# Langages de transformation incrémentaux

Pierre GENEVÈS

13 juin 2003

## Remerciements

Je tiens à remercier Vincent Quint pour m'avoir accueilli au sein du projet WAM et pour la confiance qui m'a été accordée dans l'encadrement du stage. Je voudrais également remercier Jean-Yves Vion-Dury, et Nabil Layaïda pour leur aide et leur disponibilité. Enfin, je tiens à remercier toute l'équipe WAM dans laquelle j'ai travaillé, ainsi que celle du W3C, pour leur accueil chaleureux. J'ai pu découvrir la recherche en informatique dans un cadre agréable et propice au travail.

# Table des matières


**Première partie : état de l'art** **4**

**1 Introduction** **5**
1.1 Motivations et objectifs . . . . . 5
1.1.1 Transformation de documents XML . . . . . 5
1.1.2 Optimisation de performance . . . . . 5
1.1.3 Problématique de la transformation incrémentale . . . . . 6
1.2 Cadre du travail . . . . . 7
1.3 Organisation du rapport . . . . . 7

**2 Bases** **8**
2.1 Introduction . . . . . 9
2.1.1 Notion de document . . . . . 9
2.1.2 Document linéaire . . . . . 10
2.1.3 Document structuré . . . . . 10
2.1.4 Document structuré hypermédia . . . . . 11
2.1.5 Document balisé extensible . . . . . 11
2.2 Présentation de XML . . . . . 12
2.2.1 En bref . . . . . 12
2.2.2 Origines et objectifs . . . . . 12
2.2.3 Balisage XML . . . . . 13
2.2.4 Modèle de document XML . . . . . 13
2.2.5 Validité . . . . . 13
2.2.6 Classe de documents XML . . . . . 14
2.2.7 Structure arborescente d'un document XML . . . . . 14
2.2.8 Interfaces pour le traitement des documents XML . . . . . 15
2.3 Transformation de structures XML . . . . . 16
2.3.1 Traitement de documents : problématiques . . . . . 16
2.3.2 Principe de base de la transformation XML . . . . . 17
2.3.3 Un exemple . . . . . 17
2.3.4 Conclusion . . . . . 18

**3 Langages de transformation de documents XML** **19**
3.1 Introduction . . . . . 20
3.1.1 Possibilités pour exprimer des transformations . . . . . 20
3.1.2 Définition d'un langage de transformation . . . . . 20
3.2 Le langage de sélection XPath . . . . . 21
3.2.1 Introduction . . . . . 21
3.2.2 Modèle de document . . . . . 21

3.2.3 Sélection de noeuds dans un arbre . . . . . . . . . . . . . . . 22
3.3 Principaux langages de transformation XML . . . . . . . . . . . . . 24
3.3.1 XSLT . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
3.3.2 XQuery . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
3.3.3 Circus . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
3.3.4 CDuce . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
3.4 Eléments de comparaison . . . . . . . . . . . . . . . . . . . . . . 25
3.4.1 Modèle de document considéré . . . . . . . . . . . . . . . . 25
3.4.2 Modèle de transformation . . . . . . . . . . . . . . . . . . . 27
3.4.3 Transformation incrémentale . . . . . . . . . . . . . . . . . 28
3.5 Synthèse . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
3.6 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29

**4 Traitement incrémental** **31**
4.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
4.1.1 Traitement incrémental . . . . . . . . . . . . . . . . . . . . 32
4.1.2 Fondements . . . . . . . . . . . . . . . . . . . . . . . . . . 32
4.1.3 But . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
4.2 Traitement incrémental : définitions . . . . . . . . . . . . . . . . . 33
4.2.1 Caractérisation d'un algorithme incrémental . . . . . . . . . 33
4.2.2 Rendre un traitement incrémental . . . . . . . . . . . . . . 33
4.3 Catégorisation des travaux sur l'incrémentalité . . . . . . . . . . . 33
4.4 Méthodes générales pour le traitement incrémental . . . . . . . . . 34
4.4.1 Critères de classification . . . . . . . . . . . . . . . . . . . 34
4.4.2 Principales méthodes générales . . . . . . . . . . . . . . . . 35
4.5 Méthodes spécifiques pour le traitement incrémental des documents . 37
4.5.1 Traduction incrémentale . . . . . . . . . . . . . . . . . . . . 37
4.5.2 Traitement XML paresseux (Lazy XML processing) . . . . . 38
4.5.3 Transformation incrémentale de documents XML : incXSLT . 38
4.6 Synthèse . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39
4.6.1 Techniques génériques . . . . . . . . . . . . . . . . . . . . 39
4.6.2 Techniques spécifiques . . . . . . . . . . . . . . . . . . . . 40
4.6.3 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . 40

**Seconde partie : contribution** **40**

**5 Retour à la problématique de la transformation incrémentale** **42**
5.1 Contexte . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42
5.1.1 Une forte demande dans le contexte industriel . . . . . . . . . 43
5.1.2 Un domaine en plein essor . . . . . . . . . . . . . . . . . . . 43
5.1.3 Intérêt de la transformation incrémentale . . . . . . . . . . . 44
5.2 Formulation du problème . . . . . . . . . . . . . . . . . . . . . . . 44
5.3 Caractérisation d'un algorithme pour la transformation incrémentale . 44
5.4 Partitionnement en deux sous-problèmes . . . . . . . . . . . . . . . 45
5.5 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45

**6 Un système de réécriture pour la transformation** **46**
6.1 Introduction . . . . . 47
6.1.1 Vers un modèle de transformation . . . . . 47
6.1.2 Eléments favorables pour la transformation incrémentale . . . 47
6.1.3 Un nouveau modèle de transformation . . . . . 49
6.1.4 Motivations . . . . . 50
6.1.5 Modèle de document considéré . . . . . 51
6.2 Règle de transformation . . . . . 51
6.2.1 Syntaxe des expressions . . . . . 51
6.2.2 Sémantique des expressions . . . . . 52
6.2.3 Contraintes associées à une règle . . . . . 56
6.3 Modèle d'exécution . . . . . 57
6.3.1 Règle applicable . . . . . 57
6.3.2 Exécution de la transformation . . . . . 59
6.4 Exemple . . . . . 59
6.4.1 Modèles de document considérés . . . . . 59
6.4.2 Transformation . . . . . 61
6.4.3 Exécution sur une instance . . . . . 62
6.5 Conservation de l'historique d'exécution . . . . . 65
6.6 Restriction du modèle . . . . . 67
6.6.1 Opérateurs de génération . . . . . 67
6.6.2 Optique purement générative . . . . . 67
6.7 Répercution des modifications du document source . . . . . 67
6.7.1 Modélisation des modifications du document source . . . . . 67
6.7.2 Répercution d'une insertion dans l'arbre source . . . . . 69
6.7.3 Répercution d'une modification d'un noeud texte . . . . . 74
6.7.4 Répercution d'une suppression dans l'arbre source . . . . . 74
6.7.5 Récapitulatif . . . . . 76
6.7.6 Traitement de listes de modifications élémentaires . . . . . 77
6.7.7 Exemples . . . . . 79
6.8 Synthèse . . . . . 83
6.8.1 Evaluation du modèle . . . . . 83
6.8.2 Limites du modèle . . . . . 84

**7 Conclusion** **86**
7.1 Rappel des objectifs . . . . . 86
7.2 Rappel du travail réalisé . . . . . 86
7.3 Evaluation . . . . . 87
7.4 Perspectives . . . . . 88

**Références** **95**

# Chapitre 1

# Introduction

## 1.1 Motivations et objectifs

Avec l'émergence du standard XML, l'échange d'informations structurées entre logiciels a été unifié et standardisé. Le volume d'informations XML circulant sur la planète augmente à mesure que cette norme devient plus employée par l'industrie. Cette augmentation, favorisée par le développement du Web, suscite l'élaboration de langages, de systèmes et d'outils pour la visualisation et l'utilisation de ces informations structurées.

La forte pénétration de XML dans l'industrie logicielle fait qu'aujourd'hui, sans doute plus que jamais, les traitements de documents structurés prennent une importance croissante. Parmi ces traitements, la transformation de structures XML apparaît comme une opération fondamentale.

### 1.1.1 Transformation de documents XML

La transformation consiste à produire un fichier XML qui soit conforme à certaines exigences de structure, à partir d'un autre fichier XML structuré différemment. Les transformations sont omniprésentes dans la chaîne de traitement de l'information structurée sur le Web, depuis la création jusqu'à la présentation, en passant par l'échange entre applications dans les services Web. Quelques langages spécialisés dans la transformation de documents XML commencent à émerger, XSLT étant sans doute le plus répandu.

### 1.1.2 Optimisation de performance

L'optimisation des traitements en quête de meilleures performances est un sujet récurrent en informatique. Récemment, la robustesse des technologies XML a inspiré des cas d'utilisation extrêmes de XML. Des acteurs économiques comme les compagnies aériennes, la bourse et le secteur de la finance, par exemple, produisent d'ores et déjà des masses considérables d'informations structurées en XML à traiter. L'utilisation grandissante de XML dans l'industrie a motivé les recherches en matière de performance des traitements des informations XML. Cependant, dans la course à la performance, de nombreux problèmes demeurent irrésolus ou partiellement abordés, parmi lesquels celui de l'optimisation de l'exécution de transformations traitant un document source de très grande taille. Pour l'instant, les langages de transformation

comme XSLT s'avèrent plutôt mal adaptés au traitement de tels documents, car ils nécessitent que la totalité du document source soit en mémoire, même si seule une petite partie est finalement utilisée pour produire le document cible. Les recherches ont permis d'identifier d'autres problèmes, comme celui de la transformation en streaming. Celle-ci consiste à commencer à transformer un document source alors qu'il n'est pas encore connu en totalité, du fait de sa transmission non achevée. La transformation s'effectue alors sur les fragments du document source à mesure qu'ils sont reçus. La transformation en streaming peut ainsi être vue comme un cas particulier de la transformation incrémentale.

Au moins une limite des langages de transformation XML actuels a pu être clairement identifiée : ces langages ont été conçus dans une optique "batch", où la totalité d'un document est transformée d'un coup. Or, cette approche peut s'avérer mal adaptée aux applications dynamiques, dans lesquelles le document à transformer peut subir des changements qui doivent être répercutés immédiatement au document résultat, sans relancer la totalité du processus de transformation.

### 1.1.3 Problématique de la transformation incrémentale

La transformation incrémentale est un procédé visant à optimiser l'exécution d'une transformation déjà appliquée en tirant partie de son exécution précédente.

Considérons un document XML et une transformation. Une première application de la transformation traite le document source pour produire un document cible.

Lorsqu'une modification du document source survient, le document cible précédemment obtenu n'est plus à jour. Le nouveau document cible correspondant à la transformation du document source modifié est requis. Pour l'obtenir, une première approche est de réexécuter complètement la transformation sur le nouveau document source, de manière à ce qu'il soit traité en totalité, comme si on ne l'avait jamais effectué auparavant. Si la modification apportée au document source est légère, il y a de fortes chances pour que le résultat de la seconde application de la transformation soit relativement proche du premier. En effet, de légères modifications du document d'entrée sont souvent seulement à l'origine de légers changements dans le document cible. De nombreux et coûteux calculs redondants auront donc été effectués.

Une seconde approche consiste au contraire à prendre en compte le fait que la transformation a déjà été exécutée une fois. De manière à éviter les calculs redondants, le document cible de la première application de la transformation pourrait donc être réutilisé et mis à jour pour refléter les modifications intervenues dans le document source. Cette seconde approche consiste à réagir aux modifications apportées sur le document source en les répercutant directement sur le document cible, sans réappliquer la totalité du processus de transformation. Cette approche visant à optimiser la performance est nommée incrémentale.

C'est dans cette direction que s'inscrit ce travail de DEA.

Des travaux sur la transformation incrémentale de documents XML ont déjà été menés au sein du projet WAM. Ces travaux ont notamment débouchés sur l'élaboration d'un processeur de transformation incrémental pour le langage XSLT. Ils ont également souligné l'intérêt d'une analyse statique de la transformation pour préparer l'exécution incrémentale.

Ce travail de DEA poursuit l'étude de ce domaine encore peu exploré pour tenter de comprendre quels procédés utiles peuvent être employés pour la transformation incrémentale.

## 1.2 Cadre du travail

Ce travail s'est effectué au sein du projet de recherche *Web, Adaptation, et Multimédia* (WAM) de l'INRIA Rhône-Alpes. Le projet WAM a été créé en janvier 2003 pour donner suite aux travaux effectués sur les documents structurés au sein du précédent projet OPERA.

Les thèmes scientifiques abordés par le projet WAM sont issus des évolutions du Web. Parmi ses axes de recherche, le projet se focalise notamment sur la transformation de documents, qui est considérée comme une classe de traitement générique des documents du Web.

Ce travail de DEA autour des langages de transformation XML et de la transformation incrémentale s'inscrit sur cet axe de recherche.

Pendant ce stage, l'idée d'un nouveau langage de transformation XML a germée dans l'équipe de recherche. Ce travail de DEA a en partie contribué à l'élaboration de ce nouveau langage, encore à ses prémices.

## 1.3 Organisation du rapport

Ce rapport est organisé en deux parties qui sont brièvement introduites ci-après.

**Etat de l'art** Dans une première partie, un état de l'art du domaine est présenté. Il débute en donnant les notions de base qui serviront par la suite (chapitre 2), puis poursuit en étudiant d'une part les langages de transformation de structures XML (chapitre 3), et d'autre part les méthodes employées pour le traitement incrémental (chapitre 4).

**Contribution** La seconde partie du rapport expose la contribution apportée dans le cadre de ce stage. Tout d'abord, nous précisons la problématique que nous considérons (chapitre 5). Ensuite, nous présentons notre nouveau modèle de transformation, dans l'optique de la transformation incrémentale (chapitre 6). Ce chapitre débute par quelques critères techniques qu'un modèle se doit de vérifier afin de se prêter à la transformation incrémentale. Ensuite, le modèle de transformation proposé est introduit pour tenter de répondre à ces critères. Nous présentons alors la modélisation que nous faisons des modifications apportées au document source. Sur cette base, des propositions sont faites en vue de la transformation incrémentale dans notre modèle. Ce chapitre s'achève en expliquant notamment les nombreuses pistes restant à étudier.

Finalement, une conclusion rappelant brièvement le problème et le travail réalisé se propose d'évaluer les résultats obtenus, et de donner des perspectives pour un travail ultérieur.

# Chapitre 2

# Bases

## Résumé

Ce premier chapitre introduit les bases nécessaires pour aborder la suite de ce mémoire. Il débute par une introduction à la notion de document avec des définitions et des problématiques liées au traitement des documents électroniques. Ensuite, XML et ses concepts sont présentés. Enfin, des éléments sont donnés pour permettre de déceler l'importance du rôle que joue la transformation de structures XML.

## Contenu

**2.1 Introduction** . . . . . . . . . . **9**
- 2.1.1 Notion de document . . . . . . . . . . 9
- 2.1.2 Document linéaire . . . . . . . . . . 10
- 2.1.3 Document structuré . . . . . . . . . . 10
- 2.1.4 Document structuré hypermédia . . . . . . . . . . 11
- 2.1.5 Document balisé extensible . . . . . . . . . . 11

**2.2 Présentation de XML** . . . . . . . . . . **12**
- 2.2.1 En bref . . . . . . . . . . 12
- 2.2.2 Origines et objectifs . . . . . . . . . . 12
- 2.2.3 Balisage XML . . . . . . . . . . 13
- 2.2.4 Modèle de document XML . . . . . . . . . . 13
- 2.2.5 Validité . . . . . . . . . . 13
- 2.2.6 Classe de documents XML . . . . . . . . . . 14
- 2.2.7 Structure arborescente d'un document XML . . . . . . . . . . 14
- 2.2.8 Interfaces pour le traitement des documents XML . . . . . 15

**2.3 Transformation de structures XML** . . . . . . . . . . **16**
- 2.3.1 Traitement de documents : problématiques . . . . . . . . . 16
- 2.3.2 Principe de base de la transformation XML . . . . . . . . . 17
- 2.3.3 Un exemple . . . . . . . . . . 17
- 2.3.4 Conclusion . . . . . . . . . . 18

## 2.1 Introduction

### 2.1.1 Notion de document

Le terme *document* provient étymologiquement du latin *documentum* signifiant "ce qui sert à instruire". Dans la langue française, le terme document est défini par "un écrit servant de preuve ou de renseignement". La notion de document est ainsi essentiellement porteuse du sens de "transmission de connaissance, communication d'information" . La parole constitue une manière de transmettre la connaissance. Le document a étendu cette capacité de transmission avec la mise sous forme persistante de l'information : des roches ornées de peintures préhistoriques aux DVD gravés d'informations utilisés aujourd'hui, de nombreux supports (papyrus, papier, etc.) ont été employés pour assurer la persistance.

Avec cette approche, un document est donc une forme pérenne permettant d'assurer la transmission d'une information.

Cependant, avec l'avènement de l'informatique, cette définition n'est plus suffisante pour capturer la notion de document. En particulier, à un document n'est plus associée une seule forme mais différentes formes, selon le contexte. Un document garde pour objectif la communication de connaissances, ce qui permet de l'opposer à des "données". Mais aujourd'hui un document peut prendre différentes formes afin de remplir cet objectif de communication. Un document peut à présent être :

- dynamique : il peut dépendre du temps s'il est par exemple généré dynamiquement à partir d'une base de données évoluant elle-même dynamiquement
- multi-média : il peut faire intervenir du texte, des images, de la vidéo ou du son
- multi-dimensionnel : il peut comporter des animations qui modifient sa présentation avec le temps. Par exemple un document peut contenir des animations de mouvement d'objets, ou de changement d'apparence. D'une façon générale, la description d'un document se fait à présent autour de quatre dimensions : logique, spatiale, temporelle et hypermédia [21]
- interactif : il peut s'adapter et varier avec l'interaction de l'utilisateur

La notion de document a ainsi subi une très forte évolution depuis une vingtaine d'années. En intégrant ces différentes caractéristiques elle a aussi intégré le concept qu'un document puisse prendre différentes formes selon le contexte. En conséquence, la notion de document devient à présent difficile à identifier car de nombreuses facettes différentes peuvent être considérées pour établir une définition du document. Un document se distingue notamment par les types de contenus (textes, images, vidéos, sons), l'organisation des objets qui le composent (relations entre objets) les types de support de restitution (papier, écran, synthèse vocale), les traitements que l'on peut effectuer (lecture/écoute, navigation, annotation, édition, transformation). Plusieurs définitions couvrant partiellement la notion de document ont été proposées [42].

Une définition, également issue du domaine informatique, insiste sur la notion de propriétés et d'usages du document [46] : *"Le document désigne un ensemble cohérent et fini d'informations, plus ou moins structurées, perceptibles, à usage défini et représenté sur un support concret."*

Un point intéressant de cette définition est le qualificatif perceptible associé aux informations du document. Celui-ci permet de s'abstraire de la forme avec laquelle le document est restitué. En séparant les notions de contenu sémantique et de restitution du document, cette définition permet de considérer qu'un document puisse prendre plusieurs formes, sur différents types de restitution. La perception des informations d'un document peut par exemple s'effectuer de manière graphique (avec un écran), sonore (avec un dispositif de synthèse vocale), ou pourquoi pas mécaniquement (avec un dispositif à retour d'effort ou un terminal braille). Pour chaque type de restitution, différentes présentations des informations d'un document peuvent être envisagées. Par exemple, considérons des prévisions météorologiques, ces informations ne seront pas présentées de la même manière sur un PDA, sur un ordinateur ou par téléphone.

Dans la suite, nous tentons de présenter comment les modèles de la notion de document ont évolué pour en capturer de plus en plus d'aspects.

### 2.1.2 Document linéaire

A l'origine, les premiers écrits, tout comme la parole, étaient des processus sériels adaptés aux caractéristiques de la perception humaine (mémoire auditive limitée, etc.) Un premier modèle de la notion de document, le document linéaire, modélise ainsi les documents par des flots de données. De nombreux travaux ont été effectués dans le domaine du génie documentaire et ont proposé un concept moins sériel et à présent bien répandu : celui du document structuré.

### 2.1.3 Document structuré

Les premières définitions du document structuré datent des années 80 [15, 12]. Les documents structurés sont fondés sur les relations logiques entre les différents composants du document. La notion de document structuré introduit principalement deux concepts : la *structure logique* et le *modèle* d'un document.

**Structure logique** Un document est considéré comme une structure abstraite qui rassemble des éléments typés. Dans un document tel qu'un article, des exemples d'éléments typés sont les titres, les chapitres, les sections, les paragraphes, etc. La structure elle-même est une hiérarchie de tels éléments. Le plus bas niveau d'éléments est constitué par le contenu du document : les chaînes de caractères, les symboles mathématiques, les images, et les éléments graphiques. Tous ces éléments possèdent divers types de relations : une section doit contenir un titre, des paragraphes et éventuellement des sous-sections ; un paragraphe soit suivre un autre paragraphe ; un paragraphe peut faire référence à un élément bibliographique, etc. Les éléments avec leurs relations constituent la structure logique d'un document.

**Modèle de document** La structure logique d'un document est régie par un modèle, qui définit un type de document en spécifiant tous les types d'éléments qui peuvent être utilisés dans la structure logique d'un document de ce type. Un modèle spécifie aussi les attributs qui peuvent être associés à chaque type d'éléments, et toutes les relations structurelles qui peuvent être établies entre les éléments. Tous les documents du type défini doivent se conformer à leur modèle. En définissant d'autres modèles de document, il est possible de définir d'autres types de document, qui incluent d'autres types

d'éléments avec des relations différentes. Les documents qui obéissent à un même ensemble de contraintes sont ainsi regroupés en classes de document. Lorsqu'un document respecte un modèle de document, on dit que c'est une *instance* de ce modèle. La définition formelle d'un modèle de document s'effectue en spécifiant un schéma de structure à l'aide d'un formalisme appelé *schéma*. Un schéma permet d'exprimer des contraintes sur la classe de document.

Un point intéressant introduit par les documents structurés est la **séparation entre le contenu et la présentation d'un document**. En effet, comme la structure logique d'un document est toujours conforme au modèle de document, il est possible de définir génériquement l'aspect graphique des documents : des règles de présentation peuvent être associées aux types d'éléments et d'attributs. Elles spécifient des propriétés comme la police, la couleur, l'espacement, l'indentation, etc. Les règles de présentation sont regroupées au sein de modèles de présentations : un modèle de présentation contient toutes les règles de présentation nécessaires pour spécifier l'aspect graphique de tous les types d'éléments et d'attributs définis dans le modèle de document.

### 2.1.4 Document structuré hypermédia

Le document structuré voit le document comme un ensemble d'objets organisés hiérarchiquement. Ainsi, la structure logique du document est arborescente. Cette approche doit être étendue pour couvrir les besoins hypertexte pour lesquels une organisation de liens doit être ajoutée à l'organisation hiérarchique des informations.

Les liens couvrant les besoins hypertextes peuvent avoir une sémantique de désignation, d'exploration (exemple : menus de navigation), de fragmentation (exemple : une encyclopédie fragmentée en un ensemble de pages HTML) ou de transclusion (la transclusion est la réutilisation d'un document ou d'une partie d'un document dans le rendu d'un autre document ; un exemple est le référencement au sein d'une page HTML d'une image placée sur un serveur distant ; une transclusion diffère d'une copie par le fait que seule la référence à l'objet distant est stockée).

Quelle que soit leur sémantique, des liens non hiérarchiques doivent pouvoir relier des éléments indépendamment de la structure d'arbre. Ces liens hypertexte enrichissent la notion de document structuré puisqu'ils ajoutent de l'interactivité lors de la consultation du document, en rendant possible la navigation. Ils enrichissent également la structure d'arbre du document en structure de graphe. Ces réseaux de liens rendent difficile la tâche de définir une frontière du document. Où un document s'arrête-t-il ? Le Web est-il un énorme document ?

### 2.1.5 Document balisé extensible

Récemment, les travaux effectués autour de tous ces problèmes ont débouché sur une tentative de normalisation avec un ensemble de règles, de lignes directrices, de conventions pour la conception de documents structurés : Extensible Markup Langage (XML) [68]. Ce standard défini par le World Wide Web Consortium (W3C) [61] repose sur les concepts de documents structurés présentés précédemment et ouvre de plus la voie de l'uniformisation, de l'interopérabilité, de l'indépendance de la plateforme, de

l'extensibilité, et de l'internationalisation de la notion de document. Cette norme sera utilisée tout au long de ce mémoire et fait l'objet de la section suivante.

## 2.2 Présentation de XML

### 2.2.1 En bref

La norme XML [68] fournit avant tout une manière simple et standard d'échanger des informations textuelles structurées entre des applications logicielles.

### 2.2.2 Origines et objectifs

A l'origine existait SGML, un langage de représentation de données et documents structurés assez complexe, dont l'idée principale est de fournir une représentation indépendante du système de traitement. D'autre part, HTML, une application de SGML, était bien répandu du fait de sa simplicité, mais limité par sa sémantique de présentation.

Le besoin de publier des données SGML sur le Web est l'objectif initial qui a donné lieu à la création de XML. XML tente de répondre aux objectifs suivants [68] :

1. XML doit être facilement utilisable sur le Web
2. XML doit supporter une grande variété d'applications
3. XML doit être compatible avec SGML
4. Il doit être facile d'écrire des programmes qui traitent des documents XML
5. Le nombre d'options doit être réduit au minimum, idéalement à zéro
6. Les documents XML doivent être lisibles et raisonnablement clairs
7. La conception de XML doit être menée rapidement
8. La description de XML doit être formelle et concise
9. Les documents XML doivent être faciles à créer
10. La concision du balisage XML est d'une importance minime

Au final, XML est un nouveau standard inspiré de SGML et de HTML. Il serait possible de discuter longuement jusqu'à quel point XML est conforme à chaque objectif précédent. Nous retiendrons simplement quatre grandes idées clés de XML que nous détaillerons par la suite :

- **un langage pour créer des langages :** XML permet de créer des langages adaptés au type d'information à décrire.
- **l'indépendance du balisage :** le balisage XML est indépendant des traitements à effectuer.
- **la validité :** XML introduit deux niveaux d'exigence sur les documents, ce qui favorise leur traitement.
- **le processeur :** XML définit le processeur comme une abstraction claire de l'application : celle qui analyse le document en interprétant les balises.

Ces caractéristiques rendent les documents XML particulièrement aptes à êtres traités et remplissent notamment les objectifs précédents n°2 et 4. Elles sont détaillées dans la suite.

### 2.2.3 Balisage XML

De manière à conserver la compatibilité avec SGML, XML est un langage de balisage descriptif qui utilise des balises (des mots encadrés par ’<’ et ’>’) et des attributs (de la forme nom="valeur"). Le **contenu** est structuré en **éléments typés** qualifiés par des **attributs** avec des **valeurs** :

```
<livre isbn="2-7324-2523-0">
        <auteur>Yann Arthus-Bertrand</auteur>
        <titre>La terre vue du ciel</titre>
</livre>
```

**Séparation contenu et présentation** Alors que HTML définit la signification de chaque balise et de chaque attribut (et souvent la manière dont le texte qu’ils encadrent apparaîtra dans un navigateur), XML utilise les balises seulement pour délimiter les éléments de données et laisse l’entière interprétation des données à l’application qui les lit. Les balises (tags) XML décrivent donc la structure du texte et non sa présentation.

**Indépendance** Le balisage est indépendant du système où il est créé, et indépendant des traitements à effectuer.

**Langages XML** XML est un langage pour créer des langages. Définir un langage XML (un "vocabulaire XML", une "grammaire XML") consiste d’une part à spécifier un modèle de document et d’autre part à associer une sémantique à chaque élément, attribut et assemblage de structure du langage.

### 2.2.4 Modèle de document XML

La notion de modèle de document structuré a été introduite en 2.1.3. Spécifier un modèle de document XML consiste à donner :
- les éléments de base non décomposables qui constituent le contenu
- les noms des éléments autorisés
- les noms et valeurs permises des attributs du langage
- l’assemblage des éléments dans la structure, i.e. les compositions possibles d’éléments ; le contenu autorisé de chaque élément ; quels attributs sont autorisés avec quels éléments, etc.

Concrètement, le modèle de document est spécifié à l’aide d’un méta-langage comme DTD [68], XML Schema [69] ou autre [27, 14]. Un schéma de structure XML écrit avec un tel méta-langage exprime les contraintes que doivent respecter les documents pour se conformer au modèle.

### 2.2.5 Validité

Une notion essentielle apportée par XML, et qui n’existait pas dans SGML, est la définition de deux niveaux de validité, à travers les propriétés de document *bien-formé* et de document *valide*.

**Document bien formé** Un document XML est bien-formé s'il respecte notamment les règles suivantes :

– il existe une balise englobant toutes les balises du document
– toute balise ouverte doit être refermée (à <balise> doit être associé </balise>). Lorsqu'un élément est vide, les balises peuvent être simplifiées : <balise></balise> est identique à <balise/>
– Si un élément contient d'autres éléments alors appelés éléments fils, les balises correspondantes doivent être fermées avant la fermeture de celles de leur parent. Par exemple, <r><p><a>texte</p></a></r> n'est pas un document bien formé car la balise <a> devrait être fermée avant </p>.
– Les valeurs des attributs doivent être entourées d'une paire de guillemets ou d'apostrophes.

Ces règles simples permettent d'interpréter le document sous une forme arborescente (c.f. 2.2.7) pour lui appliquer des traitements.

**Document valide** Un document XML est valide par rapport à un modèle s'il se conforme aux exigences de structure décrites dans le schéma du modèle. Un document valide est évidemment bien-formé.

Un point intéressant avec ces deux niveaux de validité est qu'il devient possible de traiter un document dès lors qu'il est bien formé, sans toutefois faire l'hypothèse beaucoup plus forte qu'il soit valide par rapport à un modèle.

### 2.2.6 Classe de documents XML

Un modèle de document définit une classe de documents XML : l'ensemble des arbres XML valides par rapport au modèle. XML permet de créer des langages adaptés au type d'information à décrire. XHTML [67] pour l'hypertexte simple, MathML [64] pour les expressions mathématiques, SVG [66] pour les graphiques vectoriels, en sont des exemples, ainsi que SMIL [65], la classe des documents XML multimédia synchronisés.

### 2.2.7 Structure arborescente d'un document XML

En se basant sur la spécification Document Object Model (DOM) [63], un document XML bien-formé peut être représenté par un arbre. Ainsi il existe deux représentations d'un document XML : la forme linéarisée (figure 2.1) et la forme arborescente (figure 2.2).

```
<livre isbn="2-7324-2523-0"><auteur>Yann
Arthus-Bertrand</auteur><titre>La terre vue du
ciel</titre><sommaire><chapitre>L'état du monde en l'an
2000</chapitre><chapitre>Du paléolithique à la
mondialisation</chapitre></sommaire></livre>
```

FIG. 2.1 – Forme linéarisée

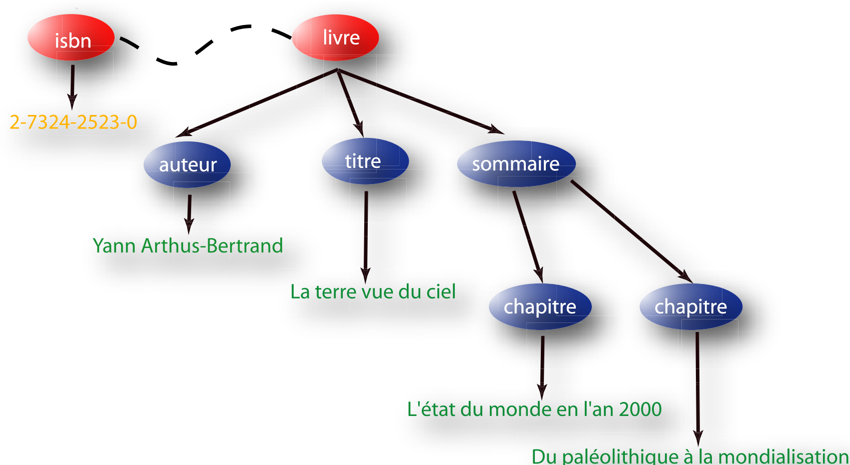


FIG. 2.2 – Forme arborescente

Trois sortes de noeuds sont présents dans l'exemple précédent :
- les noeuds éléments, qui ont un **nom**
- les noeuds attributs, qui ont un **nom** et une **valeur**
- les noeuds textes, qui ont une **valeur**

Selon la spécification DOM, les noeuds éléments et les noeuds textes sont ordonnés, tandis que les noeuds attributs ne le sont pas.

Alors que la forme linéarisée est la représentation adaptée au stockage et à l'échange de documents, il est généralement plus facile de raisonner sur la forme arborescente pour concevoir des traitements. Ces traitements sont conçus à l'aide d'une interface qui permet de manipuler les documents XML.

### 2.2.8 Interfaces pour le traitement des documents XML

La première étape d'un processeur XML traitant un document est de construire une représentation du document XML propice au traitement. On distingue actuellement deux approches possibles :
- l'approche hiérarchique, qui consiste à construire une structure hiérarchique contenant des objets représentant les éléments du document, et dont les méthodes permettent d'accéder aux propriétés. La principale interface utilisant cette approche

est Document Object Model (DOM) [63], qui repose sur la forme arborescente du document présentée précédemment.
– l'approche événementielle, qui consiste à réagir à des événements (comme le début d'un élément, la fin d'un élément, une valeur d'attribut, etc.) et de renvoyer le résultat à l'application utilisant cette API. Simple API for XML (SAX) [60] est la principale interface utilisant l'approche événementielle.

L'approche hiérarchique impose de construire un arbre en mémoire contenant l'intégralité des éléments du document en mémoire. Pour de gros documents, cette approche peut donc s'avérer peu performante car tout l'arbre du document est manipulé quelque soit le traitement qui lui est appliqué. Au contraire, l'approche événementielle permet de traiter uniquement ce qui est nécessaire, mais ne donne accès à chaque instant qu'à une petite partie du document.

## 2.3 Transformation de structures XML

### 2.3.1 Traitement de documents : problématiques

D'une manière générale, avec l'évolution du modèle de la notion de document surgissent de nouvelles problématiques concernant le traitement des documents. Lorsque de nouveaux aspects de la notion de document sont pris en compte, de nouveaux problèmes liés à ces aspects se répercutent au niveau des traitements de documents. Par exemple, les liens hypermedia étendent la notion de document structuré en y ajoutant de l'interactivité via la navigation rendue possible ; mais ces liens soulèvent le problème de la frontière du document et posent de nouveaux problèmes relatifs aux traitements (par exemple, comment définir la copie d'un document hypermédia ?)

Avec le développement de XML et du Web, les traitements de documents structurés prennent une importance croissante. Par la même occasion, de nouvelles problématiques prennent forme.

**Problème de l'évolution des modèles de document** XML facilite la création de nouveaux langages adaptées à des besoins particuliers (ex : MathML, SVG, SMIL, etc.) et donc de nouvelles classes de documents. Un des problèmes qui se posent est celui de faire face à l'évolution des modèles de documents en permettant la réutilisation de documents existants. Lorsqu'un modèle de document évolue, il est nécessaire de permettre le changement correspondant des instances. D'une manière plus générale, puisque les modèles évoluent, il faut fournir un moyen de réutiliser des documents existants en les faisant eux-mêmes évoluer pour appartenir à de nouvelles classes.

**Problème de la compatibilité des modèles pour l'interopérabilité** XML facilite l'échange d'informations entre applications en apportant une manière standard de communiquer. Un problème qui se pose est de franchir la barrière des modèles de document : adapter les documents XML issus d'un logiciel au modèle de document utilisé par un autre logiciel, afin de rendre possible l'interopérabilité de logiciels traitant chacun intrinsèquement un modèle distinct de document XML.

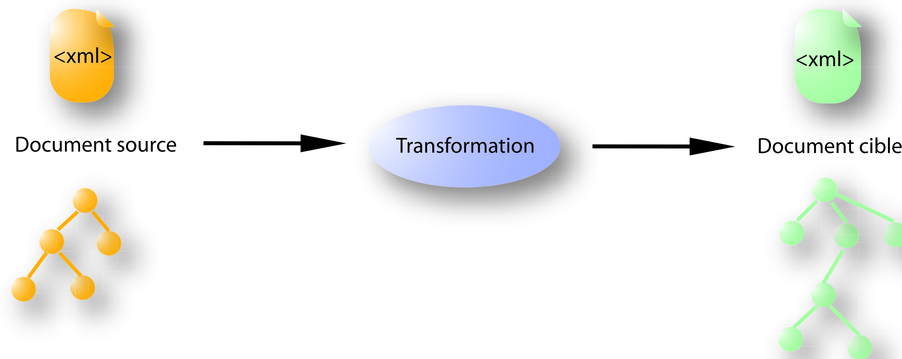


FIG. 2.3 – Processus de transformation

**Problèmes de formatage, d'adaptation, de présentation** XML facilite la séparation du contenu de la présentation, c'est-à-dire la séparation de l'information proprement dite des détails sur la manière avec laquelle elle sera présentée sur un dispositif particulier. Un problème est d'obtenir à partir de l'information un document de présentation (par exemple un document HTML ou bien SVG), formaté et adapté au contexte, pouvant être affiché ou imprimé afin de communiquer avec un lecteur humain.

Avec cette approche, et sachant qu'au final un document XML est analysé par un logiciel, il est possible d'isoler un besoin particulier fondamental : celui de transformer un document XML en un autre document, pour en faire une instance d'une autre classe.

### 2.3.2 Principe de base de la transformation XML

La transformation de structures XML est le moyen qui permet de passer d'un document XML d'une certaine classe à un autre document, de la même classe ou d'une classe différente. Le document en entrée de la transformation est appelé document source. Le document en sortie est nommé document cible (figure 2.3).

Afin d'être réutilisable et générique, c'est-à-dire applicable aux documents appartenant à une même classe, la spécification de la transformation s'appuie généralement sur le modèle du document source. Le rôle du modèle de document source est de rendre explicite la connaissance qu'on a a priori d'un document, de manière à déduire des propriétés. Par exemple, si un modèle spécifie que le document doit obligatoirement comporter un titre et que celui-ci doit se trouver à un endroit bien précis, la transformation n'a pas à tester la présence du titre. Si la transformation vise à garantir la validité du document cible par rapport à un modèle, elle peut également s'appuyer sur un modèle de document cible.

### 2.3.3 Un exemple

Sur le Web, on peut trouver différentes grammaires XML et les schémas correspondants pour modéliser la musique. L'utilisation de tels vocabulaires XML rend par

exemple possible l'incorporation de séquences musicales dans des documents Web, le contrôle du son des systèmes sonores domestiques via le Web, une notation graphique des partitions de musique, etc.

Une rapide recherche sur le Web permet de trouver au moins trois schémas différents qui ont chacun été inventé avec différents objectifs à l'esprit :

- MusicML, par exemple, fournit un moyen de représenter la notation musicale graphiquement
- ChordML est quant-à-lui conçu pour coder l'accompagnement harmonique des paroles
- enfin, Music Markup Language (MML) est une tentative pour modéliser de nombreuses sortes d'événements musicaux pour l'analyse musicologique poussée

On peut ainsi utiliser la transformation de structures XML pour traiter la musique de nombreuses manières. On pourrait par exemple convertir la musique d'une de ces représentations en une autre, par exemple de MusicML à MML. Il serait possible d'utiliser la transformation pour convertir la musique de n'importe laquelle de ces représentations en notation visuelle, en produisant un document SVG [66] contenant des graphiques vectoriels. Ces deux applications de la transformation illustrent le cas où on transforme un document d'une certaine classe en un document d'une autre classe, différente. On pourrait également se servir de la transformation de structure XML pour réaliser une transformation musicale, telle que transposer la musique à une hauteur d'accord différente. Ce cas illustre quant-à-lui une application de la transformation où la classe du document cible de la transformation est la même que celle du document source. Enfin, il serait possible d'extraire les paroles et produire une page HTML à l'aide d'une transformation. Ce dernier cas illustre l'application de la transformation certainement la plus courante aujourd'hui : celle qui consiste à produire du HTML publiable sur le Web à partir de documents XML d'une certaine classe.

### 2.3.4 Conclusion

Le but premier de XML est de fournir le moyen de créer de nouveaux langages adaptés à des besoins particuliers (ex : MathML, SVG, SMIL, etc.) et donc de nouvelles classes de documents. Le but premier de la transformation de structures XML est de fournir la possibilité de transmettre des informations d'une classe de documents XML à une autre. La transformation est donc un traitement fondamental complémentaire de XML.

# Chapitre 3

# Langages de transformation de documents XML

## Résumé

Avec l’émergence du standard XML et le nombre croissant de documents XML, quelques langages spécialisés dans la transformation de structures XML ont été proposés. Ce chapitre tente de refléter ce qui existe à l’heure actuelle en synthétisant l’étude des principaux langages de transformation.

## Contenu

**3.1 Introduction** . . . . . . . . . . . . . . . . . . . . . . . . . . . . **20**
3.1.1 Possibilités pour exprimer des transformations . . . . . . . 20
3.1.2 Définition d’un langage de transformation . . . . . . . . . 20
**3.2 Le langage de sélection XPath** . . . . . . . . . . . . . . . . . . . . **21**
3.2.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . 21
3.2.2 Modèle de document . . . . . . . . . . . . . . . . . . . . 21
3.2.3 Sélection de noeuds dans un arbre . . . . . . . . . . . . . 22
**3.3 Principaux langages de transformation XML** . . . . . . . . . . **24**
3.3.1 XSLT . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
3.3.2 XQuery . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
3.3.3 Circus . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
3.3.4 CDuce . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
**3.4 Eléments de comparaison** . . . . . . . . . . . . . . . . . . . . . **25**
3.4.1 Modèle de document considéré . . . . . . . . . . . . . . . 25
3.4.2 Modèle de transformation . . . . . . . . . . . . . . . . . . 27
3.4.3 Transformation incrémentale . . . . . . . . . . . . . . . . 28
**3.5 Synthèse** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **29**
**3.6 Conclusion** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **29**

## 3.1 Introduction

### 3.1.1 Possibilités pour exprimer des transformations

Les possibilités qui se présentent pour exprimer une transformation de structure XML se divisent en deux catégories : celles qui reviennent à utiliser un langage de programmation classique ; ou bien celles qui emploient un langage spécifiquement dédié à la transformation XML.

**Langages de programmation et librairies** Exprimer une transformation de structure XML avec un langage de programmation classique est possible. Avec cette approche, du code (C, C++, Java, etc.) est écrit pour lire le contenu du document source, interpréter sa structure et sa sémantique, et pour générer un nouveau document avec une structure potentiellement différente. Cela suppose de modéliser les structures de données employées (par exemple les types élément, attribut, document structuré) avec le langage choisi. Des librairies supportant DOM ou SAX sont disponibles pour faciliter cette implantation avec les principaux langages.

Cette approche de bas niveau est couramment employée dans les applications de base de données sur le Web dans lesquelles les données de la base sont transformées en contenu de pages HTML. Des logiciels ou boîtes à outils existent sur le marché pour tenter de faciliter l'écriture du code spécifique.

Cette approche peut offrir de très bonnes performances, puisque une application spécifique est réalisée. Cependant, dans les cas où elle est raisonnablement applicable, elle pose des problèmes importants de flexibilité, de maintenance, et de passage à l'échelle. En effet, une transformation très simple nécessite déjà du code conséquent, qui risque fort d'évoluer en code "spaghetti". L'approche souffre du fait que le langage de programmation demeure générique, et n'est pas spécifiquement étudié pour la transformation de documents. De plus, les problèmes ne sont pas uniquement de l'ordre du génie logiciel : le langage de programmation ne fournit intrinsèquement aucune propriété sur les transformations (comme la facilité de faire évoluer une transformation en fonction de l'évolution du modèle source ou cible, des propriétés concernant la transformation incrémentale, ou bien la garantie de validité du document cible). Enfin, le langage de programmation ne se prête pas non plus facilement à l'établissement de telles propriétés.

**Langages spécifiquement dédiés** De nombreuses recherches ont été effectuées afin d'identifier les transformations récurrentes, de les interpréter, et de mieux les exprimer. Ces travaux ont conduit à la définition de véritables langages de transformation, avant même l'émergence de XML. Parmi ces langages spécialement dédiés à la transformation on peut notamment citer : [56, 13, 28, 3, 20, 6]. Un état de l'art complet de ces langages de transformation a été accompli par Stéphane Bonhomme dans sa thèse [6] en 1998. Avec l'avènement de XML, et depuis 1998, de nombreux nouveaux langages de transformation de structures XML ont été proposés.

### 3.1.2 Définition d'un langage de transformation

#### Principe

Le principe de tout langage de transformation de structure XML est de permettre d'exprimer le moyen de transformer un document instance d'un modèle source en un

document instance d'un modèle cible. A cet effet, tout langage de transformation est articulé autour de trois phases (ou "fonctions" en faisant abstraction du temps) :

1. la sélection et l'extraction de fragments dans l'arbre source ;
2. la réorganisation des fragments sélectionnés, la création de nouveaux fragments et la combinaison de ceux-ci ;
3. la génération du document cible à partir de ces fragments

**Le moyen : les règles de transformation**

D'une manière analogue à un langage de programmation et au principe "diviser pour régner", un langage de transformation fournit un moyen d'éclater l'expression d'une transformation en plusieurs unités élémentaires : les *règles de transformation*. Celles-ci constituent le support des techniques citées précédemment. A cet effet, une règle de transformation est généralement constituée de deux parties :

– La partie gauche (ou LHS : Left Hand Side) permet la sélection et l'extraction de noeuds du document source, en autorisant des tests de structure et de valeurs à l'intérieur de la structure. De nombreuses techniques ont été proposées pour permettre l'exploitation des informations de l'arbre source. La sélection peut par exemple être faite avec des expressions régulières d'arbres [6, 17], par pattern matching. Récemment, ce besoin a été isolé et a fait l'objet d'une normalisation à travers le standard XPath [70] recommandé par le W3C. Cette norme fait l'objet de la section suivante.
– La partie droite (ou RHS : Right Hand Side) permet l'utilisation du contexte issu de la partie gauche (fragments de structures et valeurs extraites, etc.) pour construire l'arbre cible en utilisant des primitives de génération de fragments d'arbre.

Notons qu'une règle de transformation nécessite aussi une notion de contexte dans l'arbre cible qui précise à quel endroit dans l'arbre cible la génération a lieu. Cette notion de contexte dans l'arbre cible est fréquemment rendue implicite par le modèle d'exécution des règles (comme dans XSLT).

## 3.2 Le langage de sélection XPath

### 3.2.1 Introduction

XPath est un langage permettant de désigner des parties d'un document XML. Il permet de naviguer dans un arbre XML et de sélectionner un ensemble de noeuds.

### 3.2.2 Modèle de document

XPath n'a de sens que par rapport à un modèle de document dans lequel la notion de chemin est clairement définie. Ce modèle est la forme arborescente d'un document XML. La définition formelle du modèle de document défini par XPath se trouve dans [73]. Ce modèle est similaire à celui défini par DOM, que nous avons présenté dans le premier chapitre.

En particulier, les principales différences entre ces deux modèles, qui nous concernent dans le cadre de ce mémoire, sont les suivantes :

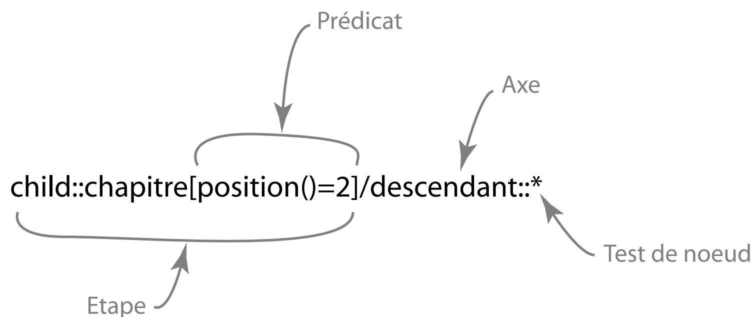


FIG. 3.1 – Exemple d'une expression XPath

– tout d'abord, contrairement à DOM, XPath définit la valeur d'un élément. Celle-ci est la concaténation des valeurs de tous les noeuds texte descendants de l'élément. Par exemple, la valeur XPath de <p>Bonjour</p> est la chaîne "Bonjour" et la valeur XPath de <p>Bonjour<em>Au revoir</em></p> est la chaîne "BonjourAu revoir" ;
– en XPath, l'élément qui contient un attribut est le parent de l'attribut, bien que l'attribut ne soit pas un enfant de l'élément [1]
– enfin, un noeud de texte en XPath contient toujours le maximum possible de caractères contigus dans la forme sérialisée. Autrement dit, aucun noeud de texte n'est adjacent à un autre noeud de texte.

### 3.2.3 Sélection de noeuds dans un arbre

#### Syntaxe et sémantique

La syntaxe et la sémantique de la norme XPath sont intégralement décrites dans [70]. La sémantique de XPath a de plus fait l'objet d'une formalisation dans [49]. Dans la section suivante, nous introduisons informellement la syntaxe et la sémantique d'une expression XPath.

Une expression XPath se compose d'une succession d'*étapes*, chacune d'elles sélectionnant un noeud ou un ensemble de noeuds, relativement au noeud ou à l'ensemble de noeuds sélectionné par le terme précédent figurant dans l'expression.

Nous introduisons la syntaxe d'une expression XPath à travers l'exemple figure 3.1.

Dans une expression XPath, les étapes successives sont séparées par le caractère "/" (slash). Chaque étape sélectionne un noeud courant dans l'arbre du document, noeud par rapport auquel l'étape suivante va s'appliquer pour ajouter une condition de sélection. une étape peut également sélectionner un ensemble de noeuds, auquel cas on parle d'ensemble courant.

Chaque étape est composée d'un *axe*, d'un *test de noeud*, et de zéro ou plusieurs *prédicats* :

– l'axe spécifie le type de relation arborescente entre le nœud contextuel et les noeuds à localiser ;

[1] C'est pour cette raison que dans ce mémoire, lorsqu'une structure XML est illustrée, les relations entre éléments et attributs sont représentées en pointillés, de manière à faire apparaître la différence avec une relation "parent-enfant" entre deux éléments.

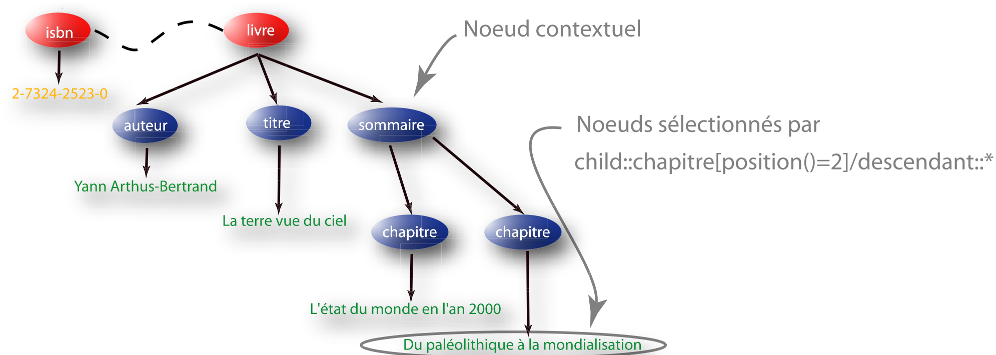


FIG. 3.2 – Exemple de sélection

– le test de noeud spécifie le type et le nom des noeuds obtenus par l'étape de localisation
– zéro ou plusieurs prédicats, qui contiennent des expressions booléennes pour filtrer l'ensemble des noeuds obtenus par l'étape de localisation

L'expression de la figure 3.1 sélectionne tous les noeuds (test de noeud "*") qui sont descendants (axe "descendant") de l'élément de nom "chapitre" et de position 2.

L'évaluation d'une expression XPath s'effectue relativement à un noeud donné dit *noeud contextuel*. La figure 3.2 illustre le noeud sélectionné par l'expression XPath dans le cas de l'arbre XML présenté dans le premier chapitre.

Une expression XPath qui débute par un "/" (slash) est absolue, c'est à dire qu'elle considère comme noeud contextuel la racine du document.

Enfin, XPath fournit également la possibilité d'exprimer des expressions numériques, booléennes ou qui manipulent des chaînes de caractères. Par exemple l'expression "count(chapitre) >= 3" teste si le nombre de chapitres est supérieur ou égal à 3, les chapitres étant des enfants du noeud contextuel implicite.

### Axes

XPath définit 13 axes permettant de naviguer dans un document. La figure 3.3 illustre la navigation dans un arbre XML qui est rendue possible par les axes de XPath.

### Syntaxe abrégée

XPath définit aussi une syntaxe abrégée pour les étapes les plus utilisées. En particulier, les deux principales abréviations définies sont les suivantes :
– "child : :" peut être omis
– "attribute : :" peut être remplacé par "@"

Une exemple d'expression employant la syntaxe abrégée est

$$para[5][@type = "warning"]$$

qui sélectionne le cinquième enfant "para" du noeud contextuel, si ce noeud "para" a un attribut "type" dont la valeur est "warning".

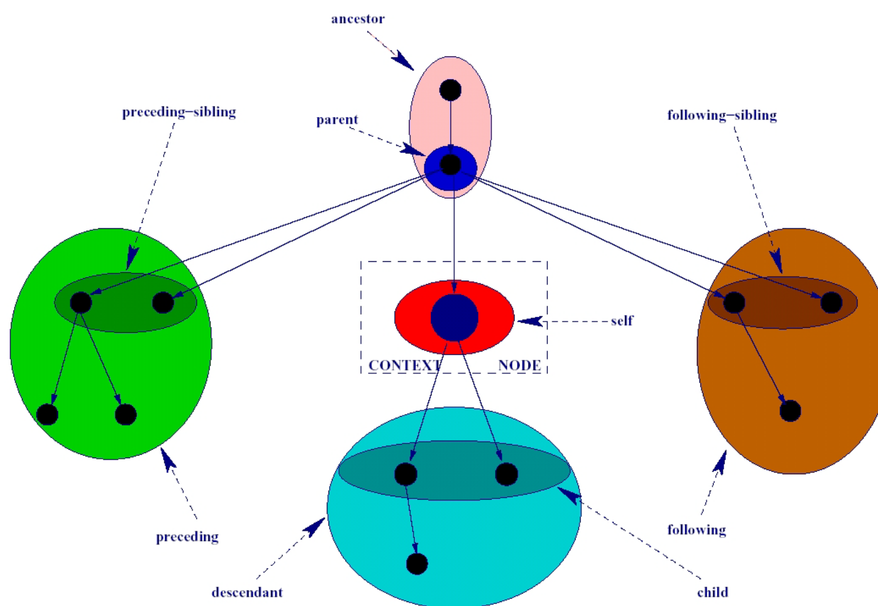


FIG. 3.3 – Navigation avec XPath 1.0

## 3.3 Principaux langages de transformation XML

Parmi les langages de transformations proposés pour la transformation de structures XML, nous avons retenu quatre langages (XSLT [74], XQuery [72], Circus [47], CDuce [5]) comme étant représentatifs de l'état actuel de la technique.

Les raisons de ce choix sont les suivantes : le langage est soutenu par une organisation importante donc ses chances de survie sont grandes (par exemple XSLT et XQuery du W3C), le langage est largement utilisé (par exemple XSLT), ou bien le langage a une approche unique et distincte de la norme XSLT (par exemple Circus et CDuce).

Tout d'abord, chaque langage ainsi que ses origines sont brièvement passés en revue.

### 3.3.1 XSLT

XSLT un langage de transformation déclaratif. Il est issu du domaine de la présentation (XSLFO). C'est un standard du W3C depuis novembre 1999. XSLT est le langage de transformation le plus répandu à l'heure actuelle. Sa particularité est de proposer un concept de template associé à celui de sélecteur. XSLT utilise le standard XPath du W3C pour réaliser la sélection.

### 3.3.2 XQuery

XQuery est un langage de requêtes sur documents XML. Il est issu du domaine des bases de données relationnelles. Il évolue petit à petit vers un langage de transformation. Conçu par le W3C, XQuery est appelé à devenir un standard. Sa particularité est

d'être bâti comme un sur-ensemble de XPath, en s'appuyant sur un modèle fonctionnel, et en étant typé.

### 3.3.3 Circus

Circus-DTE est un langage de programmation spécialisé dans la transformation de structures. Il est issu d'un noyau fonctionnel inspiré du lambda calcul typé. Circus a été conçu par Jean-Yves Vion-Dury et al, de Xerox. La particularité de Circus est de proposer un typage fort.

### 3.3.4 CDuce

CDuce est un langage fonctionnel pour la transformation de structures. Il est issu des travaux menés par Giuseppe Castagna et al du groupe langages de l'ENS de Paris et du groupe base de données du LRI à Orsay. CDuce a été élaboré sur la base de XDuce. XDuce (prononcé "transduce") est un langage fonctionnel pour la manipulation de structures XML, développé par Haruo Hosoya et al de l'université de Pennsylvanie. La particularité de CDuce est de mettre l'accent sur le contrôle statique de types.

## 3.4 Eléments de comparaison

Il serait possible de s'étendre longuement sur chaque langage de transformation. Nous ne cherchons pas ici à présenter chaque langage de transformation du point de vue de l'utilisateur. Pour cela nous avons retenu les références suivantes que nous recommandons : [52] pour XSLT, [53] pour XQuery, [54] pour Circus, et [55] pour CDuce.

Dans cette section, nous cherchons plutôt à restituer les éléments principaux qui permettent de se faire une idée de chaque langage au regard de la transformation incrémentale. A cet effet, nous avons retenu trois critères :

- la représentation qui est faite d'un document ;
- le modèle de transformation avec en particulier la constitution d'une règle de transformation ;
- et enfin les éventuelles propriétés que le langage peut présenter au regard de la transformation incrémentale.

### 3.4.1 Modèle de document considéré

#### Modélisation des structures XML

XSLT et XQuery emploient le modèle de données de XPath présenté précédemment.

Circus a pour objectif la transformation de structures en général. Son but est par conséquent de permettre une modélisation particulière pour chaque cas d'application. En particulier, plusieurs modèles des structures XML sont envisageables. Un premier modèle est décrit dans [47]. Bien que ne couvrant pas tous les aspects couverts par le modèle de XSLT et XQuery (par exemple les espaces de noms), il permet déjà d'écrire des transformations complexes comme la transformation de MathML Content vers MathML présentation. Circus a été étendu de façon à pouvoir modéliser les structures XML d'une manière plus élaborée, en capturant par exemple les contraintes d'ordre ou de nombre d'occurrence d'éléments. La modélisation des structures XML manipulées est ainsi laissée au programmeur.

CDuce propose des types atomiques avec des constructeurs de types standards (produit, enregistrements, fonctions). Les types expression régulière d'arbre présents dans XDuce sont réutilisés à travers les types récursifs et les combinaisons booléennes (union, intersection, différence). La modélisation des structures XML s'effectue avec l'algèbre de types proposée.

Au final, deux catégories de langages se dessinent : d'une part les langages spécifiquement dédiés à la transformation de structures XML, qui viennent avec un modèle intrinsèque et prédéfini des structures XML (XSLT, XQuery) ; et d'autre part les langages qui se veulent plus généraux dans la transformation de structures, en n'étant pas forcément spécifiques à XML, et de ce fait qui permettent au programmeur de définir le modèle de structures XML employé (Circus, CDuce).

#### Typage des documents

Un schéma de structure peut être vu comme un type de document, un document comme une instance d'un type, et une transformation comme une conversion entre types. Certains langages reposent sur cette représentation typée des documents XML. Ceci permet de faciliter l'expression de transformations applicables à une classe et non spécifiques à une instance particulière.

Circus modélise les arbres XML au moyen d'un type enregistrement récursif nommé XMLTree. Cet enregistrement comprend trois champs, le nom du noeud, la liste ordonnée des noeuds fils et un dictionnaire des paires (attribut, valeur) de l'élément.

CDuce type les arbres XML en utilisant ses types récursifs et ses expressions régulières.

Avec XQuery, le document XML source passe au travers du processus de validation, qui analyse le document, le valide par rapport à un schéma particulier, et le représente sous sa forme arborescente selon le modèle de données des requêtes [73]. Lorsque le document source n'a pas de modèle associé, il est validé par rapport à un schema par défaut, permissif, qui assigne des types génériques aux noeuds et valeurs atomiques. XQuery associe ainsi un type aux documents d'entrée de la transformation par l'intermédiaire de ce processus de validation. Il est possible d'invoquer le processus de validation sur le résultat d'une requête cible, et donc de typer le document cible.

Seul XSLT ne type pas les documents XML.XSLT a une approche plus syntaxique de la transformation. Le programmeur doit donc veiller à bien prendre en compte le modèle de document source lorsqu'il spécifie une transformation, de manière à ce qu'elle soit applicable à une classe. De plus, comme nous allons le voir, l'utilisateur doit aussi contrôler lui-même la validité du document résultat.

#### Garantie de validité du document cible

L'approche par typage permet d'établir des propriétés sur les transformations spécifiées, grâce au contrôle de type. En particulier, les langages qui emploient une représentation typée d'un document peuvent garantir un certain niveau de validité du document cible de la transformation par rapport à un modèle, en fonction de la puissance de leur contrôle de type.

En Circus, le seul fait de déclarer une variable de type de l'arbre cible instancie la variable à une valeur par défaut du type, avant même tout traitement. Par exemple, la déclaration suivante

```
 pam x : xtree_input, y : xtree_output.
(...)
```

instancie y avec une valeur par défaut du type xtree_output. Par la suite, le contrôle de type assure la validité incrémentalement à chaque traitement. Ainsi, on est donc assuré que le document cible de la transformation est une instance du type déclaré, i.e. est valide par rapport au modèle correspondant.

La spécification d'une transformation CDuce implique la définition des types des document XML source et cible. CDuce garantit donc la validité de l'arbre cible de la transformation par rapport au type spécifié.

En XQuery, la validation par rapport à un schéma peut être invoquée explicitement sur le résultat d'une requête ou sur des expressions intermédiaires obtenues à l'intérieur d'une requête.

### 3.4.2 Modèle de transformation

**Partie gauche : sélection dans l'arbre source**

XSLT réutilise le standard XPath comme un sous-langage pour permettre la sélection de noeuds et l'extraction de valeurs de l'arbre source.

XQuery réutilise lui aussi XPath. La différence est que XQuery est construit comme un sur-ensemble de XPath : une expression XPath est une requête XQuery valide. La philosophie derrière XQuery est d'étendre la norme XPath pour en faire un langage de requêtes sur documents XML.

Circus emploie une technique spécifique pour permettre l'exploitation de fragments dans l'arbre source. Il propose un filtrage par matching de la forme : "si *expression* correspond à *filtre* alors effectuer *action*".

CDuce emploie lui-aussi une technique qui lui est propre pour permettre l'exploitation de fragments dans l'arbre source. CDuce propose à cet effet un mécanisme de pattern-matching sur des expressions régulières d'arbres (directement issu des travaux sur XDuce).

**Partie droite : génération dans l'arbre cible**

Avec XSLT, une transformation peut se décomposer en un ensemble de templates (dont le contenu décrit un fragment d'arbre cible) associés à des sélecteurs dans l'arbre source. Ainsi la partie droite d'une règle est le contenu d'une template. Celui-ci peut être constitué d'une description en dur des noeuds à générer dans l'arbre cible, de références XPath vers des valeurs de l'arbre source et d'instructions pour contrôler la transformation.

Avec XQuery, une requête peut se décomposer en plusieurs fonctions. XQuery fournit des instructions de contrôle (boucles, etc.) de manière à pouvoir organiser au sein de l'arbre cible les fragments issus d'une requête sur l'arbre source. La partie droite d'une requête XQuery est constituée d'une description en dur des noeuds à générer dans l'arbre cible, de résultats de requêtes effectuées sur l'arbre source, et d'instructions pour contrôler la transformation.

En Circus une règle de transformation peut notamment s'exprimer sous la forme d'une polymorphic abstract machine (PAM) qui prend en paramètre un arbre d'entrée et qui produit un arbre en sortie. Le principe est de modifier la valeur par défaut de l'arbre de sortie, conformément à son type. Circus propose également un concept de règle de la forme $a\#b \Rightarrow c$. Informellement, la sémantique d'une telle règle est d'évaluer l'expression $a$ et de lui appliquer le filtre $b$. Si cette opération aboutit avec succès, $c$ est évaluée. On pourrait donc considérer qu'une expression $c$, qui génère une structure

de l'arbre cible, constitue la partie droite d'une règle Circus. Notons que la génération de la structure du document cible peut s'effectuer à l'aide de toute la puissance d'un langage de programmation générique.

Avec CDuce, une règle de transformation peut s'exprimer par une fonction. D'une manière similaire à l'approche développée dans Circus, la génération s'effectue à l'aide de la puissance d'un langage de programmation générique. Notons cependant que CDuce est basé sur un noyau fonctionnel restreint. Le principe est d'utiliser le jeu de constructeurs de types fourni pour générer la structure de l'arbre cible.

#### Modèle d'exécution

XSLT est un langage spécifique avec un modèle d'exécution spécialisé et original. La norme XSLT définit ce qu'est un *processeur de transformation XSLT*. En particulier, le *contexte d'exécution* d'une instruction de transformation et le *processus d'instanciation* des règles sont définis, ainsi que les stratégies de résolution des conflits potentiels entre règles applicables.

Les autres langages ont un modèle d'exécution qui s'apparente plus à celui d'un langage de programmation classique. Circus peut par exemple être vu comme un langage généraliste qui a évolué vers langage de transformation. D'une manière un peu similaire, CDuce repose sur un noyau fonctionnel générique. Il évolue en particulier pour supporter des primitives "XML-friendly"[5].

### 3.4.3 Transformation incrémentale

#### Support natif

Le processeur de transformations XSLT ne prend pas nativement en compte le cas de la transformation incrémentale [46]. Le W3C a en particulier pour objectif de faire évoluer XSLT pour rendre son modèle d'exécution plus adapté à la transformation de documents de très grande taille. Ceci figure comme objectif pour la version 2.0 de XSLT [75].

A l'heure actuelle, aucune solution particulière n'est proposée par les autres langages étudiés afin d'optimiser la transformation de documents de très grande taille, en les transformant de manière incrémentale.

#### Support via programmation ad-hoc

Notons que certains langages, notamment du fait de leur paradigme fonctionnel, permettent d'effectuer une transformation de manière incrémentale en la programmant de façon ad-hoc. L'idée est d'éclater une transformation en plusieurs fonctions, sans effet de bord (i.e. sans variable globale affectée localement dans une fonction). Chaque fonction serait définie de manière à traiter les conséquences de certaines modifications particulières de l'arbre source. Ces modifications, déterminées à l'avance, conditionnent l'écriture des fonctions. A l'exécution, lorsqu'une modification attendue de l'arbre source survient, il suffit d'appliquer la fonction correspondante. Puisqu'il n'y a pas d'effet de bord, la restauration du contexte d'exécution est facilitée : aucune valuation de variables n'est à restaurer. Cette approche suppose que le programmeur envisage a priori les modifications particulières du document source qu'il sera possible de traiter incrémentalement. Par conséquent, un nombre limité de modifications pourront être traitées de manière incrémentale. De plus, cette approche a un coût considérable

| Caractéristiques | XSLT | XQuery | Circus | CDuce |
|---|---|---|---|---|
| **Modèle de document** | | | | |
| Langage spécifique à XML | oui | oui | non | non |
| Typage d'un document XML | non | oui | oui | oui |
| Garantie de validité du document cible | non | oui | oui | oui |
| **Modèle de transformation** | | | | |
| Partie gauche des règles | Sélection avec XPath | Sélection avec XPath | Matching d'expressions de filtrage | Matching d'expressions régulières d'arbres |
| Partie droite des règles | Contenu de template | Contenu d'un return | Opérateurs génériques | Opérateurs génériques |
| **Transformation incrémentale** | | | | |
| Support natif | Non | Non | Non | Non |
| Via programmation ad-hoc | Non | Non | Oui | Oui |

FIG. 3.4 – Tableau récapitulatif

en terme de programmation spécifique, et par conséquent en terme de flexibilité, de maintenance et de passage à l'échelle.

## 3.5 Synthèse

Le tableau de la figure 3.4 récapitule les principales caractéristiques de chaque langage.

Tout d'abord, dans les langages de transformation considérés, la partie gauche des règles de transformation qui concerne la sélection et l'extraction de données dans l'arbre source est isolée et très bien caractérisée. Ceci notamment avec le standard XPath du W3C que nous avons présenté en 3.2. En revanche, la partie droite des règles de transformation qui concerne la génération de la structure du document cible, n'est pas encore aussi clairement identifiée et isolée. De ce point de vue, les règles de transformation ne sont pas symétriques.

D'autre part, nous pouvons constater que XSLT, bien qu'il soit le langage de transformation le plus répandu et utilisé à l'heure actuelle, souffre de plusieurs défauts. Son défaut principal est sans aucun doute l'absence de garantie de la validité du document cible. Avec XSLT, il n'est pas possible d'acquérir la certitude qu'une feuille de transformation produit des documents valides. Ceci ne peut qu'être subodoré à la suite de constatations empiriques.

Enfin et surtout, aucun langage ne semble avoir été spécifiquement étudié dans l'optique de la transformation incrémentale. Au contraire, chaque langage a été conçu dans une optique "batch" qui part du principe que la totalité du document source est transformée d'un coup.

## 3.6 Conclusion

Après avoir décelé l'importance du rôle que joue la transformation de structures XML, nous avons cerné quelques limites des langages de transformation actuels. Au-

jourd'hui où la transformation de documents de très grande taille est d'actualité, de nombreux travaux restent encore à faire. En particulier, l'approche qui consiste à optimiser les transformations en les effectuant de manière incrémentale nous semble une piste prometteuse.

Nous pensons qu'il serait intéressant de comprendre les propriétés qui permettraient de concevoir dès le départ un langage dans l'optique de la transformation incrémentale.

Le chapitre suivant tente d'analyser les techniques développées avec succès autour du traitement incrémental.

# Chapitre 4

# Traitement incrémental

## Résumé

Le traitement incrémental est un sujet abordé à travers de nombreux et divers thèmes de recherche. Les premiers travaux autour de l'incrémentalité remontent à la méthodologie de la programmation, il y a plusieurs décennies. Depuis, l'incrémentalité a principalement été utilisée comme technique d'optimisation. Des algorithmes incrémentaux ont été développés dans des domaines aussi variés que les environnements de programmation interactifs, les résolveurs de contraintes, la compilation de programmes, la transformation de documents structurés, etc. Ce chapitre est une synthèse des principales méthodes générales et spécifiques pour le traitement incrémental. Il tente de dégager les idées générales qui ont été développées avec succès autour de l'incrémentalité. Il donne également quelques références dans l'étendue des travaux existants.

## Contenu

**4.1 Introduction** . . . . . . . . . . . . . . . . . . . . . . . . . . . . **32**
4.1.1 Traitement incrémental . . . . . . . . . . . . . . . . . . . 32
4.1.2 Fondements . . . . . . . . . . . . . . . . . . . . . . . . . 32
4.1.3 But . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
**4.2 Traitement incrémental : définitions** . . . . . . . . . . . . . . . **33**
4.2.1 Caractérisation d'un algorithme incrémental . . . . . . . . 33
4.2.2 Rendre un traitement incrémental . . . . . . . . . . . . . . 33
**4.3 Catégorisation des travaux sur l'incrémentalité** . . . . . . . . . **33**
**4.4 Méthodes générales pour le traitement incrémental** . . . . . . . **34**
4.4.1 Critères de classification . . . . . . . . . . . . . . . . . . 34
4.4.2 Principales méthodes générales . . . . . . . . . . . . . . . 35
**4.5 Méthodes spécifiques pour le traitement incrémental des documents** . . . . . . . . . . . . . . . . . . . . . . . . . . . . **37**
4.5.1 Traduction incrémentale . . . . . . . . . . . . . . . . . . . 37
4.5.2 Traitement XML paresseux (Lazy XML processing) . . . . 38
4.5.3 Transformation incrémentale de documents XML : incXSLT 38
**4.6 Synthèse** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **39**
4.6.1 Techniques génériques . . . . . . . . . . . . . . . . . . . 39
4.6.2 Techniques spécifiques . . . . . . . . . . . . . . . . . . . 40

4.6.3 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . 40

# 4.1 Introduction

## 4.1.1 Traitement incrémental

Dans la langue française, le terme « incrémentiel »fait référence à ce « qui permet un traitement séquentiel immédiat des informations ». Dans de nombreux contextes en informatique, les modifications des données d'entrée doivent être traitées rapidement afin d'avoir un effet immédiat sur le résultat en sortie. Puisque de légers changements sur les données d'entrée d'un traitement sont souvent seulement à l'origine de légers changements dans le résultat, l'objectif est de calculer le nouveau résultat *incrémentalement*, c'est-à-dire en mettant à jour les parties adéquates de l'ancien résultat, plutôt que de recalculer complètement un nouveau résultat à partir de zéro. A cet effet, on cherche à utiliser un algorithme dit *incrémental*.

## 4.1.2 Fondements

Les premiers travaux autour du traitement incrémental remontent sans doute à la méthodologie de la programmation [8], vers la fin des années 70. Le maintien de manière incrémentale d'invariants de boucle permettait alors de dégager des propriétés d'efficacité sur les programmes.

## 4.1.3 But

**Une technique d'optimisation** Les méthodes incrémentales constituent une classe importante de techniques d'optimisation. Celles-ci permettent l'implantation efficace de programmes en contournant la lenteur relative des ordinateurs.

**Discussion** Les améliorations de performances qui peuvent être obtenues en appliquant des techniques incrémentales ont un coût. Le stockage des résultats de sous-calculs nécessite de la mémoire supplémentaire. De plus, le développement et l'implantation d'algorithmes incrémentaux est nettement plus complexe que celle de leurs alternatives non incrémentales. Or, d'une manière générale, la puissance des ordinateurs ne cesse de croître constamment grâce à des améliorations matérielles. La question de savoir si l'utilisation de techniques incrémentales est justifiée peut donc se poser. En effet, lorsque la puissance des ordinateurs aura suffisamment augmenté, certaines applications pourraient ne plus requérir d'optimisation. Cependant, indépendamment de la puissance des ordinateurs, il existera toujours des applications dont les performances sont insuffisantes. Ceci en partie car l'augmentation de la puissance des ordinateurs inspire constamment le développement d'applications de plus en plus intensives du point de vue calculatoire. L'augmentation de la puissance des processeurs n'enlève pas le besoin de méthodes incrémentales.

## 4.2 Traitement incrémental : définitions

### 4.2.1 Caractérisation d'un algorithme incrémental

Le problème du traitement incrémental pourrait être exprimé comme suit : le but est de calculer une fonction $f$ sur la donnée d'entrée $x$, où $x$ est une structure de données bien identifiée (par exemple un arbre dans le cas de la transformation de documents structurés) et de maintenir le résultat $f(x)$ à jour lorsque la donnée d'entrée subit des modifications. Ainsi, un algorithme incrémental pour le calcul de $f$ prend comme données d'entrée :

- l'entrée du traitement « batch »$x$ ;
- le résultat du traitement « batch »$f(x)$ ;
- éventuellement des informations auxiliaires résultant de l'analyse statique ou dynamique du traitement « batch », de l'entrée $x$, ou du résultat $f(x)$
- les modifications $\Delta x$ de l'entrée

L'algorithme incrémental calcule le nouveau résultat $f(x \oplus \Delta x)$ où $x \oplus \Delta x$ dénote l'entrée modifiée, et met à jour adéquatement les éventuelles informations auxiliaires.

### 4.2.2 Rendre un traitement incrémental

Une définition formelle du problème de « rendre un traitement incrémental »est donnée par [26] : Étant donnés un programme f et une opération $\oplus$, un programme $f'$ est appelé une *version incrémentale* de $f$ par rapport à $\oplus$ si $f'$ calcule $f(x \oplus \Delta x)$ de façon efficace en utilisant $f(x)$.

**Exemple :** supposons que f soit un compilateur C, $x$ un programme C et que l'opération $\oplus$ effectue un changement du programme C. $f'$ est alors un compilateur C incrémental : il compile en effet un nouveau programme en mettant à jour l'ancien programme plutôt que de le compiler dans sa totalité.

Souvent, des informations auxiliaires (autres que le résultat $f(x)$) doivent être maintenues et sont nécessaires pour un calcul incrémental efficace de $f(x \oplus \Delta x)$. Un programme qui calcule de telles informations est appelé une *version étendue* de $f$. Ainsi, le but de calculer efficacement consiste à construire une version étendue d'un programme $f$, puis de dériver une version incrémentale de la version étendue pour une opération $\oplus$.

## 4.3 Catégorisation des travaux sur l'incrémentalité

De nombreux travaux concernant l'incrémentalité ont été effectués. Un aperçu des différents travaux existants est donné par G. Ramalingam et T. Reps dans une bibliographie classifiée sur le traitement incrémental [37]. D'autres travaux encore sont venus s'ajouter depuis. Les algorithmes incrémentaux sont notamment utilisés pour les environnements de programmation interactifs, pour la compilation de programmes, dans les résolveurs de contraintes, pour la transformation de documents structurés, etc.

**Il est possible de diviser tous les travaux existants en deux catégories :**

**Les méthodes générales** Ce sont les méthodes pour le traitement incrémental conçues pour n'importe quel programme et n'importe quelle modification des données d'entrée. Ces techniques sont les suivantes : *incremental attribute evaluation framework* [40, 41], *function caching* [1, 35], *incremental lambda reduction* [10], *program abstraction* [16], *partial evaluation* [43], *change detailing network* [51], *strength reduction* [4] (dans le cadre de l'optimisation des compilateurs), *finite differencing* [33] (dans le cadre de la programmation transformationnelle : « transformational programming »qui consiste à dériver un programme d'une spécification abstraite), *incremental rewriting* [11, 44] (dans le cadre des systèmes de réécriture), *adaptive functional programming* [2], et enfin *strengthening invariants* [26].

**Les méthodes spécifiques** Ce sont les algorithmes incrémentaux conçus pour des problèmes spécifiques. Ce sont des algorithmes particuliers, développés « à la main »pour résoudre des problèmes particuliers et des modifications particulières des données d'entrée. Dans notre domaine, des exemples de tels algorithmes sont notamment ceux conçus autour de l'édition de documents [22], et pour la transformation de documents structurés [45]. Ces algorithmes pourraient être nommés ad-hoc dans la mesure où ils résolvent des problèmes particuliers d'incrémentalité.

Dans la suite, nous étudierons tout d'abord ces méthodes générales, en tentant de donner des critères pour être en mesure de les classer. Enfin, nous aborderons les méthodes spécifiques développées dans le domaine qui nous concerne : celui du traitement des documents et plus particulièrement de la transformation de structure XML.

## 4.4 Méthodes générales pour le traitement incrémental

### 4.4.1 Critères de classification

Cette section présente deux critères utilisables pour classer les méthodes générales.

**Gestion des informations auxiliaires** Les méthodes générales diffèrent tout d'abord dans leur manière de stocker et de maintenir les informations auxiliaires. Au moment du stockage des résultats d'un traitement en vue du traitement incrémental, les informations doivent être stockées de manière structurée pour permettre de retrouver rapidement l'information nécessaire lors des traitements suivants. De plus, la masse d'informations stockée est limitée. De nombreuses techniques, mais pas toutes, ne retiennent que les informations relatives aux dernier traitement.

**Selective recomputation versus finite differencing** Une distinction importante peut être établie entre les méthodes reposant sur « selective recomputation »et celles basées sur « finite differencing »(ou « differencial updating »). Pour expliquer la différence entre « selective recomputation »et « finite differencing », considérons un programme $f(x)$ traitant incrémentalement une donnée d'entrée $x$. Lorsque $f(x)$ est calculé, des résultats intermédiaires sont stockés. Supposons alors que le résultat du même traitement est requis pour une donnée d'entrée légèrement modifiée $x \oplus \Delta x$. Dans les méthodes basées sur « selective recomputation », les résultats intermédiaires du calcul de $f(x)$ sont utilisés pour éviter les calculs redondants avec $f(x \oplus \Delta x)$. Dans les approches basées sur « finite differencing », l'existence d'une fonction de différence $f_\Delta$ est prise comme hypothèse. $f_\Delta$ est utilisée pour calculer le résultat pour $x \oplus \Delta x$ par mise à jour du résultat précédent : $f(x \oplus \Delta x) = f(x) \otimes f_\Delta(\Delta x)$.

### 4.4.2 Principales méthodes générales

#### Différence finie (Finite differencing)

Le terme « finite differencing »a été introduit par Paige et Koenig dans [33]. Leurs idées ont été appliquées par Paige dans son papier sur la programmation avec invariants [34]. Dans la programmation avec invariants, les programmes sont transformés en programmes incrémentaux en remplaçant les fonctions par des fonctions correspondantes incrémentales prises dans une librairie. Pour chaque fonction incrémentale $f$, un invariant $E = f(x_1, ..., x_n)$ est maintenu durant l'exécution du programme. Lorsque la valeur de l'un des arguments $xi$ change, la valeur de E est mise à jour au moyen d'une fonction de différence $f_{\Delta xi}$. Seule la dernière valeur de $E$ est sauvegardée, ce qui limite la taille des informations stockées.

#### INC

INC est un langage conçu pour le traitement incrémental, par Yellin et Strom [51]. Il est basé sur la méthode « finite differencing ». Le langage INC comporte un certain nombres de fonctions prédéfinies pour lesquelles une fonction de différence existe. Un programme INC est un réseau de fonctions prédéfinies dans lequel le résultat d'une des fonctions sert de paramètre d'entrée pour la suivante. Lorsque un programme INC est exécuté avec un certain paramètre d'entrée $x$, le paramètre d'entrée et le résultat de chaque fonction sont sauvegardés. Lors de la prochaine exécution du programme avec la nouvelle donnée d'entrée $x + \Delta x$, la première fonction $f$ du programme utilise $f_\Delta$ pour calculer le nouveau résultat et la différence avec le précédent résultat. La différence de résultat est passée à la fonction suivante. Seule l'information du dernier calcul est sauvegardée.

#### Mise en cache (Functiong caching)

Introduite par Michie dans [29], « function caching »est une technique pour stocker les résultats des applications de fonctions en vue de les réutiliser lorsque une fonction est appelée pour la seconde fois avec les mêmes paramètres. Cette technique est souvent utilisée pour améliorer la performance des fonctions récursives. Pugh et Teitelbaum [35] ont décrit comment utiliser la technique « function caching »pour l'évaluation incrémentale avec des structures de données. Dans le but d'obtenir des algorithmes incrémentaux efficaces, les structures de données doivent être conçues spécifiquement de manière à ce que les traitements sur des instances similaires du type de donnée aient en commun de nombreux sous-traitements. « function caching »est conceptuellement simple mais difficile à implanter efficacement. Une des difficultés est de trouver des manières de rechercher rapidement dans le cache. Lorsque les fonctions sont appelées avec de nombreux paramètres, comparer les paramètres de l'appel de la fonction en cours avec les informations présentes dans le cache devient très coûteux. Une autre difficulté est de conserver la taille du cache raisonnablement limitée. Il est nécessaire d'implanter des techniques pour purger le cache, qui conservent les résultats ayant une forte probabilité d'être réutilisés dans des traitements ultérieurs, et suppriment les autres informations.

### Evaluation partielle (Partial evaluation)

L'évaluation partielle est une technique classique de transformation de programme pour spécialiser une fonction étant donné une partie connue de ses données d'entrée. Le résultat de cette technique s'appelle une *fonction résiduelle*, et a la propriété de donner les résultats souhaités lorsqu'elle est appliquée aux parties restantes des données d'entrée.

Une définition formelle de l'évaluation partielle au moyen des projections est donnée par Sundaresh dans [43]. Partant de cette définition, Sundaresh montre comment l'évaluation partielle peut être utilisée dans le cadre du traitement incrémental.

Le principe est que le domaine d'entrée d'une fonction peut être partitionné en utilisant un ensemble de projections. Une fois le domaine d'entrée partitionné, les fonctions résiduelles correspondant au partitionnement, c'est-à-dire à chaque projection, sont calculées. Ceci revient à stocker les portions du calcul qui ne dépendent que de chacune des parties du domaine d'entrée. Ainsi, lorsqu'une partie du domaine d'entrée change, il suffit d'appliquer à la nouvelle partie la fonction résiduelle correspondante. La difficulté principale est de trouver un bon partitionnement du domaine d'entrée pour pouvoir combiner correctement et efficacement les fonctions résiduelles [43].

### Réécriture incrémentale

Field a proposé un algorithme pour permettre la réécriture incrémentale de termes [11]. Dans la technique, les sous-termes remplaçables sont repérés avant qu'un terme T ne soit réécrit. Lorsque le terme est réécrit, les termes contextuels dans T des sous-termes repérés sont réécrits simultanément. Le résultat de la réécriture incrémentale est la forme normale du terme ainsi que les formes normales de tous les termes contextuels. Elles sont groupées ensemble en un terme, permettant de récupérer facilement chaque forme normale individuellement. Lorsqu'un second terme T' s'apprête à être réécrit, et que T' diffère de T d'un ou plusieurs sous-termes repérés, la forme normale du terme contextuel commun est utilisée comme point de départ pour la réduction incrémentale de T'. Le terme résultat est alors ajouté au stockage existant.

Cette technique a un peu plus tard été reprise par E. Van Der Meulen dans sa thèse [44]. La principale limite de cette approche est qu'elle s'applique à une classe restreinte de spécifications algébriques.

### Renforcement d'invariants (Strenghtening invariants for incrementalization)

Récemment, des efforts ont été menés pour tenter de dériver des programmes incrémentaux automatiquement (ou semi-automatiquement) à partir de programmes non incrémentaux écrits dans des langages de programmation classiques. Cette approche ambitieuse est nommée *incrémentalisation* [26]. Elle découpe le coeur du problème en trois sous-problèmes :

1. l'exploitation du résultat précédent $f(x)$ ;
2. le stockage, le maintien et l'exploitation de valeurs intermédiaires du calcul $(fx)$ ;
3. et enfin la découverte et l'exploitation d'informations auxiliaires, c'est-à-dire des valeurs non calculées par $f(x)$.

Puisque les informations auxiliaires ne sont pas calculées par l'algorithme original qui calcule $f(x)$, le fait de les ajouter renforce les invariants vérifiés lors du calcul itératif qui utilise $f(x)$, d'où le nom de cette méthode générale.

L'idée basique en réponse au premier problème est d'identifier dans le calcul de $f(x \oplus y)$ les sous-calculs qui sont aussi effectuées dans le calcul de $f(x)$ et dont les résultats peuvent être récupérés du cache $r$ de $f(x)$. Le calcul de $f(x + y)$ est symboliquement transformé pour éviter d'effectuer à nouveau ces sous-calculs en les remplaçant par les résultats correspondants. [25]

En réponse au second problème, une méthode a été donnée dans [23] pour transformer statiquement les programmes pour stocker dans un cache tous les résultats intermédiaires utiles pour le traitement incrémental. L'idée basique est (I) tout d'abord d'étendre le programme $f$ en un programme $g$ qui renvoie tous les résultats intermédiaires, (II) incrémentaliser le programme $g$ pour $\oplus$ pour obtenir une version incrémentale $h$ de $g$ en utilisant la méthode citéé plus haut pour le premier problème, et (III) d'analyser les dépendances dans $h$ pour élaguer le programme étendu $g$ en un programme $i$ qui retourne seulement les résultats intermédiaires utiles, et enfin élaguer le programme $h$ pour obtenir un programme $f'$ qui maintient incrémentalement seulement les résultats intermédiaires nécessaires.

Concernant le troisème problème, une proposition pour trouver des informations auxiliaires est donnée dans [26].

Les deux idées clefs de cette proposition sont les suivantes :

- tout d'abord, considérer comme information auxiliaire éligible pour $f$ tous les résultats intermédiaires d'une version incrémentale de $f$ qui ne dépend que de $x$. Une telle version peut être obtenue grâce aux techniques citées précédemment.
- Etendre $f$ avec toutes les informations auxiliaires éligibles, puis appliquer des techniques toujours issues des méthodes citées précédemment pour obtenir une version étendue et une version incrémentale qui ensemble calculent, exploitent et maintiennent seulement les résultats intermédiaires utiles et seulement les informations auxiliaires utiles.

La principale limite de cette technique est liée à son objectif : la méthode s'avère lourde à employer puisqu'elle tente de rendre incrémental un algorithme qui ne l'est pas.

## 4.5 Méthodes spécifiques pour le traitement incrémental des documents

### 4.5.1 Traduction incrémentale

Avant l'arrivée de XML, un algorithme incrémental a été conçu autour de l'édition documentaire [22]. Il s'agit d'un algorithme incrémental de traduction d'arbre. Habituellement, lorsqu'un utilisateur saisit une chaîne de caractères, un parsing est effectué pour reconnaître cette chaîne. Ce parsing peut d'ailleurs lui même être incrémental. Les travaux de Lindén concernent la traduction incrémentale, c'est à dire, étant donné une grammaire d'entrée et une de sortie, traduire la chaîne saisie par l'utilisateur contrainte par une grammaire d'entrée en une chaîne contrainte par une autre grammaire de sortie. Cette traduction est effectuée de manière incrémentale et réagit donc à chaque modification de la chaîne d'entrée. On pourrait considérer qu'une variante simplifiée de cet algorithme est présent de nos jours sur les téléphones cellulaires. En effet, certains téléphones permettent de saisir rapidement du texte à l'aide d'un nombre de touches inférieur au cardinal de l'alphabet réellement permis. Un contrôle incrémental de la saisie est effectué à l'aide d'un dictionnaire. L'algorithme de Lindén prend en en entrée un arbre modifié, une liste des changements, une grammaire d'entrée et une grammaire

de sortie et l'ancien arbre de sortie. L'algorithme calcule le nouvel arbre cible par mise à jour de l'ancien. La principale limite de cette approche réside dans le fait qu'elle est difficilement extensible au cadre de la transformation de documents XML car les grammaires considérées sont trop restrictives.

### 4.5.2 Traitement XML paresseux (Lazy XML processing)

Des travaux sont actuellement en cours sur le traitement paresseux de documents XML [31]. Le concept de "laziness" est traditionnellement relié à celui de l'incrémentalité. Un algorithme *paresseux (lazy)* délaye autant que possible le calcul. Il y a plusieurs raisons pour ne pas effectuer un calcul complètement : certains sous-calculs peuvent ne pas être pertinents, non nécessaires immédiatement, ou bien les résultats non visibles, ou encore redondants. Une différence entre incrémentalité et "laziness" est qu'un algorithme incrémental met à jour la solution complète après une modification d'un exemple de problème alors que l'algorithme paresseux résout différentes parties de l'exemple de problème, et on doit lui spécifier lesquelles. Cependant, l'approche commune entre laziness et incrémentalité est le traitement statique utilisé en prévision d'une exécution paresseuse ou incrémentale. Les travaux en cours sur le traitement paresseux des structures XML [31] commencent par une étude fine du parsing qui est le premier traitement qui doit nécessairement être rendu paresseux avant d'aborder des problèmes comme la transformation. Une des difficultés pointée par [31] est que DOM [63] rend la tache du parsing paresseux difficile car son interface (et d'une manière générale aussi les feuilles XSLT, etc.) impliquent une connaissance totale de l'arbre source même si seules des parties de celui-ci sont effectivement nécessaires.

### 4.5.3 Transformation incrémentale de documents XML : incXSLT

#### Introduction

incXSLT est un processeur XSLT [74] incrémental conçu par Lionel Villard et Nabil Layaïda [45]. La conception de ce processeur incrémental repose sur la modélisation du contexte d'exécution de XSLT et l'étude fine du langage de sélection XPath [70].

Suite à une modification du document source de la transformation, ce processeur incrémental est en mesure de :

- détecter quelles sont les expressions XPath qui ont besoin d'être réévaluées, grâce à un concept de *règle de réévaluation*
- reconsidérer efficacement les instanciations, c'est-à-dire le sous ensemble des règles de transformation qu'il est nécessaire de réappliquer ; ceci notamment en élaborant un *graphe de dépendance*
- maintenir la cohérence du contexte d'exécution propre à XSLT (il faut contrôler les répercussions liées à une modification indirecte du contexte d'exécution)
- enfin, d'exécuter les instructions nécessaires de façon incrémentale

#### Principe de la détection des expressions à réévaluer

Une analyse de chaque expression XPath de la feuille de transformation XSLT est effectuée afin de calculer les modifications sur le document source qui peuvent entraîner une modification du résultat de l'expression. Pour ne pas énumérer l'ensemble de ces modifications, un pattern permettant de les caractériser est calculé. A ce pattern est associée une liste d'instructions à réexécuter, correspondant à l'analyse de chaque

expression XPath de la feuille de transformation. Ce couple est nommé *règle de réévaluation*.

#### Construction des règles de réévaluation

La liste des règles de réévaluation est calculée en considérant toutes les expressions de la feuille de transformation.

Le calcul des règles de réévaluation peut être formalisé comme suit : soit $A$ l'ensemble des noeuds sélectionnés par l'expression $e$ de l'instruction $i$ avant la modification du document source. Soit $B$ l'ensemble des noeuds sélectionnés par la même expression après la modification du document source. la règle de réévaluation $(p, i)$ est ajoutée si $A \neq B$, avec $p$ le pattern modélisant la modification du document source.

Pour identifier la ou les conditions pour que $A$ et $B$ soient différents, chaque objet syntaxique d'une expression XPath est considéré : des règles de réévaluation sont tout d'abord données pour une étape sans prédicat, avec prédicat, et enfin pour l'enchainement d'étapes. [45]

#### Exécution incrémentale

Une fois que la liste des règles de réévaluation est établie, les instructions qui nécessitent d'être réexécutées sont connues. Pour cela une correspondance est établie entre les patterns des règles et le noeud qui est en train d'être modifié. Seules les instructions associées aux patterns correspondants nécessitent d'être réexécutées. Deux modèles d'exécution ont été envisagés : le premier est basé sur le parcours de l'arbre d'exécution ; le second modèle repose sur l'exécution directe de chaque instruction de la liste des patterns.

Le principal inconvénient du premier modèle est que le parcours de l'arbre d'exécution est proportionnel à la taille de l'arbre cible. Par conséquent, pour de gros documents, le temps de mise à jour peut être important [45]. Le second modèle nécessite des calculs et surtout du stockage supplémentaire pour pouvoir restaurer le contexte et s'avère finalement plus coûteux en taille mémoire que le premier. Il semble néanmoins adapté au cas où peu d'instructions nécessitent d'être réévaluées [45].

#### Conclusion

incXSLT permet d'exécuter des transformations exprimées en XSLT de manière incrémentale dès qu'une première exécution a été effectuée. Une limite de cette approche est le fait que les règles de réévaluation sont relâchées du fait de l'existence de paramètres dynamiques qu'il est impossible de connaître statiquement.

## 4.6 Synthèse

### 4.6.1 Techniques génériques

De nombreux travaux ont été effectués autour de la conception et l'utilisation d'algorithmes incrémentaux, dans des domaines variés. Aussi complexes qu'elles soient, toutes les techniques proposées sont essentiellement articulées autour de quatre techniques de base qui sont les suivantes :

1. exploitation du résultat précédent : le calcul de $f'(x \oplus \Delta x)$ s'effectue en utilisant le résultat du calcul précédent f(x). Pour cela il est nécessaire de stocker le résultat $f(x)$ dans un cache ;
2. exploitation de résultats intermédiaires : pour calculer $f(x)$ plusieurs calculs intermédiaires sont nécessaires. Ces calculs peuvent aussi être stockés et exploités pour calculer $f'(x \oplus \Delta x)$ ;
3. découverte, calcul, maintien et exploitation d'informations auxiliaires à propos de $x$, c'est-à-dire des valeurs non calculées par $f(x)$ en temps normal (dans le cas non incrémental) ;
4. évaluation partielle : cette technique s'appuie sur une partie connue et statique de la donnée en entrée. Dans certains cas, la fonction peut être simplifiée en évaluant de façon statique cette partie.

Ces techniques génériques sont employées afin de dériver un programme incrémental à partir d'un programme non incrémental.

### 4.6.2 Techniques spécifiques

Les seuls travaux existants à l'heure actuelle concernant la transformation incrémentale de documents XML sont ceux de Lionel Villard et Nabil Layaïda [45] sur incXSLT. Cette avancée repose notamment sur deux idées, plus générales que XSLT :

– analyser statiquement la transformation (et plus précisément dans le cadre de XSLT, les expressions XPath [70] de sélection dans l'arbre source). L'analyse statique a pour but d'associer un motif caractérisant des modifications du document source aux instructions de transformation qu'il est nécessaire de réexécuter
– stocker une partie de l'historique d'exécution de la transformation, de manière à pouvoir placer les informations auxiliaires qui permettent de modifier le document cible adéquatement

### 4.6.3 Conclusion

Nous avons tenté d'étudier les travaux existants autour du traitement incrémental. Enormément de travaux ont été effectués autour du traitement incrémental en général, dans des domaines très variés. Nous avons analysé les principales méthodes générales, que nous pensons être à la base d'un grand nombre de ces travaux.

Les travaux autour du traitement incrémental de documents XML sont encore en nombre très limité. Nous avons tenté de faire apparaître les idées principales sur lesquelles repose incXSLT.

L'optimisation des traitements de documents XML est un domaine encore à ses débuts, et prend une importance croissante à mesure que le standard XML se répand et induit des utilisations extrêmes de XML . Les retombées de ces travaux d'optimisation seront cruciales pour la suite.

# Seconde partie : contribution

# Chapitre 5

# Retour à la problématique de la transformation incrémentale


## Résumé

Ce chapitre revient sur l'intérêt de la transformation incrémentale de structures XML. Après avoir précisé la problématique que nous considérons, nous caractérisons un algorithme incrémental de transformation XML, et identifions deux sous-problèmes.


## Contenu


**5.1 Contexte** . . . . . . . . . . . . . . . . . . . . . . . . . . . . **42**
5.1.1 Une forte demande dans le contexte industriel . . . . . . . 43
5.1.2 Un domaine en plein essor . . . . . . . . . . . . . . . . . 43
5.1.3 Intérêt de la transformation incrémentale . . . . . . . . . . 44
**5.2 Formulation du problème** . . . . . . . . . . . . . . . . . . . . . **44**
**5.3 Caractérisation d'un algorithme pour la transformation incrémentale** . . . . . . . . . . . . . . . . . . . . . . . . . . . . **44**
**5.4 Partitionnement en deux sous-problèmes** . . . . . . . . . . . . . **45**
**5.5 Conclusion** . . . . . . . . . . . . . . . . . . . . . . . . . . . . **45**


## 5.1 Contexte

La transformation incrémentale s'inscrit dans le cadre d'un contexte industriel où la demande est importante et d'actualité. Du point de vue de la recherche scientifique, la transformation incrémentale s'inscrit dans le cadre de l'optimisation des traitements XML, un domaine qui est actuellement en plein essor, du fait de la grande expansion actuelle du standard XML. Cette section donne quelques pistes pour saisir l'importance des éventuelles retombées de ces travaux.

### 5.1.1 Une forte demande dans le contexte industriel

#### Documents de très grande taille

La transformation de structures XML est omniprésente dans la chaîne de traitement de l'information structurée. Récemment, la robustesse des technologies XML a suscité des utilisations extrêmes de XML. Il est dorénavant possible de citer de nombreux cas d'utilisation de documents XML de très grande taille.

Un exemple parmi d'autres est le NASDAQ qui emploie des documents XML pour représenter le taux de cloture quotidien des actions. Le document XML représentant les taux de cloture de l'année 1999 contient plus de 1,3 million d'éléments XML :

```
<YearOfNasdaqCloses Year="1999">
  <ClosingQuote Ticker="AAABB">
    <Date>01/01/1999</Date>
    <Price>6.25</Price>
    <Percent>0.5</Percent>
  </ClosingQuote>
  <!-- 1,353,818 éléments ClosingQuote supprimés -->
  <ClosingQuote Ticker="ZVXI">
    <Date>12/31/1999</Date>
    <Price>16.10</Price>
    <Percent>-1.05</Percent>
  </ClosingQuote>
</YearOfNasdaqCloses>
```

Lorsque ce type de document est transformé, toute optimisation de performance de la transformation est la bienvenue. En particulier, on cherche à éviter les calculs redondants.

#### Documents infinis, flots de structures XML

Un autre cas d'actualité est l'ajout d'une dimension interactive à la télévision numérique, via la transmission d'un flot de structures XML représentant l'interactivité. Ce flot est transmis en parallèle avec le flot audio-visuel classique, en permanence, pendant des heures. La taille totale du flot est telle que stocker la totalité du document XML n'est pas envisageable. La transformation doit donc s'effectuer en streaming, c'est-à-dire que le document cible doit être engendré à mesure que le flot du document source est reçu. Le but est donc de transformer sans connaître la totalité du document source ; c'est aussi un thème de recherche en plein essor. Nous donnerons quelques pistes dans le chapitre suivant pour voir comment la transformation en streaming peut être reliée avec la transformation incrémentale.

### 5.1.2 Un domaine en plein essor

La transformation des documents XML pose de nombreux problèmes lorsque les documents transformés deviennent gigantesques. Les procédés classiques de transformation atteignent leurs limites. Nous avons notamment vu dans la première partie de ce mémoire qu'aucun langage de transformation XML actuel n'était spécifiquement conçu pour effectuer une transformation de manière incrémentale.

### 5.1.3 Intérêt de la transformation incrémentale

Lorsqu'un document de très grande taille nécessite d'être transformé, nous pensons que le transformer incrémentalement est une bonne approche pour améliorer la performance. Considérons par exemple le cas du NASDAQ. Lors de l'ajout d'un sous-arbre de racine <ClosingQuote>, un problème de coût d'exécution se pose. La réexecution de la totalité de la transformation serait beaucoup trop coûteuse, compte tenu des légers changements à apporter au document cible. Nous cherchons donc à pouvoir réagir à des modifications du document source en répercutant immédiatement les changements correspondants au document cible, de manière que seules les portions affectées du résultat soient mises à jour, sans relancer la totalité du processus de transformation.

## 5.2 Formulation du problème

Le problème de la transformation incrémentale peut être formulé de différentes manières, par exemple, en terme d'optimisation : comment réutiliser au maximum l'ancien document cible pour économiser des calculs coûteux et redondants ? Comment réexécuter seulement le minimum d'instructions nécessaires ?

Une formulation du problème de la transformation incrémentale pourrait être la suivante : étant donné un document XML et une transformation, comment produire le document cible de la transformation et le maintenir à jour de manière optimale lorsque le document source subit des modifications ?

Un point intéressant de cette formulation est le caractère optimal de la mise à jour du document cible. En effet, la transformation incrémentale n'a de sens que si l'exécution d'un incrément de transformation nécessite des ressources en temps et en espace inférieures à celle de l'exécution de la transformation complète. Dans le cas où ce critère ne peut pas être respecté, la réexécution totale de la transformation est une meilleure solution.

## 5.3 Caractérisation d'un algorithme pour la transformation incrémentale

Le problème prend comme hypothèse le fait que la transformation par incrément ne s'effectue qu'après une première exécution de la transformation complète. Ainsi, il est possible de mener une analyse dynamique lors de l'exécution de la première transformation, de manière à faciliter par la suite le traitement incrémental.

Les travaux effectués autour de la transformation incrémentale [46] ont souligné l'intérêt que pouvait représenter une analyse statique de la transformation.

Un algorithme incrémental pour la transformation de documents XML prend comme données d'entrée :

- le document XML source $x$
- la transformation $T$
- le document cible $T(x)$, résultat de l'application de la transformation $T$ sur le document source $x$
- éventuellement des informations auxiliaires résultant d'une analyse statique de la transformation ou du document source
- éventuellement des informations auxiliaires issues d'une analyse dynamique menée lors de la première application de la transformation

- des modifications $\delta x$ apportées sur le document source

L'algorithme incrémental calcule de manière optimale le nouveau document cible de la transformation $T(x \oplus \delta x)$, et met à jour les éventuelles informations auxiliaires en vue du prochain traitement par incrément.

## 5.4 Partitionnement en deux sous-problèmes

Dans le problème de la transformation incrémentale nous avons clairement identifié deux sous-problèmes distincts :

- La détection des instructions à réévaluer suite à une modification du document source. Etant donné une modification du document source (par exemple l'insertion d'un élément), on cherche à déterminer les instructions de transformation qu'il faut réévaluer pour répercuter correctement cette modification sur le document cible.
- L'exécution des règles détectées comme étant à réévaluer. Une fois que les règles qu'il est nécessaire de réexécuter ont été identifiées, elles doivent être exécutées. L'état de l'art du traitement incrémental nous enseigne que deux types d'exécution peuvent être envisagés. L'exécution peut reposer sur la technique de recalcul sélectif (selective recomputation) ou bien de différence finie (finite differencing) [33] (c.f. chapitre précédent).
  incXSLT est un exemple où l'exécution est effectuée avec la technique de recalcul sélectif : l'arbre d'exécution de la transformation est parcouru (d'une manière similaire à une approche non incrémentale), avec des conditions pour éviter des calculs redondants. Le principal inconvénient de cette approche est que le parcours de l'arbre d'exécution est proportionnel à la taille de l'arbre cible. Par conséquent, pour de gros documents, le temps de mise à jour peut être important [46].
  L'approche des différences finies consisterait à exécuter directement les règles détectées comme étant à réévaluer. Cette approche nécessite la conception de fonctions de mise à jour du document cible pour réagir à chaque modification du document source. Cette approche soulève également un problème : celui de la restauration du contexte dans lequel les instructions doivent être exécutées.

## 5.5 Conclusion

Après avoir proposé une définition du problème de la transformation incrémentale, nous avons caractérisé un algorithme pour la transformation incrémentale. Nous avons ensuite identifié deux sous-problèmes distincts, ainsi qu'une limite de l'approche par recalcul sélectif.

Sur cette base, le chapitre suivant présente le modèle de transformation que nous proposons dans l'optique de la transformation incrémentale.

# Chapitre 6

# Un système de réécriture pour la transformation

## Résumé

Ce chapitre présente le modèle que nous proposons dans l'optique de la transformation incrémentale, en réponse aux critères que nous avons pu dégager. Ensuite, après une modélisation des modifications apportées au document source, des propositions sont faites pour la transformation incrémentale. Le modèle que nous avons conçu semble bien se prêter au traitement incrémental d'insertions dans l'arbre source. Ce chapitre s'achève en expliquant notamment les nombreuses pistes restant à étudier.

## Contenu

**6.1 Introduction** . . . . . . . . . . . . . . . . . . . . . . . . . . . . **47**
6.1.1 Vers un modèle de transformation . . . . . . . . . . . . . . 47
6.1.2 Eléments favorables pour la transformation incrémentale . . 47
6.1.3 Un nouveau modèle de transformation . . . . . . . . . . . 49
6.1.4 Motivations . . . . . . . . . . . . . . . . . . . . . . . . 50
6.1.5 Modèle de document considéré . . . . . . . . . . . . . . . 51
**6.2 Règle de transformation** . . . . . . . . . . . . . . . . . . . . . . **51**
6.2.1 Syntaxe des expressions . . . . . . . . . . . . . . . . . . 51
6.2.2 Sémantique des expressions . . . . . . . . . . . . . . . . 52
6.2.3 Contraintes associées à une règle . . . . . . . . . . . . . . 56
**6.3 Modèle d'exécution** . . . . . . . . . . . . . . . . . . . . . . . . **57**
6.3.1 Règle applicable . . . . . . . . . . . . . . . . . . . . . . 57
6.3.2 Exécution de la transformation . . . . . . . . . . . . . . . 59
**6.4 Exemple** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **59**
6.4.1 Modèles de document considérés . . . . . . . . . . . . . . 59
6.4.2 Transformation . . . . . . . . . . . . . . . . . . . . . . . 61
6.4.3 Exécution sur une instance . . . . . . . . . . . . . . . . . 62
**6.5 Conservation de l'historique d'exécution** . . . . . . . . . . . . . **65**
**6.6 Restriction du modèle** . . . . . . . . . . . . . . . . . . . . . . . **67**
6.6.1 Opérateurs de génération . . . . . . . . . . . . . . . . . . 67

6.6.2 Optique purement générative . . . . . . . . . . . . . . . . 67
**6.7 Répercution des modifications du document source . . . . . . . 67**
6.7.1 Modélisation des modifications du document source . . . . 67
6.7.2 Répercution d'une insertion dans l'arbre source . . . . . . 69
6.7.3 Répercution d'une modification d'un noeud texte . . . . . . 74
6.7.4 Répercution d'une suppression dans l'arbre source . . . . . 74
6.7.5 Récapitulatif . . . . . . . . . . . . . . . . . . . . . . . . 76
6.7.6 Traitement de listes de modifications élémentaires . . . . . 77
6.7.7 Exemples . . . . . . . . . . . . . . . . . . . . . . . . . 79
**6.8 Synthèse . . . . . . . . . . . . . . . . . . . . . . . . . . . . 83**
6.8.1 Evaluation du modèle . . . . . . . . . . . . . . . . . . . 83
6.8.2 Limites du modèle . . . . . . . . . . . . . . . . . . . . . 84


## 6.1 Introduction

Avant d'aborder le problème de la transformation incrémentale proprement dit, il est nécessaire de définir le modèle de transformation considéré. L'étude de l'état de l'art des langages de transformation d'une part et du traitement incrémental d'autre part nous a permis d'établir un certain nombre d'éléments favorables pour la conception d'un modèle de transformation, dans l'optique de la transformation incrémentale. Nous présentons tout d'abord ces critères, puis la démarche qui nous a permis de proposer un nouveau modèle de transformation.

### 6.1.1 Vers un modèle de transformation

**Expressivité adaptée** Tout d'abord le modèle doit permettre d'exprimer des transformations de structures complexes et pas seulement de simples fonctionnalités comme la traduction d'éléments (fonctionnalité à laquelle peut se réduire un premier modèle naïf, c.f. Annexe A). Pour cela, de l'étude des langages de transformation il apparaît que le modèle doit fournir au moins :

1. la possibilité d'extraire des informations du document source en autorisant les tests de structure (notamment sur le contexte hiérarchique des noeuds, c'est-à-dire leur chemin depuis la racine de l'arbre), de valeurs dans la structure (par exemple des tests sur les valeurs d'attributs ou sur le contenu textuel des noeuds) ;
2. la possibilité de générer des sous-structures dans l'arbre cible (par exemple des sous-arbres)

### 6.1.2 Eléments favorables pour la transformation incrémentale

D'après l'étude de l'état de l'art du traitement incrémental, il est possible de formuler intuitivement quelques éléments favorables pour qu'un modèle de transformation se prête à la transformation par incrément.

1. **Règles de transformation de granularité fine** Tout d'abord, le modèle devrait permettre de spécifier des unités de composition les plus petites et légères possibles. Dans le cas où la transformation le permet, le modèle devrait faciliter l'expression de règles de transformation qui traitent les sous-structures les plus

petites possibles du document source. De telles règles sont dites de granularité fine.

Ainsi, comme dans la technique d'évaluation partielle [43] le domaine d'entrée est partitionné en petites unités et la réaction à des modifications légères du document source est facilitée. En effet, plus petites sont les classes du domaine d'entrée partitionné, i.e. plus fine est la granularité des règles, plus grande est la probabilité qu'une règle ne soit pas concernée lorsque le document source est modifié à un endroit quelconque. Dans une certaine mesure, la granularité fine des règles permet donc de réagir à des modifications légères du document source en minimisant les calculs redondants.

2. **Règles de transformation indépendantes**

   Dans un modèle de transformation à base de règles, les règles peuvent être plus ou moins indépendantes. Une règle de transformation peut être liée à une autre règle dans la mesure où les paramètres d'entrée de la seconde règle utilisent les paramètres de sortie de la première règle. Le modèle d'exécution, qui définit la composition des règles de transformation, peut être tel que lorsque des paramètres d'entrée d'une règle changent, les paramètres d'entrée de toutes les autres règles changent également.

   Deux règles de transformation $R_1$ et $R_2$ sont dites *fortement indépendantes* si aucun paramètre de sortie de $R_1$ n'est un paramètre d'entrée de $R_2$ et inversement. La composée de règles de transformation fortement indépendantes commute, c'est-à-dire que les règles produisent le même résultat indépendamment de leur ordre respectif d'application (en notation fonctionnelle $R_1(R_2(x)) = R_2(R_1(x))$).

   Dans la mesure où la transformation le permet, le modèle de transformation devrait permettre d'exprimer des règles de transformation aussi indépendantes que possible, et si possible, fortement indépendantes. Le domaine d'entrée pourrait alors être considéré comme partitionné, et les règles comme des fonctions résiduelles du domaine d'entrée, d'une manière analogue à la technique de l'évaluation partielle [43].

3. **Facilité de la décomposition**

   Lors de la spécification d'une transformation, la décomposition de la transformation en règles indépendantes de granularité fine ne devrait pas être une tâche laissée entièrement à la charge du programmeur, mais devrait être naturelle et facilitée par le modèle de transformation.

4. **Conservation possible de l'historique d'exécution**

   Dans notre état de l'art du traitement incrémental, nous avons identifié la conservation de l'historique d'exécution comme une technique éprouvée pour faciliter un traitement incrémental ultérieur. L'exécution d'une transformation dans notre modèle devrait être limitée de façon à ce qu'il soit possible de stocker un historique d'exécution. De plus, comme le fait remarquer Lionel Villard [46], un langage sans effet de bord permet de ne pas avoir à stocker l'historique d'exécution dans sa globalité.

5. **Approche par différence finie**

   Pour éviter l'inconvénient lié à l'approche par recalcul sélectif que nous avons présenté dans le chapitre précédent, nous proposons autant que possible de tenter d'adopter une approche par différence finie.

### 6.1.3 Un nouveau modèle de transformation

#### Démarche

Définir un modèle de transformation passe par la définition des trois fonctions principales que nous avons identifié lors de l'état de l'art des langages de transformation, et que nous rappelons ci-après :

1. la sélection et l'extraction de fragments dans l'arbre source ;
2. la réorganisation des fragments sélectionnés, la création de nouveaux fragments et la combinaison de ceux-ci ;
3. la génération du document cible à partir de ces fragments

Ces trois fonctions principales sont conçues en élaborant d'une part des règles de transformation et d'autre part un modèle d'exécution de ces règles.

Comme nous l'avons constaté dans l'état de l'art, la partie gauche d'une règle de transformation a fait l'objet de nombreux travaux, et il existe plusieurs concepts mûrs appropriés (expressions régulières d'arbres, XPath, etc.)

La définition d'un modèle d'exécution, qui décrit comment les règles sont appliquées, pourrait s'inspirer des automates d'arbre, des automates à jeton [30] récemment proposés pour la transformation, ou encore d'un modèle étendu inspiré de celui que nous avons proposé pour la traduction incrémentale (c.f. Annexe A). Cependant, après avoir considéré les primitives de génération de bas niveau proposées par DOM, nous avons finalement renoncé à étendre ce modèle. La raison principale est la difficulté rencontrée pour caractériser et définir avec précision la partie droite des règles, concernant la génération de l'arbre cible, afin d'obtenir un modèle de transformation expressif.

Notre attention s'est porté sur un langage spécialisé et original [50]. Ce langage décrit une approche simple dans lesquelles les règles font intervenir le concept de réécriture. Bien que ce langage de transformation soit devenu de moindre importance du fait de son expressivité limitée et de l'émergence de XSLT, l'approche qui est proposée permet de décrire précisément quelques primitives de génération de l'arbre cible [50].

#### Contexte du travail et remerciements

A ce stade, une idée a été proposée dans l'équipe : employer XPath dans une approche similaire. Du fait de l'expressivité de XPath, un modèle de transformation basé sur cette approche n'aurait pas les mêmes limitations.

Dans le cadre de ce DEA, nous avons pris part aux tentatives exploratoires pour essayer de définir et de mieux cerner un modèle de transformation reposant sur cette idée. Cette participation s'est notamment traduite par des tentatives d'élaboration de modèles d'exécution qu'il serait possible d'associer à de telles règles, en gardant à l'esprit la transformation incrémentale. Plusieurs propositions ont été avancées et démontées au cours de réflexions constructives menées avec les membres du projet WAM [62]. C'est ainsi que la conception d'un nouveau langage de transformation XML, "Omega", a démarré dans le projet WAM.

Il faut souligner que ces travaux ont des répercutions beaucoup plus larges que le cadre de ce DEA. Nos explorations ont en partie contribué à la conception de Omega. Nous avons ainsi eu la chance de participer à l'élaboration des prémices de ce nouveau langage.

#### Choix d'orientation

Plutôt que de tenter d'élaborer un autre modèle de transformation spécialisé, nous avons été séduit par la construction d'un modèle reposant sur les idées ainsi développées dans l'équipe. Nous donnons les principales raisons de ce choix en 6.1.4.

#### Plan du chapitre

Ce chapitre débute par la présentation du modèle de transformation que nous proposons. Ce modèle tente de répondre aux requis précédemment exposés dans l'optique de la transformation incrémentale. En particulier, le modèle d'exécution a été conçu spécifiquement à cette fin.

Nous proposons ensuite une modélisation des modifications pouvant survenir sur le document source de la transformation.

Nous proposons enfin des techniques pour traiter incrémentalement ces modifications en les répercutant sur le document cible.

Au final, de nombreuses pistes restantes concernant l'étude et d'extensions de ce modèle sont soulevées.

#### Nouveau modèle proposé

Le modèle de transformation que nous proposons est à base de règles. Son originalité principale est d'utiliser XPath pour décrire la génération de l'arbre cible, ce qui est nouveau et n'a jamais encore été étudié à notre connaissance. L'autre originalité de ce modèle est que son modèle d'exécution est fondé sur celui d'un système de réécriture. Les principales raisons qui motivent ces choix sont données dans la section suivante.

Dans le cadre de ce modèle, nous avons isolé un sous-ensemble de XPath et détourné sa sémantique classique de sélection pour lui conférer une sémantique d'identification (matching) dans un arbre XML. Nous utilisons le même fragment de XPath aussi bien pour décrire l'exploitation de données de l'arbre source que la génération de l'arbre cible.

Dans notre modèle, une règle de transformation est la réécriture d'une expression XPath relativement à un contexte donné. Le contexte est exprimé par matching dans les arbres source et cible de la transformation.

### 6.1.4 Motivations

**Intérêts d'employer XPath** XPath est une norme [70] bien formalisée [49], qui fait l'objet de nombreuses études. Employer XPath permet de décrire précisément la génération de l'arbre cible. De plus, la notation compacte de XPath permet d'exprimer des règles de manière concise.

**Intérêts de la réécriture** Tout d'abord, le principe d'exécution des systèmes à base de règles, de la forme "tant qu'une règle est applicable, l'appliquer", nous semble *a priori* bien adapté au traitement incrémental de l'insertion de noeuds dans le document source de la transformation.

De plus, nous ne voyons pas d'obstacle à l'expression de règles indépendantes, et même fortement indépendantes et de granularité fine à l'aide d'un système de réécriture.

Enfin, les travaux de C. Kirchner et al. sur ELAN ont montré des résultats encourageants autour de la performance des transformations implantées sur un système de réécriture [19].

### 6.1.5 Modèle de document considéré

Nous considérons le modèle de document défini par la norme XPath [70]. Nous avons présenté ce modèle dans la première partie de ce mémoire.

## 6.2 Règle de transformation

Dans cette section, les règles de transformation qui constituent le coeur du modèle que nous proposons sont présentées. Dans ce modèle, une règle de transformation est un triplet $(N, R, \phi)$ où $R$ est une règle de réécriture de la forme suivante :

$$l\ ,\ \ r \longrightarrow r'$$

$\phi$ est un ensemble de contraintes associé à la règle. Une règle est identifiée de manière unique par son nom $N$.

Chaque composante de la règle sera détaillée dans les sections suivantes. Nous introduisons succinctement le but de chaque expression : $l$ est une expression permettant d'exploiter des informations de l'arbre source ; $r$ permet de définir un contexte dans l'arbre cible. Enfin $r \longrightarrow r'$, qui dénote la réécriture de $r$ en $r'$, décrit la génération de l'arbre cible.

Le même sous-ensemble de XPath est utilisé pour décrire l'exploitation des informations du document source (expression $l$) et pour décrire la génération de l'arbre cible (expressions $r$ et $r'$). La section suivante introduit le sous-ensemble de XPath considéré.

### 6.2.1 Syntaxe des expressions

Nous proposons une version restreinte de XPath, qui est un sous-ensemble de la version proposée dans [48]. A ce sous-ensemble nous ajoutons comme extension le test d'égalité =, que la norme Xpath [70] définit.

Nous introduisons tout d'abord la syntaxe des expressions $l$, $r$ et $r'$, inspirée de la grammaire formelle donnée dans [48]. La syntaxe est donné sous une forme BNF, dans laquelle "::=" dénote la réécriture d'un non terminal et "|" dénote l'alternance.

$\wedge$ dénote la racine d'un arbre. Les expressions sont absolues :

$$e \text{ ::= } \wedge/p \ \mid \ \wedge//p$$

Après la référence à la racine du document, l'expression définit un chemin par étapes :

$$p \text{ ::= } p|p \ \mid \ a :: N \ \mid \ (p) \ \mid \ p/p \ \mid \ p[q]$$

Les chemins peuvent faire référence aux 13 axes définis dans XPath 1.0 via le symbole $a$ :

$$a \in \{self,\ attribute,\ namespace,\ child,\ parent,\ descendant,\ ancestor,\ descendant\text{-}or\text{-}self,\ ancestor\text{-}or\text{-}self,\ following,\ preceding\ preceding\text{-}sibling,\ following\text{-}sibling\}$$

Un chemin peut aussi contenir des prédicats ($p[q]$). Une expression à l'intérieur d'un prédicat peut seulement contenir des tests d'existence de noeuds, des affectations ou bien des tests d'égalité sur le contenu des noeuds :

$$q ::= p \mid p = \$v \mid p = "U"$$

Enfin, un test de noeud $N$ peut être un nom $n$ non qualifié, ou un des types de noeuds définis dans le modèle de données de XPath :

$$N ::= n \mid text() \mid node() \mid comment() \mid processing\text{-}instruction() \mid element()$$

Des sucres syntaxiques analogues à ceux définis dans XPath et qui permettent d'abréger les étapes $a :: N$ sont ajoutés :

$$p ::= p//p \mid .. \mid . \mid N \mid @N$$

**Spécificité d'une expression $l$** Les expressions $l$ et $r$ utilisent la même syntaxe présentée précédemment. Seule une restriction est associée aux expressions $l$ : celles-ci ne peuvent pas comporter d'union (|) dans leur dernière étape. Par exemple $\wedge/a/(b|c)/d$ est une expression $l$ valide, mais pas $\wedge/a/(b|c)$. Nous verrons la raison de cette restriction lors de la description de la sémantique d'une expression $l$, dans la section suivante.

**Frontière du sous-ensemble retenu** Notons que nous considérons un sous-ensemble de XPath conséquent. Le sous-ensemble retenu dans d'autres études [32] est en comparaison beaucoup plus restreint.

Notons cependant que nous considérons une version simplifiée des prédicats par rapport à la norme XPath dans son intégralité. En particulier, nous ne considérons pas les prédicats faisant intervenir la position des noeuds, ni les prédicats faisant intervenir des expressions booléennes. Un des objectifs de poursuite du travail consiste à étendre les prédicats considérés. Nous verrons en 6.8.2 les problèmes que peut soulever une telle extension.

### 6.2.2 Sémantique des expressions

#### Exploitation des informations de l'arbre source

**Matching de sous-arbre dans l'arbre source** Bien qu'une expression $l$ soit exprimée avec un fragment de la syntaxe de XPath, nous ne lui donnons pas la sémantique classique de sélection d'une expression XPath [49]. Au contraire, nous proposons de détourner la sémantique classique pour donner à une expression $l$ le sens d'une expression qui identifie des sous-arbres dans l'arbre source par matching.

Deux différences principales existent entre la sémantique que nous proposons et la sémantique classique d'une expression XPath. La première différence est qu'avec

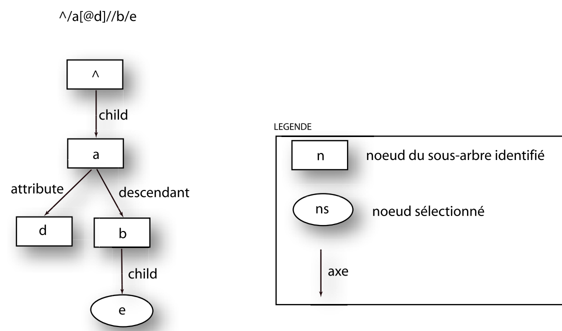


FIG. 6.1 – Sous-arbre identifié et noeud sélectionné

la sémantique classique, l'évaluation d'une expression XPath renvoie *un ensemble* de noeuds sélectionnés dans un arbre ; alors qu'ici, une expression $l$, lorsqu'elle est essayée sur un arbre XML, identifie *un seul* sous-arbre, le premier qui correspond à l'expression, dans l'ordre de l'arbre.

La seconde différence est que lorsque $l$ est essayée sur l'arbre source, elle identifie non pas *des noeuds* mais des *sous-arbres*. Tous les noeuds et les relations décrits dans $l$ sont considérés comme identifiés, et non pas seulement les noeuds terminaux classiquement désigné par $l$.

Par exemple, l'expression $\wedge/person[name]$ va identifier le premier sous-arbre du document source composé de la racine et d'un élément $person$, fils de la racine du document, avec un élément fils $name$.

La figure 6.1 illustre le sous-arbre identifié par une expression $l$ de la forme

$$\wedge/a[@d]//b/e$$

.

**Noeud sélectionné dans le sous-arbre** Dans le sous-arbre identifié par une expression $l$, le noeud désigné dans la dernière étape de $l$ occupe un rôle particulier. Nous qualifions ce noeud de "sélectionné". La figure 6.1 illustre le noeud sélectionné au sein du sous-arbre identifié par $\wedge/a[@d]//b/e$.

Notons qu'au sein d'un sous-arbre, un et un seul noeud est sélectionné. L'unicité du noeud sélectionné nous permet de décrire en 6.3.2 un moteur d'application des règles simple. C'est la raison pour laquelle les expressions $l$ ne peuvent pas comporter d'union (|) dans leur dernière étape.

**Test de valeurs de structure** Pour permettre le test de valeurs dans la structure, conformément à notre exigence en 6.1.1, l'opérateur $=$ est introduit. Au sein des prédicats de l'expression $l$, cet opérateur $=$ permet de tester les valeurs de la structure comme dans l'utilisation standard de XPath. Par exemple :

$$\wedge/person[@gender = "F"]$$

identifie les éléments $person$ fils de la racine dont l'attribut $gender$ a la valeur $"F"$. La valeur d'un noeud est la linéarisation du contenu de ses descendants, comme défini dans XPath.

**Extraction de valeurs de structure** Pour pouvoir extraire des valeurs de la structure, et transmettre des informations du document source au document cible, nous introduisons l'affectation de variables.

Dans les prédicats de l'expression de matching $l$, l'opérateur d'affectation $=$ suivi d'une variable (par exemple $\$v$) permet de récupérer la valeur (le contenu linéarisé) des noeuds. Par exemple lorsque l'expression suivante :

$$\wedge/person[name = \$name]$$

est essayée sur un arbre, la variable $\$name$ est affectée et prend la valeur linéarisée des noeuds textes descendants du noeud $\wedge/person/name$.

Notons que dans le cas d'une expression $l$, l'affectation est notée dans le sens inverse du sens usuel.

### Génération de l'arbre cible

Nous proposons d'utiliser le même fragment de XPath pour décrire la génération de l'arbre cible que celui présenté précédemment pour décrire l'exploitation d'informations de l'arbre source.

C'est un des points clés de notre modèle, puisqu'il fournit la possibilité de décrire la génération de l'arbre cible avec la même syntaxe et la même sémantique que celle utilisée pour les expressions exploitant le document source.

Nous avons analysé cette proposition originale qui s'avère viable.

**Principe de la solution** Une expression $r$ a la même sémantique sur l'arbre cible qu'une expression $l$ sur l'arbre source. Lorsqu'une expression $r$ est essayée sur l'arbre cible, elle identifie le premier sous-arbre, dans l'ordre de l'arbre cible, correspondant à l'expression. Tous les noeuds et les relations décrits dans $r$ sont considérés comme identifiés, et pas seulement le noeud feuille désigné par $r$. Ce dernier conserve la dénomination de noeud sélectionné, au sein du sous-arbre. Un exemple est donné sur la figure 6.2.

$r'$ décrit ce qu'est devenu ce sous-arbre de l'arbre cible après que l'opération de génération a eu lieu.

**Opération de génération** Nous avons construit nos règles de la forme $N : l, r \longrightarrow r'$, pour que la réécriture de $r$ en $r'$ : $r \longrightarrow r'$ spécifie l'opération de génération qui est effectuée dans l'arbre cible par la règle de nom $N$.

Pour expliquer l'opération de génération qui est décrite par $r \longrightarrow r'$, nous considérons que les expressions $r$ et $r'$ peuvent toutes deux être mises sous la forme d'un graphe. Les sous-structures identifiées par une expression correspondent aux sommets du graphe ; les relations existant entre les noeuds (fils, attribut, etc.) correspondent aux arcs du graphe.

La figure 6.2 illustre le cas de la réécriture suivante :

$$\wedge/a[@d]//b/(e|f) \longrightarrow \wedge/a//b/e[d]$$

Dans cet exemple, les sommets en forme de rectangle représentent les noeuds de l'arbre cible identifiés par l'expression. Les noeuds qui sont sélectionnés par l'expression XPath interprétée avec sa sémantique classique sont représentés en forme d'ellipse. Enfin les relations existantes entre les noeuds décorent les arcs.

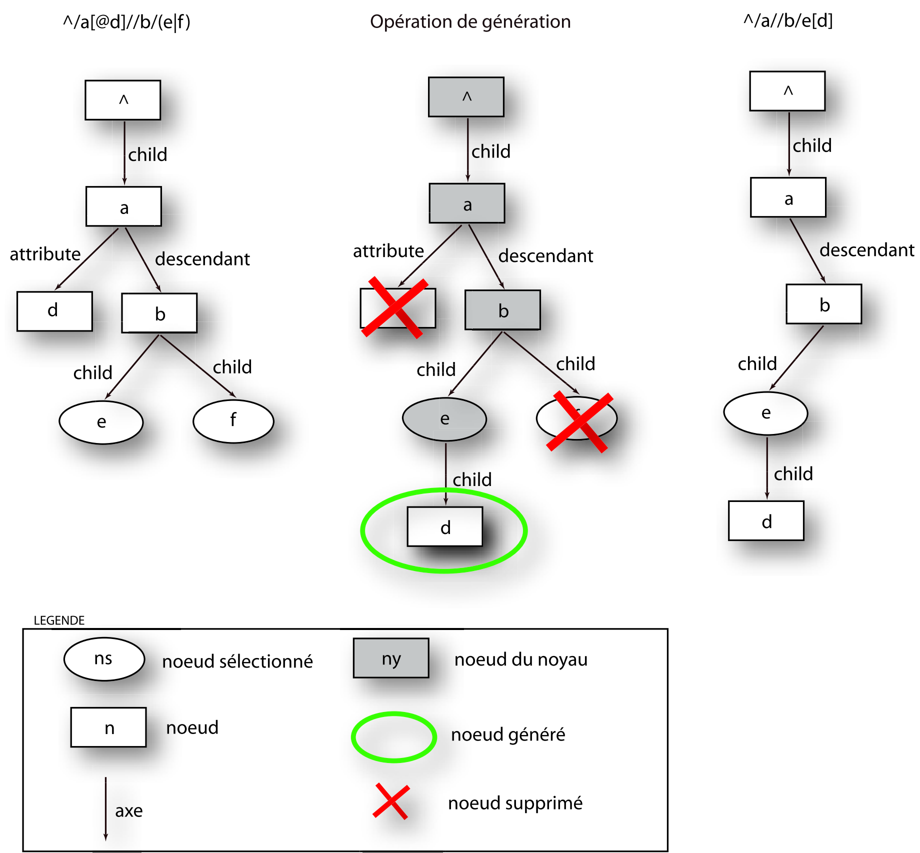


FIG. 6.2 – Opération de génération

L'identification du plus grand sous-graphe commun aux graphes de $r$ et $r'$, nommé noyau [9], permet de définir les noeuds qui seront générés dans l'arbre cible. Tous les noeuds de $r'$ ne faisant pas partie du noyau, c'est-à-dire n'étant pas communs à $r$ et $r'$ sont générés, et à l'inverse, tous les noeuds de $r$ ne faisant pas partie du noyau sont supprimés.

**Convention sur l'ordre d'insertion** Il faut noter que dans le cas où un noeud de même nom que celui qui est généré est déjà présent dans l'arbre cible avant la génération, l'insertion du nouvel élément s'effectue dans l'ordre de l'arbre cible. Par exemple, dans le cas de l'opération $\wedge/a \longrightarrow \wedge/(a|a)$ le noeud "a" généré est inséré comme fils en seconde position de la racine (à droite du noeud "a" existant).

### 6.2.3 Contraintes associées à une règle

Empiriquement, nous avons constaté qu'il est nécessaire de pouvoir spécifier des contraintes entre les expressions $l$ qui identifient des noeuds dans l'arbre source, et les expressions $r$ et $r'$ qui identifient et génèrent dans l'arbre cible.

Nous avons identifié comme une nécessité le fait de pouvoir contrôler les informations transmises du document source au document cible, ainsi que pouvoir les contraindre.

En réponse à ce besoin, nous proposons l'expression de contraintes par l'intermédiaire de variables dans les expressions $l$, $r$ et $r'$ liées par un ensemble d'équations. Cet ensemble est nommé $\phi$ et associé à la règle (c.f. 6.2).

Ainsi, lorsqu'une règle de réécriture est spécifiée, des contraintes peuvent s'exercer d'une part entre les expressions $l$ et $r$ et d'autre part entre les expressions $l$ et $r'$, par l'intermédiaire de variables libres dans $l$, $r$ et $r'$. Des équations associées à la règle de réécriture permettent de lier ces variables.

L'exemple ci-après illustre le cas où des variables lient d'une part les expressions $l$ et $r$ et d'autre part les expressions $l$ et $r'$. Pour faciliter la lecture de ce premier exemple, nous avons veillé à bien faire apparaître les séparations entre les expressions $l$, $r$ et $r'$ de la règle. Dans la suite, nous adopterons une forme un peu plus compacte.

$rule_product$ :
$\wedge/products/product[name = \$n1][price/@currency = "EUR"][amount/text() = \$p1]$
,
$\wedge/items/item[@name = \$n2]$
$\longrightarrow$
$\wedge/items/item[@name = \$n2][@usdprice = \$p2]$

$with$
$\$n2 = \$n1 \; and \; \$p2 = \$p1 * 0.9;$

Nous proposons dans un premier temps un jeu d'opérateurs limité pour exprimer ces contraintes :

- sur les chaînes de caractères : affectation (=) et concaténation (+)
- sur les entiers et flottants : affectation (=), produit (*) et somme (+)

Notons que lors de l'extension de ce jeu d'opérateurs, des restrictions sont à prévoir sur la complexité des contraintes permises, afin de conserver une exécution efficace des règles de transformation. Ceci constitue une piste ouverte : sans doute faudra-t-il limiter ces contraintes à des équations linéaires, de manière à garder la possibilité d'exprimer efficacement chaque variable en fonction des autres.

**Simplification de notation** Dans la spécification d'une règle, lorsqu'une variable de l'expression $l$ est liée à une variable de l'expression $r$ ou $r'$ par une contrainte d'égalité, on s'autorise à remplacer le nom chaque variable par le même nom et faire disparaître la contrainte d'égalité. Ainsi la règle suivante :

$\wedge/person[@gender = "M"][name = \$n],$
$\wedge$
$\longrightarrow$
$\wedge/man[@name = \$n]$

est en réalité spécifiée à la place de :

$\wedge/person[@gender = "M"][name = \$n1],$
$\wedge$
$\longrightarrow$
$\wedge/man[@name = \$n2]$

$with \quad \$n2 = \$n1$

## 6.3 Modèle d'exécution

Après avoir décrit les règles de transformation du modèle que nous proposons, nous décrivons dans cette section comment celles-ci sont appliquées et ce qu'est l'exécution d'une transformation.

### 6.3.1 Règle applicable

Une règle $(N, (l, r \longrightarrow r'), \phi)$ est applicable si et seulement si toutes les conditions suivantes sont vérifiées :
- $l$ identifie au moins un sous-arbre de l'arbre source
- ce sous-arbre n'a pas déjà été identifié auparavant par la règle $N$ (i.e. la décoration du noeud sélectionné dans le sous-arbre ne comporte pas le nom N ; c.f. 6.3.2)
- chaque équation associée à la règle de transformation (chaque élément de $\phi$) a une solution
- $r$ identifie au moins un sous-arbre de l'arbre cible

On dit qu'une règle est essayée lorsque son caractère applicable est testé. Pour tester le caractère applicable d'une règle sur un arbre source et un arbre cible donnés, on considère deux algorithmes de matching qui fonctionnent respectivement sur l'arbre source et sur l'arbre cible.

**Algorithme de matching dans l'arbre source** Le premier algorithme, étant donné $l$ et un arbre source, identifie le premier sous-arbre correspondant à $l$ et trouve l'instanciation correspondante des variables libres de $l$.

Par exemple, lorsque l'expression $l$ de la règle "rule_product" précédemment exposée est essayée pour la première fois sur le document source suivant :

```
<products>
  <product>
    <name>Memory Disk</name>
    <price currency="EUR">
      <amount>10</amount>
    </price>
  </product>
  <product>
    <name>Laptop computer</name>
    <price currency="EUR">
      <amount>3300</amount>
    </price>
  </product>
</products>
```

L'algorithme de matching identifie le premier sous-arbre du document décrit par $l$, au sein de laquelle l'élément "product" est sélectionné. L'instanciation

$$\{(n1, "Memory\ Disk"), (p1, 10)\}$$

est de plus renvoyée. Lorsque l'expression $l$ de la règle "rule_product" est essayée pour la deuxième fois, le second sous-arbre est identifié par l'algorithme et l'instanciation suivante associée est renvoyée :

$$\{(n1, "Laptop\ computer"), (p1, 3300)\}$$

.

**Algorithme de matching dans l'arbre cible** Le second algorithme est appliqué si les équations associées à la règle ont une solution. Dans ce cas, la résolution de ces équations donne une instanciation $\sigma$ des variables de $r$ liées à celles de $l$. Etant donné une expression $r$, un arbre cible, et l'instanciation $\sigma$, le second algorithme identifie le premier sous-arbre qui correspond à $(r, \sigma)$ dans l'ordre de l'arbre cible, et trouve l'instanciation correspondante des variables restant libres dans $r$.

Par exemple, reprenons l'exemple précédent dans le cas où l'arbre cible est le suivant :

```
<items>
  <item name="Mouse"/>
  <item name="Memory Disk"/>
  <item name="Screen"/>
</items>
```

L'algorithme de matching dans l'arbre source a précédemment renvoyé l'instanciation

$$\{(n1, "Memory\ Disk"), (p1, 10)\}$$

.

Les équations associées à la règle ont une solution, leur résolution produit l'instanciation

$$\{(n2, "Memory\ Disk"), (p2, 9)\}$$

.

L'expression $r$ ainsi instanciée :

$$\wedge/items/item[@name = "Memory\ Disk"]$$

est alors essayée sur l'arbre cible, ce qui retourne le sous-arbre composé du premier élément "item" de nom "Memory Disk" du document cible, c'est-à-dire ici le deuxième noeud "item" dans l'ordre du document. Alors, l'opération de génération peut avoir lieu.

### 6.3.2 Exécution de la transformation

#### Principe

L'arbre source de la transformation ne subit aucune modification durant la transformation. L'arbre cible est construit en totalité. Une règle est essayée et appliquée dans un contexte donné, c'est-à-dire étant donné un arbre source et un arbre cible.

#### Choix des règles essayées

Les règles spécifiées par le programmeur sont ordonnées : $r_1, r_2, ...r_n$. Chaque règle est essayée dans l'ordre où elle a été spécifiée. Lorsqu'une règle est applicable elle est appliquée. Lorsque la dernière règle $r_n$ a été considérée, c'est au tour de la première règle $r_1$ d'être considérée, et ainsi de suite. L'exécution de la transformation est terminée lorsque plus aucune règle n'est applicable. L'algorithme d'exécution d'une transformation est donné en figure 6.3.

#### Unicité de l'application d'une règle

Une règle ne peut pas s'appliquer à un même sous-arbre de l'arbre source plus d'une fois. Une fois qu'une règle a été essayée, si elle a identifié un sous-arbre de l'arbre source, ce sous-arbre doit être marqué comme identifié par cette règle. Pour cela, nous proposons de décorer le noeud sélectionné au sein du sous-arbre par le nom de la règle. Cette décoration a pour but d'empêcher toute identification du même sous-arbre par la même règle, et par conséquent toute réapplication ultérieure de la même règle à ce sous-arbre.

## 6.4 Exemple

Cette section présente un exemple de transformation de structure XML avec notre modèle. Elle présente tout d'abord les modèles de document source et cible considérés, puis exprime la transformation correspondante dans notre modèle. Enfin, l'exécution de la transformation sur une instance source est commentée.

### 6.4.1 Modèles de document considérés

Les modèles de document source et cible de la transformation sont donnés ci-après sous forme de grammaire d'arbre. Les schémas RelaxNG correspondants sont donnés en annexe B. L'exemple de transformation choisi est inspiré de [5]. On admet que les personnes modélisées sont identifiées de manière unique par leur nom.

```
// n : nombre de règles spécifiées
// d : indice de la dernière règle appliquée
// nb : nombre de règles testées comme non applicables
// depuis que la dernière règle a été appliquée

// Première exécution de la transformation :
d ← 0
nb ← 0
essayer_regles(1)

// fonction qui essaie les règles à partir de la règle d'indice i
DebutFonction essayer_regles(i :entier)
        j ← i
        Tant que (nb<n)
            Si (r_j est applicable)
                appliquer(r_j)
                d ← j
                nb ← 0
            Sinon
                nb ← nb + 1
            FinSi
            j ← j modulo n + 1
        FinTantque
FinFonction
```

FIG. 6.3 – Algorithme d'application des règles

**Modèle de document source**

```
Person -> person[Name, Children] {gender="M|F"}

Name -> name[#, Surname?]

Surname -> surname[#]

Children -> children[Person*]
```

**Modèle de document cible**

```
Man -> man[Sons, Daughters] {name="#"}

Sons -> sons[Man*]

Daughters -> daugthers[Woman*]

Woman -> woman[Sons, Daughters] {name="#"}
```

### 6.4.2 Transformation

La transformation d'une instance du modèle source en instance du modèle cible peut s'exprimer à l'aide des six règles suivantes :

```
//transforme les personnes dont on ne connaît pas de parent

r1 :
 ^/person[@gender="M"][name=$n],
 ^
->
 ^/man[@name=$n][sons][daughters]

r2 :
 ^/person[@gender="F"][name=$n],
 ^
->
 ^/woman[@name=$n][sons][daughters]

// transforme les personnes dont on connaît le père

r3 :
 ^//person[@gender="M"][name=$pere]/children/person[@gender="M"][name=$n],
 ^//man[@name=$pere]/sons
->
 ^//man[@name=$pere]/sons/man[@name=$n][sons][daughters]

r4 :
 ^//person[@gender="M"][name=$pere]/children/person[@gender="F"][name=$n],
 ^//man[@name=$pere]/daugthers
->
 ^//man[@name=$pere]/daughters/woman[@name=$n][sons][daughters]
```

```
// transforme les personnes dont on connaît la mère
r5 :
 ^//person[@gender="F"][name=$mere]/children/person[@gender="M"][name=$n],
 ^//woman[@name=$mere]/sons
->
 ^//woman[@name=$mere]/sons/man[@name=$n][sons][daughters]

r6 :
 ^//person[@gender="F"][name=$mere]/children/person[@gender="F"][name=$n],
 ^//woman[@name=$mere]/daughters
->
 ^//woman[@name=$mere]/daugthers/woman[@name=$n][sons][daughters]
```

### 6.4.3 Exécution sur une instance

Cette section commente l'exécution de la transformation sur une instance. L'instance source considérée est la suivante :

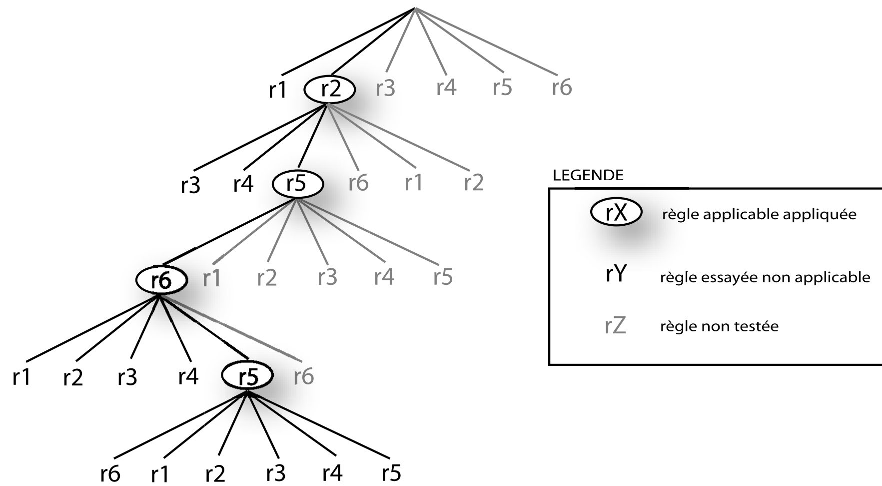


FIG. 6.4 – Séquence d'exécution

```
<person gender="F">
  <name>Julie</name>
  <children>
    <person gender="M">
        <name>Paul</name>
        <children/>
    </person>
    <person gender="F">
        <name>Dominique</name>
        <children>
            <person gender="M">
                <name>Bertrand</name>
                <children/>
            </person>
        </children>
    </person>
  </children>
</person>
```

**Exécution de la transformation** Lorsque l'exécution de la transformation débute, l'arbre cible est initialisé avec une racine $\wedge$.

Conformément au modèle d'exécution, la règle $r1$ est tout d'abord essayée mais n'est pas applicable car elle n'identifie pas d'élément dans l'arbre source. La règle $r2$ est alors essayée. Elle est applicable puisqu'elle identifie d'une part un sous-arbre dans l'arbre source. Celui-ci comprend l'élément "person" qui représente la personne de nom "Julie". D'autre part la racine de l'arbre cible est identifiée. $r2$ est donc appliquée

et à l'issu de son application le document cible est le suivant :

```
<woman name="Julie">
    <sons/>
    <daugthers/>
</woman>
```

Ensuite, les règles $r3$ et $r4$ sont essayées dans ce nouveau contexte mais ne sont pas applicables car elles n'identifient aucun élément dans l'arbre source. La règle $r5$ est essayée, elle identifie un sous-arbre dans l'arbre source comprenant l'élément "person" représentant le fils "Paul" de "Julie". Elle identifie d'autre part le sous-arbre de l'arbre cible comprenant l'élément "woman" qui a un attribut de valeur "Julie". La règle $r5$ est donc applicable ; à l'issu de son application le document cible devient le suivant :

```
<woman name="Julie">
    <sons>
      <man name="Paul">
        <sons/>
        <daugthers/>
    <sons/>
    <daugthers/>
</woman>
```

Ensuite, la règle $r6$ est essayée dans ce contexte, et ainsi de suite. La figure 6.4 donne la séquence d'exécution complète de la transformation. Une séquence d'exécution peut être vue comme un chemin dans l'arbre des choix des règles à essayer. La figure fait apparaître cet arbre avec les règles essayées et les règles effectivement appliquées.

Au final, lorsque la transformation s'achève car plus aucune règle n'est applicable, l'instance cible générée est la suivante :

```
<woman name="Julie">
    <sons>
        <man name="Paul">
            <sons/>
            <daugters/>
        </man>
    </sons>
    <daugthers>
        <woman name="Dominique">
            <sons>
                <man name="Bertrand">
                    <sons/>
                    <daugthers/>
                </man>
            </sons>
            <daugters/>
        </woman>
    </daugters>
</woman>
```

## 6.5 Conservation de l’historique d’exécution

Lorsqu’une transformation est exécutée, nous proposons de conserver l’historique d’exécution en sauvegardant l’ordre et le nom des règles appliquées.

La figure 6.5 illustre l’historique d’exécution de la transformation de l’exemple précédent.

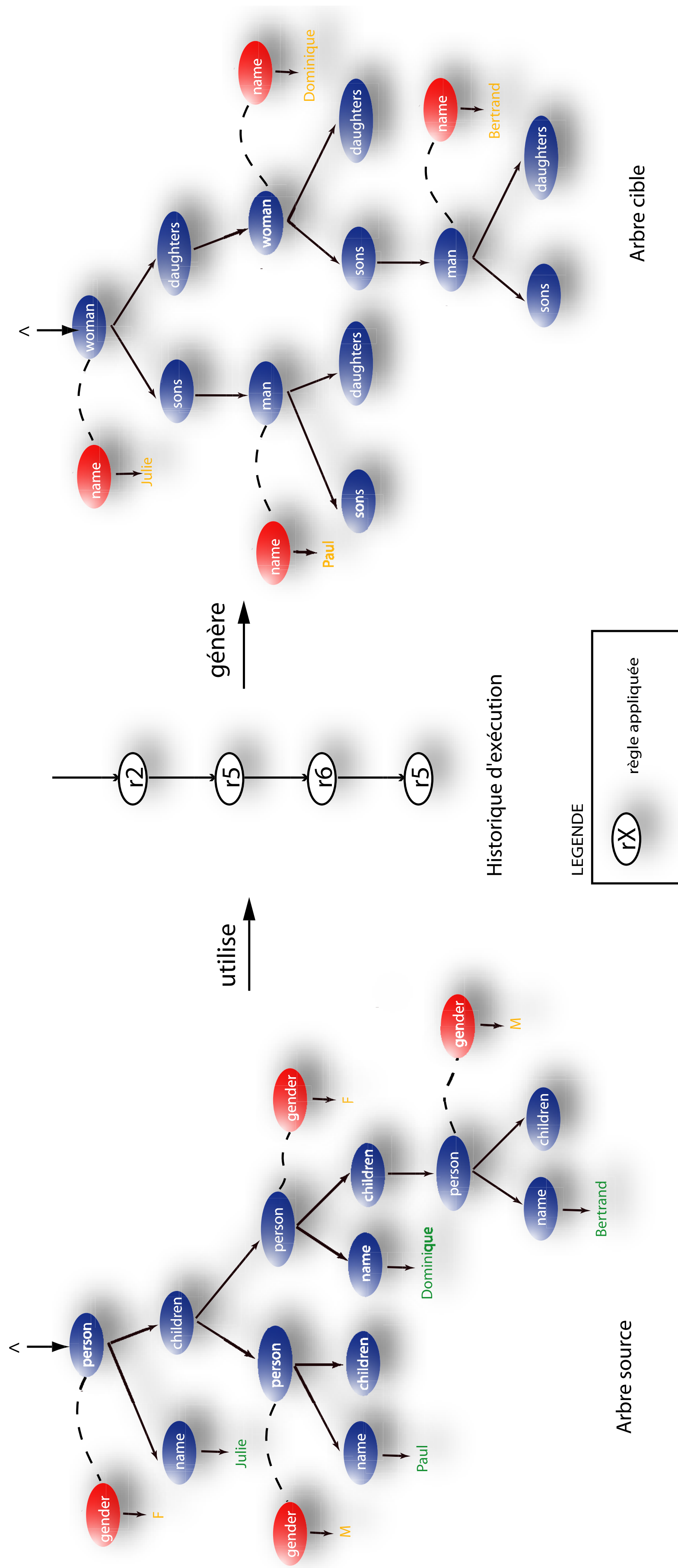


FIG. 6.5 – Historique d'exécution

## 6.6 Restriction du modèle

### 6.6.1 Opérateurs de génération

La majorité des opérateurs du sous-ensemble de XPath considéré se transpose bien dans une interprétation générative. Le sens de la plupart des opérateurs XPath interprété ainsi est assez intuitif : par exemple l'opérateur $/$ ("fils") construit un élément fils, $@$ construit un noeud attribut, etc. Dans la suite, l'opérateur descendant "//" est autorisé dans une expression $r'$ seulement s'il fait partie du noyau commun de $r$ et $r'$ (c.f. 6.2.2). Considérer l'opérateur descendant dans une expression $r'$ sans qu'il appartienne au noyau de $r$ et $r'$ nécessite d'envisager un ensemble de solutions possibles pour une même règle de transformation. En pratique, nous nous contenterons de l'opérateur "fils" ("/") qui est un cas particulier de l'opérateur descendant, suffisant pour décrire la génération de l'arbre cible.

### 6.6.2 Optique purement générative

Nous avons décrit l'opération de génération dans l'arbre cible dans toute sa généralité : elle peut générer ou supprimer des noeuds dans l'arbre cible, comme l'illustre la figure 6.2. Dans la suite, pour simplifier notre modèle en vue de la conservation de l'historique d'exécution, seule la génération de noeuds est tolérée au sein des règles de transformation ; la suppression est écartée. De ce fait, l'application des règles construit l'arbre cible de manière monotone. Du point de vue des applications cette restriction paraît raisonnable. A titre de comparaison, la norme XSLT construit elle-aussi l'arbre cible de manière monotone. Cette simplification permet d'autre part de conserver une certaine simplicité de notre modèle.

## 6.7 Répercution des modifications du document source

Nous avons introduit le modèle de transformation que nous proposons en vue de la transformation incrémentale, et détaillé l'exécution d'une transformation dans ce modèle sur un exemple.

Nous cherchons à présent à bâtir un algorithme incrémental pour la transformation de documents XML, sur la base du modèle que nous avons proposé à cet effet.

Nous considérons qu'une première exécution complète d'une transformation a été effectuée dans notre modèle. L'historique d'exécution est donc disponible. Cette section analyse les conséquences d'une modification du document source à répercuter sur le document cible, de manière à ce qu'il soit mis à jour incrémentalement.

Cette section débute par une modélisation des modifications du document source. Elle considère ensuite les conséquences de chaque type de modification. L'étude menée sur les répercutions de ces modifications identifie des problèmes et propose quelques extensions du modèle d'exécution initialement proposé en 6.3.

### 6.7.1 Modélisation des modifications du document source

#### Modifications élémentaires

Nous considérons que les modifications élémentaires suivantes peuvent survenir sur le document source d'une transformation :

– l'insertion d'un noeud feuille (élément, attribut, ou texte)

– la suppression d'un noeud feuille (élément, attribut, ou texte)
– la modification de la valeur d'un noeud texte (et donc en particulier de la valeur d'un attribut)

Notons que suppression et ajout de sous-arbres se ramènent à une suite de suppressions ou d'ajouts de feuilles (voir 6.7.1).

Dans un arbre XML, un noeud peut être identifié par son contexte. Nous proposons de caractériser les modifications élémentaires du document source en exprimant d'une part le contexte, le nom et le cas échéant la valeur du noeud modifié ; et d'autre part l'opération qui a été effectuée sur ce noeud.

**Contexte, nom et valeur du noeud modifié** Nous proposons d'identifier le noeud modifié à l'aide d'une expression XPath absolue renvoyant un singleton.

Seul un sous-ensemble de XPath est nécessaire à cet effet. Il faut tout d'abord pouvoir décrire le chemin depuis la racine du document jusqu'au noeud modifié. A cette fin, les axes fils ($/$) et attribut ($@$) sont nécessaires. Afin de pouvoir identifier un noeud texte il faut considérer le test de type de noeud $text()$. De plus, pour identifier le noeud de manière unique dans les cas où le document source possède plusieurs noeuds frères de même type, les prédicats de position parmi les noeuds frère sont nécessaires. [1] Enfin, pour décrire la valeur du noeud modifié s'il s'agit d'un noeud texte ou d'un noeud attribut, les prédicats d'égalité sur la valeur chaîne du noeud courant sont nécessaires.

Par exemple,

$$\wedge/html/body/table[2]/tr[1]/td[3]/text()[. = "toto"]$$

est l'expression caractérisant une opération sur le contenu de la troisième cellule de la première rangée d'un tableau dans une page HTML.

**Opération effectuée** Une expression absolue construite avec ce jeu d'opérateurs permet d'identifier le noeud qui a été modifié. En revanche, elle ne suffit pas pour décrire l'opération qui a été effectuée sur ce noeud (insertion, suppression, ou modification). Nous proposons de préciser l'opération effectuée avec un symbole apposé devant l'expression : $+$ pour l'insertion, $-$ pour la suppression et $*$ pour la modification.

Par exemple, $+\wedge/html/body/table[2]/tr[1]/td[3]$ décrit l'insertion d'une cellule dans un tableau. $*\wedge/html/body/@background[.="\#FFFFFF"]$ décrit une modification de la valeur de l'attribut "background" du corps d'une page HTML.

### Modifications

**Modélisation** Nous proposons de considérer n'importe quelle modification du document source comme un ensemble de modifications élémentaires effectuées dans un certain ordre. Dans notre modèle, nous considérons ainsi les modifications apportées au document source comme des listes ordonnées de modifications élémentaires.

**Validité des modifications** Dans notre modèle, nous considérons toutes les modifications pouvant survenir sur un document source. Dans le cas où un modèle de document est disponible pour le document source, la charge de vérifier si une modification conserve la validité du document source par rapport à son modèle est laissée à l'utilisateur.

[1] Notons que le sous-ensemble de XPath considéré pour les expressions des règles de transformation dans notre modèle ne suffit pas puisqu'il ne comporte pas les prédicats de position parmi les noeuds frère.

### 6.7.2 Répercution d'une insertion dans l'arbre source

L'insertion d'un noeud dans le document source impose de considérer deux aspects. D'une part, lorsqu'une insertion survient, des règles précédemment appliquées pourraient ne plus être applicables du fait de l'insertion. D'autre part, des règles pourraient devenir applicables suite à l'insertion du noeud.

Ces deux aspects soulèvent deux problèmes qui font l'objet des sections suivantes :
- la détermination des règles à remettre en cause ;
- la détermination des règles qui deviennent applicables

#### Règles remises en cause

Nous cherchons tout d'abord à déterminer les règles à remettre en cause suite à l'insertion d'un noeud. Remarquons que toutes les règles déjà appliquées ne sont pas forcément à remettre en cause. En effet, l'insertion d'un noeud attribut dans l'arbre source ne risque pas par exemple de remettre en cause l'application d'une règle qui ne référence aucun attribut.

Nous cherchons donc tout d'abord à caractériser et à isoler les règles dont l'application ne doit pas être remise en cause. Pour cela, nous cherchons *dans quelle mesure il est possible de spécifier des règles dont l'application ne sera jamais remise en question* suite à l'insertion d'un noeud dans l'arbre source.

Répondre à cette question nécessite l'étude des relations structurelles sur lesquelles sont construites les expressions de matching dans l'arbre source.

L'analyse de ces relations permet de les diviser en deux catégories :
- les relations qui restent valables après une insertion, que nous nommons relations *non perturbées* ;
- les relations qui n'existent plus à cause de l'insertion, que nous nommons relations *perturbées*

Ces catégories sont détaillées ci-après.

**Relations non perturbées** Les propriétés ou relations suivantes, qui existaient sur les noeuds présents avant l'insertion, ne sont pas remises en cause par l'insertion d'un noeud feuille (attribut, élément ou texte) :
- le nom d'un noeud (par exemple : p, article)
- le type d'un noeud (text(), node(), etc.)
- les relations entre les noeuds : la relation fils, descendant, attribut et toutes les autres relations considérées dans notre modèle (ancestor, ancestor-or-self, descendant-or-self, following, following-sibling, namespace, parent, self, preceding, preceding-sibling)
- la relation union (|)
- les prédicats ([]) comportant exclusivement les relations précédentes

Ces propriétés et relations qui existaient entre les noeuds de l'arbre source avant l'insertion existent toujours sur le nouvel arbre source modifié par l'insertion.

Par conséquent, les règles qui ont été appliquées et dont l'expression $l$ comporte exclusivement ces relations ne doivent pas être remises en cause.

**Relations perturbées** Dans le modèle de document que nous considérons, le contenu textuel d'un élément est défini comme la concaténation des valeurs de tous ses noeuds texte descendants. Le contenu textuel d'un élément est donc affecté par l'insertion d'un

noeud texte descendant. En conséquence, les opérations suivantes effectuées avant l'insertion d'un noeud texte ne sont plus pertinentes :
- le test d'égalité sur le contenu linéarisé
- l'extraction de contenu linéarisé

Toutes les règles appliquées qui comportent au moins une de ces opérations sur un élément ancêtre du noeud texte inséré doivent être remises en cause.

Par exemple, une règle comportant une expression $l$ de la forme $person[name = "Julie"]$ ou $person[name = \$n]$ doit être remise en cause lorsqu'un noeud texte est inséré comme descendant de l'élément $name$.

Notons que la valeur d'un attribut ne fait pas partie du contenu linéarisé d'un élément. Nous considérons qu'un attribut est inséré avec une valeur. L'insertion d'un noeud texte fils d'un noeud attribut est donc considérée comme une modification de la valeur du noeud texte (c.f. 6.7.3).

#### Extension du modèle

L'étude des relations structurelles a permis d'identifier quelles sont les règles appliquées qui doivent être remises en cause. Nous cherchons maintenant à défaire ces règles. Notre modèle nécessite d'être étendu pour prendre en compte cet aspect. Nous proposons d'étendre notre modèle pour être capable de mettre à jour l'arbre cible. L'extension du modèle proposée repose sur deux techniques inspirées de l'état de l'art, qui font l'objet des sections suivantes :
- tout d'abord, nous proposons de mener une analyse statique de la transformation, c'est-à-dire une analyse effectuée avant la première exécution de la transformation ;
- d'autre part, nous proposons de maintenir des informations auxiliaires au cours de l'exécution de la transformation ; des informations supplémentaires sont maintenues afin de pouvoir réaliser l'incrément de transformation nécessaire

**Analyse statique de la transformation** Le but de l'analyse statique est de repérer les règles dont l'application pourrait être remise en cause suite à une insertion, c'est-à-dire les règles qui comportent des relations perturbées.

Pour cela, les règles sont analysées statiquement afin de savoir si elles comportent au moins un test d'égalité de contenu linéarisé, ou une extraction de contenu linéarisé. Le cas échéant elles sont dites *repérées*.

**Informations auxiliaires** Nous proposons d'étendre le modèle d'exécution pour qu'il établisse les informations auxiliaires suivantes :
- **Elément source lié à la règle appliquée** Au cours de l'exécution de la transformation, nous nous plaçons au moment où une règle identifie un sous-arbre dans l'arbre source. Si la règle est repérée, nous proposons de créer un pointeur sur celle-ci dans la séquence d'exécution. Ce pointeur est associé à chaque élément de l'arbre source dont le contenu linéarisé est testé ou extrait par la règle. Notons que le pointeur est associé à un élément uniquement si aucun autre pointeur vers une règle repérée précédemment appliquée ne lui a déjà été associé. On garantit ainsi que la première application d'une règle repérée, dans l'ordre de la séquence d'exécution est pointée. La figure 6.6 illustre ces pointeurs (en vert) dans le cas de la transformation de l'exemple précédent. Sur cette figure, un pointeur vert lie un élément à la première règle repérée qui a été appliquée et qui traite son contenu linéarisé.

- **Règle appliquée liée aux noeuds générés**
  Lors de l'application d'une règle repérée, un ensemble de pointeurs est associé à la règle appliquée. Ces pointeurs indiquent chaque noeud que la règle a généré dans l'arbre cible. Ceci permet de détruire ultérieurement les noeuds qu'une règle a généré.

La figure 6.6 illustre les informations auxiliaires qui sont établies lors de l'exécution de la transformation étudiée dans l'exemple précédent (en section 6.4). Dans ce cas, toutes les règles ont été repérées puisqu'elles comportent toutes au moins une relation perturbée.

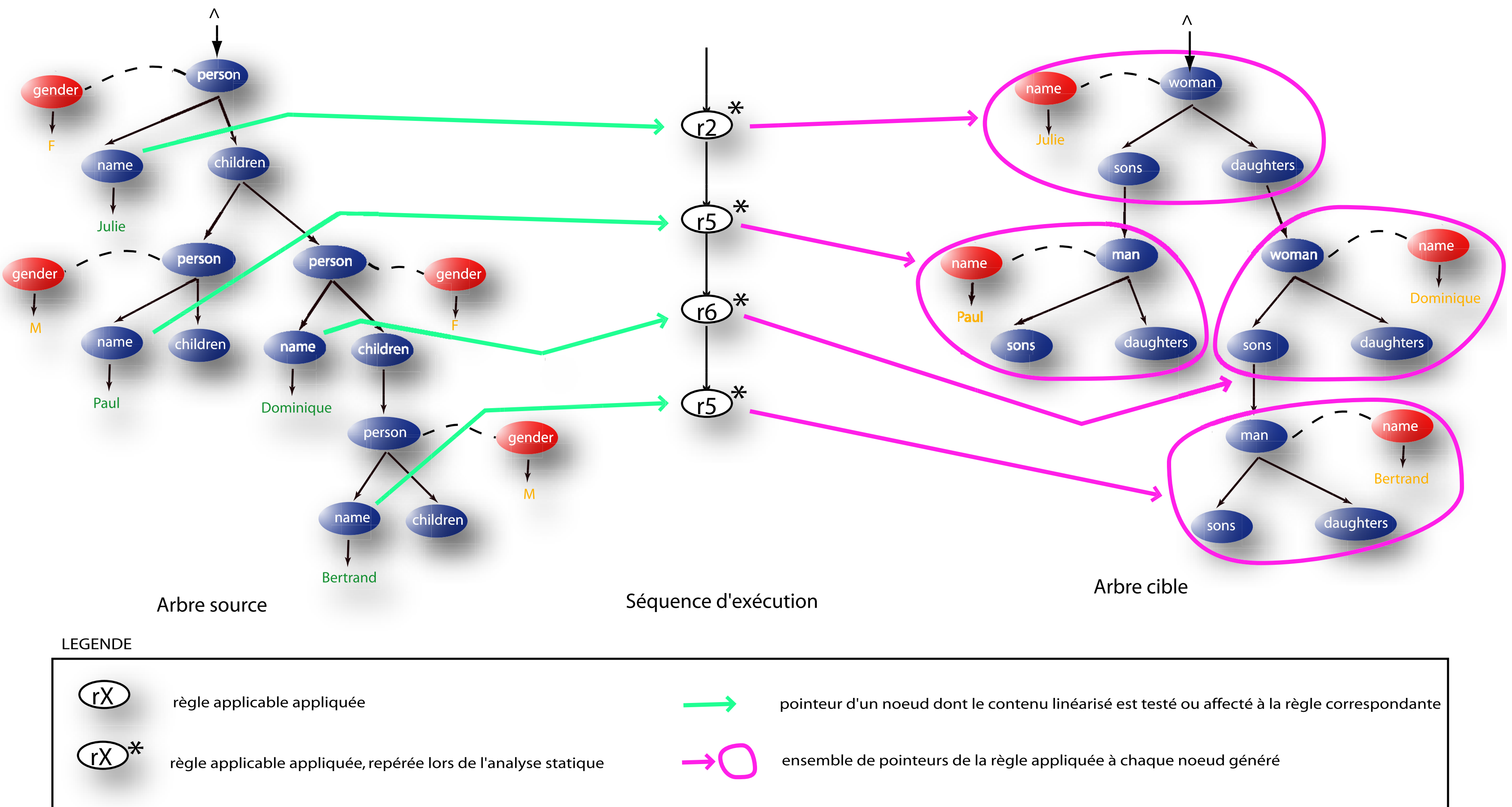


FIG. 6.6 – Informations auxiliaires

Avec ces informations auxiliaires nous sommes en mesure de remettre en question les règles adéquates. L'algorithme de réaction à l'insertion d'un noeud texte proposé en 6.7.2 emploie ces informations auxiliaires. Cependant, remettre en question les règles adéquates ne suffit pas, l'insertion d'un noeud peut aussi déclencher l'application de nouvelles règles devenues applicables.

#### Règles devenues applicables

Nous avons traité le cas des règles à remettre en cause suite à l'insertion d'un noeud texte. Nous cherchons maintenant à déterminer les règles devenues applicables suite à l'insertion d'un noeud (élément, attribut ou texte). Une règle traitant directement l'élément inséré peut notamment devenir applicable. Mais d'autres règles traitant d'autres éléments peuvent également devenir applicables. Par exemple, une règle avec une expression $l$ de la forme $person[@gender="F"]$ qui n'était pas applicable à un élément $person$ sans attribut $gender$ devient applicable lors de l'insertion de l'attribut $gender="F"$. Ainsi, a priori, plusieurs règles traitant différents noeuds de l'arbre source peuvent devenir applicables suite à l'insertion d'un seul noeud.

**Intérêt du modèle d'exécution proposé** Notre modèle de transformation repose sur le principe d'exécution "tant qu'une règle est applicable, elle est appliquée". Ainsi, les nouvelles règles devenues applicables sont naturellement envisagées par le modèle. Le même modèle d'exécution reste donc valable après qu'un noeud a été inséré dans l'arbre source, pour exécuter les nouvelles règles qui deviennent applicables.

#### Synthèse

Pour prendre en compte l'insertion d'un noeud (élément, attribut ou texte) dans l'arbre source, il faut appliquer toutes les règles devenues applicables, ce qui est naturellement supporté par notre modèle d'exécution. De plus, dans le cas de l'insertion d'un noeud texte, des règles appliquées peuvent être remises en cause. Nous avons déterminé lesquelles, et avons étendu notre modèle de manière à ce qu'il établisse des informations auxiliaires qui rendent possible cette remise en cause. Nous proposons donc deux algorithmes pour traiter incrémentalement l'insertion d'un noeud dans l'arbre source.

**Algorithme de réaction à l'insertion d'éléments ou d'attributs** Pour traiter l'insertion d'un élément ou d'un attribut et de sa valeur, nous proposons de réitérer la boucle qui essaie les règles, en repartant de la dernière règle appliquée. Avec le modèle d'exécution proposé, ceci se résume à l'appel suivant dans lequel "d" est l'indice de la dernière règle appliquée dans la séquence d'exécution :

```
// Algorithme A1
essayer_regles(d+1)
```

**Algorithme de réaction à l'insertion d'un noeud texte** Avec les informations auxiliaires que nous avons établies en 6.7.2, nous sommes en mesure de proposer un algorithme pour modifier incrémentalement le document cible lors de l'insertion d'un noeud texte dans l'arbre source.

Puisque l'arbre cible est construit de manière monotone (c.f 6.6.2), l'idée est de reprendre la construction de l'arbre cible dans l'état où il était avant l'application de la première règle remise en cause.

Lorsqu'un noeud texte est inséré, tous les contenus linéarisés de ses éléments ancêtres sont affectés. Il faut donc prendre en compte toutes ces références implicites.

Nous proposons de déterminer quelle est la première règle appliquée qui a traité le contenu linéarisé d'un des ancêtres du noeud inséré.

L'application de cette règle est remise en cause, et la génération de l'arbre cible reprend à partir de cet instant. L'algorithme est donné en 6.7.

### 6.7.3 Répercution d'une modification d'un noeud texte

#### Modification d'un noeud texte fils d'un élément

La modification d'un noeud texte affecte le contenu linéarisé des éléments ancêtres. Comme dans le cas de l'insertion d'un noeud texte, le test et l'extraction de contenu linéarisé sont deux relations perturbées. Les conséquences d'une modification de la valeur d'un noeud texte fils d'un élément dans l'arbre source sont ainsi similaires à celles d'une insertion d'un noeud texte dans l'arbre source. Dans ce cas, nous proposons d'employer la même technique que celle proposée en réaction d'une l'insertion de noeud texte (c.f. 6.7.2), en considérant non plus le noeud inséré, mais le noeud modifié.

#### Modification de la valeur d'un attribut

Lorsqu'un noeud texte fils d'un attribut est modifié, les conséquences sont similaires. La différence avec la modification d'un noeud texte est que la modification de la valeur d'un attribut n'affecte pas le contenu linéarisé des éléments ancêtres. Seule la valeur du noeud attribut est modifiée, et cette modification ne se propage pas sur les ancêtres du noeud attribut.

Nous proposons un algorithme en réaction à la modification de la valeur d'un attribut qui est une version simplifiée de l'algorithme en réaction à la modification/insertion d'un noeud texte. L'idée est que les noeuds ancêtres du noeud attribut n'ont pas besoin d'être testés pour déterminer la première règle à remettre en cause. Par conséquent le seul changement par rapport à l'algorithme $A2$ est que l'appel à la fonction "text_prem_pos()" dans $A2$ peut être remplacé par l'appel à une fonction simplifiée "att_prem_pos()", donnée sur la figure 6.8.

### 6.7.4 Répercution d'une suppression dans l'arbre source

Lorsqu'un noeud de l'arbre source est supprimé, toutes les règles appliquées à cause de l'existence de ce noeud sont remises en cause. Pour prendre en compte cet aspect, nous proposons d'étendre notre modèle en conservant de nouvelles informations auxiliaires.

#### Informations auxiliaires supplémentaires

Nous proposons de généraliser les informations auxiliaires conservées dans le cas des règles repérées à toutes les règles, pour être en mesure de mettre à jour le document cible lors d'une suppression de l'arbre source.

Lorsqu'un noeud de l'arbre source est identifié par une règle, nous proposons de conserver un pointeur de cet élément vers la règle appliquée.

```
// n : le noeud texte inséré dans l'arbre source
// p : un noeud de l'arbre source
// d : indice de la dernière règle appliquée
// lastPos, firstPos, currentPos sont des entiers dénotant des positions dans la séquence d'exécution
// rar : position de la première règle à remettre en cause dans la séquence d'exécution

//fonction trouvantla position de l'éventuelle première règle à remettre en cause :
DebutFonction text_prem_pos(n, lastPos)
    p ← n
    firstPos ← lastPos+1

    Tant que (p !=∧)
        Si (un pointeur est associé à p)
            currentPos ← position de la règle pointée dans la séquence d'exécution
            Si (currentPos<firstPos)
                firstPos ← currentPos
            FinSi
        FinSi
        p ← parent(p)
    FinTantque

    Retourner(firstPos)
FinFonction

d ← indice de la dernière règle appliquée
lastPos ← position de la dernière règle dans la séquence d'exécution
n ← noeud texte inséré
rar ← text_prem_pos(n, lastPos)
Si (rar<lastPos+1)
    détruire la séquence d'exécution à partir de la position rar comprise
        // (tous les noeuds correspondants dans l'arbre cible sont détruits,
        // ainsi que les informations auxiliaires correspondantes de l'arbre source)
    d ← indice de la règle appliquée à la position rar dans la séquence d'exécution
FinSi

essayer_regles(d)
```

FIG. 6.7 – Algorithme A2 : réaction à l'insertion d'un noeud texte

//fonction trouvantla position de l'éventuelle première règle à remettre en cause :
**DebutFonction** att_prem_pos(n, lastPos)
  p ← n
  firstPos ← lastPos+1

  **Si**(un pointeur est associé à p)
    currentPos ← position de la règle pointée dans la séquence d'exécution
    **Si** (currentPos<firstPos)
      firstPos ← currentPos
  **FinSi**
  Retourner(firstPos)
**FinFonction**

FIG. 6.8 – Simplification dans le cas d'une valeur d'attribut

| Modifications élémentaires | Remise en cause possible de règles appliquées | Nouvelles règles potentiellement applicables |
|---|---|---|
| **Insertion** | | |
| d'un élément | non | oui |
| d'un attribut et de sa valeur | non | oui |
| d'un noeud texte fils d'un élément | oui | oui |

FIG. 6.9 – Tableau récapitulatif

**Traitement de la suppression**

L'idée est de trouver la première règle à remettre en cause. Nous proposons de la déterminer en utilisant le pointeur établi sur chaque élément transformé. Si la présence de l'élément supprimé a été utilisé par une règle, cette règle est à remettre en cause. Nous proposons de détruire les conséquences de l'application de cette règle dans l'arbre cible, et de reprendre sa construction à partir de cet état.

Notons qu'avec notre proposition, les éléments ou attributs de l'arbre source peuvent chacun posséder deux pointeurs. Un pointeur peut exister sur une règle appliquée car cette règle utilise la présence du noeud ; un autre pointeur peut exister sur une règle qui utilise le contenu linéarisé de l'élément ou la valeur de l'attribut. Dans ce cas, nous proposons de déterminer la première règle qui a été appliquée parmi les deux pointées, en se référant à leur position dans la séquence d'exécution.

### 6.7.5 Récapitulatif

Le tableau de la figure 6.9 récapitule les implications de l'insertion dans notre modèle.

Notons que lorsqu'une règle est à remettre en cause et que nous défaisons l'historique d'exécution pour le reconstruire, de nouvelles règles sont évidemment applicables. Aussi, la modification et la suppression de noeuds, comme l'insertion d'un noeud texte, peuvent remettre en cause des règles appliquées et donc déclencher l'application de nouvelles règles.

### 6.7.6 Traitement de listes de modifications élémentaires

Dans notre modèle, une modification du document source est une liste ordonnée de modifications élémentaires (c.f. section 6.7.1). Ainsi, l'insertion d'un sous-arbre $\Gamma$ dans le document source est décomposée en la liste ordonnée des insertions de chaque noeud de $\Gamma$.

Nous souhaitons minimiser les calculs à effectuer pour mettre à jour le document cible lorsque plusieurs noeuds sont insérés d'un coup dans l'arbre source. Nous avons vu dans notre précédent modèle pour la traduction incrémentale (c.f. annexe A) que réagir après chaque modification élémentaire nuit à la performance. Nous cherchons donc à réagir seulement après que plusieurs noeuds ont été insérés dans l'arbre source.

Le principe du modèle d'exécution que nous avons proposé permet de réagir à un ensemble d'insertions d'éléments ou d'attributs.

Certaines modifications élémentaires entraînent seulement l'application de nouvelles règles. D'autres modifications élémentaires nécessitent de remettre en cause des règles. Dans ce cas, nous avons proposé de trouver la première règle à remettre en cause, de détruire ses conséquences et de rebâtir l'arbre cible en relançant l'application des règles.

**Liste d'insertions et de modifications** Nous considérons à présent une liste ordonnée de modifications élémentaires, consituée uniquement d'insertions (d'élément, d' attribut et de sa valeur, ou bien de noeud texte fils d'un élément) et de modifications (de la valeur d'un attribut ou d'un noeud texte). Nous proposons de traiter incrémentalement l'ensemble de ces modifications élémentaires. Le traitement incrémental proposé a lieu après que le document source a été effectivement modifié. Notons que dans un premier temps nous écartons les suppressions, qui nécessitent que le traitement incrémental soit effectué avant que les noeuds ne soient effectivement supprimés de l'arbre source. Une piste de poursuite du travail consisterait à déterminer dans quelle mesure il serait possible de traiter efficacement un ensemble de modifications élémentaires comportant des suppressions.

Pour traiter une liste composée d'insertions et de modifications, nous proposons d'abord de traiter le cas des modifications élémentaires qui peuvent remettre en cause des règles, et ensuite seulement de réappliquer les règles. Plus précisément, nous proposons de remettre en cause la première règle de l'historique d'exécution qui est concernée. Pour cela, nous proposons de déterminer, pour chaque modification élémentaire de la liste, la position dans la séquence d'exécution de l'éventuelle règle remise en cause. De ces règles à remettre en cause, nous ne retenons que celle de position minimale dans la séquence d'exécution. Nous la remettons en cause en détruisant ce qu'elle a généré dans l'arbre cible, et la suite de l'historique d'exécution. Nous relançons alors l'application des règles pour achever la transformation. La figure 6.10 donne l'esquisse de l'algorithme.

```
// l : liste de modifications élémentaires
// m : modification élémentaire
// d : indice de la dernière règle appliquée
// lastPos, currentPos sont des entiers dénotant des positions dans la séquence d'exécution

Si (non liste_vide(l))
d ← indice de la dernière règle appliquée
lastPos ← position de la dernière application de règle dans la séquence d'exécution

//trouve l'éventuelle première règle à remettre en cause dans la séquence d'exécution :
Tantque non liste_vide(l)
    m <- prem(l) // premier élément de la liste
    l <- succ(l) // liste privée de son premier élément
    n ← noeud_modifié(m)
    Selon (m)
        insertion ou modification d'un noeud texte fils d'un élément :
            currentPos ← text_prem_pos(n, lastPos)
        modification d'une valeur d'attribut :
            currentPos ← att_prem_pos(n, lastPos)
    FinSelon
    Si (currentPos<firstPos)
        firstPos ← currentPos
    FinSi
FinTantque
Si (firstPos !=lastPos+1)
    détruire la séquence d'exécution à partir de la position firstPos comprise
        // (tous les noeuds correspondants dans l'arbre cible sont détruits,
        // ainsi que les informations auxiliaires correspondantes de l'arbre source)
    d ← indice de la règle appliquée à la position firstPos dans la séquence d'exécution
FinSi

essayer_regles(d)
FinSi
```

FIG. 6.10 – Algorithme A4 : réaction à une liste d'insertions et de modifications

### 6.7.7 Exemples

**Ajout d'un sous-arbre à l'arbre source**

Supposons qu'il faille représenter une nouvelle personne dans le document source. Cette personne est un fils de "Julie" et se nomme "Jean". L'ajout de cette personne modifie le document source qui devient le suivant :

```
<person gender="F">
    <name>Julie</name>
    <children>
        <person gender="M">
            <name>Paul</name>
            <children/>
        </person>
        <person gender="M">
            <name>Jean</name>
            <children/>
        </person>
        <person gender="F">
            <name>Dominique</name>
            <children>
                <person gender="M">
                    <name>Bertrand</name>
                    <children/>
                </person>
            </children>
        </person>
    </children>
</person>
```

Dans notre modèle, cette modification se traduit par la liste de modifications élémentaires suivante :

$L = (+ \wedge /person[name = "Julie"]/children/person[2],$

$+ \wedge /person[name = "Julie"]/children/person[2]/@gender[. = "M"],$

$+ \wedge /person[name = "Julie"]/children/person[2]/name,$

$+ \wedge /person[name = "Julie"]/children/person[2]/name/text()[. = "Jean"],$

$+ \wedge /person[name = "Julie"]/children/person[2]/children)$

L'algorithme $A4$ est appliqué. Parmi ces modifications élémentaires, seule la quatrième (insertion d'un noeud texte) pourrait a priori remettre en cause des règles appliquées dans l'historique. La figure 6.11 illustre le fonctionnement de notre proposition sur cet exemple. Les ancêtres du noeud texte inséré sont parcourus (étapes n°1, 2, 3 et 4 sur la figure). Aucun ne comporte de pointeur vers une règle appliquée. Dans ce cas, aucune règle n'est à remettre en cause, puisque le contenu linéarisé des éléments

ancêtres n'est pas utilisé. L'exécution de la transformation peut donc se poursuivre en continuant l'application des règles depuis la dernière règle appliquée. Ainsi, la règle $r5$ est appliquée (étape n°5) qui génère un sous-arbre dans l'arbre cible. Le document cible de la transformation est ainsi mis à jour incrémentalement.

**Gain de performance sur l'exemple** Notons que pour cet exemple, le coût de la mise à jour incrémentale est constitué du coût $C_1$ de la remontée dans l'arbre source, et du coût $C_2$ de l'application des nouvelles règles applicables (r5 en l'occurrence). Nous considérons que $C_1$ est négligeable. En pratique, ceci a du sens car de nombreux documents de très grande taille sont en réalité des documents en "en râteau", i.e. des arbres très larges mais relativement peu profonds(voir l'exemple typique des documents utilisés par le NASDAQ dans le chapitre 5). Ainsi, sur cet exemple, le coût de la mise à jour incrémentale est réduit au coût d'application des seules règles devenues applicables. Dans ce cas, notre solution est donc clairement beaucoup plus performante que la réexécution complète de la transformation. D'une manière plus générale, nous subodorons que notre modèle est performant pour le traitement incrémental de l'ajout de sous-arbres dans l'arbre source. Comme nous le verrons, une piste de poursuite de notre travail consiste à faire une analyse complète de performance.

### Modification d'une valeur d'attribut

Supposons maintenant qu'une erreur se soit glissée dans le document source de la transformation : la personne de nom "Dominique" n'est pas une femme mais un homme. La figure 6.12 illustre la réaction de notre modèle suite à la modification de la valeur de l'attribut $gender = "F"$ en $gender = "M"$.

Cet exemple illustre le cas où une légère modification du document source peut avoir de lourdes conséquences sur l'arbre cible. Ceci est intrinsèquement lié à la transformation.

Avec notre solution, l'attribut dont la valeur a été modifiée est considéré (étape n°1 sur la figure). Celui-ci a un pointeur vers une règle appliquée dans l'historique d'exécution. La séquence d'exécution est détruite à partir de cette position (étape n°2). Tous les noeuds générés dans l'arbre cible par l'application des règles détruites (r6 et r5) sont également détruits. Ensuite, en repartant de la dernière position dans la séquence d'exécution, les nouvelles règles sont appliquées (étape n°3). Le document cible est ainsi mis à jour incrémentalement.

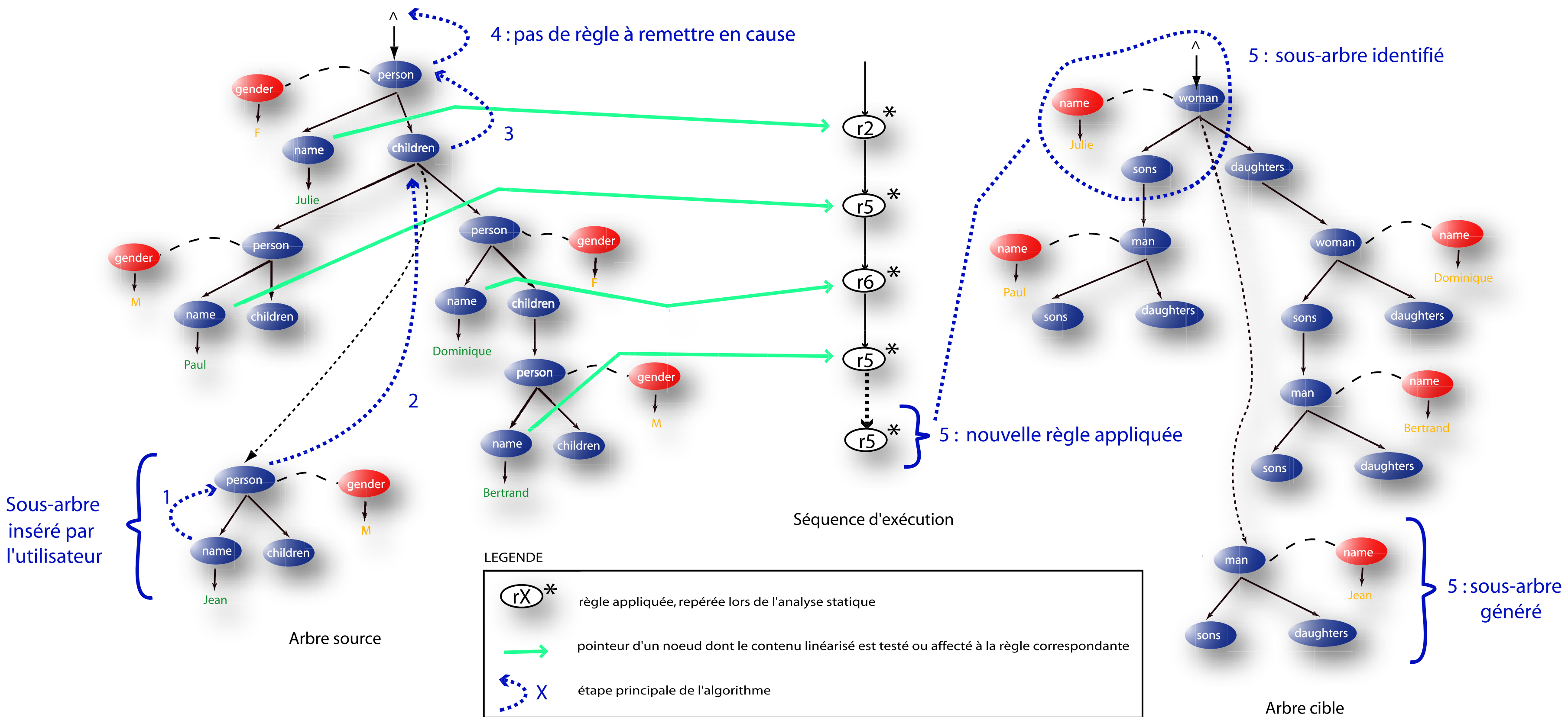


FIG. 6.11 – Exemple : ajout d'un sous-arbre dans l'arbre source

Valeur d'attribut modifiée de "F" en"M" par l'utilisateur

1 : règle à remettre en cause

2 : destruction de l'historique d'exécution

3 : nouvelles règles appliquées

3 : sous-arbre généré

Arbre source

Séquence d'exécution

Arbre cible

LEGENDE

| | |
|---|---|
| rX* | règle appliquée, repérée lors de l'analyse statique |
| → (vert) | pointeur d'un attribut dont la valeur est testée ou affectée à la règle correspondante |
| X (bleu) | étape principale de l'algorithme |
| → (rose) | ensemble de pointeurs de la règle appliquée à chaque noeud généré |
| X (rouge) | position à partir de laquelle la séquence d'exécution est détruite |

FIG. 6.12 – Exemple : modification d'une valeur d'attribut

## 6.8 Synthèse

### 6.8.1 Evaluation du modèle

#### Pertinence du modèle par rapport aux critères dégagés

**Expressivité adaptée** Notre modèle fournit la possibilité d'extraire des informations du document source grâce à l'affectation de variable permettant l'extraction de contenu linéarisé, de valeur d'attribut, et de noeud texte. De plus, l'expression de tests de structure est possible avec l'égalité comme dans XPath classique. Enfin, notre opération de génération permet de générer des fragments dans le document cible. Notons que la génération peut s'effectuer selon le contexte dans l'arbre source et selon le contexte dans l'arbre cible. Nous considérons donc que l'objectif fixé est atteint.

**Règles de transformation de granularité fine** Dans notre modèle, la granularité d'une règle peut être limitée par une description trop importante du contexte dans l'arbre source. Cependant, il y a des cas où la transformation elle-même nécessite une description importante du contexte dans l'arbre source ; dans ce cas, l'expression de règles de granularité fine est tout simplement impossible. Ceci n'incombe pas à notre modèle mais aux besoins de transformation. Nous n'avons pas identifié d'autres obstacles à la description de règles de granularité fine dans notre modèle que les besoins intrinsèquement liés à la transformation. Nous considérons donc que notre modèle permet de spécifier des règles de transformation de granularité fine lorsque la transformation le permet.

**Règles de transformation indépendantes** Dans notre modèle, deux règles peuvent être liées par le contexte de l'arbre cible. Une règle $R_1$ peut générer un nouveau contexte dans l'arbre cible qui est utilisé par une autre règle $R_2$. Ainsi, tant que $R_1$ n'a pas été appliquée, $R_2$ n'est pas applicable ; et si l'application de $R_1$ est remise en question, l'application de $R_2$ peut l'être aussi. Ainsi, dans notre modèle, les règles peuvent ne pas être indépendantes.

En revanche, notons que lorsque deux règles utilisent le même contexte dans l'arbre cible, celles-ci sont indépendantes. L'ordre d'application de ces règles n'est donc pas important. Considérons par exemple les deux premières règles de l'exemple que nous avons présenté en 6.4. Seul le modèle d'exécution définit leur ordre d'application et par conséquent l'ordre des sous-structures générées dans l'arbre cible. On pourrait donc considérer que ces deux règles sont fortement indépendantes car elles peuvent être appliquées dans n'importe quel ordre, tout en produisant un document cible conservant la même sémantique.

Au final, notre modèle permet l'expression de règles indépendantes. Pour spécifier des règles indépendantes, le programmeur doit simplement veiller à contrôler les dépendances entre les règles lorsqu'il spécifie le contexte dans l'arbre cible (l'expression $r$) de chaque règle.

**Facilité de la décomposition** Puisque notre modèle repose sur un système de règles, nous pensons qu'il facilite la décomposition de la transformation du point de vue du programmeur ; du moins, il impose au programmeur de considérer la transformation comme un ensemble de règles de transformation.

**Conservation possible de l'historique d'exécution** Notre modèle n'engendre pas d'effets de bords car les variables sont locales à une règle de transformation. Nous avons vu en 6.5 que la conservation de l'historique d'exécution était possible.

**Approche par différence finie** Suivant la définition de l'approche par différence finie (c.f notre chapitre sur le traitement incrémental), nous avons tenté de construire, pour chaque modification élémentaire considérée, la fonction correspondante qui mettait à jour l'arbre cible. Une des pistes de poursuite du travail consiste à effectuer l'analyse de performance pour déterminer dans quels cas pratiques notre proposition s'avère viable. En effet, dans certains cas, notre proposition pour traiter la suppression peut par exemple remettre en cause une des toutes premières règles appliquées. Dans ce cas, un problème de performance se pose : vaut-il mieux traiter la suppression de cette façon ou réexécuter totalement la transformation ? Nous avons certes opté pour une approche par différence finie, mais il serait nécessaire d'effectuer une analyse de performance qui permettrait de confirmer ou de mettre en défaut ce critère.

### 6.8.2 Limites du modèle

**Verrous à lever**

**Les prédicats de position** Notre modèle ne permet pas d'exprimer des prédicats de position dans les règles de transformation. Ceci limite les transformations qu'il est possible d'exprimer dans notre modèle.

Tout d'abord, les expressions $l$ et $r$ ne peuvent pas faire intervenir la position parmi les noeuds frères.

De plus, une règle de transformation ne peut qu'ajouter dans l'arbre cible, et non insérer à une position donnée parmi les noeuds frère. Ceci soulève le problème de l'ordre des sous-structures générées dans l'arbre cible. D'une part, ceci force le programmeur à écrire lui même les règles dans un certain ordre, qui dépend du modèle d'exécution, afin que les sous-structure générées dans l'arbre cible soient ordonnées de la manière souhaitée lors de la première transformation. D'autre part, en l'état actuel, les modifications incrémentales ne prennent pas en compte l'ordre dans l'arbre cible. Ce problème ne ressort pas dans l'exemple que nous avons présenté. Cependant, dans le cas général l'ordre des sous-structures de l'arbre cible importe.

Nous avons ainsi identifié l'ajout des prédicats de position comme le premier verrou à lever pour étendre notre modèle.

**Ambiguïtés de contexte dans l'arbre cible** Un autre problème identifié est lié à la manière dont nous spécifions un contexte dans l'arbre cible, qui peut amener des ambiguïtés. Le problème provient du fait que la spécification d'un contexte dans l'arbre cible peut identifier plusieurs sous-arbres de l'arbre cible. Or, généralement, le programmeur souhaite considérer un seul de ces sous-arbres, et en l'état actuel notre modèle ne permet pas toujours de l'exprimer.

Par exemple, nous ne sommes pas capables d'exprimer des transformations comme la traduction d'un arbre d'éléments "a" en arbre d'éléments "b". Considérons par exemple un arbre d'éléments de nom "a" et les deux règles suivantes :

$$r_1 : \wedge/a, \wedge \longrightarrow \wedge/b$$

$$r_2 : \wedge//a/a, \wedge//b \longrightarrow \wedge//b/b$$

La façon dont nous spécifions le contexte dans l'arbre cible au sein de la règle $r_2$ est insuffisante dans notre modèle d'exécution. En effet, à chaque fois que $r_2$ est appliquée, le contexte identifié par $r$ reste le même (le premier sous-arbre identifié par $\wedge//b$). Or nous souhaitons nous référer à l'élément "b" correspondant à la traduction du premier élément "a" dans $\wedge//a/a$.

Notons que dans l'exemple que nous avons présenté nous n'avons pas eu ce problème puisque le nom d'une personne l'identifie de manière unique.

Nous pensons qu'une piste possible serait d'enrichir le modèle d'exécution de manière à ce qu'une règle puisse s'appliquer à un contexte dans l'arbre cible qui est défini par le résultat de la règle précédemment appliquée. Nous avons commencé à explorer cette piste sur d'autres modèles d'exécution et nous sommes convaincu de son intérêt dans le cadre de ce modèle.

### Pistes restant à étudier

**Sémantique de l'opération de génération** Il faudrait définir formellement la sémantique de l'opération de génération que nous employons dans notre modèle, c'est-à-dire la réécriture d'expressions XPath en partie droite de règle. Notons que des problèmes sont à prévoir lors de l'extension du modèle pour prendre en compte les prédicats de position. En particulier, en supposant que notre modèle supporte les prédicats de position, quelle sémantique conférer à une partie droite de la forme $a/b[position() = 2] \longrightarrow a/b[position() = 4]$ ? Ce cas pose un problème si l'élément "a" n'a que deux fils "b" : doit-on générer de nouveaux éléments ? L'étude de la sémantique de l'opération de génération est ainsi la principale piste restant à étudier avant d'étendre le modèle.

# Chapitre 7

# Conclusion

## 7.1 Rappel des objectifs

Nous avons abordé dans ce mémoire le problème de la transformation incrémentale des structures XML. L'énoncé du problème peut être synthétisé ainsi : étant donné un document XML et une transformation, après que le document cible a été produit, le document source peut subir des modifications ; nous cherchons alors à mettre à jour le document cible de manière optimale.

Ce problème d'optimisation s'inscrit dans un contexte industriel où l'utilisation de documents de grande taille est d'actualité. Une des applications de la transformation incrémentale est par exemple la mise à jour d'importants sites Web, pour lesquels il serait naturel de pouvoir transformer des documents XML en HTML de manière incrémentale.

Ce problème s'inscrit également dans un domaine scientifique en plein essor qui a pour but d'améliorer la performance des transformations de manière à pouvoir concevoir de nouvelles applications, dynamiques.

Dans cette perspective, nous nous sommes fixé comme objectif de comprendre les procédés qui pourraient permettre un traitement incrémental efficace.

## 7.2 Rappel du travail réalisé

Nous avons tout d'abord étudié les langages de transformation XML actuels, et cerné leurs limites. Tous ont été conçus dans une optique "batch" où la totalité du document source est transformée d'un coup.

Afin d'être en mesure de faire des propositions pour contourner cette limite, nous avons étudié et effectué un travail de synthèse sur l'état de l'art du traitement incrémental.

Celui-ci nous a permis d'identifier quelques éléments favorables pour qu'un modèle de transformation se prête à la transformation incrémentale. Nous avons alors cherché à proposer un modèle de transformation en réponse à ces critères.

A ce stade, une nouvelle proposition pour caractériser la génération dans l'arbre cible (en utilisant XPath) a été émise dans l'équipe. Nous avons contribué à explorer les possibilités offertes par cette nouvelle approche, en concevant et étudiant différents modèles d'exécution (i.e. des procédés pour exécuter les règles de transformation).

Nous avons ainsi eu la chance de pouvoir prendre part aux premières réflexions sur la conception d'un nouveau langage de transformation dans l'équipe de recherche WAM.

Après avoir perçu l'ampleur de ces travaux, nous avons finalement conçu et isolé un modèle de transformation original et spécialisé en réponse à nos critères pour la transformation incrémentale. En particulier, nous avons cherché à bâtir un modèle d'exécution dans l'optique de la transformation incrémentale.

Nous avons compris qu'un tel modèle doit prendre en compte le fait qu'une transformation XML n'est pas limitée à une seule exécution complète sur un document source immuable. En réponse à ce besoin, nous avons proposé d'employer un modèle d'exécution similaire à celui d'un système de réécriture. Pour cela, nous avons proposé des règles de transformation reposant sur une sémantique de matching de sous-arbres aussi bien dans l'arbre source que dans l'arbre cible de la transformation.

Avec notre modèle de transformation, nous avons également proposé une modélisation des modifications qui peuvent survenir sur le document source.

Sur cette base, nous avons proposé des algorithmes permettant de traiter incrémentalement des modifications apportées au document source.

Les principaux résultats de notre travail sont l'analyse des problèmes soulevés par la transformation incrémentale ; et l'élaboration d'un modèle naissant qui se prête tout particulièrement au traitement incrémental de l'insertion de noeuds dans l'arbre source.

## 7.3 Evaluation

Notre approche est sensiblement différente de celle développée dans incXSLT. Cette dernière visait à rendre incrémental un processeur de transformation XSLT existant et non incrémental.

Notre approche consiste au contraire à explorer les possibilités offertes par la conception d'un nouveau modèle de transformation, pensé dès le départ dans l'optique de la transformation incrémentale.

Nous avons ainsi tenté de construire un modèle qui facilite l'exécution directe des règles détectées comme étant à appliquer suite à la modification du document source. En particulier, restaurer le contexte d'application d'une règle est aisé dans notre modèle.

Cependant, en l'état actuel, notre modèle demeure sévèrement limité par l'absence des prédicats de position dans les règles. Tant que ce verrou n'est pas levé, la génération dans l'arbre cible est restreinte à des ajouts de sous-structures en dernière position parmi les sous-structures soeurs, dans l'ordre de l'arbre cible. Il n'est pas possible d'effectuer de véritables insertions à une position parmi les sous-structures soeurs qui serait déterminée par le programmeur. L'expressivité du modèle en est réduite d'autant. L'ajout des prédicats de position parmi les noeuds frères de même type, tel que défini par la norme XPath, figure parmi les pistes de poursuite du travail.

Malgré ses limites actuelles, nous pensons que notre modèle pourrait être applicable et efficace pour effectuer certaines transformations de manière incrémentale. Nous pensons en particulier aux documents XML de grande taille, créés par ajout successifs de sous-structures soeurs (selon un modèle de document dit "en râteau"). Les documents utilisés par le NASDAQ pour représenter les cotations de titres boursiers à l'année (c.f. chapitre 5) en sont un exemple.

Enfin, notons que dans notre modèle, certaines modifications élémentaires comme la suppression de noeuds, peuvent nécessiter de remettre en cause la majeure partie de l'historique d'exécution. Des cas extrêmes sont donc à prévoir où les algorithmes

proposés ne sont pas plus efficaces que la réexécution complète de la transformation. Aussi, un complément de ce travail serait une partie complète de détermination des cas où les algorithmes mis en place sont significativement plus efficaces que la réexécution complète de la transformation.

## 7.4 Perspectives

Les perspectives à court terme de ce travail de DEA sont synthétisées ci-après.

**Analyse de performance** Il est nécessaire de réaliser une évaluation complète du modèle, de façon à déterminer précisément les cas où les techniques mises en places (en particulier le modèle d'exécution proposé et les algorithmes incrémentaux proposés) sont significativement plus efficaces que la réexécution de la transformation dans sa globalité.

**Formalisation de la sémantique de génération** Il est nécessaire de définir précisément la sémantique de l'opération de génération (la réécriture d'expressions XPath en partie droite de règle) sur laquelle notre modèle repose, et aussi sans doute le nouveau langage en cours de conception dans l'équipe de recherche WAM.

**Ajout des prédicats de position** Nous avons identifié un verrou à lever dans notre modèle : il faudrait l'étendre de façon qu'il puisse prendre en compte les prédicats de positions parmi les noeuds frères de même type. Cette extension permettrait d'effectuer de véritables insertions dans le document cible et non de simples ajouts. En revanche, elle pose de nouveaux problèmes. En particulier, un prédicat de position parmi les noeuds frère est une propriété qui peut être perturbée par l'insertion d'un noeud. De plus, ajouter les prédicats de position nécessite d'étendre notre opération de génération afin de prendre en compte les nouveaux aspects liés aux positions. Ceci débouche sur les autres pistes de poursuite du travail.

A plus ou moins long terme, les principales perspectives de ce travail concernent l'analyse statique de la transformation, de façon à optimiser le traitement incrémental.

**Analyse statique des dépendances entre règles** Nous pensons en effet que faire apparaître explicitement le contexte dans l'arbre cible qui est utilisé par une règle est un atout. Cet atout, conjugué à notre modèle d'exécution à base de matching de sous-arbres, ouvre la voie à une analyse statique des dépendances entre les règles.

Dans un travail ultérieur, nous aimerions ainsi nous concentrer sur une analyse statique des règles de la transformation, qui pourrait notamment servir à élaborer une structure comme un graphe de dépendance des règles. Nous pensons que cette approche, éventuellement couplée avec une approche employant un test d'inclusion pour déterminer les règles qui nécessitent d'être appliquées, est prometteuse. Elle permettrait sans doute l'élaboration de nouveaux algorithmes incrémentaux plus efficaces pour gérer notamment le cas des modifications élémentaires qui remettent en cause une grande partie de l'historique d'exécution dans notre modèle actuel.

**Transformation en streaming** La transformation en streaming consiste à émettre le document source et à commencer à produire l'arbre cible de la transformation alors que l'arbre source n'est pas encore complètement connu car partiellement transmis. Les noeuds de l'arbre source sont reçus dans l'ordre de la forme sérialisée (c'est-à-dire dans l'ordre du parcours en profondeur d'abord de l'arbre source). La transformation commence à produire les parties de l'arbre cible qu'il est possible d'engendrer au fur et à mesure que la structure de l'arbre source est transmise. La taille des structures transmises sont telles qu'il n'est pas possible de stocker la totalité de l'arbre source depuis le début de la transmission.

Mis à part cette spécificité importante, il est possible de relier la transformation incrémentale à la transformation en streaming. En effet, on pourrait considérer que la transformation en streaming est un cas particulier de la transformation incrémentale. Le cas du streaming correspond à un modèle de transformation incrémentale dans lequel la seule modification élémentaire de l'arbre source considérée est l'insertion d'un noeud feuille. De plus, cette insertion ne se fait que dans l'ordre de l'arbre source, correspondant à la transmission de la forme sérialisée.

Nous pensons donc qu'une piste intéressante serait d'étudier si ce modèle peut être viable dans le cas de la transformation en streaming d'arbres de profondeur finie. Nous donnons ci-après quelques éléments qui expliquent nos motivations.

Tout d'abord, dans le modèle que nous avons proposé, le document cible peut être mis à jour incrémentalement lors de l'insertion d'un noeud dans l'arbre source. Dans le cas du streaming, la réception de l'arbre source pourrait être décrite par une liste d'insertions de noeuds feuille (c.f. section 6.7.1). Au fur et à mesure que les noeuds sont reçus, de nouvelles règles deviennent applicables. Notre modèle d'exécution permet de réagir en mettant à jour le document cible.

Nous avons vu que dans ce modèle seule l'insertion d'un noeud texte peut remettre en cause des règles appliquées. Dans le cas de la transformation en streaming, il serait possible de mettre en attente l'application d'une règle traitant du contenu linéarisé jusqu'à ce que tous les noeuds texte descendants soient connus. De cette manière, l'application d'une règle ne serait jamais remise en question.

Tous les noeuds descendants d'un noeud $n$ sont connus lorsque le prochain frère de $n$ dans l'ordre du document source est reçu. Une analyse statique de la transformation déterminerait toujours les règles comportant des tests ou des extractions de contenu linéarisé. A l'exécution, lorsqu'un noeud de l'arbre source est reçu, son éventuel noeud frère qui le précède dans l'ordre du document est marqué comme candidat pour l'application des règles identifiées lors de l'analyse statique. Pour cela, le noeud de l'arbre source peut être décoré d'un symbole "*" qui indique que tous ses descendants ont été reçus. Les règles comportant des tests ou des extractions de contenu linéarisé ne peuvent pas s'appliquer tant que les noeuds concernés ne sont pas décorés par le symbole "*". L'application des règles qui ne comportent pas de tests ni d'extraction de contenu linéarisé n'est pas concernée par la mise en attente. Par rapport au modèle décrit précédemment, une condition supplémentaire est ajoutée pour qu'une règle $(N, R, \phi)$ soit applicable : si la règle N fait partie des règles repérées à l'analyse statique, dans le sous-arbre source identifié, chaque noeud dont le contenu linéarisé est traité doit être décoré par une *, indiquant que tous ses descendants ont été reçus. Dans le cas contraire, la règle n'est pas applicable.

Le principal problème qui reste est spécifique à la transformation en streaming et est lié au fait que l'arbre source ne peut pas être stocké dans sa globalité. Les axes inverses (c'est-à-dire les axes qui décrivent un chemin remontant dans l'arbre) devraient sans doute être bannis au moins au sein des expressions $l$. Sur la base de notre modèle,

nous pensons qu'une piste intéressante serait d'étudier comment le modèle d'exécution et l'algorithme de matching dans l'arbre source pourraient être adaptés au cas du streaming. En particulier, une piste à explorer serait le stockage temporaire de certaines parties de l'arbre source, et leur effacement au fur et à mesure que l'algorithme de matching identifie des sous-arbres.

**Extension de l'expressivité et de l'utilisabilité** De façon à étendre notre modèle pour le rendre à la fois plus expressif et surtout plus utilisable en se plaçant du point de vue du programmeur, une piste consisterait à étendre le modèle d'exécution que nous avons proposé à l'aide de stratégies. Une idée de Jean-Yves Vion-Dury est de laisser la possibilité au programmeur de définir des stratégies d'application des règles. Nous avons étudié cette idée sur d'autres modèles d'exécution, en collaboration avec l'équipe de recherche. Le principe est que le programmeur pourrait lui-même spécifier un ordre d'application des règles, pour contrôler plus facilement la transformation. Nous pensons que cette idée pourrait être aisément appliquée à notre modèle à base de matching. Enfin, dans l'optique de la transformation incrémentale, nous pensons que ce principe pourrait être vu comme une restriction des dépendances de règles à considérer.

# Bibliographie

## Bibliographie


[1] M. Abadi et al, *"Analysis and caching of dependencies"*, ACM SigPlan International Conference on Functional Programming, Philadelphia, May 1996, pp 83-91.

[2] Umut A. Acar, Guy E. Blelloch, Robert Harper, *"Adaptive Functional Programming"*, 16 January 2002, CMU-CS-01-161, School of Computer Science, Carnegie Mellon University, Pittsburgh

[3] Extase Akpotsui, *"Transformation de types dans les systèmes d'édition de documents structurés"*, Thèse en Informatique, INPG, Octobre 1993

[4] F.E. Allen, J. Cocke, and K. Kennedy, "*Reduction of operator strength*". In S.S. Muchnick and N.D. Jones, editors, Program Flow Analysis, pages 79-101. Prentice Hall, Englewood Cliffs, N.J., 1981.

[5] V. Benzaken, G. Castagna, and A. Frisch, *"CDuce : a white paper"*, Workshop PLAN-X : Programming Language Technologies for XML, Pittsburgh PA, Oct. 2002.

[6] Stéphane Bonhomme, *"Transformations de documents structurés : une combinaison des approches déclaratives et automatiques"*, Thèse en Informatique, Université Joseph Fourier, Décembre 1998.

[7] Alan Carle and Lori Pollock, "*Modular Specification of Incremental Program Transformation Systems*", ACM 1989

[8] E. W. Dijkstra. "*A Discipline of Programming*", Prentice Hall Series in Automatic Computation, Prentice Hall, Englewood Cliffs, N.J., 1976.

[9] Hartmut Ehrig, Martin Korff, Michael Löwe, *"Tutorial Introduction to the Algebraic Approach of Graph Grammars Based on Double and Single Pushouts"*, In H.-J. Kreowski H. Ehrig and G. Rozenberg, editors, Graph Grammars and Their Application to Computer Science, LNCS 532. Springer-Verlag, 1991.

[10] John Field and T. Teitelbaum,*"Incremental reduction in the lambda calculus"*, In Proc. ACM Conference on Lisp and Functional Programming, Nice, France, Juin 1990, pp. 307-322

[11] John Field, *"A simple rewriting semantics for realistic imperative programs and its application to program analysis"*, Technical report, IBM T.J. Watson Research Center, May 1992

[12] *"Interactively Editing Structured Documents"*, R. Furuta, V. Quint, J. André, , Electronic Publishing - Origination, Dissemination and Design , 1(1), pp. 19-44, April 1988.

[13] R. Furuta et P. David Stotts, *"Specifying Structured Document Transformations"*, Document Manipulation and Typography, In Proceedings of the International COnference on Electronic Publishing, Cambridge University Press, Nice, 20-22 Avril 1988.

[14] P. Genevès, *"Une relecture de DSD : a schema language for XML"*, document internet, 2003, http ://wam.inrialpes.fr/people/geneves/review_dsd.html

[15] C.F. Goldfarb, *"A generalized approach to document markup"*, SIGPLAN Notices of the ACM, June 1981.

[16] R. Hoover, "*Alphonse : incremental computation as a programming abstraction*". In proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation, pages 261-272. ACM, New York, June 1992.

[17] H. Hosoya and B. C. Pierce, "XDuce : a statically typed XML processing language", ACM Transactions on Internet Technology 3(2) :117-148, 2003.

[18] Steven D. Johnson, Yanhong A. Liu, Yuchen Zhang, "*A systematic Incrementalization Technique and its Application to Hardware Design*", technical report 524, June 1999

[19] Claude Kirchner, Zhebin Qian, Preet Kamal Singh and Jürgen Stuber, *"Xemantics : a Rewriting Calculus-Based Semantics of XSLT"*, LORIA, Rapport de recherche A01-R-386, 2001

[20] Eila Kuikka et Martti Penttonen, *"Transformation of structured documents"*, Electronic Publishing - Origination, Dissemination and Design, Vol. 8, N°4, pp. 319-341, 1995

[21] Nabil Layaïda, *"Madeus : Système d'édition et de présentation de documents structurés multimédia"*, Thèse en informatique, Université Joseph Fourier, juin 1997.

[22] Greger Lindén, "*Incremental updates in structured documents*", Thèse en informatique, Université d'Helsinki, 5 Février 1994

[23] Yanhong A. Liu et Tim Teitelbaum, "*Systematic Derivation of Incremental Programs*", Science of Computer Programming, Vol. 24, N°1, pp.1-30, Février 1995

[24] Yanhong A. Liu, "*Incremental computation : a semantics-based systematic transformational approach*", Ph'd thesis, 1996

[25] Yahnhong A. liu, *"Efficient Computation via Incremental Computation"*, Pacific-Asia Conference on Knowledge Discovery and Data Mining, 1999

[26] Yanhong A. Liu, Scott D. Stoller, Tim Teitelbaum, "*Strengthening Invariants for Efficient Computation*", 2001

[27] Dongwon Lee and Wesley W. Chu, *"A comparative Analysis of Six XML Schema Languages"*, Department of Computer Science, university of California, ACM Sigmod Record, September 2000

[28] S. A. Mamrak, M. J. Kaelbling, C. K. nicholas et M. Share, *"Chameleon : A System for Solving the Data-Translation Problem"*, IEEE Transactions on Software Engineering, pp. 1090-1108, Septembre 1998

[29] D. Michie, *"Memo functions and machine learning"*, Nature, 218 :19-22, 1968

[30] Tova Milo, Dan Suciu, Victor Vianu, *"Typechecking for XML Transformers"*, In Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, 2000

[31] Markus L. Noga, Steffen Schott, Welf Löwe, "*Lazy XML Processing*",DocEng'02, November 8-9, 2002, McLean, Virginia, USA

[32] G. Miklau and D. Suciu, *"Containment and equivalence for an XPath fragment"*, In 21st ACM Symposium on Principles of Databases Systems (PODS), Madison, Wisconsin, pages 65-76, 2002.

[33] R. Paige and S. Koenig, "*Finite differencing of computable expressions*", ACM Trans. on Program. Lang. Syst., 4(3) :402-454, July 1982.

[34] R. Paige, "*Programming with invariants*", IEEE Software, pages 56-69, Jan. 1986.

[35] W. Pugh and T. Teitelbaum. "*Incremental computation via function caching*". In Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages, pages 315-328. ACM, New York, Jan 1989.

[36] Vincent Quint, *"Multimédia sur le Web : XML, SMIL, SVG"*, cours donné à l'École Centrale de Lyon le 24 mars 2003.

[37] G. Ramalingam and Thomas Reps, "*A categorized Bibliography on Incremental Computation*", 1993

[38] G. Ramalingam and Thomas Reps, "*An incremental Algorithm for Maintaining the Dominator Tree of a Reducible Flowgraph*", ACM, 1994

[39] Thomas W. Reps and Louis B. Rall, "*Computational Divided Differencing and Divided-Difference Arithmetics*", Kluwer Academic Publishers, 2000

[40] T. Reps, T. Teitelbaum, and A. Demers. Incremental context-dependent analysis for language-based editors. ACM Trans. Program. Lang. Syst., 5(3) :449-477, July 1983.

[41] T. Reps and T. Teitelbaum, "*The Synthesizer Generator : a System for Constructing Language-Based Editors*", Springer-Verlag, New York, 1988.

[42] Cécile Roisin, *"Documents multimédia structurés"*, Habilitation à diriger des recherches, spécialité informatique, Institut National Polytechnique de Grenoble, Septembre 1999

[43] R. S. Sundaresh and P. Hudak, "*Incremental computation via partial evaluation*", in Conference Record of the 18th Annual ACM Symposium on Principles of Programming Languages, pages 1-13. ACM, New York, Jan. 1991.

[44] Emma Anna Van Der Meulen, "*Incremental rewriting*", Thèse en informatique, Université d'Amsterdam, 1994

[45] Lionel Villard, Nabil Layaïda, "*iXSLT : An Incremental XSLT Transformation Processor for XML Document Manipulation*", WWW 2002, Hawaï, 2002

[46] Lionel Villard, *"Modèles de documents pour l'édition et l'adaptation de présentations multimédias"*, Thèse en informatique, Institut National Polytechnique de Grenoble, 21 Mars 2002

[47] Jean-Yves Vion-Dury, Veronika Lux, Emmanuel Pietriga, *"Experimenting with the Circus Language for XML Modeling and Transformation"*, ACM Symposium on Document Engineering, McLean (Virginia, USA), November 2002

[48] J.-Y. Vion-Dury, Nabil Layaïda, *"Containment of XPath expressions : an Inference and Rewriting based approach"*, à paraître dans Extreme Markup Languages 2003

[49] Phil Wadler, "Two Semantics for XPath", Document Internet, http ://www.research.avayalabs.com/user/wadler/papers/xpath-semantics/xpath-semantics.pdf, 2000

[50] D. Walker, D. Petitpierre and S. Armstrong, *"XMLTrans : a Java-based XML Transformation Language for Structured Data"*, In Proc. of the 18th International Conference on Computational Linguistic, pp. 1131-1135, Vol N°2, 2000

[51] Daniel Yellin and Robert Strom, "*INC : A Language for Incremental Computations*", ACM Transactions on Programming Languages and Systems, Vol. 13, N°2, April 1991, Pages 211-236.

### Présentation des langages de transformation XML du point de vue de l'utilisateur

[52] *"The XSL Companion"*, Neil Bradley, Addison-Wesley, 2000, ISBN 0-201-67487-4

[53] *"XQuery : an XML query langage"*, D. Chamberlin, IBM Systems Journal, vol 41, n°4 2002

[54] *"The Circus-DTE tutorial"*, David Ramsey, document internet : http ://www.alphaavenue.com/staging/Xerox/Circus-DTE/tutorial/tutorial.pdf

[55] *"CDuce user's manual"*, Site Web : http ://www.cduce.org/manual.html

[56] *"Guide to OmniMark 5"*, Omnimark, 2001, http ://www.omnimark.com

### Systèmes

[57] *"ChordML DTD"*, http ://www.cifranet.org/xml/chordml.dtd

[58] *"Music Markup Language (MML)"*, Jacques Steyn, University of Pretoria, Jan. 5, 2000

[59] *"MusicML"*, The Connection Factory, 1999, http ://195.108.47.160/3.0/musicml/

[60] *"Simple API for XML (SAX)"*, Site Web : http ://www.saxproject.org/

### Organisations

[61] *"The World Wide Web Consortium (W3C)"*, http ://www.w3.org/

[62] *"Projet de recherche Web Adaptation et Multimédia (WAM)"*, INRIA Rhône-alpes, http ://wam.inrialpes.fr

### Normes et recommandations

[63] *"Document Object Model (DOM) Level 1 Specification"*, W3C Recommendation, 1 October 1998, http ://www.w3.org/TR/REC-DOM-Level-1/

[64] *"Mathematical Markup Language (MathML) Version 2.0"*, W3C Recommendation 21 February 2001, http ://www.w3.org/TR/MathML2/

[65] *"Synchronized Multimedia Integration Language (SMIL 2.0)"*, W3C Recommendation, 07 August 2001, http ://www.w3.org/TR/smil20/

[66] *"Scalable Vector Graphics (SVG) 1.1 Specification"*, W3C Recommendation, 14 January 2003 http ://www.w3.org/TR/SVG11/

[67] *"XHTML$^{TM}$ 1.0 The Extensible HyperText Markup Language (Second Edition)"*, A Reformulation of HTML 4 in XML 1.0, W3C Recommendation 26 January 2000, revised 1 August 2002, http ://www.w3.org/TR/xhtml1/

[68] *"Extensible Markup Language (XML) 1.0 (Second Edition)"*, Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, W3C Recommandation, 6 Octobre 2000, http ://www.w3.org/TR/REC-xml

[69] *"XML Schema Part 1 : Structures"*, W3C Recommendation, 2 May 2001, http ://www.w3.org/TR/xmlschema-1/

[70] *"XML Path Language (XPath)"*, James Clark and Steve DeRose, W3C Recommandation, 16 November 1999, http ://www.w3.org/TR/xpath

[71] *"XML Path Language XPath) version 2.0"*, A. Berglund, S. Boag, D. Chamberlin, M. F. Fernandez, M. Kay, J. Robie, J. Siméon, W3C Working Draft, 15 November 2002, http ://www.w3.org/TR/xpath20

[72] *"XQuery 1.0 : An XML Query Language "*, S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, J. Siméon, W3C Working Draft, 15 November 2002, http ://www.w3.org/TR/xquery/

[73] *"XQuery 1.0 and XPath 2.0 Data Model"*, W3C Working Draft, 2 May 2003, http ://www.w3.org/TR/xpath-datamodel/

[74] *"XSL Transformations (XSLT)"*, James Clark, W3C Recommandation, 16 November 1999, http ://www.w3.org/TR/xslt

[75] *"XSL Transformations (XSLT) version 2.0"*, Michael Kay, W3C Working Draft, 16 November 2002, http ://www.w3.org/TR/xslt20

# Table des figures


2.1 Forme linéarisée . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
2.2 Forme arborescente . . . . . . . . . . . . . . . . . . . . . . . . . . 15
2.3 Processus de transformation . . . . . . . . . . . . . . . . . . . . . . 17

3.1 Exemple d'une expression XPath . . . . . . . . . . . . . . . . . . . . 22
3.2 Exemple de sélection . . . . . . . . . . . . . . . . . . . . . . . . . . 23
3.3 Navigation avec XPath 1.0 . . . . . . . . . . . . . . . . . . . . . . . 24
3.4 Tableau récapitulatif . . . . . . . . . . . . . . . . . . . . . . . . . . 29

6.1 Sous-arbre identifié et noeud sélectionné . . . . . . . . . . . . . . . . 53
6.2 Opération de génération . . . . . . . . . . . . . . . . . . . . . . . . 55
6.3 Algorithme d'application des règles . . . . . . . . . . . . . . . . . . 60
6.4 Séquence d'exécution . . . . . . . . . . . . . . . . . . . . . . . . . . 63
6.5 Historique d'exécution . . . . . . . . . . . . . . . . . . . . . . . . . 66
6.6 Informations auxiliaires . . . . . . . . . . . . . . . . . . . . . . . . 72
6.7 Algorithme A2 : réaction à l'insertion d'un noeud texte . . . . . . . . 75
6.8 Simplification dans le cas d'une valeur d'attribut . . . . . . . . . . . 76
6.9 Tableau récapitulatif . . . . . . . . . . . . . . . . . . . . . . . . . . 76
6.10 Algorithme A4 : réaction à une liste d'insertions et de modifications . 78
6.11 Exemple : ajout d'un sous-arbre dans l'arbre source . . . . . . . . . . 81
6.12 Exemple : modification d'une valeur d'attribut . . . . . . . . . . . . . 82

7.1 Algorithme de traitement d'une insertion . . . . . . . . . . . . . . . . 99

# Annexe A

## Un modèle pour la traduction incrémentale d'éléments

### Introduction

Nous proposons un premier modèle permettant la traduction incrémentale d'éléments. Le modèle de document considéré est celui défini par la norme XPath, qui a été présenté dans la première partie de ce mémoire. Cependant, dans le cadre de ce modèle, nous ne considérons que les éléments d'un document. Il serait facile d'étendre ce modèle pour traduire de manière similaire les attributs et les noeuds texte. Cependant, nous avons opté pour présenter uniquement la traduction d'éléments, qui suffit pour donner les idées principales liées au traitement incrémental dans ce modèle.

### Principe

Intuitivement, le principe est le suivant : à un nom d'élément "a" de l'arbre source correspond un nom d'élément potentiellement différent dans l'arbre cible, nommé "traduction de a". L'arbre source n'est pas modifié par la transformation. L'arbre cible est construit à partir de zéro.

#### Règles de traduction

L'association entre noms d'éléments s'effectue par l'intermédiaire de règles de la forme suivante : $a \rightarrow b$. Une telle règle signifie que pour un élément de nom "a" de l'arbre source un élément de nom "b" sera construit dans l'arbre cible.

A un nom d'élément de l'arbre source est associé au plus une règle de traduction.

#### Première traduction

L'arbre source est parcouru en profondeur d'abord. Pour chaque noeud rencontré, une règle de traduction correspondant au type du noeud est recherché. S'il en existe une, elle est appliquée.

**Informations auxiliaires** Lors de l'exécution d'une règle de traduction, le noeud de l'arbre source qui est traduit se voit associer un pointeur indiquant le noeud correspondant à sa traduction dans l'arbre cible.

#### Traductions ultérieures

**Modification du document source** Dans ce modèle, deux modifications élémentaires sont considérées comme pouvant survenir sur l'arbre source :

- L'insertion d'un élément, qui consiste à ajouter un élément dans l'arbre source comme une feuille avec une position déterminée parmi ses noeuds frères ;
- La suppression d'un élément qui consiste à supprimer une feuille de l'arbre source, à une certaine position parmi ses noeuds frères.

Ces deux modifications sont caractérisées par le contexte du noeud modifié, qui permet notamment d'identifier les noeuds ancêtres du noeud modifié ainsi que la position du noeud modifié parmi ses frères.

Toute modification du document source peut se décomposer en une liste d'opérations d'insertion et de suppression.

**Traitement incrémental d'une insertion** Grâce à la conservation de pointeurs entre un noeud de l'arbre source et sa traduction dans l'arbre cible, les opérations élémentaires peuvent être traitées incrémentalement. Lorsqu'un élément est ajouté dans l'arbre source, sa traduction dans l'arbre cible est obtenue en considérant l'unique règle associé au type de l'élément. Cette traduction est alors insérée dans l'arbre cible. La position d'insertion dans l'arbre cible est déterminée en utilisant le premier ancêtre de l'élément inséré qui possède une traduction dans l'arbre cible. La figure 7.1 donne l'algorithme qui réagit à l'insertion d'un noeud dans le document source en insérant sa traduction dans l'arbre cible. Cet algorithme est appliquée après l'insertion effective du noeud dans l'arbre source.

**Traitement incrémental d'une suppression** Lorsqu'un élément de l'arbre source est supprimé, son éventuelle traduction dans l'arbre cible doit être supprimée. Si l'élément n'a pas de traduction dans l'arbre cible, ce dernier est à jour. Dans le cas contraire, si l'élément supprimé a été traduit, il suffit simplement d'utiliser le lien établi lors de la traduction pour identifier le noeud de l'arbre cible à supprimer. Ce traitement est effectué avant la suppression effective d'un noeud de l'arbre source.

### Exemple

Un exemple de traduction d'éléments est la traduction des éléments représentant des rangées d'un tableau HTML en éléments représentant une liste HTML. Cette traduction peut s'exprimer dans ce modèle par les règles de traduction suivantes :

$$html \longrightarrow html$$

$$body \longrightarrow body$$

$$tr \longrightarrow ul$$

$$td \longrightarrow li$$

### Limites du modèle

Ce premier modèle rend possible la traduction incrémentale d'éléments. Il permet seulement d'exprimer une correspondance entre des noms d'éléments. Il ne permet

**Entrées :** ns est le noeud inséré dans l'arbre source, ps est sa position parmi ses frères (entier)
**Sorties :** l'arbre cible est modifié et les informations auxiliaires sont mises à jour
**Variables :**n̄c est le noeud de l'arbre cible, pc la position de b parmi ses frères (entier), pns un pointeur sur un noeud de l'arbre source, pnc un pointeur sur un noeud de l'arbre cible, et position_trouvee un booléen

**Si** (une règle de traduction est associée au type de l'élément ns)
//La règle de traduction est appliquée à ns ce qui donne un élément nc
nc $\leftarrow$ traduction(ns)
pc $\leftarrow$ ps

// Recherche de la position d'insertion de b dans l'arbre cible
pns $\leftarrow$ ns
position_trouvee $\leftarrow$ faux
**Tant que** pns a un parent et non position_trouvee
pns $\leftarrow$ parent(pns)
**Si** (pns a une traduction pnc dans l'arbre cible)
position_trouvee<-vrai
**Else**
pc $\leftarrow$ pc + nombre de frères de pns qui sont avant pns dans l'arbre source

// Mise à jour de l'arbre cible
**Si** (non position_trouvee)
nc devient la racine du document cible
**Sinon**
nc est inséré comme fils en pc-ième position de l'élément pnc

// Mise à jour des informations auxiliaires
Associer à ns un pointeur sur nc
**Finsi**

FIG. 7.1 – Algorithme de traitement d'une insertion

pas d'envisager des transformations de structures XML. Aussi, vis-à-vis de la transformation proprement dite, des limites considérables de ce premier modèle peuvent être identifiées. Elles sont classées ci-dessous en catégories :

– **Filtrage insuffisant**
  La traduction d'un élément ne dépend que de son type. Ceci limite énormément les traductions qu'il est possible d'exprimer. En effet, dans la plupart des cas, la traduction d'un élément ne dépend pas seulement de son type mais aussi de son contexte dans l'arbre source et de valeurs de la structure.
– **Génération insuffisante**
  A un noeud de l'arbre source ne peut correspondre qu'au plus un noeud de l'arbre cible. Or, pour chaque noeud de l'arbre source, on souhaiterait pouvoir générer des structures dans l'arbre cible plus élaborées qu'un simple noeud : des sous-arbres par exemple. De plus, ces structures devraient pouvoir être placées à différents endroits de l'arbre cible.
– **Réordonnement de la structure impossible**
  Nous définissons comme ordre sur les éléments d'un document, l'ordre déterminé par le parcours en profondeur d'abord des noeuds de l'arbre du document. Etant donné deux éléments e1 et e2 de l'arbre source tel que e1 soit avant e2, les traductions respectives de ces deux éléments t1 et t2 auront forcément le même ordre (t1 sera avant t2). En effet, la construction de l'arbre cible est monotone : l'arbre cible est construit à partir de zéro par insertions successives de traductions d'éléments. Or, ces traductions d'éléments sont obtenues dans l'ordre du parcours en profondeur d'abord de l'arbre source. Ce modèle ne permet donc pas de réordonner les éléments dans la structure.
– **Mauvaise performance de la première transformation**
  La première exécution de la transformation repose sur un parcours en profondeur d'abord du document source. Ceci peut être un handicap lorsque le document source est de très grande taille. En effet, si le document cible de la transformation est de petite taille, parcourir le document source est coûteux. Dans ce cas, il vaudrait mieux produire le document cible en profondeur d'abord. Le modèle d'exécution peut donc être considéré comme insuffisant.
– **Mauvaise performance de la transformation incrémentale**
  Un autre problème de ce modèle concerne la performance d'un incrément de transformation. Ce problème est dû aux modifications élémentaires du document source considérées.
  Supposons que l'utilisateur veuille modifier l'arbre source de la transformation par exemple de a(b,c(e,f),g) en a(b,**i**(e,f),g). Dans ce cas, la décomposition en opérations élémentaires est : "suppression de f, suppression de e, suppression de c, insertion de i, insertion de e, insertion de f". Ce qui implique 6 traitements par incrément pour un simple changement de type (renommage) d'élément du document source.
  Considérer uniquement l'insertion et la suppression comme modifications élémentaires nuit donc fortement à la performance de la transformation incrémentale.

## Synthèse

Ce modèle de traduction d'éléments permet un traitement incrémental très simple en réaction à une modification du document. Les raisons principales sont les suivantes :

1. l'identification des règles à réévaluer est triviale. En effet, puisqu'à un type d'élément correspond une seule règle, lorsqu'un noeud est ajouté dans le document source, l'unique règle à appliquer est immédiatement connue ;
2. D'autre part, l'application d'une règle s'effectue uniquement en fonction des noms d'éléments, indépendamment de tout autre contexte ;
3. Enfin, l'insertion et la suppression d'un élément ne peuvent pas remettre en cause l'application d'une règle déjà appliquée. Traiter le seul élément inséré suffit donc pour mettre à jour correctement le document cible.

## Conclusion

Ce modèle est le premier modèle naïf que nous avons élaboré pour aborder le problème de la transformation incrémentale. Il a le mérite de montrer que la traduction incrémentale d'éléments ne pose pas de problème. En revanche, ce modèle est difficilement extensible du fait de son caractère extrêmement restreint. Cependant, étudier les raisons pour lesquelles le traitement incrémental est aisé dans ce modèle nous a permis d'élaborer certains des critères à l'origine de notre proposition.

# Annexe B

Cette annexe présente les modèles de documents considérés dans notre exemple de transformation sous forme de schémas RelaxNG.

## Modèle de document source

```
<define name="pattern">
  <element name="person">
    <attribute name="gender">
      <choice>
        <value type="string">M</value>
        <value type="string">F</value>
      </choice>
  <element name="name">
   <text/>
   <zeroOrMore>
     <element name="surname">
       <text/>
     </element>
  </element>
  <element name="children">
    <ref name="pattern"/>
  </element>
  </element>
</define>
```

## Modèle de document cible

```
<define name="manpattern">
  <element name="man">
    <attribute name="name">
      <text/>
    </attribute>
    <element name="sons">
       <ref name="manpattern"/>
    </element>
    <element name="daughters">
```

```
      <ref name="womanpattern"/>
    </element>
  </element>
</define>

<define name="womanpattern">
  <element name="woman">
    <attribute name="name">
    </attribute>
    <element name="sons">
       <ref name="manpattern"/>
    </element>
    <element name="daughters">
      <ref name="womanpattern"/>
    </element>
  </element>
</define>
```