3)l'utilisation des variables dans les complémentations. La figure 2.2 donne une vue comparative de ces langages.

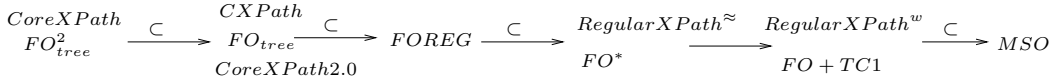


FIGURE 2.2 – Schéma de comparaison

2 Requête arbre régulière ($\mathcal{RAR}$)

Dans cette section, nous introduisons un nouveau langage de sélection de nœuds basé sur les expressions régulières qu'on appellera langage des Requêtes Arbres Régulières ($\mathcal{RARs}$). Ce choix est motivé par le caractère formel de ce langage et donc son indépendance vis-à-vis des standards existant comme XPath. Bien qu'incomparable avec le langage XPath, nous montrons en section 3, que les $\mathcal{RARs}$ permettent d'exprimer toutes les requêtes exprimables par le fragment navigationnel positif de CoreXPath.

2.1 Requête arbre régulière ($\mathcal{RAR}$) : définition

Introduisons dans un premier temps la notion de requête arbre régulière de manière intuitive en considérant la requête binaire $\mathcal{V}$: "Donner les couples d'éléments (Titre, Auteur) de tous les articles" et son évaluation sur le document semi-structuré $\mathcal{D}$ de la figure 2.3. Cette requête peut être représentée par la requête arbre régulière $\mathscr{R}$ de la figure 2.3 qui indique de manière schématique les conditions d'extraction d'un couple de nœuds (w, w') de $\mathcal{D}$: (w, w') est extrait de $\mathcal{D}$ si et seulement s'il est possible de réaliser un plongement p de l'arbre $\mathscr{R}$ dans le document $\mathcal{D}$, de telle sorte que (voir figure 2.3) (a) les nœuds de sélection grisés 00 et 01 de $\mathscr{R}$ sont précisément associés par p aux nœuds w et w' c'est-à-dire $p(00) = w$, $p(01) = w'$ et , (b) pour chaque arc (w_1, w_2) de la requête $\mathscr{R}$ il existe un chemin dans le document source de $p(w_1)$ à $p(w_2)$ dont la suite des étiquettes satisfait les contraintes exprimées par l'expression régulière étiquetant l'arc (w_1, w_2) dans $\mathscr{R}$. Formellement, une requête arbre régulière ($\mathcal{RAR}$) sur l'alphabet Σ est donc un arbre dont les arcs sont étiquetés par des expressions régulières sur Σ. Nous en donnons ci-après la définition précise.

tel-00580926, version 1 - 29 Mar 2011

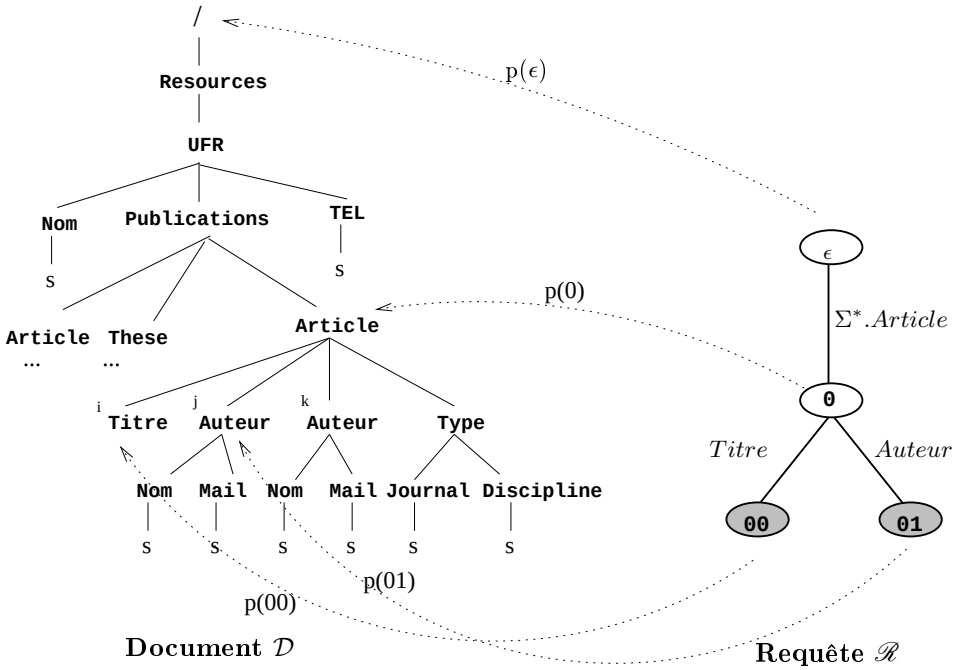


FIGURE 2.3 – Un plongement de la requête $\mathscr{R}$ dans le document $\mathcal{D}$

Définition 10. *Requête arbre régulière n-aire (RAR) : Soit Σ un alphabet fini d'étiquettes. Une requête arbre régulière n-aire $\mathscr{R}$ sur Σ est définie par $\mathscr{R} = (\mathscr{T}, \vec{s})$ où $\mathscr{T} = (\Sigma, N, M, \mathcal{E})$ est appelé arbre template de $\mathscr{R}$ ($N \subseteq \mathbb{N}^*$ est le domaine de l'arbre, $M \subseteq N \times N$ est l'ensemble de ses arcs, $\mathcal{E} : M \longrightarrow REG(\Sigma)$ est une application qui associe à chaque arc (w, w') de M une expression notée $\mathcal{E}_{(w,w')}$ qui est, soit vide, soit régulière propre, i.e $L(\mathcal{E}_{(w,w')}) \subset \Sigma^+$), et $\vec{s}$=$(w_1, ..., w_n)$ est un n-uplet de nœuds spécifiques de N, représentant les n-uplets de nœuds (non nécessairement distincts) à sélectionner. Lorsque $\vec{s}$ est réduit à un seul nœud (n=1), on dit que $\mathscr{R}$ est monadique.*

La taille de $\mathscr{R}$ notée $|\mathscr{R}|$ est définie par : $|\mathscr{R}| = |N| + \Sigma_{e \in M} |\mathcal{A}_e|$ où $\mathcal{A}_e$ est un automate fini de mots associé à l'expression régulière $\mathcal{E}_e$ et $|\mathcal{A}_e|$ désigne la taille de $\mathcal{A}_e$.
Pour détailler le contenu d'un template $\mathscr{T}$, on adopte l'écriture suivante :

$$\begin{aligned} \mathscr{T} \quad = \quad & (\epsilon = [\mathcal{E}_{(\epsilon,0)} \rightarrow 0..., \mathcal{E}_{(\epsilon,k_\epsilon)} \rightarrow k_\epsilon]; \\ & 0 = [\mathcal{E}_{(0,00)} \rightarrow 00, ..., \mathcal{E}_{(0,0k_0)} \rightarrow 0k_0]; \\ & ... \\ & w = [\mathcal{E}_{(w,w0)} \rightarrow w0, ..., \mathcal{E}_{(w,wk_w)} \rightarrow wk_w]; ...) \end{aligned}$$

Cette écriture donne pour chaque nœud w de N, la liste des expressions régulières $\mathcal{E}_{(w,wj)}$, $0 \leqslant j \leqslant k_w$ associées à l'ensemble, noté $Out(w)$, des arêtes sortantes du

tel-00580926, version 1 - 29 Mar 2011

nœud w, (k_w+1) est l'arité du nœud w i.e $(k_w + 1) = |Out(w)|$. Nous ne prenons en considération que les requêtes arbres régulières $\mathscr{R}$ pour lesquelles $Out(\epsilon)$ est réduit à un seul nœud i.e $k_\epsilon = 0$.
Nous notons $\mathscr{R}_\emptyset$ la requête $\mathcal{RAR}$ associée à un template $\mathscr{T}$ vide. Par convention, cette requête sera évaluée à l'ensemble vide sur tout document $\mathcal{D}$.

2.2 Requête arbre régulière ($\mathcal{RAR}$) : Plongement, Évaluation

L'évaluation d'une requête $\mathcal{RAR}$ sur un document semi-structuré utilise la notion de plongement.

Définition 11. *Un plongement d'une requête $\mathcal{RAR}$ $\mathscr{R} = (\mathscr{T}, \vec{s})$ avec $\mathscr{T} = (\Sigma, N, M, \mathcal{E})$ dans un document semi-structuré $\mathcal{D}$, est une fonction p injective totale de N dans $\mathcal{N}(\mathcal{D})$ vérifiant :*

- *L'image du nœud ϵ racine de $\mathscr{R}$ est le nœud racine de $\mathcal{D}$ (étiqueté par '/').*
- *$\forall w, w' \in N$, si $w < w'$ alors $p(w) < p(w')$.*
- *$\forall e = (w, w') \in M$, il existe dans $\mathcal{D}$ un chemin P_e allant de $p(w)$ à $p(w')$ tel que :*
  (a) la suite notée $\lambda(P_e) \setminus \{\lambda(p(w))\}$, des étiquettes des nœuds de P_e, excluant celle de $p(w)$ et incluant celle de $p(w')$, est un mot du langage régulier $L(\mathcal{E}_{(w,w')})$ et,
  (b) si, dans $\mathscr{T}$, $e_1 = (w, wi)$ et $e_2 = (w, wj)$ sont deux arêtes distinctes de $Out(w)$ alors les chemins P_{e_1} et P_{e_2} n'ont pas de préfixe commun.

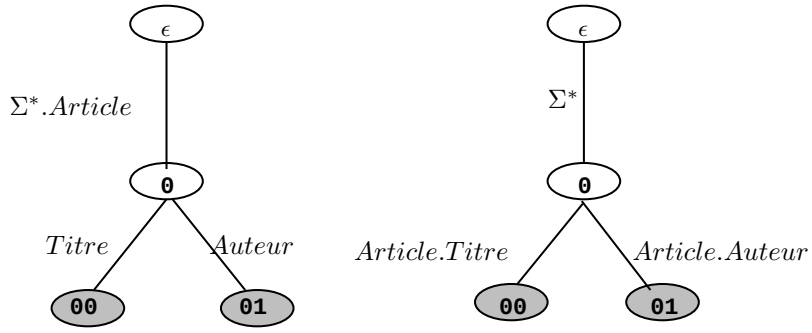
Un exemple de plongement est donnée en Figure 2.3.
Notons que, la condition (b) impose à tout plongement p de $\mathscr{R}$ dans $\mathcal{D}$, que les chemins dans $\mathcal{D}$ associés à deux arêtes sortantes d'un nœud w de $\mathscr{R}$, passent par deux fils différents de $p(w)$.
Pour illustrer la condition précédente (b), considérons les requêtes arbres $\mathscr{R}$ et $\mathscr{R}'$ de la figure 2.4. Leurs plongements, dans un document semi-structuré, sont tout à fait différents : les plongements de $\mathscr{R}$ extraient les paires (Titre, Auteur) qui sont fils d'un même nœud Article, alors que ceux de $\mathscr{R}'$ extraient les paires (Titre, Auteur) qui sont fils de deux nœuds Article distincts.

Trace d'une requête $\mathcal{RAR}$ selon un plongement On appelle trace de $\mathscr{R}$ dans le document $\mathcal{D}$ selon le plongement p, le plus petit sous-arbre de $\mathcal{D}$ contenant l'image $p(N)$. On la note : $\texttt{trace}_p(\mathscr{R}, \mathcal{D})$.

Évaluation d'une requête RAR : soit $\mathscr{R} = (\mathscr{T}, \vec{s})$ une requête RAR, $\mathcal{D}$ un document semi-structuré et $\mathcal{P}$ l'ensemble de tous les plongements de $\mathscr{R}$ dans $\mathcal{D}$.

- Le résultat, noté $\mathscr{R}_p(\mathcal{D})$, de l'évaluation de $\mathscr{R}$ sur $\mathcal{D}$ suivant le plongement p de $\mathcal{P}$, est le n-uplet d'arbres $\mathscr{R}_p(\mathcal{D}) = (\mathcal{D}(p(w_1)), \ldots, \mathcal{D}(p(w_n)))$ où $\vec{s} = (w_1, ..., w_n)$ est le n-uplet de nœuds sélectionnés par $\mathscr{R}$.

tel-00580926, version 1 - 29 Mar 2011



(a) Requête arbre $\mathscr{R}$ **(b) Reqête arbre $\mathscr{R}'$**

FIGURE 2.4 – Requêtes arbres régulières ($\mathcal{RAR}$)

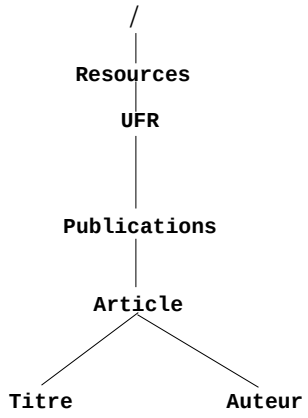


FIGURE 2.5 – Trace de la requête $\mathcal{RAR}$ $\mathscr{R}$ selon p

– Le résultat, noté $Eval_{\mathcal{D}}(\mathscr{R})$, de l'évaluation de $\mathscr{R}$ sur $\mathcal{D}$ est $Eval_{\mathcal{D}}(\mathscr{R}) = \bigcup_{p \in \mathcal{P}} \mathscr{R}_p(\mathcal{D})$.

Exemple La trace de la requête $\mathscr{R}$ selon le plongement donné en Figure 2.3 est donnée en Figure 2.5, son évaluation sur le document $\mathcal{D}$ de la figure 2.3 est l'ensemble composé de deux couples de sous-arbres : $\{(\mathcal{D}(i), \mathcal{D}(j)), (\mathcal{D}(i), \mathcal{D}(k))\}$.

2.3 Evaluation d'une requête $\mathcal{RAR}$ à partir d'un nœud

La notion de template utilisée dans la définition d'une $\mathcal{RAR}$ permet d'exprimer des conditions qui doivent être satisfaites à partir de la racine du document pour identifier les nœuds sélectionnés par la requête. Cette même notion de template peut, d'une manière plus générale, exprimer des conditions qui doivent être satisfaites

tel-00580926, version 1 - 29 Mar 2011

localement à partir d'un nœud donné, dans le sous-arbre issu de ce nœud, pour extraire certains nœuds.

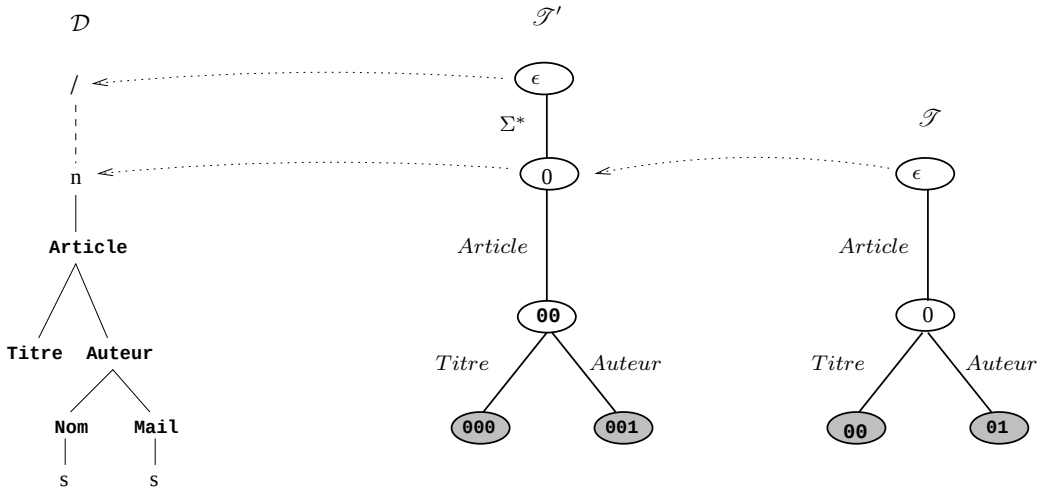


FIGURE 2.6 – Les templates $\mathscr{T}$ et $\mathscr{T}'$

Définition 12. *Soit $\mathscr{T}$ un template et $\mathcal{D}$ un document semi-structuré, on dit qu'un nœud n de $\mathcal{D}$ satisfait le template $\mathscr{T}$ (on écrit : $n \vDash_{\mathcal{D}} \mathscr{T}$) si et seulement si il existe un plongement p du template $\mathscr{T}'$ dans $\mathcal{D}$ vérifiant p(0)=n, avec :*

$$\begin{aligned}\mathscr{T}' \quad = \quad & (\epsilon = [\Sigma^* \to 0];\\ & 0 = [\mathcal{E}_{(\epsilon,0)} \to 00, ..., \mathcal{E}_{(\epsilon,k_\epsilon)} \to 0k_\epsilon];\\ & ...\\ & 0w = [\mathcal{E}_{(w,w0)} \to 0w0, ..., \mathcal{E}_{(w,wk_w)} \to 0wk_w]; ...)\end{aligned}$$

L'évaluation de la requête $\mathscr{R} = (\mathscr{T}, \vec{s})$ à partir du nœud n est définie par :
$Eval_{\mathcal{D}}(\mathscr{R}, n) = \bigcup_{p/p(0)=n} \mathscr{R}'_p(\mathcal{D})$ avec $\mathscr{R}' = (\mathscr{T}', \vec{s})$.
Remarquons que l'on a : $n \vDash_{\mathcal{D}} \mathscr{T} \Leftrightarrow Eval_{\mathcal{D}}(\mathscr{R}, n) \neq \emptyset$.

Exemple :
Soit $\mathscr{T}$ le template de la figure 2.6, le nœud étiqueté Publication du document $\mathcal{D}$ de la figure 2.3 satisfait $\mathscr{T}$.

2.4 Projection d'une requête $\mathcal{RAR}$

Nous définissons ici la notion de projection de requêtes qui nous sera utile dans les sections suivantes. Soit k, $1 \leq k \leq n$, un entier naturel, J=$(i_1, i_2, ..., i_k)$ un sous-uplet de (1,2,..,n) (on note J$\sqsubseteq$ (1,2,..,n)) et $\vec{u} = (u_1, u_2, ..., u_n)$ un n-uplet de nœuds. On définit la projection Π_J de $\vec{u}$ suivant J par : $\Pi_J(\vec{u})=(u_{1_i}, u_{i_2}, ..., u_{i_k})$.
La notion de projection s'étend naturellement aux requêtes n-aires.

tel-00580926, version 1 - 29 Mar 2011

Définition 13. *La projection suivant $J=(i_1, i_2, ..., i_k)$ de la requête n-aire $\mathscr{R}$ est la requête k-aire, notée $\Pi_J(\mathscr{R})$, définie par : $\forall\ \mathcal{D}$, $Eval_{\mathcal{D}}(\Pi_J(\mathscr{R}))=\{\Pi_J(\vec{u})\ /\ \vec{u} \in Eval_{\mathcal{D}}(\mathscr{R})\}$.*

Remarque La projection d'une union de requêtes n-aires coïncide clairement avec l'union des projections de chacune de ces requêtes.

Le lemme suivant est immédiat et énonce la stabilité par projection des requêtes RAR

Lemme 1. *Si $\mathscr{R}=(\mathscr{T},\vec{s})$ est une requête $\mathcal{RAR}$ alors $\forall\ J=(i_1, i_2, ..., i_k) \sqsubseteq (1, 2, ..., n)$, $\Pi_J(\mathscr{R})$ est la requête $\mathcal{RAR}$ $\Pi_J(\mathscr{R})=(\mathscr{T},\Pi_J(\vec{s}))$*

3 Les requêtes $\mathcal{RARs}$ versus $CoreXPath$

Dans cette section nous montrons (Théorème 6) que tout chemin de $CoreXPath^+$ est équivalent à une union finie de requêtes monadiques $\mathcal{RAR}$. $CoreXPath^+$ est le fragment navigationnel de XPath introduit en section 1.2 n'utilisant pas l'opérateur not. Pour démontrer le Théorème 6, nous introduisons dans la section 3.1 la notion d'extension de requête $\mathcal{RAR}$ par un filtre F de $CoreXPath^+$, et nous montrons quelques propriétés de cette notion. En section 3.2, nous établissons (Proposition 1) que la notion d'extension préserve les unions de requêtes $\mathcal{RARs}$. Enfin nous énonçons et prouvons le Théorème 6 en section 3.3.

3.1 Extension de requêtes par filtre

Définition 14. *Soit F un filtre de $CoreXPath^+$, $\mathscr{R} = (\mathscr{T}, \vec{s})$ une requête $\mathcal{RAR}$ n-aire et i un entier tel que $1 \leq i \leq n$.*
L'extension $(\mathscr{R}._{[i]}F)$ de la requête $\mathscr{R}$ par F est la requête dont l'évaluation sur tout document semi-structuré $\mathcal{D}$ est définie par :

- $Eval_{\mathcal{D}}(\mathscr{R}._{[i]}F)=\{\ (u_1, u_2, ..., u_n, u_{n+1}) \in \mathcal{N}(\mathcal{D})^{n+1}\ /\ (u_1, u_2, ..., u_n) \in Eval_{\mathcal{D}}(\mathscr{R})$ *et* $u_{n+1} \in [\![F]\!]u_i\ \}$.

L'opérateur d'extension d'une requête $\mathcal{RAR}$ par un filtre de $CoreXPath^+$ possède les propriétés élémentaires suivantes :

Propriétés :

1. L'extension par un filtre F de $CoreXPath^+$ d'une union de requêtes $\mathcal{RARs}$ coïncide avec l'union des extensions par F de chacune de ces requêtes.
2. $\mathscr{R}._{[i]}(/F)$ est équivalent à $\Pi_{(1,...,n,n+2)}(\mathscr{R}'._{[n+1]}F)$ avec $\mathscr{R}' = (\mathscr{T}, (\vec{s}, \epsilon))$

tel-00580926, version 1 - 29 Mar 2011

3. $\mathscr{R}_{.[i]}(axis :: \lambda[F])$ est une requête $\mathcal{RAR}$ (n+1)-aire équivalente à $\Pi_{(1,...,n+1)}((\mathscr{R}_{.[i]}axis :: \lambda)_{.[n+1]}F)$.

Démonstration. :
1. On a $Eval_{\mathcal{D}}(\cup_j \mathscr{R}_j) = \cup_j Eval_{\mathcal{D}}(\mathscr{R}_j)$ donc

$$\begin{aligned}
Eval_{\mathcal{D}}((\cup_j \mathscr{R}_j)_{.[i]}F) &= \{(u_1,u_2,...,u_n,u_{n+1}) \in \mathcal{N}(\mathcal{D})^{n+1}/ \\
&\quad (u_1,u_2,...,u_n) \in Eval_{\mathcal{D}}(\cup_j \mathscr{R}_j) \text{ et } u_{n+1} \in [\![F]\!]u_i\} \\
&= \{(u_1,u_2,...,u_n,u_{n+1}) \in \mathcal{N}(\mathcal{D})^{n+1}/ \\
&\quad (u_1,u_2,...,u_n) \in \cup_j Eval_{\mathcal{D}}(\mathscr{R}_j) \text{ et } u_{n+1} \in [\![F]\!]u_i\} \\
&= \{(u_1,u_2,...,u_n,u_{n+1}) \in \mathcal{N}(\mathcal{D})^{n+1}/\exists\ j \\
&\quad (u_1,u_2,...,u_n) \in Eval_{\mathcal{D}}(\mathscr{R}_j) \text{ et } u_{n+1} \in [\![F]\!]u_i\} \\
&= \cup_j \{(u_1,u_2,...,u_n,u_{n+1}) \in \mathcal{N}(\mathcal{D})^{n+1}/ \\
&\quad (u_1,u_2,...,u_n) \in Eval_{\mathcal{D}}(\mathscr{R}_j) \text{ et } u_{n+1} \in [\![F]\!]u_i\} \\
&= \cup_j Eval_{\mathcal{D}}((\mathscr{R}_j)_{.[i]}F) \\
&= Eval_{\mathcal{D}}(\cup_j(\mathscr{R}_{j\,.[i]}F).
\end{aligned}$$

Ainsi $(\cup_j\mathscr{R}_j)_{.[i]}F \equiv \cup_j(\mathscr{R}_{j\,.[i]}F)$.

2. On a $[\![/F]\!]u_i = [\![F]\!]root(\mathcal{D})$ donc

$$\begin{aligned}
Eval_{\mathcal{D}}(\mathscr{R}_{.[i]}/F) &= \{(u_1,u_2,...,u_n,u_{n+1}) \in \mathcal{N}(\mathcal{D})^{n+1}/ \\
&\quad (u_1,u_2,...,u_n) \in Eval_{\mathcal{D}}(\ \mathscr{R}) \text{ et } u_{n+1} \in [\![F]\!]root(\mathcal{D})\} \\
&= \{(u_1,u_2,...,u_n,u_{n+1}) \in \mathcal{N}(\mathcal{D})^{n+1}/ \\
&\quad (u_1,u_2,...,u_n,root(\mathcal{D})) \in Eval_{\mathcal{D}}(\ (\mathscr{T},(\vec{s},\epsilon))) \\
&\quad \text{et } u_{n+1} \in [\![F]\!]root(\mathcal{D})\} \\
&= \{(u_1,u_2,...,u_n,u_{n+1}) \in \mathcal{N}(\mathcal{D})^{n+1}/ \\
&\quad (u_1,u_2,...,u_n,root(\mathcal{D}),u_{n+1}) \in Eval_{\mathcal{D}}(\mathscr{R}'_{.[n+1]}F)\} \\
&= Eval_{\mathcal{D}}(\Pi_{(1,...,n,n+2)}(\mathscr{R}'_{.[n+1]}F)).
\end{aligned}$$

tel-00580926, version 1 - 29 Mar 2011

3. $Eval_{\mathcal{D}}(\mathscr{R}_{.[i]}(axis :: \lambda[F]))$=$\{$ $(u_1, ..., u_n, u_{n+1}) \in \mathcal{N}(\mathcal{D})^{n+1}$ / $(u_1, ..., u_n) \in Eval_{\mathcal{D}}(\mathscr{R})$ et $u_{n+1} \in [\![axis :: \lambda[F]]\!]u_i$ $\}$. Or $u_{n+1} \in [\![axis :: \lambda[F]]\!]u_i$ est vrai si $u_{n+1} \in [\![axis :: \lambda]\!]u_i$ et il existe $c \in \mathcal{N}(\mathcal{D})$ tel que $c \in [\![F]\!]u_{n+1}$ donc

$$\begin{aligned}
Eval_{\mathcal{D}}(\mathscr{R}_{.[i]}(axis :: \lambda[F])) &= \{(u_1, ..., u_n, u_{n+1}) \in \mathcal{N}(\mathcal{D})^{n+1} / (u_1, ..., u_n) \in Eval_{\mathcal{D}}(\mathscr{R}) \\
&\quad u_{n+1} \in [\![axis :: \lambda]\!]u_i \; et \; \exists \; c \; \in \mathcal{N}(\mathcal{D}) \; tel \; que \;\; c \in [\![F]\!]u_{n+1}\} \\
&= \{(u_1, ..., u_n, u_{n+1}) \in \mathcal{N}(\mathcal{D})^{n+1} / (u_1, ..., u_n, u_{n+1}) \in \\
&\quad Eval_{\mathcal{D}}(\mathscr{R}_{.[i]} axis :: \lambda) et \; \exists \; c \; tel \; que \;\; c \in [\![F]\!]u_{n+1}\} \\
&= \{(u_1, ..., u_n, u_{n+1}) \in \mathcal{N}(\mathcal{D})^{n+1} / \exists \; c \; tel \; que \\
&\quad (u_1, ..., u_n, u_{n+1}, c) \in Eval_{\mathcal{D}}((\mathscr{R}_{.[i]} axis :: \lambda)_{.[n+1]} F)\} \\
&= Eval_{\mathcal{D}}(\Pi_{(1,...,n+1)}((\mathscr{R}_{.[i]} axis :: \lambda)_{.[n+1]} F)).
\end{aligned}$$

3.2 Stabilité d'union de $\mathcal{RAR}$s par extension

Nous montrons dans cette section la stabilité des unions de $\mathcal{RAR}$s par extension, c'est à dire que l'extension d'une union de requêtes $\mathcal{RAR}$s est une union de requêtes $\mathcal{RAR}$s. D'après la première propriété de l'opérateur d'extension montrée en 3.1, il suffit de le montrer pour une seule requête $\mathcal{RAR}$. Pour cela nous montrons le résultat (Lemme 2), d'abord dans le cas où le filtre utilisé pour l'extension est un filtre unaire de la forme $axis :: \lambda$, puis dans le cas général (Proposition 1)

La preuve du lemme 2 utilise des opérateurs spécifiques permettant de réaliser diverses transformations sur un template $\mathscr{T} = (\Sigma, N, M, \mathcal{E})$ de requête $\mathcal{RAR}$ n-aire. Nous introduisons ci-dessous ces opérateurs :
Posons

$$\begin{aligned}
\mathscr{T} &= (\epsilon = [\mathcal{E}_{(\epsilon,0)} \to 0]; \\
&\quad 0 = [\mathcal{E}_{(0,0)} \to 00, ..., \mathcal{E}_{(0,0k_0)} \to 0k_0]; \\
&\quad ... \\
&\quad \omega = [\mathcal{E}_{(\omega,\omega 0)} \to \omega 0, ..., \mathcal{E}_{(\omega,\omega i)} \to \omega i, ..., \mathcal{E}_{(\omega,\omega k_\omega)} \to \omega k_\omega]; ...)
\end{aligned}$$

L'opérateur de renommage $Rename_{(S,m)}$ (Figure 2.7) :

Cet opérateur utilise deux paramètres : une expression régulière S sur Σ et une arête m de M. Appliqué à l'arbre template $\mathscr{T}$, il renvoie l'arbre template obtenu à partir de $\mathscr{T}$ en remplaçant l'expression régulière $\mathcal{E}_m$ par S.

tel-00580926, version 1 - 29 Mar 2011

Formellement, si $m = (\omega, \omega i)$ alors

$$\begin{aligned} Rename_{(S,m)}(\mathscr{T}) \;=\; & (\epsilon = [\mathcal{E}_{(\epsilon,0)} \to 0]; \\ & 0 = [\mathcal{E}_{(0,0)} \to 00, ..., \mathcal{E}_{(0,0k_0)} \to 0k_0]; \\ & ... \\ & \omega = [\mathcal{E}_{(\omega,\omega 0)} \to \omega 0, ..., S \to \omega i, ..., \mathcal{E}_{(\omega,\omega k_\omega)} \to \omega k_\omega]; ...) \end{aligned}$$

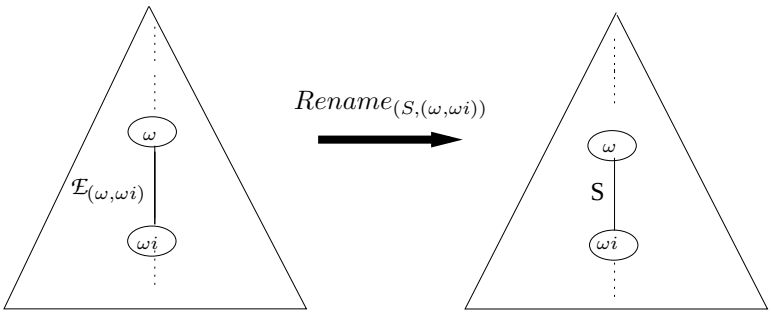


FIGURE 2.7 – Transformation par renommage

L'opérateur d'insertion horizontale $Insert^h_{S,\omega,pos}$(Figure 2.8) :

Cet opérateur utilise trois paramètres : une expression régulière S sur Σ, un nœud ω de N et un entier naturel *pos* représentant la position d'insertion. Appliqué à l'arbre template $\mathscr{T}$, il renvoie l'arbre template obtenu en insérant un nouveau nœud ωpos comme fils de ω à la position *pos*, en décalant les nœuds frères qui suivent cette position, et en choisissant S comme étiquette de la nouvelle arête (ω,ωpos). Formellement,

$$\begin{aligned} Insert^h_{S,\omega,pos}(\mathscr{T}) \;=\; & (\epsilon = [\mathcal{E}_{(\epsilon,0)} \to 0]; \\ & 0 = [\mathcal{E}_{(0,0)} \to 00, ..., \mathcal{E}_{(0,0k_0)} \to 0k_0]; \\ & ... \\ & \omega = [..., \mathcal{E}_{(\omega,\omega(pos-1))} \to \omega(pos-1), \\ & S \to \omega pos, \mathcal{E}_{(\omega,\omega pos)} \to \omega(pos+1), ...]; \\ & ...) \end{aligned}$$

L'opérateur d'insertion verticale $Insert^v_{S,\omega}$(Figure 2.9) :

Cet opérateur utilise deux paramètres : une expression régulière S sur Σ et un nœud ω de N. Appliqué à l'arbre template $\mathscr{T}$, il renvoie l'arbre template obtenu

tel-00580926, version 1 - 29 Mar 2011

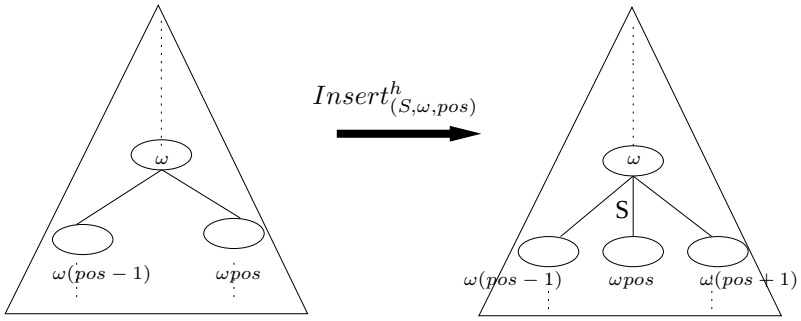


FIGURE 2.8 – Insertion horizontale

à partir de $\mathscr{T}$ en remplaçant tous les fils de ω par un seul nouveau nœud $\omega 0$, en décalant d'un niveau vers le bas l'ancienne descendance de ω vers la descendance du nouveau nœud $\omega 0$, et en choisissant S comme étiquette de la nouvelle arête (ω, $\omega 0$). Formellement,

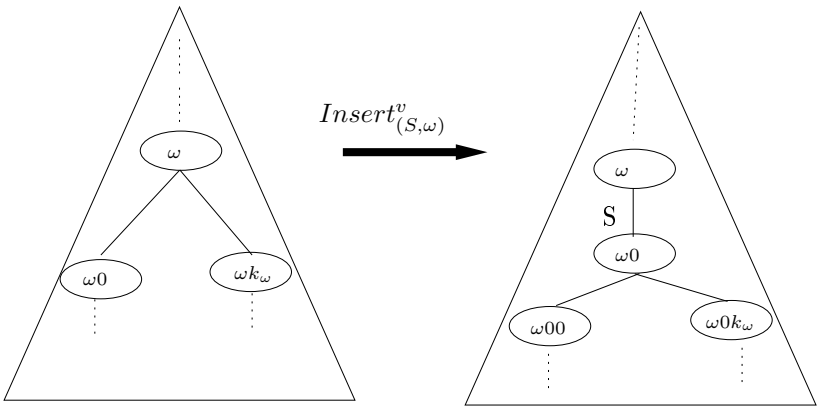


FIGURE 2.9 – Insertion verticale

$$\begin{aligned}\forall S \neq \emptyset,\ Insert^{v}_{S,\omega}(\mathscr{T}) \quad = \quad & (\epsilon = [\mathcal{E}_{(\epsilon,0)} \rightarrow 0];\\ & 0 = [\mathcal{E}_{(0,0)} \rightarrow 00, ..., \mathcal{E}_{(0,0k_0)} \rightarrow 0k_0];\\ & ...\\ & \omega = [S \rightarrow \omega 0]\\ & \omega 0 = [\mathcal{E}_{(\omega,0)} \rightarrow \omega 00, ..., \mathcal{E}_{(\omega,\omega k_\omega)} \rightarrow \omega 0 k_\omega];\\ & ...)\end{aligned}$$

Par convention, on définit $Insert^{v}_{\emptyset,\omega}(\mathscr{T}) = \mathscr{T}$

D'autre part, dans le cas de cet opérateur d'insertion verticale, si $\vec{s} = (s_1, s_2, ..., s_n)$ est un n-uplet de nœuds, on notera $Insert^{v}_{\omega}$ le n-uplet $(s'_1, s'_2, ..., s'_n)$ correspondant

tel-00580926, version 1 - 29 Mar 2011

à $\vec{s}$, après l'opération d'insertion, c'est à dire : $\forall\ k = 1, ..., n$ si s_k est un descendant de ω (i.e. $s_k = \omega u$, avec $u \in \mathbb{N}^*$) alors $s'_k = \omega 0 u$, et sinon $s'_k = s_k$.

Lemme 2. *Soit $\mathscr{R}$=($\mathscr{T}$,$\vec{s}$) est une requête $\mathcal{RAR}$ n-aire, $j \leqslant n$ un entier et $\lambda \in \Sigma \setminus \{'/'\}$ alors $\mathscr{R}_{.[j]}(axis :: \lambda)$ est équivalente à une union finie de requêtes $\mathcal{RAR}$s (n+1)-aire.*

Démonstration. Soit $\mathscr{R}$=($\mathscr{T}$,$\vec{s}$) une requête $\mathcal{RAR}$ n-aire avec $\vec{s} = (s_1, s_2, ..., s_n)$, et j un entier tel que $1 \leq j \leq n$. Appliquons maintenant l'extension de $\mathscr{R}$ par le filtre $axis :: \lambda$. Nous donnons ci-après l'analyse détaillée des différents cas possibles selon la valeur de axis. Cette analyse un peu longue peut être sautée lors d'une première lecture

Cas : *axis*=self

Si s_j=ϵ alors $\mathscr{R}_{.[j]}(self :: \lambda) \equiv \mathscr{R}_\emptyset$ car $\lambda \neq\ '/'$, sinon soit σ le père de s_j dans $\mathscr{T}$. On obtient $\mathscr{R}_{.[j]}(self :: \lambda)$ en filtrant l'expression régulière $\mathscr{E}_{(\sigma,s_j)}$ par $\Sigma^*\lambda$ pour imposer λ comme dernière lettre, on a alors :

$$\mathscr{R}_{.[j]}(self :: \lambda) \equiv (Rename_{(\mathscr{E}_{(\sigma,s_j)} \cap \Sigma^*\lambda),(\sigma,s_j)}(\mathscr{T}),(s_1, s_2, ..., s_n, s_j)).$$

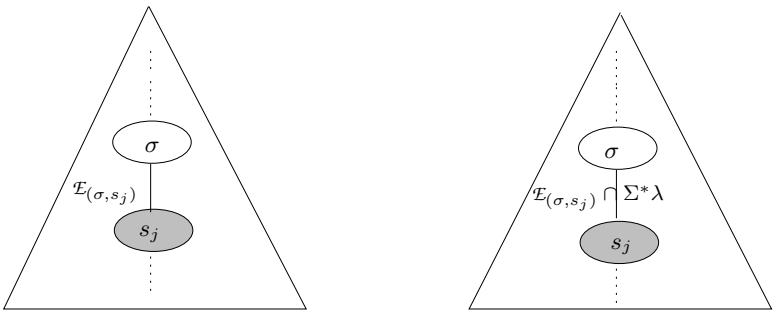


FIGURE 2.10 – Requête arbre $\mathscr{R}_{.[j]}(self :: \lambda)$

Cas *axis*=child

Si $s_j \neq \epsilon$, $\mathscr{R}_{.[j]}(child :: \lambda)$ est obtenu à partir de $\mathscr{T}$

- soit, en insérant pour tout i $\in$ [0,k_{s_j}+1] un nouveau ième fils de s_j et en étiquetant par λ la nouvelle arête $(s_j, s_j i)$. On notera par $\mathcal{U}_i$ le template obtenu : $\mathcal{U}_i = Insert^h_{\lambda,s_j,i}(\mathscr{T})$ (Figure 2.11).

tel-00580926, version 1 - 29 Mar 2011

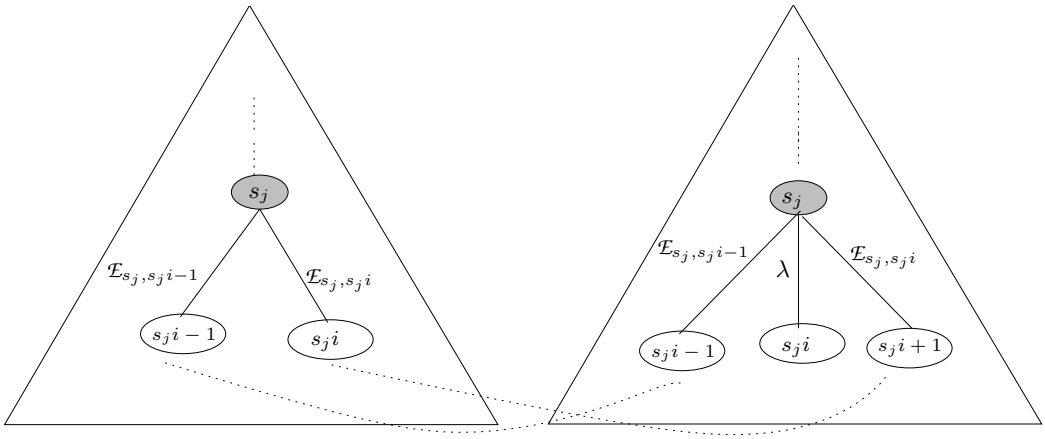


FIGURE 2.11 – $\mathcal{U}_i$ pour axis=child

– soit, en filtrant pour tout i $\in$ [0,k_{s_j}] l'expression régulière $\mathcal{E}_{(s_j,s_ji)}$ de $\mathcal{T}$ par une contrainte qui impose λ en première lettre. On peut obtenir le template $\mathcal{W}_i$ résultat de cette transformation en composant des opérateurs $Insert^v$ et $Rename$:
$\mathcal{W}_i = Rename_{\lambda,(s_j,s_ji)}(Insert^v_{\lambda^{-1}\mathcal{E}_{(s_j,s_ji)},s_ji}(\mathcal{T}))$ (Figure 2.12)

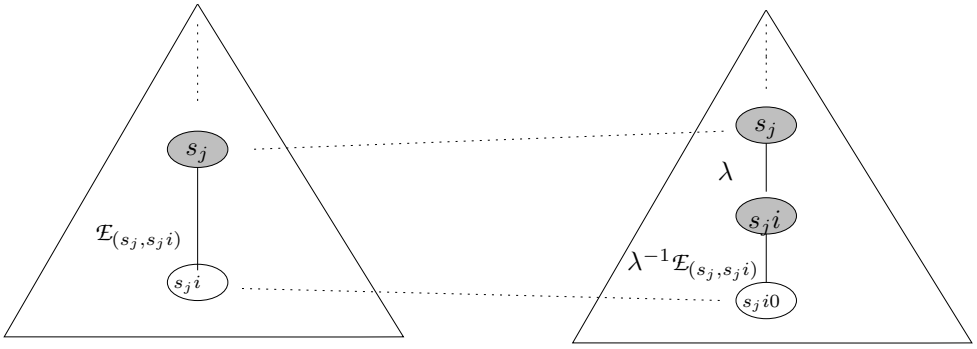


FIGURE 2.12 – $\mathcal{W}_i$ pour axis=child

Ainsi : $\mathscr{R}._{[j]}(child :: \lambda) \equiv (\bigcup_{i=0}^{k_{s_j}}(\mathcal{W}_i, (s'_1, s'_2, ..., s'_n, s_ji))) \cup (\bigcup_{i=0}^{k_{s_j+1}}(\mathcal{U}_i, (s_1, s_2, ..., s_n, s_ji))$
où $(s'_1, s'_2, ..., s'_n) = Insert^v_{s_ji}((s_1, s_2, ..., s_n))$
Si s_j=ϵ (k_{s_j}=0), on définit : $\mathscr{R}._{[i]}(child :: \lambda) \equiv (\mathcal{W}_0, (s'_1, s'_2, ..., s'_n, 0))$

tel-00580926, version 1 - 29 Mar 2011

Cas *axis*=descendant

Si $s_j \neq \epsilon$,pour chaque $\omega \in s_j\mathbb{N}^*$, descendant du nœud s_j, on construit un nouveau template à partir de $\mathscr{T}$:

- soit en insérant à la position i une nouvelle arête $(\omega, \omega i)$ étiqueté par $\Sigma^*\lambda$, on notera par $\mathcal{U}^i_\omega$ le template obtenu : $\mathcal{U}^i_\omega = Insert^h_{\Sigma^*\lambda,\omega,i}(\mathscr{T})$ (Figure 2.13)

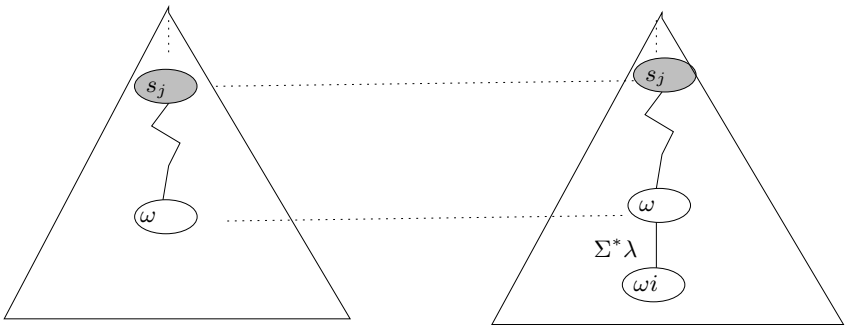


FIGURE 2.13 – $\mathcal{U}^i_\omega$ pour axis=descendant

- Soit en faisant apparaître sur une arête $\mathcal{E}_{(\omega,\omega')}$ de $\mathscr{T}$ sortant de ω, un nœud descendant de s_j étiqueté par λ. Pour spécifier la position de ce nœud sur l'arête (ω, ω'), qui peut être quelconque, on partitionne $\mathcal{E}_{(\omega,\omega')} \cap \Sigma^*\lambda\Sigma^*$ en union de produits d'expressions régulières E_1E_2 où E_1 vérifie $L(E_1) \subset \Sigma^*\lambda$. Cette partition est obtenue à partir de l'étude des calculs d'un automate fini $\mathscr{A}$ sur les mots de $L(\mathcal{E}_{(\omega,\omega')})$, comme décrit ci-après.
  Soit $\mathscr{A}$=$(\Sigma, Q, q_0, \delta, F)$ un automate déterministe de mots reconnaissant $\mathrm{L}(\mathcal{E}_{(\omega,\omega')})$, on suppose que tout état q est accessible à partir de q_0 et permet d'atteindre au moins un état final de F. On note $Q_\lambda = \{q \in Q \ tel \ que \ \exists t \in Q \ , (t, \lambda, q) \in \delta\}$.
  Si E_q (respectivement $\bar{E}_q$) est une expression régulière tel que $\mathrm{L}(E_q)$=L_q= $\mathrm{L}(\mathcal{A},\{q_0\},\{q\})$ (respectivement $\mathrm{L}(\bar{E}_q)$=$\bar{L}_q$ = $\mathrm{L}(\mathcal{A},\{q\},\mathrm{F})$). On a alors
  $\mathcal{E}_{(\omega,\omega')} \cap \Sigma^*\lambda\Sigma^* = \bigcup_{q\in Q_\lambda} (E_q \cap \Sigma^*\lambda)\bar{E}_q$.
  Cette partition de $\mathcal{E}_{(\omega,\omega')} \cap \Sigma^*\lambda\Sigma^*$ permet de mettre en évidence la position d'une occurrence de λ dans un mot de $\mathrm{L}(\mathcal{E}_{(\omega,\omega')}) \cap \Sigma^*\lambda\Sigma^*$, en utilisant l'état q de Q_λ dans lequel $\mathscr{A}$ se trouve après la lecture de cette occurrence de λ.
  Pour chaque q $\in Q_\lambda$, on construit un template $\mathcal{W}^q_{(\omega,\omega')}$ à partir de $\mathscr{T}$ en introduisant un nœud intermédiaire sur (ω, ω') et en imposant l'expression régulière $(E_q \cap \Sigma^*\lambda)\bar{E}_q$ entre ω et ω'. Le template $\mathcal{W}^q_{(\omega,\omega')}$ peut être obtenu par composition des opérateurs $Insert^v$ et $Rename$:

  $\mathcal{W}^q_{(\omega,\omega')} = Rename_{E_q\cap\Sigma^*\lambda,(\omega,\omega')}(Insert^v_{\bar{E}_q,\omega'}(\mathscr{T}))$ (Figure 2.14)

tel-00580926, version 1 - 29 Mar 2011

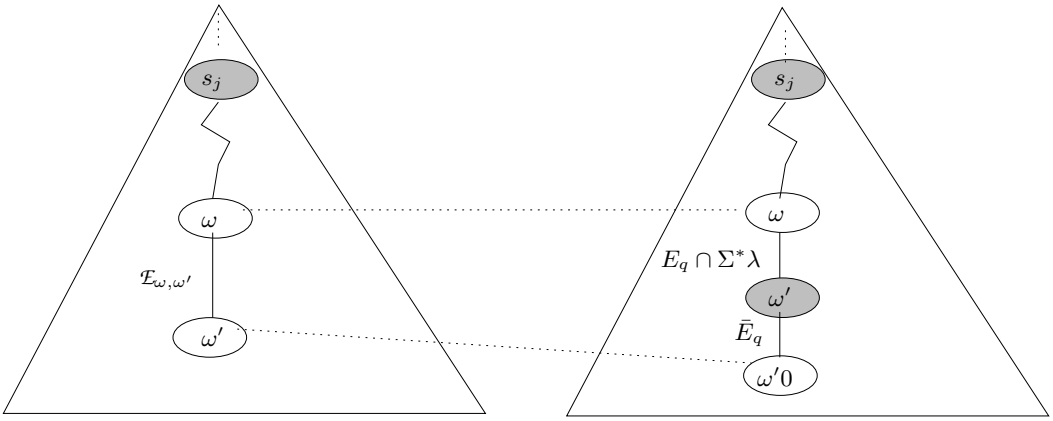


FIGURE 2.14 – $\mathcal{W}^q_{(\omega,\omega')}$ pour axis=descendant

Finalement on peut écrire :
$\mathscr{R}.[j](descendant :: \lambda) \equiv (\bigcup_{\omega \in s_n\mathbb{N}^*, i\in[0,k_\omega+1]} (\mathcal{U}^i_\omega, (s_1, s_2, ..., s_n, \omega i)))$
$\cup (\bigcup_{(\omega,\omega')\in M, \omega\in s_n\mathbb{N}^*} (\bigcup_{q\in Q_\lambda} \mathcal{W}^q_{(\omega,\omega')}, (s'_1, s'_2, ..., s'_n, \omega')))$
où M désigne l'ensemble des arêtes de $\mathscr{T}$ et $(s'_1, s'_2, ..., s'_n) = Insert^v_{\omega'}(s_1, s_2, ..., s_n)$

Si s_j=ϵ, la construction est identique excepté le fait que les $\mathcal{U}^i_\omega$ seront construits pour tout ω de $s_j\mathbb{N}^+$.

Cas $axis$=parent

Si s_j=ϵ alors $\mathscr{R}.[j](parent :: \lambda) \equiv \mathscr{R}_\emptyset$, sinon soit σ le père du nœud s_j i.e s_j=(σi) pour un certain entier i.
Il est alors nécessaire de savoir si, par tout plongement, les nœuds images de σ et de s_j conserve la relation 'père-fils' dans le document. Pour cela, on décompose l'expression régulière $\mathcal{E}_{(\sigma,s_j)}$ en $\mathcal{E}_{(\sigma,s_j)} = E^1 \cup E^2$ avec $E^1 = \mathcal{E}_{(\sigma,s_j)} \cap \Sigma$ et $E^2 = \mathcal{E}_{(\sigma,s_j)} \cap \Sigma^*\Sigma^2$.
La requête $\mathscr{R}$ est alors équivalente à l'union des deux requêtes $\mathscr{R}_1 = (\mathscr{T}_1, \vec{s})$ et $\mathscr{R}_2 = (\mathscr{T}_2, \vec{s})$ où $\mathscr{T}_1$ (respectivement $\mathscr{T}_2$) est obtenu à partir de $\mathscr{T}$ en remplaçant $\mathcal{E}_{(\sigma,s_j)}$ par E^1 (respectivement E^2).
Dans le premier cas (cas $\mathscr{R}_1$ où l'inclusion $L(E^1) \subseteq \Sigma$ assure la conservation par tout plongement de la relation père-fils entre σ et s_j), il faut imposer que le nœud plongement de σ dans le document soit étiqueté par λ.
Si $\sigma = \epsilon$ alors $\mathscr{R}.[j](parent :: \lambda) \equiv \mathscr{R}_\emptyset$, sinon soit τ le père du nœud σ dans $\mathscr{T}_1$, on construit l'arbre template $\mathcal{Y}$ en filtrant l'expression régulière $\mathcal{E}_{(\tau,\sigma)}$ par $\Sigma^*\lambda$ pour imposer λ comme dernière lettre : $\mathcal{Y} = Rename_{\mathcal{E}_{(\tau,\sigma)}\cap\Sigma^*\lambda,(\tau,\sigma)}(\mathscr{T}_1)$ (Figure 2.15).

tel-00580926, version 1 - 29 Mar 2011

Dans le deuxième cas (cas $\mathscr{R}_2$ où la conservation de la relation père-fils entre σ

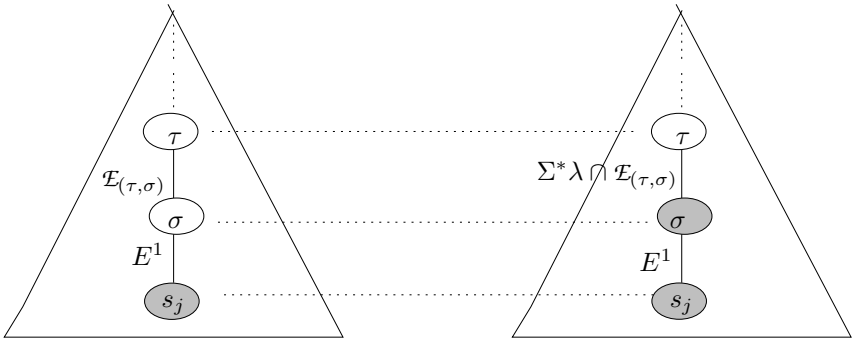


FIGURE 2.15 – Cas $\mathscr{R}_1$: $\mathcal{Y}$ pour axis=parent

et s_j n'est pas assurée par tout plongement), il faut faire apparaître un nouveau nœud entre σ et s_j représentant, après plongement, le père du plongement de s_j et imposer λ comme étiquette pour ce nœud. Pour cela on filtre E^2 en ne retenant que les mots de $E^2 \cap \Sigma^*\lambda\Sigma$. En utilisant la partition $E^2 \cap \Sigma^*\lambda\Sigma = \bigcup_{l\in\Sigma}(E^2 \cap \Sigma^*\lambda l)$, on construit pour chaque lettre $l \in \Sigma$, un template $\mathcal{W}_l$ faisant apparaître le père de s_j et n'utilisant que les mots de $E^2 \cap \Sigma^*\lambda l$. $\mathcal{W}_l$ est obtenu par la composition des opérateurs $Insert^v$ et $Rename$ (Figure 2.16) : $\mathcal{W}_l = Rename_{E^2 l^{-1}\cap\Sigma^*\lambda,(\sigma,s_j)}(Insert^v_{l,s_j}(\mathscr{T}))$

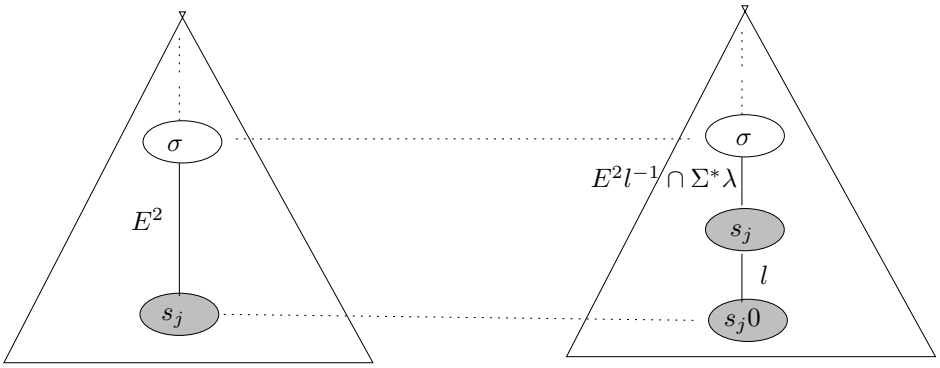


FIGURE 2.16 – $\mathcal{W}_l$ pour axis=parent

Finalement on peut écrire :
$\mathscr{R}_{.[j]}(parent :: \lambda) \equiv (\mathcal{Y}, (s_1, s_2, ..., s_n, \sigma)) \cup \bigcup_{l\in\Sigma}(\mathcal{W}_l, (s'_1, s'_2, ..., s'_n, s_j))$ avec
$(s'_1, s'_2, ..., s'_n) = Insert^v_{s_j}(s_1, s_2, ..., s_n)$.

tel-00580926, version 1 - 29 Mar 2011

Cas $axis$=ancestor

Si s_j=ϵ alors $\mathscr{R}_{.[j]}(ancestor :: \lambda) \equiv \mathscr{R}_{\emptyset}$.
Sinon ($s_j \neq \epsilon$), l'idée est de construire des templates faisant apparaître un nœud étiqueté par λ sur une arête $(\omega, \omega i)$ reliant deux ancêtres de s_j (i.e $(\omega, \omega i) \in M$ et $\omega i <= s_j$).
Dans le cas $\omega i < s_j$, il faut donc, d'une part, remplacer l'expression rationnelle $\mathcal{E}_{(\omega,\omega i)}$ par $\mathcal{E}_{(\omega,\omega i)} \cap \Sigma^*\lambda\Sigma^*$ et d'autre part, utiliser la même démarche que celle utilisée dans le cas où $axis$=descendant pour faire apparaître un nœud sur l'arête $(\omega, \omega i)$ étiqueté par une occurrence de λ d'un mot de $\mathrm{L}(\mathcal{E}_{(\omega,\omega i)}) \cap \Sigma^*\lambda\Sigma^*$. On obtient donc les templates $\mathcal{W}^q_{(\omega,\omega i)}$ définis pour tout $(\omega, \omega i) \in M$ vérifiant $\omega i < s_j$ et pour tout $q \in Q_\lambda$ (où Q_λ est défini comme dans le cas $axis$=descendant avec $\mathcal{E}_{(\omega,\omega i)} \cap \Sigma^*\lambda\Sigma^* = \bigcup_{q \in Q_\lambda} (E_q \cap \Sigma^*\lambda)\bar{E}_q$)
$\mathcal{W}^q_{(\omega,\omega i)} = Rename_{E_q \cap \Sigma^*\lambda,(\omega,\omega i)}(Insert^v_{\bar{E}_q,\omega i}(\mathscr{T}))$ (Figure 2.17)

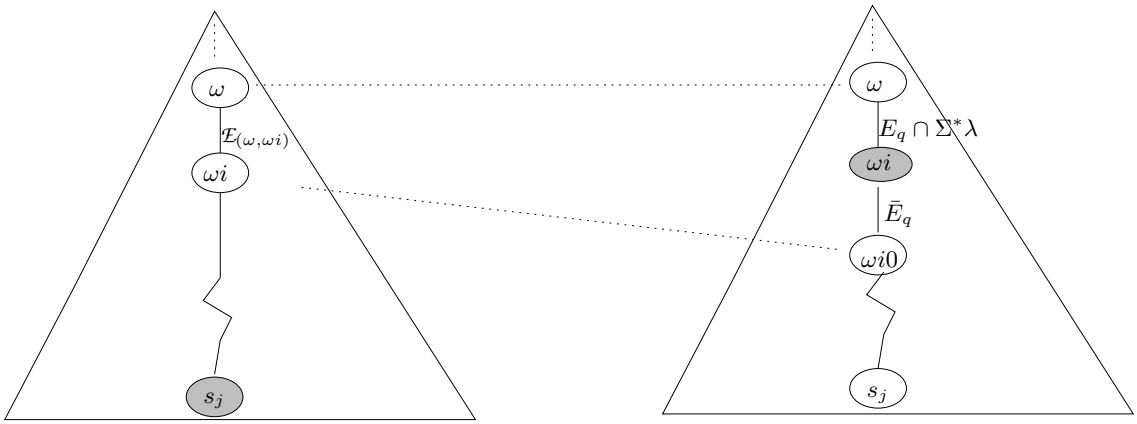


FIGURE 2.17 – $\mathcal{W}_l$ pour axis=ancestor

Dans le cas où $\omega i = s_j$, On utilise la même démarche en remplaçant $\Sigma^*\lambda\Sigma^*$ par $\Sigma^*\lambda\Sigma^+$ afin d'assurer que le nœud introduit représente un ancêtre strict de s_j. Pour tout q $\in Q_\lambda$, on construit donc le template : $\mathcal{U}^q = Rename_{E_q \cap \Sigma^*\lambda,(\omega,s_j)}(Insert^v_{\bar{E}_q,s_j}(\mathscr{T}))$ avec $\bar{E}_q$ désignant une expression régulière associée au langage $\bar{L}_q \cap \Sigma^+$.

Finalement :
$\mathscr{R}_{.[j]}(ancestor :: \lambda) \equiv (\bigcup_{(\omega,\omega i) \in M, \omega i < s_j} (\bigcup_{q \in Q_\lambda} (\mathcal{W}^q_{(\omega,\omega')}, (\vec{s'}, \omega i))) \cup (\bigcup_{q \in Q_\lambda} (\mathcal{U}^q, (\vec{s'}, s_j)))$.
où $\vec{s'} = Insert^v_{\omega i}(\vec{s})$.

tel-00580926, version 1 - 29 Mar 2011

Cas *axis*=following_sibling

Si s_j=ϵ alors $\mathscr{R}_{.[j]}(following\_sibling :: \lambda) \equiv \mathscr{R}_{\emptyset}$. Sinon ($s_j \neq \epsilon$), on note σ le père du nœud s_j i.e s_j=σi_{s_j}. Remarquons que $P_a/following\_sibling :: \lambda$ est équivalent à $P_a/parent :: */child_{d(s_j)} :: \lambda$ où $child_{d(s_j)}$ désigne un sous-ensemble de l'axe *child* correspondant aux fils de σ situés à droite de s_j. Cette équivalence nous conduit donc à combiner la démarche utilisée pour le cas *axis*=parent avec celle utilisée pour le cas *axis*=child en l'adaptant au sous-axe $child_{d(s_j)}$.

On décompose $\mathcal{E}_{(\sigma,s_j)}$ comme dans le cas *axis*=parent, en $\mathcal{E}_{(\sigma,s_j)} = E^1 \cup E^2$ avec $E^1 = \mathcal{E}_{(\sigma,s_j)} \cap \Sigma$ et $E^2 = \mathcal{E}_{(\sigma,s_j)} \cap \Sigma^2\Sigma^*$. La requête $\mathscr{R}$ est alors l'union des deux requêtes $\mathscr{R}_1$ et $\mathscr{R}_2$ utilisant respectivement E^1 et E^2 à la place de $\mathcal{E}_{(\sigma,s_j)}$.
Dans le cas de $\mathscr{R}_1$ où les plongements de σ et s_j conservent la relation père-fils, si σ=ϵ, on a $\mathscr{R}_{1\cdot[j]}(following\_sibling :: \lambda) \equiv \mathscr{R}_{\emptyset}$, sinon ($\sigma \neq \epsilon$) on construit :

(a) pour chaque i de $[i_{s_j}+1, k_\sigma+1]$, l'arbre template $\mathcal{U}_i$ obtenu en insérant un nouveau fils de σ à la position i (à droite de s_j) et en étiquetant l'arc $(\sigma, \sigma i)$ avec l'expression régulière λ : $\mathcal{U}_i = Insert^h_{\lambda,\sigma,i}(\mathscr{T}_1)$

(b) et pour chaque i de $[i_{s_j}+1, k_\sigma]$, l'arbre template $\mathcal{Y}_i$ obtenu en filtrant l'expression régulière $\mathcal{E}_{(\sigma,\sigma i)}$ (qui étiquette une arête sortante de σ située à droite de (σ, s_j)) par une contrainte qui impose λ comme première lettre :
$\mathcal{Y}_i = Rename_{\lambda,(\sigma,\sigma i)}(Insert^v_{\lambda^{-1}{}_{\mathcal{E}_{(\sigma,\sigma i)}},\sigma i}(\mathscr{T}_1))$.

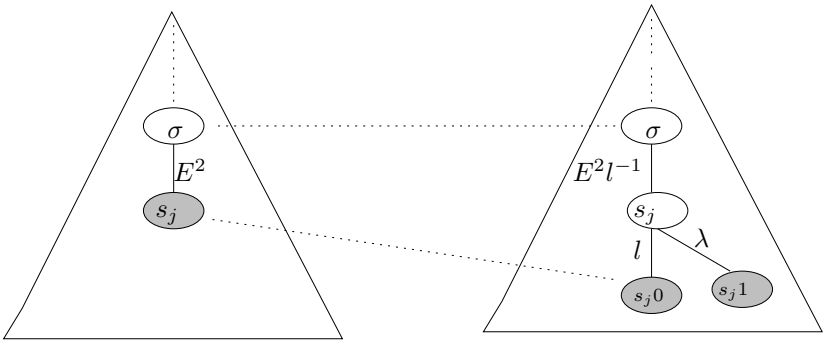


FIGURE 2.18 – $\mathcal{W}_l$ pour *axis*=following_sibling

Dans le cas de $\mathscr{R}_2$, on utilise comme dans le cas *axis*=parent, la partition $E^2 = \bigcup_{l\in\Sigma}(E^2 \cap \Sigma^* l)$ pour identifier la dernière lettre l d'un mot de $L(E^2)$. Pour chaque lettre l, on construit un template $\mathcal{W}_l$ en faisant apparaître un nouveau nœud entre σ et s_j représentant, après plongement, le père du plongement de s_j, et en ajoutant à ce nouveau nœud un second fils (à droite de s_j) à étiqueter par λ :
$\mathcal{W}_l = Insert^h_{\lambda,s_j,1}(Rename_{E^2 l^{-1},(\sigma,s_j)}(Insert^v_{l,s_j}(\mathscr{T}_2)))$

tel-00580926, version 1 - 29 Mar 2011

Finalement :
$\mathscr{R}._{[j]}(following\_sibling :: \lambda) \equiv (\bigcup_{i=i_{s_j}+1}^{k_\sigma} (\mathcal{Y}_i, (\vec{s'}, \sigma i))) \cup (\bigcup_{i=i_{s_j}+1}^{k_\sigma+1} (\mathcal{U}_i, (\vec{s}, \sigma i)))$
$\cup (\bigcup_{l\in\Sigma} (\mathcal{W}_l, (\vec{s''}, s_j 1)))$.
où $\vec{s'} = Insert^v_{\sigma i}(\vec{s})$ et $\vec{s''} = Insert^v_{s_j}(\vec{s})$.

Cas $axis$=preceding_sibling

On procède comme dans le cas précédent mais en ne créant des frères ou en ne modifiant des arêtes qu'à gauche du nœud s_j. Ainsi :
$\mathscr{R}._{[j]}(preceding\_sibling :: \lambda) \equiv (\bigcup_{i=0}^{i_{s_j}-1} (\mathcal{Y}_i, (\vec{s'}, \sigma i))) \cup (\bigcup_{i=1}^{i_{s_j}} (\mathcal{U}_i, (\vec{s}, \sigma i)))$
$\cup (\bigcup_{l\in\Sigma} (\mathcal{W}_l, (\vec{s''}, s_j 0)))$.
avec $\mathcal{U}_i = Insert^h_{\lambda,\sigma,i}(\mathscr{T}_1)$, $\mathcal{Y}_i = Rename_{\lambda,(\sigma,\sigma i)}(Insert^v_{\lambda^{-1}\mathscr{E}_{(\sigma,\sigma i)},\sigma i}(\mathscr{T}_1))$,

et $\mathcal{W}_l = Insert^h_{\lambda,s_j,0}(Rename_{E^{2l-1},(\sigma,s_j)}(Insert^v_{l,s_j}(\mathscr{T}_2)))$

où $\vec{s'} = Insert^v_{\sigma i}(\vec{s})$ et $\vec{s''} = Insert^v_{s_j}(\vec{s})$.

Cas $axis \in$ {ancestor_or_self, descendant_or_self, following, preceding}

On peut écrire :
$\mathscr{R}._{[j]}(ancestor\_or\_self :: \lambda) \equiv \mathscr{R}._{[j]}(ancestor :: \lambda) \cup \mathscr{R}._{[j]}(self :: \lambda)$

$\mathscr{R}._{[j]}(descendant\_or\_self :: \lambda) \equiv \mathscr{R}._{[j]}(descendant :: \lambda) \cup \mathscr{R}._{[j]}(self :: \lambda)$

$\mathscr{R}._{[j]}(following :: \lambda) \equiv \Pi_{1,2,\ldots,n,n+3}(((\mathscr{R}._{[j]}(ancestor\_or\_self :: *)).{}_{[n+1]}following\_sibling :: *).{}_{[n+2]}descendant\_or\_self :: \lambda)$

$\mathscr{R}._{[j]}(preceding :: \lambda) \equiv \Pi_{1,2,\ldots,n,n+3}(((\mathscr{R}._{[j]}(ancestor\_or\_self :: *)).{}_{[n+1]}preceding\_sibling :: *).{}_{[n+2]}descendant\_or\_self :: \lambda)$.

Ainsi l'examen des axes précédents permet également de conclure pour ces 4 derniers axes.

□

Nous pouvons maintenant établir la stabilité des unions de requêtes $\mathcal{RAR}s$ par extension :

tel-00580926, version 1 - 29 Mar 2011

Proposition 1. *(Stabilité par extension)*
1. Soit F un filtre de $CoreXPath^+$, alors pour tout entier j et pour toute requête $\mathcal{RAR}$ $\mathscr{R}$=($\mathscr{T}$,$\vec{s}$), $\mathscr{R}._{[j]}F$ est équivalente à une union finie de requêtes $\mathcal{RAR}$s.
2. Les unions de requêtes $\mathcal{RAR}$s sont stables par extension.

Démonstration. Le point 2 est une conséquence immédiate du point 1. On établit le point 1 en procédant par récurrence sur la structure de F.

cas de base :
a- $F = axis :: \lambda$, le résultat est immédiat on appliquant le lemme 2.
b- $F = /axis :: \lambda$, on a

$$\begin{aligned} Eval_{\mathcal{D}}(\mathscr{R}._{[j]}F) &= \{(u_1,...,u_n,u_{n+1}) \in \mathcal{N}(\mathcal{D})^{n+1} / (u_1,...,u_n) \in Eval_{\mathcal{D}}(\mathscr{R}) \text{ et} \\ &\quad u_{n+1} \in [\![F]\!]u_j\} \\ &= \{(u_1,...,u_n,u_{n+1}) \in \mathcal{N}(\mathcal{D})^{n+1} / (u_1,u_2,...,u_n) \in Eval_{\mathcal{D}}(\mathscr{R}) \text{ et} \\ &\quad u_{n+1} \in [\![axis :: \lambda]\!]\epsilon\} \\ &= \{(u_1,...,u_n,u_{n+1}) \in \mathcal{N}(\mathcal{D})^{n+1} / \\ &\quad (u_1,...,u_n,\epsilon,u_{n+1}) \in (\mathscr{T},(\vec{s},\epsilon))._{[n+1]}(axis :: \lambda)\} \end{aligned}$$

Ainsi $\mathscr{R}._{[j]}F \equiv \Pi_{1,2,...,n,n+2}((\mathscr{T},(\vec{s},\epsilon))._{[n+1]}(axis :: \lambda))$,
d'où le résultat en utilisant le cas de base (a) et le lemme 1 section 2.4.

Étape d'induction :
Soit F un filtre de $CoreXPath^+$ pour lequel le résultat a été établi et $\mathscr{R}$=($\mathscr{T}$,$\vec{s}$) une requête $\mathcal{RAR}$ n-aire.
- Si $G = axis :: \lambda[F]$ et $\mathscr{R}$=($\mathscr{T}$,$\vec{s}$) une requête $\mathcal{RAR}$ n-aire. D'après la propriété 3 section 3.1,

$$\mathscr{R}._{[j]}(G) = \mathscr{R}._{[j]}(axis :: \lambda[F]) \equiv \Pi_{(1,...,n+1)}((\mathscr{R}._{[j]}(axis :: \lambda))._{[n+1]}F).$$

D'après le lemme 2, $\mathscr{R}._{[j]}(axis :: \lambda)$ est une union de requêtes $\mathcal{RAR}$ (n+1)-aires, par la propriété 1 section 3.1 et l'hypothèse de récurrence $(\mathscr{R}._{[j]}(axis :: \lambda))._{[n+1]}F$ est une union de requêtes $\mathcal{RAR}$s. Ainsi en utilisant à nouveau le lemme 1 section 2.4, $\Pi_{(1,...,n+1)}((\mathscr{R}._{[j]}(axis :: \lambda))._{[n+1]}F)$ est une union de requêtes $\mathcal{RAR}$s d'où le résultat.

- Si G= /axis : :λ[F], on a

$$\mathscr{R}._{[j]}(G) = \mathscr{R}._{[j]}(/axis :: \lambda[F]) \equiv \Pi_{(1,...,n,n+2)}(((\mathscr{T},(\vec{s},\epsilon))._{[n+1]}(axis :: \lambda))._{[n+2]}F).$$

et le résultat est à nouveau obtenu en utilisant l'hypothèse de réccurence, le cas de base (a) et le lemme 1 section 2.4. □

tel-00580926, version 1 - 29 Mar 2011

Nous définissons le filtrage $\mathscr{R}_{[i]}[F]$ d'une requête $\mathcal{RAR}$ n-aire $\mathscr{R}$ par un filtre F de $CoreXPath$ positif par : $\mathscr{R}_{[i]}[F] = \Pi_{(1,2,\ldots,n)}(\mathscr{R}._{[i]}F)$. Le corollaire suivant établit la stabilité des unions des requêtes $\mathcal{RARS}$ par filtrage et est une conséquence directe de la proposition 1 et du lemme 1.

Corollaire 1. *Soit $\mathscr{R}$=($\mathscr{T}$,$\vec{s}$) une requête $\mathcal{RAR}$ et F un filtre de $CoreXPath^+$ alors $\mathscr{R}_{[i]}[F]$ est équivalente à une union finie de requêtes $\mathcal{RARS}$.*

3.3 Requêtes de $CoreXPath^+$ versus requêtes $\mathcal{RARS}$

Nous pouvons maintenant énoncer le résultat principal de la section 3 :

Théorème 6. *Tout chemin P_a de $CoreXPath^+$ est équivalent à une union finie de requêtes monadiques $\mathcal{RARS}$.*

Démonstration. Soit $P_a = /P$, On procède par induction sur la structure de P.

Cas de base ($P = axis :: \lambda$) :
- Si $P = child :: \lambda$ alors P_a est équivalent à la requête arbre régulière $\mathscr{R}_P = \{(\epsilon = [\lambda \rightarrow 0]), \{0\}\}$ qui sélectionne la racine du document si elle est étiquetée par λ.
- Si $P = descendant :: \lambda$ alors P_a est équivalent à la requête arbre régulière $\mathscr{R}_P = \{(\epsilon = [\Sigma^*\lambda \rightarrow 0]), \{0\}\}$ qui sélectionne les nœuds du document étiquetés par λ.
- Si $P = axis :: \lambda$ avec $axis \notin$ {child,descendant} alors P_a ne sélectionne aucun nœud et est donc équivalent à la requête arbre régulière $\mathscr{R}_\emptyset$ dont l'évaluation est toujours vide.

Étape d'induction :
Si $P_a = /P/axis :: \lambda$ ou $P_a = /P[F]$ avec /P équivalent à une union de requêtes $\mathcal{RARS}$ monadiques $\cup_k \mathscr{R}_k$, alors le lemme 2 et le corollaire 1 donnent le résultat car on a :
$/P/axis :: \lambda$ est équivalent à $\cup_k \Pi_{\{2\}}(\mathscr{R}_{k\cdot[1]}(axis :: \lambda))$ et $/P[F]$ est équivalente à $\cup_k \mathscr{R}_k[F]$.
Si $P_a = /(P_1|P_2)$ avec $/P_1$ et $/P_2$ équivalents à des unions de requêtes $\mathcal{RARS}$ monadiques alors le résultat est immédiat car P_a est équivalent à $/P_1 \cup /P_2$. □

3.4 Un fragment de requêtes $\mathcal{RARS}$

Les requêtes $\mathcal{RARS}$ peuvent exprimer des requêtes que XPath ne peut pas exprimer. Grâce à l'utilisation de l'opérateur de répétition '$*$' dans une expression régulière. Nous verrons en effet dans la section 4 que la requête $\mathcal{RAR}$ (figure 2.19) n'est pas exprimable par $CXPath$ et donc à fortiori par XPath.

tel-00580926, version 1 - 29 Mar 2011

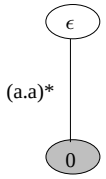


FIGURE 2.19 –

Ainsi la réciproque du théorème 6 n'est pas vrai. Cependant, dans cette section nous établissons une variante de la réciproque du théorème 6 sur un fragment de requêtes $\mathcal{RAR}s$ qui utilisent des expressions régulières simplifiées (où l'opérateur '$*$' n'est autorisé que sur Σ). Nous montrons alors que toute requête $\mathcal{RAR}$ monadique de ce fragment est exprimable par une union finie de chemins du fragment $XP^{\{[],*,//,fs,ps\}}$ de CoreXPath, étendant le fragment simple $XP^{\{*,[],//\}}$ avec les deux axes `following_sibling` (noté fs) et `preceding_sibling` (noté ps).
Formellement, on définit $XP^{\{[],*,//,fs,ps\}}$ par la grammaire :

$P \equiv axis :: \lambda \ \| \ P[F] \ \| \ P/axis :: \lambda$

F $\equiv$ axis:: λ $\|$ axis:: λ[F] $\|$ /F

Avec $\lambda \in \Sigma \cup \{*\}$ et axis $\in$ { `child, descendant, following_sibling, preceding_sibling` }.
On note $\mathcal{RAR}^{\{|,.,\Sigma^*\}}$ l'ensemble des requêtes $\mathcal{RAR}s$ utilisant des expressions régulières E définies par : $E \equiv E|E \ \| \ E.E \ \| \ \Sigma^* \ \|$ c .(c une lettre de Σ).
$\mathcal{RAR}^{\{\Sigma^*\}}$ désigne les requêtes $\mathcal{RAR}s$ n'autorisant que Σ^* ou un caractère c de Σ comme expressions régulières.
Nous commençons par un lemme établissant la relation qui existe entre le fragment $\mathcal{RAR}^{\{|,.,\Sigma^*\}}$ et le fragment $\mathcal{RAR}^{\{\Sigma^*\}}$:

Lemme 3. *Toute requête de $\mathcal{RAR}^{\{|,.,\Sigma^*\}}$ est exprimable par une union finie de requêtes de $\mathcal{RAR}^{\{\Sigma^*\}}$.*

Démonstration. Soit une requête $\mathcal{RAR}$ $\mathscr{R} = (\mathscr{T}, \vec{s})$ de $\mathcal{RAR}^{\{|,.,\Sigma^*\}}$. Nous transformons $\mathscr{R}$ en une union finie de requêtes de $\mathcal{RAR}^{\{\Sigma^*\}}$ qui lui est équivalente. La procédure de transformation examine chaque arête e de $\mathscr{T}$ et transforme $\mathscr{T}$ selon l'expression régulière E_e apparaissant sur cette arête. Cette procédure est exprimée par l'algorithme transform (Algorithme 1). □

Nous établissons tout d'abord un résultat reliant les filtres de $XP^{\{[],*,//,fs,ps\}}$ et le sous ensemble , noté $\mathscr{T}^{\{\Sigma^*\}}$, des templates qui n'utilisent comme expressions régulières que Σ^* ou une lettre de Σ.

Proposition 2. *Soit $\mathscr{T}$ un template de $\mathscr{T}^{\{\Sigma^*\}}$ et $\mathcal{D}$ un document semi-structuré , alors il existe un filtre F de $XP^{\{[],*,//,fs,ps\}}$ tel que $\forall$ $n \in \mathcal{D}$, $n \vDash_{\mathcal{D}} \mathscr{T} \Leftrightarrow n \vDash_{\mathcal{D}} F$ (*).*

tel-00580926, version 1 - 29 Mar 2011

Algorithm 1 Algorithm transform

Require: une requête $\mathscr{R} = (\mathscr{T}, \vec{s})$ de $\mathcal{RAR}^{\{|,.,\Sigma^*\}}$
  sortie : un ensemble de requête ξ de $\mathcal{RAR}^{\{\Sigma^*\}}$ dont l'union est équivalente à $\mathscr{R}$
  ξ=$\{\mathscr{R}\}$
  $\xi_1 \leftarrow \{\mathscr{R} \in \xi \ / \ \mathscr{R} \notin \mathcal{RAR}^{\{\Sigma^*\}}\}$
  while $\xi_1 \neq \emptyset$ **do**
    choisir une requête $\mathscr{R} = (\mathscr{T}, \vec{s})$ de ξ_1 et une arête e=(i,j) de $\mathscr{T}$ étiquetée ni par Σ^* ni par une lettre de Σ
    if $E_e = E_1|E_2$ **then**
      supprimer $\mathscr{R}$ de ξ
      $\xi \leftarrow \xi \cup \{\mathscr{R}_1, \mathscr{R}_2 \ / \ \mathscr{R}_1$ (respectivement $\mathscr{R}_2$) est l'arbre $\mathscr{R}$ dans lequel on a remplacé E_e par E_1 (respectivement E_2) (Figure 2.20) $\}$
    end if
    if $E_e = E_1.E_2$ **then**
      supprimer $\mathscr{R}$ de ξ
      $\xi \leftarrow \xi \cup \{\mathscr{R}' \ / \ \mathscr{R}' = (\mathscr{T}', \vec{s'})$ où $\mathscr{T}'$ est l'arbre $\mathscr{T}$ dans lequel on a remplacé E_e par E_1 et ajouté une nouvelle arête (j,j0) étiquetée par E_2, i.e $\mathscr{T}' = Rename_{E_1,(i,j)}(Insert^v_{E_2,j}(\mathscr{T}))$ et $\vec{s'} = Insert^v_j(\vec{s})$ (Figure 2.21)$\}$
    end if
    $\xi_1 \leftarrow \{\mathscr{R} \in \xi \ / \ \mathscr{R} \notin \mathcal{RAR}^{\{\Sigma^*\}}\}$
  end while
  return ξ

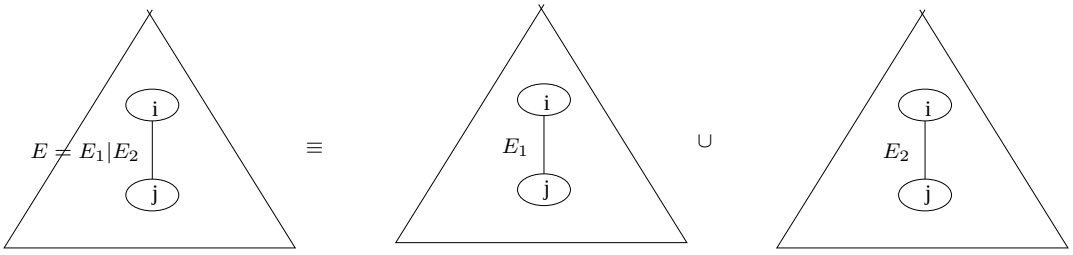


FIGURE 2.20 – Transformation de l'alternative

Démonstration. On procède par récurrence sur le nombre d'arêtes de $\mathscr{T}$.
Cas de base :
- Si $\mathscr{T} = (\epsilon\ )$, on pose F=self : :*.
- Si $\mathscr{T} = (\epsilon = [c \rightarrow 0];)$, on pose F=child : :c .
- Si $\mathscr{T} = (\epsilon = [\Sigma^* \rightarrow 0];)$, on pose F=descendant : : * .

Et on a clairement dans les trois cas : $\forall$ n $\in \mathcal{D}$, n$\vDash_{\mathcal{D}} \mathscr{T} \Leftrightarrow$ n $\vDash_{\mathcal{D}}$ F.
Etape d'induction : On suppose le résultat établi pour des templates contenant moins de n arêtes et soit un template $\mathscr{T}$ comportant n arêtes, $\mathscr{T}$ s'écrit sous la forme indiquée en (Figure 2.22), où $c_i = \Sigma^*$ ou $c_i \in \Sigma$, pour $i = 0, ..., k$.

tel-00580926, version 1 - 29 Mar 2011

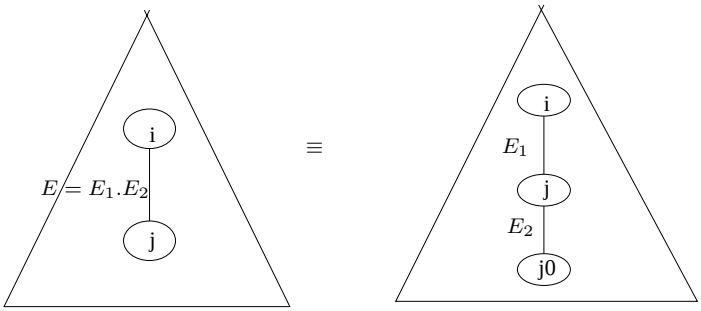


FIGURE 2.21 – Transformation de la cancatenation

Les templates $\mathscr{T}_0$,...,$\mathscr{T}_k$ ont un nombre d'arêtes $<$ n, donc d'après l'hypothèse de

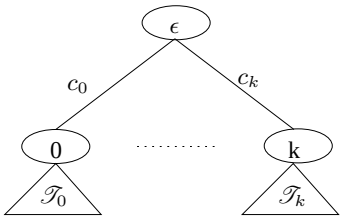


FIGURE 2.22 – Template $\mathscr{T}$

récurrence il existe des filtres F_1,..., F_k de $XP^{\{[],*,//,fs\}}$ qui vérifient :
pour tout i, $\forall$ n $\in \mathcal{D}$, n$\vDash_{\mathcal{D}} \mathscr{T}_i \Leftrightarrow$ n $\vDash_{\mathcal{D}} F_i$.
Alors :
- cas 1 : si pour tout i $c_i \neq \Sigma^*$ alors
le filtre F=$child :: c_0[F_0][fs :: c_1[F_1]....fs :: c_k[F_k]]...]$ vérifie la condition (*).
- cas 2 : si il existe j tel que $c_j = \Sigma^*$ alors on pose
$F = child :: \gamma_0[H_0][fs :: \gamma_1[H_1][...[fs :: \gamma_k[H_k]]...]$ avec

$$\gamma_i = c_i \; et \; H_i = F_i \; si \; c_i \in \Sigma$$
$$\gamma_i = * \; et \; H_i = descendant\_or\_self :: *[F_i] \; si \; c_i = \Sigma^*$$

□

Théorème 7. *Toute requête monadique $\mathscr{R}$ de $\mathcal{RAR}^{\{|,.,\Sigma^*\}}$ est exprimable par une union finie de requêtes de $XP^{\{[],*,//,fs,ps\}}$.*

Démonstration. D'après le lemme 3 $\mathscr{R}$ est équivalente à une union finie de requêtes de $\mathcal{RAR}^{\{\Sigma^*\}}$. i.e $\mathscr{R} = \cup \mathscr{R}_i$ avec $\mathscr{R}_i$ requête de $\mathcal{RAR}^{\{\Sigma^*\}}$. Donc il suffit de démontrer que toute requête $\mathscr{R}$ de $\mathcal{RAR}^{\{\Sigma^*\}}$ est exprimable par une requête de $XP^{\{[],*,//,fs,ps\}}$. On le démontre par récurrence sur la taille n de $\mathscr{R}$.

tel-00580926, version 1 - 29 Mar 2011

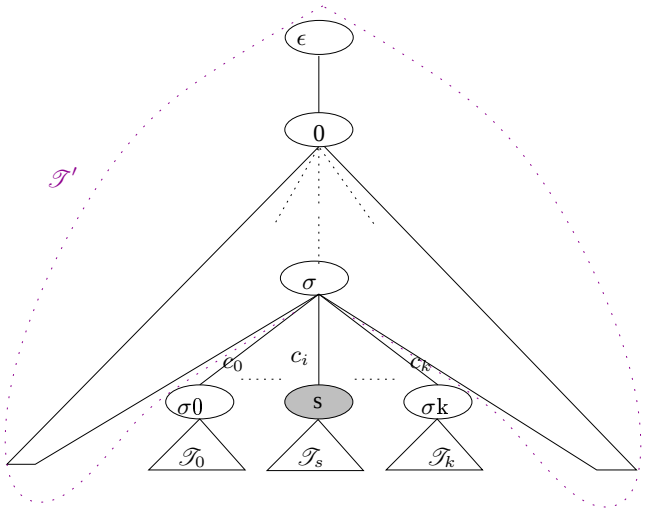


FIGURE 2.23 – Requête $\mathscr{R}$

Cas de base : n=1 l'expression équivalente est '/' .

Étape d'induction : Supposons que chaque requête monadique de $\mathcal{RAR}^{\{\Sigma^*\}}$ de taille n est équivalente à une requête de $XP^{\{[],*,//,fs,ps\}}$ et montrons le pour une requête $\mathscr{R}$ de taille n+1. Soit σ le nœud père du nœud de sélection. La requête $\mathscr{R}$ peut s'écrire sous la forme indiquée en (figure 2.23). D'après la proposition 2, il existe des filtres F_0 ,...,F_k tel que pour tout i ,$\forall$ n $\in \mathcal{D}$, n$\vDash_{\mathcal{D}} \mathscr{T}_i \Leftrightarrow$ n $\vDash_{\mathcal{D}} F_i$.
Soit la requête $\mathscr{R}'$=($\mathscr{T}'$,σ) où $\mathscr{T}'$ est le template entouré par la ligne en pointillés en figure 2.23, la taille de cette requête $\mathscr{R}'$ est inférieur à n donc d'après l'hypothèse de récurrence $\mathscr{R}'$ est exprimable par une requête P_0 de $XP^{\{[],*,//,fs,ps\}}$.
Alors $\mathscr{R}$ est exprimable par l'expression :
$P_0/child :: \gamma_i[H_i][ps :: \gamma_{i-1}[H_{i-1}][...[ps :: \gamma_0[H_0]]...][fs :: \gamma_{i+1}[H_{i+1}][...[fs :: \gamma_k[H_k]]...]$
avec

$$\gamma_i = c_i \; et \; H_i = F_i \; si \; c_i \in \Sigma$$
$$\gamma_i = * \; et \; H_i = descendant\_or\_self :: *[F_i] \; si \; c_i = \Sigma^*$$

□

4 Les requêtes $\mathcal{RARs}$ versus CXPath

Nous montrons dans cette section que les unions finies de requêtes $\mathcal{RARs}$ monadiques constituent un langage qui n'est pas comparable avec le fragment positif de $CXPath$ ($CXPath^+$).

Proposition 3. $CXPath^+ \nsubseteq \cup \; \mathcal{RARs}$

tel-00580926, version 1 - 29 Mar 2011

Démonstration. Pour montrer cette proposition, nous exhibons une requête de $CXPath^+$ qui n'est pas exprimable comme une union finie de requêtes $\mathcal{RARS}$.
Soit $\mathscr{R}$ la requête exprimée par le chemin suivant de $CXPath^+$:
$/child :: a/(child :: b[./child :: d/\ and\ ./child :: c])^+/child :: f$.

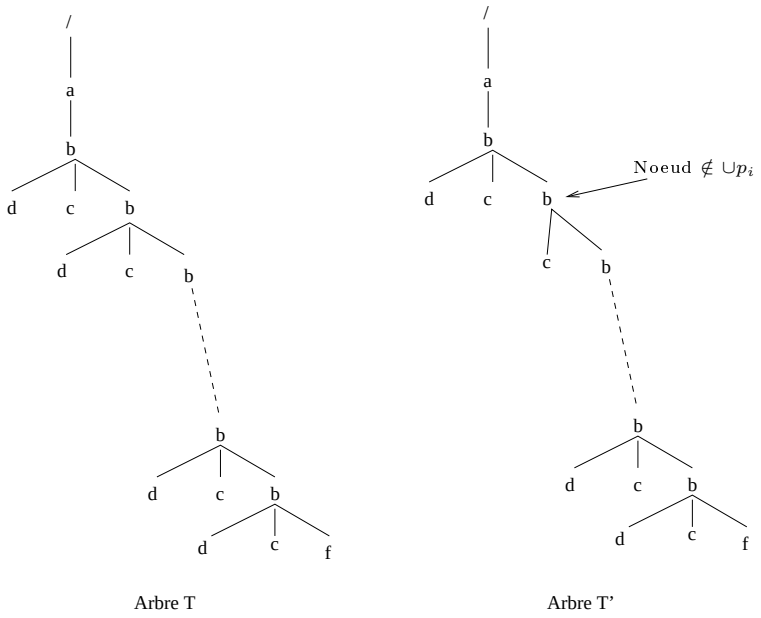


FIGURE 2.24 –

Supposons qu'il existe une union finie de requêtes $\mathcal{RARS}$ $\mathscr{R}' = \cup_i \mathscr{R}_i$ équivalente à $\mathscr{R}$. Soit k_i le nombre maximum de nœuds de $\mathscr{R}_i$ ayant une arité supérieure strictement à 1 et figurant sur un même chemin et soit N un entier supérieur strictement à Sup$\{k_i$, i $\in$ [1,n] $\}$. Considérons l'arbre T de la figure 2.24 dans lequel on suppose que le nombre de nœuds étiquetés par b et sélectionnés par $/child :: a/(child :: b[./child :: d/\ and\ ./child :: c])^+$ est supérieur strictement à N.
La feuille f est sélectionnée par la requête $\mathscr{R}$ et doit donc l'être par la requête $\mathscr{R}'$. Il existe donc i $\in$ [1,n] et un plongement p_i de $\mathscr{R}_i$ dans T, sélectionnant la feuille f.
D'autre part, par construction, il existe un nœud ν de T étiqueté par b qui n'est l'image par le plongement p_i d'aucun nœud de $\mathscr{R}_i$, d'arité >1 (Figure 2.24). Nous modifions T en supprimant le fils d du nœud ν et notons T' l'arbre ainsi modifié. La feuille f dans T' est toujours sélectionnée par $\mathscr{R}'$ car la modification n'a pas altéré le plongement p_i, mais cette feuille f n'est plus sélectionnée par la requête $\mathscr{R}$. Ceci montre donc que $\mathscr{R}$ n'est pas exprimable sous la forme d'une union de $\mathcal{RARS}$. □

Proposition 4. $\cup\ \mathcal{RAR} \not\subseteq CXPath^+$

Démonstration. Tout d'abord rappelons la définition des langages de mots sans

tel-00580926, version 1 - 29 Mar 2011

étoile : les langages de mots sans étoile sont les langages obtenus en utilisant uniquement les opérations booléennes (union, intersection et complément) et le produit (la concaténation) mais pas l'étoile.
Soit la requête arbre régulière $\mathscr{R}$ de la figure 2.25. Montrons que $R \notin CXPath^+$: Le langage (aa)* est un langage qui n'est pas sans étoile [DG08] . Il n'est donc pas définissable par une formule de FO_{tree}. Or $CXPath$ a la même expressivité que FO_{tree} (section 1.3 chapitre 2). Donc $\mathscr{R}$ ne peut être exprimée par une requête de CXPath. □

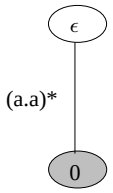


FIGURE 2.25 – requête $\mathscr{R}$

5 Les requêtes $\mathcal{RAR}s$ versus $FOREG$

Proposition 5. *Soit $\mathscr{T}$ un template et $\mathcal{D}$ un document semi-structuré , alors il existe une formule $\phi_{\mathscr{T}}$ de $FOREG$ tel que $\forall\ n \in \mathcal{D}$, $n \vDash_{\mathcal{D}} \mathscr{T} \Leftrightarrow \mathcal{D} \models \phi_{\mathscr{T}}(n)$.*

Démonstration. On procède par récurrence sur la taille (égale au nombre de nœuds) de $\mathscr{T}$.

cas de base :
$\mathscr{T} = (\epsilon = [\mathcal{E} \to 0];)$. On définit la formule $\phi_{\mathscr{T}}$ de $FOREG$ comme suit : $\forall$ n $\in$ $\mathcal{D}$, $\mathcal{D} \models \phi_{\mathscr{T}}(n) \Leftrightarrow \exists y[P]^{\downarrow}_{s,t}(n,y)$ où P est l'expression régulière définie de manière inductive sur la structure de l'expression régulière $\mathcal{E}$ par la fonction ***Reg*** dans l'algorithme 2 donné ci-après.

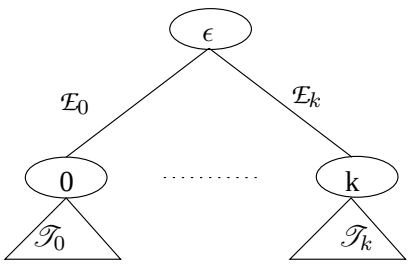


FIGURE 2.26 – Template $\mathscr{T}$

tel-00580926, version 1 - 29 Mar 2011

Algorithm 2 Algorithm fonction ***Reg***

Require: une expression régulière $\mathcal{E}$
sortie : une expression régulière P qui vérifie :
$\forall$ n $\in \mathcal{D}$, n $\vDash_{\mathcal{D}} \mathscr{T} \Leftrightarrow \exists y\ [P]^{\downarrow}_{s,t}(n,y)$ avec $\mathscr{T} = (\epsilon = [\mathcal{E} \rightarrow 0];)$
function *Reg*($\mathcal{E}$)
begin
if $\mathcal{E}$ est une lettre a **then**
$P \leftarrow \theta_a$ avec $\theta_a(s,t) = O_a(t)$
end if
if $\mathcal{E} = \mathcal{E}_1|\mathcal{E}_2$ **then**
$P \leftarrow$ ***Reg***($\mathcal{E}_1$)| ***Reg***($\mathcal{E}_2$)
end if
if $\mathcal{E} = \mathcal{E}_1.\mathcal{E}_2$ **then**
$P \leftarrow$***Reg***($\mathcal{E}_1$).***Reg***($\mathcal{E}_2$)
end if
if $\mathcal{E} = \mathcal{E}_0^*$ **then**
$P \leftarrow$ (***Reg***($\mathcal{E}_0$))*
end if
return P
end

Etape d'induction : Soit un template $\mathscr{T}$ de taille (n+1), $\mathscr{T}$ s'écrit sous la forme présentée en figure 2.26.

Si k=0 : $\mathscr{T}_0$ étant de taille $<$ (n+1), il existe d'après l'hypothèse de récurrence une formule $\phi_{\mathscr{T}_0}$ de $FOREG$ vérifiant : $\forall$ n $\in \mathcal{D}$, n $\vDash_{\mathcal{D}} \mathscr{T}_0 \Leftrightarrow \mathcal{D} \models \phi_{\mathscr{T}_0}(n)$
La formule $\phi_{\mathscr{T}}$ définie par : $\phi_{\mathscr{T}}$(s)= $\exists y\ [$***Reg***($\mathcal{E}_0$)$]^{\downarrow}_{s,t}(s,y) \wedge \phi_{\mathscr{T}_0}(y)$ satisfait clairement :
$\forall$ n $\in \mathcal{D}$, n $\vDash_{\mathcal{D}} \mathscr{T} \Leftrightarrow \mathcal{D} \models \phi_{\mathscr{T}}(n)$

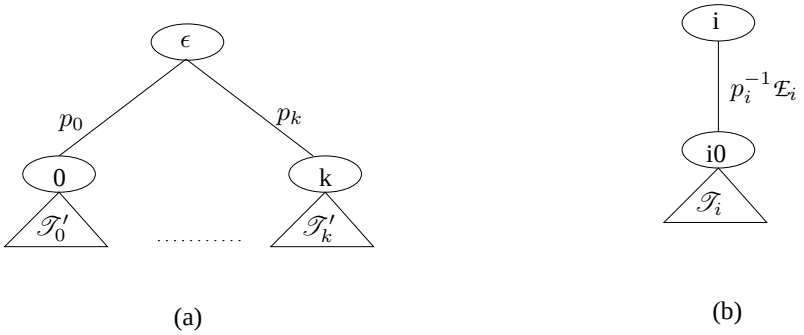


FIGURE 2.27 – (a) Template $\mathscr{T}_{\vec{p}}$, (b) Template $\mathscr{T}'_i$

tel-00580926, version 1 - 29 Mar 2011

Si k≥1, soit $\vec{p}$=$(p_0, p_1, ..., p_k)$ un k-uplet de lettres, constitué de tous les préfixes possibles de longueur 1 des langages L($\mathcal{E}_0$),L($\mathcal{E}_1$),...,L($\mathcal{E}_k$). On note $\mathscr{T}_{\vec{p}}$ le template représenté en figure 2.27 (a), où ∀ i =0,...,k $\mathscr{T}_i'$ désigne le template schématisé en figure 2.27 (b) et est formellement défini par $\mathscr{T}_i'$= $\text{Insert}^{v}_{(p_i^{-1}{}_{\mathcal{E}_i}, i)}(\mathscr{T}_i)$.
Comme k≥1, la taille de $\mathscr{T}_i'$ est inférieure strictement à (n+1) pour tout i=0,...,k donc d'après l'hypothèse de récurrence il existe des formules de $FOREG$ $\phi_{\mathscr{T}_0'}$,..., $\phi_{\mathscr{T}_k'}$ qui vérifient : pour tout i ∈ [0,k], ∀ n ∈ $\mathcal{D}$, n $\vDash_{\mathcal{D}}$ $\mathscr{T}_i'$ ⇔ $\mathcal{D} \models \phi_{\mathscr{T}_i'}(n)$.
L'expression $\phi_{\vec{p}}$ de $FOREG$ qui vérifie ∀ n ∈ $\mathcal{D}$, n $\vDash_{\mathcal{D}}$ $\mathscr{T}_{\vec{p}}$ ⇔ $\phi_{\vec{p}}(n)$ est donnée par : $\exists y_0, ..., y_k[(true)^*(s = y_0)O_{p_0}(s)(true)^*...(s = y_i)O_{p_i}(s)...(true)^*(s = y_k)O_{p_k}(s)(true)^*]^{\rightarrow}_s(n)$ $\wedge\, \phi_{\mathscr{T}_0'}(y_0) \wedge \phi_{\mathscr{T}_1'}(y_1) \wedge ... \wedge \phi_{\mathscr{T}_k'}(y_k)$.
Clairement le template $\mathscr{T}$ est équivalent à l'union finie des templates $\mathscr{T}_{\vec{p}}$ lorsque $\vec{p}$ prend toutes les valeurs possibles.
Donc si on pose $\phi_{\mathscr{T}} = \bigvee_{\vec{p}} \phi_{\vec{p}}$ on a :∀ n ∈ $\mathcal{D}$, n $\vDash_{\mathcal{D}}$ $\mathscr{T}$ ⇔ $\mathcal{D} \models \phi_{\mathscr{T}}(n)$.

□

Théorème 8. *Soit $\mathscr{R} = (\mathscr{T}, s)$ une requête $\mathcal{RAR}$ monadique. Il existe une formule $\phi_{\mathscr{R}}$ de $FOREG$ tel que* $\forall \mathcal{D}\ Eval_{\mathcal{D}}(\mathscr{R})$=$\{\ n \in \mathcal{D}\ /\ \mathcal{D} \vDash \phi_{\mathscr{R}}(n)\}$

Démonstration. On démontre le théorème par récurrence sur le nombre de nœuds de $\mathscr{T}$.

Cas de base : $\mathscr{R} = (\mathscr{T}, s)$ avec $\mathscr{T} = (\epsilon = [\mathcal{E} \rightarrow 0];)$. Soit la formule de $FOREG$ définie par $\phi_{\mathscr{T}}(n) = \exists y\ [\textbf{\textit{Reg}}(\mathcal{E})]^{\downarrow}_{s,t}(n, y)$ où ***Reg*** est la fonction de l'algorithme 2

Si s=ϵ alors $Eval_{\mathcal{D}}(\mathscr{R})$=$\{\ n \in \mathcal{D}\ /\ \mathcal{D} \vDash \phi_{\mathscr{T}}(n) \wedge$ n=root$\}$.
Si s=0 alors $Eval_{\mathcal{D}}(\mathscr{R})$=$\{\ n \in \mathcal{D}\ /\ [\textbf{\textit{Reg}}(\mathcal{E})]^{\downarrow}_{s,t}(root, n)\}$.

Étape d'induction : Supposons que pour toute requête monadique $\mathcal{RAR}$ de taille n il existe une formule $\phi_{\mathscr{R}}$ de $FOREG$ tel que $\forall \mathcal{D}\ Eval_{\mathcal{D}}(\mathscr{R})$=$\{\ n \in \mathcal{D}\ /\ \mathcal{D} \vDash \phi_{\mathscr{R}}(n)\}$, et montrons que ce résultat est vrai pour une requête $\mathscr{R} = (\mathscr{T}, s)$ de n+1 nœuds.

Cas s=ϵ : $\phi_{\mathscr{R}}(n) = \phi_{\mathscr{T}}(n) \wedge (n = root)$
Cas s≠ ϵ : soit σ le nœud père de s.

si $\sigma = \epsilon$ (voir figure 2.28), on définit $\phi_{\mathscr{R}}(n)$=$[\textbf{\textit{Reg}}(\mathcal{E})]^{\downarrow}_{s,t}(root, n) \wedge \phi_{\mathscr{T}}(n)$.

Si $\sigma \neq \epsilon$, la requête $\mathscr{R}$ peut s'écrire sous la forme présentée en figure 2.29 où σ est le nœud père du nœud de sélection s : On note par $\mathscr{T}'$ (voir figure 2.29) le template construit à partir de $\mathscr{T}$ en supprimant le sous template $\mathcal{U}$ issu du nœud σ. Soit la requête $\mathscr{R}'$=($\mathscr{T}'$,σ) : la taille de cette requête $\mathscr{R}'$ est inférieure à n donc d'après l'hypothèse de récurrence, il existe une formule $\phi_{\mathscr{R}'}$ de $FOREG$ tel que $\forall \mathcal{D}$ $Eval_{\mathcal{D}}(\mathscr{R}')$=$\{\ n \in \mathcal{D}\ /\ \mathcal{D} \vDash \phi_{\mathscr{R}'}(n)\}$.
D'autre part comme pour la proposition 5, la requête $\mathscr{R}$ est équivalente à l'union des requêtes $\mathscr{R}_{\vec{p}}$ pour tous les k-uplet $\vec{p} = (p_0, ..., p_k)$ possibles de préfixes de longueur

tel-00580926, version 1 - 29 Mar 2011

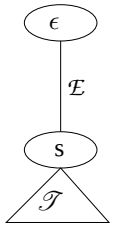


FIGURE 2.28 – Requête $\mathscr{R}$

1 des expressions régulières $\mathcal{E}_0, .., \mathcal{E}_k$, où $\mathscr{R}_{\vec{p}}$ est représentée en figure 2.30. Pour chaque requête $\mathscr{R}_{\vec{p}}$, les templates $\mathscr{T}_i'$ ont la même définition que celles données dans la proposition 5.

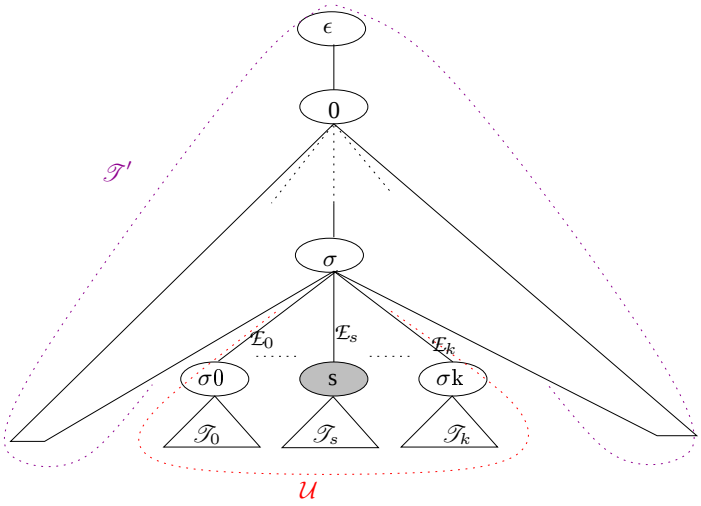


FIGURE 2.29 – Requête $\mathscr{R}$

D'après la proposition 5, il existe des formules de $FOREG$ $\phi_{\mathscr{T}_i'}$,$\phi_{\mathscr{T}_s}$, $\forall$ i $\in [0,k] \setminus$ {s} tel que pour tout i, $\forall$ n $\in \mathcal{D}$, n$\vDash_{\mathcal{D}}$ $\mathscr{T}_i' \Leftrightarrow \mathcal{D} \models \phi_{\mathscr{T}_i'}(n)$ et n$\vDash_{\mathcal{D}}$ $\mathscr{T}_s \Leftrightarrow \mathcal{D} \models \phi_{\mathscr{T}_s}(n)$.

La formule $\phi_{\vec{p}}$ qui vérifie $\forall \mathcal{D}$ $Eval_{\mathcal{D}}(\mathscr{R}_{\vec{p}})$=$\{$ $n \in \mathcal{D}$ / $\mathcal{D} \vDash \phi_{\vec{p}}(n)\}$ est donnée par :
$\phi_{\vec{p}}$(s) = $\exists y \phi_{\mathscr{R}'}(y) \wedge \exists y_0, ..., y_s, ..., y_k [(true)^*(l = y_0)O_{p_0}(l)...(l = y_s)O_{p_s}(l)...(l = y_k)O_{p_k}(l)(true)^*]_l^{\rightarrow}(y) \wedge$
$\phi_{\mathscr{T}_0'}(y_0) \wedge ..., \phi_{\mathscr{T}_{s-1}'}(y_{s-1}) \wedge \phi_{\mathscr{T}_{s+1}'}(y_{s+1}), ... \wedge \phi_{\mathscr{T}_k'}(y_k) \wedge [\boldsymbol{Reg}(p_s^{-1}\mathcal{E}_s)]_{l,t}^{\downarrow}(y_0, s) \wedge \phi_{\mathscr{T}_s'}(s)$
Enfin $\phi_{\mathscr{R}}$ est donnée par $\phi_{\mathscr{R}} = \bigvee_{\vec{p}} \phi_{\vec{p}}$. □

tel-00580926, version 1 - 29 Mar 2011

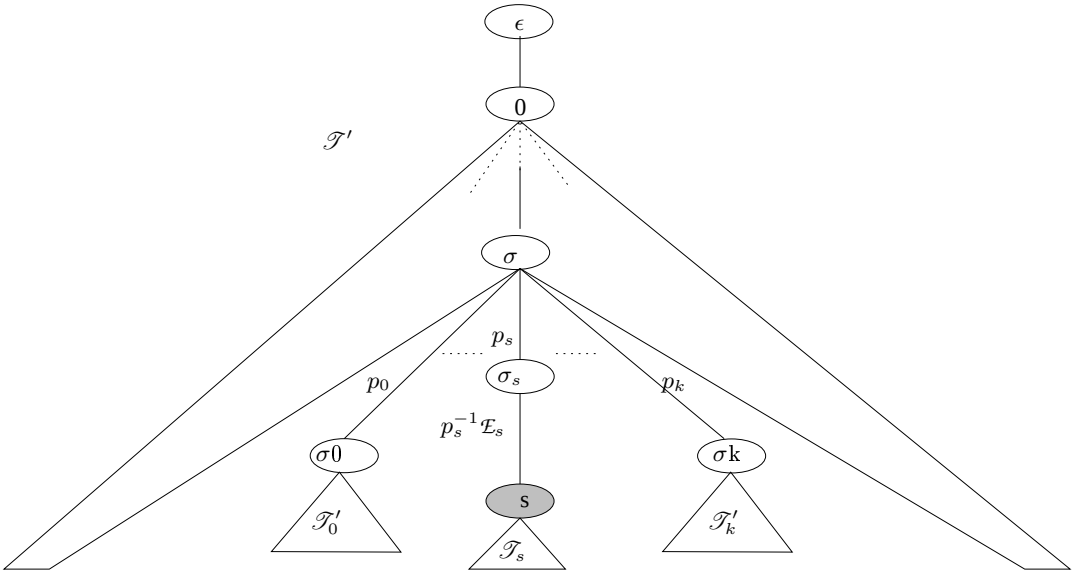


FIGURE 2.30 – Requête $\mathscr{R}_{\vec{p}}$

6 Les requêtes $\mathcal{RAR}$s versus RegularXPath

Proposition 6. *Soit $\mathscr{T}$ un template et $\mathcal{D}$ un document semi-structuré , alors il existe une expression de chemin P de RegularXPath tel que $\forall\ n \in \mathcal{D}$, $n \vDash_{\mathcal{D}} \mathscr{T} \Leftrightarrow n \in [\![< P >]\!]$.*

Théorème 9. *Toute requête RAR monadique $\mathscr{R} = (\mathscr{T}, s)$ est exprimable par une expression de chemin de RegularXPath.*

Démonstration. On procède par récurrence sur la taille (égale au nombre de nœuds) de $\mathscr{T}$.
La preuve suit le même raisonnement que celle dans le cas *FOREG*, la preuve complète est donnée dans l'annexe. □

7 Bilan

Les requêtes arbres régulières constituent un formalisme indépendant de tout standard pour la sélection de nœuds dans les données semi structurées. Leur puissance provient de leur structure arborescente, proche du modèle arbre des documents XML, et des expressions régulières qu'elles utilisent et qui constituent un outil puissant pour la navigation arborescente. Ces requêtes $\mathcal{RAR}$s ne sont pas comparables avec le langage XPath standard du W3C. Plusieurs fragments de XPath ont été définis pour simplifier l'étude de ce langage que ce soit pour la mise à jour, l'évaluation ou l'inclusion de requêtes.

tel-00580926, version 1 - 29 Mar 2011

On a montré que :
(1)- toute requête positive du fragment navigationnel de XPath ($CoreXPath^{+}$) peut être exprimée par une union finie de requêtes $\mathcal{RARS}$.
(2)- toute requête monadique $\mathcal{RAR}$, est exprimable par une formule de $FOREG$.

$CoreXPath$ n'étant pas FO complet, d'autres extensions ont été définis pour augmenter sa puissance d'expressivité. $CXPath$ est une extension de $CoreXPath$ qui est FO complet, par contre son fragment positif n'est pas comparable avec les requêtes $\mathcal{RARS}$. RegularXPath est une autre extension plus expressive que $CXPath$ mais qui est incluse strictement dans la logique monadique du second ordre. Pour trouver une équivalence logique de RegularXPath, les travaux de Cate [tS08] définissent l'opérateur W qui permet d'évaluer une expression sur une partie restreinte (sous-arbre) du document global. Ainsi il montre que le langage $RegularXPath^{W}$ a la même puissance d'expression que FO+TC1 et définit aussi des automates d'arbre NTWA (Nested tree walking automata) qui ont la même puissance d'expression. Un automate cheminant (TWA) se déplace dans l'arbre d'un nœud à un autre en suivant les arêtes. Il choisit son déplacement en fonction de l'étiquette du nœud courant, de son état courant et du type du nœud courant. Un NTWA est une extension des TWA où les transitions dépendent du fait qu'un sous automate trouve ou non un calcul sur le sous arbre issu du nœud courant.
Un résumé de la comparaison entre $\mathcal{RARS}$ et tous les langages cités auparavant est illustré par la figure 2.31.

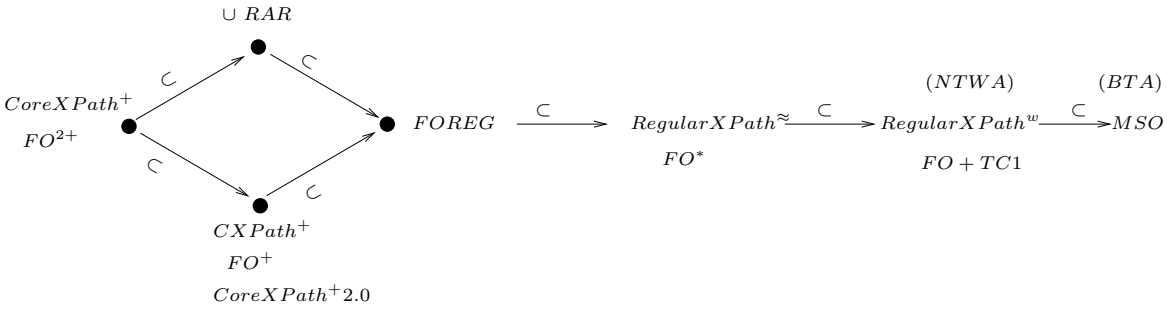


FIGURE 2.31 – Schéma de comparaison

tel-00580926, version 1 - 29 Mar 2011

«Vise toujours à la brièveté ; brève est la route de la nature, et c'est la manière de tout faire et de tout dire le plus raisonnablement possible.»

MARC-AURÈLE Pensées / Les Stoïciens

tel-00580926, version 1 - 29 Mar 2011

tel-00580926, version 1 - 29 Mar 2011

Chapitre 3

Analyse de dépendances entre Vues et Mises à jour

1 Introduction

Dans ce chapitre, nous nous intéressons au problème de l'impact d'une mise à jour sur une vue définie sur des données semi-structurées. Une vue permet de présenter partiellement les informations issues d'un ou plusieurs documents sources afin de répondre à des besoins de personnalisation ou de confidentialité. Lorsque les données sources sont volumineuses, le coût de l'évaluation de la vue peut être élevé et il est donc important de ne pas recalculer la vue si cela ne s'avère pas nécessaire. Le problème de l'impact consiste précisément à détecter si, après une mise à jour des données sources, le résultat de l'évaluation de la requête de vue risque ou non d'être modifié. Dans le cas où l'on peut *a priori* obtenir une réponse négative à cette question, une nouvelle évaluation de la vue peut être évitée et par conséquent son coût économisé.

Ce problème, classique dans le cadre des bases de données relationnelles [BLT86, BCL89, GL95, GMS93, GM95, Vis96] a été étudié dans le cadre des bases de données objet [AFP03, SLT91] et également pour des données semi-structurées [AMR+98, LD00, OCMH03, DESR03, EsWDR02, QCR00, STP+05, Nil06, DNSL05, GRS07, RS06, BC09, ZGM98].
La grande majorité de ces travaux traite ce problème en adoptant une approche dans laquelle les documents sources ainsi que les vues matérialisées sont disponibles et les méthodes mises en œuvre utilisent ces documents sources. Pourtant, dans le cas où ces informations sont inaccessibles, une analyse statique s'appuyant uniquement sur la définition des requêtes de vues et celles des mises à jour, devient nécessaire pour détecter un éventuel impact.

Dans la suite, nous adoptons ce point de vue, selon lequel nous ne disposons pas *a priori* des documents sources mais seulement de la requête de vue $\mathcal{V}$ et de la mise

à jour $\mathcal{C}$. Nous abordons le problème en faisant les choix suivants :
(1) Nous modélisons la mise à jour sous la forme d'une requête $\mathcal{C}$, permettant de sélectionner les nœuds qui seront modifiés. Dans la mesure où nous ne précisons pas non plus la façon dont les nœuds sélectionnés par $\mathcal{C}$ seront modifiés, $\mathcal{C}$ représente en fait une classe de mises à jour possibles, celle de toutes les mises à jour ne modifiant que les nœuds sélectionnés par $\mathcal{C}$. L'énoncé du problème que nous étudions est alors le suivant : "Étant données une vue $\mathcal{V}$ et une classe $\mathcal{C}$ de mises à jour, déterminer si la vue $\mathcal{V}$ est indépendante de $\mathcal{C}$, c'est-à-dire si toute mise à jour q de $\mathcal{C}$ n'a aucun impact sur l'évaluation de $\mathcal{V}$, quel que soit le document source". Dans le cas où un schéma $\mathcal{S}c$ contraignant les documents sources est connu, on peut espérer que la connaissance de $\mathcal{S}c$ permette de détecter un nombre plus élevé de cas d'indépendance qu'en l'absence de schéma. Nous étudions donc également la variante suivante du problème précédent : "Étant donnés une vue $\mathcal{V}$, une classe $\mathcal{C}$ de mises à jour et un schéma $\mathcal{S}c$, déterminer si la vue $\mathcal{V}$ est indépendante de $\mathcal{C}$ dans le contexte $\mathcal{S}c$, c'est-à-dire si toute mise à jour q de $\mathcal{C}$ n'a aucun impact sur l'évaluation de $\mathcal{V}$, quel que soit le document source $\mathcal{D}$, valide par rapport à $\mathcal{S}c$, sur lequel $\mathcal{V}$ est évaluée".

(2) Nous exprimons les requêtes de sélection correspondant à la vue $\mathcal{V}$ et à la mise à jour $\mathcal{C}$, par des requêtes arbres régulières $\mathcal{RAR}s$. Ce choix a pour avantage de positionner notre étude dans un cadre formel rigoureux indépendant de tout standard. Nous avons, dans le chapitre 2, étudié avec précision, le pouvoir d'expression de ce formalisme et montré qu'il permet notamment d'exprimer toutes les requêtes de $CoreXPath^{+}$. Ainsi l'ensemble des résultats de ce chapitre 3 s'appliquent en particulier aux cas où $\mathcal{V}$ et $\mathcal{C}$ sont exprimées sous forme de requêtes positives de $CoreXPath$.

Notre principale contribution est de fournir une condition suffisante pour qu'une vue $\mathcal{V}$ soit indépendante d'une classe de mise à jour $\mathcal{C}$ dans le contexte $\mathcal{S}c$. Cette condition est évaluable en temps polynomial par rapport à la taille des entrées $\mathcal{V}$, $\mathcal{C}$ et $\mathcal{S}c$. Nous montrons aussi que le problème de l'indépendance de $\mathcal{V}$ par rapport à $\mathcal{C}$ est en général PSPACE-difficile. Enfin nous montrons que pour des sous-classes particulières de requêtes de vues $\mathcal{V}$ et de classes de mises à jour $\mathcal{C}$, le problème d'indépendance devient polynomial.

Dans ce chapitre, les données semi-structurées (DSS) considérées sont des documents XML qui seront modélisés par des arbres ordonnés étiquetés sur un alphabet à arités variables Σ comme présentés dans le chapitre 1. Dans ce chapitre, les valeurs textuelles des éléments XML et les attributs sont ignorées (la fonction val est ignorée), seuls les éléments définissant la structure du document sont représentés par les nœuds de l'arbre. Nous donnons un exemple de représentation d'un document en figure 3.1. Formellement, un document $\mathcal{D}$ est un couple $\mathcal{D}=(d, \lambda)$ où d est un domaine d'arbre, et λ associe une étiquette $a \in \Sigma$ à chaque nœud w de d.

tel-00580926, version 1 - 29 Mar 2011

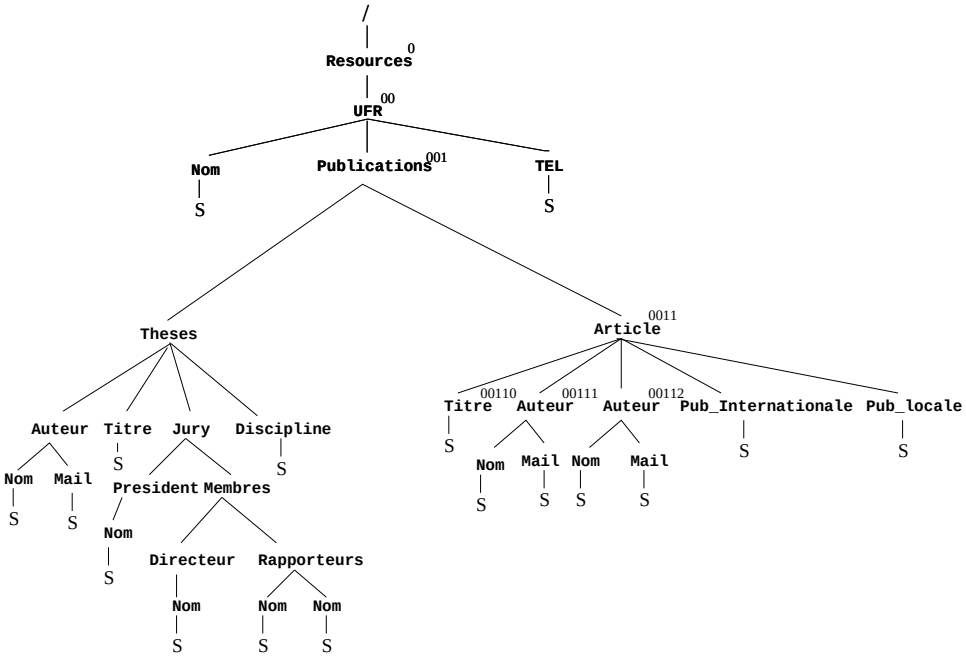


FIGURE 3.1 – Un document semi-structuré $\mathcal{D}$

2 État de l'art

2.1 Optimisation et réécriture de requêtes

De nombreux travaux existent dans la littérature mais traitent plutôt de l'optimisation de requêtes : [BÖB+04] utilisent des vues XPath matérialisées pour optimiser l'évaluation de requêtes XML et proposent un algorithme d'appariement entre ces vues matérialisées et les sous-expressions XPath d'une requête XML afin de déterminer si de telles vues peuvent être utilisées pour accélérer l'évaluation de la requête ; dans le cadre du problème général de réécriture d'une requête à partir d'une vue, [LWZ06] étudient, lorsque vue et requête sont des *tree pattern*, l'existence et la construction d'une requête '*maximale*' réécrite à partir de la vue et contenue dans la requête ; dans [ABMP07], le problème de la réécriture de requêtes à partir de vues est également étudié en présence de contraintes structurelles et de contraintes d'intégrité et en supposant que les requêtes et les vues sont exprimées dans un formalisme de *tree pattern* plus riche que celui utilisé dans [LWZ06] ; dans [SV05] la réécriture d'une requête à partir de vues est étudiée sous l'angle de la complétude du langage de réécriture c'est-à-dire sa capacité à permettre la réécriture de requêtes dont les évaluations sont déterminées par celles des vues.

tel-00580926, version 1 - 29 Mar 2011

2.2 La mise à jour incrémentale des vues utilisant les sources

Parmi les travaux existants, on peut citer quelques approches :

- L'approche [AMR+98] utilise le modèle OEM et le langage Lorel basé sur l'algèbre LORE pour les requêtes, l'algorithme proposé utilise les identifiants internes des objets du document pour effectuer la mise à jour de la vue.

- Ce travail [LD00] utilise l'architecture WHAX (WareHouse Architecture for XML) qui est une architecture permettant de définir et maintenir des vues à partir de données hiérarchiques semi-structurées et à partir de données relationnelles. Dans ce modèle, le langage utilisé pour la définition des vues est WHAX-QL, une variante du langage XML-QL.

- L'approche [EsWDR02, EsMS06] considère le langage XQuery et utilise des approches algébriques qui permettent de s'affranchir de la syntaxe du langage de requêtes considéré, basé principalement sur l'algèbre XAT développée dans le projet RAINBOW [ZZRR02]. Les requêtes XQuery sont décomposées en arbre algébrique. L'arbre est constitué des différents opérateurs XAT qui composent la requête.

- [OCMH03] : ce travail s'intéresse aux vues définies grâce au couple XPath/XSLT. Cette technique prend en compte l'ordonnancement des résultats de l'évaluation de la vue et se base sur des schémas de numérotation qui ne sont pas des schéma robustes vis à vis des mises à jour et qui nécessitent parfois une renumérotation.

- [STP+05,STP+06] : ces travaux portent également sur des vues XPath avec comme langage d'interrogation XSLT. Par contre, la source et la vue sont indépendantes ("loose-coupled") l'une de l'autre. L'approche repose sur la détection de mises à jour pertinentes par rapport à un calcul d'inclusion entre chemins.

- L'approche [Nil06] étudie la maintenance autonome de vues XQuery en se basant sur une technique d'inclusion et de ré-écriture de requêtes. Dans cette approche, les vues sont exprimées à l'aide d'expressions XQuery.

- L'approche [San07,San08] se situe dans une architecture de médiation. Ces travaux se fondent sur une extension de la XAlgèbre permettant une annotation par identifiant des données, ainsi que sur un procédé de reconstitution partielle des sources. Le langage de requêtes utilisé pour définir les vues est XQuery.

tel-00580926, version 1 - 29 Mar 2011

2.3 La détection d'indépendance

Commutativité et analyse de chemin [GRS07] : Dans ces travaux, les auteurs s'intéressent au problème de commutativité de deux mises à jour, à des fins d'optimisation. Les mises à jour sont exprimées en utilisant le langage XQuery privé des fonctions récursives, n'utilisant que les axes child, descendant, parent et ancestor et augmenté par les opérations "Insert" et "Delete".
La principale contribution des auteurs est de proposer une condition suffisante assurant la commutativité de deux mises à jour exprimées par deux expressions EX_1 et EX_2 de XQueryU.
Cette problématique est proche de la notre dans la mesure où intuitivement deux expressions EX_1 et EX_2 commutent si et seulement si elles respectent une forme d'indépendance l'une vis à vis de l'autre. L'idée est d'associer à chaque expression EX de XQueryU, trois chemins notés r, a, u et définissant respectivement une borne supérieure de l'évaluation de EX (pour r), l'ensemble des nœuds accédés lors de l'évaluation de EX (pour a), et l'ensemble des nœuds mis à jour par EX (pour u). Une analyse statique au cours de laquelle est réalisée une comparaison des chemins $<a_1,u_1>$ associés à EX_1 et $<a_2,u_2>$ associés à EX_2, permet d'exhiber la condition suffisante assurant la commutativité de EX_1 et EX_2.

Conflit de *tree pattern* [RS06] : Dans ce travail les auteurs étudient le problème de la détection de conflits entre des opérations de mises à jour et des vues monadiques spécifiées dans le langage des *tree pattern* $P^{\{*,//,[]\}}$ introduit par [MS04] (on a vu au chapitre 2 section 1.5 que $P^{\{*,//,[]\}}$ est équivalent aux expressions $CoreXPath$ n'utilisant que les axes child et descendant, le symbole $*$ et les filtres). Les opérations de mises à jour envisagées sont de deux types "Insert" et "Delete" : elles sont spécifiés par des expressions de la forme $\text{Insert}_{p,X}$ et "Delete_p" où p est un tree pattern de $P^{\{*,//,[]\}}$ et X un arbre à insérer. Dans les deux cas le tree pattern p détermine l'ensemble des nœuds à supprimer pour "Delete_p", ou bien en dessous desquels l'arbre X doit être inséré pour $\text{Insert}_{p,X}$. Les auteurs introduisent d'autre part deux sémantiques dans l'analyse du conflit entre la vue et l'opération "Insert" ou "Delete" :
- Le conflit de nœuds qui suppose que la vue renvoie un ensemble de nœuds.
- Le conflit d'arbres qui suppose que la vue renvoie un ensemble de sous-arbres.
Leur analyse se restreint au cas du conflit de nœuds mais les auteurs affirment que la démarche peut être étendue au cas du conflit d'arbres.
Il est montré que le problème de la détection de conflit de nœuds est NP-complet. Par contre, un algorithme polynomial est exhibé pour la détection de conflit dans le cas particulier des vues spécifiées par des *tree pattern* linéaires (et les nœuds de sélection sont des feuilles). Dans le cas de l'opération "Delete" l'algorithme se base sur la correspondance de pattern (pattern maching). Pour le cas "Insert", quand l'arbre inséré X est non connu, l'algorithme utilisé est dérivé de celui utilisé dans le cas de l'opération "Delete". Par contre quand l'arbre inséré X est connu, l'al-

tel-00580926, version 1 - 29 Mar 2011

gorithme est modifié pour prendre en compte l'information spécifique à X afin de trouver d'avantage de cas de non conflit.

Indépendance par les schémas [BC09] : Dans cette approche, les auteurs n'utilisent aucune information sur les chemins mais se basent sur les types définis dans le schéma contraignant les données. L'analyse statique suppose donc la connaissance d'un schéma (DTD spécialisée). L'approche est inspirée de l'approche définie dans [GRS07], La différence majeure par rapport à celle de [GRS07] est : (i) Le langage utilisé pour les mises à jour est la recommandation du W3C XQuery update facility (ii) les règles d'inférence ne renvoient pas des nœuds (ou les chemins définissant ces nœuds) mais des noms de types définis dans le schéma.
Ces règles d'inférence, qui sont une simplification des règles standards d'inférence de types de XQuery, calculent l'ensemble des types possibles des nœuds retournés par la requête. Ainsi ces règles agissent comme une fonction qui permet de dériver un ensemble de types à partir d'un environnement statique, un schéma et une requête. L'indépendance est détectée par comparaison des types inférés par les requêtes de la vue et de mise à jour.
Les travaux de [BCU10] s'inspirent de ceux de [BC09] et proposent également une autre approche basée sur l'inférence de types.

2.4 Comparaison et Positionnement

Notons tout d'abord que, même s'il en est proche, le problème de la réécriture de requêtes en utilisant des vues (travaux de la section 2.1) n'est pas le même que celui de l'indépendance de requêtes de vues et de mises à jour.
Dans l'ensemble des approches présentées dans les sections 2.1 et 2.2 , les auteurs supposent : (i) soit que la source est disponible, (ii) soit qu'il est possible de stocker une copie de la source ou des informations intermédiaires pour surveiller l'évolution de la source, puis ces informations sont exploitées pour mettre les vues à jour. Au contraire, dans notre approche, nous n'utilisons que les spécifications des requêtes de vue et de mises à jour et éventuellement un schéma structurel s'il est disponible pour faire une analyse d'indépendance.
Les approches données dans la section 2.3 sont très similaires à la nôtre. Notre approche diffère de celles utilisées dans tous ces travaux, principalement par le choix, comme langage de requêtes, du modèle formel des requêtes arbres régulières, qui impacte aussi bien les résultats obtenus que les techniques mises en œuvre.
Le langage de requête choisi est très souvent XPath, XQuery ou un formalisme de *tree pattern* tandis que nous optons ici pour un langage plus abstrait basé sur la notion de requête arbre régulière ; bien qu'incomparable avec XPath, ce langage permet cependant d'exprimer toutes les requêtes positives de $CoreXPath$ ainsi que des requêtes non exprimables en XPath. Pour traiter l'indépendance, ces travaux ainsi que notre approche ignorent les attributs et les valeurs atomiques. D'autre part dans

tel-00580926, version 1 - 29 Mar 2011

la majorité de ces travaux, les requêtes utilisées sont des requêtes monadiques et les nœuds de sélection sont des feuilles, alors que les requêtes *RARs* sont des requêtes n-aires dont les nœuds de sélection ne sont pas obligatoirement des feuilles.

3 Modélisation des vues et des mises à jour par les *RARs*

3.1 Concepts de vue

Une vue est une présentation partielle et réorganisée d'un ensemble de données sources. Elle représente de cette façon une sorte d'intermédiaire entre la base de données et l'utilisateur. Cela a de nombreux avantages :

- *Confidentialité & Sécurité :* masquer la structure des données sources et interdire l'accès à une partie des données sources.
- *Personnalisation :* regrouper les informations pertinentes pour des utilisateurs spécifiques.
- *Optimisation :* simplifier l'expression de requêtes longues et complexes et optimiser aussi le temps de réponse des requêtes si les documents sources sont de trés grande taille.

Le résultat d'une requête de vue, lorsqu'il est stocké, est appelé une vue matérialisée.

Dans le cas de documents XML l'exécution d'une vue se compose principalement de deux étapes : la première étape consiste à extraire du (ou des) document(s) source(s) un ensemble de nœuds pertinents contenant les informations souhaitées ; la seconde étape réorganise le résultat obtenu lors de la première étape pour lui donner la structure personnalisée attendue.
Ainsi on peut modéliser une vue $\mathcal{V}$ par la composition de deux applications : $\mathcal{V} = h \circ t$, où l'application t sélectionne un ensemble de nœuds à extraire des données sources et l'application h procède à la réorganisation de ces nœuds pour obtenir le résultat final. Cette application h ne jouant aucun rôle dans l'étude de l'indépendance d'une vue par rapport à un ensemble de mises à jour, nous assimilerons donc l'exécution d'une vue à sa première étape d'extraction de nœuds à partir des données sources c'est à dire $\mathcal{V} \simeq t$. Plus précisément, nous considérons dans ce travail que (1) les données sources sont constituées d'un unique document XML $\mathcal{D}$ et que (2) une requête de vue n-aire $\mathcal{V}$ exprime les conditions qui doivent être satisfaites par un n-uplet de nœuds de $\mathcal{D}$ pour être sélectionné par $\mathcal{V}$. Nous considérons que l'extraction d'un nœud w de $\mathcal{D}$ retourne le sous-arbre, noté $\mathcal{D}(w)$, issu de w dans $\mathcal{D}$. Ainsi l'exécution d'une requête de vue n-aire $\mathcal{V}$ sur le document $\mathcal{D}$ retourne un ensemble de n-uplets d'arbres de la forme $(\mathcal{D}(w_1), ..., \mathcal{D}(w_n))$ correspondant chacun à l'extraction d'un n-uplet de nœuds $(w_1, ..., w_n)$ sélectionné par $\mathcal{V}$.

tel-00580926, version 1 - 29 Mar 2011

Modélisation des vues par les $\mathcal{RAR}$s : Nous faisons le choix de modéliser les vues par les requêtes arbres régulières. Formellement une vue est représentée par une requête $\mathcal{RAR}$ $\mathcal{V} = (\mathscr{T}, \vec{s})$ où

- $\mathscr{T}$ est le template de $\mathcal{V}$.
- $\vec{s}$=$(w_1, ..., w_n)$ est le n-uplet de nœuds de sélection.

Dans le cas où $\vec{s}$ est composé d'un seul nœud, on parle de requête de vue monadique.

Exemple 1 : Considérons le document semi-structuré $\mathcal{D}$ de la figure 3.1 et la requête de vue binaire $\mathcal{V}_1$ suivante : "Donner les (Titre,Auteur) des articles publiés localement". L'exécution de $\mathcal{V}_1$ sur $\mathcal{D}$ retourne : $\{(t_1, s_1), (t_1, s_2)\}$ avec $t_1 = \mathcal{D}(00110)$, $s_1 = \mathcal{D}(00111)$ et $s_2 = \mathcal{D}(00112)$, c'est à dire les deux couples regroupant chacun le titre de l'article correspondant au nœud 0011 avec le sous arbre issu de l'un de ses deux auteurs.
Cette requête de vue est représentée par la requête arbre régulière de la figure 3.2

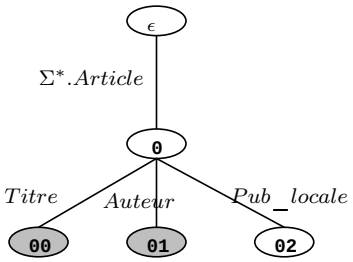


FIGURE 3.2 – $\mathcal{RAR}$ $\mathcal{V}_1$

3.2 Classe de mises à jour

Dans ce travail nous modélisons une mise à jour sur un document XML source $\mathcal{D}$ par une opération qui (1) sélectionne un ensemble de nœuds de $\mathcal{D}$ à mettre à jour et (2) remplace le sous-arbre $\mathcal{D}(w)$ issu de chaque nœud w sélectionné par un nouvel arbre. Ce type de mise à jour modélise un grand nombre de mises à jour, y compris les opérations d'insertion/suppression de sous-arbres dans le document source, si l'on considère ces dernières comme des mises à jour effectuées sur le nœud père de la position d'insertion/suppression. Ainsi une mise à jour q d'un document semi-structuré est définie par la composition, $q = f \circ \mathcal{C}$, de deux applications f et $\mathcal{C}$: l'application $\mathcal{C}$ sélectionne l'ensemble des nœuds w à mettre à jour et l'application f procède à la mise à jour, en remplaçant les sous-arbres $\mathcal{D}(w)$ issus de ces nœuds w par des arbres non vides. Cette application f joue un rôle important dans l'analyse d'indépendance, elle peut par exemple changer la valeur d'une feuille extraite par la vue. Cependant, dans un souci de simplification, nous choisissons ici d'analyser statiquement l'indépendance d'une vue $\mathcal{V}$ par rapport à une mise à jour $q = f \circ \mathcal{C}$ en faisant abstraction de la fonction de remplacement f opérée par q. Ainsi l'analyse

tel-00580926, version 1 - 29 Mar 2011

que nous proposons est plus précisément une analyse d'indépendance d'une vue $\mathcal{V}$ par rapport à un ensemble de mises à jour représenté par $\mathcal{C}$, plutôt que par rapport à une mise à jour q particulière.
L'application $\mathcal{C}$ représente en fait une classe de mises à jour : l'ensemble des mises à jour qui modifient les nœuds sélectionnés par $\mathcal{C}$. Ainsi deux mises à jour q et q' font partie de la même classe de mises à jour si et seulement si elles sont définies à partir de la même application de sélection $\mathcal{C}$.

Modélisation des mises à jour par les $\mathcal{RAR}$s : une classe de mises à jour $\mathcal{C}$, qui est un processus de sélection de nœuds, sera modélisée de la même façon que les vues par les requêtes arbres régulières.
Ainsi une mise à jour est représentée par une requête $\mathcal{RAR}$ $\mathcal{C} = (\mathscr{T}, \overrightarrow{m})$ où

- $\mathscr{T}$ est le template de $\mathcal{C}$.
- $\overrightarrow{m}$=$(m_1, ..., m_n)$ est le n-uplet de nœuds de mise à jour.

Exemple 2 : Soient les mises à jour, (q_1) "Modifier les auteurs d'articles publiés à l'international en y ajoutant un numéro de téléphone (Tel)", et (q_2) "Modifier les auteurs d'articles publiés à l'international en remplaçant l'élément Nom par la séquence des deux éléments (NomFamille, Prénom)". Ces deux mises à jour, q_1 et q_2, appartiennent à la même classe $\mathcal{C}_1$ de mises à jour (figure 3.3) qui sélectionne les auteurs d'articles publiés à l'international pour les modifier.

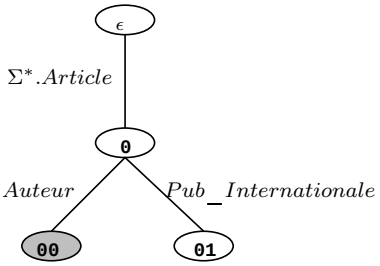


FIGURE 3.3 – Requête arbre régulière $\mathcal{C}_1$

4 Indépendance entre vues et classes de mises à jour

4.1 Impact d'une mise à jour sur une vue

Nous dirons qu'une mise à jour q a un impact sur une vue $\mathcal{V}$ d'un document $\mathcal{D}$ si et seulement si l'évaluation de $\mathcal{V}$ sur $\mathcal{D}$ avant et après la mise à jour q ne produit pas le même résultat, c'est à dire formellement, si et seulement si : $\mathcal{V}(q(\mathcal{D})) \neq \mathcal{V}(\mathcal{D})$.

Exemple 3 : Les requêtes de mise à jour q_1 et q_2 de l'exemple 2 ont clairement un impact sur la requête de vue $\mathcal{V}_1$ de l'exemple 1, puisqu'elles modifient les sous-arbres

tel-00580926, version 1 - 29 Mar 2011

s_1 et s_2 extraits.
Considérons sur le document $\mathcal{D}$ de la figure 3.1, la requête de vue $\mathcal{V}_2$ suivante : "Donner les couples d'éléments (Titre, Nom d'auteurs) des articles publiés localement". Cette requête diffère de la requête $\mathcal{V}_1$ de l'Exemple 1 par le fait que seul le nom de l'auteur est extrait avec le titre de l'article. La mise à jour q_2 de l'Exemple 2 a clairement un impact sur cette vue puisqu'elle supprime les éléments Nom. Par contre la mise à jour q_1 n'a pas d'impact sur le résultat de la vue.

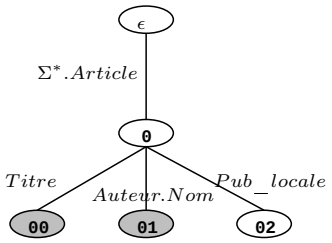


FIGURE 3.4 – Requête arbre régulière $\mathcal{V}_2$

4.2 Indépendance entre vue et classe de mises à jour :

Définition 15. *$\mathcal{V}$ est dite indépendante par rapport à la classe de mises à jour $\mathcal{C}$ si et seulement si pour tout document source $\mathcal{D}$, pour toute mise à jour q de $\mathcal{C}$, q n'a pas d'impact sur $\mathcal{V}$ dans $\mathcal{D}$.*

Notons que la notion d'indépendance utilisée dans ce contexte est différente de celle utilisée dans [LLSV01], qui définit l'indépendance de mise à jour (respectivement de requête) dans le contexte d'un entrepôt de donnée comme le fait que celui ci n'a besoin d'aucune information de la source pour se mettre à jour (respectivement pour répondre à une requête). Notre notion d'indépendance se rapproche plutôt de la notion d'"irrelevance" utilisée dans [BLT86,BCL89] pour exprimer qu'une requête n'a aucun rapport avec une autre.

Exemple 4 : Soit la classe de mise à jour $\mathcal{C}_2$ suivante (figure 3.5) : "mettre à jour les titres des thèses qui ont au moins deux rapporteurs".
Les deux vues $\mathcal{V}_1$ et $\mathcal{V}_2$ sont clairement indépendantes de cette classe de mise à jour $\mathcal{C}_2$. Par contre, elles ne sont pas indépendantes de la classe de mise à jour $\mathcal{C}_1$ de l'exemple 2, car il existe une mise à jour de $\mathcal{C}_1$ qui impacte la vue dans chaque cas.

4.3 Indépendance dans le contexte d'un schéma :

Dans de nombreux cas, on peut supposer qu'un schéma imposant des contraintes aux données sources est disponible. Cette donnée supplémentaire peut alors permettre d'affiner l'analyse d'indépendance. Notons $\texttt{valide}(\mathcal{S}c)$ l'ensemble des documents

tel-00580926, version 1 - 29 Mar 2011

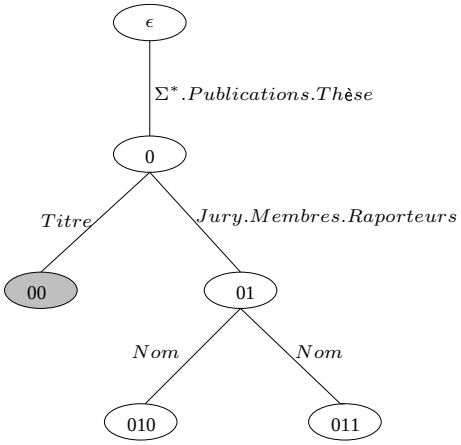


FIGURE 3.5 – Requête arbre régulière $\mathcal{C}_2$

valides par rapport au schéma $\mathcal{S}c$. Dans ce contexte la définition de l'indépendance est modifiée comme suit :

Définition 16. *$\mathcal{V}$ est indépendante d'une classe de mises à jour $\mathcal{C}$ dans le contexte du schéma $\mathcal{S}c$ si et seulement si : pour tout document valide par rapport à $\mathcal{S}c$ (i.e $\mathcal{D} \in$ `valide`$(\mathcal{S}c)$) et pour toute mise à jour q de $\mathcal{C}$ conservant la validité par rapport à $\mathcal{S}c$ (i.e $q(\mathcal{D}) \in$ `valide`$(\mathcal{S}c)$), q n'a pas d'impact sur $\mathcal{V}$ dans $\mathcal{D}$.*

Exemple 5 : Considérons le document semi-structuré $\mathcal{D}_1$ de la figure 3.6 et supposons que ce document soit contraint par le schéma $\mathcal{S}c_1$ donnée par la DTD suivante :

```
<!ELEMENT Resource  UFR >
<!ELEMENT UFR  (Nom, Publications, Tél) >
<!ELEMENT Publications  (Article*) >
<!ELEMENT Article  (Titre, Auteur*, (Pub_locale | Pub_International) ) >
<!ELEMENT Auteur  (Nom, Mail) >
<!ELEMENT Nom (#PCDATA) >
<!ELEMENT Tél (#PCDATA) >
<!ELEMENT Pub_locale (#PCDATA) >
<!ELEMENT Pub_International (#PCDATA) >
<!ELEMENT Titre (#PCDATA) >
<!ELEMENT Mail (#PCDATA) >
<!ATTLIST Article IDA  #REQUIRED >
```

Dans l'exemple 4, on a vu que les deux vues $\mathcal{V}_1$ et $\mathcal{V}_2$ ne sont pas indépendantes de la classe de mise à jour $\mathcal{C}_1$ mais dans le contexte du schéma $\mathcal{S}c_1$ ces deux vues

tel-00580926, version 1 - 29 Mar 2011

sont indépendantes de la classe de mise à jour $\mathcal{C}_1$. En effet, les nœuds sélectionnés par $\mathcal{V}_1$ ou $\mathcal{V}_2$ pour être extraits, concernent des articles publiés dans des conférences locales, et ne sont donc pas concernés par les mises à jour de $\mathcal{C}_1$ qui modifient des articles publiés à l'international. Ceci n'est vrai que dans le contexte du schéma $\mathcal{S}c_1$ qui interdit à un article d'être à la fois publié dans une conférence locale et une autre internationale.

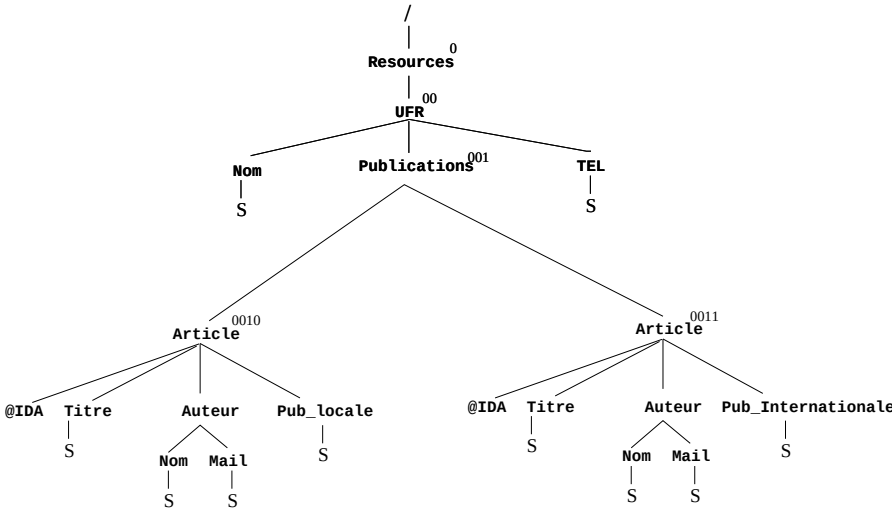


FIGURE 3.6 – Un document semi-structuré $\mathcal{D}_1 \in \texttt{valide}(\mathcal{S}c_1)$

5 Problème d'indépendance

Dans les deux prochaines sections, nous étudions le problème de l'indépendance d'une vue $\mathcal{V}$, modélisée par une requête $\mathcal{RAR}$ $\mathcal{V} = (\mathscr{T}_\mathcal{V}, \vec{s_\mathcal{V}})$, par rapport à une classe de mises à jour $\mathcal{C}$ modélisée par une requête $\mathcal{RAR}$ $\mathcal{C} = (\mathscr{T}_\mathcal{C}, \vec{s_\mathcal{C}})$.
Dans cette section, nous montrons que le problème est en général PSPACE-difficile puis, nous exhibons une condition suffisante assurant l'indépendance de $\mathcal{V}$ par rapport à la classe $\mathcal{C}$. Nous montrons ensuite en Section 5 que cette condition peut être évaluée en temps polynomial.

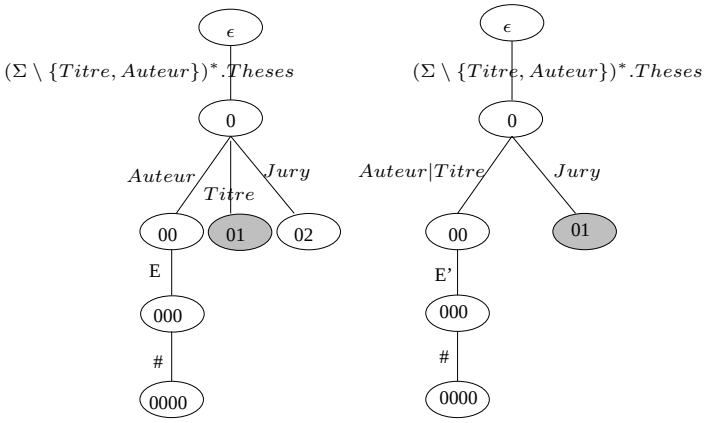
5.1 Un problème PSPACE-difficile

Proposition 7. *Décider si une vue $\mathcal{V} = (\mathscr{T}_\mathcal{V}, \vec{s_\mathcal{V}})$ est indépendante par rapport à une classe de mise à jour $\mathcal{C} = (\mathscr{T}_\mathcal{C}, \vec{s_\mathcal{C}})$ est un problème PSPACE-difficile.*

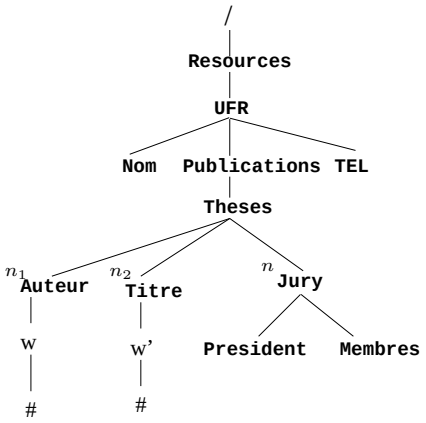
Démonstration. Nous réduisons le problème de l'inclusion d'expressions régulières, qui est PSPACE-difficile [MNSC04], au problème de l'indépendance d'une vue par rapport à une classe de mises à jour. Considérons deux expressions régulières E et

tel-00580926, version 1 - 29 Mar 2011

E' sur Σ. Nous définissons deux requêtes $\mathcal{RAR}s$ monadiques $\mathcal{V}$ et $\mathcal{C}$ comme indiqué sur la figure 3.7 ((a) et (b)) où '#' est une nouvelle étiquette qui n'apparaît ni dans E ni dans E', et nous montrons que $\mathcal{V}$ est dépendante de $\mathcal{C}$ si et seulement si $L(E) \nsubseteq L(E')$.



(a) Classe de mise à jour $\mathcal{C}$ (b) Vue $\mathcal{V}$



(c) *Arbre* $\mathcal{D}_0$

FIGURE 3.7 – Schéma de réduction

Supposons que $\mathcal{V}$ soit dépendante de $\mathcal{C}$. Il existe un document $\mathcal{D}$ et une mise à jour q de $\mathcal{C}$ tels que $\mathcal{V}(q(\mathcal{D})) \neq \mathcal{V}(\mathcal{D})$. Deux cas sont donc possibles : soit (a) un nœud n de $\mathcal{D}$ étiqueté par 'Jury' est extrait par $\mathcal{V}$ ($n \in \mathcal{V}(\mathcal{D})$) mais ne l'est plus après la mise à jour ($n \notin \mathcal{V}(q(\mathcal{D}))$), soit (b) un nouveau nœud n étiqueté par 'Jury' est extrait par $\mathcal{V}$ après la mise à jour ($n \in \mathcal{V}(q(\mathcal{D}))$) alors qu'il ne l'était pas avant ($n \notin \mathcal{V}(\mathcal{D})$). Dans les deux cas, on déduit que le nœud n appartient nécessairement

tel-00580926, version 1 - 29 Mar 2011

à une trace de la requête $\mathcal{C}$ dans $\mathcal{D}$ selon un plongement p. Par conséquent n a deux nœuds frères n_1 et n_2 étiquetés respectivement par 'Auteur' et 'Titre', qui sont les images par p des nœuds 00 et 01 de $\mathcal{C}$. Si $L(E) \subseteq L(E')$, le sous-arbre issu de n_1 remplit alors les conditions imposées par la requête $\mathcal{V}$ pour l'extraction de n, dans $q(\mathcal{D})$ après la mise à jour pour le cas (a), et dans $\mathcal{D}$ avant la mise à jour pour le cas (b). Ceci contredit les faits $n \notin \mathcal{V}(q(\mathcal{D}))$ (cas (a)) et $n \notin \mathcal{V}(\mathcal{D})$ (cas (b)).
Inversement si $L(E) \nsubseteq L(E')$, alors il existe un mot w de $L(E)$ qui n'appartient pas à $L(E')$. Soit d'autre part w' un mot de $L(E')$. Considérons l'arbre $\mathcal{D}_0$ de la figure 3.7 (c) dans lequel n_1, n_2 et n représentent les nœuds étiquetés respectivement par 'Auteur', 'Titre' et 'Jury', et le mot w (respectivement w') coïncide avec la suite des étiquettes rencontrées sur le chemin allant du nœud Auteur (respectivement Titre) au nœud #. Le nœud n sera extrait par $\mathcal{V}$ avant toute mise à jour de $\mathcal{C}$ car w' appartient à $L(E')$. Le nœud n_2 est modifié par toute mise à jour de $\mathcal{C}$ car w appartient à $L(E)$. Considérons maintenant la mise à jour q de $\mathcal{C}$ qui supprime le chemin $w'\#$ du sous-arbre issu de n_2. Le nœud n ne sera plus extrait par $\mathcal{V}$ après la mise à jour q car w n'appartient pas à $L(E')$. Ainsi $\mathcal{V}$ est dépendante de $\mathcal{C}$.

□

5.2 Un critère d'indépendance

Clairement, le caractère PSPACE difficile du problème d'indépendance d'une vue par rapport à une classe de mise à jour, est une conséquence directe de notre choix de représenter les vues et les classes de mises à jour par des requêtes $\mathcal{RARS}$.
Nous montrons cependant dans la suite de ce chapitre, que l'on peut définir un critère suffisant d'indépendance évaluable en temps polynomial. Ainsi la complexité dans le cas général du problème d'indépendance, induite par l'expressivité du modèle des requêtes $\mathcal{RARS}$ choisi, est contrebalancée par la possibilité de détecter en temps polynomial, des situations particulières d'indépendance, sans restreindre l'expressivité apportée par les requêtes $\mathcal{RARS}$.

Dans cette section, nous exhibons un critère d'indépendance d'une vue $\mathcal{V}$ par rapport à une classe de mises à jour $\mathcal{C}$, évaluable en temps polynomial. Nous commençons par une analyse du problème d'indépendance.
Pour des raisons techniques et sans perte de généralité, nous faisons l'hypothèse suivante sur le template $\mathcal{T}_c$ de la requête $\mathcal{C}$: toute arête entrante (w,s_c) dans un nœud s_c de mise à jour est étiqueté par une lettre de Σ. Cette supposition n'est pas restrictive car on peut toujours remplacer $\mathcal{C}$ par une union $\bigcup_{l\in\Sigma}\mathcal{C}_l$ de requêtes vérifiant cette condition, en posant $\mathcal{C}_l = (\mathcal{W}_l, Insert^v_{s_c}(\vec{s_c}))$ avec $\mathcal{W}_l = (Rename_{\mathcal{I}_{(w,s_c)}l^{-1},(w,s_c)}(Insert^v_{l,s_c}(\mathcal{T}_c)))$ et l'indépendance de $\mathcal{V}$ par rapport à $\mathcal{C}$ est équivaut à l'indépendance de $\mathcal{V}$ par rapport à chacune des requêtes $\mathcal{C}_l$.

Selon les définitions de la section précédente, une requête de vue $\mathcal{V}$ est dépen-

tel-00580926, version 1 - 29 Mar 2011

dante d'une classe de mises à jour $\mathcal{C}$ dans le contexte du schéma $\mathcal{S}c$ si et seulement si il existe $\mathcal{D} \in \texttt{valide}(\mathcal{S}c)$, il existe $q \in \mathcal{C}$ vérifiant $q(\mathcal{D}) \in \texttt{valide}(\mathcal{S}c)$ et il existe un n-uplet $\vec{t}$ vérifiant l'une des deux assertions :

(1) $\vec{t} \in \mathcal{V}(\mathcal{D})$ *et* $\vec{t} \notin \mathcal{V}(q(\mathcal{D}))$

ou (2) $\vec{t} \notin \mathcal{V}(\mathcal{D})$ *et* $\vec{t} \in \mathcal{V}(q(\mathcal{D}))$.

- L'assertion (1) signifie qu'un n-uplet $\vec{t}$, extrait avant la mise à jour, ne l'est plus après : il existe donc une trace $\texttt{trace}_p(\mathcal{V}, \mathcal{D})$ de $\mathcal{V}$ dans $\mathcal{D}$ selon un plongement p, et un plongement p' de $\mathcal{C}$ dans $\mathcal{D}$ permettant la mise à jour q, vérifiant l'une des conditions suivantes :
  - Un nœud de $\texttt{trace}_p(\mathcal{V}, \mathcal{D})$ a été modifié par q et les conditions d'extraction de $\vec{t}$ ne sont plus satisfaites (Cas (i) figure 3.9). Ainsi un nœud mis à jour, $p'(s_{\mathcal{C}})$, appartient à $\mathcal{N}(\texttt{trace}_p(\mathcal{V}, \mathcal{D}))$ où $s_{\mathcal{C}}$ est un nœud de $\vec{s_{\mathcal{C}}}$.
  - Un sous-arbre de $\vec{t}$ a été modifié par q et $\vec{t}$ n'apparaît plus dans le résultat de $\mathcal{V}$ (Cas (ii) figure 3.9). Ainsi un nœud mis à jour, $p'(s_{\mathcal{C}})$, appartient à $\mathcal{N}(\mathcal{V}_p(\mathcal{D}))$ où $s_{\mathcal{C}}$ est un nœud de $\vec{s_{\mathcal{C}}}$.

- L'assertion (2) signifie qu'un nouveau n-uplet $\vec{t}$ est extrait après la mise à jour. Ainsi la mise à jour par q d'un nœud $p'(s_{\mathcal{C}})$ de $\mathcal{D}$, où $s_{\mathcal{C}}$ est un nœud de $\vec{s_{\mathcal{C}}}$, a créé une trace $\texttt{trace}_p(\mathcal{V}, q(\mathcal{D}))$ de $\mathcal{V}$ dans $q(\mathcal{D})$ selon un plongement p, permettant l'extraction de $\vec{t}$ (Cas (iii) figure 3.9). Dans ce cas, $p'(s_{\mathcal{C}})$ appartient à $\mathcal{N}(\texttt{trace}_p(\mathcal{V}, q(\mathcal{D})))$

Les assertions (1) et (2) impliquent chacune l'existence d'au moins un arbre ($\mathcal{D}$ pour l'assertion (1) et q($\mathcal{D}$) pour l'assertion (2)) satisfaisant des propriétés particulières. Nous définissons formellement dans la suite le langage $\mathcal{L}$ des arbres satisfaisant ces propriétés, et nous montrons (Proposition 8) que la vacuité de $\mathcal{L}$ est une condition suffisante pour l'indépendance de $\mathcal{V}$ par rapport à $\mathcal{C}$ dans le contexte du schéma $\mathcal{S}c$.

Nous précisons tout d'abord quelque notions et notations que nous utilisons pour définir le langage $\mathcal{L}$.

- Si $\vec{t} = (\mathcal{D}_1, ..., \mathcal{D}_n)$ est un n-uplet de sous-arbres, on note $\mathcal{N}(\vec{t})$ l'union des nœuds de ces sous-arbres : $\mathcal{N}(\vec{t}) = \cup_{i=1}^{n} \mathcal{N}(\mathcal{D}_i)$.

Définition 17. *(Plongement/Trace partielle) Rappelons qu'un plongement p d'une requête $\mathcal{RAR}$ $\mathscr{R}$ dans un document semi-structuré $\mathcal{D}$ est une fonction injective totale de N (ensemble des nœuds de $\mathscr{R}$) dans $\mathcal{N}(\mathcal{D})$ préservant l'ordre.*
Si $\mathscr{R} = (\mathscr{T}, \vec{s})$, on note $\widehat{\mathscr{T}}$ le template obtenu à partir de $\mathscr{T}$ en supprimant tous les descendants des nœuds de $\vec{s}$ et en étiquetant par Σ toutes les arêtes entrantes (w, s) dans un nœud de sélection s de $\vec{s}$.
On appelle alors "plongement partiel" de $\mathscr{R}$ dans un document $\mathcal{D}$ tout plongement $\hat{p}$ de la requête $\widehat{\mathscr{R}} = (\widehat{\mathscr{T}}, \vec{s})$ dans $\mathcal{D}$. La trace, $\texttt{trace}_{\hat{p}}(\widehat{\mathscr{R}}, \mathcal{D})$, de $\widehat{\mathscr{R}}$ dans $\mathcal{D}$ selon $\hat{p}$ est appelée "trace partielle" de $\mathscr{R}$ dans $\mathcal{D}$ selon $\hat{p}$.

tel-00580926, version 1 - 29 Mar 2011

Remarquons que tout plongement p de $\mathscr{R}$ dans $\mathcal{D}$ réalise un plongement de $\widehat{\mathscr{R}}$ dans $\mathcal{D}$ que l'on notera partiel(p). Dans ce cas, la trace partielle de $\mathscr{R}$ dans $\mathcal{D}$ selon partiel(p) est appelée "trace partielle de $\mathscr{R}$ dans $\mathcal{D}$ associée à p."

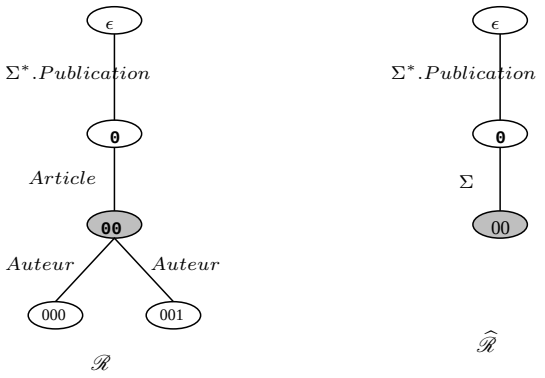


FIGURE 3.8 –

Définition 18. *Soit $\mathcal{L}$ l'ensemble des arbres $\mathcal{D}$ vérifiant :*
(i) $\mathcal{D} \in \texttt{valide}(\mathcal{S}c)$
(ii) il existe une trace $\texttt{trace}_p(\mathcal{V}, \mathcal{D})$ de $\mathcal{V}$ dans $\mathcal{D}$ selon un plongement p, il existe une trace partielle $\texttt{trace}_{p'}(\widehat{\mathcal{C}}, \mathcal{D})$ de $\mathcal{C}$ dans $\mathcal{D}$ selon un plongement partiel p' et il existe un nœud $s_\mathcal{C}$ de $\vec{s_\mathcal{C}}$ tel que $p'(s_\mathcal{C}) \in \mathcal{N}(\texttt{trace}_p(\mathcal{V}, \mathcal{D})) \cup \mathcal{N}(\mathcal{V}_p(\mathcal{D}))$.

Nous pouvons maintenant déduire de cette analyse statique une condition suffisante pour l'indépendance d'une vue $\mathcal{V}$ par rapport à une classe de mises à jour :

Proposition 8. *Si $\mathcal{L} = \varnothing$ alors $\mathcal{V}$ est indépendante de $\mathcal{C}$ dans le contexte $\mathcal{S}c$.*

Démonstration. Supposons que $\mathcal{V}$ soit dépendante de $\mathcal{C}$ dans le contexte $\mathcal{S}c$. L'assertion (1) ou (2) est donc satisfaite.
Si l'assertion (1) ($\vec{t} \in \mathcal{V}(\mathcal{D})$ et $\vec{t} \notin \mathcal{V}(q(\mathcal{D}))$) est satisfaite, nous avons vu qu'il existe un plongement p' de $\mathcal{C}$ dans $\mathcal{D}$, un plongement p de $\mathcal{V}$ dans $\mathcal{D}$, et un nœud $s_\mathcal{C}$ de $\vec{s_\mathcal{C}}$ vérifiant : $p'(s_\mathcal{C}) \in \mathcal{N}(\texttt{trace}_p(\mathcal{V}, \mathcal{D})) \cup \mathcal{N}(\mathcal{V}_p(\mathcal{D}))$. Or p' est également un plongement partiel de $\mathcal{C}$ dans $\mathcal{D}$. Le langage $\mathcal{L}$ est donc non vide.
Si l'assertion (2) ($\vec{t} \notin \mathcal{V}(\mathcal{D})$ et $\vec{t} \in \mathcal{V}(q(\mathcal{D}))$) est satisfaite, nous avons vu qu'il existe $\mathcal{D} \in \texttt{valide}(\mathcal{S}c)$, une mise à jour q dans $\mathcal{C}$ qui vérifie $q(\mathcal{D}) \in \texttt{valide}(\mathcal{S}c)$, un plongement p' de $\mathcal{C}$ dans $\mathcal{D}$ permettant la mise à jour q, un plongement p de $\mathcal{V}$ dans $q(\mathcal{D})$ permettant l'extraction de $\vec{t}$ après la mise à jour q et un nœud $s_\mathcal{C}$ de $\vec{s_\mathcal{C}}$ tel que $p'(s_\mathcal{C})$ appartient à $\mathcal{N}(\texttt{trace}_p(\mathcal{V}, q(\mathcal{D})))$. Or le plongement $partiel(p')$ de $\mathcal{C}$ dans $\mathcal{D}$ subsiste dans $q(\mathcal{D})$ puisque seuls les nœuds de mise à jour et leurs descendants sont modifiés par q : formellement, il existe un plongement partiel p'' de $\mathcal{C}$ dans $q(\mathcal{D})$ dont l'image dans $q(\mathcal{D})$ coïncide avec celle de $partiel(p')$ dans $\mathcal{D}$ et vérifie donc

tel-00580926, version 1 - 29 Mar 2011

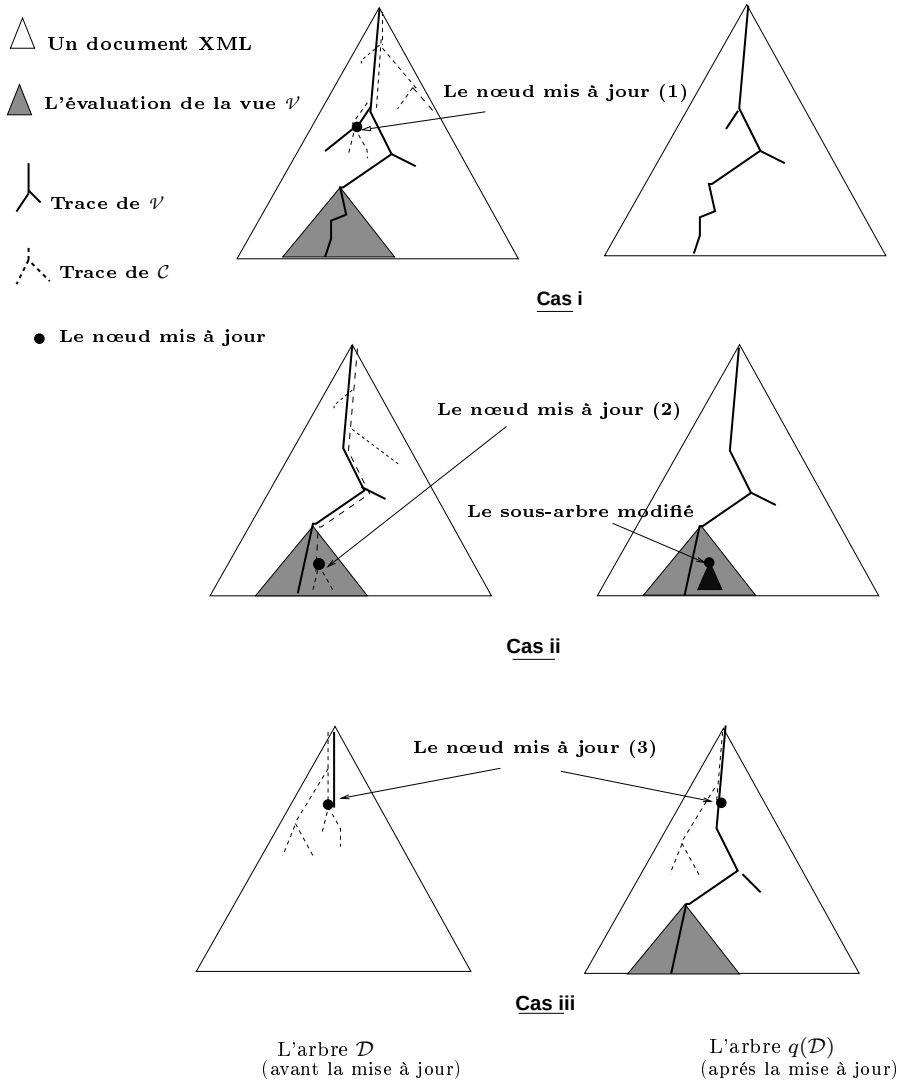


FIGURE 3.9 – Analyse de l'impact d'une mise à jour sur une vue

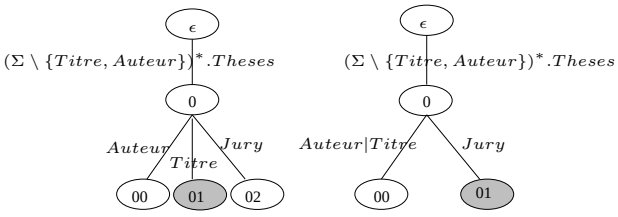
$partiel(p')(s_{\mathcal{C}}) = p''(s_{\mathcal{C}})$.
Ainsi $p''(s_{\mathcal{C}}) \in \mathcal{N}(\texttt{trace}_p(\mathcal{V}, q(\mathcal{D})))$ et $q(\mathcal{D})$ satisfait les propriétés des arbres de $\mathcal{L}$. Le langage $\mathcal{L}$ est donc non vide.

□

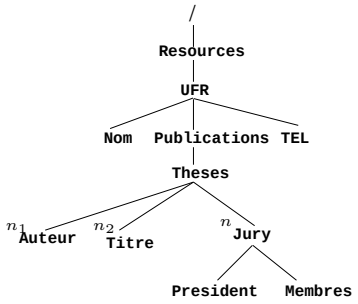
Remarque La vacuité du langage $\mathcal{L}$ n'est pas une condition nécessaire pour que la requête de vue $\mathcal{V}$ soit indépendante de la classe de mises à jour $\mathcal{C}$ dans le contexte $\mathcal{S}c$. Reprenons les requêtes $\mathcal{V}$ et $\mathcal{C}$ de la figure 3.7 ainsi que l'arbre $\mathcal{D}_0$, et supprimons, dans $\mathcal{V}$ et $\mathcal{C}$, les sous-arbres issus du nœud 00 et, dans $\mathcal{D}_0$, les sous-arbres issus des nœuds n_1 et n_2 (voir figure 3.10), l'arbre $\mathcal{D}_0$ appartient à $\mathcal{L}$ donc $\mathcal{L}$ est non vide mais on peut facilement vérifier l'indépendance de $\mathcal{V}$ par rapport à $\mathcal{C}$ puisque la

tel-00580926, version 1 - 29 Mar 2011

mise à jour n'affecte que les titres et ne s'effectue que si le nœud auteur existe. Ainsi ce dernier nœud assure la sélection du sous arbre enraciné par le nœud jury même après la mise à jour.



(a) Classe de mise à jour $\mathcal{C}$ (b)Vue $\mathcal{V}$



(c) *Arbre* $\mathcal{D}_0$

FIGURE 3.10 – $\mathcal{L} = \varnothing$, condition non nécessaire d'indépendance

5.3 Condition nécessaire et suffisante d'indépendance

Dans cette section, nous montrons qu'il existe des cas particuliers où le critère énoncé en Proposition 8 devient une condition nécessaire et suffisante d'indépendance de la vue par rapport à la classe de mises à jour. Il s'agit des cas où :
(i) les nœuds de sélection de la requête de vue $\mathcal{V} = (\mathscr{T}_v, \vec{s}_v)$ vérifient la propriété $\mathcal{P}$: "tout chemin de la racine à une feuille de $\mathscr{T}_v$ contient au moins un nœud de $\vec{s}_v$ ".
(ii) et les nœuds de mise à jour sont tous des feuilles.
Nous précisons ci-dessous quelques notations.

Notations :
- On note VP l'ensemble des vues $\mathcal{V}$ définies par les requêtes arbres régulières $(\mathscr{T}_v, \vec{s}_v)$ vérifiant la propriété $\mathcal{P}$.
- On note CF l'ensemble des classes de mises à jour dont tous les nœuds de mise à jour sont des feuilles.
Une propriété importante des vues $\mathcal{V}$ de VP est que tous les nœuds feuilles sont

tel-00580926, version 1 - 29 Mar 2011

soit des nœuds de sélection, soit des descendants de nœuds de sélection.

Exemple 6 : Soit la requête de vue $\mathcal{V}$ suivante figure 3.11 : "Donner les couples (Auteur, Membres) des thèses ayant au moins deux rapporteurs".
Cette vue appartient à l'ensemble VP car elle vérifiant la propriété $\mathcal{P}$.

Et soit la classe de mise à jour $\mathcal{C}$ de la figure 3.12. Cette classe appartient à l'ensemble CF car le nœud de mise à jour est une feuille.

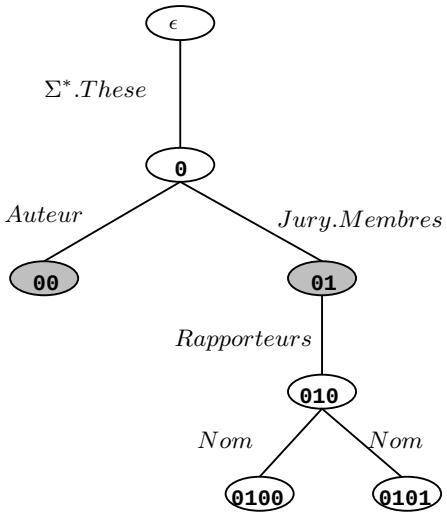


FIGURE 3.11 – Une vue de VP

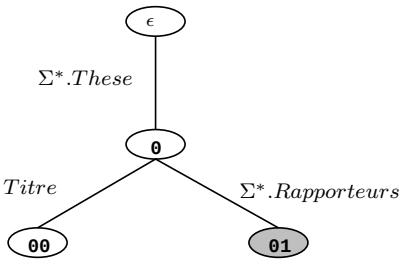


FIGURE 3.12 – Une classe de CF

Dans le cas particulier, des requêtes de vue de VP et des classes de mises à jour de CF, on démontre que la vacuité du langage $\mathcal{L}$ est une condition nécessaire et suffisante pour l'indépendance. Dans l'exemple 6, le langage $\mathcal{L}$ est non vide car il est clair qu'il existe un document $\mathcal{D}$ dans lequel l'image du nœud de mise à jour appartient à un sous arbre extrait par l'évaluation de $\mathcal{V}$; et on peut effectivement vérifier que $\mathcal{V}$ et $\mathcal{C}$ sont dépendantes : la mise à jour de $\mathcal{C}$ qui supprime les noms de tous les rapporteurs d'une certaine thèse, a un impact sur l'évaluation de la vue $\mathcal{V}$.

tel-00580926, version 1 - 29 Mar 2011

Proposition 9. *Soit $\mathcal{V} = (\mathscr{T}_\mathcal{V}, \vec{s_\mathcal{V}})$ une requête de vue de VP, $\mathcal{C} = (\mathscr{T}_\mathcal{C}, \vec{s_\mathcal{C}})$ une classe de mises à jour de CF. $\mathcal{V}$ est indépendante de la classe $\mathcal{C}$ si et seulement si $\mathcal{L}$ est vide.*

Démonstration. D'après la Proposition 8, il suffit de montrer que la vacuité du langage $\mathcal{L}$ est une condition nécessaire d'indépendance. Supposons donc $\mathcal{L} \neq \emptyset$ et soit $\mathcal{T}$ un arbre de $\mathcal{L}$. Nous montrons que $\mathcal{T}$ est un arbre témoin de la dépendance de $\mathcal{V}$ par rapport à $\mathcal{C}$. Puisque $\mathcal{T} \in \mathcal{L}$, il existe dans $\mathcal{T}$ une trace $\texttt{trace}_p(\mathcal{V}, \mathcal{D})$ de $\mathcal{V}$ selon un plongement p et il existe une trace partielle $\texttt{trace}_{p'}(\widehat{\mathcal{C}}, \mathcal{D})$ de $\mathcal{C}$ selon un plongement partiel p', dont l'image d'un nœud mis à jour $n = p'(s_\mathcal{C})$, avec $s_\mathcal{C}$ de $\vec{s_\mathcal{C}}$, appartient soit à $\mathcal{N}(\texttt{trace}_p(\mathcal{V}, \mathcal{D}))$ (cas (a) de la figure 3.13) soit à $\mathcal{N}(\mathcal{V}_p(\mathcal{T}))$ (cas (b) de la figure 3.13). Comme $\mathcal{V}$ appartient à VP, il existe une relation d'ascendance ou de descendance entre le nœud n mis à jour et un nœud m sélectionné par $\mathcal{V}$. Dans les deux cas (a) et (b), il est alors possible d'exhiber une mise à jour q de $\mathcal{C}$ ajoutant par exemple une nouvelle feuille au sous-arbre $\mathcal{D}(n)$ de telle sorte que le sous-arbre $\mathcal{D}(m)$ soit également modifié. Le résultat de la vue est donc modifié par q. On en déduit la dépendance de la vue $\mathcal{V}$ par rapport à la classe de mises à jour $\mathcal{C}$.

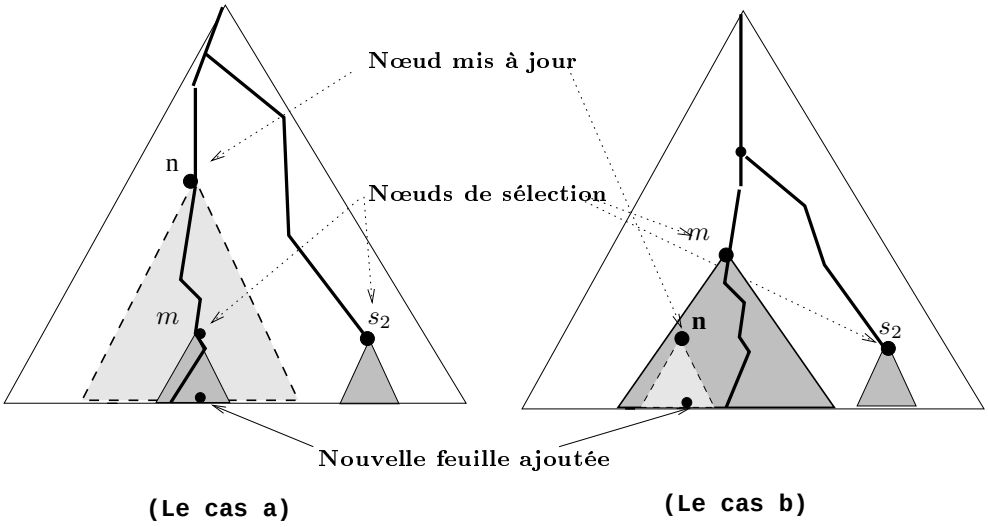


FIGURE 3.13 – Cas des vues de VP et des classes de mises à jour de CF

□

6 Vérification du critère d'indépendance

Dans cette section, nous montrons que la vérification du critère d'indépendance, $\mathcal{L} = \emptyset$, est décidable en temps polynomial par rapport aux tailles de $\mathcal{V}$, $\mathcal{C}$ et $\mathcal{S}c$. Pour cela, nous montrons que $\mathcal{L}$ est un langage régulier d'arbres reconnaissable par

tel-00580926, version 1 - 29 Mar 2011

un automate $\mathcal{A}$ dont la taille est polynomiale en les tailles de $\mathcal{V}$, $\mathcal{C}$ et $\mathcal{S}c$. Le résultat découle alors du fait que le test de vacuité d'un langage régulier d'arbres est décidable en temps polynomial en la taille d'un automate reconnaissant ce langage. Nous commençons en section 5.1 par construire un automate reconnaissant l'ensemble des arbres contenant une trace d'une requête $\mathcal{RAR}$ $\mathscr{R}$, nous donnons ensuite en section 5.2, la construction de l'automate $\mathcal{A}$, et terminons enfin en étudiant la complexité des constructions précédentes.

6.1 Automate reconnaissant une trace

Soit la requête arbre régulière $\mathscr{R} = (\mathscr{T}, \vec{s})$ avec $\mathscr{T} = (\Sigma, N, M, \mathcal{E})$, nous définissons un automate $\mathcal{A}_{\mathscr{R}} = (\Sigma, Q, \delta, F)$ (où Q est l'ensemble d'états, δ la fonction de transition et F l'ensemble des états finaux), qui reconnaît l'ensemble des arbres contenant une trace de $\mathscr{R}$.

L'automate $\mathcal{A}_{\mathscr{R}}$ est construit à partir des automates finis de mots associés aux expressions régulières apparaissant dans l'arbre template $\mathscr{T}$: pour chaque expression régulière $\mathcal{E}_e$ (où $e = (w, w')$ est un arc de M), nous notons $\widetilde{L(\mathcal{E}_e)}$ le langage rationnel formé des mots miroirs de mots de $L(\mathcal{E}_e)$. L'idée intuitive est de construire $\mathcal{A}_{\mathscr{R}}$ de telle sorte que, étant donnés un arbre $\mathcal{D}$ et un plongement p de $\mathscr{R}$ dans $\mathcal{D}$ (voir figure 3.14), un calcul *ascendant* de $\mathcal{A}_{\mathscr{R}}$ sur $\mathcal{D}$ réalise, sur la suite d'étiquettes $\widetilde{\sigma(e)} = \sigma_1\sigma_2...\sigma_{n-1}$ rencontrées lors du parcours *ascendant* du chemin P_e, une simulation d'un calcul d'un automate de mots reconnaissant le langage $\widetilde{L(\mathcal{E}_e)}$.
Pour chaque arc $e = (w, w')$ de M, on suppose donné un automate de mots finis $\mathcal{A}_e = (\Sigma, Q_e, \delta_e, t_e^0, f_e)$ reconnaissant $\widetilde{L(\mathcal{E}_e)}$. Sans perte de généralité, on suppose également satisfaites les trois propriétés suivantes :

(i) les ensembles $\{Q_e, e \in M\}$ sont disjoints deux à deux

(ii) $\mathcal{A}_e$ a un unique état initial t_e^0 et un unique état final f_e

(iii) l'ensemble R_e des états de Q_e accessibles à partir de t_e^0 par une seule transition de δ_e ne sont accessibles à partir d'aucun autre état de Q_e.

Remarquons que le principe de simulation des automates $\mathcal{A}_e$ par $\mathcal{A}_{\mathscr{R}}$ et la propriété (iii) satisfaite par chacun de ces automates, impliquent que les nœuds $p(w')$, images par p du sommet w' d'un arc $e = (w, w')$ de M, sont les seuls nœuds associés à un état de R_e, par au moins un calcul réussi de $\mathcal{A}_{\mathscr{R}}$. Ainsi, les nœuds de $\mathcal{D}$ associés par p à un nœud de sélection de $\mathscr{R}$ (i.e. à un nœud de $\mathcal{N}(\vec{s})$) sont les seuls nœuds associés, par un calcul réussi de $\mathcal{A}_{\mathscr{R}}$, à un état de l'ensemble $Select(\mathcal{A}_{\mathscr{R}}) = \bigcup_{e=(w,w') \in M / w' \in \mathcal{N}(\vec{s})} R_e$.

Nous donnons en Section 6.1 la construction formelle de $\mathcal{A}_{\mathscr{R}}$.

tel-00580926, version 1 - 29 Mar 2011

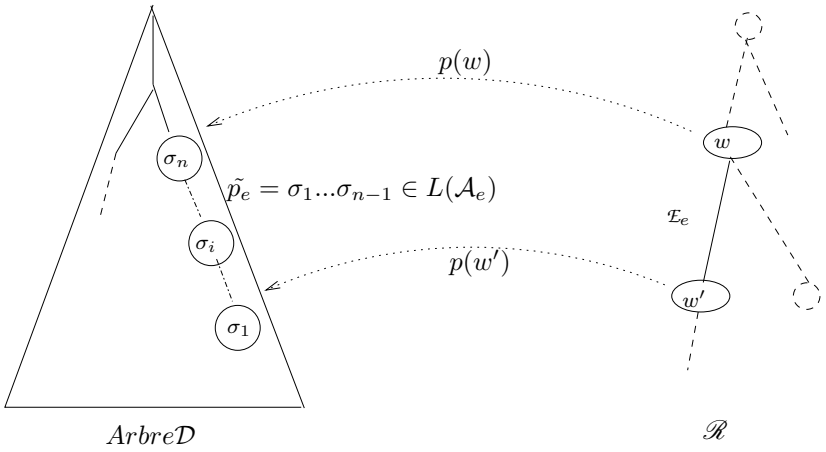


FIGURE 3.14 – Principe de construction de $\mathcal{A}_{\mathscr{R}}$

Construction de $\mathcal{A}_{\mathscr{R}} = (\Sigma, Q, \delta, F)$

Nous notons par τ la trace de $\mathscr{R} = (\mathscr{T}, \vec{s_{\mathscr{R}}})$ dans $\mathcal{D}$ qui doit être identifiée par un calcul réussi de $\mathcal{A}_{\mathscr{R}}$, et p son plongement associé.
L'ensemble des états est défini par : $Q = \cup_{e \in M} Q_e \cup \{f, g\}$ où f et g sont deux états n'apparaissant pas dans $\cup_{e \in M} Q_e$. L'état f joue le rôle d'état final, i.e. $F = \{f\}$, et l'état g est un état générique qui sera associé aux nœuds n'appartenant pas à la trace τ de $\mathscr{R}$ en cours de reconnaissance.
L'ensemble des transitions δ est donné en définissant ci-dessous, pour chaque étiquette x de Σ et état t de Q, l'ensemble $L(x,t) = \{w \in Q^* / (x,t,w) \in \delta\}$:

- $L(x,g) = g^*$. Cet ensemble de transitions permet à l'automate $\mathcal{A}_{\mathscr{R}}$ d'associer l'état générique g, à un nœud w et à ses fils, dans le cas où w n'appartient pas à la trace τ en cours de reconnaissance (Transition 1 de la figure 3.15).
- $L(/, f) = g^* f_{e_1} g^*$ si $Out(\varepsilon) = \{e_1\}$ dans $\mathscr{T}$, $L(x,f) = \emptyset$ pour tout $x \neq' /'$. Cet ensemble de transitions permet l'arrêt d'un calcul réussi de $\mathcal{A}_{\mathscr{R}}$.
- Si $t \in Q_e$ pour un arc e de M, $L(x,t)$ est l'union de trois ensembles de transitions, $L(x,t) = L_1(x,t) \cup L_2(x,t) \cup L_3(x,t)$, définis comme suit :
  - L'ensemble $L_1(x,t)$ n'est non vide que si $e = (w, w')$ avec w' feuille de $\mathscr{T}$. Cet ensemble de transitions permet à $\mathcal{A}_{\mathscr{R}}$ de démarrer un calcul de $\mathcal{A}_e$, à partir de l'image par p d'une feuille de $\mathscr{T}$, afin de reconnaître le mot miroir $\widetilde{\sigma(e)}$ (Transition 2 de la figure 3.15).

$$L_1(x,t) = \begin{cases} g^* & \text{si } w' \text{ est un nœud feuille de } \mathscr{T} \text{ et (x, } t_e^0\text{, t)} \in \delta_e \\ \emptyset & \text{sinon} \end{cases}$$

  - L'ensemble $L_2(x,t)$ permet à $\mathcal{A}_{\mathscr{R}}$ de continuer un calcul de $\mathcal{A}_e$ déjà com-

tel-00580926, version 1 - 29 Mar 2011

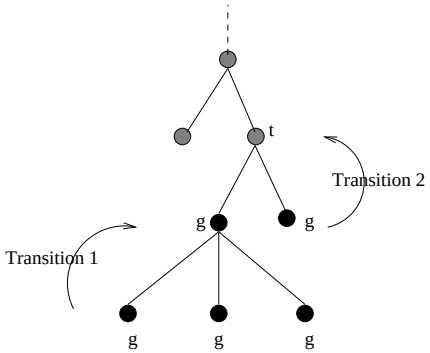


FIGURE 3.15 – Transitions 1 et 2

mencé (Transition 3 de la figure 3.16) :

$$L_2(x,t) = \bigcup_{\{t'/(x,t',t)\in\delta_e\}} g^*t'g^*$$

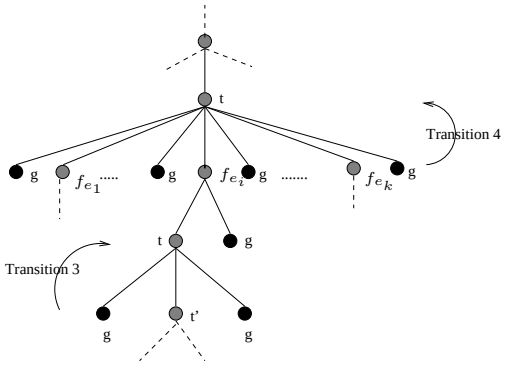


FIGURE 3.16 – Transitions

– L'ensemble $L_3(x,t)$ permet à $\mathcal{A}_{\mathscr{R}}$, si $e = (w, w')$ et $Out(w') = \{e_1, e_2, ..., e_k\}$, de démarrer un calcul de $\mathcal{A}_e$ à partir du nœud $p(w')$, afin de reconnaître le mot $\widetilde{\sigma(e)}$, lorsque les mots $\widetilde{\sigma(e_1)}, ..., \widetilde{\sigma(e_k)}$ ont déjà été reconnus (Transition 4 de la figure 3.16) :

$$L_3(x,t) = \begin{cases} g^*f_{e_1}g^*f_{e_2}g^*...g^*f_{e_k}g^* & \text{si } w' \text{ n'est pas une feuille de } \mathscr{R} \\ & \text{et } (x, t_e^0, t) \in \delta_e, \\ \emptyset & \text{sinon} \end{cases}$$

Remarque : A partir de la construction ci-dessus, on déduit que $\mathcal{A}_{\widehat{\mathscr{R}}}$ est un automate reconnaissant l'ensemble des arbres contenant une trace partielle de $\mathscr{R}$ où $\widehat{\mathscr{R}}$ est défini comme dans la définition 17.

tel-00580926, version 1 - 29 Mar 2011

Construction de $\bar{\mathcal{A}}_{\mathscr{R}}$=$(\Sigma, U, \eta, F)$

Nous modifions maintenant la construction de l'automate $\mathcal{A}_{\mathscr{R}}$ afin d'obtenir un nouvel automate $\bar{\mathcal{A}}_{\mathscr{R}}$ reconnaissant le même langage que $\mathcal{A}_{\mathscr{R}}$ mais permettant d'identifier tous les descendants de nœuds images par p des nœuds de sélection de $\mathscr{R}$. L'idée est de changer légèrement $\mathcal{A}_{\mathscr{R}}$ pour que les calculs réussis associent des états barrés à de tels nœuds.

Formellement, notons $\bar{Q}$ un ensemble de copies barrées des états de Q, $\overline{w}$ le mot $\bar{q_1}...\bar{q_k}$ si w est le mot $q_1...q_k$ de Q^*, et $\overline{S}$ l'ensemble $\{\bar{w}/w \in S\}$ si $S \subseteq Q^*$. On pose $\bar{\mathcal{A}}_{\mathscr{R}} = (\Sigma, U, \eta, F)$ avec $U = Q \cup \bar{Q}$, $F = \{f\}$ et on définit les ensembles de transitions associés à η comme suit :

- $L(x, \bar{g}) = \bar{g}^*$
- $L(x, \bar{t}) = \overline{L_1(x,t)} \cup \overline{L_2(x,t)} \cup \overline{L_3(x,t)}$
- $L(x, t) = \overline{L_1(x,t)} \cup \overline{L_3(x,t)}$ si $t \in R_e$, $e = (w, w')$ et $w' \in \mathcal{N}(\vec{s})$
- $L(x, g) = g^*$
- $L(x, t) = L_1(x,t) \cup L_2(x,t) \cup L_3(x,t)$ si $t \in Q_e$, $e = (w, w')$ et $w' \notin \mathcal{N}(\vec{s})$

Les deux premiers ensembles de transitions associent des états barrés à tous les nœuds descendants d'un nœud de sélection, tandis que le troisième permet à un calcul *ascendant* de $\bar{\mathcal{A}}_{\mathscr{R}}$ de passer d'états barrés à des états non barrés dès qu'un nœud de sélection est atteint.

Ainsi, dans tout calcul de $\bar{\mathcal{A}}_{\mathscr{R}}$ sur $\mathcal{D}$, identifiant la trace $\texttt{trace}_p(\mathscr{R}, \mathcal{D})$ d'un plongement p de $\mathscr{R}$ dans $\mathcal{D}$, les nœuds apparaissant sur cette trace ou descendant d'un nœud de sélection (c'est à dire les nœuds de $\mathcal{N}(\texttt{trace}_p(\mathcal{V}, \mathcal{D})) \cup \mathcal{N}(\mathcal{V}_p(\mathcal{D})))$ sont caractérisés par le fait qu'ils sont associés à un état de $U \setminus \{g\}$, c'est à dire à des états barrés ou à des états de $\cup_{e \in M} Q_e \cup \{f\}$. Cette propriété sera utilisée plus loin.

6.2 Construction de $\mathcal{A}$

L'automate d'arbres $\mathcal{A}$ reconnaissant le langage $\mathcal{L}$, qui intervient dans le critère d'indépendance, est construit à partir des automates $\mathcal{A}_{\widehat{\mathcal{C}}} = (\Sigma, Q_{\widehat{\mathcal{C}}}, \delta_{\widehat{\mathcal{C}}}, F_{\widehat{\mathcal{C}}})$ et $\bar{\mathcal{A}}_{\mathcal{V}} = (\Sigma, U_{\mathcal{V}}, \eta_{\mathcal{V}}, F_{\mathcal{V}})$. $\mathcal{A}_{\widehat{\mathcal{C}}}$ est l'automate reconnaissant une trace partielle de la classe de mises à jour $\mathcal{C}$ en utilisant la construction de la Section 5.1, $\bar{\mathcal{A}}_{\mathcal{V}}$ est l'automate reconnaissant une trace complète de la requête de vue $\mathcal{V}$ en utilisant la construction de la Section 5.1. La construction de $\mathcal{A}$ à partir de $\mathcal{A}_{\widehat{\mathcal{C}}}$ et $\bar{\mathcal{A}}_{\mathcal{V}}$, met en œuvre deux constructeurs classiques d'automates que nous présentons ci-dessous.

Automate produit $\mathcal{A}_1 \times \mathcal{A}_2$
Soient $\mathcal{A}_1$=$(\Sigma, Q_1, \delta_1, F_1 \subseteq Q_1)$ et $\mathcal{A}_2$=$(\Sigma, Q_2, \delta_2, F_2 \subseteq Q_2)$ deux automates d'arbres. Le langage $L(\mathcal{A}_1) \cap L(\mathcal{A}_2)$ est reconnu par l'automate produit $\mathcal{A}_1 \times \mathcal{A}_2$=$(\Sigma, Q_1 \times Q_2, \Delta, F_1 \times F_2)$ où Δ est donné par : $(x, (q, q'), (q_1, q'_1)(q_2, q'_2) \ldots (q_n, q'_n)) \in \Delta$ si et seulement si $(x, q, q_1 q_2 \ldots q_n) \in \delta_1$ et $(x, q', q'_1 q'_2 \ldots q'_n) \in \delta_2$

tel-00580926, version 1 - 29 Mar 2011

Automate à états sélectifs $\sigma(\mathcal{B}, S)$
Soient $\mathcal{B} = (\Sigma, U, \eta, F_U \subseteq U)$ un automate d'arbres et $S \subseteq U$ un sous-ensemble d'états de U. L'ensemble des arbres $\mathcal{D}$ pour lesquels il existe un calcul réussi de $\mathcal{B}$ attribuant un état de S à au moins un nœud de $\mathcal{D}$ est un langage rationnel d'arbres reconnu par l'automate $\sigma(\mathcal{B}, S)$, déduit de $\mathcal{B}$ comme suit : le fonctionnement de $\sigma(\mathcal{B}, S)$ est semblable à celui de $\mathcal{B}$ mais utilise un ensemble supplémentaire $\widehat{U}$ (copie de U) d'états chapeautés ; les calculs de $\sigma(\mathcal{B}, S)$ sur un arbre $\mathcal{D}$ sont identiques à ceux de $\mathcal{B}$ sur $\mathcal{D}$, mais attribuent des états chapeautés aux ascendants des nœuds ayant été associés à un état de S. Les états finaux de $\sigma(\mathcal{B}, S)$ sont les états finaux chapeautés de $\mathcal{B}$, ce qui assure que tout calcul réussi de $\sigma(\mathcal{B}, S)$ correspond à un calcul réussi de $\mathcal{B}$ utilisant au moins un état de S. Formellement, si π est l'application de $U \cup \hat{U}$ dans U définie $\forall u \in U$ par $\pi(\hat{u}) = \pi(u) = u$, on pose : $\sigma(\mathcal{B}, S) = (\Sigma, U \cup \hat{U}, \eta \cup \gamma, \hat{F_U})$ avec $\hat{F_U} = \{\hat{u}/u \in F_U\}$
et $\gamma = \{(x, \hat{u}, w)/w \in (U \cup \hat{U})^* S (U \cup \hat{U})^* et(x, u, \pi(w)) \in \eta\} \cup$
$\{(x, \hat{u}, w)/w \in (U \cup \hat{U})^* \hat{U} (U \cup \hat{U})^* et(x, u, \pi(w)) \in \eta\}$.
A partir de maintenant, nous supposons qu'un schéma $\mathcal{S}c$ est disponible et spécifié par un automate fini d'arbre $\mathcal{A}_{\mathcal{S}c}$, c'est à dire : $L(\mathcal{A}_{\mathcal{S}c}) = \texttt{valide}(\mathcal{S}c)$.

Proposition 10. *Le langage $\mathcal{L}$ est reconnu par l'automate d'arbres $\mathcal{A}$ défini par : $\mathcal{A} = \mathcal{A}_{\mathcal{S}c} \times \sigma(\mathcal{A}_{\widehat{\mathcal{C}}} \times \bar{\mathcal{A}}_{\mathcal{V}}, S)$ où $S = Select(\mathcal{A}_{\widehat{\mathcal{C}}}) \times (U_{\mathcal{V}} \setminus \{g\})$.*

Démonstration. Rappelons que le langage $\mathcal{L}$ est l'ensemble des arbres $\mathcal{D}$ vérifiant :
(i) $\mathcal{D} \in \texttt{valide}(\mathcal{S}c)$
(ii) il existe une trace $\texttt{trace}_p(\mathcal{V}, \mathcal{D})$ de $\mathcal{V}$ dans $\mathcal{D}$ selon un plongement p,
il existe une trace partielle $\texttt{trace}_{p'}(\mathcal{C}, \mathcal{D})$ de $\mathcal{C}$ dans $\mathcal{D}$ selon un plongement partiel p' et il existe un nœud s_c de $\vec{s_c}$ tel que $p'(s_c) \in \mathcal{N}(\texttt{trace}_p(\mathcal{V}, \mathcal{D})) \cup \mathcal{N}(\mathcal{V}_p(\mathcal{D}))$
$\mathcal{A}_{\widehat{\mathcal{C}}} \times \bar{\mathcal{A}}_{\mathcal{V}}$ reconnait l'ensemble des arbres vérifiant l'existence d'une trace $\texttt{trace}_p(\mathcal{V}, \mathcal{D})$ de $\mathcal{V}$ dans $\mathcal{D}$ et une trace partielle $\texttt{trace}_{p'}(\mathcal{C}, \mathcal{D})$ de $\mathcal{C}$ dans $\mathcal{D}$. D'autre part un calcul réussi de $\mathcal{A}_{\widehat{\mathcal{C}}} \times \bar{\mathcal{A}}_{\mathcal{V}}$ sur un arbre $\mathcal{D}$ attribue aux nœuds de $p'(\vec{s_c}) \cap (\mathcal{N}(\texttt{trace}_p(\mathcal{V}, \mathcal{D})) \cup \mathcal{N}(\mathcal{V}_p(\mathcal{D})))$ un état de $Select(\mathcal{A}_{\widehat{\mathcal{C}}}) \times (U_{\mathcal{V}} \setminus \{g\})$. Ainsi l'ensemble des arbres vérifiant la condition (ii) est reconnu par l'automate $\sigma(\mathcal{A}_{\widehat{\mathcal{C}}} \times \bar{\mathcal{A}}_{\mathcal{V}}, S)$ où $S = Select(\mathcal{A}_{\widehat{\mathcal{C}}}) \times (U_{\mathcal{V}} \setminus \{g\})$. L'ajout de la condition (i) assure l'égalité $\mathcal{L} = L(\mathcal{A})$.

6.3 Étude de la Complexité

Dans cette section nous étudions la complexité de la construction de l'automate $\mathcal{A} = \mathcal{A}_{\mathcal{S}c} \times \sigma(\mathcal{A}_{\widehat{\mathcal{C}}} \times \bar{\mathcal{A}}_{\mathcal{V}}, S)$. Nous commençons d'abord par la complexité de la construction de l'automate reconnaissant une trace.

Lemme 4. *Soit $\mathcal{A}_{\mathscr{R}} = (\Sigma, Q, \delta, F)$ l'automate construit en Section 5.1.1 à partir de la requête $\mathscr{R} = (\mathscr{T}, \vec{s})$ avec $\mathscr{T} = (\Sigma, N, M, \mathcal{E})$. Si a_m désigne l'arité maximale des*

tel-00580926, version 1 - 29 Mar 2011

nœuds de N, la taille $|\mathcal{A}_{\mathscr{R}}|$ de $\mathscr{R}$ est en $O(|\Sigma| \times |\mathscr{R}| \times a_m)$.

Démonstration. Soient $t \in Q$, $x \in \Sigma$ et $\mathcal{A}_{L(x,t)}$ un automate de mots reconnaissant le langage $L(x,t) = \{w \in Q^*/(x,t,w) \in \delta\}$. Conformément à la définition de $\mathcal{A}_{\mathscr{R}}$ on a :

$$\begin{aligned} |\mathcal{A}_{\mathscr{R}}| &= |Q| + \sum_{(x,t)\in\Sigma\times Q} |\mathcal{A}_{L(x,t)}| \\ &= |Q| + |\mathcal{A}_{L(/,f)}| + \sum_{x\in\Sigma}(|\mathcal{A}_{L(x,g)}|) \\ &\quad + \sum_{(x,t)\in\Sigma\times Q_M} (|\mathcal{A}_{L_1(x,t)}| + |\mathcal{A}_{L_2(x,t)}| + |\mathcal{A}_{L_3(x,t)}|) \end{aligned}$$

Où Q_M représente $\cup_{e\in M} Q_e$.
Rappelons tout d'abord que $|\mathscr{R}| = |N| + \sum_{e\in M} |\mathcal{A}_e|$ où $\mathcal{A}_e$ est l'automate associé à l'expression régulière $\mathcal{E}_e$. Nous montrons dans la suite que chaque terme de la somme exprimant $|\mathcal{A}_{\mathscr{R}}|$ est en $O(|\Sigma| \times |\mathscr{R}| \times a_m)$:

- $|Q| = |\bigcup_{e\in M} Q_e \cup \{f,g\}|$ est en $O(|\mathscr{R}|)$
- $\forall e \in M$, $\forall (x,t) \in \Sigma \times Q_e$, $|\mathcal{A}_{L(x,g)}|$ et $|\mathcal{A}_{L_1(x,t)}|$ sont en $O(1)$
  Ainsi $\sum_{x\in\Sigma}(|\mathcal{A}_{L(x,g)}|)$ est en $O(|\Sigma|)$ et $\sum_{(x,t)\in\Sigma\times Q_M} (|\mathcal{A}_{L_1(x,t)}|)$ est en $O(|\Sigma| \times |\mathscr{R}|)$
- Il existe une constante K telle que :
  $\forall e \in M$, $\forall (x,t) \in \Sigma \times Q_e$, $|\mathcal{A}_{L_2(x,t)}| \leqslant K \times |\{t'/(x,t',t) \in \delta_e\}|$
  De l'égalité $\sum_{(x,t)\in\Sigma\times Q_M} (|\mathcal{A}_{L_2(x,t)}|) = \sum_{e\in M}(\sum_{(x,t)\in\Sigma\times Q_e} (|\mathcal{A}_{L_2(x,t)}|))$, on déduit que $\sum_{(x,t)\in\Sigma\times Q_M} (|\mathcal{A}_{L_2(x,t)}|)$ est en $O(\sum_{e\in M} |\delta_e|)$.
  Or $\sum_{e\in M} |\delta_e| \leqslant |\mathscr{R}|$, et donc $\sum_{(x,t)\in\Sigma\times Q_M} (|\mathcal{A}_{L_2(x,t)}|)$ est en $O(|\mathscr{R}|)$.
- $\forall e \in M$, $\forall (x,t) \in \Sigma \times Q_e$, $|\mathcal{A}_{L_3(x,t)}|$ est en $O(a_m)$ et il y a au plus $\sum_{e\in M} |\delta_e|$ couples (x,t) pour lesquels $L_3(x,t)$ est non vide.
  Ainsi $\sum_{(x,t)\in\Sigma\times Q_M} (|\mathcal{A}_{L_3(x,t)}|)$ est en $O(a_m \times |\mathscr{R}|)$.
- $|\mathcal{A}_{L(/,f)}|$ est en $O(1)$

Nous étudions maintenant la taille de l'automate avec états sélectifs utilisé dans la construction de $\mathcal{A}$.

tel-00580926, version 1 - 29 Mar 2011

Lemme 5. *Soient $\mathcal{B} = (\Sigma, U, \eta, F_U \subseteq U)$ un automate d'arbres, $S \subseteq U$ un sous-ensemble d'états de U et $\Im = \sigma(\mathcal{B}, S)$ l'automate avec états sélectifs défini en Section 6.2. La taille $|\Im|$ de $\Im$ est en $O(|\mathcal{B}|)$.*

Démonstration. Nous avons :
$|\Im| = 2|U| + \sum_{(x,\hat{u})\in\Sigma\times\hat{U}} |\mathcal{A}_{L_\Im(x,\hat{u})}| + \sum_{(x,u)\in\Sigma\times U} |\mathcal{A}_{L_\Im(x,u)}|$.
Pour tout $(x,u) \in \Sigma \times U$, $L_\Im(x,u) = L_\mathcal{B}(x,u)$.
D'autre part, $L_\Im(x,\hat{u}) = \pi^{-1}(L_\mathcal{B}(x,u)) \cap (U \cup \hat{U})^*(\hat{U} \cup S)(U \cup \hat{U})^*$, et donc $|\mathcal{A}_{L_\Im(x,\hat{u})}|$ est en $O(|\mathcal{A}_{L_\mathcal{B}(x,u)}|)$.
Du fait que $|\mathcal{B}| = |U| + \sum_{(x,u)\in\Sigma\times U} |\mathcal{A}_{L_\mathcal{B}(x,u)}|$, on déduit que $|\Im|$ est en $O(|\mathcal{B}|)$.

Nous pouvons maintenant combiner les résultats des Lemmes 1 et 2.

Proposition 11. *La taille $|\mathcal{A}|$ de l'automate $\mathcal{A} = \mathcal{A}_{\mathcal{S}c} \times \sigma(\mathcal{A}_{\widehat{\mathcal{C}}} \times \bar{\mathcal{A}}_\mathcal{V}, S)$ est en $O(a_\mathcal{C} a_\mathcal{V} \times |\Sigma|^3 \times |\mathcal{A}_{\mathcal{S}c}| \times |\mathcal{C}| \times |\mathcal{V}|)$, où $a_\mathcal{C}$ et $a_\mathcal{V}$ sont les arités maximales des nœuds de $\mathcal{C}$ et $\mathcal{V}$ respectivement.*

Démonstration. Nous avons $|\mathcal{A}| \leqslant |\mathcal{A}_{\mathcal{S}c}| \times |\sigma(\mathcal{A}_{\widehat{\mathcal{C}}} \times \bar{\mathcal{A}}_\mathcal{V}, S)|$ et on déduit facilement de la construction donnée en Section 6.1 que $|\bar{\mathcal{A}}_\mathcal{V}| \leq 2|\mathcal{A}_\mathcal{V}|$. Le résultat découle alors des lemmes 1 et 2, et du fait que $|\widehat{\mathcal{C}}| \leqslant |C| + |\Sigma|$.

Proposition 12. *La complexité en temps du test de vacuité du langage $\mathcal{L}$ est en $O(a_\mathcal{C}^2\ a_\mathcal{V}^2\ |\Sigma|^6 |\mathcal{A}_{\mathcal{S}c}|^2 \times |\mathcal{C}|^2 \times |\mathcal{V}|^2)$. Ainsi le critère d'indépendance est décidable en temps polynomial.*

Démonstration. Comme $\mathcal{L} = L(\mathcal{A})$, il suffit d'utiliser l'algorithme classique de test de vacuité de $L(\mathcal{A})$ qui calcule, par saturation jusqu'à obtention d'un point fixe, le sous-ensemble des états accessibles de $\mathcal{A}$. La complexité en temps de cet algorithme est polynomiale en $O(|\mathcal{A}|^2)$. Le résultat découle alors de la Proposition 5.

7 Bilan

Dans ce chapitre nous avons étudié le problème d'indépendance entre vues et mises à jour. Notre principale contribution est d'avoir exhibé une condition suffisante d'indépendance entre une requête de vue et une classe de mises à jour, testable en temps polynomial. Cette condition est nécessaire et suffisante dans le cas particulier de requêtes de vue de l'ensemble VP et de requêtes de mises à jour de CF. Nous avons également montré que ce problème d'indépendance est en général PSPACE-difficile.
Pour cette étude, nous avons choisi de spécifier les requêtes de vue et les classes de mises à jour par des requêtes arbres régulières.
Notre analyse d'indépendance entre classes de mises à jour et requêtes de vue se

tel-00580926, version 1 - 29 Mar 2011

ramène en fait à une analyse d'indépendance entre deux requêtes $\mathcal{RAR}$s et pourrait être utilisée dans d'autres contextes d'applications : le problème de la commutativité entre deux requêtes de mises à jour étudié dans [GRS07] et dans [BBFV05] en est un exemple.

tel-00580926, version 1 - 29 Mar 2011

tel-00580926, version 1 - 29 Mar 2011

«Les machines un jour pourront résoudre tous les problèmes, mais jamais aucune d'entre elles ne pourra en poser un !.»

Albert Einstein

tel-00580926, version 1 - 29 Mar 2011

tel-00580926, version 1 - 29 Mar 2011

Chapitre 4

Contraintes d'intégrité et requêtes arbres régulières

1 Introduction

Un système de gestion de bases de données doit assurer le maintien de la cohérence des données. La cohérence des données est en général définie comme la satisfaction d'un ensemble de contraintes que l'on classe habituellement en deux types : les contraintes de schéma, définissant la structure ou types de données, et les contraintes d'intégrité exprimant des restrictions sémantiques sur les données elles-mêmes.
Un problème crucial dans les bases de données est la maintenance de ces contraintes d'intégrité après une mise à jour. Dans ce chapitre, nous étudions la préservation de la satisfaction des contraintes d'intégrité, après un certain ensemble de mises à jour, dans le contexte des bases de données XML.
Les contraintes d'intégrité comme les contraintes de clés, les contraintes d'inclusion ou les dépendances fonctionnelles ont largement été étudiées dans le modèle relationnel [Cod71, Arm74, BDFS84, BMT89]. Leur généralisation aux bases de données XML n'est pas triviale et a également fait l'objet de nombreux travaux [FS00, BDF+01, FL02, BDF+03].

La majorité de ces travaux ne traitent pas le problème de l'impact des mises à jour sur les contraintes d'intégrité. Cependant on peut les classer en quatre catégories :

- La spécification des contraintes d'intégrité [FS00, FL02, Fan05, DT05].
- L'étude des dépendances fonctionnelles et leur inférence [LLL02, VL05, FF07].
- L'axiomatisation et la normalisation [AL04, VLL04].
- La spécification des clés [BDF+02, BDF+01, BDF+03, HKL+08].

Tous ces travaux diffèrent les uns des autres par (1) leur utilisation ou non des schéma XML (DTD, XMLSchema), (2) la façon avec laquelle ils accèdent et comparent les éléments XML, (3) et la puissance d'expressivité des contraintes exprimées.

Dans ce chapitre nous nous focalisons sur les contraintes d'intégrité exprimées par des dépendances fonctionnelles et plus particulièrement au problème suivant : est-il possible de détecter si une dépendance fonctionnelle *df* est indépendante d'une classe de mises à jour $\mathcal{C}$, c'est-à-dire si tout document XML $\mathcal{D}$ satisfaisant *df* la satisfait toujours après toute mise à jour de $\mathcal{C}$. Pour cela, nous faisons le choix d'utiliser les requêtes arbres régulières vues dans le chapitre 2 comme formalisme uniforme pour représenter les dépendances fonctionnelles et les mises à jour. Nous montrons que ce choix nous permet de réaliser une analyse statique de l'indépendance d'une dépendance fonctionnelle par rapport à une classe de mises à jour $\mathcal{C}$.
Nos principaux résultats sont les suivants : (1) nous établissons que le problème d'indépendance est en général PSPACE-difficile, et (2) nous exhibons une condition suffisante testable en temps polynomial (par rapport à la taille de la dépendance fonctionnelle *df* et de la classe de mises à jour $\mathcal{C}$) assurant l'indépendance de *df* par rapport à la classe de mises à jour $\mathcal{C}$.

2 Langages d'expression de dépendances fonctionnelles

La définition des dépendances fonctionnelles nécessite la désignation de certaines parties du document XML pour une éventuelle comparaison. Pour cela il faut utiliser un langage de sélection de tuples de nœuds comme XPath ou les requêtes $\mathcal{RAR}$s .
Nous donnons dans les sections 2.1, 2.2 et 2.3 un bref aperçu des principaux langages d'expression de dépendances fonctionnelles utilisés dans les travaux antérieurs puis nous présentons en section 2.4 notre proposition d'utiliser des $\mathcal{RAR}$s pour spécifier les dépendances fonctionnelles.

2.1 Les dépendances fonctionnelles définies par des tuples d'arbre

Arenas et Libkin proposent dans [AL04] une représentation relationnelle des documents XML puis définissent les dépendances fonctionnelles sur ces relations. Ainsi un arbre XML $\mathcal{D}$ est un ensemble de tuples qui permettent de définir les dépendances fonctionnelles comme dans le cadre relationnel.
Une DTD Δ=(E,A,P,R,r) est définie dans [AL04] par :

- un ensemble fini E composé des types d'élément ;
- un ensemble fini A d'attributs ;

tel-00580926, version 1 - 29 Mar 2011

- une application P qui associe à chaque élément τ de E une définition de type P(τ) qui est soit la lettre S représentant le type chaîne de caractère soit une expression régulière α sur E ;
- une application R qui associe à chaque élément de E un sous-ensemble de A ;
- r $\in$ E est le type de la racine, et on suppose que r n'apparait dans aucune expression P(τ),$\tau \in$ E.

Soit Δ=(E,A,P,R,r) une DTD, une chaîne de caractères $w = w_1...w_n$ est un chemin de Δ si w_1=r, w_i est dans l'alphabet de P(w_{i-1}) pour tout i $\in$ [2,n-1], w_n est dans l'alphabet de P(w_{n-1}) ou peut être égale à @l si @l $\in$ R(w_{n-1}). Path(Δ) est l'ensemble de tous les chemins de Δ.
Un tuple d'arbre t de $\mathcal{D}$ selon une DTD Δ représente un sous-arbre tree$_\Delta(t)$ de $\mathcal{D} = (d, \lambda, val)$ de même racine que $\mathcal{D}$ et qui contient au plus une occurrence de chaque chemin de Δ. Plus formellement t est une fonction de Path(Δ) dans $d \cup$ I* $\cup \{\perp\}$ tel que pour tout chemin p de Path(Δ), avec t(p)$\neq\perp$:

- si last(p)=a alors t(p) $\in d$ et $\lambda(t(p)) = a$
- si p' est un préfixe de p, alors t(p')$\neq\perp$ et t(p') se trouve sur le chemin de la racine à t(p) dans $\mathcal{D}$
- si @l est défini pour t(p) et si sa valeur est une chaîne de caractère s $\in$ I* alors t(p.@l)=s

où d est le domaine d'arbre, I^* ensemble des chaines de caractères et $\perp$ est la valeur indéterminée. L'ensemble de tous les tuples d'arbre de $\mathcal{D}$ selon Δ est noté $\tau(\Delta)$.

Notation :
Si S est un sous-ensemble de Paths(Δ), et t_1 et t_2 sont des tuples d'arbre de $\mathcal{D}$ selon Δ, on note : $t_1.S = t_2.S$ si et seulement si $t_1(p)$=$t_2(p)$ $\forall p \in S$ et $t_1.S \neq \perp$ si et seulement si $t_1(p) \neq \perp$ $\forall p \in S$.

La définition d'une dépendance fonctionnelle dans [AL04] utilise en fait la notion de tuples d'arbre maximaux par rapport à la relation d'ordre $\sqsubseteq$ définie par :
soit deux tuples d'arbre t_1, t_2, alors $t_1 \sqsubseteq t_2$ si et seulement si :
- t_1= t_2 ou
- si $t_1.p \neq \perp$ implique $t_2.p \neq \perp$ et $t_1.p = t_2.p$, $\forall p \in Path(\Delta)$.

$tuple_\Delta(\mathcal{D})$ désigne l'ensemble des tuples d'arbre maximaux par rapport à $\sqsubseteq$, ainsi $tuple_\Delta(\mathcal{D})$=max$_\sqsubseteq${t $\in \tau(\Delta)$ / tree$_\Delta$(t)$\preceq \mathcal{D}$ } où
$\preceq$ est la relation d'inclusion entre document (arbre) XML définie par :
soit deux documents $\mathcal{D}_1$=(d_1, λ_1, val_1) et $\mathcal{D}_2$=(d_2, λ_2, val_2) alors $\mathcal{D}_1 \preceq \mathcal{D}_2$ si et seulement si :
- $d_1 \subseteq d_2$;
- $\forall w = kj \in d_1, \exists i, w'$ tel que $w' = ki \in d_2$, $\lambda_1(w) = \lambda_2(w')$ et $val_1(w) = val_2(w')$.

tel-00580926, version 1 - 29 Mar 2011

Définition 19. *(Dépendance fonctionnelle : définition, satisfaction [AL04])*

- *Une dépendance fonctionnelle est définie sous la forme :* $S_1 \rightarrow S_2$ *où* S_1 *et* S_2 *sont des sous-ensembles finis non vides de Path(*Δ*).*
- *Un document XML* $\mathcal{D}$ *satisfait la dépendance fonctionnelle* $S_1 \rightarrow S_2$ *si pour chaque* $t_1, t_2 \in tuple_\Delta(\mathcal{D})$ *si* $t_1.S_1 = t_2.S_1$ *et* $t_1.S_1 \neq \perp$ *alors* $t_1.S_2 = t_2.S_2$.

Remarquons que la notion de tuple d'arbre dans [AL04] qui formalise la notion d'instance d'un ensemble de chemins $\mathcal{S}$ dans un document $\mathcal{D}$, conserve la notion de préfixe, dans le sens où : si p_1 est un préfixe de p_2 dans $\mathcal{S}$ alors t(p_1) est un ancêtre de t(p_2) pour tout tuple d'arbre t de $\mathcal{D}$.

Exemple :
Soit Δ la DTD suivante :
<!ELEMENT session(candidat*)>
<!ELEMENT candidat(niveau, épreuve*, (resteAvalider|annéeEmbauche)) >
<!ELEMENT niveau (#PCDATA)>
<!ELEMENT épreuve (date, matière, note, rang) >
<!ELEMENT date (#PCDATA)>
<!ELEMENT matière (#PCDATA)>
<!ELEMENT note (#PCDATA)>
<!ELEMENT rang (#PCDATA)>
<!ELEMENT resteAvalider (matière*) >
<!ELEMENT annéeEmbauche (#PCDATA)>
<!ATTLIST candidat IDN CDATA #IMPLIED >

Un exemple de tuple d'arbre t de $tuple_\Delta(\mathcal{D})$, où $\mathcal{D}$ est le document de la figure 4.1, est donné par :
t(session)=0
t(session.candidat)=00
t(session.candidat.@IDN)=se1
t(session.candidat.niveau)=001
t(session.candidat.épreuve)=002
t(session.candidat.resteAvalider)=004
t(session.candidat.épreuve.date)=0020
t(session.candidat.épreuve.matière)=0021
t(session.candidat.épreuve.note)=0022
t(session.candidat.épreuve.rang)=0023
t(session.candidat.resteAvalider.matière)=0040
t(session.candidat.niveau.S)=D
t(session.candidat.épreuve.date.S)=12/05/09
t(session.candidat.épreuve.matière.S)=math

tel-00580926, version 1 - 29 Mar 2011

t(session.candidat.épreuve.note.S)=8
t(session.candidat.épreuve.rang.S)=15
t(session.candidat.resteAvalider.matière.S)=math

Un exemple de dépendance fonctionnelle définie sur Δ est $df : S_1 \rightarrow S_2$ avec :
S_1= {session.candidat.épreuve.matière, session.candidat.épreuve.note}
S_2= {session.candidat.épreuve.rang}.
Cette dépendance fonctionnelle exprime la contrainte que "deux candidats ayant eu la même note à une épreuve d'une même matière, sont classés identiquement".

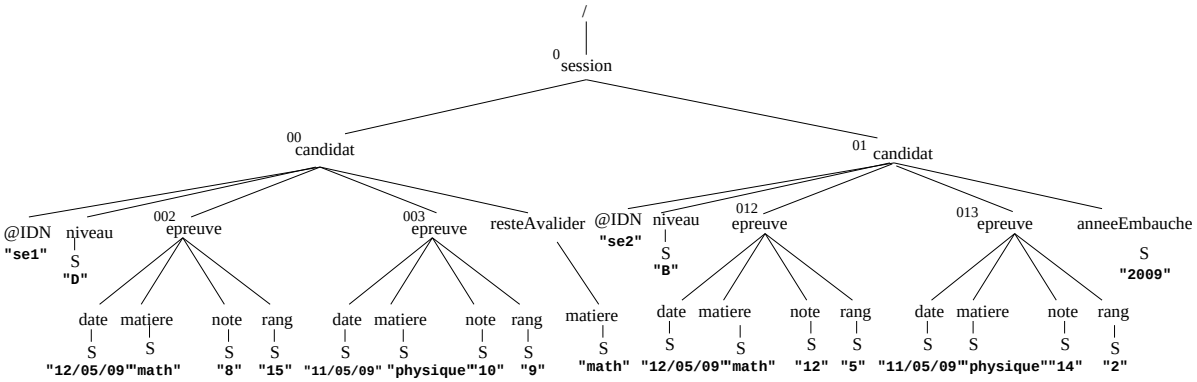


FIGURE 4.1 – Un document XML

2.2 Les dépendances fonctionnelles définies par des chemins linéaires simples

Vincent et al [VL03] définissent les dépendances fonctionnelles par des chemins et des instances de chemin et ne prennent pas en compte l'existence d'une DTD qui contraint les données. Leur approche est similaire à celle de Arenas et Libkin [AL04] dans le cas où toutes les informations sont complètes (pas de valeur nulle).

Un chemin linéaire simple p est une suite de labels $l_1...l_n$, où n ≥ 1 et l_i est soit un élément, un attribut ou un caractère spécial S pour les feuilles. Last(p) représente le dernier label du chemin p. La notation p $\cap$ q représente le chemin préfixe commun maximal entre p et q.
Une instance d'un chemin linéaire simple $l_1...l_n$ dans un document $\mathcal{D}$ est une suite de nœuds de $\mathcal{D}$: $v_1...v_n$ tel que v_1 est la racine, v_i est fils de v_{i-1} et le label de v_i est l_i pour tout i avec $2 \leq i \leq n$. L'ensemble des instances dans $\mathcal{D}$ d'un chemin linéaire simple p est noté Paths(p).
N(p) représente les nœuds terminaux des instances de p dans $\mathcal{D}$ et pour tout nœud v de $\mathcal{D}$, Nodes(v,p) désigne les descendants de v dans $\mathcal{D}$ qui sont dans N(p).

tel-00580926, version 1 - 29 Mar 2011

Définition 20. *(Dépendance fonctionnelle : définition, satisfaction [VL03])*
- *Une dépendance fonctionnelle est définie sous la forme :* $p_1, ..., p_k \rightarrow q$ *où* $p_1, ..., p_k$ *et* q *sont des chemins linéaires simples.*
- *Un document XML* $\mathcal{D}$ *satisfait (fortement) la dépendance fonctionnelle* $p_1, ..., p_k \rightarrow q$ *si pour chaque deux instances quelconques de chemins disjointes* $V{=}v_1...v_n$ *et* $V'{=}v'_1...v'_n$ *dans Paths(q)*
  Si $\forall\ 1 \leq i \leq k$
  (1) Last(p_i) *est un élément et* $x_i = y_i$ *ou*
  (2) Last(p_i) *n'est pas un élément et (* $\perp \in$ *Nodes*(x_i, p_i) *ou* $\perp \in$ *Nodes*(y_i, p_i) *ou val(Nodes*$(x_i, p_i)) \cap$ *val(Nodes*$(y_i, p_i)) \neq \emptyset$
  Alors val(v_n)*=val*(v'_n)
  où $x_i = \{v/v \in V \wedge v \in N(p_i \cap q)\}$ *et* $y_i = \{v/v \in V' \wedge v \in N(p_i \cap q)\}$

Remarquons que dans cette approche proposée par [VL03] c'est l'identification des nœuds x_i et y_i dans la définition 20 qui traduit une forme de conservation de la notion de préfixe entre les chemins p_i et q intervenant dans la spécification de la dépendance fonctionnelle et leurs instanciations dans le document : si p_i et q ont un préfixe commun, les instanciations de p_i et q dans $\mathcal{D}$ doivent partager ce préfixe commun.

Exemple : La dépendance fonctionnelle *df* : "deux candidats ayant eu la même note à une épreuve d'une même matière, sont classés identiquement" est représentée par :
session.candidat.épreuve.matière, session.candidat.épreuve.note →
session.candidat.épreuve.rang.

2.3 Les dépendances fonctionnelles définies par des chemins en PL

Dans [BHL09, Hal07] les auteurs généralisent les propositions citées précédemment dans le sens où (i) ils proposent la spécification d'un contexte pour la dépendance et (ii) ils proposent d'utiliser des chemins non nécessairement obtenus à partir d'une DTD ou d'un autre schéma XML.

Langage de chemin PL : un chemin du langage PL est défini par la grammaire : q : := ϵ | l | q/q | q//l où ϵ représente le chemin vide, l est une étiquette de Σ, le symbole "/" est l'opération de concaténation et "//" est un wildcard qui représente n'importe quelle suite finie d'étiquettes de nœuds.

tel-00580926, version 1 - 29 Mar 2011

Définition 21. *(Dépendance fonctionnelle : définition [Hal07])*
Une dépendance fonctionnelle df est une expression de la forme :
$df = (C, (\{P_1[E_1], P_2[E_2], ..., P_n[E_n]\} \rightarrow Q[E_{n+1}]))$ *où*

- *C est un chemin qui permet de désigner le nœud contexte en dessous duquel la dépendance fonctionnelle doit être respectée.*
- *{ P_1,...,P_n} est un ensemble non vide de chemins.*
- *Les symboles $E_1, ..., E_n, E_{n+1}$ représentent le type d'égalité associé à chaque chemin dans la dépendance fonctionnelle et peuvent être égaux à V pour l'"égalité de valeur ou N pour l'égalité de nœuds.*
- *$C, P_1, ..., P_n, Q$ sont des chemins de PL.*

Nous donnons plus de détails sur la notion d'égalité dans la section 2.4.

Pour introduire la définition de la satisfaction, introduisons quelques notions [BHL09] :

Motif : Soit un chemin P = $l_1/.../l_n$, où n $\geq$ 1 et Q le chemin $q_1/.../q_m$. Le chemin P est dit un préfixe du chemin Q, noté P $\subseteq$ Q, si n $\leq$ m et $l_1 = q_1, ..., l_n = q_n$.
Un *motif* est formé d'un ensemble fini de chemins dans un arbre, fermé par préfixe.

Exemple : L'ensemble de chemins M={session, session.candidat, session.candidat.@IDN, session.candidat.niveau} est un motif.

Soit SP un ensemble de chemins, le motif formé à partir des chemins de SP est le plus petit motif contenant les chemins de SP, c'est à dire l'union des chemins de SP et de leurs préfixes.

Conformité par rapport à un motif :
Soit M un motif et soit $\mathcal{D}$= (d, λ, val) un document XML.
Un *segment* dans un document $\mathcal{D}$ est un couple de positions (v_0,v_e) de d tel que v_0 est un préfixe de v_e. *Segments*($\mathcal{D}$) est l'ensemble des segments dans $\mathcal{D}$ ayant pour origine la racine.
Instances(P,$\mathcal{D}$) est l'ensemble de toutes les instances d'un chemin P dans un document $\mathcal{D}$.

$\mathcal{D}$ est conforme au *motif* M, noté $\mathcal{D} \lhd$ M, si pour tout I $\in$ *Segments*($\mathcal{D}$), il existe P $\in$ M, tel que I $\in$ *Instances*(P,$\mathcal{D}$).

Instance d'un motif M dans un document $\mathcal{D}$: soient un document $\mathcal{D}$=(d,λ,val) et un motif M. Une instance I=(d^i, λ^i, val^i) est un sous-arbre de $\mathcal{D}$ (i.e $d^i \subseteq d, \varepsilon \in d^i$, d^i est fermé par préfixe et λ^i et val^i sont les restrictions de λ et val à d) vérifiant :

1. I est conforme à M
2. $\forall$ P $\in$ M, il existe une et une seule instance de P dans I

tel-00580926, version 1 - 29 Mar 2011

tel-00580926, version 1 - 29 Mar 2011

N-uplet d'instance de motif Soit M un motif et soit un ensemble de chemins P=$\{P_1, ..., P_k\}$ tel que $P \subset M$. Soit I=(d^i, λ^i, val^i) une instance de M sur un document $\mathcal{D}$, I contient une et une seule instance de chaque P_j ; on note I_j l'instance du chemin P_j dans I, $\forall j \in [1, k]$. On note $\vec{P}$ le vecteur $P_1, ..., P_k$. Le n-uplet correspondant à $\vec{P}$ sur I , noté I[$\vec{P}$], est défini par : I[$\vec{P}$]=(val^i(Lastp(I_1),...,val^i(Lastp(I_k)).
où Lastp(I) est la position dans d de Last(I).
Deux n-uplets I[$\vec{P}$] et J[$\vec{P}$] sont égaux relativement aux opérateurs d'égalité
$\vec{E}$ =$E_1, ..., E_k$, ce qui se note I[$\vec{P}$]=$_{\vec{E}}$ J[$\vec{P}$], si et seulement si $\forall$ j $\in$[1,k],
I[P_j] =$_{E_j}$ J[P_j].

Définition 22. *(Dépendance fonctionnelle : satisfaction [Hal07]) Soit $\mathcal{D}$= (d, λ, val) un document XML et df = $(C, (\{P_1[E_1], P_2[E_2], ..., P_n[E_n]\} \rightarrow Q[E_{n+1}]))$ une dépendance fonctionnelle. Soit M le motif formé à partir de $\{C/P_1, ..., C/P_k, C/Q\}$.*
T satisfait df si et seulement si $\forall$ I,J instance de M dans T qui coïncide au moins sur le chemin C, on a :
I[$C/\vec{P}$]=$_{\vec{E}}$J[$C/\vec{P}$] $\Rightarrow$ I[C/Q]=$_{E_{n+1}}$J[C/Q]
avec $C/\vec{P}$=$\{C/P_1,...,C/P_k\}$ et $\vec{E} = (E_1, ..., E_n)$.

C est un chemin qui commence à la racine et identifie le nœud contexte sous lequel la dépendance fonctionnelle doit être respectée, les chemins $P_1, ..., P_k$ identifient les nœuds conditions dont les valeurs vont déterminer la valeur du nœud cible, ces chemins sont relatifs au nœud contexte, Q identifie le nœud cible qui est aussi relatif au nœud contexte.

Exemple : La dépendance fonctionnelle *df* : "deux candidats ayant eu la même note à une épreuve d'une même matière, sont classés identiquement" est représentée par :
(session, ({candidat.épreuve.matière[V], candidat.épreuve.note[V]} $\rightarrow$
candidat.épreuve.rang[V])), où V est l'égalité par valeur.

2.4 Expression des dépendances fonctionnelles par les requêtes arbres régulières (XDF-$\mathcal{RAR}$)

Dans ce chapitre nous proposons d'utiliser les requêtes arbres régulières vues dans le chapitre 2 pour spécifier les dépendances fonctionnelles sur des données XML. Cette proposition est justifiée par la remarque suivante : l'approche de [BHL09, Hal07] associe à chaque dépendance fonctionnelle $fd = (C, (\{P_1[E_1], P_2[E_2], ..., P_n[E_n]\} \rightarrow Q[E_{n+1}]))$ un motif formé à partir de l'ensemble des chemins $\{C/P_1$,...,C/P_n,$C/Q\}$.
Or un motif peut être représenté par une requête $\mathcal{RAR}$ permettant d'identifier les nœuds conditions et cible de la dépendance fonctionnelle.
Ainsi, l'avantage d'utiliser une requête $\mathcal{RAR}$ est (1) de permettre de spécifier précisément les positions relatives des nœuds conditions et cible dans le document et

(2) d'utiliser la puissance d'expression des $\mathcal{RAR}s$ que nous avons étudiée en section 2. Commençons par voir quelques exemples.

Dépendances fonctionnelles (XDF-$\mathcal{RAR}$) : exemples

Considérons le document XML de la Figure 4.1 .Ce document XML stocke les données de sessions d'épreuves dans un établissement d'enseignement : chaque candidat est identifié par un attribut @IDN, et reçoit une note et un rang pour chaque épreuve ; chaque candidat a aussi un niveau (de A à E) dépendant des notes reçues ; de plus si un candidat échoue à une épreuve ou ne s'y présente pas, un nœud fils "resteAvalider" est ajouté regroupant l'ensemble des matières à repasser avec succés ; au contraire si toutes les épreuves ont été passées, le candidat est diplômé et un nœud fils annéeEmbauche est ajouté stockant l'année du premier emploi.

Exemple 1 Soit la dépendance fonctionnelle suivante :
$df1$: "Dans une session, si des candidats ont obtenus la même note dans une même matière, alors ils ont obtenus le même rang dans cette matière".
Le formalisme [BHL09] présenté dans la section 2.3 exprime cette dépendance fonctionnelle par l'expression suivante :
$df1$: (/session,(candidat/épreuve/matière/, candidat/épreuve/note)→ candidat/épreuve/rang)
Dans notre formalisme, cette dépendance fonctionnelle sera représentée comme suit : $df1$=$(\mathcal{F}D_1, c)$ avec $\mathcal{F}D_1 = (\mathscr{T}_1, \vec{s_1} = (p_1, p_2, q))$ où $\mathcal{F}D_1$ est la requête arbre régulière de la Figure 4.2 permettant d'identifier les nœuds conditions et cible de $df1$.
Plus précisément, les nœuds c, p_1, p_2 et q de $\mathcal{N}(\mathscr{T}_1)$ ont les significations suivantes :

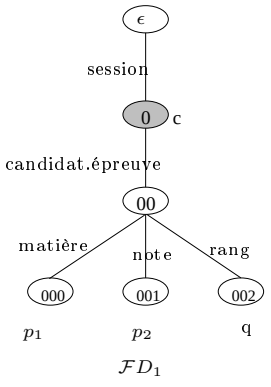


FIGURE 4.2 – XDF-$\mathcal{RAR}$ dans XML

- c (grisé dans la Figure 4.2) est le nœud contexte en dessous duquel fd_1 doit être vérifiée ; il est nécessairement un ancêtre des nœuds de $\vec{s_1}$

tel-00580926, version 1 - 29 Mar 2011

- les nœuds p_1 et p_2 de $\vec{s_1}$ représentent les nœuds conditions,
- Ce nœud q de $\vec{s_1}$ est le nœud cible.

Un document $\mathcal{D}$ satisfait fd_1 si et seulement si pour deux traces différentes $\tau_1 = \texttt{trace}_{\pi_1}(\mathcal{F}D_1, \mathcal{D})$ et $\tau_2 = \texttt{trace}_{\pi_2}(\mathcal{F}D_1, \mathcal{D})$ de $\mathcal{F}D_1$ dans $\mathcal{D}$, qui concernent la même session ($\pi_1(c) = \pi_2(c)$), la condition suivante est satisfaite : si les épreuves d'une même matière ($\pi_1(p_1)$ et $\pi_2(p_1$ ont la même valeur) sont évaluées par la même note ($\pi_1(p_2)$ et $\pi_2(p_2)$ ont aussi la même valeur), alors les rangs obtenus pour ces épreuves coïncident ($\pi_1(q)$ et $\pi_2(q)$ ont aussi la même valeur).
Le document $\mathcal{D}$ de la figure 4.1 satisfait fd_1.

Exemple 2 Soit la dépendance fonctionnelle suivante :
$df2$:"Deux épreuves différentes passées par un même candidat à la même date concernent deux matières différentes."
Le formalisme [BHL09] présenté dans la section 2.3 exprime cette dépendance fonctionnelle par l'expression suivante :
$df2$: (//candidat,(épreuve/date,épreuve/matière)→épreuve)

Dans notre formalisme, cette dépendance fonctionnelle sera représentée par $df2$=$(\mathcal{F}D_2, c)$ avec $\mathcal{F}D_2 = (\mathscr{T}_2, \vec{s_1} = (p_1, p_2, q))$ où $\mathcal{F}D_2$ est la requête arbre régulière de la Figure 4.3.

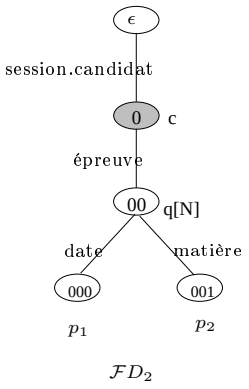


FIGURE 4.3 – XDF-$\mathcal{RAR}$ dans XML

Le document $\mathcal{D}$ de la figure 4.1 satisfait la dépendance fonctionnelle df_2.

Dépendances fonctionnelles (XDF-$\mathcal{RAR}$) : définition, satisfaction

Le traitement des dépendances fonctionnelles nécessite une définition claire de la notion d'égalité. La valeur d'un nœud est spécifiée non seulement par son label mais aussi par les valeurs de ses descendants. De plus, les contraintes ont souvent besoin de faire la différence entre l'égalité de nœuds (qui met en jeu la position du

tel-00580926, version 1 - 29 Mar 2011

nœud) et l'égalité de valeurs (qui prend en compte l'étiquette et le type du nœud ainsi que ses descendants). Nous introduisons dans cette section les notions d'égalité de nœuds (identité sur les nœuds) et d'égalité par valeur qui seront utilisées pour comparer les nœuds conditions et cible des dépendances fonctionnelles. Puis nous donnons la définition formelle précise d'une dépendance fonctionnelle ainsi que celle de sa satisfaction par un document XML.

Définition 23 (égalité de valeur). *Deux nœuds, w_1 et w_2, d'un document $\mathcal{D}=(D, \lambda, val)$ sont* égaux-par-valeur *si et seulement si :*
- $\lambda(w_1) = \lambda(w_2)$ (ils ont la même étiquette)
- w_1 et w_2 ont le même type τ
- si τ est a *(type attribut) ou* t *(type texte), alors $val(w_1) = val(w_2)$*
- si τ est e *(type élément), alors $|\{w_1 i / w_1 i \in D \wedge i \in \mathbb{N}\}| = |\{w_2 i / w_2 i \in D \wedge i \in N\}|$*
et pour tout $w_1 i$ dans D, $w_1 i$ et $w_2 i$ sont égaux-par-valeur

Nous utilisons alors les notations suivantes :
- $w_1 =_V w_2$ si w_1 et w_2 sont *égaux-par-valeur*
- $w_1 =_N w_2$ si w_1 et w_2 définissent le même nœud.

Définition 24. *(dépendance fonctionnelle dans XML) Une dépendance fonctionnelle dans XML est une expression $fd = (\mathcal{FD}, c)$ où :*
- *$\mathcal{FD} = (\mathscr{T}, \overrightarrow{s} = \{p_1[E_1], p_2[E_2], ..., p_n[E_n], q[E_{n+1}]\})$ est une requête arbre régulière dont chaque nœud sélectionné $p_1, ..., p_n$ et q est associé à un type d'égalité $E_i \in \{V, N\}$ (i=1,..., n+1)*
- *c est un nœud ancêtre de chaque nœud $p_1, p_2, ..., p_n$ et q*

c est le nœud contexte, les p_i sont les nœuds conditions et q le nœud cible.

Pour simplifier les notations, quand le type d'égalité n'est pas précisé, celui-ci vaut V par défaut et nous notons donc p_i au lieu de $p_i[V]$.

Définition 25. *(Satisfaction de dépendance fonctionnelle) Un document $\mathcal{D}$ satisfait la dépendance fonctionnelle $(\mathcal{FD}, c)$ si et seulement si quelles que soient deux traces, $\tau_1 = \mathtt{trace}_{\pi_1}(\mathcal{FD}, \mathcal{D})$ et $\tau_2 = \mathtt{trace}_{\pi_2}(\mathcal{FD}, \mathcal{D})$, de $\mathcal{FD}$ dans $\mathcal{D}$ satisfaisant*
(a) $\pi_1(c) =_N \pi_2(c)$ (les images des nœuds contextes sont identiques)
et (b) $\forall i = 1, ..., n$, $\pi_1(p_i) =_{E_i} \pi_2(p_i)$,
alors l'égalité $\pi_1(q) =_{E_{n+1}} \pi_2(q)$ est satisfaite.

Puissance d'expressivité des XFD-$\mathcal{RAR}$
Soit $fd=(C, (\{P_1[E_1], P_2[E_2], ..., P_n[E_n]\} \rightarrow Q[E_{n+1}]))$ une XFD de la forme introduite dans [BHL09] . On peut construire une XFD-$\mathcal{RAR}$ fd=$(\mathcal{FD}, c)$ dont la sémantique est trés proche de celle associé à fd dans [BHL09]. La construction de $\mathcal{FD}$ traduit les chemins $C, P_1, ..., P_n, Q$ en des mots sur Σ $w_c, w_{p_1}, ..., w_{p_n}, w_q$, et les utilise pour étiqueter les arêtes du template $\mathscr{T}$ de $\mathcal{FD}=(\mathscr{T}, \vec{s} = \{p_1, ..., p_n, q\})$; w_c étiquette une arête du nœud racine au nœud contexte c, et les mots $w_{p_1}, ..., w_{p_n}$ et w_q

tel-00580926, version 1 - 29 Mar 2011

étiquettent les chemins du nœud contexte aux nœuds conditions et cible $p_1, ..., p_n$ et q. On ajoute de plus des nœuds intermédiaires entre le nœud contexte et les nœuds $p_1, ..., p_n$ et q pour factoriser, s'il existe, le plus long préfixe commun entre les mots $w_{p_1}, ..., w_{p_n}, w_q$. Une telle construction appliquée à la df_1 de l'exemple 1 produit exactement la XFD-$\mathcal{RAR}$ $\mathcal{FD}_1$=($\mathcal{T}_1$,(p_1, p_2, q)) donnée en figure 1.2. La sémantique associée à la XFD-$\mathcal{RAR}$ fd=($\mathcal{FD}, c$) ainsi construite est très proche de celle associée à fd dans [BHL09] à ceci près que il n'y a pas d'ordre entre les chemins $P_1, ..., P_n, Q$ dans l'approche [BHL09] alors que les XFD-$\mathcal{RAR}$ imposent à chaque plongement de la requête $\mathcal{RAR}$ $\mathcal{FD}$ de respecter dans $\mathcal{D}$ l'ordre entre des nœuds frères.
Ainsi le formalisme des XFD-$\mathcal{RAR}$s permet l'expression des contraintes exprimées dans [BHL09]. L'utilisation dans les XFD-$\mathcal{RAR}$s d'expressions régulières quelconques permet, d'autre part, clairement d'exprimer des contraintes plus complexes. Les exemples 3 et 4 qui suivent montrent que, même sans utiliser des expressions régulière complexes, le formalisme des XFD-$\mathcal{RAR}$s permet d'exprimer des contraintes jusqu'alors non exprimables par les approches antérieures.

Exemple 3 Dans cet exemple, nous reprenons l'exemple de la figure 1.1 et nous supposons que les nœuds épreuve de chaque candidat sont rangés par matière et chaque candidat a au moins une épreuve dans cette matière. Soit alors les dépendances fonctionnelles suivantes :
df_3 :" Deux candidats qui ont les mêmes notes dans au moins deux matières, ont le même niveau".
df_4 :"Deux candidats qui ont les mêmes notes dans au moins deux matières et qui ont de plus des épreuves à repasser, ont le même niveau".

Ces deux dépendances fonctionnelles sont représentée par les XDF-$\mathcal{RAR}$s utilisant les $\mathcal{FD}_3$ et $\mathcal{FD}_4$ de la figure 4.4.
-La dépendance fonctionnelle df_3 ne peut être exprimée par le formalisme [BHL09], car l'utilisation des motifs dans la sémantique de [BHL09] ne permet pas de distinguer dans $\mathcal{D}$ deux épreuves différentes d'un même candidat. Par contre notre modèle de requête $\mathcal{RAR}$ le permet avec les deux arêtes étiquetées épreuve puisque la sémantique des plongements des $\mathcal{RAR}$s impose aux chemins associés à ces deux arêtes de ne pas avoir de préfixe commun dans $\mathcal{D}$.
-La dépendance fonctionnelle df_4 est un exemple de XDF-$\mathcal{RAR}$ dans laquelle des nœuds feuilles autres que les nœuds condition ou cible, sont utilisés pour imposer l'existence d'une certaine structure comme condition d'extraction des nœuds conditions et cibles, par exemple le nœud resteAvalider dans $\mathcal{FD}_4$.

tel-00580926, version 1 - 29 Mar 2011

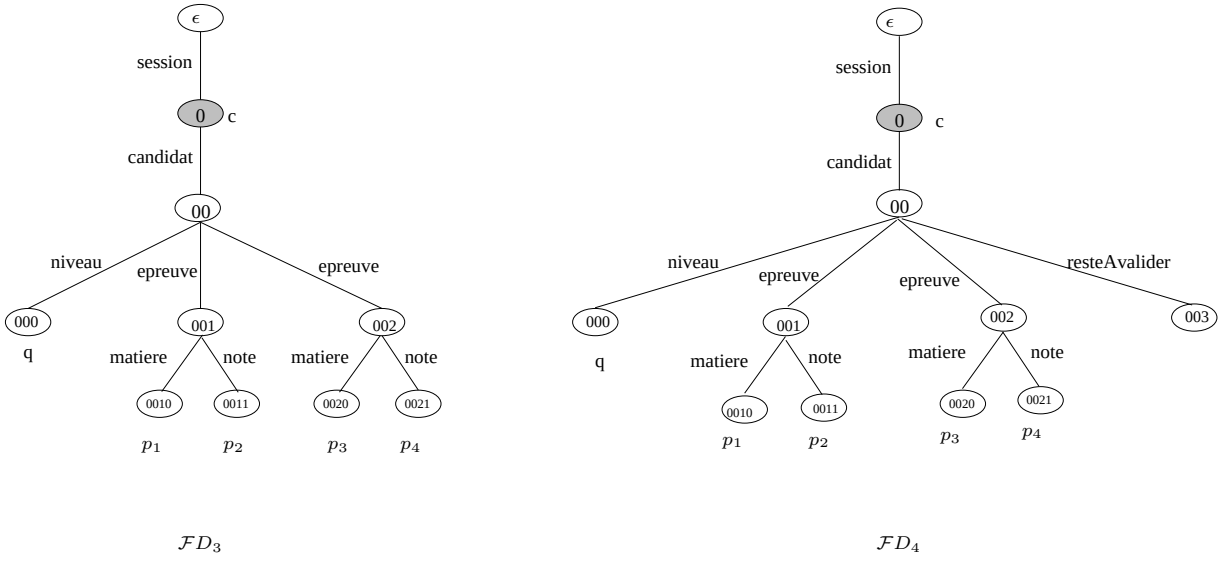


FIGURE 4.4 – XDF-$\mathcal{RAR}$ dans XML

3 Impact des mises à jour sur les contraintes d'intégrité

3.1 La problématique de l'impact

Intuitivement pour introduire la notion d'impact nous allons dans l'exemple qui suit exhiber l'intérêt d'une analyse statique des structures de la dépendance fonctionnelle et de la mise à jour pour détecter un éventuel impact sur la satisfaction de la dépendance fonctionnelle. Considérons le document XML de la Figure 4.5, et soit la dépendance fonctionnelle suivante :
df : "Deux candidats qui ont le même niveau académique et qui ont trouvé du travail, ont été embauchés la même année dans leur premier job". Le document XML de la Figure 4.5 satisfait bien la dépendance fonctionnelle *df* car les candidats 1 et 2 ont le même niveau "D" et aussi la même année d'embauche "2009". Soit maintenant la mise à jour suivante :
$\mathcal{C}$: "Mettre à jour le niveau de tous les candidats à qui il reste des épreuves à valider", une mise à jour qui change par exemple le niveau du candidat 3 à "D" a un impact sur *df* car le document mis à jour ne satisfait plus *df* puisque les deux candidats 2 et 3 ont le même niveau mais deux années d'embauche différentes.
Supposons maintenant l'existence d'un schéma qui contraint les données en imposant que tous les nœuds candidats ne doivent pas avoir en même temps un nœud "annéeEmbauche" et un nœud "resteAvalider" : dans ce cas la mise à jour $\mathcal{C}$ n'aura pas d'impact sur la satisfaction de *df*, puisque les nœuds "niveau" mis à jour appartiendront à des nœuds candidats qui n'ont pas de nœud "annéeEmbauche".

tel-00580926, version 1 - 29 Mar 2011

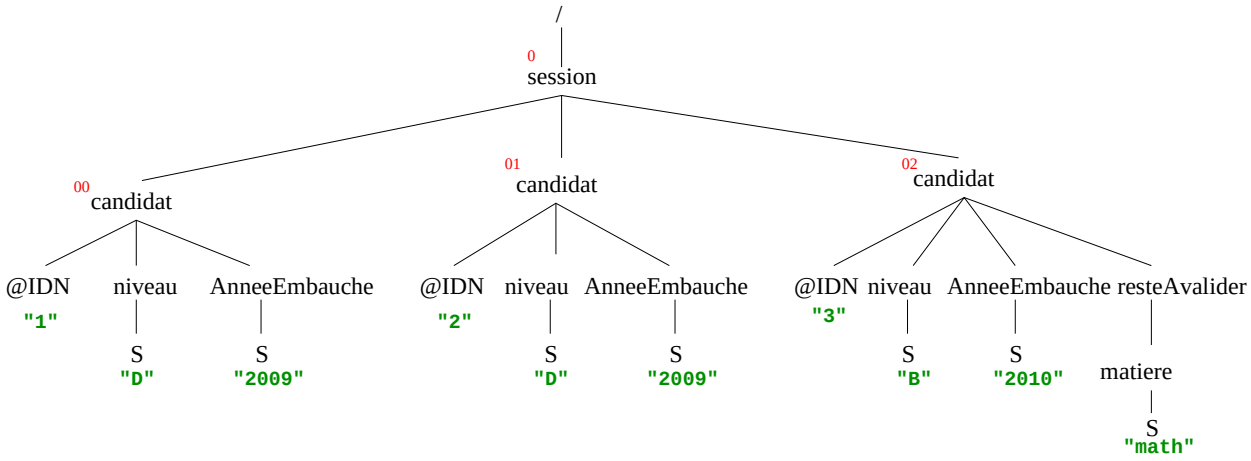


FIGURE 4.5 – Un document XML

3.2 Travaux reliés

Dans [BBG+02], Benedikt et al définissent un langage ΔX déclaratif de contraintes d'intégrité, basé sur XMLschema et un fragment simple de XPath ($XP^{\{*,//,[]\}}$ augmenté par l'utilisation des attributs et l'opérateur d'égalité). Les contraintes d'intégrité sont écrites sous une représentation formelle par la logique du premier ordre avec comptage (FOC), pour faire la vérification incrémentale après une mise à jour. ΔX est aussi un compilateur qui génère du code pour la vérification incrémentale des contraintes d'intégrité. Pour générer ce code il transforme les contraintes d'intégrité en logique prédicatif. Le problème est en général indécidable, leur algorithme se base sur des méthodes incomplètes.

Parmi les travaux plus récents qui prennent en compte les mises à jour , le plus intéressant est celui de [Lim07, BHL09] en effet, dans [Lim07, BHL09] les auteurs proposent un algorithme de vérification incrémentale d'un ensemble de dépendances fonctionnelles après une liste de mises à jour :
- Les hypothèses faites sont différentes des nôtres, en effet (1) leur approche utilise l'instance et des informations préalablement stockées lors de validations antérieures de cette instance alors que nous n'utilisons pour notre analyse statique que les spécifications des contraintes d'intégrité et celle des mises à jour et un schéma de structure $\mathcal{S}c$ si il est disponible. (2) les mises à jour sont spécifiées complètement c'est-à-dire en précisant non seulement les nœuds à modifier mais aussi les modifications qui doivent leur être appliquées. (3) les nœuds à modifier sont identifiés par leur position dans l'instance et non pas par une requête de mise à jour q : ainsi le coût de l'exécution de cette requête sur l'instance pour déterminer les positions de mise à

tel-00580926, version 1 - 29 Mar 2011

jour n'est pas du tout pris en compte.
- Lorsque le test d'indépendance est positif, le parcours de l'instance effectué est entièrement évité par notre approche. Les instances pouvant être largement plus volumineuses que les spécifications des contraintes d'intégrité et de celle des mises à jour, l'économie de leurs parcours peut permettre un gain de temps important. En revanche lorsque le test est négatif, leur méthode s'avère plus précise car elle peut détecter des cas de non violation de contraints que notre approche ne pourrait détecter, en utilisant l'instance, des informations stockées et les changements à effectuer.
- Lorsque la taille de l'instance est réduite, on peut penser que l'approche [Lim07, BHL09] est supérieure à la notre, le temps de parcours de l'instance pouvant être assez proche du temps d'exécution du test d'indépendance et permettant de conclure dans tous les cas (même lorsque le test d'indépendance est négatif). Cependant, la non prise en compte du coût de détermination des positions de mise à jour affaiblit cette thèse : en effet, l'utilisation d'une requête de mise à jour au lieu de positions de mise à jour peut rendre impossible le parcours linéaire proposé.

4 Indépendance entre classe de mises à jour et dépendances fonctionnelles (XDF-$\mathcal{RAR}$)

Dans cette section nous donnons la définition de l'impact d'une mise à jour sur une dépendance fonctionnelle, ainsi que la notion d'indépendance d'une dépendance fonctionnelle par rapport à une classe de mises à jour. Nous montrons alors que le problème de savoir si une dépendance fonctionnelle est indépendante par rapport à une classe de mises à jour est un problème PSPACE-difficle, puis nous exhibons une condition suffisante d'indépendance testable en temps polynomial.

Impact Nous dirons qu'une mise à jour q a un impact sur une dépendance fonctionnelle df si et seulement si il existe un document $\mathcal{D}$ tel que $\mathcal{D}$ satisfait df et $q(\mathcal{D})$ ne la satisfait pas.
Indépendance Comme pour l'analyse statique, effectuée dans le chapitre 3 pour la détection de l'indépendance d'une vue par rapport à une classe de mises à jour, notre objectif est d'analyser les spécifications de la dépendance fonctionnelle df et d'une mise à jour q pour détecter un éventuel impact sans utilisation du document source , qui peut ne pas être disponible durant l'analyse. Dans un souci de simplification, et parce que nous avons choisi de représenter les dépendances fonctionnelles au moyen de $\mathcal{RAR}$, nous étudions en fait le problème de l'indépendance d'une dépendance fonctionnelle df par rapport à une classe de mises à jour $\mathcal{C}$. En effet, une classe $\mathcal{C}$ de mises à jour étant elle aussi représentable par une $\mathcal{RAR}$, l'analyse de l'indépendance va pouvoir être menée de manière similaire à l'analyse d'indépendance menée au

tel-00580926, version 1 - 29 Mar 2011

chapitre 3 dans le contexte des vues.

Définition 26. *Une dépendance fonctionnelle df =$(\mathcal{F}D, c)$ est indépendante par rapport à une classe de mises à jour $\mathcal{C}$ si et seulement si pour tout document $\mathcal{D}$ satisfaisant df et pour toute mise à jour q de $\mathcal{C}$, alors $q(\mathcal{D})$ satisfait aussi df.*

Indépendance dans le contexte d'un schéma Dans le cas où un schéma imposant des contraintes aux données sources est disponible, on affine notre analyse par l'ajout de cette information pour détecter un nombre éventuellement plus important de cas d'indépendance. Rappelons que `valide`($\mathcal{S}c$) est l'ensemble des documents valides par rapport au schéma $\mathcal{S}c$. Dans ce contexte la définition de l'indépendance est modifiée comme suit :

Définition 27. *Une dépendance fonctionnelle df =$(\mathcal{F}D, c)$ est indépendante par rapport à une classe de mises à jour $\mathcal{C}$ dans le contexte d'un schéma $\mathcal{S}c$ si et seulement si, tout document $\mathcal{D}$ valide par rapport à $\mathcal{S}c$ (i.e $\mathcal{D} \in$ `valide`($\mathcal{S}c$)) et satisfaisant df, satisfait toujours df après toute mise à jour q de $\mathcal{C}$ préservant $\mathcal{S}c$.*

4.1 Problème PSPACE-difficile

Proposition 13. *Décider si une dépendance fonctionnelle df est indépendante par rapport à une classe de mises à jour $\mathcal{C}$ est un problème PSPACE-difficile.*

Démonstration. Nous réduisons le problème de l'inclusion d'expressions régulières, qui est PSPACE-difficile [MNSC04], au problème de l'indépendance d'une dépendance fonctionnelle par rapport à une classe de mises à jour. Considérons l'alphabet d'étiquettes $\Sigma = \{A, B, C, D, F, G, H, \#\}$, deux expressions η et η' de $REG(\Sigma)$ dans lesquelles le symbole '#' n'apparaît pas. Nous définissons deux requêtes arbres régulières $\mathcal{F}D$ et $\mathcal{C} = (\mathscr{T}_{\mathcal{C}}, s_{\mathcal{C}})$ comme indiqué sur la Figure 4.6, et nous montrons que *df*=($\mathcal{F}D$,c) est dépendante de $\mathcal{C}$ si et seulement si $\eta \not\subseteq \eta'$.

Supposons que *df* soit dépendante de $\mathcal{C}$. Il existe un document $\mathcal{D}$ qui satisfait *df* et une mise à jour q de $\mathcal{C}$ tels que $q(\mathcal{D})$ ne satisfait plus *df*. Alors il existe deux plongements π_1 et π_2 de $\mathcal{F}D$ dans $q(\mathcal{D})$ dont les traces $\tau^1_{\mathcal{F}D}$ et $\tau^2_{\mathcal{F}D}$ sont témoins de la violation de *df* dans $q(\mathcal{D})$, l'idée est de montrer que l'on peut trouver à partir de ces deux traces deux témoins de la violation de fd dans $\mathcal{D}$.

- Pour i=1,2 examinons la trace $\tau^i_{\mathcal{F}D}$, deux cas sont possibles :
  - (a) aucun nœud de $\tau^i_{\mathcal{F}D}$ ne provient de l'application de la mise à jour q et dans ce cas $\tau^i_{\mathcal{F}D}$ était également une trace de $\mathcal{F}D$ dans $\mathcal{D}$.
  - (b) $\tau^i_{\mathcal{F}D}$ apparait dans q($\mathcal{D}$) suite à l'application de q. Ainsi il existe un plongement π de $\mathcal{C}$ dans $\mathcal{D}$ tel que $\pi(s_u)$ est un nœud de $\tau^i_{\mathcal{F}D}$. Compte tenu de la structure des templates $\mathcal{C}$ et $\mathcal{F}D$, on en déduit que l'arbre $\tau^0_{\mathcal{F}D}$ de la figure 4.7 existait dans $\mathcal{D}$, pour permettre l'application de q, et contenait le sous-arbre t_0 de $\tau^i_{\mathcal{F}D}$ mis en évidence par les pointillés de la figure 4.7. Ainsi si $\eta \subseteq \eta'$,

tel-00580926, version 1 - 29 Mar 2011

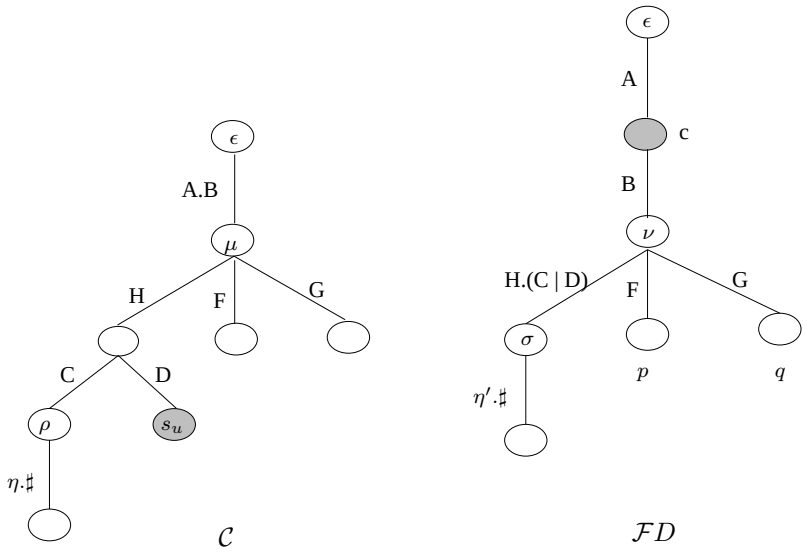


FIGURE 4.6 – Schéma de réduction

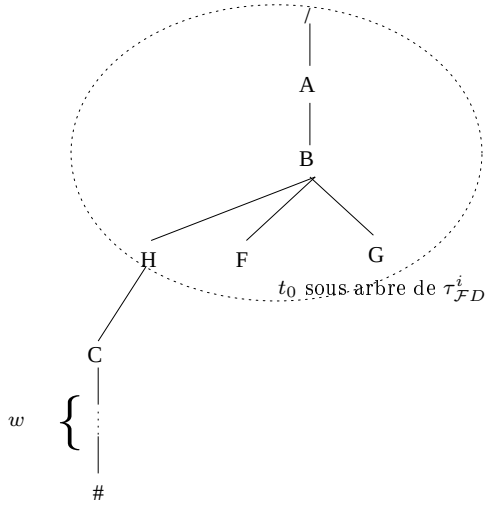


FIGURE 4.7 – $\tau^0_{\mathcal{F}D}$

tel-00580926, version 1 - 29 Mar 2011

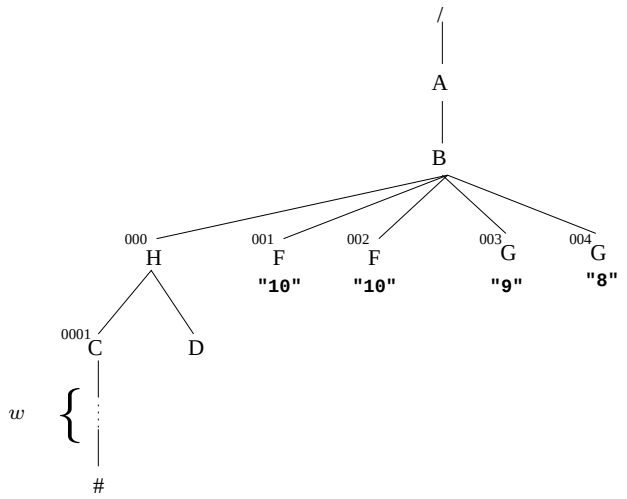


FIGURE 4.8 – Arbre de dépendance

$\tau^0_{\mathcal{F}D}$ est une trace de *df* dans $\mathcal{D}$ dont le nœud condition étiqueté par F et le nœud cible étiqueté par G ont les mêmes valeurs que dans $q(\mathcal{D})$.

Finalement, si $\eta \subseteq \eta'$, en utilisant pour chaque i=1,2, $\tau^i_{\mathcal{F}D}$ (pour le cas (a)) ou $\tau^0_{\mathcal{F}D}$(pour le cas (b)). On obtient deux traces de $\mathcal{F}D$ dans $\mathcal{D}$ qui témoignent de la violation de $\mathcal{F}D$ dans $\mathcal{D}$ ce qui est contraire à l'hypothèse.

Inversement, si $\eta \nsubseteq \eta'$, nous considérons le document $\mathcal{D}$ représenté dans la Figure 4.8 où la suite des étiquettes, apparaissant sur le chemin du nœud 'C' au nœud '$\#$', forment un mot w de $L(\eta)$ qui n'appartient pas à $L(\eta')$. $\mathcal{D}$ satisfait *df* puisque w n'appartient pas à $L(\eta')$. Le nœud 0001 est mis à jour par n'importe quelle mise à jour de $\mathcal{C}$ car w est dans $L(\eta)$. Considérons la mise à jour q de $\mathcal{C}$ qui ajoute un chemin descendant $w'\#$ au nœud 0001, où w' est un mot de $L(\eta')$. Clairement $q(\mathcal{D})$ ne satisfait pas *df* car les deux nœuds étiquetés par G ont deux valeurs différentes alors que ceux étiquetés par F sont égaux-par-valeur. □

4.2 Un critère d'indépendance

Dans cette section on établit une condition suffisante pour qu'une classe de mises à jour $\mathcal{C}$ ne viole pas une dépendance fonctionnelle *df*=($\mathcal{F}D$,c) . Ce critère suffisant d'indépendance est évaluable en temps polynomial.

L'analyse statique de l'indépendance de *df* par rapport à une classe de mise à jour est similaire à l'analyse effectuée dans le chapitre 3. Nous définissons dans un premier temps un langage $\mathcal{L}$ de documents XML satisfaisant un certain nombre de conditions, puis nous établissons la relation entre la vacuité de $\mathcal{L}$ et le problème d'indépendance d'une dépendance fonctionnelle *df* par rapport à une classe de mises à jour dans le contexte d'un schéma.

Soit $\mathcal{C}$=($\mathscr{T}_\mathcal{C}, \vec{s}_\mathcal{C}$) une classe de mises à jour. Rappelons que $\mathcal{C}^p$=($\mathscr{T}^p, \vec{s}_\mathcal{C}$) est une

tel-00580926, version 1 - 29 Mar 2011

requête $\mathcal{RAR}$ où $\mathscr{T}^p$ est le template obtenu à partir de $\mathscr{T}_\mathcal{C}$ en supprimant tous les descendants des nœuds de $\vec{s}_\mathcal{C}$ et en étiquetant par Σ toutes les arêtes entrantes (w,s) dans un nœud de sélection s de $\vec{s}_\mathcal{C}$.
Rappelons également que, pour tout document $\mathcal{D}$, et tout plongement π de $\mathcal{F}D$ dans $\mathcal{D}$, la notation $\mathcal{F}D_\pi(\mathcal{D})$ désigne le n-uplet des sous-arbres extraits par $\mathcal{F}D$ dans $\mathcal{D}$ selon π, i.e $\mathcal{F}D_\pi(\mathcal{D})$=$(\mathcal{D}(\pi(s_1)), ..., \mathcal{D}(\pi(s_n)))$ où $(s_1, s_2, ..., s_n)$=$\vec{s}_{\mathcal{F}D}$ et si π' est un plongement partiel de $\mathcal{C}$ dans $\mathcal{D}$, $\mathcal{N}(\pi'(\vec{s}_\mathcal{C}))$ désigne l'ensemble des images par π' des nœuds $\vec{s}_\mathcal{C}$.

Définition 28. *Soit $\mathcal{L}$ l'ensemble des arbres $\mathcal{D}$ vérifiant :*
(i) $\mathcal{D} \in \mathtt{valide}(\mathcal{S}c)$
(ii) il existe une trace de $\mathcal{F}D$ dans $\mathcal{D}$ selon un plongement π, $\tau_{\mathcal{F}D} = trace_\pi(\mathcal{F}D, \mathcal{D})$, et il existe une trace partielle $\tau_{\mathcal{C}^p} = trace_{\pi'}(\mathcal{C}^p, \mathcal{D})$ de $\mathcal{C}$ dans $\mathcal{D}$, selon un plongement partiel π', tel que
$\mathcal{N}(\pi'(\vec{s}_\mathcal{C})) \bigcap (\mathcal{N}(trace_\pi(\mathcal{F}D, \mathcal{D})) \cup \mathcal{N}(\mathcal{F}D_\pi(\mathcal{D}))) \neq \emptyset$

Proposition 14. *Si $\mathcal{L}$ est vide alors df est indépendante de $\mathcal{C}$ dans le contexte $\mathcal{S}c$.*

Démonstration. Supposons que *df* soit dépendante de $\mathcal{C}$ dans le contexte $\mathcal{S}c$. Alors il existe un document $\mathcal{D} \in \mathtt{valide}(\mathcal{S}c)$ qui satisfait *df* et une mise à jour q de $\mathcal{C}$ préservant la validité par rapport à $\mathcal{S}c$ tels que $q(\mathcal{D})$ ne satisfait plus *df*. Par conséquent il existe au moins un nœud n mis à jour par q qui génère un cas de de violation de *df* dans $q(\mathcal{D})$, de sorte que l'une des deux conditions (a) ou (b) ci-dessous soit satisfaite :

- (a) La violation de *df* dans $q(\mathcal{D})$ implique une trace $\tau_{\mathcal{F}D}$ de $\mathcal{F}D$ dans $q(\mathcal{D})$ qui existait aussi dans $\mathcal{D}$ mais qui n'a pas été touchée par q ($\tau_{\mathcal{F}D} = trace_\pi(\mathcal{F}D, \mathcal{D})$). Donc n est un nœud qui appartient nécessairement à un des sous-arbres enraciné par un nœud condition ou un nœud cible et sa mise à jour modifie la valeur de ce sous-arbre générant ainsi une violation de *df* dans $q(\mathcal{D})$ (cas 1 Figure 4.9). Dans ce cas n est un nœud de $\mathcal{N}(\pi'(\overrightarrow{s}_\mathcal{C})) \cap \mathcal{N}(\mathcal{F}D_\pi(\mathcal{D}))$ avec π'=partiel(π_2) et π_2 est le plongement de $\mathcal{C}$ dans $\mathcal{D}$ extrayant n pour sa mise à jour. Ainsi $\mathcal{D}$ est un document de $\mathcal{L}$.
- (b) la mise à jour de n crée une nouvelle trace $\tau_{\mathcal{F}D}$ de $\mathcal{F}D$ dans $q(\mathcal{D})$ qui est impliquée dans la violation de *df* dans $q(\mathcal{D})$: par conséquent n est un nœud de $\tau_{\mathcal{F}D}$ dans $q(\mathcal{D})$ (cas 2 Figure 4.9). Or le plongement partiel, partiel(π'), de $\mathcal{C}$ dans $\mathcal{D}$ subsiste dans $q(\mathcal{D})$ puisque seul les nœuds de mise à jour et leurs descendants sont modifiés par q : formellement, il existe un plongement partiel π'' de $\mathcal{C}$ dans $q(\mathcal{D})$ dont l'image dans $q(\mathcal{D})$ coïncide avec celle de partiel(π') dans $\mathcal{D}$. Finalement n est un nœud de $\mathcal{N}(\pi''(\overrightarrow{s}_\mathcal{C})) \cap \mathcal{N}(trace_\pi(\mathcal{F}D, q(\mathcal{D})))$ et donc $q(\mathcal{D})$ est un document de $\mathcal{L}$.

□

tel-00580926, version 1 - 29 Mar 2011

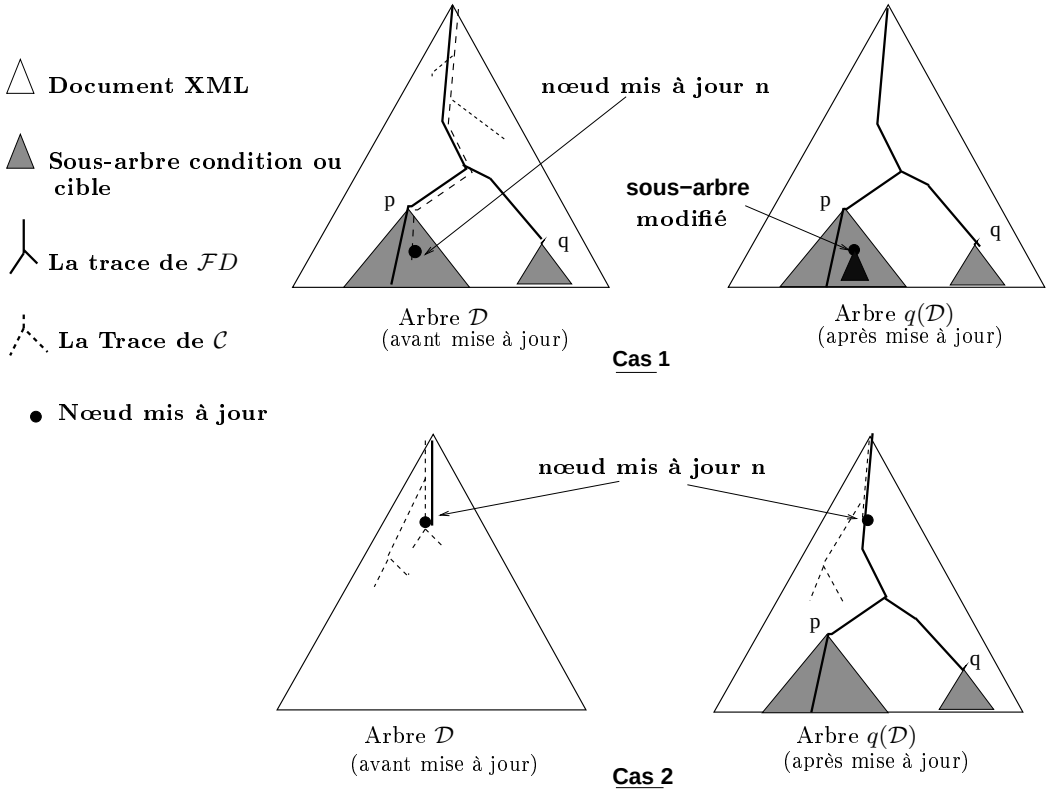


FIGURE 4.9 – Analyse d'impact

tel-00580926, version 1 - 29 Mar 2011

tel-00580926, version 1 - 29 Mar 2011

4.3 Vérification du critère d'indépendance

Nous montrons maintenant que la vérification du critère d'indépendance, $\mathcal{L} = \emptyset$, est décidable en temps polynomial par rapport aux tailles de df, $\mathcal{C}$ et $\mathcal{S}c$. Pour cela, nous montrons que $\mathcal{L}$ est un langage régulier d'arbres reconnaissable par un automate $\mathcal{A}$ dont la taille est polynomiale en les tailles de df, $\mathcal{C}$ et $\mathcal{S}c$. Comme dans le chapitre 3, le résultat découle alors du fait que le test de vacuité d'un langage régulier d'arbres est décidable en temps polynomial en la taille d'un automate reconnaissant ce langage.

Proposition 15. *Soit $fd = (\mathcal{F}D, c)$ une dépendance fonctionnelle, $\mathcal{C} = (\mathscr{T}_{\mathcal{C}}, \vec{s}_{\mathcal{C}})$ une classe de mise à jour et $\mathcal{A}_{\mathcal{S}c}$ un automate régulier d'arbre spécifiant un schéma $\mathcal{S}c$,*

- *Un automate régulier d'arbre $\mathcal{A}$ reconnaissant le langage $\mathcal{L}$ peut être construit à partir de l'automate $\mathcal{A}_{\mathcal{S}c}$ et des requêtes $\mathcal{RAR}$s $\mathcal{C}$ et $\mathcal{F}D$.*
- *La taille $|\mathcal{A}|$ de l'automate $\mathcal{A}$ est en $O(a_{\mathcal{C}} a_{\mathcal{F}D} \times |\Sigma|^2 \times |\mathcal{A}_{\mathcal{S}c}| \times |\mathcal{C}| \times |\mathcal{F}D|)$, où $a_{\mathcal{C}}$ et $a_{\mathcal{F}D}$ sont les arités maximales de $\mathcal{C}$ et $\mathcal{F}D$ respectivement.*
- *Le critère d'indépendance est polynomial, plus précisément la vacuité du langage $\mathcal{L}$ est testable en temps $O(a_{\mathcal{C}}^2\ a_{\mathcal{F}D}^2 \times |\Sigma|^4 \times |\mathcal{A}_{\mathcal{S}c}|^2 \times |\mathcal{C}|^2 \times |\mathcal{F}D|^2)$.*

Démonstration. La démonstration de cette proposition se déroule de la même façon que celle dans le chapitre 3 section 6. □

5 Bilan

Dans ce chapitre nous avons proposé une spécification des dépendances fonctionnelles pour documents XML avec le formalisme des requêtes arbres régulières $\mathcal{RAR}$s. Nous avons montré comment ce formalisme peut fédérer la plupart des approches proposées pour la modélisation des dépendances fonctionnelles en XML tout en permettant d'exprimer de nouveaux types de contraintes jusqu'alors non exprimables par les approches antérieures.
En modélisant conjointement les classes de mises à jour par les requêtes arbres régulières, nous avons montré qu'il est possible de définir une condition suffisante pour assurer l'indépendance entre une dépendance fonctionnelle et une classe de mises à jour dans le contexte d'un schéma (si ce dernier est disponible). Cependant, le problème de l'indépendance est en général PSPACE-difficile.

« Raison des effets. - La concupiscence et la force sont les sources de toutes nos actions : la concupiscence fait les volontaires ; la force, les involontaires.»

Blaise PASCAL / Pensées / Œuvres complètes

tel-00580926, version 1 - 29 Mar 2011

tel-00580926, version 1 - 29 Mar 2011

Chapitre 5

Conclusion

1 Les principaux résultats

Nous avons défini un nouveau formalisme de sélection de n-uplets de nœuds : les requêtes arbres régulières $\mathcal{RARs}$. Une requête arbre régulière ($\mathcal{RAR}$) sur un alphabet Σ est donc un arbre dont les arcs sont étiquetés par des expressions régulières sur Σ et dont certains nœuds sont grisés pour identifier le n-uplet de nœuds de sélection. Cet arbre joue le rôle d'un template spécifiant les conditions qui doivent être satisfaites par un n-uplet de nœuds pour être sélectionné.
Nous avons choisi d'utiliser ce formalisme, indépendant de tout standard, pour étudier l'impact de mises à jour sur des vues et sur des dépendances fonctionnelles. Les principales contributions de cette thèse sont rappelées ci-après.

Positionnement et pouvoir d'expression des $\mathcal{RARs}$:

XPath est le principal langage utilisé pour la sélection de nœuds. De ce fait, il est important de positionner les requêtes $\mathcal{RARs}$ par rapport à XPath. Dans sa spécification complète, XPath permet l'utilisation de fonctions et d'opérateurs sur les valeurs des nœuds. Il permet donc de définir des conditions de sélection de nœuds que les requêtes $\mathcal{RARs}$ ne peuvent exprimer. Cependant, nous montrons que son pouvoir d'expression est incomparable avec celui des requêtes $\mathcal{RARs}$.
La plupart des travaux sur XPath se restreignent à des fragments pour faciliter l'identification de leurs pouvoirs d'expression comme langages de requêtes. Les fragments les plus connus sont $CoreXPath$ et le fragment simple $XP^{\{*,//,[]\}}$. Alors que les $\mathcal{RARs}$ ne sont pas comparables avec XPath, nous avons montré que toute expression de $CoreXPath^{+}$ (le fragment positif de $CoreXPath$) est exprimable par une union finie de requêtes $\mathcal{RARs}$.
Maarten Marx a défini le langage XPath conditionnel, $CXPath$, en étendant $CoreXPath$ de manière à obtenir un langage FO complet. Le fragment positif de $CXPath$ reste incomparable avec les unions de requêtes $\mathcal{RARs}$. Cependant nous avons montré

que toute requête $\mathcal{RAR}$ est équivalente à une formule de $FOREG$: l'extension de la logique du premier ordre par des expressions régulières.

L'étude de l'indépendance entre vues et classes de mises à jour :

Le problème de la détection d'impact de mises à jour sur des vues dans le contexte XML reste un problème ouvert et important. Dans toute approche qui traite cette problématique, les langages utilisés pour représenter les requêtes de vues et de mises à jour, jouent un rôle crucial aussi bien pour la méthode utilisée que pour les résultats obtenus. Nous avons fait le choix dans ce travail, de modéliser les requêtes de vues et de mises à jour par des requêtes arbres régulières $\mathcal{RARs}$.
L'analyse d'indépendance que nous proposons permet l'utilisation de schémas contraignant les données : notre approche permet en effet de prendre en compte de tels schémas, ce qui conduit à priori à déterminer un nombre plus important de cas d'indépendance.

Nous avons montré que ce problème d'indépendance est en général PSPACE-difficile. Par ailleurs nous avons exhibé une condition suffisante d'indépendance, testable en temps polynomial, et basée sur la définition d'un langage régulier d'arbres $\mathcal{L}$ qui est construit à partir des spécifications de la classe de mises à jour $\mathcal{C}$, de la vue $\mathcal{V}$ et du schéma $\mathcal{S}c$: la vacuité de $\mathcal{L}$ est une condition suffisante d'indépendance entre la requête de vue $\mathcal{V}$ et la classe de mises à jour $\mathcal{C}$ lorsque les documents XML considérés sont conformes au schéma $\mathcal{S}c$.

Enfin, nous avons exhibé une classe de requêtes de vues et de mises à jour pour lesquelles cette condition devient une condition nécessaire et suffisante d'indépendance : il s'agit du cas où les requêtes de vues sont définies par des $\mathcal{RARs}$ dont tout nœud est descendant ou ancêtre d'un nœud de sélection (requêtes VP) et où les mises à jour ne modifient que des feuilles (classes CF).

La vérification du critère d'indépendance, $\mathcal{L} = \emptyset$, est décidable en temps polynomial par rapport aux tailles de $\mathcal{V}$, $\mathcal{C}$ et $\mathcal{S}c$. Pour cela, nous avons montré que $\mathcal{L}$ est un langage régulier d'arbres reconnaissable par un automate $\mathcal{A}$ dont la taille est polynomiale en les tailles de $\mathcal{V}$, $\mathcal{C}$ et $\mathcal{S}c$. Le résultat découle alors du fait que le test de vacuité d'un langage régulier d'arbres est décidable en temps polynomial en la taille d'un automate reconnaissant ce langage.

L'étude de l'indépendance entre dépendances fonctionnelles et classes de mises à jour :

Plusieurs types de contraintes d'intégrité ont été proposées et étudiées dans le cadre XML. Parmi ces contraintes d'intégrité, les dépendances fonctionnelles sont les plus utilisées. Dans ce travail nous avons proposé une spécification des dépen-

tel-00580926, version 1 - 29 Mar 2011

dances fonctionnelles pour documents XML qui fédère les principales propositions existantes : cette spécification utilise le formalisme des requêtes arbres régulières $\mathcal{RARs}$. Les dépendances fonctionnelles modélisées par les $\mathcal{RARs}$ permettent également d'exprimer de nouveaux types de contraintes jusqu'alors non exprimables par les approches antérieures.

La modélisation conjointe des classes de mises à jour et des dépendances fonctionnelles par les requêtes arbres régulières conduit à utiliser une approche similaire à celle utilisée dans le cas des vues pour analyser l'indépendance entre mises à jour et dépendances fonctionnelles.
Ainsi, nous avons montré qu'il est possible de définir une condition suffisante (la vacuité d'un certain langage régulier d'arbres) testable en temps polynomial pour assurer l'indépendance entre une dépendance fonctionnelle et une classe de mises à jour dans le contexte d'un schéma (si ce dernier est disponible). Cependant, en général ce problème d'indépendance reste PSPACE-difficile.

2 Perspectives

Les travaux présentés dans cette thèse laissent encore un certain nombre de perspectives de recherches que nous présentons ci-dessous.

Évaluation et pouvoir d'expression :
Même si les requêtes $\mathcal{RARs}$ sont un formalisme expressif pour sélectionner des nœuds dans un document XML, elle présentent certaines limitations :
- Les $\mathcal{RARs}$ ne permettent pas d'exprimer des conditions faisant intervenir les valeurs textuelles des documents ;
- Les $\mathcal{RARs}$ ne sont pas stables ni par intersection, ni par complémentation ;
- Les $\mathcal{RARs}$ ne traitent ni la négation, ni la disjonction ;
- L'évaluation de ces requêtes n'est pas triviale.

Il est clair que le problème de l'évaluation des requêtes $\mathcal{RARs}$ reste à étudier. On peut envisager, par exemple, de restreindre les expressions régulières utilisées pour faciliter leur traitement. L'introduction de l'opérateur de négation dans les $\mathcal{RARs}$ est également une voie à explorer.

Extension de l'analyse d'indépendance :
Notre analyse d'indépendance entre classes de mises à jour et requêtes de vue se ramène en fait à une analyse d'indépendance entre deux requêtes $\mathcal{RARs}$ et pourrait être utilisée dans d'autres contextes d'applications :
Les systèmes de données qui évoluent de manière permanente, nécessitent des mécanismes d'optimisation pour l'exécution des suites de mises à jour qu'ils reçoivent.

tel-00580926, version 1 - 29 Mar 2011

On peut envisager par exemple de paralléliser certaines de ces mises à jour ou bien de modifier leur ordre d'exécution. Déterminer si deux mises à jour commutent, c'est à dire si l'ordre dans lequel elles sont effectuées sur les données n'a pas d'impact sur le résultat final, peut ainsi permettre d'optimiser le temps d'exécution final de la suite de mises à jour reçue.
Ainsi nous pensons que notre approche pourrait être adaptée à d'autres contextes d'applications nécessitant une analyse de dépendances entre requêtes.

Implémentation :

Il reste cependant crucial de montrer par une implantation réelle, que l'évaluation de ce critère d'indépendance est moins coûteux en temps que la réévaluation de la vue ou la revérification de la dépendance fonctionnelle, après la mise à jour.

***RARs* et contraintes d'intégrités :**

Notre proposition de modéliser les dépendances fonctionnelles sur des documents XML reste à être validée. Pour cela, une axiomatisation et une normalisation de telles dépendances fonctionnelles restent à entreprendre, ainsi que la mise en œuvre d'algorithmes efficaces de vérification.
D'autres contraintes d'intégrité sont également souvent utilisées comme les contraintes de clés ou les contraintes d'inclusion. La modélisation de ces contraintes par les *RARs* peut être également envisagée.

tel-00580926, version 1 - 29 Mar 2011

tel-00580926, version 1 - 29 Mar 2011

Bibliographie

[ABMP07] Andrei Arion, Véronique Benzaken, Ioana Manolescu, and Yannis Papakonstantinou. Structured materialized views for XML queries. In *VLDB '07 : Proceedings of the 33rd international conference on Very large data bases*, 2007.

[ABS00] Serge Abiteboul, Peter Buneman, and Dan Suciu. *Data on the Web : from relations to semistructured data and XML*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.

[AFP03] M. Akhtar Ali, Alvaro A.A. Fernandes, and Norman W. Paton. MOVIE : An incremental maintenance system for materialized object views. *Data & Knowledge Engineering*, 47 :131–166, 2003.

[AL04] Marcelo Arenas and Leonid Libkin. A normal form for XML documents. *ACM Trans. Database Syst.*, 29(1) :195–232, 2004.

[AMR+98] Serge Abiteboul, Jason McHugh, Michael Rys, Vasilis Vassalos, and Janet L. Wiener. Incremental maintenance for materialized views over semistructured data. In *Proc. 24th Int. Conf. Very Large Data Bases, VLDB*, pages 38–49, 1998.

[Arm74] William Ward Armstrong. Dependency structures of data base relationships. *IFIP Congress*, pages 580–583, 1974.

[Bar05] Pablo Barcelo. Temporal logics over unranked trees. In *LICS '05 : Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science*, pages 31–40, Washington, DC, USA, 2005. IEEE Computer Society.

[BBFV05] Michael Benedikt, Angela Bonifati, Sergio Flesca, and Avinash Vyas. Verification of tree updates for optimization. In *CAV*, volume 3576, pages 379–393. Springer, 2005.

[BBG+02] Michael Benedikt, Glenn Bruns, Julie Gibson, Robin Kuss, and Amy Ng. Automated update management for XML integrity constraints. *Proc. Workshop on Programming Languages for XML (PLAN-X)*, 2002.

[BC09] Michael Benedikt and James Cheney. Schema-based independence analysis for XML updates. In *VLDB '09 : Proceedings of the 35th International Conference on Very Large Data Bases*, 2009.

[BCCM08] Véronique Benzaken, Giuseppe Castagna, Dario Colazzo, and Cédric Miachon. Pattern by example : type-driven visual programming of XML queries. In *PPDP '08 : Proceedings of the 10th international ACM SIGPLAN conference on Principles and practice of declarative programming*, pages 131–142, New York, NY, USA, 2008. ACM.

[BCL89] José A. Blakeley, Neil Coburn, and Per-Ake Larson. Updating derived relations : Detecting irrelevant and autonomously computable updates. *ACM Transactions on Database Systems*, 14(3) :369–400, 1989.

[BCU10] Nicole Bidoit, Dario Colazzo, and Federico Ulliana. Detecting XML query-update independence. In *Proc. 26èmes Journées Bases de Données Avancées, BDA*, 2010.

[BDF+01] Peter Buneman, Susan Davidson, Wenfei Fan, Carmem Hara, and Wang-Chiew Tan. Keys for xml. In *WWW '01 : Proceedings of the 10th international conference on World Wide Web*, pages 201–210, New York, NY, USA, 2001. ACM.

[BDF+02] Peter Buneman, Susan B. Davidson, Wenfei Fan, Carmem S. Hara, and Wang Chiew Tan. Reasoning about keys for XML. In *DBPL '01 : Revised Papers from the 8th International Workshop on Database Programming Languages*, pages 133–148, London, UK, 2002. Springer-Verlag.

[BDF+03] Peter Buneman, Susan Davidson, Wenfei Fan, Carmem Hara, and Wang-Chiew Tan. Reasoning about keys for XML. *Inf. Syst.*, 28(8) :1037–1063, 2003.

[BDFS84] Catriel Beeri, Martin Dowd, Ronald Fagin, and Richard Statman. On the structure of armstrong relations for functional dependencies. *Journal of the ACM*, 31 :30–46, 1984.

[BDM+06] Mikolaj Bojańczyk, Claire David, Anca Muscholl, Thomas Schwentick, and Luc Segoufin. Two-variable logic on data trees and XML reasoning. In *PODS '06 : Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 10–19, New York, NY, USA, 2006. ACM.

[BHL09] Béatrice Bouchou, Mírian Halfeld Ferrari, and Maria-Adriana Lima. Contraintes d'intégrité pour XML. visite guidée par une syntaxe homogène. *Technique et Science Informatiques*, 28(3) :331–364, 2009.

[BLT86] Jose A. Blakeley, Per-Ake Larson, and Frank Wm. Tompa. Efficiently updating materialized views. In *SIGMOD Conference*, pages 61–71, 1986.

[BMT89] Dina Bitton, Jeffrey Millman, and Solveig Torgersen. A feasibility and performance study of dependency inference. In *Proceedings of the Fifth*

tel-00580926, version 1 - 29 Mar 2011

International Conference on Data Engineering, pages 635–641, Washington, DC, USA, 1989. IEEE Computer Society.

[BÖB+04] Andrey Balmin, Fatma Özcan, Kevin S. Beyer, Roberta J. Cochrane, and Hamid Pirahesh. A framework for using materialized XPath views in XML query processing. In *VLDB '04 : Proceedings of the Thirtieth international conference on Very large data bases*, 2004.

[BPSM+08] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible markup language XML 1.0 (fifth edition), `http://www.w3.org/TR/2008/REC-xml-20081126/`., 26 November 2008.

[Büc60] J. R. Büchi. On a decision method in restricted second-order arithmetic. In *Internat. Congress Logic, Method, and Philos. Sci.*, pages 1–11. Press, Stanford, Calif.ACM, 1960.

[CDG+07] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on : `http://www.grappa.univ-lille3.fr/tata`, 2007. release October, 12th 2007.

[CM01] James Clark and Murata Makoto. Relax ng specification, committee specification 3 december 2001), `http://www.oasis-open.org/committees/relax-ng/spec-20011203.html`., 2001.

[Cod71] E. F. Codd. Further normalization of the data base relational model. *IBM Research Report, San Jose, California*, RJ909, 1971.

[Deb05] Denis Debarbieux. *Modélisation et requêtes des documents semi-structurés : exploitation de la structure de graphe*. Phd, Université des sciences et technologies de Lille, 2005.

[DESR03] K. Dimitrova, M. El-Sayed, and E. Rundensteiner. Order-sensitive view maintenance of materialized xquery views, 2003.

[DG08] Volker Diekert and Paul Gastin. First-order definable languages. In Jörg Flum, Erich Grädel, and Thomas Wilke, editors, *Logic and Automata : History and Perspectives*, volume 2 of *Texts in Logic and Games*, pages 261–306. Amsterdam University Press, 2008.

[DNSL05] Tuyet-Tram Dang-Ngoc, Virginie Sans, and Dominique Laurent. Classifying XML materialized views for their maintenance on distributed web sources. In *EGC*, pages 433–444, 2005.

[Don70] John Doner. Tree acceptors and some of their applications. 1970.

[DT05] Alin Deutsch and Val Tannen. Xml queries and constraints, containment and reformulation. *Theor. Comput. Sci.*, 336(1) :57–87, 2005.

[EF95] Heinz-Dieter Ebbinghaus and Jorg Flum. *finite model theory*. Springer Verlag, New York, USA, 1995.

tel-00580926, version 1 - 29 Mar 2011

tel-00580926, version 1 - 29 Mar 2011

[EH07] Joost Engelfriet and Hendrik Jan Hoogeboom. Automata with nested pebbles capture first-order logic with transitive closure. *CoRR*, abs/cs/0703079, 2007.

[EsMS06] Maged F. El-sayed, Prof Murali Mani, and Prof Jayavel Shanmugasundaram. Incremental maintenance of materialized xquery views. In *In ICDE*, page 129, 2006.

[EsWDR02] Maged El-sayed, Ling Wang, Luping Ding, and Elke A. Rundensteiner. An algebraic approach for incremental maintenance of materialized xquery views. In *In WIDM*, pages 88–91, 2002.

[Fan05] Wenfei Fan. Xml constraints : Specification, analysis, and applications. In *In Proc. DEXA*, pages 805–809, 2005.

[FF07] Fabio Fassetti and Bettina Fazzinga. Fox : Inference of approximate functional dependencies from XML data. *Database and Expert Systems Applications, International Workshop on*, 0 :10–14, 2007.

[FL02] Wenfei Fan and Leonid Libkin. On XML integrity constraints in the presence of dtds. *J. ACM*, 49(3) :368–406, 2002.

[FNTT07] Emmanuel Filiot, Joachim Niehren, Jean-Marc Talbot, and Sophie Tison. Polynomial time fragments of XPath with variables. In *PODS '07 : Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 205–214, New York, NY, USA, 2007. ACM.

[FS00] Wenfei Fan and Jérôme Siméon. Integrity constraints for XML. In *PODS '00*, pages 23–34, New York, NY, USA, 2000. ACM.

[GKP02] Georg Gottlob, Christoph Koch, and Reinhard Pichler. Efficient algorithms for processing XPath queries. In *In VLDB*, pages 95–106, 2002.

[GL95] Timothy Griffin and Leonid Libkin. Incremental maintenance of views with duplicates. pages 328–339, 1995.

[GM95] A. Gupta and I. S. Mumick. Maintenance of materialized views : Problems, techniques, and applictions. In *Proc. 24th Int. Conf. Very Large Data Bases, VLDB*, 1995.

[GMS93] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *ACM SIGMOD international conference on Management of data*, pages 157–166. ACM, 1993.

[GO99] Erich Grädel and Martin Otto. On logics with two variables. *Theoretical Computer Science*, 224 :73–113, 1999.

[Gol90] Charles F. Goldfarb. *The SGML handbook*. Oxford University Press, Inc., New York, NY, USA, 1990.

[GRS07] Giorgio Ghelli, Kristoffer Hogsbro Rose, and Jérôme Siméon. Commutativity analysis in XML update languages. In *ICDT*, pages 374–388, 2007.

[Hal07] Mírian Halfeld Ferrari Alves. *Les aspects dynamiques de XML et les services web*. Habilitation à diriger des recherches, Université François Rabelais de Tours, 2007.

[HKL+08] S. Hartmann, H. Kohler, S. Link, T. Trinh, and J. Wang. On the notion of an XML key. In *SDKB 2008*, volume 4925, pages 114–123. LNCS - Springer, 2008.

[HP01] Haruo Hosoya and Benjamin Pierce. Regular expression pattern matching for xml. *SIGPLAN Not.*, 36(3) :67–80, 2001.

[Kel04] Uwe Keller. Some remarks on the definability of transitive closure in first-order logic and datalog, 2004.

[Kle56] S. Kleene. Representation of events in nerve nets and finite automata. *In C. Shannon and J. McCarthy, editors, Automata Studies*, 3(41), 1956.

[LD00] Hartmut Liefke and Susan B. Davidson. View maintenance for hierarchical semistructured data. In *Data Warehousing and Knowledge Discovery*, pages 114–125, 2000.

[Leo06] LIBKIN Leonid. Logics for unranked trees : An overview. *Logical Methods in Computer Science*, 2 :1–31, 2006.

[Lim07] Maria Adriana Lima. *Maintenance incrémentale des contraintes d'intégrité en XML*. Phd, Université François Rabelais de Tours, France, 2007.

[LLL02] Mong-Li Lee, Tok Wang Ling, and Wai Lup Low. Designing functional dependencies for xml. In *EDBT '02 : Proceedings of the 8th International Conference on Extending Database Technology*, pages 124–141, London, UK, 2002. Springer-Verlag.

[LLSV01] D. Laurent, J. Lechtenbörger, N. Spyratos, and G. Vossen. Monotonic complements for independent data warehouses. *The VLDB Journal*, 10(4) :295–315, 2001.

[LWZ06] Laks V. S. Lakshmanan, Hui Wang, and Zheng Zhao. Answering tree pattern queries using views. In *VLDB '06 : Proceedings of the 32nd international conference on Very large data bases*, 2006.

[Mar04a] Maarten Marx. Conditional XPath , the first order complete XPath dialect. In *PODS '04 : Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 13–22, New York, NY, USA, 2004. ACM.

[Mar04b] Maarten Marx. Xpath with conditional axis relations. *LNCS Advances in Database Technology - EDBT 2004*, 2992/2004 :579–580, 2004.

tel-00580926, version 1 - 29 Mar 2011

[Mar05a] Maarten Marx. Conditional XPath. *ACM Trans. Database Syst.*, 30(4) :929–959, 2005.

[Mar05b] Maarten Marx. First order paths in ordered trees. In *In ICDT'05*, pages 114–128. Springer, 2005.

[Md05] Maarten Marx and Maarten de Rijke. Semantic characterizations of navigational XPath. *SIGMOD Rec.*, 34(2) :41–46, 2005.

[MLMK05] Makoto Murata, Dongwon Lee, Murali Mani, and Kohsuke Kawaguchi. Taxonomy of XML schema languages using formal language theory. *ACM Trans. Internet Technol.*, 5(4) :660–704, 2005.

[MNSC04] Wim Martens, Frank Neven, Thomas Schwentick, and Limburgs Universitair Centrum. Complexity of decision problems for simple regular expressions. In *In Proceedings of the 29th International Symposium on Mathematical Foundations of Computer Science (MFCS 2004)*, pages 889–900. Springer, 2004.

[Mor75] M. Mortimer. On languages with two variables. *Logik Grundlagen Math*, 21 :135–140, 1975.

[MS04] G. Miklau and D. Suciu. Containment and equivalence for a fragment of XPath. *ACM*, 51 :2–45, 2004.

[Mur98] Makoto Murata. Data model for document transformation and assembly (extended abstract). In *In Proceedings of the workshop on Principles of Digital Document Processing*, pages 140–152. Springer-Verlag, 1998.

[Nev02] Frank Neven. Automata, logic, and xml. In *In CSLâ02 - Annual Conference of the European Association for Computer Science Logic (invited talk*, pages 2–26. Springer, 2002.

[Nil06] Shirish K. Nilekar. *Self Maintenance of Materialized XQuery Views via Query Containment and Re-Writing.* Master thesis report, Worcester Polytechnic Institute, 2006.

[NS00] Frank Neven and Thomas Schwentick. Expressive and efficient pattern languages for tree-structured data (extended abstract). In *PODS '00 : Proceedings of the nineteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 145–156, New York, NY, USA, 2000. ACM.

[NS02] Frank Neven and Thomas Schwentick. Query automata over finite trees. *Theor. Comput. Sci.*, 275(1-2) :633–674, 2002.

[NSsuJ01] Frank Neven, Thomas Schwentick, and Friedrich schiller-universitat Jena. Automata- and logic-based pattern languages for tree-structured data. pages 160–178. Springer, 2001.

tel-00580926, version 1 - 29 Mar 2011

[OCMH03] Makoto Onizuka, Fong Yee Chan, Ryusuke Michigami, and Takashi Honishi. Incremental maintenance for materialized XPath/XSLT views. In *WWW*, 2003.

[QCR00] L. Quan, L. Chen, and E. Rundensteiner. Efficient refresh in an XQL-based web caching system, 2000.

[RS06] M. Raghavachari and O. Shmueli. Conflicting XML updates. In *Advances in Database Technology - EDBT*, volume 3896, pages 552–569, 2006.

[San07] Virginie Sans. Maintenance de vues XML matérialisées à partir de sources web. *Technique et Science Informatiques*, 26(6) :751–780, 2007.

[San08] Virginie Sans. *Maintenance de vues XML matérialisées à partir de sources web non coopérantes*. Phd, Université de Cergy-Pontoise, 2008.

[SLT91] Marc H. Scholl, Christian Laasch, and Markus Tresch. Updatable views in object-oriented databases. In *Proc. 2nd Intl. Conf. on Deductive and Object-Oriented Databases (DOOD)*, number 566, 1991.

[STP+05] Arsany Sawires, Junichi Tatemura, Oliver Po, Divyakant Agrawal, and K. SelÇuk Candan. Incremental maintenance of path-expression views. In *ACM SIGMOD international conference on Management of data*, pages 443–454, 2005.

[STP+06] Arsany Sawires, Junichi Tatemura, Oliver Po, Divyakant Agrawal, Amr El Abbadi, and K. Selçuk Candan. Maintaining XPath views in loosely coupled systems. In *VLDB '06 : Proceedings of the 32nd international conference on Very large data bases*, pages 583–594. VLDB Endowment, 2006.

[Suc98] Dan Suciu. Semistructured data and xml, 1998.

[SV05] Luc Segoufin and Victor Vianu. Views and queries : Determinacy and rewriting. In *Symposium on Principles of Database Systems (PODS)*, 2005.

[ten06] Balder ten Cate. The expressivity of XPath with transitive closure. In *PODS '06 : Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 328–337, New York, NY, USA, 2006. ACM.

[tS08] Balder ten Cate and Luc Segoufin. Xpath, transitive closure logic, and nested tree walking automata. In *PODS '08 : Proceedings of the twenty-seventh ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 251–260, New York, NY, USA, 2008. ACM.

[Tur37] Alan M. Turing. Computability and lambda-definability. *J. Symb. Log.*, 2(4) :153–163, 1937.

tel-00580926, version 1 - 29 Mar 2011

[TW68] J. W. Thatcher and J. B. Wright. Generalized finite automata theory with an application to a decision problem of second-order logic. In *Theory of Computing Systems*, pages 57–81. Springer New York, mars 1968.

[Vis96] Dimitra Vista. *Optimizing Incremental View Maintenance Expressions In Relational Databases*. Phd, University of Toronto, 1996.

[VL03] Millist W. Vincent and Jixue Liu. Functional dependencies for xml. In *APWeb*, pages 22–34, 2003.

[VL05] Millist W. Vincent and Jixue Liu. Checking functional dependency satisfaction in xml. In *XSym*, pages 4–17, 2005.

[VLL04] Millist W. Vincent, Jixue Liu, and Chengfei Liu. Strong functional dependencies and their application to normal forms in XML. *ACM Trans. Database Syst.*, 29(3) :445–462, 2004.

[W3C] World wide web consortium . Available on : `http://www.w3.org/`.

[XPa99] WWW Consortium XPath. XML path language(XPath) version 1.0. 16 Novembre 1999.

[XPa07] WWW Consortium XPath 2.0. XML path language(XPath) version 2.0. 23 January 2007.

[ZGM98] Yue Zhuge and Hector Garcia-Molina. Graph structured views and their incremental maintenance. In *Proc. 14th IEEE Conf. Data Engineering, ICDE*, pages 116–125. IEEE Computer Society, 1998.

[ZZRR02] Xin Zhang, Xin Zhang, Elke A. Rundensteiner, and Elke A. Rundensteiner. Xat : XML algebra for the rainbow system. Technical report, 2002.

tel-00580926, version 1 - 29 Mar 2011

tel-00580926, version 1 - 29 Mar 2011

Table des figures

tel-00580926, version 1 - 29 Mar 2011

tel-00580926, version 1 - 29 Mar 2011

Annexe

Démonstration. (Proposition 6) On procède par récurrence sur la taille (égale au nombre de nœuds) de $\mathscr{T}$.

cas de base :
$\mathscr{T} = (\epsilon = [\mathcal{E}_{(\epsilon,0)} \rightarrow 0];)$. On définit P de manière inductive sur la structure de l'expression régulière $\mathcal{E}_{(\epsilon,0)}$ par la fonction récursive calculeP donnée dans l'algorithme 3.

Algorithm 3 Algorithm fonction calculeP

Require: un template $\mathscr{T} = (\epsilon = [\mathcal{E}_{(\epsilon,0)} \rightarrow 0];)$.
sortie : Le chemin P de RegularXPath qui vérifie $\forall$ n $\in \mathcal{D}$, n $\vDash_{\mathcal{D}} \mathscr{T} \Leftrightarrow n \in [\![< P >]\!]$
$\mathcal{E} \leftarrow \mathcal{E}_{(\epsilon,0)}$
function $calculeP(\mathcal{E})$
begin
if $\mathcal{E} = \lambda$ **then**
P $\leftarrow child/?\lambda$
end if
if $\mathcal{E} = \mathcal{E}_0^*$ **then**
P $\leftarrow (calculeP(\mathcal{E}_0))^*$
end if
if $\mathcal{E} = \mathcal{E}_1|\mathcal{E}_2$ **then**
P $\leftarrow calculeP(\mathcal{E}_1) \cup calculeP(\mathcal{E}_2)$
end if
if $\mathcal{E} = \mathcal{E}_1.\mathcal{E}_2$ **then**
P $\leftarrow calculeP(\mathcal{E}_1)/calculeP(\mathcal{E}_2)$
end if
return P
end

Etape d'induction : Soit un template $\mathscr{T}$ de taille (n+1), $\mathscr{T}$ s'écrit sous la forme présentée en Figure 5.1 :
Les templates $\mathscr{T}_0$,...,$\mathscr{T}_k$ sont de taille $<$ (n+1), donc d'après l'hypothèse de récurrence il existe des chemins de RegularXPath P_1,..., P_k qui vérifient : pour tout i $\in$

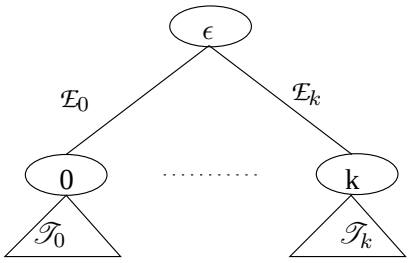


FIGURE 5.1 – Template $\mathscr{T}$

[1,k], $\forall$ n $\in \mathcal{D}$, n$\vDash_{\mathcal{D}} \mathscr{T}_i \Leftrightarrow n \in [\![< P_i >]\!]$.
Soit $\vec{p}$=$(p_0, p_1, ..., p_k)$ un k-uplet de lettres, constitué de tous les préfixes possibles de longueur 1 des langages L($\mathcal{E}_0$),L($\mathcal{E}_1$),...,L($\mathcal{E}_k$). On note $\mathscr{T}_{\vec{p}}$ le template représenté en figure 5.2. Clairement le template $\mathscr{T}$ est équivalent à l'union finie des templates $\mathscr{T}_{\vec{p}}$ lorsque $\vec{p}$ prend toutes les valeurs possibles.
Soit h_i i $\in$ [1,k], les expressions de chemin de RegularXPath équivalentes aux expressions régulières $p_i^{-1}.\mathcal{E}_i$: h_i=calculeF$(p_i^{-1}.\mathcal{E}_i)$, $\forall i \in [1, k]$.
L'expression de chemin $P_{\vec{p}}$ de RegularXPath qui vérifie $\forall$ n $\in \mathcal{D}$, n $\vDash_{\mathcal{D}} \mathscr{T}_{\vec{p}} \Leftrightarrow n \in [\![< P_{\vec{p}} >]\!]$. est donnée par :
$P_{\vec{p}}$=$< /child :: p_0[h_0[F_0]][fs :: p_1[h_1[F_1]]...[fs :: p_k[h_k[F_k]...] >$.
L'union des $P_{\vec{p}}$ donne le résultat pour $\mathscr{T}$. □

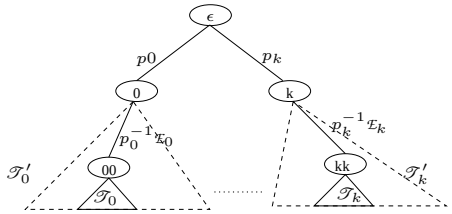


FIGURE 5.2 – Template $\mathscr{T}_{\vec{p}}$

Démonstration. (Théorème 9) On démontre le théorème par récurrence sur le nombre de nœuds n de $\mathscr{T}$.

Cas de base : n=1 l'expression équivalente est '/'.

Étape d'induction : Supposons que chaque requête monadique RAR de taille n soit équivalente à une expression de chemin de RegularXPath et montrons que ce résultat est vrai pour une requête $\mathscr{R}$ de n+1 nœuds.
La requête $\mathscr{R}$ peut s'écrire sous la forme présentée en Figure 5.3 où σ est le nœud père du nœud de sélection : On note par $\mathscr{T}'$ (voir figure 5.3) le template construit à partir de $\mathscr{T}$ en supprimant le sous template $\mathcal{U}$ issu du nœud σ ; d'autre part le

tel-00580926, version 1 - 29 Mar 2011

template $\mathcal{U}$ sera représenté comme pour la proposition 6. Comme pour la proposition 6, la requête $\mathscr{R}$ est équivalente à l'union des requêtes $\mathscr{R}_{\vec{p}}$ pour tous les k-uplet $\vec{p} = (p_0, ..., p_k)$ possibles de préfixes de longueur 1 des expressions régulières $\mathcal{E}_0, .., \mathcal{E}_k$ (Figure 5.4).
D'après la proposition 6, il existe des expressions de nœuds de RegularXPath P_0 ,...,P_k tel que pour tout i, $\forall$ n $\in \mathcal{D}$, n$\vDash_{\mathcal{D}} \mathscr{T}_i \Leftrightarrow n \in [\![< P_i >]\!]$.
Soit la requête $\mathscr{R}'$=($\mathscr{T}'$,σ) : la taille de cette requête $\mathscr{R}'$ est inférieur à n donc d'après l'hypothèse de récurrence $\mathscr{R}'$ est exprimable par une expression de chemin P' de RegularXPath.
Soit h_i i $\in$ [1,k], les expressions de chemin de RegularXPath équivalentes aux expressions régulières $p_i^{-1}.\mathcal{E}_i$.
Alors le chemin
$P_{\vec{p}}$=P'/child : :p_i[H_1][H_2]/h_i[$< P_i >$] est un chemin de RegularXPath équivalent à $\mathscr{R}_{\vec{p}}$, avec
H_1=ps : :p_{i-1}[h_{i-1}[F_{i-1}]...[ps : :p_0 [h_0[F_0]...
H_2=fs : :p_{i+1} [h_{i+1}[F_{i+1}]...[fs : :p_k [h_k[F_k]...
Ainsi l'union des chemin $P_{\vec{p}}$ est une expression de chemin de RegularXPath qui est équivalente à $\mathscr{R}$.

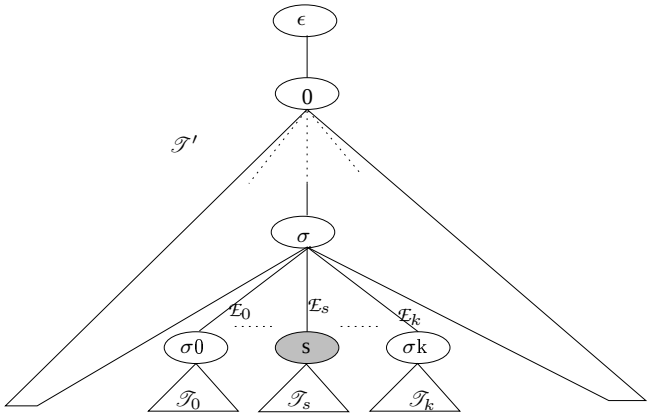


FIGURE 5.3 – Requête $\mathscr{R}$

□

tel-00580926, version 1 - 29 Mar 2011

tel-00580926, version 1 - 29 Mar 2011

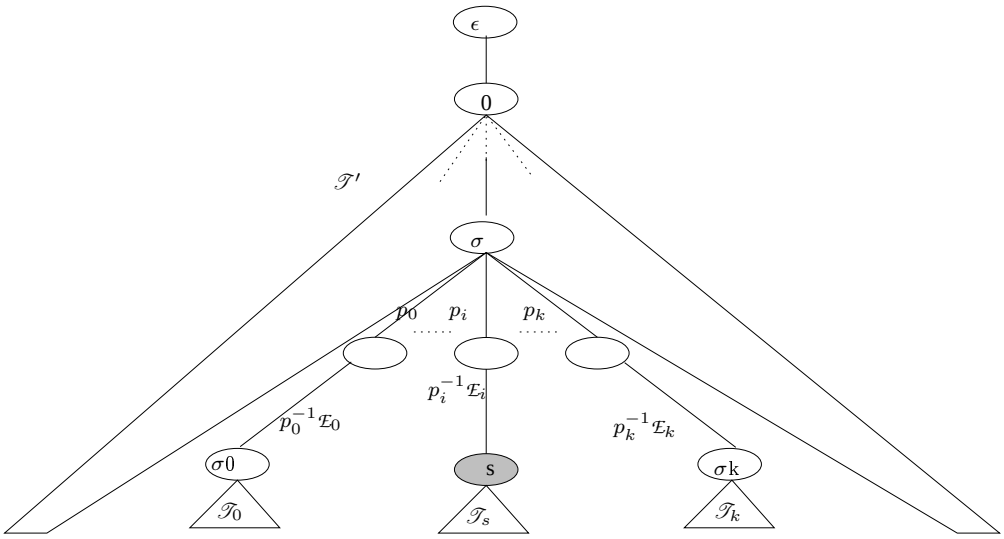


FIGURE 5.4 – Requête $\mathscr{R}_p$