Comparaison et évolution de schémas XML

Joshua Amavi

▶ **To cite this version:**

Joshua Amavi. Comparaison et évolution de schémas XML. Ordinateur et société [cs.CY]. Université d'Orléans, 2014. Français. NNT : 2014ORLE2053 . tel-01171845

HAL Id: tel-01171845
https://theses.hal.science/tel-01171845

Submitted on 6 Jul 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Centre Val de Loire UNIVERSITÉ
Spirit of the Loire Valley

UNIVERSITÉ D'ORLÉANS

UNIVERSITE D'ORLEANS

ÉCOLE DOCTORALE MIPTIS

MATHÉMATIQUES, INFORMATIQUE, PHYSIQUE THÉORIQUE ET INGÉNIEURIE DES SYSTÈMES

Laboratoire d'Informatique Fondamentale d'Orléans

THÈSE

présentée par :

Joshua AMAVI

soutenue le : **28 Novembre 2014**

pour obtenir le grade de : **Docteur de l'université d'Orléans**

Discipline/ Spécialité : **Informatique**

Comparaison et évolution de schémas XML

THÈSE DIRIGÉE PAR :

Mirian HALFELD FERRARI	Professeur des Universités, Université d'Orléans
Pierre RÉTY	Maître de Conférence HDR, Université d'Orléans

RAPPORTEURS :

François GOASDOUÉ	Professeur des Universités, Université de Rennes 1
Sophie TISON	Professeur des Universités, Université de Lille

JURY :

Véronique BENZAKEN	Professeur des Universités, Université Paris-Sud - *Président*
Jacques CHABIN	Maître de Conférence, Université d'Orléans
Michaël RUSINOWITCH	Directeur de Recherche, INRIA Nancy

À mes parents Valère et Cécile
À mes sœurs chéries Addis et Merveille
À mes amours Éssé et Sydney-Prude
À mon cher oncle Prudencio, tu me manques énormément

Remerciements

C'est avec une immense joie et beaucoup de plaisir que j'écris les lignes qui suivent pour remercier tous ceux et toutes celles qui ont contribué de près et de loin au bon déroulement de ma thèse. Cette partie de ma thèse étant aussi difficile à rédiger que les travaux qui vont suivre, j'espère avoir oublié personne.

Je m'y lance donc en commançant par mes deux directeurs de thèse Mirian et Pierre. Je vous remercie infiniment pour votre encadrement, vos encouragements dans la bonne humeur, vos conseils et toute la patiente que vous m'avez accordée. Vous avez su trouver les bons mots pour me guider et m'éviter les pistes douteuses. Il n'y a pas de mots adéquat pour vous exprimer toute ma gratitude.

J'adresse mes sincères remerciements à tous les membres de mon jury de thèse pour avoir accepté de s'investir dans cette tâche : à Véronique Benzaken, qui m'a fait l'honneur de présider ma soutenance ; à François Goasdoué et à Sophie Tison, qui ont accepté d'être mes rapporteurs et dont les commentaires et critiques ont été d'une grande aide ; à Jacques Chabin et à Michaël Rusinowitch, qui ont accepté d'être mes examinateurs.

Je tiens à remercier Agata, Béatrice, et Jacques, dont les collaborations, aides et conseils ont été très précieux dans l'aboutissement de ma thèse.

J'adresse mes sincères remerciements à tous les membres du Laboratoire d'Informatique Fondamentale d'Orléans, du département informatique et de l'IUT informatique. Je remercie particulièrement :

- l'équipe PaMDA qui m'a accueilli ;
- le directeur du LIFO, Jérôme ;
- le nouveau et l'ex directeur du pôle informatique du Collégium Sciences et Techniques, Cathérine et Ali ;
- le nouveau et l'ex directeur du département informatique de l'IUT, Emmanuel et Sébastien ;
- Ioan ancien directeur de l'école doctorale MIPTIS pour ses conseils ;
- Isabelle et Florence pour leur aides dans les tâches administratives ;
- les collègues avec qui j'ai eu le plaisir d'enseigner et d'échanger ;
- mes collègues doctorants du LIFO.

Mes remerciements vont aussi à ceux avec qui j'ai eu à partager mon bureau : Mouhamadou, Simone, Thomas, Vincent et aussi à Abdel qui vient souvent me voir à mon bureau. Nous avons partagé de bons moments.

Je suis extrêmement reconnaissant à mes ami(e)s, frères et sœurs d'Orléans : Adjé et Olivia, Alain, Albéric, Brice, Camille, Daniel, Dietrich, Edouard, Elie, Fabrice, Francine, Jarib, Juste, Lala, Natacha, Nikoué, Oumou, Sandry, Sélom, Sitsopé, Sony, Rita, Thierry, Victorine et André qui m'ont toujours témoigné d'un attachement et d'une affection sincère

et inébranlable.

Un grand merci à ma moitié Éssé qui était à mes côtés dès le début de ma thèse. Elle m'a soutenu tout au long de cette thèse aussi bien dans les moments de joie comme ceux de doute. J'avoue que sans toi, j'aurais eu la tâche plus difficile. Merci pour ton soutien, Merci pour tes conseils, Merci pour m'avoir supporté, Merci pour tes sacrifices et Merci pour ton amour. Je remercie aussi mes beaux parents Jean et Mélanie.

Pour terminer j'adresse mes remerciements à toute ma famille. Merci à mes parents Valère et Cécile, sans qui je ne serai pas là, à écrire ce manuscrit. Merci à vous pour votre éducation, pour vos encouragements et soutiens, pour m'avoir permis de grandir dans cette belle famille. Je remercie mon oncle Prudencio et sa femme Euphrasie pour m'avoir accueilli à Orléans pour mes études et pour avoir assuré le rôle de parents. Merci à toi Prudencio qui a partagé toutes mes années d'études universitaires y compris celles de la thèse et vraiment dommage que tu n'a pas pu rester quelques mois de plus pour fêter la fin de ma thèse avec moi. Tu es tout le temps dans mes pensées et je sais que tu seras aussi heureux que moi le jour de ma soutenance. Merci à mes sœurs Addis, qui a fait le déplacement du Canada pour assister à ma soutenace, et Merveille. Merci à mes tantes et oncles : Liliane (alias tatata), Centy, Davi, Mamia, Franck Gaba, May, Irène et Faustin, François Banabaya, Kwotcho et Gilles, Ivy, Dédé, Yolande, ... Merci à mes cousins et cousines : Rémus et Romulus, Sicio et Christelle, Annick et Claude, Didier, Christina, Mila, Carine, Daki, Dévi, Pamela, Hénoc, Maurile, ... Je vous adore tous. Que les involontaires oubliés me pardonnent !

Table des matières

Table des matières **vii**

Liste des figures **ix**

1 Introduction **1**
- 1.1 Contributions . . . 3
  - 1.1.1 Comparaison des schémas de données . . . 4
  - 1.1.2 Intégration de dépendances fonctionnelles . . . 5
  - 1.1.3 La correction d'un document par rapport à un schéma XML . . . 6
  - 1.1.4 L'évolution des schémas et l'adaptation des documents XML . . . 6
  - 1.1.5 Contributions logicielles . . . 7
- 1.2 Structure du manuscrit . . . 7

2 Arbres et schémas XML **9**
- 2.1 Documents XML . . . 9
- 2.2 Schémas XML . . . 12

3 Inclusion relâchée de schémas XML **17**
- 3.1 Préliminaires . . . 19
- 3.2 Inclusion relâchée pour les Regular Tree Grammars . . . 20
  - 3.2.1 Cas de grammaire non récursive . . . 20
  - 3.2.2 Cas de grammaire récursive . . . 22
- 3.3 Implémentation et Résultats Expérimentaux . . . 28
- 3.4 Travaux liés . . . 31
- 3.5 Conclusion et perspectives . . . 32

4 Intégration de dépendances fonctionnelles XML **35**
- 4.1 Introduction . . . 35
- 4.2 Motivation . . . 37
- 4.3 Préliminaires . . . 39
  - 4.3.1 Langage de chemins linéaire . . . 40
  - 4.3.2 Branches dans un arbre . . . 44
  - 4.3.3 Informations incomplètes et arbre complet . . . 46
- 4.4 Dépendances Fonctionnelles pour XML . . . 48
  - 4.4.1 Définition . . . 48
  - 4.4.2 Satisfaction de XFD . . . 50
- 4.5 Raisonner sur les dépendances fonctionnelles XML . . . 52
  - 4.5.1 Système d'axiomes . . . 53
  - 4.5.2 Axiomes additionnels . . . 56
  - 4.5.3 Dépendances triviales . . . 57
  - 4.5.4 Fermeture d'un ensemble de chemins . . . 58

4.6 Fermeture d'un ensemble de chemins . . . 59
4.6.1 Algorithme pour la fermeture d'un ensemble de chemins . . . 59
4.6.2 Analyse de la complexité . . . 61
4.6.3 Algorithme pour la fermeture d'un ensemble de dépendances . . . 62
4.6.4 Notion de couverture . . . 62
4.7 Dépendances Fonctionnelles pour l'interopérabilité . . . 63
4.7.1 Méthode pour calculer l'ensemble maximal de XFD $\mathcal{F}$ . . . 64
4.7.2 Vers une version optimisée . . . 68
4.7.3 Méthode pour calculer une couverture de $\mathcal{F}$ . . . 68
4.7.4 Calcul de la couverture de $\mathcal{F}$: Preuve de correction . . . 75
4.7.5 Analyse de la complexité de l'Algorithme 5 . . . 80
4.8 Résultats Expérimentaux . . . 81
4.8.1 Jeu de données . . . 81
4.8.2 Résultats . . . 82
4.9 État de l'art . . . 84
4.10 Conclusion . . . 86

5 Correction d'un document XML par rapport à un schéma 87
5.1 Préliminaires . . . 88
5.2 Opérations sur les arbres XML . . . 92
5.3 Correction d'un document XML . . . 97
5.3.1 Calcul de distance entre deux arbres . . . 97
5.3.2 Calcul des corrections pour un mot dans un dictionnaire de mots . . . 99
5.3.3 Exemple de correction d'un document XML . . . 101
5.3.4 Présentation de l'algorithme de correction d'un document XML . . . 106
5.3.5 Complexité . . . 112
5.4 Discussion, adaptations et extensions . . . 114
5.5 Résultats expérimentaux . . . 117
5.6 Travaux liés à la correction de document . . . 124
5.6.1 Mesurer la distance entre un document XML et un schéma . . . 125
5.6.2 Trouver une seule correction, ou toutes les corrections de coût minimal . . . 127
5.6.3 Calculer un ensemble de séquences d'opérations d'édition pour corriger des documents XML vers un schéma . . . 128
5.6.4 La revalidation et la correction après des mises à jour du document ou du schéma . . . 128
5.7 Conclusion . . . 129

6 Mapping entre schémas XML 133
6.1 Langage pour le mapping de schémas . . . 134
6.1.1 Représentation sous forme d'arbres des règles de production . . . 134
6.1.2 Opérations d'édition de grammaire . . . 136
6.1.3 Script d'édition de grammaire d'arbres et définition de mapping . . . 140
6.2 Les opérateurs de mapping de schémas . . . 142
6.3 Cas d'applications du mapping de schémas (*MappingGen*) . . . 144
6.3.1 Évolution conservatrice de schémas . . . 144
6.3.2 Substitution de schéma . . . 148
6.4 Adaptation de documents XML guidée par le mapping (*XTraM*) . . . 150
6.5 État de l'art . . . 157
6.5.1 Mapping de schémas basé sur le formalisme logique . . . 157
6.5.2 Mapping de schémas basé sur l'évolution incrémentale des schémas . . . 158

6.6 Discussions et remarques 159

7 Les solutions logicielles **161**
7.1 *XMLCorrector* 161
7.2 *DTDGrabber* 164

8 Conclusion **169**

Bibliographie **173**

A Annexe générale **187**
A.1 Preuve de correction du calcul de la grammaire de $WI(L(G))$ 187
A.1.1 Terminaison 187
A.1.2 Correction ($L(G') \subseteq WI(L(G))$) 188
A.1.3 Complétude ($L(G') \supseteq WI(L(G))$) 190
A.2 Automate d'états finis pour un chemin P dans PL 194
A.3 Calcul du plus long préfixe commun 195
A.4 Système d'axiomes : Preuve de correction 197
A.5 Système d'axiomes : Preuve de complétude 204
A.6 Algorithme pour la fermeture d'un ensemble de chemins : Preuve de correction 212
A.7 Preuves de correction et de complétude de l'algorithme de correction 214
A.8 Tableaux de comparaison 220

Liste des figures

1.1 Exemple d'application où des systèmes locaux $S_1, \ldots, S_n$ évoluent en un seul système global S. 2
1.2 Application de composition de services web 4
1.3 Intégration de dépendances fonctionnelles. 5
1.4 Correction de document XML. 6
1.5 Échange de données entre deux schémas. 7

2.1 Exemple de document XML et son arbre 9
2.2 Document XML sous forme d'arbre 11
2.3 Tableau résumé des types des grammaires 14

3.1 Arbres XML t et t' valides par rapport à G et G', respectivement. 18
3.2 Cas 1 : Exemple de grammaires pour le cas non récursif 21
3.3 Cas 2 : Exemple de grammaires pour le cas récursif 21
3.4 Cas 3 : Exemple de grammaires pour le cas récursif 21
3.5 Exemple de graphe issu de la relation $>$ pour une grammaire RTG. 29

4.1 Système global S obtenu à partir d'une évolution conservatrice des systèmes locaux 37

4.2 Deux documents XML de différentes sources locaux. 38
4.3 Automates associé à un schéma et un chemin PL 42
4.4 Document XML avec des informations manquantes 47
4.5 Exemple d'arbres complets et d'arbres non complets 47
4.6 Documents XML concernant les diplômes Licence dans une université 49
4.7 Documents XML violant la dépendance XFD4 52
4.8 Deux schémas représentant les mêmes données de façon différentes 64
4.9 Schéma de l'application pour calculer $\mathcal{F}$. 66
4.10 Résultat du Corollaire 4.1. 68
4.11 Arbre motif 81
4.12 Arbre $\mathcal{T}$ construit en répétant n fois l'arbre motif 82
4.13 Scénario 1 : Temps CPU nécessaire pour le calcul de $cover\mathcal{F}$ et l'évolution de sa taille 83
4.14 Scénario 2 : Temps CPU nécessaire pour le calcul de $cover\mathcal{F}$ et l'évolution de sa taille 84

5.1 Exemple d'arbre XML 89
5.2 Exemple de sous-arbre et d'arbre partiel 89
5.3 Un exemple de description de schéma. 90
5.4 Résultat de la dérivation pour une opération add sur un arbre 94
5.5 Arbre XML produit par la dérivation de la séquence de l'Exemple 5.7 95
5.6 Arbre partiels et le calcul d'une case de ma matrice de la distance d'édition . . 98
5.7 Exemple de matrice pour la distance d'édition entre deux arbres 99
5.8 Exploration d'un automate pour la correction d'un mot par rapport à un langage 100
5.9 Contenu de la matrice M 102
5.10 Nouvelle matrice calculée par l'appel récursif. 103
5.11 Illustration du calcul d'une cellule de la matrice d'édition 104
5.12 Trois candidats possibles t'_1, t'_2 et t'_3 pour l'arbre t de la Figure 5.1. 104
5.13 Contenu de la matrice M pour le mot $u = abbbb$ 105
5.14 Contenu de la matrice M pour le mot $u = abbc$ 105
5.15 Contenu de la matrice M pour le mot $u = abc$ après un autre retour en arrière 105
5.16 La DTD correspondante au fichier XML utilisé pour les expérimentations . . . 117
5.17 Algorithme de correction : cas de scénario 1 118
5.18 Algorithme de correction : cas de scénario 2 119
5.19 Algorithme de correction : cas de scénario 3 120
5.20 Algorithme de correction : cas de scénario 4-1 121
5.21 Algorithme de correction : cas de scénario 4-2 122
5.22 Algorithme de correction : cas de scénario 5 123
5.23 Algorithme de correction : cas de scénario 6 124

6.1 Représentation sous forme d'arbre d'une règle de production 135
6.2 Arbres bien formés et mal formés 136
6.3 Exemple d'*opérations d'édition élémentaires sur les grammaires RTG*. 138
6.4 Exemple d'*opérations d'édition non-élémentaires sur les grammaires RTG*. 141
6.5 Utilisation de la composition et de l'inversion pour faciliter l'évolution de schéma. 143
6.6 Les DTDs des services d'un hôpital 145
6.7 DTD global pour l'hospital 146
6.8 Exemple d'arbres valides par rapport aux schémas de l'hôpital 146

6.9 RTG obtenue après la fusion de toutes les règles de production des grammaires locales. 147
6.10 LTG obtenue par l'algorithme dans [42] à partir de la RTG de la Figure 6.9. . . 147
6.11 Un arbre XML valide et annoté par rapport au schéma global de la Figure 6.10. 150
6.12 Un arbre XML valide et annoté par rapport au schéma de la Figure 6.9. 152
6.13 Exemple d'utilisation des fonctions *updateNodePositions* et *updateForestPositions*. 154

7.1 Illustration de *XMLCorrector*. 161
7.2 Fenêtre principale de *XMLCorrector*. 162
7.3 Contenu de l'onglet résultat. 163
7.4 Architecture de l'application *DTDGrabber*. 165
7.5 Fenêtre principale de *DTDGrabber*. 166
7.6 Arbre des chemins possibles pour les documents associés à un schéma XML. 167
7.7 Visualisation des modifications apportées par un mapping sur un schéma XML. 167
7.8 Fenêtre de création d'une opération d'édition. 168

A.1 Example for the tree $t \in L_G^{nt}(A)$ with $f = (t_1, \cdots, t_n)$ 192
A.2 Example of the two possibilities for the tree $t' \lhd t$, with $f' = (f'_1, \cdots, f'_n)$ and $[f'_i = A_i(f''_i)$ or $f'_i = f''_i]$ 193
A.3 Automate associé au plus long préfixe commun 197
A.4 Trois situations obtenues à partir des conditions du Lemme A.11. 198
A.5 Représentation graphique des chemins et des projections possibles (preuve Axiome4). 202
A.6 Un arbre XML montrant l'impossibilité d'avoir toujours deux instances pour tous chemins P et Q tels que $P \prec Q$ et $Last(I_{P_1}) =_V Last(I_{P_2})$ et $Last(I_{Q_1}) \neq_V Last(I_{Q_2})$ 205
A.7 Construction d'un nouveau arbre bi-instance t' avec $Q_1[E''] \in (C, X[\vec{E}])^+$ . . . 207
A.8 Construction d'un nouveau arbre bi-instance t' avec $Q_1[E''] \notin (C, X[\vec{E}])^+$ . . . 208
A.9 L'ajout d'une instance de P dans l'arbre de (a) implique $|Instances(Q, t)| \geq 3$ pour l'arbre de (b) 209
A.10 Illustration of the two cases (a)$v^1_{i,k+1} = v^2_{i,k+1}$ and (b)$v^1_{i,k+1} \neq v^2_{i,k+1}$. 211
A.11 Example of matrix representation for the proof 215
A.12 Combining operation sequences in case of $i = 0$ 216
A.13 Combining operation sequences in the first column of the distance matrix . . . 217
A.14 Combining operation sequences in case of a diagonal correction with $w = u_1 \dots u_{j-1}$ 218

Introduction

1

Sommaire

1.1 Contributions . . . 3
1.1.1 Comparaison des schémas de données . . . 4
1.1.2 Intégration de dépendances fonctionnelles . . . 5
1.1.3 La correction d'un document par rapport à un schéma XML . . . 6
1.1.4 L'évolution des schémas et l'adaptation des documents XML . . . 6
1.1.5 Contributions logicielles . . . 7
1.2 Structure du manuscrit . . . 7

De nos jours l'interopérabilité de différents systèmes est souvent obtenue via l'échange de documents XML (eXtensible Markup Language). La situation est plus complexe lorsque le besoin de comparer ou ajuster des schémas XML (décrivant la structure des documents) se fait sentir. Diverses applications éprouvent ce besoin comme la composition des services web (par exemple, lors du remplacement d'un service non disponible) ou l'intégration de données. Dans ces situations, il est intéressant d'avoir la possibilité de comparer et faire évoluer non seulement les schémas mais aussi les informations sémantiques liées à ces schémas (à savoir, les contraintes d'intégrité ainsi que les concepts utilisés dans chaque application), et les documents XML.

Considérons l'application suivante : soient les systèmes locaux $S_1, \ldots, S_n$ de la Figure 1.1, qui travaillent respectivement sur les collections de documents $X_1, \ldots, X_n$. Un système local S_i est représenté par un schéma XML D_i qui renseigne sur les contraintes de structure, un ensemble de dépendances fonctionnelles XML $\mathcal{F}_i$ qui renseigne sur les contraintes d'intégrité des données. Chaque ensemble de documents XML X_i respecte les contraintes imposées par D_i et $\mathcal{F}_i$. Les systèmes $S_1, \ldots, S_n$ interagissent avec un système global S. Le système S est une intégration des systèmes locaux, et est vu comme une évolution conservatrice de chaque système local. Cette évolution est conservatrice parce que : (i) le schéma XML D du système global S accepte tous les documents de X_i, (ii) l'ensemble de dépendances fonctionnelles $\mathcal{F}$ du système global S est respecté par tous les documents de X_i.

Le système global S communique avec les systèmes locaux via des échanges de documents. Les communications sont possibles dans les deux sens : global-à-local et local-à-global. L'objectif est de laisser la possibilité aux systèmes locaux de travailler par eux mêmes tout en permettant des diagnostics et des modifications sur le système global. Dans ce contexte, S peut aussi recevoir de nouveaux documents qui ne proviennent pas des systèmes locaux éventuellement non valides par rapport à chacun des systèmes locaux. Dans ce cadre, nous sommes intéressés par la création d'outils pour assister les évolutions de

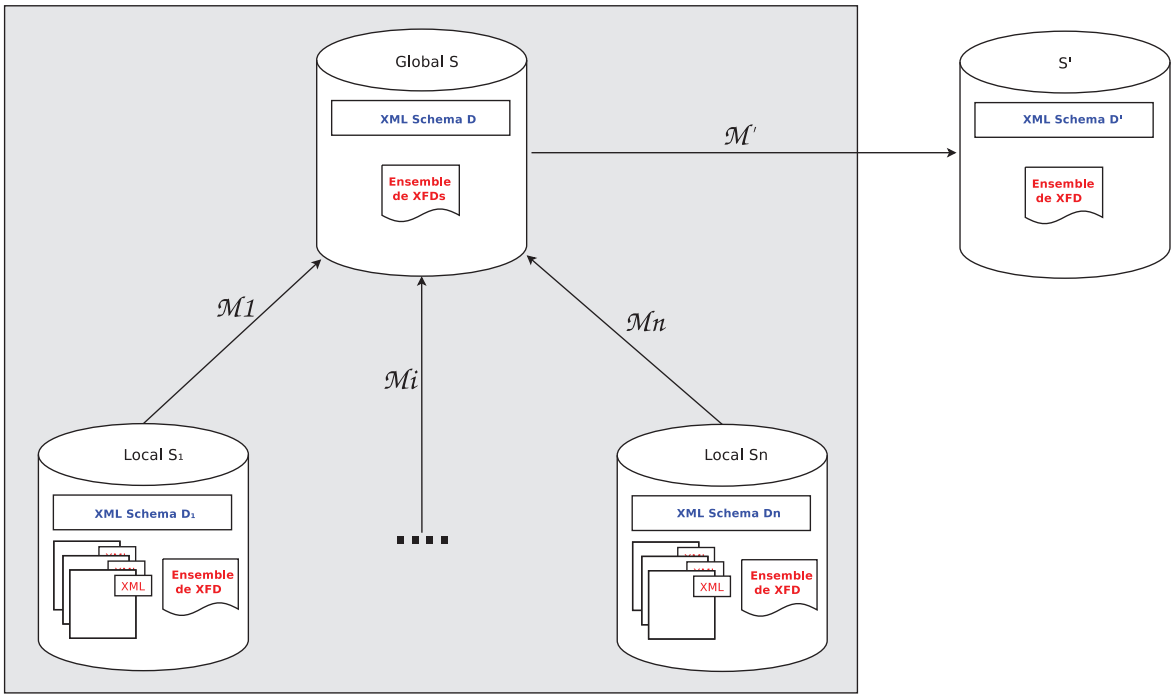


FIGURE 1.1 – *Exemple d'application où des systèmes locaux $S_1, \ldots, S_n$ évoluent en un seul système global S.*

schémas et l'adaptation de documents. Les aspects suivants nous semblent particulièrement importants :

- Pouvoir comparer des schémas ainsi qu'exprimer leur évolution. D'un côté, nous souhaitons exprimer l'évolution de chaque système local par un mapping $\mathcal{M}_i$ qui spécifie l'évolution du schéma D_i, appelé le schéma source, vers le schéma D, appelé le schéma cible. D'autre coté, nous voulons avoir la possibilité de comparer D avec un autre schéma D' de manière souple, simplement en vérifiant si D' accepte des documents XML qui contiennent toutes les informations qui se retrouvent dans les documents XML acceptés par D. Cette comparaison (nommée relâchée) permettrait de vérifier une certaine cohérence de contenu sans prendre en compte toute la structure des documents.
- A partir du mapping $\mathcal{M}_i$, il serait important de traduire les documents valides par rapport à un système local S_i vers le système global S, assurant ainsi le transfert de données et une adaptation automatique. Le mapping entre deux schémas est fondamentale dans le cadre de l'évolution et de l'adaptation automatique des documents. Par exemple, dans une évolution au niveau du système global où S (avec schéma D) évolue vers S' (avec schéma D'), la composition des mappings $\mathcal{M}_i$ et $\mathcal{M}'$ (où $\mathcal{M}'$ est le mapping entre D et D', cf. Figure 1.1) guide l'adaptation des documents. Cette évolution peut aussi se faire au niveau des systèmes locaux quand S_k évolue vers S'_k et, ici encore, les mappings exprimant les modifications des schémas serviront de base pour la traduction des documents. Des outils permettant de telles adaptations automatiques sont essentiels pour assurer la communication global-à-local et local-à-global, garantissant un fonctionnement concomitant qui peut être souhaitable, au moins temporairement.
- Si nous nous intéressons à l'adaptation d'un document X (valide par rapport à un schéma D_i) à un autre schéma D, il est naturel de penser de manière plus générale et proposer une méthode pour corriger X pour qu'il respecte les contraintes de structure

imposées par D. L'idée étant de réparer X, en proposant des documents X' qui lui sont proches tout en étant valides par rapport à D.

– Faire évoluer le schéma implique en faire évoluer aussi des contraintes d'intégrité. Les dépendances fonctionnelles des systèmes locaux peuvent être utilisées pour obtenir un ensemble global $\mathcal{F}$ de dépendances fonctionnelles pour le système global.

L'objectif de ma thèse est de proposer des solutions à des problèmes liés à l'intégration et l'évolution des applications XML. Pour cela nous nous sommes intéressés à tous les aspects énumérés ci-dessus. Dans la suite, nous résumons nos contributions dans chacun de ces cas.

1.1 Contributions

La définition de mapping pour exprimer l'évolution des schémas est le fil conducteur reliant les contributions de ma thèse. Revenons à l'application que nous avons décrite dans la Section 1, illustrée dans la Figure 1.1.

– Nous souhaitons un schéma global capable de reconnaître comme valides des documents originaires de n'importe quel système local. L'algorithme introduit dans [42] propose une extension de schéma de ce genre. Dans [9], nous avons complété cette proposition en implémentant une méthode pour calculer automatiquement un mapping entre une grammaire RTG (Regular Tree Grammar) G et son extension minimale G' qui est une grammaire LTG (Local Tree Grammar). La grammaire G' résultante reconnaît le plus petit langage LTL (Local Tree Language) qui contient $L(G)$. Cette méthode est utilisée pour calculer les mappings $\mathcal{M}_1, \ldots, \mathcal{M}_n$ entre le schéma D du système global et les schémas $D_1, \ldots, D_n$ des systèmes locaux. Le schéma D est calculé à partir de l'algorithme dans [42], en faisant une extension minimale de l'union des schémas $D_1, \ldots, D_n$. La Section 1.1.4 donne les premières idées de cette contribution, décrite en détails dans le Chapitre 6.

– Dans le processus évolutif, nous souhaitons aussi pouvoir comparer des schémas. La comparaison que nous avons envisagée est une *inclusion relâchée* entre deux grammaires d'arbre. Dans [10], nous avons proposé une méthode pour effectuer cette comparaison. Remarquer qu'une évolution peut être choisie sur les bases de cette inclusion relâchée et qu'ensuite, en suivant les mêmes lignes de [9], il est possible de calculer le mapping entre l'ancien et le nouveau schéma. Dans la Section 1.1.1 nous présentons un résumé de cette contribution qui sera traitée dans le Chapitre 3.

– L'évolution des systèmes implique aussi une évolution des contraintes d'intégrité. Les dépendances fonctionnelles jouent un rôle important dans la qualité des données stockées. Il est donc important de pouvoir maintenir le plus grand nombre de contraintes (non contradictoires) lors de l'intégration de plusieurs systèmes locaux vers un système global. Dans cette thèse nous proposons un algorithme permettant d'obtenir le plus grand ensemble de dépendances fonctionnelles XML (XFD) satisfaites par tous les schémas locaux. La Section 1.1.2 montre les grandes lignes de cette contribution, décrite en détails dans le Chapitre 4.

– La méthode pour adapter les documents XML à partir d'un mapping $\mathcal{M}$ entre deux schémas, est basée sur le correcteur de document XML par rapport à un schéma. Nous avons développé ce correcteur de manière général, comme une extension des travaux dans [46], pour ensuite l'utiliser comme base pour les adaptations de documents, rendant plus simple le transfert de données La Section 1.1.3 résume cette contribution, détaillée dans le Chapitre 5.

Le reste de cette section est consacré à un aperçu de toutes ces contributions.

1.1.1 Comparaison des schémas de données

Nous proposons de comparer deux schémas XML via une inclusion relâchée. Cette inclusion consiste à relâcher la relation père-fils entre les nœuds d'un document XML, offrant ainsi la possibilité de regarder à l'intérieur des arbres XML pour vérifier la présence et la position des informations demandées par une application. De cette manière, les informations peuvent être extraites et, éventuellement, adaptées à un autre schéma.

La comparaison considérée est une inclusion relâchée entre deux schémas XML dont l'un est considéré comme schéma source et l'autre comme schéma cible. Cette inclusion relâchée offre la possibilité de regarder à l'intérieur des arbres XML retournés par le schéma cible, pour vérifier si on a bien toutes les informations demandées par le schéma source. Ainsi nous repérons à quel endroit les informations se trouvent dans l'arbre du schéma cible, afin de les extraire et de reconstruire un arbre conforme au schéma source.

Exemple 1.1 Dans la Figure 1.2, nous avons une composition entre les services web A, C, D. Lorsque l'un des services n'est plus disponible, il est important de pouvoir le remplacer par un autre service qui peut fournir les mêmes données que le service en panne. Nous

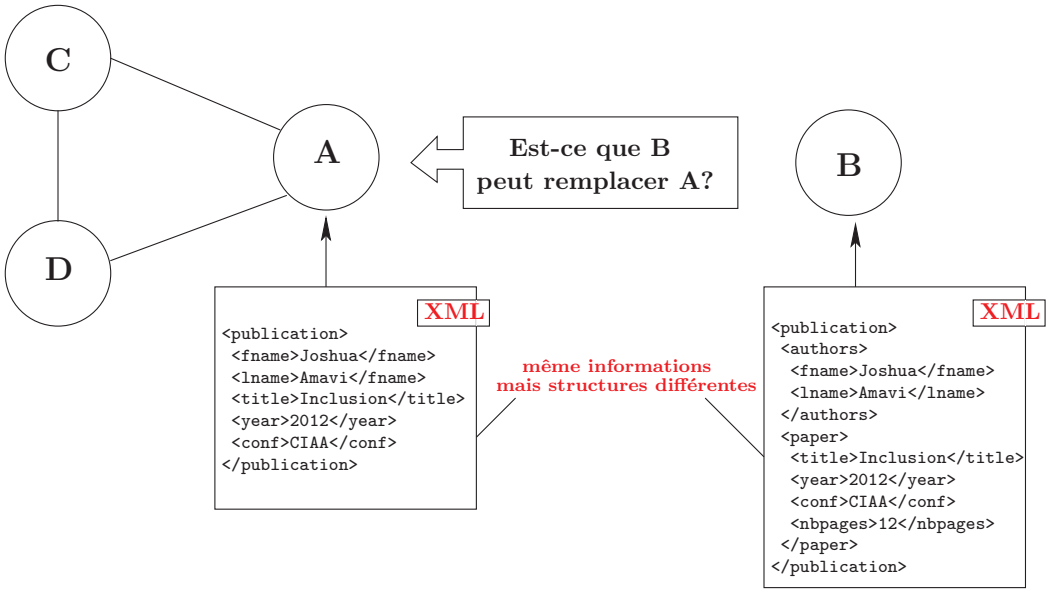


FIGURE 1.2 – *Composition entre les services web A, C et D.*

pouvons remarquer que le service B de la Figure 1.2 fournit bien des documents ayant toutes les informations contenues dans les documents fournis par le service A. Le service B est un bon candidat pour remplacer A dans la composition de services car il va pouvoir fournir toutes les informations de l'ancien service A tout en gardant sa structure.

⊠

Nos contributions permettent de comparer de façon relâchée des langages réguliers d'arbres représentés sous forme de grammaire RTG (Regular Tree Grammar). Supposons deux grammaires G_1 et G_2. Pour savoir si $L(G_1)$ [1] est inclus relâché dans $L(G_2)$, notre solution consiste à calculer la grammaire $WI(G_2)$ des arbres relâchés de la grammaire G_2. Ensuite nous vérifions que le langage généré par G_1 (la grammaire du schéma à remplacer) est inclus (au sens inclusion d'ensembles) dans le langage généré par la grammaire $WI(G_2)$. Dans une première contribution, [8], nous supposons que les grammaires RTG ne sont pas récursives. Ensuite, une nouvelle contribution, [10] a été apportée pour résoudre le cas général.

1. $L(G_1)$ est le langage généré par la grammaire G_1.

1.1.2 Intégration de dépendances fonctionnelles

Suite à une évolution conservatrice des systèmes locaux S_i en un système global S (cf. Figure 1.1), nous souhaitons calculer l'ensemble global de dépendances fonctionnelles $\mathcal{F}$ qui est respecté par tous les documents de chaque systèmes local S_i. Ainsi $\mathcal{F}$ est le plus grand ensemble de dépendances fonctionnelles (issues des différents $\mathcal{F}_i$) respecté par tous les documents de chaque ensemble X_i. Pour calculer $\mathcal{F}$ (Figure 1.3), nous allons analyser chaque dépendance fonctionnelle f_i^k dans chaque ensemble $\mathcal{F}_i$ pour décider si nous devons la garder dans $\mathcal{F}$ ou pas. La dépendance f_i^k n'est pas gardée dans $\mathcal{F}$ seulement si un document dans X_j ($i \neq j$) peut violer f_i^k. Nos contributions pour cette partie, publiées dans [11, 12], sont :

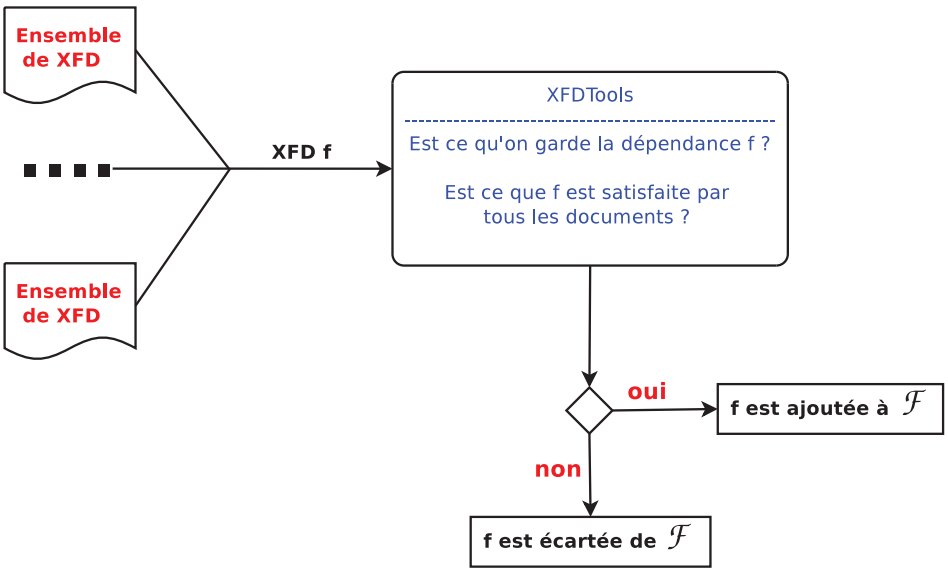


FIGURE 1.3 – *Intégration de dépendances fonctionnelles.*

1. une première contribution est l'élaboration d'un système d'axiomes correct et complet pour pouvoir raisonner sur les dépendances fonctionnelles XML (dénotée XFD) définies dans [82, 66, 33]. A l'aide de ce système d'axiomes, nous pouvons déduire des XFD à partir d'un ensemble de dépendances fonctionnelles $\mathcal{F}$, et aussi calculer la fermeture de $\mathcal{F}$;
2. notre deuxième contribution est une méthode pour calculer une couverture de l'ensemble $\mathcal{F}$. Nous calculons une couverture de $\mathcal{F}$, notée $cover\mathcal{F}$ et qui est un ensemble équivalent à $\mathcal{F}$, car l'ensemble $\mathcal{F}$ contient un nombre très élevé de dépendances, ce qui fait qu'il est impossible de pouvoir travailler avec $\mathcal{F}$ directement. Cet ensemble $cover\mathcal{F}$ sera utilisé par le système global pour, par exemple, valider de nouveaux documents qu'il va recevoir. Une originalité de notre travail, est la technique pour supprimer une dépendance fonctionnelle f (non souhaitée) d'un ensemble de dépendances fonctionnelles G sans pour autant affaiblir les raisonnements et déductions que l'on peut faire à partir de G. La suppression de f dans G est faite de telle sorte que G continue à dériver toutes les dépendances qu'il dérivait auparavant, à l'exception de la dépendance supprimée f.

Un résumé de ses travaux a été publié dans [11], puis une version longue dans [12].

1.1.3 La correction d'un document par rapport à un schéma XML

Étant donné un arbre XML t qui n'est pas valide par rapport à un schéma S, nous proposons un correcteur pour trouver tous les arbres t' valides par rapport à S tel que la distance d'édition entre t et t' est inférieure à un seuil donné. Le correcteur se base sur trois opérations élémentaires à savoir l'ajout et la suppression d'une feuille et le renommage d'un nœud, pour pouvoir modifier un arbre XML. La principale contribution de notre approche est d'offrir une première étude approfondie du problème de correction d'arbre vers un langage d'arbres. Non seulement nous mesurons la distance entre le document et le schéma mais aussi nous trouvons les arbres candidats à la correction. Nous ne nous limitons pas seulement à la recherche des solutions minimales. Nous trouvons toutes les solutions à une distance inférieure à un seuil donné. Ce travail est une continuation et une extension de ceux de [46, 29, 28]. Dans [46, 29, 28], les auteurs proposent de corriger des sous-arbres détectés invalides lors de la validation d'un arbre XML par rapport à un schéma. A partir de l'algorithme proposé dans nos travaux, nous avons développé le correcteur nommé *XMLCorrector* (Figure 1.4) pour proposer des corrections à un document XML non valide. L'outil *XMLCorrector* permet aussi de calculer les solutions de coût minimal. Nos travaux sur la correction de document ont été publiés dans le journal [6] et une démonstration de l'outil *XMLCorrector* a été effectuée lors de la conférence internationale CIAA 2012 à Porto.

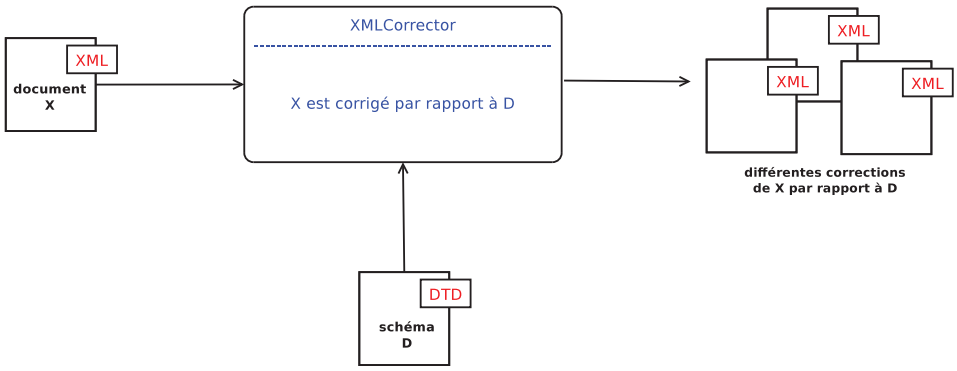


FIGURE 1.4 – *Correction de document XML.*

Dans l'application décrite dans la Section 1 et illustrée dans la Figure 1.1, nous utilisons *XMLCorrector* pour :

1. réparer un document XML X par rapport au schéma D ;
2. adapter les documents du schéma D en des documents valides par rapport à D', lorsque le schéma D évolue vers le schéma D' ;
3. choisir le système local S_i qui va stocker un nouveau document XML que le système global S reçoit. L'idée est de comparer la distance entre le nouveau document XML et chacun des schémas XML locaux D_i et ensuite d'orienter le nouveau document vers le système dont le schéma a obtenu la plus petite distance.

1.1.4 L'évolution des schémas et l'adaptation des documents XML

Lorsqu'un schéma D évolue en un autre schéma D' (Figure 1.5), nous souhaitons faire aussi évoluer ou adapter les documents liés au schéma D. Pour exprimer l'évolution de D vers D', nous proposons un langage de mapping basé sur des opérations d'édition de schémas XML (représentés par des RTG). Ensuite, en se servant du mapping ainsi exprimé, nous proposons une méthode pour adapter les documents XML du schéma D vers le schéma D'. Pour faire face à d'éventuelles évolutions de D ou de D', nous définissions la

composition et l'inversion d'un mapping. Ces opérateurs sont fondamentales pour adapter un mapping $\mathcal{M}$ aux évolutions des schémas originaux (D et D') qui ont servis à sa définition.

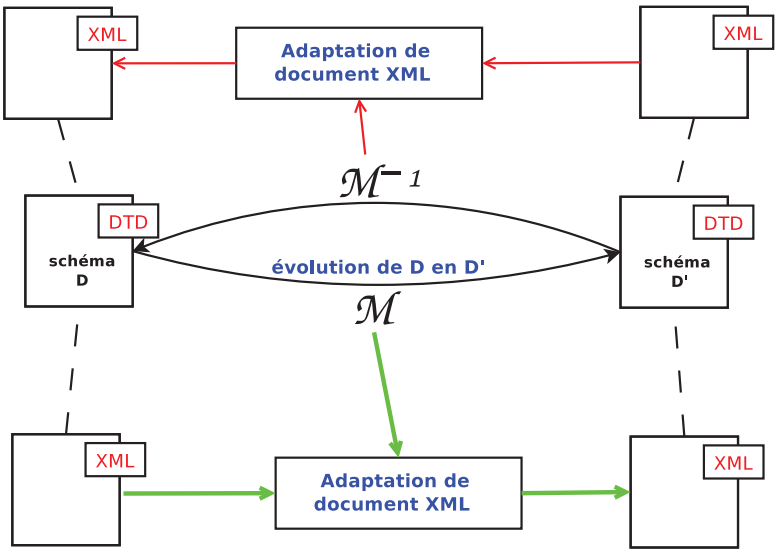


FIGURE 1.5 – *Échange de données entre deux schémas.*

1.1.5 Contributions logicielles

A partir des prototypes développés pour nos expérimentations, nous avons conçu l'application *XMLCorrector*[2] et nous sommes en train de développer l'application *DTDGrabber* qui met en commun nos diverses contributions en un seul outil (qui est bien avancé) pour faciliter la gestion d'une base de données XML. Par gestion nous entendons : (i) la définition d'une base de donnée XML à partir d'un schéma XML (DTD ou grammaire RTG), (ii) l'ajout de dépendances fonctionnelles et la vérification de ces contraintes sur les documents XML de la base de données, (iii) l'ajout de nouveaux documents XML à la base et la correction éventuelle des nouveaux documents qui sont pas conformes au schéma de la base de données, (iv) la création de mapping pour permettre l'évolution d'un schéma source vers un nouveau schéma cible et l'adaptation des documents de la source vers la cible. Pour être un vrai SGBD, il manque à notre application toute la partie interrogation de données et mise à jour de données, et une API (interface de programmation) pour accéder facilement à ces fonctionnalités via d'autres programmes. Nos applications sont développées en *Java*.

1.2 Structure du manuscrit

Le manuscrit est ordonné de la manière suivante. Le Chapitre 2 regroupe les définitions d'arbres XML et de schémas XML qui sont des notions communes aux autres chapitres.

Les chapitres suivants détaillent les contributions de ma thèse. *Les différents aspects du processus évolutif traités dans ce manuscrit sont présentés de manière indépendantes.* C'est seulement dans le chapitre 6 que le lien entre plusieurs de ces aspects sera fait.

Ainsi, le Chapitre 3 présente les résultats obtenus sur la comparaison de schémas XML. Le problème d'intégration de dépendances fonctionnelles est abordé dans le Chapitre 4. Le

2. *XMLCorrector* est disponible en ligne : `http://www.info.univ-tours.fr/~savary/English/xmlcorrector.html`

Chapitre 5 sera consacré aux détails sur la correction d'un document XML par rapport à un schéma XML. Le Chapitre 6 définit notre langage de mapping de schémas et propose une méthode pour l'adaptation des documents XML et des algorithmes pour calculer (i) les mappings entre des schémas locaux et le schéma global, et (ii) le mappings entre un schéma G' et le schéma de ses sous-arbres relâchés $WI(G')$. Nos applications *XMLCorrector* et *DTDGrabber* sont décrites dans le Chapitre 7. La majorité des preuves de lemmes, propriétés et théorèmes sont en Annexe A. Certaines preuves sont en anglais.

Arbres et schémas XML

2

Sommaire

2.1 Documents XML 9
2.2 Schémas XML 12

2.1 Documents XML

Nous manipulons les documents XML sous forme d'arbres d'arité non bornée. Un arbre t est une fonction d'un ensemble fini de positions (le **domaine**) vers un ensemble d'étiquettes (ou labels) Σ. L'ensemble Σ est aussi appelé **alphabet**. Une position est un identifiant unique d'un nœud de t et est représentée par une séquence d'entiers positifs. Par exemple, la Figure 2.1 montre un document XML et sa représentation sous forme d'arbre. Les noms des attributs sont précédés du symbole @ et les valeurs sont attachées aux feuilles et sont en italique. Dans la Figure 2.1, le premier élément de label *author* est en position 0.5.0. On constate dans cette figure que chaque position est toujours préfixée par celle de son père. Dans la suite nous confondrons les notions de nœud et de position.

```
<pub>
  <paper id="1">
    <title>XML</title>
    <conf>SAC</conf>
    <year>2013</year>
    <pages>6</pages>
    <authors>
      <author id="a1">
       <nameA>Amavi</nameA>
       <affiliation>LIFO</affiliation>
      </author>
      <author id="a2">
       <nameA>Halfeld</nameA>
       <affiliation>LIFO</affiliation>
      </author>
    </authors>
  </paper>
</pub>
```

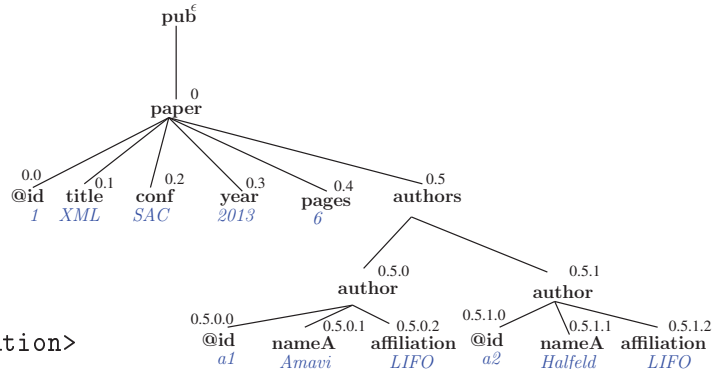


Figure 2.1 – *Exemple de document XML et son arbre*

Nous allons travailler avec un alphabet Σ qui est un ensemble fini d'étiquettes (ou labels) d'alphabet Σ et un ensemble de positions $U \subseteq \mathbb{N}^*$ qui est l'ensemble de tous les mots sur les entiers. La concaténation est dénotée par le point.

Définition 2.1 (Positions)
Soit $\mathbb{N}$ l'ensemble des entiers positifs, et $\mathbb{N}^*$ l'ensemble des mots où un mot est une séquence d'entiers positifs. La concaténation est dénotée par le point et la séquence vide est notée ϵ. Une **position** p est un élément de $\mathbb{N}^*$. □

La façon dont nous numérotons les positions est aussi appelée "Dewey numbering" [75] qui est une numérotation basée sur les préfixes.

Définition 2.2 (Comparaison de positions)
Soient u et v des positions. La position u est un **préfixe** (ou **ancêtre**) de v (dénotée $u \leq v$) si et seulement si il existe une séquence (éventuellement vide) d'entiers w telle que $u.w = v$. Lorsque w n'est pas vide alors nous disons que u est un **préfixe propre** de v (dénotée $u < v$). La position u est **à gauche** de v (dénotée $u \prec v$) s'il existe des séquences p, q, r et des entiers i,$j \in \mathbb{N}$ tels que $u = p.i.q$, $v = p.j.r$ et $i < j$.
La position u est **parallèle** à v (dénotée $u \parallel v$) si $\neg(u < v) \wedge \neg(v < u)$.

□

L'ensemble des positions D ($D \subseteq \mathbb{N}^*$) est **clos par préfixe** si $v \in D$ et $u < v$ implique $u \in D$.

Définition 2.3 (Domaine d'un arbre)
Le *domaine* de l'arbre t, noté $Pos(t)$, est l'ensemble des positions de t *clos par préfixe* et qui satisfait la condition $(\forall j \geq 0,\ u.j \in Pos(t)) \Rightarrow (\forall i\ \ 0 \leq i < j,\ u.i \in Pos(t))$. □

Intuitivement, la dernière condition indique que si la position 1.2 existe, alors les positions 1.0 et 1.1 (les frères gauches) existent aussi.

Définition 2.4 (Représentation d'un document XML)
Soit un alphabet $\Sigma = \Sigma_{ele} \cup \Sigma_{att} \cup \Sigma_{data}$, dans lequel Σ_{ele} est l'ensemble des noms d'éléments, Σ_{att} est l'ensemble des noms d'attributs et Σ_{data} est l'ensemble des noms d'éléments ayant des données. Un document XML est représenté par un triplet $\mathcal{T} = (t, type, value)$ tel que :

- l'arbre t est la fonction $t : Pos(t) \rightarrow \Sigma \cup \{\lambda\}$. Pour chaque position $p \in Pos(t)$, $t(p) = a$ indique que le symbole $a \in \Sigma$ est associé au nœud de position p. La racine est associée à la position ϵ et l'arbre vide est l'arbre $\{(\epsilon, \lambda)\}$ où $\lambda \notin \Sigma$ est un symbole réservé pour l'arbre vide. L'arbre t est appelé **arbre XML**.
- la fonction $type : t \times Pos(t) \rightarrow \{data, élément, attribut\}$ est défini par : $type(t,p) = \begin{cases} data & \text{si } t(p) \in \Sigma_{data} \\ élément & \text{si } t(p) \in \Sigma_{ele} \\ attribut & \text{si } t(p) \in \Sigma_{att} \end{cases}$
- la fonction $value : t \times Pos(t) \rightarrow Pos(t) \cup \mathbf{V}$ est défini par :
  $value(t,p) = \begin{cases} p & \text{si } type(t,p) = élément \\ val \in \mathbf{V} & \text{sinon} \end{cases}$
  où $\mathbf{V}$ est un ensemble récursivement énumérable.

□

L'ensemble des positions feuilles dans un arbre XML t, noté *leaves*, est défini comme

suit :

$$leaves(t) = \{u \in Pos(t) \mid \nexists\, i \in \mathbb{N} \text{ tel que } u.i \in Pos(t)\}$$

Nous notons par $|t|$ la taille de l'arbre t c'est-à-dire le nombre de positions dans (ou la cardinalité de) $Pos(t)$, et par $\bar{t}$ le nombre de fils directs du nœud racine de t. L'arbre vide (pour lequel $Pos(t) = \emptyset$) est dénoté par $t_\emptyset$.

Exemple 2.1 Par exemple considérons le document XML $\mathcal{T} = (t, type, value)$ de la Figure 2.2, nous avons $t(\epsilon) = library$, $t(0) = book$, $type(t, 1.0) = data$ et $value(t, 1.0) = BackPack$. Aussi $|t| = 26$ et $\bar{t} = 2$.

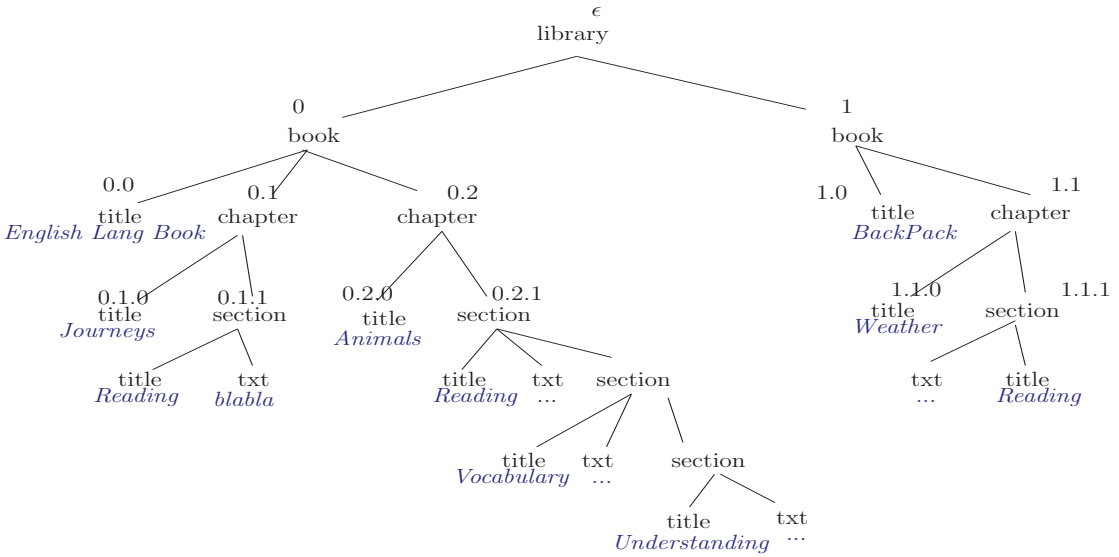


FIGURE 2.2 – *Document XML : une librairie de livres en anglais. Chaque nœud a une position et un label. Par exemple, $t(\epsilon) = library$ et $t(0.1.1) = section$.*

⊠

Nous allons introduire les notions de *sous-arbre* et *arbre partiel*.

Définition 2.5 (Sous-arbre et arbre partiel)
Étant donné un arbre XML non vide t, une position $p \in \mathbb{N}^*$ and $i \in \mathbb{N}$ tel que $-1 \leq i \leq \bar{t} - 1$, les notions de sous-arbre et d'arbre partiel sont définies comme suit :

- $t|_p$ dénote le **sous-arbre** dont la racine est en position $p \in Pos(t)$, plus précisément $Pos(t|_p) = \{s \mid p.s \in Pos(t)\}$ et pour chaque position $s \in Pos(t|_p)$ on a $t|_p(s) = t(p.s)$.
- $t\langle i \rangle$ dénote **l'arbre partiel** de t qui contient la racine de t et les sous-arbres $t|_0$, $t|_1$, ..., $t|_i$. Formellement nous avons $Pos(t\langle i \rangle) = \{v \mid v \in Pos(t)$, et $[v = \epsilon$ ou $v = k.u$ avec $0 \leq k \leq i$ et $u \in \mathbb{N}^*]\}$ et pour chaque position $v \in Pos(t\langle i \rangle)$ on a $t\langle i \rangle(v) = t(v)$.

□

Tout sous-arbre ou arbre partiel d'un arbre XML est lui-même un arbre XML au sens de la Définition 2.4. Notons que d'après la Définition 2.5, pour tout arbre non vide t on a $t|_\epsilon = t$, $t\langle \bar{t} - 1 \rangle = t$, et $t\langle -1 \rangle = \{(\epsilon, t(\epsilon))\}$.

Les documents XML que nous représentons sous forme d'arbres, doivent vérifier des contraintes de schémas qui sont exprimées par des DTD ou XML Schema du W3C ou Relax-NG. Nous allons définir dans la section suivante les différents types de schémas pour XML.

2.2 Schémas XML

Dans ce manuscrit, les schémas XML seront exprimés par des grammaires d'arbres. Nous verrons ensuite le lien entre les grammaires d'arbres que nous allons définir avec les schémas XML, c'est-à-dire DTD ou XML Schema du W3C ou Relax-NG. Les définitions suivantes ont été formalisées dans [89, 42].

Définition 2.6 (Regular Tree Grammar (RTG))
Une grammaire **regular tree grammar** (RTG) (appelée aussi **grammaire de haies**) est un 4-uplet $G = (N, \Sigma, S, P)$, où N est un ensemble fini de *symboles non-terminal* ; Σ est un ensemble fini de *symboles terminal* ; S est un ensemble de *symboles initiaux* tel que $S \subseteq N$ et P est un ensemble fini de *règles de production* qui sont de la forme $X \to a\,[R]$, avec $X \in N$, $a \in \Sigma$, et R est une expression régulière sur l'ensemble N. Nous rappelons que les expressions régulières sur $N = \{A_1, \ldots, A_n\}$ sont définies par induction par :

$$R ::= \epsilon \mid A_i \mid R|R \mid R.R \mid R^+ \mid R^* \mid R^? \mid (R).$$

□

Dans la suite, nous allons considérer uniquement des RTGs en forme réduite et normale. Les définitions de ces différentes formes sont les suivantes :

Définition 2.7 (Grammaire en forme normale et réduite [86])
Une grammaire RTG est **réduite** si (i) chaque non-terminal est accessible à partir d'un non-terminal initial, et (ii) tous les non-terminaux génèrent au moins un arbre ne contenant que des symboles terminaux. Une grammaire RTG est en forme **normale** si différentes règles de production ont différents non-terminaux comme membre gauche. Ainsi lorsqu'on a une RTG $G = (N, \Sigma, S, P)$ réduite et en forme normale, pour chaque non-terminal $A \in N$ il existe dans P exactement une seule règle de production de la forme $A \to a[E]$, c'est-à-dire qui a comme membre gauche A.

□

Étant donné une RTG G qui n'est pas en forme normale, il est toujours possible de la transformer en une RTG en forme normale équivalente. Voici un exemple qui illustre une telle transformation :

Exemple 2.2 La grammaire $G_0 = (N_0, \Sigma, S, P_0)$, telle que $N_0 = \{X, A, B\}$, $\Sigma = \{f, a, c\}$, $S = \{X\}$, et $P_0 = \{X \to f\,[A.B],\ A \to a[\epsilon],\ B \to a[\epsilon],\ A \to c[\epsilon]\}$, n'est pas en forme normale. La conversion de G_0 en forme normale donne les ensembles $N_1 = \{X, A, B, C\}$ et $P_1 = \{X \to f\,[(A|C).B],\ A \to a[\epsilon],\ B \to a[\epsilon],\ C \to c[\epsilon]\}$.

⊠

Comme dans une grammaire en forme normale, il existe une seule règle de production $A \to a[E]$ pour un non-terminal A, nous notons par $term(A) = a$ et $reg(A) = E$ le terminal et l'expression régulière respectivement, associés à A.

A partir d'une grammaire RTG G, la définition suivante nous montre comment dériver un arbre t qui appartient au langage $L(G)$ généré par G.

Définition 2.8 (Dérivation)
Soit $G = (N, \Sigma, S, P)$ une RTG (en forme normale). Considérons un non-terminal $A \in N$, et soit $A \to a[E]$ la règle de production unique de P qui a comme partie gauche A. Le langage $L^w(E)$ dénote l'ensemble des mots sur les non-terminaux générés par E, et T_Σ dénote

l'ensemble de tous les arbres qui ne contiennent que des symboles terminaux. L'ensemble $L_G(A)$ d'arbres générés par A est défini récursivement par :

$$L_G(A) = \{a(t_1, \ldots, t_n) \mid \exists u \in L^w(E),\, u = A_1 \ldots A_n,\, \forall i, t_i \in L_G(A_i)\}.$$

Le langage $L(G)$ généré par G est $L(G) = \{t \in T_\Sigma \mid \exists A \in S,\, t \in L_G(A)\}$.

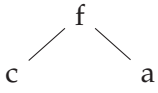
Un arbre t défini sur $N \cup \Sigma$ **se dérive** (en une étape) en un arbre t' si et seulement si (i) il existe une position p dans t telle que $t|_p = A \in N$ et une règle de production $A \to a\,[E]$ appartenant à P, et (ii) $t' = t[p \leftarrow a(w)]$ [1] où $w \in L^w(E)$. Nous notons cette dérivation par $t \to_{[p, A \to a\,[E]]} t'$. Une **dérivation** (en plusieurs étapes) est une séquence (éventuellement vide) de dérivations en une étape. Nous la notons $t \to_G^* t'$. Le langage $L(G)$ défini ci-dessus peut être aussi défini de la façon suivante : $L(G) = \{t \in T_\Sigma \mid \exists A \in S,\, A \to_G^* t\}$.

Un *nt-tree* est un arbre dont toutes les étiquettes sont des non-terminaux. L'ensemble $L_G^{nt}(A)$ de *nt-trees* généré par A est défini récursivement par :

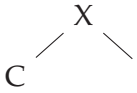
$$L_G^{nt}(A) = \{A(t_1, \ldots, t_n) \mid \exists u \in L^w(E),\, u = A_1 \ldots A_n,\, \forall i\, (t_i \in L_G^{nt}(A_i) \vee t_i = A_i)\}.$$

□

Exemple 2.3 Considérons la grammaire $G_1 = (N_1, \Sigma, S, P_1)$, telle que $N_1 = \{X, A, B, C\}$, $\Sigma = \{f, a, c\}$, $S = \{X\}$, et $P_1 = \{X \to f\,[(A|C).B],\, A \to a[\epsilon],\, B \to a[\epsilon],\, C \to c[\epsilon]\}$. Le *nt-tree* pour l'arbre



est :



.

⊠

La classe des langages d'arbres réguliers RTL possède des sous-classes que nous allons maintenant aborder. Pour différencier les sous-classes de RTL, nous devons comprendre la notion de non-terminaux en concurrence.

Définition 2.9 (Non-terminaux en concurrence)
Soit une RTG $G = (N, \Sigma, S, P)$. Deux non-terminaux A et B sont **en concurrence** si $A \neq B$ et G contient des règles de productions de la forme $A \to a[E]$ et $B \to a[E']$, c'est-à-dire que A et B partagent le même symbole terminal. □

Nous allons maintenant considérer les différentes sous-classes de RTG proposées dans [89].

Définition 2.10 (Local Tree Grammar (LTG))
Une grammaire **local tree grammar** (LTG) est une RTG qui ne possède pas de non-terminaux en concurrence. Un **local tree language** (LTL) est un langage qui peut être généré par au moins une LTG. □

Définition 2.11 (Single-Type Tree Grammar (STTG))
Une grammaire **single-type tree grammar** (STTG) est une RTG en forme normale telle que (i) pour chaque règle de production, les non-terminaux contenus dans son expression régulière ne sont pas en concurrence et (ii) les non-terminaux initiaux ne sont pas en concurrence. Un **single-type tree language** (STTL) est un langage qui peut être généré par au moins une STTG. □

1. La notation $t[p \leftarrow a(w)]$ signifie qu'on remplace le sous-arbre à la position p dans t par le sous-arbre $a(w)$.

Exemple 2.4 Soient G_1, G_2 et G_3 les grammaires suivantes :

G_1 :	G_2 :	G_3
B $\rightarrow$ book$[Tb.S^*]$	B $\rightarrow$ book$[Tb.S^*]$	B $\rightarrow$ book$[Tb.S^*]$
Tb $\rightarrow$ titleBook$[St^?]$	Tb $\rightarrow$ title$[St^?]$	Tb $\rightarrow$ title$[St^?]$
St $\rightarrow$ subtitle$[\epsilon]$	St $\rightarrow$ subtitle$[\epsilon]$	St $\rightarrow$ subtitle$[\epsilon]$
S $\rightarrow$ section$[Ts.Para^+]$	S $\rightarrow$ section$[Ts.Para^+]$	S $\rightarrow$ section$[Ts.Para_1.Para_2]$
Ts $\rightarrow$ titleSection$[\epsilon]$	Ts $\rightarrow$ title$[\epsilon]$	Ts $\rightarrow$ title$[\epsilon]$
Para $\rightarrow$ paragraph$[Ct^*]$	Para $\rightarrow$ paragraph$[Ct^*]$	$Para_1 \rightarrow$ paragraph$[Ts.Ct]$
		$Para_2 \rightarrow$ paragraph$[Ct^*]$
Ct $\rightarrow$ content$[\epsilon]$	Ct $\rightarrow$ content$[\epsilon]$	Ct $\rightarrow$ content$[\epsilon]$

La grammaire G_2 n'est pas une LTG car les non-terminaux Tb et Ts sont en concurrence. La grammaire G_3 n'est pas une STTG car les non-terminaux $Para_1$ et $Para_2$ sont en concurrence et se retrouvent tous les deux dans l'expression régulière $Ts.Para_1.Para_2$ associée au non-terminal S. Voici un tableau résumé des types pour les grammaires de cet exemple :

	RTG	STTG	LTG
G_1	Oui	Oui	Oui
G_2	Oui	Oui	Non
G_3	Oui	Non	Non

FIGURE 2.3 – *Tableau résumé des types des grammaires*

⊠

Dans [89] le pouvoir d'expression de ces classes de langages est discuté. Nous pouvons donc rappeler que LTL $\subset$ STTL $\subset$ RTL (RTL correspond à **local tree grammar**). Cependant LTL et STTL sont clos par intersection mais pas par union ; tandis que RTL est clos par union, intersection et complément. Nous pouvons aussi remarquer que la convertion d'une LTG en forme normale est aussi une LTG.

Dans [89] il est aussi établi la correspondance entre ces langages théoriques et les schémas XML telles que les DTD ou XML Schema du W3C ou Relax-NG. Dans ce sens, nous avons :

1. LTG a le même pouvoir d'expression qu'une DTD ;
2. STTG a le même pouvoir d'expression qu'un XML Schema du W3C ;
3. RTG a le même pouvoir d'expression que Relax-NG.

Pour terminer ce chapitre, nous allons rappeler quelques définitions et résultats concernant les expressions régulières. Ainsi le standard du W3C établit que seules les expressions régulières 1-unambiguous sont autorisées dans les DTDs. Une expression régulière est 1-unambiguous si chaque symbole dans un mot peut être rattaché à une seule occurrence de symbole dans l'expression régulière lors de l'analyse du mot, sans savoir ce qui suit dans le mot. Par exemple, considérons l'expression régulière $E = (A|B)^*.A.A^*$ et le mot $w = BAA \in L(E)$. Le mot w peut être parsé de deux façons différentes. Si nous numérotons chaque symbole de E par des entiers distincts, nous obtenons $E' = (A_1|B_2)^*.A_3.A_4^*$. En considérant les mots avec indices $B_2A_1A_3$ et $B_2A_3A_4$ nous avons deux façons d'engendrer w. Par conséquent, l'expression régulière E n'est pas 1-unambiguous. Nous référons à [35] pour la définition formelle de cette notion. Il a aussi été montré qu'une expression régulière E est 1-unambiguous si et seulement si son automate de Glushkov correspondant est déterministe [35, 38, 128].

Définition 2.12 (Expression régulière monadique et stricte)
Une expression régulière E est **monadique** si chaque non-terminal de E n'apparaît qu'une seule fois dans E. Une expression régulière E est **stricte** si elle ne contient pas d'opérateurs $+$ (la fermeture positive) et ? (optionnel). Une grammaire est **monadique** (respectivement **stricte**) si toutes les expressions régulières sont monadiques (respectivement strictes).

□

Le lemme qui suit, est une conséquence immédiate des notions précédentes.

Lemme 2.1 *Une expression régulière monadique est 1-unambiguous. Par conséquent, une LTG monadique est déterministe (une LTG ou DTD est déterministe si toutes ses expressions régulières sont 1-unambiguous [35]).*

Inclusion relâchée de schémas XML 3

Sommaire

3.1 Préliminaires 19
3.2 Inclusion relâchée pour les Regular Tree Grammars 20
3.2.1 Cas de grammaire non récursive 20
3.2.2 Cas de grammaire récursive 22
3.3 Implémentation et Résultats Expérimentaux 28
3.4 Travaux liés 31
3.5 Conclusion et perspectives 32

De nos jours, XML est le format d'échange de données le plus utilisé sur le web. Pour permettre l'interopérabilité entre plusieurs systèmes, il est utile d'obtenir des informations à partir d'un autre système. Dans le contexte des données modélisées sous forme d'arbres, cette opération correspond à la recherche de sous-arbres selon une requête de l'application donnée. Cette recherche de sous-arbre peut être approximative, en essayant de trouver le document XML qui est le plus adapté pour un certain nombre de contraintes. La situation est beaucoup plus complexe lorsque l'on souhaite comparer (ou retrouver) des schémas XML qui définissent des documents proches (pas forcément égaux) aux documents générés par un schéma XML donné. Dans ce chapitre nous offrons la possibilité de comparer les schémas XML d'une manière relâchée qui n'a pas encore été étudiée dans la littérature. Il existe des travaux [22, 44, 45, 74, 98] sur la comparaison d'arbres de façon relâchée mais la possibilité de faire une telle comparaison sur des langages d'arbres (ou schémas XML) est nouvelle. Ainsi toutes les applications pour lesquelles une telle comparaison est nécessaire ne sont pas encore connues. Pour les arbres, la comparaison de manière relâchée est par exemple utilisée pour exprimer et retrouver de l'information dans une base donnée structurée. L'une des applications possibles pour la comparaison de manière relâchée de schémas XML, est de pouvoir substituer un service web faisant partie d'une composition de services par un autre service web. Cette application est illustrée dans l'exemple suivant.

Exemple 3.1 Supposons que nous avons une application où nous souhaitons remplacer un schéma XML G par un nouveau schéma G' (comme par exemple une composition de services web où un service remplace un autre, chacun des services est associé à son propre schéma XML). Avant d'effectuer le remplacement, nous voulons vérifier si les documents (ou messages) XML supportés par G' contiennent (d'une manière approximative) ceux supportés par G. Les schémas XML sont vus comme des grammaires (Chapitre 2, Définition 2.6) où nous ne considérons que la structure des documents XML sans prendre en compte les données attachées aux feuilles. Ainsi pour définir les feuilles, nous considérons les règles de

production de la forme $A \to a[\epsilon]$. Nous allons supposer que G et G' contiennent les règles suivantes :

G :	G' :
P $\to$ publi$[(F.L)^+.T.B^?]$	P $\to$ publi$[A^*.Pa]$
B $\to$ biblio$[P^+]$	A $\to$ author$[F.L]$
F $\to$ firstName$[\epsilon]$	Pa $\to$ paper$[T.Y.B^?]$
L $\to$ lastName$[\epsilon]$	B $\to$ biblio$[P^+]$
T $\to$ title$[\epsilon]$	F $\to$ firstName$[\epsilon]$
	L $\to$ lastName$[\epsilon]$
	T $\to$ title$[\epsilon]$
	Y $\to$ year$[\epsilon]$

Le non-terminal P définit une publication et le non-terminal B définit une bibliographie.

Nous souhaitons savoir si les messages XML valides par rapport à G peuvent être acceptés (d'une manière approximative) par G'. Le schéma G accepte les documents XML comme l'arbre t de la Figure 3.1 qui ne sont pas valides par rapport à G', mais qui représentent les mêmes types d'informations qui sont traités par G'. En effet dans G', la même information est organisée comme l'arbre t' de la Figure 3.1.

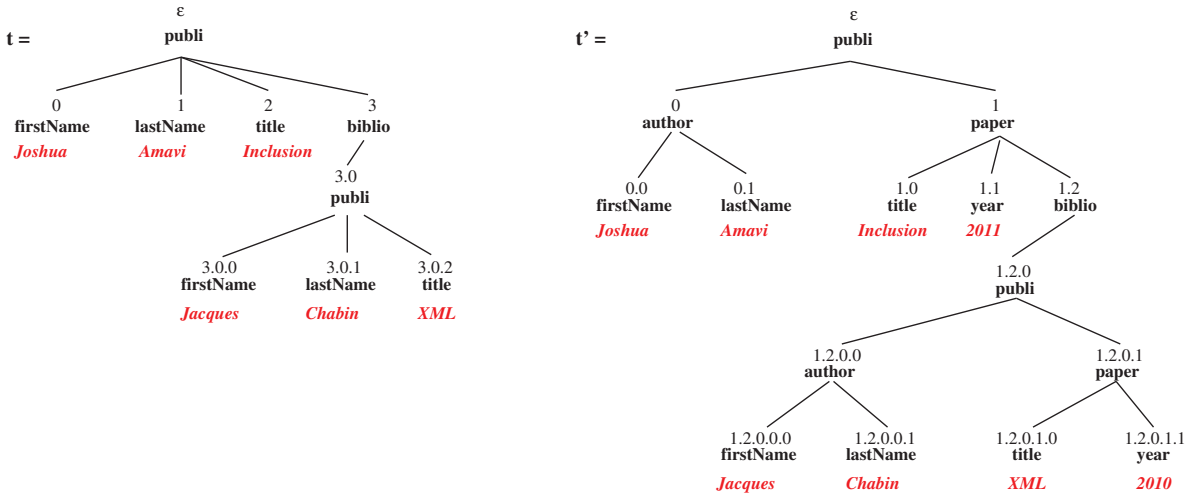


FIGURE 3.1 – *Arbres XML t et t' valides par rapport à G et G', respectivement.*

⊠

Le critère d'approximation qui est souvent utilisé pour comparer les arbres, consiste à relâcher la relation père-fils entre les nœuds d'un arbre. Dans ce chapitre, nous considérons ce critère dans le contexte des langages d'arbres. Nous appelons cette relation *inclusion relâchée* pour éviter des confusions avec l'inclusion classique de langages, c'est-à-dire l'inclusion d'un ensemble d'arbres dans un autre.

Étant donné deux grammaires G et G', $L(G)$ et $L(G')$ sont les ensembles (aussi appelés langages) de documents XML valides par rapport à G and G', respectivement. Nous proposons dans ce chapitre, une méthode pour décider si $L(G)$ est *inclus relâché* dans $L(G')$ dans le but de savoir si la substitution de G par G' peut être envisagée. Le langage d'arbres $L(G)$ est *inclus relâché* dans $L(G')$ si pour tout arbre $t \in L(G)$, il existe un arbre $t' \in L(G')$ tel que t *inclus relâché* dans t'. Intuitivement t est inclus relâché dans t' (dénoté $t \lhd t'$) si nous pouvons obtenir t en supprimant des nœuds dans t' (un nœud supprimé, est remplacé par ses enfants s'il en a). Par exemple dans la Figure 3.1, t peut être obtenu à partir de

t' en supprimant les nœuds *author*, *paper*, *year* de t' ce qui fait que nous avons $t \lhd t'$. La grammaire G' de l'Exemple 3.1 a deux niveaux en plus, à savoir les nœuds *author*, *paper* et une information en plus, le nœud *year*, par rapport à la grammaire G de l'Exemple 3.1. Nous pouvons donc vérifier facilement que $L(G)$ est *inclus relâché* dans $L(G')$. Nous allons voir dans ce chapitre comment faire cette vérification à partir d'un algorithme que nous avons prouvé correct et complet et sur lequel nous avons fait des expérimentations. Dans ce chapitre nous abordons le cas général : les grammaires sont des Regular Tree Grammars (RTG) et peuvent être récursives.

La suite du chapitre sera organisée comme suit : dans la section 3.1 nous trouvons les notions préliminaires, l'algorithme pour l'inclusion relâchée de schémas XML est décrite en section 3.2 tandis que dans la section 3.3 nous analysons quelques résultats expérimentaux. Nous terminerons par l'état de l'art en section 3.4, la conclusion et les perspectives dans la section 3.5. Les preuves sont en Annexe A.1.

3.1 Préliminaires

Nous allons premièrement définir l'inclusion relâchée sur les arbres et ensuite la définir sur les grammaires d'arbres RTG.

Définition 3.1 (Arbre résultat après une suppression de nœud)
Pour un arbre t' et une position non vide q de t', nous dénotons par $Rem_q(t') = t$ l'arbre t obtenu en supprimant le nœud à la position q de t' (un nœud supprimé, est remplacé par ses enfants s'il en a). Formellement nous avons :

1. $t(\epsilon) = t'(\epsilon)$,
2. $\forall p \in Pos(t')$ tel que $p < q : t(p) = t'(p)$,
3. $\forall p \in Pos(t')$ tel que $p \prec q : t|_p = t'|_p$,
4. Soient $q.0, q.1, ..., q.n \in Pos(t')$ les positions filles de la position q. Si q n'a pas de fils, soit $n = -1$. Nous supposons que $q = s.k$ où $s \in \mathbb{N}^*$ et $k \in \mathbb{N}$. Nous avons :
   - $t|_{s.(k+n+i)} = t'|_{s.(k+i)}$ pour tout i tel que $i > 0$ et $s.(k+i) \in Pos(t')$ (les frères situés à droite de q décalent vers la droite),
   - $t|_{s.(k+i)} = t'|_{s.k.i}$ pour tout i tel que $0 \leq i \leq n$ (les enfants montent d'un niveau dans t').

□

Définition 3.2 (Inclusion relâchée d'arbres)
L'arbre t est **inclus relâché** dans t', dénoté par $t \lhd t'$, s'il existe une séquence de positions $q_1 \dots q_n$ telle qu'on a $t = Rem_{q_n}(\cdots Rem_{q_1}(t'))$.

□

Exemple 3.2 Considérons la Figure 3.1. L'arbre t peut être obtenu à partir de t' en appliquant les opérations $t = Rem_0(Rem_1(Rem_{1.1}(Rem_{1.2.0.0}(Rem_{1.2.0.1}(Rem_{1.2.0.1.1}(t'))))))$ et donc nous concluons que $t \lhd t'$. Remarquons que pour chaque nœud dans t, il existe un nœud dans t' avec la même étiquette, et ce matching préserve l'ordre vertical (c'est-à-dire que si un nœud a est ancêtre de b dans t' alors a est aussi ancêtre de b dans t) et l'ordre gauche-droite (c'est-à-dire que si un nœud a est à gauche de b dans t' alors a est aussi à gauche de b dans t). Cependant l'arbre $t_1 = paper(biblio, year)$ n'est pas inclus relâché dans t' puisque le nœud *biblio* devrait être à droite du nœud *year*.

⊠

Définition 3.3 (Inclusion relâchée de RTLs)
Soit G, G' deux RTGs. L'**ensemble des arbres inclus relâchés** dans $L(G')$, noté $WI(L(G'))$, est défini par $WI(L(G')) = \{t \mid \exists t' \in L(G'),\ t \lhd t'\}$. Le langage $L(G)$ est **inclus relâché** dans $L(G')$, dénoté par $L(G) \lhd L(G')$, si et seulement si $L(G) \subseteq WI(L(G'))$. En d'autres mots $L(G)$ est **inclus relâché** dans $L(G')$, si pour tout arbre t dans $L(G)$ il existe un arbre t' dans $L(G')$ tel que $t \lhd t'$.

□

Exemple 3.3 Le langage $\{a(b(c)), a(b,d), a(g), a(c)\}$ est *inclus relâché* dans le langage $\{a(f, b(c), e, d), a(f(g), d)\}$. Aussi le langage $L(G)$ généré par la grammaire G de l'Exemple 3.1 est *inclus relâché* dans $L(G')$ généré par la grammaire G' de l'Exemple 3.1.

⊠

3.2 Inclusion relâchée pour les Regular Tree Grammars

Pour décider si $L(G)$ est inclus relâché dans $L(G')$, nous procédons par les différentes étapes suivantes :

1. En partant de G', nous calculons la grammaire $WI(G')$ qui génère le langage d'arbres $WI(L(G')) = \{t \mid \exists t' \in L(G'),\ t \lhd t'\}$;
2. Ensuite nous vérifions si $L(G) \subseteq WI(L(G'))$, c'est-à-dire l'inclusion classique de langages réguliers d'arbres. Nous avons $L(G) \lhd L(G')$ ssi $L(G) \subseteq WI(L(G'))$.

Nous allons nous intéresser au calcul de l'ensemble des arbres inclus relâchés $WI(L(G'))$ pour un langage $L(G')$ donné. Notre méthode va transformer la grammaire G' pour obtenir la grammaire qui génère le langage $WI(L(G'))$. La méthode que nous proposons, permet de résoudre le cas où la grammaire G' est récursive. Des preuves de correction et de complétude de notre algorithme seront données. Nous allons d'abord introduire une méthode qui traite les grammaires d'arbres non récursives et ensuite montrer la difficulté pour les grammaires d'arbres récursives. Nous n'aborderons pas l'algorithme pour tester l'inclusion classique entre deux langages réguliers d'arbres. Ce test est décidable puisque G et $WI(G')$ sont des grammaires régulières. Dans [26, 43, 50, 51], nous pouvons trouver des algorithmes efficaces pour tester cette inclusion mais la complexité de tels algorithmes reste exponentiel dans le pire des cas.

3.2.1 Cas de grammaire non récursive

Pour calculer la grammaire qui génère $WI(L(G'))$, nous avons proposé dans [8] une méthode pour le cas où la grammaire G' n'est pas récursive. L'intuition de la méthode est la suivante :

1. pour chaque non-terminal A qui apparaît dans une règle de production, on remplace le non-terminal A par l'expression $A|reg(A)$ dans la règle de production. Ainsi nous pouvons générer le non-terminal ou directement ses enfants, et plus généralement ses descendants en sautant comme cela plusieurs niveaux dans la grammaire ;
2. chaque non-terminal remplacé est marqué, pour éviter de le remplacer de nouveau. Une fois que tous les non-terminaux des règles de productions sont marqués, nous obtenons la grammaire qui génère le langage $WI(L(G'))$.
3. la méthode termine bien car nous avons supposé que la grammaire n'est pas récursive.

G' :	G'' :
$A \to a[B]$	$A \to a[B\|(C\|\epsilon)]$
$B \to b[C]$	$B \to b[(C\|\epsilon)]$
$C \to c[\epsilon]$	$C \to c[\epsilon]$

FIGURE 3.2 – *Cas 1 : Exemple de grammaires pour le cas non récursif*

Exemple 3.4 Soit G' la grammaire de la Figure 3.2. En appliquant la méthode décrite ci-dessus pour calculer la grammaire des sous-arbres relâchés de G' on obtient G''. La grammaire G' génère l'arbre unique $a(b(c))$ alors que la grammaire G' permet de générer les arbres a, $a(b)$, $a(c)$ et $a(b(c))$ qui sont des sous-arbres relâchés de l'arbre $a(b(c))$.

⊠

Malheureusement, cette méthode ne marche pas lorsque la grammaire G' est récursive car elle ne termine pas dans ce cas. Considérons la grammaire G'_1 de la Figure 3.3. Cette grammaire est récursive à cause de la règle de A qui contient un A dans son expression régulière. Si nous appliquons la méthode précédente, nous obtenons pour le non-terminal A la règle suivante : $A \to a[(B|\epsilon).(A|((B|\epsilon).(A|\epsilon).(C|\epsilon))|\epsilon).(C|\epsilon)]$. Cependant une nouvelle occurrence de A apparaît dans la partie droite de la nouvelle règle ce qui fait que cette méthode ne pourra pas terminer. Si nous arrêtons de remplacer les nouveaux A introduits après un certain nombre d'étapes, la grammaire obtenue ne génère pas $WI(L(G'_1))$. Dans cet exemple simple, nous pouvons facilement voir que la grammaire qui génère $WI(L(G'_1))$ est la grammaire G''_1 de la Figure 3.3. Avec G''_1, nous pouvons obtenir un nombre quelconque de b à gauche de a et un nombre quelconque de c à droite de a dans un arbre généré par G''_1. Aussi les b sont toujours à gauche de a et les c sont toujours à droite de a.

G'_1 :	G''_1 :
$A \to a[B.(A\|\epsilon).C]$	$A \to a[B^*.(A\|\epsilon).C^*]$
$B \to b[\epsilon]$	$B \to b[\epsilon]$
$C \to c[\epsilon]$	$C \to c[\epsilon]$

FIGURE 3.3 – *Cas 2 : Exemple de grammaires pour le cas récursif*

Considérons maintenant la grammaire G'_2 de la Figure 3.4 qui est aussi récursive. Dans G'_2 nous avons deux appels récursifs à A. Pour les arbres relâchés de cette grammaire, nous pouvons avoir un c ou un a à gauche d'un b dans un arbre généré par $WI(L(G'_2))$, ce qui n'est pas le cas pour G'_2 et aussi pour $WI(L(G'_1))$. En fait le langage $WI(L(G'_2))$ peut être généré par la grammaire G''_2 de la Figure 3.4. Dans les arbres générés par G''_2, il n'y a plus d'ordre entre les fils de a comme c'était le cas pour G'_2 où comme fils de a on avait un b suivi de a qui est suivi de c.

G'_2 :	G''_2 :
$A \to a[B.(A.A\|\epsilon).C]$	$A \to a[(A\|B\|C)^*]$
$B \to b[\epsilon]$	$B \to b[\epsilon]$
$C \to c[\epsilon]$	$C \to c[\epsilon]$

FIGURE 3.4 – *Cas 3 : Exemple de grammaires pour le cas récursif*

Nous pouvons conclure avec ces exemples que le cas des grammaires récursives est beaucoup plus compliqué. Dans la section suivante, nous proposons une méthode pour résoudre le cas récursif.

3.2.2 Cas de grammaire récursive

Dans cette section nous allons nous intéresser aux grammaires récursives pour le calcul de la grammaire des sous-arbres relâchés. Nous pouvons noter que notre méthode pour résoudre le cas des grammaires récursives, marchera aussi pour les grammaires non récursives.

Pour résoudre le cas récursif, nous divisons les non-terminaux de la grammaire en 3 catégories : non-récursif ou 0-récursif, 1-récursif et 2-récursif. Intuitivement, un non-terminal A de la grammaire G est 2-récursif s'il existe un arbre $t \in L_G^{nt}(A)$ et A apparaît dans t à au moins deux positions non vide p, q telles que p et q sont parallèles ($p \parallel q$). Un non-terminal A est 1-récursif si A n'est pas 2-récursif, et si A apparaît à une position non vide dans $L_G^{nt}(A)$. Un non-terminal n'est pas récursif si A n'est pas 2-récursif et A n'est pas 1-récursif.

Exemple 3.5 Considérons la grammaire G de l'Exemple 3.1. Nous avons : P est 2-récursif puisque P génère B et B peut générer l'arbre $biblio(P, P)$. B aussi est 2-récursif. Les non-terminaux F, L, T, Y sont non-récursifs. La grammaire G ne possède pas de non-terminaux qui sont 1-récursif.

⊠

Nous allons maintenant définir les types de récursivités des non-terminaux de façon formelle.

Définition 3.4 (Relation de dépendance sur les non-terminaux)
Soit $G = (N, \Sigma, S, P)$ une RTG en forme normale. Pour une expression régulière E, $nt(E)$ dénote l'ensemble des non-terminaux apparaissant dans E.

- La relation $>$ sur les non-terminaux est définie par :
  $A > B$ si $\exists\, A \to a[E] \in P$ tel que $B \in nt(E)$.
- La relation $>$ sur les multi-ensembles [1] de non-terminaux de taille au plus 2, est définie par :
  - $\{A\} > \{B\}$ si $A > B$,
  - $\{A, B\} > \{C, D\}$ si $A = C$ et $B > D$,
  - $\{A\} > \{C, D\}$ s'il existe une règle de production $A \to a[E]$ dans G et un mot $u \in L^w(E)$ de la forme $u = u_1 C u_2 D u_3$, c'est-à-dire que les non-terminaux C et D sont contenus dans le même mot.

□

Remarque 3.1 *Pour vérifier que $\{A\} > \{C, D\}$, c'est-à-dire que l'on peut retrouver C et D dans un même mot généré par l'expression $reg(A)$=E, nous pouvons utiliser la fonction* **in** *définie récursivement sur E par $in(C, D, E) =$:*

1. *si E est un non-terminal ou $E = \epsilon$ alors retourner faux ;*
2. *si E=E_1^* ou E=E_1^+ alors retourner* $(\mathbf{C} \in \mathbf{nt}(\mathbf{E_1})) \wedge (\mathbf{D} \in \mathbf{nt}(\mathbf{E_1}))$ *;*
3. *si E=$E_1^?$ alors retourner* $\mathbf{in}(\mathbf{C}, \mathbf{D}, \mathbf{E_1})$ *;*
4. *si E=$E_1|E_2$ alors retourner* $\mathbf{in}(\mathbf{C}, \mathbf{D}, \mathbf{E_1}) \vee \mathbf{in}(\mathbf{C}, \mathbf{D}, \mathbf{E_2})$ *;*
5. *si E=$E_1.E_2$ alors retourner* $((\mathbf{C} \in \mathbf{nt}(\mathbf{E_1})) \wedge (\mathbf{D} \in \mathbf{nt}(\mathbf{E_2}))) \vee$
   $((\mathbf{C} \in \mathbf{nt}(\mathbf{E_2})) \wedge (\mathbf{D} \in \mathbf{nt}(\mathbf{E_1}))) \vee \mathbf{in}(\mathbf{C}, \mathbf{D}, \mathbf{E_1}) \vee \mathbf{in}(\mathbf{C}, \mathbf{D}, \mathbf{E_2})$ *;*

1. Du fait que nous avons les multi-ensembles, remarquons que $\{A, B\} = \{B, A\}$ et $\{C, D\} = \{D, C\}$.

La fonction **in** *termine car les appels récursifs sont appelés sur des sous-expressions de E. La complexité en temps est de $O(|E|)$ où $|E|$ est la taille de E (vu comme un arbre).*

Définition 3.5 (Type de récursivité des non-terminaux)
Soit $>^+$ la fermeture transitive de la relation $>$. Nous avons :

- le non-terminal A est 2-récursif ssi $\{A\} >^+ \{A, A\}$;
- le non-terminal A est 1-récursif ssi $A >^+ A$ et A n'est pas 2-récursif ;
- le non-terminal A est 0-récursif ssi A n'est ni 2-récursif, ni 1-récursif.

□

Pour calculer la fermeture transitive de $>$, nous pouvons utiliser l'algorithme de Warshall [123]. L'algorithme de Warshall prend en paramètre une matrice qui représente la relation $>$ et calcule la matrice pour la relation $>^+$. Si nous avons n non-terminaux dans G, alors nous avons $p = n + \frac{n.(n+1)}{2}$ multi-ensembles de taille au plus 2. Ainsi une matrice booléenne $p \times p$ peut représenter $>$, et par conséquent la complexité en temps pour le calcul de $>^+$ est en $O(p^3) = O(n^6)$ qui est polynomial.

Exemple 3.6 Considérons la grammaire G de l'Exemple 3.1. Nous avons $\{P\} > \{B\} > \{P, P\}$ et donc P est 2-récursif. Nous avons aussi $\neg(\{F\} >^+ \{F, F\})$ et $\neg(F >^+ F)$, par conséquent F n'est pas récursif. ⊠

Maintenant pour définir l'algorithme pour calculer la grammaire qui génère le langage des sous-arbres relâchés $WI(L(G))$ de G, nous avons besoin des notions supplémentaires suivantes : une relation de classe d'équivalence entre les non-terminaux, les descendants d'un non-terminal, les non-terminaux pouvant être à gauche (ou à droite) d'un non-terminal A dans un mot. Ces notions sont formalisées dans la définition suivante.

Définition 3.6 (Notion d'équivalence et fonctions sur les non-terminaux)
Soit $G = (N, \Sigma, S, P)$ une RTG en forme normale.

1. La relation $\equiv$ est définie sur les non-terminaux par $A \equiv B$ si $A >^* B \wedge B >^* A$, où $>^*$ dénote la fermeture transitive-réflexive de $>$. La relation $\equiv$ est une relation d'équivalence car si $A \equiv B$ alors A et B ont le même type de récursivité. Nous utilisons $\widehat{A}$ pour dénoter la classe d'équivalence de A.
2. $Succ(A)$ est l'ensemble des non-terminaux tels que $Succ(A) = \{X \in N \mid A >^* X\}$. Pour un ensemble $Q = \{A_1, \ldots, A_n\}$, $Succ(Q) = Succ(A_1) \cup \cdots \cup Succ(A_n)$.
3. $Left(A)$ est l'ensemble de non-terminaux définis par :

$$Left(A) = \{X \in N \mid \exists B, C \in \widehat{A}, \exists B \to b[E] \in P, \exists u \in L^w(E), u = u_1 X u_2 C u_3\}.$$

   Similairement, $Right(A)$ est défini par :

$$Right(A) = \{X \in N \mid \exists B, C \in \widehat{A}, \exists B \to b[E] \in P, \exists u \in L^w(E), u = u_1 C u_2 X u_3\}.$$

4. $reg(A)$ est l'expression régulière de la règle de production de A.
5. $\widehat{reg}(A) = reg(A)|reg(B_1)|\cdots|reg(B_n)$ where $\widehat{A} = \{A, B_1, \ldots, B_n\}$.

□

Exemple 3.7 Considérons la grammaire G' de l'Exemple 3.1. Nous avons :

- $P \equiv B$, car $P > Pa > B$ et $B > P$.

- $\widehat{P} = \{P, Pa, B\}$.
- $Succ(A) = \{A, F, L\}$.
- $Left(P)$ est défini à partir des non-terminaux équivalents ($\equiv$) à P, c'est-à-dire B, P, Pa, et de la grammaire G' qui contient les règles de productions (parmi d'autres) :
  B $\rightarrow$ biblio$[P^+]$, P $\rightarrow$ publi$[A^*.Pa]$, Pa $\rightarrow$ paper$[T.Y.B^?]$.
  P^+ peut générer $P.P$, et donc nous avons $Left(P) = \{P\} \cup \{A\} \cup \{T, Y\} = \{P, A, T, Y\}$.
- $reg(P) = A^*.Pa$
- $\widehat{reg}(P) = reg(P)|reg(Pa)|reg(B) = (A^*.Pa)|(T.Y.B^?)|P^+$.

⊠

Nous avons les propriétés suivantes : (i) lorsque deux non-terminaux A et B sont équivalents, alors les successeurs, les non-terminaux pouvant être à gauche (ou à droite) de A sont respectivement les mêmes que les successeurs, les non-terminaux pouvant être à gauche (ou à droite) de B ; (ii) La classe d'équivalence d'un non-terminal A non récursif est un singleton qui ne contient que A. La définition formelle de ces propriétés et les preuves sont les suivantes.

Lemme 3.1 *Soit A, B deux non-terminaux. Nous avons :*

- *Si $A \equiv B$ alors $Succ(A) = Succ(B)$, $Left(A) = Left(B)$, $Right(A) = Right(B)$, $\widehat{reg}(A) = \widehat{reg}(B)$.*
- *Si A, B ne sont pas récursifs, alors $A \neq B$ implique $A \not\equiv B$, par conséquent $\widehat{A} = \{A\}$ et $\widehat{B} = \{B\}$, c'est-à-dire que les classes d'équivalence des non-terminaux non récursifs sont des singletons.*

Démonstration. La première partie du lemme est triviale.
Voici la preuve de la deuxième partie du lemme. Le fait qu'un non-terminal A est (1 ou 2)-récursif est exprimé par $(A >^+ A) \implies (A >^+ A \wedge \neg(\{A\} >^+ \{A, A\})) \vee (\{A\} >^+ \{A, A\})$. Par conséquent, si A n'est pas récursif alors $A \not>^+ A$. Maintenant supposons que A et B ne sont pas récursifs, $A \neq B$ et $A \equiv B$ (preuve par contradiction). Dans ce cas nous avons $A >^+ B \wedge B >^+ A$ qui conduit à $A >^+ A$, ceci est impossible car si A n'est pas récursif nous avons $A \not>^+ A$. ◁

L'intuition de notre algorithme est basée sur les idées suivantes :

1. Rappelons que l'inclusion relâchée de deux arbres t et t' consiste à supprimer une série de positions dans t' pour obtenir t. Lorsqu'on supprime un nœud, on le remplace par ses enfants. Ainsi il est possible de faire remonter les descendants d'un nœud de plusieurs niveaux. Les descendants possibles que peut avoir un nœud généré par le non-terminal A sont calculés par la fonction $Succ(A)$.
2. Lorsqu'un non-terminal A est non récursif, nous allons appliquer la même méthode expliquée dans la section 3.2.1 pour calculer la nouvelle règle du non-terminal A. Cette méthode garde la structure (l'ordre entre les non-terminaux) des expressions régulières, en remplaçant chaque non-terminal B par $B|reg(B)$. Dans le nouvel algorithme, comme nous avons des non-terminaux 1-récursif et 2-récursif, $B|reg(B)$ sera remplacé par la nouvelle expression de B lorsque B est 1-récursif ou 2-récursif. Ce cas sera illustré dans l'Exemple 3.8.
3. Lorsqu'un non terminal A est récursif, nous avons une répétition dans le sens de la verticale dans un arbre généré par la grammaire. Avec la suppression possible d'un nœud qui revient à mettre à plat les niveaux dans la grammaire, la répétition verticale entraîne une répétition horizontale dans la nouvelle expression régulière de A.

4. Lorsqu'un non-terminal A est 2-récursif, nous avons vu dans l'exemple de la section 3.2.1 que l'ordre entre les fils d'un nœud généré par A n'est pas préservé. Ainsi tous les descendants de A peuvent arriver à n'importe quelle position dans la nouvelle expression régulière. Par exemple si A, B, C sont les descendants de A alors sa nouvelle expression régulière sera $(A|B|C)^*$. Ce cas sera illustré dans l'Exemple 3.9.
5. Lorsqu'un non-terminal A est 1-récursif, Nous avons vu dans l'exemple de la section 3.2.1 que l'ordre gauche-droite entre les fils potentiellement à gauche de A et ceux potentiellement à droite de A est conservé. Ce cas est beaucoup plus compliqué que les autres cas (0-récursif et 2-récursif) car le non-terminal A qui est 1-récursif doit être traité à la fois comme un non-terminal 0-récursif et comme un non-terminal 2-récursif. Il doit être traité comme un 0-récursif dans le sens où l'ordre gauche-droite entre les fils à gauche de A et les fils à droite de A est conservé donc on ne peut pas avoir juste une expression régulière comme une répétition de ses descendants comme pour le cas 2-récursif ; et il doit aussi être traité comme un 2-récursif dans le sens où l'ordre entre que les fils potentiellement à gauche (resp. potentiellement à droite) n'est pas préservé. Plus précisément si B et C sont à gauche de A alors on peut obtenir un C à gauche de B et inversement. Donc pour résumer, dans la nouvelle expression régulière de A, la partie concernant les non-terminaux à gauche (resp. non-terminaux à droite) est gérée de la même façon que le cas 2-récursif et le lien entre les non-terminaux à gauche et les non-terminaux à droite est géré de la même façon que le cas 0-récursif. Ce cas sera illustré dans les Exemples 3.10 et 3.11.
6. Les non-terminaux qui sont dans la même classe d'équivalence ont le même type de récursivité (Définition 3.6), et un point très intéressant est qu'ils possèdent la même expression régulière dans la grammaire qui génère le langage $WI(L(G))$.

En nous basant sur les idées établies ci-dessus, nous pouvons définir l'algorithme qui résout le cas récursif.

Définition 3.7 (Expression régulière *Ch* des non-terminaux de la nouvelle grammaire)
Pour chaque non-terminal A, nous définissons l'expression régulière Ch de façon récurcive. Nous allons considérer chaque ensemble de non-terminaux $\{A_1, \ldots, A_n\}$ comme étant l'expression régulière $(A_1| \cdots |A_n)$.

- si A est 2-récursif, alors $Ch(A) = (Succ(A))^*$
- si A est 1-récursif, alors $Ch(A) = (Succ(Left(A)))^*.Ch^{rex}_{\widehat{A}}(\widehat{reg}(A)).(Succ(Right(A)))^*$
- si A n'est pas récursif, alors $Ch(A) = Ch^{rex}_{\widehat{A}}(reg(A))$

avec $Ch^{rex}_{\widehat{A}}(E)$ étant l'expression régulière obtenue à partir de E en remplaçant chaque non-terminal B apparaissant dans E par $Ch_{\widehat{A}}(B)$, où

$$Ch_{\widehat{A}}(B) = \left| \begin{array}{ll} \text{- } \widehat{B}|\epsilon & \text{si } B \text{ est 1-récursif et } B \in \widehat{A} \\ \text{- } Ch(B) & \text{si } (B \text{ est 2-récursif}) \text{ ou } (B \text{ est 1-récursif et } B \notin \widehat{A}) \\ \text{- } B|Ch(B) & \text{si } B \text{ n'est pas récursif} \end{array} \right.$$

Par convention $Ch^{rex}_{\widehat{A}}(\epsilon) = \epsilon$ et $Ch(\epsilon) = \epsilon$.

□

Par exemple lorsque nous prenons $Ch(A)$ pour un non-terminal A qui est 1-récursif, $(Succ(Left(A)))^*$ représente la partie de l'expression régulière qui génère les non-terminaux potentiellement à gauche de A et leur descendant ; $(Succ(Right(A)))^*$ représente la partie de l'expression régulière qui génère les non-terminaux potentiellement à droite de A et leur descendant ; et $Ch^{rex}_{\widehat{A}}(\widehat{reg}(A))$ représente la partie de l'expression régulière qui génère la partie qui conserve l'ordre gauche droite et qui fait le lien entre les non-terminaux à gauche

de A et les non-terminaux à droite de A. Cette dernière partie est la même que la façon dont est calculé Ch pour un non-terminal non récursif. Nous pouvons aussi remarquer qu'avec $(Succ(Left(A)))^*$ et $(Succ(Right(A)))^*$, l'ordre entre les non-terminaux n'est pas respecté.

Notre algorithme pour calculer la grammaire générant $WI(L(G))$ pour une grammaire G éventuellement récursive est le suivant :

Algorithme 1 Calcul de la grammaire générant tous les sous-arbres relâchés d'un langage

Entrée : Une RTG $G = (N, \Sigma, S, P)$ en forme normale
Sortie : La RTG G' qui génère le langage $WI(L(G))$
1: $P' := \emptyset$
2: **pour tout** $A \to a[E] \in P$ **faire**
3: Ajouter $A \to a[Ch(A)]$ à P'
4: **fin pour**
5: **retourner** (N, Σ, S, P')

Comme remarque, la nouvelle grammaire a exactement le même nombre de règles de production que la grammaire initiale. Aussi l'ensemble de non-terminaux N, l'alphabet Σ et l'ensemble de non-terminaux initiaux de la nouvelle grammaire sont les mêmes que ceux de la grammaire initiale.

Théorème 3.1 *Le calcul de Ch termine toujours, et nous avons* $L(G') = WI(L(G))$.

La preuve du Théorème 3.1 établissant que le langage $L(G')$ est le même que $WI(L(G))$ se trouve en Annexe, section A.1. Nous allons considérer des exemples pour donner plus d'intuitions sur l'Algorithme 1 et aussi montrer les différentes situations que l'on peut rencontrer.

Exemple 3.8 Considérons la grammaire G qui contient les règles de productions suivantes :

$$A \to a[B],\ B \to b[C],\ C \to c[\epsilon]$$

A est le non-terminal initial. Notons que A, B, C ne sont pas récursifs.
$Ch(C) = Ch^{rex}_{\widehat{C}}(reg(C)) = Ch^{rex}_{\widehat{C}}(\epsilon) = \epsilon$.
$Ch(B) = Ch^{rex}_{\widehat{B}}(reg(B)) = Ch^{rex}_{\widehat{B}}(C) = Ch_{\widehat{B}}(C) = C|Ch(C) = C|\epsilon$.
$Ch(A) = Ch^{rex}_{\widehat{A}}(reg(A)) = Ch^{rex}_{\widehat{A}}(B) = Ch_{\widehat{A}}(B) = B|Ch(B) = B|(C|\epsilon)$.

Ainsi, nous obtenons les règles suivantes pour la nouvelle grammaire G' :

$$A \to a[B|(C|\epsilon)],\ B \to b[C|\epsilon],\ C \to c[\epsilon]$$

qui génère bien $WI(L(G))$. Dans ce cas particulier, où il n'y a pas de non-terminaux récursifs, nous obtenons bien la même grammaire que pour la méthode décrite en Section 3.2.1 et dont les détails se trouvent dans [8], malgré que l'algorithme soit formalisé d'une autre manière. ⊠

Exemple 3.9 Considérons la grammaire G qui contient les règles de productions suivantes :

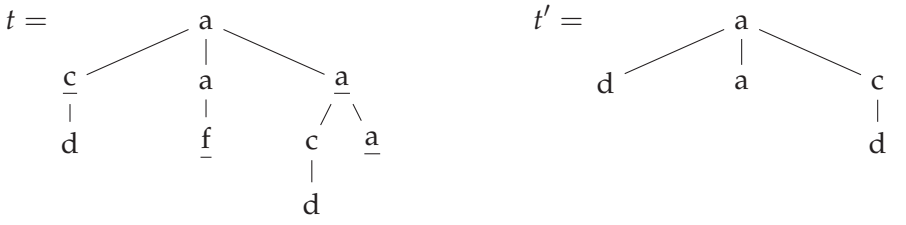
$$A \to a[(C.A.A^?)|F|\epsilon],\ C \to c[D],\ D \to d[\epsilon],\ F \to f[\epsilon]$$

Pour cette grammaire nous avons A qui est 2-récursif ; C, D, F qui ne sont pas récursifs.
$Ch(D) = Ch(F) = \epsilon$. $Ch(C) = D|\epsilon$.
$Succ(A) = \{A, C, D, F\}$. En considérant $Succ(A)$ comme l'expression $A|C|D|F$ nous avons $Ch(A) = (A|C|D|F)^*$.

Ainsi, nous obtenons les règles suivantes pour la nouvelle grammaire G' :

$$A \to a[(A|C|D|F)^*],\ C \to c[D|\epsilon],\ D \to d[\epsilon],\ F \to f[\epsilon]$$

L'arbre t ci-dessous est généré par G. En supprimant les nœuds soulignés, nous obtenons $t' \lhd t$, et t' est bien généré par la grammaire G'. Notons que a est un frère gauche de c dans t', qui est impossible pour un arbre généré par G.



⊠

Exemple 3.10 Considérons la grammaire G qui contient les règles de productions suivantes (A est le symbole initiale) :

$$A \to a[(B.C.A^?.H)|F],\ B \to b[\epsilon],\ C \to c[D],\ D \to d[\epsilon],\ H \to h[\epsilon],\ F \to f[\epsilon]$$

Pour cette grammaire nous avons A qui est 1-récursif ; B, C, D, H, F qui ne sont pas récursifs. Nous avons aussi $Ch(B)=Ch(D)=Ch(H)=Ch(F)=\epsilon$; $Ch(C)=D|\epsilon$; $\widehat{A}=\{A\}$ et donc $\widehat{reg}(A) = reg(A) = (B.C.A^?.H)|F$.

$$\begin{aligned} Ch^{rex}_{\widehat{A}}(\widehat{reg}(A)) &= Ch^{rex}_{\widehat{A}}((B.C.A^?.H)|F) = (Ch_{\widehat{A}}(B).Ch_{\widehat{A}}(C).Ch_{\widehat{A}}(A)^?.Ch_{\widehat{A}}(H))|Ch_{\widehat{A}}(F) \\ &= ((B|Ch(B)).(C|Ch(C)).(A|\epsilon)^?.(H|Ch(H)))|(F|Ch(F)) \\ &= ((B|\epsilon).(C|(D|\epsilon)).(A|\epsilon)^?.(H|\epsilon))|(F|\epsilon) \\ &= (B^?.(C|D|\epsilon).A^?.H^?)|F^? \quad \text{(après simplification)} \end{aligned}$$

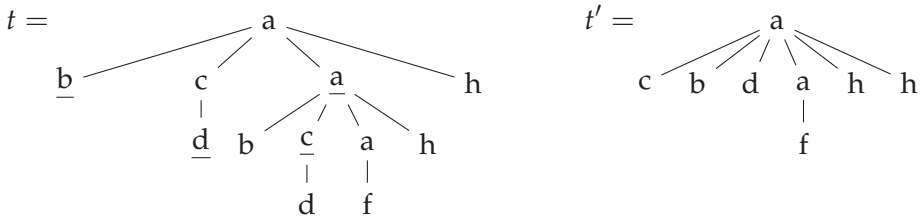
$Left(A)=\{B,C\}$ et $Succ(Left(A))=\{B,C,D\}$; $Right(A)=\{H\}$ et $Succ(Right(A))=\{H\}$. En les considérant comme des expressions régulières (au lieu d'ensembles), $Succ(Left(A)) = B|C|D$ et $Succ(Right(A)) = H$. Ainsi nous obtenons pour $Ch(A)$:

$$\begin{aligned} Ch(A) &= (Succ(Left(A)))^*.Ch^{rex}_{\widehat{A}}(\widehat{reg}(A)).(Succ(Right(A)))^* \\ &= (B|C|D)^*.[(B^?.(C|D|\epsilon).A^?.H^?)|F^?].H^* \\ &= (B|C|D)^*.(A^?|F^?).H^* \quad \text{(après simplification)} \end{aligned}$$

La nouvelle grammaire G' possède les règles suivantes :

$$A \to a[(B|C|D)^*.(A^?|F^?).H^*],\ B \to b[\epsilon],\ C \to c[D|\epsilon],\ D \to d[\epsilon],\ H \to h[\epsilon],\ F \to f[\epsilon]$$

L'arbre t ci-dessous est généré par G. En supprimant les nœuds soulignés, nous obtenons $t' \lhd t$, et t' est bien généré par la grammaire G'. Notons que c est un frère gauche de b dans t', qui est impossible pour un arbre généré par G. Comme b et c sont des nœuds à gauche de a dans t par exemple, ils peuvent apparaître dans n'importe quel ordre dans t'. Par contre les nœuds b, c, d sont nécessairement à gauche de h dans t'.



⊠

Exemple 3.11 L'exemple précédent ne montrait pas le rôle des classes d'équivalences. Considérons la grammaire G ayant comme règles de productions :

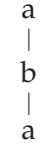
$$A \to a[B^?],\ B \to b[A]$$

où A et B sont 1-récursifs. Nous avons $A \equiv B$ et donc $\widehat{A} = \widehat{B} = \{A, B\}$; $Left(A) = Left(B) = Right(A) = Right(B) = \emptyset$. Nous obtenons pour $Ch(A)$:

$$\begin{aligned} Ch(A) &= Ch^{rex}_{\widehat{A}}(\widehat{reg}(A)) = Ch^{rex}_{\widehat{A}}(B^?|A) = (Ch_{\widehat{A}}(B))^?|(Ch_{\widehat{A}}(A)) \\ &= (\widehat{B}|\epsilon)^?|(\widehat{A}|\epsilon) = (A|B|\epsilon)^?|(A|B|\epsilon) \\ &= A|B|\epsilon \quad \text{(après simplification)} \end{aligned}$$

Nous pouvons noter que $\widehat{A}$ et $\widehat{B}$ ont été remplacés par $A|B$, qui est nécessaire comme illustré par les arbres t et t' ci-dessous. Puisque $A \equiv B$ alors $Ch(B) = Ch(A)$. Nous déduisons donc que les règles de la grammaires G' sont :

$$A \to a[A|B|\epsilon],\ B \to b[A|B|\epsilon]$$

L'arbre $t =$  est généré par G. En supprimant b on obtient $t' =$ qui est généré par G' comme souhaité.

⊠

3.3 Implémentation et Résultats Expérimentaux

Nous avons implémenté un prototype en Java et les expériences ont été effectuées sur un Intel Quad Core i3-2310M avec 2.10GHz et 8GB de mémoire RAM. Le calcul de la grammaire des sous-arbres relâchés se fait en deux temps : en premier nous recherchons le type de récursivité de la grammaire initiale ; et ensuite en fonction du type de récursivité d'un non-terminal, nous déterminons sa nouvelle règle de production. En comparant les deux étapes, il se fait que la première étape (c'est-à-dire celle qui calcule le type de récursivité des non-terminaux) est celle qui consomme plus de temps. La plus difficile tâche est de décider si un non-terminal est 2-récursif or 1-récursif. Pour cette partie, nous avons implémenté deux algorithmes :

1. l'un qui utilise la méthode Warshall pour calculer $>^+$ et dont nous avons montré que la complexité en temps est de $O(n^6)$ où n est le nombre de non-terminaux [2],
2. l'autre méthode consiste en la comparaison de cycles dans un graphe représentant la relation $>$ sur les non-terminaux (et non sur les multi-ensembles). Dans le pire des cas, la complexité de cette méthode est exponentielle puisque nous calculons tous les cycles dans le graphe. Le calcul de tous les cycles est un problème d'énumération et a une complexité exponentielle dans le pire des cas. Le pire des cas arrive lorsque le graphe est complet, c'est-à-dire qu'il y a une arête entre chaque sommet du graphe avec chacun des autres sommets. Dans notre contexte le graphe issu de la relation $>$ dans une RTG est complet si tous les non-terminaux apparaissent dans toutes les

2. Puisque les grammaires sont en forme normales, n est aussi le nombre de règles de production.

règles, ce qui est rare car dans le cas pratique les schémas XML (DTD, XML Schema) ne possèdent pas de tels règles. Par conséquent nous avons peu de cycles dans les graphes et ceci rend cette méthode beaucoup plus efficace.

Pour résumer, le premier algorithme basé sur la méthode de Warshall dépend du nombre de non-terminaux n, tandis que le deuxième algorithme dépend du nombre de cycles dans un graphe issu de la relation $>$.

Pour mieux comprendre le deuxième algorithme, considérons la grammaire G qui contient les règles suivantes :

$$A \to a[B|C.D],\ B \to b[I|\epsilon],\ C \to c[A|\epsilon],\ D \to d[A|\epsilon],$$

$$I \to i[E.B.F],\ E \to e[\epsilon],\ F \to f[\epsilon]$$

Le graphe obtenu à partir de la grammaire G est le graphe $G_>$ de la Figure 3.5. A partir de

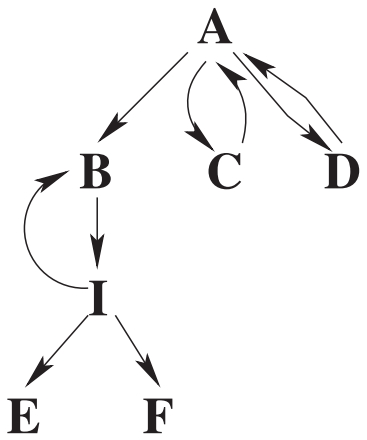


FIGURE 3.5 – *Exemple de graphe issu de la relation $>$ pour une grammaire RTG.*

ce graphe, nous pouvons calculer tous les cycles (classés par ordre alphabétique par rapport au premier symbole) suivants : ACA, ADA et BIB. Tous les non-terminaux qui sont dans un cycle sont ont moins 1-récursif. Aussi tous les non-terminaux qui sont dans le même cycle, appartiennent à la même classe d'équivalence. Pour savoir si un non-terminal est 2-récursif, il faut comparer tous les cycles qui le contiennent, deux à deux (la comparaison d'un cycle avec lui même est aussi possible). Nous comparons deux cycles débutant par le même non-terminal jusqu'à ce qu'on ait la confirmation que le non-terminal est 2-récursif. Si après les comparaisons de tous cycles contenant ce non-terminal, nous avons pas obtenu que le non-terminal en question est 2-récursif, alors nous pouvons affirmer que le non-terminal est 1-récursif. La comparaison se fait de la façon suivante : pour le non-terminal A par exemple, nous allons comparer les cycles ADA et ACA.

- nous commençons par les deuxièmes symboles des cycles et nous comparons les symboles aux mêmes positions dans les cycles. Pour notre exemple, D sera comparé à C et le dernier symbole A du cycle ADA sera comparé avec le dernier symbole A du cycle ACA.
- A chaque position i, nous vérifions si les deux symboles aux positions i peuvent se trouver dans un même mot généré par l'expression régulière associée au symbole à la position $i-1$ (qui doit être la même dans les deux cycles). Si le test est vrai alors tous les non-terminaux des deux cycles sont 2-récursifs et sont dans la même classe d'équivalence.

– Dans le cas où le test échoue, on passe à la position suivante $i+1$ seulement si les deux symboles au position i sont les mêmes ; sinon on passe à la comparaison d'autres cycles.

– Dans notre exemple D et C sont dans un même mot généré par l'expression régulière $B|C.D$ de A ce qui conduit au résultat : A, C, et D sont 2-récursif.

Pour la grammaire G, nous avons A, C, et D qui sont 2-récursif, B et I qui sont 1-récursif et E, F qui ne sont pas récursifs.

Nous avons fait tourner notre prototype sur des grammaires générées aléatoirement et sur des cas réels de grammaires. Dans le Tableau 4.1, nous montrons une comparaison entre les deux algorithmes pour calculer les types de récursivité des non-terminaux. Les notations #0-rec, #1-rec et #2-rec dénotent respectivement le nombre de non-terminaux qui sont 0-récursif, 1-récursif et 2-récursif ; #Cycles le nombre de cycles, et $|G|$ (resp. $|WI(G)|$) dénote la somme des tailles des expressions régulières[3] de la grammaire initiale (resp. la grammaire résultat $WI(G)$). Les résultats des lignes 1 à 4 concernent les DTDs générés, tandis que les lignes 5 et 6 correspondent à des cas réels de DTDs. Les résultats montrent que si $n < 50$, l'algorithme basé sur la méthode de Warshall est très efficace (il met moins de 6 secondes pour terminer) et pour des valeurs de $n > 50$ l'algorithme met beaucoup de temps à répondre. Dans la plupart des cas, l'algorithme basé sur la comparaison de cycles (Algorithme B) s'exécute plus rapidement que l'algorithme basé sur la méthode de Warshall (Algorithme A). Un exemple avec $n = 111$ (ligne 3) met 7 minutes pour l'Algorithme A alors que la réponse est immédiate pour l'Algorithme B. Lorsque le nombre de cycles est inférieur à 100, l'Algorithme B est immédiat malgré que sa complexité est exponentielle dans le pire des cas.

Maintenant considérons les DTDs qui concernent les cas réels. La DTD de la ligne 6 spécifie des annotations linguistiques des entités nommées, effectuées dans le projet Corpus National Polonais. Cette DTD est décrite en détaille dans le Chapitre 5, page 117. A partir de cette DTD nous avons 17 règles de productions et des certains non-terminaux récursifs. Les deux algorithmes donnent des réponses dans l'immédiat pour cette grammaire avec peu de règles et peu de cycles. La DTD de la ligne 6 est le schéma du XHTML[4] qui possède $n = 85$ non-terminaux et beaucoup de cycles ($\#Cycles = 9620$). Les deux algorithmes calculent les types des non-terminaux en environ 2 minutes.

	Grammaires RTG				**Temps CPU (s)**		**Tailles**	
	#0-*rec*	#1-*rec*	#2-*rec*	#Cycles	Warshall	Cycle-compar.	$\|G\|$	$\|WI(G)\|$
1	9	2	38	410	5.48	0.82	183	1900
2	34	4	12	16	5.51	0.08	126	1317
3	78	12	21	30	445	0.2	293	4590
4	8	2	16	788	0.38	1.51	276	397
5	14	0	2	1	0.08	0.01	30	76
6	30	0	55	9620	136.63	113.91	1879	22963

Tableau 3.1 – *Temps CPU (en secondes) nécessaires pour les deux algorithmes pour la détection des types de récursivité.*

3. La taille d'une expression régulière E est le nombre d'occurence de non-terminaux dans E.
4. http ://www.w3.org/TR/xhtml1/dtds.html

3.4 Travaux liés

Le problème d'inclusion relâchée d'arbres a été premièrement étudié dans [74] avec la proposition d'un algorithme, et ensuite des solutions améliorées ont été apportées dans [22, 44, 45, 98]. Notre approche diffère de ces travaux du fait que nous considérons l'inclusion relâchée sur des langages d'arbres (et non sur des arbres). Dans [98], Richter décrit un algorithme d'inclusion relâché d'arbres qui consiste à calculer une fonction de mapping entre l'arbre pattern P et l'arbre cible T. La fonction de mapping est une association entre chacun des nœuds de l'arbre pattern P à un des nœuds de l'arbre cible T. Si un mapping entre les nœuds des deux arbres existe alors P est inclus relâché dans T. L'algorithme s'arrête dès qu'il trouve une fonction de mapping (car il peut avoir plusieurs mapping entre les nœuds de P et T). L'algorithme proposé par Richter a une complexité de $O(|\Sigma_P| \cdot |T| + \#matches \cdot DEPTH(T))$ (où Σ_P est l'ensemble des étiquettes de l'arbre pattern et $\#matches$ est le nombre de pairs $(v, w) \in P * T$ avec $Label(v) = Label(w)$). Son algorithme améliore celui proposé par Mannila et Kilpelainen [74] ayant une complexité de $O(|P| \cdot |T|)$. Mannila et Kilpelainen [74] sont les premiers à proposer un algorithme en temps polynomial qui résout le problème d'inclusion relâché d'arbres. Une des applications au problème d'inclusion relâché d'arbres, comme l'est proposée dans [74, 98], est la recherche de sous-arbres dans l'arbre d'analyse syntaxique d'une phrase en langage naturel. On peut ainsi extraire une ou plusieurs sous-phrases de la phrase initiale, en ne gardant que les informations souhaitées par l'utilisateur. Un autre algorithme pour le problème d'inclusion relâchée d'arbres, qui a une meilleure complexité que l'algorithme proposé par Richter [98], est proposé dans [44]. Au lieu de calculer une fonction mapping entre l'arbre pattern P et l'arbre cible T, les auteurs proposent de partir de l'arbre T et ensuite de trouver une série de positions dans T telle que si l'on supprime les nœuds (un nœud supprimé est remplacé par ses enfants) à ces positions dans T, on obtient P. L'algorithme dans [44] a une complexité de $O(|T| \cdot min\{DEPTH(P), |leaves(P)|\})$. Dans [98], nous pouvons trouver trois autres définitions d'inclusion d'arbres :

- **inclusion chemin ordonné :** pour cette inclusion lorsque que l'on supprime un nœud alors tous ses descendants sont aussi supprimés. Pour cette inclusion, on ne peut avoir que des nœuds manquants entre nœuds frères et il n'est pas possible d'avoir de nœuds manquants entre un nœud et ses ancêtres. Ainsi, la relation père-fils est donc préservée entre les nœuds de l'arbre pattern P et ceux de l'arbre cible T.
- **inclusion région ordonnée :** pour cette inclusion lorsque que l'on supprime un nœud alors tous ses descendants sont aussi supprimés, et aussi il n'est pas possible de supprimer des nœuds entre deux nœuds voisins. La relation père-fils et la relation frère (nœud voisin immédiat) sont donc préservées entre les nœuds de l'arbre pattern P et ceux de l'arbre cible T.
- **inclusion forte :** correspond au faite que l'arbre pattern P est un sous-arbre (au sens classique Chapitre 2, Définition 2.5) de l'arbre cible T.

Il existe une hiérarchie entre ces trois définitions d'inclusion et l'inclusion relâchée. Une solution au problème d'inclusion forte est aussi une solution au problème d'inclusion région ordonnée. Une solution au problème d'inclusion région ordonnée est aussi une solution au problème d'inclusion chemin ordonné. Et enfin une solution au problème d'inclusion chemin ordonné est aussi une solution au problème d'inclusion relâchée. Nous pouvons remarquer que l'inclusion relâchée est la moins contraignante, d'où son utilisation dans l'inclusion relâchée de langages d'arbres. Aussi dans [98], nous pouvons trouver la définition d'une autre inclusion d'arbres qu'est l'inclusion d'abres non ordonnés. Cette inclusion ne tient pas compte de la relation gauche-droite entre les nœuds des arbres P et T. Par exemple

avec cette inclusion, nous avons l'arbre $a(b, c)$ qui est inclus dans l'arbre $a(c, d, b)$. Richter dans [98], montre que le problème d'inclusion d'arbres non ordonnés est NP-complet.

L'inclusion (au sens classique) de langages d'arbres est considérée dans [26, 43, 50, 51, 87]. Une étude de complexité est faite dans [87], où les auteurs identifient des cas où l'inclusion de langages est efficace. Dans [43], un algorithme polynomial pour vérifier si $L(A) \subseteq L(D)$ est donné, où A est un automate pour les arbres d'arités non borné et D est une DTD déterministe. L'inclusion relâchée est plus flexible que l'inclusion d'ensemble car ce dernier exige que le plus grand ensemble contienne tous les éléments du plus petit. Dans l'inclusion relâchée de langages d'arbres L_1 et L_2, il suffit que chacun des arbres de L_1 soient liés (en considérant l'inclusion relâché d'arbres) à au moins un arbre de L_2. En considérant l'application de substitution de services web, le fait d'utiliser l'inclusion relâchée pour rechercher un schéma XML B pour remplacer un autre schéma XML A dans une composition de services web est plus intéressant que l'utilisation de l'inclusion d'ensembles car (i) nous n'imposons pas que les arbres de B soient exactement les mêmes que ceux de A ; (ii) ce qui est important est que les arbres de B possèdent toutes les informations de ceux de A sans pour autant qu'ils aient la même structure ; (iii) les arbres de B peuvent contenir des informations qui n'existent pas dans les arbres de B. Il serait aussi intéressant de ne pas tenir compte de l'ordre dans la comparaison des arbres de A et B en utilisant l'inclusion d'arbres non ordonné mais ce problème est très difficile (NP-complet). Aussi notre choix de prendre en compte l'ordre se justifie par le fait que dans les documents XML l'ordre horizontal des nœuds est important. Une perspective serait quand même de tester l'inclusion relâchée des schémas XML sans tenir compte de l'ordre horizontal entre les nœuds des arbres. Ceci serait utile pour les XML schema du W3C qui à partir de l'élément *xsd:all* permet de définir des expressions régulières qui ne tiennent pas compte de l'ordre entre les non-terminaux.

3.5 Conclusion et perspectives

Dans cette thèse, étant donné une grammaire d'arbres d'arité non bornée G (ou grammaire de haies), nous présentons une méthode directe (c'est-à-dire manipulant les grammaires) pour calculer une grammaire G' qui génère l'ensemble de tous les arbres inclus relâchés dans les arbres générés par G. La méthode que nous avons proposée dans [8], était aussi une méthode directe mais sur des grammaires de haies qui ne sont pas récursives. Dans mon stage de fin d'année de Master [5], nous avons proposé une autre méthode plus complexe que celle décrite dans ce chapitre, pour résoudre le cas récursif. Cette méthode est basée sur les langages synchronisés de couples d'arbres [97] qui ont un pouvoir d'expression plus fort que les langages algébriques et qui sont définis par des programmes logiques sur des arbres d'arité bornée. Cette méthode calcule G' en transformant les langages d'arbres d'arité non bornée en langages d'arbres binaires en utilisant l'encodage « first-child next-sibling » [52, 90, 106]. Ensuite l'inclusion relâchée $\lhd$ est exprimée par un langage synchronisé de couples d'arbres, et en se servant de la projection régulière nous obtenons une grammaire régulière d'arbres G_1. Enfin la transformation de la grammaire binaire G_1 en grammaire de haies donne la grammaire G'. Cette méthode est plus complexe et produit des grammaires complexes.

Pour terminer ce chapitre nous allons considérer une perspective qui est nécessaire pour compléter l'application (décrite dans l'Exemple 3.1) où l'on souhaite substituer un service A dans une composition de services par un autre service B. A partir de la méthode décrite dans ce chapitre, nous pouvons décider si un service A peut être remplacé par un service

B. Le service *B* trouvé peut donc fournir les mêmes informations que celles contenues dans les documents de *A* mais la structure des documents de *B* n'est pas la même que celle des documents de *A*. Pour que le service *B* puisse remplacer le service *A* dans la composition de services web, il faudra pouvoir traduire les documents de *B* vers des documents valides par rapport aux autres services de la composition, et faire la traduction dans le sens contraire c'est-à-dire traduire les documents des autres services de la composition vers des documents du service *B*. Des solutions pour cette phase de traduction bi-directionnelle peuvent être trouvées dans le chapitre 6. Dans le chapitre 6 nous proposons un langage basé sur des opérations d'éditions pour exprimer un mapping entre deux schémas dont l'un est considéré comme la source et l'autre comme la cible. A partir du mapping et d'un document t valide par rapport au schéma source, nous proposons une méthode qui permet de transformer t en un arbre t' valide par rapport au schéma cible. Cette méthode s'inspire fortement du mapping pour avoir une transformation qui est conforme aux exigences de ce dernier. A partir de la méthode décrite dans ce chapitre nous avons les informations nécessaires pour exprimer un mapping entre le service *A* et *B*. Ensuite en se servant des opérations de composition et d'inversion définies sur les mappings (chapitre 6) qui sont très utiles pour l'évolution de schéma et évitent de recalculer les mappings entre les schémas, nous pouvons exprimer des mappings entre un autre schéma de la composition de services et le service *B*. Notre proposition pour la traduction de documents dans le contexte de la substitution de services web, sera expliquée plus en détails dans le chapitre 6 avec des exemples pour bien comprendre son fonctionnement.

Intégration de dépendances fonctionnelles XML

4

Sommaire

4.1 Introduction . . . 35
4.2 Motivation . . . 37
4.3 Préliminaires . . . 39
4.3.1 Langage de chemins linéaire . . . 40
4.3.2 Branches dans un arbre . . . 44
4.3.3 Informations incomplètes et arbre complet . . . 46
4.4 Dépendances Fonctionnelles pour XML . . . 48
4.4.1 Définition . . . 48
4.4.2 Satisfaction de XFD . . . 50
4.5 Raisonner sur les dépendances fonctionnelles XML . . . 52
4.5.1 Système d'axiomes . . . 53
4.5.2 Axiomes additionnels . . . 56
4.5.3 Dépendances triviales . . . 57
4.5.4 Fermeture d'un ensemble de chemins . . . 58
4.6 Fermeture d'un ensemble de chemins . . . 59
4.6.1 Algorithme pour la fermeture d'un ensemble de chemins . . . 59
4.6.2 Analyse de la complexité . . . 61
4.6.3 Algorithme pour la fermeture d'un ensemble de dépendances . . . 62
4.6.4 Notion de couverture . . . 62
4.7 Dépendances Fonctionnelles pour l'interopérabilité . . . 63
4.7.1 Méthode pour calculer l'ensemble maximal de XFD $\mathcal{F}$ . . . 64
4.7.2 Vers une version optimisée . . . 68
4.7.3 Méthode pour calculer une couverture de $\mathcal{F}$ . . . 68
4.7.4 Calcul de la couverture de $\mathcal{F}$: Preuve de correction . . . 75
4.7.5 Analyse de la complexité de l'Algorithme 5 . . . 80
4.8 Résultats Expérimentaux . . . 81
4.8.1 Jeu de données . . . 81
4.8.2 Résultats . . . 82
4.9 État de l'art . . . 84
4.10 Conclusion . . . 86

4.1 Introduction

Dans ce chapitre, nous nous intéressons au problème d'échanges de données XML dans un environnement multi-système (décrit dans le Chapitre 1 : Introduction) où un système

global doit recevoir et traiter des données provenant de différents systèmes locaux. Le système global est une évolution conservatrice des systèmes locaux dans le sens où il concerve la possibilité d'accepter les documents XML de chaque système local. Le système global possède son propre schéma et son propre ensemble de contraintes d'intégrité (généré à partir d'une fusion de ceux des systèmes locaux), et peut accepter et traiter des documents XML non locaux (éventuellement n'étant pas valides par rapport aux systèmes locaux).

Notre travail a pour but d'enrichir les propositions d'évolution de schéma en prenant en compte les contraintes d'intégrité qui permettent de spécifier la sémantique des données et qui jouent un rôle important dans le maintien de la consistance des données. Les propositions de fusion de schémas sont souvent basées sur de simple modèle de données. Les schémas peuvent être plus expressif que les DTD et XSD, et sont associés à des contraintes d'intégrités (comme dans [33]) ou exprimés par un modèle de données sémantiquement plus riche (comme dans [124]).

Un algorithme pour l'évolution conservatrice de schémas, qui étend de façon minimale une grammaire d'arbres régulier, est proposé dans [42]. Cette approche pour l'extension de schéma est purement syntaxique : seul les aspects structuraux de XML sont considérés et de nouvelles grammaires sont construites par des manipulations syntaxiques des règles de production. Nous souhaitons enrichir ce modèle en offrant la possibilité de calculer à partir d'ensembles locaux de XFD (Dépendances Fonctionnelles XML), une couverture du plus grand ensemble de dépendances fonctionnelles qui n'est pas violé par les documents des systèmes locaux. Ceci est une première étape vers une proposition d'évolution de schéma qui prend en compte les contraintes d'intégrité. Cette extension vise à enrichir l'évolution de schéma mais est conçue comme une procédure indépendante. Dans ce sens, elle peut être appliquée ou adaptée à d'autres approches d'évolution de schéma.

Certaines applications de nos travaux sont : dans le domaine des Bibliothèques Numériques, à cause de leur besoin d'évolution lorsque de nouvelles sources de données deviennent disponibles ou lorsque la fusion de deux bibliothèques peut être intéressante [53] ; dans la construction de services innovantes avec des données provenant de diverses organisations qui manipulent des informations similaires (mais pas identiques), et qui permettent d'envisager des adaptations possibles aux applications du « big data » [39]. Dans ces cas, il est important d'avoir un ensemble non contradictoire de dépendances fonctionnelles (cet ensemble peut être construit à partir des ensembles locaux de contraintes).

Nous supposons des systèmes locaux $S_1, \ldots, S_n$, comme illustré dans la Figure 4.1, qui travaillent respectivement sur des ensembles de documents XML $X_1, \ldots, X_n$, et qui interagissent avec un système global S. Le système global S est une évolution conservatrice des systèmes locaux $S_1, \ldots, S_n$. Chaque ensemble X_i est conforme aux contraintes de schéma $\mathcal{D}_i$ et aux contraintes d'intégrité $\mathcal{F}_i$ et suivent une ontologie O_i. Notre but est d'associer à S un schéma et des contraintes d'intégrité qui représentent une évolution conservatrice des contraintes locales. Plus précisément, étant donnés différents triplets $(\mathcal{D}_1, \mathcal{F}_1, O_1), \ldots, (\mathcal{D}_n, \mathcal{F}_n, O_n)$, nous sommes intéressés par la génération de $(\mathcal{D}, cover\mathcal{F}, \mathcal{A})$ où :

(i) $\mathcal{D}$ est une extension de schéma qui accepte tous les documents locaux ;

(ii) $cover\mathcal{F}$ est un ensemble de XFD équivalent à $\mathcal{F}$: le plus grand ensemble de XFD construit à partir de $\mathcal{F}_1, \ldots, \mathcal{F}_n$, qui est satisfait par tous les documents dans $X_1, \ldots, X_n$;

(iii) et $\mathcal{A}$ est un ensemble de correspondances entre les éléments de $\mathcal{D}_1, \ldots, \mathcal{D}_n$, et qui guide la construction de $\mathcal{D}$ et $\mathcal{F}$ en termes de sémantique.

Ce chapitre est consacré à la génération de $cover\mathcal{F}$ qui contient les XFD pour lesquelles aucune violation n'est possible pour les ensembles de documents $X_1, \ldots, X_n$. Il est important de noter que notre algorithme est basé sur un système d'axiomes, et ainsi $cover\mathcal{F}$ est obtenu à partir de $\mathcal{F}_1, \ldots, \mathcal{F}_n$ sans tenir compte des données.

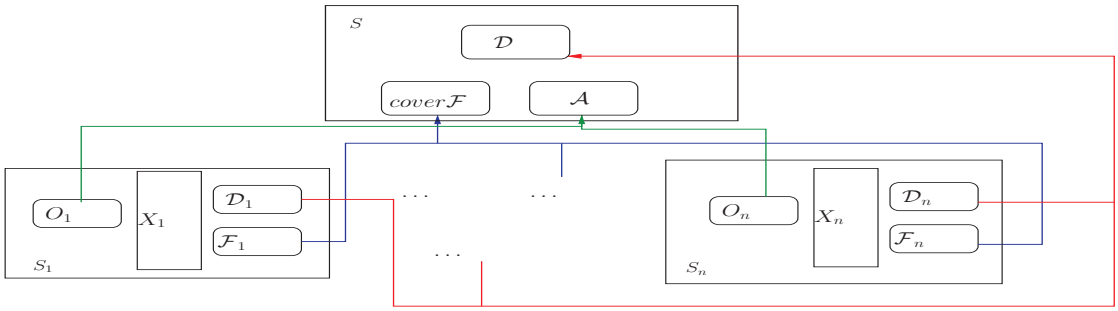


FIGURE 4.1 – *Le schéma et les contraintes d'intégrité du système S sont construits à partir des contraintes dans les systèmes locaux. Le schéma $\mathcal{D}$ accepte tous les documents de $\mathcal{D}_1, \ldots, \mathcal{D}_n$. Les contraintes d'intégrité dans $cover\mathcal{F}$ sont obtenues en filtrant les XFD non-contradictoires dans $\mathcal{F}_1, \ldots, \mathcal{F}_n$. L'ensemble de correspondances $\mathcal{A}$, construit à partir des ontologies $O_1, \ldots, O_n$, permet de retrouver les concepts équivalents dans les schémas. Chaque système local S_i traite les données de la collection de documents X_i.*

La contribution de ce chapitre est double : d'un côté nous introduisons un système d'axiomes avec des preuves de correction et complétude ; et de l'autre côté, nous présentons un algorithme efficace pour calculer, en nous aidant du système d'axiomes, l'ensemble $cover\mathcal{F}$. Nous prouvons que l'ensemble $cover\mathcal{F}$ a de bonnes propriétés et est bien équivalent à l'ensemble $\mathcal{F}$ et des expériences montrent l'efficacité de notre approche.

Le reste du chapitre est organisé comme suit : dans la Section 4.2 nous présentons un exemple de calcul de $cover\mathcal{F}$ pour motiver notre approche. Les notions préliminaires pour la compréhension de nos méthodes sont introduites dans la Section 4.3. Nous définissons les dépendances fonctionnelles pour XML avec des exemples dans la Section 4.4. Ces définitions de dépendances fonctionnelles proviennent de [33, 66, 82] où l'on peut trouver l'algorithme de satisfaction de telles dépendances fonctionnelles sur un document XML. Dans la Section 4.5, nous présentons le système d'axiomes pour ces dépendances fonctionnelles, nécessaire pour pouvoir raisonner sur les dépendances fonctionnelles. Un algorithme pour calculer la fermeture d'un ensemble de chemins par rapport à un ensemble de XFD, basé sur le système d'axiomes, est présenté en Section 4.6 avec la définition de la notion de couverture. Dans la Section 4.7, nous pouvons trouver les méthodes pour calculer $\mathcal{F}$ et sa couverture $cover\mathcal{F}$, les preuves de correction de ces méthodes et l'analyse de leur complexité. Nos résultats expérimentaux sont présentés en Section 4.8, les travaux similaires aux nôtres sont présentés dans la Section 4.9 et pour terminer nous concluons le chapitre en Section 4.10.

4.2 Motivation

Nous supposons des universités de la même région de France qui veulent implanter un système central pour obtenir et traiter les informations concernant leurs cours et leurs étudiants, indépendamment des systèmes locaux déjà existants. Leur but est d'obtenir un

système central qui assure un nombre maximal de contraintes d'intégrité locaux et non-contradictoires.

Chaque université a établit localement, ses propres contraintes. Par exemple, considérons les arbres XML de la Figure 4.2 comme étant des documents de deux différentes universités. Chaque document est valide par rapport aux dépendances fonctionnelles présentées dans le Tableau 4.1, c'est-à-dire les documents dans X_1 sont valides par rapport à $\mathcal{F}_1$, et ceux dans X_2 sont valides par rapport à $\mathcal{F}_2$.

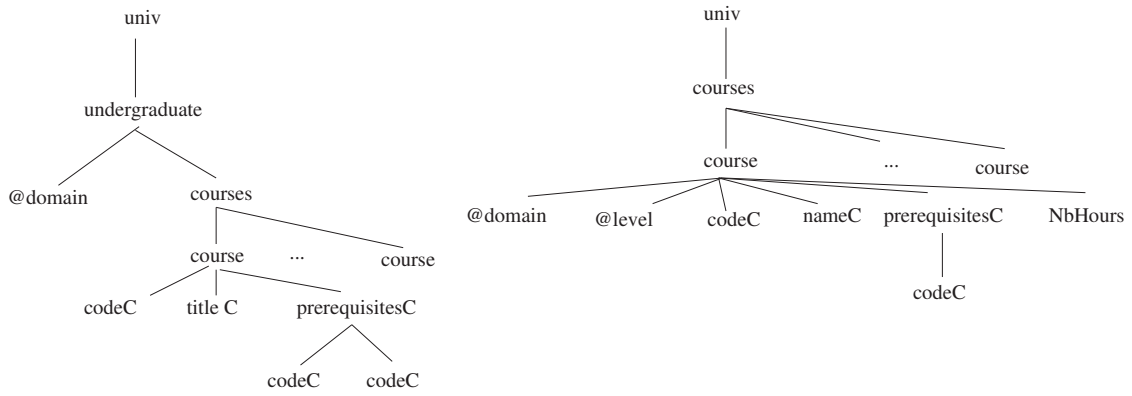


FIGURE 4.2 – *Deux documents XML de différentes sources locaux.*

Dans le domaine XML, une dépendance fonctionnelle (XFD) est définie par des chemins sur un arbre. Chaque chemin sélectionne un nœud dans l'arbre. Les valeurs ou positions des nœuds sélectionnés sont regroupées pour construire des n-uplets qui seront utilisés pour vérifier si un document XML donné satisfait une dépendance. Par exemple, considérons la XFD *f : (univ, (undergraduate/courses/course/codeC → undergraduate/courses/course/titleC))* sur le premier document de la Figure 4.2. Elle spécifie que le contexte est *univ*, c'est-à-dire que la contrainte doit être vérifiée sur les données en dessous du nœud étiquetté par *univ*. Dans ce contexte, *f* implique la construction des n-uplets composés par les valeurs obtenues en suivant les chemins : *univ/undergraduate/courses/course/codeC, univ/undergraduate/courses/course/titleC*. Comme dans le modèle relationnel, un document est valide par rapport à *f* si n'importe quels deux n-uplets égaux sur les valeurs obtenues à partir de *univ/undergraduate/courses/course/codeC*, sont aussi égaux sur les valeurs obtenues à partir de *univ/undergraduate/courses/course/titleC*. Par conséquent dans une université, le code d'un cours détermine son nom.

Similairement, la XFD *f1 : (univ, (undergraduate/courses/course/codeC → undergraduate/courses/course/prerequisitesC))* implique des n-uplets où le chemin *univ/undergraduate/courses/course/prerequisitesC* conduit aux sous-arbres ayant comme racine *prerequisitesC* (c'est-à-dire les sous-arbres contenant les informations sur "prerequisites"). Cette contrainte indique que les cours ayant le même code doivent aussi avoir les mêmes prérequis. Un document est valide par rapport à *f1* si n'importe quels deux n-uplets égaux sur les valeurs obtenues à partir de *univ/undergraduate/courses/course/codeC*, sont aussi égaux sur les valeurs obtenues à partir de *univ/undergraduate/courses/course/prerequisitesC*, c'est-à-dire les sous-arbres obtenus sont isomorphes.

Considérons les trois premières XFD dans le Tableau 4.1, concernant la source 1. Elles indiquent que dans une université, le code d'un cours détermine son nom, son domaine et ses prérequis. En d'autres mots, le cours est identifié par son code.

A partir des différentes ontologies nous assumons que le Tableau 4.2 est disponible, faisant la correspondance entre les chemins des différents ensembles locaux de documents. Ainsi il est possible de conclure que, par exemple, les XFD *f : (univ, (undergraduate/cour-*

ses/course/codeC → undergraduate/courses/course/titleC)) et *(univ, (courses/course/codeC → courses/course/nameC))* sont équivalentes, c'est-à-dire qu'elles représentent la même contrainte puisqu'elles concernent les mêmes concepts : dans une université, le code détermine son nom.

$\mathcal{F}$	XFD
1	$(univ, (undergraduate/courses/course/codeC \rightarrow undergraduate/courses/course/titleC))$
1	$(univ, (undergraduate/courses/course/codeC \rightarrow undergraduate/courses/course/prerequisitesC))$
1	$(univ, (undergraduate/courses/course/codeC \rightarrow undergraduate/@domain))$
2	$(univ, (courses/course/codeC \rightarrow courses/course/nameC))$
2	$(univ, (courses/course/codeC \rightarrow courses/course/@domain))$
2	$(univ, (courses/course/codeC \rightarrow courses/course/@level))$
2	$(univ, (\{courses/course/nameC, courses/course/@level\} \rightarrow courses/course/NbHours))$

Tableau 4.1 – *XFD dans $\mathcal{F}_1$ et $\mathcal{F}_2$.*

Chemins dans $\mathcal{D}_1$	Chemins dans $\mathcal{D}_2$
univ/undergraduate/courses/course/codeC	*univ/courses/course/codeC*
univ/undergraduate/courses/course/titleC	*univ/courses/course/nameC*
univ/undergraduate/courses/course/prerequisitesC	*univ/courses/course/prerequisitesC*
univ/undergraduate/courses/course/prerequisitesC/codeC	*univ/courses/course/prerequisitesC/codeC*
univ/undergraduate/@domain	*univ/courses/course/@domain*

Tableau 4.2 – *Extrait de la table de correspondance.*

Supposons que nous avons ces deux sources locaux, et nous souhaitons avoir à partir de $\mathcal{F}_1$ et $\mathcal{F}_2$, le plus grand ensemble de XFD $\mathcal{F}$ qui ne contredit aucun document dans X_1 et X_2. Pour atteindre ce but, nous considérons toutes les XFD dérivables à partir de $\mathcal{F}_1$ et $\mathcal{F}_2$, qui résultent en un très grand ensemble de XFD. En effet l'ensemble $\mathcal{F}$ est très grand pour qu'on puisse l'exploiter. Une meilleure solution consiste à calculer *cover$\mathcal{F}$, une couverture de $\mathcal{F}$* (qui est un ensemble de XFD plus petit et qui est équivalent à $\mathcal{F}$), sans pour autant calculer toutes les XFD que l'on peut dériver à partir de $\mathcal{F}_1$ et $\mathcal{F}_2$. Nous proposons un algorithme qui génère *cover$\mathcal{F}$*.

Dans notre exemple, l'ensemble *cover$\mathcal{F}$* résultat contient les XFD du Tableau 4.3. Dans le Tableau 4.3, la première et la quatrième XFD sont équivalentes. Elles ont été ajoutées à *cover$\mathcal{F}$* puisque tous les documents dans X_1 et X_2 sont valides par rapport à ces deux dépendances. Le même raisonnement est appliqué à la deuxième et à la troisième dépendance dans le Tableau 4.3. Les deux dernières dépendances concernent des concepts qui apparaissent seulement dans X_2, et par conséquent ne peuvent pas être violées par les documents dans X_1. Enfin remarquons que la dépendance $(univ, (undergraduate/courses/course/codeC \rightarrow undergraduate/courses/course/prerequisitesC))$ dans $\mathcal{F}_1$, qui nous dit que les cours ayant le même code doivent avoir les mêmes prérequis, n'est pas dans *cover$\mathcal{F}$*. La raison est que, selon la table de correspondance, cette dépendance est équivalente à *(univ, (courses/course/codeC → courses/course/prerequisitesC))* dans $\mathcal{F}_2$. Cependant comme $\mathcal{F}_2$ ne contient pas cette dépendance, les documents dans X_2 peuvent la violer (puisque les concepts concernés existent dans X_2).

4.3 Préliminaires

Les dépendances fonctionnelles (XFD) que nous allons définir, ont été introduites dans [27, 33]. Dans cette section nous allons rappeler les notions nécessaires pour la défini-

$(univ, (undergraduate/courses/course/codeC \rightarrow undergraduate/courses/course/titleC))$
$(univ, (undergraduate/courses/course/codeC \rightarrow undergraduate/@domain))$
$(univ, (courses/course/codeC \rightarrow courses/course/@domain))$
$(univ, (courses/course/codeC \rightarrow courses/course/nameC))$
$(univ, (courses/course/codeC \rightarrow courses/course/@level))$
$(univ, (\{courses/course/nameC, courses/course/@level\} \rightarrow courses/course/NbHours))$

Tableau 4.3 – *XFD dans l'ensemble résultat cover$\mathcal{F}$.*

tion de la sémantique de nos XFD (dépendances fonctionnelles pour XML).

4.3.1 Langage de chemins linéaire

Plusieurs langages de chemins sont possibles pour désigner des parties d'un document XML, allant de la séquence d'étiquettes (chemin simple) [36] à des langages plus expressifs comme XPath [48]. Dans ce manuscrit nous utilisons le langage de chemin PL, qui est une sous-classe de XPath, pour adresser une partie d'un document XML.

Définition 4.1 (Langage de chemins)
PL est le langage de chemins (mots sur l'alphabet $\Sigma \cup \{/, //, []\}$) définit récursivement par :

$$\rho ::= [] \mid \ l \mid \rho/\rho \mid \rho//l$$

où $[]$ est le chemin vide, l est un label dans Σ, le symbole "/" est l'opération concaténation, "//" représente n'importe quelle séquence (éventuellement vide) de labels. Notons que $l/[] = []/l = l$ et $[]//l = //l$.

□

On appelle *chemins simples* des chemins qui sont de simples séquences d'étiquettes (sans //). Nous désignons par $\mathbb{P}$ l'ensemble de *tous les chemins simples qui se trouvent dans n'importe quel arbre XML t respectant un schéma $\mathcal{D}$ donné*. L'ensemble $\mathbb{P}$ est généré à partir d'un schéma $\mathcal{D}$ donné et est un ensemble *fini* de chemins simples, car on suppose que $\mathcal{D}$ est limité en profondeur. En d'autres termes le langage de mots $L(A_{\mathcal{D}})$, obtenu à partir d'un automate à états finis $A_{\mathcal{D}}$, est fini. L'automate $A_{\mathcal{D}}$ est construit à partir du schéma $\mathcal{D}$. Ce type de schéma est exprimé, par exemple, à partir d'une DTD non récursive. Dans ce sens, nous sommes plus général que [27, 33] où $\mathbb{P}$ est l'ensemble contenant tous les chemins possible dans *un* arbre donné.

Il est important de noter qu'un chemin avec // peut être associé à un ensemble de chemins simples dans $\mathbb{P}$. Pour un chemin P dans PL, son ensemble de chemins simples est le langage $L(A_P)$ où A_P est l'automate de mots à états finis obtenu sur le principe des deux définitions ci-après. L'automate A_P reconnaît donc l'ensemble des chemins simples représentés par le chemin P dans le contexte de $\mathbb{P}$.

L'automate A_P est construit en deux étapes. Tout d'abord on construit un automate B_P à partir du chemin P c'est-à-dire reconnaît tous les chemins simples définis sur l'alphabet Σ et qui respectent l'expression du chemin P. Ensuite l'automate A_P est obtenu en faisant une intersection entre l'automate B_P et l'automate $A_{\mathcal{D}}$ qui reconnaît tous les chemins simples du schéma $\mathcal{D}$. En général, l'automate B_P reconnaît un langage infini, et une fois que l'intersection de B_P est faite avec $A_{\mathcal{D}}$ on obtient un automate qui reconnaît un langage fini du fait que $L(A_{\mathcal{D}})$ est fini. L'Algorithme 14 (Annexe A.2) permet de construire l'automate d'états

finis (FSA) B_P à partir d'un chemin donné P dans PL sur l'alphabet Σ. Le FSA B_P est un 5-uplet $(Q, \Sigma, s_0, \delta, s_f)$ où

- Q est l'ensemble des états,
- Σ est l'alphabet,
- s_0 est l'état initial,
- δ est la fonction transition tel que $\delta : Q \times \Sigma \to Q$
- et s_f est l'état final.

L'automate B_P est construit en analysant le chemin P, à partir de l'ensemble des états $Q = \{s_0\}$. L'état s_0 est l'état initial et l'état courant s est initialisé à s_0. En analysant le chemin P, si on est sur un label $a \in \Sigma$ alors :

1. lorsque le label a est précédé du symbole '/' (*i.e.* $/a$), on rajoute la transition $\delta(s, a) = s_i$ à l'automate B_P où s_i est un nouvel état. L'état courant s devient l'état s_i
2. lorsque le label a est précédé du symbole '//' (*i.e.* $//a$), on boucle sur l'état courant en rajoutant les transitions $\forall\, b \in \Sigma,\ \delta(s, b) = s$ à l'automate B_P. L'état courant s passe ensuite à un nouvel état s_i après avoir ajouté la transition $\delta(s, a) = s_i$ à l'automate B_P.

Aucune action n'est faite sur l'automate B_P lorsqu'on rencontre le symbole du chemin vide '[]'. A la fin de l'analyse du chemin P, l'état final s_f est égal à l'état courant s.

L'automate B_P a toujours le même format à cause du symbole // qui fait ajouter des transitions avec toutes les étiquettes de l'alphabet Σ à un état de B_P. Nous pouvons facilement construire l'automate à états finis déterministe de B_P, avec juste quelques transitions en plus.

Définition 4.2 (FSA A_P associé à P dans le contexte de $\mathbb{P}$)
Soit un chemin P dans PL et un ensemble de chemins simples $\mathbb{P}$. L'automate A_P associé à P est défini tel que $L(A_P) = L(B_P) \cap \mathbb{P}$ où B_P est l'automate construit par l'Algorithme 14.
□

Exemple 4.1 Soit la DTD $\mathcal{D}$ concernant une librairie de livres en anglais, acceptant des documents comme celui de la Figure 2.2. La DTD $\mathcal{D}$ impose à une section d'avoir au plus deux sous-niveaux de sous-sections. L'automate $A_{\mathcal{D}}$ construit à partir de la DTD $\mathcal{D}$ est illustré par la Figure 4.3(a) et on a $L(A_{\mathcal{D}}) = \mathbb{P}$. Notons que l'automate $A_{\mathcal{D}}$ reconnaît aussi les préfixes des chemins définis dans la DTD $\mathcal{D}$. Soit $P = library//section/title$. L'automate B_P construit à partir de P et sa déterminisation sont illustrés par la Figure 4.3(b). En effet $L(B_P) \cap \mathbb{P}$ contient l'ensemble les chemins simples associés à P en considérant $\mathbb{P}$. Plus précisément, $L(B_P) \cap \mathbb{P}$ contient les chemins simples dans $L(B_P)$ qu'on retrouve aussi dans l'ensemble $\mathbb{P}$. L'ensemble $L(A_P) = \{$ *library/book/chapter/section/title, library/book/chapter/section/section/title, library/book/chapter/section/section/section/title*$\}$ et on a $A_P = A_{\mathcal{D}} \cap B_P$.
⊠

Un chemin donné P est **valide** dans $\mathcal{D}$ si : (i) il est conforme à la syntaxe PL, (ii) $L(A_P) \neq \emptyset$, (iii) pour chaque label l dans P, si $l \in \Sigma_{data} \cup \Sigma_{att}$ alors le label l est le dernier symbole dans P.

Nous allons maintenant définir des fonctions sur les chemins PL qui sont aussi valables sur les chemins simples.

Définition 4.3 (Fonctions *Parent* et *Last* sur les chemins)
Soit $P = l_1/\ldots/l_n$ où $n \geq 1$, un chemin PL.

- Le dernier label de P est obtenu par la fonction *Last* : $Last(P) = l_n$. Le label l_n est aussi appelé le dernier pas de P.

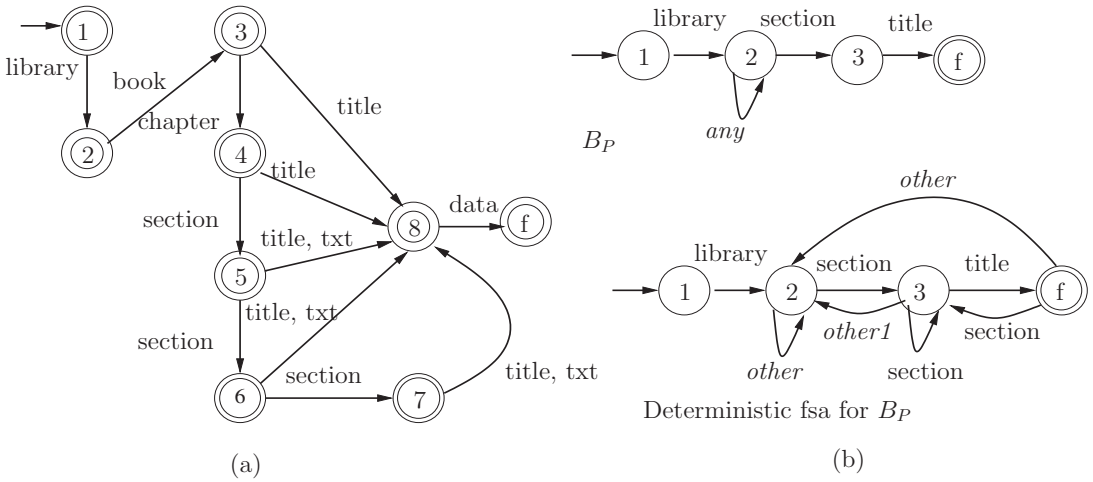


FIGURE 4.3 – *(a) Automate $A_{\mathcal{D}}$ construit à partir de la DTD $\mathcal{D}$ ($L(A_{\mathcal{D}}) = \mathbb{P}$). (b) Automate B_P pour le chemin P = library//section/title et sa déterminisation (où any est utilisé pour Σ, other est utilisé pour $\Sigma \setminus \{section\}$ et other1 est utilisé pour $\Sigma \setminus \{section, title\}$).*

– Le parent de P représente le(s) chemin(s) obtenu(s) en enlevant le dernier pas à P. La fonction $Parent(P) = \{l_1/\ldots/l_{n-1} \mid l_1/\ldots/l_{n-1}/l_n \in L(A_P)$ pour $n > 1\}$.

□

Définition 4.4 (Préfixe d'un chemin)
Soit P un chemin valide dans $\mathcal{D}$ et A_P son automate associé (Définition 4.2). Soit $PREFIX(A_P)$ l'automate à états finis qui reconnaît tous les préfixes de $L(A_P)$. Le chemin Q est un préfixe de P, noté $Q \preceq_{PL} P$, si nous avons $L(A_Q) \subseteq L(PREFIX(A_P))$. Entre deux chemins simples, la notion de préfixe est notée $\preceq$ (sans le PL).

□

Exemple 4.2 Considérons le document XML de la Figure 2.2 et l'ensemble $\mathbb{P} = L(A_{\mathcal{D}})$ de la Figure 4.3(a). Soit un chemin $P = library//section/title$. Nous avons :

– $Last(P) = title$
– $Parent(P) = library//section$
– $library/book \preceq library/book/chapter/section/txt$
– $L(PREFIX(A_P)) = \{library, library/book, librairy/book/chapter, library/book/chapter/section, library/book/chapter/section/section, library/book/chapter/section/section/section, library/book/chapter/section/section/section/title\}$
– $library//section \preceq_{PL} P$, $library/book/chapter \preceq_{PL} P$, $library \preceq_{PL} P$

⊠

Définition 4.5 (Le plus long préfixe commun (intersection de chemin))
Soit P' et Q' deux chemins simples, P et Q deux chemins PL valides, et A_P, A_Q leurs automates associés (Définition 4.2).

– Le plus long préfixe commun de deux chemins simples P' et Q' (noté par $P' \cap Q'$) est le chemin R tel que $R \preceq P'$ et $R \preceq Q'$ et il n'existe pas un autre chemin R' tel que $R \prec R'$, $R' \preceq P'$ et $R' \preceq Q'$.
– Le plus long préfixe commun de P et Q, noté $P \cap Q$, décrit l'ensemble des chemins simples $\{P' \cap Q' \mid P' \in L(A_P) \wedge Q' \in L(A_Q)\}$.

□

Exemple 4.3 Considérons le document XML de la Figure 2.2 et l'ensemble $\mathbb{P} = L(A_{\mathcal{D}})$ de la Figure 4.3(a). Nous avons :

- $library/book/chapter/section/section/title \cap library/book/chapter/section/txt = library/book/chapter/section$
- $library//section/title \cap library//section/txt = library//section$

⊠

L'Algorithme 15 (Annexe A.3) permet de calculer l'automate $A_{P \cap R}$ associé au plus long préfixe commun $P \cap R$ des chemins P et R.

Instances de chemin. Nous pouvons représenter avec le langage PL un chemin dans un document XML. Maintenant il est important de savoir exactement quelle partie du document XML nous intéresse pour un chemin donné. Les parties de notre document XML seront donc les instanciations des chemins.

Définition 4.6 (Instance d'un chemin P dans un arbre t)
Soit P un chemin PL, A_P son automate associé et $L(A_P)$ le langage reconnu par A_P. Soit $I = p_1/\ldots/p_n$ une séquence de positions tel que chaque p_i est le fils de p_{i-1} dans t. Nous considérons I comme une instance de P dans l'arbre t si et seulement si la séquence $t(p_1)/\ldots/t(p_n) \in L(A_P)$. L'ensemble de toutes les instances de P dans un arbre t est défini par la fonction $Instances(P, t)$. □

Nous allons définir sur les instances de chemins les mêmes fonctions qui ont été défini sur les chemins.

Définition 4.7 (Fonctions sur les instances de chemins)
Soit $I = p_1/\ldots/p_n$ et $J = u_1/\ldots/u_m$ deux instances de chemin.

- $Last(I) = p_n$ est la dernière position dans I.
- Pour $n > 1$, $Parent(I)$ est la séquence $p_1/\ldots/p_{n-1}$.
- L'instance I est un préfixe de l'instance J, noté par $I \preceq J$, si $n \leq m$ et $p_1 = u_1$, …, $p_n = u_n$. Nous avons $I = J$ Lorsque $I \preceq J$ and $J \preceq I$.
- Le plus long préfixe commun de I et J, noté par $I \cap J$, est l'instance K tel que $K \preceq I$ et $K \preceq J$ et il n'existe pas une autre instance K' tel que $K \prec K'$, $K' \preceq I$ et $K' \preceq J$.

□

Le plus long préfixe commun de deux instances, détermine aussi le plus petit ancêtre commun entre deux nœuds et son chemin est donné par la Définition 4.5.

Exemple 4.4 Dans le document de la Figure 2.2, $I_1 = \epsilon/0/0.1/0.1.0$, $I_2 = \epsilon/0/0.2/0.2.0$ et $I_3 = \epsilon/1/1.1/1.1.0$ sont des instances du chemin $P =$*library/book/chapter/title*. Nous avons $Last(I_1) = 0.1.0$ et $Parent(I_1) = \epsilon/0/0.1$. Le plus long préfixe commun de I_1 et I_2 est $\epsilon/0$, et le plus long préfixe commun de I_1 et I_3 est ϵ. ⊠

Comparaison de nœuds. Comme beaucoup d'auteurs, nous utilisons deux types d'égalité pour comparer des nœuds dans un arbre XML qui sont : l'*égalité par valeur* et l'*égalité par nœud* (ou *identité de nœud*). Deux nœuds sont égaux par nœud lorsqu'ils sont à la même position. Deux nœuds sont égaux par valeur lorsqu'ils sont des racines de sous-arbres isomorphes. Plus précisément nous avons :

Définition 4.8 (Égalité par valeur)
Soit un arbre XML t. Deux nœuds de position p et q sont égaux par valeur, égalité notée par $p =_V q$, si les conditions suivantes sont respectées :

(i) $t(p) = t(q)$ (ils ont la même étiquette) ;

(ii) $type(t, p) = type(t, q)$ (ils ont le même type) ;

(iii) si $type(t, p) = data$ ou $type(t, p) = attribut$ alors $value(t, p) = value(t, q)$ (ils ont la même valeur sur leur donnée) ;

(iv) si $type(t, p) = élément$ alors il existe une fonction bijective entre les fils de p et ceux de q qui associe chaque position $p.i$ à une position $q.j$ telle que $p.i =_V q.j$.

□

Par exemple dans la Figure 2.2, les nœuds en positions 1.1.1 et 0.1.1 sont égaux par valeur, mais les nœuds 1.1 et 0.1 ne le sont pas (les sous-arbres des éléments *title* sont associées à différente données). Notons que la définition de l'égalité par valeur ne prend pas en compte l'ordre entre les fils d'un nœud dans l'arbre t.

Pour exprimer ces deux égalités on utilise le symbole E, qui peut être remplacé par V pour l'égalité par valeur ou N pour l'égalité par nœud.

Définition 4.9 (Concordance de nœud)
Soient deux nœuds n_1 et n_2 de même étiquette. Les nœuds n_1 et n_2 concordent, ce qui se note $n_1 =_E n_2$, si et seulement si, ou bien $n_1 =_V n_2$ (égalité en valeur, dans ce cas $E = V$), ou bien $n_1 =_N n_2$ (identité de nœud, dans ce cas $E = N$). □

Remarquons que $n_1 =_N n_2$ implique $n_1 =_V n_2$. La réciproque est fausse c'est à dire que deux nœuds égaux par valeur, ne sont pas forcément à la même position.

4.3.2 Branches dans un arbre

Dans la section précédente, nous avons vu ce que c'est qu'un chemin et des fonctions sur les chemins. Nous allons maintenant introduire la notion de *branche* aussi appelé motif dans la littérature [32, 120].

Une branche est vue comme un ensemble non vide de chemins simples qui ont un préfixe commun (au moins la racine). La projection de l'arbre sur une branche (ou l'instance de la branche dans l'arbre) détermine les positions de l'arbre correspondant aux chemins. Ainsi, la projection est un ensemble clos par préfixe d'instances de chemins qui respectent certaines conditions nécessaires. Avec cette définition intuitive nous pouvons nous dire qu'un chemin PL est aussi une branche mais, nous allons voir tout à l'heure en formalisant les choses, qu'un chemin PL n'est pas du tout une branche.

Définition 4.10 (Branche dans un arbre)
Une *branche* ou *motif* est un ensemble fini de chemins (simples) d'un arbre t, *clos par préfixe*.
□

Pour pouvoir instancier une branche dans un arbre, nous avons besoin de regrouper les instances des chemins de la branche qui sont les plus proches, c'est à dire qui partagent le même préfixe. Par exemple lorsque nous considérons la branche M comprenant les chemins $library/book/title$, $library/book/chapter/title$ et leur préfixes, une instance de M dans l'arbre de la Figure 2.2 serait composée de $\epsilon/0/0.0$, $\epsilon/0/0.1/0.1.0$ et leur préfixes. L'ensemble composé de $\epsilon/0/0.0$, $\epsilon/1/1.1/1.1.0$ et leur préfixes, n'est pas une instance de M car

les instances $\epsilon/0/0.0$ et $\epsilon/1/1.1/1.1.0$ ne sont pas proches. Dans ce dernier cas, on essaie d'associer le chapitre *weather* au livre *English Lang Book* alors que le livre *English Lang Book* ne comporte pas de chapitre ayant le titre *weather*. La notion de *plus proche* nous est donnée par la définition suivante :

Définition 4.11 (Test de Proximité entre deux instances)
Soit P et Q deux chemins valides sur l'arbre t. La fonction booléenne $isInst_lcp(P, I, Q, J)$ renvoie *vraie* lorsque toutes les conditions suivantes sont vérifiées :

1. $I \in Instances(P, t)$;
2. $J \in Instances(Q, t)$ et
3. $I \cap J$ est une instance du chemin $P \cap Q$;

Dans le cas contraire, la fonction retourne *faux*.

□

L'instance d'une branche est donc définie comme suit :

Définition 4.12 (Projection d'un arbre $\mathcal{T}$ sur une branche M)
Soit M une branche sur l'arbre $\mathcal{T}$. Soit $Long_M$ l'ensemble des chemins dans M qui ne sont pas préfixes d'autres chemins dans M. Soit $SetPathInst$ l'ensemble des instances de chemins (simples) qui vérifie :

1. $\forall\, P \in Long_M$, il existe une et une seule instance $inst \in Instances(P, t)$ dans l'ensemble $SetPathInst$.
2. $\forall\, inst \in SetPathInst$, il existe un chemin $P \in Long_M$ tel que $inst \in Instances(P, t)$.
3. pour toutes instances $inst$ et $inst'$ dans $SetPathInst$, si $inst \in Instances(P, t)$ et $inst' \in Instances(Q, t)$ alors $isInst_lcp(P, inst, Q, inst')$ est vraie.

Une projection de $\mathcal{T}$ sur M, notée par $\Pi_M(\mathcal{T})$, est un triplet $(t^i, type^i, value^i)$ où $type^i(t^i, p) = type(t, p)$, $value^i(t^i, p) = value(t, p)$ et t^i est une fonction $\Delta \to \Sigma$ telle que :

- $\Delta = \bigcup_{inst \,\in\, SetPathInst}\{p \mid p \text{ est une position dans } inst\}$
- $t^i(p) = t(p)$, $\forall p \in \Delta$

Dans la suite, pour faire simple, nous allons confondre $\Pi_M(\mathcal{T})$ et la fonction t^i. □

Étant donné les projections de deux *branches*, $\Pi_{M_1}(\mathcal{T})$ et $\Pi_{M_2}(\mathcal{T})$, l'union $\Pi_{M_1}(\mathcal{T}) \cup \Pi_{M_2}(\mathcal{T})$ est obtenue naturellement en considérant toutes les instances des chemins qui se trouvent dans les deux projections.

Exemple 4.5 Considérons le document XML $\mathcal{T}$ de la Figure 2.2. Soit M une *branche* définie à partir de l'ensemble

$$\{\textit{library/book/title, library/book/chapter/title, library/book/chapter/section/title }\},$$

i.e., M contient ces chemins et tous leur préfixes.
Un exemple de projection pour $\mathcal{T}$ sur M est égale à l'ensemble

$$\{(\epsilon, library), (0, book), (0.0, title), (0.1, chapter), (0.1.0, title), (0.1.1, section), (0.1.1.0, title)\}.$$

Cependant, l'ensemble

$$\{(\epsilon, library), (0, book), (0.0, title), (0.1, chapter), (0.1.0, title), (0.2.1, section), (0.2.1.0, title)\}$$

n'est pas une projection de $\mathcal{T}$ sur M. En effet, en prenant la Définition 4.12, si nous considérons $P = library/book/chapter/title$ et son instance $inst = \epsilon/0/0.1/0.1.0$ avec $Q = library/book/chapter/section/title$ et son instance $inst' = \epsilon/0/0.2/0.2.1/0.2.1.0$, nous avons $isInst_lcp(P, inst, Q, inst') = faux$. Nous pouvons remarquer que le plus long préfixe commun $P \cap Q$ est $library/book/chapter$ alors que celui des instances $inst \cap inst'$ est ϵ/o. Nous pouvons trouver dans $\mathcal{T}$ deux autres projections sur M :

1. $\{(\epsilon, library), (0, book), (0.0, title), (0.2, chapter), (0.2.0, title), (0.2.1, section), (0.2.1.0, title)\}$ et
2. $\{(\epsilon, library), (1, book), (1.0, title), (1.1, chapter), (1.1.0, title), (1.1.1, section), (1.1.1.1, title)\}$

⊠

D'après la Définition 4.12, nous pouvons remarquer que la projection de $\mathcal{T}$ sur une *branche M* contient exactement une instance pour chaque chemin dans M. Dans la suite, si besoin, *nous notons par* $\Pi_M(\mathcal{T})[P]$ *l'unique instance du chemin P dans* $\Pi_M(\mathcal{T})$. En effet, en écrivant $\Pi_M(\mathcal{T})[P]$ nous limitons la projection de T sur M à l'instance (dans la projection) d'un seul chemin P.

Le lemme suivant nous garantie que dans une projection, toutes les instances des chemins sont proches entre elles.

Lemme 4.1 *Soit* $\Pi_M(\mathcal{T})$ *une projection du document* $\mathcal{T}$ *sur une branche M. Pour chaque chemin P et Q dans M si* $I = \Pi_M(\mathcal{T})[P]$ *et* $J = \Pi_M(\mathcal{T})[Q]$ *alors on a* $isInst_lcp(P, I, Q, J) = vraie$.

Démonstration. Soit P et Q deux chemins dans la *branche M*. Soit $\Pi_M(\mathcal{T})$ la projection de M sur l'arbre $\mathcal{T}$ et soit I et J les instances de P et Q dans $\Pi_M(\mathcal{T})$. Nous avons les deux cas suivants :

1. Cas 1 : $P \prec Q$. Dans ce cas $P \cap Q = P$. Puisque I et J sont les unique instances des chemins P et Q dans $\Pi_M(\mathcal{T})$ alors $I \cap J = I$. Ainsi on obtient $isInst_lcp(P, I, Q, J) = vraie$.
2. Cas 2 : $P \not\prec Q$ et $Q \not\prec P$. Nous pouvons distinguer trois situations différentes :
   (a) P et Q sont des chemins dans $Long_M$. D'après la Définition 4.12 nous savons que $isInst_lcp(P, I, Q, J) = vraie$.
   (b) $P \in Long_M$ et $Q \notin Long_M$. Soit $Q' \in Long_M$ tel que $Q \prec Q'$. Soit $J' = \Pi_M(\mathcal{T})[Q']$. D'après la Définition 4.12, nous savons que $isInst_lcp(P, I, Q', J') = vraie$ et par conséquent $I \cap J'$ est l'instance de $P \cap Q'$ dans la projection $\Pi_M(\mathcal{T})$. Nous savons aussi que $I \cap J' \prec J'$. D'après l'item 1 nous avons $isInst_lcp(Q, J, Q', J') = vraie$ et aussi $J \prec J'$. Comme $J \not\prec I$ (puisque $Q \not\prec P$) nous avons $I \cap J' \prec J$. Puisque $J \prec J'$, nous concluons que $I \cap J = I \cap J'$ et donc $isInst_lcp(P, I, Q, J) = vraie$.
   (c) $P \notin Long_M$ et $Q \notin Long_M$ est similaire à la situation (2b).

◁

4.3.3 Informations incomplètes et arbre complet

Un document XML $\mathcal{T}$ représente couramment des informations incomplètes. C'est le cas pour le document $\mathcal{T}$ de la Figure 4.4, dans lequel un livre (position 1) ne possède pas de fils *chapter*. Par exemple le chemin $library/book/chapter$ possède une instance incomplète sur $\mathcal{T}$, à savoir $\epsilon/1/1.1$.

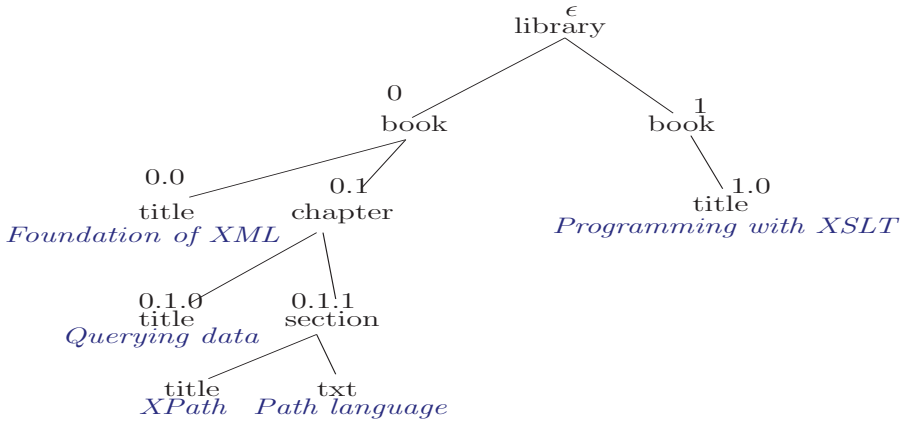


FIGURE 4.4 – *Document XML avec des informations manquantes*

Le traitement correct des informations incomplètes est fondamental, aussi bien pour le calcul des requêtes que pour la vérification des contraintes d'intégrité. Différentes propositions, établies dans le cadre du modèle relationnel [17, 49, 72, 73, 81, 83], peuvent être transposées aux documents XML [2, 15, 67, 120].

A l'opposée un *arbre complet*, terme aussi utilisée dans [118], est un arbre XML qui ne possède pas d'informations manquantes. L'*arbre complet* est une extension de la notion de *relation complète* (dans le modèle relationnel). La définition de l'*arbre complet* nous est donnée par la définition suivante :

Définition 4.13 (Arbre complet)
Soit $\mathbb{P}$ un ensemble de chemins simples. Un arbre $\mathcal{T}$ est complet par rapport à $\mathbb{P}$ si lorsqu'il existe des chemins P et P' dans $\mathbb{P}$ tels que $P' \prec P$ et il existe une instance de I' pour P', alors il existe une instance I pour P telle que le nœud $Last(I')$ est un ancêtre du nœud $Last(I)$.
□

Ainsi un arbre qui ne respecte pas les conditions de la Définition 4.13, possède des informations incomplètes.

Exemple 4.6

- En considérant $\mathbb{P}$ comme l'ensemble des chemins reconnus par l'automate de la Figure 4.3(a), l'arbre de la Figure 2.2 est un arbre complet par rapport à $\mathbb{P}$ alors que l'arbre de la Figure 4.4 n'est pas complet par rapport à $\mathbb{P}$.
- Soit $\mathbb{P}'$ composé des chemins $R/A/C$, $R/A/D$, R/B et de leur préfixes. La Figure 4.5 montre des arbres qui sont complets par rapport à $\mathbb{P}'$, et d'arbres qui ne le sont pas.

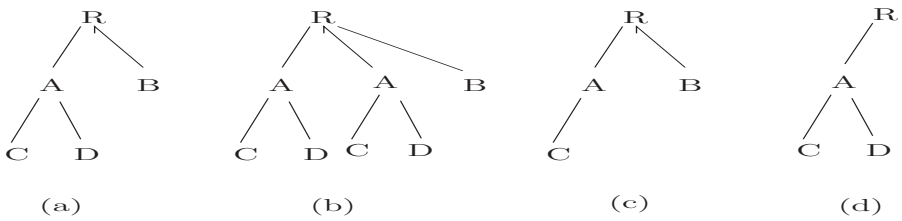


FIGURE 4.5 – *Exemple d'arbres complets ((a) et (b)) et d'arbres non complets ((c) et (d)) par rapport à $\mathbb{P}'$*

⊠

4.4 Dépendances Fonctionnelles pour XML

4.4.1 Définition

Dans le modèle relationnel, une dépendance fonctionnelle pour un document XML (XFD) se note $X \rightarrow Y$ (où X et Y sont des ensembles de chemins) et impose que pour chaque paire de n-uplets[1] t_1 et t_2 si $t_1[X] = t_2[X]$ alors $t_1[Y] = t_2[Y]$. Nos dépendances fonctionnelles peuvent être imposées dans une partie spécifique du document XML, et donc pour cette raison nous spécifions un *chemin contexte*. Nous allons introduire la définition des dépendances fonctionnelles pour XML qui a été formalisée dans [32, 33, 66, 82].

Définition 4.14 (Dépendance fonctionnelle pour un document XML)
Soit un arbre XML t, une dépendance fonctionnelle XML (XFD) est une expression de la forme

$$f = (C, (\{P_1\ [E_1], \ldots, P_k\ [E_k]\}\ \rightarrow \{Q_1\ [E'_1], \ldots, Q_n\ [E'_n]\}))$$

où :

- C est un chemin commençant à la racine de t et se terminant au *nœud contexte*. Le chemin C représente le *contexte* dans lequel la dépendance fonctionnelle doit être respectée (la dépendance est alors dite *relative*) ;
- $\{P_1, \ldots, P_k\}$ et $\{Q_1, \ldots, Q_n\}$ sont des ensembles non vides de chemins. Les chemins P_i et Q_j commencent à partir du *nœud contexte*. L'ensemble $\{P_1, \ldots, P_k\}$ représente la partie gauche (LHS) de la XFD ou la partie déterminante, et l'ensemble $\{Q_1, \ldots, Q_n\}$ représente la partie droite (RHS) ou la partie dépendante. Nous allons omettre l'accolade lorsque la partie droite est un singleton ;
- les symboles $E_1, \ldots, E_k$ et $E'_1, \ldots, E'_n$ représentent le type d'égalité associé à chaque chemin de la dépendance : si $E_i = V$ alors cette partie de la dépendance fonctionnelle doit être vérifiée avec l'égalité de valeurs, et si $E_i = N$ alors cette partie doit être vérifiée avec l'identité de nœud. Quand les symboles $E_1, \ldots, E_k$ ou $E'_1, \ldots, E'_n$ sont omis, l'égalité par valeur est utilisée par défaut.

□

Remarquons que pour une XFD, l'ensemble de chemins $\{C/P_1, \ldots, C/P_k, C/Q_1, \ldots, C/Q_n\}$ et leur préfixes définissent une *branche*. La définition de nos XFD autorise la combinaison de deux types d'égalités comme dans [122], et généralise les propositions de dépendances fonctionnelles sur XML des articles [15, 67, 119, 120]. Nous revenons sur ces propositions dans l'état de l'art de ce chapitre (section 4.9).

Exemple 4.7 Une université possède plusieurs spécialités pour le diplôme Licence. Par exemple à l'Université d'Orléans il y a : les licences informatique, MIAGE, Mathématiques, Chimie, Gestion, Droit etc. La licence se fait généralement en trois ans (niveaux L1,L2,L3). Pour une université, nous stockons pour chaque licence son intutilé, l'année académique, les informations concernant l'ensemble des étudiants, l'ensemble des cours et les inscriptions de chaque étudiant à un cours. Un cours possède des pré-réquis qui sont d'autres cours, et aussi l'inscription d'un étudiant à un cours est faite pour un niveau donné. La Figure 4.6 illustre des parties de documents XML contenant ces informations sur les licences de l'Université d'Orléans. Le nœud *students* à la position 2.2 dans l'arbre $\mathcal{T}_1$ de la Figure 4.6 a

1. Les n-uplets sont formés par les valeurs des derniers nœuds des instances des chemins de X et de Y sur un document $\mathcal{T}$.

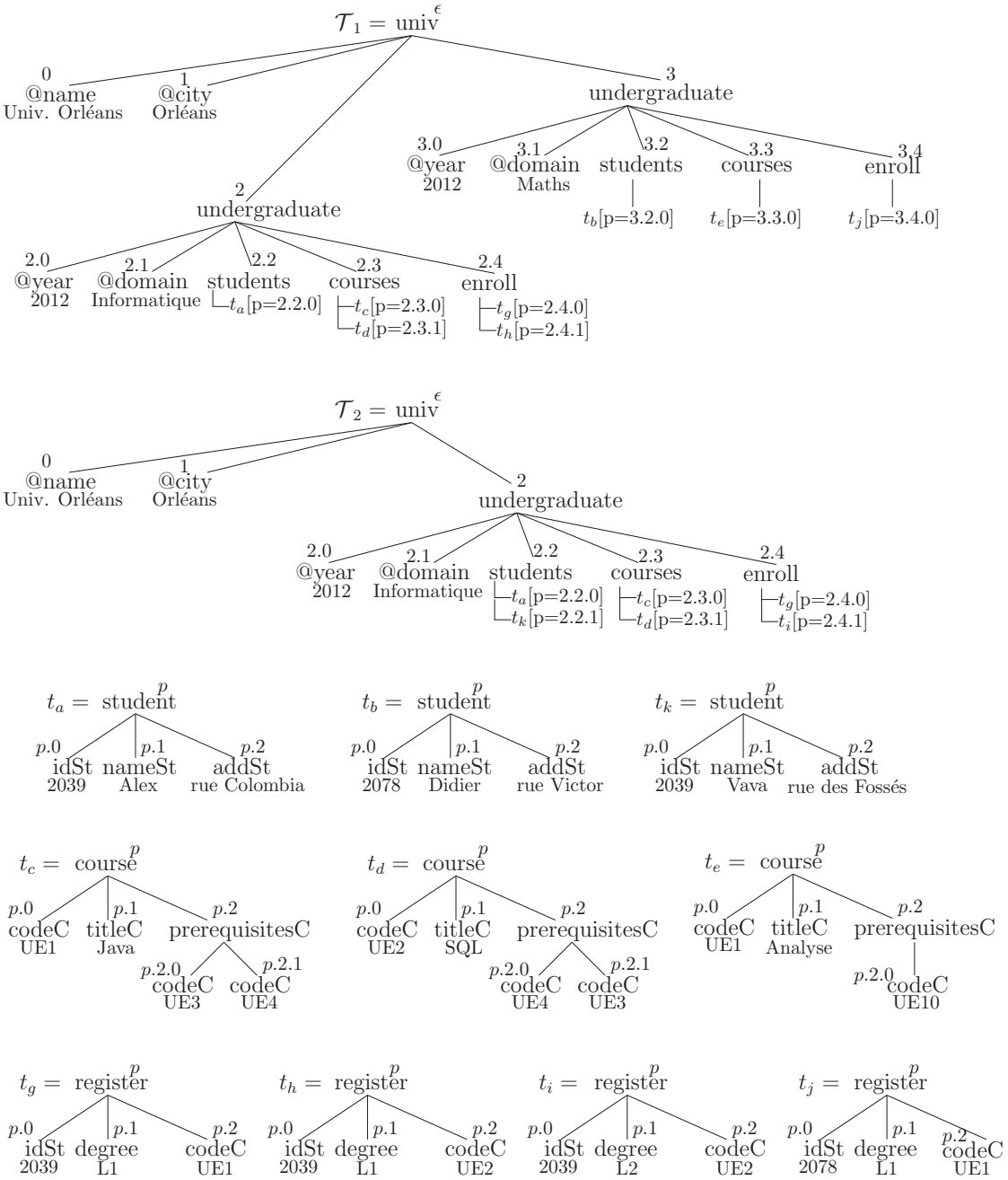


FIGURE 4.6 – *Documents XML concernant les diplômes Licence dans une université*

comme fils le sous-arbre $t_a[p = 2.2.0]$, c'est-à-dire l'arbre t_a de la Figure 4.6 où l'on remplace toutes les positions p par 2.2.0. Il en est de même pour les autres nœuds de la même forme.

Voici des exemples de XFD, qui expriment des dépendances pour des données comme celles contenues dans le document de la Figure 4.6 :

XFD1 : $(univ//courses, (\{course/codeC\} \rightarrow course/titleC))$

En considérant l'ensemble des cours d'une même Licence (contexte), les cours qui ont le même code doivent avoir aussi le même titre.

XFD2 : $(univ, (\{undergraduate//course/codeC\} \rightarrow undergraduate//course/titleC))$

En considérant l'ensemble des cours d'une même université (contexte), les cours qui ont le

même code doivent avoir aussi le même titre. Cette dépendance doit être vérifiée dans tout le document XML.

XFD3 : $(univ, (\{undergraduate/@year, undergraduate/@domain, undergraduate//register/idSt\} \rightarrow undergraduate//register/degree))$
Dans une université, lors d'une année académique pour une Licence donnée, un étudiant ne peut choisir que des cours correspondant à une même année de Licence. Cette dépendance n'autorise pas qu'un étudiant puisse être à cheval sur deux niveaux d'une même Licence.

XFD4 : $(univ, (\{undergraduate//idSt, undergraduate/@year\} \rightarrow undergraduate/@domain))$
Chaque année, un étudiant ne peut s'inscrire qu'aux cours d'une seule Licence. Cette dépendance n'autorise donc pas la double inscription.

XFD5 : $(univ//students, (\{student/idSt\} \rightarrow student[N]))$
En considérant l'ensemble des étudiants d'une Licence donnée, deux étudiants ne peuvent pas avoir le même identifiant et un étudiant n'apparaît qu'une fois.

⊠

4.4.2 Satisfaction de XFD

Une XFD définit des conditions que doivent vérifier les n-uplets formés des valeurs atteintes par les chemins spécifiés. Les définitions qui vont suivrent proviennent de [32, 66, 82]. La définition suivante précise les n-uplets considérés.

Définition 4.15 (n-uplets dans la projection d'un arbre sur une *branche*)
Soit M une *branche* et soit un ensemble de chemins $X = \{P_1, \ldots, P_k\}$ tel que $X \subset M$. Soit $\tau = \Pi_M(\mathcal{T}) = (t^i, type^i, value^i)$ une projection d'un arbre $\mathcal{T}$ sur M. On note τ_j l'instance du chemin P_j dans τ, $\forall j \in [1 \ldots k]$. Le n-uplet correspondant à X sur τ, noté $\tau[X]$, est tel que[2] : $\tau[X] = (P_1 : value^i(t^i, Last(\tau_1)), \ldots, P_k : value^i(t^i, Last(\tau_k)))$. On note par $\tau[P_j]$ la valeur $value^i(t^i, Last(\tau_j))$.

□

Le n-uplet $\tau[X]$ est l'ensemble des valeurs obtenues d'un document $\mathcal{T}$ à partir d'une projection sur une *branche* M, et est construit selon la perspective nommée des bases de données relationnelles [1] (c'est à dire que les noms des attributs composant le n-uplet sont connus). Deux n-uplets $\tau^1[X]$ et $\tau^2[X]$ sont égaux relativement à $\vec{E} = E_1, \ldots, E_k$, ce qui se note $\tau^1[X] =_{\vec{E}} \tau^2[X]$, si et seulement si :

$$\forall j \in [1 \ldots k], \tau^1[P_j] =_{E_j} \tau^2[P_j].$$

La définition ci-dessous formalise la sémantique des XFD avec les deux notions d'égalité, comme dans l'approche proposée dans [122].

Définition 4.16 (Satisfaction de XFD)
Soient $f = (C, (\{P_1\ [E_1], \ldots, P_k\ [E_k]\} \rightarrow \{Q_1\ [E'_1], \ldots, Q_n\ [E'_n]\}))$ un XFD et $\mathcal{T}$ un document XML complet par rapport à $\mathbb{P}$. Soit M la *branche* définie à partir de f. L'arbre $\mathcal{T}$ satisfait f (que l'on note $\mathcal{T} \models f$) si et seulement si pour tout $\tau^1 = \Pi^1_M(\mathcal{T})$ et $\tau^2 = \Pi^2_M(\mathcal{T})$ qui sont des projections de $\mathcal{T}$ sur M et coïncident au moins sur le contexte C, on a :

$$\text{Si } \tau^1[C/X] =_{\vec{E}} \tau^2[C/X] \text{ alors } \tau^1[C/Y] =_{\vec{E'}} \tau^2[C/Y] \text{ avec}$$

2. Lorsqu'il n'y a pas d'ambiguïté pour le tuple, nous allons omettre les chemins associés aux valeurs.

$C/X = \{C/P_1, \ldots, C/P_k\}$; $C/Y = \{C/Q_1, \ldots, C/Q_n\}$; $\vec{E} = E_1, \ldots, E_k$ et $\vec{E'} = E'_1, \ldots, E'_n$. □

Exemple 4.8 Considérons les documents $\mathcal{T}_1$ et $\mathcal{T}_2$ de la Figure 4.6 et les cinq dépendances de l'Exemple 4.7.

- XFD1 : pour la *branche M* formée à partir de l'ensemble de chemins dans la partie gauche $C/X = \{univ//courses/course/codeC\}$ et du chemin $C/Q = univ//courses/course/titleC$, nous obtenons dans $\mathcal{T}_1$ les trois projections τ^1, τ^2 et τ^3, telles que $\tau^1[C/X] = (UE1)$, $\tau^2[C/X] = (UE2)$, $\tau^3[C/X] = (UE1)$. Seulement τ^1, τ^2 s'intersectent sur le contexte $univ//courses$. Puisque $\tau^1[C/X] \neq_V \tau^2[C/X]$ et qu'on ne peut pas comparer $\tau^3[C/X]$ avec $\tau^1[C/X]$ et $\tau^2[C/X]$ alors le document $\mathcal{T}_1$ respecte la dépendance XFD1.
- XFD2 : pour la dépendance XFD2 nous obtenons dans $\mathcal{T}_1$ les trois projections τ^1, τ^2 et τ^3, telles que $\tau^1[C/X] = (UE1)$, $\tau^2[C/X] = (UE2)$, $\tau^3[C/X] = (UE1)$ et τ^1, τ^2, τ^3 s'intersectent bien sur le contexte $univ$. Puisque $\tau^1[C/X] =_V \tau^3[C/X]$ et que $\tau^1[C/Q] \neq_V \tau^3[C/Q]$ ('Java' $\neq_V$ 'Analyse'), alors la dépendance XFD2 est violée par le document $\mathcal{T}_1$.
- XFD3 : la dépendance XFD3 n'autorise pas qu'un étudiant puisse être à cheval sur deux niveaux d'une même Licence. La *branche M* est formée des chemins dans $C/X = \{univ/undergraduate/@year, univ/undergraduate/@domain, univ/undergraduate//register/idSt\}$ et du chemin $C/Q = univ/undergraduate//register/degree$. Pour M, nous obtenons dans $\mathcal{T}_1$ les trois projections τ^1, τ^2 et τ^3, telles que $\tau^1[C/X] = (2012, Informatique, 2039)$, $\tau^2[C/X] = (2012, Informatique, 2039)$, $\tau^3[C/X] = (2012, Maths, 2078)$ et τ^1, τ^2, τ^3 s'intersectent bien sur le contexte $univ$. Puisque $\tau^1[C/X] =_V \tau^2[C/X]$ et que $\tau^1[C/Q] =_V \tau^2[C/Q]$ alors le document $\mathcal{T}_1$ respecte la dépendance XFD3.

  Par contre dans le document $\mathcal{T}_2$, les projections τ^1, τ^2 sont telles que $\tau^1[C/X] = \tau^2[C/X] = (2012, Informatique, 2039)$ et $\tau^1[C/Q] = L1$, $\tau^2[C/Q] = L2$. Le document $\mathcal{T}_2$ ne respecte donc pas la dépendance XFD3 puisque nous avons $\tau^1[C/X] =_V \tau^2[C/X]$ et que $\tau^1[C/Q] \neq_V \tau^2[C/Q]$.
- XFD4 : la dépendance XFD4 n'autorise pas qu'un étudiant s'inscrive dans deux Licences distinctes la même année. Nous pouvons vérifier de la même manière que les documents $\mathcal{T}_1$ et $\mathcal{T}_2$ vérifient la dépendance XFD4. L'étudiant d'identifiant 2039 n'apparaît que dans la Licence 'Informatique', et l'étudiant d'identifiant 2078 n'apparaît aussi que dans la Licence 'Maths'. Pour le chemin PL $univ/undergraduate//idSt$, nous avons au moins deux instances possibles dans chaque sous-arbre concernant une Licence. Pour le sous-arbre de la Licence 'Informatique', deux instances sont $\epsilon/2/2.2/2.2.0/2.2.0.0$ et $\epsilon/2/2.4/2.4.0/2.4.0.0$ correspondant chacune respectivement aux chemins simples $univ/undergraduate/students/student/idSt$ et $univ/undergraduate/enroll/register/idSt$.
- XFD5 : $univ//students, (\{student/idSt\} \rightarrow student[N])$. La dépendance XFD5 utilise l'identité de nœud pour le chemin de droite. La dépendance XFD5 impose dans l'ensemble des étudiants d'une même Licence que deux étudiants ne peuvent pas avoir le même identifiant. La *branche M* est formée des chemins dans $\{univ//students/student/idSt, univ//students/student\}$. En considérons le document $\mathcal{T}_2$, nous obtenons les trois projections τ^1, τ^2 et τ^3 telles que $\tau^1[C/X, C/Q] = (2039, 2.2.0)$, $\tau^2[C/X, C/Q] = (2039, 2.2.1)$, $\tau^3[C/X, C/Q] = (2078, 3.2.0)$ et seul τ^1,

τ^2 s'intersectent sur le contexte *univ//students*. Puisque $\tau^1[C/X] =_V \tau^2[C/X]$ et que $\tau^1[C/Q] \neq_N \tau^3[C/Q]$, alors la dépendance XFD5 est violée par le document $\mathcal{T}_2$.

⊠

Il est important de remarquer qu'il est difficile (voir impossible) d'exprimer la dépendance XFD4 avec que des chemins simples sans utiliser de chemins PL. Soit le document XML $\mathcal{T}_3$ de la Figure 4.7. Nous allons supposer qu'il n'y a pas de nœud *idSt* ayant la valeur 2039 dans le sous-arbre enraciné en 3.4.0. Ainsi le document $\mathcal{T}_3$ exprime le fait que l'étudiant d'identifiant 2039 est inscrit en Licence 'Informatique' mais suit des cours en Licence 'Maths'. Il est évident que $\mathcal{T}_3 \not\models$ XFD4. En essayant d'exprimer la dépendance XFD4 via les deux dépendances XFD4a *univ*, ({*undergraduate/students/student/idSt*, *undergraduate/@year*} → *undergraduate/@domain*) et XFD4b *univ*, ({*undergraduate/enroll/register/idSt*, *undergraduate/@year*} → *undergraduate/@domain*), nous constatons que $\mathcal{T}_3 \models$ XFD4a, XFD4b. Il en est de même pour la dépendance XFD4c : *univ*, ({*undergraduate/students/student/idSt*, *undergraduate/enroll/register/idSt*, *undergraduate/@year*} → *undergraduate/@domain*) qui ne peut pas remplacer XFD4 car $\mathcal{T}_3 \models$ XFD4c.

Nous voyons donc que le fait d'utiliser un language de chemins plus puissant dans une XFD, permet d'exprimer plus de choses en comparaison avec une XFD ne comportant que des chemins simples.

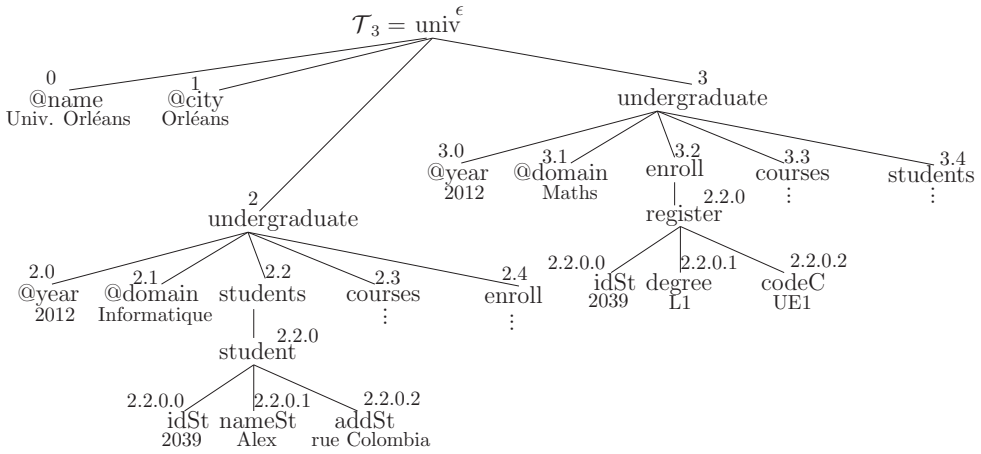


FIGURE 4.7 – *Documents XML violant la dépendance XFD4*

4.5 Raisonner sur les dépendances fonctionnelles XML

Un document XML $\mathcal{T}$ satisfait un ensemble de XFD $\mathcal{F}$, noté par $\mathcal{T} \models \mathcal{F}$, si $\mathcal{T} \models f$ pour toute XFD f dans $\mathcal{F}$. Il est en effet important de pouvoir raisonner sur le fait qu'une dépendance f est aussi satisfaite sur un document $\mathcal{T}$ lorsqu'un ensemble $\mathcal{F}$ est satisfait. La définition suivante formalise l'implication de XFD.

Définition 4.17 (Implication de XFD)
Soit un ensemble de XFD $\mathcal{F}$. L'ensemble $\mathcal{F}$ implique logiquement f, noté par $\mathcal{F} \models f$, si pour tout document XML $\mathcal{T}$ tel que $\mathcal{T} \models \mathcal{F}$ alors $\mathcal{T} \models f$.

□

En nous basant sur la notion d'implication, nous pouvons introduire la définition de la fermeture d'un ensemble de XFD.

Définition 4.18 (Fermeture d'un ensemble de XFD)
La fermeture d'un ensemble de dépendances $\mathcal{F}$, noté $\mathcal{F}^+$, est l'ensemble contenant toutes les dépendances XFD qui sont logiquement impliquées par $\mathcal{F}$, c'est à dire, $\mathcal{F}^+ = \{f \mid \mathcal{F} \models f\}$. □

Notation : Dans le reste du manuscrit, étant donné une XFD $(C, (X \rightarrow A))$ où $X = \{P_1, \ldots, P_n\}$ est un ensemble de chemins et A est un chemin, on utilise C/X comme un raccourci pour l'ensemble $\{C/P_1, \ldots, C/P_n\}$.

4.5.1 Système d'axiomes

Pour calculer $\mathcal{F}^+$, on a besoin de règles d'inférence qui expliquent comment des dépendances peuvent engendrer d'autres dépendances. Dans cette section nous présentons notre système d'axiomes et prouvons qu'il est correct (*i.e.* nous ne pouvons pas déduire à partir de $\mathcal{F}$ une fausse dépendance) et complet (*i.e.* nous arrivons à déduire toutes les dépendances vraies à partir de $\mathcal{F}$). Notre système d'axiomes est similaire à celui proposé dans [118], mais diffère sur deux points importants : (i) nos XFD sont définies par rapport à un contexte (donc pas forcément par rapport à la racine) et (ii) nous avons deux types d'égalités. Dans [118] les auteurs utilisent l'égalité par nœud pour les nœuds internes et l'égalité par valeur pour les feuilles. Nous considérons des documents XML complets, c'est à dire ne possédant pas d'informations manquantes.

Définition 4.19 (Règles d'inférence)
Soit un document $\mathcal{T}$ complet par rapport à $\mathbb{P}$ et des XFD définies sur les chemins dans $\mathbb{P}$, nos axiomes sont :

A1 : **Réflexivité**
$(C, (\{P_1[E_1], \ldots, P_n[E_n]\} \rightarrow P_i[E_i])), \forall i \in [1 \ldots n]$.

A2 : **Augmentation**
Si $(C, (\{P_1[E_1], \ldots, P_n[E_n]\} \rightarrow \{Q_1[E'_1], \ldots, Q_m[E'_m]\}))$ alors
$(C, (\{P_0[E_0], P_1[E_1], \ldots, P_n[E_n]\} \rightarrow \{P_0[E_0], Q_1[E'_1], \ldots, Q_m[E'_m]\}))$.

A3 : **Transitivité**
Si $(C, (\{P_1[E_1], \ldots, P_n[E_n]\} \rightarrow \{Q_1[E'_1], \ldots, Q_m[E'_m]\}))$ et
$(C, (\{Q_1[E'_1], \ldots, Q_m[E'_m]\} \rightarrow S[E_s]))$ alors $(C, (\{P_1[E_1], \ldots, P_n[E_n]\} \rightarrow S[E_s]))$.

A4 : **Branche Préfixe**
Si $(C, (\{P'_1[E'_1], \ldots, P'_n[E'_n]\} \rightarrow P_{n+1}[E_{n+1}]))$ et il existe des chemins $C/P_1, \ldots, C/P_n$ (pas nécessairement distincts) tels que :

(i) $P'_i \cap P_{n+1} \preceq_{PL} P_i$ et

(ii) $P_i \preceq_{PL} P'_i$ ou $P_i \preceq_{PL} P_{n+1}$

alors $(C, (\{P_1[E_1], \ldots, P_n[E_n]\} \rightarrow P_{n+1}[E_{n+1}]))$.

A5 : **Unicité des Ascendants**
Si Q est un préfixe de P alors $(C, (P[N] \rightarrow Q[N]))$.

A6 : **Unicité de l'Attribut**
Si $Last(P) \in \Sigma_{att}$ alors $(C, (Parent(P)[E] \rightarrow P[E]))$.

A7 : **Unicité du contexte**
$(C, (\{P_1[E_1], \ldots, P_n[E_n]\} \rightarrow [][E_{n+1}]))$.

A8 : **Extension du Contexte**
Si $(C, (\{P_1\,[E_1], \ldots, P_n\,[E_n]\} \rightarrow P_{n+1}\,[E_{n+1}]))$ et il existe un chemin Q tel que $P_1 = Q/P'_1$, $\cdots$, $P_{n+1} = Q/P'_{n+1}$ alors $(C/Q, (\{P'_1\,[E_1], \ldots, P'_n\,[E_n]\} \rightarrow P'_{n+1}\,[E_{n+1}]))$.

A9 : **De l'identité par nœud à l'égalité par valeur**
Pour n'importe quel chemin, $(C, (P\,[N] \rightarrow P\,[V]))$.

□

Les axiomes A1-A3 sont similaires aux axiomes d'Armstrong pour le modèle relationnel. Leur intuition est donc la même que pour le modèle relationnel.

Exemple 4.9 Considérons le document XML $\mathcal{T}_1$ de la Figure 4.6.

A1 : $(univ/undergraduate, (\{idSt, nameSt, addSt\} \rightarrow addSt))$. L'axiome A1 est toujours vérifié quelque soit le document XML. Il concerne donc les dépendances triviales.

A2 : si $(univ//courses, (\{course/codeC\} \rightarrow course/titleC))$ alors $(univ//courses, (\{course/codeC, course/prerequisitesC\} \rightarrow \{course/titleC, course/prerequisitesC\}))$.
Il est clair que si les cours ayant le même code correspondent à un seul titre alors les cours ayant le même code et les mêmes pré-réquis vont correspondre aussi au même titre et mêmes pré-réquis.

A3 : si $univ/undergraduate, (\{courses//codeC\} \rightarrow course[N])$ et $(univ/undergraduate, (\{courses/course[N]\} \rightarrow titleC[N]))$ alors $(univ/undergraduate, (\{courses//codeC\} \rightarrow titleC[N]))$. En prenant une Licence comme contexte, nous considérons qu'un cours est défini par son code (*i.e.*, *CodeC* est la clé), et donc il n'existe pas deux cours qui ont le même *codeC*. Par ailleurs, un cours n'a qu'un seul titre. Dans ce contexte, nous pouvons déduire via l'axiome A3 que le code d'un cours (*codeC*) détermine de façon unique son titre (*titleC*).

⊠

Pour une dépendance f, l'axiome A4 spécifie que l'ensemble obtenu en remplaçant un ou plusieurs chemins de la partie gauche par certains chemins parmi les préfixes des chemins de f, détermine aussi le chemin à droite. Les chemins qui nous intéressent sont ceux situés entre (i) le plus long préfixe commun d'un chemin de la partie gauche et du chemin droit, et (ii) le chemin de gauche considéré ou le chemin de droite.

Exemple 4.10 Considérons le document XML $\mathcal{T}_1$ de la Figure 4.6.

A4 : soit la XFD $(univ, (\{undergraduate/students//idSt, undergraduate/@year\} \rightarrow undergraduate/@domain))$. Les plus long préfixes commun sont :

- $undergraduate/students//idSt \cap undergraduate/@domain = undergraduate$ et
- $undergraduate/@year \cap undergraduate/@domain = undergraduate$.

Avec l'axiome A4, nous obtenons par exemple :

- $(univ, (\{undergraduate/students/student, undergraduate/@year\} \rightarrow undergraduate/@domain))$ ou
- $(univ, (\{undergraduate/students, undergraduate/@year\} \rightarrow undergraduate/@domain))$ ou
- $(univ, (\{undergraduate\} \rightarrow undergraduate/@domain))$

La dépendance n'autorise pas qu'un étudiant s'inscrive dans deux Licences distinctes la même année. A partir de cette dépendance, nous pouvons déduire parmi d'autres XFD, qu'une licence n'est associée qu'a un seul domaine.

⊠

En considérant l'identité de nœud, l'axiome A5 exprime qu'un chemin détermine n'importe quel chemin qui est son préfixe. Ceci est possible parce qu'une position identifie de façon unique un nœud dans un arbre et un nœud possède un seul nœud ancêtre pour un chemin préfixe. L'axiome A6 exprime qu'un chemin parent à un attribut détermine cet attribut. Ceci est vrai parce que dans un document XML, un nœud ne possède qu'un seul nœud attribut pour chacun de ses chemins attribut. Puisqu'il existe qu'une seule racine dans un document XML, l'axiome A7 indique que n'importe quel ensemble de chemins détermine le chemin racine.

Exemple 4.11 Considérons le document XML $\mathcal{T}_1$ de la Figure 4.6.

A5 : Soit le chemin $P = undergraduate//register/idSt$, avec l'axiome A5 nous pouvons engendrer par exemple, $(univ, (\{undergraduate//register/idSt[N]\} \rightarrow undergraduate//register[N]))$.

A6 : Soit le chemin $P = undergraduate/@year$, avec l'axiome A6, nous pouvons engendrer la dépendance $(univ, (\{undergraduate[N]\} \rightarrow undergraduate/@year[N]))$.

⊠

Remarquons que l'axiome $A5$ est fausse en prenant en compte l'égalité par valeur. L'arbre $\mathcal{T}_1$ de la Figure 4.6 ne satisfait pas la dépendance $(univ, (\{undergraduate//course/-prerequisitesC[V]\} \rightarrow undergraduate//course[V]))$. En fait nous avons

$$Last(\epsilon/2.3/2.3.0/2.3.0.2) =_V Last(\epsilon/2.3/2.3.1/2.3.1.2)$$

alors que $Last(\epsilon/2.3/2.3.0) \neq_V Last(\epsilon/2.3/2.3.1)$.

Remarquons aussi que malgré que nous avons l'égalité par valeur, la règle suivante $(C, (P[V] \rightarrow P/Q[V]))$ est fausse. Considérons la XFD $(univ, (\{undergraduate//prerequisitesC[V]\} \rightarrow undergraduate//prerequisitesC/codeC[V]))$. L'arbre $\mathcal{T}_1$ de la Figure 4.6 ne satisfait pas cette dépendance parce que nous avons

$$Last(\epsilon/2.3/2.3.0/2.3.0.2) =_V Last(\epsilon/2.3/2.3.1/2.3.1.2)$$

alors que $Last(\epsilon/2.3/2.3.0/2.3.0.2/2.3.0.2.0) \neq_V Last(\epsilon/2.3/2.3.1/2.3.1.2/2.3.1.2.0)$.

Pour une XFD f, avec l'axiome A8 nous pouvons étendre (ou rallonger) le chemin contexte C de f pour obtenir une nouvelle XFD avec le contexte C/Q, où le chemin Q est un préfixe pour tous les chemins apparaissant dans la dépendance f.

Exemple 4.12 Considérons le document XML $\mathcal{T}_1$ de la Figure 4.6.

A8 : Si $(univ/undergraduate, (\{students/student/idSt\} \rightarrow students/student/nameSt))$ alors $(univ/udergraduate/students, (\{student/idSt\} \rightarrow student/nameSt))$. Si dans le contexte *undergraduate*, le chemin *idSt* identifie le nom d'un étudiant, alors cela est aussi vrai pour le contexte *students*.

⊠

L'axiome A9 affirme que pour un chemin donné, l'identité de nœud entraîne l'égalité par valeur. Ceci est vrai car deux nœuds se trouvant à la même position ont forcément les mêmes valeurs.

La grande différence entre notre système d'axiomes et celui de [118] est la présence des axiomes A7, A8, A8. Aussi la différence entre la définition de XFD va influencer sur les

preuves de correction et de complétude du système d'axiomes.

L'intérêt d'avoir un système d'axiomes (ou des règles d'inférences) pour les XFD, est de pouvoir répondre à la question suivante : « *Étant donné un ensemble de XFD $\mathcal{F}$ et une XFD f, est ce que $\mathcal{F}$ engendre f ?* ». Pour y répondre il faudra montrer qu'à partir de l'ensemble $\mathcal{F}$ et en appliquant les règles d'inférences, on arrive à engendrer la dépendance f. La définition suivante nous en dit plus sur ce qu'est une séquence de dérivation.

Définition 4.20 (Dérivation de XFD)
Soit $\mathcal{F}$ un ensemble de XFD, on dit qu'une XFD f est dérivable à partir des dépendances dans $\mathcal{F}$ par les règles d'inférences de la Définition 4.19, notée $\mathcal{F} \vdash f$ (*i.e.* $\mathcal{F}$ implique sémantiquement f), ssi il existe une séquence de XFD $f_1, f_2, \ldots, f_n$ telle que :

1. $f = f_n$ et
2. pour tout $i = 1, \ldots, n$ la dépendance f_i est dans $\mathcal{F}$ ou est obtenue à partir de $f_1, f_2, \ldots f_{i-1}$ en appliquant un axiome parmi les règles du système d'axiomes (Définition 4.19).

La séquence de XFD $f_1, f_2, \ldots, f_n$ est appelée séquence de dérivation (ou une démonstration) de f à partir de $\mathcal{F}$. La notation $\mathcal{F} \vdash f$ se lit aussi $\mathcal{F}$ engendre f, ou $\mathcal{F}$ entraîne f.
Soit α une séquence de dérivation. Nous notons par $\alpha(\mathcal{F})$ la sous-séquence de α qui ne contient que les dépendances de $\mathcal{F}$.

□

Exemple 4.13 Voici un exemple de dérivation. Prenons par exemple l'ensemble $\mathcal{F} = \{$
$(univ/undergraduate, (\{students//idSt\} \rightarrow students/student))$,
$(univ/undergraduate, (\{students//nameSt\} \rightarrow students//addSt))$ $\}$.
La XFD $f = (univ/undergraduate, (\{students//idSt\} \rightarrow students//addSt))$ est engendrée par l'ensemble $\mathcal{F}$ car la suite $\alpha = f_1, f_2, f_3, f_4$ ci-dessous constitue une dérivation de f à partir de $\mathcal{F}$:

f_1 : $(univ/undergraduate, (\{students//nameSt\} \rightarrow students//addSt))$ est dans $\mathcal{F}$

f_2 : $(univ/undergraduate, (\{students/student\} \rightarrow students//addSt))$ est obtenue en appliquant l'axiome A4 (Branche Préfixe) à f_1

f_3 : $(univ/undergraduate, (\{students//idst\} \rightarrow students/student))$ est dans $\mathcal{F}$

f_4 : $(univ/undergraduate, (\{students//idSt\} \rightarrow students//addSt))$ est obtenue en appliquant A3 (Transitivité) sur f_3 et f_2

Nous avons la sous-séquence $\alpha(\mathcal{F}) = f_1, f_3$.

⊠

4.5.2 Axiomes additionnels

A partir des règles d'inférences du système d'axiomes introduit dans la Définition 4.19, nous pouvons dériver d'autres règles que nous appellons des axiomes additionnels au système d'axiomes. Ces nouveaux axiomes nous seront très utiles dans des preuves, comme par exemple celle de la complétude (Section A.5) de notre système d'axiomes, même s'ils ne sont pas indispensables pour prouver la complétude.

Définition 4.21 (Règles d'inférences additionnelles)
Soit un document $\mathcal{T}$ complet par rapport à $\mathbb{P}$ et des XFD définies sur les chemins dans $\mathbb{P}$, nos axiomes additionnels sont les suivants :

A10 : **Union**
Si $(C, (\{P_1\,[E_1], \ldots, P_n\,[E_n]\} \rightarrow Q\,[E_{n+1}]))$ et $(C, (\{P_1\,[E_1], \ldots, P_n\,[E_n]\} \rightarrow R\,[E_{n+2}]))$ alors $(C, (\{P_1\,[E_1], \ldots, P_n\,[E_n]\} \rightarrow \{Q\,[E_{n+1}], R\,[E_{n+2}]\}))$.

A11 : **Décomposition**
Si $(C, (\{P_1\,[E_1], \ldots, P_n\,[E_n]\} \rightarrow \{Q_1\,[E'_1], \ldots, Q_m\,[E'_m]\}))$ et l'ensemble des chemins $\{R_1, \ldots, R_k\} \subseteq \{Q_1, \ldots, Q_m\}$ alors $(C, (\{P_1\,[E_1], \ldots, P_n\,[E_n]\} \rightarrow \{R_1\,[E''_1], \ldots, R_k\,[E''_k]\}))$.

A12 : **Pseudo-transitivité**
Si $(C, (\{P_1\,[E_1], \ldots, P_n\,[E_n]\} \rightarrow \{Q_1\,[E'_1], \ldots, Q_m\,[E'_m]\}))$ et $(C, (\{Q_1\,[E'_1], \ldots, Q_m\,[E'_m], R_1\,[E''_1], \ldots, R_k\,[E''_k]\} \rightarrow S\,[E_s]))$ alors $(C, (\{P_1\,[E_1], \ldots, P_n\,[E_n], R_1\,[E''_1], \ldots, R_k\,[E''_k]\} \rightarrow S\,[E_s]))$.

A13 : **Valeur Unique d'un sous-arbre**
Si $(C, (\{P_1\,[E_1], \ldots, P_n\,[E_n]\} \rightarrow P_{n+1}\,[E_{n+1}]))$ et pour tout $i \in [1 \ldots n]$ le plus long préfixe commun de C/P_i et C/P_{n+1} est le chemin contexte C (*i.e.*, $P_i \cap P_{n+1} = []$) alors $(C, (P_0\,[E_0] \rightarrow P_{n+1}\,[E_{n+1}]))$ pour n'importe quel chemin C/P_0.

□

L'intuition des axiomes A10, A11 et A12 est la même que pour le modèle relationnel. Remarquons que grâce aux axiomes A10 et A11 (Décomposition) que nous pouvons écrire nos XFD avec qu'un seul chemin à droite. En effet une conséquence de ces deux axiomes est que la dépendance $(C, (\{P_1\,[E_1], \ldots, P_k\,[E_k]\} \rightarrow \{Q_1\,[E'_1], \ldots, Q_m\,[E'_m]\}))$ est satisfaite si et seulement si les dépendances $\forall\, i \in [1, \ldots, m]$, $(C, (\{P_1\,[E_1], \ldots, P_k\,[E_k]\} \rightarrow \{Q_i\,[E'_i]\}))$ sont aussi satisfaites. Ainsi, c'est suffisant d'avoir une XFD avec un seul chemin à droite. En ce qui concerne l'axiome A13, il exprime le fait que toutes les instances pour le sous-arbre désigné par le chemin de droite P_{n+1} ont la même valeur dans le contexte C. Puisque l'intersection entre chacun des chemins à gauche de la dépendance avec le chemin de droite est le chemin vide, nous pouvons associer n'importe quelles instances des chemins de gauche à toutes les instances du chemin de droite. Pour que la dépendance soit donc vérifiée, il faudrait que (i) toutes les instances du chemin de droite aient la même valeur dans le cas de l'égalité par valeur ou soient la même instance dans le cas d'identité de nœud. Si on a bien la condition (i) alors toutes les dépendances ayant le chemin P_{n+1} à droite, seront aussi vérifiées.

Exemple 4.14 Pour illustrer l'axiome A13, nous allons considérer la dépendance $(univ/undergraduate, (\{students//idSt\} \rightarrow @domain))$. Dans le contexte *univ/undergraduate*, un étudiant est associé à un seul domaine. Puisque la dépendance vérifie les conditions de l'axiome $A13$, nous pouvons déduire la dépendance $(univ/undergraduate, (\{courses//codeC\} \rightarrow @domain))$ qui exprime que les cours ayant le même *code*C appartiennent au même domaine.

⊠

4.5.3 Dépendances triviales

La notion de XFD triviale (c'est à dire toujours satisfaite ou contient des informations redondantes, pour tout arbre t contenant les instances de ses chemins) a été étudiée dans les articles [14, 120, 63], d'où nous pouvons synthétiser la propriété suivante :

Proposition 4.1 (XFD triviale)
Une XFD $(C, (\{P_1\,[E_1], \ldots, P_n\,[E_n]\} \rightarrow Q\,[E]))$ *où* $P_1, \ldots, P_n, Q$ *sont des chemins, est une XFD triviale si l'une des conditions suivantes est respectée :*

1. $Q = []$. *Cas où la partie droite correspond au contexte.*

2. *$Q = P_i$ et $E = E_i$ pour un i tel que $1 \leq i \leq n$. Cas où un chemin dans la partie gauche est identique au chemin de la partie droite. La dépendance est aussi triviale dans le cas où $E_i = N$ et $E = V$.*
3. *$P_i = Q/P'_i$ et $E_i = N$ pour un $i \in [1, \ldots, n]$. Dans ce cas, il existe un chemin P_i dans la partie gauche dont Q est un préfixe. L'égalité considérée est l'identité de nœud.*
4. *$Last(Q) \in \Sigma_{att}$, $P_i = Parent(Q)$ pour un $i \in [1, \ldots, n]$. Dans ce cas, un nœud (père) détermine la valeur de son attribut.*
5. *il existe $i, j \in [1, \ldots, n]$, $j \neq i$ tel que $P_j \preceq P_i$ et $E_i = N$. On est ici dans le cas où un chemin dans la partie gauche est préfixe d'un autre chemin dans la partie gauche.*
6. *il existe $i, j \in [1, \ldots, n]$, $j \neq i$ tel que $P_i = Parent(P_j)$, $Last(P_j) \in \Sigma_{att}$. Dans ce cas, un chemin dans la partie gauche est le parent d'un autre chemin qui est un attribut et se trouvant aussi dans la partie gauche.*

□

Pour les cas 1 à 4, les dépendances sont toujours satisfaites. Pour les cas 5 à 6, les dépendances ne sont pas significatives puisque les informations contenues dans P_j sont redondantes. Nous pouvons remarquer que les dépendances triviales tournent autour des axiomes A1 (Réflexivité), A5 (Unicité des Ascendants), A6 (Unicité de l'Attribut), et A7 (Unicité du contexte) et A9 (De l'identité de nœud à l'égalité par valeur).

Pour illuster les dépendances triviales, prenons le document $\mathcal{T}_1$ de la Figure 4.6, la dépendance $(univ, (\{undergraduate/students\,[N]\} \rightarrow undergraduate\,[N]))$ est toujours satisfaite. De même, la dépendance $(univ, (\{undergraduate\,[N]\} \rightarrow undergraduate/@domain\,[N]))$ est une XFD triviale.

4.5.4 Fermeture d'un ensemble de chemins

A la fin de la Section 4.5.1, plus précisément avec la Définition 4.20, nous avons vu ce qu'est une dérivation d'une XFD f à partir d'un ensemble de XFD $\mathcal{F}$. On a aussi conclut que lorsqu'on arrive à construire une dérivation de f en partant de $\mathcal{F}$, alors on peut décider que $\mathcal{F} \vdash f$ ($\mathcal{F}$ engendre f). Nous allons maintenant fournir un algorithme qui va permettre de décider si $\mathcal{F} \vdash f$ ou pas. L'algorithme se base sur la notion de fermeture d'un ensemble de chemins, et principalement sur le Lemme 4.2.

Définition 4.22 (Fermeture d'un ensemble de chemins)
Soient $X = \{P_1, \ldots, P_n\}$ un ensemble de chemins et C un chemin simple qui sera le contexte. Soit $\vec{E} = E_1, \ldots, E_n$ un vecteur de types d'égalité associé à l'ensemble X. La fermeture de (C, X) par rapport à $\mathcal{F}$, notée $(C, X[\vec{E}])^+_{\mathcal{F}}$, est l'ensemble maximal de chemins $\{C/Q_1[E'_1], \ldots, C/Q_m[E'_m]\}$ tel que la dépendance $(C, (X\,[\vec{E}] \rightarrow \{Q_1\,[E'_1], \ldots, Q_m\,[E'_m]\}))$ peut être engendrée à partir de $\mathcal{F}$ en utilisant le système d'axiomes. Formellement, $(C, X\,[\vec{E}])^+_{\mathcal{F}} = \{C/Q[E'] \mid \mathcal{F} \vdash (C, (X\,[\vec{E}] \rightarrow Q\,[E']))\}$. Lorsqu'il n'y a pas d'ambiguïté sur l'ensemble $\mathcal{F}$ utilisé, on le note $(C, X[\vec{E}])^+$. □

Comme dans le modèle relationnel, le résultat central sur la fermeture d'un ensemble de chemins est qu'il permet de donner un aperçu sur le fait qu'une XFD est déduite d'un ensemble $\mathcal{F}$ en utilisant la fermeture d'un ensemble de chemins. Le prochain lemme formalise cela :

Lemme 4.2 *Soient C un chemin simple et $X = \{P_1, \ldots, P_n\}$, $Y = \{Q_1, \ldots, Q_m\}$ des ensembles de chemins. Soient $\vec{E} = E_1, \ldots, E_n$ et $\vec{E'} = E'_1, \ldots, E'_m$ des vecteurs de types d'égalités associés à X et*

Y respectivement. Nous avons :

$$\mathcal{F} \vdash (C, (\{P_1\,[E_1], \ldots, P_n\,[E_n]\} \rightarrow \{Q_1\,[E'_1], \ldots, Q_m\,[E'_m]\}))\ \textit{ssi}\ C/Y[\vec{E'}] \subseteq (C, X[\vec{E}])^+.$$

Démonstration.

1. Supposons que $\mathcal{F} \vdash (C, (\{P_1\,[E_1], \ldots, P_n\,[E_n]\} \rightarrow \{Q_1\,[E'_1], \ldots, Q_m\,[E'_m]\}))$. En utilisant l'axiome A11 (Décomposition) nous obtenons pour tout $i \in [1, \ldots, m]$, $\mathcal{F} \vdash (C, (\{P_1\,[E_1], \ldots, P_n\,[E_n]\} \rightarrow Q_i\,[E'_i]))$. On en déduit donc que chacun des chemins $C/Q_1\,[E'_1], \ldots, C/Q_m\,[E'_m]$ est dans $(C, X\,[\vec{E}])^+$ et donc nous avons $C/Y[\vec{E'}] \subseteq (C, X\,[\vec{E}])^+$.
2. Supposons que $C/Y[\vec{E'}] \subseteq (C, X[\vec{E}])^+$. Par la définition de $(C, X[\vec{E}])^+$, nous savons que $\mathcal{F} \vdash (C, (\{P_1\,[E_1], \ldots, P_n\,[E_n]\} \rightarrow Q_i\,[E'_i]))$ pour chaque chemin $Q_i[E_i]$ tel que $i \in [1, \ldots, m]$. En appliquant l'axiome A10 (Union), nous obtenons $\mathcal{F} \vdash (C, (\{P_1\,[E_1], \ldots, P_n\,[E_n]\} \rightarrow \{Q_1\,[E'_1], \ldots, Q_m\,[E'_m]\}))$.

◁

Grâce au Lemme 4.2 nous pouvons donner l'algorithme (qui est le même que celui utilisé dans le modèle relationnel) pour décider si $\mathcal{F}$ engendre $f = (C, (X\,[\vec{E}] \rightarrow Y\,[\vec{E'}]))$ qui est le suivant :

1. calculer $(C, X[\vec{E}])^+_{\mathcal{F}}$;
2. si $C/Y[\vec{E'}] \subseteq (C, X[\vec{E}])^+_{\mathcal{F}}$ alors $\mathcal{F} \vdash f$, sinon $\mathcal{F} \nvdash f$.

Nous introduisons l'algorithme qui permet de calculer $(C, X[\vec{E}])^+_{\mathcal{F}}$ dans la section 4.6. Le résultat du Lemme 4.2 nous sera très utile dans la preuve de complétude du système d'axiomes.

Les preuves de correction et de complétude du système d'axiomes se trouvent respectivement dans l'Annexe A.4 et dans l'Annexe A.5.

4.6 Fermeture d'un ensemble de chemins

Pour compléter l'algorithme qui permet de décider si un ensemble de dépendances $\mathcal{F}$ engendre une dépendance f, nous allons proposer un algorithme pour calculer la fermeture $(C, X[\vec{E}])^+$ d'un ensemble de chemins C/X. Nous présenterons l'algorithme qui calcule $(C, X[\vec{E}])^+$, et nous allons aussi prouver qu'il calcule bien la fermeture d'un ensemble de chemins. Enfin une analyse de sa complexité sera faite pour démontrer son efficacité.

4.6.1 Algorithme pour la fermeture d'un ensemble de chemins

L'Algorithme 2 prend en entrée un ensemble de XFD $\mathcal{F}$, l'ensemble des chemins simples $\mathbb{P}$, le chemin contexte C et un ensemble de chemins X (tel que $C/X \subseteq \mathbb{P}$) pour lequel on veut calculer la fermeture. L'algorithme se divise en deux phases : une phase d'initialisation et une phase d'itération. Dans chaque phase nous allons calculer l'ensemble de chemins $X^{(i)}$ où i est le numéro d'itération avec $i = 0$ pour la phase d'initialisation. L'ensemble $X^{(i)}$ est le résultat intermédiaire pour $(C, X[\vec{E}])^+$ et chaque ensemble $X^{(i)}$ est calculé à partir de l'ensemble précédent $X^{(i-1)}$; sauf pour la phase d'initialisation.La phase d'initialisation consiste donc à calculer $X^{(0)}$ à partir des ensembles suivants où le type d'égalité prise en compte est aussi stocké :

- S qui contient le chemin contexte ($C/[]$, ligne 1). L'égalité par nœud est considérée ;
- T qui contient les préfixes de chaque chemin dans X (ligne 2). Notons que l'ensemble $X[\vec{E}]$ est aussi ajouté à T car dans le cas de l'égalité par valeur les préfixes d'un chemin ne sont pas ajoutés à T.
- V qui contient les chemins attributs pour des chemins dans T qui sont leur parents (ligne 3) ;
- W qui contient les chemins associés à l'égalité par valeur sachant que ces mêmes chemins avec l'identité de nœud sont dans $S \cup T \cup V$ (ligne 4).

Algorithme 2 *closurePath*$(C, X, \mathcal{F})$: Fermeture d'un ensemble de chemins

Entrée :
- un ensemble fini de chemins simple $\mathbb{P}$
- un ensemble de XFD $\mathcal{F}$
- un chemin contexte C et un ensemble de chemins X tel que $C/X \subseteq \mathbb{P}$
- un vecteur de types d'égalité $\vec{E}$ associé à X

Sortie : L'ensemble de chemins $(C, X[\vec{E}])^+$

1: $S := \{[][N]\}$ **% *grâce à l'axiome A7* %**
2: $T := \{P[N] \mid P \preceq Q \text{ et } Q[N] \in X[\vec{E}]\} \cup X[\vec{E}]$ **% *grâce aux axiomes A1 et A5* %**
3: $V := \{P[N] \mid Q[E'] \in T \cup S, Parent(P) = Q \text{ et } Last(P) \in \Sigma_{att}\}$ **% *grâce à l'axiome A6* %**
4: $W := \{P[V] \mid P[N] \in S \cup T \cup V\}$ **% *grâce à l'axiome A9* %**
5: $X^{(0)} := S \cup T \cup V \cup W$

6: **Tant que** $X^{(i)} \neq X^{(i-1)}$ **faire**
7: $Y := \{P[E_{n+1}] \mid$ il existe une XFD $f \in \mathcal{F}$ où $f = (C, (\{P_1[E_1], \ldots, P_n[E_n]\} \rightarrow P[E_{n+1}]))$ ou $f = (C', (\{Q/P_1[E_1], \ldots, Q/P_n[E_n]\} \rightarrow Q/P[E_{n+1}]))$ avec $C = C'/Q$, telle qu'une des conditions ci-dessous soit satisfaite :
(a) $\{P_1[E_1], \ldots, P_n[E_n]\} \subseteq X^{(i)}$ ou **% *grâce à l'axiome A3* %**
(b) $\exists P'_1[E_1], \ldots, P'_n[E_n] \in X^{(i)}$ tel que $\forall k \in [1, \ldots, n]$, $P_k \cap P \preceq P'_k$ et $(P'_k \preceq P_k$ ou $P'_k \preceq P)$ **% *grâce à l'axiome A4* %**
$\}$
8: $T := \{P[N] \mid P \preceq Q \text{ et } Q[N] \in Y\} \cup Y$ **% *grâce à l'axiome A5* %**
9: $V := \{P[N] \mid Q[E'] \in T, Parent(P) = Q \text{ et } Last(P) \in \Sigma_{att}\}$ **% *grâce à l'axiome A6* %**
10: $W := \{P[V] \mid P[N] \in T \cup V\}$ **% *grâce à l'axiome A9* %**
11: $X^{(i+1)} := X^{(i)} \cup T \cup V \cup W$
12: **fin Tant que**

13: **retourner** $C/X^{(i)}$

La phase d'itération $i\,(>0)$ consiste à calculer $X^{(i)}$ à partir des ensembles suivants :

- $X^{(i-1)}$ qui est le résultat de l'itération précédente $i-1$;
- Y (ligne 7) qui contient tous les chemins P tel que :
  (a) il existe une dépendance f dans $\mathcal{F}$ telle que tous les chemins à gauche de f sont dans $X^{(i-1)}$ et P est à droite de f
  (b) ou il existe une dépendance f dans $\mathcal{F}$ telle qu'un sous ensemble de $X^{(i-1)}$ vérifie les conditions de l'axiome A4 pour les chemins à gauche de f et P est à droite de f
- un nouveau T qui contient les préfixes de chaque chemin dans Y (ligne 8) ;

– un nouveau V qui contient les chemins attributs pour des chemins dans T (ligne 9) ;
– et un nouveau W qui contient les chemins associés à l'égalité par valeur sachant que ces mêmes chemins avec l'identité de nœud sont dans $S \cup T \cup V$ (ligne 10).

L'algorithme s'arrête lorsqu'on ne peut plus rajouter de chemins dans $X^{(i)}$.

Exemple 4.15 Soit l'ensemble de chemins $\mathbb{P}$ qui contient les chemins de l'ensemble $\{C/R/A/I, C/R/B, C/R/H/J, C/R/D/E/@E, C/R/D/F, C/R/D/G\}$ et ses préfixes ; soient les XFD $f_1 = (C/R, (A[V] \rightarrow B[V]))$, $f_2 = (\{C/R, (A/I[N], H/J[N]\} \rightarrow D/E[V]))$, $f_3 = (C, (\{R/D/F[V], R/B[V]\} \rightarrow R/D/G[V]))$ et l'ensemble de XFD $\mathcal{F} = \{f_1, f_2, f_3\}$. Nous allons maintenant calculer $(C/R, X[\vec{E}])^+$ pour $X = \{A/I, H/J, D/F\}$ et $\vec{E} = N, N, V$.

	$X[\vec{E}]$ ou Y	S	T	V	W
$X^{(0)}$	$A/I[N]$, $H/J[N]$, $D/F[V]$	$[][N]$	$A[N]$, $H[N]$		$A/I[V]$, $H/J[V]$, $[][V]$, $A[V]$, $H[V]$
$X^{(1)}$	$B[V]$ (f_1), $D/E[V]$ (f_2)			$D/E/@E[N]$	$D/E/@E[V]$
$X^{(2)}$	$D/G[V]$ (f_3)				

Tableau 4.4 – *Illustration du calcul de la fermeture d'un ensemble de chemins.*

Les étapes du calcul de $(C/R, X[\vec{E}])^+$ sont illustrées dans le Tableau 4.4. L'ensemble $X^{(i)}$ contient les chemins des lignes précédentes en plus de ceux sur sa ligne. Le résultat pour $X = \{A/I, H/J, D/F\}$ est $C/X^{(2)}$ qui contient tous les chemins dans le Tableau 4.4. Remarquons que pour pouvoir utiliser la dépendance f_3, l'axiome A8 a été utilisée pour faire passer le contexte de C à C/R.

⊠

Dans l'Annexe A.6, nous prouvons que l'Algorithme 2 est correct.

4.6.2 Analyse de la complexité

Faisons une analyse de la complexité de l'Algorithme 2. Dans cet algorithme nous pouvons noter que :

1. la boucle à la ligne 6 est exécutée au plus $|\mathbb{P}|$ fois lorsque nous supposons qu'à chaque itération un seul chemin est ajouté à $X^{(i)}$.
2. vérifier que $X^{(i)} \neq X^{(i+1)}$ a eu complexité dans le pire des cas de $O(|\mathbb{P}|^2)$ lorsque $X^{(i)} = X^{(i+1)} = \mathbb{P}$.
3. Pour calculer l'ensemble Y à la ligne 7, nous considérons toutes les XFD dans $\mathcal{F}$ et vérifions pour chacune si l'une des conditions (a)-(b) est satisfaite. La complexité de cette partie est $O(|f_{max}| \cdot |\mathbb{P}| \cdot |\mathcal{F}|)$ où $|\mathcal{F}|$ est la cardinalité de $\mathcal{F}$ et $|fmax|$ est la taille (nombre de chemins dans la XFD) de la plus longue XFD dans $\mathcal{F}$.
4. Pour calculer les préfixes et attributs aux lignes 8 et 9, la complexité dans le pire des cas est $O(|\mathbb{P}|)$ si l'on considère qu'on les cherche dans l'ensemble $\mathbb{P}$.

Nous obtenons donc pour le tout $O(|\mathbb{P}| \cdot (|f_{max}| \cdot |\mathbb{P}| \cdot |\mathcal{F}| + |\mathbb{P}|^2 + |\mathbb{P}|))$, ou après avoir factorisé $O(|\mathbb{P}|^2 \cdot (|f_{max}| \cdot |\mathcal{F}| + |\mathbb{P}| + 1))$. Ainsi la complexité en temps de l'Algorithme 2, dans le pire des cas (peu probable) est $O(|\mathbb{P}|^2 \cdot (|\mathcal{F}| + |\mathbb{P}|))$.

4.6.3 Algorithme pour la fermeture d'un ensemble de dépendances

Du fait que nous avons prouvé que notre système d'axiomes est correct et complet (voir section A.4 et A.5), il en découle qu'on peut aussi définir la fermeture $(C, X[\vec{E}])^+$ comme l'ensemble de tous les chemins $P[E']$ tel que $\mathcal{F} \models (C, (X[\vec{E}] \rightarrow P[E']))$. Une autre conséquence est la suivante : la fermeture d'un ensemble de dépendances $\mathcal{F}^+$ que nous avons défini comme l'ensemble contenant toutes les dépendances XFD qui sont logiquement impliquées par $\mathcal{F}$, peut être considérée comme étant l'ensemble de toutes les dépendances XFD qui sont dérivées par $\mathcal{F}$ en se servant du système d'axiomes.

En prenant en compte cette nouvelle façon de représenter $\mathcal{F}^+$, nous pouvons proposer un algorithme pour calculer $\mathcal{F}^+$ en se servant de l'Algorithme 2 qui calcule la fermeture d'un ensemble de chemins. Pour avoir toutes les dépendances dans $\mathcal{F}^+$, l'algorithme 3 va calculer pour chaque sous-ensemble X de $\mathbb{P}$ sa fermeture X^+. Ensuite les dépendances $X \rightarrow X^+$ seront ajoutées dans $\mathcal{F}^+$. En ayant calculé la fermeture des combinaisons de tous les chemins de $\mathbb{P}$, on est sûr d'avoir toutes les dépendances dérivées par $\mathcal{F}$.

Algorithme 3 *closureXFD*$(\mathcal{F})$: Fermeture d'un ensemble de XFD

Entrée :

- un ensemble fini de chemins simple $\mathbb{P}$
- un ensemble de XFD $\mathcal{F}$

Sortie : L'ensemble de XFD $\mathcal{F}^+$

```
1: Fplus := ∅
2: pour tout sous-ensemble X de ℙ faire
3:     pour tout chemin contexte C dans ℱ faire
4:         si le chemin C est préfixe de tous les chemins dans X alors
5:             Xplus := closurePath(C, X, ℱ)
6:             Fplus := Fplus ∪ {(C, (X → Xplus))}
7:         fin si
8:     fin pour
9: fin pour
10: retourner Fplus
```

Soit n égal au nombre de chemins simples dans $\mathbb{P}$. Le nombre de sous-ensemble de $\mathbb{P}$ est de 2^n ce qui fait que le nombre de XFD dans $\mathcal{F}^+$ est de l'ordre de 2^n. Du fait que le nombre de dépendances dans $\mathcal{F}^+$ est exponentiel par rapport à n, le calcul de $\mathcal{F}^+$ est très coûteux même pour une petite valeur de n. La complexité en temps de l'Algorithme 3 est : $O(2^n \cdot |\mathbb{P}|^2 \cdot (|\mathcal{F}| + |\mathbb{P}|))$, c'est-à-dire l'Algorithme 3 est exponentiel en fonction de la taille de $\mathbb{P}$.

4.6.4 Notion de couverture

Définition 4.23 (Ensembles équivalents et couverture)
Soient $\mathcal{F}_1$, et $\mathcal{F}_2$ deux ensembles de XFD. Nous disons que $\mathcal{F}_1$ est équivalent à $\mathcal{F}_2$, noté $\mathcal{F}_1 \equiv \mathcal{F}_2$ si et seulement si on a $\mathcal{F}_1^+ = \mathcal{F}_2^+$. Nous disons aussi que $\mathcal{F}_1$ est une *couverture* de $\mathcal{F}_2$ lorsque nous avons $\mathcal{F}_1 \equiv \mathcal{F}_2$.

□

Pour montrer qu'un ensemble $\mathcal{F}_1$ est une couverture d'un autre ensemble $\mathcal{F}_2$ ($\mathcal{F}_1 \equiv \mathcal{F}_2$), on vérifie que $\mathcal{F}_1^+ = \mathcal{F}_2^+$ c'est à dire :

- pour toute XFD $f \in \mathcal{F}_1$ on vérifie que $\mathcal{F}_2 \vdash f$. Ceci montre que $\mathcal{F}_1^+ \subseteq \mathcal{F}_2^+$;

– pour toute XFD $f \in \mathcal{F}_2$ on vérifie que $\mathcal{F}_1 \vdash f$. Ceci montre que $\mathcal{F}_2^+ \subseteq \mathcal{F}_1^+$.

Avec ces deux inclusions on a bien $\mathcal{F}_1^+ = \mathcal{F}_2^+$. A la fin de la section 4.5.4, nous proposons un algorithme qui se sert de l'Algorithme 2 pour tester si un ensemble $\mathcal{F}$ dérive une dépendance f. Vérifier que $\mathcal{F}_1^+ = \mathcal{F}_2^+$ en se servant de l'Algorithme 2 sera beaucoup plus efficace que de calculer $\mathcal{F}_1^+$ et $\mathcal{F}_2^+$ à l'aide de l'Algorithme 3 et de les comparer ensuite.

4.7 Dépendances Fonctionnelles pour l'interopérabilité

Étant donné des ensembles de XFD $\mathcal{F}_1, \cdots, \mathcal{F}_n$, nous voulons calculer le plus grand ensemble de dépendances fonctionnelles $\mathcal{F}$ tel que :

$$\text{pour tout document } T, (\exists\, i \in [1, \ldots, n], T \models \mathcal{F}_i) \Longrightarrow T \models \mathcal{F}.$$

Autrement dit, tout document XML qui satisfait les dépendances d'un système local doit satisfaire aussi les dépendances du système global. Avoir $\mathcal{F}$ comme ensemble de XFD au niveau de notre système global, va nous permettre d'assurer la cohérence entre les données XML se trouvant au niveau des systèmes locaux. Pour remplir cette condition, l'ensemble $\mathcal{F}$ doit être maximal, c'est à dire qu'il ne sera donc pas possible de trouver une dépendance XFD (i) qui peut être engendrée par tous ou une partie parmi les ensembles $\mathcal{F}_1, \cdots, \mathcal{F}_n$; (ii) qui est satisfaite par tous les documents XML dans $X_1, \cdots, X_n$; et (iii) qui ne soit pas dans $\mathcal{F}$.

Pour des raisons de simplicité, nous allons supposer que nous avons deux sources locales mais notre méthode peut être facilement adaptée pour n sources locales. Dans le cas où $n = 2$, nous disposons donc de deux sources locales $S_1 = (\mathcal{D}_1, \mathcal{F}_1, X_1)$ et $S_2 = (\mathcal{D}_2, \mathcal{F}_2, X_2)$ où D_k est le schéma, $\mathcal{F}_k$ est l'ensemble des XFD et X_k est l'ensemble des documents XML. Il est possible que la manière de représenter les données dans chaque système ne soit pas forcément la même. Une même information peut être représentée différemment selon qu'on est dans le système S_1 ou qu'on est dans le système S_2. Prenons par exemple la Figure 4.8(a) qui illustre l'ensemble des chemins simples $\mathbb{P}_1$ pour le schéma $\mathcal{D}_1$, et la la Figure 4.8(b) qui illustre l'ensemble des chemins simples $\mathbb{P}_2$ pour le schéma $\mathcal{D}_2$. Dans le schéma de la Figure 4.8(a), les publications sont organisées comme une collection de papiers où pour chaque papier on connaît les informations sur le titre, la conférence dans laquelle il est publié, l'année de publication, le nombre de pages et les auteurs. Par contre dans le schéma de la Figure 4.8(b), les publications sont organisées comme une collection de conférences où pour chaque conférence on connaît son nom, l'année et la ville où elle a lieu, la liste des papiers avec leur titre, nombre de pages et auteurs. Comme nous pouvons le constater, tous les deux schémas représentent les mêmes données (des publications) mais la façon dont elles sont organisées n'est pas pareille. Aussi un schéma possède des informations qu'un autre n'a pas. Par exemple comme information commune, le chemin $pub/paper/authors$ dans $\mathbb{P}_1$ représente la même information que le chemin $pub/conf/paper/writers$ à savoir les auteurs ; et comme information diverse l'ensemble $\mathbb{P}_2$ contient le chemin $pub/conf/@city$ alors que son chemin équivalent n'est pas dans $\mathbb{P}_1$.

Pour résoudre ce problème de sémantique entre les deux sources S_1 et S_2, nous allons supposer que la table de correspondance (Tableau 4.5) entre les chemins des deux schémas D_1 et D_2 nous a été donnée. Dans cette table, chaque chemin de l'ensemble $\mathbb{P}_1$ est associé à son chemin équivalent dans $\mathbb{P}_2$ et vice-verca. Cela va donc permettre de savoir qu'une donnée dans D_1 représente la même information qu'une autre donnée dans D_2. A partir du Tableau 4.5 nous définissons deux fonctions de traductions Φ_1 et Φ_2 :

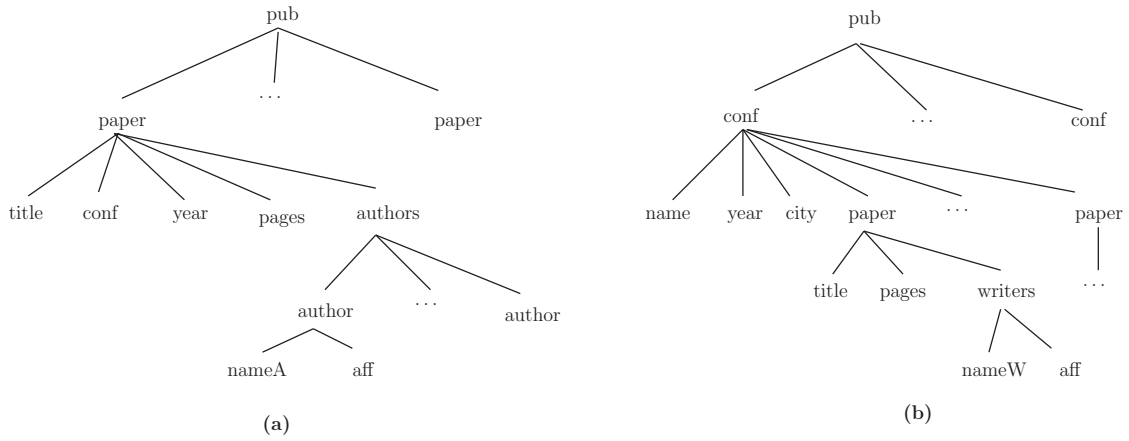


FIGURE 4.8 – *Deux schémas $\mathcal{D}_1$ (a) et D_2 (b) représentant les mêmes données de façon différentes.*

- la fonction Φ_1 est définie de $\mathbb{P}_2 \rightarrow \mathbb{P}_1 \cup \mathbb{P}_2$: étant donné le chemin $P \in \mathbb{P}_2$, $\Phi_1(P)$ donne son chemin équivalent dans $\mathbb{P}_1$ s'il existe ; sinon il renvoie P.
- la fonction Φ_2 est définie de $\mathbb{P}_1 \rightarrow \mathbb{P}_1 \cup \mathbb{P}_2$: étant donné le chemin $P \in \mathbb{P}_1$, $\Phi_2(P)$ donne son chemin équivalent dans $\mathbb{P}_2$ s'il existe ; sinon il renvoie P.

Dans la suite nous allons utiliser k et $\overline{k}$ pour indiquer deux sources qui sont symétriques (par exemple lorsque $k = 1$, $\overline{k} = 2$ et lorsque $k = 2$, $\overline{k} = 1$).

Chemin dans $\mathbb{P}_1$	Chemin dans $\mathbb{P}_2$
pub/paper/title	*pub/conf/paper/title*
pub/paper/year	*pub/conf/year*
pub/paper/conf	*pub/conf/name*
pub/paper/authors	*pub/conf/paper/writers*
pub/paper/year	*pub/conf/paper/year*

Tableau 4.5 – *Tableau de correspondances entre les chemins de $\mathbb{P}_1$ et $\mathbb{P}_2$.*

Exprimer la sémantique entre deux schémas est un vrai problème, qui est surtout posé lors de l'interopérabilité entre deux systèmes. Il faut permettre aux deux systèmes de pouvoir parler le même langage pour qu'il puissent se comprendre lors de leurs échanges. Il existe plein de travaux dans ce sens dans la littérature et diverses solutions ont été proposées, comme les outils Coma 2.0/3.0, XClust [79], XS3 [116], Cupid, AgreementMaker. Les auteurs de l'article [3] font un panorama sur les différentes techniques utilisées pour faire le matching entre schémas et proposent un comparatif entre ces techniques.

4.7.1 Méthode pour calculer l'ensemble maximal de XFD $\mathcal{F}$

A partir de deux ensembles de XFD $\mathcal{F}_1$, $\mathcal{F}_2$ nous souhaitons calculer le plus grand ensemble de XFD $\mathcal{F}$ pour permettre la cohabitation entre les documents XML qui vérifient $\mathcal{F}_1$ et ceux qui vérifient $\mathcal{F}_2$ sans qu'il n'y ait de contradiction entre les documents. L'objectif est donc de faire une intégration des dépendances XFD au niveau d'un système global qui centralise les données provenant de plusieurs systèmes locaux.

L'Algorithme 4 permet d'obtenir le plus grand ensemble de dépendances fonctionnelles en se basant sur la démarche suivante :

1. calculer $\mathcal{F}_1^+$ et $\mathcal{F}_2^+$ pour avoir toutes les XFD qu'on peut déduire à partir de $\mathcal{F}_1$ et $\mathcal{F}_2$;

2. ensuite chaque dépendance f dans $\mathcal{F}_k^+$ ($k \in [1,2]$) est analysée pour savoir si elle peut être violée par les documents provenant de la source symétrique $S_{\overline{k}}$. Cette vérification n'est pas faite au niveau des documents XML mais plutôt au niveau de l'ensemble $\mathcal{F}_{\overline{k}}$. La dépendance $f = (C, (X \rightarrow B))$ est analysée et ajoutée à $\mathcal{F}$ lorsqu'une des conditions ci-dessous est satisfaite :
   (a) il n'existe pas de chemin dans $\mathbb{P}_{\overline{k}}$ équivalent au chemin de la partie droite C/B de f (ligne 4). Dans ce cas les documents de $S_{\overline{k}}$ ne peuvent pas violer $f \in \mathcal{F}_k^+$. L'information que porte le chemin C/B n'est pas présente dans les documents de $S_{\overline{k}}$ ce qui fait que la dépendance f ne les concerne pas ;
   (b) il n'existe pas d'ensemble de chemins dans $\mathbb{P}_{\overline{k}}$ équivalent aux chemins de la partie gauche C/X de f (ligne 6). Les informations que portent les chemins dans C/X ne sont pas présentes dans les documents de $S_{\overline{k}}$ ce qui fait que la dépendance f ne les concerne pas. On est donc dans la même situation qu'au (a), les documents de $S_{\overline{k}}$ ne peuvent pas violer $f \in \mathcal{F}_k^+$;
   (c) lorsque les deux conditions (a) et (b) ne sont pas vérifiées, cela veut dire que les informations que portent les chemins dans C/X et C/B sont présentes dans les documents de $S_{\overline{k}}$. Dans ce cas on va vérifier si $\mathcal{F}_{\overline{k}}$ peut engendrer f (ligne 8). Si c'est le cas alors f est satisfaite par tous les documents de $S_{\overline{k}}$ et donc peut être ajoutée à $\mathcal{F}$. Dans le cas contraire il peut exister des documents de $S_{\overline{k}}$ qui violent f et donc on a une incohérence entre les documents des deux sources. La dépendance f est alors à écarter.

   La conclusion qu'on peut tirer est la suivante : si f peut être violée par certains documents XML alors f ne peut pas être dans $\mathcal{F}$, sinon f est ajoutée à $\mathcal{F}$.

Algorithme 4 *integrationXFD*$(\mathcal{F}_1, \mathcal{F}_2)$: XFD assurant l'interoperabilité de S_1 et S_2

Entrée :
- Ensemble de XFD $\mathcal{F}_1$ pour le schéma $\mathcal{D}_1$
- Ensemble de XFD $\mathcal{F}_2$ pour le schéma $\mathcal{D}_2$
- L'ensemble des chemins $\mathbb{P}_1$ obtenus à partir de $\mathcal{D}_1$
- L'ensemble des chemins $\mathbb{P}_2$ obtenus à partir de $\mathcal{D}_2$
- Fonctions de traductions Φ_1 et Φ_2

Sortie : L'ensemble des XFD $\mathcal{F}$ pour le système global S

1: $\mathcal{F} := \emptyset$
2: **pour** $k := 1$ **à** 2 **faire**
3: **pour tout** $(C, (X \rightarrow B)) \in \mathcal{F}_k^+$ **faire**
4: **si** $\Phi_{\overline{k}}(C/B) \notin \mathbb{P}_{\overline{k}}$ **alors**
5: $\mathcal{F} := \mathcal{F} \cup \{(C, (X \rightarrow B))\}$
6: **sinon si** $\Phi_{\overline{k}}(C/X) \not\subseteq \mathbb{P}_{\overline{k}}$ **alors**
7: $\mathcal{F} := \mathcal{F} \cup \{(C, (X \rightarrow B))\}$
8: **sinon si** $\Phi_{\overline{k}}(C/B) \in \Phi_{\overline{k}}(C, X)^+_{\mathcal{F}_{\overline{k}}}$ **alors**
9: $\mathcal{F} := \mathcal{F} \cup \{(C, (X \rightarrow B))\}$
10: **fin si**
11: **fin pour**
12: **fin pour**
13: **retourner** $\mathcal{F}$

Le schéma de notre application pour calculer $\mathcal{F}$ est illustrée dans la Figure 4.9. Le « matching » entre les schémas est assuré par les outils comme Coma ou XClust dédiés à cela ou un humain. Le tableau de correspondance entre les chemins des schémas est le résultat de la phase de « matching ». Notre système d'axiomes que nous avons prouvé correct et complet est utilisé pour raisonner sur les XFD. L'application reçoit donc les ensembles de XFD $\mathcal{F}_1$ et $\mathcal{F}_2$, applique l'Algorithme 4 et nous donne en sortie l'ensemble $\mathcal{F}$. Nous allons maintenant prouver que l'ensemble $\mathcal{F}$ calculé par l'Algorithme 4 est bien l'ensemble que nous souhaitons.

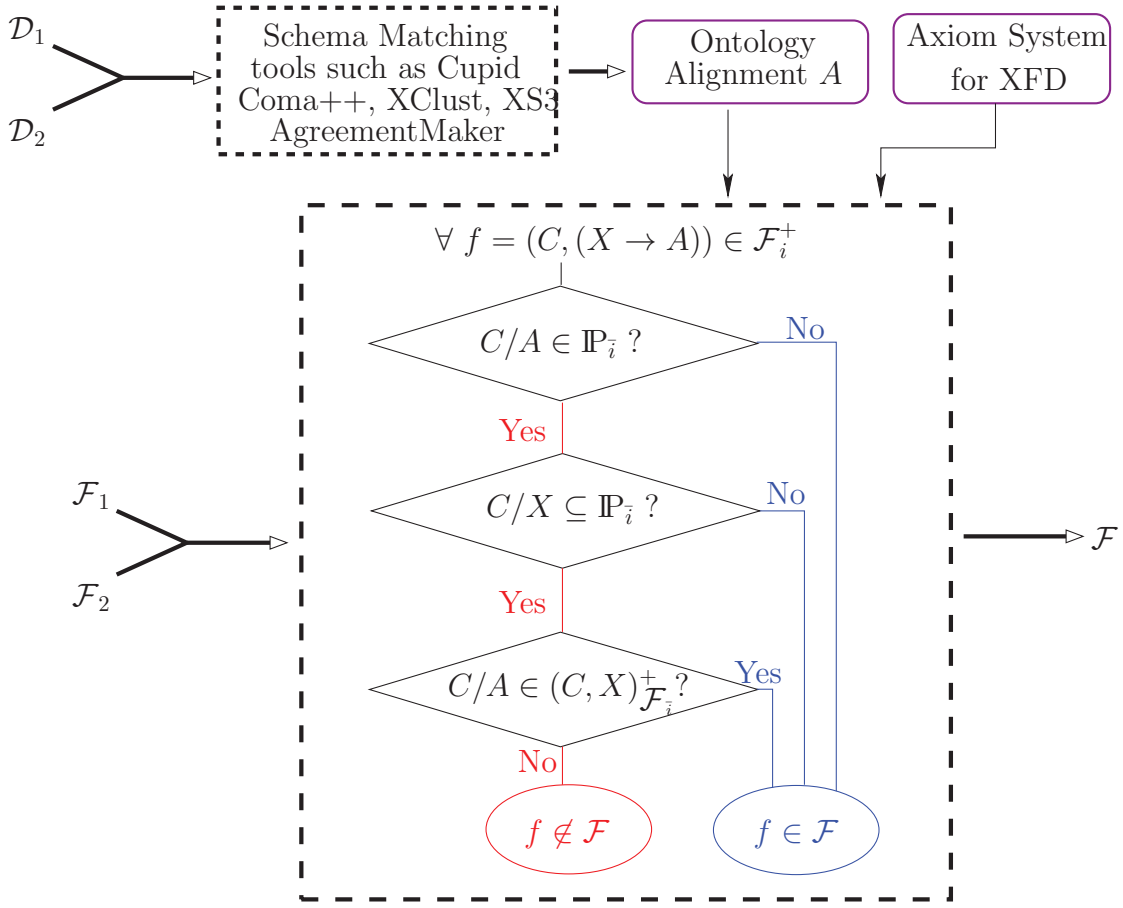


FIGURE 4.9 – *Schéma de l'application pour calculer $\mathcal{F}$.*

Définition 4.24 (XFD concernant un ensemble et pas un autre)
Soient $\mathcal{F}_1$ un ensemble de XFD défini sur l'ensemble des chemins $\mathbb{P}_1$, et $\mathcal{F}_2$ un ensemble de XFD défini sur l'ensemble des chemins $\mathbb{P}_2$. Nous définissons l'ensemble des XFD concernant $\mathcal{F}_1$ et pas $\mathcal{F}_2$, noté $\mathcal{K}_{\mathcal{F}_1/\mathcal{F}_2}$, l'ensemble suivant :

$$\mathcal{K}_{\mathcal{F}_1/\mathcal{F}_2} = \{(C, (X \to A)) \mid (C, (X \to A)) \in \mathcal{F}_1^+ \text{ et } [C/X \subseteq (\mathbb{P}_1 \setminus \mathbb{P}_2) \text{ ou } C/A \in (\mathbb{P}_1 \setminus \mathbb{P}_2)]\}$$

□

Intuitivement pour deux ensembles $\mathcal{F}_1$ et $\mathcal{F}_2$, $\mathcal{K}_{\mathcal{F}_1/\mathcal{F}_2}$ contient toutes les XFD f qu'on peut engendrer à partir de $\mathcal{F}_1$ et qui ne peuvent pas être violées par les documents dans X_2 pour l'une des raisons suivantes :

1. le chemin droit de f est un chemin dans $\mathbb{P}_1$ mais qui n'est pas $\mathbb{P}_2$ ou

2. les chemins dans la partie gauche de f sont des chemins dans $\mathbb{P}_1$ mais qui ne sont pas $\mathbb{P}_2$.

Nous allons prouver que $\mathcal{F}$ est un ensemble dans $\mathcal{F}_1^+ \cup \mathcal{F}_2^+$ qui contient les dépendances caractérisées par le théorème ci-dessous :

Théorème 4.1 *Soient $\mathcal{F}_1$, $\mathcal{F}_2$ deux ensembles de XFD. L'ensemble $\mathcal{F}$ retourné par la fonction integrationXFD$(\mathcal{F}_1, \mathcal{F}_2)$ (Algorithme 4), est tel que :*

$$\mathcal{F} = (\mathcal{F}_1^+ \cap \mathcal{F}_2^+) \cup \mathcal{K}_{\mathcal{F}_1/\mathcal{F}_2} \cup \mathcal{K}_{\mathcal{F}_2/\mathcal{F}_1}$$

Démonstration. La preuve est triviale en suivant les étapes de l'Algorithme 4. Nous retrouvons le calcul de :

- $(\mathcal{F}_1^+ \cap \mathcal{F}_2^+)$ entre les lignes 8 et 10
- $\mathcal{K}_{\mathcal{F}_1/\mathcal{F}_2}$ entre les lignes 4 et 7 lorsque k vaut 1
- $\mathcal{K}_{\mathcal{F}_2/\mathcal{F}_1}$ entre les lignes 4 et 7 lorsque k vaut 2

◁

Il reste à prouver que $\mathcal{F}$ est le plus grand ensemble qu'on peut avoir pour le système global S dans l'ensemble $\mathcal{F}_1^+ \cup \mathcal{F}_2^+$.

Théorème 4.2 *Soient $\mathcal{F}_1$, $\mathcal{F}_2$ deux ensembles de XFD tels que $\mathcal{F}_1$ est défini sur les documents de X_1 et $\mathcal{F}_2$ est défini sur les documents de X_2. L'ensemble $\mathcal{F}$ retourné par la fonction integrationXFD$(\mathcal{F}_1, \mathcal{F}_2)$ (Algorithme 4), est le plus grand sous-ensemble de $\mathcal{F}_1^+ \cup \mathcal{F}_2^+$ tel que, sans prendre en compte les données, nous pouvons garantir que $X_1 \models \mathcal{F}$ et $X_2 \models \mathcal{F}$.*

Démonstration. Prouver que $X_1 \models \mathcal{F}$ et $X_2 \models \mathcal{F}$ est très facile. Nous savons que $X_1 \models \mathcal{F}_1$ et par conséquent $X_1 \models \mathcal{F}_1^+$. Et comme $(\mathcal{F}_1^+ \cap \mathcal{F}_2^+) \subseteq \mathcal{F}_1^+$ alors $X_1 \models (\mathcal{F}_1^+ \cap \mathcal{F}_2^+)$. Nous savons aussi que $\mathcal{K}_{\mathcal{F}_1/\mathcal{F}_2} \subseteq \mathcal{F}_1^+$ ce qui a pour conséquence $X_1 \models \mathcal{K}_{\mathcal{F}_1/\mathcal{F}_2}$. Comme les dépendances qui sont dans $\mathcal{K}_{\mathcal{F}_2/\mathcal{F}_1}$ n'ont rien avoir avec les documents de X_1 alors on a $X_1 \models \mathcal{K}_{\mathcal{F}_2/\mathcal{F}_1}$. Nous pouvons donc conclure que $X_1 \models \mathcal{F}$. Avec un raisonnement similaire, on obtient $X_2 \models \mathcal{F}$.

Ensuite nous allons prouver par contradiction que $\mathcal{F}$ est le plus grand sous-ensemble de $\mathcal{F}_1^+ \cup \mathcal{F}_2^+$ assurant la propriété du théorème. Supposons qu'il existe un ensemble XFD G inclus dans $\mathcal{F}_1^+ \cup \mathcal{F}_2^+$ tel que $\mathcal{F} \subset G$ et $X_k \models G$ pour $k = 1, 2$. Soit $f = (C, (X \to A))$ une XFD dans G qui n'est pas dans $\mathcal{F}$. Comme $(C, (X \to A)) \notin \mathcal{F}$ et que $(C, (X \to A)) \in \mathcal{F}_1^+ \cup \mathcal{F}_2^+$, nous savons que :

1. C/A est dans $\mathbb{P}_1$ et C/A est dans $\mathbb{P}_2$; sinon f aurait été ajouté à $\mathcal{F}$ à l'étape 4 dans l'Algorithme 4 ;
2. C/X est inclus dans $\mathbb{P}_1$ et $\mathbb{P}_2$; sinon f aurait été ajouté à $\mathcal{F}$ à l'étape 6 dans l'Algorithme 4 ;
3. $C/A \in X^+_{\mathcal{F}_1}$ et $C/A \notin X^+_{\mathcal{F}_2}$ ou l'inverse ; sinon f aurait été ajouté à $\mathcal{F}$ à l'étape 8 dans l'Algorithme 4 ;

D'après le cas 3, nous savons que $(C, (X \to A)) \in \mathcal{F}_1^+$ mais $(C, (X \to A)) \notin \mathcal{F}_2^+$. Dans cette situation, d'après les cas 1-2 ci-dessus, les documents dans X_2 peuvent violer $(C, (X \to A))$. Ceci est donc une contradiction avec notre hypothèse qui nous dit qu'il est toujours vrai que $X_i \models G$ pour $k = 1, 2$.

◁

Nous pouvons prouver le résultat suivant, découlant du Théorème 4.2 et qui positionne la fermeture de l'ensemble $\mathcal{F}$ par rapport à $\mathcal{F}_1^+ \cup \mathcal{F}_2^+$. Une illustration du Corollaire 4.1 est faite dans la Figure 4.10.

Corollaire 4.1 $\mathcal{F}^+ \cap (\mathcal{F}_1^+ \cup \mathcal{F}_2^+) = \mathcal{F}$.

Démonstration. Soit $(C, (X \rightarrow A))$ une XFD dans $\mathcal{F}^+$ qui n'est pas $\mathcal{F}$. Ainsi $(C, (X \rightarrow A))$ n'est pas non plus dans $\mathcal{F}_1^+ \cup \mathcal{F}_2^+$. En effet si $(C, (X \rightarrow A))$ était dans $\mathcal{F}_1^+ \cup \mathcal{F}_2^+$, d'après le Théorème 4.2, il ne serait pas toujours possible d'avoir $X_k \models (C, (X \rightarrow A))$. Cette situation est impossible car $(C, (X \rightarrow A)) \in \mathcal{F}^+$ et nous avons l'assurance que $X_k \models \mathcal{F}$ pour $k = 1, 2$. ◁

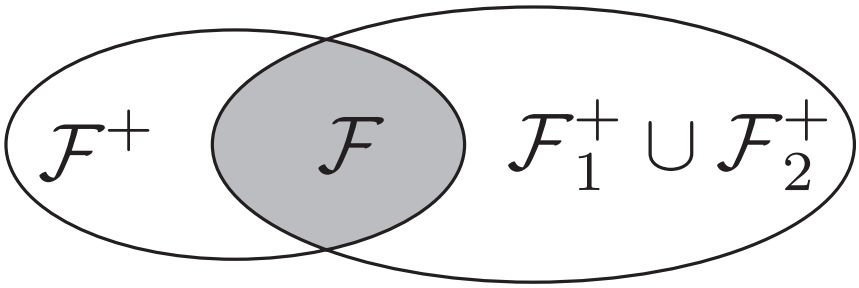


FIGURE 4.10 – *Résultat du Corollaire 4.1.*

4.7.2 Vers une version optimisée

L'Algorithme 4 dépend de deux principales fonctions : l'Algorithme 3 qui calcule la fermeture d'un ensemble de XFD ($\mathcal{F}^+$) et l'Algorithme 2 qui calcule la fermeture d'un ensemble de chemins ($(C, X)^+$). Comme prévu c'est le calcul des $\mathcal{F}_k^+$ ($k \in [1, 2]$) qui prend la majeure partie du temps nécessaire pour l'Algorithme 4. Ainsi l'Algorithme 4 analyse un nombre exponentiel de XFD (créées par l'Algorithme 3) ce qui fait que sa complexité est exponentielle par rapport à la taille des $\mathbb{P}_k$.

Il est important de noter que nous ne pouvons pas juste remplacer $\mathcal{F}_i^+$ par $\mathcal{F}_i$ à la ligne 3 dans l'Algorithme 4. Pour comprendre cela, considérons les ensembles $\mathcal{F}_1$ et $\mathcal{F}_2$ à partir desquels on peut engendrer la dépendance f par différentes séquences de dérivation. Supposons que pour $\mathcal{F}_1$ nous avons $f_1, \ldots, f_k, \ldots, f$ tandis que pour $\mathcal{F}_2$ nous avons $f'_1, \ldots, f'_k, \ldots, f$. De plus supposons aussi qu'à cause des conditions des lignes 4, 6 et 8, les dépendances f_k et f'_k ne sont pas ajoutées dans $\mathcal{F}$ et donc la dérivation de f ne sera pas possible par $\mathcal{F}$ retourné par l'Algorithme 4. Ce serait donc une grosse erreur que $\mathcal{F} \not\models f$, puisque f est dérivée par $\mathcal{F}_1$ et $\mathcal{F}_2$. C'est pour cela que l'Algorithme 4 prend en compte toutes les XFD dans la fermeture de $\mathcal{F}_1$ et $\mathcal{F}_2$. Mais cette solution implique la génération d'un ensemble avec un très grand nombre de XFD et qui n'est pas manipulable. Pour pouvoir améliorer l'Algorithme 4 nous allons proposer une autre méthode qui utilise les ensembles $\mathcal{F}_k$ au lieu de $\mathcal{F}_k^+$ mais qui traite plus finement les cas où une dépendance n'est pas ajoutée à $\mathcal{F}$ afin d'éviter que $\mathcal{F}$ ne puisse pas engendrer d'autres dépendances que les ensembles $\mathcal{F}_1$ et $\mathcal{F}_2$ dérivent. La complexité de la nouvelle méthode n'aura pas le facteur 2^n ($n = |\mathbb{P}_k|$) qui est présent dans celui de l'Algorithme 4. En conclusion, la nouvelle méthode va s'exécuter plus rapidement et produira un ensemble avec beaucoup moins de XFD, et qui sera exploitable.

4.7.3 Méthode pour calculer une couverture de $\mathcal{F}$

Dans cette section nous présentons l'Algorithme 5 qui génère l'ensemble de XFD *cover*$\mathcal{F}$ équivalent au plus grand ensemble de XFD $\mathcal{F}$ calculé par l'Algorithme 4. Elle prend aussi en paramètre deux ensembles de XFD $\mathcal{F}_1$ et $\mathcal{F}_2$.

Avant de comprendre le fonctionnement de l'Algorithme 5, nous allons définir deux sous-ensembles de la fermeture d'un ensemble de chemins. Pour un ensemble de chemins, nous définissons une étape de la fermeture en faisant un pas vers l'avant ou un pas vers l'arrière.

Définition 4.25 (Un pas de la fermeture d'un ensemble de chemins)
Soit X un ensemble de chemins et C un chemin simple qui sera le contexte. Soit $\vec{E}$ le vecteur de types d'égalité associé à l'ensemble X. La fermeture en un seul pas de (C, X) par rapport à $\mathcal{F}$, notée $(C, X[\vec{E}])^{(1)}_{\mathcal{F}}$, est l'ensemble de chemins $C/\{P_1[E'_1], \ldots, P_n[E'_n]\}$ tel que chaque dépendance $(C, (X[\vec{E}] \rightarrow P_i[E'_i]))$, $i \in [1, \cdots, n]$ peut être engendrée à partir de $\mathcal{F}$ avec une séquence de dérivation α_i et la longueur de la sous-séquence $\alpha_i(\mathcal{F})$ est d'au plus 1. Formellement, $(C, X[\vec{E}])^{(1)}_{\mathcal{F}} = \{C/P[E'] \mid \mathcal{F} \vdash (C, (X[\vec{E}] \rightarrow P[E'])) \textit{ est obtenue avec une séquence de dérivation } \alpha_i, \textit{ et la longueur de la sous-séquence } \alpha_i(\mathcal{F}) \textit{ est d'au plus 1}\}$.
Lorsqu'il n'y a pas d'ambiguïté sur l'ensemble $\mathcal{F}$ utilisé, on le note $(C, X[\vec{E}])^{(1)}$.

□

Définition 4.26 (Un pas de la fermeture inverse d'un ensemble de chemins)
Soit X un ensemble de chemins et C un chemin simple qui sera le contexte. Soit $\vec{E}$ le vecteur de types d'égalité associé à l'ensemble X. La fermeture inverse en un seul pas de (C, X) par rapport à $\mathcal{F}$, notée $(C, X[\vec{E}])^{(-1)}_{\mathcal{F}}$, est l'ensemble minimal d'ensemble de chemins $\{C/Y_1[\vec{E'_1}], \ldots, C/Y_n[\vec{E'_n}]\}$ tel que $C/X[\vec{E}]$ soit inclus dans chaque ensemble $(C, Y_1[\vec{E'_1}])^{(1)}, \cdots, (C, Y_n[\vec{E'_n}])^{(1)}$. Formellement, $(C, X[\vec{E}])^{(-1)}_{\mathcal{F}} = \{C/Y[\vec{E'}] \mid C/X[\vec{E}] \in (C, Y[\vec{E'}])^{(1)} \textit{ et il n'existe pas } C/Y'[\vec{E''}] \in (C, X[\vec{E}])^{(-1)}_{\mathcal{F}} \textit{ tel que } C/Y'[\vec{E''}] \subset C/Y[\vec{E'}]\}$.
Lorsqu'il n'y a pas d'ambiguïté sur l'ensemble $\mathcal{F}$ utilisé, on le note $(C, X[\vec{E}])^{(-1)}$.

□

Nous allons maintenant expliquer le fonctionnement de l'Algorithme 5. L'Algorithme 5 utilise le même raisonnement que dans l'Algorithme 4 en analysant les XFD des ensembles $\mathcal{F}_k$ ($k \in [1, 2]$) au lieu de celles des ensembles $\mathcal{F}^+_k$ ($k \in [1, 2]$). Chaque dépendance $f = (C, (X \rightarrow B))$ dans $\mathcal{F}_k$ est analysée et ajoutée à $cover\mathcal{F}$ selon les conditions établies aux points (2a)-(2c) au début de la section 4.7.1 (Page 65). Nous retrouvons la vérification de ces conditions aux lignes 7 à 12 dans l'Algorithme 5. Si ces conditions sont vérifiées, c'est à dire f ne viole pas de documents du système symétrique, alors f est ajoutée à $cover\mathcal{F}$ sinon un traitement spécial est fait à la dépendance f.

Aux lignes 15 à 20, l'Algorithme 5 prend en compte le fait qu'en travaillant avec $\mathcal{F}_k$, certaines XFD dans $\mathcal{F}^+_k$ peuvent être oubliées. L'exemple 4.16 illustre bien cette situation qui est expliquée dans la section 4.7.2.

Exemple 4.16 Soient $\mathcal{F}_1 = \{(C, (A \rightarrow B)), (C, (B \rightarrow M)), (C, (M \rightarrow D)), (C, (D \rightarrow E)), (C, (O \rightarrow Z)), (C, (\{F, D\} \rightarrow E))\}$ et $\mathcal{F}_2 = \{(C, (A \rightarrow B)), (C, (B \rightarrow M)), (C, (B \rightarrow O)), (C, (O \rightarrow E)), (C, (D \rightarrow N)), (C, (\{F, M\} \rightarrow O))\}$. Sans les lignes 15-20 de l'Algorithme 5, les dépendances $(C, (A \rightarrow E))$ et $(C, (\{F, M\} \rightarrow E))$ dérivées par $\mathcal{F}_1$ et $\mathcal{F}_2$, ne seront pas dérivées par $cover\mathcal{F}$ qui sera égal à l'ensemble $\{(C, (A \rightarrow B)), (C, (B \rightarrow M)), (C, (O \rightarrow Z)), (C, (D \rightarrow N))\}$.

⊠

Comme nous pouvons le constater dans l'exemple 4.16, puisque certaines dépendances de $\mathcal{F}_1$ et $\mathcal{F}_2$ ne peuvent pas être ajoutées à $cover\mathcal{F}$, $cover\mathcal{F}$ ne peut pas engendrer $(C, (A \rightarrow$

Algorithme 5 *integrationXFDv*$_2(\mathcal{F}_1, \mathcal{F}_2)$: XFD assurant l'interoperabilité de S_1 et S_2

Entrée :

- Ensemble de XFD $\mathcal{F}_1$ pour le schéma $\mathcal{D}_1$
- Ensemble de XFD $\mathcal{F}_2$ pour le schéma $\mathcal{D}_2$
- L'ensemble des chemins $\mathbb{P}_1$ obtenus à partir de $\mathcal{D}_1$
- L'ensemble des chemins $\mathbb{P}_2$ obtenus à partir de $\mathcal{D}_2$
- Fonctions de traductions Φ_1 et Φ_2

Sortie : L'ensemble des XFD *cover*$\mathcal{F}$ pour le système global S

1: *cover*$\mathcal{F} := \emptyset$
2: **pour** $k := 1$ **à** 2 **faire**
3: $G_1 := \mathcal{F}_k$ **% $\mathbf{G_1}$ *contient les XFD à analyser* %**
4: $G_2 := \emptyset$ **% $\mathbf{G_2}$ *contient les XFD à écarter de* *cover*$\mathcal{F}$ %**
5: $G_3 := \emptyset$ **% $\mathbf{G_3}$ *contient les XFD à ajouter à* *cover*$\mathcal{F}$ %**
6: **pour tout** $(C, (X \rightarrow B)) \in G_1$ **faire**
7: **si** $\Phi_{\overline{k}}(C/B) \notin \mathbb{P}_{\overline{k}}$ **alors**
8: $G_3 := G_3 \cup \{(C, (X \rightarrow B))\}$
9: **sinon si** $\Phi_{\overline{k}}(C/X) \not\subseteq \mathbb{P}_{\overline{k}}$ **alors**
10: $G_3 := G_3 \cup \{(C, (X \rightarrow B))\}$
11: **sinon si** $\Phi_{\overline{k}}(C/B) \in \Phi_{\overline{k}}(C, X)^{+}_{\mathcal{F}_{\overline{k}}}$ **alors**
12: $G_3 := G_3 \cup \{(C, (X \rightarrow B))\}$
13: **sinon**
14: **% *une XFD* f *est ajoutée à* $\mathbf{G_1}$ *seulement si* f $\notin \mathbf{G_1} \cup \mathbf{G_2} \cup \mathbf{G_3}$ %**
15: $H :=$ **closure1Step**$(C, B, G_1 \cup G_3) \setminus \{C/B\}$
16: $G_1 := G_1 \cup \{(C, (X \rightarrow D)) \mid C/D \in H\}$
17: $K :=$ **inverseClosure1Step**$(C, X, G_1 \cup G_3) \setminus \{C/X\}$
% *Rappelons que* K *est un ensemble d'ensembles de chemins* %
18: $G_1 := G_1 \cup \{(C, (Y \rightarrow B)) \mid C/Y \in K\}$
19: $G_1 := G_1 \cup \{(C, (Z \rightarrow B)) \mid (C, (Z \rightarrow B))$ est obtenue en utilisant l'axiome $A4$ sur $(C, (X \rightarrow B))\}$ **% Z *contient les préfixes des chemins de* X *ou* B %**
20: $G_1 := G_1 \cup \{(C, (X, W \rightarrow V)) \mid (C, (X, W \rightarrow V))$ est obtenue en utilisant l'axiome $A12$ sur $(C, (X \rightarrow B))$ et $(C, (B, W \rightarrow V)) \in G_1 \cup G_3\}$
21: $G_2 := G_2 \cup \{(C, (X \rightarrow B))\}$
22: **fin si**
23: $G_1 := G_1 \setminus \{(C, (X \rightarrow B))\}$
24: **fin pour**
25: *cover*$\mathcal{F}$:= *cover*$\mathcal{F} \cup G_3$
26: **fin pour**
27: **retourner** *cover*$\mathcal{F}$

$E))$ sans les lignes 15-20 de l'Algorithme 5. Soit $f = (C, (X \rightarrow B))$ une des dépendances ayant échouées aux tests des lignes 7 à 12 de l'Algorithme 5. Pour éviter que les dépendances comme $(C, (A \rightarrow E))$ de l'exemple 4.16 ne soient pas dérivées par *cover*$\mathcal{F}$, l'Algorithme 5 calcule aux lignes 15 à 20 pour chaque XFD f, les dépendances de la forme $f_j = (C, (Y \rightarrow A))$ telles que :

(i) $C/X \in (C, Y)^{(1)}$ et $A = B$ ou

(ii) $C/Y = C/X$ et $C/A \in (C, B)^{(-1)}$ ou

(iii) $C/A = C/B$ et f_j est obtenue en appliquant l'axiome A4 sur f, ou

(iv) $Y = X \cup Y_1$ et f_j est obtenue en appliquant l'axiome A12 sur f et $(C, (B, Y_1 \rightarrow A))$

Les tests des lignes 7-12 seront ensuite exécutés sur les nouvelles dépendances f_j. Dans ce sens, nous ne calculons pas toute la fermeture de $\mathcal{F}_k$ mais nous calculons plutôt une partie de la fermeture si besoin. Ainsi pour calculer juste une partie de la fermeture d'un ensemble de XFD, nous utilisons deux fonctions : *closure*1*Step* et *inverseClosure*1*Step*. La fonction *closure*1*Step* calcule une étape de la fermeture d'un ensemble de chemins ($(C, X)^{(1)}$). Quand à la fonction *inverseClosure*1*Step*, elle considère les XFD dans le sens inverse et calcule une étape de la fermeture inverse ($(C, X)^{(-1)}$) en allant vers l'arrière. Ces deux fonctions seront expliquées en détails après. L'exemple suivant illustre le calcul effectué aux lignes 15-20 de l'Algorithme 5.

Exemple 4.17 Considérons les ensembles $\mathcal{F}_1$ et $\mathcal{F}_2$ de l'exemple 4.16. Le Tableau 4.6 nous montre les dépendances obtenues en considérant chaque XFD dans $\mathcal{F}_1$ (ligne 3 de l'Algorithme 5). La première colonne de ce tableau indique les dépendances de G_1 qui sont analysées. La seconde colonne indique les XFD qui sont ajoutées à G_1 lors de l'exécution des lignes 15-20. Finalement la troisième colonne indique les XFD qui sont insérées dans *cover*$\mathcal{F}$.

G_1 (XFD analysées)	Ajout à G_1	Contenu de cover$\mathcal{F}$
$(C, (A \rightarrow B))$		$(C, (A \rightarrow B))$ (cond. ligne 11)
$(C, (B \rightarrow M))$		$(C, (B \rightarrow M))$ (cond. ligne 11)
$(C, (M \rightarrow D))$	$(C, (M \rightarrow E))$ $(C, (B \rightarrow D))$ $(C, ([\,] \rightarrow D))$ $(C, (\{F, M\} \rightarrow E))$	
$(C, (D \rightarrow E))$	$(C, ([\,] \rightarrow E))$	
$(C, (O \rightarrow Z))$		$(C, (O \rightarrow Z))$ (cond. ligne 7)
$(C, (M \rightarrow E))$	$(C, (B \rightarrow E))$	
$(C, (B \rightarrow D))$	$(C, (A \rightarrow D))$	
$(C, (\{F, M\} \rightarrow E))$		$(C, (\{F, M\} \rightarrow E))$ (cond. ligne 11)
$(C, (B \rightarrow E))$		$(C, (B \rightarrow E))$ (cond. ligne 11)
$(C, (A \rightarrow D))$	$(C, (A \rightarrow E))$	
$(C, (A \rightarrow E))$		$(C, (A \rightarrow E))$ (cond. ligne 11)

Tableau 4.6 – *Calcul d'une partie de cover*$\mathcal{F}$ *: XFD obtenue en considérant* $\mathcal{F}_1$

Le Tableau 4.6 est construit en suivant l'exécution de l'Algorithme 5. Par exemple considérons la troisième ligne de ce tableau : le cas où la XFD $(C, (M \rightarrow D))$ appartenant à $\mathcal{F}_1$ est choisie à la ligne 6 de l'Algorithme 5. Cette dépendance ne vérifie aucune condition parmi les conditions des lignes 7, 9 et 11. Nous avons :

- lorsque la ligne 15 est exécutée, l'ensemble $H = \{C/E\}$ est calculé puisque *closure*1*Step*$(C,\ D,\ \mathcal{F}_1)$ donne $\{C/D, C/E\}$. Par conséquent la XFD $(C, (M \rightarrow E))$ est ajoutée à G_1 (ligne 16) ;
- lorsque la ligne 17 est exécutée, l'ensemble $K = \{\{C/B\}\}$ est calculé puisque *inverseClosure*1*Step*$(C,\ D,\ \mathcal{F}_1)$ donne $\{\{C/B\}, \{C/M\}\}$. Par conséquent la XFD $(C, (B \rightarrow D))$ est ajoutée à G_1 (ligne 18) ;
- lorsque la ligne 19 est exécutée, la XFD $(C, ([] \rightarrow D))$ est ajoutée à G_1 ;

– lorsque la ligne 20 est exécutée, la XFD $(C, (\{F, M\} \rightarrow E))$ est ajoutée à G_1 puisque en se servant de l'axiome A12 (Pseudo-transitivité) sur $(C, (M \rightarrow D))$ et $(C, (\{F, D\} \rightarrow E))$ on obtient $(C, (\{F, M\} \rightarrow E))$.

Notons que ces quatre XFD seront analysées plus tard (lignes 6, 7 et 8 du Tableau 4.6). Les dépendances $(C, (M \rightarrow E))$ et $(C, (B \rightarrow D))$ ne seront pas ajoutées à $cover\mathcal{F}$ mais vont générées d'autres XFD comme par exemple $(C, (A \rightarrow E))$ qui sera finalement ajoutée à $cover\mathcal{F}$. La dépendance $(C, (\{F, M\} \rightarrow E))$ sera ajoutée à $cover\mathcal{F}$ lors de son analyse. Nous avons vu dans l'exemple 4.16 que sans les lignes 15-20 les dépendances $(C, (A \rightarrow E))$ et $(C, (\{F, M\} \rightarrow E))$ ne peuvent pas être engendrées par $cover\mathcal{F}$, et dans cet exemple nous voyons comment les lignes 15-20 permettent à $cover\mathcal{F}$ d'engendrer $(C, (A \rightarrow E))$ et $(C, (\{F, M\} \rightarrow E))$.

⊠

Fonctions closure1Step et inverseClosure1Step. Pour pouvoir calculer une partie de la fermeture d'un ensemble de XFD, nous allons introduire deux fonctions : *closure*1*Step* et *inverseClosure*1*Step*.

La fonction *closure*1*Step* calcule une étape de la fermeture d'un ensemble de chemins $((C, X)^{(1)})$. Plus précisément si nous prenons l'Algorithme 2, pour l'ensemble de chemins C/X, la fonction *closure*1*Step* calcule $C/X^{(1)}$. Le code de la fonction *closure*1*Step* revient à supprimer la boucle « tant que » de l'Algorithme 2 pour que les instructions ne soient exécutées qu'une seule fois. Nous trouverons donc le code de la fonction *closure*1*Step* dans l'Algorithme 6.

La fonction *inverseClosure*1*Step* considère les XFD dans le sens inverse et calcule, pour un ensemble de chemins, une étape de la fermeture inverse $((C, X)^{(-1)})$ en allant vers l'arrière. Cette fonction est implémentée par l'Algorithme 7. Le résultat est un ensemble S contenant des ensembles de chemins. Mais avant de passer à l'algorithme, nous allons définir l'opérateur $\uplus$ sur des ensembles d'ensembles :

Définition 4.27 (Distribution de l'union)
Soient deux ensembles S_1, S_2 qui contiennent des ensembles d'éléments. La distribution de l'union de S_1 et S_2, notée $S_1 \uplus S_2$, est définie comme suit :

$$S_1 \uplus S_2 = \{U \mid U = U_1 \cup U_2 \text{ où } U_1 \in S_1 \text{ et } U_2 \in S_2\}$$

$S_1 \uplus S_2$ contient tous les ensembles résultant de l'union de chaque élément U_1 dans S_1 avec chaque élément U_2 dans S_2.

□

Pour illustrer l'opérateur $\uplus$ prenons l'exemple suivant :

Exemple 4.18 Considérons les ensembles $S_1 = \{\emptyset\}$, $S_2 = \{\{A, B\}, \{D\}\}$ et $S_3 = \{\{M, N\}\}$. Nous avons :

– $S_1 \uplus S_2 = \{\{A, B\}, \{D\}\} = S_2$
– $S_1 \uplus S_3 = \{\{M, N\}\} = S_3$
– $S_2 \uplus S_3 = \{\{A, B, M, N\}, \{D, M, N\}\}$

⊠

Algorithme 6 $closure1Step(C, X, \mathcal{F})$: Fermeture en un pas d'un ensemble de chemins

Entrée :
- un ensemble fini de chemins simple $\mathbb{P}$
- un ensemble de XFD $\mathcal{F}$
- un chemin contexte C et un ensemble de chemins X tel que $C/X \subseteq \mathbb{P}$
- un vecteur de types d'égalité $\vec{E}$ associé à X

Sortie : L'ensemble de chemins $(C, X[\vec{E}])^{(1)}$

1: $S := \{[][N]\}$ **% *grâce à l'axiome A7* %**
2: $T := \{P[N] \mid P \preceq Q \text{ et } Q[N] \in X[\vec{E}]\} \cup X[\vec{E}]$ **% *grâce aux axiomes A1 et A5* %**
3: $V := \{P[N] \mid Q[E'] \in T \cup S, Parent(P) = Q \text{ et } Last(P) \in \Sigma_{att}\}$ **% *grâce à l'axiome A6* %**
4: $W := \{P[V] \mid P[N] \in S \cup T \cup V\}$ **% *grâce à l'axiome A9* %**
5: $X^{(0)} := S \cup T \cup V \cup W$

6: $Y := \{P[E_{n+1}] \mid$ il existe une XFD $f \in \mathcal{F}$ où $f = (C, (\{P_1[E_1], \ldots, P_n[E_n]\} \rightarrow P[E_{n+1}]))$ ou $f = (C', (\{Q/P_1[E_1], \ldots, Q/P_n[E_n]\} \rightarrow Q/P[E_{n+1}]))$ avec $C = C'/Q$, telle qu'une des conditions ci-dessous soit satisfaite :

(a) $\{P_1[E_1], \ldots, P_n[E_n]\} \subseteq X^{(0)}$ ou **% *grâce à l'axiome A3* %**

(b) $\exists P'_1[E_1], \ldots, P'_n[E_n] \in X^{(0)}$ tel que $\forall k \in [1, \ldots, n]$, $P_k \cap P \preceq P'_k$ et $(P'_k \preceq P_k$ ou $P'_k \preceq P)$ **% *grâce à l'axiome A4* %**

$\}$
7: $T := \{P[N] \mid P \preceq Q \text{ et } Q[N] \in Y\} \cup Y$ **% *grâce à l'axiome A5* %**
8: $V := \{P[N] \mid Q[E'] \in T, Parent(P) = Q \text{ et } Last(P) \in \Sigma_{att}\}$ **% *grâce à l'axiome A6* %**
9: $W := \{P[V] \mid P[N] \in T \cup V\}$ **% *grâce à l'axiome A9* %**
10: $X^{(1)} := X^{(0)} \cup T \cup V \cup W$
11: **retourner** $C/X^{(1)}$

La ligne 1 initialise l'ensemble résultat S. Ensuite pour chaque chemin P dans X, l'Algorithme 7 calcule $(C, \{P[E']\})^{(-1)}$ et ensuite combine les résultats par la distribution de l'union pour obtenir les ensembles de chemins dans $(C, X[\vec{E}])^{(-1)}$. L'intuition derrière cette façon de procéder est la suivante :

- si $X \rightarrow \{A, B\}$ alors on a $X \rightarrow A$ et $X \rightarrow B$;
- si $\{X, Y\} \rightarrow \{A, B\}$ alors on a $X \rightarrow A$ et $Y \rightarrow B$, ou $X \rightarrow B$ et $Y \rightarrow A$, ou simplement que $\{X, Y\} \rightarrow A$ et $\{X, Y\} \rightarrow B$.

Les lignes 2-11 font l'initialisation de $(C, \{P[E']\})^{(-1)}$. Pour cela l'Algorithme 7 stocke dans R tous les ensembles de chemins qui impliquent le chemin P en utilisant les axiomes A7 (Unicité du contexte), A1 (Réflexivité), A5 (Unicité des ascendants), A6 (Unicité de l'attribut) et A9 (De l'identité de nœud à l'égalité par valeur) :

- si $P = []$ comme n'importe quel chemin implique P (axiome A7), l'ensemble R est initialisé avec des singletons formé chacun par un chemin de $\mathbb{P}$.
- si $Last(P)$ est un élément (ligne 5), comme avec l'axiome A5 n'importe quel chemin Q dont P est préfixe détermine P, l'ensemble R est initialisé avec des singletons formé chacun d'un chemin Q. Rappelons que $P \preceq P$, donc $\{P[E']\} \in R$.
- si $Last(P)$ est un attribut (ligne 7), en appliquant A6 nous avons $(C, (Parent(P) \rightarrow P)$. L'ensemble R est initialisé avec deux ensembles : l'un contenant P lui même (A1) et l'autre contenant son parent $Parent(P)$ (A6).
- si $Last(P)$ est une donnée (ligne 9) alors l'ensemble R est initialisé avec $\{P\}$ à cause de l'axiome A1.

Algorithme 7 *inverseClosure*1*Step*$(C, X, \mathcal{F})$: Une étape de la fermeture inverse d'un ensemble de chemins

Entrée :
- un ensemble fini de chemins simple $\mathbb{P}$
- un ensemble de XFD $\mathcal{F}$
- un chemin contexte C et un ensemble de chemins X tel que $C/X \subseteq \mathbb{P}$
- un vecteur de types d'égalité $\vec{E}$ associé à X

Sortie : L'ensemble de chemins $(C, X[\vec{E}])^{(-1)}$

1: $S := \{\emptyset\}$
2: **pour tout** chemin $P[E'] \in X[\vec{E}]$ **faire**
3: **si** $P = []$ **alors**
4: $R := \{\{Q[E'']\} \mid Q \in \mathbb{P}, E'' = N \text{ ou } E'' = V\}$
5: **sinon si** $Last(P) \in \Sigma_{ele}$ **alors**
6: $R := \{\{Q[N]\} \mid P \preceq Q\}$
7: **sinon si** $Last(P) \in \Sigma_{att}$ **alors**
8: $R := \{\{P[E']\}\} \cup \{\{Parent(P)[E''] \mid E'' = N \text{ ou } E'' = V\}\}$
9: **sinon si** $Last(P) \in \Sigma_{data}$ **alors**
10: $R := \{\{P[E']\}\}$
11: **fin si**
12: $R := \{\{Q[N]\} \mid \{Q[V]\} \in R\}$
13:
14: $Y_1 := \{\{P_1[E_1], \ldots, P_n[E_n]\} \mid$ il existe une XFD $[(C, (\{P_1[E_1], \ldots, P_n[E_n]\} \rightarrow P'[E_{n+1}]))$ ou $(C', (\{Q/P_1[E_1], \ldots, Q/P_n[E_n]\} \rightarrow Q/P'[E_{n+1}]))$ avec $C = C'/Q]$ dans $\mathcal{F}$ telle que $P' = P\}$
15:
16: $Y_2 := \{\{P_1[E_1], \ldots, P_n[E_n]\} \mid$ il existe une XFD $[(C, (\{P'_1[E_1], \ldots, P'_n[E_n]\} \rightarrow P'[E_{n+1}]))$ ou $(C', (\{Q/P'_1[E_1], \ldots, Q/P'_n[E_n]\} \rightarrow Q/P'[E_{n+1}]))$ avec $C = C'/Q]$ dans $\mathcal{F}$ telle que $P' = P$ et $\forall k \in [1, \ldots, n]$, $P'_k \cap P \preceq P_k$ et $(P_k \preceq P'_k$ ou $P_k \preceq P)\}$
17:
18: $S := S \uplus (R \cup Y_1 \cup Y_2)$
19: **fin pour**
20: **retourner minimiser**(C/S)

- si l'égalité par valeur est associé à un chemin Q de R alors $Q[N]$ est aussi un élément de R (axiome A9).

A la ligne 14, Y_1 contient tous les ensembles de chemins Z apparaissant à gauche d'une dépendance dans $\mathcal{F}$ qui a le chemin P à droite c'est à dire $(C, (Z \rightarrow P)) \in \mathcal{F}$. A la ligne 16, Y_2 contient tous les ensembles de chemins respectant les conditions de l'axiome A4 (Branche Préfixe) sur une dépendance de $\mathcal{F}$ ayant P comme chemin droit. N'oublions pas que pour Y_1 et Y_2 nous considérons aussi les dépendances de $\mathcal{F}$ sur lesquels on doit appliquer l'axiome A8 (Extension du contexte) avant d'obtenir une dépendance avec le contexte C. Ensuite à la ligne 18, la nouvelle valeur de S est obtenue en faisant la distribution de l'union ($\uplus$) de l'ancienne valeur de S et l'union de Y_1, Y_2 et R. Finalement l'ensemble S est réduite à la ligne 20 par la fonction *minimiser*. La fonction *minimiser* supprime de S tous les ensembles qui contiennent au moins un autre ensemble de S. L'exemple suivant illustre le fonctionnement de l'Algorithme 7.

Exemple 4.19 Considérons l'ensemble $\mathcal{F}$ avec trois XFD : $(C, (D \rightarrow O))$, $(C, (\{A, B\} \rightarrow O))$

et $(C, (\{M, N\} \rightarrow Q))$. Aussi nous affirmons que $Q \prec J$ et que $O \prec I$. Soit $X = \{O, Q\}$ où les derniers nœuds de O et Q sont des nœuds éléments. Cet exemple est simplifié en ne tenant pas compte des types d'égalités. En déroulant l'Algorithme 7, nous obtenons à chaque étape les résultats dans le tableau suivant :

	1ère étape de la boucle	2ème étape de la boucle **pour tout**
chemin $P \in X$	O	Q
R (ligne 5)	$\{\{O\}, \{I\}\}$	$\{\{Q\}, \{J\}\}$
Y_1 (ligne 14)	$\{\{A, B\}, \{D\}\}$	$\{\{M, N\}\}$
Y_2 (ligne 16)	$\{\{[]\}\}$	$\{\{[]\}\}$
S (ligne 18)	$\{\{O\}, \{I\}, \{A, B\}, \{D\}, \{[]\}\}$	$\{\{O, Q\}, \{O, J\}, \{O, M, N\}, \{O, []\},$ $\{I, Q\}, \{I, J\}, \{I, M, N\}, \{I, []\},$ $\{A, B, Q\}, \{A, B, J\},$ $\{A, B, M, N\}, \{A, B, []\},$ $\{D, Q\}, \{D, J\}, \{D, M, N\}, \{D, []\},$ $\{[], Q\}, \{[], J\}, \{[], M, N\}, \{[]\}\}$
minimiser(S) (ligne 20)	$\{\{O\}, \{I\}, \{A, B\}, \{D\}, \{[]\}\}$	$\{\{O, Q\}, \{O, J\}, \{O, M, N\},$ $\{I, Q\}, \{I, J\}, \{I, M, N\},$ $\{A, B, Q\}, \{A, B, J\}, \{A, B, M, N\},$ $\{D, Q\}, \{D, J\}, \{D, M, N\}, \{[]\}$

Tableau 4.7 – *Calcul de S contenant les ensembles de chemin qui implique un chemin P en un pas.*

L'ensemble résultat indique que les XFD comme $(C, (\{I, Q\} \rightarrow X))$ ou $(C, (\{A, B, M, N\} \rightarrow X))$ sont dérivées à partir de l'ensemble initial $\mathcal{F}$. Ainsi, l'Algorithme 7 retourne l'ensemble $S = \{\{C/O, C/Q\}, \{C/O, C/J\}, \cdots, \{C/[]\}\}$. Remarquons aussi que pour cet exemple, à la ligne 17 de l'Algorithme 5 l'ensemble K va contenir tous les ensembles de chemins dans S sauf $\{C/O, C/Q\}$ qui correspond à notre X.

⊠

4.7.4 Calcul de la couverture de $\mathcal{F}$: Preuve de correction

Dans cette section nous allons prouver que l'ensemble *cover*$\mathcal{F}$ retourné par l'Algorithme 5 contient des XFD correctes, et est bien une couverture de $\mathcal{F}$ calculé par l'Algorithme 4. Nous allons introduire le lemme suivant qui pour un ensemble de XFD $\mathcal{F}$, nous indique les dépendances qu'il faut ajouter à l'ensemble $\mathcal{F} \setminus \{f\}$ pour continuer à assurer la dérivation de $\mathcal{F}^+ \setminus \{f\}$, c'est-à-dire continuer à dériver toutes les dépendances (si possible f) que $\mathcal{F}$ dérive.

Lemme 4.3 *Soit $\mathcal{F}$ un ensemble de XFD défini sur l'ensemble des chemins $\mathbb{P}$, tel que $(C, (X \rightarrow Y)) \in \mathcal{F}$. Soit $H = \{(C, (X, V \rightarrow Y, V)) \mid V \subseteq \mathbb{P}\}$ c'est-à-dire l'ensemble des XFD obtenues en appliquant l'axiome A2 (Augmentation) sur la dépendance $(C, (X \rightarrow Y))$. Soit $(C, (Z_1 \rightarrow Z_2))$ une XFD différente de $(C, (X \rightarrow Y))$ et telle que $(C, (Z_1 \rightarrow Z_2)) \notin H$. Si $\mathcal{F} \vdash (C, (Z_1 \rightarrow Z_2))$ alors $G \vdash (C, (Z_1 \rightarrow Z_2))$ où G est l'ensemble de XFD défini comme suit :*

$$G = \mathcal{F} \cup \mathcal{F}_1 \cup \mathcal{F}_2 \cup \mathcal{F}_3 \cup \mathcal{F}_4 \setminus \{(C, (X \rightarrow Y))\}$$

avec

$$\mathcal{F}_1 = \{(C, (X \rightarrow V)) \mid V \in (C, Y)_{\mathcal{F}}^{(1)}\}$$

$$\mathcal{F}_2 = \{(C, (W \rightarrow Y)) \mid W \in (C, X)_{\mathcal{F}}^{(-1)}\}$$

$$\mathcal{F}_3 = \{(C, (X' \rightarrow Y)) \mid X' \textit{ respecte les conditions de l'axiome A4 sur } (C, (X \rightarrow Y))\}$$

$$\mathcal{F}_4 = \{(C,(X,W \to V)) \mid (C,(Y,W \to V)) \in \mathcal{F} \; et \; W \subseteq I\!P\}$$

Démonstration. Supposons en premier que $(C,(Z_1 \to Z_2)) \in \mathcal{F}$. Comme $(C,(Z_1 \to Z_2))$ est différent de $(C,(X \to Y))$ et G contient toutes les dépendances de $\mathcal{F}$ excepté $(C,(X \to Y))$ alors nous avons $G \vdash (C,(Z_1 \to Z_2))$.

Dans le cas contraire $(C,(Z_1 \to Z_2)) \notin \mathcal{F}$. Comme $\mathcal{F} \vdash (C,(Z_1 \to Z_2))$ cela veut dire qu'il existe une séquence de dérivation α qui est une démonstration de $(C,(Z_1 \to Z_2))$ à partir de $\mathcal{F}$. Lorsque α ne contient pas $(C,(X \to Y))$ alors il est clair que $G \vdash (C,(Z_1 \to Z_2))$. Nous allons maintenant considérer que α contient $(C,(X \to Y))$. La preuve va consister à construire à partir de α une nouvelle séquence de dérivation qui sera la démonstration de $(C,(Z_1 \to Z_2))$ à partir de G. Pour construire la nouvelle séquence de dérivation, on va isoler la partie de la séquence α où $(C,(X \to Y))$ intervient et on va remplacer toute cette partie par une ou plusieurs dépendances qui appartiennent à G et qui sont dérivées en utilisant $(C,(X \to Y))$.

1. Soit la séquence de dérivation $\alpha_1 = f_1, \cdots, f_n$ telle que :
   (a) chaque f_i $(1 \leq i \leq n)$ est dans α
   (b) chaque f_i $(1 \leq i \leq n)$ est de la forme $(C,(X_i \to X)$ c'est à dire que $X_1, \cdots, X_n$ sont des ensembles de chemins dans $(C,X)^{(-1)}_{\mathcal{F}}$.

   Nous disons que α_1 est la sous-séquence de α qui dérive l'ensemble X en une étape. Puisque $\mathcal{F} \vdash (C,(X_i \to X))$ et $\mathcal{F} \vdash (C,(X \to Y))$ alors par transitivité (A3) $\mathcal{F} \vdash (C,(X_i \to Y))$. Nous construisons la séquence α^1 en remplaçant dans α la dépendance

   $$(C,(X \to Y))$$

   par la séquence

   $$(C,(X_1 \to Y)), \cdots, (C,(X_n \to Y)).$$

   En considérant α^1 nous venons de prouver que $G \vdash (C,(Z_1 \to Z_2))$ puisque chaque XFD $(C,(X_i \to Y)) \in G$ (ensemble $\mathcal{F}_2$ du lemme).

2. De façon similaire au cas 1, soit la séquence α_2 qui est la sous-séquence de α qui dérive les chemins $Y_1, \cdots, Y_2$ en une étape à partir du chemin Y. Les chemins $Y_1, \cdots, Y_n$ sont dans $(C,Y)^{(1)}_{\mathcal{F}}$ et puisque $\mathcal{F} \vdash (C,(X \to Y))$ et $\mathcal{F} \vdash (C,(Y \to Y_i))$ alors par transitivité (A3) $\mathcal{F} \vdash (C,(X \to Y_i))$. Nous construisons la séquence α^2 en remplaçant dans α la dépendance

   $$(C,(X \to Y))$$

   par la séquence

   $$(C,(X \to Y_1)), \cdots, (C,(X \to Y_n)).$$

   En considérant α^2 nous venons de prouver que $G \vdash (C,(Z_1 \to Z_2))$ puisque chaque XFD $(C,(X \to Y_i)) \in G$ (ensemble $\mathcal{F}_1$ du lemme).

3. De façon similaire au cas 1, soit la séquence α_3 qui est la sous-séquence de α qui dérive le chemin Y en une étape à partir de chaque ensemble X_i $(i \in [1, \cdots, n])$ et en utilisant l'axiome l'axiome A4 sur la dépendance $(C,(X \to Y))$. Nous construisons la séquence α^3 en enlevant la dépendance $(C,(X \to Y))$ de α. Puisque chaque XFD $(C,(X_i \to Y)) \in G$ (ensemble $\mathcal{F}_3$ du lemme), nous venons de prouver que $G \vdash (C,(Z_1 \to Z_2))$.

4. De façon similaire au cas 1, soit la séquence α_4 qui est la sous-séquence de α qui dérive les dépendances $(C,(X,W_i \to V_i))$ $(i \in [1, \cdots, n])$ à partir de $(C,(X \to Y))$ et $(C,(Y,W_i \to V_i))$ de deux manières différentes :

- soit en utilisant l'axiome A2 pour obtenir $(C, (X, W_i \rightarrow Y, W_i))$ et ensuite la transitivité (A3) pour obtenir $(C, (X, W_i \rightarrow V_i))$;
- soit en appliquant directement l'axiome A12 sur $(C, (X \rightarrow Y))$ et $(C, (Y, W_i \rightarrow V_i))$ pour obtenir $(C, (X, W_i \rightarrow V_i))$.

Nous construisons la séquence α^4 en remplaçant dans α la séquence α_4 par la séquence $(C, (X, W_n \rightarrow V_n)), \cdots, (C, (X, W_n \rightarrow V_n))$. Puisque chaque XFD $(C, (X, W_i \rightarrow V_i)) \in G$ (ensemble $\mathcal{F}_4$ du lemme), nous venons de prouver que $G \vdash (C, (Z_1 \rightarrow Z_2))$.

Nous venons de montrer qu'en utilisant des XFD dans l'ensemble $\mathcal{F}_1$ ou $\mathcal{F}_2$ ou $\mathcal{F}_3$ ou $\mathcal{F}_4$, il est possible de trouver une démonstration de $(C, (Z_1 \rightarrow Z_2))$ à partir de G. Nous allons montrer qu'il existe au moins une telle séquence en se servant des XFD de $\mathcal{F}_1 \cup \mathcal{F}_2 \cup \mathcal{F}_3 \cup \mathcal{F}_4$. Pour cela montrons que si $\mathcal{F}_1 \cup \mathcal{F}_2 \cup \mathcal{F}_3 \cup \mathcal{F}_4 = \emptyset$ et que $\mathcal{F} \vdash (C, (Z_1 \rightarrow Z_2))$ alors la séquence α considérée précédemment ne peut pas contenir $(C, (X \rightarrow Y))$. Cette preuve sera faite par contradiction. Supposons que α contienne $(C, (X \rightarrow Y))$ et que α est construit à partir des axiomes élémentaires A1-A9. Notons que parmi les axiomes du système d'axiomes, ceux qui s'appuient sur une ou plusieurs XFD pour dériver une nouvelle XFD sont les axiomes A2, A3 et A4. Nous avons les différents cas suivants :

- α contient une seule dépendance qui est $(C, (X \rightarrow Y))$. Dans ce cas $(C, (X \rightarrow Y))$ est le même que $(C, (Z_1 \rightarrow Z_2))$ qui nous donne une contradiction avec notre hypothèse : $(C, (Z_1 \rightarrow Z_2))$ est différent de $(C, (X \rightarrow Y))$.
- α contient plus d'une dépendance y compris $(C, (X \rightarrow Y))$. Si la dépendance $(C, (X \rightarrow Y))$ est dans la séquence α alors l'un ou plusieurs des axiomes A2, A3 et A4 ont été utilisé. Si l'axiome A2 a été utilisé alors on est dans le même cas que le cas 4 et donc l'ensemble $\mathcal{F}_4 \neq \emptyset$ ce qui contredit notre hypothèse. Si l'axiome A3 a été utilisé alors on est dans le cas 1 ou le cas 2 et donc l'ensemble $\mathcal{F}_1 \neq \emptyset$ ou l'ensemble $\mathcal{F}_3 \neq \emptyset$ ce qui contredit notre hypothèse. Si l'axiome A4 a été utilisé alors on est dans le même cas que le cas 3 et donc l'ensemble $\mathcal{F}_3 \neq \emptyset$ ce qui contredit notre hypothèse.

Ceci prouve donc que si $\mathcal{F}_1 \cup \mathcal{F}_2 \cup \mathcal{F}_3 \cup \mathcal{F}_4 = \emptyset$ et que $\mathcal{F} \vdash (C, (Z_1 \rightarrow Z_2))$ alors la séquence α considérée précédemment ne peut pas contenir $(C, (X \rightarrow Y))$.

◁

Remarquons que la dérivation de f à partir du nouvel ensemble G n'est pas garantie mais reste possible. Dans le cas où $G \vdash f$ nous avons G équivalent à $\mathcal{F}$. Nous pouvons aussi voir l'ensemble $\mathcal{F}_1 \cup \mathcal{F}_2 \cup \mathcal{F}_3 \cup \mathcal{F}_4$ du Lemme 4.3 comme le sous-ensemble indispensable de l'ensemble des XFD qu'on peut « ne pas pouvoir dériver » en enlevant f de $\mathcal{F}$. Grâce au Lemme 4.3 nous pouvons maintenant prouver que l'ensemble $cover\mathcal{F}$ est équivalent à $\mathcal{F}$ en d'autres termes que $cover\mathcal{F}^+ = \mathcal{F}^+$.

Théorème 4.3 *$cover\mathcal{F} \equiv \mathcal{F}$: l'ensemble $cover\mathcal{F}$, retourné par l'Algorithme 5, est équivalent à (ou est une couverture de) l'ensemble $\mathcal{F} = (\mathcal{F}_1^+ \cap \mathcal{F}_2^+) \cup \mathcal{K}_{\mathcal{F}_1/\mathcal{F}_2} \cup \mathcal{K}_{\mathcal{F}_2/\mathcal{F}_1}$.*

Démonstration. La preuve sera composée de trois parties : (A) l'Algorithme 5 termine, (B) $\forall f \in cover\mathcal{F}$, $\mathcal{F} \vdash f$ et (C) $\forall f \in \mathcal{F}$, $cover\mathcal{F} \vdash f$.

Fait A : l'Algorithme 5 termine.
Pour chaque XFD $h = (C, (Y \rightarrow A))$ qui est considérée à la ligne 6 de l'Algorithme 5, la dépendance h est ajoutée à $cover\mathcal{F}$ grâce aux tests des lignes 7-12 ou h est analysée et implique l'ajout de 0 ou plusieurs dépendances dans G (lignes 13-20).

Rappellons que l'ensemble $(C, Y)^+$ par rapport à $\mathcal{F}_k$ est fini et est calculé en ajoutant à chaque itération i de l'Algorithme 2, un ensemble de chemins Z_k à $(C, Y)^+_{\mathcal{F}_k}$. Cette procédure

continue jusqu'à ce que aucun ajout ne soit possible dans $(C, Y)^+_{\mathcal{F}_k}$. L'ensemble de chemins Z_k ($1 \leq k \leq n$) est obtenue par plusieurs appels successives de la fonction *closure1Step* (Algorithme 6). Ainsi, nous affirmons que $Z_n \subseteq Y_n = closure1Step(C, Y_{n-1}, \mathcal{F}_k)$, ..., $Z_2 \subseteq Y_2 = closure1Step(C, Y, \mathcal{F}_k)$.

Les fonctions $closure1Step(C, A, \mathcal{F}_i)$ et $inverseClosure1Step(C, Y, \mathcal{F}_i)$ suivent les XFD vers l'avant ou l'arrière, en visitant les chemins qui sont dans $(C, Y)^+_{\mathcal{F}_k}$. Et comme $(C, Y)^+_{\mathcal{F}_k}$ est fini, le nombre d'appels de $closure1Step(C, A, \mathcal{F}_k)$ ou $inverseClosure1Step(C, Y, \mathcal{F}_k)$ est aussi fini. Nous concluons que l'Algorithme 5 termine à cause de l'une des raisons suivantes :

- une nouvelle XFD ajoutée à G grâce à h, est aussi ajoutée à $cover\mathcal{F}$;
- une nouvelle XFD engendrée aux lignes 15-20 à partir de h, correspond à une XFD déjà traitée et donc n'est plus ajoutée à G ;
- la dépendance $h = (C, (Y \rightarrow A))$ est telle que $inverseClosure1Step(C, Y, \mathcal{F}_i) = \emptyset$ et $closure1Step(C, A, \mathcal{F}_i) = \emptyset$, et donc aucune XFD n'est ajoutée à G.

Fait B : $\forall f \in cover\mathcal{F}, \mathcal{F} \vdash f$ (ou $cover\mathcal{F}^+ \subseteq \mathcal{F}^+$).
En fait nous pouvons prouver que $cover\mathcal{F} \subseteq \mathcal{F}$ (qui est plus fort que prouver juste que $\mathcal{F} \vdash f$ pour toute XFD $f \in cover\mathcal{F}$). A partir de l'Algorithme 5, deux types de XFD sont ajoutées à $cover\mathcal{F}$:

1. Celles qui sont dans $\mathcal{F}_1$ ou $\mathcal{F}_2$ et qui réussissent aux tests des lignes 7, 9 et 11. Sans surprise, selon le contenu de $\mathcal{F}$ (Théorème 4.1), ces XFD sont aussi $\mathcal{F}$.
2. celles qui sont dérivées à partir de $\mathcal{F}_1$ ou $\mathcal{F}_2$ (lignes 15 à 20) et qui aussi réussissent aux tests des lignes 7, 9 et 11. Pour ce cas, considérons une XFD $g = (C, (X \rightarrow B))$ appartenant G pour qui les lignes 15 à 20 de l'Algorithme 5 sont exécutées :
   - d'après la ligne 16 on obtient $g_1 = (C, (X \rightarrow D))$ telle que $C/D \in (C, B)^+_{\mathcal{F}_k}$, qui est dans $\mathcal{F}^+_k$.
   - d'après la ligne 18 on obtient $g_2 = (C, (Y \rightarrow B))$ telle que $C/X \in (C, Y)^+_{\mathcal{F}_k}$, qui est dans $\mathcal{F}^+_k$.
   - d'après la ligne 19 on obtient $g_3 = (C, (Y \rightarrow B))$ telle que $C/B \in (C, Y)^+_{\mathcal{F}_k}$, qui est aussi dans $\mathcal{F}^+_k$.
   - d'après la ligne 20 on obtient $g_4 = (C, (X, W \rightarrow V))$ telle que $C/B, C/V \in (C, X \cup W)^+_{\mathcal{F}_k}$, qui est aussi dans $\mathcal{F}^+_k$.

   Puisque ces nouvelles dépendances sont dans $\mathcal{F}^+_1$ ou $\mathcal{F}^+_2$ et satisfont les conditions des lignes 7, 9 ou 11 alors ils sont aussi dans $\mathcal{F}$ (Théorème 4.1).

Nous venons de prouver que $cover\mathcal{F} \subseteq \mathcal{F}$.

Fait C : $\forall f \in \mathcal{F}, cover\mathcal{F} \vdash f$ (ou $\mathcal{F}^+ \subseteq cover\mathcal{F}^+$).
Cette partie est un peu plus difficile puisque nous prétendons que $cover\mathcal{F}$ qui est beaucoup plus petit et inclus dans $\mathcal{F}$, dérive les mêmes dépendances que $\mathcal{F}$. Pour y parvenir, nous allons prendre chaque dépendance $f \in \mathcal{F} = (\mathcal{F}^+_1 \cap \mathcal{F}^+_2) \cup \mathcal{K}_{\mathcal{F}_1/\mathcal{F}_2} \cup \mathcal{K}_{\mathcal{F}_2/\mathcal{F}_1}$ et montrer que f est aussi dans $cover\mathcal{F}^+$ (*i.e.*, $cover\mathcal{F} \vdash f$).

1. Soit $(C, (Y \rightarrow A))$ une XFD appartenant à l'ensemble $\mathcal{F}^+_1 \cap \mathcal{F}^+_2$. Avec cette hypothèse nous savons que $(C/Y \cup \{C/A\}) \subseteq \mathbb{P}_1$, $(C/Y \cup \{C/A\}) \subseteq \mathbb{P}_2$, $C/A \in (C, Y)^+_{\mathcal{F}_1}$ et $C/A \in (C, Y)^+_{\mathcal{F}_2}$.

(a) Si $(C,(Y \rightarrow A)) \in \mathcal{F}_1$ alors comme $C/A \in (C,Y)^+_{\mathcal{F}_2}$, d'après la ligne 11 de l'Algorithme 5, nous avons $(C,(Y \rightarrow A)) \in cover\mathcal{F}$. Il est donc évident que $cover\mathcal{F} \vdash (C,(Y \rightarrow A))$ ou $(C,(Y \rightarrow A)) \in cover\mathcal{F}^+$.

(b) Sinon si $(C,(Y \rightarrow A)) \in \mathcal{F}_2$ alors avec les mêmes arguments qu'au cas 1a, nous pouvons affirmer que $(C,(Y \rightarrow A)) \in cover\mathcal{F}$ et donc $(C,(Y \rightarrow A)) \in cover\mathcal{F}^+$.

(c) Sinon nous avons $f = (C,(Y \rightarrow A)) \notin \mathcal{F}_1$ et $(C,(Y \rightarrow A)) \notin \mathcal{F}_2$. Comme $C/A \in (C,Y)^+_{\mathcal{F}_1}$, il existe une séquence de dérivation $\alpha = f_1,\ldots,f_n$ qui dérive f.
En premier supposons que $cover\mathcal{F}$ contiennent toutes les dépendances de $\mathcal{F}_1$ qui sont dans la séquence α. Dans ce cas, il est évident que $cover\mathcal{F} \vdash f$.
Maintenant supposons le cas contraire, c'est à dire qu'il existe au moins une XFD de $\mathcal{F}_1$ qui est dans la séquence α mais qui n'est pas dans $cover\mathcal{F}$. Notons cette dépendance f_k. Nous savons que f_k ne satisfait pas les conditions des lignes 7, 9, 11 et donc sera considérée aux lignes 15-20. D'après les lignes 15-20, nous savons aussi que f_k sera enlevée de G et va impliquer l'ajout d'autres XFD h. D'après le Lemme 4.3, on obtient que $G \vdash f$.

Toutes les nouvelles XFD h seront analysées à la ligne 6. Elles seront ajoutées à $cover\mathcal{F}$ si elles satisfont les conditions des lignes 7, 9, 11. Dans le cas contraire elles sont traitées aux lignes 15-20 et la procédure continue jusqu'à ce que f soit ajoutée à G et ensuite à $cover\mathcal{F}$. Lorsque les dépendances h n'ont pas entraîné l'ajout de f dans $cover\mathcal{F}$, alors en suivant le même raisonnement que dans le Lemme 4.3 (qui consiste à remplacer les dépendances h par certaines parmi les nouvelles calculées), on arrive à construire à partir de α une séquence composée de XFD dans $cover\mathcal{F}$ qui dérive f.

Et pour comprendre pourquoi f sera éventuellement ajoutée à G, rappellons les faits suivants. D'après l'hypothèse, nous savons que $f = (C,(Y \rightarrow A))$ et $f \in \mathcal{F}_1^+$. Nous savons aussi que les nouvelles XFD h qui sont pas ajoutées à $cover\mathcal{F}$, provoquent l'ajout dans G de dépendances de la forme $(C,(Z \rightarrow W))$ avec $C/Z \subseteq (C,Y)^+_{\mathcal{F}_1}$ et $C/W \subseteq (C,Y)^+_{\mathcal{F}_1}$. Ceci est vrai parce que en calculant $(C,Y)^+_{\mathcal{F}_1}$ nous trouvons des chemins dans C/Z et C/W qui sont ajoutés à $(C,Y)^+_{\mathcal{F}_1}$ durant une étape de l'Algorithme 2. Plus précisément, il existe des ensembles de chemins $C/Z_1,\cdots,C/Z_n \subseteq (C,Y)^+$ tels que $C/W \subseteq closure1Step(C,Z_n,\mathcal{F}_1),\cdots,Z_2 \subseteq closure1Step(C,Z_1,\mathcal{F}_1), Z_1 \subseteq closure1Step(C,Z,\mathcal{F}_1)$. Comme la XFD f est l'une de ces XFD décrites au-dessus, elle peut éventuellement être ajoutée à G, et ensuite être ajoutée à $cover\mathcal{F}$ à la ligne 11 du fait que $C/A \in (C,Y)^+_{\mathcal{F}_1}$ et $C/A \in (C,Y)^+_{\mathcal{F}_2}$.

2. Soit $f = (C,(Y \rightarrow A)) \in \mathcal{K}_{\mathcal{F}_1/\mathcal{F}_2}$. Dans ce cas $f \in \mathcal{F}_1^+$, $(C/Y \cup \{C/A\}) \subseteq \mathbb{P}_1$, et $(C/Y \cup \{C/A\}) \not\subseteq \mathbb{P}_2$. Nous allons montrer que $f \in cover\mathcal{F}^+$. Comme $C/A \in (C,Y)^+_{\mathcal{F}_1}$, il existe une séquence $\alpha = f_1,\ldots,f_n$ correspondant à la dérivation de f. Avec les mêmes arguments que ceux dans le cas 1c, nous savons que $cover\mathcal{F}$ contient toutes les XFD nécessaires à la dérivation de f ou f est ajoutée à G. Lorsque f est ajoutée à G, son analyse à la ligne 6 va aboutir à l'ajout de f à $cover\mathcal{F}$ aux lignes 8 ou 10 parce que $(C/Y \cup \{C/A\}) \subseteq \mathbb{P}_1$, et $(C/Y \cup \{C/A\}) \not\subseteq \mathbb{P}_2$.
3. Soit $f = (C,(Y \rightarrow A)) \in \mathcal{K}_{\mathcal{F}_2/\mathcal{F}_1}$. La preuve que $f \in cover\mathcal{F}^+$, est similaire à celle du cas 2.

Nous pouvons donc conclure que $\mathcal{F}^+ \subseteq cover\mathcal{F}^+$.

◁

Nous obtenons bien un ensemble $cover\mathcal{F}$ qui est équivalent à $\mathcal{F}$ et de plus, $cover\mathcal{F}$ est aussi inclus dans $\mathcal{F}$. Passons maintenant à une brève analyse de la complexité pour calculer $cover\mathcal{F}$.

4.7.5 Analyse de la complexité de l'Algorithme 5

L'Algorithme 5 dépend principalement de l'Algorithme 2 qui calcule la fermeture d'un ensemble de chemins ($(C, X)^+$), et de l'Algorithme 7 qui calcule juste une étape de la fermeture inverse d'un ensemble de chemins ($(C, X)^{(-1)}$). L'Algorithme 6 qui calcule une étape de la fermeture d'un ensemble de chemins ($(C, X)^{(1)}$) est une version simplifiée de l'Algorithme 2. Nous avons vu dans la section 4.6.2, que la complexité du calcul de $(C, X)^+$ est de l'ordre de $O(|\mathbb{P}|^2 \cdot (|\mathcal{F}| + |\mathbb{P}|))$ dans le pire des cas où $|\mathcal{F}|$ est le nombre de XFD dans $\mathcal{F}$ et $|\mathbb{P}|$ est le nombre de chemins dans $\mathbb{P}$.

Analysons maintenant la complexité du calcul de $(C, X)^{(-1)}$ (Algorithme 7). Cette complexité est basée sur les remarques suivantes. A chaque itération nous calculons l'ensemble S en nous servant des ensembles Y_1 et Y_2 qui contiennent des ensembles de chemins.

- le calcul de Y_1 est fait en temps $O(|\mathcal{F}|)$.
- pour calculer Y_2 nous avons besoin de considérer toutes les XFD dans $\mathcal{F}$ et pour chaque $f \in \mathcal{F}$ ayant P à sa droite, nous calculons les préfixes de tous les chemins à gauche. Ainsi le calcul de Y_2 est fait en temps $O(|\mathcal{F}| \cdot (n \cdot m + m^n)$, où m est le nombre de labels dans le plus long chemin de f et n est le nombre de chemins à gauche de f. En fait nous avons :
  (i) les préfixes d'un chemin Q sont obtenus en temps $O(m)$. Nous devons calculer les préfixes de n chemins ;
  (ii) les nouveaux ensembles dans Y_2 sont obtenus en combinant chaque préfixe d'un chemin P_1 (à gauche de f) avec un préfixe des chemins $P_2, \cdots, P_n$. Ce calcul est fait en temps $O(m^n)$.
- Parmi tous les chemins de X, soit l le nombre maximal de dépendances dans $\mathcal{F}$ qui possède l'un des chemins de X comme chemin droit. Le nombre d'ensemble dans R (ligne 12, Algorithme 7) est m ; dans Y_1 est l et dans Y_2 est $(m^n \cdot l)$. Remarquons pourtant que dans le cas où il existe un chemin Q qui est préfixe de tous les chemins dans la partie gauche, une seule dépendance est gardée dans Y_2 au lieu de $m^n \cdot l$ à cause de la fonction *minimiser*.
- A chaque itération l'on calcule la distribution de l'union de deux ensembles S_1 et S_2 qui a une complexité de $O(|S_1| \cdot |S_2|)$. Comme on a X itérations, le calcul de S est fait en temps $O((m + l + m^n \cdot l)^{|X|})$. Les variables m et n sont des constantes.

Nous obtenons donc une complexité pour le calcul de $(C, X)^{(-1)}$ qui est de l'ordre de $O(l^{|X|})$ dans le pire des cas.

Dans le pire des cas, l'Algorithme 5 va traiter environ $|\mathcal{F}_i| \cdot |\mathbb{P}_i|$ XFD pour chaque ensemble $\mathcal{F}_i$. Nous obtenons le pire des cas lorsque pour chaque XFD $f = (C, (X \to P))$ dans $\mathcal{F}_i$, $(C, X)^+$ contient $|\mathbb{P}_i|$ chemins et un seul chemin est ajouté à $(C, X)^+$ pendant chaque étape de la boucle de l'Algorithme 2 et aucune XFD n'est ajoutée à $cover\mathcal{F}$. Ainsi, dans ce cas, les lignes 15-18 de l'Algorithme 5 seront exécutées $|\mathbb{P}_i|$ fois pour chaque XFD dans $\mathcal{F}_i$. La complexité de l'Algorithme 5 est donc de $O(|\mathcal{F}_i| \cdot |\mathbb{P}_i| \cdot (g + h))$ où g est la complexité de

l'Algorithme 2 et h est la complexité de l'Algorithme 7. Les variables qui sont déterminantes dans cette complexité sont les cardinalités de $\mathcal{F}$ et $\mathbb{P}$. Dans la pratique $|X|$ et n ne sont pas plus grand que 5, et par conséquent peuvent être négligées par rapport aux tailles de $\mathcal{F}$ et $\mathbb{P}$.

4.8 Résultats Expérimentaux

Afin d'examiner les performances de l'Algorithme 5, nous avons effectué plusieurs expérimentations sur des données synthétiques. Rappelons que l'Algorithme 5 prend en entrée deux systèmes locaux $S_1 = (\mathcal{D}_1, \mathcal{F}_1, O_1)$ et $S_2 = (\mathcal{D}_2, \mathcal{F}_2, O_2)$, et calcule l'ensemble $cover\mathcal{F}$ contenant les dépendances fonctionnelles pour lesquelles il n'y a pas de violation lorsque l'on considère les documents provenant des systèmes S_1 et S_2. Dans cette section nous présentons les résultats obtenus lors des expérimentations. Mais avant cela, nous présentons d'abord notre jeu de données.

4.8.1 Jeu de données

L'arbre utilisé dans nos expériences est celui de la Figure 4.12. Cet arbre $\mathcal{T}$ est construit en répétant plusieurs fois l'arbre motif de la Figure 4.11. Pour différencier les sous-arbres de $\mathcal{T}$, les nœuds de l'arbre motif ont été renommé en ajoutant l'indice k ($k \geq 1$). Ainsi nous utiliserons le *sous-arbre k* pour faire référence au $k^{\text{ième}}$ arbre motif dans $\mathcal{T}$.

Nos expériences ont consisté à générer $cover\mathcal{F}$ à partir d'ensembles $\mathcal{F}_1$ et $\mathcal{F}_2$ qui grandissent à chaque test tout en assumant l'existence de grands ensembles $\mathbb{P}_1$ et $\mathbb{P}_2$, et par conséquent des arbres $\mathcal{T}$. Dans la suite nous allons souvent faire référence à l'arbre $\mathcal{T}$ pour indiquer le type de documents (le schéma) avec lequel nous travaillons. Dans ce contexte, nous définissons $\mathbb{P}_1^j$ comme l'ensemble des chemins contenant tous les chemins de $\mathcal{T}$ sauf les chemins $C/R_{1,k}/G_{1,k}$ (avec $k \leq j$), et $\mathbb{P}_2^j$ comme l'ensemble des chemins contenant tous les chemins de $\mathcal{T}$ sauf les chemins $C/R_{1,k}/F_{1,k}$ (avec $k \leq j$). Nous supposons que l'ensemble des chemins $\mathbb{P}_1^j$ (respectivement $\mathbb{P}_2^j$) sont générés à partir de $\mathcal{D}_1$ (respectivement $\mathcal{D}_2$).

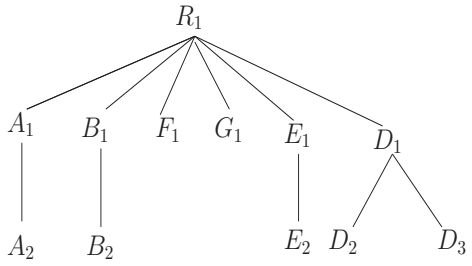


FIGURE 4.11 – *Arbre motif*

L'ensemble des chemins $\mathcal{F}_1^j$ (respectivement $\mathcal{F}_2^j$) est défini sur des chemins dans $\mathbb{P}_1^j$ (respectivement $\mathbb{P}_2^j$). Le Tableau 4.8 montre les XFD qui sont dans $\mathcal{F}_1^j$ et $\mathcal{F}_2^j$. Les ensembles $\mathcal{F}_1^j$ et $\mathcal{F}_2^j$ contiennent en commun les XFD (1) et (2). Cependant les XFD $(3a)$, $(4a)$ et $(5a)$ se retrouvent seulement dans $\mathcal{F}_1^j$ et les XFD $(3b)$, $(4b)$ et $(5b)$ se retrouvent seulement dans $\mathcal{F}_2^j$. Avec les XFD $(4a)$ et $(5a)$, nous pouvons dériver la XFD (6) $(C/R_{1,k}, (\{A_{1,k}/A_{2,k}, B_{1,k}/B_{2,k}\} \rightarrow E_{1,k}/E_{2,k}))$ et avec les $(4b)$ et $(5b)$, nous pouvons aussi dériver la XFD (6). Ainsi $\mathcal{F}_1^j$ et $\mathcal{F}_2^j$ dérivent la XFD (6) mais de différentes manières. Nous pouvons remarquer que $|\mathcal{F}_1^1| = |\mathcal{F}_2^1| = 5$, $|\mathcal{F}_1^2| = |\mathcal{F}_2^2| = 10$ et $\mathcal{F}_1^1 \subset \mathcal{F}_1^2$, $\mathcal{F}_2^1 \subset \mathcal{F}_2^2$.

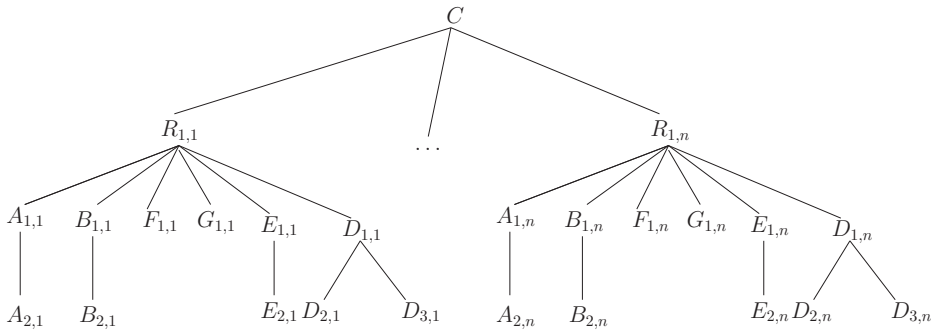


FIGURE 4.12 – *Arbre $\mathcal{T}$ construit en répétant, sous la même racine, n fois l'arbre motif de la Figure 4.11*

$\mathcal{F}_1^j$	$\mathcal{F}_2^j$
(1) $(C/R_{1,k}, (\{A_{1,k}, B_{1,k}\} \rightarrow D_{1,k}))$	(1) $(C/R_{1,k}, (\{A_{1,k}, B_{1,k}\} \rightarrow D_{1,k}))$
(2) $(C/R_{1,k}, (\{D_{1,k}\} \rightarrow E_{1,k}))$	(2) $(C/R_{1,k}, (\{D_{1,k}\} \rightarrow E_{1,k}))$
(3*a*) $(C/R_{1,k}, (\{E_{1,k}\} \rightarrow F_{1,k}))$	(3*b*) $(C/R_{1,k}, (\{E_{1,k}\} \rightarrow G_{1,k}))$
(4*a*) $(C/R_{1,k}, (\{A_{1,k}/A_{2,k}, B_{1,k}/B_{2,k}\} \rightarrow D_{1,k}/D_{2,k}))$	(4*b*) $(C/R_{1,k}, (\{A_{1,k}/A_{2,k}, B_{1,k}/B_{2,k}\} \rightarrow D_{1,k}/D_{3,k}))$
(5*a*) $(C/R_{1,k}, (\{D_{1,k}/D_{2,k}\} \rightarrow E_{1,k}/E_{2,k}))$	(5*b*) $(C/R_{1,k}, (\{D_{1,k}/D_{3,k}\} \rightarrow E_{1,k}/E_{2,k}))$

Tableau 4.8 – *Contenus des ensembles de XFD $\mathcal{F}_1^j$ et $\mathcal{F}_2^j$ utilisés dans les expérimentations*

4.8.2 Résultats

Nous avons pris en considération deux paramètres dans les expérimentations : (i) le nombre de chemins obtenus à partir de $\mathcal{D}_1$ et $\mathcal{D}_2$, et (ii) le nombre de XFD dans $\mathcal{F}_1$ et $\mathcal{F}_2$ ($\mid \mathcal{F}_1 \mid + \mid \mathcal{F}_2 \mid$). L'algorithme a été implémenté en Java et les tests ont été effectués sur une machine ayant les caractéristiques suivantes : un processeur Intel Quad Core $i3 - 2310M$ avec $2.10GHz$ et $8GB$ de mémoire RAM. Nous avons utilisé trois scénarios lors des tests.

Dans le premier scénario nous examinons l'influence de la taille de $\mathcal{F}_1$ et $\mathcal{F}_2$ sur le temps d'exécution de l'Algorithme 5. Nous avons utilisé $\mathcal{F}_1^j$ and $\mathcal{F}_2^j$ pour des valeurs de j comprises entre 1 et 45. La Figure 4.13 montre des temps d'exécution raisonnables (environ 2 minutes) pour calculer *cover*$\mathcal{F}$ à partir d'ensembles de XFD $\mathcal{F}_1$ et $\mathcal{F}_2$ tels que $|\mathcal{F}_1| + |\mathcal{F}_2| = 450$. La Figure 4.13 nous montre aussi comment la taille de *cover*$\mathcal{F}$ croît : à chaque étape puisqu'on rajoute 25 XFD à $|\mathcal{F}_1| + |\mathcal{F}_2|$, l'ensemble *cover*$\mathcal{F}$ a environ 50 XFD de plus que sa précédente version.

Dans le deuxième scénario nous examinons encore l'influence de la taille de $\mathcal{F}_1$ et $\mathcal{F}_2$ sur le temps d'exécution de l'Algorithme 5. Notons que dans le premier scénario, les XFD ayant les indices k ne concernaient qu'un seul sous-arbre de $\mathcal{T}$. Dans ce deuxième scénario, nous allons ajouter la possibilité qu'une XFD ayant l'indice $k = 1$ peut dériver des XFD d'indice $k = 2$, et ainsi de suite. Pour cela, nous ajoutons à l'ensemble $\mathcal{F}_1^j$ (respectivement $\mathcal{F}_2^j$) la XFD de la forme $(7a)$ $(C, (\{R_{1,k}/E_{1,k-1}/E_{2,k-1}\} \rightarrow R_{1,k}/D_{1,k}/D_{2,k}))$ (respectivement $(7b)$ $(C, (\{R_{1,k}/E_{1,k-1}/E_{2,k-1}\} \rightarrow R_{1,k}/D_{1,k}/D_{3,k}))$), avec $2 \leq k \leq j$.

Comme illustré dans la Figure 4.14, le temps d'exécution pour calculer *cover*$\mathcal{F}$ est plus important que celui obtenu avec le premier scénario. Par exemple, pour les ensembles $\mathcal{F}_1$ et $\mathcal{F}_2$ (tel que $|\mathcal{F}_1| + |\mathcal{F}_2| = 262$) nous avons besoin de 53 minutes pour calculer *cover*$\mathcal{F}$. Ce comportement est expliqué par deux facteurs :

– les XFD de la forme $(7a)$ et $(7b)$ ne sont pas ajoutés à *cover*$\mathcal{F}$ à cause de la condition à

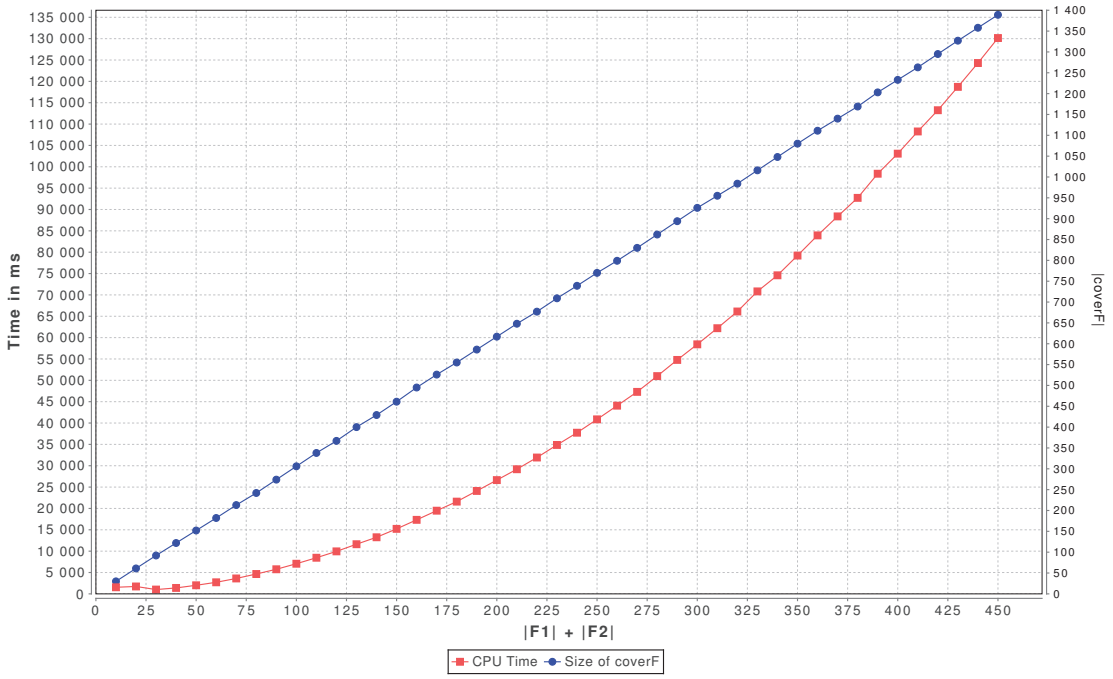


FIGURE 4.13 – *Scénario 1 : Temps CPU nécessaire pour le calcul de cover$\mathcal{F}$ et l'évolution de sa taille*

la ligne 11 de l'Algorithme 5. Vérifier cette condition est une tâche coûteuse parce que le calcul de $(C, R_{1,k}/E_{1,k-1}/E_{2,k-1})^+_{\mathcal{F}_i}$ contient beaucoup de chemins ;

– pour cet exemple, les lignes 15-18 de l'Algorithme 5 génèrent beaucoup de XFD qui augmentent considérablement le nombre de XFD dans *cover*$\mathcal{F}$. En effet $|cover\mathcal{F}|$ a environ 10610 XFD lorsque $|\mathcal{F}_1^j| + |\mathcal{F}_2^j| = 262$.

Dans le troisième scénario, nous comparons l'Algorithme 4 qui calcule l'ensemble maximal $\mathcal{F}$ à partir de $\mathcal{F}_1^+$ et $\mathcal{F}_2^+$ (présenté dans la section 4.7.1) avec l'Algorithme 5 qui calcule *cover*$\mathcal{F}$. Dans la section 4.7.4, nous avons prouvé que *cover*$\mathcal{F}$ est équivalent à $\mathcal{F}$. Le Tableau 4.9 présente un comparatif de ces deux algorithmes. La ligne 1 du Tableau 4.9 montre les résultats pour les ensembles $\mathcal{F}_1^1$ et $\mathcal{F}_2^1$ tandis que la ligne 4 montre les résultats pour $\mathcal{F}_1^2$ et $\mathcal{F}_2^2$ et la ligne 5 montre les résultats pour $\mathcal{F}_1^3$ et $\mathcal{F}_2^3$. Les ensembles $\mathcal{F}_1^1$, $\mathcal{F}_1^2$ et $\mathcal{F}_1^3$ (respectivement $\mathcal{F}_2^1$, $\mathcal{F}_2^2$ et $\mathcal{F}_2^3$) sont les mêmes que ceux du scénario 1. Lorsque nous calculons l'ensemble $\mathcal{F}$ pour les ensembles $\mathcal{F}_1^3$ et $\mathcal{F}_2^3$ avec l'Algorithme 4, nous obtenons une erreur de dépassement mémoire après 5 minutes de temps d'exécution. Pour les mêmes ensembles, l'Algorithme 5 met environ 3 secondes et *cover*$\mathcal{F}$ contient 92 XFD. Puisque les tests concernant la ligne 5 n'ont pas été concluant pour l'Algorithme 4, nous avons effectué d'autres tests sur une simplification de l'arbre $\mathcal{T}$. La simplification a consisté à supprimer les feuilles de l'arbre $\mathcal{T}$ pour $k = 2$, c'est à dire les nœuds $A_{2,2}$, $B_{2,2}$, $D_{2,2}$, $D_{3,2}$ et $E_{2,2}$, de façon à travailler avec des ensembles $\mathbb{P}_1$ et $\mathbb{P}_2$ qui ont moins de chemins. Les résultats de ces tests sont révélés aux lignes 2 et 3 du Tableau 4.9. L'arbre considéré à la ligne 3 contient les nœuds $F_{1,2}$ et $G_{1,2}$ en plus des nœuds de l'arbre considéré à la ligne 2. Comme espéré dans tous les cas, l'Algorithme 5 est beaucoup plus efficace que l'Algorithme 4. De plus, la taille de $\mathcal{F}$ croît de façon exponentielle alors que celle de *cover*$\mathcal{F}$ est polynomiale.

Nos expériences confirment la complexité présentée dans les sections 4.7.2 et 4.7.5, et renforcent l'importance de calculer un ensemble de XFD plus petit *cover*$\mathcal{F}$ au lieu de son équivalent $\mathcal{F}$ qui contient un trop grand nombre de XFD. Le pire des cas arrive lorsque les documents ont beaucoup de chemins équivalents et les fermeture des ensembles de chemins contiennent beaucoup de chemins. Nous avons utilisé l'algorithme décrit dans la sec-

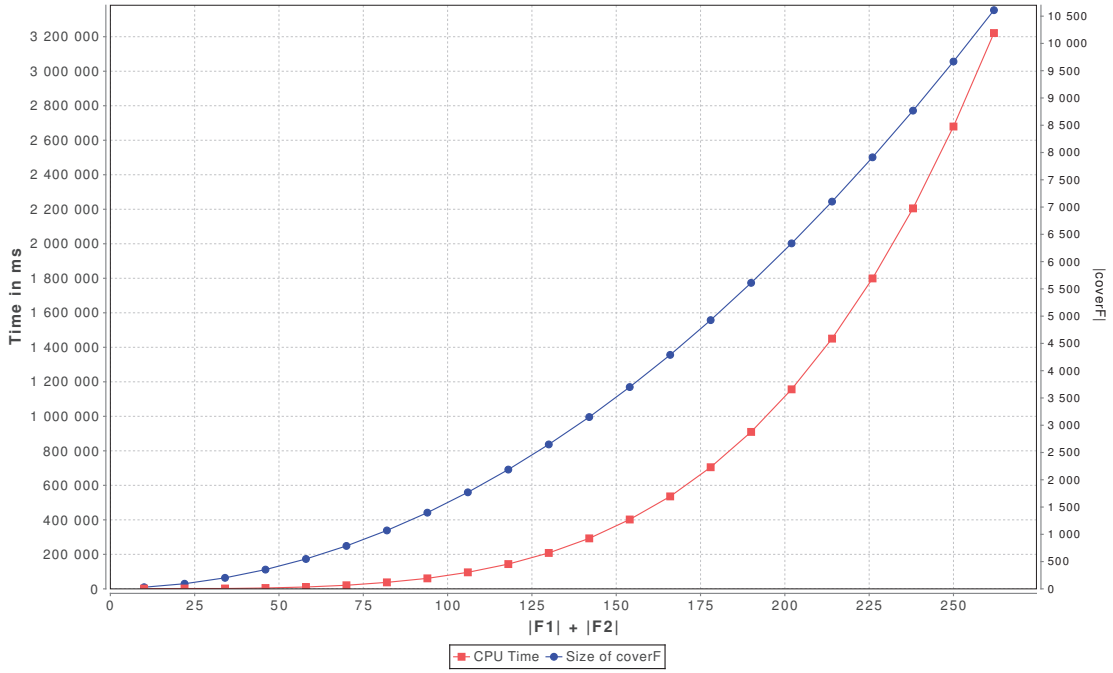


FIGURE 4.14 – *Scénario 2 : Temps CPU nécessaire pour le calcul de cover$\mathcal{F}$ et l'évolution de sa taille*

	$\lvert\mathbb{P}_1\rvert+\lvert\mathbb{P}_2\rvert$	$\lvert\mathcal{F}_1\rvert$	$\lvert\mathcal{F}_2\rvert$	$\lvert\mathcal{F}_1^+\rvert$	$\lvert\mathcal{F}_2^+\rvert$	$\lvert\mathcal{F}\rvert$	t_1 (ms)	$\lvert cover\mathcal{F}\rvert$	t_2 (ms)
1	12	5	5	3 835	3 835	5 755	19 211	30	160
2	17	7	7	3 865	3 865	5 785	24 401	32	195
3	18	8	8	3 928	3 928	5 911	26 476	34	204
4	23	10	10	7 670	7 670	11 510	165 460	61	503
5	34	15	15	?	?	?	$>$ 5min	92	2 949

Tableau 4.9 – *Comparaison des deux méthodes : t_1 est le temps nécessaire pour calculer $\mathcal{F}$ et t_2 est le temps nécessaire pour calculer cover$\mathcal{F}$.*

tion 4.6.4 pour tester l'équivalence entre les ensembles $\mathcal{F}$ et $cover\mathcal{F}$ sur plusieurs exemples. Ces tests ont contribué à la validation de la correction de notre méthode pour le calcul de $cover\mathcal{F}$ et confirment la preuve de la section 4.7.4.

4.9 État de l'art

La motivation de ce chapitre est de fournir au système global la capacité de préserver le plus grand ensemble de contraintes locales non contradictoires. Pour accomplir notre objectif, nous utilisons un système d'axiomes.

Plusieurs autres propositions pour définir les dépendances fonctionnelles XML (XFD) existent. Nous pouvons citer [15, 67, 78, 101, 120, 118, 127] comme exemples et [67, 122] pour une comparaison de certaines d'entre elles.

Le langage de chemin utilisé dans [15, 120, 118] autorise seulement la spécification de chemins simples. Certaines propositions de XFD comprennent aussi un système d'axiomes, comme dans [67, 77, 118]. Nous avons décider d'adopter la définition de XFD présenté

dans [27, 82, 66] pour lequel un algorithme de validation a été proposé dans [33]. En effet, [33] offre un algorithme général pour vérifier la satisfaction de différents types de contraintes d'intégrité, incluant les XFD. Une approche complémentaire est présentée dans [64], qui considère l'idée de validation incrémentale via des vérifications statiques de dépendances fonctionnelles suivant des mises à jour. Dans [64], les XFD sont définies comme des requêtes d'arbre, ce qui augmente considérablement la complexité d'une implémentation, et aussi la partie dynamique nécessaire à la vérification (c'est-à-dire en considérant les valeurs existantes dans le document XML) qui doit encore être effectuée sur les parties concernées par les mises à jour.

Un système d'axiomes et un algorithme pour l'implication de clés en XML sont proposés dans [37]. Comme les XFD sont plus générales, nous pouvons exprimer les contraintes de clé dans [37] avec nos XFD. Nous avons aussi vérifié que notre système d'axiomes peut dériver les règles proposées dans [37] pour l'implication de contraintes de clés XML.

Dans ce chapitre nous proposons un système d'axiomes pour les XFD définies dans[27, 33], ainsi qu'un algorithme efficace pour calculer la fermeture d'un ensemble de chemins. Notre travail sur le système d'axiome est comparable à celui proposé dans [118]. Les principales différences sont :

(i) nous proposons un langage de chemins plus expressif autorisant l'utilisation de // ;

(ii) nos XFD sont vérifiées par rapport à un contexte et non seulement par rapport à une racine, c'est-à-dire que les XFD peuvent être relatives ;

(iii) nos XFD peuvent être définies en prenant en compte deux types d'égalité : l'égalité par valeur, et l'identité de nœud ;

(iv) nous utilisons de simples concepts (comme les branches et la projection) qui, nous le pensons, permettent de prouver que le système d'axiomes est correct et complet d'une manière simple.

Nous utilisons notre système d'axiomes dans le développement d'un outil pratique : pour filtrer des dépendances XFD locales dans le but d'obtenir un ensemble contenant que des XFD ne pouvant pas être violées par aucun des documents locaux. L'objectif de notre système d'axiomes est de traiter des documents provenant des systèmes locaux, mais aussi d'effectuer la fusion de données. Dans ce contexte, notre travail présente un aspect intéressant puisque nous ne nous intéressons pas à mettre ensemble les informations des sources locaux, mais plutôt à les manipuler. C'est pour cela que notre approche ne traite pas les données, mais traite seulement les contraintes disponibles (les XFD dans notre cas). Il existe des travaux qui considèrent l'intégration de base de données locales en globale. Ces propositions viennent habituellement avec l'idée de la fusion de données. La fusion de données XML est considérée dans [41, 93]. Dans [93], l'auteur considère le problème suivant : lorsque l'on fait la fusion des données de plusieurs systèmes locales avec chacune ayant le même ensemble de XFD, une XFD $f : X \rightarrow A$ peut être violée en considérant l'ensemble des données de la fusion. L'auteur propose une approche probabiliste pour décider des données qui sont les plus pertinentes. Dans [47], l'échange de données est considérée. Un schéma cible est construit à partir d'un schéma source en se servant d'un mapping. L'objectif de leur proposition est de construire une instance sur le schéma cible, basée sur le mapping et le schéma source, afin de répondre à des requêtes sur les données cibles d'une manière cohérente avec les données sources.

La majorité des travaux sur l'évolution de schémas ne prennent pas en compte l'évolution des contraintes d'intégrité, qui ont un rôle important dans la maintenance de la consistance

des données, associées aux schémas. Dans [69], une perspective sur l'adaptation des dépendances fonctionnelles XML suite à l'évolution de schéma est proposée. Cette perspective a été réalisée dans [101]. L'évolution de schéma proposée dans [42] est la mieux adaptée à notre proposition ; il est donc possible de combiner leur évolution de schéma avec notre filtre de XFD afin de générer un ensemble de contraintes (structure et intégrité) permettant l'interopérabilité de systèmes.

4.10 Conclusion

Dans ce papier nous sommes motivés par les applications sur un environnement multi-systèmes et nous présentons une méthode pour établir le plus grand ensemble de XFD qui doit être satisfait par tous les documents de chaque système local respectant déjà les XFD locaux. Une originalité importante de notre travail, est qu'il ne traite pas les données, mais traite seulement les dépendances fonctionnelles disponibles. Notre approche ne s'intéresse pas seulement aux applications sur les multi-systèmes, mais aussi à une perspective d'évolution conservatrice de contraintes. Pour atteindre notre but, un nouveau système d'axiomes élaboré pour les XFD définies avec un contexte et deux types d'égalité, a été introduit et prouvé correct et complet lorsqu'on considère les arbres complets.

Comme directions futures de notre travail, nous mentionnons :

- l'extension de notre méthode à d'autres types de contraintes d'intégrité comme les dépendances d'inclusion ;
- un calcul incrémental de $cover\mathcal{F}$, suivant l'évolution des contraintes locales ou systèmes locaux ;
- la détection de XFD locales qui ne sont pas ajoutées dans $cover\mathcal{F}$ mais qu'on peut inclure dans $cover\mathcal{F}$ en corrigeant par exemple les documents XML qui ne respectent pas ces XFD ;
- l'implantation d'un validateur de XFD sur les systèmes locaux, dans une approche map-reduce, en considérant l'ensemble $cover\mathcal{F}$ comme ensemble de contraintes qui vont être vérifiées sur les données du multi-systèmes.

Correction d'un document XML par rapport à un schéma 5

Sommaire

5.1 Préliminaires . . . 88
5.2 Opérations sur les arbres XML . . . 92
5.3 Correction d'un document XML . . . 97
5.3.1 Calcul de distance entre deux arbres . . . 97
5.3.2 Calcul des corrections pour un mot dans un dictionnaire de mots . . . 99
5.3.3 Exemple de correction d'un document XML . . . 101
5.3.4 Présentation de l'algorithme de correction d'un document XML . . . 106
5.3.5 Complexité . . . 112
5.4 Discussion, adaptations et extensions . . . 114
5.5 Résultats expérimentaux . . . 117
5.6 Travaux liés à la correction de document . . . 124
5.6.1 Mesurer la distance entre un document XML et un schéma . . . 125
5.6.2 Trouver une seule correction, ou toutes les corrections de coût minimal . . . 127
5.6.3 Calculer un ensemble de séquences d'opérations d'édition pour corriger des documents XML vers un schéma . . . 128
5.6.4 La revalidation et la correction après des mises à jour du document ou du schéma . . . 128
5.7 Conclusion . . . 129

Dans ce chapitre seront présentés nos travaux sur la correction de document XML par rapport à un schéma. Si nous prenons l'exemple d'un mot qui est tapé dans un éditeur de texte (ayant un correcteur orthographique) et que ce mot est erroné, l'éditeur peut nous proposer des corrections possibles pour ce mot. Nos travaux consistent à faire la même chose sur les arbres (qui remplacent les mots) et à proposer des corrections par rapport à un langage d'arbres (qui remplacent le dictionnaire de mots). La distance d'édition d'arbres est utilisée pour mesurer à quel point les solutions proposées sont proches de l'arbre à corriger. Les travaux sur la correction d'arbres ont débuté avec la thèse d'Ahmed Cheriat. Dans sa thèse [46], ils se sont intéressés à la validation d'un document XML t par rapport à un schéma et au moment où ils trouvent des sous-arbres de t qui ne sont pas valides alors une routine de correction est utilisée pour rendre valide les sous-arbres incorrects avant de poursuivre la validation de t. Les résultats de ces travaux sont dans [28, 29]. Nous y reviendrons dans l'état de l'art de ce chapitre (Section 5.6). Pendant mon année d'ingénieur dans le projet ANR Codex, effectué juste avant de commencer ma thèse en Octobre 2011, j'ai eu à

travailler avec Béatrice Bouchou et Agata Savary sur la correction de documents XML dans un contexte plus général. L'idée était d'avoir une méthode permettant de trouver tous les arbres t' valides par rapport à un schéma S et qui sont à une distance de t ne dépassant pas un seuil fixé. Aussi la méthode doit pouvoir fournir les différents changements à appliquer à t pour obtenir les arbres t'. Les applications aux problèmes de correction de document sont très variées, comme l'a été montré dans [114], et comprennent :

1. l'échange de données (XML data exchange en anglais) et l'intégration de données,
2. la recherche et la composition de service web,
3. l'adaptation d'un document XML par rapport à une base de données [108, 117],
4. l'exécution de requêtes consistantes sur des documents XML inconsistants [104],
5. la classification de document XML [125], ou comment ranger un document XML par rapport à un ensemble de DTDs [19, 20, 115],
6. l'évolution de documents XML et de schéma XML [28, 29, 30, 31, 65, 102, 109, 110].

J'ai continué par travailler sur la correction de documents pendant ma première année de thèse, et ces travaux ont abouti à une publication [6] dans le journal international « The Computer Journal », apparue récemment (Mai 2014) dans son volume 57 [7]. Ce chapitre basé sur notre publication [7] est dédié à une présentation détaillée d'un algorithme pour corriger les documents XML : les principes, les algorithmes, des preuves de propriétés et des résultats expérimentaux sont fournis pour une compréhension totale de son fonctionnement. Notre algorithme est unique dans sa complétude dans le sens où étant donné un seuil th, l'algorithme trouve toutes les corrections possibles t' valides par rapport à S telles que la distance entre t et t' est inférieur à th. Aussi il est possible (en l'absence du seuil th) de trouver toutes les solutions qui sont à la plus petite distance de t. Les principales particularités de notre approche sont présentées dans l'état de l'art de ce chapitre (Section 5.6) tout en nous positionnant par rapport aux autres travaux. Nous pouvons aussi affirmer que notre travail est le premier cas d'une présentation à part entière d'une solution dans cet important domaine, même s'il y a plusieurs propositions qui ont été publiées ces dix dernières années.

Un outil nommé *XMLCorrector* a été développé à partir des algorithmes de ce chapitre et est disponible en ligne [1] sous la licence GNU LPL v3. Cette licence autorise la modification du code source pour l'adapter à une application particulière, ou étendre ses fonctionnalités (par exemple pour traiter les XML schemas (XSD) en suivant les directives fournies en Section 5.4). L'outil *XMLCorrector* est présenté dans la Section 7.1 du chapitre 7.

Le chapitre est organisé comme suit. Dans les section 5.1 et 5.2 nous présentons toutes les définitions pour pouvoir comprendre notre algorithme, présenté en Section 5.3. Dans la Section 5.4, une discussion est effectuée sur l'algorithme de correction et sur des extensions et adaptations possibles de cet algorithme. Nous détaillons ensuite les résultats expérimentaux en Section 5.5. Nous terminons par une discussion sur les travaux liés en Section 5.6 et une conclusion en Section 5.7.

5.1 Préliminaires

Dans ce chapitre nous considérons un document XML comme un arbre ordonné d'arité non borné, que nous appelons arbre XML. La définition est donnée dans le Chapitre 2, Définition 2.4. L'exemple suivant fait un rappel des notations et fonctions définies sur un arbre XML t.

1. Page web de *XMLCorrector* : `http://www.info.univ-tours.fr/~savary/English/xmlcorrector.html`

Exemple 5.1 Dans l'arbre XML t de la Figure 5.1 nous avons :

- $\Sigma = \{root, a, b, c, d\}$
- $Pos(t) = \{\epsilon, 0, 0.0, 0.1, 1, 1.0, 2, 2.0\}$
- $t = \{(\epsilon, root), (0, a), (0.0, c), (0.1, d), (1, b), (1.0, c), (2, b), (2.0, c)\}$
- $t(\epsilon) = root,\ t(0) = a,\ t(0.0) = c$, etc.
- $leaves(t) = \{0.0, 0.1, 1.0, 2.0\}$
- $|t| = 8,\ \ \bar{t} = 3$

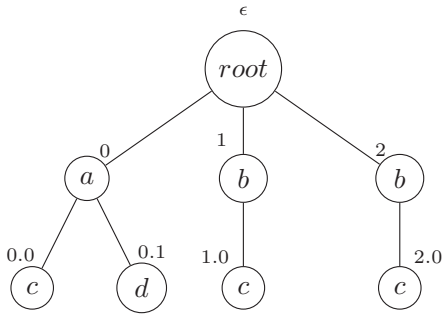


FIGURE 5.1 – *Arbre XML*

La Figure 5.2 montre le sous-arbre $t|_1$ et l'arbre partiel $t\langle 1\rangle$ de l'arbre t de la Figure 5.1.

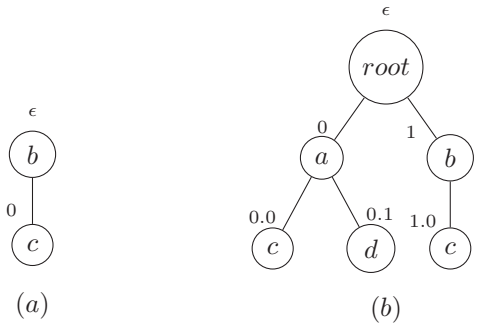


FIGURE 5.2 – *(a) Sous-arbre $t|_1$ et (b) arbre partiel $t\langle 1\rangle$ de l'arbre t de la Figure 5.1.*

⊠

Les documents XML que nous représentons sous forme d'arbres, doivent vérifier des contraintes de schémas qui sont exprimées par des DTD ou XML Schema du W3C ou Relax-NG. Dans le chapitre 2, nous avons vu que ces schémas sont exprimés dans la théorie par des grammaires d'arbres réguliers (RTG). Dans ce chapitre nous allons travailler avec le langage DTD et exprimer les contraintes de schémas par un ensemble de couples de label et d'automate d'états finis (FSA). Le langage DTD n'autorise qu'une seule description pour un label donné. Nous allons donc représenter l'expression régulière de la grammaire par un automate d'états finis. La définition suivante décrit une DTD par un ensemble d'automates où chaque automate est lié à un label.

Définition 5.1 (Description de schéma)
Une **description de schéma** S est un triplet (Σ, *root*, *Rules*) où Σ est un alphabet composé des noms d'éléments ou labels, *root* est l'élément racine, et *Rules* est l'ensemble de couples (a, FSA_a) tel que $a \in \Sigma$ est un label et FSA_a est l'automate d'états finis représentant tous les mots possibles sur Σ formés par les fils du nœud étiqueté par a. Plus précisément :

1. $root \in \Sigma$
2. $Rules = \{(a, FSA_a) \mid a \in \Sigma\}$
3. $\forall\, a \in \Sigma$, $FSA_a = (\Sigma_a, S_a, s_0^a, F_a, \Delta_a)$, est un automate d'états finis avec $\Sigma_a \subseteq \Sigma$.

Les symboles Σ_a, S_a, s_0^a, F_a et Δ_a sont respectivement l'alphabet, l'ensemble des états, l'état initial, et l'ensemble des fonctions de transitions. Pour simplifier, nous allons noter la fonction de transition Δ_a par $FSA_a.\Delta$. □

Exemple 5.2 Dans la Figure 5.3 nous trouvons une description de schéma S dont la racine est l'étiquette *root*.

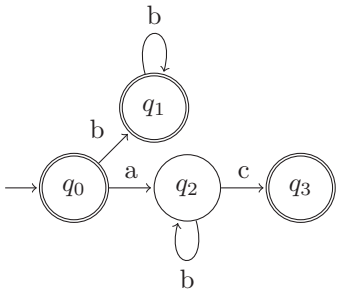
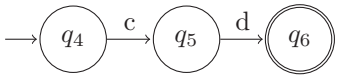
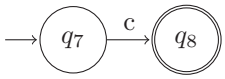
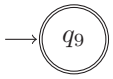
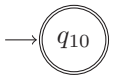
Étiquette	Expression régulière	Automate d'états finis (FSA)
root	$b^*\|ab^*c$	
a	cd	
b	c	
c	ϵ	
d	ϵ	

FIGURE 5.3 – *Un exemple de description de schéma.*

⊠

En vérifiant un arbre XML par rapport aux contraintes de structure imposées par son schéma, l'arbre XML peut être valide c'est-à-dire respecte toutes les contraintes, ou localement valide c'est-à-dire est un sous-arbre d'un arbre valide, ou partiellement valide c'est-à-dire est un arbre partiel d'un arbre valide, ou non valide. Les définitions suivantes formalisent ce que c'est qu'un arbre valide, un arbre localement valide et un arbre partiellement valide.

Définition 5.2 (Arbre localement valide)
Soit une description de schéma $S = (\Sigma, root, Rules)$. Un arbre XML t est **localement valide** par rapport à S si et seulement si ses labels appartiennent à Σ, et ils respectent les contraintes définies dans l'ensemble $Rules$. Formellement nous avons :

1. $\forall\, p \in Pos(t)$, $t(p) \in \Sigma$.
2. $\forall\, p \in Pos(t) \setminus leaves(t)$, le mot $t(p.0)t(p.1)\ldots t(p.(\overline{t|_p} - 1)) \in L(FSA_{t(p)})$, c'est-à-dire le mot formé par les labels des positions fils de p est accepté par l'automate associé au label de la position p.

3. $\forall\, p \in leaves(t)$, $\epsilon \in L(FSA_{t(p)})$, chaque automate associé au label d'une position feuille accepte le mot vide ϵ.

□

Définition 5.3 (Arbre valide)
Soit une description de schéma $S = (\Sigma, root, Rules)$. Un arbre XML t est **valide** par rapport à S si et seulement si l'arbre t est localement valide par rapport à S et $t(\epsilon) = root$.

□

Définition 5.4 (Arbre partiellement valide)
Soit une description de schéma $S = (\Sigma, root, Rules)$. Un arbre XML t est **partiellement valide** par rapport à S si et seulement si l'arbre t est un arbre partiel d'un arbre localement valide par rapport à S.

□

Exemple 5.3 L'arbre t de la Figure 5.1 n'est pas valide par rapport à la description de schéma S de l'Exemple 5.2 car le mot $abb \notin L(FSA_{root})$. Tous les sous-arbres $t|_0, t|_1, t|_2$ sont localement valides par rapport à S. Les arbres partiels $t\langle -1\rangle, t\langle 0\rangle, t\langle 1\rangle, t\langle 2\rangle$ (Définition 2.5) sont aussi partiellement valides par rapport à S. Si le nœud $(0.0, c)$ est supprimé de t alors aucun arbre partiel de t n'est partiellement valide par rapport à S, sauf l'arbre partiel $t\langle -1\rangle$.
⊠

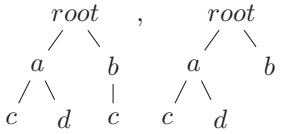
Définition 5.5 (Langages d'arbres)
Soient une description de schéma $S = (\Sigma, root, Rules)$, un label $c \in \Sigma$ et un mot u qui est un préfixe d'un mot $w \in L(FSA_c)$. Nous introduisons les notations suivantes pour les langages d'arbres définis par la description S :

1. $L(S)$ dénote l'ensemble de tous les arbres qui sont valides par rapport à S.
2. $L_{loc}(S)$ dénote l'ensemble de tous les arbres qui sont localement valides par rapport à S.
3. $L_{part}(S)$ dénote l'ensemble de tous les arbres qui sont partiellement valides par rapport à S.
4. $L_u^c(S)$ dénote l'ensemble de tous les arbres qui sont partiellement valides par rapport à S tel que le mot formé par les fils de la racine c est u. Précisément nous avons : $L_u^c(S) = \{t \mid t \in L_{part}(S),\, t(\epsilon) = c \text{ et } t(0)\dots t(\bar{t}-1) = u\}$.

□

Nous pouvons remarquer que pour un arbre valide $t \in L(S)$ on a : (i) t est localement valide, (ii) tous les sous-arbres de t sont localement valides et (iii) tous les arbres partiels de t sont partiellement valides.

Exemple 5.4 Prenons comme exemple le schéma S de l'Exemple 5.2 avec la modification suivante : l'expression régulière associée à b est $c|\epsilon$ au lieu c. Nous avons :

$L_{ab}^{root}(S) = \{$  , $\}$

⊠

Nous avons la proposition suivante :

Proposition 5.1

(i) *Si* $u \in L(FSA_c)$ *alors* $L_u^c(S) \subseteq L_{loc}(S)$;
(ii) *si* $u \in L(FSA_{S.root})$ *alors* $L_u^{S.root}(S) \subseteq L(S)$;
(iii) $\bigcup_{u \in L(FSA_{S.root})} L_u^{S.root}(S) = L(S)$.

□

Le point (i) de la Proposition 5.1 nous dit que tout arbre dans l'ensemble $L_u^c(S)$ est un arbre localement valide par rapport à S. Le point (ii) de la Proposition 5.1 nous dit que tout arbre dans $L_u^{S.root}(S)$ est un arbre valide par rapport à S. Et le point (iii) de la Proposition 5.1 nous dit que l'ensemble $L(S)$ est obtenu en faisant l'union de tous les ensembles $L_u^{S.root}(S)$ avec u qui est un mot du langage $L(FSA_{S.root})$.

5.2 Opérations sur les arbres XML

Il est possible de faire évoluer ou modifier un arbre en se servant d'une ou plusieurs opérations d'éditions de nœuds que sont l'ajout, la suppression et le renommage d'un nœud.

Nous allons utiliser des ensembles de positions qui caractérisent des positions spéciales liées à une position donnée.

Définition 5.6 (Ensemble de positions pour la modification)
Soient t un arbre XML, p une position telles que $p \in Pos(t)$ et $p = \epsilon$ ou $p = u.i$ (avec $u \in \mathbb{N}^*$ et $i \in \mathbb{N}$). Nous sommes intéressés par les ensembles de positions suivantes dans t :

- La **frontière d'insertion** de t, notée $InsFr(t)$, est l'ensemble des positions qui ne sont pas dans t dans lesquelles il est possible d'opérer une insertion. Précisément nous avons :
  1. $InsFr(t_\emptyset) = \{\epsilon\}$
  2. Si $t \neq t_\emptyset$ alors $InsFr(t) = \{v.j \notin Pos(t) \mid v \in Pos(t), j \in \mathbb{N}$ et $[(j = 0)$ ou $(j \neq 0$ et $v.(j-1) \in Pos(t))]\}$
- L'ensemble des **positions à changer** par rapport à p, noté $ChangePos_p(t)$, est l'ensemble des positions qui doivent être soit supprimées, soit décalées à gauche, soit décalées à droite en cas d'insertion ou de suppression de nœud à la position p. Précisément nous avons :
  1. $ChangePos_\epsilon(t) = \{\epsilon\}$
  2. Si $p \neq \epsilon$ alors $ChangePos_p(t) = \{w \mid w \in Pos(t), p = u.i, w = u.k.u', i \leq k < \bar{t}_u$ et $u' \in \mathbb{N}^*\}$
- L'ensemble des **positions décalées à droite** par rapport à p, noté $ShiftRightPos_p(t)$, est l'ensemble des positions qui sont obtenues après le décalage vers la droite d'une partie de t à cause de l'insertion d'un nœud à la position p. Précisément nous avons :
  1. $ShiftRightPos_\epsilon(t) = \emptyset$
  2. Si $p \neq \epsilon$ alors $ShiftRightPos_p(t) = \{w \mid p = u.i, w = u.(k+1).u', u.k.u' \in Pos(t), i \leq k < \bar{t}|_u$ et $u' \in \mathbb{N}^*\}$
- L'ensemble des **positions décalées à gauche** par rapport à p, noté $ShiftLeftPos_p(t)$, est l'ensemble des positions qui sont obtenues après le décalage vers la gauche d'une partie de t à cause de la suppression d'un nœud à la position p. Précisément nous avons :

1. $ShiftLeftPos_{\epsilon}(t) = \emptyset$.
2. Si $p \neq \epsilon$ alors $ShiftLeftPos_p(t) = \{w \mid p = u.i, w = u.(k-1).u', u.k.u' \in Pos(t), i+1 \leq k < t\rceil_u$ et $u' \in \mathbb{N}^*\}$.

□

Exemple 5.5 Pour l'arbre XML t de la Figure 5.1 nous avons :

- $InsFr(t) = \{0.0.0, 0.1.0, 0.2, 1.0.0, 1.1, 2.0.0, 2.1, 3\}$
- $ChangePos_1(t) = \{1, 1.0, 2, 2.0\}$
- $ShiftRightPos_1(t) = \{2, 2.0, 3, 3.0\}$
- $ShiftLeftPos_1(t) = \{1, 1.0\}$
- $ChangePos_2(t) = \{2, 2.0\}$
- $ShiftRightPos_2 = \{3, 3.0\}$
- $ShiftLeftPos_2 = \emptyset$

⊠

Définition 5.7 (Opérations d'édition de nœud)
Soient un alphabet Σ et le caractère réservé $/ \notin \Sigma$, une **opération d'édition de nœud** ed est le triplet (op, p, l) où $op \in \{relabel, add, delete\}$, $p \in \mathbb{N}^*$ et $l \in \Sigma \cup \{/\}$. Soit un arbre XML t, l'opération d'édition de nœud ed est **définie sur** t si et seulement si l'une des conditions suivantes est satisfaite :

- $op = relabel$, $l \in \Sigma$ et $p \in Pos(t)$
- $op = add$, $l \in \Sigma$ et $p \in Pos(t) \setminus \{\epsilon\} \cup InsFr(t)$
- $op = delete$, $l = /$ et $p \in leaves(t)$

Soit une opération d'édition de nœud ed, une ed**-derivation** D_{ed} est une fonction partielle de tout arbre t sur lequel ed est définie vers un autre arbre t', notée $t \xrightarrow{D_{ed}} t'$ ou simplement $t \xrightarrow{ed} t'$, qui transforme t en t' selon les conditions suivantes :

- Pour l'opération **relabel**, la dérivation remplace l'étiquette de la position p. Si $ed = (relabel, p, l)$ alors :
  1. $Pos(t') = Pos(t)$,
  2. $t'(p) = l$,
  3. $\forall\, p' \in Pos(t') \setminus \{p\}$, $t'(p') = t(p')$.
- Pour l'opération **add**, la dérivation insère un nœud à la position p en décalant vers la droite les positions à droite de p. Si $ed = (add, p, l)$ alors :
  1. $Pos(t') = Pos(t) \setminus ChangePos_p(t) \cup ShiftRightPos_p(t) \cup \{p\}$,
  2. $t'(p) = l$,
  3. $\forall\, p' \in (Pos(t) \setminus ChangePos_p(t))$, $t'(p') = t(p')$,
  4. $\forall\, p' \in ShiftRightPos_p(t)$, $[p' = u.(k+1).u'$ et $k \in \mathbb{N}$ et $u, u' \in \mathbb{N}^*] \Rightarrow t'(p') = t(u.k.u')$.
- Pour l'opération **delete**, la dérivation supprime la feuille nœud à la position p en décalant vers la gauche les positions à droite de p. Si $ed = (delete, p, /)$ alors :
  1. $Pos(t') = Pos(t) \setminus ChangePos_p(t) \cup ShiftLeftPos_p(t)$,
  2. $\forall\, p' \in (Pos(t) \setminus ChangePos_p(t))$, $t'(p') = t(p')$,

3. $\forall\, p' \in \mathit{ShiftLeftPos}_p(t), [p' = u.(k-1).u'$ et $k \in \mathbb{N}$ et $u, u' \in \mathbb{N}^*] \Rightarrow t'(p') = t(u.k.u')$.

□

Exemple 5.6 Considérons l'arbre XML t de la Figure 5.1 et l'opération d'édition de nœud $ed = (add, 1, a)$ définie sur t. La dérivation de ed transforme t en t' illustré en Figure 5.4.

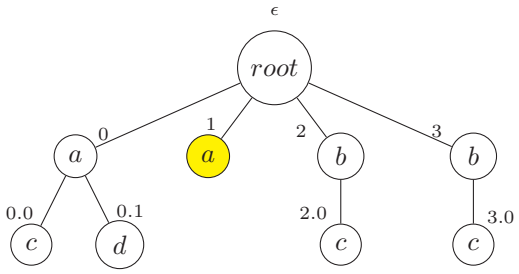


FIGURE 5.4 – *Résultat de la dérivation pour $ed = (add, 1, a)$ sur l'arbre t en Figure 5.1.*

⊠

Définition 5.8 (Séquence d'opérations d'édition de nœud)
Soient t un arbre XML, $n \geq 0$ et $ed_1, ed_2, \ldots ed_n$ des opérations d'édition de nœud. La **séquence d'opérations d'édition de nœud** $nos = \langle ed_1, ed_2, \ldots ed_n \rangle$ est **définie sur** t si et seulement si il existe une séquence d'arbres $t_0, t_1, \ldots, t_n$ telle que :

- $t_0 = t$
- $\forall\, 0{<}k{\leq}n$, ed_k est définie sur t_{k-1} et $t_{k-1} \xrightarrow{ed_k} t_k$

Soit une séquence d'opérations d'édition de nœud nos, la **dérivation** D_{nos} est une fonction partielle de tout arbre t, sur lequel nos est défini, vers un autre arbre t' notée $t \xrightarrow{D_{nos}} t'$ ou simplement $t \xrightarrow{nos} t'$. La fonction partielle D_{nos} transforme t en t' si et seulement si il existe une séquence d'arbres $t_0, t_1, \ldots, t_n$ comme défini ci-dessus et $t_n = t'$.
La **séquence vide d'opérations d'édition de nœud** est notée $\langle\rangle$ et on a $t \xrightarrow{\langle\rangle} t$.
Deux séquences d'opérations d'édition de nœud nos_1 et nos_2 sont dites **équivalentes** si et seulement si pour tout arbre t sur lequel elles sont définies, la dérivation de t par nos_1 et la dérivation de t par nos_2 produisent le même arbre : $nos_1 \equiv nos_2$ si et seulement si $\forall\, t$, [nos_1 et nos_2 sont définies sur t, $t \xrightarrow{nos_1} t_1$ et $t \xrightarrow{nos_2} t_2] \Rightarrow t_1 = t_2$.

□

Exemple 5.7 Considérons l'arbre t de la Figure 5.1 et la séquence d'opérations d'édition de nœud $nos{=}\langle(relabel, 0.1, c), (delete, 0.0, /), (relabel, 0, b)\rangle$. La séquence nos est définie sur t et sa dérivation produit l'arbre t'_1 de la Figure 5.12. Nous avons aussi $nos \equiv \langle(delete, 0.0, /), (relabel, 0, b)\rangle \equiv \langle(relabel, 0, b), (delete, 0.0, /)\rangle$.

⊠

Pour un arbre XML donné t et un ensemble de séquences d'opérations d'édition de nœud $Nos{=}\{nos_1, \ldots, nos_n\}$ définies sur t, la dérivation de t par Nos produit l'ensemble des arbres $\{t_1, \ldots, t_n\}$ tel que pour chaque $nos_i \in Nos$ on a $t \xrightarrow{nos_i} t_i$. La dérivation de t par Nos est notée $t \xrightarrow{Nos} \{t_1, \ldots, t_n\}$. Nous allons maintenant introduire des séquences d'opérations d'édition de nœud particulières correspondant à des opérations d'édition plus « haut niveau » qui permettent l'ajout ou le retrait de sous-arbres entiers au lieu de simple

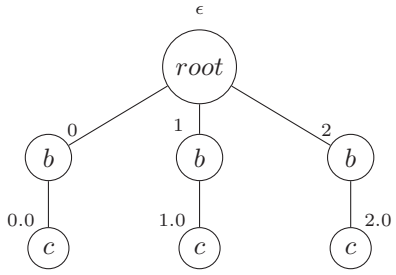


FIGURE 5.5 – *Arbre XML produit par la dérivation de la séquence nos sur l'arbre t de la Figure 5.1*

nœud. Par exemple la séquence d'ajout aux positions 1, 1.0 et 1.1 dans un arbre donné peut être vu comme une seule opération où l'on insère un sous-arbre de trois nœuds à la position 1. De façon similaire, la séquence de suppression aux positions 2.0 et 2 correspond à l'opération de suppression du sous-arbre à la position 2.

Définition 5.9 (Opérations d'édition d'arbre)
Soient l'alphabet Σ, une **opération d'édition d'arbre** *ted* est le triplet (op, p, τ) où $op \in \{insert, remove, replace\}$, $p \in \mathbb{N}^*$ et τ est un arbre XML dont les étiquettes sont dans Σ. Pour un arbre XML t, l'opération d'édition d'arbre *ted* est **définie sur** t si et seulement si l'une des conditions suivantes est satisfaite :

- $op = insert$ et $p \in Pos(t) \setminus \{\epsilon\} \cup InsFr(t)$
- $op = remove$, $p \in Pos(t)$ et $\tau = t_\emptyset$
- $op = replace$ et $p \in Pos(t)$

Soit une opération d'édition d'arbre *ted*, une ***ted*-derivation** D_{ted} est une fonction partielle de tout arbre t sur lequel *ted* est définie vers un autre arbre t', notée $t \xrightarrow{D_{ted}} t'$ ou simplement $t \xrightarrow{ted} t'$, qui transforme t en t' selon les conditions suivantes :

- Pour l'opération **insert**, la dérivation insère un nouvel arbre à la position p en décalant vers la droite les positions à droite de p. Si $ted = (insert, p, \tau)$ alors :
  $t = t_0 \xrightarrow{(add, p.v_1, \tau(v_1))} t_1 \xrightarrow{(add, p.v_2, \tau(v_2))} t_2 \ldots \xrightarrow{(add, p.v_n, \tau(v_n))} t_n = t'$ où $v_1, \ldots, v_n$ sont les positions de τ visitées dans l'ordre préfixe.
- Pour l'opération **remove**, la dérivation supprime le sous-arbre enraciné à la position p en décalant vers la gauche les positions à droite de p. Si $ted = (remove, p, t_\emptyset)$ alors :
  $t = t_0 \xrightarrow{(delete, p.v_1, /)} t_1 \xrightarrow{(delete, p.v_2, /)} t_2 \ldots \xrightarrow{(delete, p.v_n, /)} t_n = t'$ où $v_1, \ldots, v_n$ sont les positions du sous-arbre $t|_p$ visitées dans l'ordre postfixe inverse (de droite à gauche).
- Pour l'opération **replace**, une dérivation possible est de supprimer le sous-arbre enraciné en p puis d'insérer τ à sa place, mais la dérivation de coût minimal est de *corriger* le sous-arbre en p vers le sous-arbre τ en appliquant l'algorithme décrit en section 5.3.

□

Toute opération d'édition d'arbre *ted* peut s'exprimer en terme de séquence d'opérations d'édition de nœud *nos* selon la Définition 5.9. Nous disons que *ted* et *nos* sont **t-equivalentes**, noté $ted \equiv_t nos$. D'après la Définition 5.9, à chaque opération d'édition d'arbre correspond exactement une séquence d'opérations d'édition de nœud t-equivalente.

Exemple 5.8 L'opération d'édition d'arbre $ted = (insert, 3, \tau_1)$ avec $\tau_1 = \{(\epsilon, b), (0, c)\}$ est définie sur l'arbre t de la Figure 5.1 et elle est t-équivalente à $\langle (add, 3, b), (add, 3.0, c)$.

⊠

Toute dérivation a un coût de transformation non négatif. Nous allons fixer le coût à 1 pour chaque opération d'édition de nœud, mais cela peut être défini autrement dans le cas général et donc varier selon le cas d'application [2]. Nous pouvons par exemple supposer une application où l'opération **relabel** à un coût deux fois plus cher que le coût de l'opération **add**.

Définition 5.10 (Coût d'une séquence d'opérations)
Pour toute opération d'édition de nœud ed, le coût $cost(ed)$ est défini comme le coût non négatif de dérivation d'un arbre par ed. Soit la séquence d'opérations d'édition de nœud $nos = \langle ed_1, ed_2, \ldots, ed_n \rangle$, le coût de nos est défini par $Cost(nos) = \Sigma_{i=1}^{n}(cost(ed_i))$. Le coût d'une opération d'édition d'arbre ted est égal au coût de la séquence nos t-équivalente, c'est-à-dire $Cost(ted) = Cost(nos)$ où $ted \equiv_t nos$. Soit l'ensemble de séquences d'opérations d'édition de nœud Nos, le coût minimum de Nos est défini par $MinCost(Nos) = \min_{nos \in Nos}\{Cost(nos)\}$.

□

Nous arrivons aux définitions de la distance entre deux arbres et entre un arbre et un langage d'arbres, fondamentales pour l'algorithme de correction de document. On notera que la première correspond bien à la distance entre deux arbres proposée par [100].

Définition 5.11 (Distance d'arbres)
Soient t et t' des arbres et soit $NOS_{t \to t'}$ l'ensemble de toutes séquences d'opérations d'édition de nœud nos telles que $t \xrightarrow{nos} t'$. La distance entre deux arbres t et t' est définie par :

$$dist(t, t') = MinCost(NOS_{t \to t'}).$$

La distance entre un arbre t et un langage d'arbres L est définie par :

$$DIST(t, L) = \min_{t' \in L}\{dist(t, t')\}.$$

□

Exemple 5.9 Considérons l'arbre t de la Figure 5.1 et le schéma S de l'Exemple 5.2. Nous avons $DIST(t, L) = dist(t, t'_2) = Cost(\langle (add, 3, c) \rangle) = 1$, avec t'_2 de la Figure 5.12.

⊠

Définition 5.12 (Ensemble d'arbres solution pour la correction)
Soient un arbre t, une description de schéma $S = (\Sigma, root, Rules)$ et un seuil th ($th \geq 0$). **L'ensemble correction** de t par rapport à S en dessous du seuil th est défini comme l'ensemble de tous les arbres valides et dont la distance avec t n'est pas plus grande que th. Formellement nous avons :

$$L_t^{th}(S) = \{t' \mid t' \in L(S), dist(t, t') \leq th\}.$$

□

Le but de l'algorithme que nous allons présenter dans la section 5.3 est de montrer comment obtenir l'ensemble d'arbres solution pour un arbre t donné. L'algorithme calcule l'ensemble de toutes les séquences d'opérations d'édition de nœud à partir desquelles on

2. Dans le prototype développé pour l'algorithme de correction de document le coût de chaque opération est un paramètre

obtient les arbres t' qui sont dans l'ensemble correction de t. Chaque séquence d'opérations de nœud peut être exprimée en opérations équivalentes à celles définies par Selkow [100] (renommage de nœud, insertion de sous-arbre et suppression de sous-arbre). La traduction entre les séquences d'opérations d'édition de nœud et d'arbre est trivialement obtenue à partir de la Définition 5.9. En se servant de la Définition 5.5, **l'ensemble correction** peut être aussi défini comme suit :

$$L_t^{th}(S) = \left\{ t' \mid t' \in \bigcup_{u \in L(FSA_{S.root})} L_u^{S.root}(S) \text{ et } dist(t, t') \leq th \right\} \quad (\alpha)$$

Nous allons dans la suite prouver que la Définition 5.12 et (α) sont équivalentes.

5.3 Correction d'un document XML

La génération de la correction d'un document XML par rapport à un schéma XML est basée sur deux algorithmes fondamentales. Le premier concernant les arbres, est la proposition de Selkow pour la distance d'édition entre deux arbres [100]. Le second concerne les mots (chaînes de caractères), a été proposé par Oflazer et permet de trouver la correction d'erreur d'orthographe [92]. Ce deuxième algorithme fait une exploration dynamique d'un automate d'états finis (FSA) qui représente un dictionnaire de mots. Nous allons faire une brève introduction de ces deux algorithmes fondamentales avant de proposer notre algorithme pour la correction de document.

5.3.1 Calcul de distance entre deux arbres

Le problème de la distance d'édition entre deux arbres abordé par Selkow [100] généralise le problème du calcul de la distance d'édition entre deux mots [121] à celui entre deux arbres étiquetés non bornés. Trois opérations d'éditions sont utilisées : (i) le renommage de l'étiquette d'un nœud, (ii) la suppression d'un sous-arbre et (iii) l'insertion d'un sous-arbre. Les deux dernières opérations peuvent être décomposées en des séquences d'opérations de suppression de feuille, et d'insertion de feuille respectivement. Un coût est attribué à chacune de ces opérations et le problème consiste à trouver le coût minimal de toutes les séquences d'opérations qui transforment un arbre t en un autre arbre t'. La distance d'édition entre t et t' est donc égale au coût minimal.

Le calcul de la distance d'édition est basé sur une matrice H où chaque case $H[i, j]$ contient la distance d'édition entre deux arbres partiels $t\langle i \rangle$ et $t'\langle j \rangle$. (cf. Figure 5.6(a)). Rappelons que l'arbre partiel $t\langle i \rangle$ d'un arbre t comporte la racine de t et ses sous-arbres $t|_0, \ldots, t|_i$. Nous utilisons $C_{i,j}$ pour dénoter le coût minimal pour transformer $t\langle i \rangle$ en $t'\langle j \rangle$. Selkow a montré que $C_{i,j}$ est le coût minimum des trois séquences d'opérations suivantes :

1. la transformation de $t\langle i \rangle$ en $t'\langle j-1 \rangle$ en insérant le sous-arbre $t'|_j$;
2. la transformation de $t\langle i-1 \rangle$ en $t'\langle j-1 \rangle$ en transformant $t|_i$ en $t'|_j$;
3. la transformation de $t\langle i-1 \rangle$ en $t'\langle j \rangle$ en supprimant le sous-arbre $t|_i$.

La matrice H est calculée colonne par colonne, de la gauche vers la droite et de bas en haut. Ainsi chaque case $H[i, j]$ est déduite à partir de ses trois voisines $H[i-1, j-1]$, $H[i-1, j]$ et $H[i, j-1]$, comme illustré dans la Figure 5.6(b). La case $H[i, j]$ contient la plus petite valeur entre *(1)* la somme de sa voisine gauche et du coût minimum d'insertion du sous-arbre $t'|_j$ (Figure 5.6(b), flèche (1)) ; *(2)* la somme de sa voisine haut-gauche et du coût

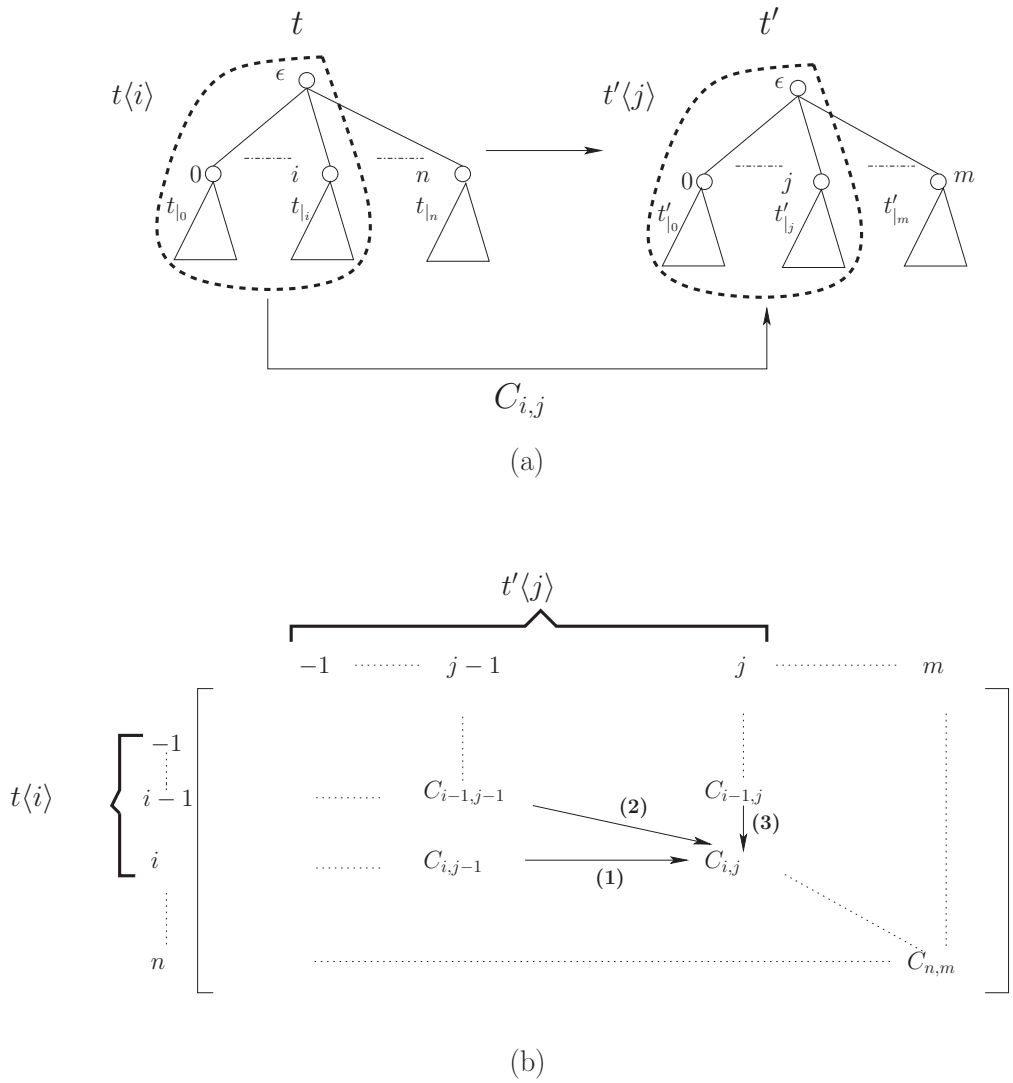


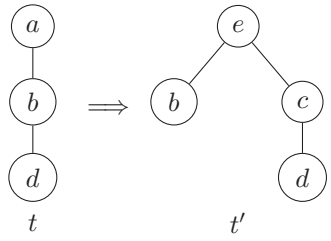
FIGURE 5.6 – *(a) Deux arbres partiels $t\langle i\rangle$ et $t'\langle j\rangle$. (b) Matrice pour la distance d'édition : calcul de la case $H[i,j] = C_{i,j}$.*

minimum de la transformation du sous-arbre $t|_i$ en $t'|_j$ (Figure 5.6 (b), flèche (2)) ; et *(3)* la somme de sa voisine haut et du coût minimum de suppression du sous-arbre $t|_i$ (Figure 5.6 (b), flèche (3)).

Exemple 5.10 Soient t et t' les deux arbres de la Figure 5.7(a). Dans cet exemple nous fixons à 1 le coût de chaque opération d'édition élémentaire (insertion, suppression, renommage de nœud). La matrice d'édition de distance H entre t et t' est donnée dans la Figure 5.7(b). Nous avons l'arbre t du côté des lignes, et l'arbre t' du côté des colonnes. Les lignes et les colonnes de la matrice H sont numérotées comme suit :

1. -1 lorsque la racine d'un arbre est concernée,
2. un entier i lorsque l'enfant à la position i de la racine est concerné. Les indices des lignes et des colonnes sont accompagnées par les étiquettes des nœuds correspondants.

La dernière case en bas à droite ($H[0,1]$) contient la distance d'édition entre t et t', c'est-à-dire le coût de la séquence d'opérations d'édition minimal pour transformer t en t'. Cette séquence est composée des opérations suivantes : le renommage de la racine en e, l'insertion de b comme premier fils de la racine et le renommage du nœud b qui a pour fils d, en



(a)

H		-1	o	1
		e	b	c
-1	a	1	2	4
o	b	3	2	3

(b)

FIGURE 5.7 – *Matrice pour la distance d'édition entre t et t′.*

c. L'ordre de calcul des cases de H est le suivant : $H[-1,-1]$, $H[0,-1]$, $H[-1,0]$, $H[0,0]$, $H[-1,1]$, $H[0,1]$.

⊠

Il est important de remarquer que le calcul de la distance d'édition entre deux arbres t et t' implique le calcul de la distance d'édition entre les sous-arbres de t et t'. La complexité en temps de l'algorithme de Selkow est de $O(\Sigma_{i=0}^{min(d_t,d_{t'})} h_i h'_i)$, où d_t et $d_{t'}$ sont les profondeurs de t et t', h_i et h'_i sont les nombres de nœuds à la profondeur i dans t et t', respectivement.

Le calcul de la distance d'édition d'arbres $dist(t,t')$ de Selkow est notre première notion fondamentale mais nous avons besoin de faire plus de choses : notre but est de calculer les séquences d'opérations de coût minimal pour transformer un arbre XML t qui n'est pas valide par rapport à un schéma XML, en des arbres XML valides. Pour pouvoir atteindre notre but, nous ne calculons pas seulement une distance entre un arbre donné t et un schéma S mais nous calculons des séquences d'édition pouvant transformer t en des arbres valides par rapport à S. De plus, nous nous limitons pas seulement au calcul des séquences minimales car nous cherchons aussi tous les arbres valides t' tels que $dist(t,t') \leq th$, où th est un seuil pour la distance entre t et les arbres t'. Nous allons maintenant introduire la deuxième notion fondamentale qui est un algorithme permettant de rechercher des corrections possibles pour un mot erroné dans un dictionnaire de mots.

5.3.2 Calcul des corrections pour un mot dans un dictionnaire de mots

Étant donné un mot X qui n'appartient pas à un dictionnaire ou langage, l'algorithme d'Oflazer [92] propose toutes les corrections pour X, qui sont à une distance inférieure ou égale à un seuil th du mot X.

Cet algorithme malgré qu'il ne soit pas présenté dans un récent état de l'art [34] sur la correction de mot par rapport à un langage, peut être classé (en se basant sur la classification faite dans [34]) comme une méthode directe basée sur la notion d'arbre préfixe implémenté comme un « tri de mots ». Plus précisément, cette méthode est basée sur une exploration dynamique d'un automate reconnaissant un langage de mots. Un candidat partiel $Y = a_1 a_2 \ldots a_k$ est progressivement généré en concaténant les étiquettes des transitions de A en commençant par l'état initial q_0 jusqu'à un état final. A chaque étape, si l'automate est dans son état q_m, pour compléter le mot Y par l'étiquette b d'une transition sortante de q_m, l'algorithme vérifie si la distance d'édition « cut-off » entre X et le mot $Y = a_1 a_2 \ldots a_k b$ reste inférieure à un seuil th. La distance d'édition « cut-off » entre deux mots est une mesure introduite par [57] qui permet d'optimiser le processus d'exploration de l'automate en le stoppant dès qu'il ne peut plus y avoir de nouvel Y dans le seuil th. Dans cette situation, la

dernière transition est annulée ce qui entraîne que le dernier caractère est retiré de Y et il y a un retour arrière vers l'état q_m et l'exploration continue en essayant une nouvelle transition sortante de q_m. Lorsqu'un état final est atteint, si $dist(X, Y) \leq th$ alors Y est un candidat valide pour la correction du mot X. Notre prochain exemple, que nous pouvons retrouver dans [92], illustre les idées ci-dessus.

Exemple 5.11 La Figure 5.8 montre le graphe $G1$ représentant l'automate d'états finis correspondant à l'expression régulière $E = (aba|bab)^*$ et dans $G2$ nous avons l'exploration de ce graphe en considérant le mot $X = ababa$ qui n'appartient pas au langage $L(E)$. Les trois chemins qui sont entourés dans $G2$, représentent trois corrections possibles *abaaba*, *ababab* et *bababa*. Pour chaque nœud n dans $G2$, nous avons entre crochet la distance d'édition « cut-off » entre le mot incorrect *ababa* et le mot correspondant au chemin reliant l'état initial q_0 à l'état pour le nœud n. Si nous prenons une distance seuil $th = 1$ alors les trois mots entourés sont valides. Nous pouvons noter qu'il n'est pas possible de prolonger ces trois chemins pour obtenir d'autres candidats avec le seuil $th = 1$ car la distance d'édition « cut-off » passe à 2 pour tous les prochains états.

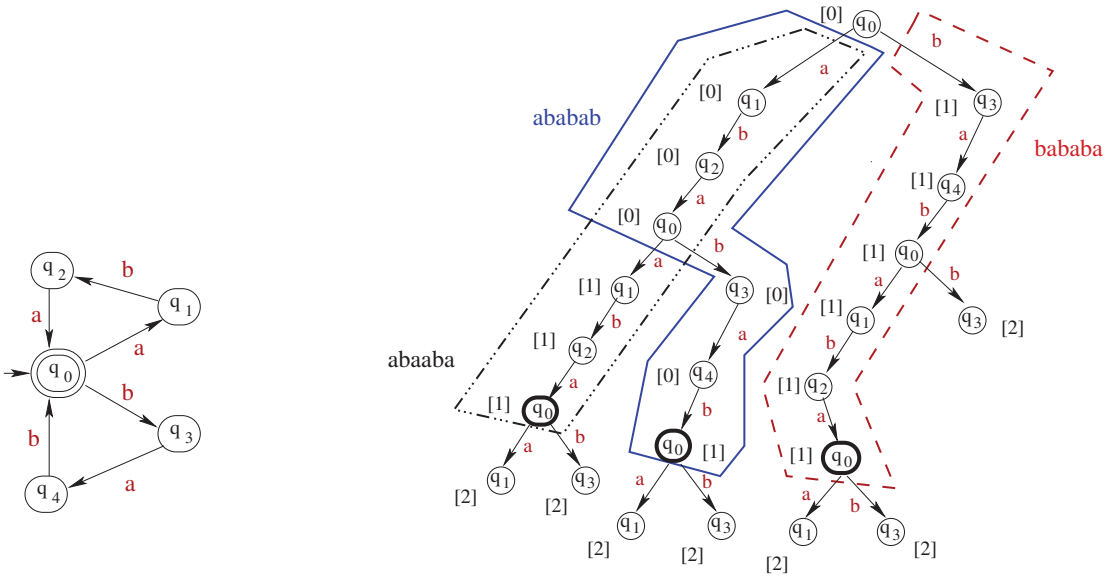


FIGURE 5.8 – *G1 : graphe représentant le FSA qui correspond à* $(aba|bab)^*$, *G2 : graphe représentant l'exploration des chemins pour la correction du mot ababa pour le seuil* $th = 1$

⊠

De la même manière que dans [57], une matrice d'édition de distance H entre X et les candidats potentiels Y est dynamiquement calculée où la case $H[i, j]$ contient la distance entre les préfixes de longueur i et j des deux mots X et (respectivement) Y. La plus-value de la proposition de Oflazer dans [92] réside dans la représentation du lexique/dictionnaire par l'automate A de sorte que, lorsqu'un mot est recherché dans le lexique, les colonnes de la matrice qui correspondent à un même préfixe de mots du lexique ne sont calculées qu'une seule fois.

Pour résumer, l'algorithme de correction de documents que nous avons mis au point combine les propositions fondamentales de [92, 100]. Nous utilisons le calcul de la distance d'édition entre deux arbres de Selkow, défini pour les trois opérations de nœud *relabel*, *add* et *delete*, pour calculer dynamiquement les distances entre l'arbre partiel courant de t et un

arbre partiel généré progressivement par une exploration dynamique des automates à états finis présents dans les règles de transition de l'automate de schéma A. L'arbre partiel joue ici pour les arbres le rôle que joue les préfixes pour les mots. Finalement, nous étendons les deux propositions fondamentales de [92, 100] pour corriger un arbre par rapport à un langage d'arbres de manière similaire à ce qu'a fait Oflazer pour la correction d'un mot par rapport à un langage de mots.

Dans la section suivante, nous allons illustrer notre algorithme de correction de documents par un exemple.

5.3.3 Exemple de correction d'un document XML

Dans cette section, un exemple montre comment nous combinons les deux approches introduites en section 5.3.1 et 5.3.2 pour calculer les séquences d'opérations d'édition de nœuds pour transformer un arbre t en arbres valides t' tels que $dist(t, t') \leq th$.

Soient $\Sigma = \{root, a, b, c, d\}$ un alphabet, t l'arbre XML de la Figure 5.1 et S la description de schéma de la Figure 5.3. L'automate d'états finis associé à la racine *root* correspond à l'expression régulière $b^*|ab^*c$. Remarquons que l'arbre t n'est pas valide par rapport à S car le mot *abb* formé par les fils de la racine de t n'appartient pas au langage $L(b^*|ab^*c)$.

Nous allons calculer l'ensemble des arbres valides $\{t'_1, \cdots, t'_n\}$ ayant une distance par rapport à t plus petite que le seuil $th = 2$. En réalité nous calculons pour chaque arbre t'_k ($k \in [1 \ldots n]$) une séquence d'opérations d'édition de nœud qui permet de l'obtenir à partir de t. Pour trouver les séquences, on se sert d'une matrice de distance d'édition M d'un arbre à un langage d'arbres. Cette matrice M contient des ensembles de séquences d'opérations (ayant des coûts inférieurs au seuil) nécessaires pour transformer les arbres partiels de t en arbres partiels de t'_k (il est possible d'avoir plusieurs séquences). Nous allons utiliser l'arbre t' pour représenter les arbres valides t'_k. La case se trouvant à la ligne i et colonne j est dénotée par $M[i][j]$ ou (i, j). La première case $(0, 0)$ de M contient la séquence d'opérations pour transformer la racine de t en la racine attendue pour les arbres valides appartenant au langage $L(S)$. Dans notre exemple, t a la même racine que la racine établie par le schéma S ce qui fait que la racine de t ne sera pas changée. C'est pour cela que la case $(0, 0)$ de la Figure 5.9 contient la séquence vide dénotée $\langle\rangle$. Par la suite, pour calculer les autres cases (i, j), nous allons nous servir des cases qui ont été déjà calculées. En fait nous concaténons chaque séquence prise dans l'une des cases voisines suivantes : gauche $(i-1, j)$ ou haut-gauche $(i-1, j-1)$ ou haut $(i, j-1)$, avec l'une des trois opérations tout en s'assurant que le coût de la séquence obtenue ne dépasse pas le th :

(i) l'ajout de sous-arbre (dénoté par $\rightarrow$) à partir du contenu de la case à gauche : ses séquences d'opérations sont concaténées avec celle correspondant à l'insertion du sous-arbre $t'|_j$ (Définition 5.9). Plusieurs sous-arbres différents et localement valides peuvent être ajoutés du moment qu'ils ont à leur racine l'étiquette correspondante à la colonne j. Pour cette raison, nous trouvons dans chaque case un ensemble de séquences d'opérations d'édition de nœud.

(ii) la correction de sous-arbre (dénotée par $\searrow$) à partir du contenu de la case haut-gauche : ses séquences d'opérations sont concaténées avec celles correspondant à la correction du sous-arbre $t|_i$ de t en un sous-arbre localement valide de l'arbre candidat en cours de construction. La racine du nouveau sous-arbre aura l'étiquette correspondante à la colonne j. Cette correction est effectuée par un appel récursif de l'algorithme de correction qui construit une autre matrice de distance d'édition.

(iii) la suppression de sous-arbre (dénotée par $\downarrow$) à partir du contenu de la case du haut : ses séquences d'opérations sont concaténées avec celle correspondant à la suppression du

sous-arbre $t|_i$ (Définition 5.9). Dans ce cas une seule séquence est possible en prenant la séquence d'opérations d'édition de nœud t-equivalente.

	M	0 root	1 b	2 b	3 b	**4** ***b***
0	root	$\{\langle\rangle\}$	$\{\langle(add, 0, b), (add, 0.0, c)\rangle\}$	$\emptyset$	$\emptyset$	$\emptyset$
1	a	$\emptyset$	$\{os_1{=}\langle(relabel, 0, b), (delete, 0.1, /)\rangle\}$	$\emptyset$	$\emptyset$	$\emptyset$
2	b	$\emptyset$	$\emptyset$	$\{os_1\}$	$\emptyset$	$\emptyset$
3	b	$\emptyset$	$\emptyset$	$\emptyset$	$\{os_1\}$	$\emptyset$

FIGURE 5.9 – *Contenu de la matrice M*

Soit la matrice M de la Figure 5.9. Elle est construite pas-à-pas en suivant FSA_{root} (Figure 5.3). Ainsi le calcul de M est effectué colonne par colonne. Chaque nouvelle colonne est ajoutée en suivant une transition dans l'automate FSA_{root}. Par exemple dans la Figure 5.9, pour la colonne $j = 1$, la transition (q_0, b, q_1) peut être utilisée et cette colonne sera référencée par l'étiquette b.

Dans la suite nous allons considérer les solutions obtenues en suivant les différentes transitions de FSA_{root}.

Correction 1 : Suivant la transition (q_0, b, q_1) de FSA_{root} (Figure 5.3).
La matrice M obtenue est celle de la Figure 5.9.

- Le calcul sur la colonne 0 suit le raisonnement suivant. En allant de la case $(0, 0)$ à la case $(1, 0)$, le sous-arbre de t (Figure 5.1) enraciné en position 0 est supprimé et cette suppression a un coût de 3. Comme le seuil $th = 2$ est dépassé, la case $(1, 0)$ est vide ainsi que toutes les cases en dessous de cette case. La dernière case de la première colonne correspond à la transformation de tout l'arbre t en la racine de l'arbre valide t'.
- Les étiquettes pour les colonnes ($0 < j$) dans M forment un mot u. La Figure 5.9 montre le contenu des cases de la matrice M pour le mot $u = bbbb$. Les lignes pour une nouvelle colonne sont calculées du haut vers le bas. Voici une illustration du calcul des cases pour une colonne en prenant pour exemple la case $(1, 1)$:
  - (i) la case $M[1][0]{=}\emptyset$ ne contient pas de séquences et par conséquent elle ne pourra pas générer de séquences d'opérations pour la case $(1, 1)$.
  - (ii) la case $M[0][0] = \{\langle\rangle\}$ contient la séquence vide de coût 0. La séquence vide est concaténée avec la séquence d'opérations $os_1{=}\langle(relabel, 0, b), (delete, 0.1, /)\rangle$ qui est obtenue en corrigeant le sous-arbre $\{(\epsilon, a), (0, c), (1, d)\}$ à la position 0 dans t ($t|_0$) en un arbre localement valide ayant comme racine b. Le sous-arbre que nous obtenons est $\{(\epsilon, b), (0, c)\}$. Le coût de la séquence os_1 est 2 et ne dépasse pas le seuil $th = 2$. La matrice M' (Figure 5.10) est utilisée pour transformer le sous-arbre $\{(\epsilon, a), (0, c), (1, d)\}$ en $\{(\epsilon, b), (0, c)\}$. M' est en effet construite lors de l'appel récursif fait pour corriger le sous-arbre $t|_0$.
    - (a) Initialement, la correction retenue à la case $M'[0][0]$ concerne le renommage de la racine de $t|_0$ qui change de a en b.
    - (b) Dans la suite $M'[1][0]$ est obtenue en supprimant le nœud c à la position 0 dans $t|_0$. La séquence dans $M'[1][0]$ permet la transformation de $\{(\epsilon, a), (0, c)\}$ en $\{(\epsilon, b)\}$ avec un coût de 2.

(c) La case $M'[2][0]$ indique que la transformation de $\{(\epsilon, a), (0, c), (1, d)\}$ en $\{(\epsilon, b)\}$ a un coût supérieur au th. En effet il fallait rajouter l'opération $(delete, 1, /)$ à la séquence $\{\langle(relabel, \epsilon, b), (delete, 0, /))\rangle\}$ dans $M'[1][0]$, qui a déjà un coût de 2.

(d) La case $M'[0][1]$ résulte de l'ajout du nœud c à la position 0 dans $\{(\epsilon, b)\}$.

(e) La case $M'[1][1]$ propose la transformation de $\{(\epsilon, a), (0, c)\}$ en $\{(\epsilon, b), (0, c)\}$. En partant de la case $M'[0][0]$, comme nous avons déjà renommé a par b, il n'y a plus rien à faire. C'est pour cela que nous obtenons aussi la séquence $\langle(relabel, \epsilon, b)\rangle$ dans $M'[1][1]$. En partant des cases $M'[1][0]$ et $M'[0][1]$ il n'est pas possible d'avoir de nouvelles séquences car les nouveaux coûts vont dépasser le seuil th.

(f) Finalement la case $M'[2][1]$ propose de transformer $\{(\epsilon, a), (0, c), (1, d)\}$ en $\{(\epsilon, b), (0, c)\}$. En partant de la case $M'[1][1]$, il nous reste à supprimer le nœud d. On obtient la séquence $\{\langle(relabel, \epsilon, b), (delete, 1, /)\rangle\}$ de coût 2.

A la fin de l'appel récursif la séquence os_1 provient de la séquence $\{\langle(relabel, \epsilon, b), (delete, 1, /)\rangle\}$ de $M'[2][1]$ dans laquelle les positions ont été préfixées de la position 0. Remarquons que c'est à la position 0 dans t qu'a été effectué l'appel récursif pour corriger le sous-arbre $t|_0$.

M′	0 b	1 c
0 a	$\{\langle(relabel, \epsilon, b)\rangle\}$	$\{\langle(relabel, \epsilon, b), (insert, 0, c))\rangle\}$
1 c	$\{\langle(relabel, \epsilon, b), (delete, 0, /))\rangle\}$	$\{\langle(relabel, \epsilon, b)\rangle\}$
2 d	$\emptyset$	$\{\langle(relabel, \epsilon, b), (delete, 1, /)\rangle\}$

FIGURE 5.10 – *Nouvelle matrice calculée par l'appel récursif.*

(iii) la case $M[0][1] = \{\langle(add, 0, b), (add, 0.0, c)\rangle\}$ avec un coût de 2. A cette séquence est concaténée la séquence $os_2 = \{\langle(delete, 0.1, /), (delete, 0.0, /), (delete, 0, /)\rangle\}$, qui permet de supprimer le sous-arbre à la position 0 dans t. Cependant le coût de suppression de ce sous-arbre est de 3. Le coût total de la nouvelle séquence (obtenue après concaténation) est de 5, dépassant le seuil $th = 2$. Ainsi la case $(0, 1)$ ne pourra pas générer de séquences d'opérations pour la case $(1, 1)$.

Le calcul de la case $(1, 1)$ selon les items (i),(ii),(iii) ci-dessus, est illustré dans la Figure 5.11.

Pour les autres cases de la matrice M (Fig. 5.9), la transition (q_1, b, q_1) est utilisée. Lorsque le mot formé par les étiquettes des colonnes est dans $L(FSA_{root})$ (c'est-à-dire lorsqu'on atteint un état final) alors la dernière case en bas de la colonne contient des séquences d'opérations pouvant dériver un arbre candidat t' à partir de t. Dans notre exemple, malgré que chaque colonne associée à l'étiquette b correspond à un état final, la dernière case des colonnes 1, 2 et 4 sont vides et donc ne produisent pas d'arbres candidats. Par contre la case $(3, 3)$ de la colonne 3 contient la séquence os_1=$\langle(relabel, 0, b), (delete, 0.1, /)\rangle$ qui transforme t en un arbre valide t'_1 (Figure 5.12).

Toutes les cases de le colonne $j = 4$ sont vides, ce qui veut dire que nous ne pouvons pas avoir une séquence de coût inférieur au $th = 2$ pour un mot ayant $bbbb$ comme préfixe.

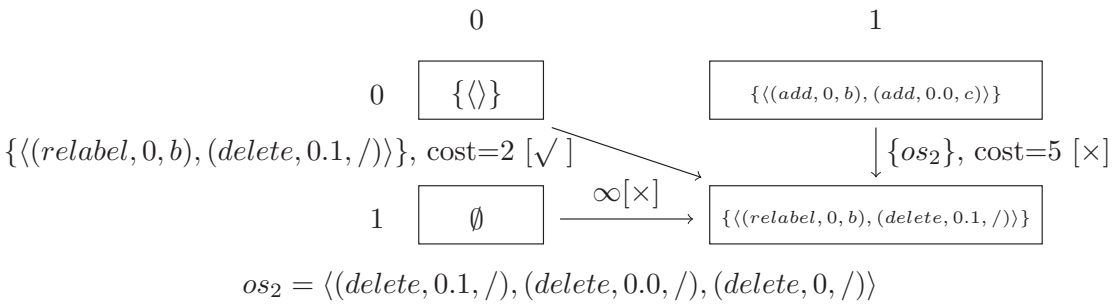


FIGURE 5.11 – *Calcul de la case (1,1) de la matrice M.*

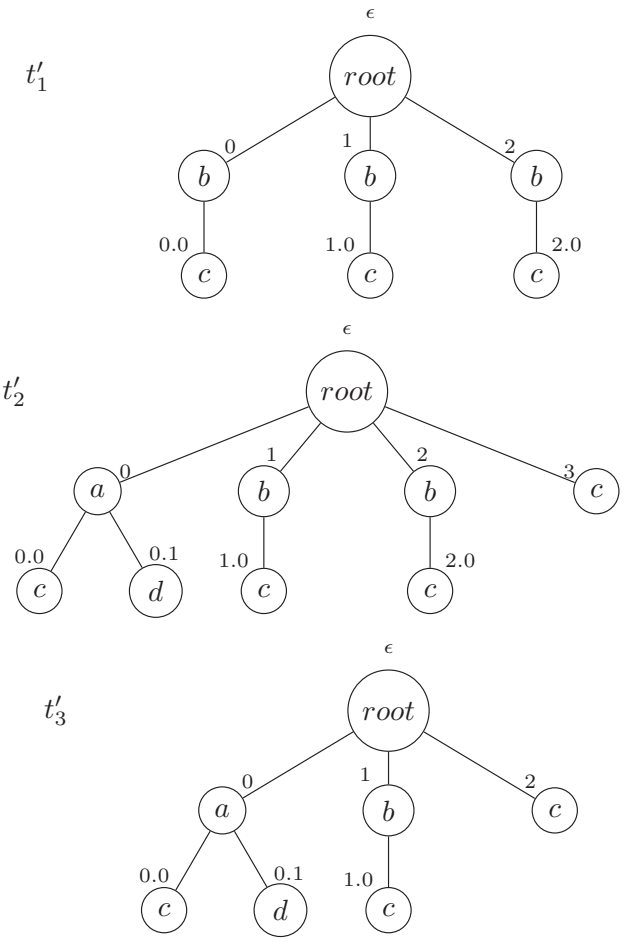


FIGURE 5.12 – *Trois candidats possibles t'_1, t'_2 et t'_3 pour l'arbre t de la Figure 5.1.*

Dans cette situation, un retour arrière est requis pour essayer d'autres transitions. Le retour arrière va supprimer les colonnes $j = 4, 3, 2, 1$ ce qui correspond à un retour à l'état q_0. La prochaine transition est (q_0, a, q_2) qui permet d'ajouter la seconde colonne de la matrice illustrée dans la Figure 5.13. Les autres colonnes de la matrice M sont calculées en suivant la transition (q_2, b, q_2) jusqu'à ce que l'on atteint une colonne vide. Un retour arrière est encore effectuée pour atteindre la colonne $j = 3$ et nous utilisons la transition (q_2, c, q_3) pour calculer la nouvelle colonne $j = 4$. La matrice courante obtenue après ces changements est illustrée dans la Figure 5.14.

Le mot *abbc* formé par les colonnes de la matrice courante (Figure 5.14) appartient au

M	0 root	1 a	2 b	3 b	4 b	**5** ***b***
0 root	$\{\langle\rangle\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
1 a	$\emptyset$	$\{\langle\rangle\}$	$\{\langle(add, 1, b), (add, 1.0, c)\rangle\}$	$\emptyset$	$\emptyset$	$\emptyset$
2 b	$\emptyset$	$\{\langle(delete, 1.0, /), (delete, 1, /)\rangle\}$	$\{\langle\rangle\}$	$\{\langle(add, 2, b), (add, 2.0, c)\rangle\}$	$\emptyset$	$\emptyset$
3 b	$\emptyset$	$\emptyset$	$\{\langle(delete, 2.0, /), (delete, 2, /)\rangle\}$	$\{\langle\rangle\}$	$\{\langle(add, 3, b), (add, 3.0, c)\rangle\}$	$\emptyset$

FIGURE 5.13 – *Contenu de la matrice M pour le mot $u = abbbb$ (après un retour arrière à l'état q_1 de l'automate FSA_{root}))*

M	0 root	1 a	2 b	3 b	**4** ***c***
0 root	$\{\langle\rangle\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
1 a	$\emptyset$	$\{\langle\rangle\}$	$\{\langle(add, 1, b), (add, 1.0, c)\rangle\}$	$\emptyset$	$\emptyset$
2 b	$\emptyset$	$\{\langle(delete, 1.0, /), (delete, 1, /)\rangle\}$	$\{\langle\rangle\}$	$\{\langle(add, 2, b), (add, 2.0, c)\rangle\}$	$\emptyset$
3 b	$\emptyset$	$\emptyset$	$\{\langle(delete, 2.0, /), (delete, 2, /)\rangle\}$	$\{\langle\rangle\}$	$\boldsymbol{\{\langle(add, 3, c)\rangle\}}$

FIGURE 5.14 – *Contenu de la matrice M après un retour arrière et l'utilisation de la transition (q_2, c, q_3) de FSA_{root}*

langage $L(FSA_{root})$ et donc la dernière case en bas de la colonne $j = 4$ contient une séquence d'opérations ayant un coût ne dépassant pas le seuil $th = 2$. Cette séquence dérive donc un nouvel arbre t'_2 illustré dans la Figure 5.12. L'état q_3 ne possède pas de transition sortante, ce qui nous conduit à un retour arrière à la colonne $j = 2$ pour essayer la transition donnant le mot *abc*. La Figure 5.15 montre la matrice correspondante au mot *abc*. La dernière case de la colonne $j = 3$ a un coût ne dépassant pas le seuil $th = 2$. Cette séquence est obtenue par le calcul d'une nouvelle matrice en faisant un appel récursif pour corriger le sous-arbre $\{(\epsilon, b), (0, c)\}$ à la position 2 dans t en $\{(\epsilon, c)\}$. L'arbre candidat t'_3 est illustré dans la Figure 5.12. Pour le seuil $th = 2$, il n'est pas possible de trouver d'autres arbres candidats autre que t'_1, t'_2 et t'_3.

M	0 root	1 a	2 b	**3** ***c***
0 root	$\{\langle\rangle\}$	$\emptyset$	$\emptyset$	$\emptyset$
1 a	$\emptyset$	$\{\langle\rangle\}$	$\{\langle(add, 1, b), (add, 1.0, c)\rangle\}$	$\emptyset$
2 b	$\emptyset$	$\{\langle(delete, 1.0, /), (delete, 1, /)\rangle\}$	$\{\langle\rangle\}$	$\boldsymbol{\{\langle(add, 2, c)\rangle\}}$
3 b	$\emptyset$	$\emptyset$	$\{\langle(delete, 2.0, /), (delete, 2, /)\rangle\}$	$\boldsymbol{\{\langle(relabel, 2, c), (delete, 2.0, /)\rangle\}}$

FIGURE 5.15 – *Contenu de la matrice M pour le mot $u = abc$ après un autre retour en arrière*

5.3.4 Présentation de l'algorithme de correction d'un document XML

Nous allons présenter notre algorithme pour la correction de documents et prouver ses propriétés (c'est-à-dire qu'il termine, qu'il est correct et complet). Nous ferons ensuite une analyse de sa complexité. Avant de présenter notre algorithme de correction, nous allons d'abord introduire des opérateurs sur les ensembles de séquences d'opérations, qui sont utilisés dans l'algorithme. Soient deux ensembles de séquences d'opérations S et S' ; ces opérateurs permettent par exemple d'ajouter des préfixes aux positions de toutes les opérations dans S, de concaténer des séquences d'un ensemble S à celles d'un ensemble S' en ne gardant que celles qui ont un coût inférieur à un seuil th donné, de faire l'union de deux ensembles S et S' en ne gardant que les séquences qui n'ont pas de séquences équivalentes avec un coût plus petit.

Définition 5.13 (Union restreinte aux coûts minimaux)
Soient Nos_1 et Nos_2 deux ensembles de séquences d'opérations d'édition, et $Nos = Nos_1 \cup Nos_2$. Nous dénotons par $Nos_1 \Cup Nos_2$ l'**union restreinte aux coûts minimaux** de Nos_1 et Nos_2 définie par :

$$Nos_1 \Cup Nos_2 = \{nos \mid nos \in Nos \ \wedge [\nexists nos' \in Nos \text{ tq. } nos' \equiv nos \text{ et } Cost(nos') < Cost(nos)]\}$$

□

Soient Nos_1 et Nos_2 deux ensembles de séquences d'opérations d'édition. Nous dénotons par $Nos_1 \cdot Nos_2$ la concaténation de Nos_1 et Nos_2 qui donne :

$$Nos_1 \cdot Nos_2 = \{nos_1.nos_2 \mid nos_1 \in Nos_1, nos_2 \in Nos_2\}.$$

Exemple 5.12 Considérons les ensembles de séquences d'opérations d'édition
$S_1 = \{\langle(add, 1, c), (delete, 2, /), (relabel, 0, d)\rangle, \langle(relabel, 2, c), (delete, 2.0, /)\rangle\}$, et
$S_2 = \{\langle(relabel, 0, a)\rangle\}$. Nous avons : $S_1 \cdot S_2 = \{\langle(add, 1, c), (delete, 2, /), (relabel, 0, d), (relabel, 0, a)\rangle, \langle(relabel, 2, c), (delete, 2.0, /), (relabel, 0, a)\rangle\}$.
Remarquons que si Nos_1 ou Nos_2 est l'ensemble vide $\emptyset$ alors $Nos_1 \cdot Nos_2 = \emptyset$.

⊠

Définition 5.14 (Concaténation bornée par un seuil)
Soient Nos_1 et Nos_2 deux ensembles de séquences d'opérations d'édition, et le seuil th tel que $th \geq 0$. Nous définissons la **concaténation bornée par le seuil th** de Nos_1 et Nos_2, dénotée par $Nos_1 \cdot_{th} Nos_2$, comme étant le sous-ensemble de la concaténation $Nos_1 \cdot Nos_2$ dans lequel toutes les séquences ont un coût plus petit que th. Formellement nous avons :

$$Nos_1 \cdot_{th} Nos_2 = \{nos_1.nos_2 \mid nos_1 \in Nos_1, nos_2 \in Nos_2 \text{ et } Cost(nos_1.nos_2) \leq th\}.$$

□

Exemple 5.13 Considérons les ensembles de séquences d'opérations d'édition
$Nos_1 = \{\langle(add, 1, a), (add, 1.0, b), (add, 1.1, c)\rangle, \langle(add, 3, e), (add, 3.0, f)\rangle\}$, $Nos_2 = \{\langle(add, 0, g), (add, 0.0, h)\rangle\}$, et $Nos_3 = \{\langle(relabel, \epsilon, root), (add, 2, b)\rangle, \langle(delete, 4, /)\rangle, nos_\emptyset\}$.
Nous avons :
$Nos_1 \cdot_3 Nos_3 = \{\langle(add, 1, a), (add, 1.0, b), (add, 1.1, c)\rangle, \langle(add, 3, e), (add, 3.0, f), (delete, 4, /)\rangle, \langle(add, 3, e), (add, 3.0, f)\rangle\}$

$Nos_2 \cdot_3 Nos_3 = \{\langle(add, 0, g), (add, 0.0, h), (delete, 4, /)\rangle, \langle(add, 0, g), (add, 0.0, h)\rangle\}$

$Nos_1 \cdot_3 Nos_2 = \emptyset$.

⊠

Définition 5.15 (Ajout de préfixe à un ensemble de séquences d'opérations)
Soit Nos un ensemble de séquences d'opérations d'édition, et $u \in \mathbb{N}^*$. Nous définissons l'**ajout du préfixe u à un ensemble de séquences d'opérations**, dénoté par $AddPrefix(Nos, u)$, comme étant l'ensemble résultat de la concaténation du préfixe u à toutes les positions des opérations qui sont dans Nos. Formellement nous avons : $AddPrefix(Nos, u) = \{\langle ed_1, \ldots, ed_n\rangle \mid ed_i = (op_i, u.pos_i, l_i)$ pour $1 \leq i \leq n$ et $\exists\ \langle ed'_1, \ldots, ed'_n\rangle \in Nos$ tel que $ed'_i = (op_i, pos_i, l_i)\}$.

□

L'algorithme de correction prend en paramètre un schéma $S = (\Sigma, root, Rules)$, un arbre XML t et un seuil entier th. Pour corriger un arbre XML t par rapport à S en dessous du seuil th, nous utilisons une méthode de programmation dynamique qui calcule une matrice à deux dimensions d'édition d'arbre-à-langage M_u^c où c est une étiquette et $u = u_1 u_2 \ldots u_k$ est un mot (une séquence d'étiquettes). Chaque case de la matrice M_u^c contient un ensemble de séquences d'opérations d'édition de nœud. Plus précisément la case $M_u^c[i][j]$ contient l'ensemble de séquences d'opérations d'édition de nœud permettant de transformer l'arbre partiel $t\langle i{-}1\rangle$ en $t_1, \ldots, t_n$ tels que chaque arbre candidat :

- est partiellement valide ;
- a comme racine c et les fils de la racine forment un préfixe[3] du mot u, de longueur j ;
- la distance d'édition avec t ne dépasse pas le seuil th.

Formellement nous avons :

$$\exists\ t_1, \ldots, t_n\ [t\langle i{-}1\rangle \xrightarrow{M_u^c[i][j]} \{t_1, \ldots, t_n\} \text{ et } \forall\ 1 \leq k \leq n\ [t_k \in L^c_{u[1\ldots j]}(S) \text{ et } dist(t, t_k) \leq th].$$

En posant $c = root$, $i = \bar{t}$ et $j = |u|$, la case $M_u^c[i][j]$ contient l'ensemble de séquences d'opérations permettant de transformer l'arbre t en un ensemble d'arbres appartenant à l'ensemble correction $L_t^{th}(S)$ ($L_t^{th}(S)$ est l'ensemble correction dans la Définition 5.12).

La matrice M_u^c est calculée par l'Algorithme 8. Il prend en paramètre l'étiquette c (du schéma) vers lequel on veut corriger l'arbre, l'arbre à corriger t, le schéma S et le seuil th. L'Algorithme 8 retourne l'ensemble des séquences d'opérations permettant de transformer l'arbre t en arbres localement valides ayant c comme racine. Lorsque l'Algorithme 8 est appelé avec $c = root$, il retourne bien tout l'ensemble correction $L_t^{th}(S)$.

Le premier appel à la fonction *correction* (Algorithme 8) implique d'autres appels dont les résultats sont collectés dans l'ensemble *Result* qui sera ensuite retourné à la fin.

Lorsque le seuil th vaut 0 et que l'arbre initial t est localement valide et possède la bonne racine c alors le résultat de la correction est la séquence vide (lignes 1–2) du fait qu'aucune opération n'est nécessaire pour transformer t. Dans le cas où t n'est pas localement valide, avec un seuil th inférieur à 0, il n'existe pas de solutions (lignes 3–4) ce qui entraîne un ensemble de solutions vide (qui est différent d'un ensemble de solution qui contient uniquement la séquence vide). Avec un seuil positif et non nul, la matrice M_u^c est initialisée avec une colonne qui correspond au mot $u = \epsilon$ et la matrice a autant de lignes que le nombre de sous-arbres fils de la racine additionné à 1 puisque la ligne 0 correspond à la racine (lignes 6–8). Les cases de la première colonne sont ensuite calculées (lignes 10–18). La première case $M_u^c[0][0]$ reçoit le(s) opérations nécessaire(s) pour introduire la bonne racine c, c'est à dire

(i) l'insertion de c lorsque t est vide (lignes 10–11),

3. Pour le mot $u = u_1 u_2 \ldots u_j \ldots u_n \in \Sigma^*$ nous dénotons par $|u|$ la longueur de u, c'est-à-dire $|u| = n$, et par $u[1..j]$ le préfixe de u de longueur j, c'est-à-dire $u[1..j] = u_1 u_2 \ldots u_j$ avec $1 \leq j \leq n$.

(ii) la séquence vide lorsque la racine de t est correcte (lignes 12–13),

(iii) le renommage de la racine lorsque la racine de t n'est pas correcte (lignes 14–15).

Remarquons que si t est égal à l'arbre vide $t_\emptyset$ alors la matrice M_u^c ne possède qu'une seule ligne ce qui entraîne le calcul de la seule case $M_u^c[0][0]$.

Toutes les cases en dessous de $M_u^c[0][0]$ vont contenir les séquences d'opérations tranformant les arbres partiels $t\langle i-1\rangle$ en un arbre possédant seulement la racine c. En suivant les lignes 17–18, ces séquences contiennent l'opération calculée pour la case $M_u^c[0][0]$ concaténée aux suppressions de tous les sous-arbres de la racine de t. Notons que les suppressions (i) sont effectuées de la droite vers la gauche dans le but d'éviter le décalage des positions, (ii) sont exprimées, pour des raisons de simplicité, en opérations d'édition d'arbres « *remove* ». Enfin la matrice M_u^c avec la première colonne initialisée est passée à la fonction *correctionState* (ligne 21), qui va poursuivre le calcul et retourner le résultat final.

Algorithme 8 correction(t, S, th, c) : Correction de l'arbre XML t par rapport au schéma S

Entrée :
- l'arbre XML à corriger t
- la description de schéma S
- le seuil th
- l'étiquette c qui sera la racine des arbres résultats

Sortie : L'ensemble des séquences d'opérations qui permettent d'obtenir les arbres résultats

1: **si** $th = 0$ et $t \in L_{loc}(S)$ et $t(\epsilon) = c$ **alors**
2: **retourner** $\{\langle\rangle\}$ % ***arrêt de l'appel récursif*** %
3: **sinon si** $th \leq 0$ **alors**
4: **retourner** $\emptyset$ % ***arrêt de l'appel récursif*** %
5: **sinon**
6: $u := \epsilon$
7: $n := \bar{t}$ % n ***est le nombre de fils de la racine de*** t %
8: $M_u^c := initMatrix(n+1, 1)$ % ***initialise la matrice avec*** $n+1$ ***ligne et une colonne*** %

9: % ***calcul de la première colonne de la matrice*** %
10: **si** $t = t_\emptyset$ **alors**
11: $M_u^c[0][0] := \{\langle(add, \epsilon, c)\rangle\}$
12: **sinon si** $c = t(\epsilon)$ **alors**
13: $M_u^c[0][0] := \{\langle\rangle\}$
14: **sinon**
15: $M_u^c[0][0] := \{\langle(relabel, \epsilon, c)\rangle\}$
16: **fin si**
17: **pour** i:=1 **à** n **faire**
18: $M_u^c[i][0] := \{\langle(remove, i-1, t_\emptyset)\rangle\} \cdot_{th} M_u^c[i-1][0]$
19: **fin pour**

20: % ***on corrige les fils de la racine de*** t %
21: $Result := correctionState(t, S, th, c, M_u^c, initialState(FSA_c), Result)$ % ***la fonction initialState retourne l'état initial du FSA associé à*** c %
22: **retourner** $Result$
23: **fin si**

La fonction *correctionState* (Algorithme 9) effectue une exploration en profondeur

d'abord de l'automate FSA_c associé à la racine c (étiquette racine vers lequel se fait la correction de t). Lorsque l'état s en paramètre est un état final alors le mot u (obtenu en traversant les transitions utilisées dans l'automate jusqu'à l'appel actuel de la fonction) appartient au langage $L(FSA_c)$. Par conséquent, toutes les séquences accumulées dans la dernière case vers le bas de la dernière colonne de M_u^c conduisent à des arbres localement valides et donc peuvent être ajoutées à l'ensemble solution (lignes 2–3). Ensuite chaque transition δ partant de l'état courant s est considérée (lignes 6–7). La fonction *correctionState* retourne le résultat final après avoir traité toutes les transitions de l'automate FSA_c. Remarquons que toutes les transitions sont traitées après plusieurs appels récursifs. Un appel de la fonction *correctionState* permet de prendre en compte les transitions d'un état.

Algorithme 9 correctionState(t, S, th, c, M_u^c, s, *Result*) : Exploration en profondeur d'abord à partir de l'états s

Entrée :
- l'arbre XML à corriger t, la description de schéma S, le seuil th
- l'étiquette c qui sera la racine des arbres résultats
- la matrice courante M_u^c
- l'état courant s dans l'automate FSA_c
- l'ensemble des séquences *Result*, qui sera complété par les séquences calculées pour l'état s

Sortie : L'ensemble des séquences d'opérations qui permettent d'obtenir les arbres résultats

1: **% *dans le cas où*** $u \in L(FSA_c)$**, *on ajoute les séquences de la dernière case de la colonne courante au résultat* %**
2: **si** $s \in FSA_c.F$ **alors**
3: $Result := Result \Cup M_u^c[\bar{t}][|u|]$
4: **fin si**
5: **% *exploration de l'automate*** FSA_c **%**
6: **pour tout** $\delta \in FSA_c.\Delta$ tel que $\delta = (s, a, s')$ **faire**
7: $Result := correctionTransition(t, S, th, c, M_u^c, s', a, Result)$
8: **fin pour**
9: **retourner** *Result*

La fonction *correctionTransition* (Algorithme 10) utilise la transition courante δ avec son étiquette a pour corriger les arbres partiels de t en des arbres partiels valides ayant a comme racine de leur dernier sous-arbre. La fonction *correctionTransition* procède (i) en calculant une colonne (la dernière de la matrice), (ii) en vérifiant s'il est possible de poursuivre le chemin d'exploration, et si c'est le cas, (iii) prolonge le chemin en appelant de nouveau la fonction *correctionState*. A partir des fonctions *correctionState* et *correctionTransition*, nous pouvons noter que le nombre de colonnes calculées pour un processus de correction de correction de l'arbre t par rapport à l'étiquette c est borné par $f_c^{\bar{t}+th}$ où f_c est le nombre maximal de transitions sortantes pour tous les états dans FSA_c. Cette remarque peut être vérifiée dans l'exemple détaillé en section 5.3.3.

Le mot $v = u.a$ est formé par les étiquettes des transitions utilisées dans l'automate FSA_c jusqu'à l'appel actuel et de l'étiquette a de la transition courante (ligne 2). La matrice M_v^c est initialisée avec les mêmes cases que M_u^c (ligne 5), avec une colonne en plus (correspondant à la transition courante) qui est calculée aux lignes 9–22. La matrice M_u^c possède $v+1$ colonnes et $n+1$ lignes où n est le nombre de sous-arbre fils de la racine de t (lignes 3–5). Pour calculer

Algorithme 10 correctionTransition(t, S, th, c, M_u^c, s, a, $Result$) :

Entrée :
- l'arbre XML à corriger t, la description de schéma S, le seuil th
- l'étiquette c qui sera la racine des arbres résultats
- la matrice courante M_u^c
- l'état cible s de la transition courante dans l'automate FSA_c
- l'étiquette a de la transition courante
- l'ensemble des séquences $Result$ (contenant les séquences calculées pour les états déjà visités)

Sortie : L'ensemble des séquences d'opérations qui permettent d'obtenir les arbres résultats

1: $nbColonneNonVide := 0$
2: $v := u.a$
3: $m := |v|$ % *m est la longueur du mot courant* %
4: $n := \bar{t}$ % *n est le nombre de fils de la racine de t* %
5: $M_v^c := initMatrix(n{+}1, m{+}1, M_u^c)$ % ***initialise la nouvelle matrice avec avec** $n{+}1$ **lignes et** $m{+}1$ **colonnes, et copie toutes les colonnes de** M_u^c **dans** M_v^c* %
6: % ***calcul des arbres localement valides pour l'étiquette** a* %
7: $T := correction(t_\emptyset, S, th - MinCost(M_v^c[0][m{-}1]), a)$

8: % ***début du calcul de la dernière colonne de** M_v^c* %
9: $M_v^c[0][m] := M_v^c[0][m{-}1] \cdot_{th} AddPrefix(T, m{-}1)$
10: **si** $M_v^c[0][m] \neq \emptyset$ **alors**
11: $nbColonneNonVide := nbColonneNonVide + 1$
12: **fin si**
13: **pour** $i := 1$ **à** n **faire**
14: $T := correction(t_\emptyset, S, th - MinCost(M_v^c[i][m{-}1]), a)$
15: $T' := correction(t|_{i-1}, S, th - MinCost(M_v^c[i{-}1][m{-}1]), a)$
16: $M_v^c[i][m] := M_v^c[i][m{-}1] \cdot_{th} AddPrefix(T, m{-}1) \Cup$ % ***correction issue de l'horizontale*** %
17: $M_v^c[i{-}1][m{-}1] \cdot_{th} T' \Cup$ % ***correction issue de la diagonale*** %
18: $\{(remove, i{-}1, t_\emptyset)\} \cdot_{th} M_v^c[i{-}1][m]$ % ***correction issue de la verticale*** %
19: **si** $M_v^c[i][m] \neq \emptyset$ **alors**
20: $nbColonneNonVide := nbColonneNonVide + 1$
21: **fin si**
22: **fin pour**
23: % ***fin du calcul de la dernière colonne de** M_v^c* %

24: % ***lorsque la dernière colonne contient au moins une case non vide, la recherche se poursuit en appellant correctionState sur l'état** s**. Dans le cas contraire la recherche est stoppée*** %
25: **si** $nbColonneNonVide \geq 1$ **alors**
26: $Result := correctionState(t, S, th, c, M_v^c, s, Result)$
27: **fin si**
28: **retourner** $Result$

les cases de la dernière colonne, nous calculons en premier les séquences d'opérations qui transforment l'arbre vide $t_\emptyset$ en des arbres localement valides ayant a comme racine (lignes 7 et 14). Ces séquences sont t-équivalent aux opérations d'éditions d'arbres qui insèrent des arbres localement valides à la position $m{-}1$ dans t. Chacune de ses opérations n'intervient seulement après avoir corrigé l'arbre partiel $t\langle i{-}1\rangle$ (par exemple pour $i = 0$, nous avons seulement la racine) en des arbres partiellement valides $t' \in L_u^c$. Ensuite, le coût autorisé pour insérer un arbre avec comme racine a à la position $m{-}1$ ne peut pas dépasser le seuil

global th auquel on enlève le coût minimal de la correction précédente de $t\langle i{-}1\rangle$ en t' ($th - MinCost(M_v^c[i][m{-}1])$). Comme cette valeur minimale varie, cette correction est effectuée pour chaque case de la dernière colonne.

La première case de la colonne m de la matrice correspond à la transformation de l'arbre partiel $t\langle -1\rangle$ en chacun des arbres de $t'' \in L_v^c$. Ainsi son contenu est formé par les corrections précédentes de $t\langle -1\rangle$ en $t' \in L_u^c$ concaténées avec les insertions de sous-arbre à la position $m{-}1$ préfixées par la position d'insertion $m{-}1$ (ligne 9). A chaque fois que des séquences d'opérations sont concaténées ($\cdot_{th}$), nous ne gardons que les séquences résultats dont les coûts ne dépassent pas le th. Les autres cases de la colonne m sont calculées en prenant en compte les trois possibilités issues de la proposition de Selkow (cf. section 5.3.1) :

- la première est de transformer l'arbre partiel $t\langle i{-}1\rangle$ en $t' \in L_u^c$, et ensuite d'ajouter un arbre localement valide par rapport à la racine a à la position $m{-}1$ dans t (ligne 16). Ceci correspond à la correction issue de l'horizontale dans la matrice illustrée dans la Figure 5.6(b) et dans la Figure 5.11.
- la deuxième est de transformer l'arbre partiel $t\langle i{-}2\rangle$ en $t' \in L_u^c$, et ensuite de corriger le sous-arbre $t|_{i-1}$ en un sous-arbre localement valide par rapport à la racine a (ligne 17). Ceci correspond à la correction issue de la diagonale dans la matrice illustrée dans la Figure 5.6(b) et dans la Figure 5.11.
- la troisième est de supprimer le sous-arbre $t|_{i-1}$, et ensuite de transformer l'arbre partiel $t\langle i{-}2\rangle$ en $t' \in L_v^c$ (ligne 18). Ceci correspond à la correction issue de la verticale dans la matrice illustrée dans la Figure 5.6(b) et dans la Figure 5.11.

Toutes les séquences calculées à partir de ses trois possibilités sont stockées en s'assurant que : (i) leur coût ne dépasse pas le seuil th (cette vérification est effectuée au moment de la concaténation bornée $\cdot_{th}$) (ii) elles n'ont pas de séquences équivalentes avec un coût inférieur (ceci est garantie par l'opération de l'union avec coût minimum $\Cup$). Certaines cases de la colonne courante peuvent être vide après l'application de la concaténation bornée $\cdot_{th}$. La variable *nbColonneNonVide* compte le nombre de cases dans la colonne courante qui contiennent au moins une solution (lignes 1, 10–11 et 19–20). Si la colonne courante contient au moins une solution alors la correction récursive continue avec l'état d'arrivée de la transition courante (ligne 26). Dans le cas contraire, le chemin d'exploration est arrêté et un retour-arrière à partir de la transition courante vers une transition précédente est effectué (c'est-à-dire que la matrice courante avec la dernière colonne ne sera plus utilisée).

Il est important de remarquer que le résultat de la fonction $correction(t, S, th, c)$ (Algorithme 8) est exactement l'ensemble de séquences d'opérations d'édition de nœud suivant :

$$\left\{ nos \mid nos \in \bigcup_{u \in L(FSA_c)} M_u^c[\overline{t}][|u|] \text{ et } Cost(nos) \leq th \right\}$$

A cause des lignes 6–7 de la fonction *correctionState*, nous essayons tous les mots possibles dans le langage $L(FSA_c)$ et la linge 2 ajoute au résultat seulement les séquences des cases $M_u^c[n][|u|]$ pour lesquels $u \in L(FSA_c)$. Ainsi seules les cases $M_u^c[\overline{t}][|u|]$ sont sélectionnées et elles correspondent aux séquences qui mènent à des arbres valides. De plus l'opérateur concaténation bornée ($\cdot_{th}$) combiné à l'opérateur union de coût minimum ($\Cup$) n'autorisent que des séquences d'opérations, non redondantes qui ont des coûts ne dépassant pas le seuil th, dans les cases de la matrice M_u^c. Dans les lignes qui vont suivre, nous allons prouver que la fonction $correction(t, S, th, c)$ (Algorithme 8) calcule bien l'ensemble que nous venons de décrire.

Exemple 5.14 Nous pouvons vérifier que les corrections trouvées dans l'exemple de la section 5.3.3 sont précisément celles qui sont calculées par la fonction *correction* (Algorithme 8) ⊠

Propriétées.

Nous allons commencer par les aspects simples de l'algorithme. Les preuves des lemmes et théorèmes de cette section sont données en Annexe, section A.7. Le premier aspect intéressant est de montrer que la correction de l'arbre vide donne bien des arbres localement valides en dessous du seuil. Cet aspect est formulé par le lemme suivant :

Lemme 5.1 *Soient une étiquette $c \in \Sigma$, un schéma S et un seuil th. Supposons le schéma $S' = (\Sigma, c, S.Rules)$. L'appel de correction($t_\emptyset$,S,th,c) termine et retourne l'ensemble Result tel qu'on ait :*

$$t_\emptyset \xrightarrow{Result} L^{th}_{t_\emptyset}(S').$$

Nous allons maintenant supposer que le $th = 0$. Ce cas est considéré séparément car il n'y a pas création de matrice.

Lemme 5.2 *Soient une étiquette $c \in \Sigma$, un schéma S et un seuil th. Supposons le schéma $S' = (\Sigma, c, S.Rules)$. L'appel de correction(t,S,0,c) termine et retourne l'ensemble Result tel qu'on a :*

$$t \xrightarrow{Result} L^0_t(S').$$

Nous pouvons maintenant traiter le cas général c'est-à-dire la correction d'un arbre non vide avec un seuil th non nul. Nous allons montrer que chaque case de la matrice d'édition calculée par notre algorithme, permet de transformer les arbres partiels de t en des arbres partiellement valides en dessous du seuil th.

Lemme 5.3 *Soient une étiquette $c \in \Sigma$, un schéma S, un arbre XML $t \neq t_\emptyset$, et un seuil $th > 0$. Soit un mot $u \in \Sigma^*$ tel que $u \in L(FSA_{S.root})$ et $L^c_u(S) \neq \emptyset$. L'appel de $correction(t, S, th, c)$ calcule la matrice M^c_u telle que pour $i \in [0 \dots \bar{t}]$ et $j \in [0 \dots |u|]$, la formule suivante est vraie :*

$$t\langle i-1\rangle \xrightarrow{M^c_u[i][j]} \{t' \mid t' \in L^c_{u[1..j]}(S), dist(t\langle i-1\rangle, t') \leq th\}.$$

Théorème 5.1 *Soient un arbre XML t, un schéma S et un seuil $th \geq 0$. L'appel de $correction(t, S, th, S.root)$ termine et retourne l'ensemble Result tel qu'on a :*

$$t \xrightarrow{Result} L^{th}_t(S).$$

5.3.5 Complexité

Dans le processus de correction, il existe deux sources de récursivité : la description de schéma S d'une part et l'arbre à corriger d'autre part. Les deux sources de récursivité interviennent en parallèle sauf pour la correction de l'arbre vide.

En effet la récursivité selon la description de schéma est la seule utilisée pour la correction de l'arbre vide vers une étiquette racine a. Ce cas correspond à l'insertion du nœud a qui sera la racine de l'arbre solution. Pour insérer l'arbre ayant la racine a, nous utilisons la règle associée à a dans S et nous construisons le mot correct formé par les étiquettes des fils de a, et chaque nœud fils devient la racine d'un arbre correct à insérer. Ainsi pour chaque

nouveau nœud, nous considérons encore la règle correspondante dans S et nous essayons de construire le mot correct des fils, et ainsi de suite. Le processus de correction de l'arbre vide vers l'étiquette racine a est limitée par le seuil th. Puisque la taille maximale des arbres corrects est th, le processus s'arrête dès que th nœuds ont été insérés dans l'arbre vide. Ainsi le nombre maximale de transitions utilisées dans S (en prenant en compte tous les automates concernés) est toujours limité par $(f_S)^{th}$, où f_S est le nombre maximal de transitions sortantes en considérant tous les états dans les automates d'états finis de S. Nous pouvons remarquer que la correction de l'arbre vide vers une étiquette racine a peut s'arrêter bien avant la limite générale. Par exemple si l'automate contient un seul état qui est final, le processus de correction s'arrête immédiatement. Il est évident que la forme des automates dans S est un paramètre important dans la récursivité : moins il y a de choix, la borne f_S est petite. Les répétitions dans le schéma (c'est-à-dire les cycles dans les automates de S) implique des mots des fils de longueur potentiellement infini. La récursivité du schéma aussi implique des arbres valides de profondeur potentiellement infini. Ces deux derniers cas sont limités par le seuil th.

La deuxième source de récursivité est sur l'arbre à corriger t. Chacun de ses nœuds est susceptible d'être corrigé, ainsi un parcours récursif complet de l'arbre est nécessaire. Ce parcours est effectué par des appels récursifs au processus de correction pour calculer des matrices d'édition. Pour chaque matrice d'édition, nous pouvons déterminer la borne générale pour : (i) le nombre de lignes, et (ii) le nombre maximal de colonnes calculées pour toutes les matrices confondues.

Pour chaque matrice, son *nombre de lignes* est limité par le nombre maximal de fils en prenant en compte tous les nœuds de l'arbre, dénoté f_t, auquel on additionne 1 (pour la racine). Le nombre de colonnes calculées pour corriger l'arbre t dépend de l'exploration dynamique des automates de S. En considérant le graphe d'exploration illustré dans la Figure 5.8, pour un mot incorrect w, la longueur maximale des chemins menant à des mots corrects est inférieure ou égale à $|w|+th$. Par ailleurs, pour chaque nœud dans le graphe d'exploration il existe au plus f nœuds suivants à explorer où f est le nombre maximal de transitions sortantes des états de l'automate. Ainsi, *le nombre de colonnes* calculées dans l'algorithme de Oflazer est limité dans le pire des cas par $f^{|w|+th}$. En généralisant ce résultat à notre contexte, la limite pour la longueur des mots corrects, $|w| + th$, devient $|t| + th$, la taille maximale des arbres corrects. On obtient donc *le nombre maximal de colonnes calculées dans les matrices confondues* qui est limité dans le pire des cas par $(f_S)^{|t|+th}$.

Nous obtenons ainsi le nombre de cases calculées dans le pire des cas qui vaut $(f_t + 1) \times (f_S)^{|t|+th}$. Le calcul de chaque case consiste à insérer, supprimer ou renommer un nœud. La récursivité sur le sous-arbre enraciné en ce nœud est prise en compte dans le nombre global de colonnes multiplié par le nombre maximal de colonnes. Toutefois, comme nous calculons *toutes les solutions* en dessous du seuil th, pour chaque case nous concaténons toutes les séquences d'opérations. Nous allons essayer de borner le nombre de séquences obtenues. Le coût de chaque séquence d'opérations dans la matrice est limité par th. Rappelons que chaque opération sur l'arbre t est un triplet (op, p, a) tel que :

- pour $op = delete$: p est une position dans t.
- pour $op = relabel$: a appartient à Σ ; p est une position dans t.
- pour $op = add$: a appartient à Σ ; p appartient à l'ensemble des positions $Pos(t) \setminus \{\epsilon\} \cup InsFr(t)$, dont la taille est égale 1 pour un arbre vide et $2 \times |t| - 1$ pour un arbre non-vide [4].

4. Notons que la frontière d'insertion contient exactement une nouvelle position fille pour chaque nœud existant dans l'arbre initial.

Par conséquent, il y a un large choix possible pour p dans le cas d'une séquence contenant que des insertions de nœud. Aussi après l'insertion d'un nœud dans t, une nouvelle position est créée dans t. Ainsi l'insertion d'un nouveau nœud (le cas échéant) peut apparaître dans l'une des $2 \times (|t| + 1) - 1$ positions résultantes. Et si l'on continue l'insertion d'un autre nouveau nœud peut concerner l'une des $2 \times (|t| + 2) - 1$ positions, et ainsi de suite. Au total le nombre de séquences de positions possibles pour les opérations est limité par $(2 \times (|t| + th))^{th}$. Pour chacune de ces opérations nous pouvons choisir parmi l'une des trois opérations, et l'une parmi $|\Sigma|$ étiquettes cibles. Pour ces raisons, la taille des ensembles possible de séquences d'opérations est limitée par $(3 \times |\Sigma| \times 2 \times (|t| + th))^{th}$. Nous pouvons conclure que la complexité en temps de l'algorithme de correction d'arbre, dans le pire des cas est de :

$$O((f_t + 1) \times (f_S)^{|t|+th} \times 6 \times |\Sigma| \times (|t| + th))^{th}.$$

Dans la pratique deux facteurs diminuent la limite précédente, ces deux facteurs sont difficiles à formaliser dans l'analyse de la complexité. Le premier facteur découle de l'exploration dynamique de l'automate qui est vite limitée par le seuil th : nous arrêtons d'ajouter de nouvelles colonnes dès que toutes les cases dans la colonne courante sont vides (une case est vide lorsque toutes les séquences qu'elle doit contenir ont un coût dépassant th). Le deuxième facteur provient du fait que les calculs des arbres partiels sont factorisés, c'est-à-dire que les colonnes initiales sont calculées seulement une fois pour tous les mots ayant le même préfixe (ce phénonème est aussi le même pour la proposition de Oflazer pour la correction des mots). Ces facteurs sont sûrement les raisons pour lesquelles les résultats expérimentaux décrits dans la section 5.5 montrent difficilement des courbes exponentielles en temps, lorsque les courbes sont en fonction de la taille du document ou du seuil. Par ailleurs, des optimisations ont été introduites dans l'implémentation dans le but de réduire le temps d'exécution. L'une de ses optimisations consiste à sauvegarder les résultats intermédiaires de correction dans une structure auxiliaire pour ne calculer qu'une seule fois la correction d'un sous-arbre de t vers une étiquette racine a et en dessous d'un seuil th. Par conséquent si l'on a besoin de corriger de nouveau le même sous-arbre de t vers a et en dessous d'un seuil $th_< th$, nous recherchons les solutions dans la structure auxiliaire au lieu de faire un nouvel appel récursif pour corriger le sous-arbre. Les détails sur l'optimisation peuvent être trouvées dans l'annexe de notre papier [7]. Pour les documents qui contiennent des séquences de sous-arbres larges et qui ont une structure identique, le stockage intermédiaire améliore de façon significative les performances de correction.

5.4 Discussion, adaptations et extensions

Notre approche pour corriger un document XML t *par rapport à* un schéma S est unique du fait qu'il est complet, dans le sens qu'à partir d'un seuil non négatif th, l'algorithme trouve tous les arbres t' valides par rapport à S tels que la distance d'édition entre t et t' ne dépasse pas th. C'est ainsi qu'il offre la garantie d'avoir toutes les solutions vérifiant un critère précis (ici selon un seuil donné). Cela nous permet d'être sûr que la meilleure solution (qui satisfait le critère) est réellement dans l'ensemble résultat. Il est à noter que les caractéristiques qui définissent ce qu'est exactement *la meilleure solution* dépendent fortement du contexte d'application et parfois peuvent être assez informelles (ainsi difficile à sélectionner automatiquement). Cependant produire un ensemble ayant plusieurs solutions, en particulier pour l'utilisateur final, est généralement contre-productif. Pour cette raison, chaque application basée sur notre approche devra exécuter des tâches de pré-traitements dans le

but de tirer profit des solutions calculées dans le contexte d'application particulier. Les pré-traitements dépendants du contexte peuvent être vraiment utiles. La mise à disposition du code source sous une licence libre permet les différentes formes d'adaptation.

Par exemple, lorsqu'une mise à jour de schéma rend invalide plusieurs documents (cf Section 5.6.4), certains d'entre eux peuvent être concernés par les mêmes corrections. Dans ce cas, une application précise peut utiliser notre solution pour pouvoir explorer les corrections alternatives pour un seul document. Ensuite l'application peut demander à l'utilisateur de sélectionner la solution adaptée et ainsi appliquer la séquence d'édition à tous les autres documents concernés.

Puisque notre algorithme trouve plusieurs solutions en dessous d'un seuil, une question inévitable est de savoir comment sélectionner la bonne valeur pour le seuil. D'une part si la valeur du seuil est inutilement élevée, le temps d'exécution peut être inacceptable. D'autre part si la valeur du seuil est très petite, l'algorithme peut échouer à proposer la solution souhaitée (ou aucune solution). Dans ces cas aussi, la réponse dépend fortement du contexte d'utilisation. Pour une application donnée, un scénario de test comme celui de la section 5.5 peut être utilisé pour estimer la valeur optimale pour une taille de document donnée. Cette estimation peut commencer en déterminant la distance d'édition entre t et $L(S)$, qui est calculée efficacement avec l'un des algorithmes spécialement conçus pour ce problème (cf. Section 5.6.1).

Notre algorithme peut être utilisé pour estimer la valeur optimale du th avec le scénario suivant : si le document n'est pas valide alors la distance est d'au moins 1, ainsi on peut chercher des solutions avec $th = 1$; si aucune solution de coût 1 n'est trouvée alors la recherche de solutions pour le $th = 2$ est exécutée, et ainsi de suite jusqu'à trouver une solution. Notre outil *XMLCorrector* utilise ce scénario lorsque l'utilisateur souhaite avoir seulement les corrections minimales. L'une des raisons pour utiliser ce scénario est le fait que nous sommes sûrs de ce que calcule notre algorithme. L'un des inconvénients est la complexité théorique d'un tel procédé, qui reste exponentielle et donc supérieure aux autres algorithmes capables de trouver la distance entre un document et un schéma. Cependant, comme montré dans les six scénarios de tests dans la Section 5.5, le comportement expérimental de telles solutions est polynomial plutôt qu'exponentiel.

Les avantages et les inconvénients de calculer toutes les solutions satisfaisant un critère bien défini est l'une des discussions dans plusieurs autres domaines, comme par exemple le cas des mots. En effet, pas trop loin de notre cas, le récent et très intéressant état de l'art dans [34] montre comment le problème d'approximation de « matching »de mots a reçu une attention particulière par différentes communautés scientifiques, qui ont souvent construit des algorithmes comparables, parfois en parallèle, pour leur contexte d'applications pratiques spécifiques. En prenant en compte maintenant la place que prend XML dans les systèmes d'informations, nous pouvons supposer que toutes les situations pour lesquelles la correction d'arbre à langage est utile [5] ne sont pas encore connues. C'est pour cela qu'une approche délibérée d'application indépendante, comme celle proposée dans ce chapitre malgré qu'elle soit trop complète, est une précieuse contribution.

Nous allons maintenant présenter des extensions qui ne sont pas encore implémentées dans l'outil *XMLCorrector* associé à l'article [7]. Le premier concerne les types de langages que notre algorithme peut traiter. Nous avons d'abord mis l'accent sur les DTDs, qui correspondent au « Local Tree Grammars (LTG) » dans la taxonomie de [89], mais nous montrons

5. étendue avec des tâches de pré-traitements et de post-traitements dépendant du contexte.

que le traitement des grammaires « Single Type Tree Grammars (STTG) », c'est-à-dire des schémas exprimés en XML Schema (XSD), est une extension simple.

Dans un schéma définissant un « Local Tree Grammars (LTG) » (tel qu'un DTD), chaque nom d'élément (ou symbole non-terminal) est associé à une unique définition de contenu (ou expression régulière). Dans un schéma définissant un « Single Type Tree Grammars (STTG) » (tel qu'un XML Schema (XSD)), il est possible d'avoir plus d'une définition de contenu pour le même nom d'élément, à condition que les présumés nom d'éléments en *concurrence* ne se retrouvent jamais simultanément dans une même définition de contenu. En d'autres termes, dans un XML Schema (XSD) il est possible d'associer plusieurs définitions de contenus à un même nom d'élément mais ils sont différenciés par leur contexte (ils peuvent être vus comme des définitions locales). Ainsi, la seule différence avec le cas des DTDs est que l'on doit résoudre la concurrence des noms d'éléments en utilisant les définitions des *nœuds parents*.

Notre algorithme peut facilement être adapté pour traiter cette classe de schémas puisque la récursivité va du haut vers le bas. Pour le nœud racine, il est garanti que la concurrence de noms d'éléments pour une étiquette donnée ne se produit jamais (dans un schéma XSD, seul les noms d'éléments globalement définis peuvent être une racine et il est interdit de définir deux noms d'éléments globaux avec la même étiquette). Lorsqu'on corrige un nœud qui n'est pas une racine, nous connaissons déjà le type de son nœud parent, et ainsi nous pouvons déterminer de façon unique la règle pour ce nœud dans son contexte. Comme il y a plusieurs éléments racine, l'appel initial à l'algorithme de correction doit être déclenché pour chacun des éléments racine et ceci n'ajoute qu'un facteur multiplicatif à la complexité globale.

D'autres extensions concernent des optimisations en temps et en espace de notre algorithme, inspirées d'autres travaux. Premièrement, rappelons que chaque case dans notre matrice d'édition M^c_u cf Section 5.3.4) est calculée à partir de ses trois cases voisines (gauche, haut, gauche-haut). Ainsi, en suivant l'idée énoncée dans [34] et implantée dans [104], seules deux colonnes consécutives doivent être stockées à chaque instant. Notons qu'un retour-arrière entraîne le recalcul des colonnes précédemment supprimées. Deuxièmement, comme aussi énoncé dans [34], plusieurs optimisations ont été proposées pour accélérer les implémentations de correction de mot à langage. Puisque notre algorithme étend ce problème, certaines de ces optimisations peuvent être appliquées, comme par exemple « bit-parallelism » (mis à jour simultanée de nombres dans un seul mot calculé) ou « FB-tries » (évoqué à la fin de la Section 5.5).

Une autre perspective intéressante est d'élargir la diversité des opérations d'édition élémentaires et leurs coûts. Par exemple nous pouvons considérer (similairement à [108], [105] et [117]) la possibilité d'insérer ou supprimer des nœuds internes, pas nécessairement des feuilles. Nous pensons que ce problème est lié à la *matrice d'édition étendue* sur les mots [34], dans laquelle les substitutions de mots arbitraires, plutôt que des opérations d'édition d'un seul caractère, sont autorisées. Dans le contexte des arbres, insérer ou supprimer un nœud interne va correspondre à remplacer une séquence de racines voisines par un nouveau nœud, ou vice verca. Aussi, pouvoir échanger des sous-arbres ou les déplacer dans tout l'arbre en une seule opération pourrait donner une bonne modélisation de la proximité entre documents dans différents domaines d'applications, comme discuté dans la littérature de correction d'un arbre vers un arbre [21, 54]. Finalement, l'utilisation de coût d'opération dépendant des positions où les opérations sont appliquées, pourrait être utile comme illustré dans [34].

```
<!ELEMENT NKJP_names (head,meta,sent*)>
<!ELEMENT head (schema)>
<!ELEMENT schema EMPTY>
<!ELEMENT meta (id,subId)>
<!ELEMENT id (#PCDATA)>
<!ELEMENT subId (#PCDATA)>
<!ELEMENT sent (seg|ne)+>
<!ELEMENT seg (orth,base)>
<!ELEMENT orth (#PCDATA)>
<!ELEMENT base (#PCDATA)>
<!ELEMENT ne (((orth, base)|(when, orth)),
              (derivType, (derivedFrom)?)?,
               cert, certComment?, children)>
<!ELEMENT when (#PCDATA)>
<!ELEMENT derivType (#PCDATA)>
<!ELEMENT derivedFrom (#PCDATA)>
<!ELEMENT cert (#PCDATA)>
<!ELEMENT certComment (#PCDATA)>
<!ELEMENT children (seg|ne)+>
```

FIGURE 5.16 – *La DTD correspondante au fichier XML utilisé pour les expérimentations*

5.5 Résultats expérimentaux

Plusieurs expérimentations ont été menées dans le but d'examiner les performances de notre algorithme sur un cas réel de données en fonction de différents paramètres : (i) la taille du document (ii) la valeur du seuil (iii) le nombre d'erreurs (iv) la position des erreurs (v) la nature du DTD. Dans cette section nous décrivons les paramètres des six scénarios de test et nous fournissons leurs résultats.

Nous avons utilisé un large document XML, que nous allons appeler le *fichier test*. Ce *fichier test* contient des annotations linguistiques des entités nommées, effectuées dans le projet Corpus National Polonais [99]. Le fichier test est conforme au DTD présenté dans la Figure 5.16. Cette DTD est assez riche pour couvrir différents aspects qui peuvent influencer les performances de l'algorithme de correction. Précisément, la DTD définit les éléments concernés par un degré variable de flexibilité (qui potentiellement donne un nombre variable de corrections).

Les métadonnées que sont les deux premiers éléments ⟨`head`⟩ et ⟨`meta`⟩ ainsi que leur trois nœuds fils ⟨`schema`⟩, ⟨`id`⟩ et ⟨`subId`⟩, correspondent à la partie de la DTD dans laquelle les opérateurs `?`, `|`, `+` et `*` ne sont pas autorisées dans les expressions régulières respectives. Ainsi si une erreur est introduite dans cette partie du fichier test, il n'y aura pas d'ambiguïté sur la façon de corriger cette erreur. C'est pour cela que cette partie du fichier test sera appelée *la partie non-ambiguë*.

Le reste du fichier test (qui sera appelé *la partie ambiguë*) est composé de 2638 phrases (⟨`sent`⟩) divisées en ⟨`seg`⟩ments (c'est-à-dire groupes de mots) et des entités nommées (⟨`ne`⟩). Chaque ⟨`seg`⟩ment contient sa forme ⟨`orth`⟩ographique (c'est-à-dire la forme flexionnelle apparaissant dans le texte, par exemple *domu*) et sa forme de ⟨`base`⟩ (c'est-à-dire le lemme, par exemple *dom*). La description des entités nommées est plus compliquée et autorise les opérateurs `?`, `|`, `+`, `*` et la récursivité. En particulier, les fils (⟨`children`⟩) d'une entité nommée (⟨`ne`⟩) sont des ⟨`seg`⟩ments et/ou d'autres entités nommées (par exemple *[ulica [[Kazimierza]*

[Pułaskiego]]] '[[[Kazimierz] [Pułaski]] Street]'). Ceci permet une récursivité non bornée incorporée d'entité nommées. Cependant dans la pratique pas plus de peu de niveaux incorporés apparaissent (3 dans notre fichier test). Comme résultat, notre fichier test XML a une structure assez plate. Il y a plusieurs phrases et chaque phrase contient plusieurs segments et/ou entités nommées (c'est-à-dire la racine a des milliers de fils et de petits-fils), tandis que les segments et les entités nommées sont représentés par des structures relativement peu profondes (c'est-à-dire la profondeur du document ne dépasse pas 10).

Le fichier test a été transformé de différentes façons afin de concevoir différents scénarios de test. Quatre paramètres d'entrée ont été pris en compte :

- la taille du document à corriger ;
- la distance seuil ;
- le nombre d'erreurs introduites ;
- les positions auxquelles sont introduites les erreurs (la partie ambiguë vs. la partie non-ambiguë, et le début vs. la fin du fichier test).

Deux types de résultats ont été examinés :

- les temps CPU nécessaires pour le processus de correction ;
- le nombre de résultats obtenus.

L'algorithme a été implanté en Java 1.6 et les tests sont exécutés sur une machine ayant un Intel Core i3-2310M 2.10GHx4 sous Ubuntu Oneiric Linux 11.10, avec 8 GB de RAM et un disque dur de 500 GB.

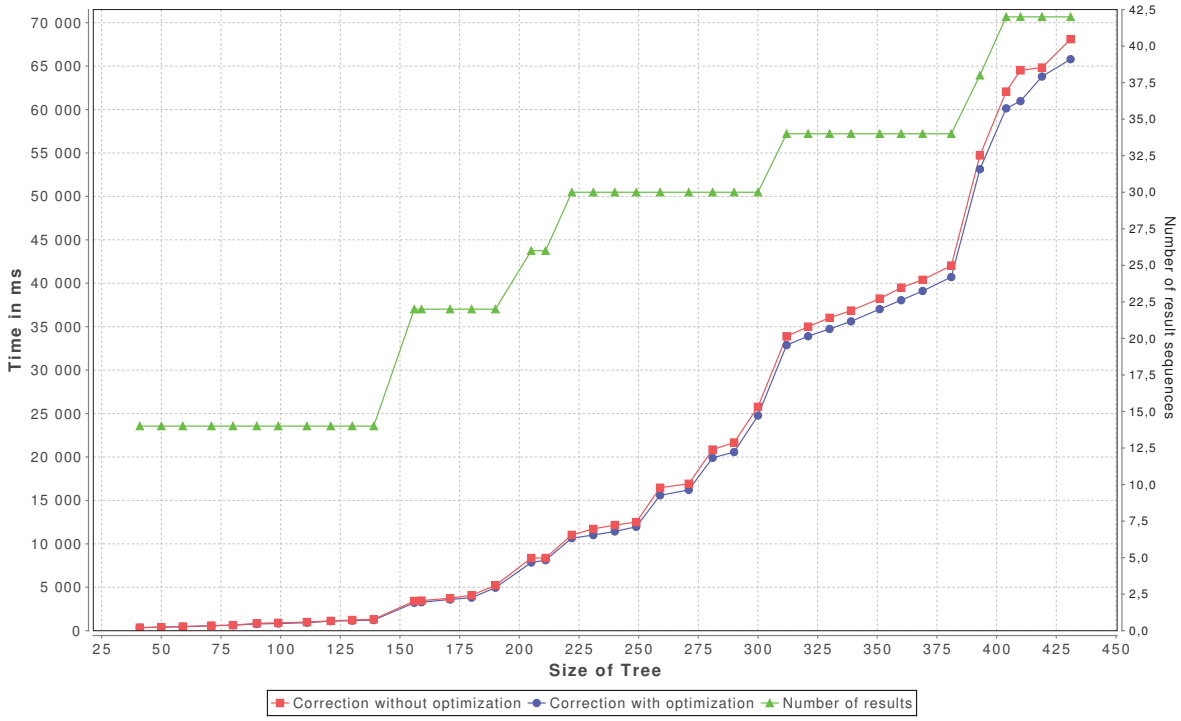


FIGURE 5.17 – *Temps CPU nécessaire pour le processus de correction et le nombre de candidats trouvés en fonction de la taille du document (nombre de nœuds) pour le th* = 2.

Le *premier* scénario est conçu pour tester le temps de correction et le nombre de candidats obtenus en fonction de la taille du document. Dans ce scénario :

- le fichier test a été réduit et répété dans des fichiers contenants les métadonnées, la première et la dernière phrase, certaines phrases qui suivent la première. Dans ce sens

nous obtenons une série de fichiers valides $f_1^1, \ldots, f_n^1$ tels que f_i^1 et f_{i+1}^1 diffèrent de 10 à 20 nœuds.

– une erreur a été introduite dans la première phrase de tous les fichiers f_i^1 (l'élément ⟨base⟩ a été supprimé en dessous de l'élément ⟨ne⟩). Ainsi, l'erreur apparaît au début de chaque fichier dans la partie ambiguë.

– le seuil est de $th = 2$.

La Figure 5.17 montre les résultats de ce scénario. Notons que le temps CPU a un comportement polynomial, malgré la complexité théorique exponentielle décrite dans la Section 5.3.5. Nous pensons que la grande différence peut résulter d'une estimation assez grossière de la complexité due à la nature très récursive de notre algorithme (les factorisations des préfixes d'un chemin offertes par l'automate, sont difficiles à exprimer dans l'estimation du pire des cas). L'optimisation décrite à la fin de la section 5.3.5 (qui consiste à stocker les résultats intermédiaires dans une structure auxiliaire) a une influence mineure sur le temps d'exécution : ceci peut s'expliquer par le fait que la correction de sous-arbres avec moins de 150-200 nœuds est moins coûteuse que d'utiliser la structure auxiliaire. Dans nos expériences, les sous-arbres similaires n'ont pas plus de 50 nœuds. Nous pouvons aussi noter le temps CPU montre quelque irrégularités ayant un rapport étroit avec le nombre de corrections trouvées. Par exemple pour des tailles de fichiers entre 300 et 370, le temps CPU croit linéairement tant que le nombre de résultats est inférieure à 26. Cependant le temps CPU croit rapidement lorsque le nombre de résultats candidats atteint 30 et 33.

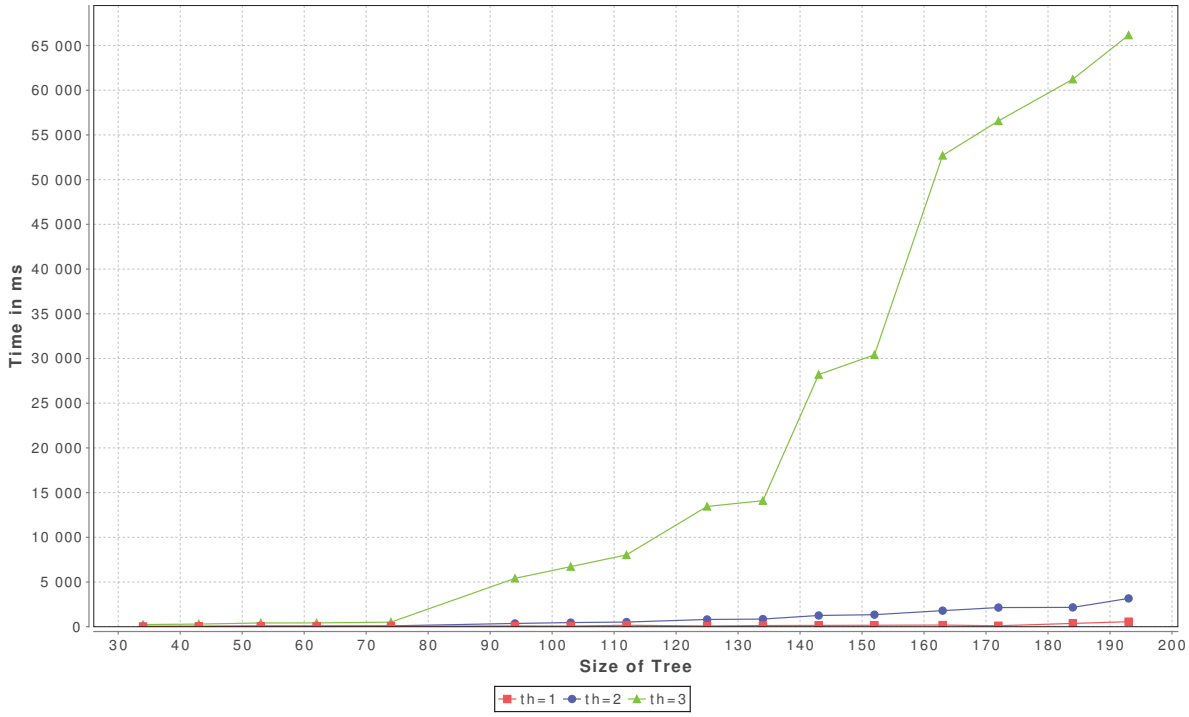


FIGURE 5.18 – *Temps CPU nécessaire pour le processus de correction avec différentes valeurs du seuil, en fonction de la taille du document (nombre de nœuds).*

Le *second* scénario est destiné à montrer comment la valeur de la distance seuil *th* a une influence sur le temps d'exécution. Dans ce scénario :

– Le fichier test a été réduit et répété dans des fichiers valides $f_1^2, \ldots, f_n^2$, comme dans le scénario précédent.

– Une erreur a été introduite au début de chaque fichier f_i^2 dans sa partie non-ambiguë (l'élément ⟨schema⟩ a été supprimé sous l'élément ⟨head⟩).

– Le seuil a été fixé à $th = 1$, $th = 2$ et $th = 3$ pour les trois étapes de l'expérience, respectivement.

Comme illustré dans la Figure 5.18, le temps de correction croit considérablement pour de gros documents et des seuils élevés. Par exemple pour un document de 193 nœuds, le temps nécessaire pour le $th = 3$ est d'environ 20 et 114 fois plus grand pour $th = 2$ et $th = 1$, respectivement. Ceci s'explique simplement par le fait que la valeur du seuil est l'un des facteurs importants limitant l'espace de recherche dans l'automate exploré durant le processus de correction (les chemins d'exploration sont coupés dès que la distance d'édition de l'arbre solution partiel précédemment obtenu dépasse le seuil th).

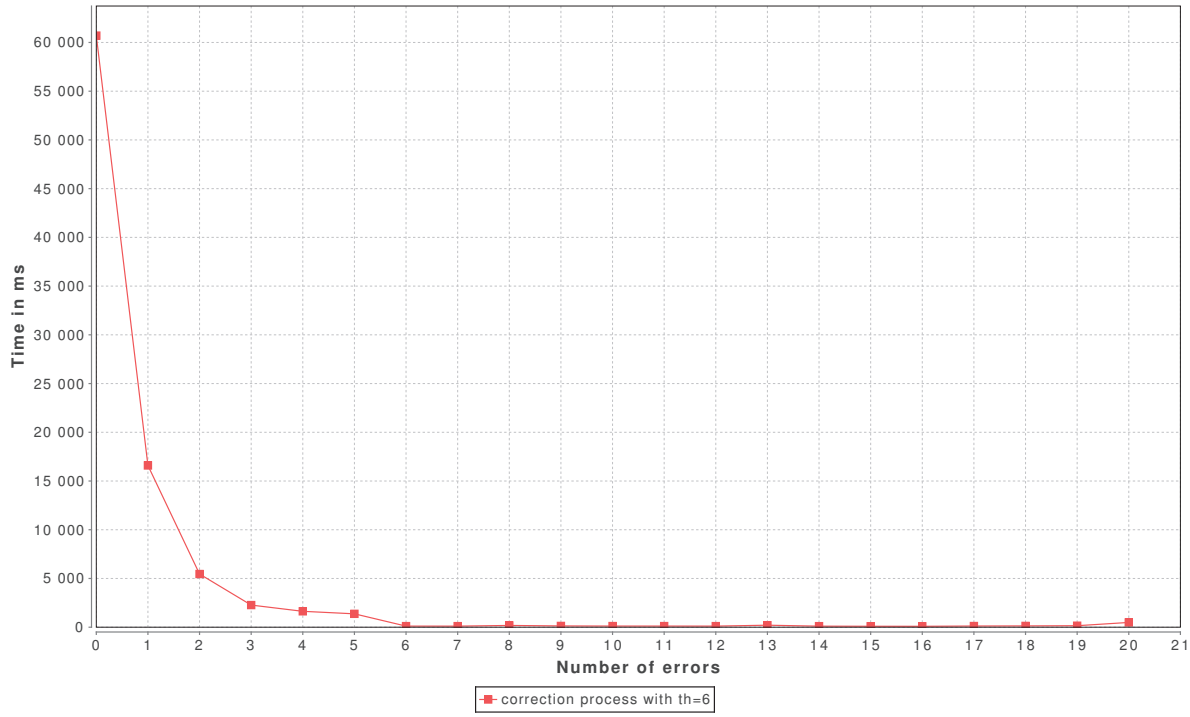


FIGURE 5.19 – *Temps CPU nécessaire pour le processus de correction avec le seuil $th = 6$ en fonction du nombre d'erreurs dans le fichier test*

Dans le *troisième* scénario, nous examinons l'influence du nombre d'erreurs introduits dans le fichier test sur le temps d'exécution durant le processus de correction. Dans ce scénario :

– Le fichier test réduit en un fichier valide f^3 contenant une phrase seulement de 44 nœuds.

– Les erreurs introduites sont répétées dans le fichier f^3, en commençant à gauche et finissant avec les nœuds de droites. Comme résultat, 21 fichiers $f^3_0, \ldots, f^3_{20}$ sont produits tels que le fichier f^3_{i+1} a une erreur de plus que le fichier f^3_i.

– Le seuil est fixé à $th = 6$.

Comme illustré dans la Figure 5.19, le temps CPU nécessaire pour le processus de correction est plus grand pour les fichiers avec peu d'erreurs, et faible pour des fichiers avec plusieurs erreurs. Ceci s'explique par le fait que le processus de correction peut continuer seulement si les sous-arbres placés à gauche du nœud courant peuvent être corrigés. Puisque chaque arbre correction partiel fait croître la distance d'édition, l'espace de recherche est réduit rapidement si plusieurs erreurs apparaissent (particulièrement proche du début du fichier).

Si plus de 6 erreurs apparaissent, il n'y aura pas de correction possible ce qui fait que le temps de correction est réduit considérablement. Remarquons aussi que la Figure 5.19 montre un temps important pour le document avec 0 erreurs, c'est-à-dire un document valide. Comme nous sommes plutôt intéressés par la correction de fichier seulement s'ils sont pas corrects, ce résultat n'est pas pertinent. Dans ce cas la correction doit être précédée par la validation. Notre algorithme répond à un problème plus général : celui de trouver tous les arbres valides dont la distance avec l'arbre à corriger (sans tenir compte de sa validité) n'est pas plus grand que le seuil.

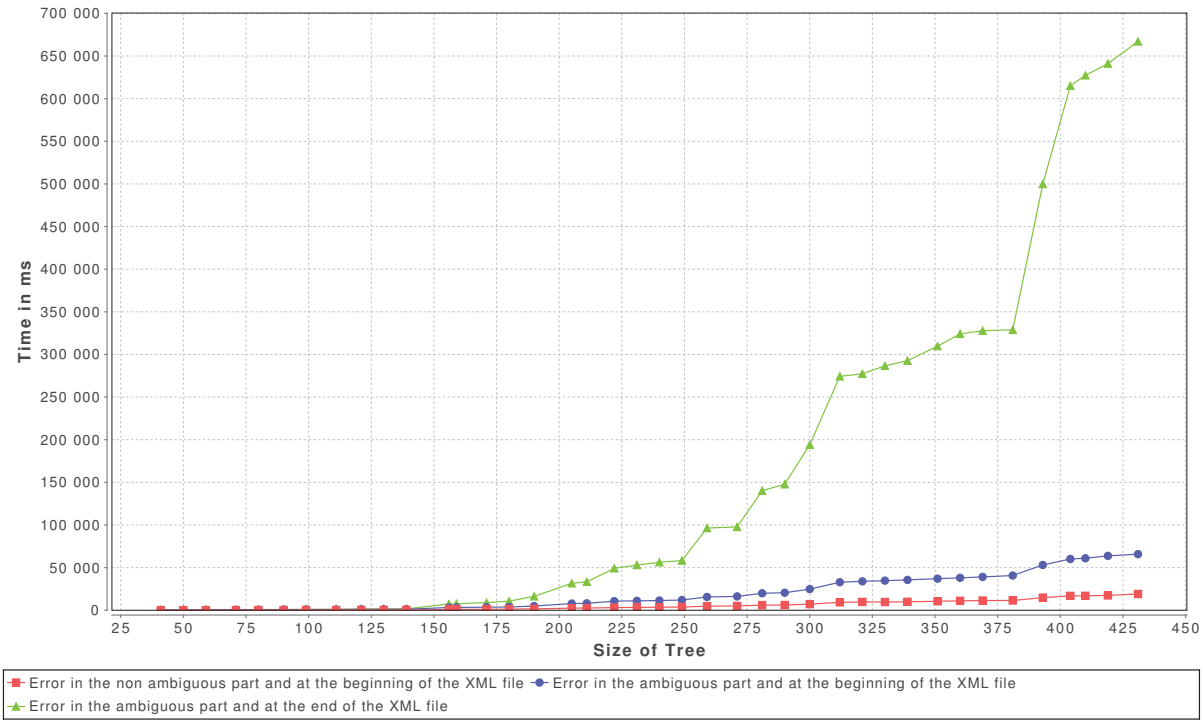


FIGURE 5.20 – *Temps CPU nécessaire pour le processus de correction avec une erreur placée à différentes positions du fichier test, et avec* $th = 2$*, en fonction de la taille du document (nombres de nœuds)*

L'hypothèse pour le *quatrième* scénario était d'examiner si la position de l'erreur dans le fichier test, et sur la nature de la DTD ont une influence sur le temps de correction, et le nombre de solutions candidates produites. Dans ce scénario :

- Le fichier test a été réduit en un fichier valide f^4 contenant 530 nœuds.
- Le fichier test était ensuite réduit et répété en des fichiers valides $f^4_1, \ldots, f^4_n$, comme dans le premier scénario.
- Un fichier invalide $f^4_{(i,1)}$ est créé en introduisant une erreur dans les métadonnées du fichier f^4_i, pour chaque $1 \leq i \leq n$ (l'élément ⟨`schema`⟩ est supprimé sous l'élément ⟨`head`⟩). Ainsi, cette erreur apparaît dans le début du document et est non-ambiguë, c'est-à-dire qu'il existe une seule façon de corriger l'erreur.
- Deux autres fichiers invalides $f^4_{(i,2)}$ et $f^4_{(i,3)}$ sont créés en introduisant une erreur soit dans la première phrase, ou soit dans la dernière phrase du fichier f^4_i (l'élément ⟨`base`⟩) est supprimé sous l'élément ⟨`ne`⟩, c'est-à-dire dans la partie ambiguë du fichier). Ces erreurs sont ambiguës, c'est-à-dire qu'il existe plusieurs façons de les corriger.
- Le seuil a été fixé à $th = 2$.

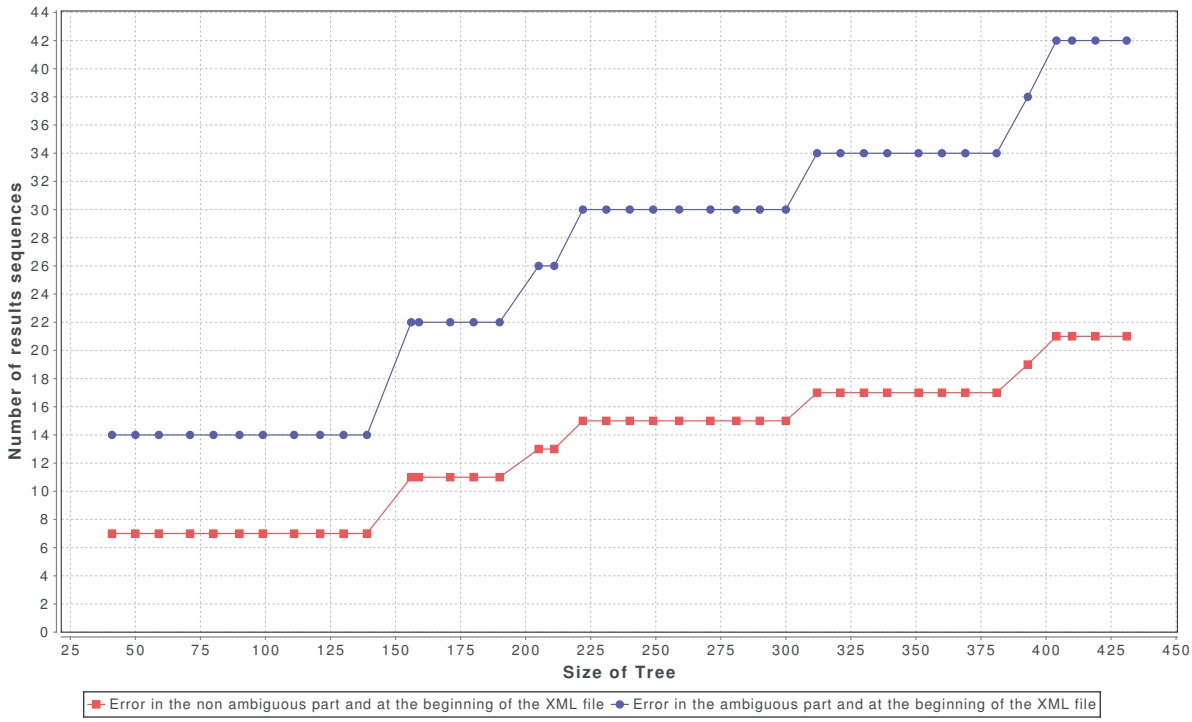


FIGURE 5.21 – *Nombre de solutions candidats trouvés avec une erreur placée à différentes positions du fichier test, et avec th* = 2*, en fonction de la taille du document (nombres de nœuds)*

Comme illustré dans la Figure 5.20, le temps CPU nécessaire pour la correction reste relativement bas lorsqu'une erreur apparaît dans le début du fichier, tandis que le temps croit considérablement lorsque l'erreur apparaît à la fin du fichier. Ceci provient probablement du fait que les sous-arbres à gauche doivent être corrigés avant le nœud courant qui doit être examiné. Ainsi, si les erreurs sont proches du début du fichier, la distance d'édition augmente rapidement et l'espace de recherche est très tôt réduit. Inversement, si tous les sous-arbres gauches sont valides, la distance d'édition est égale à 0 et l'espace de recherche reste limiter par le seuil initial.

La Figure 5.21 montre que la correction pour une erreur apparaissant dans la partie non-ambiguë est plusieurs fois plus rapide que dans la partie ambiguë. Dans cette figure aussi, ce résultat est étroitement lié avec le nombre de corrections trouvées, qui est deux fois plus petit dans le premier cas que dans le second cas à cause des alternatives autorisées par la DTD pour l'élément ⟨ne⟩.

Le *cinquième* scénario permet d'examiner l'influence du seuil *th* sur le temps de correction et le nombre de solutions candidates trouvées. Dans ce scénario :

– Le document à corriger est le document vide.

– Le seuil varie entre 1 et 15.

Comme illustré dans la Figure 5.22, le temps CPU et le nombre de candidats trouvés sont de nature polynomiale par rapport à la valeur du seuil.

Le dernier et *sixième* scénario a pour but de montrer comment notre algorithme se comporte, lorsqu'il est utilisé pour déterminer l'ensemble des corrections de coût minimal (par exemple pour aider l'utilisateur à choisir la bonne valeur du seuil nécessaire pour son application, comme expliqué dans la Section 5.4). Dans ce scénario :

– Le même ensemble de 21 fichiers (avec un nombre d'erreurs croissant) que celui utilisé dans le troisième scénario.

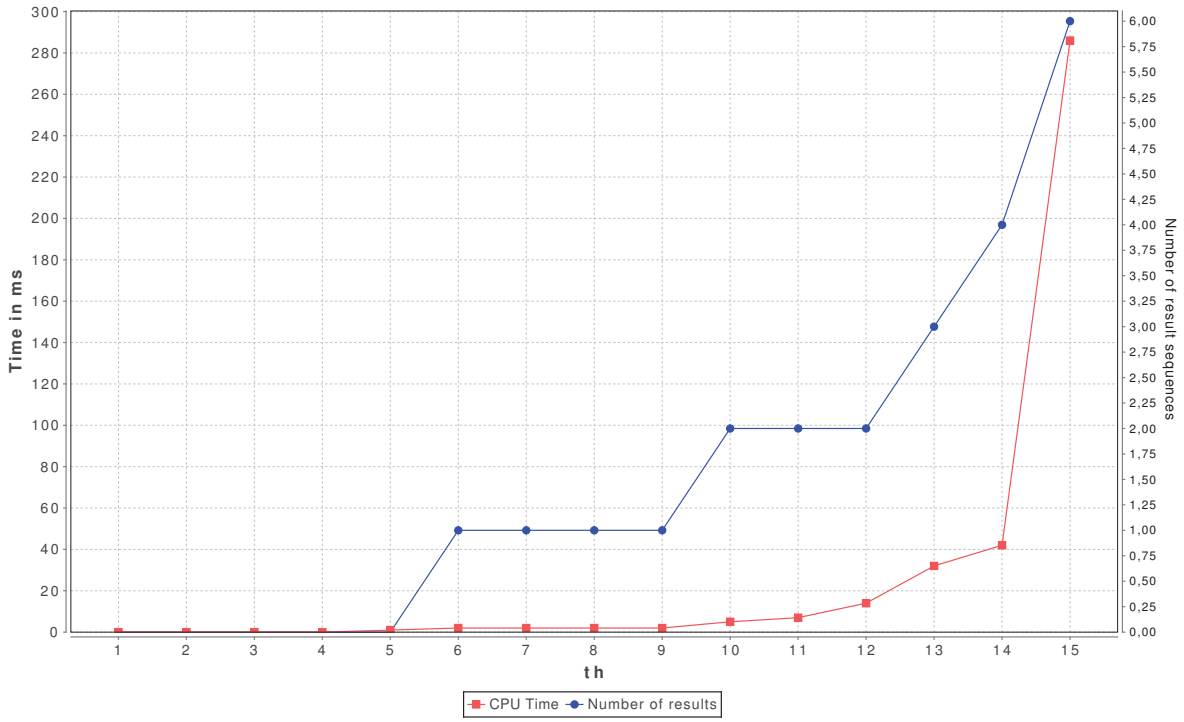


FIGURE 5.22 – *Temps CPU nécessaire et le nombre de candidats trouvés pendant la correction de l'arbre vide, en fonction du seuil th.*

– L'algorithme a été exécuté à plusieurs reprises avec $th = 1$, ensuite avec $th = 2$, etc., jusqu'à ce que la première solution est trouvée.

Comme illustrée dans la Figure 5.23, le comportement de la courbe de ce scénario est aussi polynomial plutôt qu'exponentiel.

Pour conclure, en dépit de la complexité théorique exponentielle, notre algorithme montre un comportement qui est plutôt polynomial en fonction de la valeur du seuil (Figure 5.22), et de la taille du document (Figures 5.17, 5.18 et 5.20). Curieusement, plus la distance du document invalide au schéma est grande, plus le temps de correction est petit (Figure 5.19). Ceci est probablement dû au fait que les erreurs apparaissant proche du début du fichier réduisent rapidement le temps de correction (Figure 5.20), qui résulte de la correction des sous-arbres voisins du même niveau de la gauche vers la droite. Une optimisation possible serait d'effectuer la correction des sous-arbres voisins aussi bien de la gauche vers la droite que de la droite vers la gauche, comme pour les « FB-tries » de Mihov et Schulz décrit dans [34]. Comme attendu, si les erreurs apparaissent dans la partie ambiguë du fichier (c'est-à-dire les parties concernées par les opérateurs `?`, `|`, `+` et `*` du schéma) alors leur correction prend un temps plus important que la correction des erreurs se trouvant dans la partie non-ambiguë (Figure 5.20). Finalement, le temps de correction est étroitement lié avec le nombre de corrections candidates trouvées (Figures 5.17 et Figures 5.22), qui à son tour se traduit à partir des facteurs mentionnés ci-dessus : la taille du document d'entrée, la valeur du seuil, la position d'une erreur et la nature du schéma. Cette corrélation entre le temps de correction et le nombre de candidats peut expliquer, au moins en partie, pourquoi le problème de correction d'un arbre vers un langage défini dans notre approche est plus difficile à résoudre que dans d'autres travaux discutés dans la Section 5.6. En particulier ce problème est souvent réduit dans la littérature à trouver seulement la distance entre un arbre et un langage sans proposer une séquence pour la correction, ou proposer un nombre fini

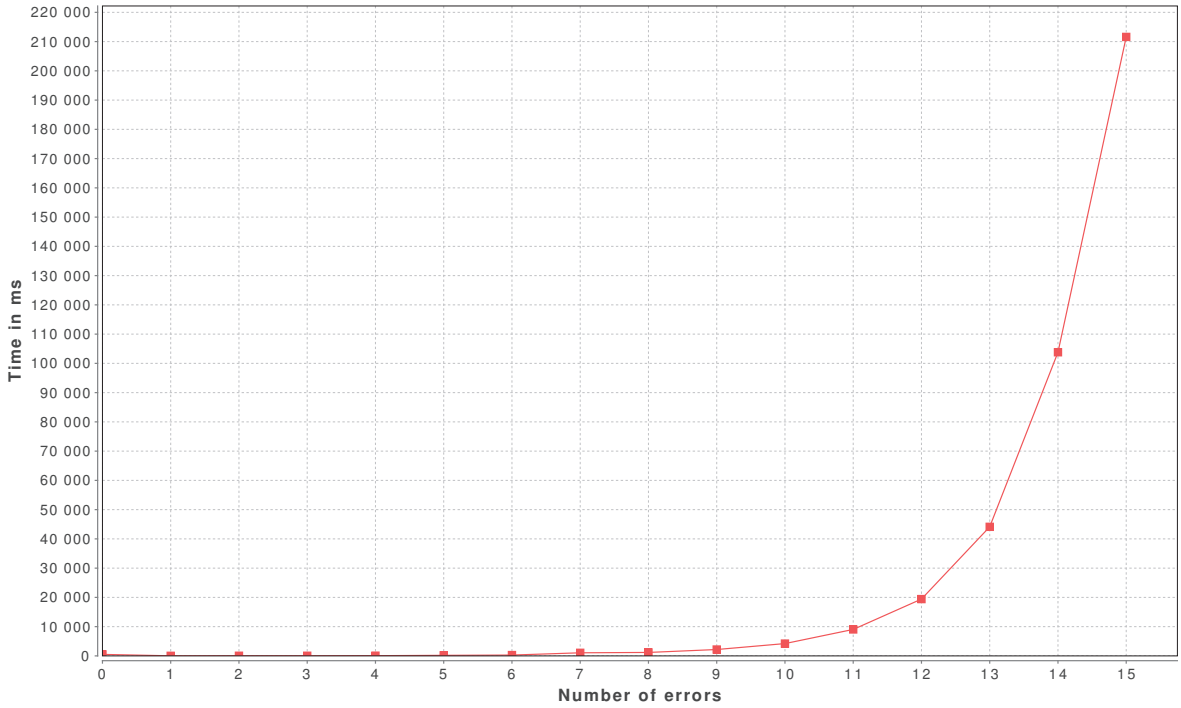


FIGURE 5.23 – *Temps CPU nécessaire pour la recherche de la distance entre un arbre et un schéma, en fonction du nombres d'erreurs introduites dans le fichier test.*

de séquences minimales seulement (cf. Tableau A.1). Si la complétude de notre correction est requise alors le temps de correction croît en conséquence.

5.6 Travaux liés à la correction de document

Dans cette section, nous allons résumer plusieurs définitions du problème de la correction d'arbre vers un schéma (« tree-to-schema correction » en anglais) ainsi que les différents modèles d'algorithme correspondants qui existent dans la littérature. Nous pouvons trouver trois catégories importantes dans les travaux proches de notre algorithme de correction de documents XML :

(i) *la mesure de la distance entre un document XML et un schéma* qui est souvent lié à

(ii) *la recherche d'une ou toutes les solutions de coût minimal*,

(iii) et la tâche la plus complexe qu'est *le calcul d'un ensemble de séquences d'opérations d'édition pour corriger un document XML par rapport à un schéma*.

De plus, la *revalidation et la correction* intervenant *après des mises à jour effectuées sur le document ou le schéma* peuvent être vues comme une instance particulière du problème de correction d'un arbre vers un schéma. Nous abordons toutes ces directions dans les sous-sections suivantes. Dans notre papier [7], une analyse comparative des différentes proposition a été faite, ce qui nous a permis de situer notre travail par rapport aux autres travaux. Nous pouvons trouver une synthèse de cette analyse dans le Tableau A.1 (type d'opérations, validité de la structure, validité des attributs, type de schéma, solutions minimales, utilisation de seuil, séquences d'édition, modèle de document et modèle de schéma) et le Tableau A.2 (estimation de la complexité en temps et en espace, les preuves, nature des données utilisées dans les expérimentations, la disponibilité du code source, la licence, les benchmark) en Annexes, section A.8.

5.6.1 Mesurer la distance entre un document XML et un schéma

Une étude d'évaluation globale dédiée au problème de comparaison entre document XML et grammaire, ainsi que le problème de calcul de distance ou de similarité entre document XML et grammaire est présentée dans [114]. Les auteurs présentent aussi un aperçu complet des applications pour des solutions au problème, qui sont aussi des applications potentielles pour notre travail.

Dans [19, 20] la notion de *structures similaires* entre un document XML t et une DTD D est définie et un algorithme pour calculer cette mesure est présenté. Cette similarité entre structures consiste en la valeur maximale d'une fonction qui évalue un ensemble de mappings à partir de t vers D. L'évaluation repose sur le calcul de trois paramètres dépendant de plusieurs configurations requises. Ces trois paramètres sont : (i) le nombre de parties en *commun* c, (ii) le nombre de parties en *extra* (c'est-à-dire qui ne sont pas dans le schéma) m, (iii) le nombre de parties *manquantes* dans le document. L'algorithme pour calculer la similarité est basé sur la fonction *Match* qui calcul les meilleurs mappings et qui a une complexité en $O((|t| + |D|) \times a^2)$ où $|t|$ est la taille du document, $|D|$ est la taille de la DTD et a est le nombre maximal d'arité de t. Les résultats expérimentaux montrent de bonnes performances. Le principal avantage d'une telle mesure est sa flexibilité à prendre en compte d'autres facteurs (comme par exemple l'importance du niveau auquel sont détectées les parties en communs ou divergentes, ou l'égalité des étiquettes contre la similarité d'étiquettes, etc.) qui peuvent être configurés par les utilisateurs et qui dépendent du domaine d'applications. Cette flexibilité n'est pas le cas dans la proposition [115] qui aussi mesure la similarité entre un document et une DTD, mais en modélisant le document et la DTD comme des arbres ordonnés et étiquetés. Pour cela ils adaptent la mesure de la distance d'édition entre arbres à la particularité de leur DTD représentés sous forme d'arbres spéciaux. Ils obtiennent donc un algorithme polynomial pour trouver la séquence d'opérations d'édition de coût minimal entre l'arbre XML et l'arbre représentant la DTD. Cette séquence et son coût sont considérés comme la mesure de similarité.

A première vue, [125] traite un problème qui est similaire au nôtre : celui de trouver une séquence d'opérations élémentaires d'éditions (renommer un nœud, supprimer ou insérer une feuille) de coût minimal qui transforme un document XML en un autre document XML conforme à un schéma. Leur travail prend en compte aussi bien des DTDs que des XML schemas (XSD). Contrairement à ce but initial, leur algorithme qui est décrit propose seulement la distance d'édition entre l'arbre et le schéma, et ne permet pas de trouver la séquence d'édition pour corriger l'arbre initial. Leur algorithme est basé sur la programmation dynamique (comme le nôtre) et leur complexité est de $O(n \times p \times \log p)$, où n et p sont les tailles de l'arbre et du schéma respectivement. Cependant leur pseudo-code ne permet pas de vérifier cette complexité. Un point intéressant dans ce papier est la description d'une application « framework » qu'est la classification de documents XML. Des arguments convaincants ont été donnés en faveur de l'utilisation de la distance entre document et schéma plutôt qu'entre document et document pour cette application. Cependant c'est la distance d'édition qui est utilisée et non la séquence d'édition, ce qui fait qu'il est difficile de savoir si la procédure de correction a été réellement étudiée et implémentée.

Dans [104] le problème de la distance d'édition entre un document XML d et une DTD D est défini, comme dans notre approche, comme le coût minimal parmi les coûts de toutes les séquences transformant d en un arbre valide d' de D. Les opérations d'éditions considérées sont la suppression et l'insertion de feuille. De complexes *macro-opérations* sont définies sur ses opérations d'éditions (comme dans le papier [28]) et sont les suivantes : (i)

la suppression de sous-arbre, (ii) l'insertion d'un sous-arbre valide de taille minimale avec une étiquette racine donnée, (iii) la correction récursive d'un sous-arbre. L'algorithme est basé sur un *graphe de restauration* ressemblant à une matrice avec N lignes et $M+1$ colonnes, où N est le nombre d'états dans l'automate d'états finis correspondant à l'étiquette racine (FSA_r), et M est le nombre de fils pour le nœud racine. Le sommet q_i^j dans le graphe à la ligne i et colonne j représente à la fois l'état courant dans FSA_r ($0 \leq i < N$) et la position courante dans le mot formé par les fils de la racine ($0 \leq j \leq M$). Chaque arête dans le graphe correspond à une macro-opération appliquée aux fils de la racine et ayant un poids égal au coût de cette macro-opération. En particulier le coût de la correction d'un sous-arbre nécessite une construction récursive du graphe de restauration sur le sous-arbre courant. Le problème de trouver la distance d'édition entre un document et une DTD est ainsi réduit à la recherche du chemin le plus court dans le graphe de restauration à partir de l'arête q_0^0 vers une arête représentant un état final dans la dernière colonne. La complexité en temps indiquée dans le papier est de $O(|D|^2 \times |T|)$ où $|D|$ et $|T|$ sont les tailles de la DTD et du document respectivement. L'analyse de la complexité est vraiment brève et n'est pas claire sur le fait que la récursivité est prise en compte (la correction du même nœud vers le même automate doit nécessairement être faite plusieurs fois dans l'arbre). La complexité en espace annoncée est de $O(|D|^2 \times height(T))$ puisque seulement deux colonnes consécutives du graphe de restauration ont besoin d'être sauvegardées à la fois. Il est mentionné que la restauration des séquences d'opérations à partir du graphe est triviale puisque chaque séquence provient de l'un des chemins du graphe de longueur minimum. Cependant aucune indication n'est donnée sur les chemins qui sont considérés (si c'est tous les chemins ou juste un seul et lequel). De plus aucune information n'est donnée sur les positions de mise à jour dans l'arbre modifié. Les résultats expérimentaux sur des documents XML invalides générés aléatoirement montrent un comportement linéaire de l'algorithme en fonction de la taille du document (avec une DTD de taille 5) et aussi un comportement polynomial en fonction de la taille de la DTD.

Dans [105] les auteurs reconsidèrent le problème de la distance d'édition entre un document et une DTD en représentant la DTD par un « *streaming tree automaton* (STA) », c'est-à-dire un simple automate à pile qui opère sur un document XML en entrée vu comme une séquence de balises ouvrantes et fermantes (dans l'ordre standard du document) plutôt qu'un arbre. Étant donné un STA pour la DTD, un *automate de réparation* est défini et il ressemble aussi à une matrice dont : (i) le nombre de lignes est égal au nombre d'états dans la STA, (ii) le nombre de colonnes est égal au nombre de balises ouvrantes et fermantes dans le document XML. Les transitions dans l'automate de réparation sont pondérées et ses transitions représentent les opérations d'édition (et leur coûts). Comme opération nous avons (i) le renommage de nœud, (ii) l'insertion d'un nœud (pas nécessairement une feuille), (iii) la suppression d'un nœud (pas nécessairement une feuille). La suppression ou l'insertion de la racine n'est pas autorisée. Le calcul de la distance entre un document et une DTD est , ici aussi, réduit à la recherche d'un mot/chemin acceptant de poids minimal dans l'automate de réparation. Cette approche semble prometteuse du fait : (i) qu'elle considère un ensemble riche d'opérations d'édition (l'insertion et la suppression de nœuds internes sont autorisées), (ii) la bonne formation du document et la validité des attributs sont toutes les deux prises en compte au sein du même framework pour la validité de la structure du document. L'algorithme est décrit dans un problème plus complexe qu'est l'interrogation d'ensembles de documents XML (éventuellement incompatibles), prouvé EXPTIME-complet. La complexité en temps du problème du calcul de distance n'est pas explicitement donnée. Nous regret-

tons qu'il n'y est pas de résultats expérimentaux pour cette proposition, en particulier par rapport à de gros documents et de DTDs complexes.

5.6.2 Trouver une seule correction, ou toutes les corrections de coût minimal

La référence [25] est fréquemment citée car il est probablement le premier à annoncer le défi suivant : étant donné un arbre T et une grammaire d'arbres G (telle que $T \notin L(G)$, $L(G)$ étant le langage d'arbres défini par G), comment trouver un arbre T' qui est dans $L(G)$ et qui n'est pas trop distant de T. Comme dans beaucoup d'autres frameworks, les arbres représentent les documents XML et la grammaire est une DTD. Cependant dans ce papier les arbres sont des arbres d'arité bornée, ce qui n'est pas le cas des documents XML, et donc les algorithmes travaillent sur une représentation binaire des arbres qui n'est pas présentée. Les opérations d'éditions élémentaires suivantes sont considérées : le *renommage* de nœud, l'*insertion* et la *suppression* de nœud à n'importe quel niveau dans l'arbre. Les auteurs mentionnent que l'un d'entre eux a prouvé dans un autre article l'existence d'un proptotype pour la distance d'édition d'arbre avec une opération d'édition spéciale appelée *move*, qui permet de déplacer tout un sous-arbre d'un nœud à un autre nœud dans l'arbre. Ce papier est focalisé sur des résultats expérimentaux pour une implémentation d'algorithmes précédemment publiés. Ces algorithmes traitent la classique distance d'édition d'arbres (sans l'opération *move*). Le premier consiste à marquer les nœuds parents d'un nœud erreur durant un parcours de bas en haut de l'arbre, et ensuite un nouveau parcours de haut en bas pour modifier le voisinage des nœuds marqués. La façon dont ces modifications sont exécutées, n'est pas présentée. La complexité de ma méthode est linéaire en la taille de l'arbre et exponentielle en fonction du nombres d'erreurs. Le but du second algorithme est d'éviter le facteur exponentiel mais il calcul un arbre T' qui n'est peut être pas optimal. Les expériences faites concerne seulement le deuxième algorithme, et 4 DTDs sont prises en compte. Pour chaque DTD, 5 documents XML sont automatiquement générés avec des tailles entre 50 et 800 nœuds et toujours avec 10 erreurs dont le type et les positions ne sont pas spécifiées. Les temps d'exécution (pour trouver T') sont rangés entre 900 et 2400 millisecondes. Une page web est donnée pour tester l'implémentation mais cette page n'est plus maintenue.

Dans [112, 113] le problème de corriger un document XML vers un schéma est basé sur les définitions dans [28] de Béatrice Bouchou et Agata Savary qui sont mes co-auteurs dans [7] à la base de ce chapitre. Les auteurs [112, 113] ont étendu les définitions de [28] pour : (i) traiter les grammaires STTG, c'est-à-dire des XML schemas (XSD) et pas seulement des DTDs, (ii) intégrer la correction d'attributs et pas seulement des éléments. Leur algorithme adapte aussi les idées de [104] qui consiste à modéliser les corrections par la recherche de chemins de longueur minimal dans le graphe de correction, cependant non seulement il trouve la distance d'édition entre l'arbre et le schéma mais aussi il trouve les séquences d'édition nécessaires pour obtenir les arbres solutions. Ils utilisent aussi l'idée de mise en cache des résultats partiels, comme dans notre approche. Les auteurs mentionnent les résultats expérimentaux dont le temps utilisé montre un comportement linéaire en fonction de la taille du document. Cependant la nature des données et les conditions de l'expérience ne sont pas décrites. L'implémentation de leur approche est téléchargeable mais les données utilisées dans leur expérience ne sont pas fournies. Par soucis de comparaison, nous avons essayer de lancer leur implémentation sur nos données tests, qui a donné comme résultat un crash de programme. Nous pouvons noter que la forme de leur schémas n'est pas standard et il semble que les définitions récursives ne sont pas autorisées. Il est aussi suggéré que trouver toutes les solutions possibles en dessous d'un seuil et pas seulement les solu-

tions minimales (ce qui va rendre leur framework très proche du nôtre), est possible après quelques modifications de l'algorithme original. Cependant ces modifications n'ont pas été définies.

5.6.3 Calculer un ensemble de séquences d'opérations d'édition pour corriger des documents XML vers un schéma

En dehors des papiers [112, 113] mentionnés ci-dessus, la proposition dans [108] est la seule à notre connaissance qui calcule l'ensemble de scripts d'édition (c'est-à-dire des séquences d'opérations) entre un document XML t et une grammaire d'arbres (les attributs ne sont pas considérés). Leur approche n'utilise pas un seuil th, mais un nombre K des corrections optimales désirées, c'est-à-dire que leur approche cherche les K corrections de plus petit coût. Le premier résultat de ce papier est la preuve que ce problème est NP-hard [6]. Le second résultat est un algorithme en temps pseudo-polynomial pour les grammaires « one-unambiguous ». Les trois opérations d'éditions considérées, qui sont différentes des nôtres, sont : $ren(n_i, a)$ qui assigne l'étiquette a au nœud n_i, $add(a, n_h, n_j)$ qui ajoute le nœud $n_{h,j}$ étiqueté par a comme le parent des frères $n_h, \ldots, n_j$, et $del(n_i)$ qui supprime le nœud interne n_i et remplace ce nœud par ses enfants. Comme restriction, seulement l'opération ren peut être appliquée à la racine et il n'est pas possible de supprimer des nœuds. Aussi un coût est associé à chaque opération d'édition. L'auteur définit en premier un graphe $H_K(t, N)$ contenant $|t| + 1$ nœuds et environ $|N| \times K \times 2 \times |t|$ arêtes, avec N étant l'ensemble des non-terminaux dans la grammaire. Le graphe $H_K(t, N)$ représente tous les changements qui peuvent être appliqués à t. Certains chemins dans ce graphe correspondent à des séquences de frères, en outre les arêtes sont associées à des opérations d'édition et leur coûts. Ensuite l'auteur propose de considérer des sous-graphes correspondant à tous les fils du même nœud n dans le but de calculer *les intersections* entre les sous-graphes et les automates (d'états finis non-déterministe) qui représentent les règles de productions associées à l'étiquette de n. Trouver les K premiers script d'édition consiste à calculer les k chemins les plus courts dans le graphe résultant. Comme il est possible de trouver K scripts d'édition optimals pour t à condition qu'il existe des scripts d'éditions optimaux connus pour chaque sous-arbre propre de t, l'algorithme s'exécute à partir des arêtes faibles vers les arêtes élevées dans $H_K(t, N)$, tout en suivant un ordre partiel spécifique. Malgré qu'il n'est pas toujours pertinent de fournir K scripts d'édition optimaux (comment l'algorithme choisit entre deux scripts ayant le même coût ?), les opérations d'édition utilisées dans cette approche sont intéressantes et l'algorithme semble prometteuse, considérant le fait que sa complexité est dite pseudo-polynomiale. Cependant le papier ne mentionne pas de résultats expérimentaux et il n'est pas possible de déterminer s'il existe une implémentation de ces idées depuis sa publication.

5.6.4 La revalidation et la correction après des mises à jour du document ou du schéma

Certaines instances particulières du problème de correction d'un arbre vers un schéma apparaissent dans le contexte de l'évolution des documents XML et schémas, notamment sur le web.

D'une part, les *documents XML* qui sont connus valides peuvent être *soumis à des mises à jour*, qui peuvent les invalider. Deux approches complémentaires sont proposées dans ce cas :

6. Trouver une solution est exponentiel en la taille des entrées.

- on peut essayer de déduire un nouveau schéma à partir des mises à jour du document, comme dans [31] et plus récemment dans [102], où l'évolution de schéma est concernée dans une approche streaming (de flux).
- on peut aussi essayer de corriger le document pour restaurer sa validité, cependant la priorité est donnée aux changements récents afin de les préserver. Les travaux dans [28, 29] offrent une telle solution dans un framework *incremental*. En fait à partir d'un document valide t, sur lequel a été effectué des changements, les auteurs ciblent le processus de correction sur les parties du document concernées par les changements. Durant la phase de revalidation incrémentale, une routine de correction est activée seulement si un sous-arbre invalide est rencontré. A la fin de la revalidation, les corrections générées par chaque appel à la routine sont combinées et plusieurs versions de documents valides sont proposées à l'utilisateur.

D'autre part, on peut souhaiter d'effectuer des *mises à jour sur le schéma*. Ceci peut entraîner l'invalidité de documents précédemment valides (malgré que ceci n'est pas toujours le cas comme montré dans [30]) et donc les documents invalides nécessitent une correction par rapport au nouveau schéma. Dans [109, 110], l'auteur démontre que la déduction de K transformations optimales de document XML à partir d'un script de mise à jour de DTD est NP-hard (même pour $K = 1$). Ils présentent aussi une solution pour le cas particulier de ce problème, c'est-à-dire lorsque le script de mise à jour de la DTD est de longueur égal à 1. Plus précisément, étant donné une DTD D, une opération élémentaire de mise à jour u transformant D en D' et un document t valide par rapport à D, l'algorithme proposé permet de calculer la liste des top-K transformations de t (déduites à partir de u) tel que chaque liste transforme t en un document valide par rapport à D'. L'algorithme a une complexité polynomiale en la taille de D, t et K. Le problème d'adapter un document précédemment valide après une modification du schéma est aussi adressé dans [65] : le schéma est un XML schema (XSD) et seulement une transformation heuristique choisie est calculée. Les auteurs ne donnent pas d'évaluation de la complexité mais des résultats expérimentaux démontrent un temps linéaire croissant en fonction de la taille du document.

Les instances du problème de correction de document vers schéma considérées dans cette sous-section sont particulières dans le sens où les documents sont initialement valides avant l'application de changements sur le document ou sur le schéma. Il est possible de prendre l'avantage de cette connaissance comme montré dans [28, 29, 94, 95, 109, 110]. Malgré ce fait, [109, 110] prouvent que la transformation de document en un ensemble d'arbres valides (aussi bien les K-optimum ou tous ceux en dessous d'un seuil th) n'est pas une tâche facile, même si le document est connu à l'avance comme étant valide. L'une des difficultés est de précisément établir les propriétés des solutions calculées par rapport à celles qui ne sont pas calculées. De même dans [28, 29], les auteurs peuvent éviter la revalidation et la correction des parties substantielles du document mais l'ensemble des solutions ne contient pas toutes les solutions en dessous du th. Ainsi il est impossible de savoir si la meilleure solution souhaitée par l'utilisateur est bien contenue dans cet ensemble de solutions.

5.7 Conclusion

La correction d'arbre vers un langage est un problème théorique qui a des applications intéressantes et existantes dans le domaine de traitement de XML, et certainement d'autres applications futures.

Étant donné un document XML bien formé vu comme un arbre t, une DTD vue comme une description de schéma S et un seuil positif th, nous avons présenté un algorithme pour calculer tous les arbres t' valides par rapport à S tels que la distance d'édition entre t et t' ne dépasse pas le seuil th.

La distance d'édition entre arbres, inspirée de [100], est basée sur trois opérations élémentaires d'édition de nœud : (i) le renommage de nœud, (ii) l'ajout d'une feuille, (iii) la suppression d'une feuille. Ces opérations peuvent être regroupées en des opérations plus complexes : (iv) insertion de sous-arbre, (v) suppression de sous-arbre. Le schéma est représenté comme un ensemble d'automates d'états fini associé à des étiquettes. L'algorithme étend les idées de [92] en faisant un parcours en profondeur d'abord borné par th de l'automate associé à l'étiquette du nœud courant, et une matrice de distance d'édition est complétée colonne par colonne chaque fois qu'une nouvelle transition est traversée. Cependant puisque nous corrigeons des arbres et non des mots, suivre chaque transition peut potentiellement provoquer une correction récursive d'un sous-arbre du nœud courant. De plus la matrice d'édition sauvegarde les séquences d'opérations d'édition pertinentes pour obtenir les arbres candidats pour la correction.

Nous avons prouvé que l'algorithme de correction est correct est complet. Nous avons aussi montré que sa complexité est $O((f_t + 1) \times (f_S)^{|t|+th} \times (6 \times |\Sigma| \times (|t| + th))^{th}$, où f_t est le nombre maximum de fils d'un nœud dans t, f_S est le nombre maximal de transitions sortantes de tous les automates de S, $|t|$ est le nombre de nœuds dans t, Σ est l'alphabet de S, et th est le seuil pour la correction. Cette complexité théorique exponentielle est liée au fait que les séquences d'édition et les corrections correspondantes sont générées et que l'ensemble correction est complet.

Nous avons réalisé des tests expérimentaux sur des données réelles axés sur l'influence de différents paramètres d'entrée (la taille de t, le seuil th, le nombre d'erreurs dans t, la position d'une erreur dans t) sur le temps de corrections et le nombre de solutions obtenues. En réalité, le temps CPU consommé dans les expériences montre un comportement polynomial plutôt qu'exponentiel.

A la lumière de l'analyse détaillée des travaux liés, la principale contribution de notre approche est d'offrir une première étude à part entière et approfondie du problème de correction d'arbre vers un langage d'arbres. Nous ne mesurons pas seulement la distance entre le document et le schéma mais aussi trouvons les arbres candidats à la correction. Nous ne nous limitons pas seulement à la recherche des solutions minimales mais aussi nous trouvons toutes les solutions à une distance inférieure à un seuil donné. Ainsi, nous considérons la correction comme un problème d'énumération plutôt qu'un problème de décision contrairement à la plupart des autres approches.

Certaines approches récentes telles que [105, 108, 113] apportent un nouveau regard à la correction de document vers un schéma puisqu'elles introduisent les opérations d'éditions agissant sur les nœuds internes de l'arbre, et offrent des optimisations de structure de données via une modélisation basée sur les graphes. L'une de nos perspectives est d'examiner comment ces propositions peuvent être intégrées aux nôtres pour pouvoir proposer un framework universel dans lequel différentes variantes du problème de correction peuvent être résolues plus efficacement.

Les travaux de ce chapitre ont pour objectif de fournir des outils pour effectuer l'évolution de schémas et assurer l'échange de données. L'évolution de schémas est exprimée par un script d'édition sur des grammaires d'arbres réguliers. Pour ce langage de mapping, deux opérateurs de mapping, la composition et l'inversion, sont définis et ont pour but d'adapter

un mapping $\mathcal{M}$ lorsque son schéma source ou son schéma cible évolue en un autre schéma. Ensuite nous avons proposé des méthodes pour :

(i) exprimer des mappings entre des systèmes locaux et leur évolution conservatrice en un système global, en nous basant sur la méthode d'extension minimale d'une grammaire RTG en LTG [42]. L'aspect conservateur garantit une flexibilité lorsque le schéma global coexiste avec les systèmes locaux ;

(ii) exprimer un mapping entre une grammaire RTG G et la grammaire $WI(G)$ de ses sous-arbres relâchés en nous basant sur l'algorithme présenté dans le Chapitre 3 pour calculer $WI(G)$ à partir de G.

Enfin nous avons présenté notre méthode pour adapter des documents XML suite à une évolution de leur schéma.

Un prototype a été implanté (en Java) et testé. Comme premier scénario de test, nous avons produit une LTG en fusionnant (calcul de l'extension minimale) les grammaires obtenues à partir de la DTD de *dblp*[7] et du XML Schema (XSD) de *HAL*[8]. Notre outil *MappingGen* retourne un mapping composé de 19 opérations d'édition. Ensuite l'outil *XTraM* a été utilisé pour adapter un document de 52 nœuds valide par rapport à la LTG calculé vers la grammaire pour *HAL*, et nous obtenons pour ce cas 36 solutions en 22.6 s. Dans ce test, toutes les traductions possibles ont été considérées mais l'utilisateur peut interférer sur les étapes intermédiaires en faisant des choix avant la fin du calcul des solutions. Dans ce sens, l'utilisateur guide et réduit le nombre de solutions. Cette fonctionnalité sera implémentée dans notre logiciel qui est décrit dans le Chapitre 7 - Section 7.2, pour faciliter l'interférence de l'utilisateur à des étapes intermédiaires lors du processus.

Pour de futurs travaux, il serait intéressant de pouvoir calculer automatiquement le mapping entre deux schémas donnés pour faciliter la tâche à l'utilisateur. Pour le moment le mapping peut être défini par l'utilisateur ou généré automatiquement pour une utilisation spécifique dans le cadre des deux applications (Section 6.3). Ainsi celui-ci pourra juste valider le mapping généré ou le corriger s'il ne répond pas à ces attentes. Aussi dans notre approche, l'adaptation de document tient seulement compte de sa structure. Comme dans les propositions [24, 60] (détaillées en Section 6.5), nous souhaiterions mettre à profit les dépendances fonctionnelles pour XML (Chapitre 4) en utilisant la technique de « chase » [85] pour compléter et générer les valeurs manquantes dans les solutions obtenues par *XTraM*.

Une autre perspective serait d'utiliser le mapping pour adapter les dépendances fonctionnelles définies sur le schéma source. De tels travaux peuvent rejoindre les travaux sur la réécriture des requêtes sur les documents en présence d'un mapping [23, 76] car les requêtes et les dépendances fonctionnelles manipulent des chemins dans les documents XML.

7. `http://dblp.uni-trier.de/xml/dblp.dtd`
8. `http://import.ccsd.cnrs.fr/xsd/generationAuto.php?instance=hal`

Mapping entre schémas XML

6

Sommaire

6.1 Langage pour le mapping de schémas 134
6.1.1 Représentation sous forme d'arbres des règles de production 134
6.1.2 Opérations d'édition de grammaire 136
6.1.3 Script d'édition de grammaire d'arbres et définition de mapping 140
6.2 Les opérateurs de mapping de schémas 142
6.3 Cas d'applications du mapping de schémas (*MappingGen*) 144
6.3.1 Évolution conservatrice de schémas 144
6.3.2 Substitution de schéma 148
6.4 Adaptation de documents XML guidée par le mapping (*XTraM*) 150
6.5 État de l'art 157
6.5.1 Mapping de schémas basé sur le formalisme logique 157
6.5.2 Mapping de schémas basé sur l'évolution incrémentale des schémas 158
6.6 Discussions et remarques 159

La construction de nouvelles applications visant à intégrer des données provenant de différentes sources tout en permettant l'utilisation des systèmes locaux d'origine n'est pas une tâche facile. Notre objectif est d'établir un environnement multi-système composé par un système global et central qui est une évolution *conservatrice* de systèmes locaux, capable d'effectuer des changements qui pourront être transmis aux systèmes locaux. Dans cet environnement, la communication doit être possible dans les deux sens : de local-à-global et de global-à-local. Nous aimerions permettre aux systèmes locaux de continuer à travailler sur leurs propres données tout en permettant un diagnostic et des modifications basées sur une vue générale et complète du système global. Ce scénario nécessite des outils pour traiter l'évolution de schéma et l'adaptation de leurs documents. Il peut être intéressant pour une entreprise d'avoir une configuration temporaire jusqu'à ce que les systèmes locaux soient prêts pour être complètement remplacés par le système global, ou simplement d'adopter cette architecture flexible.

Pour cela nous aimerions exprimer des mappings entre les systèmes locaux et le système global. Les mappings de schémas jouent un rôle important dans les tâches de gestion de données comme l'intégration de données ou l'échange de données, et permettent d'assurer l'interopérabilité entre les systèmes. L'échange de données se définit par la manière de transférer des données conformes à un schéma source vers des données conformes à un schéma cible. Le mapping de schémas exprime une relation entre le schéma source et le schéma cible.

Dans ce contexte nous rappelons le scénario du Chapitre 1. Supposons que $S_1, \ldots, S_n$ sont des systèmes locaux qui travaillent respectivement sur des collections de documents XML $X_1, \ldots, X_n$, et interagissent avec un système global S. Chaque ensemble X_i est conforme au schéma XML $\mathcal{D}_i$, tandis que $\mathcal{D}$ est un schéma XML pour S qui accepte n'importe quel document venant d'un $\mathcal{D}_i$. Nous supposons aussi que le système global S peut évoluer en S' pour accepter plus de documents ou en rejeter certains existants. Notre but est de proposer des outils permettant la transformation automatique de schéma et l'adaptation automatique des documents. Aussi les outils doivent permettre l'adaptation d'un mapping lorsque l'un des schémas concernés évolue : l'utilisation de la composition et de l'inversion de mapping sont nécessaires et apportent une adaptation facile et efficace.

Ce chapitre est organisé de la façon suivante : dans la Section 6.1, nous définissons le langage pour le mapping de schémas basé sur des opérations d'édition de grammaire RTG ; la composition et l'inversion de mapping sont définis dans la Section 6.2. La Section 6.3 contient des algorithmes pour calculer automatiquement des mappings dans le contexte d'évolution conservatrice de schémas d'une part et la substitution de schéma d'autre part. L'adaptation des documents XML suite à l'évolution de leur schéma est définie dans la Section 6.4. Une discussion sur les différents travaux portant sur l'évolution de schéma, est faite dans la Section 6.5 ; et enfin une conclusion et une discussion sur de futurs travaux clôturent ce chapitre (Section 6.6).

6.1 Langage pour le mapping de schémas

Dans le contexte de l'évolution de schéma, un schéma source va évoluer en un schéma cible. Un mapping de schémas est défini comme un triplet (S, T, σ) où S est le schéma source, T est le schéma cible et σ est le langage qui spécifie la relation entre le schéma source et le schéma cible. Généralement σ est un ensemble de dépendances (ou contraintes) qui sont exprimées par des formules logiques du type std ou tgd (« source-to-target dependency ») [16, 61, 4]. Pour décrire la relation entre le schéma source et le schéma cible, nous allons utiliser des opérations d'édition de schéma à la place de l'ensemble des dépendances « source-to-target » σ. Les opérations d'édition de schéma expriment les différents changements à appliquer au schéma source S pour le transformer en schéma cible T. Nos opérations d'édition de schéma sont une extension des opérations d'édition d'arbre vues dans le chapitre 5. L'ensemble σ sera donc remplacé par m qui est un script d'édition, c'est-à-dire une séquence d'opérations d'édition de schéma.

Dans cette section, nous allons définir nos opérations d'édition sur un schéma ou grammaire RTG G et ensuite définir formellement un mapping de schémas. L'idée est de représenter les règles de production de la grammaire sous forme d'arbres. Le problème de transformation d'une RTG en une autre RTG, est traité comme un problème d'édition d'arbres.

6.1.1 Représentation sous forme d'arbres des règles de production

Soit $X \to a\,[R]$ une règle de production. Nous notons $reg(X)$ (resp. $term(X)$) l'expression régulière R associée au non-terminal X (resp. le symbole terminal a associé à X). Notons que $reg(X)$ et $term(X)$ sont uniques puisque nous supposons que la grammaire est en forme normale et donc X n'apparaît qu'une seule fois à gauche d'une règle de production. Nous traitons l'expression régulière R comme un arbre d'arité variable dénoté t_R. L'ensemble

des non-terminaux qui apparaissent dans R est noté par $nt(R)$. Formellement, t_R est défini récursivement comme suit :

- si $R = \epsilon$ alors t_R est juste un seul nœud étiqueté par ϵ.
- si $R = A$ avec $A \in N$ alors t_R est juste un seul nœud étiqueté par A.
- si $R = R_1.\cdots.R_n$ alors $t_R = .(t_{R_1}, \cdots, t_{R_n})$ est l'arbre ayant pour racine '.' avec les sous-arbres fils $t_{R_1}, \cdots, t_{R_n}$.
- si $R = R_1|\cdots|R_n$ alors $t_R = |(t_{R_1}, \cdots, t_{R_n})$ est l'arbre ayant pour racine '|' avec les sous-arbres fils $t_{R_1}, \cdots, t_{R_n}$.
- si $R = R_1^*$ alors $t_R = *(t_{R_1})$ est l'arbre ayant pour racine '$*$' avec le seul sous-arbre fils t_{R_1}.
- si $R = (R_1)$ alors $t_R = t_{R_1}$.

Nous représentons la partie droite d'une règle de production $X \to a\,[R]$ comme un arbre dénoté t^r_X tel que $t^r_X = a(t_R)$. La racine de t^r_X est le symbole terminal a qui a pour seul fils le sous-arbre t_R. Nous avons $t^r_X|_0 = t_R$.

Exemple 6.1 Soit $R = (A.B.C)^*|(D^*.E)$ une expression régulière. La Figure 6.1 montre l'arbre t_R et l'arbre t^r_X représentant la partie droite de la règle de production $X \to root\,[R]$.

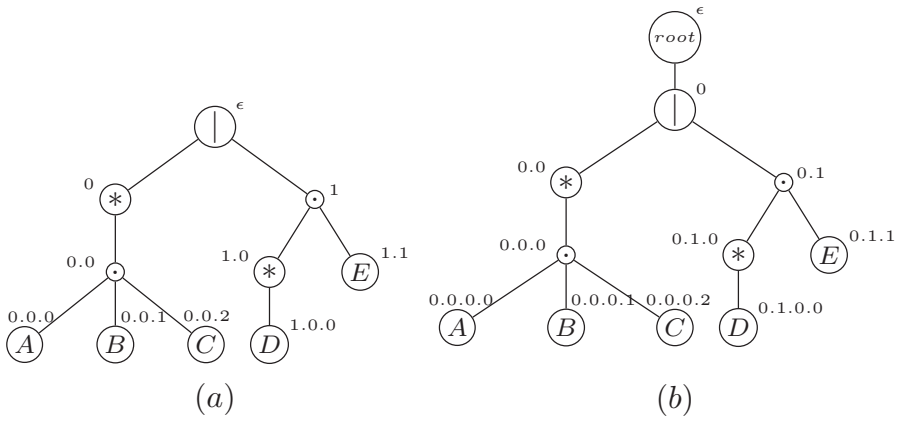


FIGURE 6.1 – *(a) Arbre t_R avec $R = (A.B.C)^*|(D^*.E)$ et (b) Arbre t^r_X avec $X \to root\,[R]$.*

⊠

Nous allons souvent confondre l'expression régulière R par sa représentation sous forme d'arbre t_R.

Définition 6.1 (Arbre de production bien formé)
Un arbre t représentant la partie droite d'une règle de production est bien formé si et seulement si les conditions suivantes sont vérifiées :

(i) la racine est un symbole terminal, c'est-à-dire $t(\epsilon) \in \Sigma$, et a exactement un seul sous-arbre fils ;

(ii) les nœuds feuilles sont dans $NT \cup \{\epsilon\}$;

(iii) les nœuds internes sont dans l'ensemble $\{|, ., *\}$ tel que :
- si un nœud interne est dans $\{*\}$ alors le nœud a exactement un seul fils ;
- sinon si un nœud interne est dans $\{|, .\}$ alors le nœud a au moins un fils.

□

Exemple 6.2 Dans la Figure 6.2, les arbres (a), (b) sont bien formés et les arbres (c), (d), (e) ne sont pas bien formés. L'arbre (c) n'est pas bien formé car le nœud '$*$' a plus d'un fils alors qu'il devrait avoir exactement un seul. L'arbre (d) n'est pas bien formé car la racine a trois fils alors qu'elle ne devrait avoir qu'un seul. Et enfin l'arbre (e) n'est pas bien formé car : (ii) le non-terminal Co a un fils alors qu'il devrait être une feuille, et (iii) le nœud '$*$' n'a pas de fils et est donc une feuille alors qu'il doit être un nœud interne et doit avoir exactement un fils. Si l'on traduit les trois arbres (c), (d), (e) en règles de production, on obtient respectivement $bib\,[Jour * Paper]$, $bib\,[Jour^*, conf, Rep]$ et $info\,[T.Y^{Co}.*]$ qui ne sont pas des expressions régulières bien formées. ⊠

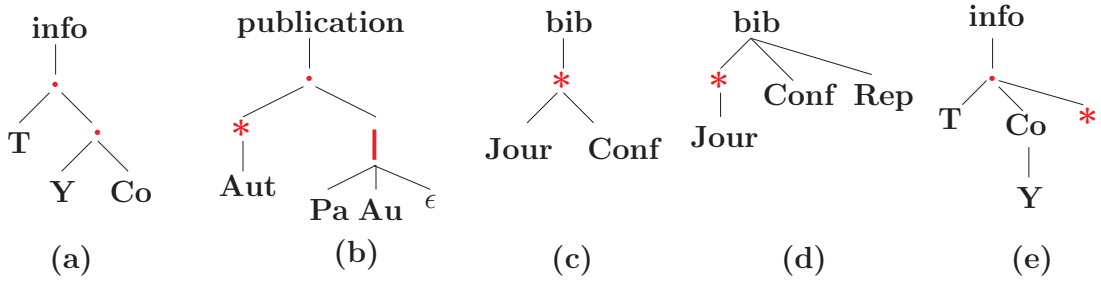
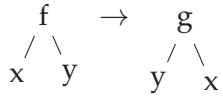
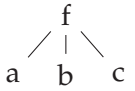
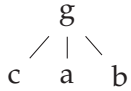


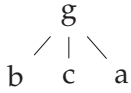
FIGURE 6.2 – *Arbres bien formés (a), (b) et mal formés (c), (d), (e).*

6.1.2 Opérations d'édition de grammaire

Nous allons définir les opérations d'édition en utilisant la réécriture. Étant donné un ensemble de variables V, une règle de réécriture (notée $l \to r$) est une paire de termes construits sur l'ensemble $\{|, ., *\} \cup N \cup V$, assumant que les variables n'ont pas de fils.

Une haie est une séquence d'arbres (possiblement vide) comme $[t_0, \ldots, t_n]$. Soit h une haie, $|h|$ dénote le nombre d'arbres dans h. Par exemple si $h = [t_0, \ldots, t_n]$ alors $|h| = n + 1$. Une substitution σ est une application de X vers un ensemble de haies, étendue de façon homomorphique aux arbres. Soient t et t' des arbres, t se réécrit en t' à la position u par la règle de réécriture $l \to r$ et avec la substitution σ (qui se note $t \to_{[u,l\to r,\sigma]} t'$), si on a $t|_u = \sigma(l)$ et $t' = t[u \leftarrow \sigma(r)]$. Par exemple, étant donné la règle  (x et y sont des variables), nous avons  qui se réécrit :

- en  (avec la substitution $x/[a, b]$, $y/[c]$), et aussi

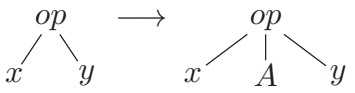
- en  (avec la substitution $x/[a]$, $y/[b, c]$).

Dans la définition suivante, les termes (ou arbres) sont toujours réécrits à la position u, avec la substitution σ, assurant que la condition (le cas échéant) est satisfaite. Nous allons seulement mentionner la règle de réécriture qui diffère selon les cas.

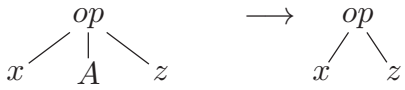
Définition 6.2 (Opérations d'édition élémentaires de grammaire)
Étant donné une RTG $G = (N, \Sigma, S, P)$ en forme normale, une **opération d'édition élémentaire** *ed* est une fonction partielle qui transforme G en une nouvelle RTG G'. L'**opération d'édition élémentaire** *ed* peut être appliquée à G si et seulement si *ed* est définie sur G. Nous distinguons quatre types d'opérations d'édition élémentaires sur une RTG :

1. Les opérations d'édition pour modifier l'ensemble des symboles initiaux S :
   - `set_startelm`(A) : ajoute le non-terminal A à S, où $A \in N$.
   - `unset_startelm`(A) : supprime le non-terminal A de S, où $A \in N$.

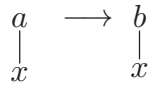
2. Les opérations d'édition pour modifier un non-terminal ou un terminal dans une règle de production :
   - `ins_elm`$(X, A, u.i)$: (cf. Figure 6.3(ed_1)) applique la règle de réécriture

     sur t^r_X à la position u, où $X \in N$, $A \in N \cup \{\epsilon\}$, $|\sigma(x)| = i$ et $op \in \{|, .\}$.
   - `del_elm`$(X, A, u.i)$: (cf. Figure 6.3(ed_2)) applique la règle de réécriture

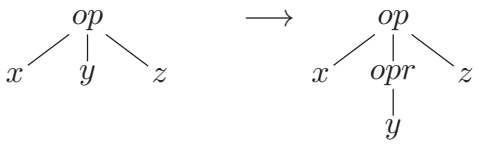
     

     sur t^r_X à la position u, où $X \in N$, $|\sigma(x)| = i$, $A \in N \cup \{\epsilon\}$, $|\sigma(x)| + |\sigma(z)| \geq 1$ et $op \in \{|, .\}$.
   - `rel_root`(X, a, b) : (cf. Figure 6.3(ed_3)) applique la règle de réécriture

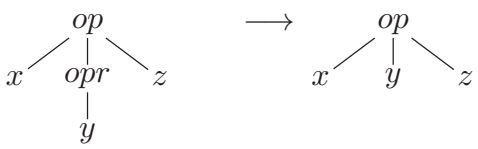
     

     sur t^r_X à la position ϵ, où $X \in N$, $a, b \in \Sigma$ et $|\sigma(x)| = 1$.
   - `rel_elm`(X, A, B, u) : (cf. Figure 6.3(ed_4)) applique la règle de réécriture $A \longrightarrow B$ sur t^r_X à la position u, où $X \in N$, $A, B \in N \cup \{\epsilon\}$.

3. Les opérations d'édition pour modifier un opérateur dans l'expression régulière d'une une règle de production :
   - `ins_opr`$(X, opr, u.i, n)$: (cf. Figure 6.3(ed_5)) applique la règle de réécriture

     sur t^r_X à la position u, où $X \in N$, $n \geq 1$, $op \in \{|, ., *\} \cup \Sigma$, $|\sigma(x)| = i$, $|\sigma(y)| = n$ et si $n = 1$ alors $opr \in \{|, ., *\}$ sinon $opr \in \{|, .\}$.
   - `del_opr`$(X, opr, u.i, n)$: (cf. Figure 6.3(ed_6)) applique la règle de réécriture

sur t^r_X à la position u, où $X \in N$, $op \in \{|, ., *\} \cup \Sigma$, $opr \in \{|, ., *\}$, $|\sigma(x)| = i$, et $|\sigma(y)| = n$. Si $op \in \{*\} \cup \Sigma$ alors $|\sigma(y)| = 1$ et $|\sigma(x)| + |\sigma(z)| = 0$.

– `rel_opr`(X, op, opr, u) : (cf. Figure 6.3(ed_7)) applique la règle de réécriture

$$\begin{array}{ccc} op & \longrightarrow & opr \\ | & & | \\ x & & x \end{array}$$

sur t^r_X à la position u, où $X \in N$, $op, opr \in \{|, ., *\}$, $|\sigma(x)| \neq 0$ et si $opr = *$ alors $|\sigma(x)| = 1$.

4. Les opérations d'édition pour modifier l'ensemble des règles de production P :
   – `ins_rule`(A, a) : ajoute la nouvelle règle de production $A \to a\,[\epsilon]$ à P et le non-terminal A à S, où $A \notin N$.
   – `del_rule`(A, a) : supprime la règle de production associée à A de P, où $A \in N$ et $reg(A) = \epsilon$. Aussi le non-terminal A ne doit apparaître nulle part dans les expressions régulières des autres règles de production. Si $A \in S$ alors A est aussi supprimé de S.

Après chaque opération d'édition, les ensembles Σ et N sont automatiquement mis à jour pour contenir uniquement les terminaux (resp. non-terminaux) apparaissant dans P.

□

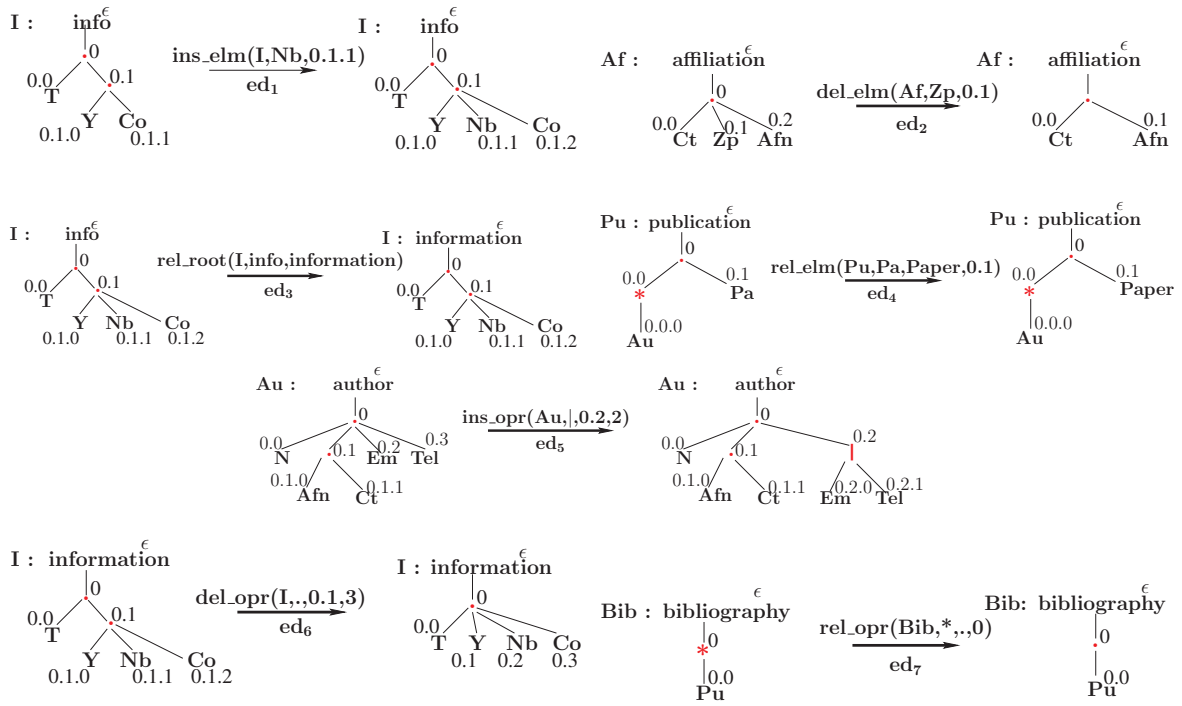


FIGURE 6.3 – *Exemple d'opérations d'édition élémentaires sur les grammaires RTG.*

A partir de la Définition 6.2, nous avons les propositions suivantes :

Proposition 6.1 *Une opération d'édition appliquée à une RTG G en forme normale et réduite produit une RTG G′ qui est aussi en forme normale et réduite.* □

Démonstration. L'application des opérations d'édition respecte certaines conditions. Avant d'ajouter une règle on s'assure que le non-terminal associé à la nouvelle règle n'existe pas

dans N. Aussi avant de supprimer une règle on s'assure que le non-terminal de la règle supprimée n'intervient pas dans les autres règles de production. Ces deux vérifications font que nous obtenons toujours une nouvelle grammaire en forme normale et réduite.

◁

Proposition 6.2 *Soient G et G' deux RTG données. Il existe toujours une séquence d'opérations d'édition élémentaires (Définition 6.2) capable de transformer G en G'.* □

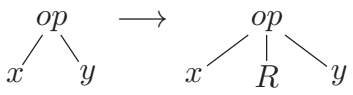
Démonstration. La séquence d'opérations la plus facile à trouver, consiste à supprimer toutes les règles de G, et à insérer toutes les règles de G'.

◁

En combinant les **opérations d'édition élémentaires**, nous allons définir les **opérations d'édition non-élémentaires** . Les opérations d'édition non-élémentaires sont considérées comme des raccourcis et sont vues comme un seul bloc d'opérations indissociables plus facile à manipuler et qui simplifieront la longueur des séquences d'opérations.

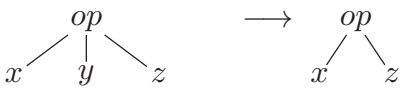
Définition 6.3 (Opérations d'édition non-élémentaires de grammaire)
Soient une RTG $G = (N, \Sigma, S, P)$ en forme normale, une **opération d'édition non-élémentaire** *ed* est une fonction partielle qui transforme G en une nouvelle RTG G'. L'**opération d'édition non-élémentaire** *ed* peut être appliquée à G si et seulement si *ed* est définie sur G. Chaque **opération d'édition non-élémentaire** peut être exprimée comme une séquence d'opérations d'édition élémentaires. Nous définissons les opérations d'édition non-élémentaires suivantes :

1. `ins_tree`$(X, R, u.i)$ (cf. Figure 6.4(ed_1)) : applique la règle de réécriture



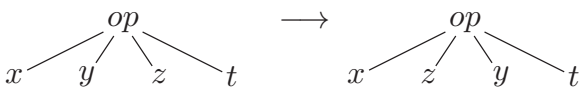
sur t^r_X à la position u, où $X \in N$, $|\sigma(R)| = 1$, $|\sigma(x)| = i$ et $op \in \{|, .\}$. L'opération `ins_tree`$(X, R, u.i)$ est équivalente à la séquence des opérations `ins_elm`$(X, t_R(v), u.i.v)$, $\forall\, v \in Pos(t_R)$. Les insertions des nœuds se faisant de la racine aux feuilles. Intuitivement l'opération `ins_tree` permet l'insertion d'un sous-arbre (expression régulière bien formée) dans t^r_X.

2. `del_tree`$(X, R, u.i)$ (cf. Figure 6.4(ed_2)) : applique la règle de réécriture



sur t^r_X à la position u, où $X \in N$, $|\sigma(x)| = i$, $|\sigma(y)| = 1$ et $\sigma(y) = t_R$, $|\sigma(x)| + |\sigma(z)| \geq 1$ et $op \in \{|, .\}$. L'opération `del_tree`$(X, R, u.i)$ est équivalente à la séquence des opérations `del_elm`$(X, t_R(v), u.i.v)$, $\forall\, v \in Pos(t^r_X|_{u.i})$. Les suppressions des nœuds se faisant des feuilles à la racine. Intuitivement l'opération `del_tree` permet la suppression d'un sous-arbre dans t^r_X.

3. `inv_tree`$(X, u.i)$ (cf. Figure 6.4(ed_3)) : applique la règle de réécriture



sur t^r_X à la position u, où $X \in N$, $op \in \{|,.\}$, $|\sigma(x)| = i$, $|\sigma(y)| = |\sigma(z)| = 1$. Cette opération $\texttt{inv_tree}(X, u.i)$ est équivalente à la séquence des opérations : $\texttt{ins_tree}(X, t^r_X|_{u.(i+1)}, u.i)$, $\texttt{del_tree}(X, t^r_X|_{u.(i+2)}, u.(i+2))$. Intuitivement l'opération `inv_tree` permet l'échange de deux sous-arbres à des positions consécutives.

4. $\texttt{ins_treerule}(A, a, R)$: ajoute la nouvelle règle de production $A \to a\,[R]$ à P et le non-terminal A à S, où $A \notin N$ et $nt(R) \subseteq N$. L'opération $\texttt{ins_treerule}(A, a, R)$ est équivalente à la séquence suivante : $\texttt{ins_rule}(A, a)$, $\texttt{ins_opr}(A, |, 0, 1)$, $\texttt{ins_tree}(A, t_R, 0.1)$, $\texttt{del_elm}(A, \epsilon, 0.0)$, $\texttt{del_opr}(A, |, 0, 1)$. Intuitivement l'opération `ins_treerule` permet l'ajout d'une règle de production dont l'expression régulière est différente de ϵ.

5. $\texttt{del_treerule}(A, a, R)$: supprime la règle de production $A \to a\,[R]$ de P et le non-terminal A de S (si $A \in S$), où $A \in N$. L'opération $\texttt{del_treerule}(A, a, R)$ est équivalente à la séquence suivante : $\texttt{ins_opr}(A, |, 0, 1)$, $\texttt{ins_elm}(A, \epsilon, 0.0)$, $\texttt{del_tree}(A, t_R, 0.1)$, $\texttt{del_opr}(A, |, 0, 1)$, $\texttt{del_rule}(A, a)$. Intuitivement l'opération `del_treerule` permet la suppression d'une règle de production dont l'expression régulière est différente de ϵ.

6. $\texttt{ins_elm_treerule}(X, A, u.i, a, R)$ est équivalente à la séquence suivante : $\texttt{ins_treerule}(A, a, R)$, $\texttt{ins_elm}(X, A, u.i)$, $\texttt{unset_startelm}(A)$. Intuitivement l'opération `ins_elm_treerule` permet l'ajout d'un non-terminal $A \notin N$ dans une expression régulière, en prenant le soin d'ajouter la règle de production correspondante à A.

7. $\texttt{del_elm_treerule}(X, A, u.i, a, R)$ est équivalente à la séquence suivante : $\texttt{set_startelm}(A)$, $\texttt{del_elm}(X, A, u.i)$, $\texttt{del_treerule}(A, a, R)$. Intuitivement l'opération `del_elm_treerule` permet la suppression d'un non-terminal A d'une expression régulière, en prenant le soin de supprimer la règle de production correspondante à A.

8. $\texttt{agg_elm}(X, A, u)$ (cf. Figure 6.4(ed_4)) : applique la règle de réécriture $x \longrightarrow A$ sur t^r_X à la position u, où $X \in N$, $u \neq \epsilon$, $A \in N$, $|\sigma(x)| = 1$, et $reg(A) = \sigma(x)$. L'opération $\texttt{agg_elm}(X, A, u)$ est équivalente à la séquence suivante : $\texttt{ins_opr}(X, |, u, 1)$, $\texttt{ins_elm}(X, A, u.1)$, $\texttt{del_tree}(X, t^r_X|_{u.0}, u.0)$, $\texttt{del_opr}(X, |, u, 1)$. Intuitivement l'opération `agg_elm` permet de remplacer une sous-expression de t^r_X par un non-terminal A. Ceci est possible seulement si la sous-expression est équivalente à $reg(A)$.

9. $\texttt{ext_elm}(X, A, u)$ (cf. Figure 6.4(ed_5)) : applique la règle de réécriture $A \longrightarrow t_{reg(A)}$ sur t^r_X à la position u, où $X, A \in N$. L'opération $\texttt{ext_elm}(X, u)$ est équivalente à la séquence suivante : $\texttt{ins_opr}(X, |, u, 1)$, $\texttt{ins_tree}(X, t^r_A|_0, u.1)$, $\texttt{del_elm}(X, A, u.0)$, $\texttt{del_opr}(X, |, u, 1)$. Intuitivement l'opération `ext_elm` permet de remplacer un non-terminal A dans t^r_X par son expression régulière associée $reg(A)$.

□

6.1.3 Script d'édition de grammaire d'arbres et définition de mapping

Soit ed une opération d'édition (élémentaire ou non) définie sur la grammaire RTG G. Nous notons par $ed(G)$ la grammaire obtenue en appliquant l'opération ed à G.

Définition 6.4 (Script d'édition)
Un **script d'édition** $m = \langle ed_1, ed_2, \ldots ed_n \rangle$ est une séquence d'*opérations d'édition* ed_k où $1 \leq k \leq n$. Soit G une RTG, un **script d'édition** $m = \langle ed_1, ed_2, \ldots ed_n \rangle$ est définie sur G si et seulement si il existe une séquence de RTG $G_0, G_1, \ldots, G_n$ telle que :

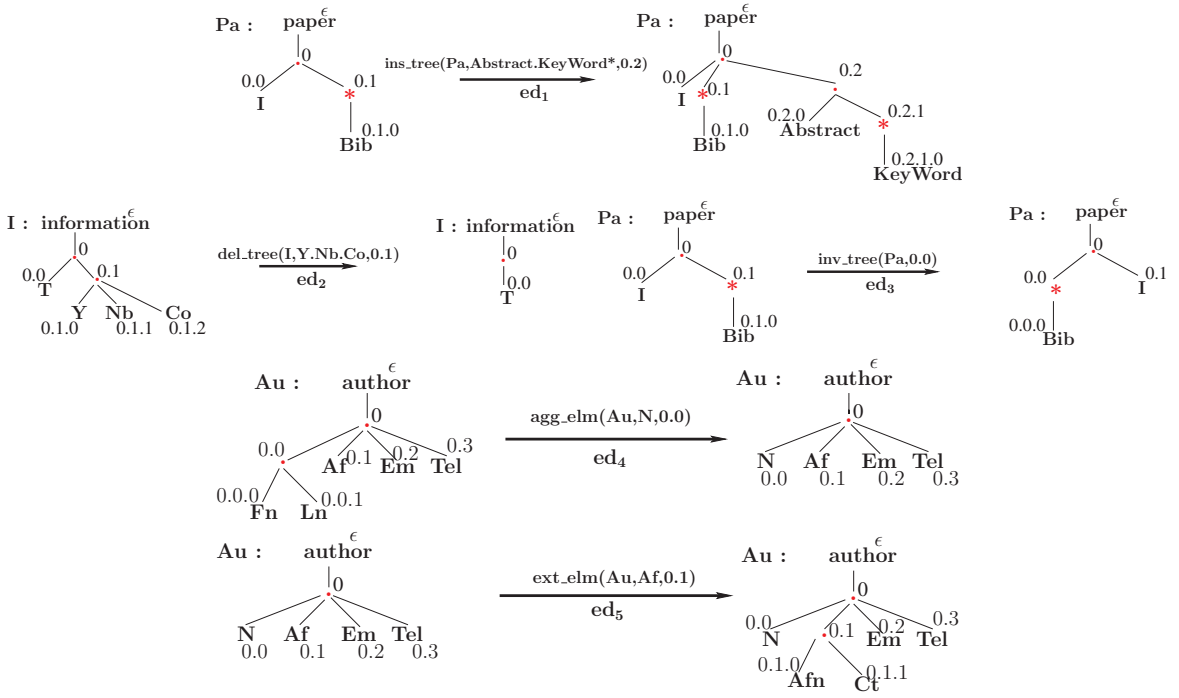


FIGURE 6.4 – *Exemple d'opérations d'édition non-élémentaires sur les grammaires RTG.*

- $G_0 = G$
- $\forall 1 \leq k \leq n$, ed_k est définie sur G_{k-1} et $ed_k(G_{k-1}) = G_k$.

Ainsi nous avons $m(G) = G_n$ c'est-à-dire qu'en appliquant le *script d'édition* m sur la RTG G nous obtenons la RTG G_n. Le *script d'édition* m nous dit comment obtenir la RTG G_n à partir de G. Le *script d'édition* vide est noté $\langle\rangle$.

□

Maintenant pour chaque opération d'édition ed, nous définissons un coût positif qui dépend d'une application spécifique donnée. D'une part nous supposons que les opérations dont l'application sur une RTG G ne changent pas le langage généré par G, sont de coût nul. Ces opérations sont définies dans la Définition 6.5. Leur but est de simplifier une expression régulière donnée de la grammaire. D'autre part nous supposons que les opérations d'édition élémentaires (Définition 6.2) ont un coût de 1, tandis que les opérations d'édition non-élémentaires (Définition 6.3) ont un coût de 5. Bien sûr que les coûts des opérations d'édition peuvent être adpatés en fonction du besoin de l'application. La fonction de coût sur l'opération ed est notée $cost(ed)$.

Définition 6.5 (Les opérations d'édition de coût nul)
Les **opérations d'édition de coût nul** ne changent pas le langage généré par la grammaire G sur laquelle elles sont définies. En fait nous avons $L(G) = L(ed(G))$ où ed est une **opération d'édition de coût nul**. Les opérations d'édition suivantes sont de coût nul :

- $\texttt{inv_tree}(X, u.i)$ où $t^r_X(u) = '|'$;
- $\texttt{del_elm}(X, \epsilon, u.i)$, ou $\texttt{ins_elm}(X, \epsilon, u.i)$ avec $t^r_X(u) = '.'$;
- $\texttt{del_opr}(X, opr, u.i)$ où $t^r_X(u) = t^r_X(u.i) = opr$;
- $\texttt{del_opr}(X, opr, u.i)$ où $t^r_X(u.i) \in \{|, .\}$ et $t^r_X(u.i)$ a exactement qu'un seul fils.

□

Définition 6.6 (Coût d'un script d'édition)
Soit le script d'édition $m = \langle ed_1, ed_2, \ldots, ed_n \rangle$. Le coût de m est définie par $Cost(m) = \Sigma_{i=1}^{n}(cost(ed_i))$.

□

Définition 6.7 (Mapping de schémas)
Un mapping de schéma XML est un triplet $\mathcal{M} = (S, T, m)$, où S est le schéma source, T est le schéma cible, et m est un *script d'édition* qui transforme S en T (c'est-à-dire $m(S) = T$). Nous disons que $\mathcal{M}$ est syntaxiquement *spécifié* ou *exprimé* par m.

□

6.2 Les opérateurs de mapping de schémas

Dans [62], la composition et l'inversion ont été présentés comme deux opérateurs fondamentaux pour l'adaptation de mapping dans le contexte de l'évolution de schéma. Soit $\mathcal{M}_1 = (S, T, m_1)$ un mapping entre les schémas S et T. Lorsque S ou T évolue, le mapping $\mathcal{M}_1$ doit être adapté. En utilisant les opérateurs composition et inversion, nous pouvons éviter le recalcul de mappings. L'Exemple 6.3 (inspiré de [62]), décrit comment sont utilisés les opérateurs composition et inversion pour adapter un mapping lors de l'évolution de schéma.

Exemple 6.3 Nous allons considérer les schémas S et T de la Figure 6.5 et le mapping $\mathcal{M}_1$ entre ses deux schémas. Nous pouvons considérer les deux situations suivantes :

1. Évolution du schéma cible : assumons que le schéma T évolue en T' et que cette évolution soit modélisée par le mapping $\mathcal{M}_2$. Intuitivement $\mathcal{M}_2$ va permettre la transformation des données de T en des données de T'. En composant $\mathcal{M}_1$ et $\mathcal{M}_2$, nous obtenons un nouveau mapping qui va de S à T'. La composition opère en général sur deux mappings consécutifs $\mathcal{M}_1$ et $\mathcal{M}_2$, tels que le schéma cible de $\mathcal{M}_1$ est le schéma source de $\mathcal{M}_2$. Le résultat de la composition donne le mapping $\mathcal{M}_1 \circ \mathcal{M}_2$ qui a le même effet que d'appliquer $\mathcal{M}_1$ et ensuite $\mathcal{M}_2$.
2. Évolution du schéma source : assumons que le schéma source évolue en S' et que cette évolution soit modélisée par le mapping $\mathcal{M}_3$. Intuitivement $\mathcal{M}_3$ va permettre la transformation des données de S en des données de S'. Nous avons besoin d'un nouveau mapping qui reflète le mapping original $\mathcal{M}_1$ (ou plutôt $\mathcal{M}_1 \circ \mathcal{M}_2$ après l'évolution du schéma cible) mais en partant du schéma source S'. Notons que dans ce cas nous ne pouvons pas directement composer $\mathcal{M}_3$ avec $\mathcal{M}_1 \circ \mathcal{M}_2$, puisque $\mathcal{M}_3$ et $\mathcal{M}_1 \circ \mathcal{M}_2$ ne sont pas consécutifs. Pour pouvoir appliquer la composition, nous avons besoin d'appliquer l'opérateur inversion sur $\mathcal{M}_3$ pour obtenir le mapping $\mathcal{M}_3^{-1}$ qui inverse l'effect de $\mathcal{M}_3$. A partir de $\mathcal{M}_3^{-1}$, nous pouvons appliquer l'opération composition pour produire $\mathcal{M}_3^{-1} \circ \mathcal{M}_1 \circ \mathcal{M}_2$. Le mapping résultat, qui va de S' vers T', est une adaptation du mapping original m pour les nouveaux schémas obtenus après évolution.

⊠

Nous allons maintenant définir les opérateurs composition et inversion sur nos mappings entre les grammaires RTG.

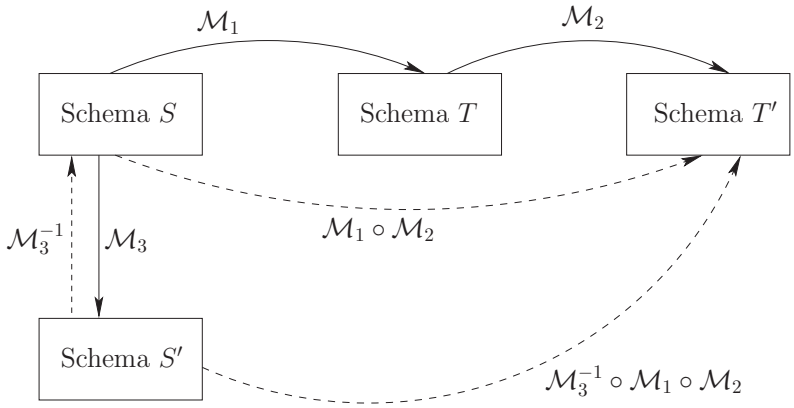


FIGURE 6.5 – *Utilisation de la composition et de l'inversion pour faciliter l'évolution de schéma.*

Définition 6.8 (Composition de mappings)
Soient deux mappings $\mathcal{M}_1$ et $\mathcal{M}_2$ tels que $\mathcal{M}_1 = (S, T, m_1)$ et $\mathcal{M}_2 = (T, V, m_2)$; la composition de $\mathcal{M}_1$ et $\mathcal{M}_2$ donne le mapping $\mathcal{M}_1 \circ \mathcal{M}_2 = (S, V, m_1 \,.\, m_2)$ où $m_1 \,.\, m_2$ est la concaténation de m_1 à m_2.

□

Définition 6.9 (Inversion de mapping)
Soit le mapping $\mathcal{M} = (S, T, m)$ où $m = \langle ed_1, \cdots, ed_n \rangle$, l'inverse de $\mathcal{M}$ est le mapping $\mathcal{M}^{-1} = (T, S, m^{-1})$ avec $m^{-1} = \langle ed_n^{-1}, \cdots, ed_1^{-1} \rangle$ et $ed_k^{-1} (1 \leq k \leq n)$ est l'inverse de l'opération d'édition ed_k qui est défini dans le Tableau 6.1.

	opération d'édition ed		**opération d'édition** ed^{-1}
Élémentaire	$\texttt{set_startelm}(A)$	$\rightleftharpoons$	$\texttt{unset_startelm}(A)$
	$\texttt{ins_elm}(X, A, u.i)$	$\rightleftharpoons$	$\texttt{del_elm}(X, A, u.i)$
	$\texttt{rel_root}(X, a, b)$	$\rightleftharpoons$	$\texttt{rel_root}(X, b, a)$
	$\texttt{rel_elm}(X, A, B, u)$	$\rightleftharpoons$	$\texttt{rel_elm}(X, B, A, u)$
	$\texttt{ins_opr}(X, opr, u.i, n)$	$\rightleftharpoons$	$\texttt{del_opr}(X, opr, u.i, n)$
	$\texttt{rel_root}(X, opr_1, opr_2, u)$	$\rightleftharpoons$	$\texttt{rel_root}(X, opr_2, opr_1, u)$
	$\texttt{ins_rule}(A, a)$	$\rightleftharpoons$	$\texttt{del_rule}(A, a)$
Non-élémentaire	$\texttt{ins_tree}(X, R, u.i)$	$\rightleftharpoons$	$\texttt{del_tree}(X, R, u.i)$
	$\texttt{inv_tree}(X, u.i)$	$\rightleftharpoons$	$\texttt{inv_tree}(X, u.i)$
	$\texttt{ins_treerule}(A, a, R)$	$\rightleftharpoons$	$\texttt{del_treerule}(A, a, R)$
	$\texttt{ins_elm_treerule}(X, A, u.i, a, R)$	$\rightleftharpoons$	$\texttt{del_elm_treerule}(X, A, u.i, a, R)$
	$\texttt{agg_elm}(X, A, u)$	$\rightleftharpoons$	$\texttt{ext_elm}(X, A, u)$

Tableau 6.1 – *Tableau de correspondance des opérations inverses.*

□

Exemple 6.4 Soient les scripts d'édition
$m_1 = \langle \texttt{ins_elm}(I, Nb, 0.1.1), \texttt{rel_root}(I, info, information) \rangle$ et
$m_2 = \langle \texttt{ins_opr}(Au, |, 0.2, 2), \texttt{ins_rule}(Nb, nbPages) \rangle$. La concaténation de m_1 et m_2 donne
$m = \langle \texttt{ins_elm}(I, Nb, 0.1.1), \texttt{rel_root}(I, info, information), \texttt{ins_opr}(Au, |, 0.2, 2),$

$\texttt{ins_rule}(Nb, nbPages)\rangle$. L'inverse de m donne $m^{-1} = \langle \texttt{del_rule}(Nb, nbPages)$, $\texttt{del_opr}(Au, |, 0.2, 2), \texttt{rel_root}(I, information, info), \texttt{del_elm}(I, Nb, 0.1.1)\rangle$.

⊠

6.3 Cas d'applications du mapping de schémas (*MappingGen*)

Dans cette section nous allons aborder deux cas d'application du mapping de schémas. Le premier cas d'application concerne une évolution conservatrice de schémas locaux $D_1, \ldots, D_n$ en un schéma global D. L'algorithme qui permet de calculer le nouveau schéma D qui génère le plus petit langage LTL (Local Tree Language) contenant les langages générés par $D_1, \ldots, D_n$ a été formalisé dans [42]. A partir de l'algorithme proposé dans [42], nous souhaitons calculer les mappings entre chacun des schémas locaux D_i ($1 \leq i \leq n$) et le schéma global D pour ensuite pouvoir traduire les documents XML entre les schémas D_i et D lors de leurs échanges. La traduction guidée par le mapping des documents XML sera expliquée dans la section 6.4. Nous pouvons remarquer que le mapping m_i obtenu en nous basant sur l'algorithme d'extension de schéma dans [42] permet de transformer le schéma D_i en D. Pour traduire les documents de D vers ceux de D_i nous avons besoin du mapping inverse m_i^{-1} qui permet de transformer le schéma D en D_i. La méthode pour obtenir le mapping inverse m_i^{-1} à partir de m_i se trouve dans la Définition 6.9. Pour résumer, nous savons passer du schéma D_i à D en se servant du mapping m_i ; et à partir de l'inverse m_i^{-1} du mapping m_i, nous pouvons passer du schéma D à D_i. Enfin pour assurer les échanges entre D_i et D, nous utilisons le mapping m et son inverse pour traduire (l'algorithme de traduction de documents sera présenté dans la section 6.4) les documents XML d'un schéma vers un autre. Dans la section 6.3.1, nous présentons l'algorithme qui permet de calculer le mapping m_i en suivant les principes de l'algorithme d'extension de schéma dans [42].

Le deuxième cas d'application concerne la substitution d'un service web par un autre service web. Nous avons vu dans le chapitre 3, l'utilisation de l'inclusion relâchée pour comparer deux schémas XML exprimés comme des RTG. Comme expliqué dans la section 3.5, lorsque le service A communique avec le service C, et qu'on remplace le service A par B (en ayant pris le soin de vérifier à l'aide de l'inclusion relâchée que B fournit les mêmes informations que A), il est nécessaire de pouvoir traduire les documents entre C et B pour permettre les échanges. Nous allons supposer que le mapping entre C et A est connu. Dans la section 6.3.2, nous allons proposer une méthode pour calculer le mapping entre A et B en suivant le principe de l'algorithme de calcul de la grammaire des sous-arbres relâchés de B. Une fois que nous avons le mapping entre A et B, à l'aide de la composition de mapping (Définition 6.8), nous pouvons calculer le mapping entre C et B. Ensuite comme dans le cas précédent, avec l'algorithme de traduction de documents (présenté en section 6.4) et du mapping entre C et B, nous pouvons traduire les documents XML entre B et C.

Les deux algorithmes pour générer des mappings dans le contexte des deux applications décrites ci-dessous seront dénommés par l'outil *MappingGen*.

6.3.1 Évolution conservatrice de schémas

Nous allons d'abord donner un exemple pour motiver l'application, et ensuite proposer un algorithme pour calculer le mapping souhaité.

Exemple. Nous considérons les données d'un hôpital maintenues par trois services : un service qui gère les informations des patients et leurs traitements, un autre service qui est responsable de l'édition des factures et le dernier service qui est en contact avec les sociétés d'assurances et décide quand un traitement est couvert par l'assurance du patient. La Figure 6.6 montre une version simplifiée des DTDs de chaque service. Les définitions d'éléments de type PCDATA sont omises.

```
Service des patients et traitements
<!ELEMENT hospital  (info*)>
<!ELEMENT info      (patient|treatment)>
<!ELEMENT patient   (SSN,pname,visitInfo*)>
<!ELEMENT visitInfo (trId,date)>
<!ELEMENT treatment (trId,tname,procedure)>
<!ELEMENT procedure (treatment*)>
```

```
Service de la couverture d'assurance
<!ELEMENT hospital (info*)>
<!ELEMENT info     (cover|policy)>
<!ELEMENT cover    (SSN,plname)>
<!ELEMENT policy   (plname,trId*)>
```

```
Service des factures
<!ELEMENT hospital (info*)>
<!ELEMENT info     (bill)>
<!ELEMENT bill     (SSN,item*,date)>
<!ELEMENT item     (trId,price)>
```

FIGURE 6.6 – *Les DTDs des services d'un hôpital*

Sans interférer sur les services locaux, en leur demandant de garder leur propre système local, la direction de l'hôpital aimerait avoir une vue globale sur l'ensemble des services dans le but de faire des rapports et statistiques comme par exemple le pourcentage de compagnies d'assurance qui couvrent la radiothérapie ou la chirurgie de la myopie ; le nombre de patients qui payent eux mêmes leur propres traitements, etc. Le système global peut recevoir directement des informations : un docteur ayant l'accès au schéma global peut introduire des informations sur un nouveau traitement (par exemple le prix qu'il a fixé pour son traitement) avec les patients qu'il souhaite traiter. De plus, en analysant les données globales, la direction peut décider de changer sa politique. Par exemple, la direction peut décider d'appliquer un bas prix pour les patients qui n'ont pas une bonne assurance, et ceci peut provoquer la modification (ou l'évolution) de schéma.

Cette flexibilité peut être enrichie en permettant quelques actions basiques. Premièrement, nous avons la construction d'un schéma global conservateur capable d'accepter d'une part tous les documents locaux, et d'autre part les nouveaux documents soumis directement au schéma global. Notons que les données locales n'ont pas besoin d'être traduites, puisqu'elles sont déjà valides par rapport au schéma global. Deuxièmement, la traduction des documents à partir du système global vers un système local, peut autoriser des mises à jour au niveau global qui seront passées au niveau local. Finalement, avec l'évolution du schéma global en gardant un mapping disponible, nous pouvons transformer les schémas locaux sources vers le nouveau schéma global. Des actions similaires sont autorisées pour traiter le cas où un schéma local évolue.

La Figure 6.7 montre la DTD globale résultante de la méthode qui est proposée en [42]. Nous allons proposer dans la partie suivante un algorithme qui construit les mappings entre les DTDs locaux et la DTD globale (construit à partir des idées de [42]). Connaissant le mapping qui transforme un schéma local en schéma global, il est trivial d'obtenir son inverse. Le mapping inverse sera la base pour traduire les documents à partir du schéma global vers un schéma local.

```
<!ELEMENT hospital  (info*)>
<!ELEMENT info      ((patient|treatment)|(cover|policy)|bill)>
<!ELEMENT patient   (SSN,pname,visitInfo*)>
<!ELEMENT visitInfo (trId,date)>
<!ELEMENT treatment (trId,tname,procedure)>
<!ELEMENT procedure (treatment*)>
<!ELEMENT cover    (SSN,plname)>
<!ELEMENT policy   (plname,trId*)>
<!ELEMENT bill     (SSN,item*,date)>
<!ELEMENT item     (trId,price)>
```

FIGURE 6.7 – *DTD global pour l'hospital*

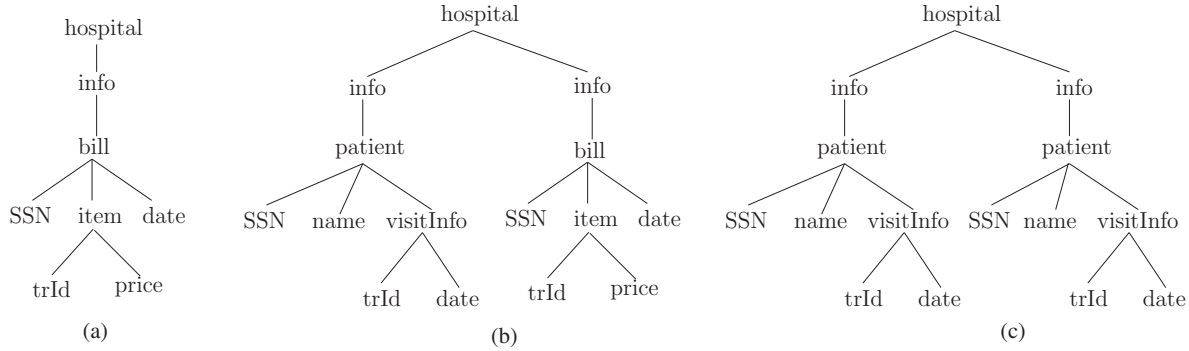


FIGURE 6.8 – *(a) Un arbre XML valide par rapport au schéma local des factures. (b) Un arbre XML valide par rapport au schéma global de la Figure 6.7. (c) Arbre résultant après la transformation de (b) vers le schéma local des patients de la Figure 6.6.*

Dans la Figure 6.8(a) nous trouvons un document valide par rapport au schéma local sur les factures. Notons qu'il est aussi valide par rapport au schéma global de la Figure 6.7. Le document de la Figure 6.8(b) montre un document XML concernant des patients et des traitements. Ce document est valide par rapport au schéma global mais n'est pas valide par rapport à aucun des schémas locaux. La transformation du document de la Figure 6.8(b) vers un document respectant le schéma sur les patients, donne le document de la Figure 6.8(c). Cette traduction est guidée par le mapping de schéma du schéma global de la Figure 6.7 vers le schéma local sur les patients de la Figure 6.6.

L'algorithme dans [42] (nommé *ExtSchemaGenerator*) permet de calculer une grammaire LTG ou STTG qui étend minimalement une grammaire RTG donnée. Dans ce papier sont prouvées la correction et la minimalité de la grammaire générée. Nous allons suivre les idées de *ExtSchemaGenerator* pour générer une LTG à partir d'une RTG. Nous fusionons les règles de production associées aux non-terminaux en concurrence. L'expression régulière de la nouvelle règle est une disjonction des expressions régulières associées aux non-terminaux en concurrence.

La Figure 6.9 montre la grammaire RTG obtenue par l'union des règles de production des trois grammaires locales tandis que la Figure 6.10 montre la LTG résultante de l'extension de schéma. La LTG obtenue est une extension de la RTG originale (Figure 6.9) car elle génère *tous* les arbres qui sont générés par la RTG originale et éventuellement d'autres arbres comme celui de la Figure 6.8(b). Il est aussi vrai que la LTG obtenue est aussi une extension de chaque grammaire des services de l'hôpital.

$H_1 \rightarrow hospital[I_1^*]$
$I_1 \rightarrow info[P \mid T]$
$P \rightarrow patient[S \cdot N \cdot V^*]$
$V \rightarrow visitInfo[Id \cdot D]$
$T \rightarrow treatment[Id \cdot TN \cdot PR]$
$PR \rightarrow procedure[T^*]$

$H_2 \rightarrow hospital[I_2^*]$
$I_2 \rightarrow info[C \mid Pol]$
$C \rightarrow cover[S \cdot PN]$
$Pol \rightarrow policy[PN \cdot Id^*]$

$H_3 \rightarrow hospital[I_3^*]$
$I_3 \rightarrow info[B]$
$B \rightarrow bill[S \cdot It^* \cdot D]$
$It \rightarrow item[Id \cdot PZ]$

FIGURE 6.9 – *RTG obtenue après la fusion de toutes les règles de production des grammaires locales.*

$H_1 \rightarrow hospital[I_1^* \mid I_1^* \mid I_1^*]$
$I_1 \rightarrow info[(P \mid T) \mid (C \mid Pol) \mid B]$
$P \rightarrow patient[S \cdot N \cdot V^*]$
$V \rightarrow visitInfo[Id \cdot D]$
$T \rightarrow treatment[Id \ \cdot \ TN \ \cdot \ PR]$
$PR \rightarrow procedure[T^*]$
$Pol \rightarrow policy[PN \cdot Id^*]$
$B \rightarrow bill[S \cdot It^* \cdot D]$
$It \rightarrow item[Id \cdot PZ]$

FIGURE 6.10 – *LTG obtenue par l'algorithme dans [42] à partir de la RTG de la Figure 6.9.*

Algorithme. L'Algorithme 11 génère un mapping qui convertit une RTG en une LTG en suivant les idées de [42], et cet algorithme a été illustré dans l'exemple ci-dessus. L'Algorithme 11 commence par déterminer l'ensemble des non-terminaux en concurrence EC_a (lignes 2-3). Ensuite nous choisissons arbitrairement dans EC_a, un des non-terminaux (nommé X_0) pour représenter les autres c'est-à-dire lorsque nous fusionnons les règles des non-terminaux en concurrence, l'un d'entre eux est choisi pour représenter la règle résultante de la fusion (ligne 4). Rappelons que les opérations d'édition portent toujours sur une règle de production vue comme un arbre. La nouvelle règle de production de X_0 est construite en deux étapes. Nous ajoutons l'opérateur '|' comme parent à l'expression régulière originale $reg(X_0)$ (ligne 5) et ensuite nous insérons toutes les expressions régulières associées aux non-terminaux en concurrence avec X_0 comme frères du sous-arbre $reg(X_0)$ (ligne 7). A la ligne 8 nous remplaçons dans toutes les règles de production, les non-terminaux dans EC_a par X_0. Les règles de production originales des non-terminaux dans EC_a sont supprimées (ligne 11) après un ajustement possible de l'ensemble des symboles initiaux S (ligne 9).
Si nous considérons la RTG de la Figure 6.9, L'Algorithme 11 retourne le mapping m suivant :

1	`ins_opr`$(H_1, \mid, 0, 1)$	7	`ins_tree`$(I_1, C \mid Pol, 0.1)$
2	`ins_tree`$(H_1, I_2^*, 0.1)$	8	`rel_elm`$(H_1, I_2, I_1, 0.1.0)$
3	`del_treerule`$(H_2, hospital, I_2^*)$	9	`del_treerule`$(I_2, info, C \mid Pol)$
4	`ins_tree`$(H_1, I_3^*, 0.2)$	10	`ins_tree`$(I_1, B, 0.2)$
5	`del_treerule`$(H_3, hospital, I_3^*)$	11	`rel_elm`$(H_1, I_3, I_1, 0.2.0)$
6	`ins_opr`$(I_1, \mid, 0, 1)$	12	`del_treerule`$(I_3, info, B)$

Lorsque $m = \langle 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 \rangle$ est appliqué à la RTG de la Figure 6.9, nous obtenons la LTG de la Figure 6.10.

Théorème 6.1 *Soit m le mapping obtenu via l'Algorithme 11 à partir de la RTG G. Le langage $L(m(G))$ est le plus petit LTL qui contient $L(G)$. De plus, la grammaire $m(G)$ est égale à celle obtenue par ExtSchemaGenerator.*

Algorithme 11 Génération d'un mapping pour transformer une RTG en LTG

Entrée : Une RTG $G = (NT, \Sigma, S, P)$
Sortie : Un script d'édition m entre G et la LTG G' tel que $L(G) \subseteq L(G')$

1: $m := \langle\rangle$
2: **pour tout** terminal $a \in \Sigma$ **faire**
3: $EC_a := \{X_0, \ldots, X_k\}$ contient les non-terminaux en concurrence où $term(X_i) = a$
4: Le non-terminal X_0 est choisi pour représenter $X_0, \ldots, X_k$
5: Ajout de $\texttt{ins_opr}(X_0, |, 0, 1)$ à m **% *début de la construction de la nouvelle règle* %**
6: **pour tout** non-terminal $X_i \in \{X_1, \ldots, X_k\}$ **faire**
7: Ajout de $\texttt{ins_tree}(X_0, reg(X_i), 0.i)$ à m
8: Ajout de $\texttt{rel_elm}(Y, X_i, X_0, u)$ à m, pour tout u tel que u est la position de X_i dans la règle $Y \rightarrow b\,[R] \in P$
9: Ajout de $\texttt{set_startelm}(X_0)$ à m lorsque $X_0 \notin S$ et $X_i \in S$
10: **% *La règle de production du non-terminal* X_i *est supprimée de* P %**
11: Ajout de $\texttt{del_treerule}(X_i, a, reg(X_i))$ à m
12: **fin pour**
13: **fin pour**
14: **retourner** m

Démonstration. En suivant les lignes de l'Algorithme 11 qui correspondent aux mêmes changements effectués par l'algorithme *ExtSchemaGenerator*, nous pouvons conclure que la grammaire $m(G)$ est égale à celle obtenue par *ExtSchemaGenerator*. Comme il a été prouvé dans [42] que la grammaire obtenue par *ExtSchemaGenerator* est le plus petit LTL qui contient $L(G)$ alors nous avons aussi que $L(m(G))$ est le plus petit LTL qui contient $L(G)$.

◁

Nous pouvons facilement vérifier que le mapping $m^{-1} = \langle a, b, c, d, e, f, g, h, i, j, k, l\rangle$ (voir ci-dessous) est l'inverse du mapping m, et permet de transformer la LTG de la Figure 6.10 en la RTG de la Figure 6.9.

a	$\texttt{ins_treerule}(I_3, info, B)$	g	$\texttt{del_opr}(I_1, \|, 0, 1)$
b	$\texttt{rel_elm}(H_1, I_1, I_3, 0.2.0)$	h	$\texttt{ins_treerule}(H_3, hospital, I_3^*)$
c	$\texttt{del_tree}(I_1, B, 0.2)$	i	$\texttt{del_tree}(H_1, I_3^*, 0.2)$
d	$\texttt{ins_treerule}(I_2, info, C \mid Pol)$	j	$\texttt{ins_treerule}(H_2, hospital, I_2^*)$
e	$\texttt{rel_elm}(H_1, I_1, I_2, 0.1.0)$	k	$\texttt{del_tree}(H_1, I_2^*, 0.1)$
f	$\texttt{del_tree}(I_1, C \mid Pol, 0.1)$	l	$\texttt{del_opr}(H_1, \|, 0, 1)$

6.3.2 Substitution de schéma

Le problème est le suivant : nous voulons décider si une RTG G est incluse de façon relâchée dans une RTG G'. Pour cela nous sommes amenés à calculer la grammaire des sous-arbres inclus relâchés $WI(G')$ de la RTG G'. Dans le dernier paragraphe de la Section 3.5 , nous avons évoqué une perspective intéressante qui serait de pouvoir calculer un mapping entre les RTG G' et $WI(G')$ afin de pouvoir connecter G à G' et traduire les documents dans les deux sens entre G' et G. Nous allons donc proposer une méthode pour calculer un mapping entre G' et $WI(G')$ en s'inspirant de la méthode pour calculer $WI(G')$ à partir de G'.

Algorithme 12 $genChMap(A)$: Calcul du mapping pour construire $Ch(A)$

Entrée : Une RTG $G = (N, \Sigma, S, P)$ en forme normale et un non-terminal A
Sortie : Le mapping m pour construire $Ch(A)$ si $Ch(A)$ n'est pas défini

$m := \langle\rangle$
si $Ch(A)$ n'est pas défini **alors**
  si A est 2-récursif **alors**
    Ajout de $\texttt{ins_opr}(A, |, 0, 1)$ à m
    Ajout de $\texttt{ins_tree}(A, (Succ(A))^*, 0.1)$ à m
    Ajout de $\texttt{del_tree}(A, t^r_A|_{0.0}, 0.0)$ à m **% *suppression de l'ancienne expression* %**
  sinon si A est 1-récursif **alors**
    % *remplacement de* $reg(A)$ ***par*** $\widehat{reg}(A)$ **%**
    Ajout de $\texttt{ins_opr}(A, |, 0, 1)$ à m
    Ajout de $\texttt{ins_tree}(A, \widehat{reg}(A), 0.1)$ à m
    Ajout de $\texttt{del_tree}(A, t^r_A|_{0.0}, 0.0)$ à m
    $m := m\,.\,genChAMap(A)$ **% *construction de la partie du milieu* %**
    Ajout de $\texttt{ins_opr}(A, ., 0, 1)$ à m
    Ajout de $\texttt{ins_tree}(A, (Succ(Left(A)))^*, 0.0)$ à m **% *construction de la partie gauche* %**
    Ajout de $\texttt{ins_tree}(A, (Succ(Right(A)))^*, 0.2)$ à m **% *construction de la partie droite* %**
  sinon
    $m := m\,.\,genChAMap(A)$
  fin si
fin si
retourner m

Algorithme 13 $genChAMap(A)$: Calcul du mapping pour construire $Ch^{rex}_{\widehat{A}}$

Entrée : Une RTG $G = (N, \Sigma, S, P)$ en forme normale, et un non-terminal A
Sortie : Le mapping m pour construire $Ch^{rex}_{\widehat{A}}$ si $Ch^{rex}_{\widehat{A}}$ n'est pas défini

$m := \langle\rangle$ **% *si*** $Ch^{rex}_{\widehat{A}}$ ***est défini alors on retourne*** $\langle\rangle$ **%**
pour tout non-terminal B à la position u dans t^r_A **faire**
  si B est 1-récursif et $B \in \widehat{A}$ **alors**
    % B *est remplacé par* $\widehat{B}|\epsilon$ %
    Ajout de $\texttt{ins_opr}(A, |, u, 1)$ à m
    Ajout de $\texttt{ins_tree}(A, \widehat{B}|\epsilon, u.1)$ à m
    Ajout de $\texttt{del_elm}(A, B, u.0)$ à m
    Ajout de $\texttt{del_opr}(A, |, u, 1)$ à m
  sinon si (B est 2-récursif) ou (B est 1-récursif et $B \notin \widehat{A}$) **alors**
    % B *est remplacé par* *Ch*(B) %
    $m := m\,.\,genChMap(B)$
    Ajout de $\texttt{ext_elm}(A, B, u)$ à m
  sinon
    % B *est remplacé par* $B|$*Ch*(B) %
    $m := m\,.\,genChMap(B)$
    Ajout de $\texttt{ins_opr}(A, |, u, 1)$ à m
    Ajout de $\texttt{ins_elm}(A, B, u.1)$ à m
    Ajout de $\texttt{ext_elm}(A, B, u.1)$ à m
  fin si
fin pour
retourner m

L'Algorithme 12 permet de construire un mapping pour l'expression régulière $Ch(A)$ (Définition 3.7) et l'Algorithme 13 permet de construire un mapping pour l'expression ré-

gulière $Ch^{rex}_{\widehat{A}}(E)$ (Définition 3.7). Pour remplacer par exemple $reg(A)$ par $(Succ(A))^*$ dans la règle de production de A, nous rajoutons l'opérateur '|' comme parent de $reg(A)$, ensuite nous rajoutons $(Succ(A))^*$ comme frère de $reg(A)$ et enfin nous supprimons $reg(A)$ (lignes 4 - 6, Algorithme 12). Dans l'Algorithme 13, pour remplacer un non-terminal B par $B|Ch(B)$ nous procédons de la manière suivante : nous ajoutons l'opérateur '|' comme parent de B, ensuite nous rajoutons un B comme frère de B, et enfin nous transformons le dernier B ajouté en $Ch(B)$ via l'opération `ext_elm`. Au moment où nous appliquons l'opération `ext_elm`, on a $reg(B) = Ch(B)$ ce qui fait qu'on a bien remplacé B par $B|Ch(B)$. Pour que l'Algorithme 12 et l'Algorithme 13 terminent, nous calculons qu'une seule fois $Ch(A)$ (resp. $Ch^{rex}_{\widehat{A}}(E)$) pour un non-terminal A donné (resp. pour une expression régulière E donnée). Nous allons supposer que nous stockons les résultats déjà calculés pour ensuite les utiliser lorsqu'on en a besoin de nouveau. Ceci nous évite donc de recalculer $Ch(A)$ (resp. $Ch^{rex}_{\widehat{A}}(E)$).

6.4 Adaptation de documents XML guidée par le mapping (*XTraM*)

Dans les sections précédentes, nous avons vu un langage de mapping, qui consiste en une suite d'opérations d'édition de grammaire RTG, permettant de transformer une RTG G en une autre RTG G'. Lorsque nous avons un arbre t qui a été généré par G, il est intéressant de savoir comment transformer t en un arbre t' généré par G' en se servant des modifications exprimées par le mapping entre G et G'. Le but de cette section est de proposer une méthode qui permet de faire une telle adaptation de documents, intervenant ainsi dans la tâche d'échanges de données entre des schémas XML (appelée « data exchange » en anglais). Cette tâche joue un rôle important dans la maintenance de l'interopérabilité entre différents systèmes. L'outil permettant d'exécuter cette tâche est nommé *XTraM*.

Exemple 6.5 Dans cet exemple nous considérons la grammaire G' de la Figure 6.10. Supposons que nous voulons faire évoluer la grammaire G' en G'' en renommant le terminal $info$ en $information$. Sur cette grammaire, cela correspond à l'opération $\texttt{rel_root}(I_1, info, information)$. Si nous prenons l'arbre t_1 de la Figure 6.11, les changements issus de l'opération $\texttt{rel_root}(I_1, info, information)$ consistent à renommer l'étiquette des trois nœuds aux positions 0, 1 et 2 en $information$.

⊠

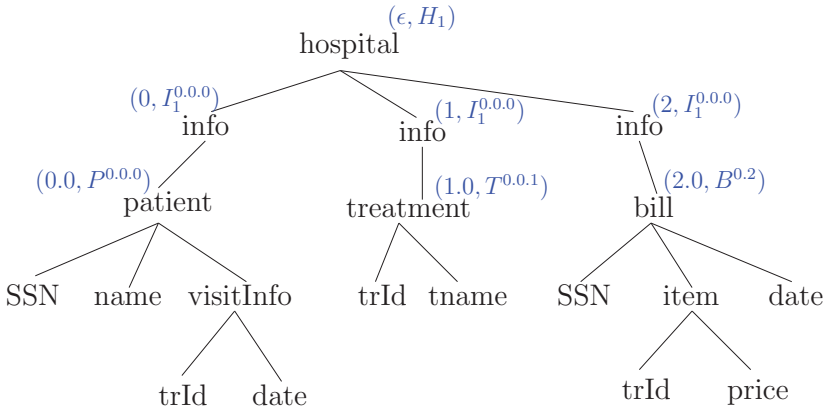


FIGURE 6.11 – *Un arbre XML valide et annoté par rapport au schéma global de la Figure 6.10.*

Notre méthode consiste à appliquer une liste de changements sur un arbre XML t, en accord avec les opérations d'édition de grammaire trouvées dans le mapping m. Par

exemple, l'ajout ou la suppression d'une expression régulière comme fils de l'opérateur '.' est une opération du mapping qui provoque respectivement l'insertion ou la suppression d'un sous-arbre dans l'arbre original t (pour maintenir sa validité par rapport au nouveau schéma obtenu). Lorsqu'une correction locale sur des sous-arbres est nécessaire, nous utilisons *XMLCorrector* (dont l'algorithme est décrit dans le Chapitre 5) pour assurer la validité des documents. Notre méthode suit les changements proposés par le mapping et effectue si besoin des corrections locales de sous-arbres en se servant de *XMLCorrector*. Considérons un arbre XML t valide par rapport au schéma (ou grammaire) S, et un mapping m qui va de S vers T. Notre méthode peut être résumée en ses deux étapes :

1. Puisque l'arbre t appartient au langage $L(S)$, il est possible d'avoir un arbre de dérivation de t c'est-à-dire d'associer un non-terminal A à chaque nœud de t (ayant la position p) qui a été généré par ce non-terminal. Nous analysons donc t pour détecter chaque non-terminal qui génère ses nœuds, et nous annotons chaque nœud de t avec le non-terminal correspondant et sa position u dans la règle de production utilisée. Cette annotation respecte le format (p, A^u) que nous avons utilisé pour annoter les nœuds de l'arbre XML de la Figure 6.11, où p est la position du nœud concerné dans l'arbre t, A est le non-terminal qui le génère et u est la position de A dans la règle de production utilisée. Dans la Figure 6.11, le nœud *bill* à la position 2.0 est généré par le non-terminal B dont la position dans $t^r_{I_1}$ (grammaire de la Figure 6.10) est 0.2, et tout cela est noté $(2.0, B^{0.2})$.
2. Chaque opération d'édition *ed* dans m active un ensemble de modifications sur t. Lorsque l'opération d'édition *ed* transforme la grammaire G en une nouvelle grammaire G' qui contient G (c'est-à-dire que $L(G) \subseteq L(G')$), l'ensemble des modifications sur t est vide. Dans le cas contraire, notre méthode effectue un parcours de t (annoté comme dans l'étape 1) afin de trouver les positions de t qui seront affectées par *ed*. Les modifications de t sont définies selon chaque opération d'édition. Nous verrons un cas dans le prochain exemple et les détails des modifications suivront. Il est donc possible qu'aucune position ne soit affectée et dans ce cas t n'est pas modifié.

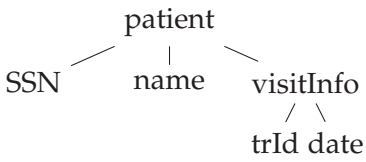
Exemple 6.6 Soient G la grammaire de la Figure 6.9 et G' la grammaire de la Figure 6.10. D'après la Section 6.3.1 nous savons que $L(G) \subseteq L(G')$. Comme tous les arbres générés par G appartiennent à $L(G')$, nous allons plutôt utiliser le mapping m^{-1} (en fin de Section 6.3.1) qui permet de transformer G' en G pour adapter le document t_1 de Figure 6.11 par rapport à la grammaire G. Nous avons bien $t_1 \in L(G')$. Voici comment est modifié l'arbre t_1 :

(a) Les nœuds de t_1 sont annotés en bleu, cf. Figure 6.11.

(b) L'opération $\texttt{ins_treerule}(I_3, info, B)$ n'implique aucun changement à t_1. En insérant une nouvelle règle de production à la grammaire G', nous obtenons une nouvelle grammaire qui contient G'. La nouvelle grammaire génère aussi t_1.

(c) L'opération $\texttt{rel_elm}(H_1, I_1, I_3, 0.2.0)$ renomme chaque non-terminal I_1, tel que $t^r_{H_1}(0.2.0) = I_1$, en I_3. Comme dans t_1, il n'y a pas d'annotation tel que $I_1^{0.2.0}$ est un fils de l'annotation (ϵ, H_1), alors il n' y a pas de modification à effectuer sur t_1.

(d) L'opération $\texttt{del_tree}(I_1, B, 0.2)$ supprime le non-terminal B tel que $t^r_{I_1}(0.2) = B$. Comme dans la Figure 6.11 l'annotation $(2.0, B^{0.2})$ (nœud *bill*) est un fils de l'annotation $(2, I_1^{0.0.0})$, nous aurons des changements à faire sur t_1. Puisque nous supprimons B de l'expression $(P \mid T) \mid (C \mid Pol) \mid B$, qui est un cas où un non-terminal est supprimé en dessous de l'opérateur '|', nous devons remplacer le sous-arbre généré par B par un sous-arbre généré par $(P \mid T) \mid (C \mid Pol)$ pour préserver la validité de t_1 par rapport à la nouvelle grammaire. L'outil *XMLCorrector* nous permet de corriger le

sous-arbre généré par B à la position 2.0 par rapport à l'expression $(P \mid T) \mid (C \mid Pol)$. Le résultat obtenu est le sous-arbre 

avec un coût (minimal) de 5. L'arbre t obtenu avec le nouveau sous-arbre est l'arbre t_2 de la Figure 6.12. Les annotations dans t_2 sont mises à jour du fait des changements opérés.

(e) Toutes les autres opérations du mapping m^{-1} n'impliquent pas de changements sur t_2 à cause des mêmes raisons qu'en (b) ou (c). Comme on peut s'y attendre, l'arbre résultat est t_2 qui est bien reconnu par la grammaire G.

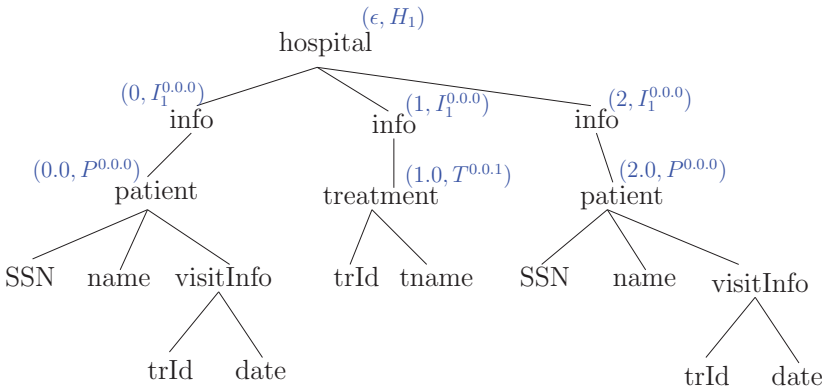


FIGURE 6.12 – *Un arbre XML valide et annoté par rapport au schéma de la Figure 6.9.*

⊠

Dans la suite, nous allons (i) donner les types d'opérations qui n'impliquent pas de changements sur l'arbre qu'on souhaite adapter, (ii) montrer comment on annote notre arbre et l'utilisation qu'on en fait, et (iii) donner les différentes modifications en fonction du type d'opérations.

Lemme 6.1 *Soient G une RTG et ed une opération d'édition définie sur G. Si ed satisfait l'une des conditions ci-dessous alors nous avons $L(G) \subseteq L(ed(G))$.*

1. $ed = \texttt{set_startelm}(A)$
2. $ed = \texttt{ins_elm}(X, A, u.i)$ *et*
   - *(i)* $t^r_X(u) = '.'$ *et* $A = \epsilon$, *ou*
   - *(ii)* $t^r_X(u) = '|'$
3. $ed = \texttt{del_elm}(X, A, u.i)$ *et*
   - *(i)* $t^r_X(u) = '.'$ *et* $A = \epsilon$, *ou*
   - *(ii)* $t^r_X(u) = '|'$ *et* $L(A) \subseteq L(t^r_X|_{u.k})$ *pour un entier* $k \neq i$
4. $ed = \texttt{rel_elm}(X, A, B, u)$ *et* $term(A) = term(B)$ *et* $L(reg(A)) \subseteq L(reg(B))$
5. $ed = \texttt{ins_opr}(X, opr, u.i, n)$ *et*
   - *(i)* $opr = t^r_X(u)$, *ou*
   - *(ii)* $t^r_X(u) = '*'$ *et* $opr \in \{., |\}$, *ou*
   - *(iii)* $t^r_X(u) \in \{., |\}$ *et* $opr = '*'$
6. $ed = \texttt{del_opr}(X, opr, u.i, n)$ *et*

(i) $opr = t^r_X(u)$ *ou*

(ii) $t^r_X(u) = '*'$ *et* $opr \in \{., |\}$

7. $ed = \texttt{rel_opr}(X, op, opr, u)$ *et l'opérateur* op *ayant un seul fils et* $op \neq '*'$
8. $ed = \texttt{ins_rule}(A, a)$ *ou* $ed = \texttt{ins_treerule}(A, a, R)$
9. $ed = \texttt{ins_tree}(X, R, u.i)$ *et*

(i) $t^r_X(u) = '.'$ *et* $\epsilon \in L(R)$, *ou*

(ii) $t^r_X(u) = '|'$

10. $ed = \texttt{del_tree}(X, R, u.i)$ *et*

(i) $t^r_X(u) = '.'$ *et* $L(R) = \{\epsilon\}$, *ou*

(ii) $t^r_X(u) = '|'$ *et* $L(R) \subseteq L(t^r_X|_{u.k})$ *pour un entier* $k \neq i$

11. $ed = \texttt{inv_tree}(X, u.i)$ *et* $t^r_X(u) = '|'$
12. $ed = \texttt{ext_elm}(X, A, u)$ *et* $A \in L(reg(A))$.

Le Lemme 6.1 nous indique les opérations d'édition de grammaire qui permettent d'agrandir un langage donné. Le langage généré par la nouvelle grammaire obtenue contient le langage de l'ancienne grammaire. Par exemple (Lemme 6.1 (3ii)) le fait de supprimer un non-terminal A en dessous de l'opérateur '$|$' et que la condition suivante « les sous-expressions qui sont des frères directs de A peuvent générer A » est vérifiée, alors la grammaire obtenue génère le même langage que celle de départ. Lorsque l'on adapte un document t, pour ces opérations, aucun changement n'est appliqué à t.

La technique que nous utilisons pour annoter (ou construire le $nt{-}tree$ correspondant) un arbre t par rapport à une grammaire RTG est assez simple. Prenons par exemple un nœud quelconque de notre arbre et supposons que nous connaissons le non-terminal A qui l'a généré. Pour savoir comment sont générés les fils de ce nœud, l'idée est de créer une expression régulière à partir des non-terminaux potentiels qui génèrent chacun de ses fils, afin de faire une intersection avec l'expression régulière associée à A. Par exemple pour la grammaire G' de la Figure 6.9, pour le terminal $info$ il est possible de le générer via les non-terminaux I_1, I_2 et I_3. Si nous prenons par exemple le mot $info.info.info$ nous allons construire l'expression régulière $(I_1|I_2|I_3).(I_1|I_2|I_3).(I_1|I_2|I_3)$. L'intersection de cette expression avec l'expression I_1^* du non-terminal H_1 nous donne le singleton $\{I_1.I_1.I_1\}$. Ainsi nous savons les différentes façons de générer le mot $info.info.info$ à partir de l'expression I_1^* du non-terminal H_1. En combinant cette façon de faire pour tous les nœuds de t nous obtenons les différents $nt{-}tree$ correspondant à t. Notons que lorsque la grammaire RTG est déterministe alors la cardinalité de l'ensemble des mots sur les non-terminaux est fortement réduite car nous avons une disjonction que pour les non-terminaux qui sont en concurrences. Par contre si la RTG n'est pas déterministe (dans ce cas elle comporte des expressions régulières qui ne sont pas 1-unambiguous), il faudra prendre aussi en compte les positions des non-terminaux lors de la génération de l'expression régulière pour un mot donné. Le cas le plus simple est celui des LTG déterministes du fait qu'il existe un seul $nt{-}tree$ possible pour un arbre donné et cela s'explique par le fait qu'il n'y a pas de non-terminaux en concurrence. Le W3C recommande des DTD déterministes pour décrire les schémas ce qui fait que dans la pratique le nombre de $nt{-}tree$ n'explose pas.

A partir d'un $nt{-}tree$ pour un arbre t, nous utilisons deux fonctions $updateNodePositions$ et $updateForestPositions$ qui permettent respectivement de déterminer les positions de t qui sont affectées par une modification causée par une opération d'édition ed. La fonction

updateNodePositions détermine une seule position de nœud qui est due par exemple à l'insertion d'un non-terminal dans une expression, et la fonction *updateForestPositions* détermine une série de positions consécutives ou forêt qui est due par exemple à la suppression d'une sous-expression dans une expression. Voici un résumé et un exemple de ce que fait ces deux fonctions.

- $updateNodePositions(t^0, X, u)$: cette fonction calcule à partir de l'arbre annoté t^0 de l'arbre XML t, toutes les positions qui sont concernées par une mise à jour. En premier cette fonction identifie les positions v de t qui sont étiquetées par $term(X)$. Ensuite elle retourne les positions filles des positions v qui sont affectées par la suppression ou l'insertion d'un non-terminal à la position u dans t^r_X. La fonction retourne une liste de positions de t.
- $updateForestPositions(t^0, X, u, n)$: cette fonction calcule à partir de l'arbre annoté t^0 de l'arbre XML t, toutes les forêts de positions qui sont concernées par une mise à jour. En premier cette fonction identifie les positions v de t qui sont étiquetées par $term(X)$. Ensuite elle retourne les positions consécutives filles des positions v qui sont affectées par la suppression ou l'insertion d'un opérateur, ou la suppression d'une sous-expression à la position u dans t^r_X. L'entier n correspond au nombre de nœuds (la position u et ses $n-1$ frères directs). La fonction retourne une liste de forêts de positions de t.

Exemple 6.7 Prenons pour exemple la règle de production $X \to R[A^*.B.((C.D)|F).G]$ de la Figure 6.13(a) et l'arbre annoté t^0 de la Figure 6.13(b). Nous avons :

- $updateNodePositions(t^0, X, 0.0.0)$ retourne les positions 0 et 1.
- $updateNodePositions(t^0, X, 0.2.0.0)$ retourne une liste vide.
- $updateNodePositions(t^0, X, 0.2.1)$ retourne la position 3.
- $updateNodePositions(t^0, X, 0.4)$ retourne la position 5. Ceci correspond à l'insertion à une nouvelle position dans t^r_X. Les positions retournées suivent immédiatement une position dans t^0 générée par le non-terminal G à la position 0.3 dans t^r_X.

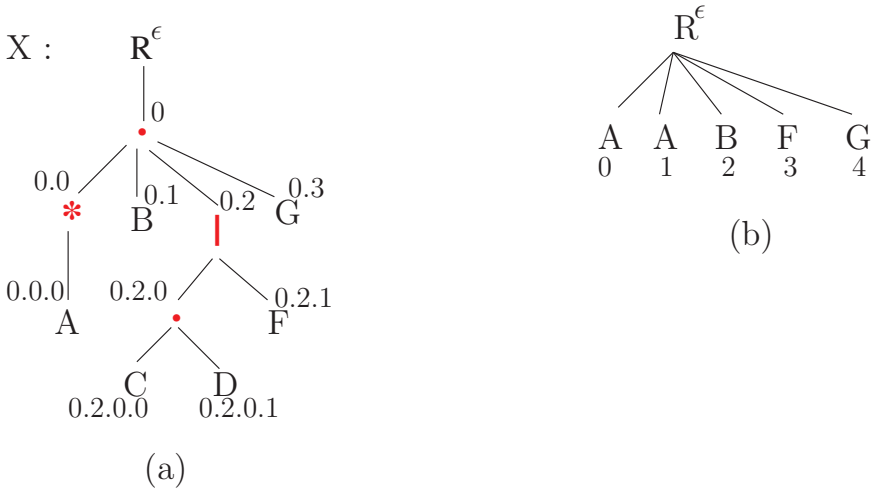


FIGURE 6.13 – *Exemple d'utilisation des fonctions updateNodePositions et updateForestPositions.*

- $updateForestPositions(t^0, X, 0.0, 1)$ retourne la forêt (0,1) car A^* a généré le mot $A.A$.
- $updateForestPositions(t^0, X, 0.0, 4)$ retourne la forêt (0,1,2,3,4). Ceci est équivalent à $updateForestPositions(t^0, X, 0, 1)$.

– $updateForestPositions(t^0, X, 0.1, 2)$ retourne la forêt (2,3) car l'on recherche les positions générées par l'expression $B.((C.D)|F)$.
– $updateForestPositions(t^0, X, 0.2.0, 1)$ retourne une liste vide car l'on recherche les positions pour le mot $C.D$.

Définition 6.10 (Changements provoqués par une opération d'édition sur un arbre XML)
L'algorithme d'adaptation de documents procède par l'analyse de chaque opération dans le mapping $\mathcal{M}$. Nous énumérons ci-dessous les opérations d'édition *ed* avec les situations particulières (en écartant celles du Lemme 6.1), pour lesquelles des changements dans l'arbre XML sont nécessaires. Notons que la correction de sous-arbre (ou de forêt) par rapport à un non-terminal (ou une expression régulière) est assurée par *XMLCorrector*.

1. si $ed = \texttt{ins_elm}(X, A, u.i)$, alors nous sommes dans la situation où $t^r_X(u) = '.'$; ainsi pour chaque position w dans $updateNodePositions(t^0, X, u.i)$, un nouvel arbre localement valide par rapport au non-terminal A est inséré à la position w dans l'arbre t.

2. si $ed = \texttt{del_elm}(X, A, u.i)$, alors nous sommes dans la situation où $t^r_X(u) \in \{., |\}$; ainsi pour chaque position w dans $updateNodePositions(t^0, X, u.i)$ nous avons :
   (a) si $t^r_X(u) = '.'$ alors le sous-arbre à la position w dans t est supprimé ;
   (b) sinon si $t^r_X(u) = '|'$ alors le sous-arbre à la position w dans t, est remplacé par un(e) nouvel(le) arbre/forêt valide par rapport à l'expression régulière $t^r_X|_{u.k}$ où k est un entier tel que $k \neq i$. Le remplacement de $t|_w$ est effectué par un appel de *XMLCorrector* pour corriger $t|_w$ par rapport à l'expression régulière $t^r_X|_{u.k}$.

3. si $ed = \texttt{rel_root}(X, a, b)$ alors nous renommons par b chaque position w dans t telle que X génère le nœud à la position w et $t(w) = a$.

4. si $ed = \texttt{rel_elm}(X, A, B, u)$ alors pour chaque position w dans $updateNodePositions(t^0, X, u)$, le sous-arbre à la position w dans t est corrigé à l'aide de *XMLCorrector* par rapport au non-terminal B.

5. si $ed = \texttt{ins_opr}(X, opr, u.i, n)$ alors nous sommes dans la situation où $u \neq \epsilon$, $t^r_X(u) \neq opr$, $t^r_X(u) \in \{., |\}$ et $opr \in \{., |\}$; ainsi pour chaque forêt $(w_1, \ldots, w_n)$ dans $updateForestPositions(t^0, X, u.i, n)$ nous avons :
   (a) si $t^r_X(u) = '|'$ et $opr = '.'$ alors la forêt aux positions $(w_1, \ldots, w_n)$ dans t doît être corrigée par rapport à l'expression régulière $t^r_X|_{u.i}$ obtenue après application de *ed* sur la grammaire originale.
   (b) si $t^r_X(u) = '.'$ et $opr = '|'$ alors idem (5a).

6. si $ed = \texttt{del_opr}(X, opr, u.i, n)$ alors nous sommes dans la situation où $t^r_X(u) \neq opr$; ainsi pour chaque forêt $(w_1, \ldots, w_n)$ dans $updateForestPositions(t^0, X, u.i.0, n)$ nous avons :
   (a) si $t^r_X(u) = '|'$ et $opr = '.'$ alors idem (5a).
   (b) si $t^r_X(u) = '.'$ et $opr = '|'$ alors idem (5a).
   (c) si $t^r_X(u) \in \{., |\} \cup \Sigma$ et $opr = '*'$ alors idem (5a).

7. si $ed = \texttt{rel_opr}(X, opr, opr', u)$ alors pour chaque forêt $(w_1, \ldots, w_n)$ dans $updateForestPositions(t^0, X, u, 1)$, nous corrigeons la forêt d'arbres aux positions $(w_1, \ldots, w_n)$ dans t par rapport à l'expression régulière $t^r_X|_u$ obtenue après avoir appliquée l'opération *ed* sur la grammaire originale.

8. si $ed = \texttt{ins_tree}(X, R, u.i)$ alors nous sommes dans la situation où $t^r_X(u) = '.'$; ainsi pour chaque position w dans $updateNodePositions(t^0, X, u.i)$, une nouvelle forêt de sous-arbres localement valide par rapport à l'expression régulière R est insérée à la position w dans l'arbre t.
9. si $ed = \texttt{del_tree}(X, R, u.i)$, alors nous sommes dans la situation où $t^r_X(u) \in \{., |\}$; ainsi pour chaque forêt $(w_1, \ldots, w_n)$ dans $updateForestPositions(t^0, X, u.i, 1)$ nous avons :
   (a) si $t^r_X(u) = '.'$ alors la forêt aux positions $(w_1, \ldots, w_n)$ dans t est supprimée ;
   (b) sinon si $t^r_X(u) = '|'$ alors la forêt aux positions $(w_1, \ldots, w_n)$ dans t, est corrigée par rapport à l'expression régulière $t^r_X|_{u.k}$ où k est un entier tel que $k \neq i$.
10. si $ed = \texttt{inv_tree}(X, u.i)$ alors nous sommes dans la situation où $t^r_X(u) = '.'$; ainsi on inverse chaque forêt de sous-arbres aux positions $(w_1, \ldots, w_n)$ dans $updateForestPositions(t^0, X, u.i, 1)$ avec la forêt qui la suit immédiatement se trouvant aux positions $(w'_1, \ldots, w'_m)$ dans $updateForestPositions(t^0, X, u.(i+1), 1)$.
11. si $ed = \texttt{agg_elm}(X, A, u)$ alors pour chaque forêt $(w_1, \ldots, w_n)$ dans $updateForestPositions(t^0, X, u, 1)$, nous modifions l'arbre t de telle sorte que $term(A)$ devienne le parent de la forêt de sous-arbres aux positions $(w_1, \ldots, w_n)$.
12. si $ed = \texttt{ext_elm}(X, A, u)$ alors pour chaque position w dans $updateNodePositions(t^0, X, u)$, nous modifions l'arbre t de telle sorte que le nœud $term(A)$ à la position w dans t soit remplacé par ses enfants.
13. si $ed = \texttt{unset_startelm}(A)$ ou $ed = \texttt{del_rule}(A, a)$ alors dans le cas où la racine de t a été générée par A, une correction de tout l'arbre t est faite par rapport à un symbole initial de la grammaire originale.

□

Théorème 6.2 *Soient deux RTG G et G', un arbre t tel que $t \in L(G)$ et une opération d'édition ed qui transforme G en G'. En appliquant les transformations de la Définition 6.10 issues de ed sur t, on obtient un arbre $t' \in L(G')$.*

Démonstration. La preuve de ce théorème repose essentiellement sur la preuve de correction et de complétude de l'algorithme utilisé par *XMLCorrector* et sur le fait que l'opération ed transforme bien G en G'. Notre méthode consiste à faire des corrections locales dans l'arbre t à des endroits que nous détectons comme invalides. En fonction du type de l'opération ed, soit nous insérons des sous-arbres ou des forêts localement valides dans t ; soit nous supprimons des sous-arbres ou des forêts dans t ; soit nous remplaçons des sous-arbres ou des forêts par des sous-arbres ou des forêts localement valides.

◁

Par défaut, nous proposons des adaptations $t'_1, \ldots, t'_n$ de t qui sont à la même distance minimale de t. L'utilisateur pourra ensuite choisir les documents qu'il souhaite garder. Puisque *XMLCorrector* offre la possibilité d'avoir les corrections de coûts minimaux mais aussi les corrections dont le coût est inférieur à un seuil th donné, nous pouvons aussi proposer à l'utilisateur toutes les adaptions t' dont la distance entre t et t' est inférieure à th. En fonction du type d'application, il est possible qu'on ne souhaite pas forcément avoir la solution de coût minimal. Ceci donne une flexibilité à notre proposition.

Pour le moment nous annotons l'arbre de départ tout entier. Une optimisation possible, est de détecter dans le mapping les non-terminaux et règles concernées afin d'annoter seulement les parties de l'arbre qui sont générées par ces non-terminaux.

6.5 État de l'art

6.5.1 Mapping de schémas basé sur le formalisme logique

Dans le contexte de l'évolution de schéma, un schéma source peut évoluer en un schéma cible. Un mapping de schéma est défini comme un triplet (S, T, Σ) où S est le schéma source, T est le schéma cible et Σ est le langage qui spécifie la relation entre le schéma source et le schéma cible. Généralement Σ est un ensemble de dépendances (ou contraintes) qui sont exprimées par des formules logiques du premier ordre du type std ou tgd (« source-to-target dependency ») [24, 60, 61]. Les formules logiques utilisées sont de la forme $\forall \bar{x}(\phi(\bar{x}) \rightarrow \exists \bar{y}(\psi(\bar{x}, \bar{y})))$ où $\bar{x}$ et $\bar{y}$ sont des vecteurs de variables, $\phi(\bar{x})$ est une conjonction de formules atomiques sur le schéma source et $\psi(\bar{x}, \bar{y})$ est une conjonction de formules atomiques sur le schéma cible.

Dans [24, 60, 61], les contraintes sont de différents types. Nous avons les correspondances entre le schéma source et le schéma cible, mais aussi les dépendances d'inclusion, et les dépendances fonctionnelles. Ces dernières peuvent être exprimées sur le schéma source et traduites (via les correspondances) vers le schéma cible au moment de la validation. Les correspondances entre les deux schémas permettent de mieux comprendre la relation entre les deux schémas et sont trouvées lors de la phase de matching. Le matching de schéma consiste donc à créer des liens entre les éléments (représentant les mêmes concepts) des schémas. Le matching de schémas est souvent effectué par l'utilisateur, mais aussi par des outils de matching, comme Clio et Coma, qui produisent des correspondances que l'utilisateur valide. Un panorama sur les techniques utilisées pour faire le matching de schéma est proposée dans [3, 96, 103]. Dans [18, 96, 103], certains outils de matching sont décrits et un comparatif de leurs caractéristiques est proposé. Comme outils qui proposent le matching automatique de schémas XML nous avons : Coma [55, 56], Cupid [84], Harmony [88], YAM [58, 59], XClust [79]. Même si la définition de mappings dans [24, 60, 61] peut être utilisée à la fois pour le relationnel et XML, les dépendances du type « source-to-target dependencies » plus adaptées à XML ont été introduites dans [4, 16]. Les auteurs utilisent des « tree patterns » pour exprimer les formules atomiques.

Il peut y avoir plusieurs instances solutions pour une instance du schéma source et un mapping donné. Parmi les instances solutions, une solution particulière appelée solution universelle est considérée. Une solution universelle est une instance du schéma cible qui contient ni plus, ni moins d'informations que ce que le mapping requiert. Pour calculer une solution universelle pour une instance du schéma source, des techniques de « chase » [85] et des fonctions « Skolem » [71] sont utilisées pour générer des valeurs basées sur les valeurs de l'instance du schéma et aussi générer des valeurs manquantes dans l'instance calculée pour le schéma cible. Dans [61], les auteurs proposent des méthodes pour calculer la solution universelle, et ils montrent sous certaines conditions qu'une solution universelle, si elle existe, peut être calculée en temps polynomiale.

Des algorithmes pour composer des mappings et inverser des mappings sont proposées dans [62] pour le cas relationnel et dans [4, 126] pour le cas XML. Dans ces papiers il est étudié le langage adéquat pour exprimer la composition de mapping car la logique de second ordre est nécessaire dans certains cas. L'objectif de ses travaux est de trouver un langage de mapping qui est clos par composition et par inversion et qui a de bonnes propriétés pour les algorithmes.

Pour adapter ou traduire une instance du schéma source, les mappings exprimés sont transformés en des scripts exécutables pour modifier les instances du schéma de départ dans le but d'obtenir des instances pour le schéma d'arrivée. Lorsque les schémas sont des schémas relationnels, des scripts SQL sont générés et lorsque des schémas XML sont utilisés,

des scripts XQuery ou XSLT sont générés. Ces scripts sont ensuite exécutés sur l'instance du schéma source pour obtenir une instance du schéma cible conforme au mapping. En considérant les dépendances du type « source-to-target dependencies » adaptées à XML [4, 16], il n'existe pas encore de méthodes pour calculer une solution d'une instance du schéma source pour un mapping donné. Les techniques décrites pour réaliser l'échange de données, sont utilisées dans l'outil de mapping Clio [60].

Les problèmes rencontrés dans ce genre de technique (énumérés dans [60]) sont les suivants : l'algorithme pour faire le « chase » en se servant des dépendances fonctionnelles ou des dépendances d'inclusion, ne termine pas dans tous les cas (car on peut tomber sur des cycles entre les dépendances). La complexité de l'algorithme de « chase » est exponentiel dans le pire des cas. La récursivité dans les schémas XML est une source de problème car elle peut entraîner une infinité de contraintes dans le mapping. Pour notre cas, le fait que le schéma soit récursif ne cause aucun problème pour exprimer le mapping.

Dans notre approche, nous nous intéressons à la modification de la structure du document source en prenant en compte le mapping. Vu l'importance des dépendances fonctionnelles et d'inclusions dans le processus d'échange de données, à partir des dépendances fonctionnelles et d'inclusions pour XML définies dans le Chapitre 4 et dans [28, 29, 46], nous pouvons envisager d'utiliser la technique de « chase » [85] pour compléter et générer les valeurs manquantes dans le document solution calculé.

6.5.2 Mapping de schémas basé sur l'évolution incrémentale des schémas

Plusieurs autres approches [40, 70, 80, 91, 111] se basent sur les scripts d'éditions pour exprimer l'évolution de schémas XML.

Dans [111], les auteurs s'intéressent au problème de transformation de document XML suite à la mise à jour d'une DTD. L'évolution de DTD se fait via un script d'édition sur les DTD en modélisant comme nous les expressions régulières sous forme d'arbres. En ce qui concerne les opérations d'édition, notre méthode est plus générale du fait que nous traitons les grammaires RTG alors qu'ils ne traitent que les grammaires LTG. Ils n'autorisent pas, par exemple, la modification des symboles terminaux et ne possèdent pas d'opérations pour renommer les non-terminaux, ou les opérateurs. Certes, il est possible de simuler le renommage par la suppression de l'ancien nœud et l'insertion du nouveau nœud mais l'impact sur l'adaptation des documents est important. Dans [111] nous trouvons aussi une restriction sur l'insertion et la suppression d'opérateurs car ils autorisent seulement l'insertion d'un opérateur comme fils du même opérateur. Dans ce cas, le langage engendré par l'ancienne expression régulière reste inchangé. Avec ces restrictions, l'impact des opérations d'insertion et de suppression d'opérateurs n'a pas d'effet sur l'adaptation de documents sauf pour le cas où l'opérateur '$*$' est supprimé où l'on doit supprimer les sous-arbres de trop. Dans notre approche ses restrictions ne sont pas prises en compte. En utilisant l'algorithme de correction de documents lors du processus d'adaptation, nous arrivons à prendre en compte les modifications causées sur les documents par ses restrictions. Pour des raisons d'efficacité, l'adaptation de documents proposée dans [111] ne doit pas être ambiguë, c'est-à-dire qu'il doit exister une seule solution pour un document XML et un script d'édition donné. Pour garantir cela, un algorithme en temps polynomial est proposé pour détecter si la transformation d'un document est ambiguë par rapport à un script d'édition. Les mêmes auteurs proposent dans [107], une méthode basée sur les scripts d'édition pour tester l'inclusion entre deux DTD. Une restriction est faite sur les opérations d'édition. Ce problème étant PSPACE-complet, lorsqu'ils arrivent à trouver un script d'édition avec leur

opérations restreintes alors l'inclusion est vérifiée, sinon ils ne peuvent pas décider. Leur algorithme s'exécute en temps polynomial.

Une méthode pour calculer automatiquement un script d'édition entre deux RTGs est proposée dans [70]. Nous représentons nos règles de production de la même manière qu'eux, c'est-à-dire sous forme d'arbres. Rappelons que la structure de l'arbre est de telle sorte que le terminal est la racine de l'arbre, les opérateurs sont des nœuds internes, et les non-terminaux sont des feuilles. Leurs opérations d'édition de RTG sont donc similaires aux nôtres. Les seules différences sont : (i) nous n'autorisons pas la modification d'un non-terminal à gauche d'une règle de production alors que cela est possible dans [70]. Dans [70], les auteurs montrent que le problème de calcul de script d'édition entre deux RTGs est NP-difficile et cela est dû à la modification de la partie gauche d'une règle de production (vu les différentes combinaisons que l'on peut avoir). Nous supposons que nos grammaires RTG sont en forme normale et nous n'autorisons pas de changer le non-terminal à gauche d'une règle par un autre non-terminal ; (ii) nous modifions un opérateur seulement par un autre opérateur, et un non-terminal seulement par un non-terminal ou le symbole ϵ alors que dans [70], ces restrictions ne sont pas prises en compte et il est donc possible de modifier l'étiquette des nœuds de l'arbre représentant une règle de production par n'importe quelle étiquette (qu'elle soit un non-terminal ou un opérateur). Le problème traité dans [70] est aussi intéressant pour nous, mais les solutions de script d'édition qu'ils proposent ne sont pas compatibles avec la façon dont nous adaptons ensuite les documents XML car nous ne savons pas quelles modifications seront appliquées à nos documents lorsqu'un opérateur est, par exemple, renommé en un non-terminal. Pour pouvoir calculer automatiquement le script d'édition entre deux RTGs, nous devons adapter l'algorithme d'édition entre deux arbres à nos restrictions sur les opérations d'édition d'arbre.

ELaX (Evolution Language for XML-Schema) [91] et Exup [40] sont des langages spécifiques qui proposent des modifications sur XML Schema (XSD) du W3C. ELaX est basé sur trois opérations primitives (l'ajout (add), la suppression (delete) et la mise à jour (update)) qui sont adaptées à chaque type d'élément dans XML Schema, et sur le langage de chemin XPath. Leur approche peut se résumer au traitement du schéma XML Schema comme un seul arbre sur lequel sont appliquées les primitives d'opérations. Il est aussi possible dans [40, 91] d'adapter les documents en se servant des opérations dans le mapping exprimé entre deux schémas.

Dans [68], un état de l'art est fait sur les différents outils de mappings de schéma. Les auteurs décrivent aussi comment la plupart des SGBD (Système de Gestion de Base de Données) commerciaux, qui offrent la possibilité de stocker des documents XML, traitent l'évolution de schéma. Les SGBD comme *Oracle XML Schema Evolution*, *Microsoft SQL Server*, et *IBM DB2* utilisent un langage propriétaire pour exprimer les primitives d'évolution de schéma et n'autorisent pas les primitives qui invalident les documents existants. L'insertion d'un élément optionnel dans le schéma est, par exemple, une primitive autorisée car elle n'a pas d'effet sur les documents existants.

6.6 Discussions et remarques

Les travaux de ce chapitre ont pour objectif de fournir des outils pour effectuer l'évolution de schémas et assurer l'échange de données. L'évolution de schémas est exprimée par un script d'édition sur des grammaires d'arbres réguliers. Pour ce langage de mapping, deux

opérateurs de mapping, la composition et l'inversion, sont définis et ont pour but d'adapter un mapping $\mathcal{M}$ lorsque son schéma source ou son schéma cible évolue en un autre schéma. Ensuite nous avons proposé des méthodes pour :

(i) exprimer des mappings entre des systèmes locaux et leur évolution conservatrice en un système global, en nous basant sur la méthode d'extension minimale d'une grammaire RTG en LTG [42]. L'aspect conservateur garantit une flexibilité lorsque le schéma global coexiste avec les systèmes locaux ;

(ii) exprimer un mapping entre une grammaire RTG G et la grammaire $WI(G)$ de ses sous-arbres relâchés en nous basant sur l'algorithme présenté dans le Chapitre 3 pour calculer $WI(G)$ à partir de G.

Enfin nous avons présenté notre méthode pour adapter des documents XML suite à une évolution de leur schéma.

Un prototype a été implanté (en Java) et testé. Comme premier scénario de test, nous avons produit une LTG en fusionnant (calcul de l'extension minimale) les grammaires obtenues à partir de la DTD de *dblp* [1] et du XML Schema (XSD) de *HAL* [2]. Notre outil *MappingGen* retourne un mapping composé de 19 opérations d'édition. Ensuite l'outil *XTraM* a été utilisé pour adapter un document de 52 nœuds valide par rapport à la LTG calculé vers la grammaire pour *HAL*, et nous obtenons pour ce cas 36 solutions en 22.6 s. Dans ce test, toutes les traductions possibles ont été considérées mais l'utilisateur peut interférer sur les étapes intermédiaires en faisant des choix avant la fin du calcul des solutions. Dans ce sens, l'utilisateur guide et réduit le nombre de solutions. Cette fonctionnalité sera implémentée dans notre logiciel qui est décrit dans le Chapitre 7 - Section 7.2, pour faciliter l'interférence de l'utilisateur à des étapes intermédiaires lors du processus.

Pour de futurs travaux, il serait intéressant de pouvoir calculer automatiquement le mapping entre deux schémas donnés pour faciliter la tâche à l'utilisateur. Pour le moment le mapping peut être défini par l'utilisateur ou généré automatiquement pour une utilisation spécifique dans le cadre des deux applications (Section 6.3). Ainsi celui-ci pourra juste valider le mapping généré ou le corriger s'il ne répond pas à ces attentes. Aussi dans notre approche, l'adaptation de document tient seulement compte de sa structure. Comme dans les propositions [60, 24] (détaillées en Section 6.5), nous souhaiterions mettre à profit les dépendances fonctionnelles pour XML (Chapitre 4) en utilisant la technique de « chase » [85] pour compléter et générer les valeurs manquantes dans les solutions obtenues par *XTraM*.

Une autre perspective serait d'utiliser le mapping pour adapter les dépendances fonctionnelles définies sur le schéma source. De tels travaux peuvent rejoindre les travaux sur la réécriture des requêtes sur les documents en présence d'un mapping [23, 76] car les requêtes et les dépendances fonctionnelles manipulent des chemins dans les documents XML.

1. `http://dblp.uni-trier.de/xml/dblp.dtd`
2. `http://import.ccsd.cnrs.fr/xsd/generationAuto.php?instance=hal`

Les solutions logicielles

7

Sommaire

7.1 *XMLCorrector* 161
7.2 *DTDGrabber* 164

Nous allons aborder dans ce chapitre deux applications : la première nommée *XMLCorrector* qui est disponible en ligne [1] et la deuxième (en cours de développement) nommée *DTDGrabber* qui est en cours de développement. Ces deux applications s'appuient essentiellement sur les travaux abordés dans ma thèse et aussi sur d'autres travaux antérieurs.

7.1 *XMLCorrector*

L'application *XMLCorrector* est une implantation de l'algorithme de correction décrit dans le Chapitre 5, qui permet de corriger un document XML par rapport à un schéma XML exprimé comme une DTD. *XMLCorrector* est une application open source et est distribuée sous la licence GNU LGPL v3. Le principe de *XMLCorrector* est illustré dans la Figure 7.1.

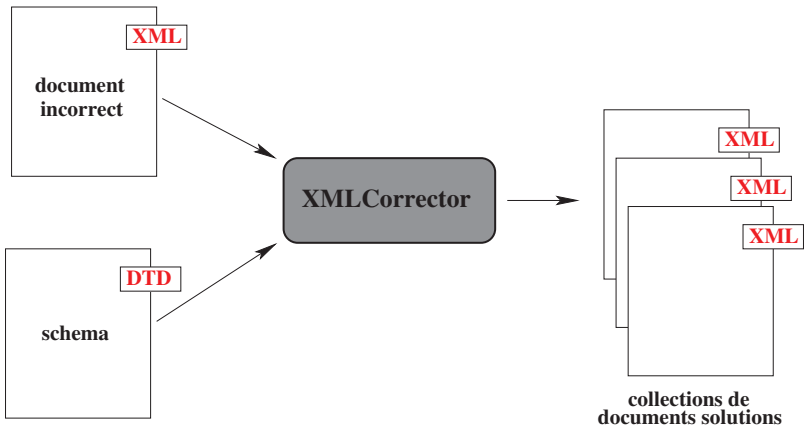


Figure 7.1 – *Illustration de XMLCorrector.*

A partir des opérations définies dans la Section 5.2 du Chapitre 5, *XMLCorrector* propose des séquences d'opérations d'édition d'arbres pour corriger un arbre XML. *XMLCorrector*

1. http://www.info.univ-tours.fr/~savary/English/xmlcorrector.html

ne gère pas les valeurs aux feuilles dans l'arbre XML. Il existe deux modes d'utilisation de *XMLCorrector* pour corriger un arbre XML t par rapport à une DTD. Le premier mode, qui est une contribution importante dans le domaine, utilise un seuil entier th pour trouver toutes les solutions t' valides par rapport à la DTD et dont la distance avec l'arbre XML t est inférieure au seuil th. Le second mode permet de trouver uniquement toutes les corrections de coût minimal, c'est-à-dire la distance entre chacune de ses solutions et l'arbre t est exactement égale à la distance entre l'arbre t et la DTD concernée. Les autres fonctionnalités de *XMLCorrector* sont :

1. la possibilité de pouvoir visualiser les séquences résultats (la liste des opérations), les séquences de même coût qui lui sont équivalentes et l'arbre obtenu après l'application des séquences sur l'arbre de départ ;
2. la possibilité d'avoir les statistiques d'une correction à savoir : le temps CPU mis pour la correction, le nombre d'arbres différents trouvés, les valeurs des paramètres et les fichiers dans lesquels sont stockés les résultats ;
3. la possibilité de changer les coûts des opérations d'édition.

La fenêtre principale de *XMLCorrector* est illustrée dans la Figure 7.2.

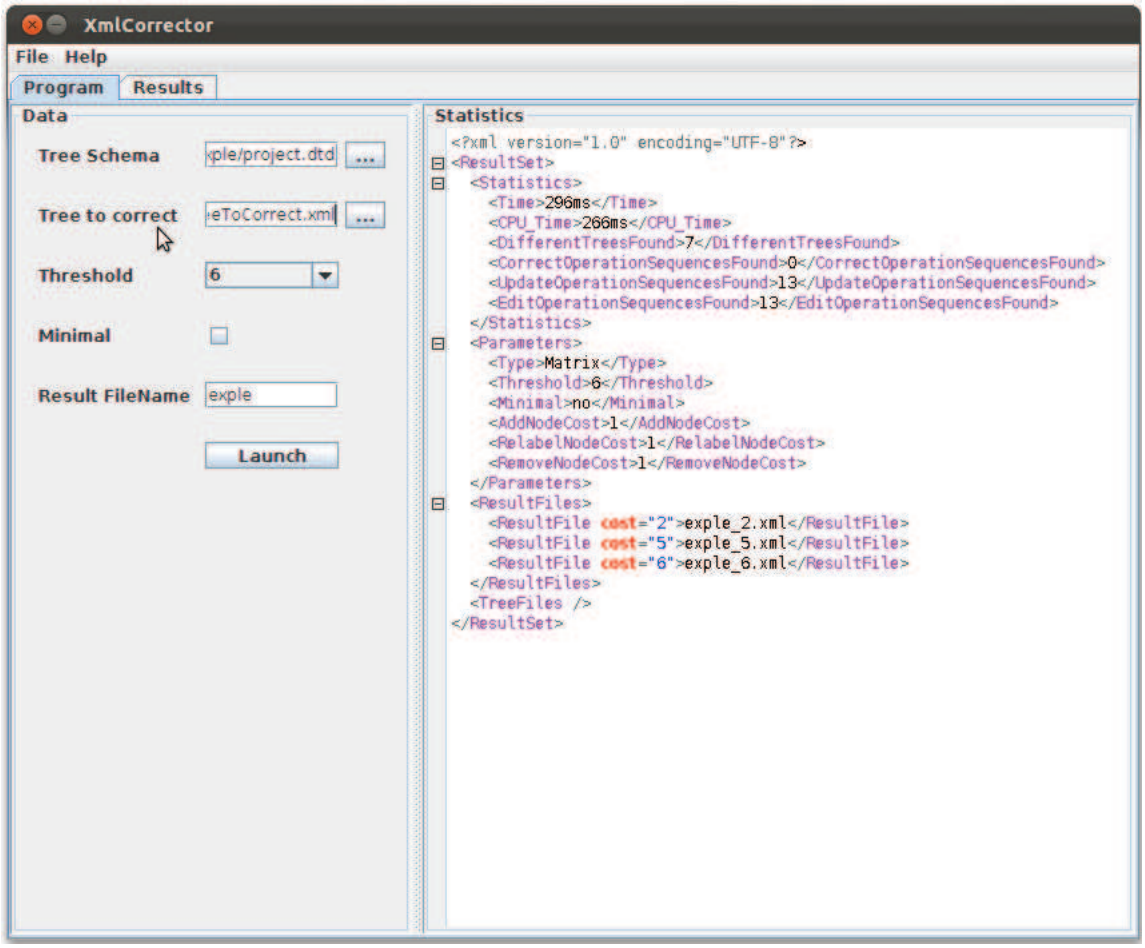


FIGURE 7.2 – *Fenêtre principale de XMLCorrector.*

Pour pouvoir corriger un document XML, l'utilisateur a besoin de renseigner sur le fichier XML contenant le document, le schéma DTD, le seuil et s'il souhaite avoir que les solutions minimales ou toutes les solutions en dessous du seuil. Les statistiques concernant une correction sont données en format XML (cf. Figure 7.3). Les séquences résultats sont stockées

dans des fichiers XML de tel sorte que toutes les séquences ayant le même coût sont dans le même fichier. Aussi chaque arbre solution est stocké dans un fichier XML différent. Nous pouvons visualiser les séquences résultats et les arbres qu'ils donnent à partir de l'arbre de départ dans l'onglet « Results ». Dans la Figure 7.3, l'arbre de départ se trouve dans le panneau droit-haut. La liste des fichiers résultats énumérés dans les statistiques se trouve dans le panneau gauche-haut. Lorsqu'on choisit un fichier, son contenu est affiché dans le panneau gauche-milieu. Enfin lorsqu'une séquence est sélectionnée, son détail est affiché dans le panneau gauche-bas et l'arbre XML correspondant est dessiné dans le panneau droit-bas. La modification des coûts des opérations d'édition se fait via le menu *File* → *Options*.

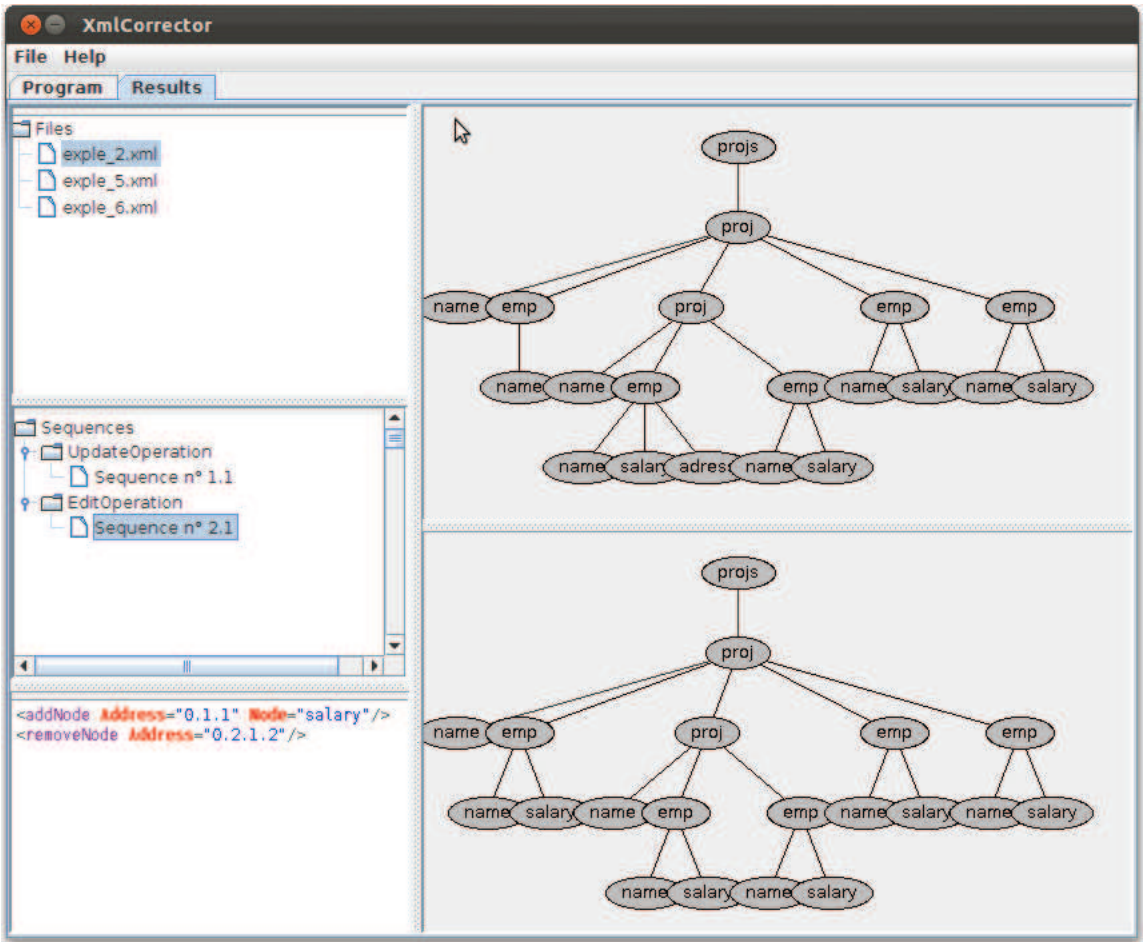


FIGURE 7.3 – *Contenu de l'onglet résultat.*

Améliorations futures. Pour améliorer *XMLCorrector*, nous comptons rajouter les fonctionnalités suivantes à l'application :

- pouvoir visualiser pas à pas les modifications d'une séquence d'opérations sur un arbre XML ;
- pouvoir prendre en compte les valeurs des feuilles dans l'arbre XML, et aussi les attributs ;
- l'ajout de nouvelles opérations d'éditions comme l'ajout et la suppression de nœuds internes dans l'arbre XML. Cette partie nécessite des travaux de recherche ;
- traiter d'autres formats de schéma XML comme XML Schema du W3C et aussi Relax-NG ;

- définir et utiliser des fonctions de coûts pour corriger un arbre XML à la place de valeurs affectées à chaque opérations d'éditions ;
- paralléliser l'algorithme de correction pour traiter efficacement de gros documents XML ;
- avoir une API pour permettre l'utilisation de l'algorithme de correction dans d'autres applications ;
- définir des séquences d'opérations interdites ou non souhaitées : l'algorithme de correction doit éviter d'utiliser ces séquences dans son processus. Cette fonctionnalité peut réduire fortement le nombres de solutions.

7.2 *DTDGrabber*

L'application *DTDGrabber* a été développée dans le but de mettre en commun les différentes contributions de ma thèse, et d'apporter des fonctionnalités intéressantes (comme la manipulation des XFDs) qui ne sont pas prises en compte par des SGBD pour XML. L'application *DTDGrabber* est en cours de développement, mais est bien avancée. L'application *DTDGrabber* est principalement développée par moi. Certaines parties ont été développées par des étudiants de l'IUT d'Orléans via des stages ou des projets que j'ai encadré (stagiaires : Cédric Viet et Joevin Vuillemet). *DTDGrabber* intègre aussi un module externe *xfd-val* [2] qui est un validateur de dépendances fonctionnelles XML, développé par Maria Adriana Lima dans sa thèse de Maria-Adriana Lima [82].

La Figure 7.4 illustre l'architecture de *DTDGrabber*. Les données stockées dans la base sont le schéma XML, un ensemble de dépendances fonctionnelles XML, et une collection de documents XML devant respecter les contraintes de structure imposées par le schéma et les contraintes d'intégrité imposées par les dépendances fonctionnelles. Lorsque de nouveaux documents XML sont insérés dans la base de données, le module *XML Parser* vérifie s'ils sont valides par rapport au schéma. Dans le cas où certains documents ne sont pas valides par rapport au schéma XML, ils sont fournis au module *XMLCorrector* qui propose des corrections pour les documents incorrects (parmi lesquelles l'utilisateur doit choisir) pour une insertion dans la base de données. Le module *XFDEditor* permet de créer de nouvelles dépendances fonctionnelles pour une base de données et le module *xfd-val* permet de vérifier les dépendances sur les documents XML de la base. Lorsqu'un document viole une dépendance, il y a un rapport d'erreurs qui est créé et l'utilisateur peut s'en servir pour corriger les valeurs erronées dans le document XML. Enfin les modules *MappingGen*, *XTraM*, et *XFDTools* (vus dans les chapitres précédents) permettent d'effectuer l'évolution conservatrice de plusieurs systèmes locaux en un seul système global en calculant les mappings entre les schémas locaux et le schéma global (*MappingGen*), en effectuant la traduction de documents entre le schéma global et les schémas locaux (*XTraM*) et en calculant l'ensemble des XFDs à vérifier par le système global sur tous les documents locaux (*XFDTools*). L'application *DTDGrabber* offre aussi la possibilité de pouvoir éditer un mapping entre deux schémas (pour faire évoluer un schéma vers un autre schéma) à l'aide du module de mapping et ensuite de pouvoir adapter les documents du schéma source vers le schéma cible à l'aide du module *XTraM*.

2. *xfd-val* est disponible à la page : `https://code.google.com/p/xdf-val/`

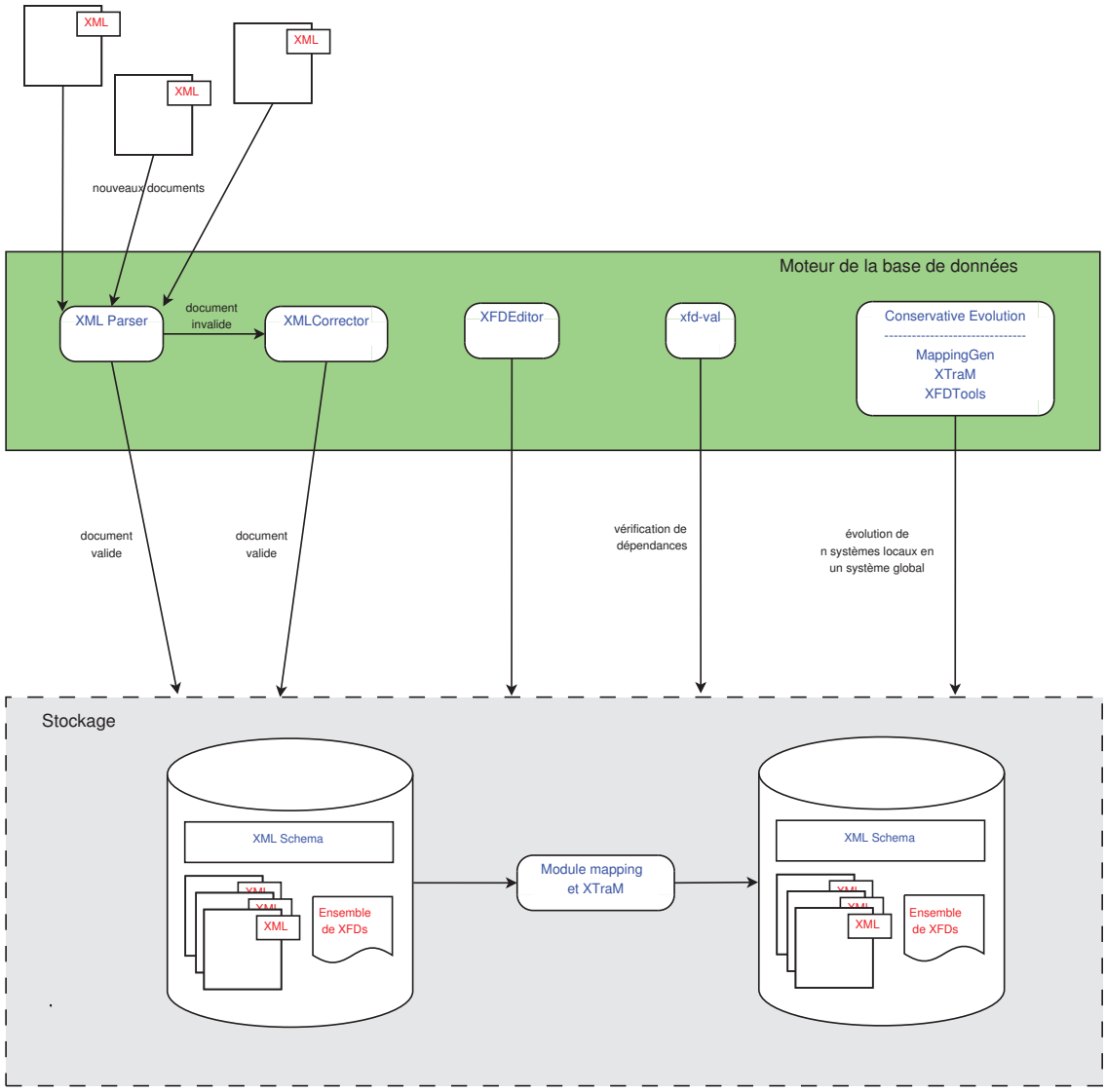


FIGURE 7.4 – *Architecture de l'application DTDGrabber.*

Une autre fonctionnalité de *DTDGrabber* consiste à pouvoir analyser les modifications de chaque opération d'édition du mapping sur un schéma donné.

La Figure 7.5 illustre la fenêtre principale de *DTDGrabber*. Une nouvelle base de données est créée à partir d'un schéma XML. Chaque onglet de la fenêtre principale concerne une base de donnée qui est ouverte. L'onglet est divisé en deux panneaux. Le panneau de gauche renseigne sur les dépendances fonctionnelles tandis que le panneau de droite renseigne sur les documents XML qui sont dans la base de données. Lorsqu'un nouveau document XML est ajouté à la base de données (après vérification de sa structure et correction si possible), le module *xfd-val* est lancé pour vérifier si le document XML viole les dépendances fonctionnelles de la base. Des rapports d'erreurs sont générés en cas de violation. Par exemple dans la Figure 7.5, le document "publication_tree_3" viole au moins une dépendance fonctionnelle car son statut vaut "erreur". Le rapport d'erreur concernant ce document est le suivant :

```
<ValidationLogs>
  <xml>data/publication_1/collection-xmldoc/publication_tree_3.xml</xml>
  <isValid>false</isValid>
```

```
  <logInfo>
    <xfd>publication_1_xfd_5 : (pub, ({paper/year,paper/conf,paper/authors[n]}
                                --> paper/title[n] ))</xfd>
    <log>[]: [2013, SAC, #] : r.0.2.0 HAS KEY [2013, SAC, #] : r.0.0.0</log>
  </logInfo>
</ValidationLogs>
```

Le rapport d'erreur nous dit que le document "publication_tree_3.xml" viole la dépendance "publication_1_xfd_5" à cause des tuples mentionnés ci-dessus qui possèdent les mêmes valeurs sur les chemins à gauche de la dépendance mais pour le chemin "paper/title" à droite de la dépendance le nœud concerné n'est pas le même. L'utilisateur peut par exemple corriger cette incohérence en ouvrant le document à partir de l'application pour le modifier.

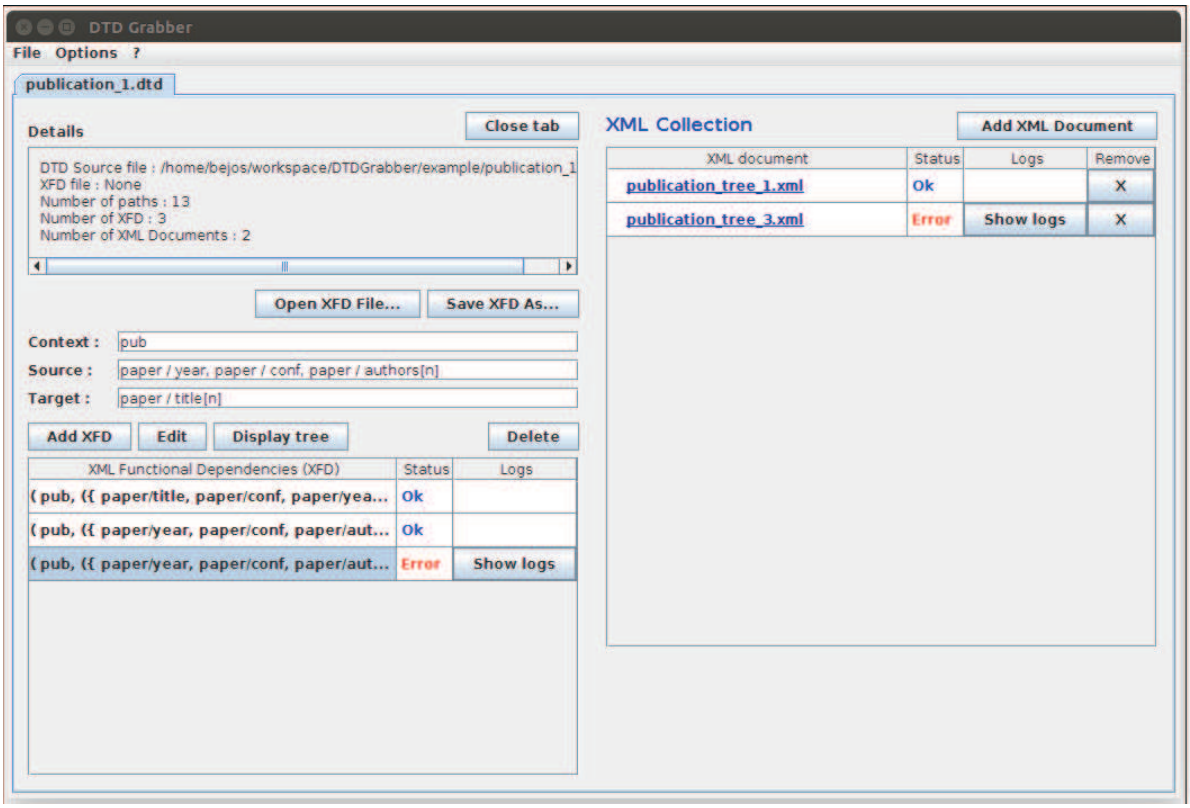


FIGURE 7.5 – *Fenêtre principale de DTDGrabber.*

De même le statut de la troisième dépendance "publication_1_xfd_5" ayant pour valeur "erreur", indique que la troisième dépendance est violée par au moins un document. Son rapport d'erreur va nous donner les documents qui la violent.
Le bouton "Display tree" permet d'afficher l'arbre représentant l'ensemble des chemins qui peuvent apparaître dans les documents XML de la base. Cet arbre qui est illustré dans la Figure 7.6, est utilisé pour remplir les champs "Context", "Source" et "Target" du panneau de gauche. Ces champs peuvent aussi être remplis manuellement. Ces trois champs permettent d'éditer (création ou modification) d'une dépendance fonctionnelle. Lorsqu'une dépendance fonctionnelle est éditée, le module *xfd-val* est lancé pour vérifier si les documents respectent la dépendance. Les champs statuts des documents sont alors actualisés en fonction des résultats obtenus.
La fenêtre qui permet de visualiser un mapping et les modifications apportées à un schéma, est montrée en Figure 7.7. Il est possible de naviguer parmi les opérations du mapping et

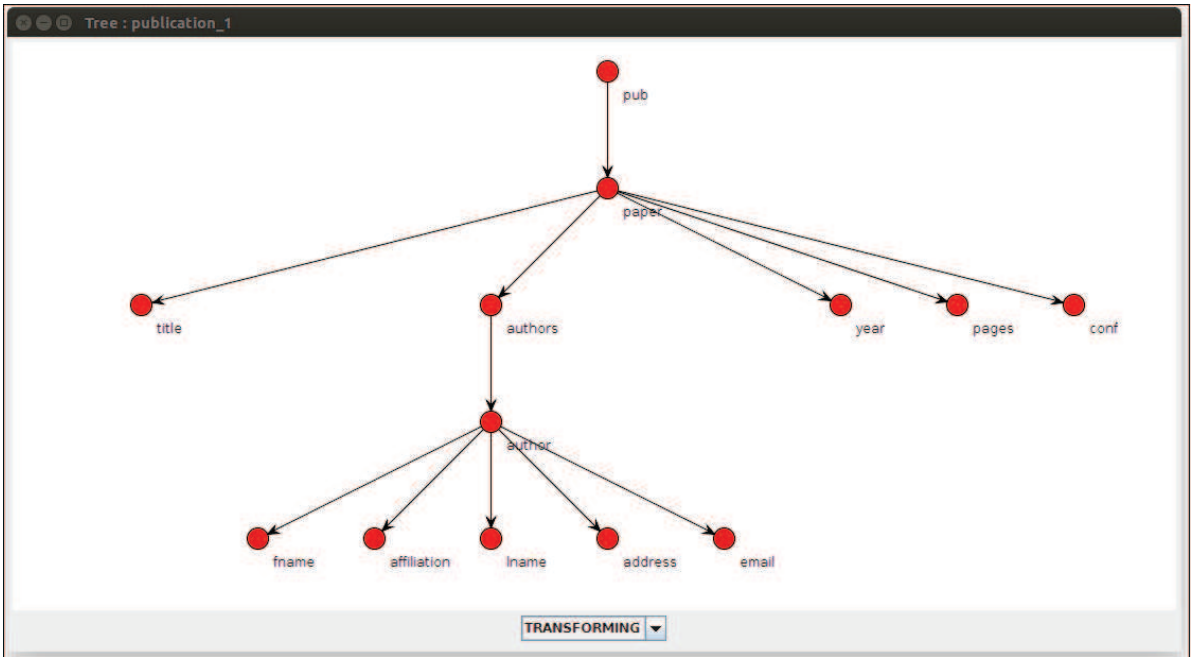


FIGURE 7.6 – *Arbre des chemins possibles pour les documents associés à un schéma XML.*

voir les différents changements sur le schéma en temps réel. C'est à partir de cette fenêtre que peut être créé et modifié un mapping de schéma.

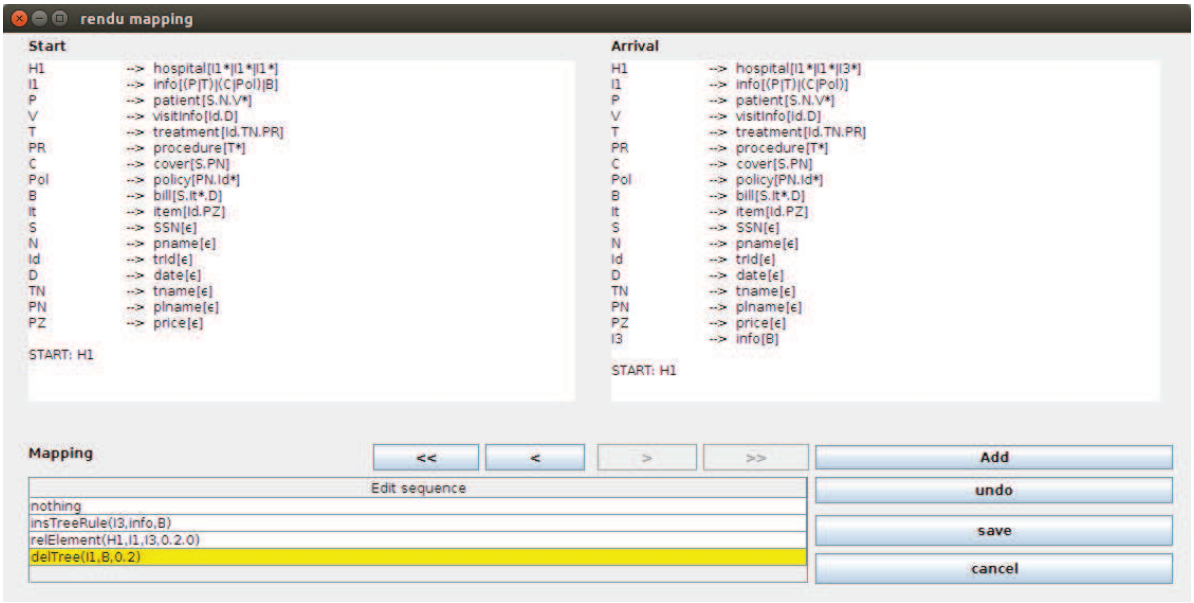


FIGURE 7.7 – *Visualisation des modifications apportées par un mapping sur un schéma XML.*

L'ajout d'une opération au mapping se fait via le bouton "Add" qui ouvre une nouvelle fenêtre pour effectuer cette action. La fenêtre concernée est illustrée dans la Figure 7.8. Pour créer par exemple l'opération $\texttt{del_tree}(I1, C|Pol, 0.1)$, l'on doit renseigner les champs du panneau gauche. A droite est affichée l'arbre représentant la règle de production (cf. Section 6.1 , Chapitre 6) qui est modifiée. Le bouton "validate" permet de revenir sur la fenêtre de la Figure 7.7 et ajoute l'opération créée au mapping courant.

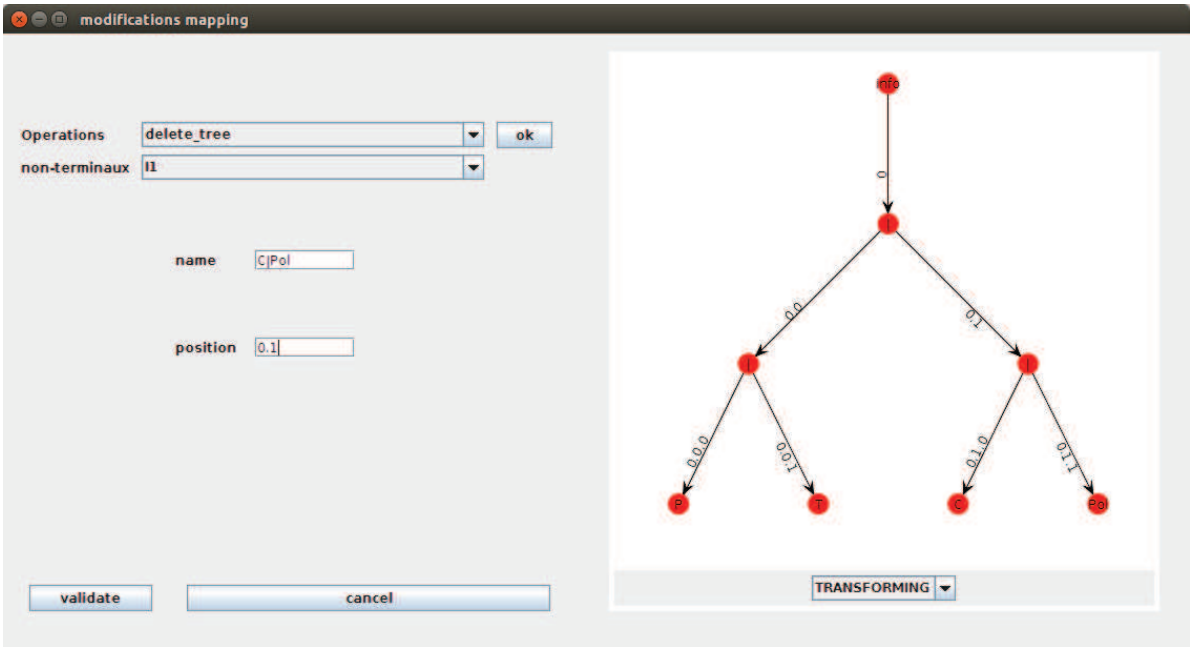


FIGURE 7.8 – *Fenêtre de création d'une opération d'édition.*

Améliorations futures. Nous pouvons citer les améliorations suivantes :

- l'ajout d'un nouveau document sans connaissance du schéma. Dans ce cas l'application *DTDGrabber* doit proposer, parmi les schémas existants dans la base, celui qui est le plus proche du nouveau document. Le module *XMLCorrector* est alors utilisé pour connaître la distance entre le nouveau document et chacun des schémas, et ensuite est choisi le schéma pour lequel la distance calculée est la plus petite ;
- l'ajout d'un moteur d'exécution de requête à l'application *DTDGrabber* pour en faire un véritable SGBD ;
- lors de l'évolution d'un schéma vers un autre, nous aimerions pouvoir aussi adapter les dépendances fonctionnelles en se servant du mapping ;
- l'ajout d'un module pour calculer automatiquement un mapping entre deux schémas donnés ;
- étendre les dépendances fonctionnelles aux dépendances d'inclusion.

Conclusion

8

Nous nous sommes intéressés dans cette thèse à proposer des techniques pour faire coexister des systèmes locaux (schéma XML + dépendances fonctionnelles + documents XML) avec un système global qui est une évolution conservatrice des systèmes locaux. Nous représentons les schémas XML par des grammaires régulières d'arbres d'arité non bornée (RTG) et les documents XML par des arbres d'arité non bornée. Dans un tel environnement, il est nécessaire de permettre l'échange de données entre le système global et les systèmes locaux d'où l'intérêt d'avoir un mapping entre les schémas. Les mappings de schémas jouent un rôle important dans les tâches de gestion de données comme l'intégration de données ou l'échange de données, et permettent d'assurer l'interopérabilité entre les systèmes. Nous spécifions un mapping par un script d'édition sur des grammaires d'arbres réguliers (RTG) dont les règles de production sont représentées comme des arbres d'arité non bornée. Pour faciliter le calcul de nouveaux mappings lors de l'évolution du schéma source ou du schéma cible, la composition et l'inversion de mapping ont été définies. L'utilisation de ses opérateurs évitent de recalculer un nouveau mapping entre les nouveaux schémas obtenus.

Dans [42], nous pouvons trouver l'algorithme *ExtSchemaGenerator* qui étend une RTG en une LTG. Cet algorithme est utilisé pour calculer le schéma global à partir des schémas locaux comme suit :

1. fusionner les règles de productions des schémas locaux pour obtenir une RTG ;
2. étendre la RTG issue de la fusion en une LTG par l'algorithme *ExtSchemaGenerator*.

Pour obtenir les mappings entre le schéma global et les schémas locaux, nous nous sommes inspirés des idées de [42]. Les différents changements opérés par *ExtSchemaGenerator* sur la RTG ont été facilement interprétés en des opérations d'éditions de grammaires. Ainsi nous obtenons bien les mappings désirés.

Aussi il est important de calculer les XFD du système global à partir de celles des systèmes locaux. Pour cela nous proposons une méthode qui évite la vérification des XFD sur tous les documents, en travaillant sur les ensembles de XFD locaux. Les XFD utilisées (définies dans [33, 66, 82]) ont la particularité :

- de posséder un contexte pour définir des dépendances fonctionnelles qui seront vérifiées localement dans des sous-arbres ;
- de posséder deux types d'égalités : l'égalité par valeur et l'identité de nœud ;
- d'utiliser le langage de chemins PL qui permet de décrire un ensemble de chemins simples.

Pour pouvoir calculer les XFD du système global constituant le plus grand ensemble de XFD qui doit être satisfait par tous les documents locaux, nous proposons un système d'axiomes pour nos XFD. Ce système d'axiomes va nous permettre de dériver d'autres dépendances

à partir d'un ensemble de XFD. Le système d'axiomes est prouvé correct et complet. Notre méthode pour calculer les XFD du système global consiste en un filtre des XFD locaux. Lorsqu'une XFD f peut être violée par un document local, nous l'écartons en prenant le soin de pouvoir garder une dépendance f' qui est satisfaite par tous les documents et qui nécessite f pour sa dérivation. Ainsi nous obtenons l'ensemble des dépendances pouvant dériver toutes les dépendances qui sont satisfaites par tous les documents locaux.

Lorsque le remplacement d'un système par un autre est nécessaire, nous proposons aussi une méthode pour comparer les schémas de façon relâchée. Le nouveau système devra traiter les documents XML de l'ancien système. L'inclusion relâchée assure au nouveau schéma de contenir toutes les informations se trouvant dans l'ancien schéma et offre la possibilité au nouveau schéma d'avoir une structure différente de l'ancien tout en respectant la relation ancêtre-descendant et gauche-droite entre les nœuds. Notre contribution est une méthode pour calculer la grammaire $WI(G')$ qui génère l'ensemble de tous les arbres inclus-relâchés dans les arbres générés par la grammaire G'. Cette méthode traite le cas des grammaires récursives en séparant les non-terminaux de la grammaire en trois types de récursivité : 2-rec, 1-rec et 0-rec. En se servant de $WI(G')$, nous pouvons décider si une grammaire G est inclus-relâchée dans G' en vérifiant que $L(G) \subseteq L(WI(G'))$. A partir de la méthode pour calculer $WI(G')$, nous proposons une méthode pour calculer le mapping entre G' et $WI(G')$.

A partir des contributions décrites ci-dessus, nous savons calculer le mapping entre schémas dans les deux cas suivants :

1. entre des schémas locaux et leur évolution conservatrice en un schéma global ;
2. entre une grammaire RTG G' et la grammaire $WI(G')$ de ses sous-arbres relâchés.

Pour assurer l'échange de données entre ces schémas, nous proposons une méthode pour adapter les documents d'un schéma source vers un schéma cible en se servant des informations contenues dans le mapping. L'intuition de cette méthode est la suivante :

- lorsqu'on ajoute un nouveau non-terminal A dans une règle de production, il faudra ajouter un nouveau sous-arbre bien formé par rapport A dans notre document XML ;
- lorsqu'on supprime un non-terminal A d'une règle de production, il faudra supprimer les sous-arbres générés par A dans le document XML ;
- lorsqu'on renomme un non-terminal A par un autre non-terminal B, il faudra corriger les sous-arbres générés par A en des sous-arbres générés par B.

La correction d'un document XML par rapport à un schéma XML est aussi une de nos contributions. Dans cette dernière contribution nous avons réalisé des tests sur des données réelles axés sur l'influence de différents paramètres, qui montrent un comportement polynomial de notre algorithme alors que la complexité théorique de notre algorithme est exponentielle. La principale contribution de notre approche est d'offrir une étude approfondie du problème de correction d'arbre vers un langage d'arbres. Dans notre approche, nous ne mesurons pas seulement la distance entre le document et le schéma mais aussi trouvons les arbres candidats à la correction. Nous ne nous limitons pas seulement à la recherche des solutions minimales mais aussi nous trouvons toutes les solutions à une distance inférieure à un seuil donné. Ainsi, nous considérons la correction comme un problème d'énumération plutôt qu'un problème de décision contrairement à la plupart des autres approches.

Pour résumer, nos travaux ont conduit à 1 publication dans une revue internationale, 4 publications dans des conférences internationales, et 1 publications dans des conférences nationales. Nous avons aussi développé un logiciel réunissant ces différentes contributions

pour faire évoluer les schémas XML et leur documents XML associés ainsi que leurs dépendances fonctionnelles. Ce logiciel sera prochainenement mis en ligne.

Perspectives.

Pour nos travaux sur les dépendances fonctionnelles, nous envisageons d'étendre notre méthode aux dépendances d'inclusion. Aussi lorsque le schéma évolue, nous souhaitons calculer l'ensemble des dépendances du nouveau schéma de façon incrémentale en se servant aussi du mapping entre le schéma source et le schéma cible. Il serait aussi intéressant d'avoir un validateur de XFD sur les documents locaux en adaptant la méthode proposée dans [33] dans une approche map-reduce. L'ensemble de dépendances du système global sera l'ensemble à vérifier sur les documents locaux. Une version map-reduce de la vérification de XFD pourrait être intégrée dans des plateformes qui gèrent de gros volumes de données web comme AMADA [13] (qui stocke des documents XML et des graphes RDF).

Pour compléter notre outil, il serait intéressant de pouvoir aborder la partie sémantique car actuellement nous ne traitons pas les valeurs et les attributs lors du processus d'adaptation des documents XML. L'utilisation des dépendances fonctionnelles serait probablement nécessaire pour atteindre ce but. L'adaptation des dépendances fonctionnelles est aussi nécessaire lors d'une évolution non conservatrice du système global par exemple. Les travaux sur l'adaptation de dépendances fonctionnelles peuvent rejoindre les travaux sur la réécriture des requêtes sur les documents en présence d'un mapping [23, 76] car les requêtes et les dépendances fonctionnelles manipulent des chemins dans les documents XML.

Un autre défi très important serait de pouvoir calculer automatiquement un mapping entre deux schémas donnés. Pour le moment le mapping peut être défini par l'utilisateur ou généré automatiquement pour une utilisation spécifique (évolution conservatrice de schéma ou grammaire des arbres inclus relâchés). Nous pouvons remarquer que les règles de production sont représentées par des arbres dans lesquelles les opérateurs sont des nœuds internes et les non-terminaux des feuilles. Nos opérations d'édition des règles de production concernant les opérateurs $(|, ., *)$ modifient les nœuds internes de l'arbre qui représente une règle de production. Pour pouvoir calculer automatiquement un mapping entre deux schémas, une première piste serait d'avoir une méthode pour transformer un arbre en un autre arbre en autorisant les opérations d'ajout et de suppression de nœuds internes. Ensuite il faudra adapter cette méthode en autorisant uniquement des opérations facilement traduisibles en opérations d'édition pour l'adaptation des documents XML. Par exemple le fait de renommer un opérateur en un non-terminal ne peut pas être traduit lorsqu'on passe à l'adaptation des documents. Les opérations de ce type ne doivent pas être utilisées dans le mapping de schémas. Ces opérations d'édition agissant sur les nœuds internes d'un document XML, sont utilisées dans la correction de document vers un schéma dans [105, 108, 113]. L'idée ici serait d'adapter ces méthodes en autorisant seulement l'utilisation d'opérations spécifiques. Ainsi il ne sera pas possible de modifier un nœud en n'importe quel nœud. Un problème qui peut se poser en suivant cette démarche est le suivant : est-il toujours possible d'avoir un script d'édition entre deux schémas en appliquant les méthodes de [105, 108, 113] avec des opérations spécifiques ? Dans [107], les auteurs se sont intéressés au problème d'inclusion d'une DTD D_1 dans une autre DTD D_2 en cherchant un script d'édition entre les deux DTD composés uniquement d'opérations qui permettent

d'agrandir le schéma D_1 (ces opérations sont du même type que celles du Lemme 6.1). La méthode n'aboutit pas toujours. Si un tel script est trouvé alors on peut affirmer que D_1 est inclus dans D_2. Sinon on ne peut rien conclure.

Bibliographie

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley Publishing Company, 1995.
Cité page 50.

[2] S. Abiteboul, L. Segoufin, and V. Vianu. Representing and querying XML with incomplete information. *ACM Trans. Database Syst.*, 31(1) :208–254, Mar. 2006.
Cité page 47.

[3] A. Algergawy, R. Nayak, and G. Saake. Element similarity measures in XML schema matching. *Information Sciences*, 180(24) :4975–4998, Dec. 2010.
Cité pages 64 et 157.

[4] S. Amano, L. Libkin, and F. Murlak. XML schema mappings. In *Proceedings of the 28th ACM Symposium on Principles of Database Systems*, PODS '09, pages 33–42, New York, NY, USA, 2009. ACM.
Cité pages 134, 157 et 158.

[5] J. Amavi. Comparaison des langages d'arbres pour la substitution de services web (in french). Technical Report RR-2010-13, LIFO, Université d'Orléans, 2010.
Cité page 32.

[6] J. Amavi, B. Bouchou, and A. Savary. On correcting XML documents with respect to a schema. *The Computer Journal*, 56(4), 2013.
Cité pages 6 et 88.

[7] J. Amavi, B. Bouchou, and A. Savary. On correcting XML documents with respect to a schema. *The Computer Journal*, 57(5) :639–674, 2014.
Cité pages 88, 114, 115, 124 et 127.

[8] J. Amavi, J. Chabin, M. Halfeld Ferrari, and P. Réty. Weak inclusion for xml types. In *Proceedings of the 16th International Conference on Implementation and Application of Automata*, volume 6807 of *CIAA '11*, pages 30–41. Springer Berlin Heidelberg, Blois, France, 2011.
Cité pages 4, 20, 26 et 32.

[9] J. Amavi, J. Chabin, M. Halfeld Ferrari, and P. Réty. A toolbox for conservative XML schema evolution and document adaptation. In *25th International Conference on Database and Expert Systems Applications, DEXA*, Lecture Notes in Computer Science. Springer, 2014.
Cité page 3.

[10] J. Amavi, J. Chabin, and P. Réty. Weak inclusion for recursive xml types. In *Proceedings of the 17th International Conference on Implementation and Application of Automata*, volume 7381 of *CIAA '12*, pages 78–89. Springer Berlin Heidelberg, Porto, Portugal, 2012.
Cité pages 3 et 4.

[11] J. Amavi and M. Halfeld Ferrari. Filtering XFD toward interoperability. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, pages 868–871, New

York, NY, USA, 2013. ACM.
Cité page 5.

[12] J. Amavi and M. Halfeld Ferrari. Maximal Set of XML Functional Dependencies over Multiple Systems. In *BDA*, Nantes, France, Oct. 2013.
Cité page 5.

[13] A. Aranda-Andújar, F. Bugiotti, J. Camacho-Rodríguez, D. Colazzo, F. Goasdoué, Z. Kaoudi, and I. Manolescu. AMADA : Web data repositories in the amazon cloud. In *Proceedings of the 21st ACM International Conference on Information and Knowledge Management*, CIKM '12, pages 2749–2751, New York, NY, USA, 2012. ACM.
Cité page 171.

[14] M. Arenas and L. Libkin. A normal form for XML documents. In *Proceedings of the 21th ACM symposium on Principles of database systems*, PODS '02, pages 85–96, New York, NY, USA, 2002. ACM.
Cité page 57.

[15] M. Arenas and L. Libkin. A normal form for XML documents. *ACM Transactions on Database Systems (TODS)*, 29 No.1 :195–232, 2004.
Cité pages 47, 48 et 84.

[16] M. Arenas and L. Libkin. XML data exchange : Consistency and query answering. In *Proceedings of the 24th ACM Symposium on Principles of Database Systems*, PODS '05, pages 13–24, New York, NY, USA, 2005. ACM.
Cité pages 134, 157 et 158.

[17] P. Atzeni and N. M. Morfuni. Functional dependencies and constraints on null values in database relations. *Information and Control*, 70(1) :1 – 31, 1986.
Cité page 47.

[18] Z. Bellahsene and F. Duchateau. Tuning for schema matching. In Z. Bellahsene, A. Bonifati, and E. Rahm, editors, *Schema Matching and Mapping*, Data-Centric Systems and Applications, pages 293–316. Springer Berlin Heidelberg, 2011.
Cité page 157.

[19] E. Bertino, G. Guerrini, and M. Mesiti. A Matching algorithm for measuring the structural similarity between an XML documents and a DTD and its applications. *Information Systems*, 29 :23–46, 2004.
Cité pages 88, 125, 221 et 222.

[20] E. Bertino, G. Guerrini, and M. Mesiti. Measuring the structural similarity among XML documents and DTDs. *Journal of Intelligent Information Systems*, 30 :55–92, 2008.
Cité pages 88, 125, 221 et 222.

[21] P. Bille. A survey on tree edit distance and related problems. *Theoretical Computer Science*, 337 :217–239, 2005.
Cité page 116.

[22] P. Bille and I. L. Gørtz. The tree inclusion problem : In optimal space and faster. In *Automata, Languages and Programming, 32nd ICALP*, pages 66–77, 2005.
Cité pages 17 et 31.

[23] A. Bonifati, E. Chang, T. Ho, L. V. Lakshmanan, R. Pottinger, and Y. Chung. Schema mapping and query translation in heterogeneous P2P XML databases. *The VLDB Journal*, 19(2) :231–256, Apr. 2010.
Cité pages 131, 160 et 171.

[24] A. Bonifati, G. Mecca, P. Papotti, and Y. Velegrakis. Discovery and correctness of schema mapping transformations. In Z. Bellahsene, A. Bonifati, and E. Rahm, editors, *Schema Matching and Mapping*, Data-Centric Systems and Applications, pages 111–147. Springer Berlin Heidelberg, 2011.
Cité pages 131, 157 et 160.

[25] U. Boobna and M. de Rougemont. Correctors for XML Data. In *Proceedings of XSym 04, Toronto, Canada*, volume 3186 of *Lecture Notes in Computer Science*, pages 97–111. Springer, 29–30 August 2004.
Cité pages 127, 221 et 222.

[26] A. Bouajjani, P. Habermehl, L. Holík, T. Touili, and T. Vojnar. Antichain-based universality and inclusion testing over nondeterministic finite tree automata. In *Int. Conf. on Implementation and Applications of Automata, CIAA*, pages 57–67. Springer, 2008.
Cité pages 20 et 32.

[27] B. Bouchou, A. Cheriat, M. Halfeld Ferrari, D. Laurent, M.-A. Lima, and M. Musicante. Efficient constraint validation for updated XML databases. *Informatica*, 31(3) :285–310, 2007.
Cité pages 39, 40 et 85.

[28] B. Bouchou, A. Cheriat, M. Halfeld Ferrari, and A. Savary. Integrating Correction into Incremental Validation. In *Proceeding of BDA 06, Lille, France*, 17–20 October 2006.
Cité pages 6, 87, 88, 125, 127, 129 et 158.

[29] B. Bouchou, A. Cheriat, M. Halfeld Ferrari, and A. Savary. XML Document Correction : Incremental Approach Activated by Schema Validation. In *Proceedings of IDEAS 06, Delhi, India*, pages 228–238. IEEE Computer Society, 11–14 December 2006.
Cité pages 6, 87, 88, 129 et 158.

[30] B. Bouchou and D. Duarte. Assisting XML Schema Evolution that Preserves Validity. In *Proceedings of SBBD 07, João Pessoa, Paraíba, Brasil*, pages 270–284. SBC, 15–19 October 2007.
Cité pages 88 et 129.

[31] B. Bouchou, D. Duarte, M. Halfeld Ferrari, D. Laurent, and M. A. Musicante. Schema Evolution for XML : A Consistency-Preserving Approach. In *Proceedings of MFCS 04, Prague, Czech Republic*, volume 3153 of *Lecture Notes in Computer Science*, pages 876–888. Springer, 22–27 August 2004.
Cité pages 88 et 129.

[32] B. Bouchou, M. Halfeld Ferrari, and M. Lima. Contraintes d'intégrité pour XML. visite guidée par une syntaxe homogène. *Technique et Science Informatiques*, 28(3) :331–364, 2009.
Cité pages 44, 48 et 50.

[33] B. Bouchou, M. Halfeld Ferrari, and M.-A. V. Lima. A grammarware for the incremental validation of integrity constraints on XML documents under multiple updates. *Transactions on Large-Scale Data and Knowledge-Centered Systems, TLDKS Journal*, 6(LNCS 7600), 2012.
Cité pages 5, 36, 37, 39, 40, 48, 85, 169 et 171.

[34] L. Boytsov. Indexing methods for approximate dictionary searching : Comparative analysis. *ACM Journal of Experimental Algorithmics*, 16(1), 2011.
Cité pages 99, 115, 116 et 123.

[35] A. Brüggeman-Klein and D. Wood. One-unambiguous regular languages. *Information and Computation*, 142(2) :182–206, 1998.
Cité pages 14 et 15.

[36] P. Buneman, S. Davidson, W. Fan, C. Hara, and W. C. Tan. Keys for XML. In *WWW10, May 2-5*, 2001.
Cité page 40.

[37] P. Buneman, S. B. Davidson, W. Fan, C. S. Hara, and W. C. Tan. Reasoning about keys for XML. *Inf. Syst.*, 28(8) :1037–1063, 2003.
Cité page 85.

[38] P. Caron and D. Ziadi. Characterization of Glushkov automata. *Theor. Comput. Sci. (TCS)*, 233(1-2) :75–90, 2000.
Cité page 14.

[39] E. Castanier, R. Coletta, P. Valduriez, and C. Frisch. Public data integration with websmatch. In *BDA 2012*, 2012.
Cité page 36.

[40] F. Cavalieri, G. Guerrini, and M. Mesiti. Updating XML schemas and associated documents through Exup. In *Proceedings of the IEEE 27th International Conference on Data Engineering*, ICDE '11, pages 1320–1323, Washington, DC, USA, 2011. IEEE Computer Society.
Cité pages 158 et 159.

[41] F. Cecchin, C. D. de Aguiar Ciferri, and C. S. Hara. XML data fusion. In *DaWak*, pages 297–308, 2010.
Cité page 85.

[42] J. Chabin, M. Halfeld Ferrari Alves, M. A. Musicante, and P. Réty. Conservative type extensions for XML data. *T. Large-Scale Data- and Knowledge-Centered Systems, TLDKS Journal*, 9(LNCS 7980) :65–94, 2013.
Cité pages xi, 3, 12, 36, 86, 131, 144, 145, 146, 147, 148, 160 et 169.

[43] J. Champavère, R. Guilleron, A. Lemay, and J. Niehren. Efficient inclusion checking for deterministic tree automata and DTDs. In *Int. Conf. Language and Automata Theory and Applications*, volume 5196 of *LNCS*, pages 184–195. Springer, 2008.
Cité pages 20 et 32.

[44] Y. Chen and Y. Chen. A new tree inclusion algorithm. *Information Processing Letters*, 98(6) :253 – 262, 2006.
Cité pages 17 et 31.

[45] Y. Chen, Y. Shi, and Y. Chen. Tree inclusion algorithm, signatures and evaluation of path-oriented queries. In *Symp. on Applied Computing*, pages 1020–1025, 2006.
Cité pages 17 et 31.

[46] A. Cheriat. *Une méthode de correction de la structure de documents XML dans le cadre d'une validation incrémentale*. PhD thesis, LI, Université François Rabelais de Tours, 2006.
Cité pages 3, 6, 87 et 158.

[47] R. Chirkova, L. Libkin, and J. L. Reutter. Tractable XML data exchange via relations. In *Proceedings of the 20th ACM Conference on Information and Knowledge Management, CIKM 2011*, pages 1629–1638, 2011.
Cité page 85.

[48] J. Clark and S. DeRose. XML path language (XPath) - version 1.0. Available at http ://www.w3.org/TR/xpath, 1999.
Cité page 40.

[49] E. F. Codd. Extending the database relational model to capture more meaning. *ACM Trans. Database Syst.*, 4(4) :397–434, Dec. 1979.
Cité page 47.

[50] D. Colazzo, G. Ghelli, L. Pardini, and C. Sartiani. Linear inclusion for XML regular expression types. In *Proceedings of the 18th ACM Conference on Information and Knowledge Management, CIKM*, pages 137–146. ACM Digital Library, 2009.
Cité pages 20 et 32.

[51] D. Colazzo, G. Ghelli, and C. Sartiani. Efficient asymmetric inclusion between regular expression types. In *Proceeding of International Conference of Database Theory, ICDT*, pages 174–182. ACM Digital Library, 2009.
Cité pages 20 et 32.

[52] 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.
Cité page 32.

[53] G. Crane. What do you do with a million books ? *D-Lib Magazine*, 12(3), March 2006.
Cité page 36.

[54] D. T. Barnard, G. Clarke and N. Duncan. Tree-to-tree Correction for Document Trees. Technical Report 95-372, Department of Computing and Information Science, Queen's University, Kingston, Ontario, 1995.
Cité page 116.

[55] H.-H. Do and E. Rahm. COMA : a system for flexible combination of schema matching approaches. In *Proceedings of the 28th International Conference on Very Large Data Bases*, VLDB '02, pages 610–621. VLDB Endowment, 2002.
Cité page 157.

[56] H.-H. Do and E. Rahm. Matching large schemas : Approaches and evaluation. *Information Systems*, 32(6) :857–885, Sept. 2007.
Cité page 157.

[57] M. W. Du. and S. C. Chang. A model and a fast algorithm for multiple errors spelling correction. *Acta Informatica*, 29 :281–302, 1992.
Cité pages 99 et 100.

[58] F. Duchateau, R. Coletta, Z. Bellahsene, and R. J. Miller. (Not) Yet Another Matcher. In *Proceedings of the 18th ACM Conference on Information and Knowledge Management*, CIKM '09, pages 1537–1540, New York, NY, USA, 2009. ACM.
Cité page 157.

[59] F. Duchateau, R. Coletta, Z. Bellahsene, and R. J. Miller. YAM : a schema matcher factory. In *Proceedings of the 18th ACM Conference on Information and Knowledge Management*, CIKM '09, pages 2079–2080, New York, NY, USA, 2009. ACM.
Cité page 157.

[60] R. Fagin, L. Haas, M. Hernández, R. J. Miller, L. Popa, and Y. Velegrakis. Clio : Schema mapping creation and data exchange. In A. Borgida, V. Chaudhri, P. Giorgini, and E. Yu, editors, *Conceptual Modeling : Foundations and Applications*, volume 5600 of *Lecture Notes in Computer Science*, pages 198–236. Springer Berlin Heidelberg, 2009.
Cité pages 131, 157, 158 et 160.

[61] R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. Data exchange : semantics and query answering. *Theoretical Computer Science*, 336(1) :89 – 124, 2005. Database Theory.
Cité pages 134 et 157.

[62] R. Fagin, P. G. Kolaitis, L. Popa, and W. C. Tan. Schema mapping evolution through composition and inversion. In Z. Bellahsene, A. Bonifati, and E. Rahm, editors, *Schema*

Matching and Mapping, pages 191–222. Springer, 2011.
Cité pages 142 et 157.

[63] F. Fassetti and B. Fazzinga. Approximate functional dependencies for XML data. In *ADBIS Research Communications*, 2007.
Cité page 57.

[64] F. Gire and H. Idabal. Regular tree patterns : a uniform formalism for update queries and functional dependencies in XML. In *EDBT/ICDT Workshops*, 2010.
Cité page 85.

[65] G. Guerrini, M. Mesiti, and M. Sorrenti. XML Schema Evolution : Incremental Validation and Efficient Document Adaptation. In *Proceedings of XSym 07, Vienna, Austria*, volume 4704, pages 92–106. Springer, 23–24 September 2007.
Cité pages 88 et 129.

[66] M. Halfeld Ferrari Alves. *Les aspects dynamiques de XML spécification des interfaces de services web avec PEWS*. Habilitation à diriger des recherches, Université François Rabelais de Tours, 2007.
Cité pages 5, 37, 48, 50, 85 et 169.

[67] S. Hartmann and T. Trinh. Axiomatising functional dependencies for XML with frequencies. In *Proceedings of the 4th international conference on Foundations of Information and Knowledge Systems*, FoIKS'06, pages 159–178, Berlin, Heidelberg, 2006. Springer-Verlag.
Cité pages 47, 48 et 84.

[68] M. Hartung, J. Terwilliger, and E. Rahm. Recent advances in schema and ontology evolution. In Z. Bellahsene, A. Bonifati, and E. Rahm, editors, *Schema Matching and Mapping*, Data-Centric Systems and Applications, pages 149–190. Springer Berlin Heidelberg, 2011.
Cité page 159.

[69] Q. He and T. W. Ling. Extending and inferring functional dependencies in schema transformation. In *Proceedings of the 2004 ACM CIKM International Conference on Information and Knowledge Management*, pages 12–21, 2004.
Cité page 86.

[70] K. Horie and N. Suzuki. Extracting differences between regular tree grammars. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, pages 859–864, New York, NY, USA, 2013. ACM.
Cité pages 158 et 159.

[71] R. Hull and M. Yoshikawa. ILOG : declarative creation and manipulation of object identifiers. In *Proceedings of the Sixteenth International Conference on Very Large Databases*, pages 455–468, San Francisco, CA, USA, 1990. Morgan Kaufmann Publishers Inc.
Cité page 157.

[72] T. Imieliński. Incomplete information in logical databases. *IEEE Data Eng. Bull.*, 12(2) :29–40, 1989.
Cité page 47.

[73] T. Imieliński and W. Lipski. Incomplete information in relational databases. *J. ACM*, 31(4) :761–791, Sept. 1984.
Cité page 47.

[74] P. Kilpeläinen and H. Mannila. Ordered and unordered tree inclusion. *SIAM J. Comput.*, 24(2) :340–356, 1995.
Cité pages 17 et 31.

[75] D. E. Knuth. *Fundamental Algorithms, The Art of Computer Programming*, volume 1, 2nd ed. Addison-Wesley, 1973.
Cité page 10.

[76] G. Koloniari and E. Pitoura. Content-based routing of path queries in peer-to-peer systems. In E. Bertino, S. Christodoulakis, D. Plexousakis, V. Christophides, M. Koubarakis, K. Böhm, and E. Ferrari, editors, *Advances in Database Technology - EDBT 2004*, volume 2992 of *Lecture Notes in Computer Science*, pages 29–47. Springer Berlin Heidelberg, 2004.
Cité pages 131, 160 et 171.

[77] L. Kot and W. M. White. Characterization of the interaction of XML functional dependencies with DTDs. In *ICDT- 11th International Conference on Database Theory*, pages 119–133, 2007.
Cité page 84.

[78] M.-L. Lee, T. W. Ling, and W. L. Low. Designing functional dependencies for XML. In *Extending Database Technology (EDBT)*, pages 124–141, 2002.
Cité page 84.

[79] M. L. Lee, L. H. Yang, W. Hsu, and X. Yang. Xclust : Clustering XML schemas for effective integration. In *Proceedings of the 11th International Conference on Information and Knowledge Management*, CIKM '02, pages 292–299, New York, USA, 2002. ACM.
Cité pages 64 et 157.

[80] E. Leonardi, T. T. Hoai, S. S. Bhowmick, and S. K. Madria. Dtd-diff : A change detection algorithm for dtds. *Data Knowledge Engineering*, 61(2) :384–402, 2007.
Cité page 158.

[81] M. Levene and G. Loizou. *A guided tour of relational databases and beyond*. Springer-Verlag, 1999.
Cité page 47.

[82] M.-A. Lima. *Contraintes d'intégrité en XML*. PhD thesis, LI, Université François Rabelais de Tours, 2007.
Cité pages 5, 37, 48, 50, 85, 164 et 169.

[83] W. Lipski. On databases with incomplete information. *J. ACM*, 28(1) :41–70, Jan. 1981.
Cité page 47.

[84] J. Madhavan, P. A. Bernstein, and E. Rahm. Generic schema matching with Cupid. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, pages 49–58, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.
Cité page 157.

[85] D. Maier, A. O. Mendelzon, and Y. Sagiv. Testing implications of data dependencies. *ACM Transactions on Database Systems (TODS)*, 4(4) :455–469, 1979.
Cité pages 131, 157, 158 et 160.

[86] M. Mani and D. Lee. XML to Relational Conversion using Theory of Regular Tree Grammars. In *VLDB Workshop on EEXTT*, pages 81–103. Springer, 2002.
Cité page 12.

[87] W. Martens, F. Neven, and T. Schwentick. Complexity of decision problems for simple regular expressions. In *Int. Symp. Mathematical Foundations of Computer Science, MFCS*, pages 889–900, 2004.
Cité page 32.

[88] P. Mork, L. Seligman, A. Rosenthal, J. Korb, and C. Wolf. The harmony integration workbench. In S. Spaccapietra, J. Z. Pan, P. Thiran, T. Halpin, S. Staab, V. Svatek, P. Shvaiko, and J. Roddick, editors, *Journal on Data Semantics XI*, pages 65–93. Springer-Verlag, Berlin, Heidelberg, 2008.
Cité page 157.

[89] M. Murata, D. Lee, M. Mani, and K. Kawaguchi. Taxonomy of XML schema languages using formal language theory. *ACM Trans. Inter. Tech.*, 5(4) :660–704, 2005.
Cité pages 12, 13, 14 et 115.

[90] F. Neven. Automata theory for XML researchers. *Sigmod Record*, 31(3), 2002.
Cité page 32.

[91] T. Nösinger, M. Klettke, and A. Heuer. XML schema transformations (the ELaX approach). In *Proceedings of the 24th International Conference on Database and Expert Systems Applications*, volume 8055 of *DEXA '13*, pages 293–302. Springer Berlin Heidelberg, 2013.
Cité pages 158 et 159.

[92] K. Oflazer. Error-tolerant Finite-state Recognition with Applications to Morphological Analysis and Spelling Correction. *Computational Linguistics*, 22(1) :73–89, 1996.
Cité pages 97, 99, 100, 101 et 130.

[93] T. Pankowski. Reconciling inconsistent data in probabilistic XML data integration. In *Sharing Data, Information and Knowledge, 25th British National Conference on Databases - BNCOD*, pages 75–86, 2008.
Cité page 85.

[94] M. Raghavachari and O. Shmueli. Efficient Schema-Based Revalidation of XML. In *Proceedings of EDBT 04, Heraklion, Crete, Greece*, pages 639–657, 14–18 March 2004.
Cité page 129.

[95] M. Raghavachari and O. Shmueli. Efficient Revalidation of XML Documents. *IEEE Trans. Knowl. Data Eng.*, 19(4) :554–567, 2007.
Cité page 129.

[96] E. Rahm. Towards large-scale schema and ontology matching. In Z. Bellahsene, A. Bonifati, and E. Rahm, editors, *Schema Matching and Mapping*, Data-Centric Systems and Applications, pages 3–27. Springer Berlin Heidelberg, 2011.
Cité page 157.

[97] P. Réty, J. Chabin, and J. Chen. Synchronized ContextFree Tree-tuple Languages. Technical Report RR-2006-13, http ://www.univ-orleans.fr/-lifo/prodsci/rapports/RR/RR2006/RR-2006-13.ps.gz, LIFO, 2006.
Cité page 32.

[98] T. Richter. A new algorithm for the ordered tree inclusion problem. In A. Apostolico and J. Hein, editors, *Combinatorial Pattern Matching*, volume 1264 of *LNCS*, pages 150–166. Springer, 1997.
Cité pages 17, 31 et 32.

[99] A. Savary, J. Waszczuk, and A. Przepiórkowski. Towards the Annotation of Named Entities in the Polish National Corpus. In *Proceedings of LREC 10, Valletta, Malta*. European Language Resources Association, 17-23 May 2010.
Cité page 117.

[100] S. M. Selkow. The Tree-to-Tree Editing Problem. *Information Processing Letters*, 6(6) :184–186, 1977.
Cité pages 96, 97, 100, 101, 130 et 218.

[101] M. S. Shahriar and J. Liu. Preserving functional dependency in XML data transformation. In *ADBIS '08 : Proceedings of the 12th East European conference on Advances in Databases and Information Systems*, pages 262–278. Springer-Verlag, 2008.
Cité pages 84 et 86.

[102] M. Shoaran and A. Thomo. Evolving schemas for streaming XML. *Theoretical Computer Science*, 412(35) :4545–4557, 2011.
Cité pages 88 et 129.

[103] P. Shvaiko and J. Euzenat. A survey of schema-based matching approaches. In S. Spaccapietra, editor, *Journal on Data Semantics IV*, volume 3730 of *Lecture Notes in Computer Science*, pages 146–171. Springer Berlin Heidelberg, 2005.
Cité page 157.

[104] S. Staworko and J. Chomicki. Validity-Sensitive Querying of XML Databases. In *Proceedings of EDBT 06, Munich, Germany, Revised Selected Papers*, volume 4254 of *Lecture Notes in Computer Science*, pages 164–177. Springer, 26–31 March 2006.
Cité pages 88, 116, 125, 127, 221 et 222.

[105] S. Staworko, E. Filiot, and J. Chomicki. Querying Regular Sets of XML Documents. In *Proceedings of LiD 08, Rome, Italy*, 2008.
Cité pages 116, 126, 130, 171, 221 et 222.

[106] D. Suciu. Typechecking for semistructured data. In *International workshop on database programming languages*, volume 2397, pages 1–20. LNCS, Springer, 2001.
Cité page 32.

[107] N. Suzuki. An edit operation-based approach to the inclusion problem for DTDs. In *Proceedings of the 22nd ACM Symposium on Applied Computing*, SAC '07, pages 482–488, New York, NY, USA, 2007. ACM.
Cité pages 158 et 171.

[108] N. Suzuki. Finding K Optimum Edit Scripts between an XML Document and a Regular Tree Grammar. In *Proceedings of EROW 07, Barcelona, Spain*. CEUR-WS.org, 13 January 2007.
Cité pages 88, 116, 128, 130, 171, 221 et 222.

[109] N. Suzuki. On Inferring K Optimum Transformations of XML Document from Update Script to DTD. In *Proceedings of COMAD 08, Mumbai, India*, pages 210–221. Computer Society of India / Allied Publishers, 17-19 December 2008.
Cité pages 88 et 129.

[110] N. Suzuki. An algorithm for inferring *k* optimum transformations of XML document from update script to DTD. *IEICE Transactions*, 93-D(8) :2198–2212, 2010.
Cité pages 88 et 129.

[111] N. Suzuki and Y. Fukushima. An XML document transformation algorithm inferred from an edit script between DTDs. In *Proceedings of the 19th Conference on Australasian Database - Volume 75*, ADC '08, pages 175–184, Darlinghurst, Australia, 2007. Australian Computer Society, Inc.
Cité page 158.

[112] M. Svoboda. Processing of Incorrect XML Data. Master's thesis, Charles University in Prague, 2010.
Cité pages 127, 128, 221 et 222.

[113] M. Svoboda and I. Mlýnková. Correction of Invalid XML Documents with Respect to Single Type Tree Grammars. In *Proceedings of NDT 11, Macau, China*, volume 136

of *Communications in Computer and Information Science*, pages 179–194. Springer, 11–13 July 2011.
Cité pages 127, 128, 130, 171, 221 et 222.

[114] J. Tekli, R. Chbeir, A. Traina, and C. Traina. XML document-grammar comparison : related problems and applications. *Central European Journal of Computer Science*, 1 :117–136, 2011.
Cité pages 88 et 125.

[115] J. Tekli, R. Chbeir, and K. Yétongnon. Structural Similarity Evaluation Between XML Documents and DTDs. In *Proceedings of WISE 07, Nancy, France*, pages 196–211, 3–7 December 2007.
Cité pages 88, 125, 221 et 222.

[116] J. Tekli, R. Chbeir, and K. Yetongnon. An overview on XML similarity : Background, current trends and future directions. *Computer Science Review*, 3(3) :151–173, Aug. 2009.
Cité page 64.

[117] A. Thomo, S. Venkatesh, and Y. Y. Ye. Visibly Pushdown Transducers for Approximate Validation of Streaming XML. In *Proceedings of FoIKS 08, Pisa, Italy*, volume 4932 of *Lecture Notes in Computer Science*, pages 219–238. Springer, 11–15 February 2008.
Cité pages 88, 116, 221 et 222.

[118] M. Vincent, J. Liu, and M. Mohania. The implication problem for 'closest node' functional dependencies in complete XML documents. *Journal of Computer and System Sciences*, 78(4) :1045 – 1098, 2012.
Cité pages 47, 53, 55, 84 et 85.

[119] M. W. Vincent, J. Liu, and C. Liu. Functional dependencies, from relational to XML. In *Ershov Memorial Conference*, pages 531–538, 2003.
Cité page 48.

[120] M. W. Vincent, J. Liu, and C. Liu. Strong functional dependencies and their application to normal forms in XML. *ACM Trans. Database Syst.*, 29(3) :445–462, 2004.
Cité pages 44, 47, 48, 57 et 84.

[121] R. A. Wagner and M. J. Fischer. The String-to-String Correction Problem. *Journal of the ACM*, 21(1) :168–173, 1974.
Cité page 97.

[122] J. Wang and R. Topor. Removing XML data redundancies using functional and equality-generating dependencies. In *ADC '05 : Proceedings of the 16th Australasian database conference*, pages 65–74, Darlinghurst, Australia, Australia, 2005. Australian Computer Society, Inc.
Cité pages 48, 50 et 84.

[123] S. Warshall. A theorem on boolean matrices. *J. ACM*, 9(1) :11–12, Jan. 1962.
Cité page 23.

[124] X. Wu, T. W. Ling, M.-L. Lee, and G. Dobbie. Designing semistructured databases using ORA-SS model. In *Proceedings of the 2nd International Conference on Web Information Systems Engineering, WISE (1)*, 2001.
Cité page 36.

[125] G. Xing, C. R. Malla, Z. Xia, and S. D. Venkata. Computing Edit Distances Between an XML Document and a Schema and its Application in Document Classification. In *Proceedings of SAC 06, Dijon, France*, pages 831–835. ACM, 23–27 April 2006.
Cité pages 88, 125, 221 et 222.

[126] C. Yu and L. Popa. Semantic adaptation of schema mappings when schemas evolve. In *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, pages 1006–1017. VLDB Endowment, 2005.
Cité page 157.

[127] X. Zhao, J. Xin, and E. Zhang. XML functional dependency and schema normalization. In *HIS '09 : Proceedings of the 9th International Conference on Hybrid Intelligent Systems*, pages 307–312, 2009.
Cité page 84.

[128] D. Ziadi, J. L. Ponty, and J. Champarnaud. Passage d'une expression rationnelle à un automate fini non-deterministe. *Bull. Belg. Math. Soc*, 4 :177–203, 1997.
Cité page 14.

Annexes

Annexe générale

A

Sommaire

A.1 Preuve de correction du calcul de la grammaire de $WI(L(G))$ 187
A.1.1 Terminaison 187
A.1.2 Correction ($L(G') \subseteq WI(L(G))$) 188
A.1.3 Complétude ($L(G') \supseteq WI(L(G))$) 190
A.2 Automate d'états finis pour un chemin P dans PL 194
A.3 Calcul du plus long préfixe commun 195
A.4 Système d'axiomes : Preuve de correction 197
A.5 Système d'axiomes : Preuve de complétude 204
A.6 Algorithme pour la fermeture d'un ensemble de chemins : Preuve de correction 212
A.7 Preuves de correction et de complétude de l'algorithme de correction 214
A.8 Tableaux de comparaison 220

A.1 Preuve de correction du calcul de la grammaire de $WI(L(G))$

A.1.1 Terminaison

Définition A.1 (Relation $>$ on equivalence classes)
Relation $>$ over non-terminals was introduced in Definition 3.4. We extend it to equivalence classes by $\widehat{A} \hat{>} \widehat{B} \iff A >^+ B \wedge A \not\equiv B$.

□

Remarque A.1 *Suppose $\widehat{A'} = \widehat{A}$, $\widehat{B'} = \widehat{B}$ and $\widehat{A} \hat{>} \widehat{B}$. We have $A' \equiv A$ ($A' \in \widehat{A}$) and $B' \equiv B$ then $A' >^* A >^+ B >^* B'$, consequently $A' >^+ B'$. On the other hand, if $A' \equiv B'$ then $A \equiv B$ which is impossible. Therefore $\widehat{A'} \hat{>} \widehat{B'}$.*

Lemme A.1 *$\hat{>}$ is transitive.*

Démonstration. Suppose $\widehat{A} \hat{>} \widehat{B} \hat{>} \widehat{C}$, therefore $A >^+ B >^+ C$ then $A >^+ C$. If $A \equiv C$, $C >^* A$ and as $A >^+ B$ we have $C >^* B$. By hypothesis, $B >^+ C$ then $B \equiv C$ which is impossible because $\widehat{B} \hat{>} \widehat{C}$. So $A \not\equiv C$, and $A >^+ C$ then $\widehat{A} \hat{>} \widehat{C}$.

◁

Lemme A.2 *$\hat{>}$ is noetherian.*

Démonstration. Suppose $\widehat{A_1} \hat{>} \widehat{A_2} \hat{>} \cdots \hat{>} \widehat{A_n} \hat{>} \cdots$. Since the set NT of non-terminal symbols is finite, the set of equivalence classes over NT is also finite, then $\exists i, j$ such that $i > j$ and $\widehat{A_i} = \widehat{A_j} = \widehat{A}$. Therefore $\widehat{A} \hat{>} \cdots \hat{>} \widehat{A}$. From previous lemma $\hat{>}$ is transitive, then $\widehat{A} \hat{>} \widehat{A}$ which implied $A \not\equiv A$, which is impossible.

◁

Lemme A.3 *For each non-terminal A, the computation of $Ch(A)$ terminates.*

Démonstration. We prove that, when computing $Ch(A)$, if we obtain $Ch(B)$ as a recursive call, then $\widehat{A} \hat{>} \widehat{B}$. As $\hat{>}$ is noetherian we will conclude that the computing of $Ch(A)$ terminates.
If A is 2-recursive, the definition of $Ch(A)$ is not recursive. Suppose that A is 1-recursive or not recursive. $Ch(B)$ is contained in the expression $Ch^{rex}_{\widehat{A}}(\widehat{reg}(A))$ (if A is not recursive, $\widehat{reg}(A) = reg(A)$ because $\widehat{A} = \{A\}$). B is a non-terminal in $\widehat{reg}(A)$, then there exists $A_i \in \widehat{A}$ and a rule $A_i \to a_i[E_i]$ such that $B \in NT(E_i)$. We deduce $A_i > B$. $Ch(B)$ must be contained in $Ch_{\widehat{A}}(B)$. If $A_i \equiv B$, A_i and B are in $\widehat{A}$ then B is 1-recursive (because A is not 2-recursive), but in this case, by definition $Ch_{\widehat{A}}(B)$ does not contain $Ch(B)$. We deduce that if $Ch_{\widehat{A}}(B)$ contains $Ch(B)$ then $A_i \not\equiv B$ and then $\widehat{A_i} \hat{>} \widehat{B}$. Since $\widehat{A_i} = \widehat{A}$, we have $\widehat{A} \hat{>} \widehat{B}$.

◁

A.1.2 Correction ($L(G') \subseteq WI(L(G))$)

Définition A.2 (Relation R)
We define the relation R over $\widehat{NT} \times \mathbb{N}$ by $(\widehat{A}, n)\, R\, (\widehat{A'}, n') \iff (\widehat{A} \hat{>} \widehat{A'}) \vee (\widehat{A} = \widehat{A'} \wedge n > n')$. In other words, $R = (\hat{>}, >)$ lexicographically where $\hat{>}$ is defined above and $>$ is the usual order over natural integers. □

Lemme A.4 *R is noetherian.*

Démonstration. Because $\hat{>}$ and $>$ are noetherian.

◁

Lemme A.5
If $B_1 \cdots B_n \in L^\omega(Ch(A))$, then $\begin{array}{ccc} & A & \\ \swarrow & | & \searrow \\ B_1 & \cdots & B_n \end{array} \in WI(L_G(A))$.

Démonstration. The proof is by noetherian induction on R over $(\widehat{A}, n)$.

– If A is not recursive. Let $A \to a[E] \in G$, we have $Ch(A) = Ch^{rex}_{\widehat{A}}(reg(A)) = Ch^{rex}_{\widehat{A}}(E)$. So, there exists a word $C_1 \cdots C_k \in L^\omega(E)$ and words $w_1, \cdots, w_k$ such that $\forall i \in \{1, \cdots, k\}$, $w_i \in L^\omega(Ch(C_i))$ if C_i is 1-recursive or 2-recursive, and $w_i \in L^\omega(Ch(C_i))$ or $w_i = C_i$ if c_i is not recursive. Let's note $B_1 \cdots B_n = w_1, \cdots, w_k$. We have $\forall i A > C_i$ and $A \not\equiv C_i$ because A is not recursive. Then $\widehat{A} \hat{>} \widehat{C_i}$ then $(\widehat{A}, n)\, R\, (\widehat{C_i}, |w_i|)$.
By induction hypothesis, $\begin{array}{c} C_i \\ | \\ w_i \end{array} \in WI(L^{nt}_G(C_i))$ (except when $w_i = C_i$)

As $\begin{array}{ccc} & A & \\ \swarrow & | & \searrow \\ C_1 & \cdots & C_k \end{array} \in WI(L^{nt}_G(A))$ we have $\begin{array}{ccc} & A & \\ \swarrow & | & \searrow \\ C_1? & \cdots & C_k? \\ | & & | \\ w_1 & & w_k \end{array} \in WI(L^{nt}_G(A))$

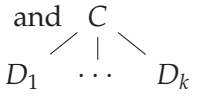
(subtree $\begin{array}{c} C_i? \\ | \\ w_i \end{array}$ is w_i when $w_i = C_i$).

Then $\begin{array}{ccc} & A & \\ w_1 & \cdots & w_k \end{array} = \begin{array}{ccc} & A & \\ B_1 & \cdots & B_k \end{array} \in WI(L_G^{nt}(A))$.

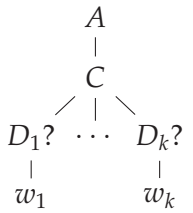
– If A is 1-recursive.

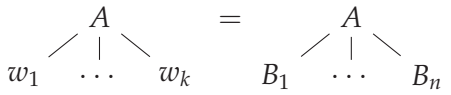
- If $B_1 \in Succ(Left(A))$, by definition of $Succ$ and $Left$, and as A is 1-recursive, it exists in $L_G^{nt}(A)$ a term with A as root, a leaf A and at the left of the path from root to the leaf A, a leaf B_1. We deduce that the tree $\begin{array}{ccc} & A & \\ B_1 & & A \end{array} \in WI(L_G^{nt}(A))$. However $B_2 \cdots B_n \in L^\omega(Ch(A))$ (due to construction of $Ch(A)$ when A is 1-recursive), and $|B_2 \cdots B_n| = n-1$ so $(\widehat{A}, n)\, R\, (\widehat{A}, n-1)$. We can apply induction hypothesis to obtain $\begin{array}{ccc} & A & \\ B_2 & \cdots & B_n \end{array} \in WI(L_G^{nt}(A))$ then $\begin{array}{ccc} & A & \\ A_1 & & A \\ & B_2 \quad \cdots \quad B_n & \end{array} \in WI(L_G^{nt}(A))$ and we conclude $\begin{array}{ccc} & A & \\ B_1 & \cdots & B_n \end{array} \in WI(L_G^{nt}(A))$.
- Else if $B_n \in Succ(Right(A))$, we apply the same reasoning that the previous case and we also conclude $\begin{array}{ccc} & A & \\ B_1 & \cdots & B_n \end{array} \in WI(L_G^{nt}(A))$.
- Else $B_1 \cdots B_n \in L^\omega(Ch^{rex}_{\widehat{A}}(\widehat{reg}(A)))$. $\exists C \longrightarrow c[E] \in G$, $C \in \widehat{A}$ and $B_1 \cdots B_n \in L^\omega(Ch^{rex}_{\widehat{A}}(E))$. Then $\exists D_1 \cdots D_k \in L^\omega(E)$ and words $w_1, \cdots, w_k$ such that $\forall i \in \{1, \cdots, k\}$
  - $w_i \in \widehat{D_i}$ or $w_i = \epsilon$ if $D_i \in \widehat{A}$.
  - $w_i = D_i$ or $w_i \in L^\omega(Ch(D_i))$ if D_i is not recursive.
  - $w_i \in L^\omega(Ch(D_i))$ else.

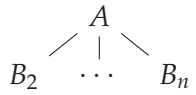
  and $B_1 \cdots B_n = w_1 \cdots w_k$. We have $\forall i$, $A >^* C > D_i$ then $A >^+ D_i$.
  - If $A \equiv D_i$ then $D_i \in \widehat{A}$ and ($w_i \in \widehat{D_i}$ or $w_i = \epsilon$) then we have a tree in $L_G^{nt}(D_i)$ with root D_i and w_i as leaves (with perhaps other leaves).
    Then $\begin{array}{c} D_i? \\ | \\ w_i \end{array} \in WI(L_G^{nt}(D_i))$.
  - If D_i is not recursive and $w_i = D_i$ then $\begin{array}{c} D_i? \\ | \\ w_i \end{array} \in WI(L_G^{nt}(D_i))$.
  - If $w_i \in L^\omega(Ch(D_i))$ and $A \not\equiv D_i$. From $A >^+ D_i$ and $A \not\equiv D_i$ we have $\widehat{A} \gtrdot \widehat{D_i}$ then $(\widehat{A}, n)\, R\, (\widehat{D_i}, |w_i|)$. By induction hypothesis, we have $\begin{array}{c} D_i \\ | \\ w_i \end{array} \in WI(L_G^{nt}(D_i))$.

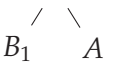
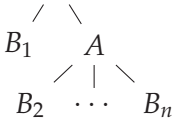
However $C \in \widehat{A}$ then it exists a tree in $L_G^{nt}(A)$ with root A and  as

subterm (perhaps other subterms exist).

Then  $\in WI(L_G^{nt}(A))$

we can conclude 

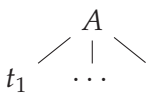
– If A is 2-recursive. We have $B_2 \cdots B_n \in L^\omega(Ch(A))$ (due to construction of $Ch(A)$ when A is 2-recursive) and $(\widehat{A}, n)\, R\, (\widehat{A}, n-1)$. We can apply induction hypothesis to obtain  $\in WI(L_G^{nt}(A))$.

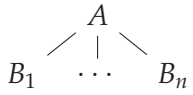
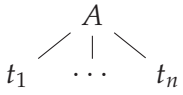
Moreover, $B_1 \in Succ(A)$ and A is 2-recursive, then it exists in $L_G^{nt}(A)$ a tree with root A and 2 leaves B_1 and A (perhaps other leaves exist). Then  $\in WI(L_G^{nt}(A))$, therefore  $\in WI(L_G^{nt}(A))$ and we conclude $\in WI(L_G^{nt}(A))$.

◁

Théorème A.1 *Correctness*
$\forall A \in NT, \forall t \in Tree(NT), t \in L_{G'}^{nt}(A) \Longrightarrow t \in WI(L_G^{nt}(A))$.

Démonstration. By induction on $|t|$.

Let  $\in L_{G'}^{nt}(A)$, then $\exists B_1 \cdots B_n \in L^\omega(Ch(A))$ such that $\forall i, t_i \in L_{G'}^{nt}(B_i)$.

$\forall i, |t_i| < |t|$. By induction hypothesis, $t_i \in WI(L_G^{nt}(B_i))$. From Lemma A.5 we get  $\in WI(L_G^{nt}(A))$. Then $t =$  $\in WI(L_G^{nt}(A))$.

◁

A.1.3 Complétude ($L(G') \supseteq WI(L(G))$)

Définition A.3 (Forest)
A *forest* is a (possibly empty) sequence of trees. The empty forest is denoted by ϵ. The size $|f|$ of the forest $f = t_1 \cdots t_n$ is defined by $|f| = |t_1| + \cdots + |t_n|$.
Given a word $u = A_1 \ldots A_n$ over non-terminals, $L_G^f(u)$ is the set of forests (composed of nt-trees) defined by $L_G^f(u) = \{t_1 \cdots t_n \mid \forall i,\, t_i \in L_G^{nt}(A_i)\}$. By convention, $L_G^f(\epsilon) = \{\epsilon\}$.

□

Lemme A.6 $\forall\, A \in NT, \epsilon \in L^w(Ch(A))$.

Démonstration. The proof is by noetherian induction on $\hat{>}$.

- If A is not recursive, we have $Ch(A) = Ch^{rex}_{\widehat{A}}(reg(A))$. $\forall B \in reg(A)$, $A > B$ and $A \not\equiv B$ (because A is not recursive) and then $\widehat{A} \hat{>} \widehat{B}$. By induction hypothesis $\epsilon \in L^w(Ch(B))$ and so we conclude that $\epsilon \in L^w(Ch(A))$.
- If A is 1-recursive, we must show that $\epsilon \in Ch^{rex}_{\widehat{A}}(\widehat{reg}(A))$. Let $B \in \widehat{reg}(A)$. We have $A >^+ B$ and two possible cases :
  - If $B \not\equiv A$, then $\widehat{A} \hat{>} \widehat{B}$. By induction hypothesis we have $\epsilon \in L^w(Ch(B))$.
  - Else $B \equiv A$, and therefore $B \in \widehat{A}$. To obtain $Ch^{rex}_{\widehat{A}}(\widehat{reg}(A))$, B was replaced by $\widehat{B}|\epsilon$ (which generates ϵ).

  By considering this two cases, if we replace each B in $\widehat{reg}(A)$ we can generate ϵ and therefore we have $\epsilon \in Ch^{rex}_{\widehat{A}}(\widehat{reg}(A))$.
- If A is 2-recursive, due to the value of $Ch(A) = (Succ(A))^*$, it is easy to verify that $\epsilon \in L^w(Ch(A))$.

◁

Corollaire A.1 $\forall A, B \in NT, \epsilon \in L^w(Ch_{\widehat{A}}(B))$.

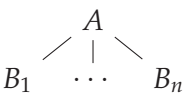
Lemme A.7 *If A is 1-recursive or 2-recursive, then $A \in L^w(Ch(A))$.*

Démonstration.

- If A is 2-recursive, we have $A \in Succ(A)$ and therefore $A \in L^w(Ch(A))$.
- If A is 1-recursive, we show that $A \in Ch^{rex}_{\widehat{A}}(\widehat{reg}(A))$.
  For all $C \in \widehat{reg}(A)$, from Lemma A.6, we have $\epsilon \in L^w(Ch(C)) \subseteq L^w(Ch_{\widehat{A}}(C))$. Since A is 1-recursive, we know also that there exists $B \in \widehat{reg}(A)$ such that $B \equiv A$. To obtain $Ch^{rex}_{\widehat{A}}(\widehat{reg}(A))$, we replace B by $Ch_{\widehat{A}}(B) = \widehat{B}|\epsilon = \widehat{A}|\epsilon$ and we have $A \in L^w(Ch_{\widehat{A}}(B))$. Since we can generate ϵ with $Ch_{\widehat{A}}(C))$, we conclude that $A \in Ch^{rex}_{\widehat{A}}(\widehat{reg}(A))$.

◁

Lemme A.8



$\in L^{nt}_{G'}(A) \iff B_1 \cdots B_n \in L^{\omega}(Ch(A))$.

Démonstration. Trivial by the definition of the Algorithm.

◁

Lemme A.9 *If $A \equiv B$ then $Ch(A) = Ch(B)$.*

Démonstration. If A is not recursive, from Lemma 3.1, $B = A$; then $Ch(B) = Ch(A)$. Otherwise, from Lemma 3.1 we have $Succ(A) = Succ(B)$; $Succ(Left(A)) = Succ(Left(B))$; $Succ(Right(A)) = Succ(Right(B))$; $\hat{reg}(A) = \hat{reg}(B)$. Therefore $Ch(A) = Ch(B)$.

◁

Corollaire A.2 $\forall C \in NT$, *If $A \equiv B$ then $Ch_{\widehat{C}}(A) = Ch_{\widehat{C}}(B)$.*

Lemme A.10 $\forall B \in NT, (B \in NT(Ch(A)) \implies A >^* B)$.

Démonstration. The proof is by noetherian induction on $\hat{>}$.

- If A is 2-recursive, $Ch(A) = (Succ(A))^*$. We deduce that $B \in NT(Succ(A))$ and so $A >^* B$.

– If A is 1-rec, $Ch(A) = (Succ(Left(A)))^*.Ch^{rex}_{\widehat{A}}(\widehat{reg}(A)).(Succ(Right(A)))^*$
We have two possible cases :

a) $B \in NT(Succ(Left(A)))$ or $B \in NT(Succ(Right(A)))$: in this case there exists $C \in \widehat{A}$ such that $C >^* B$, and since $A >^* C$ (because $C \in \widehat{A}$) we obtain $A >^* B$.

b) $B \in NT(Ch^{rex}_{\widehat{A}}(\widehat{reg}(A)))$. Then $\exists C \in \widehat{reg}(A)$, $B \in NT(Ch_{\widehat{A}}(C))$.
As $C \in \widehat{reg}(A)$, there exists $A_1 \in \widehat{A}$ s.t. $C \in reg(A_1)$. Then $A \equiv A_1$ and $A_1 > C$. Then $A >^* A_1$ and $A_1 > C$. Therefore $A >^+ C$.
If $C = B$ then we have automatically $A >^* B$. Otherwise we have three situations for $Ch_{\widehat{A}}(C)$:

i) C is 1-recursive and $C \in \widehat{A}$: $Ch_{\widehat{A}}(C) = \widehat{C}|\epsilon$ and so $B \in NT(\widehat{C}|\epsilon)$. Then $B \in \widehat{C} = \widehat{A}$, then $B \equiv A$, which establishes $A >^* B$.

ii) (C is 2-recursive) or (C is 1-recursive and $C \notin \widehat{A}$) : $Ch_{\widehat{A}}(C) = Ch(C)$ and so $B \in NT(Ch(C))$. As C is 2-recursive and A is 1-recursive or $C \notin \widehat{A}$ we have in the two cases that $A \not\equiv C$. Since $A >^+ C$ and $A \not\equiv C$, by Definition A.1 we have $\widehat{A} \hat{>} \widehat{C}$. We know that $B \in NT(Ch(C))$ and $\widehat{A} \hat{>} \widehat{C}$, by applying the induction hypothesis we obtain $C >^* B$. And since $A >^+ C$, we conclude that $A >^* B$.

iii) C is not recursive : $Ch_{\widehat{A}}(C) = C|Ch(C)$ and so $B \in NT(C|Ch(C))$. If $B = C$, we have automatically $A >^* B$. If $B \in NT(Ch(C))$, as C is not recursive and A is 1-recursive we have $A \not\equiv C$. By the same argument as in *ii*) we claim that $A >^* B$.

– If A is not recursive, the proof is as in the case 1-recursive (b), except that we replace $\widehat{reg}$ by reg.

◁

Théorème A.2 $\forall p \in \mathbb{N}$, $[\forall A \in NT, \forall w \in L^w(reg(A)), \forall f \in L^f_G(w), \forall f' \lhd f, (|f| \leq p \implies \exists w' \in L^w(Ch(A)), f' \in L^f_{G'}(w'))]$.

Démonstration. The proof is by induction on p. Let $w = (A_1, \cdots, A_n)$ and $f = (t_1, \cdots, t_n)$. We have $t = A(t_1, \cdots, t_n) \in L^{nt}_G(A)$ which is described by Figure A.1.

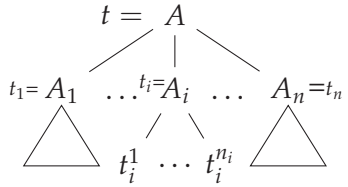
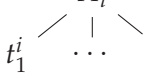


FIGURE A.1 – *Example for the tree* $t \in L^{nt}_G(A)$ *with* $f = (t_1, \cdots, t_n)$

Let $f' = (f'_1, \cdots, f'_n)$ such that $\forall i \in [1, \ldots, n]$, $f'_i \lhd t_i$ (note that the root of t_i can be removed, in other words, t_i may be a forest).

– if $p = 0$ then $w = \epsilon$, $w' = \epsilon$, $f = \epsilon$ and $f' = \epsilon$. By Lemma A.6, $\epsilon \in L^w(Ch(A))$ and therefore $f' \in L^f_{G'}(\epsilon) = \{\epsilon\}$.

– else $p \neq 0$ and then $n \neq 0$. $t_i =$  as described in Figure A.1, and

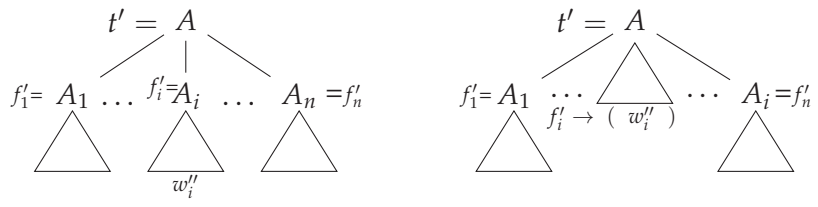


FIGURE A.2 – *Example of the two possibilities for the tree $t' \lhd t$, with $f' = (f'_1, \cdots, f'_n)$ and $[f'_i = A_i(f''_i)$ or $f'_i = f''_i]$*

$f_i = (t_i^1, \cdots, t_n^{n_i})$. Thus $f'_i = \begin{array}{c} A_i \\ | \\ f''_i \end{array}$ (illustrated in Figure A.2(a)) or

$f'_i = f''_i$ (illustrated in Figure A.2(b)). Since $|f_i| < |f|$, by induction hypothesis we obtain : $\exists\, w''_i \in L^w(Ch(A_i))$, $f''_i \in L^f_{G'}(w''_i)$. However :

$$\begin{cases} \text{if } f'_i = \begin{array}{c} A_i \\ | \\ f''_i \end{array} \text{ since } A_i \to a_i[Ch(A_i)] \in G', \text{ we have } f'_i \in L^{nt}_{G'}(A_i). \text{ Let } w'_i = A_i \\ \text{else } f'_i = f''_i \in L^f_{G'}(w''_i) \text{ with } w''_i \in L^w(Ch(A_i)). \text{ Let } w'_i = w''_i \end{cases}$$

Let $w' = (w'_1, \cdots, w'_n)$, we have $f' \in L^f_{G'}(w')$.

We now show that $w' \in L^w(Ch(A))$:

i) A is 2-recursive : since $Ch(A) = (Succ(A))^*$, It is sufficient to show that each non-terminal in w' is in $Succ(A)$. We have $w = (A_1, \cdots, A_n) \in L^w(reg(A))$, and therefore $A > A_i$.

- if $w'_i = A_i$, we have $w_i \in Succ(A)$,
- if $w'_i = w''_i \in L^w(Ch(A_i))$, let $w'_i = B_1 \cdots B_k$. By Lemma A.10, $\forall\, j \in [1, \ldots, k]$ $A_i >^* B_j$, thus $B_j \in Succ(A)$.

ii) A is not recursive : to obtain $Ch(A) = Ch^{rex}_{\widehat{A}}(reg(A))$ from $reg(A)$, we replace in $reg(A)$ each A_i by :

- $Ch(A_i)$ if A_i is 2-recursive or 1-recursive (note that the case A_i is 1-recursive and $A_i \in \widehat{A}$ is impossible). And by Lemma A.7 we have $A_i \in L^w(Ch(A_i))$
- $A_i | Ch(A_i)$ if A_i is not recursive

However $w = (A_1 \cdots A_n) \in L^w(reg(A))$, $w' = (w'_1 \cdots w'_n)$ and $\forall i, w'_i = A_i$ or $w'_i \in L^w(Ch(A_i))$. Thus $w' \in L^w(Ch(A))$.

iii) A is 1-rec : $Ch(A) = (Succ(Left(A)))^*.Ch^{rex}_{\widehat{A}}(\widehat{reg}(A)).(Succ(Right(A)))^*$

- Suppose that $\forall\, i$, $A_i \not\equiv A$: we show that $w' \in L^w(Ch^{rex}_{\widehat{A}}(\widehat{reg}(A))) \subseteq L^w(Ch(A))$ and the proof is similar to *ii)*.
- Else $\exists\, i$, $A_i \equiv A$ and then A_i is 1-recursive. Therefore
  $\begin{cases} A_i, \ldots, A_{i-1} \in Left(A_i) = Left(A) \subseteq Succ(Left(A)) \\ A_{i+1}, \ldots, A_n \in Right(A_i) = Right(A) \subseteq Succ(Right(A)) \end{cases}$
  Also $\forall\, j \in [1, \ldots, i-1]$, $(w''_j = B_1 \cdots B_k \implies \forall\, p, A_i >^* B_p \implies B_p \in Succ(Left(A)))$. Therefore all the non-terminals of $w'_1 \cdots w'_{i-1}$ are in $Succ(Left(A))$. By using the same argument all the non-terminals of

$w'_{i+1} \cdots w'_n$ are in $Succ(Right(A))$.
In the other hand to obtain $Ch^{rex}_{\widehat{A}}(\widehat{reg}(A))$ from $\widehat{reg}(A)$, we replace all the non-terminals B by $Ch_{\widehat{A}}(B)$. By using Corollary A.1, we have $\epsilon \in L^w(Ch_{\widehat{A}}(B))$. Also on the other hand, when $B = A_i$ and $Ch_{\widehat{A}}(A_i) = A_i|\epsilon$, it follows that $A_i \in L^w(Ch^{rex}_{\widehat{A}}(\widehat{reg}(A)))$. We now analyse the two possible values for the word w'_i :

* if $w'_i = A_i$ then $w' = (w'_1, \cdots, w'_i, \cdots, w'_n) \in L^w(Ch(A))$
* else $w'_i = w''_i \in L^w(Ch(A_i))$. By Lemma A.9,

$$\begin{aligned} Ch(A_i) = &\quad (Succ(Left(A_i)))^*.Ch^{rex}_{\widehat{A_i}}(\widehat{reg}(A_i)).(Succ(Right(A_i)))^* \\ = &\quad (Succ(Left(A)))^*.Ch^{rex}_{\widehat{A}}(\widehat{reg}(A)).(Succ(Right(A)))^* \end{aligned}$$

And then $w' = (w'_1, \cdots, w'_i, \cdots, w'_n) \in$

$L^w(Succ(Left(A))^*.[Succ(Left(A))^*.Ch^{rex}_{\widehat{A}}(\widehat{reg}(A)).Succ(Right(A))^*].Succ(Right(A))^*)$
$= L^w(Succ(Left(A))^*.Ch^{rex}_{\widehat{A}}(\widehat{reg}(A)).Succ(Right(A))^*) = L^w(Ch(A))$
This establishes the proof of Theorem A.2.

◁

Corollaire A.3 $\forall A \in NT, \forall t \in L_G(A), \forall t' \lhd t,\ t' \in L_{G'}(A)$.

Démonstration. Let $t \in L_G(A)$. When we replace each terminal symbol in t by the corresponding non-terminal symbol, we obtain $tt = \begin{array}{ccc} & A & \\ t_1 & \cdots & t_n \end{array} \in L^{nt}_G(A)$ and $\forall i \in [1, \ldots, n]$, $t_i \in L^{nt}_G(A_i)$. We also replace each terminal symbol in t' by the corresponding non-terminal symbol and we obtain :
$tt' = \begin{array}{ccc} & A & \\ t'_1 & \cdots & t'_m \end{array}$ and $\forall j \in [1, \ldots, m]$, $t'_j(\epsilon) = A'_j$.
Let $f = (t_1, \cdots, t_n)$, $w = (A_1, \cdots, A_n)$, $f' = (t'_1, \cdots, t'_m)$ and $w' = (A'_1, \cdots, A'_m)$. Furthermore we have $t \in L_G(A)$ and $t' \lhd t$ which implies that $w \in L^w(reg(A))$ and $f' \lhd f$. Thus by using Theorem A.2, we have $w' \in L^w(Ch(A))$ and $f' \in L^f_{G'}(w')$. Therefore with Lemma A.8, we claim that $tt' \in L^{nt}_{G'}(A)$ as $w' \in L^w(Ch(A))$. We can conclude that $t' \in L_{G'}(A)$ after replacing non-terminal symbols by terminal symbols.

◁

A.2 Automate d'états finis pour un chemin P dans PL

L'Algorithme 14 (Annexe A.2) permet de construire l'automate d'états finis (FSA) B_P à partir d'un chemin donné P dans PL sur l'alphabet Σ. Le FSA B_P est un 5-uplet $(Q, \Sigma, s_0, \delta, s_f)$ où

- Q est l'ensemble des états,
- Σ est l'alphabet,
- s_0 est l'état initial,
- δ est la fonction transition tel que $\delta : Q \times \Sigma \to Q$
- et s_f est l'état final.

L'automate B_P est construit en analysant le chemin P, à partir de l'ensemble des états $Q = \{s_0\}$. L'état s_0 est l'état initial et l'état courant s est initialisé à s_0. En analysant le chemin P, si on est sur un label $a \in \Sigma$ alors :

1. lorsque le label a est précédé du symbole '/' (*i.e.* /a), on rajoute la transition $\delta(s, a) = s_i$ à l'automate B_P où s_i est un nouvel état. L'état courant s devient l'état s_i
2. lorsque le label a est précédé du symbole '//' (*i.e.* //a), on boucle sur l'état courant en rajoutant les transitions $\forall\, b,\ \delta(s, b) = s$ à l'automate B_P. L'état courant s passe ensuite à un nouvel état s_i après avoir ajouté la transition $\delta(s, a) = s_i$ à l'automate B_P.

Aucune action n'est faite sur l'automate B_P lorsqu'on rencontre le symbole du chemin vide '[]'. A la fin de l'analyse du chemin P, l'état final s_f est égal à l'état courant s.

Algorithme 14 Automate d'états finis B_P construit à partir du chemin P

Entrée : Chemin P dans PL et Alphabet Σ
Sortie : FSA $B_P = (Q, \Sigma, s_0, \delta, s_f)$

$Q := \{s_0\}$
$op :=$ '/' % *op contient le dernier '/' ou '//' que l'on a rencontré en parsant* P %
$i := 1$ % *indice pour les nouveaux états* s_i *dans* Q %
$s := s_0$ % s *est l'état courant dans l'automate que nous construisons* %
pour tout symbole a dans P **faire**
    si $a \in \Sigma$ **alors**
        $Q := Q \cup \{s_i\}$
        si $op =$ '/' **alors**
            $\delta(s, a) := s_i$
        sinon si $op =$ '//' **alors**
            $\delta(s, a) := s_i$
            pour tout label b dans Σ **faire**
                $\delta(s, b) := s$
            fin pour
        fin si
        $s := s_i$
        $i := i + 1$
    sinon si $a =$ '/' ou $a =$ '//' **alors**
        $op := a$
    sinon si $a =$ '[]' **alors**
        % *rien à faire* %
    fin si
fin pour
$s_f := s$
$B_P := (Q, \Sigma, s_0, \delta, s_f)$
retourner B_P

A.3 Calcul du plus long préfixe commun

Étant donné l'automate A_P associé au chemin P et l'automate A_R associé au chemin R, l'Algorithme 15 permet de calculer l'automate $A_{P \cap R}$ associé au plus long préfixe commun $P \cap R$ des chemins P et R. Nous allons supposer que l'automate A_P et A_R sont déterministes.

Soit $A_P = (Q_P, \Sigma, I_P, \delta_P, F_P)$ et $A_R = (Q_R, \Sigma, I_R, \delta_R, F_R)$. L'Algorithme 15 est similaire à l'algorithme d'intersection de deux automates d'états finis. Voici donc comment fonctionne l'Algorithme 15 :

- chaque état de l'automate $A_{P\cap R}$ est un *couple-d'état* (p, q) tel que $p \in Q_P$ et $q \in Q_R$.
- les *couple-d'états* initiaux de $A_{P\cap R}$ proviennent de l'ensemble $I_P \times I_R$ où $I_P \times I_R$ est le produit cartésien de I_P et I_R.
- un *couple-d'état* (p', q') est un état de l'ensemble Q lorsqu'il existe $(p, q) \in Q$ et $a \in \Sigma$ tel que $\delta_P(p, a) = p'$ et $\delta_R(q, a) = q'$ (*i.e.* (p', q') est accessible à partir d'un *couple-d'état* $(p, q) \in Q$).
- la différence entre l'Algorithme 15 et l'algorithme d'intersection est la façon dont est calculée l'ensemble des *couple-d'états* finaux. Dans l'algorithme d'intersection, un *couple-d'état* (p, q) est final si $p \in F_P$ et $q \in F_R$. Dans l'Algorithme 15, un *couple-d'état* (p, q) est final si $p \in F_P$ ou $q \in F_R$, ou s'il existe $a, b \in \Sigma$ tel que $\delta_P(p, a)$ et $\delta_R(q, b)$ sont définis et $a \neq b$ (cette dernière condition indique le moment où les chemins divergent).

Algorithme 15 Calcul du plus long préfixe commun de deux chemins

Entrée :
- $A_P = (Q_P, \Sigma, I_P, \delta_P, F_P)$, FSA associé à P
- $A_R = (Q_R, \Sigma, I_R, \delta_R, F_R)$, FSA associé à R

Sortie : $A_{P\cap R}$, FSA associé à $P \cap R$ (le plus long préfixe commun de P et R)

1: $Q := \emptyset$ **% *initialisation de l'ensemble des couple-d'états* %**
2: $I := I_P \times I_R$ **% *initialisation de l'ensemble des couple-d'états initiaux* %**
3: $F := \emptyset$ **% *initialisation de l'ensemble des couple-d'états finaux* %**
4: $worklist := \emptyset$ **% *liste pour stocker les états non traités* %**
5: $worklistprec := \emptyset$ **% *liste pour stocker les états déjà traités* %**
6: **pour tout** $(p, q) \in I$ **faire**
7: $worklist$.**ajouter**$((p, q))$
8: **fin pour**
9: **Tant que** $worklist$.**taille**$() > 0$ **faire**
10: $(p, q) := worklist$.**supprimerTete**$()$
11: $worklistprec$.**ajouter**$((p, q))$ **% $(\mathbf{p}, \mathbf{q})$ *est marqué comme traité* %**
12: $Q := Q \cup \{(p, q)\}$
13: **% *calcul des nouveaux couple-d'états* %**
14: **pour tout** $a \in \Sigma$ tel que $\delta_P(p, a)$ et $\delta_R(q, a)$ sont définis **faire**
15: $\delta((p, q), a) := (\delta_P(p, a), \delta_R(q, a))$
16: **si** $(\delta_P(p, a), \delta_R(q, a)) \notin worklistprec$ **alors**
17: $worklist$.**ajouter**$((\delta_P(p, a), \delta_R(q, a)))$
18: **fin si**
19: **fin pour**
20: **% *vérification si* $(\mathbf{p}, \mathbf{q})$ *est un couple-d'état final* %**
21: **si** $p \in F_P$ ou $q \in F_R$ **alors**
22: $F := F \cup \{(p, q)\}$
23: **sinon si** $\exists\, a, b \in \Sigma$ tel que $\delta_P(p, a)$ et $\delta_R(q, b)$ sont définis et $a \neq b$ **alors**
24: $F := F \cup \{(p, q)\}$
25: **fin si**
26: **fin Tant que**
27: **retourner** $(Q, \Sigma, I, \delta, F)$

Exemple A.1 Nous allons considérer le document XML de la Figure 2.2 et l'ensemble $\mathbb{P} = L(D)$ de la Figure 4.3(a). Soit P = *library/section/title* et R = *library/section/txt*, le plus long préfixe commun $P \cap R$ est égale à *library/section* puisque *library/section* décrit l'ensemble { *library/book/chapter/section, library/book/chapter/section/section, library/book/chapter/section/section/section*}. Dans la Figure A.3 sont illustrés les automates A_P, A_R associés aux chemins P, R, et l'automate construit par l'Algorithme 15 pour $P \cap R$.

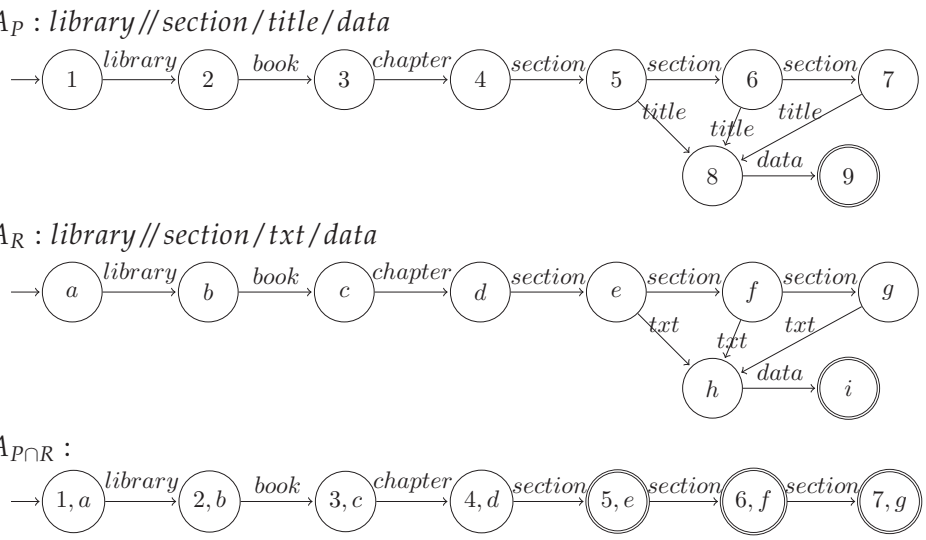


FIGURE A.3 – *FSA associés aux chemins P, R et P ∩ R.*

⊠

A.4 Système d'axiomes : Preuve de correction

Dans cette section nous allons prouver que notre système d'axiomes est correct, c'est à dire que nos axiomes dérivent toujours des résultats vrais lorsque nous considérons les arbres XML complets. Nous commençons par prouver quelques lemmes. Le premier concerne une propriété sur le plus long préfixe commun de chemins. L'exemple suivant illustre la propriété dont il s'agit.

Exemple A.2 Nous considérons le document XML $\mathcal{T}_1$ de la Figure 4.6 et les chemins suivants :

- $P_K = univ/undergraduate/@domain$
- $P_J = univ/undergraduate/students/student/idSt$
- $P_I = univ/undergraduate/students/student/nameSt$

En considérant ces chemins ci-dessus nous avons $P_J \cap P_K = univ/undergraduate$ et aussi $P_I \cap P_J = univ/undergraduate/students/student$. A partir de ces plus long préfixes commun on a (i) $P_J \cap P_K \preceq P_I \cap P_J$. Soient les instances $K = \epsilon/2/2.1$, $J = \epsilon/2/2.2/2.2.0/2.2.0.0$ et $I = \epsilon/2/2.2/2.2.0/2.2.0.1$. Nous obtenons que (ii) $isInst_lcp(P_I, I, P_J, J) = vrai$ et que $isInst_lcp(P_I, I, P_K, K) = vrai$. Comme les conditions (i) et (ii) sont vérifiées, nous pouvons conclure par transitivité que $isInst_lcp(P_J, J, P_K, K) = vrai$.

⊠

L'exemple ci-dessus suggère qu'une sorte de propriété sur la transitivité peut être établie sur la fonction *isInst_lcp*. Le lemme ci-dessous nous prouve que cela est possible sous certaines conditions.

Lemme A.11 *Soit $\mathcal{T}$ un document XML et $\mathbb{P}$ son ensemble de chemins simples associés. Soit P_I, P_J, P_K des chemins différents dans $\mathbb{P}$. Si nous avons :*

1. $P_J \cap P_K \preceq P_I \cap P_J$ *et*
2. $isInst_lcp(P_I, I, P_J, J) = vrai$ *et* $isInst_lcp(P_I, I, P_K, K) = vrai$

alors isInst_lcp$(P_J, J, P_K, K) = vrai$.

Démonstration. Soit v_i, v_j et v_k les nœuds correspondant à $Last(I)$, $Last(J)$ et $Last(K)$, respectivement. A partir des conditions établies par le lemme, nous pouvons distinguer les trois situations illustrées dans la Figure A.4. Lorsque $P_J \cap P_K \prec P_I \cap P_J$ nous sommes dans la situation de la Figure A.4(a). Et lorsque $P_J \cap P_K = P_I \cap P_J$ nous sommes dans la situation de la Figure A.4(b) ou (c).

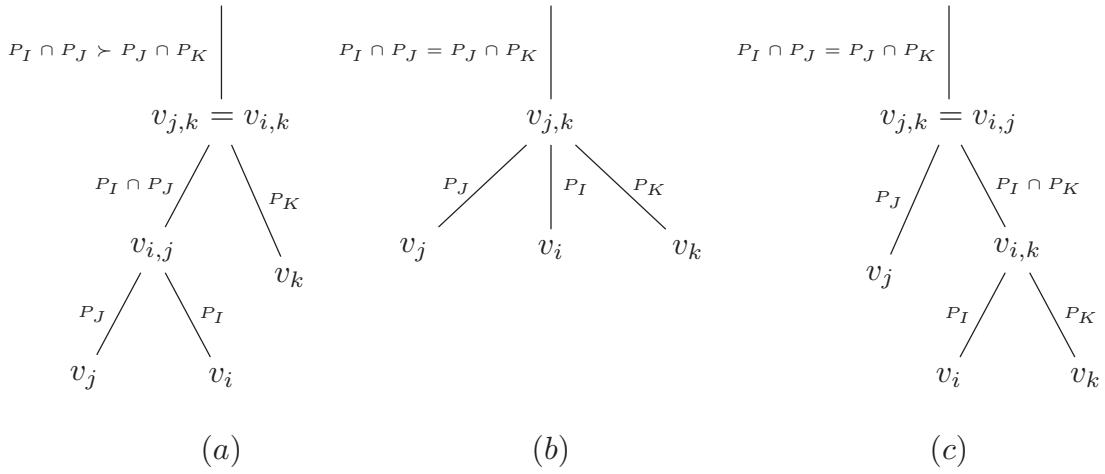


FIGURE A.4 – *Trois situations obtenues à partir des conditions du Lemme A.11.*

A partir de la condition 2 et de la Définition 4.12 nous savons que $I \cap J$ est l'instance du plus long préfixe commun de P_I et P_J ($P_I \cap P_J$). Le même raisonnement est utilisé pour $I \cap K$. Nous dénotons par $v_{i,j}$ le nœud $Last(I \cap J)$ et par $v_{i,k}$ le nœud $Last(I \cap K)$. Il est évident que $v_{i,j}$ est un ancêtre de v_i et v_j tandis que $v_{i,k}$ est un ancêtre de v_i et v_k.

- *Cas 1 - Figure A.4(a)* : Dans ce cas nous avons $P_I \cap P_K = P_J \cap P_K$. A partir de ce fait avec la condition 1 nous en déduisons que le nœud $v_{i,k}$ est un ancêtre de $v_{i,j}$, et donc par conséquent $v_{i,k}$ est aussi un ancêtre de v_j. Puisque $P_I \cap P_K = P_J \cap P_K$ nous obtenons que $Last(J \cap K) = Last(I \cap K) = v_{i,k}$. Ainsi nous avons *isInst_lcp*$(P_J, J, P_K, K) = vrai$.
- *Cas 2 - Figure A.4(b)* : Dans ce cas nous avons aussi $P_I \cap P_K = P_J \cap P_K$ et la preuve est similaire au cas 1.
- *Cas 3 - Figure A.4(c)* : Dans ce cas nous avons $P_J \cap P_K \prec P_I \cap P_K$. A partir de la condition 1 et du fait que $P_J \cap P_K \prec P_I \cap P_K$, nous en déduisons que le nœud $v_{i,j}$ est un ancêtre de $v_{i,k}$ et donc par conséquent $v_{i,j}$ est aussi un ancêtre de v_k. Puisque $P_I \cap P_J = P_J \cap P_K$ nous obtenons que $Last(J \cap K) = Last(I \cap J) = v_{i,j}$. Ainsi nous avons *isInst_lcp*$(P_J, J, P_K, K) = vrai$.

◁

Le lemme suivant traite de l'extension d'une branche (et de sa projection) en lui ajoutant un chemin (et son instance). Comme nous travaillons avec les arbres complets, il est toujours possible d'étendre une projection en lui ajoutant l'instance d'un chemin qui n'appartient pas à sa branche.

Lemme A.12 *Soit M une branche sur un arbre complet $\mathcal{T}$ et soit P' un chemin valide. Soit M' une branche obtenue par l'union $M \cup \{P'\}$. Si il existe une projection $\Pi_M(\mathcal{T})$ alors il existe une instance de P' dans $\mathcal{T}$ (que nous notons $\Pi_{\{P'\}}(\mathcal{T})[P']$) telle que $\Pi_{M'}(\mathcal{T}) = \Pi_M(\mathcal{T}) \cup \Pi_{\{P'\}}(\mathcal{T})$.*

Démonstration. D'après la Définition 4.12, nous considérons $Long_M = \{P_1, \ldots, P_n\}$ et l'ensemble des instances $SetPathInst_M = \{\Pi_M(\mathcal{T})[P_1], \ldots, \Pi_M(\mathcal{T})[P_n]\}$. Nous allons renommer le chemin P' en P_{n+1}. Considérons l'ensemble $\{P_1 \cap P_{n+1}, \ldots, P_n \cap P_{n+1}\}$ des plus long préfixes communs de $P_1, \ldots, P_n$ et de P_{n+1}. Ces chemins peuvent être totalement ordonés selon la relation préfixe $\preceq$ puisque chaque chemin est un préfixe de P_{n+1}. Nous allons ensuite renommé les indices de $P_1, \ldots, P_n$ tels que :

$$i < j \Rightarrow (P_i \cap P_{n+1}) \preceq (P_j \cap P_{n+1}) \tag{A.1}$$

Considérons le chemin P_n suivant le résultat obtenu de (A.1). Comme $\mathcal{T}$ est complet, il existe une instance $\Pi_{\{P_{n+1}\}}(\mathcal{T})[P_{n+1}]$ du chemin P_{n+1} telle que :

$$isInst_lcp(P_n, \Pi_M(\mathcal{T})[P_n], P_{n+1}, \Pi_{\{P_{n+1}\}}(\mathcal{T})[P_{n+1}]) = vrai \tag{A.2}$$

Soit $\Pi_{M'}(\mathcal{T}) = \Pi_M(\mathcal{T}) \cup \Pi_{\{P_{n+1}\}}(\mathcal{T})$. La preuve va consister à montrer que l'ensemble des instances de chemins $SetPathInst_{M'} = SetPathInst_M \cup \Pi_{\{P_{n+1}\}}(\mathcal{T})[P_{n+1}]$ vérifie aussi les trois conditions de la Définition 4.12. La vérification des deux premières conditions est évidente. Nous allons nous intéresser à la vérification de la troisième condition à savoir :

$$\forall\, i, j \in [1, \ldots, n+1],\ isInst_lcp(P_i, \Pi_{M'}(\mathcal{T})[P_i], P_j, \Pi_{M'}(\mathcal{T})[P_j]) = vrai \tag{A.3}$$

Par hypothèse, la construction de $SetPathInst_M$ assume que $\forall\, i, j \in [1, \ldots, n]$, $isInst_lcp(P_i, \Pi_M(\mathcal{T})[P_i], P_j, \Pi_M(\mathcal{T})[P_j]) = vrai$. Puisque $\Pi_{M'}(\mathcal{T})[P_i] = \Pi_M(\mathcal{T})[P_i]$ (pour $i \in [1, \ldots, n]$), et en considérant (A.2), la condition (A.3) se réduit à :

$$\forall\, i \in [1, \ldots, n-1],\ isInst_lcp(P_i, \Pi_M(\mathcal{T})[P_i], P_{n+1}, \Pi_{M'}(\mathcal{T})[P_{n+1}]) = vrai \tag{A.4}$$

Enfin en utilisant le Lemme A.11 avec $P_I = P_n$, $P_J = P_{n+1}$, $P_K = P_i$, et le fait que $(P_i \cap P_{n+1}) \preceq (P_n \cap P_{n+1})$ (selon (A.1)), nous obtenons (A.4) qui termine la preuve du lemme.

◁

Pour illustrer le Lemme A.12, considérons l'exemple suivant.

Exemple A.3 Soit le document XML $\mathcal{T}_1$ de la Figure 4.6. Soit $M = \{$*univ/undergraduate/courses/course/codeC* , *univ/undergraduate/courses/course/titleC*$\}$. Soit la projection M qui contient les instances de chemins $\epsilon/2/2.3/2.3.0/2.3.0.0$ et $\epsilon/2/2.3/2.3.0/2.3.0.1$. Soit $P' =$ *univ/undergradute/@domain* et $M' = M \cup \{P'\}$. L'instance $\epsilon/2/2.1$ de P' est celle pour laquelle nous avons $\Pi_{M'}(\mathcal{T}) = \Pi_M(\mathcal{T}) \cup \Pi_{\{P'\}}(\mathcal{T})$.

⊠

L'exemple suivant illustre une situation particulière où une XFD qui est violée par un document, possède dans sa partie gauche un chemin qui est un préfixe du chemin à droite de la XFD.

Exemple A.4 Nous considérons le document XML $\mathcal{T}_1$ de la Figure 4.6, la XFD $f = ($*univ/-undergraduate*, ($\{$*courses//titleC*, *courses//prerequisitesC*$\}$ $\rightarrow$ *courses//prerequisitesC/-CodeC*)) et la branche M définie par f. L'arbre $\mathcal{T}_1$ ne satisfait pas f. Notons que $P_2 =$ *univ/undergradute/courses/course/prerequisitesC* dans la partie gauche de f, est un préfixe de $P =$ *univ/undergradute/courses/course/prerequisitesC/CodeC* qui est dans la partie droite de f. Aussi remarquons que nous pouvons trouver deux projections de $\mathcal{T}_1$ sur M telles que $Last(\Pi^1_M(\mathcal{T}_1)[C/P_2]) =_N Last(\Pi^2_M(\mathcal{T}_1)[C/P_2]) = 2.3.0.2$

⊠

Dans les situations comme celle illustrée dans l'Exemple A.4, le prochain lemme nous dit ceci :

Lemme A.13 *Soient $\mathcal{T}$ un document XML, $f = (C, (\{P_1[E_1], \ldots, P_n[E_n]\} \rightarrow P_{n+1}[E_{n+1}]))$ une XFD et M est la branche $\{C/P_1, \ldots, C/P_{n+1}\}$. Si $\mathcal{T} \not\models f$ et il existe $j \in [1 \ldots n]$ tel que $P_j \preceq P_{n+1}$ alors nous pouvons trouver deux projections $\Pi^1_M(\mathcal{T})$ et $\Pi^2_M(\mathcal{T})$ de $\mathcal{T}$ sur M telles que $Last(\Pi^1_M(\mathcal{T})[C/P_j]) =_N Last(\Pi^2_M(\mathcal{T})[C/P_j])$.*

Démonstration. La preuve sera faite par contradiction. Supposons que pour n'importe quelles projections $\Pi^1_M(\mathcal{T})$ et $\Pi^2_M(\mathcal{T})$ de $\mathcal{T}$ sur M nous avons $Last(\Pi^1_M(\mathcal{T})[C/P_j]) \neq_N Last(\Pi^2_M(\mathcal{T})[C/P_j])$. Puisque $\mathcal{T} \not\models f$ alors d'après la Définition 4.16, nous déduisons qu'il existe deux projections (soient τ^1 et τ^2 ces deux instances) de $\mathcal{T}$ sur M telles que $\tau^1[C/P_1, \ldots, C/P_n] =_{\vec{E}} \tau^2[C/P_1, \ldots, C/P_n]$ et $\tau^1[C/P_{n+1}] \neq_{E_{n+1}} \tau^2[C/P_{n+1}]$ avec $\vec{E} = E_1, \ldots, E_n$.

- si $E_j = N$ alors $Last(\Pi^1_M(\mathcal{T})[C/P_j]) =_N Last(\Pi^2_M(\mathcal{T})[C/P_j])$ qui contredit l'hypothèse $Last(\Pi^1_M(\mathcal{T})[C/P_j]) \neq_N Last(\Pi^2_M(\mathcal{T})[C/P_j])$.
- sinon si $E_j = V$ alors $Last(\Pi^1_M(\mathcal{T})[C/P_j]) =_V Last(\Pi^2_M(\mathcal{T})[C/P_j])$. En premier supposons que les instances $\Pi^1_M(\mathcal{T})[C/P_j]$, $\Pi^2_M(\mathcal{T})[C/P_j]$ sont les préfixes pour deux instances du chemin P_{n+1}. Soit $\Pi^1_M(\mathcal{T})[C/P_j]$ le préfixe de I et J, et soit $\Pi^2_M(\mathcal{T})[C/P_j]$ le préfixe de I' et J'.
  1. si $Last(I) \neq_V Last(J)$ alors considérons que :
     • $\Pi^1_M(\mathcal{T})[C/P_{n+1}] = I$;
     • il existe une projection $\Pi^3_M(\mathcal{T})$ qui coïncide avec $\Pi^1_M(\mathcal{T})$ sauf pour l'instance du chemin P_{n+1}, puisque nous posons $\Pi^3_M(\mathcal{T})[C/P_{n+1}] = J$.
     Dans cette situation, $Last(\Pi^1_M(\mathcal{T})[C/P_j]) =_N Last(\Pi^3_M(\mathcal{T})[C/P_j])$ ce qui conduit à une contradiction avec notre hypothèse de départ qui dit que $Last(\Pi^1_M(\mathcal{T})[C/P_j]) \neq_N Last(\Pi^3_M(\mathcal{T})[C/P_j])$. Le même argument est utilisé lorsque $Last(I') \neq_V Last(J')$.
  2. sinon nous sommes dans la situation où $Last(I) =_V Last(J)$ et $Last(I') =_V Last(J')$. Puisque $Last(\Pi^1_M(\mathcal{T})[C/P_{n+1}]) \neq_V Last(\Pi^2_M(\mathcal{T})[C/P_{n+1}])$ alors cela implique aussi que $Last(\Pi^1_M(\mathcal{T})[C/P_j]) \neq_V Last(\Pi^2_M(\mathcal{T})[C/P_j])$ car nous avons $Last(I) \neq_V Last(I')$. Nous obtenons donc une contradiction avec notre hypothèse $Last(\Pi^1_M(\mathcal{T})[C/P_j]) =_V Last(\Pi^2_M(\mathcal{T})[C/P_j])$.

  Maintenant supposons que chaque instance $\Pi^1_M(\mathcal{T})[C/P_j]$ et $\Pi^2_M(\mathcal{T})[C/P_j]$ est préfixe d'une seule instance du chemin P_{n+1}. Puisque $Last(\Pi^1_M(\mathcal{T})[C/P_{n+1}]) \neq_V Last(\Pi^2_M(\mathcal{T})[C/P_{n+1}])$, cela implique aussi que $Last(\Pi^1_M(\mathcal{T})[C/P_j]) \neq_V Last(\Pi^2_M(\mathcal{T})[C/P_j])$. Nous obtenons une contradiction avec notre hypothèse $Last(\Pi^1_M(\mathcal{T})[C/P_j]) =_V Last(\Pi^2_M(\mathcal{T})[C/P_j])$.

◁

Nous pouvons maintenant prouver la correction de notre système d'axiomes sur les arbres complets.

Théorème A.3 *Les Axiomes A1-A9 sont corrects pour les XFD dans les arbres XML complets.*

Démonstration. Nous supposons un arbre XML $\mathcal{T}$ qui est complet par rapport à $\mathbb{P}$.

A1 : Soit $f = (C, (\{P_1[E_1], \ldots, P_n[E_n]\} \rightarrow P_i[E_i]))$. La preuve sera faite par contradiction. Supposons que $\mathcal{T} \not\models f$. D'après la Définition 4.16, nous déduisons qu'il existe deux projections τ^1 and τ^2 pour la branche $M = \{C/P_1, \ldots, C/P_n\}$ dans $\mathcal{T}$ telles que

$$\tau^1[C/P_1, \ldots, C/P_n] =_{E_i, i \in [1 \ldots n]} \tau^2[C/P_1, \ldots, C/P_n]$$

$$\text{et pour un } j \in [1,\ldots,n],\ \tau^1[C/P_j] \neq_{E_j} \tau^2[C/P_j].$$

Comme il existe une seule instance du chemin C/P_j dans τ^1 et τ^2, pour satisfaire la partie gauche de la dépendance on a $\tau^1[C/P_j] =_{E_j} \tau^2[C/P_j]$. Cette égalité contredit le fait que $\tau^1[C/P_j] \neq_{E_j} \tau^2[C/P_j]$.

A2 : Soit $f = (C, (\{P_1\,[E_1], \ldots, P_n\,[E_n]\} \rightarrow \{Q_1\,[E'_1], \ldots, Q_m\,[E'_m]\}))$
et $f' = (C, (\{P_0\,[E_0], P_1\,[E_1], \ldots, P_n\,[E_n]\} \rightarrow \{P_0\,[E_0], Q_1\,[E'_1], \ldots, Q_m\,[E'_m]\}))$.
Nous allons faire une preuve par contraposée. Nous prouvons que $\mathcal{T} \not\models f'$ alors $\mathcal{T} \not\models f$. Posons $Q_0 = P_0$ et $E'_0 = E_0$. D'après la Définition 4.16, nous déduisons qu'il existe Q_j avec $j \in [0,\ldots,m]$ tel que $\tau^1[C/Q_j] \neq_{E'_j} \tau^2[C/Q_j]$. Supposons en premier que $Q_j = Q_0$. En utilisant le même argument que pour A1, nous obtenons une contradiction puisque selon la Définition 4.16, $\tau^1[C/Q_0] =_{E'_0} \tau^2[C/Q_0]$ mais $\tau^1[C/Q_0] \neq_{E'_0} \tau^2[C/Q_0]$. D'où nous supposons plutôt que $j > 0$. Comme $\mathcal{T} \not\models f'$, d'après la Définition 4.16 nous avons

$$\forall\, i \in [1,\ldots,n],\ \tau^1[C/P_i] =_{E_i} \tau^2[C/P_i]$$

$$\text{et il existe un } j > 0, \text{ tel que } \tau^1[C/Q_j] \neq_{E'_j} \tau^2[C/Q_j].$$

Par conséquent nous avons aussi $\mathcal{T} \not\models f$ comme prévu.

A3 : Soit $f = (C, (\{P_1\,[E_1], \ldots, P_n\,[E_n]\} \rightarrow \{Q_1\,[E'_1], \ldots, Q_m\,[E'_m]\}))$, $f' = (C, (\{Q_1\,[E'_1], \ldots, Q_m\,[E'_m]\} \rightarrow S\,[E_s]))$ et $f'' = (C, (P_1\,[E_1], \ldots, P_n\,[E_n]\} \rightarrow S\,[E_s]))$. La preuve sera faite par contraposée : nous allons prouver que si $\mathcal{T} \not\models f''$ alors soit $\mathcal{T} \not\models f'$ ou soit $\mathcal{T} \not\models f$. Posons $P_{n+1} = S$ et $E_{n+1} = E_s$. Puisque $\mathcal{T} \not\models f''$ alors d'après la Définition 4.16, il existe deux projections τ^1 et τ^2 pour la branche $M = \{C/P_1, \ldots, C/P_{n+1}\}$ dans $\mathcal{T}$ telles que

$$\tau^1[C/P_1, \ldots, C/P_n] =_{E_i, i\in[1\ldots n]} \tau^2[C/P_1, \ldots, C/P_n]$$

$$\text{et } \tau^1[C/P_{n+1}] \neq_{E_{n+1}} \tau^2[C/P_{n+1}].$$

Nous allons étendre la branche $M = \{C/P_1, \ldots, C/P_{n+1}\}$ pour obtenir $M' = \{C/P_1, \ldots, C/P_{n+1}, C/Q_1, \ldots, C/Q_m\}$. Après plusieurs applications du Lemme A.12 nous pouvons construire des projections u^1 à partir de τ^1 et u^2 à partir de τ^2. En considérant les projections u^1 et u^2 nous pouvons obtenir l'un des deux situations suivantes :

1. $u^1[C/Q_1, \ldots, C/Q_m] =_{E'_i, i\in[1\ldots m]} u^2[C/Q_1, \ldots, C/Q_m]$ et dans ce cas nous avons $\mathcal{T} \not\models f'$ parce que $u^1[C/P_{n+1}] \neq_{E_{n+1}} u^2[C/P_{n+1}]$,
2. ou il y a un $j \in [1 \ldots m]$ tel que $u^1[C/Q_j] \neq_{E'_j} u^2[C/Q_j]$. Dans ce cas $\mathcal{T} \not\models f$.

Nous pouvons conclure que l'axiome $A3$ est correcte.

A4 : Soit $f = (C, (\{P'_1\,[E'_1], \ldots, P'_n\,[E'_n]\} \rightarrow P_{n+1}\,[E_{n+1}]))$ et $f' = (C, (\{P_1\,[E_1], \ldots, P_n\,[E_n]\} \rightarrow P_{n+1}\,[E_{n+1}]))$. La preuve sera faite par contradiction. Supposons que $\mathcal{T} \models f$ mais que $\mathcal{T} \not\models f'$. D'après l'Axiome A1, nous pouvons affirmer que pour tout $i \in [1 \ldots n]$, $P_i \neq P_{n+1}$. Puisque $\mathcal{T} \not\models f'$ alors d'après la Définition 4.16, il existe deux projections τ^1 et τ^2 pour la branche $M = \{C/P_1, \ldots, C/P_{n+1}\}$ dans $\mathcal{T}$ telles que $\tau^1[C/P_1, \ldots, C/P_n] =_{E_i, i\in[1\ldots n]} \tau^2[C/P_1, \ldots, C/P_n]$ et $\tau^1[C/P_{n+1}] \neq_{E_{n+1}} \tau^2[C/P_{n+1}]$. Nous allons montrer qu'il existe deux projections u^1 et u^2, construits à partir de τ^1 et τ^2, pour la branche $M' = \{C/P'_1, \ldots, C/P'_n, C/P_{n+1}\}$ dans $\mathcal{T}$ telles que :

$$u^1[C/P'_1, \ldots, C/P'_n] =_{E'_i, i\in[1\ldots n]} u^2[C/P'_1, \ldots, C/P'_n] \text{ et} \tag{A.5}$$

$$u^1[C/P_{n+1}] \neq_{E_{n+1}} u^2[C/P_{n+1}]. \tag{A.6}$$

Cependant d'après notre hypothèse, nous savons que si pour tous deux projections u^1 et u^2 telles que (A.5) est satisfaite alors nous avons $u^1[C/P_{n+1}] =_{E_{n+1}} u^2[C/P_{n+1}]$. Si u^1 et u^2 existent vraiment alors nous avons une contradiction avec (A.6) et l'Axiome A4 sera satisfait.

La preuve va consister à montrer qu'il est possible d'avoir deux projections pour M' satisfaisant (A.5) et (A.6). Nous allons considérer que $u^1[C/P_i] = \tau^1[C/P_i]$ et $u^2[C/P_i] = \tau^2[C/P_i]$ $\forall i \in [1 \dots n+1]$.

1. Si $\exists k \in [1 \dots n]$ tel que $P_k \preceq P_{n+1}$ (Figure A.5(a)) alors en se servant du Lemme A.13, nous avons :

$$Last(u^1[C/P_k]) =_N Last(u^2[C/P_k]). \tag{A.7}$$

   Puisque t est complet alors il existe des instances J_i telles que $\forall i \in [1 \dots n]$, $u^1[C/P_i] \preceq J_i$ et $J_i \in Instances(C/P'_i, t)$ (cf. Figure A.5(a)). $\forall i \in [1 \dots n]$, posons $u^1[C/P'_i] = u^2[C/P'_i] = J_i$. En considérant ces instances J_i pour les chemins C/P'_i et en utilisant le Lemme A.11, nous pouvons affirmer que $\forall i, j \in [1 \dots n+1]$ (rappelons que nous avons P'_{n+1}=P_{n+1}) :

$$isInst_lcp(C/P'_i, u^1[C/P'_i], C/P'_j, u^1[C/P'_j]) = vrai \tag{A.8}$$

$$\text{et } isInst_lcp(C/P'_i, u^2[C/P'_i], C/P'_j, u^2[C/P'_j]) = vrai. \tag{A.9}$$

   Ainsi dans ce cas, nous avons bien des projections u^1 et u^2 satisfaisant A.5 et A.6.

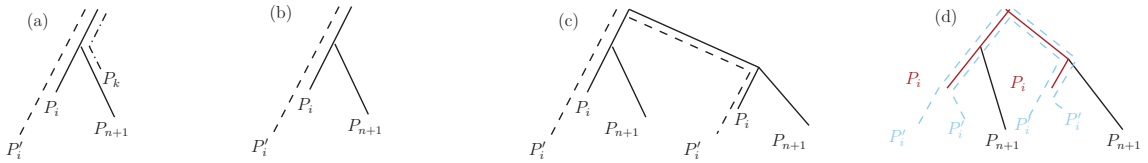


FIGURE A.5 – *Représentation graphique des chemins et des projections possibles. Cas (a) : Chemin $P_k \prec P_{n+1}$ où P_k est l'un des chemins de la partie gauche de f' et $P_i \preceq P'_i$. Cas (b) : Identité de nœud pour les derniers nœuds de P_i. Les deux projections u^1 et u^2 ont les mêmes instances pour le chemin P'_i. Cas (c) et (d) : Égalité par valeur pour les derniers nœuds de P_i. Dans le cas (c) il y a qu'une seule instance de P'_i ($P_i \preceq P'_i$). Dans le cas (d) il y a deux instances de P'_i ($P_i \preceq P'_i$).*

2. Sinon lorsque pour tout $i \in [1 \dots n]$, $P_i \not\preceq P_{n+1}$ et $P_i \preceq P'_i$, nous pouvons considérer les situations suivantes :
   (a) si nous considérons l'identité de nœud, nous avons $Last(u^1[C/P_i]) =_N Last(u^2[C/P_i])$ (Figure A.5(b)). Puisque t est complet alors il existe une instance J_i telle que $u^1[C/P_i] \preceq J_i$ et $J_i \in Instances(C/P'_i, t)$. Posons $u^1[C/P'_i] = u^2[C/P'_i] = J_i$;
   (b) si nous considérons l'égalité par valeur, nous avons $Last(u^1[C/P_i]) =_V Last(u^2[C/P_i])$. Puisque t est complet alors ils existe des instances J^1_i, J^2_i telles que $u^1[C/P_i] \preceq J^1_i$, $u^2[C/P_i] \preceq J^2_i$ et $J^1_i, J^2_i \in Instances(C/P'_i, t)$.
      - si $Last(J^1_i) =_V Last(J^2_i)$ alors posons $u^1[C/P'_i] = J^1_i$ et $u^2[C/P'_i] = J^2_i$ (Figure A.5(c)) ;

- sinon si $Last(J_i^1) \neq_V Last(J_i^2)$ alors puisque $P_i \preceq P'_i$ et $Last(u^1[C/P_i]) =_V Last(u^2[C/P_i])$, il existe deux instances J_i^3, J_i^4 telles que $u^1[C/P_i] \preceq J_i^3$, $u^2[C/P_i] \preceq J_i^4$ et $J_i^3, J_i^4 \in Instances(C/P'_i, t)$, $Last(J_i^1) =_V Last(J_i^4)$ et $Last(J_i^2) =_V Last(J_i^3)$. Dans ce cas, posons $u^1[C/P'_i] = J_i^1$ et $u^2[C/P'_i] = J_i^4$ (Figure A.5(d)).

Ensuite en considérant ces instances J_i pour les chemins C/P'_i et en utilisant le Lemme A.11, nous pouvons affirmer que nous avons (A.8) et (A.9) dans chaque cas. Puisque u^1 et u^2 existe et les conditions (A.5), (A.6) sont satisfaites, nous pouvons conclure que A4 est correct.

A5 : Soit $f = (C, (P[N] \rightarrow Q[N]))$. La preuve sera faite par contradiction. Supposons que $\mathcal{T} \not\models f$. D'après la Définition 4.16 il existe deux projections τ_1 et τ_2 pour la branche $M = \{C/P, C/Q\}$ dans $\mathcal{T}$ telles que $\tau^1[C/P] =_N \tau^2[C/P]$ et $\tau^1[C/Q] \neq_N \tau^2[C/Q]$. Cependant, puisque $\tau^1[C/P] =_N \tau^2[C/P]$ nous avons grâce à l'identité de nœud que l'instance du chemin C/P est la même dans τ_1 et τ_2. Et par conséquent puisque C/Q est un préfixe de C/P, le chemin C/Q a aussi la même instance dans τ_1 et τ_2. Nous concluons donc que $\tau^1[C/Q] =_N \tau^2[C/Q]$ qui est contradiction avec notre hypothèse $\tau^1[C/Q] \neq_N \tau^2[C/Q]$.

A6 : Soit $f = (C, (Parent(P)[E] \rightarrow P[E]))$. La preuve sera faite par contradiction. Supposons que $\mathcal{T} \not\models f$. D'après la Définition 4.16 il existe deux projections τ_1 et τ_2 pour la branche $M = \{C/P, C/Parent(P)\}$ dans $\mathcal{T}$ telles que $\tau^1[C/Parent(P)] =_E \tau^2[C/Parent(P)]$ et $\tau^1[C/P] \neq_E \tau^2[C/P]$. Cependant, puisque $Parent(P) \prec P$ et $\mathcal{T} \not\models f$ alors en appliquant le Lemme A.13 on obtient que l'instance du chemin $C/Parent(P)$ est la même dans τ_1 et τ_2. D'après la définition d'un arbre XML nous savons que l'instance de $C/Parent(P)$ dans τ_1 et τ_2 n'a qu'un seul attribut pour le label $Last(P)$ et donc l'instance du chemin C/P est aussi la même dans τ_1 et τ_2. Ainsi $\tau^1[C/P] =_E \tau^2[C/P]$ qui est contradiction avec notre hypothèse $\tau^1[C/P] \neq_E \tau^2[C/P]$.

A7 : Puisque le nœud contexte est unique, cet axiome est automatiquement satisfait.

A8 : Soit $f = (C, (\{P_1[E_1], \ldots, P_n[E_n]\} \rightarrow P_{n+1}[E_{n+1}]))$ et $f' = (C/Q, (\{P'_1[E_1], \ldots, P'_n[E_n]\} \rightarrow P'_{n+1}[E_{n+1}]))$. La preuve sera faite par contraposée. Nous allons prouver que si $\mathcal{T} \not\models f'$ alors $\mathcal{T} \not\models f$. Supposons que $\mathcal{T} \not\models f'$. D'après la Définition 4.16 nous déduisons qu'il existe deux projections τ_1 et τ_2 pour la branche $M = \{C/Q/P'_1, \ldots, C/Q/P'_{n+1}\}$ dans $\mathcal{T}$ telles que

$$\tau^1[C/Q/P'_1, \ldots, C/Q/P'_n] =_{E_i, i \in [1 \ldots n]} \tau^2[C/Q/P'_1, \ldots, C/Q/P'_n]$$

$$\text{et } \tau^1[C/Q/P'_{n+1}] \neq_{E_{n+1}} \tau^2[C/Q/P'_{n+1}].$$

Puisque $P_1 = Q/P'_1, \ldots, P_{n+1} = Q/P'_{n+1}$ alors τ_1 et τ_2 sont aussi des projections pour la branche $M = \{C/P_1, \ldots, C/P_{n+1}\}$. En considérant les projections τ_1 et τ_2, nous avons $\tau^1[C/P_{n+1}] \neq_{E_{n+1}} \tau^2[C/P_{n+1}]$. Ainsi on obtient $\mathcal{T} \not\models f$.

A9 : Puisque deux nœuds égaux par identité de nœuds ont également les mêmes valeurs, alors la dépendance $(C, (P[N] \rightarrow P[V]))$ est toujours vraie.

◁

Comme nous venons de prouver que les axiomes A1-A9 sont corrects, nous pouvons les utiliser pour prouver que les axiomes A10-A13 sont aussi corrects.

Théorème A.4 *Les Axiomes A10-A13 sont corrects pour les XFD dans les arbres XML complets.*

Démonstration. Nous supposons un arbre XML $\mathcal{T}$ qui est complet par rapport à $\mathbb{P}$.

A10 : Soit $f_1 = (C, (\{P_1\,[E_1], \ldots, P_n\,[E_n]\} \rightarrow Q\,[E_{n+1}]))$. Nous pouvons augmenter f_1 avec $\{C/P_1, \ldots, C/P_n\}$ en utilisant l'axiome A2 pour dériver $f_1' = (C, (\{P_1\,[E_1], \ldots, P_n\,[E_n]\} \rightarrow \{P_1\,[E_1], \ldots, P_n\,[E_n], Q\,[E_{n+1}]\}))$.
Soit $f_2 = (C, (\{P_1\,[E_1], \ldots, P_n\,[E_n]\} \rightarrow R\,[E_{n+2}]))$. Nous pouvons augmenter f_2 avec le chemin C/Q en utilisant l'axiome A2 pour dériver $f_2' = (C, (\{P_1\,[E_1], \ldots, P_n\,[E_n], Q\,[E_{n+1}]\} \rightarrow \{Q\,[E_{n+1}], R\,[E_{n+2}]\}))$.
A partir de f_1', f_2' et en utilisant l'axiome A3 on peut dériver $(C, (\{P_1\,[E_1], \ldots, P_n\,[E_n]\} \rightarrow \{Q\,[E_{n+1}], R\,[E_{n+2}]\}))$.

A11 : Soit $f = (C, (\{P_1\,[E_1], \ldots, P_n\,[E_n]\} \rightarrow \{Q_1\,[E_1'], \ldots, Q_m\,[E_m']\}))$. Puisque $\{R_1, \ldots, R_k\} \subseteq \{Q_1, \ldots, Q_m\}$, alors en utilisant l'axiome A1 et A10 on obtient $f' = (C, (\{Q_1\,[E_1'], \ldots, Q_m\,[E_m']\} \rightarrow \{R_1\,[E_1'], \ldots, R_k\,[E_k'']\}))$. A partir de f, f' et en utilisant l'axiome A3 on peut dériver $(C, (\{P_1\,[E_1], \ldots, P_n\,[E_n]\} \rightarrow \{R_1\,[E_1''], \ldots, R_k\,[E_k'']\}))$.

A12 : Soit $f_1 = (C, (\{P_1\,[E_1], \ldots, P_n\,[E_n]\} \rightarrow \{Q_1\,[E_1'], \ldots, Q_m\,[E_m']\}))$. Nous pouvons augmenter f_1 avec $\{C/R_1, \ldots, C/R_k\}$ en utilisant l'axiome A2 pour dériver $f_1' = (C, (\{P_1\,[E_1], \ldots, P_n\,[E_n], R_1\,[E_1''], \ldots, R_k\,[E_k'']\} \rightarrow \{Q_1\,[E_1'], \ldots, Q_m\,[E_m'], R_1\,[E_1''], \ldots, R_k\,[E_k'']\}))$.
Soit $f_2 = (C, (\{Q_1\,[E_1'], \ldots, Q_m\,[E_m'], R_1\,[E_1''], \ldots, R_k\,[E_k'']\} \rightarrow S\,[E_s]))$. A partir de f_1', f_2 et en utilisant l'axiome A3 nous dérivons $(C, (\{P_1\,[E_1], \ldots, P_n\,[E_n], R_1\,[E_1''], \ldots, R_k\,[E_k'']\} \rightarrow S\,[E_s]))$.

A13 : Soit $f = (C, (\{P_1\,[E_1], \ldots, P_n\,[E_n]\} \rightarrow P_{n+1}\,[E_{n+1}]))$. Avec l'axiome A7 nous avons $\forall\ C/P_0$, $f' = (C, (P_0\ [E_0] \rightarrow []\ [E]))$. $\forall i \in [1, \ldots, n]$ soit $P_i' = []$, nous avons $P_i \cap P_{n+1} = []$ et $[] \preceq_{PL} P_i' \preceq_{PL} P_i$ et en utilisant l'axiome A4 sur la XFD f on obtient $f'' = (C, ([]\ [E] \rightarrow P_{n+1}\,[E_{n+1}]))$. Finalement en utilisant l'axiome A3 sur f' et f'', nous obtenons $(C, (P_0\ [E_0] \rightarrow P_{n+1}\,[E_{n+1}]))$.

◁

A.5 Système d'axiomes : Preuve de complétude

La preuve de complétude des axiomes d'Armstrong dans le modèle relationnel est basée sur une relation particulière r ayant deux tuples de valeurs. Cette relation particulière est définie telle que pour un ensemble d'attribtut V inclus dans l'ensemble des attributs de r,

- les valeurs des deux tuples pour les attributs qui sont dans V^+ sont égaux et
- les valeurs des deux tuples pour les attributs qui sont pas dans V^+ sont différents.

Nous allons utiliser la même approche en définissant un arbre XML particulier avec ces mêmes propriétés. Nous souhaitons donc construire un arbre XML possédant deux instances (sauf pour la racine) pour chaque chemin $P \in \mathbb{P}$. Cependant l'exemple ci-dessous montre que selon certaines conditions imposées aux chemins, il n'est pas possible d'avoir *deux instances* pour *chaque chemin* P dans $\mathbb{P}$.

Exemple A.5 Nous voulons construire un arbre complet ayant exactement deux instances pour chaque chemin dans $\mathbb{P}$. Considérons l'égalité par valeur et deux chemins C/P et C/Q tels que $P \prec Q$. Soient I_{P_1} et I_{P_2} les deux instances de C/P dans l'arbre t. Et soient I_{Q_1} et I_{Q_2} les deux instances de C/Q dans l'arbre t. Supposons que $Last(I_{P_1}) =_V Last(I_{P_2})$ et $Last(I_{Q_1}) \neq_V Last(I_{Q_2})$. Dans cette situation, la dépendance fonctionnelle $(C, (P \rightarrow Q))$ n'est pas satisfaite par cet arbre. Nous pouvons donc appliquer le Lemme A.13 pour conclure qu'il existe une instance de P qui est un préfixe de deux instances du chemin Q. Comme

nous voulons deux instances pour chaque chemin, pour avoir deux instances de Q il faut que $Last(I_{P_1}) =_N Last(I_{P_2})$. Par conséquent nous ne pouvons pas avoir dans l'arbre deux instances pour P. En effet dans cette situation, la Figure A.6 illustre le fait qu'un arbre ayant deux instances de P et respectant les contraintes $Last(I_{P_1}) =_V Last(I_{P_2})$ et $Last(I_{Q_1}) \neq_V Last(I_{Q_2})$, doit avoir quatres instances de Q.

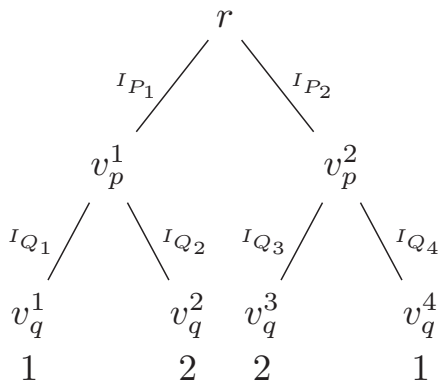


FIGURE A.6 – *Un arbre XML montrant l'impossibilité d'avoir toujours deux instances pour tous chemins P et Q tels que $P \prec Q$ et $Last(I_{P_1}) =_V Last(I_{P_2})$ et $Last(I_{Q_1}) \neq_V Last(I_{Q_2})$*

Maintenant supposons que l'identité de nœud est une condition imposée sur les deux instances d'un chemin P. Nous aurons donc $Last(I_{P_1}) =_N Last(I_{P_2})$, ce qui fait que dans cette situation P ne peut avoir qu'une seule instance.

⊠

En nous basant sur les remarques de l'Exemple A.5, nous allons maintenant définir notre arbre spécial qui aura donc *au plus* deux instances pour chaque chemin dans $\mathbb{P}$.

Définition A.4 (Arbre bi-instance)
Soit $\mathcal{F}$ un ensemble de XFD. Soit $\mathcal{T} = (t, type, value)$ un document XML où l'arbre t construit à partir des propriétés ci-dessous, est appelé un *arbre bi-instance*. Soient $\mathbb{P}$ l'ensemble des chemins simples associés à $\mathcal{T}$, $C/X \subseteq \mathbb{P}$ et $\vec{E}$ les types d'égalités associés à X. Par $|Instances(C/P, t)|$, nous désignons le nombre d'instances du chemin C/P dans t.
PROPRIÉTÉES DE CONSTRUCTION :

1. Pour chaque $C/P \in \mathbb{P}$, $|Instances(C/P, t)|$ est au plus égale à 2 (Soient I_1 et I_2 ces instances) et lorsque $|Instances(C/P, t)| = 2$:
   (a) nous avons $C/P[V] \in (C, X[\vec{E}])^+$ ssi $Last(I_1) =_V Last(I_2)$;
   (b) nous avons $C/P[E'] \notin (C, X[\vec{E}])^+$ ssi $Last(I_1) \neq_{E'} Last(I_2)$.
2. Pour chaque $C/P \in \mathbb{P}$, $|Instances(C/P, t)| = 2$ sauf dans les cas suivants :
   (a) le chemin C/P est le contexte ($P = []$), ou
   (b) $(C, (X\,[\vec{E}] \rightarrow P\,[N]))$ ou
   (c) en considérant l'égalité par valeur, $|Instances(C/P, t)| = 2$ provoque la violation de la condition 1 pour un autre chemin $C/Q \in \mathbb{P}$ tel que $P \prec Q$, ou
   (d) $Last(P) \in \Sigma_{att}$ et $Parent(C/P)$ vérifie les conditions 2a, ou 2b, ou 2c.

□

Lemme A.14 *Soit $\mathcal{F}$ un ensemble de XFD, soit $\mathcal{T} = (t, type, value)$ un document XML où t est un arbre bi-instance. Soit $\mathbb{P}$ l'ensemble de chemins associés à $\mathcal{T}$ et soit $C/X \subseteq \mathbb{P}$ et $\vec{E}$ les types d'égalités associés à X. Les propriétés suivantes sont vérifiées pour t :*

1. *si* $C/P \in \mathbb{P}$ *et* $|Instances(C/P,t)| = 1$ *alors* $C/P[E'] \in (C, X[\vec{E}])^+$.
2. *si* $P, Q \in \mathbb{P}$, $P \preceq Q$ *et il existe une instance* $I_P \in Instances(P,t)$, *et deux instances* I_{Q_1} *et* $I_{Q_2} \in Instances(Q,t)$ *telles que* $I_P \preceq I_{Q_1}$ *et* $I_P \preceq I_{Q_2}$ *alors* $|Instances(P,t)| = 1$.
3. *si* $C/P \in \mathbb{P}$ *alors* $C/P[E'] \in (C, X[\vec{E}])^+$ *ssi* $\mathcal{T} \models (C, (X[\vec{E}] \rightarrow P[E']))$.

Démonstration.

1. Selon la Définition A.4, lorsque $|Instances(C/P,t)| = 1$ alors nous sommes dans l'une des conditions 2a-2d.

   En premier considérons le cas 2a et supposons que $P = []$. En appliquant l'axiome A7 (Unicité du contexte) nous avons $\forall P_i \in X$, $(C, (P_i[E_i] \rightarrow P[E']))$ et donc $P[E'] \in (C, X[\vec{E}])^+$.

   En second lieu considérons le cas 2b et supposons que $(C, (X[\vec{E}] \rightarrow P[N]))$. En appliquant l'axiome A9 (De l'identité par nœud à l'égalité par valeur) nous avons $(C, (P[N] \rightarrow P[V]))$ et par transitivité (A3) nous obtenons $(C, (X[\vec{E}] \rightarrow P[V]))$. Ainsi nous obtenons aussi dans ce cas que $P[E'] \in (C, X[\vec{E}])^+$.

   En troisième lieu considérons le cas 2c et supposons que $P \neq []$ et que $Last(P) \notin \Sigma_{att}$. Soit I_{P_1} l'instance de P dans t. Nous allons faire une preuve par induction sur le nombre de chemins $Q_1, \ldots, Q_n$ ($\forall i \in [1 \ldots n], C/Q_i \in \mathbb{P}$ et $P \prec Q_i$) qui impose à C/P d'avoir qu'une seule instance dans t (comme indiqué par la condition 2c de la Définition A.4). Nous avons les situations suivantes pour $n = 1$ (seul C/Q_1 impose à C/P d'avoir une seule instance dans t) :

   (a) Si $|Instances(C/Q_1,t)| = 1$ alors selon la Définition A.4(2), soit $(C, (X[\vec{E}] \rightarrow Q_1[N]))$ ou soit il existe un chemin Q' qui impose au chemin Q_1 d'avoir une seule instance. Dans le premier cas puisque $(C, (X[\vec{E}] \rightarrow Q_1[N]))$ et en appliquant l'axiome A5 (Unicité des Ascendants) on a $(C, (Q_1[N] \rightarrow P[N]))$ alors par transitivité on obtient $(C, (X[\vec{E}] \rightarrow P[N]))$. En appliquant l'axiome A9 (De l'identité par nœud à l'égalité par valeur) nous avons $(C, (P[N] \rightarrow P[V]))$ et par transitivité (A3) nous obtenons $(C, (X[\vec{E}] \rightarrow P[V]))$. Ainsi nous obtenons que $C/P[E'] \in (C, X[\vec{E}])^+$. Dans le deuxième cas puisque $P \prec Q_1$ alors le chemin Q' impose aussi à P d'avoir qu'une seule instance dans t. Ce cas n'est donc pas possible puisque Q_1 est supposé être le seul chemin qui impose à P d'avoir qu'une seule instance dans t.

   (b) Si $|Instances(C/Q_1,t)| = 2$ et $C/Q_1[E''] \in (C, X[\vec{E}])^+$ alors soient $I_{Q_{1,1}}$ et $I_{Q_{1,2}}$ les deux instances de Q_1. Supposons que le sous-arbre concernant l'instance de P ait le format illustré dans la Figure A.7(a). Comme $C/Q_1[E''] \in (C, X[\vec{E}])^+$, nous avons $Last(I_{Q_{1,1}}) =_V Last(I_{Q_{1,2}})$. Afin d'assurer que t est un arbre bi-instance, nous allons vérifier s'il n'est pas possible d'avoir un arbre bi-instance avec deux instances de P et qui respecte les conditions imposées sur Q_1. Considérons l'arbre t' de la Figure A.7(b) identique à t sauf pour le sous-arbre concernant l'instance de P. L'arbre t' a deux instances de P et deux instances de Q_1 ($I'_{Q_{1,1}}$ et $I'_{Q_{2,1}}$) telles que $Last(I'_{Q_{1,1}}) =_V Last(I'_{Q_{2,1}}) =_V Last(I_{Q_{1,1}})$. Notons que l'arbre t' est un arbre bi-instance même si $C/P[E'] \in (C, X[\vec{E}])^+$ ou $C/P[E''] \notin (C, X[\vec{E}])^+$. Si $C/P[E'] \in (C, X[\vec{E}])^+$ alors les deux instances I'_{P_1} et I'_{P_2} de P dans t' sont telles que $Last(I'_{P_1}) =_V Last(I'_{P_2})$. Sinon si $C/P[E'] \notin (C, X[\vec{E}])^+$ alors pour les deux instances on a $Last(I'_{P_1}) \neq_V Last(I'_{P_2})$. Ainsi t' est un arbre bi-instance. L'existance d'un arbre bi-instance t' ayant deux instances de P prouve que lorsque $C/Q_1[E''] \in (C, X[\vec{E}])^+$ et $|Instances(C/Q_1,t)| = 2$, l'arbre t ne respecte pas la

condition 2c de la Définition A.4. Ainsi t n'est pas un arbre bi-instance. En résumé il n'est pas possible d'avoir $C/Q_1[E''] \in (C, X[\vec{E}])^+$ avec deux instances de Q_1 et que Q_1 soit le chemin qui impose à P d'avoir une seule instance.

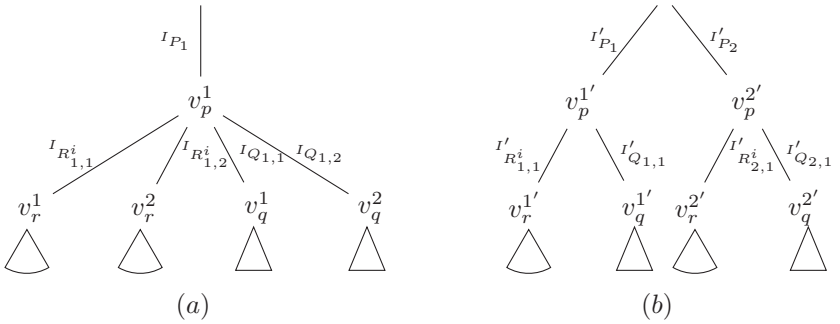


FIGURE A.7 – *Construction d'un nouveau arbre bi-instance t' avec $Q_1[E''] \in (C, X[\vec{E}])^+$*

(c) Si $|Instances(C/Q_1, t)| = 2$ et $C/Q_1[E''] \notin (C, X[\vec{E}])^+$ alors soient $I_{Q_{1,1}}$ et $I_{Q_{1,2}}$ les deux instances de Q_1. Selon la Définition A.4, nous avons $Last(I_{Q_{1,1}}) \neq_V Last(I_{Q_{1,2}})$. Supposons que le sous-arbre concernant l'instance de P ait le format illustré dans la Figure A.8(a). Afin d'assurer que t est un arbre bi-instance, nous allons vérifier s'il n'est pas possible d'avoir un arbre bi-instance avec deux instances de P et qui respecte les conditions imposées sur Q_1. Nous avons :

- Premièrement considérons que $C/P[E'] \notin (C, X[\vec{E}])^+$ et que l'arbre t' de la Figure A.8(b) est identique à t sauf pour le sous-arbre concernant l'instance de P dans la Figure A.8(a). Avec les mêmes arguments que dans le cas 1b, l'existence d'un arbre bi-instance t' contredit le fait que t soit un arbre bi-instance.
- Finallement considérons que $C/P[E'] \in (C, X[\vec{E}])^+$. Puisque $C/Q_1[E''] \notin (C, X[\vec{E}])^+$, alors en appliquant le Lemme A.13, nous pouvons déduire qu'un arbre avec deux instances de P est l'arbre t' de la Figure A.8(c) identique à t sauf pour le sous-arbre concernant l'instance de P dans la Figure A.8(a). En effet t' n'est pas un arbre bi-instabce parce que $|Instances(C/Q_1, t)| = 4$. Par conséquent le fait que t est un arbre bi-instance est justifié et donc nous avons $C/P[E'] \in (C, X[\vec{E}])^+$.

En résumé les seules situations qui justifient le fait que P ne possède qu'une seule instance dans t, est lorsqu'il existe un chemin $C/Q_1 \in \mathbb{P}$ tel que $P \prec Q_1$ et :

(i) $|Instances(C/Q_1, t)| = 1$ et $C/Q_1[N] \in (C, X[\vec{E}])^+$, ou

(ii) $|Instances(C/Q_1, t)| = 2$ et $C/Q_1[V] \notin (C, X[\vec{E}])^+$.

Ainsi $|Instances(C/P, t)| = 1$ et nous avons $C/P[E'] \in (C, X[\vec{E}])^+$.

Maintenant supposons que $\forall\, m < n$, les chemins $C/Q_1, \ldots, C/Q_m$ imposent à C/P d'avoir qu'une seule instance dans t et $\forall\, i \in [1 \ldots m]$, $P \prec Q_i$, (i) $|Instances(C/Q_i, t)| = 1$ et $C/Q_i[N] \in (C, X[\vec{E}])^+$ ou (ii) $|Instances(C/Q_i, t)| = 2$ et $C/Q_i[V] \notin (C, X[\vec{E}])^+$, et on a $C/P[E'] \in (C, X[\vec{E}])^+$. Nous allons prouver que $C/P[E'] \in (C, X[\vec{E}])^+$ lorsque les chemins $C/Q_1, \ldots, C/Q_n$ imposent à C/P d'avoir une seule instance dans t. Nous avons :

- s'il existe un chemin R tel que $Parent(R) = P$ et $|Instances(C/R, t)| = 1$ alors il existe un chemin Q' qui impose à R d'avoir une seule instance. Par hypothèse

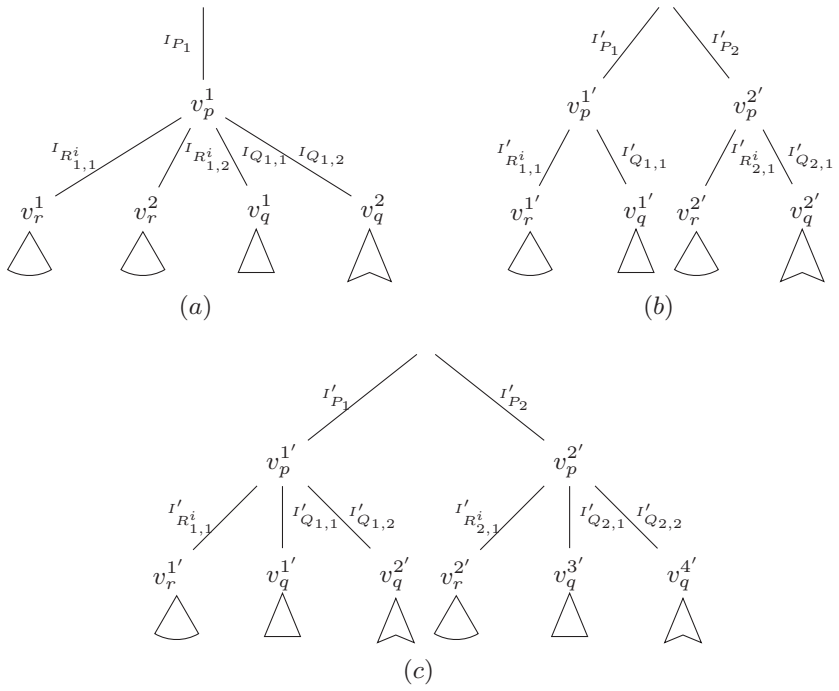


FIGURE A.8 – *Construction d'un nouveau arbre bi-instance t' avec $Q_1[E''] \notin (C, X[\vec{E}])^+$*

d'induction $R \prec Q'$, (i) $|Instances(C/Q', t)| = 1$ et $C/Q'[N] \in (C, X[\vec{E}])^+$ ou (ii) $|Instances(C/Q', t)| = 2$ et $C/Q'[V] \notin (C, X[\vec{E}])^+$, et on a $C/R[E'] \in (C, X[\vec{E}])^+$. Pour les mêmes raisons que pour les situations 1a et 1c ci-dessus nous pouvons conclure qu'à cause de Q', $C/P[E'] \in (C, X[\vec{E}])^+$.

– Sinon nous sommes dans la situation où pour tous les chemins R_i fils de P, $|Instances(C/R_i, t)| = 2$. Nous pouvons distinguer les deux cas suivants :
  - pour tous les chemins R_i nous avons $C/R_i[V] \in (C, X[\vec{E}])^+$. Cette situation est similaire à la situation 1b. Par conséquent nous pouvons construire un arbre avec deux instances de P qui soit una arbre bi-instance respectant les contraintes sur les chemins R_i. Ainsi l'arbre t (avec une seule instance de P) considéré au début n'est pas un arbre bi-instance.
  - il existe un chemin R_j tel que $C/R_j[V] \notin (C, X[\vec{E}])^+$. Cette situation est similaire à la situation 1c. Par conséquent nous ne pouvons pas construire un arbre avec deux instances de P qui soit un arbre bi-instance respectant les contraintes sur le chemin R_j. Ainsi nous concluons que $C/P[E'] \in (C, X[\vec{E}])^+$.

En dernier considérons le cas 2d de la Définition A.4. Nous supposons que $Last(P) \in \Sigma_{att}$. Nous allons en premier prouver par contradiction que $|Instances(C/Parent(P), t)| = 1$. Supposons que $|Instances(C/Parent(P), t)| \neq 1$. Par conséquent $|Instances(C/Parent(P), t)| = 2$ puisque t est un arbre bi-instance. Et comme l'arbre t est un arbre complet, chacun des deux instances de $Parent(P)$ doit avoir un nœud attribut $Last(P)$ et donc $|Instances(C/P, t)| = 2$. Nous obtenons une contradiction avec l'hypothèse $|Instances(C/P, t)| = 1$. En conclusion l'arbre t ne possède qu'une seule instance pour le chemin $Parent(P)$. Notons que puisque $Last(Parent(P)) \in \Sigma_{ele}$, nous avons déjà prouvé que $C/Parent(P)[E'] \in (C, X[\vec{E}])^+$.

En appliquant l'axiome A6 (Unicité de l'attribut) nous obtenons $(C, (Parent(P)\,[E'] \rightarrow P\,[E']))$ et nous pouvons conclure que $C/P[E'] \in (C, X[\vec{E}])^+$.

2. La preuve sera faite par contradiction et nous supposons que $|Instances(P,t)| \neq 1$. En considérant l'arbre t, $|Instances(P,t)|$ doit être égale à 2. Par hypothèse, nous savons qu'il existe une instance I_P de P telle que $I_P \preceq I_{Q_1}$ et $I_P \preceq I_{Q_2}$. A présent soit I'_P la seconde instance de P. Puisque t est complet alors il existe une instance $I_{Q_3} \in Instances(Q,t)$ telle que $I_{P'} \preceq I_{Q_3}$ et donc $|Instances(Q,t)| \geq 3$. Il y a donc une contradiction avec le fait que l'arbre bi-instance t a exactement un ou deux instances pour chaque chemin dans $\mathbb{P}$. Cette propriété est illustrée dans la Figure A.9.

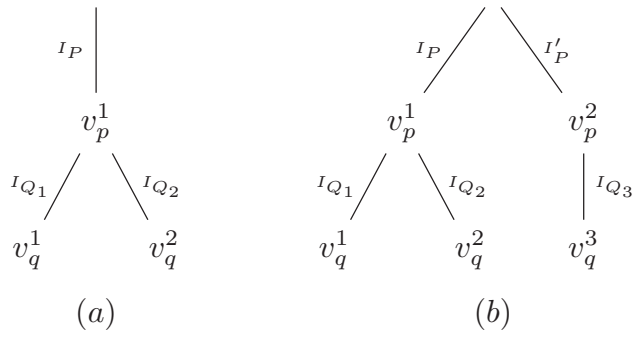


FIGURE A.9 – *L'ajout d'une instance de P dans l'arbre de (a) implique $|Instances(Q,t)| \geq 3$ pour l'arbre de (b)*

3. $(\rightarrow)$: Selon la Définition A.4 et le Lemme A.14(1) nous avons : si $C/P[E'] \in (C, X[\vec{E}])^+$ alors $|Instances(C/P,t)| = 1$ ou $|Instances(C/P,t)| = 2$ et il existe I_1, I_2 instances du chemin C/P telles que $I_1 =_V I_2$. Puisque $C/X[\vec{E}] \subseteq (C, X[\vec{E}])^+$, nous avons aussi que pour tout $P_i \in X$, $|Instances(C/P_i,t)| = 1$ ou $|Instances(C/P_i,t)| = 2$ et il existe I_1^i, I_2^i instances du chemin C/P_i telles que $I_1^i =_V I_2^i$. Ainsi en se référant à la Définition 4.16 nous pouvons conclure que $\mathcal{T} \models (C, (X\,[\vec{E}] \rightarrow P\,[E']))$.
$(\leftarrow)$: Si $|Instances(C/P,t)| = 1$ alors en utilisant le Lemme A.14(1) nous avons $C/P[E'] \in (C, X[\vec{E}])^+$. Sinon si $|Instances(C/P,t)| = 2$, nous savons qu'à partir de l'hypothèse que les deux instances I_1, I_2 du chemin C/P sont telles que $I_1 =_V I_2$. Selon la Définition A.4 nous obtenons $C/P[E'] \in (C, X[\vec{E}])^+$.

$\triangleleft$

Nous avons maintenant le matériel nécessaire pour prouver que notre système d'axiomes est complet, c'est à dire qu'étant donné un ensemble de XFD $\mathcal{F}$ et à partir de notre système d'axiomes nous pouvons engendrer toutes les XFD f telle que $\mathcal{F} \models f$.

Théorème A.5 *Si $\mathcal{F} \models f$ alors $\mathcal{F} \vdash f$.*

Démonstration. Pour pouvoir atteindre notre but, nous allons prouver la contraposée :

$$\text{si } \mathcal{F} \nvdash f \text{ alors } \mathcal{F} \not\models f.$$

Soit $f = (C, (\{P_1\,[E_1], \ldots, P_n\,[E_n]\} \rightarrow \{P_{n+1}\,[E_{n+1}] \ldots P_{n+m}\,[E_{n+m}]\}))$. Nous posons $X = \{P_1, \ldots, P_n\}$, $Y = \{P_{n+1}, \ldots, P_{n+m}\}$, C/X, C/Y sont inclus dans $\mathbb{P}$, $\vec{E} = E_1, \ldots, E_n$ et $\vec{E_f} = E_{n+1}, \ldots, E_{n+m}$. Si $\mathcal{F} \not\models f$ alors il existe un document XML qui satisfait $\mathcal{F}$ mais qui ne satisfait pas f. La preuve va consister à prouver l'existence d'un tel document.

Nous allons supposer un document XML $\mathcal{T} = (t, type, value)$ où t est un arbre bi-instance défini sur l'ensemble de chemins $C/X = \{C/P_1, \ldots, C/P_n\}$, et montrer que $\mathcal{T}$ est l'un des documents que nous cherchons.

Fait 1 : $\mathcal{T} \models \mathcal{F}$.
Nous allons faire une preuve par contradiction. Supposons que $\mathcal{T} \not\models g$ où g est une XFD (C, ($\{Q_1\,[E'_1], \ldots, Q_k\,[E'_k]\} \rightarrow Q_{k+1}\,[E'_{k+1}]$)) dans $\mathcal{F}$. D'après la Définition 4.16, comme $\mathcal{T} \not\models g$ nous pouvons déduire qu'il existe deux projections $\tau^1 = \Pi^1_M(\mathcal{T})$ et $\tau^2 = \Pi^2_M(\mathcal{T})$ pour la branche $M = \{C/Q_1, \ldots, C/Q_{k+1}\}$ dans $\mathcal{T}$ telle que :

$$\tau^1[C/Q_1, \ldots, C/Q_k] =_{\vec{E'}} \tau^2[C/Q_1, \ldots, C/Q_k] \text{ avec } \vec{E'} = E'_1, \ldots, E'_k \text{ et} \tag{A.10}$$

$$\tau^1[C/Q_{k+1}] \neq_{E'_{k+1}} \tau^2[C/Q_{k+1}]. \tag{A.11}$$

A partir de (A.11) nous savons que $\Pi^1_M(\mathcal{T})[C/Q_{k+1}] \neq \Pi^2_M(\mathcal{T})[C/Q_{k+1}]$ et que $|Instances(Q_{k+1}, t)| = 2$. D'après la Définition A.4(1), nous obtenons :

$$C/Q_{k+1}[E'_{k+1}] \notin (C, X[\vec{E}])^+. \tag{A.12}$$

D'après la Définition 4.12, nous savons que les instances de chemins pour une même projection coïncident sur leur plus long préfixe commun. Formellement, pour chaque combinaison de chemins Q_i et Q_j telles que $1 \leq i \leq k+1$ et $1 \leq j \leq k+1$, nous avons :
En considérant $\Pi^1_M(\mathcal{T})$

$$isInst_lcp(C/Q_i, \Pi^1_M(\mathcal{T})[C/Q_i], C/Q_j, \Pi^1_M(\mathcal{T})[C/Q_j]) = vrai \text{ et} \tag{A.13}$$

En considérant $\Pi^2_M(\mathcal{T})$

$$isInst_lcp(C/Q_i, \Pi^2_M(\mathcal{T})[C/Q_i], C/Q_j, \Pi^2_M(\mathcal{T})[C/Q_j]) = vrai \tag{A.14}$$

et nous pouvons aussi déterminer les nœuds spéciaux suivants pour $1 \leq i \leq k$:

$$v^1_{i,k+1} = Last(\Pi^1_M(\mathcal{T})[C/Q_i] \cap \Pi^1_M(\mathcal{T})[C/Q_{k+1}]) \text{ et} \tag{A.15}$$

$$v^2_{i,k+1} = Last(\Pi^2_M(\mathcal{T})[C/Q_i] \cap \Pi^2_M(\mathcal{T})[C/Q_{k+1}]) \tag{A.16}$$

En se basant sur (A.13) et (A.14) avec la Définition 4.5 nous savons que les positions $v^1_{i,k+1}$ et $v^2_{i,k+1}$ existent dans t. Nous pouvons distinguer les deux cas suivants :

(a) $v^1_{i,k+1} = v^2_{i,k+1}$

(b) $v^1_{i,k+1} \neq v^2_{i,k+1}$

et pour chaque valeur de i dans $[1 \ldots k]$ nous allons choisir un chemin $C/R_i \in \mathbb{P}$ respectant la propriété suivante qui est la condition nécessaire pour appliquer l'axiome A4 (Branche préfixe) :

$$C/R_i[E'_i] \in (C, X[\vec{E}])^+ \text{ et } Q_i \cap Q_{k+1} \preceq R_i \preceq Q_i \tag{A.17}$$

– lorsque nous sommes dans le cas (a), nous allons poser $R_i = Q_i \cap Q_{k+1}$ qui respecte clairement la propriété (A.17). Ce cas est illustré dans la Figure A.10(a). En se servant de (A.11) nous savons que $\Pi^1_M(\mathcal{T})[C/Q_{k+1}] \neq_{E_{k+1}} \Pi^2_M(\mathcal{T})[C/Q_{k+1}]$.
Puisque $v^1_{i,k+1} = v^2_{i,k+1}$, les instances du plus long préfixe commun de (A.15) et (A.16) sont les mêmes et correspond à l'instance de R_i (notons le I_{R_i}) c'est à dire,

$$I_{R_i} = \Pi^1_M(\mathcal{T})[C/Q_i] \cap \Pi^1_M(\mathcal{T})[C/Q_{k+1}] = \Pi^2_M(\mathcal{T})[C/Q_i] \cap \Pi^2_M(\mathcal{T})[C/Q_{k+1}] \tag{A.18}$$

Comme nous avons $I_{R_i} = \Pi^1_M(\mathcal{T})[C/Q_i] \cap \Pi^1_M(\mathcal{T})[C/Q_{k+1}] \preceq \Pi^1_M(\mathcal{T})[C/Q_{k+1}]$ et aussi $I_{R_i} = \Pi^2_M(\mathcal{T})[C/Q_i] \cap \Pi^2_M(\mathcal{T})[C/Q_{k+1}] \preceq \Pi^2_M(\mathcal{T})[C/Q_{k+1}]$, en appliquant le Lemme A.14(2), nous obtenons $|Instances(C/R_i, \mathcal{T})| = 1$. Ainsi en se servant du Lemme A.14(1), nous obtenons que $C/R_i[E'_i] \in (C, X[\vec{E}])^+$.

– lorsque nous sommes dans le cas (b), nous allons poser $R_i = Q_i$ qui aussi respecte la propriété (A.17). Ce cas est illustré dans la Figure A.10(b). Comme $v^1_{i,k+1} \neq v^2_{i,k+1}$, les instances $\Pi^1_M(\mathcal{T})[C/Q_i]$ et $\Pi^2_M(\mathcal{T})[C/Q_i]$ sont toujours différents et de même pour les instances associées au chemin R_i. Nous avons donc $|Instances(C/R_i, t)| = 2$. A partir de ce fait et du résultat obtenu dans (A.10) nous avons $\Pi^1_M(\mathcal{T})[C/Q_i] =_V \Pi^2_M(\mathcal{T})[C/Q_i]$. En se basant sur la Définition A.4(1) nous concluons que $C/R_i[E'_i] \in (C, X[\vec{E}])^+$.

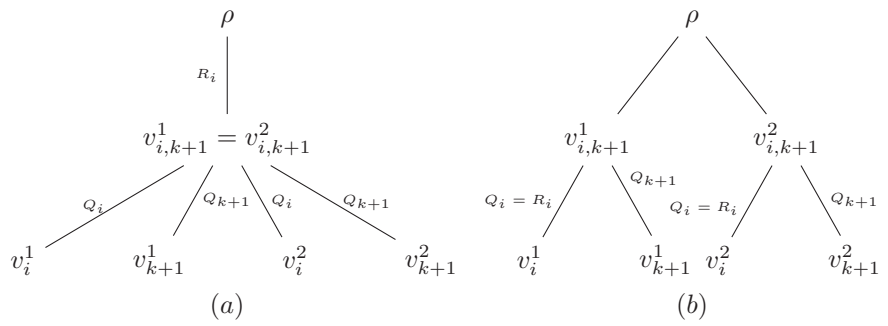


FIGURE A.10 – *Illustration of the two cases (a)*$v^1_{i,k+1} = v^2_{i,k+1}$ *and (b)*$v^1_{i,k+1} \neq v^2_{i,k+1}$.

Maintenant comme nous avons la propriété (A.17), en appliquant l'axiome A4 (Branche préfixe) sur la XFD $g = (C, (\{Q_1[E'_1], \ldots, Q_k[E'_k]\} \rightarrow Q_{k+1}[E'_{k+1}]))$ dans $\mathcal{F}$, nous engendrons la XFD $g' = (C, (\{R_1[E'_1], \ldots, R_k[E'_k]\} \rightarrow Q_{k+1}[E'_{k+1}]))$. Ensuite nous affirmons que si $C/\{R_1, \ldots, R_k\}[\vec{E'}] \subseteq (C, X[\vec{E}])^+$ alors $C/Q_{k+1}[E'_{k+1}] \in (C, X[\vec{E}])^+$. En effet, en utilisant le Lemme 4.2 nous savons que $(C, (X[\vec{E}] \rightarrow \{R_1[E'_1], \ldots, R_k[E'_k]\}$. A partir de la XFD que nous venons d'obtenir, et de g' nous pouvons dériver par transitivité $(A3)$ la XFD $(C, (X[\vec{E}] \rightarrow Q_{k+1}[E'_{k+1}]))$ Ainsi selon la Définition 4.22, nous obtenons $C/Q_{k+1}[E'_{k+1}] \in (C, X[\vec{E}])^+$ qui est en contradiction avec (A.12) : $C/Q_{k+1}[E'_{k+1}] \notin (C, X[\vec{E}])^+$. Nous pouvons donc conclure que $\mathcal{T} \models g$ pour n'importe quelle XFD $g \in \mathcal{F}$, c'est à dire que $\mathcal{T} \models \mathcal{F}$.

Fait 2 : $\mathcal{T} \not\models f$.
Rappellons qu'au début de la preuve nous avons poser $f = (C, (X[\vec{E}] \rightarrow Y[\vec{E_f}]))$. Comme $C/X[\vec{E}] \subseteq (C, X[\vec{E}])^+$, alors pour tout $P_i \in X$ nous avons $Last(\Pi^1_M(\mathcal{T})[P_i]) =_{E_i} Last(\Pi^2_M(\mathcal{T})[P_i])$ pour deux projections de M sur $\mathcal{T}$. D'après notre hypothèse $\mathcal{F} \not\vdash f$ et donc $C/Y[\vec{E_f}] \not\subseteq (C, X[\vec{E}])^+$. Ainsi il existe au moins un chemin $P \in Y$ qui a deux instances I_1 et I_2 dans $\mathcal{T}$ telles que $Last(I_1) \neq_E Last(I_2)$. Nous déduisons donc que $\mathcal{T} \not\models f$.

Pour résumer nous avons trouver un arbre $\mathcal{T}$ tel que $\mathcal{T} \models \mathcal{F}$ et $\mathcal{T} \not\models f$ ce qui fait que la preuve du Théorème A.5 est achevée.

◁

A.6 Algorithme pour la fermeture d'un ensemble de chemins : Preuve de correction

Dans cette section, nous prouvons que l'Algorithme 2 est correct.

Théorème A.6 *L'ensemble calculé par l'Algorithme 2 est égal à* $(C, X[\vec{E}])^+$.

Démonstration. Nous allons d'abord prouver que l'algorithme termine, ensuite prouver qu'il est correct et enfin prouver qu'il est complet. Pour plus de simplicité, nous allons omettre le contexte c'est à dire écrire $X[\vec{E}] \rightarrow P[E']$ au lieu de la dépendance $(C, (X[\vec{E}] \rightarrow P[E']))$. Nous allons aussi parfois omettre les types d'égalités.

Terminaison : L'algorithme termine parce que $\mathbb{P}$ est fini et comme $C/X^{(i)}$ est un sous-ensemble de $\mathbb{P}$, nous allons atteindre la valeur j pour i telle que $X^{(j)} = X^{(j+1)}$. Par conséquent nous avons la garantie que la boucle termine et l'ensemble $X^{(j)}$ est retourné par l'algorithme. Nous allons maintenant prouver que X^+ est égal à $X^{(j)}$ pour cette valeur de i (à la fin $i = j$).

Correction ($X^{(j)} \subseteq X^+$) : Nous allons montrer par induction sur i que si un chemin $P[E']$ est placé dans $X^{(i)}$ pendant l'éxécution de l'Algorithme 2, alors $P[E']$ est dans $X[\vec{E}]^+$ (ou $X \rightarrow P[E']$ selon la Définition 4.22).

- Cas de base : $i = 0$. A la ligne 5 de l'Algorithme 2, l'ensemble $X^{(0)}$ contient des chemins P qui sont dans S, ou T, ou V ou W.
  1. si $P[E'] = [][N]$ alors avec l'axiome A7 on obtient $X \rightarrow P$.
  2. sinon si $P[E'] \in T$ alors :
     (a) soit $P[E'] \in X[\vec{E}]$ et donc grâce à l'axiome A1 (Réflexivité) nous avons $X[\vec{E}] \rightarrow P[E']$;
     (b) soit il existe un chemin $Q[N] \in X[\vec{E}]$ tel que $P \preceq Q$. Comme nous avons $X[\vec{E}] \rightarrow Q[N]$, en appliquant l'axiome A5 (Unicité des Ascendants) nous avons $Q[N] \rightarrow P[N]$ et par l'Axiome A3 (Transitivité) on obtient $X[\vec{E}] \rightarrow P[N]$;
  3. sinon si $P[N] \in V$ alors $Last(P) \in \Sigma_{att}$ et il existe un chemin $Q[E'] \in T \cup S$ tel que $Q = Parent(P)$. Comme nous avons $X[\vec{E}] \rightarrow Q[E']$, en appliquant l'axiome A6 (Unicité de l'attribut) nous avons $Q[E'] \rightarrow P[N]$ et par l'Axiome A3 on obtient $X[\vec{E}] \rightarrow P[N]$.
  4. sinon si $P[V] \in W$ alors $P[N] \in S \cup T \cup V$. Comme nous avons $X[\vec{E}] \rightarrow P[N]$, en appliquant l'axiome A9 (De l'identité de nœud à l'égalité par valeur) nous avons $P[N] \rightarrow P[V]$ et par l'Axiome A3 on obtient $X[\vec{E}] \rightarrow P[V]$.

  Nous venons de montrer que pour chaque $P[E'] \in X^{(0)}$, nous avons $X[\vec{E}] \rightarrow P[E']$.
- Induction : Supposons que $i > 0$ et que $X^{(i-1)}$ contient que des chemins qui sont dans $X[\vec{E}]^+$ (hypothèse). Nous allons maintenant prouver que si $P[E']$ est placé dans $X^{(i)}$ alors $P[E'] \in X[\vec{E}]^+$. A la ligne 11 de l'Algorithme 2, nous avons $X^{(i)} = X^{(i-1)} \cup T \cup V \cup W$.
  1. Si $P[E'] \in X^{(i-1)}$ alors par hypothèse d'induction, $P[E']$ appartient aussi à $X[\vec{E}]^+$.
  2. Sinon si le chemin $P[E_{n+1}]$ est dans Y alors il existe une XFD f dans $\mathcal{F}$ telle que l'une des conditions (a)-(b) de la ligne 7 soit satisfaite. Lorsque nous posons $Z = \{P_1, \ldots, P_n\}$ alors si f est de la forme $(C, (\{P_1\,[E_1], \ldots, P_n\,[E_n]\} \rightarrow P\,[E_{n+1}]))$

cela revient à écrire $(C, (Z \to P))$; sinon si f est de la forme $(C', (\{Q/P_1\,[E_1], \ldots, Q/P_n\,[E_n]\} \to Q/P\,[E_{n+1}]))$ avec $C = C'/Q$, en appliquant l'axiome A8 (Extension du contexte) on obtient aussi $(C, (Z \to P))$. Analysons ces deux conditions :

(a) En premier supposons que $Z[\vec{E'}] = \{P_1, \ldots, P_n\}[\vec{E'}] \subseteq X^{(i-1)}$. Puisque $Z[\vec{E'}] \subseteq X^{(i-1)}$, nous savons que $Z[\vec{E'}] \subseteq X[\vec{E}]^+$ par l'hypothèse d'induction. Ainsi en appliquant le Lemme 4.2, on obtient $X \to Z$. Par Transitivité (Axiome A3), $X \to Z$ et $Z \to P$ entraînent $X \to P$ et donc $P[E'] \in X[\vec{E}]^+$;

(b) Finalement supposons qu'il existe des chemins $P'_1[E_1], \ldots, P'_n[E_n] \in X^{(i-1)}$ et $\forall\, k \in [1, \ldots, n]$, $P_k \cap P \preceq P'_k$ et ($P'_k \preceq P_k$ ou $P'_k \preceq P$). Comme $Z' = \{P'_1, \ldots, P'_n\}$ respecte ces conditions, nous pouvons appliquer l'axiome A4 (Branche Préfixe) pour obtenir $Y \to P$. Puisque $Z' \subseteq X^{(i-1)}$, par hypothèse d'induction et en utilisant le Lemme 4.2 on a $X \to Z'$. Par Transitivité (Axiome A3), $X \to Z'$ et $Z' \to P$ entraînent $X \to P$.

3. sinon si $P \in T \cup V \cup W$ alors avec les mêmes arguments qu'au cas de base, nous avons $P[E']$ appartient à $X[\vec{E}]^+$.

Nous venons de montrer que pour chaque $P[E'] \in X^{(i)}$, nous avons $X[\vec{E}] \to P[E']$.

Complétude ($X^+ \subseteq X^{(j)}$) : Montrons maintenant qu'on oublie pas de chemins c'est à dire que si un chemin $P[E'] \in X[\vec{E}]^+$ alors $P[E'] \in X^{(j)}$. La preuve de cette partie sera faite par contradiction. Nous allons supposer que $P[E'] \in X[\vec{E}]^+$ mais que $P[E'] \notin X^{(j)}$. Rappelons que l'Algorithme 2 retourne un résultat seuleument si $X^{(j)} = X^{(j+1)}$.

Supposons que nous avons l'arbre bi-instance t défini sur l'ensemble $X^{(j)}$. Cet arbre bi-instance t possède au plus deux instances pour chaque chemin dans $X^{(j)}$ tel que les derniers nœuds des instances :

– sont égaux pour les chemins qui sont dans $X^{(j)}$, et

– sont différents pour les chemins qui ne sont pas dans $X^{(j)}$.

Nous affirmons que t satisfait $\mathcal{F}$. Si ce n'est pas le cas alors il existe une dépendance $f = U \to Q$ dans $\mathcal{F}$ qui est violée par t. Pour que f soit violée il faut trouver des instances de chemin dont les derniers nœuds sont égaux pour des chemins dans U et différents pour le chemin Q. Si tel est le cas alors à cause des propriétés de t on a $U \subseteq X^{(j)}$ et $V \notin X^{(j)}$. Si f est violée alors nous pouvons utiliser les mêmes arguments que dans la preuve de la complétude de notre système d'axiome (Théorème A.5, Fait 1) pour montrer que $f = U \to Q$ n'est pas violée par t. Il en découle donc que comme $V \not\subseteq X^{(j)}$ et $U \to Q$ n'est pas violée par t, alors l'ensemble $X^{(j+1)}$ devrait contenir le chemin Q. Pour cette raison $X^{(j+1)}$ ne peut pas être le même que $X^{(j)}$ comme on l'a supposé. Comme il y a contradiction nous concluons finalement que t satisfait $\mathcal{F}$.

Par conséquent l'arbre bi-instance t doit aussi satisfaire $X[\vec{E}] \to P[E']$. La raison est que $P[E']$ est supposé être dans $X[\vec{E}]^+$ (hypothèse), et donc $X[\vec{E}] \to P[E']$ en découle de $\mathcal{F}$ en utilisant le système d'axiomes. Puisque le système d'axiomes est correct, tout arbre qui satisfait $\mathcal{F}$ va aussi satisfaire $X \to P$. Mais la seule raison pour que $X \to P$ soit satisfaite par t est que $P[E']$ appartienne à $X^{(j)}$. Cela amène donc à une contradiction puisque $P[E']$ est supposé ne pas appartenir à $X^{(j)}$ (hypothèse). Nous concluons donc que $P[E'] \in X^{(j)}$ qui est l'ensemble calculé par l'Algorithme 2.

◁

A.7 Preuves de correction et de complétude de l'algorithme de correction

Lemma 5.1 *Given a tag $c \in \Sigma$, a schema S and a threshold th, let S' be the schema $(\Sigma, c, S.Rules)$. A call to correction($t_\emptyset$,S,th,c) always terminates and returns the set Result s.t. the following proposition holds :*

$$t_\emptyset \xrightarrow{Result} L^{th}_{t_\emptyset}(S').$$

Proof. The proof is done by induction on the threshold th.

- **Basis** : If $th = 0$ then the function $correction(t_\emptyset, S, 0, c)$ returns in line 4 (an empty tree is not locally valid) with $Result = \emptyset$. Note that $t_\emptyset \xrightarrow{\emptyset} \emptyset$. Note also that $L^0_{t_\emptyset}(S') = \{t' \mid t' \in L(S'), dist(t_\emptyset, t') = 0\} = \emptyset$ since each t' in $L^0_{t_\emptyset}(S')$ must be equal to $t_\emptyset$ and $t_\emptyset$ is never valid *w.r.t.*a schema (it has no root). We can conclude that $t_\emptyset \xrightarrow{\emptyset} L^0_{t_\emptyset}(S')$.
- **Induction step** : Suppose that $0 < th$ and for each $0 \leq th' < th$ the call $correction(t_\emptyset, S, th', c)$ terminates and the proposition holds for its result. We will show that $correction(t_\emptyset, S, th, c)$ also terminates and the proposition holds for its result.

(a) Termination and soundness : We wish to show that $correction(t_\emptyset, S, th, c)$ terminates and each node operation sequence in *Result* leads to a tree in $L^{th}_{t_\emptyset}(S')$, i.e. a locally valid tree with root c and whose distance from $t_\emptyset$ is no greater than th.

Note that for $0 < th$ the call $correction(t_\emptyset, S, th, c)$ constructs a matrix M^c_v where $M^c_v[0][0] = \{(add, \epsilon, c)\}$ and $cost((add, \epsilon, c)) = 1$ (see function *correction*, line 11). The calculation of this first cell is straightforward so it clearly terminates. Then, for each $0 < j \leq |v|$, $M^c_v[0][j]$ is obtained by concatenating sequences from $M^c_v[0][j-1]$ with results of a new correction of $t_\emptyset$ *w.r.t.*S (see procedure *correctionTransition*, lines 7–9) with $th'' = th - MinCost(M^c_v[0][j-1])$. Since the correction of an empty tree induces at least one node insertion we have $th'' < th$. By hypothesis, this new correction terminates and yields sequences leading to locally valid trees. Finally, lines 2–3 in *correctionState* guarantee that each final concatenation (if any) added to *Result* leads to a tree t' which has a root c, and whose root's children form a word valid *w.r.t.*FSA_c. Thus, t' is necessarily locally valid, which proves soundness. Note also that the number of columns in M^c_v cannot exceed th. Thus, the algorithm terminates after at most th recursive calls.

(b) Completeness : The proof is done by contradiction. Suppose that there exists a locally valid tree t' with root c such that the distance between t' and $t_\emptyset$ is no greater than th, and t' cannot be obtained with a node operation sequence in *Result*.

Note that if $t' \in L(S')$ and $dist(t_\emptyset, t') \leq th$ then : (i) each subtree $t'|_i$ (with $0 \leq i \leq \bar{t'} - 1$) is locally valid, (ii) $dist(t_\emptyset, t'|_i) < th$, (iii) $\sum_i dist(t_\emptyset, t'|_i) < th$, (iv) the word v formed by t's root's children is valid *w.r.t.*FSA_c. Note that lines 6–7 in *correctionState* guarantee that we test all outgoing transitions for every state reached in FSA_c. Thus,

$$correctionTransition(\, t_\emptyset, S, th-1, c, M^c_\epsilon, s', t'(0), Result_0)$$

will have to be called. By hypothesis, the sequence nos_0 leading to the subtree $t'|_0$ must be contained in the *Result* of this call. Then,

$$correctionTransition(\, t_\emptyset, S, th-MinCost, c, M^c_{t'(0)}, s'', t'(1), Result_1)$$

will have to be called with $0 < MinCost \leq cost(nos_0) < th$. By hypothesis, the sequence nos_1 leading to subtree $t'|_1$ must again be contained in the $Result_1$ of this call. The same holds for all subtrees of t'. Thus, t' can be obtained from $t_\emptyset$ by the operation sequence

$$nos = \langle\ (insert, \epsilon, c)\rangle \cdot_{th} AddPrefix(nos_0, 0) \cdot_{th} AddPrefix(nos_1, 1) \cdot_{th} \cdots \cdot_{th} AddPrefix(nos_{\bar{t}'-1}, \bar{t}' - 1)$$

This sequence will necessarily be created in *correctionTransition*, line 9, and further added to *Result* in *correctionState*, line 3.

◁

Lemma 5.2 *Given a tag $c \in \Sigma$, a schema S, and a tree t, let S' be the schema $(\Sigma, c, S.Rules)$. A call to correction(t,S,0,c) always terminates and returns the set Result s.t.*

$$t \xrightarrow{Result} L_t^0(S').$$

Proof. Note that $L_t^0(S') = \{t' \mid t' \in L(S') \text{ and } dist(t, t') = 0\}$. This set contains only t if t is locally valid, and no tree otherwise. Note also that the call to $correction(t, S, 0, c)$ terminates : (i) in line 2 with $Result = \{nos_\emptyset\}$ if t is valid, (ii) in line 4 with $Result = \emptyset$ otherwise. In case (i) *Result* leads from t to t itself, and in case (ii) to no tree. Thus, the lemma holds.

◁

Lemma 5.3 *Let $c \in \Sigma$ be a tag, S be a schema, $t \neq t_\emptyset$ be a tree, and $th > 0$ be a threshold. Let $u \in \Sigma^*$ be a word such that $u \in L(FSA_{S.root})$ and $L_u^c(S) \neq \emptyset$. The call to $correction(t, S, th, c)$ computes the matrix M_u^c such that, for each $0 \leq i \leq \bar{t}$ and for each $0 \leq j \leq |u|$, the following proposition holds :*

$$t\langle i-1\rangle \xrightarrow{M_u^c[i][j]} \{t' \mid t' \in L^c_{u[1..j]}(S), dist(t\langle i-1\rangle, t') \leq th\}$$

Proof. Firstly, note that for a non-empty tree t and a positive threshold $0 < th$ a distance matrix is necessarily created (function *correction*, line 8) and filled out.

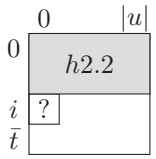


FIGURE A.11 – *Example of matrix representation for the proof*

Secondly, let's consider the case of $i = 0$ and $j = 0$. Note that for a non-empty tree t the cell $M_u^c[0][0]$ can only be filled out in the function *correction*, lines 13 and 14. Each of these 2 actions clearly terminates and results in an operation sequence transforming t's root (i.e. $t\langle -1\rangle$) into a root-only tree $\{(\epsilon, c)\}$ with cost no higher than 1. Note also that $L^c_{u[1..0]}(S) = L^c_\epsilon(S) = \{(\epsilon, c)\}$. Thus, the proposition holds for $i = 0$ and $j = 0$.

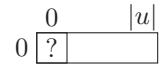
The proof for the remaining cases is done by induction on the depth of the tree t, then on the row index i, and finally on the column index j. We will use the representation of the distance matrix M_u^c as in Figure A.11 to say that we want to verify the cell which contains the question mark '?' knowing that the cells in gray are concerned by the hypothesis.

1. Basis ($depth(t) = 1$**)** : If $depth(t) = 1$ then t contains only the root tag, i.e. $t = \{(\epsilon, x)\}$ and $\bar{t} = 0$. Consequently, there exists only one i *s.t.*$0 \leq i \leq \bar{t}$, namely $i = 0$. Thus, we only need to show that for each $0 \leq j \leq |u|$:

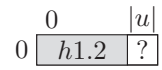
$t\langle -1\rangle \xrightarrow{M_u^c[0][j]} \{t' \mid t' \in L^c_{u[1..j]}(S), dist(t\langle -1\rangle, t') \leq th\}$

Further proof is done by induction on column index j.

1.1. Basis ($depth(t) = 1, i = 0, j = 0$**)** : If $j = 0$ then we are considering the same case as above, i.e. $i = 0$ and $j = 0$. We have already shown that the proposition is true for this particular case.



1.2. Induction step ($depth(t) = 1, i = 0, 0 < j$**)** : Suppose that the proposition is true for a $0 \leq j' = j - 1$ **(h1.2)**, i.e.



$t\langle -1\rangle \xrightarrow{M_u^c[0][j-1]} \{t' \mid t' \in L^c_{u[1..(j-1)]}, dist(t\langle -1\rangle, t') \leq th\}$.

We will prove that it also holds for j.

Note that with $0<j$ the cell $M_u^c[0][j]$ can only be filled out in function *correctionTransition*, line 9. The contents of this cell stems from the threshold-bound concatenation of node operation sequences in : (i) the cell $M_u^c[0][j-1]$, (ii) the result of correcting an empty tree with an appropriately diminished threshold, and with target root u_j. This situation is depicted in Fig. A.12. By hypothesis $h1.2$, the calculation of (i) terminates and each element in (i) transforms $t\langle -1\rangle$ into a partially valid tree with word $u[1..(j-1)]$ formed by the root's children. By Lemma 5.1 the calculation of (ii) terminates and each element of (ii) creates a locally valid tree with root u_j. Thus, each concatenation of these elements creates a tree whose :

- root is c,
- root's children form the word $u[1..j]$
- all root's subtrees are locally valid.

In other words this tree is partially valid. The threshold-bound concatenation guarantees that its distance from t is no greater than th. That proves the termination and the soundness of the lemma. The completeness can be proved similarly to Lemma 5.1. In each $t' \in L^c_{u[1..j]}$ all subtrees are locally valid, have the appropriate distance from t and $u[1..j] \in FSA_c$. Thus, each transition in FSA_c labeled with u_k ($1 \leq k \leq j$) has to be followed, and by hypothesis each subtree $t'|_k$ has to be reachable by a sequence stemming from a call to *correctionTransition*. The whole tree t' can be obtained by concatenating the root correction operation with all such sequences for $1 \leq k \leq j$ (prefixed by k).

Thus, the proposition holds for any $0 \leq j$ with $i = 0$.

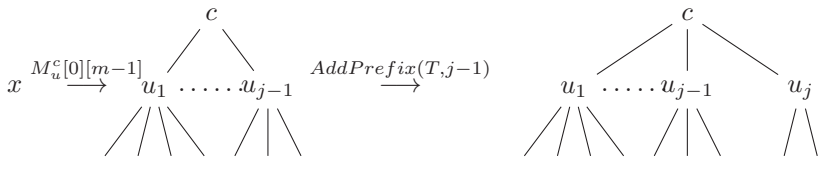


FIGURE A.12 – *Combining operation sequences in case of* $i = 0$

2. Induction step ($depth(t) > 1$**)** : Suppose that the proposition is true for any tree t' with $0 \leq depth(t') < d$ **(h2)**. We will prove that it also holds for any tree t with $depth(t) = d$.

The proof is done by induction on the row index i.

2.1. Basis ($depth(t) > 1, i = 0$**)** : With $i = 0$ we need to show that for each $0 \leq j \leq |u|$:

$t\langle -1\rangle \xrightarrow{M_u^c[0][j]} \{t' \mid t' \in L^c_{u[1..j]}(S), dist(t\langle -1\rangle, t') \leq th\}$.

The proof is the same as in the case of a tree of depth 1 (see above).

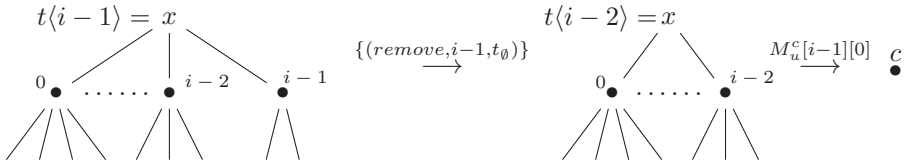
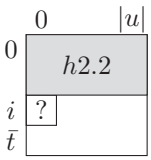


FIGURE A.13 – *Combining operation sequences in the first column of the distance matrix*

2.2. Induction step ($depth(t) > 1, i > 0$**)** : Suppose that the proposition is true for any $0 \leq i' < i$ **(h2.2)**. We will prove that is also holds for i. The proof is done by induction on column index j.

2.2.1. Basis ($depth(t) > 1, i > 0, j = 0$) : With $j = 0$ we need to show that :

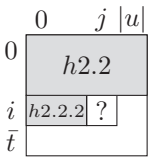
$t\langle i-1\rangle \xrightarrow{M_u^c[i][0]} \{t' \mid t' \in \{(\epsilon, c)\}, dist(t\langle i-1\rangle, t') \leq th\}$



Recall that cell $M_u^c[0][0]$ contains at most one operation leading to the correct root c. Note also that all other cells in the first column can only be filled out in function *correction*, line 18. The contents of each of these cells stems from the threshold-bound concatenation of : (i) removing a subtree in t, (ii) node operation sequences in the cell above. Thus, the cell $M_u^c[i][0]$ contains at most one operation sequence which represents removing all subtrees in $t\langle i-1\rangle$ (from right to left) and possibly relabeling the root, as depicted in Fig. A.13. Obtaining this sequence (if any) clearly terminates and leads to the root-only tree $\{(\epsilon, c)\}$, which proves termination and soundness.

The completeness is straightforward since the only possible element (if any) in $\{t' \mid t' \in \{(\epsilon, c)\}, dist(t\langle i-1\rangle, t') \leq th\}$ is the root-only tree $\{(\epsilon, c)\}$. This tree can be obtained precisely by the unique operation sequence (if any) described above.

2.2.2. Induction step $(depth(t) > 1, i > 0, j > 0)$: Suppose that the proposition is true for any $0 \leq j' = j - 1$ **(h2.2.2)**.



We will prove that it also holds for j. Note that if $i > 0$ and $j > 0$ the cell $M_u^c[i][j]$ can only be filled out in procedure *correctionTransition*, lines 16–18. Three cases are to be examined.

(i) The horizontal correction (line 16) yields threshold-bound concatenations of : (i) the cell $M_u^c[i][j-1]$, (ii) the result of correcting an empty tree with an appropriately diminished threshold, and with target root u_j. By hypothesis $h2.2.2$, the calculation of (i) terminates and each element (if any) in (i) transforms $t\langle i-1\rangle$ into a partially valid tree with word $u[1..j-1]$ formed by the root's children. By Lemma 5.1 the calculation of (ii) terminates and each element (if any) of (ii) creates a locally valid tree with root u_j. Thus, each concatenation of elements in (i) and (ii), provided that it does not exceed the threshold, creates a tree whose :

- root is c,
- the root's children form the word $u[1..j]$
- all root's subtrees are locally valid.

In other words this tree is partially valid. The threshold-bound concatenation guarantees that its distance from $t\langle i-1\rangle$ is no greater than th. That proves the termination and the soundness of this case.

(ii) The diagonal correction (line 17) yields threshold-bound concatenations of : (i) the cell $M_u^c[i-1][j-1]$, (ii) the result of correcting subtree $t|_{i-1}$ with an appropriately diminished threshold th', and with target root u_j. By hypothesis $h2.2$, the calculation of (i) terminates and each element (if any) in (i) transforms $t\langle i-2\rangle$ into a partially valid tree with word $u[1..j-1]$ formed by the root's children. Note that for (ii) two cases are possible. Firstly, th may be equal to 0. In that case, by Lemma 5.2, the result of (ii) is either empty ($t|_{i-1}$ is not locally valid) or equal to $t|_{i-1}$ (otherwise). Secondly, th may be positive. In that case, each element in the result of (ii) necessarily stems from the cell $M_v^{u_j}[t|_{i-1}^{-}][|v|]$ for a certain $v \in FSA_{u_j}$ (see procedure *correctionState*, line 3). Since the subtree $t|_{i-1}$ necessarily has a smaller depth than t, by hypothesis $h2$, (ii) terminates and
$t|_{i-1} \xrightarrow{M_v^{u_j}[t|_{i-1}^{-}][|v|]} \{t' \mid t' \in L_v^{u_j}(S), dist(t|_{i-1}, t') \leq th\}$
In other words, each element in (ii) transforms $t|_{i-1}$ into a locally valid tree with root u_j. Thus, each concatenation of elements in (i) and (ii), provided that it does not exceed the threshold, again creates a partially valid tree within the threshold, as depicted in Fig. A.14. That proves the termination and the soundness of this case.

(iii) The vertical correction (line 18) yields threshold-bound concatenations of : (i) removing subtree $t|_{i-1}$, (ii) node operation sequences in the cell $M_u^c[i-1][j]$. Clearly, (i) terminates. By hypothesis $h2.2$, the calculation of (ii) terminates and each element in (ii) transforms $t\langle i-2\rangle$ into a partially valid tree with word $u[1..j]$ formed by the root's children. This process, when preceded by deleting subtree $t|_i$ transforms the partial tree $t\langle i-1\rangle$ into a partially valid tree. That proves the termination and the soundness of this case.

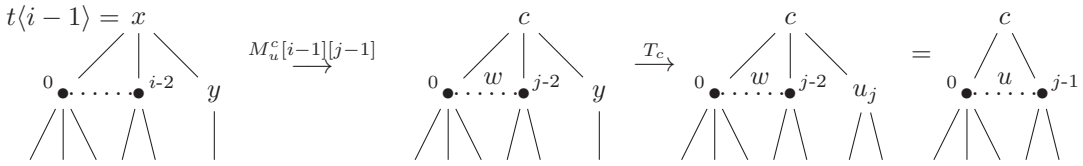


FIGURE A.14 – *Combining operation sequences in case of a diagonal correction with* $w = u_1 \dots u_{j-1}$

We prove the completeness by contradiction. Let's suppose that there exists a tree $t' \in L^c_{u[1..j]}(S)$ such that $dist(t\langle i-1\rangle, t') \leq th$, and no operation sequence in $M_u^c[i][j]$ leads from $t\langle i-1\rangle$ to t'. According to the tree distance definition in [100], t' can be obtained from $t\langle i-1\rangle$ by at least one of the three types of corrections (horizontal, diagonal, or vertical correction).

(i) In the horizontal correction : (i) the partial tree $t\langle i-1\rangle$ is transformed into the partial tree $t'\langle j-2\rangle$, (ii) the subtree $t'|_{j-1}$ is inserted. By hypothesis $h2.2.2$ the cell $M_u^c[i][j-1]$ must contain the sequence allowing to obtain $t'\langle j-2\rangle$ since this partial tree is within the threshold and its root's children form the word $u[1..j-1]$. Note also that, by Lemma 5.1, the subtree $t'|_{j-1}$ must be reachable by a sequence calculated in *correctionTransition*, line 11. Thus, at least one sequence leading to t' must be obtained.

(ii) In the diagonal correction the partial tree $t\langle i-2\rangle$ is transformed into the partial tree $t'\langle j-2\rangle$ and the subtree $t|_{i-1}$ is transformed into the subtree $t'|_{j-1}$. By hypotheses $h2.2$ and $h2$ at least one corresponding operation sequence must be obtained.

(iii) In the vertical correction the partial tree $\langle i-2\rangle$ is transformed into t' and the subtree $t|_{i-1}$ is deleted. By hypothesis $h2.2$, and by line 18 in *correctionTransition*, at least one corresponding operation sequence must be obtained.

◁

Theorem 5.1 *Given a tree t, a schema S and a threshold $0 \leq th$, a call to $correction(t, S, th, S.root)$ terminates and returns a set $Result$ such that the following proposition holds :*

$$t \xrightarrow{Result} L_t^{th}(S).$$

Proof. If $th = 0$ then the proposition holds by Lemma 5.2. Suppose now that $0 < th$.

(a) Termination : The proof is done by induction on the depth of the tree t. If $depth(t) = 0$ the tree is empty and, by Lemma 5.2, the call to $correction(t_\emptyset, S, th, S.root)$ terminates. Suppose now that the call terminates for each tree t' $s.t. 0 \leq depth(t') < d$. We will prove that it also terminates for a tree t with depth d. With $0 < th$ the set $Result$ can receive new operation sequences only in the procedure *correctionState*, line 3. These corrections stem from the matrix $M_u^{S.root}$ for some $u \in FSA_{S.root}$. By Lemma 5.3, the calculation of each matrix cell terminates. Note that the recursion is induced in *correctionTransition* by lines 7, 14, 17 and 26. The recursion in lines 7 and 14 terminates by Lemma 5.1. The recursion in line 17 terminates by the induction hypothesis since the subtree $t|_{i-1}$ has a smaller depth than t. Moreover the cells filled out in lines 9, 16 and 18 necessarily have a higher cost than the cells they have been deduced from ($M_v^c[0][m-1]$, $M_v^c[i][m-1]$ and $M_v^c[i-1][m]$, respectively) because at least one node insertion or deletion is concatenated. Only the concatenation in line 17 can lead to sequences of the same cost as in the preceding cell $M_v^c[i-1][m-1]$. That can however happen only for locally valid subtrees, whose number is bounded by $\bar{t}$. Thus after at most $\bar{t} + th$ recursive calls between procedures *correctionState* (line 7) and *correctionTransition* (line 26) all solutions in the current column must have a cost exceeding th and the recursion stops.

(b) Soundness : With $0 < th$ the set $Result$ can receive new operation sequences only in the procedure *correctionState*, line 3. These corrections stem from the bottom right-hand cell of the matrix, i.e. $M_u^{S.root}[\bar{t}][|u|]$, for some $u \in FSA_{S.root}$. By Lemma 5.3 we have :
$t\langle \bar{t}-1 \rangle \xrightarrow{M_u^{S.root}[\bar{t}][|u|]} \{t' \mid t' \in L_u^{S.root}(S), dist(t\langle \bar{t}-1 \rangle, t') \leq th\}$.
Since $t\langle \bar{t}-1 \rangle = t$ we get : $t \xrightarrow{M_u^{S.root}[\bar{t}][|u|]} \{t' \mid t' \in L_u^{S.root}(S), dist(t, t') \leq th\}$
Note that $L_u^{S.root}(S) \subseteq L(S)$, thus each element in $Result$ leads to a tree that belongs to $L_t^{th}(S)$, which proves the soundness.

(c) Completeness : The proof is done by contradiction. Suppose that there exists a tree $t' \in L_t^{th}(S)$ such that no operation in $Result$ leads from t to t'. Let v be the word formed by the children of t's root, i.e. $v = t'(0) \dots t'(\bar{t}' - 1)$. Note that $t'(\epsilon) = S.root$ since t' is valid. The function *correction* necessarily creates a matrix $M_\epsilon^{S.root}$ in line 8, fills out its first column and the cell $M_\epsilon^{S.root}[0][0]$ contains the node operation leading from $t(\epsilon)$ to $S.root$. Further on, the procedure *correctionState* necessarily tests the transition in $FSA_{S.root}$ labeled with $t'(0)$. By Lemma 5.3 the second column in $M_{t'(0)}^{S.root}$ necessarily contains sequences leading from partial trees of t to the partial tree $t'\langle 0 \rangle$. By definition of the tree distance, at least one of these sequences has a cost no higher than th. Thus, necessarily the procedure *correctionState* is again called by procedure *correctionTransition*, line 26 and the transition labeled with $t'(1)$ is examined. These calls continue until $t'(\bar{t}' - 1)$ and each new column contains at least one sequence within the threshold. After the transition labeled with $t'(\bar{t}' - 1)$ is examined the contents of the cell $M_v^{S.root}[\bar{t}][\bar{t}']$ is added to $Result$. By the proof of soundness, this cell leads from t to t'.

◁

A.8 Tableaux de comparaison

Reference	Elementary edit operations	Validity aspects			Algorithm's output					Schema type	Document model	Schema model
		Well-formedness	Structure	Attributes	Tree-to-schema edit distance	Corrections			Edit sequences			
						minimal	k closest	all within a threshold				
Boobna, de Rougemont [25]	node relabeling node insertion node deletion		✓			not always the minimal one				restricted DTD [a]	ranked ordered labeled tree	set of reg. exp.
Bertino et al. [19, 20]	no edit operation		✓		✓					DTD	unranked ordered labeled tree	ordered labeled tree
Xing et al. [125]	node relabeling leaf insertion leaf deletion		✓		✓					DTD and XML schema	unranked ordered labeled tree	regular hedge grammar
Staworko, Chomicki [104]	leaf insertion leaf deletion		✓		✓					DTD	unranked ordered labeled tree	top-down finite tree automaton
Suzuki [108]	node relabeling node insertion node deletion		✓		✓		✓		✓	DTD and XML schema	unranked ordered labeled tree	regular tree grammar
Tekli et al. [115]	node relabeling leaf insertion leaf deletion		✓	✓ [b]	✓	✓				restricted DTD [c]	unranked ordered labeled tree	ordered labeled tree
Staworko et al. [105]	node relabeling node insertion node deletion (except a root)	✓	✓	✓	✓					DTD	serialized unranked ordered labeled tree	streaming tree automaton
Thomo et al. [117]	node relabeling node insertion node deletion (+ multi operations)	✓	✓		✓ not exactly			no but could serve to do it		Extended DTD	serialized unranked ordered labeled tree [d]	visibly pushdown automaton
Svoboda, Mlýnková [112, 113]	node relabeling leaf insertion leaf deletion		✓	✓	✓	✓			✓	XML schema	unranked ordered labeled tree	top-down finite tree automaton
Ours	node relabeling leaf insertion leaf deletion		✓		✓			✓	✓	DTD	unranked ordered labeled tree	set of reg. exp.

Tableau A.1 – *Composants de définition de problème dans les approches de correction de document vers un langage*

a. Called a unary normal form DTD.
b. The treatment of attributes is limited in this approach. Moreover, attributes of the same element are seen as sequences rather than sets.
c. Called a disjunctive normal form DTD.
d. Called XML formatted word

Reference	Complexity estimation [a]		Proofs				Nature of data used in experiments	Availability				
	Time	Space	Correctness	Completeness	Termination	Complexity		Executable	Source code	License	Benchmark data	Web page
Boobna, de Rougemont [25]	$O(\|t\|)$						synthetic data, up to 800 nodes					[b]
Bertino et al. [19, 20]	$O((\|t\|+\|S\|) \times f_t^2)$						synthetic and real life data 11.111 nodes					
Xing et al. [125]	$O(\|t\| \times \|S\| \times \log\|S\|)$											
Staworko, Chomicki [104]	$O(\|S\|^2 \times \|t\|)$	$O(\|S\|^2 \times h(t))$					synthetic data 50 nodes	✓	✓	unknown		[c]
Suzuki [108]	$O(k \times \|\Sigma\| \times \|t\|^2 \times \|S\| \times R + k \times logk$		✓									
Tekli et al. [115]	$O((max(\|t\|,\|S\|)^3)$						synthetic and real-life data for classifying					
Staworko et al. [105]						[d]		✓	✓	unknown		[e]
Thomo et al. [117]	$O(th \times \|S\| \times \|\Sigma\|^2)^f$	$O(th \times \|S\| \times \|\Sigma\|^2)^g$	✓	✓		✓						
Svoboda, Mlýnková [112, 113]							synthetic data, 10,000 nodes	✓	✓	unknown		[h]
Ours	$O((f_t+1)\times (f_S)^{\|t\|+th}\times 6 \times \|\Sigma\|\times (\|t\|+th))^{th}$		✓	✓	✓	✓	real-life data, 450 tree nodes th=0,...,16	✓	✓	LGPL v3	✓	[i]

Tableau A.2 – *Propriétés et résultats des approches de correction de document vers un langage*

a. $|\Sigma|$ = size of the alphabet, $|t|$ = size of the tree, f_t = maximum fan-out of the tree, $h(t)$ = height of the tree, $|S|$ size of the schema, f_S = maximum fan-out of the schema, th = threshold, k = number of computed valid documents, R = maximum size of regular expressions in S

b. http ://www.lri.fr/∼mdr/xml.This demo does no longer produce any output.

c. http ://researchers.lille.inria.fr/ staworko/research/rhino-0.1.zip. It is unclear if this page contains the software described by this bibliographical reference.

d. Proofs of exponential complexity are provided for a broader problem – the one of consistent querying of XML documents.

e. http ://researchers.lille.inria.fr/ staworko/research/hippo/index.html. It is unclear if this page contains the software described by this bibliographical reference.

f. This is the time for building a VPA that recognizes all documents that are up to th far from a schema S.

g. This is the space for storing the VPA.

h. http ://www.ksi.mff.cuni.cz/ svoboda/projects/corrector/downloads.php

i. http ://codex.saclay.inria.fr/deliverables.php

Joshua AMAVI

Comparaison et évolution de schémas XML

Résumé. XML est devenu le format standard d'échange de données. Nous souhaitons construire un environnement multi-système où des systèmes locaux travaillent en harmonie avec un système global, qui est une évolution conservatrice des systèmes locaux. Dans cet environnement, l'échange de données se fait dans les deux sens. Pour y parvenir nous avons besoin d'un mapping entre les schémas des systèmes. Le but du mapping est d'assurer l'évolution des schémas et de guider l'adaptation des documents entre les schémas concernés. Nous proposons des outils pour faciliter l'évolution de base de données XML. Ces outils permettent de : (i) calculer un mapping entre le schéma global et les schémas locaux, et d'adapter les documents ; (ii) calculer les contraintes d'intégrité du système global à partir de celles des systèmes locaux ; (iii) comparer les schémas de deux systèmes pour pouvoir remplacer un système par celui qui le contient ; (iv) corriger un nouveau document qui est invalide par rapport au schéma d'un système, afin de l'ajouter au système. Des expériences ont été menées sur des données synthétiques et réelles pour montrer l'efficacité de nos méthodes.

Mots clés. XML, grammaire de haies, dépendances fonctionnelles, interopérabilité, mapping de schémas, échanges de données, inclusion relâchée, correction de documents en fonction d'un schéma, distance d'édition d'arbres.

Comparison and evolution of XML Schema

Abstract. XML has become the *de facto* format for data exchange. We aim at establishing a multi-system environment where some local original systems work in harmony with a global integrated system, which is a conservative evolution of local ones. Data exchange is possible in both directions, allowing activities on both levels. For this purpose, we need schema mapping whose is to ensure schema evolution, and to guide the construction of a document translator, allowing automatic data adaptation wrt type evolution. We propose a set of tools to help dealing with XML database evolution. These tools are used : (i) to compute a mapping capable of obtaining a global schema which is a conservative extension of original local schemas, and to adapt XML documents ; (ii) to compute the set of integrity constraints for the global system on the basis of the local ones ; (iii) to compare XML types of two systems in order to replace a system by another one ; (iv) to correct a new document with respect to an XML schema. Experimental results are discussed, showing the efficiency of our methods in many situations.

Keywords. XML, regular unranked-tree grammar, functional dependency, interoperability, schema mapping, data exchange, approximative inclusion, document-to-schema correction, tree edit distance.

Laboratoire d'Informatique Fondamentale d'Orléans
Bâtiment 3IA, rue Léonard de Vinci, B.P. 6759
45067 ORLEANS cedex 2, FRANCE