Transformation de documents structurés, une combinaison des approches explicite et automatique

Stéphane Bonhomme

▶ **To cite this version:**

Stéphane Bonhomme. Transformation de documents structurés, une combinaison des approches explicite et automatique. Autre [cs.OH]. Université Joseph-Fourier - Grenoble I, 1998. Français. NNT : . tel-00010471

HAL Id: tel-00010471
https://theses.hal.science/tel-00010471

Submitted on 7 Oct 2005

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

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

U.F.R. INFORMATIQUE ET MATHEMATIQUES APPLIQUEES

THÈSE

pour obtenir le grade de

DOCTEUR DE L'UNIVERSITE JOSEPH FOURIER

Discipline : Informatique

présentée et soutenue publiquement

par

Stéphane BONHOMME

le 21 décembre 1998

Titre:

Transformation de documents structurés, une combinaison des approches explicite et automatique

---

Directeur de thèse :

M. Vincent Quint

JURY

M. Claude Chrisment	**Rapporteur**
Mme Christine Vanoirbeek	**Rapporteur**
M. Vincent Quint	**Directeur**
Mme Marie–France Bruandet	**Examinateur, président du jury**
M. Jacques Le Maître	**Examinateur**
Mme Cécile Roisin	**Examinateur**

Remerciements

Je tiens à remercier premièrement Vincent Quint, directeur de recherches à l'INRIA et directeur de la branche européenne du consortium World Wide Web, de m'avoir accueilli au sein de l'équipe qu'il dirige et de m'avoir donné la possibilité de réaliser travail.

Je tiens à remercier les membres du jury de l'intérêt qu'ils ont porté à ce travail :

Marie–France Bruandet, professeur à l'Université Joseph Fourier de Grenoble, de m'avoir fait l'honneur d'être le président de ce jury.

Christine Vanoirbeek, professeur à l'École Polytechnique Fédérale de Lausanne et Claude Chrisment professeur à l'Université Paul Sabatier de Toulouse d'avoir évalué ce travail et des leur remarques constructives qui ont contribué à la qualité de ce document.

Jacques Le Maître, professeur à l'université de Toulon et du Var d'avoir accepté de faire partie de mon jury.

Je tiens à remercier tout particulièrement Cécile Roisin pour les nombreuses discussions qui ont étayé ma réflexion, pour son soutient permanent au cours de ces trois années et la patiente relecture des premières versions de ce manuscrit.

Je voudrais adresser un remerciement particulier à Irène Vatton, l'un des parents légitimes de Thot, qui m'a permis de me sortir de nombreux problèmes techniques lors du développement des logiciels qui sont les fruits de ce travail.

Je remercie aussi tous les membres permanents ou de passage du projet Opéra et de l'équipe du W3C : Muriel, Nabil, Dominique, Manuel, Loay, Fredo, Lionel, Laurent, Laurent, Maria, Yves, Daniel, Ramzi, José, Fred 'grenadine', Marion, Régis...

Je tiens également à remercier Jean–Pierre Verjus, directeur de l'unité de recherche de l'INRIA Rhône–Alpes, ainsi que tous les passagers du paquebot, pour la chaleur de leur accueil au sein de l'UR.

Enfin et surtout, je remercie ma femme Sophie, mes parents et ma famille pour le soutien permanent qu'ils m'ont apporté tout au long de ces années.

Table des matières

Chapitre I Introduction

I.1 Motivations et objectifs 14
I.2 Cadre de travail 15
I.3 Démarche suivie 16
I.4 Plan de la thèse 17

Chapitre II Modèles de documents et transformation dans les applications documentaires

II.1 Modèles et formats de documents électroniques 20
II.2 Documents linéaires 22
II.2.1 Documents texte plein 22
II.2.2 Documents numérisés 22
II.2.3 Documents formatés 23
II.2.4 Postscript et PDF 23
II.3 Documents structurés 24
II.3.1 Modèle de documents structurés 24
II.3.1.1 Structure logique 24
II.3.1.2 Structure logique générique et classe de documents . . . 25
II.3.1.3 Présentation des documents structurés 27
II.3.2 Formats de documents structurés 27
II.3.3 Documents structurés et applications documentaires 30
II.3.4 Systèmes d'édition de documents structurés 32
II.4 Environnement applicatif de la transformation de documents 34
II.4.1 Production et édition de documents 35
II.4.1.1 Structuration d'éléments importés 36
II.4.1.2 Conversion et restructuration pour l'édition de documents structurés 38
II.4.2 Publication de documents 40
II.4.3 Gestion de bases documentaires 41
II.4.4 Synthèse 41
II.5 Objectif de notre travail 42

Chapitre III Techniques et outils de transformation explicite

III.1 Un exemple de transformation 47
III.2 Filtres de conversion 50
III.3 Transformations explicites 53
III.3.1 Méthodes de transformation explicite 54
III.3.1.1 Transformations dirigées par la source 54
III.3.1.2 Transformations par requêtes 55
III.3.2 Langages de transformation 57
III.3.3 Revue des systèmes de transformation explicite 57
III.3.3.1 IDM 58
III.3.3.2 Balise 60
III.3.3.3 CoST 2 64
III.3.3.4 Exrep 65
III.3.3.5 Chameleon 66
III.3.3.6 DSSSL 69
III.3.3.7 XSL 70
III.3.3.8 SgmlQL 72
III.3.3.9 Scrimshaw 74
III.3.4 Synthèse 76
III.3.5 Spécification d'un système de transformation explicite pour l'édition de documents structurés 77
III.4 Les transformations explicites dans Amaya 79
III.4.1 Interface utilisateur 80
III.4.2 Spécification des transformations 81
III.4.2.1 Requêtes 81
III.4.2.2 Règles de génération 83
III.4.3 Déroulement d'une transformation 89
III.4.4 Application du système de transformations explicites d'Amaya 94
III.5 Synthèse 95

Chapitre IV Système de types des documents structurés

IV.1 Caractéristiques des systèmes de types des documents structurés . 101
IV.1.1 Conformité des documents 102
IV.1.2 Attributs 103
IV.1.3 Modularité 103
IV.1.4 Éléments optionnels, inclusions, exclusions 104
IV.2 Définitions 104
IV.2.1 Types de base 105
IV.2.2 Constructeurs 106
IV.2.3 Arbres de types canoniques 107
IV.3 Relations entre arbres de types pour la transformation de documents structurés 110
IV.3.1 Structurations 110
IV.3.2 Réductions 113
IV.3.3 Combinaison des relations de massif et d'absorption 115
IV.4 Transformations fondées sur les systèmes de types 118
IV.5 Synthèse 120

Chapitre V Transformations automatiques

V.1 Préliminaires 125
V.2 Comparaison des arbres de types 126
V.2.1 Génération des empreintes de type 126
V.2.1.1 Mise sous forme canonique 127
V.2.1.2 Ordre pour les arbres de types 127
V.2.1.3 Linéarisation des arbres de types canoniques 128
V.2.2 Comparaison des empreintes 131
V.2.2.1 Principe de la comparaison 131
V.2.2.2 Automate de comparaison 132
V.2.2.3 Un exemple de comparaison 134
V.2.2.4 Validité de l'automate 137
V.2.2.5 Algorithme de comparaison 142
V.3 Création des éléments cible 145
V.4 Extensions de l'algorithme de comparaison 148
V.4.1 Forme réduite de l'empreinte 149
V.4.2 Comparaison avec les empreintes réduites 150
V.4.3 Spécialisation de l'empreinte du type source 150
V.4.4 Comparaison des types récursifs 152
V.5 Application des transformations automatiques 154
V.6 Discussion 155

Chapitre VI Une combinaison des approches automatique et explicite

VI.1 Rappel des objectifs 159
VI.2 Principe de la méthode proposée 161
VI.3 Pré–couplage 162
VI.3.1 Définitions 162
VI.3.2 Expression d'un pré–couplage 163
VI.4 Comparaison de types avec pré–couplage 166
VI.4.1 Principe de la comparaison 166
VI.4.1.1 Contexte d'un pré–couple 167
VI.4.1.2 Comparaison de types hétérogènes 170
VI.4.1.3 Algorithme 172
VI.4.2 Pré–couplage et types récursifs 175
VI.5 Applications de la transformation avec pré–couplage 176
VI.5.1 Importation de documents XML dans Thot 177
VI.5.1.1 Principe de l'application 177
VI.5.1.2 Étapes de la transformation 178
VI.5.1.3 Modification des pré–couples 185
VI.5.1.4 Mémorisation des pré–couples 185
VI.5.2 Pré–couplage et applications des transformations 186
VI.6 Conclusion 186

Chapitre VII Conclusion

VII.1 Rappel des objectifs 189
VII.2 Rappel du travail réalisé 190
VII.2.1 Démarche suivie 190
VII.2.2 Résultats théoriques 191
VII.2.3 Résultats pratiques 192
VII.3 Bilan et évaluation 193
VII.4 Perspectives 194

Annexe A Systèmes de types des langages de programmation

A.1 Langages de programmation et systèmes de types 197
A.1.1 Langages typés et non typés 197
A.1.2 Systèmes de types 198
A.1.3 Contrôle de type et sécurité à l'exécution 200
A.2 Un contrôleur de types simple 200
A.3 Limites et extensions du contrôle de types simple 204
A.4 Inférence de type 205
A.5 Équivalence de types 207
A.5.1 Algorithme d'unification 208

Chapitre I

Introduction

L'édition de documents et la gestion de bases documentaires constituent une part importante des applications informatiques. Outre le gain de place apporté par le support électronique pour l'archivage de documents, l'édition et la publication sont grandement facilités par l'usage des techniques numériques. De nouveaux outils sont continuellement créés et permettent des traitements auparavant fastidieux ou impossibles (indexation automatique, navigation dans les hypertextes, diffusion instantanée à l'échelle planétaire, travail coopératif).

En conséquence, de nombreux documents sont aujourd'hui accessibles sous forme numérique, qu'ils aient été numérisés à partir d'épreuves papier ou produits à l'aide d'un éditeur. Pendant longtemps les formats disponibles ont été très variés, peu homogènes et le plus souvent spécifiques aux outils qui permettaient de les manipuler.

Pour répondre à ces contraintes, des normes ont été élaborées telles que SGML (Standard Generalized Mark–up Language [ISO 86]) ou plus récemment XML (eXtensible Mark–up Language [Bray 98a]). Ces normes ne se contentent pas d'offrir une représentation unifiée des documents indépendante des applications, mais elles introduisent aussi la notion de modèles génériques pour définir des classes de documents, on parle alors de *documents structurés*.

Ces normes sont un des facteurs à l'origine de la révolution des systèmes d'information, dont le World Wide Web est l'une des illustrations. Les applications utilisant les documents structurés permettent de partager plus aisément des documents entre systèmes mais font également émerger de nouveaux besoins dont la *transformation de structure* qui constitue l'objet de cette thèse.

I.1 Motivations et objectifs

Avec le développement du Web, l'échange et la ré−utilisation de documents sont devenues une préoccupation de premier plan. Si les documents structurés offrent une représentation unifiée des structures de document, l'organisation et les types des éléments de ces structures sont toujours spécifiques aux applications qui les manipulent. Le besoin de transformer les documents pour pouvoir les partager entre applications est donc devenu une préoccupation de premier plan pour la conception de systèmes utilisant les documents structurés.

Fondamentalement, le problème de la transformation de documents structurés peut être posé de la façon suivante :

> *Étant donné un document appartenant à une classe, elle−même définie par une grammaire hors−contexte (SGML, XML), par quel procédé est−il possible de transformer l'intégralité ou une partie de ce document pour le rendre compatible avec une autre classe, définie par une autre grammaire ?*

Cet énoncé évoque plusieurs techniques algorithmiques classiques telles que l'analyse syntaxique, la réécriture d'arbres ou la transformation de programmes. Dans ce travail, nous étudierons l'adéquation de ces techniques au domaine de la transformation des documents structurés.

En 1988 déjà, [Furuta 88a] et [Furuta 88b] identifiaient le besoin de transformation pour la conception d'applications utilisant les documents structurés. Alors que l'approche structurée apporte des avantages certains dans la manipulation des documents, la description de leurs structures par des grammaires hors contexte est trop rigide pour permettre des opérations qui sont élémentaires dans les systèmes de manipulation de documents non structurés. Par exemple, la commande *Coller* consiste à insérer les éléments contenus dans le presse−papier dans la structure du document en cours d'éditon. Cette opération ne peut pas toujours être réalisée car ces éléments peuvent ne pas être conformes à la grammaire définissant la classe du document.

L'étude menée dans cette thèse distingue les deux approches principales de la transformation :

- L'approche *explicite* se fonde un ensemble de spécifications exprimant les transformations que doivent subir les éléments du document source pour les rendre compatibles avec la structure cible.
- L'approche *automatique* exploite l'information codée dans la grammaire des classes de documents pour trouver des relations structurelles entre

ces classes. Ces relations sont ensuite utilisées pour transformer les éléments du document source.

Ces deux approches comportent des avantages et des limites que nous identifierons au cours de cette étude. De cette analyse, nous proposerons une nouvelle méthode de transformation qui combine ces deux approches de façon à bénéficier de leurs avantages respectifs.

Dans tout travail de recherche effectué dans un cadre applicatif (ici, l'édition de documents structurés) il est fondamental de confronter les propositions théoriques avec des applications réelles. C'est pourquoi, nous avons tenté de mener tout au long de cette thèse une activité équilibrée entre la réflexion théorique et la démarche expérimentale. Deux types de résultats sont donc attendus :

1. Des résultats théoriques, sous la forme de langages de spécification de transformation qui prennent en compte les aspects liés à l'édition des documents : localité des transformations, simplicité et efficacité des mécanismes d'analyse du langage ; et des méthodes algorithmiques de transformation fondées sur le modèle de documents structurés pour l'approche automatique et pour l'interprétation des langages proposés.
2. Des résultats pratiques, sous la forme de modules de transformation intégrés à des applications d'édition et permettant de spécifier et de réaliser des transformations de complexité significative.

I.2 Cadre de travail

Ce travail de thèse s'est déroulé au sein du projet Opéra de l'Inria Rhône–Alpes. Il a été mené dans le cadre du programme Génie de coopération entre l'Inria et la société Dassault–Aviation et dans le cadre de la coopération entre le projet Opéra et le consortium W3C.

Le programme GENIE est ainsi destiné à renforcer la capacité de diffusion des technologies de l'information adaptées à l'ingénierie concourante au sein de l'industrie. Ce programme vise à résoudre les problèmes documentaires rencontrés actuellement : la documentation structurée, l'hypertexte et la modélisation objet.

Le projet Opéra s'intéresse aux documents électroniques : documents structurés, hypertexte et multimédia. Le modèles de documents conçus intègrent la dimension logique, la présentation graphique, leur contenu et leur organisation temporelle. Plusieurs études sur les transformation de structures ont déjà été

menées dans ce contexte ([Akpotsui 93], [Claves 95]). Le projet met au point des techniques d'édition qui s'appuient sur ces modèles.

Cette thèse s'incrit dans la continuité de ces travaux en ayant pour but d'une part d'évaluer de façon expérimentale les différentes méthodes de trasnformation, incluant celles proposées par le projet Opéra, d'autre part en proposant une technique originale permettant de répondre aux limitations des travaux existants.

La partie théorique de ce travail de thèse s'inscrit dans les activités de modélisation du projet.

Les actions de recherche du projet Opéra trouvent leur application dans les logiciels développés dans le projet :

- Thot est un système d'édition paramétrable pour l'édition interactive de documents structurés XML.
- Byzance est un système d'édition coopérative sur le Web.
- Madeus est un logiciel expérimental d'édition de documents multimédia temporisés.

Notre expérimentation s'intégrera également dans Amaya, un outil d'édition, de navigation et de publication sur le Web développé par le W3C.

Le travail effectué dans cette thèse à été intégré dans deux de ces systèmes : les logiciels Thot et Amaya. Les résultats se répercutent sur Byzance qui utilise les mêmes modalités d'édition qu'Amaya.

I.3 Démarche suivie

Nous pouvons décomposer ce travail en trois phases qui se reflètent dans la structure de ce manuscrit.

- L'étude du processus d'élaboration et de gestion de documents nous permet d'identifier les besoins de transformation spécifiques à chaque application de la chaîne documentaire. Cette étude permet également de cerner précisément dans quel champ d'application nous plaçons notre contribution à savoir l'édition de documents structurés.
- Les différentes techniques de transformation sont abordées suivant une démarche expérimentale. Chaque méthode fait l'objet d'une étude du domaine, d'une étude des systèmes existants et d'une implémentation dans les prototypes développés par le projet. Ces expérimentations permettent d'évaluer si ces techniques peuvent être validées dans notre domaine d'application.

- Notre contribution à travers le développement d'une algorithmique pour la transformation de documents structurés combinant les approches explicite et automatique est ensuite présentée. Cette algorithmique est utilisée pour le développement d'un module de transformation intégré à un environnement d'édition interactif.

I.4 Plan de la thèse

Le plan de ce mémoire reprend la démarche suivie.

Chapitre II

Le chapitre II introduit les différentes représentations de documents électroniques, et en particulier le modèle de documents structurés. L'utilité de la représentation structurée des documents pour les applications documentaires est analysée. La seconde partie du chapitre traite de l'intérêt de la transformation de structure dans les applications de composition et de gestion de document. Nous concluons sur les objectifs que nous nous fixons pour le développement d'un service de transformation pour une application interactive d'édition de documents structurés.

Chapitre III

Le chapitre III présente un état de l'art des systèmes de transformation explicites. Cette étude consiste d'abord à identifier les classes de transformation explicites et leurs différences. Différents systèmes sont ensuite présentés et comparés à l'aide d'un exemple de transformation complet permettant d'évaluer leurs capacités. Nous présentons également le système de transformation que nous avons développé pour l'éditeur Amaya en le comparant aux travaux similaires présentés dans ce chapitre.

Chapitre IV

Le chapitre IV est une étude théorique des systèmes de types des documents structuré. Cette étude a été effectuée à partir de travaux relatifs à d'autres domaines de l'informatique (compilation et analyse de types dans les langages de programmation). La spécification formelle d'un système de types pour les documents permet d'identifier un ensemble de relations entre les structures des documents sur lesquelles se fonde le système de transformation automatique. Nous présentons également d'autres méthodes de transformation qui exploitent la notion de système de types.

Chapitre V

Dans le chapitre V, nous détaillons la méthode de transformation automatique, fondée sur la recherche de relations entre les structures source et cible, utilisant sur le modèle de types proposé au chapitre précédent. Cette méthode était proposée dans [Akpotsui 93] et son expérimentation est décrite dans [Claves 95]. Nous proposons de généraliser l'algorithme de comparaison de structure essentiel pour la suite de notre travail. Nous proposons également des extensions de cet algorithme permettant de traiter des cas d'échec de la comparaison. Ce chapitre se conclut par une évaluation des avantages et des limites de l'approche automatique.

Chapitre VI

Dans le chapitre VI, nous proposons une méthode de transformation combinant les transformations automatiques avec des déclarations spécifiques. Cette méthode permet de résoudre l'ambiguïté liée au manque de sémantique disponible lors des comparaisons automatiques par un processus cyclique de spécifications et de comparaisons. Ce processus permet d'aboutir à une solution pertinente au problème de la transformation tout en nécessitant une spécification minimale de la part de l'utilisateur. Nous présentons également une application de cette méthode pour l'import de document XML dans l'éditeur Thot.

Chapitre VII

La conclusion résume l'apport essentiel de ce travail et propose des perspectives pour l'application des méthodes proposées durant ce travail.

Chapitre II

Modèles de documents et transformation dans les applications documentaires

Ce chapitre présente les besoins de transformation dans les systèmes de documents électroniques.

Les différentes représentations des documents sont d'abord présentées. Nous étudions en particulier les formes de stockage et de représentation des documents électroniques en discutant de l'utilisation de chacune. Les applications documentaires sont également étudiées ainsi que la façon dont elles tirent parti de l'information codée dans le document.

Nous exposons ensuite les avantages de l'utilisation des documents structurés dans les applications documentaires. Nous étudions également les problèmes posés par l'édition ces documents.

Nous poursuivons par le rôle de la transformation de documents dans la chaîne éditoriale. Cette partie met en évidence les besoins de transformation au cours de la vie des documents. Les différentes formes de transformations sont identifiées ainsi que la problématique qu'elles apportent.

La fin de ce chapitre est consacrée à la transformation intervenant lors de l'étape d'édition des documents structurés. Nous identifions les spécificités des transformations dans ce champ d'application pour cerner le domaine de ce travail.

II.1 Modèles et formats de documents électroniques

Le terme *document électronique* désigne une entité abstraite qui, pour être manipulée et stockée, doit être représentée sous une forme concrète.

Le *modèle* définit la représentation des documents au niveau abstrait. Il définit la nature de l'information représentée dans le document et comment cette information est organisée.

Un document peut être un flot d'informations, mêlant le contenu du document et les informations de formatage. Les modèles qui définissent cette représentation des document sont dits *linéaires*. Les modèles de documents linéaires sont présentés dans la section II.2.

D'autres modèles représentent les documents comme une hiérarchie d'éléments typés et définissent les relations hiérarchiques possibles entre les types de ces éléments. Ce sont les modèles de *documents structurés* présentés dans la section II.3.

Le stockage des documents nécessite leur représentation physique dans un ou plusieurs fichiers. La représentation physique d'un document est définie par une convention exprimant la représentation physique du modèle de document : le *format* de document.

Les formats de documents peuvent être *propriétaire*, c'est à dire spécifiques à une application particulière, ou *standard* pour permettre l'échange de document entre applications différentes. Les formats standard de documents sont définis par des organismes de standardisation (ISO [ISO], W3C [W3C]). Ces normes garantissent la pérennité du format de document.

Un format de document présente deux spécificités : la *forme* de stockage qui définit le niveau lexical du format et la *représentation* du document qui définit les aspects syntaxiques et sémantiques des informations contenues dans le document.

Un format peut utiliser une forme de stockage *textuelle* ou *binaire*. Le stockage sous forme textuelle produit un fichier ASCII, ISO–Latin [ISO 98b] ou Unicode [Unicode 96] lisible immédiatement. Le contenu du document et les informations supplémentaires (informations de présentation, de structure) sont codées à l'aide de marqueurs textuels. Ces marqueurs peuvent prendre la forme de macros, comme c'est le cas du format LaTeX [Lamport 85] (voir figure 1) ou de balises comme dans les formats HTML [Raggett 98], SGML [ISO 86] et XML [Bray 98a] (voir figure 2).

```
\section{Mod\'{e}les et formats de documents
\'{e}lectroniques}
Un format peut utiliser une forme de stockage textuelle ou
binaire. Le stockage sous forme textuelle produit un fichier
ISO-Latin \cite{bib0} ou Unicode \cite{bib0} lisible
imm\'{e}diatement. Le contenu du document et les
informations suppl\'{e}mentaires (informations de
pr\'{e}sentation, de structure) sont cod\'{e}es \'{a} l'aide
de marqueurs textuels. Ces marqueurs peuvent prendre la
forme de macros, comme c'est le cas du format \LaTeX\
\cite{bib19} (voir figureFig.~\ref{fig0}) ou de balises
comme dans les formats SGML \cite{bib17} et XML \cite{bib27}
(voir figure Fig.~\ref{fig1}).
```

Figure 1 : Extrait d'un document LaTeX

```
<h2><a name="secta1">1.Mod&eagrave;les et formats de
documents &eacute;lectroniques </a></h2>
<p>Un format peut utiliser une forme de stockage textuelle
ou binaire. Le stockage sous forme textuelle produit un
fichier ISO-Latin <a href="#bib0"></a> ou Unicode <a
href="#bib0"></a> lisible imm&eacute;diatement. Le contenu
du document et les informations suppl&eacute;mentaires
(informations de pr&eacute;sentation, de structure) sont
cod&eacute;es &agrave; l'aide de marqueurs textuels. Ces
marqueurs peuvent prendre la forme de macros, comme c'est
le cas du format LaTeX <a href="these.html#bib19">[Lam]</a>
(voir figure<a href="these.html#fig1"> 1</a>) ou de balises
comme dans les formats SGML <a href="these.html#bib17">
[ISO86]</a> et XML <a href="these.html#bib27">
[W3C98]</a> (voir figure <a href="these.html#fig2"> 2</a>).
</p>
```

Figure 2 : Extrait d'un document HTML

L'information contenue dans un document stocké sous forme binaire est un mélange du contenu du document, généralement sous forme textuelle mais parfois aussi codé en binaire, et de codes de contrôle décrivant les informations associées qui sont toujours représentées en binaire.

La forme binaire permet un stockage plus compact des documents. La lecture du document par une application est plus performante car elle ne nécessite pas d'analyse lexicale de l'information associée. La forme textuelle permet la modification d'un document hors d'une application spécialisée : un simple éditeur de texte suffit (voir section II.3.2). Mais si le document peut être modifié directement par un utilisateur, rien ne garantit qu'il soit encore conforme à son modèle après modification.

L'aptitude à permettre la ré–utilisation de documents que présente la forme textuelle est un facteur dépréciant pour la publication de ces documents. En effet, la diffusion sous forme textuelle d'un livre ou d'un journal électronique ne garantit pas l'intégrité des documents publiés. Les éditions électroniques de quotidiens nationaux par exemple utilisent un format binaire (comme PDF) pour

garantir que le journal conserve son intégrité de contenu comme de présentation.

Dans les deux sections suivantes nous détaillons les modèles et formats de document linéaires et structurés en soulignant l'expressivité de chacun d'eux, la facilité d'échange avec les autres formats et leur aptitude à être utilisés par les applications documentaires.

II.2 Documents linéaires

Les documents linéaires sont modélisés par un flot de données qui mêle le contenu du document aux informations de présentation et/ou de typage. Les formats de documents linéaires sont très proches de leurs modèles : ils sont une simple transcription de l'information contenue dans le modèle.

II.2.1 Documents texte plein

Le modèle de document dit « texte plein » ne contient que la composante textuelle du document. Aucune information supplémentaire n'est représentée dans ce modèle.

Ces documents peuvent facilement être importés dans des applications documentaires sous leur forme originelle. Pour la conversion de ces documents dans un autre format, l'apport d'une information supplémentaire est nécessaire. Cette information permet d'analyser le contenu du document pour identifier les entités qui le composent. La section II.4.1.1 présente cette opération de structuration.

Les messages électroniques (ou e–mail) sont un format de documents sous forme de texte plein, mais ils comportent un en–tête comprenant un ensemble de champs (expéditeur, destinataire, date et heure, sujet, etc.) et un corps contenant le message proprement dit. Une application de messagerie électronique doit analyser le contenu du message pour identifier ces composantes et les traduire dans son modèle de données.

II.2.2 Documents numérisés

Un document numérisé est une image du document. Cette image est représentée par une matrice de points; le contenu textuel du document n'apparaît pas explicitement. Les techniques d'O.C.R. (Optical Character Recognition) permettent de retrouver le contenu textuel du document. De plus, des techniques d'analyse d'image permettent d'exploiter les caractéristiques graphiques du

document (taille et type des fontes, position des blocs de texte) pour permettre sa conversion dans des formats plus explicites (voir section II.4.1.1.)

Les formats de documents numérisés sont utilisés pour l'acquisition de documents papier dans un système de documentation électronique (formats TIFF, JPEG). Ils sont également utilisés pour la télécopie de documents (formats CCITT)

II.2.3 Documents formatés

Le modèle de documents formatés est linéaire. Le contenu textuel du document est entrecoupé d'indications de changement de format du flot de texte. Ces indications peuvent être sous forme binaire, comme dans la plupart des formats utilisés par les traitements de texte, ou sous forme textuelle, ce qui permet leur interprétation par différentes applications.

Le format RTF (Rich Text Format [Microsoft 98]) utilise une forme textuelle pour les indications de formatage de documents, des macros sont déclarées dans l'en–tête du document pour définir des styles. Ces macros sont utilisées dans le corps du document pour indiquer le style des caractères et le formatage du texte (marges, retraits, interlignes, etc.). C'est aussi la structure des documents au format TeX.

LaTeX est un ensemble de macros TeX permettant une expression plus abstraite du formatage du document. LaTeX permet de formater un document en donnant un style (article, chapitre, etc.) et en déclarant les composantes de ce document. Par exemple, pour un article on dispose entre autres de macros identifiant le titre, les auteurs, les titres des différents niveaux de section.

II.2.4 Postscript et PDF

Les formats PostScript [Adobe 91] et PDF [Adobe] développés par la société Adobe occupent une place particulière dans la classification que nous donnons. Leur singularité se manifeste tant au niveau de leur utilisation que de la représentation qu'ils utilisent.

PostScript est un langage de programmation avec des instructions graphiques. Un document Postscript est donc un programme. Ce format est principalement utilisé pour l'impression de qualité. Le langage Postscript est interprété par une machine virtuelle qui produit une image de qualité du document.

Bien que les documents Postscript aient une forme textuelle, comme tous les programmes, ils ne peuvent que difficilement être modifiés ou utilisés par une autre application qu'un interpréteur Postscript.

PDF (Portable Document Format) est un format binaire pour la représentation de l'aspect graphique d'un document. Ce format utilise le même modèle que PostScript, mais se différencie par le fait qu'il utilise un langage simplifié et une forme compressée des données ce qui produit des documents plus compacts. Il est possible d'éditer un document PDF pour y ajouter de l'information telle que des annotations, des zones sensibles et des références, mais le contenu du document originel ne peut pas être modifié.

II.3 Documents structurés

Contrairement aux formes de documents présentées précédemment, les documents structurés sont fondés sur les relations logiques entre les différents composants du document plutôt que sur leur apparence et leur position dans la page. Cette différence se répercute sur les formats de stockage qui dissocient l'information de structure du document des informations nécessaires à son exploitation : formatage, édition, etc.

II.3.1 Modèle de documents structurés

Le modèle de documents structurés représente le document par une organisation logique de ses composants. Les autres aspects du document, par exemple sa présentation, sont le résultat d'un traitement appliqué à cette structure logique.

II.3.1.1 Structure logique

Un document structuré est une collection d'éléments typés composés entre eux selon des relations principalement hiérarchiques.

Par exemple, le document de type *chapitre* présenté dans la figure 3 est composé d'un élément *titre*, d'un élément *introduction* et d'un élément *section*. L'introduction est elle–même composée de deux éléments *paragraphe*, les sections sont composées d'un élément *titre_section*, d'une suite d'éléments *paragraphe* et d'une suite de (sous) sections, ainsi de suite jusqu'aux éléments de base qui représentent le contenu du document.

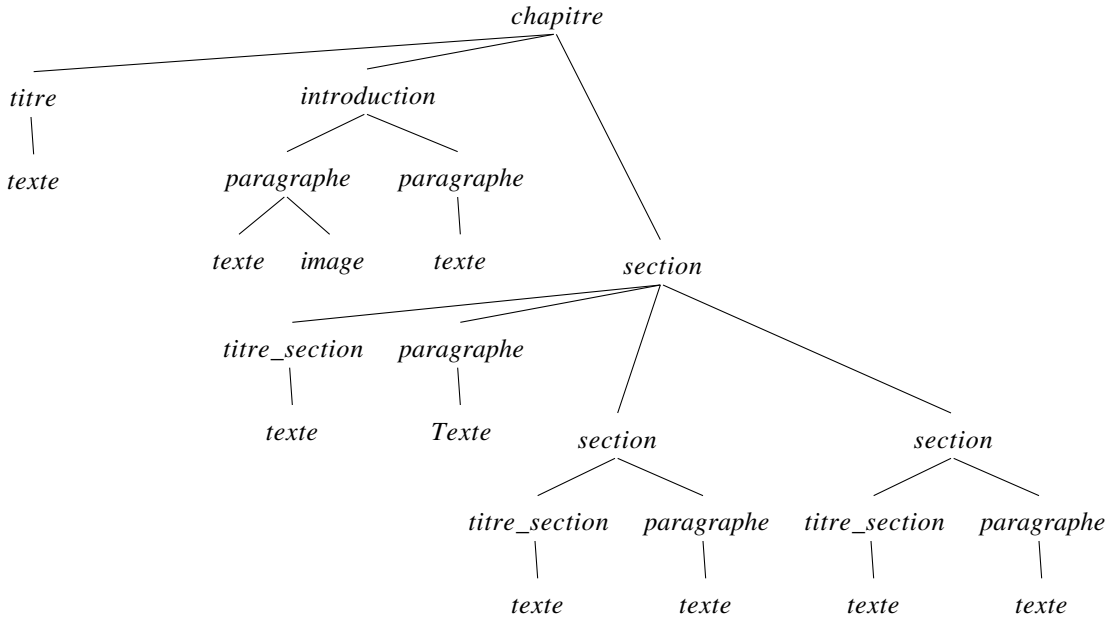


Figure 3 : Structure logique d'un chapitre

Les éléments sont porteurs de sens non seulement par leur contenu, mais aussi par leur type et leur position dans la hiérarchie. Ainsi le titre d'une section du premier niveau ne joue pas le même rôle que le titre du chapitre ou que le titre d'une section de second niveau. Grâce à leurs types et à la structure logique, ces éléments seront considérés de façon différente par une application ou par une autre.

II.3.1.2 Structure logique générique et classe de documents

La hiérarchie d'éléments composant la structure logique est définie par la *structure logique générique*. Cette structure définit les types des éléments pouvant intervenir dans un document et la façon dont ils peuvent être combinés pour former la structure du document. La structure logique générique définit une classe de documents. Tous les documents dont la structure logique est conforme aux définitions de la structure logique générique font partie de la classe que celle–ci définit.

Par exemple, la structure logique générique des documents de la classe Chapitre énonce qu'un chapitre est composé successivement des éléments Titre, Introduction et d'une liste de sections. Cette structure énonce également que l'introduction est composée d'une liste de paragraphes et qu'une section est composée d'un titre de section, d'une liste de paragraphes et d'une liste de sections.

Les structures logiques génériques sont généralement exprimées sous la forme d'un ensemble de règles de production et peuvent être comparées aux grammaires hors–contexte utilisées pour décrire les langages de programmation :

- Il existe une règle de départ décrivant l'élément racine d'un document structuré, respectivement le programme.
- Les symboles décrits dans une grammaire peuvent être terminaux ou non. Les types d'éléments décrits dans une structure logique générique peuvent être des types construits ou des types de base.
- Chaque règle est composée d'un symbole non terminal et de la définition de ce symbole à l'aide de constructeurs, de symboles non terminaux et de symboles terminaux.

Les constructeurs utilisés dans les règles de définition des structures logiques génériques sont les suivants :

- L'agrégat qui définit un ensemble ordonné ou non d'éléments de type différents ;
- La liste qui définit un ensemble ordonné d'éléments de même type ;
- Le choix qui définit une alternative entre plusieurs types d'éléments.

La figure 4 donne la structure logique générique de la classe de documents *chapitre*. Les constructeurs sont indiqués en gras et les types d'éléments terminaux en italique.

```
chapitre -> agrégat (titre, introduction, liste (section))
titre -> texte
introduction -> liste (paragraphe)
section -> agrégat (titre_section, liste (paragraphe), liste
   (section))
titre_Section -> Texte
Paragraphe -> liste (choix (Texte, Image))
```

Figure 4 : Structure générique de la classe de documents chapitre

Un document structuré est dit *conforme* si deux conditions sont vérifiées : les types des éléments qui composent sa structure logique sont définis dans la structure générique qui définit sa classe ; les relations hiérarchiques entre les éléments de la structure logique du document sont conformes aux règles de la structure logique générique.

La structure logique d'un document structuré peut également comporter des attributs. Les attributs permettent d'associer une information aux éléments de la structure logique du document. La structure logique générique définit les

attributs qui peuvent être utilisés dans une classe de documents. Cette structure précise également à quels types d'éléments les attributs peuvent être associés. Dans un document chapitre, un attribut peut donner le numéro de la première page du chapitre, un attribut peut également définir la langue des éléments.

II.3.1.3 Présentation des documents structurés

La présentation des documents structurés n'est pas incluse dans le document comme c'est le cas pour les documents linéaires mais elle est définie relativement à la structure logique générique de la classe de documents.

Bien qu'elle ne soit pas notre préoccupation première, la présentation est un exemple typique d'une application exploitant la structure logique des documents. Pour construire l'image du document, le processus de formatage parcourt la structure logique du document en appliquant les règles associées aux types des éléments rencontrés.

Les règles de présentation peuvent définir une présentation fixée des types d'éléments, comme avec le langage CSS (Cascading Style Sheets) [Bos 98], elles peuvent aussi à leur tour tirer profit de la structure logique du document pour mettre en œuvre des mécanismes d'héritage pour la présentation des éléments. Le système Thot utilise un langage appelé P (pour présentation) qui permet également de décrire les caractéristiques physiques d'un élément en fonction de ses ancêtres, de ses voisins ou de ses descendants.

II.3.2 Formats de documents structurés

Les formats de documents structurés sont définis pour stocker la structure logique des documents. Ils linéarisent cette structure en vue de son écriture dans un fichier. Ils ont en commun une structure générale de la forme :

```
Document := Element
Element  := Declaration_type Declaration_attributs Contenu
Contenu  := (Element | Element_de_base)*
```

Chaque élément est décrit par son type, l'ensemble de ses attributs et son contenu. Le contenu est la linéarisation de la descendance de l'élément dans la structure logique. Cette descendance peut contenir des éléments structurés ou des éléments de base (texte, symboles graphiques, images, etc.)

Thot, avant d'adopter la norme XML, utilisait un format binaire dans lequel le début de chacune des composantes du document était identifié par une marque. Les structures logiques génériques et les règles de présentation sont stockées dans des fichiers communs à l'ensemble des documents d'une classe appelés respectivement schémas de structure et de présentation.

Les formats SGML (Standard Generalized Markup Language) et XML (eXtensible Markup Language) délimitent les éléments de structure par une balise de début d'élément et une balise de fin d'élément. Chaque balise de début d'élément contient le type de l'élément et ses attributs, s'il en a.

Les structures logiques génériques des classes de documents XML et SGML sont appelées DTD (Document Type Definition).

A titre d'exemple, DTD *mail* présentée dans la figure 5 définit une structure générique de messages électroniques. Cette définition exprime qu'un document *mail* est composé des éléments de type *date*, *recipient*, *sender*, *subject* et *textbody*, les quatre premiers contenant un élément texte. L'élément *textbody* est composé d'une suite de paragraphes de type *p* qui peuvent contenir des éléments de type texte et des éléments de type *cite*. La figure 6 représente un document XML conforme à la DTD *mail*. De plus, cette DTD définit un attribut textuel *ref* portant sur l'élément *mail*.

```
<!ELEMENT mail           (recipient, sender,
                          subject, textbody) >
<!ELEMENT date           (#PCDATA)           >
<!ELEMENT recipient      (#PCDATA)           >
<!ELEMENT sender         (#PCDATA)           >
<!ELEMENT subject        (#PCDATA)           >
<!ELEMENT textbody       (p)+                >
<!ELEMENT p              (#PCDATA | cite)*   >
<!ELEMENT cite           (#PCDATA)           >
<!ATTLIST mail    ref    CDATA    #REQUIRED   >
```

Figure 5 : La DTD mail

```
<mail ref="sb332">
 <date>30-07-1998</date>
  <recipient>Cecile.Roisin@inrialpes.fr</recipient>
 <sender>Stephane.Bonhomme@inrialpes.fr</sender>
 <subject>littérature XML</subject>
 <textbody>
  <p>Bonjour Cécile,</p>
  <p>As-tu lu <cite>SGML, Java and the Future of the
   Web</cite></p>
  <p>Stéphane.</p>
 </textbody>
</mail>
```

Figure 6 : Un document mail

Contrairement aux documents SGML, dont la structure est toujours liée à une DTD, le format XML définit deux niveaux de conformité des documents :

- Un document est *bien formé* si sa structure logique est correctement décrite : à chaque balise de début d'élément correspond une balise de fin d'élément de façon que l'ensemble des balises forment une expression

bien parenthésée. Un document bien formé ne fait pas nécessairement référence à une DTD.

- Un document *valide* est un document bien formé dont la structure est conforme à une DTD donnée.

La forme textuelle, telle qu'elle est proposée par SGML et XML nécessite une analyse plus coûteuse que la forme binaire lors de la lecture d'un document. Par contre, elle facilite le partage des documents entre applications. Ceci est dû à deux facteurs :

- L'utilisation d'une syntaxe universelle pour le codage des documents et des structures génériques permet de partager les outils d'analyse syntaxique (*parser*) de documents entre des applications différentes.
- Chaque application peut effectuer un traitement différent d'un même document. L'interprétation des balises peut varier selon les besoins de l'application. Par exemple une application de transfert de messages n'interprétera que les éléments *mail*, *recipient* et *sender* de la structure des documents *mail*, tandis qu'une application de lecture de courrier électronique interprétera les éléments *p* et *cite* afin de construire une représentation graphique du message.

Ce dernier facteur est commun à l'ensemble des formats de documents structurés, cependant la forme textuelle permet d'avoir un aperçu de la structure du document en visualisant le fichier avec un simple éditeur de texte. Cette possibilité permet de développer des outils très simples de traitement de documents basés sur cette forme.

SGML qui existe en tant que standard depuis 1986, n'a cependant connu une application grand public que depuis peu de temps. Il a longtemps été utilisé dans des domaines techniques spécialisés comme la documentation aéronautique (avec la DTD ATA [ATA]), la représentation de documents littéraires (avec la DTD TEI [Burnard 95]) ou multimédia (DTD HyTime [ISO 98a]). La première – et jusqu'a présent la seule – application « grand public » de SGML est le World Wide Web et la DTD définissant la structure des documents hypertexte HTML (HyperText Markup Language [Raggett 98]).

La norme SGML comporte des aspects syntaxiques complexes qui se sont révélés d'une utilisation délicate (inclusions, omissions de marqueurs d'éléments) dans les outils de traitement SGML ([Pepper], [Cover]).

Le format XML [Bray 98a] est une norme de description de documents structurés proposée par le W3C. XML utilise une syntaxe simplifiée par rapport à SGML et tire les enseignements des expériences d'utilisation de ce dernier.

Des propositions, telles que l'utilisation d'espaces de noms [Bray 98b] permettant d'utiliser plusieurs jeux de balises spécifiques à des applications différentes dans un même document, sont en cours d'élaboration et devraient favoriser l'utilisation de ce format pour l'échange de documents entre applications.

Des DTD XML sont d'ores et déjà proposées pour la représentation structurée des formules mathématiques [Ion 98] ainsi que pour la définition de graphiques structurés. Ces DTD répondent aux limitations de HTML pour le codage de ces types de documents et connaîtront certainement un succès plus important que leur équivalent SGML.

Les applications de la norme XML s'étendent au delà de la représentation des documents structurés. Le format XML est utilisé pour la représentation des données dans divers domaines tels que les bases de données, la planification de travail de groupe (Workflow) ou la recherche d'information. On peut citer des propositions telles que RDF (Resource Description Framework) [Lassila 98] ou XLinks [Maler 98] :

- RDF permet de décrire des métadonnées (métadata) associées au documents du Web. Ces métadonnées fournissent aux applications des informations sur la nature du document et leur permettent d'adapter leurs traitements en fonction de cette nature.
- XLink spécifie un mécanisne de liens entre objets. Ces liens sont typés, c'est à dire qu'ils permettent non seulement de pointer vers un objet (comme le font les liens définis par l'élément A de HTML) mais aussi d'exprimer la sémantique du lien et des objets pointés. Bien qu'elle soit définie pour les applications documentaires, la norme XLink peut représenter des liens vers des objets divers tels que ceux des bases de données.

II.3.3 Documents structurés et applications documentaires

Les applications exploitant des documents structurés basent leurs traitements sur la structure logique de ces documents. Deux classes d'applications manipulent les documents structurés, chacune utilisant des modalités d'interaction différentes avec le document : les applications interactives et celles qui ne le sont pas.

Les *applications non interactives* ne permettent pas la manipulation de la structure et du contenu du document via une interface utilisateur. Les composants utilisés par ces applications sont illustrés dans la figure 7. Ils assurent la

lecture d'un document à l'aide d'un parser et offrent un ensemble de primitives permettant d'accéder aux éléments, aux attributs et au contenu du document, de les modifier ou d'en créer de nouveaux. Ces primitives sont accessibles à travers une interface de programmation ou API. Par exemple, le système Thot propose une API décrite dans [Quint 97] qui a servi à la conception de nombreuses applications comme Amaya et Byzance. DOM (Document Object Model [Wood 98]) est une recommandation du consortium W3C proposant des primitives de manipulation de documents structurés.

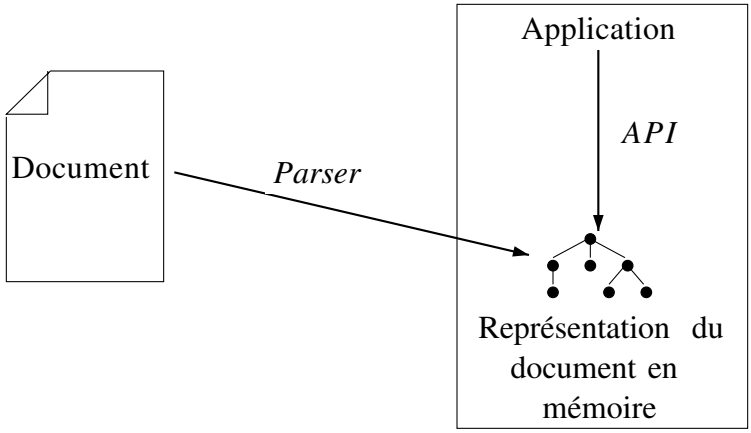


Figure 7 : Application de traitement de documents structurés

À la différence des précédentes, les *applications interactives* supportent les interactions d'un utilisateur avec le document.

En plus des composants utilisés par les applications non interactives, un mécanisme de notification (figure 8) permet à une application d'associer des traitements à des événements déclenchés par les actions de l'utilisateur sur le document structuré. Ces actions incluent l'insertion, la modification et la destruction d'éléments ou d'attributs, leur lecture ou leur sauvegarde, la sélection ou l'activation d'éléments. Les traitements peuvent porter sur un élément quelconque du document ou être spécifiques à un type d'élément.

Les documents structurés et les mécanismes permettant ce mode d'interaction sont désignés sous le terme de *document actif* dans [Quint 94].

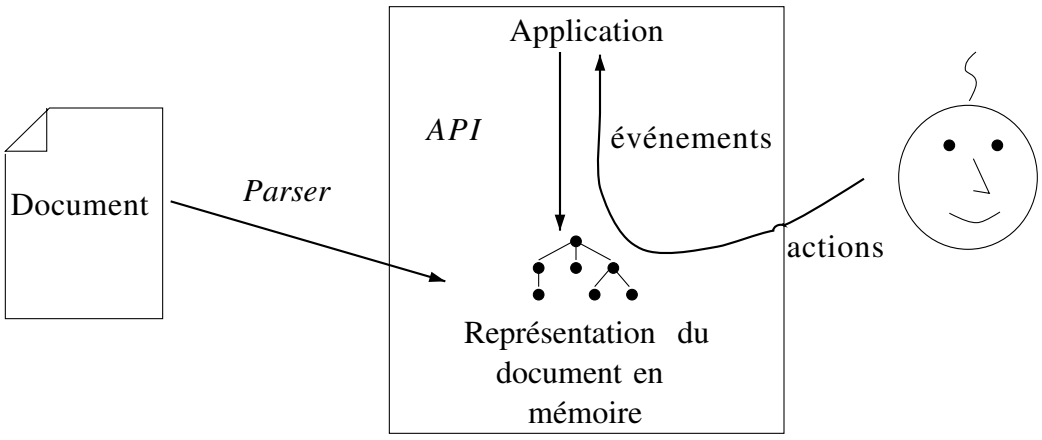


Figure 8 : Application interactive d'édition de documents structurés

II.3.4 Systèmes d'édition de documents structurés

Les systèmes d'édition de documents structurés (SEDS) tels que Thot ([Bonhomme 97]), FrameMaker ([Adobe 95]), Rita ([Cole 90], [Cole 92]) permettent la création et l'édition interactive des documents structurés. Ces systèmes permettent non seulement l'édition de la composante textuelle et de la présentation graphique des documents, mais ils prennent aussi en compte la structure des documents.

Grâce à la notion de structure générique, les éditeurs de documents structurés permettent une édition homogène d'objets de natures différentes [Furuta 88a]. Le même modèle de documents permet de représenter diverses structures tels que des articles, des documentations techniques, des supports de présentation, des graphiques structurés ou encore de documents multimédia [Layaida 97]. La composante graphique de ces documents, quel qu'en soit le support physique (écran d'ordinateur, papier, rétroprojection) est le résultat d'un traitement sur la structure (voir II.3.1.3). Du fait de cette unité de représentation et de présentation, la création d'éléments de diverses natures et leur insertion dans le document est effectuée par le même processus.

Par conséquent les documents manipulés par les SEDS sont conformes au modèle qui a permis de les créer. Pour obtenir ce résultat, deux approches sont possibles, qui présentent chacune des avantages et des inconvénients : le contrôle de la conformité du document peut être strict ou relâché. Un éditeur se fondant sur un contrôle de structure strict assure que le document est conforme au modèle tout au long de sa rédaction. Il ne permet que les opérations d'édition qui maintiennent le document en conformité avec le modèle.

Cette approche pose des problèmes lors de l'édition du document :

- Au cours de sa rédaction, le document ne peut être complet en permanence et il arrive que certains éléments imposés par le modèle ne soient pas encore créés. Pour répondre à cela, l'éditeur Thot crée spontanément les éléments vides imposés par le schéma de structure du document. Ces éléments sont clairement indiqués par des pavés grisés, incitant l'utilisateur à leur donner un contenu.

  Ainsi lorsqu'un document *mail* est créé, les éléments vides *sender*, *recipient*, *subject* et *textbody* sont également créés, indiquant à l'auteur que leur présence est imposée et qu'il doit les remplir.
- La structure de certains éléments peut devoir changer au cours de l'édition. Par exemple dans la rédaction d'un article, l'importance d'une section peut s'avérer moindre que ce qui était prévu au départ et devenir un groupe de paragraphes titrés. À l'opposé, il peut être utile de changer le niveau d'une sous−section pour en faire une section.

  Un éditeur de documents structurés fondé sur un contrôle strict doit donc permettre de changer le type ou de déplacer des éléments de structure. Ce problème est abordé dans la section II.4.1.2.
- Certains éléments peuvent être intégrés à partir d'une forme primaire non structurée. Lors de leur importation dans le système d'édition, ils doivent se conformer au modèle du document.

Le contrôle de structure relâché permet temporairement d'éditer un document structuré indépendamment de son modèle et de créer des éléments dont le type ne se conforme pas aux définitions de la structure générique. Cette approche répond aux trois points précédents, mais elle impose une phase de mise en conformité du document édité avec le modèle. En effet l'édition sous contrôle relâché pouvant mener à une divergence entre la structure du document édité et la structure générique, le document doit être transformé *a posteriori* pour se conformer à son modèle.

Ainsi, l'édition de documents structurés, que l'on utilise un contrôle de structure strict ou relâché, nécessite des transformations de structure de document. Nous exposons dans la partie suivante d'une manière plus fine les types de transformations nécessaires lors des différentes étapes d'édition et de gestion des documents électroniques.

II.4 Environnement applicatif de la transformation de documents

Les activités des intervenants lors de la conception et de la rédaction d'un document diffèrent selon l'environnement utilisé, la nature du document, son but. Nous pouvons cependant dégager certaines activités communes à la conception et à la réalisation de la plupart des documents :

- La *planification* et l'*élaboration* du document mettent en scène un (ou plusieurs) auteurs. La rédaction du document se fait généralement à l'aide d'un éditeur ou d'un traitement de texte. Cette phase peut nécessiter la récupération d'informations provenant d'autres documents (citations, notes d'auteur, etc.). L'importation de fragments de documents de différents formats est donc une fonction importante pour un éditeur de documents.
- La *révision* est un échange (éventuellement répété) entre un ensemble de relecteurs (ou comité de lecture, dans le cas d'une revue) et les auteurs du document. Les lecteurs proposent des corrections et des modifications que les auteurs intègrent dans les versions successives du document.
- La *publication* rend le document accessible à un ensemble plus ou moins large de lecteurs. La publication « traditionnelle » est la diffusion d'épreuves papier du document, tous les exemplaires du document étant identiques. La publication électronique permet de personnaliser la diffusion du document selon les lecteurs tant au niveau de la présentation que du contenu du document.
- La *maintenance* d'un document le fait évoluer en fonction des techniques et des outils. Cette tâche est généralement assurée par d'autres personnes que celles qui interviennent dans les phases précédentes (documentalistes).

  Bien qu'il soit figé, un document archivé doit pouvoir être exploité par les outils de recherche documentaire développés postérieurement à son archivage. Par exemple, le support électronique permet aux lecteurs d'attacher et de partager des informations autour du document. Celui–ci s'enrichit d'informations annexes au fur et à mesure de son utilisation.

La figure 9 présente l'évolution d'un document au cours de ces différentes étapes. La conception d'un document passe par la réutilisation d'informations existantes provenant de sources diverses (a). Cette information doit être adaptée aux modèles utilisés par l'outil d'édition (b). L'étape d'édition

proprement dite (c) fait subir de nombreuses modifications au document qui finalement est publié et/ou archivé (d).

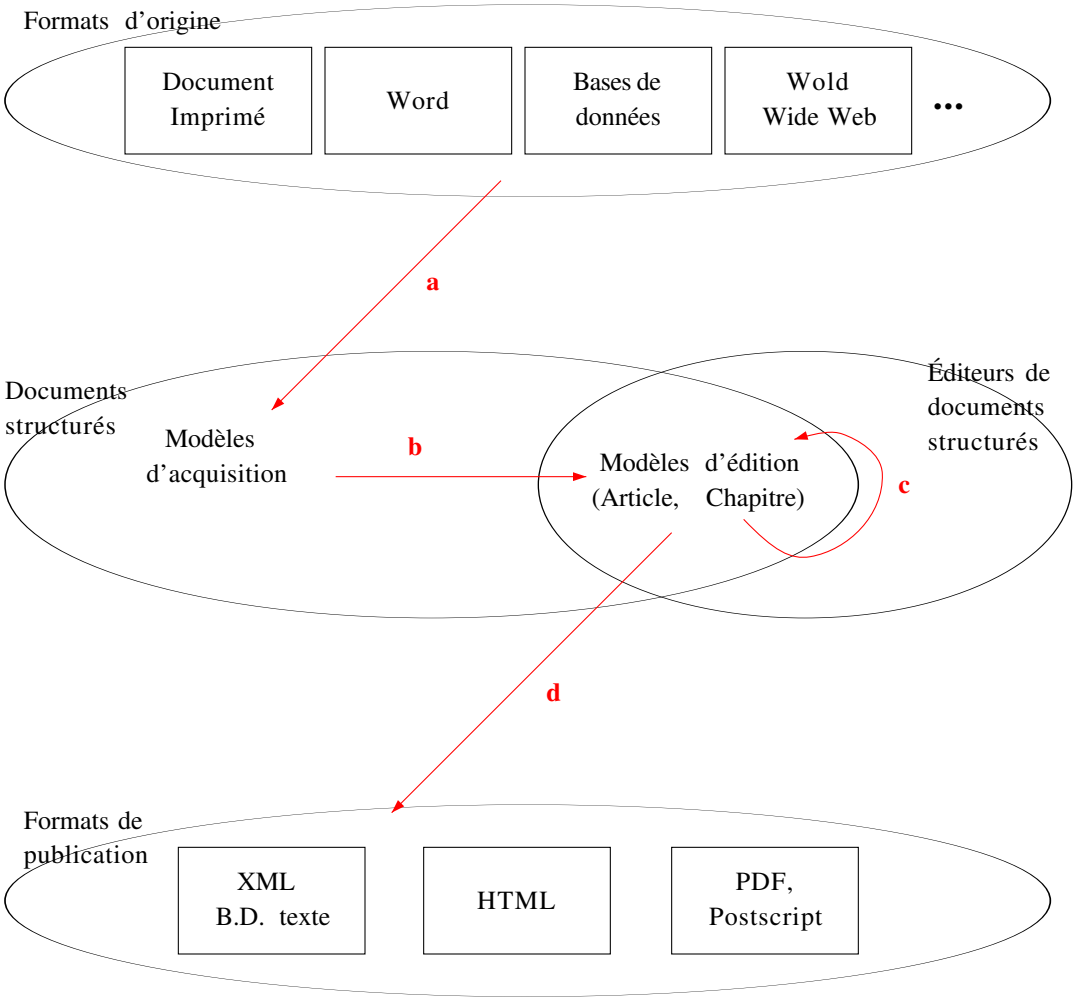


Figure 9 : Les transformations dans la chaîne éditoriale

Chacune de ces activités implique des personnes différentes et met en œuvre des outils divers. Les documents structurés, qui ont la particularité de dissocier la représentation du document des traitements possibles, sont particulièrement adaptés à une gestion homogène des documents tout au long du processus. Cela se reflète dans les projets d'applications documentaires qui utilisent actuellement ce modèle de document.

II.4.1 Production et édition de documents

De nombreux éditeurs permettent aujourd'hui de produire des documents électroniques. Les traitements permis par ces éditeurs dépendent du format dans lequel est représenté le document en mémoire.

Lorsque le document est saisi avec un éditeur de texte, en intégrant éventuellement des informations de formatage (LaTeX) ou de structure (SGML, HTML, XML), aucune aide à l'édition n'est disponible. L'auteur doit assurer lui−même la conformité de son document avec le format choisi pour le stocker.

Les traitements de texte permettent des traitements sur le contenu (correction orthographique, recherche et substitution de texte, indexation) et sur l'organisation du document (plan, feuille de style). Mais le modèle de document qu'ils utilisent est « pauvre » et ne permet pratiquement pas d'autres traitements. Ils permettent également d'insérer des éléments de différentes natures (tableaux, graphiques, images), mais la manipulation de ces éléments nécessite l'utilisation d'outils spécifiques.

Nous avons évoqué dans la section II.3.4 les contraintes imposées par les SEDS. Nous avons également souligné l'utilité des transformations de structure pour lever ces contraintes. C'est ce qui donne toute son importance à la restructuration d'éléments pour l'édition de documents : elle permet de bénéficier des avantages du modèle de documents structurés tout en apportant une réponse aux problèmes posés par les contraintes d'édition de ces documents.

Nous nous proposons dans cette partie d'identifier les transformations nécessaires non seulement pour l'édition de documents structurés, mais pour tout le cycle de vie de ces documents. Ces transformations sont de différentes sortes :

- Transformations entre formats de documents : pour importer des parties non structurées dans le document (structuration), pour exporter un document structuré dans un format de publication (conversion),
- Transformations au sein du format d'édition : pour manipuler interactivement la structure du document et échanger des éléments entre modèles (restructuration et conversion).

II.4.1.1 Structuration d'éléments importés

La structuration « a posteriori » ou *up−translation* est le traitement qui consiste à adopter un document non structuré dans un formalisme structuré. Cette opération implique la déduction des types des éléments et de leur organisation logique conformes à un modèle donné, en s'appuyant sur des informations intrinsèques au format d'origine.

La structuration d'un document se fait généralement en deux étapes :

1. Détermination de la structure logique et/ou physique (structuration).

   Cette étape conduit à une composition hiérarchique des éléments du document. Cette structure peut être libre de contraintes ou se conformer à une DTD d'acquisition (Rainbow [Sklar 94], HTML [Raggett 98]).

2. Application d'un modèle de document : la deuxième étape consiste à transformer la structure logique produite et à la conformer à une DTD voulue. Pour cela les éléments doivent être typés et la conformité de la hiérarchie d'éléments avec la structure générique doit être assurée.

Différentes techniques s'appliquent selon le format d'origine du document à structurer :

- La reconnaissance de caractères est utilisée pour extraire le contenu textuel d'un document numérisé.
- L'analyse d'image permet de déduire la structure physique d'un document numérisé. Le document est découpé en blocs selon la position des éléments dans le document. Les positions relatives des blocs est ensuite analysée pour en déduire une structure. La segmentation en blocs peut également se fonder sur l'interprétation des espaces dans l'image du document [Rus 94].
- Le contenu textuel d'un document ou le résultat d'une reconnaissance de caractères peut être exploité par plusieurs méthodes :

  L'analyse de la typographie du document permet d'identifier des éléments caractéristiques comme les listes (items précédés de tirets), les énumérations, les titres de section (précédés du numéro de section), les paragraphes (retrait sur la première ligne), etc.

  Le contenu du document peut être également analysé au niveau linguistique pour déduire des éléments de structure logique : les titres ne sont pas forcément des phrases (absence de verbe), un résumé est très descriptif.

- Analyse du style du document : les formats de traitement de texte contiennent des informations de style (balises RTF) pouvant être interprétées par un système de structuration. Cette approche est utilisée par la plupart des outils de structuration de documents issus de traitements de texte tels que DynaTag [Inso].

Pour la reconnaissance de la structure d'un document à partir de son image, [Belaïd 97] propose une combinaison des différentes techniques de structuration. Deux approches sont proposées :

- L'approche *perceptuelle* produit une structure représentant l'organisation physique du document. Cette structure est composée d'objets assemblés entre eux par les constructeurs *agrégat* (des objets alignés d'aspect différents), *séquence* (des objets alignés et identiques), *choix* (un objet isolé) et *mosaïque* (des objets non alignés). La structuration est fondée sur une interprétation de la disposition des objets résultant de l'analyse d'image et engendrée par une analyse ascendante des boites composant l'image d'un document.
- L'approche *conceptuelle* cherche à déduire directement la structure logique du document en isolant les concepts apparaissant dans le document. L'analyse de l'image est combinée avec une analyse lexicale de la sortie de l'OCR. Cette approche est adaptée aux micro−structures. Le projet GRAPHEIN [Belaïd 97], par exemple, cherche à structurer des références bibliographiques à partir d'images numériques.

Les techniques présentées dans cette section permettent d'effectuer la première phase de la structuration de documents. Cette phase consiste à produire un document dont la structure reflète uniquement les informations contenues dans le format d'origine du document (style, position des éléments). Cette structure ne peut généralement pas être utilisée en l'état car elle ne représente pas la structure logique du document (niveaux de section, titres).

La difficulté est de convertir les structures produites par la première étape dans des structures logiques utilisables par une application cible. Ces structures logiques sont généralement définies par les DTD utilisées par ces applications. Nous exposons dans la suite des techniques de transformation permettant de réaliser la seconde étape.

II.4.1.2 Conversion et restructuration pour l'édition de documents structurés

Un système d'édition structuré doit produire des documents conformes au modèle utilisé. Or certaines commandes standard d'édition comme le déplacement d'éléments (couper / coller) peuvent mener à des structures incorrectes.

Pour résoudre ce problème certains éditeurs (FrameMaker [Adobe 95]) permettent, durant l'édition, de manipuler une structure incorrecte. L'utilisateur doit alors revenir à un état cohérent avant la sauvegarde du document. Cette approche permet l'utilisation des commandes d'édition traditionnelles, mais

impose à l'utilisateur des manipulations supplémentaires pour conformer le document au modèle.

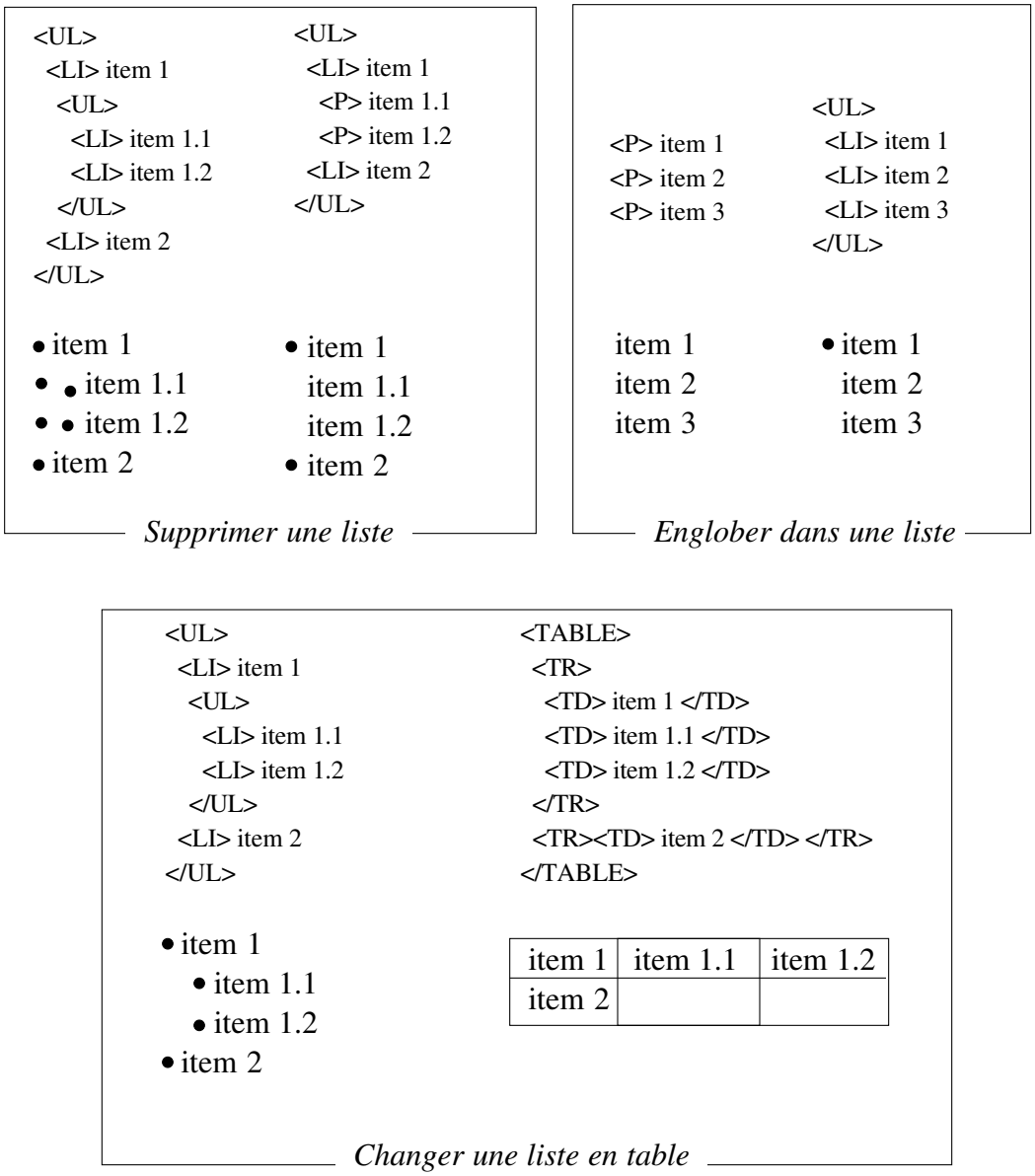


Figure 10 : Transformations pour l'édition interactive de documents HTML

Une autre solution consiste à restructurer les éléments sur lesquels porte la commande pour les rendre conformes au modèle de structure. Nous détaillerons dans la section suivante les techniques qui peuvent être mises en œuvre pour la restructuration dans le cadre de l'édition interactive.

La restructuration permet d'implémenter des commandes d'édition spécifiques aux éditeurs de documents structurés. La suppression de niveaux de structure, l'englobement dans un élément composite ou le changement de type

d'un élément en font partie. La figure 10 montre des exemples de code HTML avant et après l'exécution de quelques commandes de restructuration.

Ces exemples montrent que la correspondance entre les structures source et destination n'est pas triviale : des niveaux de structure peuvent disparaître (supprimer), d'autres peuvent être créés (englober), des éléments doivent éventuellement être insérés si le contexte les rend obligatoires.

Ces transformations peuvent être déclarées explicitement si l'éditeur ne permet de manipuler qu'un seul modèle de document. C'est le cas d'Amaya [Quint 98] qui permet de spécifier par des règles les transformations proposées à l'utilisateur [Bonhomme 96]. On reviendra sur ces règles au chapitre III.4.2.

Si l'on considère un éditeur permettant de manipuler des documents de différentes classes, la définition explicite des transformations devient impossible à cause du nombre de relations éventuelles entre types d'éléments. Une méthode fondée sur la recherche de correspondance entre structures génériques comme celle que nous proposons dans cette thèse semble plus appropriée.

II.4.2 Publication de documents

Lors du processus de publication, un document subit des transformations entre la version rédigée par les auteurs et l'épreuve diffusée. Le format du document original dépend de l'outil utilisé par l'auteur. Le document soumis peut être une épreuve papier et doit alors être capturé par OCR ou ressaisi. Il doit ensuite être formaté pour sa publication.

La soumission peut également être sous forme électronique. L'éditeur peut imposer un format pour la soumission (Latex, HTML, traitement de texte avec feuille de style). Le document sera alors mis en forme par l'éditeur pour assurer une cohérence graphique de la publication. L'auteur doit, dans ce cas, traduire le document depuis le format de production dans le format de soumission.

La publication électronique, apparue avec le développement des réseaux de télécommunications (Internet, Intranet) et les protocoles d'échange de documents électroniques (HTTP [Fielding 98]) permet de diffuser instantanément des documents à un large nombre de lecteurs. Les formats couramment utilisés pour l'impression d'épreuves papier ou la publication électronique (PDF, Postscript, HTML) sont généralement différents des formats d'édition et de soumission (LaTeX [Lamport 85], word [Microsoft 92], RTF [Microsoft 98]). L'éditeur doit donc transformer les documents vers le format de publication.

Dans le cas de la publication électronique, la transformation du document dans le format de publication peut être faite dynamiquement pour engendrer un

document personnalisé selon le profil ou une requête d'un lecteur. Certains journaux électroniques proposent ainsi une sélection d'articles en fonction des centres d'intérêt du lecteur.

II.4.3 Gestion de bases documentaires

Comme nous l'avons noté précédemment la publication d'un document ne signifie pas la fin de son traitement informatique. Les documents électroniques continuent à être utilisés après leur publication par des systèmes de recherche d'information, des outils de consultation et d'annotation. Certains passages peuvent être cités ou inclus par de nouveaux documents, etc.

Les documents formatés (PDF, PostScript), largement utilisés pour la publication électronique, ne seront pas réutilisables par de futures applications. Le format de documents structurés, au contraire, peut être exploité par n'importe quelle application, du moment qu'elle garantit la concordance du document avec son modèle. Le même document peut être manipulé et échangé par les applications existantes et à venir.

Le développement de nouveaux outils s'accompagne d'une évolution des structures génériques des classes de documents structurés. Le problème de la conversion des documents se pose alors pour la maintenance d'une base de documents structurés. Deux approches sont possibles pour résoudre ce problème :

- Un traducteur de documents est écrit à chaque modification d'un modèle de document. Certains outils (IDM [Digithome], Scrimshaw [Arnon 93], SIMON [Feng 93] étudiés dans le chapitre suivant) permettent de programmer des filtres de conversion à partir de règles de correspondance entre l'ancienne et la nouvelle structure logique générique.
- Un système de conversion générique traduit automatiquement les documents, se basant sur la comparaison des modèles de document comme nous le proposons dans le chapitre V.

II.4.4 Synthèse

Outre les besoins de transformation de documents dans un atelier éditorial, cette partie a mis en avant l'intérêt du format de documents structurés pour permettre la production et l'exploitation d'un même document tout au long de son cycle de vie. Les documents structurés présentent l'avantage de pouvoir être facilement transformés et de pouvoir offrir des représentations différentes à partir d'une structure unique.

II.5 Objectif de notre travail

Cette partie définit le contexte de ce travail et délimite la portée de la contribution que nous voulons apporter. Nous situons ici la position dans la chaîne éditoriale du système de transformation que nous proposons.

Notre thème de recherche est centré sur les systèmes d'édition interactive de documents structurés. Nous nous intéressons donc principalement aux transformations appliquées à l'édition de documents structurés. Cependant, dans notre étude, nous serons amenés à examiner des systèmes de transformation appliqués à d'autre domaines (génération de documents depuis des bases de données, formatage de documents). Les transformations qui nous intéressent particulièrement interviennent alors que les tâches suivantes sont effectuées (figure 11) :

- Importation de documents (a) pour permettre la réutilisation des documents existants. Nous considérons l'importation de documents texte plein et de documents structurés.

  Les documents formatés utilisent des formats aux syntaxes trop diverses pour pouvoir être importés selon une méthode générique. De plus ils peuvent être obtenus par l'application de filtres de conversion vers des modèles d'acquisition de documents structurés.
- Transformation en cours d'édition (b) : les transformations intervenant au cours de l'édition permettent la modification dynamique de la structure du document. Ces transformations incluent le changement de type d'éléments, l'insertion et la suppression d'éléments de structure intermédiaires et le déplacement d'éléments dans la structure du document.
- Conversion de documents entre modèles (c) : cette classe de transformations permet de convertir un document dans son intégralité d'une DTD d'édition dans une autre. Ces transformations sont nécessaires lorsqu'une DTD évolue ou lorsque les documents doivent être échangés entre applications utilisant des DTD différentes. Par exemple lors de la sauvegarde de documents, il est parfois nécessaire de les convertir dans un format spécifique pour publication dans une revue, ou en HTML pour publication sur le Web (c).

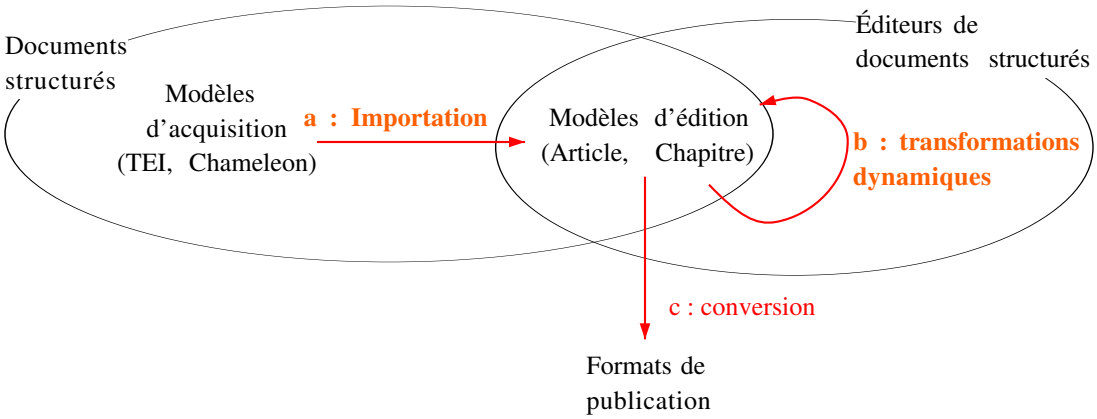


Figure 11 : Les transformations pour l'édition de documents structurés

Les besoins de transformation exprimés ci–dessus nous permettent de définir les caractéristiques du système de transformation de documents structurés que nous proposons dans ce travail :

- Les transformations doivent être rapides pour pouvoir être exécutées au cours de l'édition. Cette contrainte implique que les algorithmes utilisés soient efficaces.
- Les transformations doivent pouvoir porter sur un document entier, pour l'importation et la conversion. Elles doivent également pouvoir porter sur un sous–ensemble localisé du document pour les transformations dynamiques.
- La structure d'un document ayant subi une transformation doit être conforme au modèle du document, en particulier les transformations locales ne doivent pas mener à une structure de document incorrecte.

Ce chapitre à présenté une approche applicative de la transformation de documents. Nous allons nous intéresser dans les chapitres suivants aux techniques de transformation et à la manière dont ces dernières sont exprimées. L'étude menée dans ces chapitres nous permettra de compléter les trois caractéristiques énoncées ci–dessus en prenant en compte les aspects techniques et l'expression des transformations.

Nous analysons les techniques de transformation de documents existantes suivant une méthodologie particulière. Nous avons identifié trois familles de techniques de transformation fondées sur des principes différents : les filtres, les systèmes de transformation explicite et les systèmes de transformation automatique.

- Les **filtres** (cf. section III.2) sont utilisés pour traduire des documents dans leur intégralité. Un filtre est défini pour des formats source et destination spécifiques.

  On peut classer les filtres en deux catégories :

  – Les filtres permettant la traduction de documents entre formats, par exemple de Word vers PDF.

  – Les filtres permettant de traiter des documents structurés prennent en compte lors de la traduction non seulement le format utilisé mais aussi les structures logiques source et destination.

- Les systèmes de **transformation explicite** (c.f. section III.3) se basent sur un ensemble de règles guidant la transformation. Ces règles permettent de paramétrer les transformations et leurs permettent, à la différence des filtres, de s'appliquer à différents formats de documents. Cette technique peut être mise en œuvre par des applications dédiées aux transformations ou intégrées dans d'autres applications (éditeurs de documents, bases de données).

- La comparaison des structures génériques des classes de documents présentée dans le chapitre IV permet une **transformation automatique** de documents qui sera présentée dans le chapitre V. Cette méthode étant fondée sur une comparaison des structures génériques, elle ne peut s'appliquer qu'aux transformations entre documents possédant une représentation riche, comme les documents structurés.

Dans la suite, nous présentons la technique de transformation utilisée par chaque famille en trois parties :

- nous donnons d'abord les principes de la technique de transformation ;
- Nous présentons ensuite les outils et prototypes existant fondés sur cette technique ;
- finalement, nous décrivons notre expérimentation de la technique de transformation. Ainsi nous avons développé différents systèmes de transformation dont certains ont été intégrés aux logiciels diffusés par le projet.

Dans ce cadre, nous avons expérimenté un filtre de transformation lors de l'évolution d'une structure générique de document structuré. Le filtre de conversion des documents Article en Rapport à été utilisé pour la conversion des documents du projet (cf. section III.2).

Amaya propose une transformation de type pour l'édition de documents HTML. Cette fonction à été initialement développée pour expérimenter un système de transformation explicite dans le contexte d'un outil d'édition (voir section III.4).

L'éditeur Thot propose des commandes de restructuration génériques basées sur la technique de transformation automatique décrite au chapitre V.

Cette approche expérimentale de l'état de l'art nous permet non seulement de connaître les techniques utilisées pour la transformation de documents structurés, mais aussi d'évaluer l'intérêt de ces techniques dans le cadre spécifique de l'édition de documents.

Chapitre III

Techniques et outils de transformation explicite

Nous présentons dans ce chapitre les techniques de transformation nécessitant une spécification additionnelle aux structures et aux instances de documents. Ces techniques regroupent les filtres qui utilisent une spécification implicite de la transformation et les systèmes de transformation explicites qui interprètent un ensemble de directives servant à paramétrer les transformations. Ces deux familles de transformations sont regroupées dans ce chapitre car il existe des similarités entre les techniques utilisées, comme on le verra plus loin.

La première section présente un exemple de transformation simple qui nous permet d'illustrer les techniques de transformation étudiées. Nous abordons ensuite les filtres de conversion, puis nous détaillons les systèmes de transformation explicite existant avant de présenter les transformations utilisées par Amaya.

III.1 Un exemple de transformation

Pour illustrer les différentes techniques de conversion, nous utiliserons un exemple de transformation de document structuré. Cet exemple nous sert dans ce chapitre à montrer la syntaxe utilisée. Il nous sert également à mettre en valeur les limites et les points forts de certains systèmes de transformation.

Nous avons choisi un exemple à la fois simple et représentatif de l'ensemble des transformations de documents structurés. Il met en œuvre les opérations de

transformation décrites dans le chapitre précédent (structuration, déplacement d'éléments, suppression et insertion de niveaux de structure, changement de type).

Cet exemple reprend la DTD *mail* présentée au chapitre précédent (et rappelée dans la figure 12) qui décrit la structure générique d'un message électronique. Un document *mail* (figure 13) comprend les éléments *recipient*, *sender*, *subject* et *textbody* correspondant respectivement aux destinataire, expéditeur, sujet et contenu du message. Le contenu est composé d'une suite de paragraphes eux–mêmes composés de texte et de citations identifiées par l'élément *cite*. L'élément *mail* porte l'attribut *ref* défini dans la DTD qui peut être utilisé par les applications pour référencer le message.

```
<!ELEMENT mail           (date, recipient, sender,
                          subject, textbody) >
<!ELEMENT date           (#PCDATA)            >
<!ELEMENT recipient      (#PCDATA)            >
<!ELEMENT sender         (#PCDATA)            >
<!ELEMENT subject        (#PCDATA)            >
<!ELEMENT textbody       (p)+                 >
<!ELEMENT p              (#PCDATA | cite)*    >
<!ELEMENT cite           (#PCDATA)            >
<!ATTLIST mail    ref    CDATA    #REQUIRED   >
```

Figure 12 : La DTD XML des documents mail

```
<mail ref="sb332">
 <date>30-07-1998</date>
  <recipient>Cecile.Roisin@inrialpes.fr</recipient>
 <sender>Stephane.Bonhomme@inrialpes.fr</sender>
 <subject>littérature XML</subject>
 <textbody>
  <p>Bonjour Cécile,</p>
  <p>As-tu lu <cite>SGML, Java and the Future of the
   Web</cite></p>
  <p>Stéphane.</p>
 </textbody>
</mail>
```

Figure 13 : Un document mail

Notre exemple consiste à transformer le document *mail* de la figure 13 en un document *message* dont la structure générique est décrite par la DTD de la figure 14. Les deux DTD diffèrent sur les 6 points suivants :

1. Tous les noms des éléments changent entre les deux DTD.
2. L'élément *cite* de la DTD *mail* n'a pas d'équivalent dans la DTD *message*, mais il doit être transformé sous forme d'un texte mis entre guillemets.

3. La DTD *message* contient un niveau de structure supplémentaire : l'élément *contenu*, regroupant les éléments *titre* et *corps* correspondant respectivement aux éléments *subject* et *textbody* de la structure source.
4. L'attribut *ref* de la DTD *mail* correspond à un élément *référence* dans la DTD *message*.
5. L'élément *date* apparaît comme un attribut dans la DTD *message*.
6. L'ordre des éléments *destinataire* et *expéditeur* est inversé par rapport aux éléments *recipient* et *sender*.

Tous les systèmes de transformation ne permettent pas de traiter tous les aspects de la transformation de `mail` en `message`. Nous donnerons, lors de la présentation des systèmes de transformation, l'ensemble de leur possibilités en référence à cette liste. Nous pourrons ainsi différencier les systèmes par leur capacité à réaliser les différents aspects de cette transformation.

```
<!ELEMENT message        (référence, expéditeur,
                          destinataire, contenu)  >
<!ELEMENT référence      (#PCDATA)                >
<!ELEMENT expéditeur     (#PCDATA)                >
<!ELEMENT destinataire   (#PCDATA)                >
<!ELEMENT contenu        (titre, corps)           >
<!ELEMENT titre          (#PCDATA)                >
<!ELEMENT corps          (para)+                  >
<!ELEMENT para           (#PCDATA)                >
<!ATTLIST message       date   CDATA   #REQUIRED  >
```

Figure 14 : La DTD XML des documents message

Le document que l'on désire obtenir à l'issue de la transformation est donné dans la figure 15.

```
<message date="30-07-1998">
 <référence>sb332</référence>
  <expéditeur>Stephane.Bonhomme@inrialpes.fr</expéditeur>
  <destinataire>Cecile.Roisin@inrialpes.fr</destinataire>
 <contenu>
  <titre>littérature XML</titre>
  <corps>
   <para>Bonjour Cécile,</para>
   <para>As-tu lu «SGML, Java and the Future of the
   Web»</para>
   <para>Stéphane.</para>
  </corps>
 </contenu>
</message>
```

Figure 15 : Document message

III.2 Filtres de conversion

Un filtre est une application permettant de convertir un document d'un format donné dans un autre. Il existe des filtres permettant la transformation de documents entre toutes sortes de modes de représentation (traitement de texte, PostScript, PDF, LaTeX ou documents structurés). On peut citer les filtres de traduction de LaTeX vers HTML, de document formatés vers des documents SGML, conformes à des DTD particulières comme les DTD d'acquisition (TEI).

La caractéristique qui distingue les filtres de conversion des autres systèmes de transformation est la définition figée des transformations qu'ils permettent d'effectuer. Ces transformations sont définies lors du développement du filtre qui ne peut pas être paramétré pour effectuer des transformations différentes. Par exemple, un filtre permettant de traduire les documents *mail* en *message* comme montré dans notre exemple ne pourra effectuer que cette transformation et elle se fera toujours en suivant les mêmes règles.

Les filtres sont conçus pour répondre aux besoins spécifiques de certaines applications :

- La conversion d'un ensemble de documents homogènes : lors de la migration d'un large ensemble de documents d'un format vers un autre, des filtres adaptés sont développés.
- L'évolution d'une DTD : lors de la révision d'une DTD, il est préférable de maintenir une compatibilité avec l'ancienne version de la DTD. Lorsque cette compatibilité ne peut être maintenue (ajout d'éléments obligatoires, suppression de niveaux de structure), le développeur de la DTD peut fournir un filtre permettant de convertir les documents existant pour les rendre conformes à la nouvelle DTD.
- La structuration de documents existants : les informations sur lesquelles repose la transformation sont parfois spécifiques au format source (balises de formatage) ou au document (contenu). La conception de systèmes généraux permettant d'interpréter les formalismes de stockage de diverses applications nécessite des mécanismes complexes, comme ceux fournis par des outils d'analyse syntaxique tels que Yacc. Cela va à l'encontre du besoin de simplicité d'expression des transformations identifié dans le chapitre précédent. C'est pourquoi une approche fondée sur des filtres spécifiques à un couple de formats est plus adaptée à la structuration des documents.

Il convient de noter que de nombreuses applications documentaires utilisent des filtres de conversion pour traduire en HTML les document qu'elles produisent. Il est également prévisible qu'à l'avenir ces applications utiliseront le format XML pour le stockage. Ceci relativise l'intérêt de cet aspect de la transformation de documents et nous pousse à nous concentrer sur la transformation de documents structurés.

- Dans certains cas très spécifiques, une technique similaire à celle utilisée par les filtres peut servir à implémenter des commandes d'édition. Par exemple, de nombreux éditeurs de graphiques structurés proposent des opérations de groupement et de dégroupement d'objets. Cette opération consiste à insérer un élément *groupe* dans la structure du document et à insérer un ensemble d'éléments graphiques à l'intérieur.

L'implémentation de filtres de conversion est très dépendante des formats des documents source. Si le format source est linéaire (contenu textuel et directives de formatage, comme c'est le cas pour les documents produits par les traitements de texte), le filtre parcourt le document, interprétant les directives de formatage pour produire leur équivalent dans le format cible.

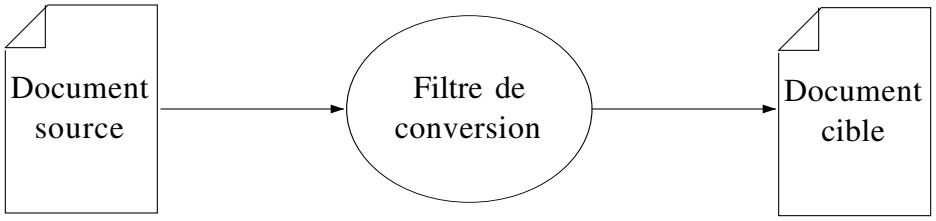


Figure 16 : Principe du filtre de conversion

Si le modèle des documents source est structuré (SGML, XML), le filtre intègre généralement un analyseur syntaxique (parser) permettant d'associer des actions – en l'occurrence la génération du document cible – aux débuts et fins d'éléments de la structure.

Certains des systèmes de transformation explicite présentés dans la section suivante permettent de programmer des filtres de conversion de documents de façon interactive (Chameleon [Mamrak 89]) ou en utilisant une spécification déclarative de la transformation (IDM [Digithome], XSL [Clark 98], DSSSL [ISO 94]).

Pour notre part, nous avons expérimenté la technique de conversion de documents par filtres lors de la création d'un nouveau schéma, nommé *rapport*, pour la classe de documents *article* définie dans l'éditeur Thot. Ce nouveau schéma se différencie du précédent principalement sur deux points :

- Le schéma *rapport* définit les sections de façon récursive, alors que le schéma *article* définit un type d'éléments pour chaque niveau de section (*section*, *sous−section*, *sous−sous−section*).
- La bibliographie d'un document *article* est une référence à une structure externe au document. Dans le schéma *rapport*, la bibliographie est intégrée à la structure du document.

D'autres différences mineures concernent les noms des types d'éléments et l'introduction de niveaux de structure supplémentaires dans la structure générique définie par le schéma *rapport*.

Notre filtre est programmé en utilisant l'API de Thot [Quint 97] qui fourni des primitives d'accès aux éléments du document source et des primitives de construction pour créer le document cible. La fonction de transformation crée l'équivalent d'un élément du document source passé en paramètre, cette fonction est ensuite appelée récursivement sur chacun des fils de l'élément source. L'arborescence ainsi obtenue est ensuite insérée dans la structure du document cible. De plus, la fonction de transformation effectue un traitement particulier sur les éléments appartenant à certains types (bibliographie, références) au lieu de traiter récursivement leur descendance.

Cette application nous a permis de nous familiariser avec la transformation de documents, mais nous a également montré que, bien qu'utilisée couramment par les membres du projet Opéra pour la transformation d'Articles en Rapports, une telle application n'est pas ré−utilisable pour d'autres transformations.

Les filtres de conversion ont pour atout leur efficacité car ils appliquent des algorithmes de transformation adaptés aux formats traités. Aucune information autre que le contenu du document ne doit être analysée ce qui accroît les performances de ces outils. Les filtres sont adaptés au traitement de collections importantes de documents suivant une méthode déterminée à l'avance et ne nécessitant donc pas d'interaction avec l'utilisateur. Dans certains cas spécifiques, les filtres peuvent être utilisés dans les systèmes d'édition interactifs, notamment lors de commandes d'édition précises portant sur des types d'éléments déterminés (comme la transformation d'une portion de feuille de calcul en tableau en vue de son intégration dans un traitement de texte).

La contrepartie de cette non−interactivité est que l'utilisateur ne peut pas guider le cours de la transformation. De plus, les filtres ne sont pas adaptables, c'est à dire que leur conception est dépendante des formats qu'ils manipulent, ils ne peuvent pas servir non plus à transformer les documents entre deux DTD quelconques.

Dans notre problématique, les filtres de conversion présentent un intérêt pour convertir des documents depuis des formats divers vers un format structuré sur lequel nous pouvons ensuite appliquer les techniques de transformation exposées plus loin.

III.3 Transformations explicites

Un système de transformation explicite s'appuie sur une information additionnelle aux documents et à leurs modèles, exprimant la façon dont un document doit être transformé. Cette information permet de paramétrer les transformations et par conséquent de les adapter aux besoins.

Les systèmes de transformation explicite prennent comme données un document source (ou une partie de document) et un ensemble de spécifications de transformation. Les spécifications de transformation sont interprétées puis utilisées pour convertir l'instance source en une instance cible (figure 17).

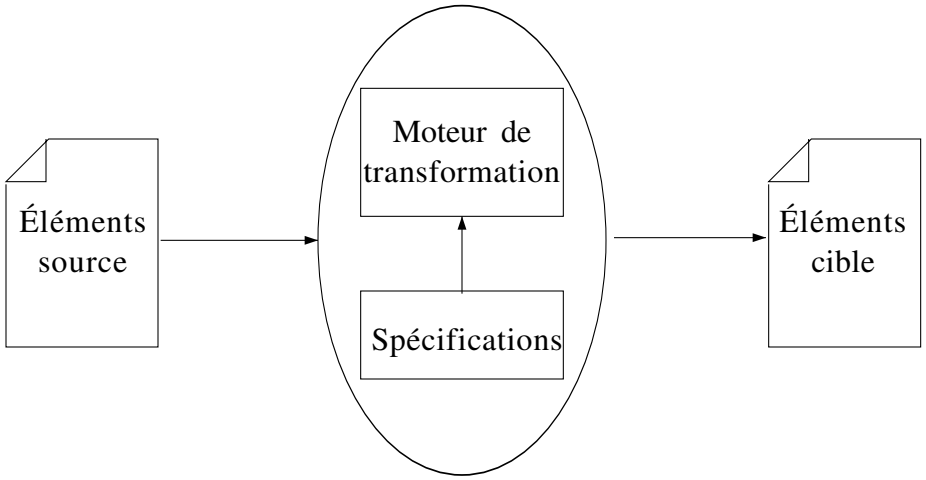


Figure 17 : Architecture générale d'un système de transformation explicite

Ainsi, deux aspects sont considérés dans l'étude de ces systèmes : le langage de spécification des transformations et la mise en œuvre de la conversion à partir de ces spécifications dans le moteur de transformation. Il faut noter que le langage dans lequel sont exprimées les transformations et le moteur qui les réalise sont dépendants l'un de l'autre.

Nous proposons dans la section III.3.1 une classification des méthodes de transformation explicite. Les classes de langages utilisés par ces méthodes sont présentés dans la section III.3.2. La section III.3.3 est une revue des principaux systèmes utilisant la technique explicite. Un bilan des outils de transformation explicite est donné dans la section III.3.4.

La section III.4 est une étude particulière d'un travail réalisé dans le cadre de cette thèse, qui visait à évaluer la technique de transformation explicite dans un contexte d'édition interactive de documents structurés. Cette expérience à débouché sur la réalisation d'un système de transformations explicites pour l'éditeur Amaya.

III.3.1 Méthodes de transformation explicite

Les moteurs de transformation se différencient selon deux critères principaux :

L'ordre d'application des règles

- Les transformations explicites peuvent être dirigées par la structure du document à transformer. Dans ce cas, une transformation est exprimée par des actions liées aux types des éléments du document source. Nous désignerons cette approche par le terme *transformation dirigée par la source.*
- L'autre approche consiste à spécifier les transformations par identification dans le document source des éléments à composer pour produire le document . Nous appellerons cette méthode *transformation par requêtes.* Ces deux approches sont développées ci–dessous.

Le mode de génération du résultat

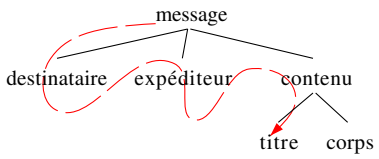
- Le document résultant de la transformation peut être engendré en écrivant directement sur un flot de sortie du système de transformation ; la génération du document est alors dite *linéaire*.
- Le document peut être construit sous la forme d'un arbre représentant sa structure avant de produire la forme de sortie ; cette approche est appelée génération *arborescente*.

III.3.1.1 Transformations dirigées par la source

Dans cette approche, une transformation est conduite par un parcours dans l'ordre préfixe de la structure du document source. Un ensemble de règles ou de primitives de génération du document cible est associé à un événement du parcours (début de document, début d'élément, fin d'élément, présence d'attribut, valeur d'attribut, etc.).

Dans certains formats de documents tels que MIF ou PDF, les éléments se sont pas ordonnés dans la forme stockée. Les documents stockés dans de tels formats sont lus par le système de transformation qui rétablit l'ordre logique des éléments avant leur traitement.

La figure 18 illustre la génération des événements lors de la transformation d'un document de la classe message. L'événement de début d'élément message est généré, puis la descendance de cet élément est parcourue pour produire les événements qui y sont associés, l'événement fin d'élément message est produit lorsque toute sa descendance à été parcourue.




```
Début message
 Début destinataire
 Fin destinataire
 Début expéditeur
 Fin expéditeur
 Début contenu
  Début titre
  ...
 Fin contenu
Fin message
```

Figure 18 : Production des événements lors de la transformation dirigée par la source.

Les transformations dirigées par la source permettent une sélection des événements en fonction du contexte dans lequel intervient l'élément de structure auquel est attaché cet événement. Ce contexte couvre les éléments qui ont été traités précédemment par le système au moment où il applique une règle : les ancêtres de l'élément courant, ses voisins de gauche et leurs descendants.

La transformation dirigée par la source permet une génération linéaire du document cible. Dans ce cas, les transformations ont la propriété de préserver l'ordre des éléments. La génération peut également être arborescente si le système de transformation fournit des fonctions de construction de structure comme la création, l'insertion ou la copie d'éléments.

L'approche des transformations dirigées par la source est proche du mécanisme d'un analyseur syntaxique et la plupart des outils proposant cette approche sont fondés sur des analyseurs de documents structurés (CoST 2 [English 96a], STIL [Schrod 95]). Ces systèmes bénéficient des fonctionalités de l'analyseur pour la production des événements et fournissent un mécanisme de génération de la structure cible.

III.3.1.2 Transformations par requêtes

Les transformations par requêtes permettent de construire le document cible par extraction d'éléments du document source. Une transformation est décrite par un ensemble de règles « requête–actions ». Une *requête* permet de sélectionner un sous–ensemble des éléments du document source sur lesquels doit être appliquée une (ou plusieurs) *action(s)* de transformation.

Le processus de transformation par requêtes se fait en deux phases :

- La sélection d'une transformation parmi l'ensemble de transformations déclarées.
- L'application d'un ensemble de règles de génération d'un document cible.

Des formes de requêtes différentes interviennent lors de ces deux phases : les requêtes utilisées par la première permettent la reconnaissance d'une structure particulière de l'instance source. Ces requêtes se comportent comme des filtres sur la structure du document. La seconde phase utilise des requêtes pour extraire l'information du document source pour l'intégrer dans le document cible.

Selon les systèmes de transformation, une importance différente est donnée à ces deux phases. Certains systèmes (IDM, Balise) ne proposent que la phase d'application des règles de génération, celle–ci se faisant de façon séquentielle et inconditionnelle. Si la génération se fait de manière linéaire, les déclarations sont ordonnées dans l'ordre de la structure du document produit. Une telle transformation peut être exprimée à l'aide du langage de script du système IDM présenté dans la section III.3.3.1.

Si le système utilise les requêtes pour faire une sélection des règles à appliquer, l'ensemble des requêtes est évalué et les règles associées à celles retournant un résultat sont appliquées. L'ordre d'application de ces règles peut varier d'un système à l'autre. Cet ordre peut être :

- donné dans l'expression : les règles *priority–expression* de DSSSL [ISO 94] permettent de définir un ordre de priorité sur les requêtes ; les expressions de transformation de Scrimshaw contiennent aussi l'ordre d'application dans leur description ;
- implicite : XSL [Clark 98] applique en priorité les règles associées aux requêtes les plus spécifiques ;
- celui de la structure du document source, comme dans le système de transformation d'Amaya (c.f. section III.4.3).
- celui de la structure du document à construire : les langages SgmlQL et Cost 2 permettent de décrire des transformations qui utilisent des requêtes sur le document source.

III.3.2 Langages de transformation

Les transformations explicites sont exprimées soit dans un langage spécifique soit dans un langage de programmation classique.

Les langages de transformation peuvent être de nature déclarative ou impérative :

- Les systèmes se basant sur une expression déclarative ne proposent que le mode de transformation dirigé par la source. La conduite de la transformation est fixée par le moteur et ne peut pas être spécifiée par l'expression. En contrepartie, ces langages permettent une expression plus concise et plus lisible des transformations. C'est le cas du langage de script d'IDM [Digithome], de Chameleon [Mamrak 89] ou de XSL [Clark 98].
- Les langages impératifs permettent de définir des procédures de transformation et de piloter le flot d'exécution de la transformation. Les systèmes basés sur de tels langages permettent de développer des transformations sur le modèle dirigé par la source (Balise [AIS 96], CoST 2 [English 96a]) ou sur le modèle par requêtes (SgmlQL [Le Maître 98], Scrimshaw[Arnon 93]).

Certains systèmes utilisant une expression déclarative proposent une interface graphique pour spécifier les transformations. C'est le cas par exemple de Chameleon qui permet d'associer de manière interactive les types d'éléments d'une structure générique source avec ceux d'une structure cible.

III.3.3 Revue des systèmes de transformation explicite

Cette section décrit plusieurs systèmes de transformation explicite couvrant l'ensemble des méthodes et utilisant les formes de langages exposées précédemment. Nous illustrerons les langages associés à ces systèmes en donnant l'expression de la transformation des documents *mail* en documents *message* décrite au début de ce chapitre.

Nous présentons d'abord les systèmes de transformation utilisant des langages ayant une plus forte composante impérative. Puis nous décrirons des systèmes intégrant une approche plus déclarative.

Le tableau de la figure 19 recense les systèmes présentés dans la suite et les caractéristiques des méthodes de transformation qu'ils utilisent.

Système	Méthode de transformation		Langage
	Ordre d'application des règles	Génération des éléments cibles	
IDM	source ou requêtes	mixte	impératif
Balise	source ou requêtes	mixte	impératif
Cost / STIL	source ou requêtes	linéaire	impératif
DSSSL	requêtes	arborescente	fonctionnel
XSL	requêtes	arborescente	déclaratif + impératif
Chameleon	source	linéaire	déclaratif
SgmlQL	requêtes	arborescente	déclaratif
Scrimshaw	requêtes	linéaire	déclaratif

Figure 19 : Caractéristiques des systèmes de transformation explicites

III.3.3.1 IDM

IDM (Intelligent Document Manager [Digithome]), développé par la société Digithome, est un environnement de manipulation de documents électroniques orienté vers la transformation de documents structurés. Cet environnement fournit une interface de programmation (API) permettant de programmer facilement des filtres de conversion de documents structurés vers des formats divers (HTML, LaTeX, Microsoft Windows Help).

L'API IDM permet de localiser des éléments de structure en utilisant des requêtes sur les ancêtres et les descendants d'un nœud, sur les voisins, sur la présence et la valeur des attributs, ou encore sur le contenu des éléments textuels. L'API permet d'implémenter des transformations procédurales et dirigées par la destination. Les primitives de l'API sont fournies par un ensemble de classes C++, dont la classe DI qui fournit les méthodes de sélection de nœud ou la classe IDM qui contient les méthodes de lecture du document source et d'écriture sur le flot de sortie.

L'exemple suivant montre la transformation du document *mail* en *message* et illustre ainsi la méthode de transformation par requêtes avec une génération linéaire des éléments cibles.

Les requêtes utilisent les méthodes de la classe `DI` préfixées par `Move` pour déplacer le nœud courant dans la structure. La méthode MoveB permet de déplacer la sélection courante vers son premier fils, MoveT permet de la déplacer sur le parent, MoveR et MoveL permet de la déplacer vers l'élément suivant ou précédent. La partie génération utilise les méthodes `DI->GetData` pour accéder au contenu des éléments et `IDM->Progress` pour écrire sur le flot de sortie le résultat de la transformation.

```
DI->MoveTT();                    // accès à l'élément racine
DI->MoveB();                     // accès à l'élément date
IDM->Progress ("<message date=\"%s\">", DI->GetData());

DI->MoveT();                     // accès à l'élément mail
IDM->Progress  ("<référence>%s</référence>\n",
                DI->GetAttrValue ("ref"));

DI->MoveB();                    //accès à l'élément sender
DI->MoveR();
DI->MoveR();
IDM->Progress  ("<expéditeur>%s</expéditeur>\n",
                 DI->GetData());

DI->MoveL();                    //accès à l'élément recipient
IDM->Progress  ("<destinataire>%s</destinataire>\n",
                 DI->GetData());

IDM->Progress  ("<contenu>\n");

DI->MoveR();
DI->MoveR();                     // accès à l'élément subject
IDM->Progress  ("<titre>%s</titre>\n",
                DI->GetData());

DI->MoveR();                     // accès à l'élément textbody
IDM->Progress  ("<corps>\n");
DI->MoveB();
do
   if (DI->IsGI("cite") == IDTRUE)
       IDM->Progress ("« %s »", DI->GetData);
   else
       IDM->Progress ("%s", DI->GetData);
while (DI->MoveNext == IDTRUE)
```

Dans cet exemple nous supposons que la structure du document source est correcte et que tous les éléments de cette structure sont présents. Un traitement supplémentaire serait nécessaire pour vérifier l'existence de chacun de ces éléments.

IDM propose de plus un langage de script qui permet de spécifier des transformations dirigées par la source. La déclaration de la transformation est alors plus concise, mais ne permet pas de réaliser tous les aspects de la

transformation de l'exemple : la spécification suivante ne remplit que les critères 1, 2 et 3 énoncés dans la section III.1 :

```
START_mail
   Progress ("<message>")
END_mail
   Progress ("</message>")
START_recipient
   Progress ("<destinataire>")
END_recipient
   Progress ("</destinataire>")
START_sender
   Progress ("<expéditeur>")
END_sender
   Progress ("</expéditeur>")
START_subject
   Progress ("<contenu> <titre>")
END_subject
   Progress ("</titre>")
START_textbody
   Progress ("<corps>")
END_textbody
   Progress ("</corps> </contenu>")
START_cite
   Progress ("«")
END_cite
   Progress ("»")
!DATA
   Progress ("$DATA")
```

Avec ce langage de script déclaratif, la génération de ce document est également linéaire. Par conséquent, la transformation dirigée par la source, spécifiée par le langage de script ne permet pas de modifier l'ordre des éléments (critères 4, 5 et 6).

Grâce à sa nature procédurale, IDM permet des transformations complexes de documents structurés entre DTD ainsi que vers de nombreux formats de documents. Néanmoins l'utilisation de cet outil requiert une bonne pratique de la programmation et de l'algorithmique pour mettre en œuvre ces transformations. Le langage de script est très simple, mais ne permet d'effectuer que des transformations simples : la génération de l'instance cible est linéaire, et les traitements sont associés à un type d'élément indépendamment du contexte dans lequel ils apparaissent.

III.3.3.2 Balise

Balise [AIS 96] est un environnement de développement d'applications documentaires basées sur les standards SGML et XML. Cet environnement comprend un langage de programmation. Il est développé par la société AIS Berger–Levrault. Le langage Balise permet de manipuler la représentation arborescente des documents à travers des fonctions d'accès et de modification de la structure. Ces fonctions permettent soit de modifier la structure d'un

document existant, comme montré ci–dessous, soit de générer un nouveau document en insérant les éléments créés dans une nouvelle arborescence comme le montre le deuxième exemple. Les instructions de contrôle du langage Balise sont les mêmes que celles d'un langage de programmation impératif classique (Pascal, C).

Ce langage fournit également des primitives de création et de copie d'éléments. Ces fonctions permettent de programmer un grand nombre de transformations. Une fonction transformant des documents *mail* en *message* s'écrit :

```
function Mail2Message (doc) {
  var elem = root (doc);          // accès à l'élément racine
  changeGI (elem, "message");     // change le type du noeud
                                  // mail

  var reg = attr ("ref", elem);  // accès à l'attribut ref
  var newelem = Move (soc, "référence", ref);
  insertSubTree (elem, 1, newelem); //cree l'élément ref

  elem = firstChild (elem);       // accès à l'élément date
  addAttr (root(doc), date, content (elem));

  elem = rightSibling (elem);     // accès à l'élément
   recipient
  changeGI (elem, "destinataire");

  elem = rightSibling (elem);     // accès à l'élément sender
  changeGI (elem, "expéditeur");
  CutSubTree (elem);              // déplace l'élément
                                  // expéditeur
  insertSubTree (root(doc), 1, elem);

  elem = rightSibling (elem);     // accès à l'élément title
  changeGI (elem, "titre");
  var body = rightSibling (elem); //accès à l'élément
   textbody
  cutSubtree (elem);              // déconnecte le nœud titre
  insertSubtree (body, 0, elem); // insère titre avant le
                                  // 1er fils de body
  changeGI (body, "contenu");     // change body en contenu
  elem = Node (doc, "corps");     // crée le noeud corps
  insertSubtree (body, 1, elem); // insère corps après le
                                  // 1er fils de contenu
  var elpar = rightSibling (body);
  while (elpar != nothing) {      // parcourt les éléments p
    cutSubtree (elpar);           // déplace chaque p dans
    insertSubtree (elpar, -1, body) // l'élément corps
    changeGI (elpar, Para);
    for elem in searchElemNodes (elpar, "cite") {
                                  // génère les guillemets
      changeContent (elem, format ("«%s»", content(elem)));
      flattenSubTree (elem);      // retire les éléments cite
    }
  }
}
```

L'algorithme utilisé dans cet exemple parcourt le document source en appliquant des traitements spécifiques aux éléments rencontrés. Ces traitements permettent d'implémenter les six aspects énoncés dans l'exemple. La nature

impérative du langage Balise impose une description complète de la transformation. En particulier, pour ignorer les éléments *cite*, il faut explicitement les retirer (fonction `flattenSubTree`) ou rechercher leur contenu d'une manière spécifique.

Balise permet également de spécifier des transformations dirigées par la source. Pour cela, Balise permet d'associer des traitements aux événements de l'analyseur XML intégré. Des traitements sont associés aux événements de début et de fin pour les éléments, des clauses contextuelles peuvent être ajoutées à cette définition (par exemple : début des éléments de type *p* se trouvant à l'intérieur d'un élément *textbody*). Pour limiter l'occupation de la mémoire lors de la transformation, le contexte est restreint aux ancêtres et aux voisins précédents de l'élément concerné par l'événement. Nous donnons ci–dessous un extrait d'une transformation dirigée par la source exprimée dans le langage Balise. Cette spécification utilise le style déclaratif pour associer des traitements aux événements de l'analyseur, ces traitements utilisant un langage procédural pour générer l'arborescence du document cible.

```
//initialisation
main {
   eventLookAhead(true);
}
// transformation de l'élément mail
element mail {
  on start {
     changeGI ("message");
     var root = Elem ("mail");
           }
  }
// transformation de l'élément date en attribut
element date {
  on start {
      var ladate = content (currentNode());
      AddAttr (root, "date", ladate);
           }
  }
// transformation de l'élément recipient
element recipient {
  on start {
      changeGI ("destinataire");
      // insère expéditeur comme premier fils de body
      cutTree (currentNode());
      insertSubTree (root, 0, currentNode());
           }
  }
// transformation de l'élément sender
element sender {
   on start {
       changeGI ("expéditeur");
       // insère expéditeur comme premier fils de body
       cutTree (currentNode());
       insertSubTree (root, 0, currentNode());
            }
  }
// transformation de l'élément subject
```

```
element suject {
     on start {
         changeGI ("titre");
        // crée l'élément contenu
        // et l'insère comme dernier fils de body
        var elcontenu = Node ("contenu");
        insertSubTree (root, -1, elcontenu);
        // insere titre dans contenu
        insertSubTree (elcontenu, 0, currentNode());
        // crée l'élément corps
        // et l'insère comme dernier fils de contenu
        var elcorps = Node ("corps");
        insertSubTree (elcontenu, -1, elcorps);
             }
  }
// transformation des éléments para
element p {
   on end {
     changeGI ("para");
     cutTree (currentNode());
     insertSubTree (elcontenu, -1, currentNode());
   }
// élimination des éléments cite, génération des guillemets
element cite {
   on begin {
    changeContent (currentNode(), format ("«%s»",
    content(currentNode())));
    flattenSubTree();
             }
   }
```

Balise, avec l'approche dirigée par la source, se différencie du langage de script d'IDM par le fait que la génération ne se fait plus de manière linéaire, mais par la construction d'une structure arborescente du document. Cette approche fournit une souplesse accrue dans la façon de placer les éléments cibles. De plus elle n'impose pas la génération des balises de début et de fin d'élément. En contrepartie, elle ne permet de produire que des documents structurés au format SGML ou XML. De plus, ce mode de génération nécessite la représentation en mémoire de l'arbre du document cible qui peut être importante.

La combinaison de la transformation dirigée par la source avec la modification de la structure du document source peut produire des effets de bord négatifs : le traitement d'un événement peut entraîner des modifications dans la structure du document et conduire à l'omission de sous–arbres entiers lors de la transformation. C'est pour éviter ces effets de bord que Balise permet de dupliquer préalablement la structure du document source (avec la fonction `DumpSubTree`) pour obtenir une instance servant à la génération du document, l'autre étant modifiée par les traitements de ces événements.

En conclusion, comme pour IDM, la composante impérative du langage Balise nécessite une connaissance de la programmation. Par contre, son approche déclarative permet la spécification de transformations plus complexes que celles

d'IDM grâce à la possibilité de sélectionner des ancêtres lors de l'association d'une règle à un type d'élément et à la génération arborescente.

III.3.3.3 CoST 2

CoST 2 [English 96a] (Copenhagen SGML Tool) est un outil qui permet de spécifier des transformations de documents structurés en Tcl. Cet outil est un logiciel libre développé par la société ART (Advanced Rotorcraft Technology, Inc.) Ce langage intègre de puissantes fonctions de traitement de la composante textuelle des documents telles que la reconnaissance d'expressions régulières et leur remplacement. Cet outil de transformation est associé au parser sgmls [Clark].

CoST 2 propose une méthode de transformation par requêtes. Celles–ci servent à sélectionner les éléments du document source qui doivent être transformés mais également à identifier ceux qui sont utilisés lors de la génération.

Les requêtes de sélection des éléments source sont définies par les mots clés `foreachNode` et `withNode` associés à un traitement Tcl. Le traitement est appliqué au premier élément ou à l'ensemble des éléments respectant la requête. Lors de l'application du traitement, l'élément ayant satisfait la requête est appelé élément courant.

Les traitements sont des scripts Tcl et permettent d'écrire de façon linéaire le document cible sur un flot de sortie. Ces traitements peuvent utiliser des requêtes sur le document source pour en extraire le contenu. Ces requêtes permettent d'accéder aux éléments environnant l'élément courant (ancêtres, voisins et descendants), aux attributs des éléments et au contenu des feuilles.

La transformation de *mail* en *message* à l'aide de CoST 2 s'exprime de la façon suivante :

```
puts "<message date=\""
withNode withgi date {puts "[query content]"}
puts "\">"
puts "<reference>"
withNode withgi message hasatt ref {
  puts "[query attval ref]"
  }
puts  "</référence><expéditeur>"
withNode withgi sender {puts "[query content]"}
puts "</expéditeur><destinataire>"
withNode withgi sender {puts "[query content]"}
puts  "</destinataire><contenu><titre>"
withNode withgi title {puts "[query content]"}
puts  "</titre><corps>"
foreachNode withgi p {
   puts "<para>"
   puts "[query content]"
   puts "</para>
   }
puts  "</corps></contenu></message>"
```

Les requêtes d'extraction du contenu sont introduites par le mot–clé `query`, tandis que les requêtes de sélection sont illustrées ici par le mot–clé `foreachNode`. La transformation présentée ci–dessus ne permet pas de traduire les element *cite* de la structure source, car la composition des requêtes n'est pas suffisamment expressive et l'opérateur `foreachNode` ne permet pas de différencier le traitement à associer au début, au contenu et à la fin de chaque élément. Pour pallier ce manque d'expressivité, CoST 2 permet d'ordonnancer les traitements selon la structure du document source en les associant aux événements de l'analyseur. Les actions TCL utilisant les requêtes sont alors associées aux événements de début et de fin d'élément.

Cet outil, profitant de la modularité apportée par le langage de script Tcl permet le développement de modules, ce qui conduit à une expression simple mais limitée des transformations. Parmi ces modules, on peut citer Simple Cost, qui permet une spécification simple de transformations dirigées par la source avec une génération linéaire par écriture sur le flot de sortie, RatFink [English 96b] qui permet la spécification de transformations de documents structurés SGML en documents au format RTF.

STIL [Schrod 95] et SGMLSpm [Megginson 96] sont des outils de transformation qui utilisent des principes similaires à ceux de CoST. STIL est développé à l'université de Darmstadt, il utilise le langage Lisp pour la description des requêtes. SGMLSpm développé par David Megginson de l'université d'Ottawa est un module perl permettant l'expression de requêtes et de règles de transformation similaires à celles de CoST2.

III.3.3.4 Exrep

Exrep [Lambolez 95] est un outil de réécriture permettant de traiter un flot de texte en entrée et d'y appliquer un ensemble de règles de réécriture. Exrep est particulièrement adapté aux domaine de l'extrction et de la réutilisation de contenu de document.

Chaque règle de réécriture est formée d'une expression régulière permettant la reconnaissance d'une sous–chaîne dans la chaîne d'entrée, et d'une chaîne de remplacement se substituant à la sous–chaîne reconnue par l'expression régulière.

Les apport de Exrep sur les outils de réecritures standards d'Unix, tels que lex [Levine 92] et awk [Dougherty 92] sont :

- la présence de dictionnaires, permettant de gérer des ensembles de sysnonymes et des glossaires. Cette propriété est particulièrement utile pour traiter le contenu textuel d'un document ;

- l'indépendance de l'outil par rapport à la structure de lignes du flot d'entrée ; les caractères de fin de ligne sont traités comme les autres, permettant la reconnaisssance de sous chaînes divisées en plusieurs lignes.
- la possibilité d'associer un contexte à un ensembe de règles de réécriture.

A cause de sa généricité lui permettant de traiter tout type de format de document, Exrep ne propose pas de primitive permettant de gérer facilement les balises SGML ou XML. La spécification d'une transformation doit donc contenir les expressions régulières permettant de rcoonaitre la syntaxe des balises de marquage.

Les transformations proposées par le système Exrep peuvent être considérées comme des transformations par requêtes avec génération linéaire. Ces transformations permettent le remplacement des balises (la substitution de <mail> par <message>, le remplacement de l'élément <date> par un attribut ou bien de l'attribut ref par un élément). Les transformations permettent l'insertion de balises en fonction du contenu d'un élément (remplacement des guillemets par l'élément cite). La principale limitiation d'Exrep est de ne pas permettre le déplacement d'éléments dans la structure, des systèmes tels que Scrimshaw (III.3.3.9) permettent de manipuler des variables permettant la mémorisation d'éléments pour leur réutilisation au cours de la transformation, ce qui permet les déplacement et leur réplication.

III.3.3.5 Chameleon

Chameleon [Mamrak 89] est un outil interactif de définition de structures génériques et de transformation entre formats de documents. Chaméleon, aussi appelé ICA pour Integrated Chameleon Architecture a été développé à l'université de l'Ohio en coopération avec la compagnie HaL Computer Systems, Inc.

SGML est utilisé par Chameleon comme format de description générique des documents et sert de pivot entre les documents de différents formats (figure 20).

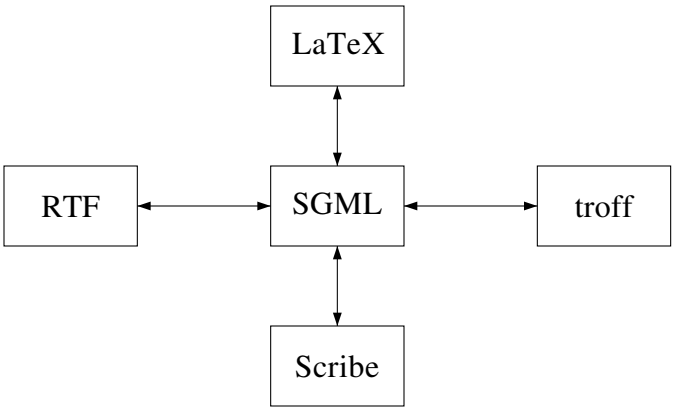


Figure 20 : Utilisation de SGML comme format pivot pour la transformation entre formats de documents

L'information de formatage des documents originaux est d'abord convertie sous forme de balises SGML. Les documents ainsi marqués sont transformés vers une structure générique SGML, enfin, ces documents sont exportés vers le format de destination voulu.

Ces transformations successives sont spécifiées par plusieurs modules de Chameleon (cf. figure 21) :

- *Devegram* permet de créer des grammaires et des DTD pour la structure générique SGML.
- *retag* permet de remplacer les balises de structure et/ou de formatage des documents.
- *intag* est complémentaire à retag et permet d'insérer les informations dans le document quand celles−ci sont requises par la DTD.
- *Spec2gen* permet de spécifier la correspondance entre le marquage spécifique et la structure générique.
- *Gen2spec* permet d'exporter un document SGML vers le format de sortie.

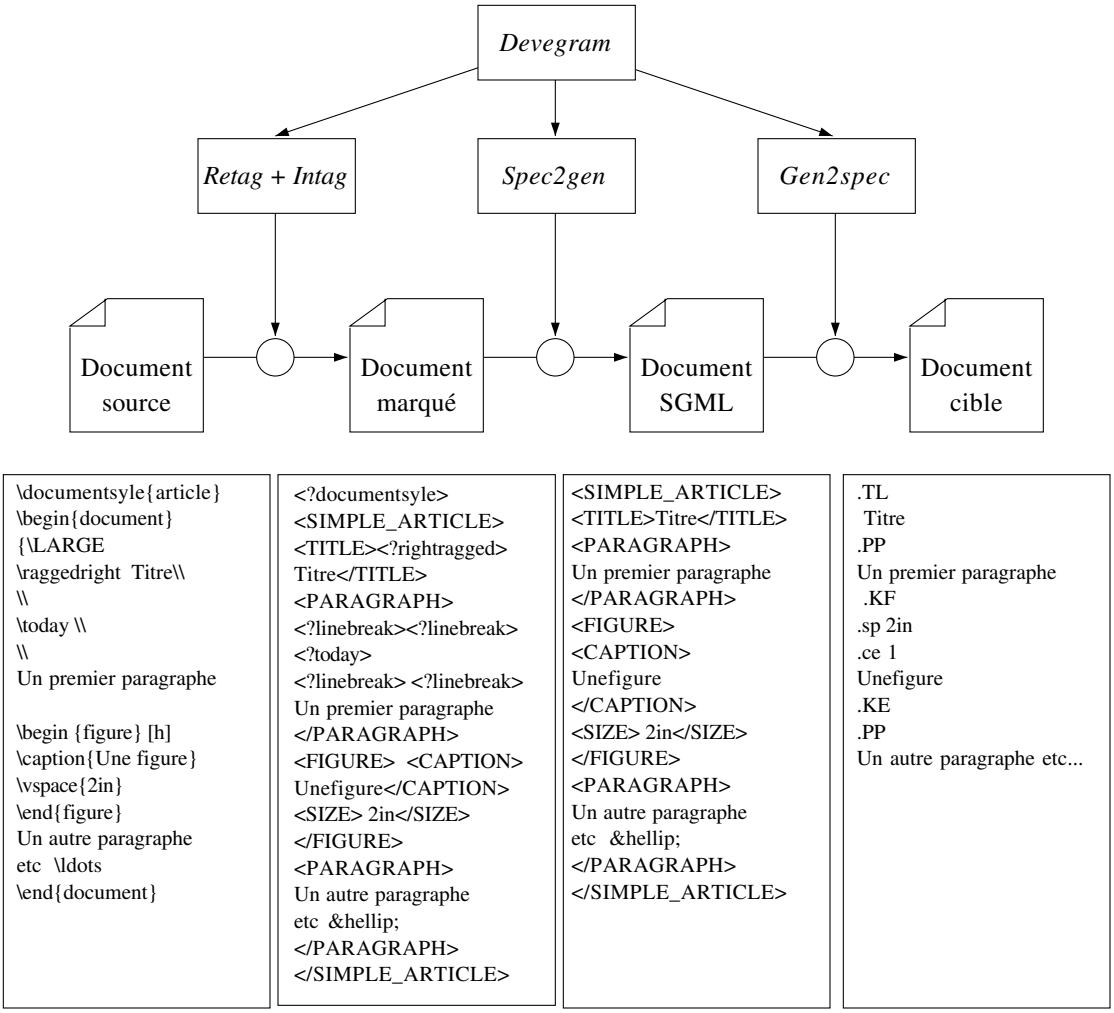


Figure 21 : Étapes de la transformation d'un document LaTeX en Troff avec ICA

Chaque module permet la génération de filtres, représentés par des cercles sur la figure 21, qui permettent le passage du document d'un état dans un autre (source, marqué, SGML, cible).

L'originalité de ce système est l'utilisation de SGML comme format central pour la conversion de documents. Cela permet de s'appuyer sur des DTD existantes pour la représentation commune des formats source et cible. Pour utiliser cette méthode de transformation, deux étapes sont nécessaires : la structuration permet de représenter le document d'origine sous une forme structurée, l'export permet de traduire le document structuré dans le format de destination.

Les modules de spécification de transformations s'appuient sur les outils d'analyse syntaxique Lex et Yacc. Les modules *Retag*, *Intag* et *Gen2spec* ne permettent que de spécifier des réécritures de l'information de formatage ou de structure associée aux documents. Le module *Spec2gen* est le plus intéressant

du point de vue de la transformation car c'est lui qui permet les opérations de manipulation de structure des documents.

Le module *Spec2gen* permet l'association des balises du document source avec les types définis dans la DTD SGML. Cette association est décrite à l'aide d'expressions régulières permettant d'exprimer l'organisation des balises sources. Certaines balises peuvent être perdues lors de cette étape car la structure générique SGML ne permet généralement pas de représenter toutes les informations nécessaires à la présentation du document. Dans l'exemple ci–dessus, le document est découpé en paragraphes, mais les sauts de ligne sont perdus.

Les expressions régulières permettent d'associer des règles de transformation (en l'occurrence un type d'élément SGML) à une séquence de balises dans le document marqué. Ces séquences d'éléments sont exprimées en termes de listes, d'alternatives et peuvent contenir des éléments optionnels. Les expressions régulières ne permettent pas d'exprimer la hiérarchie des balises source ; elles sont donc adaptées pour spécifier la transformation d'un ensemble de balises sans relations hiérarchiques, mais il est nécessaire d'étendre leur syntaxe et leur sémantique pour les utiliser dans un contexte de documents structurés (voir section III.4.2.1).

III.3.3.6 DSSSL

DSSSL [ISO 94] est une norme de l'ISO pour définir la sémantique ou le style des documents SGML. DSSSL comprend deux parties : STTP (SGML Tree Transformation Process) et SSP (Semantic Specific Process), c'est la partie STTP qui assure les transformations de structure. Le but de la transformation de structure proposée par DSSSL est l'adaptation des documents SGML aux différents support de présentation.

Les spécifications de transformations sont exprimées dans le langage *scheme* et décrivent l'association entre les types de la DTD d'origine et ceux de la DTD destination. Ces spécifications d'associations sont composées de trois parties :

- *query−expression* est l'expression d'une requête destinée à la sélection des nœuds de l'arbre source concernés par la transformation.
- *transform−expression* est l'expression de la transformation évaluée pour chaque nœud sélectionné par la requête. Cette expression rend comme résultat la description du(des) nœud(s) de l'arbre destination correspondant au nœud sélectionné.

- *priority−expression* est un composant optionnel qui spécifie quelles expressions de transformation sont utilisées lorsqu'un nœud de l'arbre répond à plusieurs requêtes.

La norme DSSSL, bien que datant de 1994, n'a connu qu'une implémentation partielle : Jade [Clark], associée à l'analyseur sgmls. La proposition XSL présentée dans la suite joue un rôle similaire à celui de DSSSL pour les documents XML.

III.3.3.7 XSL

XSL [Clark 98] est une proposition du consortium W3C pour la définition de feuilles de styles (stylesheets) pour les documents XML. Le langage XSL comprend deux parties :

- un langage de transformation auquel nous nous intéressons particulièrement ;
- une syntaxe de spécification de formatage des documents.

Le langage XML utilise le mécanisme des NameSpaces [Bray 98b] qui est également une proposition du consortium W3C. Les namespaces permettent de définir des espaces de noms dans les documents : les noms de types d'éléments sont précédés d'un préfixe indiquant à quel espace de nom ils appartiennent. XSL utilise l'espace de nom *xsl* pour identifier les directives de transformation des éléments de structure crées par la transformation.

Une feuille de style est composé d'un ensemble d'expressions (nommés *template*s) décrivant chacune une transformation élémentaire. Chaque *template* est représenté par un élément XML du type *xsl:template*.

Le modèle de transformation de XSL repose sur des requêtes spécifiant la structure des éléments source. Une requête permet de sélectionner un élément en fonction de son contexte ascendant (parents, ancêtres), la présence et la valeur des attributs des éléments constituant ce contexte.

Les requêtes XSL sont contenues par l'attribut *match* des éléments *template*.

Le contenu d'un élément *template* est constitué de l'ensemble des éléments qui sont créés dans le document cible par la transformation et de directives XSL qui permettent un traitement lors de la transformation. Ces directives permettent :

- d'insérer des éléments texte ;
- d'utiliser des compteurs ;

- d'appliquer une autre transformation sur d'autres éléments ;
- de traiter des éléments textuels, en particulier les valeurs des attributs ;
- de contrôler l'exécution de la transformation avec :
  - des instructions conditionnelles (if...then...else) portent sur une requête ;
  - des boucles (for each) permettant de traiter de façon itérative tous les éléments reconnus par une requête.

Lorsqu'une feuille de style XSL est appliquée, le template dont la requête identifie l'élément racine de la structure source est d'abord appliqué. ce template peut explicitement provoquer l'application des autres templates à d'autre éléments de la structure source.

La feuille de style suivante permet de transformer un document *mail* en *message* :

```
<xsl:stylesheet>

 <!-- transformation de l'élémnt mail         -->
 <xsl:template match="mail">
  <message date="{date}">
   <reference>
    <xsl:value-of expr="attribute(ref)"/>
   </reference>
   <!-- traite les fils de mail dans l'ordre où
        ils apparaissent dans la structure
       message                                 -->
   <xsl:process select="sender"/>
   <xsl:process select="recipient"/>
   <xsl:process select="textbody"/>
  </message>
 </xsl:template>

 <!-- transformation de l'élément sender       -->
 <xsl:template match="sender">
  <expéditeur>
   <xsl:process-children/>
  </expéditeur>
 </xsl:template>

 <!-- transformation de l'élément recipient    -->
 <xsl:template match="recipient">
  <destinataire>
   <xsl:process-children/>
  </destinataire>
 </xsl:template>

 <!-- transformation de l'élément subject      -->
 <xsl:template match="subject">
  <titre>
   <xsl:process-children/>
  </titre>
 </xsl:template>

 <!-- transformation de l'élément textboby     -->
```

```
<xsl:template match="textbody">
 <contenu>
  <!-- transformation de l'élément subject    -->
   <xsl:process select="ancestor(mail)/subject"/>
  <corps>
    <xsl:process-children/>
  </corps>
 </contenu>
</xsl:template>

<!-- traite l'élément p -->
<xsl:template match="p">
 <para>
   <xsl:process-children/>
 </para>
</xsl:template>

<!--transformation de l'élément cite          -->
<xsl:template match="cite">
 «
  <xsl:process-children/>
 »
 </xsl:template>
</xsl:stylesheet>
```

XSL permet de traiter tous les aspects de la transformation de *mail* en *message* présentée dans la section III.1. Ce langage, encore en cours de définition, est parmi ceux présentés ici celui permet d'exprimer les transformations les plus complexes. Pour cela, il intègre la méthode transformation par requêtes, une construction descriptive de la structure cible et des instructions de contrôle permettant de guider la transformation.

La norme XSL étant destinée à la transformation de document pour produire une structure proche de son image physique il a été conçu pour la transformation des documents dans leur intégralité. Cependant, les déclarations XSL sont modulaires : une transformation est décrite par un ensemble de templates qui portent chacun sur un sous–ensemble du document source. Les templates peuvent donc être utilisés pour une transformation locale et faire partie de la spécification d'une transformation plus générale.

La contrepartie de cette déclaration modulaire est la nécessité de spécifier la façon de transformer chaque élément du document source.

III.3.3.8 SgmlQL

SgmlQL [Le Maître 95], [Le Maître 98] est un système de transformation fondé sur un langage reprenant les principes du langage d'interrogation de bases de données SQL. Ce système est développé au Laboratoire Parole et Langages (LPL) à l'université de Toulon. Ce système permet la construction de documents à partir de l'information contenue dans une base de documents SGML.

Chaque script SgmlQL décrit la construction d'un document SGML en utilisant des opérateurs de construction de document, d'élément et d'attribut. Le langage permet d'évaluer des requêtes construites avec les opérateurs SQL (`Empty, Count, Sum, Indexing, Concatenation, Shrink, Exists, Forall, Select... from... where, Transpose... from... where`) mais appliquées à des documents SGML au lieu de bases de données relationnelles.

SgmlQL permet par exemple d'extraire l'information contenue dans un ensemble de messages dont la structure est décrite dans notre exemple. Supposons que le fichier messages.sgml contienne un ensemble de messages. On affecte son contenu à une variable globale par l'expression :

```
global $mess = file "messages.sgml"
```

Les requêtes utilisent cette variable globale pour en interroger le contenu :

Nombre de messages

```
count (every message within $mess)
```

Les dix premiers messages

```
(every message within $mess)[1:10]
```

Tous les messages sont–ils adressés à Cécile ?

```
forall $m in
(every destinataire within $mess):
text($m) match "Cécile"
```

L'ensemble des messages adressés à Cécile

```
select
  every message within $m
from
  $m in $mess
where
  text (first destinataire within
  $m) match "Cécile"
```

Le résultat des requêtes peut être appliqué aux opérateurs de construction d'éléments pour que le script produise un document SGML. Le script permettant de convertir le document *mail* en *message* s'écrit :

```
document docdtd: "message.dtd"
  body:
   element MESSAGE
    attr:{DATE = text(first DATE within $mess)}
    content: [
     element REFERENCE
      content:[(first MAIL within $mess)->REF],
     element EXPEDITEUR
      content:[content(first SENDER within $mess)],
     element DESTINATAIRE
      content:[content(first RECIPIENT within $mess)],
     element CONTENU
      content:[
       element TITRE
        content:[content(first TITLE within $mess)],
       element CORPS
        content:[replace every P as $p
```

```
                within $mess
                by element PARA
                    content: content (
                         replace every CITE as $c
                         within $p
                         by "« ".text($c)." »")
              ]
            ]
          ]
```

La transformation de *mail* en *message* n'est pas la plus adaptée pour montrer les possibilités du langage SgmlQL, qui est conçu pour la construction de documents à partir d'une collection de documents SGML. Ce système répond à un besoin de transformation assez différent de celui concerné par notre problématique : l'utilisation et la génération de documents structurés à partir d'information provenant de sources diverses (bases de données, serveurs d'information, moteurs de recherche, World Wide Web, etc.).

Néanmoins l'exemple ci–dessus montre que les expressions SgmlQL sont compactes. De plus ce système permet de mettre en œuvre tous les aspects de la transformation donnés en III.1.

En contrepartie, le temps d'évaluation des requêtes qui est un facteur déterminant pour l'utilisation d'une telle approche dans un éditeur interactif est important pour les requêtes du type SQL. L'approche de la transformation proposée par ce système est plus adaptée au traitement automatique de bases de données documentaires qu'à la transformation d'instances au coup par coup, telle qu'elle est utilisée dans les éditeurs interactifs.

III.3.3.9 Scrimshaw

Scrimshaw [Arnon 93] est un langage de pattern–matching entre arbres permettant de spécifier des transformations de documents structurés. Ce langage se fonde sur une approche requêtes/action et une génération linéaire du document cible.

Scrimshaw permet de manipuler des variables pouvant contenir des listes de sous–arbres de la structure des documents. Ces variables sont affectées par le résultat de requêtes sur la structure du document. Les requêtes utilisent des opérateurs de pattern–matching classiques appliqués au document source :

- *grep* {PATTERN–>EXPRESSION} recherche la première occurrence de PATTERN dans l'ordre préfixe du document, et lorsqu'une occurrence est trouvée, l'expression est évaluée et son résultat est copié sur le flot de sortie.

Dans la partie PATTERN associée à un opérateur grep, il est possible d'assigner des sous–parties à des variables. Ces variables peuvent être utilisées dans la partie EXPRESSION de la règle.

- *grepSearch* {PATTERN–>EXPRESSION} recherche toutes les occurrences de PATTERN dans la structure du document et les copie sur le flot de sortie.
- *map* et *mapSearch* sont identiques à *grep* et *grepSearch* à la différence que tous les éléments du flot d'entrée ne coïncidant pas avec la partie PATTERN sont recopiés sur le flot de sortie.

Scrimshaw fournit également un mécanisme de *pipeline* permettant d'appliquer plusieurs transformations successives à une même instance.

La transformation de *mail* en *message* dans le langage Scrimshaw s'exprime de la façon suivante :

```
{mail [
   date [<vdate:#>]
   recipient [<vrec:#>]
   sender [<vsend:#>]
   title [<vtitle:#>]
   textbody [
      <vlp:map[{p[<vp:#*>]->para[<vp>]}]
         ||mapSearch{cite [vcite:#*] -> "«",<vcite>,"»"}]
     >
            ]
        ] -> message [ expediteur [<vsend>]
                       destinataire [<vrec>]
                       contenu [
                         titre [<vtitle>]
                         corps [<vlp>]
                               ]
                     ]
}
```

La partie requête de cette transformation assigne respectivement les variables `vrec`, `vsent` et `vtitle` au contenu des éléments *recipient*, *sender* et *title* du document d'origine. La requête traitant les paragraphes est plus complexe : un opérateur `map` change d'abord les éléments *p* en *para* puis au résultat de cette transformation est appliqué un opérateur `mapSearch` permettant de remplacer les éléments *cite* par des guillemets. Le résultat de la transformation de la suite de paragraphes constituant le corps du *mail* est assigné à la variable `vlp`.

Scrimshaw est limité à la transformation des éléments et ne permet pas de spécifier des transformations impliquant les attributs. Cette limitation ne permet pas de prendre en compte les éléments *date* et *reference* dans la transformation de *mail* en *message*.

La partie génération de la transformation est une déclaration de la structure du document cible en utilisant les variables initialisées dans la partie requête.

Au delà de la syntaxe utilisée, le principe de transformation proposé par Scrimshaw se différencie de celui de DSSSL par la façon de générer le document cible. Au lieu d'utiliser les expressions de priorité pour définir l'ordre des transformations, Scrimshaw permet d'intégrer les opérateurs de pattern–matching à l'intérieur des requêtes. Ainsi les transformations peuvent être faites « à la volée » lors du pattern–matching (opérateurs `map` et `mapsearch` dans l'exemple). Le mécanisme de pipeline permet également de contrôler l'ordre dans lequel sont effectuées les transformations.

III.3.4 Synthèse

La diversité des systèmes de transformation explicite vient du fait qu'ils répondent à des besoins applicatifs différents. Nous identifions ici les relations entre les méthodes de transformation et le contexte applicatif (voir section II.4) dans lequel elles sont utilisées :

- Certaines bibliothèques de développement d'applications de documents structurés (Balise, IDM, Thot) intègrent des services de transformation. Ces outils proposent des primitives génériques permettant de programmer des transformations adaptées aux applications qu'ils permettent de développer. Cependant ils nécessitent un effort de développement important et ne peuvent pas être ré–utilisés dans un contexte autre que celui pour lequel ils ont été développés, parce que le système de transformation est intégré à l'application qui l'utilise.

  Ces systèmes peuvent également proposer des facilités pour spécifier des transformations dirigées par la source en reproduisant le comportement d'un parser (langage de script d'IDM, mécanisme d'événements de Balise). Cette possibilité permet de réécrire la structure et le contenu du document, pour l'exportation vers un autre format de documents par exemple, mais ne permet pas de transformations complexes entraînant une modification de l'ordre des éléments ou leur déplacement dans les niveaux de structure. En contrepartie, les transformations sont plus faciles à spécifier et ne réclament pas l'expertise nécessaire au développement de systèmes de transformation plus complexes.
- Les systèmes dédiés à la transformation permettent de produire des filtres de conversion (CoST 2, STIL, Chameleon) ou de construction de documents (SgmlQL). Ces systèmes sont généralement basés sur le

modèle requête–action et proposent des fonctions de transformation plus avancées que le modèle dirigé par la source (réordonnancement des éléments, gestion des attributs, sélection d'éléments en fonction de leur contexte).

Ces transformations utilisent des algorithmes plus coûteux en temps de calcul. Les requêtes pouvant être appliquées à la globalité du document, la structure de celui–ci doit être parcourue à chaque évaluation. La place mémoire nécessaire à ces processus est également importante car elle doit contenir la structure dans son intégralité. C'est une des raisons pour lesquelles les transformations par requêtes ne sont pas utilisées dans des systèmes interactifs, mais trouvent leur application dans la conversion de grandes quantités de documents, mettant en œuvre des processus de conversion non interactifs.

L'expression des transformations par requêtes n'est pas aussi simple que celle des transformations dirigées par la source car les correspondances entre les éléments source et les éléments produits par la transformation ne sont pas directes. Par conséquent la programmation de ces transformations nécessite une connaissance plus approfondie des structures génériques des documents et de l'interprétation des requêtes par le système.

III.3.5 Spécification d'un système de transformation explicite pour l'édition de documents structurés

Cette partie expose une réflexion sur la possibilité d'expérimenter un système de transformations explicites dans le cadre applicatif de l'édition de document. Nous présentons ici les choix que nous avons fait préalablement à cette expérience.

Pour l'édition de documents structurés, les transformations explicites doivent remplir des conditions de performances et de localité adaptées à cette application. Le système doit pouvoir prendre en entrée un ensemble de transformations et se baser sur différents critères pour n'en retenir qu'une. Ces critères sont :

- la structure de l'élément source. Cet élément (ou ce groupe d'éléments) est déterminé par la sélection active au moment où la transformation est demandée ;
- le résultat attendu après la transformation qui est déterminé par la commande invoquée par l'utilisateur final et par le contexte dans lequel les

éléments sont transformés. Le type des éléments résultant de la transformation est choisi par l'utilisateur. Dans certains cas, plusieurs transformations peuvent conduire à un résultat du type demandé et l'utilisateur doit déterminer la transformation à appliquer. Par exemple, la transformation d'un élément liste en une table peut conduire à la création d'une table contenant un ligne ou une colonne comme l'illustre la figure 22.

La structure résultant d'une transformation peut ne pas être conforme à la structure générique définissant la classe de documents. Le système doit contrôler que la transformation ne produise que des structures conformes.

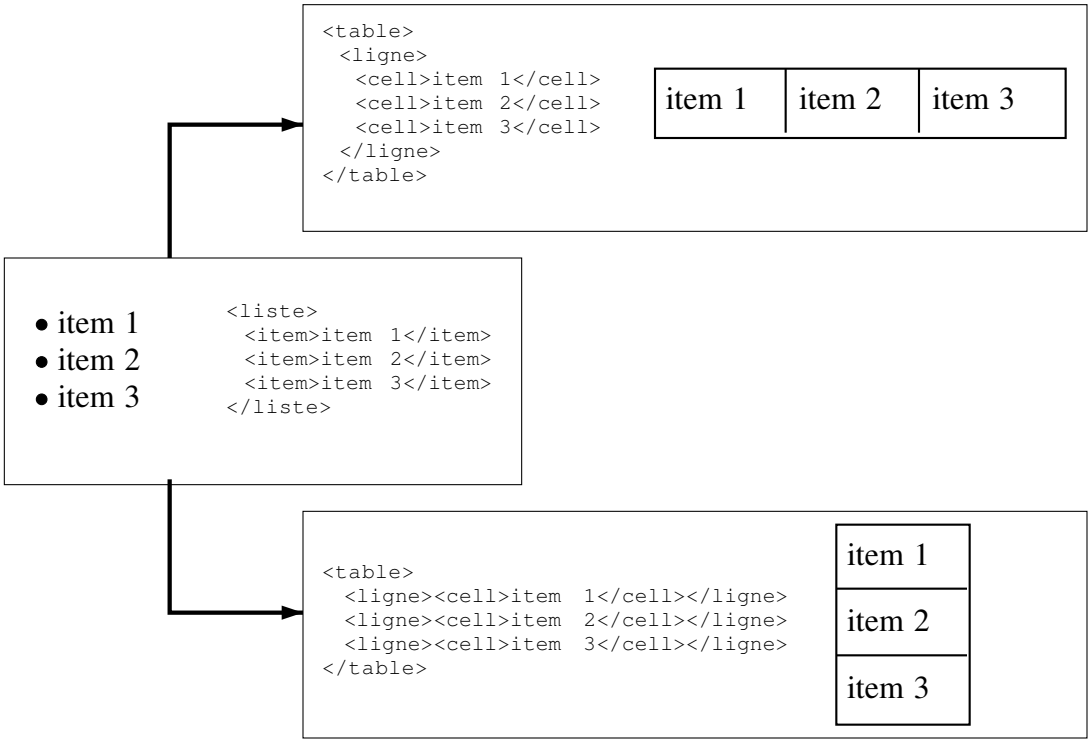


Figure 22 : deux transformations de liste en table

Les systèmes de transformation dirigés par la source permettent une application rapide des transformations en n'utilisant qu'une place mémoire minimale. De plus ils permettent une expression simple des transformations. Cependant ces systèmes, généralement associés à une génération linéaire du document cible, sont trop limités pour des transformations impliquant un déplacement des éléments dans la structure du document.

Les requêtes de sélection permettent une sélection des transformations en fonction de la structure de l'instance source. Ce type de requête permet au système de proposer à l'utilisateur un choix de transformations pré−sélectionnées.

L'utilisation de requêtes de sélection permet de spécifier la génération des éléments cibles par construction (SgmlQL, XSL). Cette spécification permet de mêler des éléments fixes et des éléments identifiés dans le document source par les requêtes.

Cette approche permet une expression plus lisible de la structure produite par la transformation mais est limitée lorsque cette structure est dépendante de la structure de l'instance source. Par exemple la spécification de la transformation d'une liste ne doit pas présupposer de connaître le nombre d'éléments contenus dans la liste. Dans ce cas il est nécessaire d'étendre la syntaxe des règles de génération avec des opérateurs de contrôle permettant de spécifier qu'une structure doit être crée autant de fois qu'il y a d'éléments dans la structure source. La déclaration de la production du document perd alors son aspect déclaratif et intègre une composante procédurale plus difficile à appréhender.

La complexité induite par cette approche est la raison pour laquelle nous avons cherché à combiner le mode de génération dirigé par la source et par construction des éléments cibles. La partie action des expressions de transformation que nous proposons est composé d'un ensemble de règles de génération décrivant chacune la construction d'une branche dans la structure des éléments engendrés. Ces règles sont relatives à des éléments identifiés dans la requête et déclenchées lors d'un parcours dans l'ordre préfixe des éléments reconnus par cette même requête.

Ainsi, nous avons expérimenté la technique de transformation explicite dans un environnement d'édition de documents HTML. Nous avons étendu le mécanisme et nous avons adapté l'interface pour prendre en compte l'édition de nouvelles structures telles que HTML 4.0 et MathML (Mathematical Markup Language). Ce système de transformation est présenté dans la section suivante.

III.4 Les transformations explicites dans Amaya

L'éditeur de documents HTML Amaya [Quint 94] est développé à l'INRIA par une équipe du consortium W3C. Nous avons implémenté dans cet éditeur une fonction de transformation d'éléments HTML et XML [Bonhomme 96]. Cette expérience nous a permis d'évaluer les techniques de transformation explicite,

de sélection de structures par requêtes et d'application des règles dirigées par la source. L'intégration de ce système dans un éditeur interactif lui confère quelques spécificités que n'ont pas les systèmes de transformation présentés précédemment.

Il est nécessaire que les transformations intervenant lors de l'édition du document soient performantes : le délai d'attente ne doit pas excéder la seconde. C'est pourquoi nous n'utilisons que des méthodes rapides de transformation (expressions régulières, application des règles dirigées par la source).

Amaya utilise une représentation structurée des documents édités. La structure des documents édités reste continuellement en conformité avec la DTD HTML (contrôle de structure strict). Par conséquent, les structures produites par le système de transformation doivent également produire des éléments conformes à la DTD.

Les transformations effectuées lors de l'édition ne portent généralement pas sur l'ensemble du document mais sur un nombre limité d'éléments choisis par l'utilisateur.

L'interface utilisateur doit être simple et rendre dans la mesure du possible les transformations transparentes. Pour cela, nous nous appuyons sur la sélection effectuée par l'utilisateur et les menus et boutons d'édition proposant à l'utilisateur les types d'éléments définis par la DTD. Les transformations sont spécifiées à l'avance, dans un fichier. L'utilisateur n'a pas besoin de connaître ce fichier et n'a pas à spécifier les transformation au moment où il veut les effectuer.

Nous avons voulu proposer un système de transformation explicite demandant une expression minimale des transformations. Plus précisément, nous voulons permettre de n'exprimer que les transformations des éléments de structure qui subissent des modifications, le système devant se charger de copier des éléments dont la structure est inchangée.

III.4.1 Interface utilisateur

Deux interfaces utilisateur sont proposées pour les transformations dans Amaya impliquant deux modes d'interaction avec le système de transformation. L'utilisateur peut demander explicitement la transformation de la sélection courante par l'entrée de menu *Transformer*. Il peut également déclencher une transformation en sélectionnant un passage du document et en sélectionnant un type d'élément dans une palette.

Dans le premier cas, l'ensemble des transformations pouvant s'appliquer à la sélection est proposé à l'utilisateur dans un menu construit dynamiquement, lui permettant ainsi de sélectionner la transformation voulue. Nous appellerons par la suite ce mode d'interaction *appel explicite.*

Dans le second cas, la première transformation (dans l'ordre de leur déclaration) produisant le type d'élément demandé et applicable à la sélection courante est déclenchée. L'utilisateur n'a pas forcément conscience qu'il provoque une transformation de structure et n'interagit qu'une fois avec le système. C'est, par exemple, ce qui se produit lorsqu'une table est sélectionnée et que l'on clique sur l'icône *Liste* de la barre de boutons d'Amaya. La table sélectionnée est alors transformée en en listes imbriquées. Ce mode d'interaction est nommé *appel induit*.

III.4.2 Spécification des transformations

Un fichier portant l'extension `.trans` contient la description de l'ensemble des transformations prédéfinies relatives à une DTD. Chaque transformation comprend trois composantes :

- Le nom de la transformation qui sera présenté à l'utilisateur de façon qu'il puisse faire le bon choix dans le cas d'un appel explicite.
- Une requête identifiant la structure des éléments source auxquels peut s'appliquer la transformation.
- Un ensemble de règles de génération relatives à des éléments identifiés dans la requête.

Une spécification de transformations a donc la syntaxe suivante :

```
Transformations := Transformation +
Transformation  := Nom ':' Requete '{' Règles '}'
Nom             := TEXTE
```

III.4.2.1 Requêtes

Les requêtes sont des *expressions* comprenant les opérateurs choix (|), séquence (,), liste (+), élément optionnel (?) et contenu ([ ]) utilisés pour combiner des *nœuds*. Chaque nœud permet d'identifier un ensemble d'éléments de la structure du document lorsque celle–ci est confrontée aux requêtes.

```
Requête        :=  Exp_requete
Exp_requete    :=  Noeud | '(' Exp_choix ')' |
                   '(' Exp_seq ')' | Exp_liste |
                    Exp_option
Exp_choix      :=  Exp_requete ( '|' Exp_requete ) +
Exp_seq        :=  Exp_requete ( ',' Exp_requete ) +
Exp_liste      :=  Exp_requete Op_liste
Op_liste       :=  '+'
Exp_option     :=  '?' Exp_requete
```

Ces expressions décrivent une organisation de nœuds qui est utilisée comme filtre sur la structure des éléments source. Les nœuds déclarés dans les requêtes peuvent être des nœuds simples ; les éléments qu'ils permettent d'identifier sont alors reconnus quelque soit leur contenu. Une expression peut aussi spécifier des nœuds avec un contenu, appelés nœuds composés ; les éléments qu'ils permettent d'identifier sont alors reconnus si et seulement si leur contenu est reconnu.

Un nœud simple est défini par un nom de type d'élément. L'ensemble des éléments identifiés par un nœud peut être restreint par la spécification d'une présence ou d'une valeur d'attribut. Les nœuds de la requête peuvent également être nommés pour faire le lien avec les règles de génération. Plusieurs nœuds peuvent porter le même nom pour leur appliquer la même règle de génération :

```
Noeud          :=  Noeud_simple | Noeud_composé
Noeud_composé  :=  Noeud_simple '[' Exp_requete ']'
Noeud_simple   :=  [Nom ':'] Type_elem |
                   [Nom ':'] '<' Type_elem ( ' ' Attr_spec )* '>'
Nom            :=  TEXTE
Type_elem      :=  TEXTE
Attr_spec      :=  Nom_attr | Nom_attr '=' Val_attr
Nom_attr       :=  TEXTE
Val_attr       :=  '"' TEXTE '"'
```

Le type de l'élément peut être soit un nom de type tel qu'il est défini dans la DTD, soit un jocker (*) qui représente n'importe quel type d'élément. Si une spécification d'attribut n'est pas associée à une valeur, seule la présence de cet attribut est requise pour que l'élément soit reconnu. La valeur associée à un attribut permet de filtrer les éléments qui ont un attribut du type spécifié qui a cette valeur.

Les quelques exemples suivants illustrent la syntaxe des requêtes utilisées par Amaya. La première requête identifie une liste non vide d'éléments de type *li*. La seconde identifie un élément *div* contenant un élément *h2* suivi d'une liste éventuellement vide d'éléments *p* ou autres.

```
li +
div [h2, ?(p | *)*]
```

L'exemple suivant permet de différentier les éléments de type *h2* selon qu'ils sont dans des divisions de premier niveau (`T1`) ou de second niveau (`T2`).

```
div [T1:h2, *+, div [T2:h2, *+]+ ]
```

La requête suivante identifie les éléments de type p portant un attribut *id* de valeur quelconque et un attribut *classe* ayant la valeur `commentaire` parmi une liste d'éléments quelconques.

```
( <p id classe="commentaire"> | * )+
```

Les requêtes du système de transformation d'Amaya servent d'une part à la sélection des extraits de documents sur lesquels les transformations peuvent être appliquées, d'autre part à l'identification d'éléments du document pour l'application de règles de transformation. Ces requêtes sont proches de celles proposées par XSL.Une première différence avec ce dernier est qu'Amaya permet d'associer des règles à plusieurs nœuds de la requête tandis que XSL ne permet que l'utilisation d'un seul élément identifié par la requête dans une règle de construction. Une seconde différence est que les requêtes d'Amaya permettent de spécifier un contexte plus étendu dans lequel il est possible de désigner les voisins des éléments ; dans les requêtes XSL seuls les ascendants des éléments source peuvent être précisés.

Amaya ne s'appuie pas sur le mécanisme de requêtes pour la génération mais sur un parcours dirigé par la structure avec une génération arborescente des éléments résultant de la transformation. C'est pourquoi la syntaxe des requêtes et la sémantique associée est différente des systèmes Scrimshaw et SgmlQL qui utilisent les requêtes comme mode de construction du document cible.

La syntaxe que nous avons présentée est inspirée de celle des expressions régulières et plus compacte que celle de XSL qui est exprimée en XML. Cette caractéristique permet une analyse plus rapide des fichiers de transformation. La vitesse de l'interprétation est un facteur important car il est possible de modifier le fichier contenant les expressions de transformation au cours d'une session d'édition et de profiter immédiatement de ces modifications. Le fichier est alors interprété à la volée lorsqu'une transformation est déclenchée.

III.4.2.2 Règles de génération

Les règles de génération des éléments cibles sont composées d'un identifiant et de la spécification d'une branche à générer dans l'arbre de structure du document. Ces deux composantes sont séparées par le caractère '>'.

```
Règles      :=  Règle+
Règle       :=  Identifiant '>' Branche ';'
Identifiant :=  TEXTE
```

L'identifiant fait la relation entre la règle de transformation et un ensemble de nœuds de la requête. Il peut être un nom défini dans la partie requête et la règle sera alors appliquée à l'ensemble des éléments reconnus par les nœuds de la requête portant ce nom. Il est ainsi possible d'associer la même règle à un ensemble d'éléments de types différents. Par exemple, la transformation d'une table HTML en liste s'écrit :

```
Liste : table [tbody [tr [(Cell:td|Cell:th)]+ ] ]
   {
   tr   > :ol.li;
   Cell > ol:li;
   }
```

Il est également possible, grâce au renommage des nœuds de la requête, d'associer des règles différentes à des nœuds différents, mais représentant le même type d'élément. Par exemple, la transformation suivante transforme une liste simple HTML en liste titrée ; le premier item de la liste simple (`Premier:li`) étant transformé en titre de liste.

```
Liste titrée : ul [Premier:li, (Autres:li)+]
   {
   Premier > dl:dt;
   Autres  > dl.dd;
   }
```

La spécification de la branche engendrée par une règle est relative à la structure engendrée par les règles précédemment appliquées. Lorsque la première règle est appliquée, la branche complète est crée. Le règles suivantes peuvent soit engendrer une nouvelle branche complète, soit engendrer une branche qui est attaché à l'un des éléments créés par la règle précédente, grâce aux éléments de placement définis ci–dessous. L'élément de plus haut niveau de la dernière branche entièrement crée est appelée élément *racine* de la structure produite.

Les éléments crées par l'application d'une règle sont toujours insérés **après** (comme frère suivant) les éléments crées par les règles précédemment appliquées. La descendance est écrite sous la forme d'une liste dont les éléments sont séparés par les caractères '.' et ':', le caractère ':' ne pouvant apparaître qu'une seule fois par règle. Ce caractère permet de séparer la branche en deux ensembles d'éléments :

- Les *éléments de placement* sont ceux spécifiés à la gauche du séparateur ':'. Ces éléments permettent de spécifier la position où seront insérés les éléments créés par la règle dans la structure de l'instance en construction.
- Les *éléments de construction* sont ceux spécifiés à la droite du séparateur ':'. Ces éléments sont systématiquement créés lors de l'application de la règle.

```
Branche     :=  [ Placement ] ':' [ Génération ] |
                [ Génération ]
Placement   :=  Élément ( '.' Élément )*
Génération  :=  Élément ( '.' Élement )* [ Fin_génération ] |
                '/'
Fin_génération := [ '.' '$' | '.' '"' TEXTE '"' ] ['#']
Élément     :=  Élément_simple | Élément_attr
Élément_simple :=  Nom
```

L'un et l'autre de ces ensembles étant optionnels, de nombreuses combinaisons sont possibles. Nous exposons tous les cas ci–dessous en les illustrant d'un schéma de la structure engendrée par la règle. Les arbres de gauche représentent la structure engendrée avant l'application de la règle, l'arbre de droite, cette même structure après l'application de la règle. Les nœuds de placement sont représentés en vert et les nœuds de génération sont présentés en rouge. Lorsqu'ils ont une influence, nous avons représenté, dans l'arbre de gauche, les nœuds de placement et de génération de la règle précédente.

elem > T1:T2; Cas général : des éléments de placement et de construction sont spécifiés. Les éléments de construction sont insérés comme dernier fils de l'élément de placement de plus bas niveau déjà existant.

Si tous les éléments de placement ne sont pas présents dans la branche la plus à droite de la structure créée préalablement à l'application de la règle, ceux–ci sont également créés.

elem > T1.T2:; les éléments de construction sont omis : le contenu de l'élément source (reconnu par `elem` dans la requête, et identifié par *data* dans le schéma) est copié à la position déterminée par les éléments de placement.

elem > :T1.T2; Les éléments de placement sont omis : les éléments de construction sont engendrés comme nouvelle racine de la transformation.

elem > T1.T2; Les éléments de placement sont omis : les éléments de construction sont crées à la même position que ceux de la dernière règle appliquée.

elem > :; Les éléments de placement et de construction sont omis : le contenu de l'élément source (*data*) est copié comme frère de la racine de la transformation.

elem >; Les éléments de placement et de construction sont omis : le contenu de l'élément source (*data*) est copié à la position déterminée par les éléments de placement de la dernière règle appliquée.

La distinction entre éléments de placement et éléments de construction permet de définir de façon précise où les éléments doivent être créés ou copiés dans la structure du document. Cette approche est intermédiaire entre la génération linéaire et la génération arborescente : l'ordre des éléments dans la structure est conservé, mais ils peuvent être insérés à des niveaux de structure spécifiques. Ce choix permet de proposer une syntaxe simple pour les expressions de transformation (une liste d'éléments) tout en répondant aux besoins de transformations au cours de l'édition. Ces transformations sont localisées et interviennent sur un petit nombre d'éléments, elles ne font pas de réordonnancement d'éléments qui peuvent être perturbants pour l'utilisateur.

Contrairement aux systèmes utilisant une génération linéaire des documents cibles, il n'est pas nécessaire de spécifier l'intégralité des structures produites par la transformation, Amaya s'appuie sur le modèle générique des documents et applique une politique visant à conserver l'intégralité du contenu des éléments transformés. En conséquence, les règles de transformation ne spécifient que les niveaux de structure du document où interviennent les changements. Par exemple, les règles de la transformation exposée ci–dessus et permettant de transformer les listes en listes titrées ne spécifient que la transformation des

éléments liste (*ul*) et item (*li*) en éléments liste titrée (*dl*), titre de liste (*dt*) et corps de liste titrée (*dd*). Le contenu des listes est implicitement transféré de la liste source dans la liste titrée, à la condition que la structure générique l'autorise.

Des symboles spéciaux peuvent être utilisés dans une règle de génération. Ces symboles permettent un autre comportement que le comportement par défaut de la génération de l'instance cible.

- Le symbole '$' remplace le dernier élément de la règle. Il permet de transférer le nœud reconnu par la requête au lieu de ne transférer que son contenu. Ce symbole est utilisé lorsque qu'un élément source a été reconnu par un joker.

  La transformation suivante permet par exemple de transformer une séquence d'élément en liste HTML :

  ```
  Liste : (item:*)+
  {
  item > ul:li.$;
  }
  ```

- Le symbole '/ ' permet d'ignorer le nœud source, ce symbole remplace la totalité de la partie génération de la règle. Le contenu de l'élément reconnu par la requête n'est alors pas transféré dans la structure cible.

  La transformation suivante permet de transformer une table en liste, en ignorant le titre de la table (*caption*) :

  ```
  Liste : table [caption, tbody[tr[td+]+]]
  {
  caption > /;
  td      > ul:li;
  }
  ```

- Le symbole '#' est utilisé lorsque plusieurs règles sont spécifiées pour un même nœud de la requête. Les règles sont alors appliquées dans l'ordre de leur déclaration. Le contenu de l'élément source reconnu est transféré lors de l'application de la règle identifiée par le symbole '#'. Dans la transformation suivante, une image est insérée avant chaque élément lien HTML (a), et le contenu de l'ancre est présenté en gras :

  ```
  A2Img : <a href>
  {
   a > <a href=a.href>.<img src="link.gif">;
   a > a:strong #;
  }
  ```

  Ce symbole peut être utilisé dans plusieurs règles pour dupliquer la descendance des éléments source lors de la transformation.

  Le symbole '$' peut également être utilisé lorsque plusieurs règles relatives à un même nœud sont déclarées, son rôle est alors également

d'identifier la règle sur laquelle doit être transféré l'élément source et son contenu. Par exemple, la transformation suivante permet de transformer une séquence d'éléments quelconques en liste HTML et de précéder chacun d'entre eux d'une titre :

```
Promo : (Item:*)+
{
Item > ul:li.h2."Promotion : " ;
Item > ul.li:$ ;
}
```

Le système de transformation d'Amaya permet également de générer des attributs sur les éléments spécifiés dans les règles de génération. Ces attributs peuvent prendre des valeurs fixes ou copier des valeurs d'attributs présents dans la structure source.

```
Élément_attr    :=  '<' Nom ( ' ' Attribute)* '>'
Attribute       :=  Nom '=' Value
Value           :=  Nom '.' Nom | '"' TEXTE '"'
```

La transformation suivante transforme une suite de paragraphes, dont certains portent des attributs *style* et d'autres contiennent des éléments *b*, en une suite de paragraphes ne portant que des attributs *style* et des éléments *span* portant un attribut *style*.

```
BtoCSS : ( p [(*|b)+])+;
   {
      p > <p style=p.style>:;
      b > p:<span style="font-face:bold;">;
   }
```

La règle `p` spécifie que chaque paragraphe de la structure source est transformé en un paragraphe portant un attribut *style* qui prend la valeur de l'attribut *style* du paragraphe original. La règle `b` transforme chaque élément *b* en un élément *span* et crée un attribut *style* attaché à cet élément. Avec la manipulation des attributs, les transformations proposées par Amaya permettent de créer une présentation spécialisée des éléments en utilisant le mécanisme des CSS2 (Cascading Style Sheets, level 2) [Bos 98] proposé par le W3C pour la présentation des documents HTML.

Il est possible de générer des feuilles de texte au cours de la transformation : le dernier nœud de la partie génération d'une règle peut être une chaîne textuelle (première règle de la transformation Promo). Si une règle qui provque la création d'éléments textuels est la seule attachée à un élément de la requête, sa descendance est alors remplacée par la feuille de texte spécifiée.

III.4.3 Déroulement d'une transformation

Nous montrons, à l'aide d'un exemple, le déroulement d'une transformation explicite dans Amaya. Nous décrivons d'abord comment sont représentées les requêtes, puis comment cette représentation est utilisée pour leur interprétation, ensuite nous expliquons comment se déroule la génération de la structure cible.

Prenons comme exemple la transformation dans un document HTML d'une suite de titres de deux niveaux (`h1` et `h2`) entrecoupés d'éléments quelconques (*) en une structure de liste titrées imbriquées s'écrit :

```
Listes  Titrées:(h1,(C1:*)+,(h2,(C2:*)+)*)+;
  {
    h1 > dl:dt;
    C1 > dl.dd:*;
    h2 > dl.dd.dl:dt ;
    C2 > dl.dd.dl.dd:*;
  }
```

L'ensemble des requêtes de toutes les expressions de transformation déclarées est représenté sous la forme d'un automate qui est éclaté en fonction de la profondeur des requêtes. Ainsi, l'interprétation de l'expression de transformation donnée en exemple produit l'automate de reconnaissance de structure présenté figures 23. La génération de cet automate ne se fait que lors de la première transformation d'une session d'édition. L'automate est alors gardé en mémoire et utilisé lors des transformations suivantes.

L'automate construit est éclaté de façon à obtenir un automate pour chaque niveau de structure de l'ensemble dans les requêtes interprétées. Chacun de ces automates de niveau permettant de déterminer quelles parties de requêtes reconnaissent une séquence d'éléments voisins, appelée *sous−requête* (partie `Exp_requête` de la règle `Noeud_composé` de la grammaire de la section III.4.2.1).

Les transitions d'un automate de reconnaissance sont étiquetées par le nom du type correspondant à la transition et un ensemble de couples d'identificateurs : le premier identifie de façon unique la sous−requête à laquelle appartient le nœud représenté par la transition, il est appelé *ide*. Si la descendance de ce nœud est spécifiée par la requête, le second identificateur, appelé *idinf*, identifie la sous−requête définissant cette descendance dans l'automate du niveau inférieur, il est nul sinon.

L'automate présenté dans la figure 23, ne représente qu'une sous−requête et aucune transition ne fait référence à une sous−requête car aucune descendance de nœud n'est définie dans la requête. Par conséquent le couple d'identificateurs associé à chaque transition est : (ide = 1, idinf = 0).

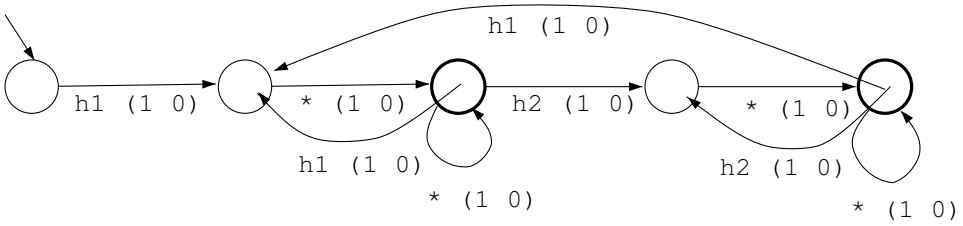
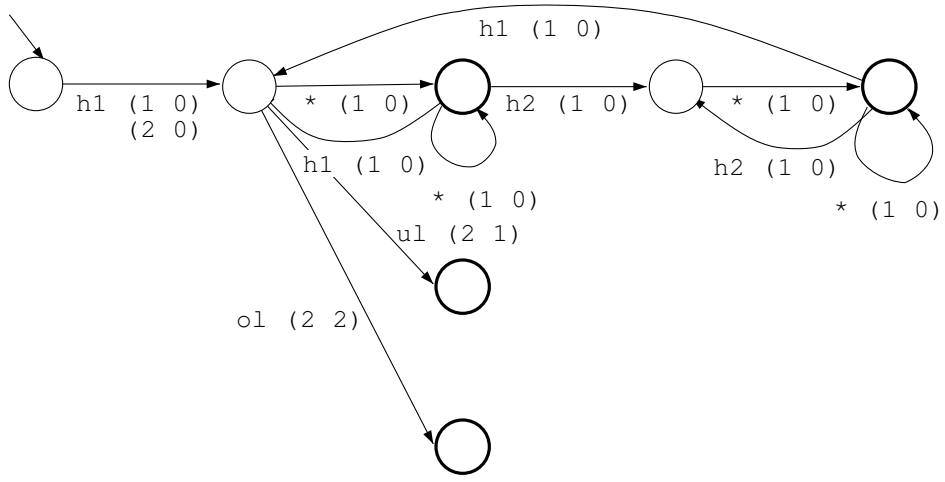
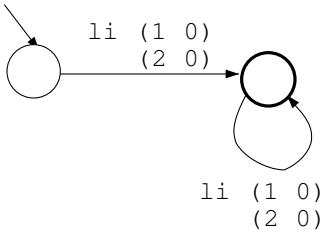


Figure 23 : Automate de reconnaissance de la requête
`(h1,(C1:*)+,(h2,(C2:*)+)*)+`

La figure 24 montre les automates des deux niveaux de structure construits lors de l'interprétation d'une seconde requête : `h1, (ol[li+]|ul[li+])` qui reconnaît un titre suivi d'une liste numérotée (`ol`) ou non (`ul`). L'interprétation de cette requête étend l'automate de niveau 0 de la figure 23 et produit un automate de niveau 1 pour reconnaître les sous–requêtes (`li+`). Les transitions associées aux nœuds parents des sous–requêtes référencent les transitions correspondantes dans la sous–requête par le deuxième paramètre de leur couple d'identificateurs (`ul (2 1)` et `ol (2 2)`). Notons que deux couples d'identificateurs sont associés à la transition représentant le nœud h1 dans l'automate de niveau 0 car cette transition est utilisée pour la reconnaissance des deux requêtes.



Automate de reconnaissance – niveau 0



Automate de reconnaissance – niveau 1

Figure 24 : Automates de reconnaissance des deux requêtes

L'algorithme de reconnaissance parcourt l'arbre de structure de l'instance source depuis le niveau le plus bas jusqu'aux éléments constituant la sélection. Chaque ensemble d'éléments frères (qui ont le même parent) est analysé par l'automate du niveau correspondant. Si cet ensemble est reconnu en utilisant des transitions portant toutes un ou plusieurs identificateurs *ide*, ces identificateurs sont remontés au niveau supérieur et seules les transitions ayant un identificateur *idinf* appartenant à cet ensemble permettent la reconnaissance du niveau supérieur.

Un ensemble d'éléments frères est reconnu par un automate si et seulement si il existe une suite de transitions qui a les propriétés suivantes :

1. La suite des noms de types associés aux transitions est égale à la suite des noms de types des éléments successifs.
2. La suite des transitions mène à un état final de l'automate (représenté en gras sur les figures 23 et 24).

3. Il existe au moins un *ide* tel que chaque transition soit définie par un couple (*ide*, *idinf*) où *idinf* est égal à 0 ou bien appartient à l'ensemble des *ide* reconnus par l'automate de niveau inférieur appliqué aux descendants de l'élément reconnu par le nœud associé à la transition.

Le résultat de la reconnaissance sur un niveau est l'ensemble des *ide* répondant à la troisième propriété.

Une table de nommage des éléments est construite durant la reconnaissance pour associer aux éléments de la source les règles relatives aux nœuds de la requête. L'algorithme ne détaille pas la reconnaissance d'attributs qui est une extension de l'égalité entre le type de l'élément et le type exprimé dans la requête.

Le résultat de l'évaluation de requêtes est ensemble de transformation dont les requêtes ont été reconnues par l'instance source. Dans le cas d'un appel explicite, cet ensemble est présenté à l'utilisateur pour lui permettre de sélectionner une transformation de son choix, Dans le cas d'un appel induit, la transformation correspondant à la première requête de l'ensemble est appliquée.

L'application des règles de transformation se fait lors d'un parcours de l'arbre de structure de la source. L'instance source est préalablement déconnectée du corps du document pour pouvoir générer les éléments définis dans les règles de transformation directement dans la structure du document et pouvoir ainsi vérifier qu'ils sont conformes à la structure générique du document.

La figure 25 montre la construction de l'instance cible par l'application des règles `h1`, `C1`, `h2` et `C2`, et illustre le rôle des nœuds de placement et des nœuds de construction.

- La règle `h1` (présentée en mauve) crée l'élément de placement *dl* et l'élément de construction *dt*.
- La première application de la règle `C1` (rouge foncé) ne crée pas l'élément de placement *dl* car c'est déjà le dernier élément du niveau le plus haut de la structure du document cible, par contre l'élément de placement *dd* est créé car il ne correspond pas au type du dernier élément du second niveau de structure (*dt*). L'étoile en fin de règle signifie que l'élément *p* est copié dans la structure cible.
- La seconde application de la règle `C1` (rouge clair) ne génère pas de nœuds de placement car ceux−ci sont déjà présents dans la branche la plus à droite de la structure crée. Seul l'élément *ol* est copié.

- La première application de la règle `h2` (bleu foncé) crée l'élément de placement *dl* et l'élément de construction *dt*, tandis que la seconde ne crée que l'élément *dt* (l'élément *dl* étant déjà créé par les règles précédentes).
- L'application des règles `C2` copie les éléments source (*ul* et *table*) dans la structure cible.

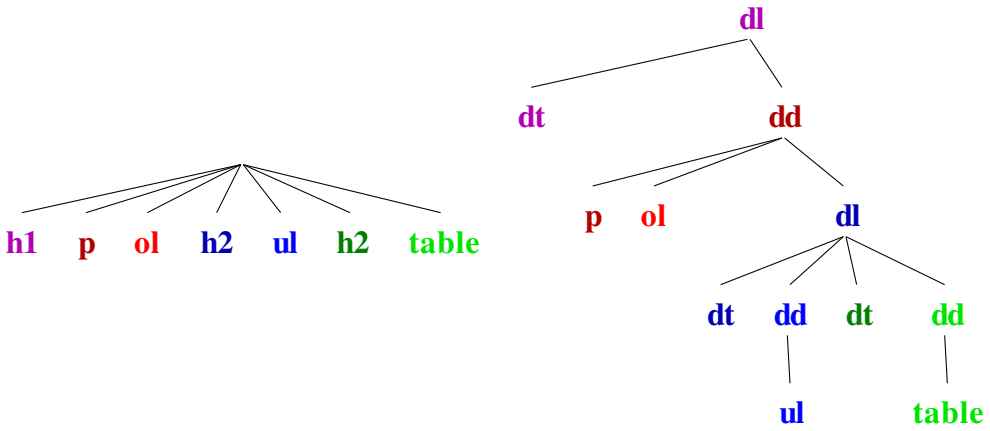


Figure 25 : Application des règles de transformation en liste titrée

Nous n'avons pas représenté sur la figure la descendance des éléments source. Cette descendance est simplement transférée depuis l'instance source dans les éléments correspondant dans l'instance destination. Par exemple, la descendance de *h1* est transférée dans l'élément *dt*. Les éléments *p*, *ol*, *ul* et *table* ainsi que leur descendance sont transférés dans l'instance destination car les règles qui leur sont associées se terminent par le symbole '*'.

La transformation est effectuée par deux parcours de l'arbre de structure des éléments source. Le premier évalue les requêtes des différentes expressions de transformation, le second permet la génération des éléments de la structure cible.

Ces deux parcours sont rendus nécessaires pour laisser à l'utilisateur la possibilité de sélectionner une transformation après que la reconnaissance des requêtes a été effectuée et qu'un sous–ensemble des transformations a été pré–sélectionné. Cette interaction doit évidemment se faire préalablement à la phase de génération de l'instance cible. Dans le cas d'un appel induit, les deux parcours de la structure sont également effectués car l'algorithme de reconnaissance des requêtes effectue un parcours ascendant de l'arbre de structure

de la source, tandis que la génération fait un parcours dans l'ordre préfixe, donc en descendant dans l'arbre de structure.

III.4.4 Application du système de transformations explicites d'Amaya

Le système de transformation d'Amaya a été conçu pour assister l'utilisateur dans l'élaboration de documents HTML. Alors que les autres éditeurs HTML ne proposent pas de commandes d'édition permettant la manipulation de structure, les transformations permettent à l'utilisateur de facilement structurer des documents faiblement structurés. Prenons l'exemple d'une liste que l'on voudrait changer en table : dans un éditeur quelconque il aurait fallu créer une table vide (éventuellement en spécifiant ses dimensions), et transférer un à un les contenus de chaque item de la liste dans les cellules de la table nouvellement créée. Avec Amaya, une seule transformation permet de changer une liste en table simplement en sélectionnant la liste à transformer et en cliquant sur l'icône table.

Les transformations dans Amaya se sont également avérées très utiles pour l'édition de formules mathématiques selon la DTD MathML [Ion 98]. Cette DTD XML est développée par le consortium W3C pour intégrer des formules mathématiques dans des pages Web. Les formules mathématiques MathML peuvent être très structurées : la figure 26 montre une formule mathématique et sa représentation MathML. Chaque opérateur est représenté par un (ou plusieurs) élément de structure, reflétant l'arbre syntaxique de la formule.

$$\sqrt{\frac{x^2 + y^2}{2}}$$

```
<msqrt> <mfrac>
  <mrow>
   <msup>
    <mi>x</mi>
    <mn>2</mn>
   </msup>
   <mo>+</mo>
   <msup>
    <mi>y</mi>
    <mn>2</mn>
   </msup>
  </mrow>
  <mn>2</mn>
</mfrac> </msqrt>
```

Figure 26 : Une formule mathématique et sa représentation MathML

La nature structurée des équations mathématiques est utile pour les applications interprétant la structure de ces équations, mais elle ne se prête pas facilement à l'édition. Par exemple, il est naturel pour un utilisateur de saisir le numérateur avant d'indiquer qu'il en train de saisir une fraction, ce comportement implique la création du contenu d'un élément avant de créer l'élément

lui–même (ici la fraction). Il est nécessaire d'opérer une transformation de structure pour englober le numérateur dans l'élément fraction.

Amaya permet, grâce aux transformations, de sélectionner une expression mathématique et de choisir ensuite l'opérateur dans laquelle cette expression doit être insérée. Ce mode d'édition, combiné avec une palette contenant les opérateurs mathématiques permet une édition interactive des formules mathématiques reproduisant le schéma naturel de la rédaction.

Nous avons expérimenté le mécanisme de transformation d'Amaya dans l'éditeur Thot qui permet de manipuler des objets de diverses natures. L'objectif de cette expérience était d'évaluer l'apport de cette technique dans un environnement d'édition générique pour répondre à la problématique de cette thèse.

Cette expérience nous a montré que les transformations explicites ne sont pas adaptées aux environnements d'édition de documents structurés génériques. En effet, la transformation des éléments nécessite la spécification des transformations qui sont applicables à la DTD qui définit ces éléments. Les documents édités par Thot appartiennent à un ensemble de DTD susceptible d'augmenter au cours du temps. La réalisation d'un système de transformation complet pour un tel environnement passe par la spécification de transformation pour chacune des DTD utilisées, mais aussi en ces DTD. Sauf dans certains cas d'application très spécifiques, dans lesquels les transformations par filtre décrites au début de ce chapitre peuvent suffire, la définition des transformations devient alors une tâche trop importante comparativement à l'utilisation qui en est faite.

III.5 Synthèse

Après avoir présenté les méthodes de transformation explicites et les outils exploitant ces différentes méthodes ainsi que notre expérimentation de la transformation explicite, nous tirons des conclusions sur l'utilisation de cette approche dans notre problématique.

Les systèmes de transformation dirigés par la source sont performants (de l'ordre du nombre de nœuds de l'instance source). Cependant ce jugement doit être nuancé lorsque le contexte des éléments cibles doit être pris en compte. Le coût supplémentaire est minime lorsque seulement les frères précédents et ancêtres de l'élément (contexte gauche immédiat) doivent être considérés, cette information pouvant être conservée lors de l'application des règles précédentes. Le surcoût peut s'avérer important dès qu'un ensemble d'éléments constitue le

contexte, principalement les éléments suivant l'élément en cours de transformation, ou la descendance de cet élément (contexte droit).

Les règles de transformation dépendantes du contexte posent des problèmes lors de l'implémentation du mécanisme de transformation :

- Place mémoire : le contexte doit être stocké en mémoire pour être utilisé lors de la transformation. Pour de gros documents, le nombre d'éléments à transformer peut être important (de l'ordre d'une dizaine de milliers d'éléments pour un document d'une centaine de pages). Pour pallier à cet inconvénient, certains systèmes tels que Balise limitent l'étendue du contexte gauche et ne mémorisent que le frère précédent et les ancêtres de l'élément traité.

  Ce problème se pose pour les applications qui traitent le document comme un flot de données. C'est le cas des applications bâties sur un parser. Un outil qui stocke tout l'arbre du document en mémoire pour d'autres fonctions que la transformation peut accéder à tous les éléments de la structure sans nécessiter de mémoire supplémentaire.
- Performance : la sélection des règles de transformation en fonction du contexte de l'élément source implique la confrontation de la spécification avec le contexte courant. Les algorithmes de comparaison d'arbres sont coûteux.

Un autre point important en la défaveur des transformations dirigées par la source est qu'elles ne permettent pas de réaliser toutes les catégories de transformations. En particulier, elles ne permettent pas de mettre en œuvre des transformations impliquant un ordonnancement différent des éléments dans la structure.

La transformation par requêtes nécessite une représentation de la structure d'un document en mémoire pour pouvoir appliquer des algorithmes de pattern−matching sur ces structures ([Kilpeläinen 92], [Cai 90], [Hoffmann 82]).

L'utilisation des requêtes pour la sélection d'une transformation ou d'un sous−ensemble des règles de transformations telles qu'elles sont utilisées dans XSL n'est pas pénalisante du fait de la localité des requêtes. Par contre, les requêtes permettant l'extraction d'information doivent traiter la structure entière du document à chaque évaluation, au détriment des performances du système d'évaluation de ces requêtes.

En contrepartie, des transformations complexes peuvent être réalisées en utilisant l'approche par requêtes, comme celles impliquant des déplacements d'éléments dans la structure des documents ou impliquant un changement dans

l'ordre des éléments. Ces transformations ne peuvent pas être réalisées avec l'approche dirigée par la source et la génération linéaire.

Pour l'édition interactive, l'intérêt de l'approche par requête est la possibilité de sélection des règles à appliquer en fonction de la structure du document source. Ainsi, comme il a été montré dans le système de transformation d'Amaya, une même expression de transformation peut s'appliquer à un ensemble d'éléments n'ayant pas exactement la même configuration structurelle.

Le tableau de la figure 27 synthétise les caractéristiques des systèmes de transformation explicites étudiés :

- Le langage dans lequel doivent être exprimées les transformations peut être un langage de programmation ou un langage spécifique (en italique). Les langages spécifiques ont l'avantage d'être adaptés à l'expression des transformations, mais nécessitent un apprentissage particulier.
- Les méthodes de transformation proposées telles qu'elles ont été exposées dans la section III.3.1.
- La méthode de génération du document cible : dans le cas de la transformation arborescente, le tableau précise si la transformation permet la génération d'un nouveau document ou change la structure du document source (transformation sur place).
- Le domaine d'application du système de transformation (voir section III.3.4).

	Langage	Méthode de transformation		Génération		Applications
		Dirigée par la source	Requête action	Linéaire	Arborescente	
IDM	C++ Script	oui (scripts)	non	oui	non	filtres de conversion
Balise	*Balise*	oui	non	non	sur place / nouveau	
CoST 2	Tcl	oui	non	oui	non	
DSSSL	Scheme	non	oui	non	nouveau	formatage
XSL	XML ECMA	non	oui	non	nouveau	formatage
Chameleon	Graphique	oui	non			
Scrimshaw	*Scrimshaw*	non	non	oui	non	
Amaya	*trans*	oui (ordre)	oui	non	sur place	édition interactive

Figure 27 : Tableau comparatif des systèmes de transformation explicite.

La plupart des systèmes de transformation explicites nécessitent une expression extensive de la transformation des documents. C'est à dire que la spécification nécessite une description de la transformation de chacun des éléments du document source..

La déclaration des transformations entre documents structurés nécessite non seulement une connaissance de la syntaxe du langage de transformation, mais également la connaissance des DTD source et cible. Cependant, la plupart des outils de transformation ne vérifient pas la conformité du document produit par rapport à sa DTD. Parmi les systèmes présentés, seul Amaya, qui est un environnement d'édition basé sur la syntaxe, Balise, qui est une bibliothèque de développement d'applications utilisant les documents structurés, et Chameleon, qui permet de spécifier des transformations à partir d'une représentation sous forme de grammaire des DTD, vérifient la conformité du document cible avec son modèle.

Notre objectif est de proposer un service de transformation de documents structurés prenant en compte la validité du document cible par rapport à sa DTD.

Cette caractéristique est à notre sens essentielle pour un système de transformation. En effet, l'édition de documents structurés utilise des mécanismes qui se fondent sur la structure et qui supposent que les documents sont conformes à leur structure générique.

Lors de la création d'éléments comme lors de la transformation d'éléments, la conformité d'un document dépend non seulement de son type, de celui de son père et de ses frères, mais en plus, les mécanismes d'inclusion, d'exclusion (au sens SGML) et d'héritage impliquent que la validité d'une instance dépende du contexte du document entier. Ainsi, il n'est pas possible de décider de la validité d'une transformation à partir de sa seule spécification. Par conséquent, la validité des éléments insérés ne peut être vérifiée par le système que lorsque la transformation est effectuée.

Nous abordons dans le chapitre suivant une approche plus formelle du modèle de documents structurés. Nous définissons clairement ce que sont les types d'éléments que nous manipulons et les relations entre ces types qui seront utilisées pour la transformation. Ainsi nous pourrons proposer un système de transformation de basant sur les structures génériques des types d'éléments nous garantissant par construction la validité de la structure des éléments crées.

Notre objectif n'est pas seulement de proposer des transformations garantissant le maintien de la validité du document, mais aussi de s'appuyer sur les relations entre les types des éléments concernés pour proposer un mécanisme de transformation automatique.

Chapitre IV

Système de types des documents structurés

Ce chapitre propose une modélisation formelle des types d'éléments des documents structurés. Il propose également un ensemble de relations entre ces types sur lesquelles nous fondons les principes de la transformation présentée dans la section suivante. Nous abordons ici une étude du système de types des documents qui fait suite à une réflexion plus générale sur les systèmes de types dans les langages de programmation. Dans cette réflexion, donnée en annexe A, nous décrivons l'algorithme d'unification permettant de calculer l'équivalence de deux types.

Il existe une analogie entre les systèmes de types des langages de programmation, qui définissent des *types de données*, et ceux des documents structurés qui définissent des *types d'éléments de documents.* Cette analogie laisse à penser que les algorithmes de vérification de types, d'inférence et d'unification (voir annexe A) peuvent s'appliquer à la transformation de documents structurés. Dans ce chapitre, nous présentons les différences entre les deux notions de type au point de vue de leur conception mais aussi dans l'utilisation qui en est faite.

IV.1 Caractéristiques des systèmes de types des documents structurés

Les différences entre les structures génériques de documents structurés et les systèmes de types des langages de programmation sont en partie dues à la

différence de nature de l'information traitée : les documents structurés détiennent une dimension sémantique qui n'est pas accessible aux applications de traitement, alors que l'ensemble de l'information d'un programme – à l'exception des commentaires – est interprétée par le compilateur. Les documents sont également traités de façon plus incrémentale que les programmes : un programme doit être complet avant d'être compilé ou exécuté, tandis que les applications documentaires, en particulier les éditeurs, travaillent sur les documents en cours de leur construction. Par conséquent, ces applications doivent pouvoir représenter et traiter des documents qui ne sont pas complets.

Ces différences de nature entre documents et programmes se répercutent sur les systèmes de types.

IV.1.1 Conformité des documents

Lors de son élaboration, un document, comme un programme, passe par des phases pendant lesquelles sa structure est incomplète. Les programmes incomplets ne peuvent être compilés ni exécutés car le système à besoin de la totalité de la sémantique pour l'exécution. Si certaines parties du programme ne sont pas implémentées, les interface des fonctions externes – utilisées par le reste du programme – doivent cependant exister pour que le programme soit conforme au système de types du langage.

Il existe plusieurs niveaux de conformité du document par rapport à la structure générique d'une classe. Suivant les applications, les besoins de conformité peuvent être plus ou moins stricts. L'algorithme présenté dans l'annexe A détermine une conformité au sens strict du document par rapport à la structure générique qui définit la classe à laquelle il appartient. Un document présentant cette propriété est un document *valide*. Cet algorithme est utilisé par les programmes de vérification de la validité des documents [W3C a] et les analyseurs XML validants [W3C b].

Certaines applications documentaires doivent pouvoir traiter les documents non complets et donc non valides. C'est le cas des éditeurs qui doivent permettre de stocker, d'ouvrir, de modifier et d'échanger des documents en cours de réalisation. En revanche, d'autres applications, par exemples utilisées pour la publication ou l'archivage, nécessitent qu'un document soit valide pour pouvoir le traiter.

Ces deux niveaux de conformité de documents se retrouvent dans la norme XML qui différencie les documents valides des documents bien formés.

IV.1.2 Attributs

Une composante importante des documents structurés que l'on ne retrouve pas dans les systèmes de types concerne les attributs. Ceux–ci servent à expliciter une partie de la sémantique inhérente au contenu du document. Ils sont interprétés par les différentes applications pour procéder à des traitements particuliers. Ces traitements sont, par exemple, l'indexation automatique et la mise en valeur des mots portant un attribut *mot important* ou le choix du dictionnaire lors de la correction orthographique d'un document multilingue qui est déterminé par un attribut *langue*.

Les normes de documents structurés XML et SGML définissent des types d'attributs : ceux–ci peuvent être énumérés, textuels ou désigner des entités externes comme d'autres documents ou des ressources d'une autre nature (images, graphiques, etc.). Dans une DTD, un ensemble d'attributs est défini relativement à un type d'élément, les attributs peuvent être obligatoires ou optionnels. Le modèle de types doit donc être étendu pour que chaque nœud du graphe de types puisse supporter des attributs.

Thot permet de définir quatre types d'attributs : les attributs entiers, énumérés, textuels et référence. Les attributs référence désignent des éléments de structure dont le type peut être fixé ou non. Comme dans SGML et XML, les attributs sont relatifs à un type d'élément, mais ils peuvent également être définis de façon globale et être alors associés à tout type d'élément.

IV.1.3 Modularité

Les systèmes de documents structurés proposent des mécanismes permettant une modularité des documents de façon que des structures génériques puissent être partagées par plusieurs classes de documents.

SGML et XML permettent l'inclusion d'éléments externes par l'intermédiaire des entités. Une extension de XML appelée *namespaces* [Bray 98b] définit un mécanisme pour intégrer des éléments de plusieurs DTD dans un même document. Ce mécanisme permet à plusieurs applications d'ajouter des éléments spécifiques dans un document en utilisant son propre espace de noms, c'est le cas XSL, présenté dans la section III.3.3.7 qui définit un espace de nom *xsl* pour représenter les directives de transformation.

D'autres applications utilisent les espaces de noms pour définir des objets de natures différentes, tout en utilisant le même modèle de documents structurés. Ainsi, Amaya permet de mélanger la structure HTML avec des équations mathématiques définies par la DTD MathML. Thot utilise également les

namespaces pour représenter des natures qui permettent l'inclusion d'éléments de structures génériques différentes dans un document. Ainsi les classes de documents *rapport*, *chapitre*, *exposé* et *lettre* partagent la même définition des *paragraphes*, des *tables* et des *graphiques*.

La modularité des structures génériques implique que plusieurs systèmes de types coexistent au sein d'un même document. Ces systèmes de types utilisent les mêmes constructeurs (agrégat, liste, choix, identité) mais définissent des espaces de noms de types différents. Deux types définis dans des structures génériques différentes et portant le même nom peuvent coexister dans le même document.

IV.1.4 Éléments optionnels, inclusions, exclusions

Toutes les normes de documents structurés permettent de définir des éléments optionnels dans la DTD. Ces éléments peuvent être présents dans la structure du document, mais ne sont pas requis pour la conformité du document. Cela implique que certaines parties du graphe de types ne doivent pas être utilisées systématiquement par les algorithmes de comparaison de types.

Thot et SGML proposent des mécanismes d'inclusion et d'exclusion, permettant respectivement d'autoriser ou d'interdire certains types d'éléments dans la construction d'un autre type d'élément. Par exemple, une archive de messages tels que ceux donnés dans la figure 14, qui contient des messages anonymes ou non et leur attribue un numéro d'archivage peut être définie par la DTD SGML suivante qui autorise la présence d'un élément *idarchive* dans les éléments *anonyme* et *message* et exclut l'élément *expéditeur* des éléments *messages* contenus dans les éléments *anonyme* :

```
<!ELEMENT archive    - - (anonyme|message)+ +idarchive >
<!ELEMENT anonyme    - - (message)          -expéditeur>
```

IV.2 Définitions

Dans la section précédente nous avons présenté les différences d'interprétation par les applications de la sémantique liée aux systèmes de types selon que ces applications sont du domaine de la gestion de documents ou de la programmation. Comme les langages de programmation, les documents structurés utilisent des systèmes de types particuliers. Au delà de leur utilisation par les applications, les systèmes de types proposés par les normes comme SGML et XML ne sont pas adaptés à la comparaison de types «classique».

- Contrairement aux langages de programmation, dans lesquels les données de base sont typées (entier, caractère, booléen, etc.), il n'existe qu'un unique type de base représenté dans les documents XML : le type texte (PCDATA). Les objets de types différents, quand ils existent, sont externes et sont référencés par le document.
- La norme XML permet de définir des types d'éléments comme étant vides (empty). Ces types d'éléments ne sont pas construits à partir de types de base du langage.
- Dans les langages de programmation, les types sont définis par un constructeur unique et un ensemble de types plus élémentaires. Dans une DTD XML, un type d'élément peut être une composition plus complexe utilisant plusieurs constructeurs. Par exemple, un paragraphe est défini comme une liste de choix entre éléments de type *texte*, *lien* ou *mise en valeur*.

Le premier argument implique que la part de sémantique contenue par le type d'un objet de base dans un langage de programmation n'est pas représentée dans le système de types des documents structurés. Cette sémantique est remontée au niveau du contenu textuel de l'élément et réclame des outils d'analyse plus puissants que ceux proposés par les algorithmes de l'annexe A. Les deux autres arguments révèlent une incapacité plus fonctionnelle de ces algorithmes, ne leur permettant pas de s'appliquer à un arbre de types déduit directement de la DTD.

Nous définissons donc dans la suite de ce chapitre une représentation adaptée des systèmes de types des documents permettant de prendre en compte ces facteurs. Ce système de types précise la définition des documents structurés énoncée en II.3.1.

Dans la deuxième partie de ce chapitre nous nous servirons de ce système de types pour définir des relations entre les structures des types source et cible pour permettre la transformation.

IV.2.1 Types de base

Dans la norme XML, le système de types est décrit dans les règles de définition d'une DTD (règle `elementDecl` de la spécification [Bray 98a]). Pour faciliter le stockage et l'échange des documents, la norme XML ne décrit qu'un seul type de base : le type texte (CDATA), les autres composantes atomiques du document (images, vidéo, graphiques) sont des références à des objets externes au document. Ces références sont définies par des attributs textuels.

Pour conserver la nature des objets de base des documents lors des transformations de structure, nous avons choisi d'intégrer ces références dans le système de types des documents structurés. Pour une meilleure lisibilité, nous donnons comme nom aux types de base de notre système, les types des éléments désignés par les références. Ainsi un élément faisant référence à une image (par exemple, l'élément vide `<IMG href="http://www.inria.fr/logo.gif">` en HTML), sera pour nous un élément de type *Image*. Nous définissons ainsi un ensemble de type de base commun à l'ensemble des DTD : **B** = {*texte*, *symbole*, *graphique*, *image*, *référence*}.

Pour intégrer les éléments vides qui ne référencent pas d'objets externes nous étendons l'ensemble **B** avec les représentants de ces types pour former l'ensemble des types de base définis par une DTD S, noté $\mathbf{B}_S$.

IV.2.2 Constructeurs

Les constructeurs que nous définissons dans notre système sont :

- L'*identité* définit un type comme étant identique au type qui le définit.

  Un élément dont le constructeur de type est l'identité n'a qu'un seul fils. Dans une DTD XML, un élément de type identité est déclaré par :

  ```
  <!ELEMENT annexe section                    >
  ```

- Le *choix* définit une alternative entre un ensemble de types. Un élément dont le constructeur de type est le choix n'a qu'un seul fils dont le type est une alternative parmi les types proposés par le choix. Dans une DTD XML, un élément dont le type est construit par un choix est déclaré de la façon suivante :

  ```
  <!ELEMENT para   (simple | list | titré) >
  ```

- La *liste* définit une séquence d'éléments de même type. Un élément dont le constructeur de types est la liste peut avoir plusieurs fils du type définissant la liste. XML permet spécifier des listes comportent un nombre non nul de fils (opérateur +) ou permet les listes vides (opérateur *).

  Le langage S, qui permet de définir les structures génériques de Thot, permet d'exprimer un intervalle donnant les cardinalités maximale et minimale d'un élément de type liste. Dans une DTD XML, un élément dont le constructeur de type est liste est défini de la façon suivante :

  ```
  <!ELEMENT list   (item)+                    >
  ```

- L'*agrégat* définit un ensemble d'éléments de types différents. En XML, un type agrégat définit une séquence d'éléments dont certains sont optionnels. Un élément de type agrégat est déclaré de la façon suivante :

  ```
  <!ELEMENT description (nom, adresse, (tel)?)  >
  ```

  Le langage S permet de définir des agrégats ordonnés et non ordonnés. Un agrégat non ordonné ne définit pas l'ordre dans lequel apparaissent les éléments fils.

  Ce constructeur n'existe pas dans la norme XML mais il peut être remplacé par une liste d'éléments dont le constructeur est le choix. La sémantique des ces constructeurs n'est cependant pas la même : un fils d'un agrégat ne peut être représenté qu'une seule fois dans la structure du document, tandis qu'un type défini dans une liste de choix peut avoir plusieurs occurrences.

Le rôle des constructeurs dans le système de types de documents permet de faire le parallèle avec les types de données. Le constructeur liste s'apparente au constructeur tableau (une séquence d'éléments du même type), l'agrégat s'apparente au type structure du langage C, et le constructeur choix à l'union.

Dans le système de types que nous proposons pour les documents structurés, nous notons l'ensemble des constructeurs de type **C**. Ces constructeurs sont : **C** = {`Choix`, `Liste`, `Agrégat`, `Identité`}.

IV.2.3 Arbres de types canoniques

Chaque structure générique de document, décrite par une DTD, peut être décrite par une expression de type. Les types définis par la DTD sont nommés et peuvent être circulaires. Comme les expressions de type des langages de programmation, les structures génériques des documents structurés peuvent être représentées sous forme de graphes de types orientés. La figure 28 montre un exemple de fragment de DTD et le graphe de types associé. Dans le graphe, les noms des types d'éléments sont présentés en caractères italiques, les constructeurs et les types de base sont indiqués en caractères romains.

```
<!ELEMENT répertoire  (entrée)+                          >
<!ELEMENT entrée       (nom, (ref_entrée | description) >
<!ELEMENT description (adr_perso, tel, adr_prof, tel)  >
<!ELEMENT adr_perso   (adresse)                          >
<!ELEMENT adr_prof    (adresse)                          >
<!ELEMENT nom         (#PCDATA)                          >
<!ELEMENT adresse     (#PCDATA)                          >
<!ELEMENT tel         (#PCDATA)                          >
<!ELEMENT ref_entrée  EMPTY                              >
<!ATTLIST ref_entrée  referred  CDATA                    >
```

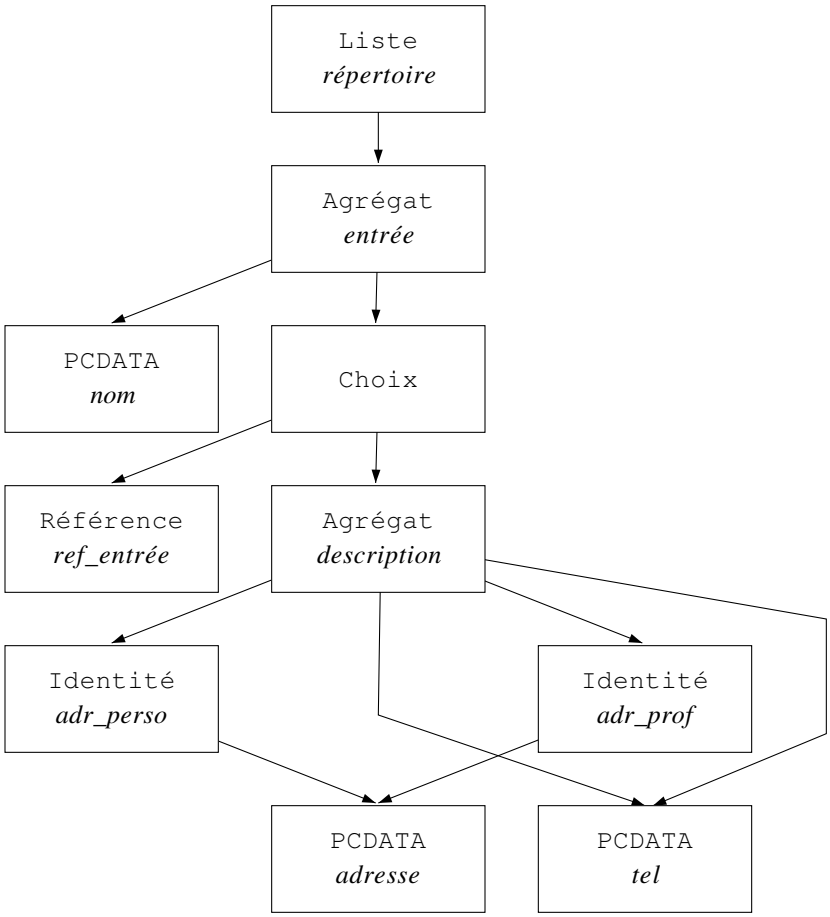


Figure 28 : DTD de la classe de documents Répertoire et graphe de types associé

La représentation de graphe n'est pas une structure de données adaptée à la comparaison. En effet, la recherche de relations autres que l'équivalence entre graphes est un problème difficile. Aussi allons nous définir une structure d'arbre adaptée aux algorithmes de comparaison que nous proposons dans le chapitre suivant.

Une structure générique S définit un ensemble de types $\mathbf{T}_S$ de façon récursive à l'aide des règles de constructions suivantes :

1. $\forall b \in \mathbf{B}_S$, $b \in \mathbf{T}_S$; *types de base* ;
2. $\forall t = c\ (t_1, t_2,...t_n)$, avec $c \in \mathbf{C}$ et $t_1, t_2,...t_n \in \mathbf{T}_S$, $t \in \mathbf{T}_S$; *types construits.*

Un *arbre de types canonique* [Akpotsui 93] est un développement du graphe de types. Ce graphe est exploré depuis le type qui doit être représenté et chaque nœud rencontré au cours de ce parcours provoque la création d'un nœud dans l'arbre canonique. Ainsi la figure 29 présente–t–elle l'arbre canonique du type *description* de l'exemple présenté ci–dessus. Dans cette représentation, les nœuds *adresse* et *tel* ont été répliqués pour obtenir une arborescence.

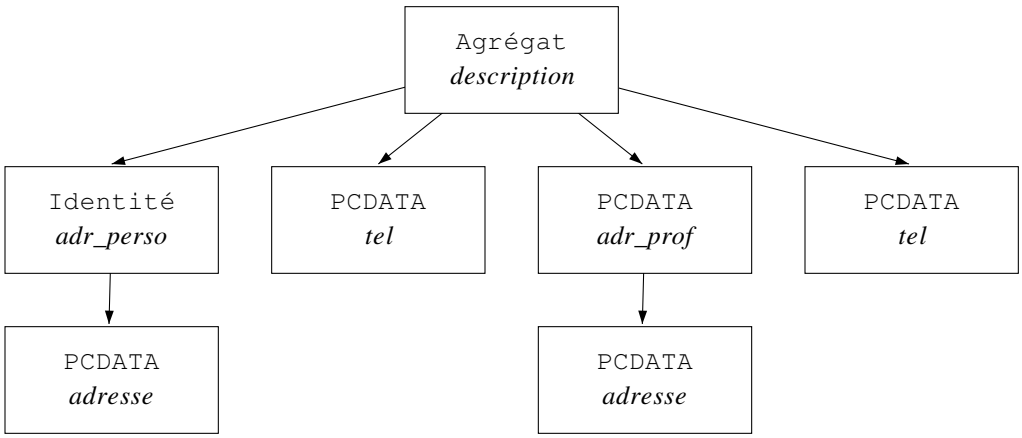


Figure 29 : Arbre canonique du type description

Une DTD peut définir des types d'éléments de façon récursive. Dans ce cas, le graphe de types comporte des cycles et il n'est pas possible de créer un arbre de type. Nous définissons un pseudo–type de base, appelé *récurrence*, qui est associé aux types qui ont une définition récurrente. Ainsi, nous pouvons représenter tout graphe de type sous forme d'arbre canonique. L'algorithme de comparaison présenté dans le chapitre suivant prend en compte les nœuds définis récursivement en effectuant un traitement particulier des nœuds terminaux du type *récurrence.*

Ces définitions étant posées nous allons nous intéresser à une relation portant sur les arbres de types servant de base à un système de transformation de document permettant la réalisation de classes de transformation que nous avons identifiées au chapitre II.

IV.3 Relations entre arbres de types pour la transformation de documents structurés

La section IV.1 a montré les spécificités des systèmes de types des documents structurés. Il en ressort que la notion d'équivalence diffère entre types de données et types d'éléments.

Pour la transformation de structures, les applications documentaires requièrent une relation entre types moins stricte que la relation d'équivalence utilisée par les compilateurs des langages de programmation. Par conséquent, nous définissons une relation de *transformabilité* entre types permettant de réaliser les deux principales classes de transformations que nous avons identifiées au chapitre II : les structurations et les réductions. Cette relation met en correspondance les nœuds de l'arbre des types des éléments source avec un arbre de type donné.

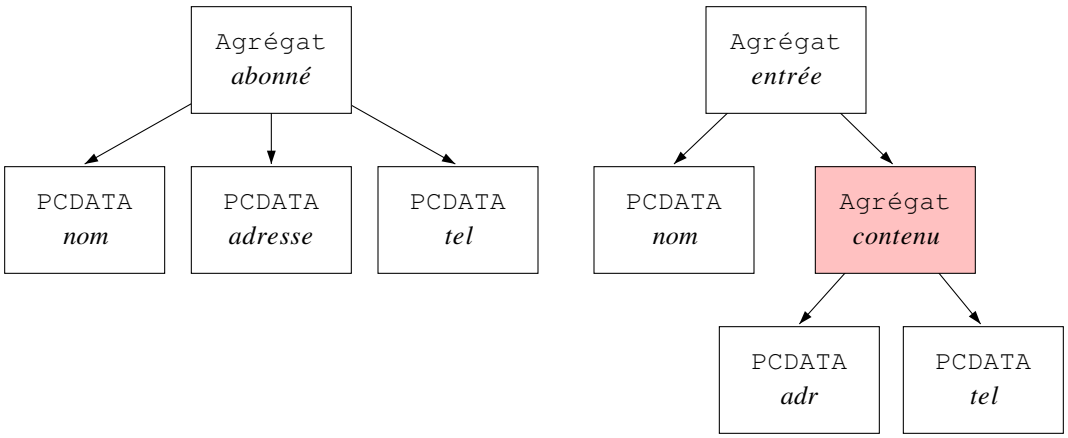
IV.3.1 Structurations

La première classe de transformations que nous considérons est celle des structurations. Une structuration est une transformation qui nécessite la création d'éléments de structure intermédiaires. Par exemple, la transformation d'un élément de type *abonné*, défini par la DTD *annuaire* donnée dans la figure 30, en un élément de type *entrée* défini dans la DTD *répertoire* nécessite la création de l'élément *contenu* qui n'a pas d'équivalent dans la structure *annuaire* au sens de l'équivalence défini par l'algorithme d'unification.

L'équivalence entre les arbres de types *abonné* et *entrée* n'est pas établie par l'algorithme d'unification à cause de la présence du nœud *contenu*. Cependant il est possible de transformer un élément du type *annuaire* en *répertoire*. Pour cela nous définissons une relation anti–symétrique d'un arbre de type dans un autre appelée *relation de massif*. Cette relation injective met en correspondance chaque nœud de l'arbre de types source avec un nœud de l'arbre de types destination ayant le même constructeur, en préservant les relations structurales de parenté et d'ordre entre les nœuds.

```
<!ELEMENT annuaire
              (département)*>
<!ELEMENT département
                    (commune)*>
<!ELEMENT commune  (abonné)*>
<!ELEMENT abonné
         (nom, adresse, tel)>
<!ELEMENT nom      (#PCDATA)>
<!ELEMENT adresse  (#PCDATA)>
<!ELEMENT tel      (#PCDATA)>
```

```
<!ELEMENT répertoire
                    (entrée)*>
<!ELEMENT entrée
               (nom,contenu)>
<!ELEMENT contenu  (adr,tel)>
<!ELEMENT nom      (#PCDATA)>
<!ELEMENT adr      (#PCDATA)>
<!ELEMENT tel      (#PCDATA)>
```



Classe annuaire *Classe répertoire*

Figure 30 : DTD et graphes de types des classes de documents répertoire et annuaire

Définitions

Sous−forêt : un ensemble de nœuds $\{t_1, ..., t_n\}$ est une *sous−forêt* d'un nœud t si et seulement si :

- $\forall i \in [1, n]$, t_i est un descendant de t.
- $\forall i \in [1, n-1]$, t_{i+1} n'est pas un descendant de t_i.
- $\forall i \in [1, n-1]$, t_{i+1} est un successeur de t_i dans l'ordre préfixe de l'arbre dont la racine est t.

Dans la figure 31, l'ensemble $\{t_{1.1}, t_{1.2}, t_2, t_{3.2}, t_4\}$ est une sous−forêt du nœud t.

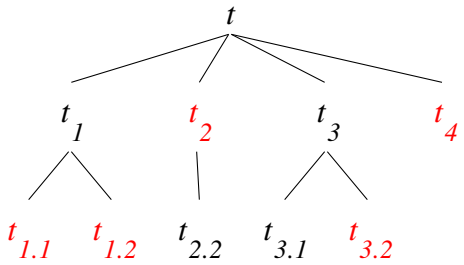


Figure 31 : Sous–forêt d'un arbre de types

Massif : un ensemble de nœuds $\{t_1, ..., t_n\}$ est un massif d'un nœud t si et seulement si $\forall\ i \in [1, n]$, t_i est un descendant de t.

Dans la figure 32, l'ensemble $\{t,\ t_{1.1},\ t_{1.2},\ t_2\ ,\ t_3,\ t_{3.1.1},\ t_{3.1.3},\ t_{3.2}\}$ est un massif du nœud t.

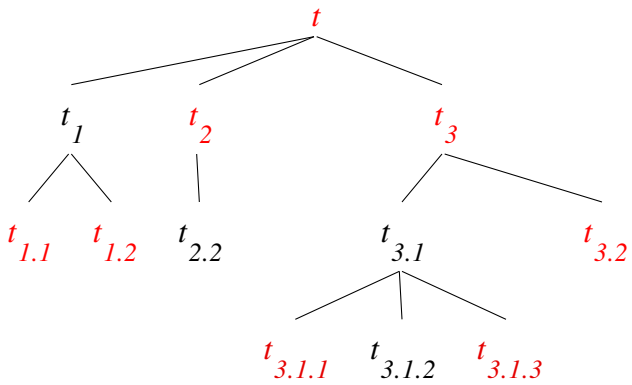


Figure 32 : Massif d'un arbre de types

Relation de massif : Une relation de massif entre un arbre de type t et un arbre de type t' est une relation injective M de t dans t' telle que le massif image de t par M préserve les constructeurs et les relations hiérarchiques des nœuds de t. M est formellement de définie par :

- t et $M(t)$ sont le même type de base, ou
- t et $M(t)$ sont de constructions respectives $c(t_1, \ldots, t_n)$ et $c'(t'_1, \ldots, t'_m)$, où c et $c' \in \mathbf{C}$, et

  1. $c = c'$
  2. il existe une sous–forêt de $M(t)$, $E_{t'} = \{t'_{q1}, ..., t'_{qn}\}$ de cardinalité n, telle que $\forall i \in [1, n]$, $t'_{qi} = M(t_i)$.

Dans la figure 33, une relation de massif associe les nœuds de l'arbre de types t et un massif de l'arbre de types t', présenté en rouge.

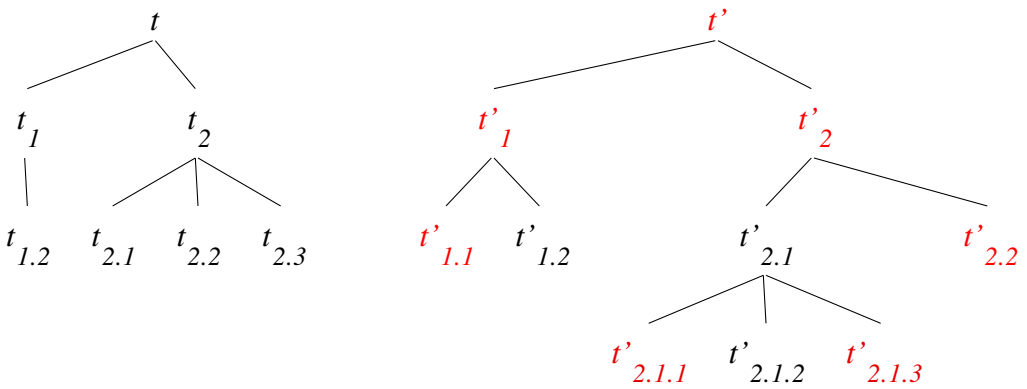


Figure 33 : Relation de massif entre arbre de types

La relation de massif a la propriété de conserver les relations hiérarchiques entre éléments : si un nœud t_i est un descendant de t dans l'arbre de types source, le nœud $M(t_i)$ est un descendant du nœud $M(t)$ dans l'arbre de types cible.

Notation

S'il existe une relation de massif entre un arbre de types U et un arbre de types V nous noterons U $\subseteq$ V.

IV.3.2 Réductions

Les réductions sont les transformations inverses des structurations : ces transformations induisent l'élimination d'éléments de structure car un ensemble de nœuds de l'arbre de type source n'ont pas de correspondance dans l'arbre de types cible. Un exemple de réduction est la transformation inverse de la précédente qui permet de transformer une instance d'*annuaire* dans le type *répertoire*. Cette différence implique que les arbres de types source et cible ne peuvent pas être mis en correspondance par l'algorithme d'unification ni par une relation de massif.

Nous définissons une nouvelle relation entre un arbre de types source et un arbre de type cible représentant une structure réduite : la relation d'*absorption*.

Définitions

Nœud absorbable : un nœud t d'un arbre de types est *absorbable* si et seulement si il est défini par un constructeur c et son nœud parent est défini par le même constructeur.

Dans l'exemple de la figure 30, le nœud *contenu* est absorbable car son constructeur est *agrégat* et le constructeur de son père (*entrée*) est également *agrégat*.

Équivalence à un nœud près : Nous définissons l'équivalence à un nœud près comme une approximation de l'équivalence calculée par l'algorithme d'unification, en ignorant un nœud dans l'arbre source et un nœud ayant la même position dans l'image par la relation d'équivalence E.

Deux arbres de types t et t' sont équivalents à t_0, t'_0 près si et seulement si :

- t_0 est un descendant de t, t'_0 est un descendant de t' ;
- il existe une relation d'équivalence E entre les arbres $t - \{t_0\}$ et $t' - \{t'_0\}$;
- t'_0 est fils de l'image du parent de t_0 par E ;
- t'_0 est le successeur de l'image du prédécesseur de t_0 par E.

Relation d'absorption : il existe une relation d'absorption N entre deux arbres de types t et t' si et seulement si il existe un nœud $t_a = c\,(t_1,\ldots, t_n)$ appartenant à la descendance de t et un nœud $t'_a = c\,(t'_1,\ldots, t'_m)$ descendant de t' tels que :

1. Les arbres de types t et t' sont équivalents à t_a, t'_a près.
2. il existe $i \in [1...n]$ tel que $t_i = c\,(t_{i.1},..., t_{i.p})$ soit absorbable, et
   - $m = n+p-1$
   - Les types $t_1,..., t_{i-1}, t_{i+1},..., t_n$ sont respectivement équivalents aux types $t'_1,..., t'_{i-1}, t'_{i+p},..., t'_m$
   - Les types $t_{i.1},..., t_{i.p}$ sont respectivement équivalents aux types $t_i,..., t_{i+p-1}$.

Un type peut être absorbé si l'image de son parent peut contenir les images de ses fils. Ainsi dans l'exemple (figure 34), le type *abonné* est en relation avec le type *entrée* par absorption du type *contenu* : les constructeurs des ces trois types sont des agrégats et les fils du type *contenu* sont équivalents aux types *adresse* et *tel*.

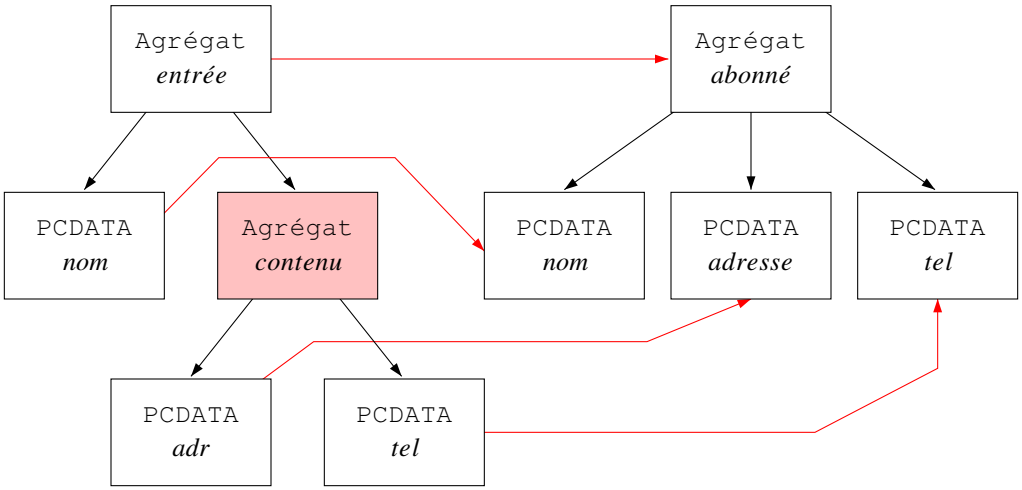


Figure 34 : Relation d'absorption entre les types abonné et entrée

La relation d'absorption a les mêmes propriétés que la relation de massif : elle conserve l'ensemble des types de base de l'arbre de types source, car par définition, seuls des nœuds intermédiaires d'un arbre de types peuvent avoir un constructeur similaire à celui de leur parent. Les nœuds feuille sont des types de base. Les relations hiérarchiques entre les nœuds de l'arbre sont également conservées.

La relation d'absorption ne permet pas d'effectuer l'ensemble des réductions possibles d'un type d'élément. Certaines transformations sont fondées sur des relations qui suppriment des éléments dont le type n'est pas absorbable ; c'est en particulier le cas lorsqu'une alternative d'un type choix n'est représenté ni dans la DTD cible, ni dans la structure du document source. Pour prendre en compte ces transformations, nous proposons dans le chapitre suivant une extension de la notion d'absorption qui consiste à éliminer de l'arbre de types source les nœuds qui ne sont pas instanciés dans la structure du document source (voir chapitre V, section V.4.3).

IV.3.3 Combinaison des relations de massif et d'absorption

Les transformations les plus courantes ne sont généralement pas de simples réduction ou structurations, mais une combinaison de ces relations. Pour cette raison, nous définissons la relation de transformabilité, qui est une combinaison de relations d'absorption et d'une relation de massif.

Définition

Relation de transformabilité : soit t et t' deux arbre de types, il existe une relation de transformabilité entre t et t' si seulement si il existe un ensemble d'arbres de types t_0, ..., t_n tel que :

1. il existe une relation d'absorption N_0 telle que $t_0 = N_0\ (t)$;
2. $\forall\ i \in [1,\ n]$, il existe une relation d'absorption N_i telle que $t_i = N_i\ (t_{i-1})$;
3. il existe une relation de massif M telle que $M(t_n) = t$'.

Par composition des relations, les nœuds terminaux de l'arbre de types source sont conservés par la relation de transformabilité. De même, les images conservent la hiérarchie des nœuds de l'arbre source.

La figure 35 montre la composition de relations d'absorption et de massif qui mettent en relation les arbres de types des structures génériques *annuaire* et *répertoire*.Les deux absorptions permettent de supprimer les nœuds *commune* (1) et *département* (2) qui ne peuvent pas être représentés dans la structure répertoire. Une relation de massif (3) met en correspondances le nœud de l'arbre de type *image* de ces deux absorptions avec les nœuds de l'arbre de types *répertoire*.

Dans le chapitre suivant nous proposons un algorithme de recherche de ces relations appliqué aux structures génériques de documents.

```
<!ELEMENT annuaire (départ)* >
<!ELEMENT départ   (commune)*>
<!ELEMENT commune  (abonné)* >
<!ELEMENT abonné
          (nom, adresse, tel)>
<!ELEMENT nom      (#PCDATA) >
<!ELEMENT adresse  (#PCDATA) >
<!ELEMENT tel      (#PCDATA) >
```

```
<!ELEMENT répertoire (entrée)*>
<!ELEMENT entrée (nom,contenu)>
<!ELEMENT contenu    (adr,tel)>
<!ELEMENT nom      (#PCDATA)   >
<!ELEMENT adr      (#PCDATA)   >
<!ELEMENT tel      (#PCDATA)   >
```

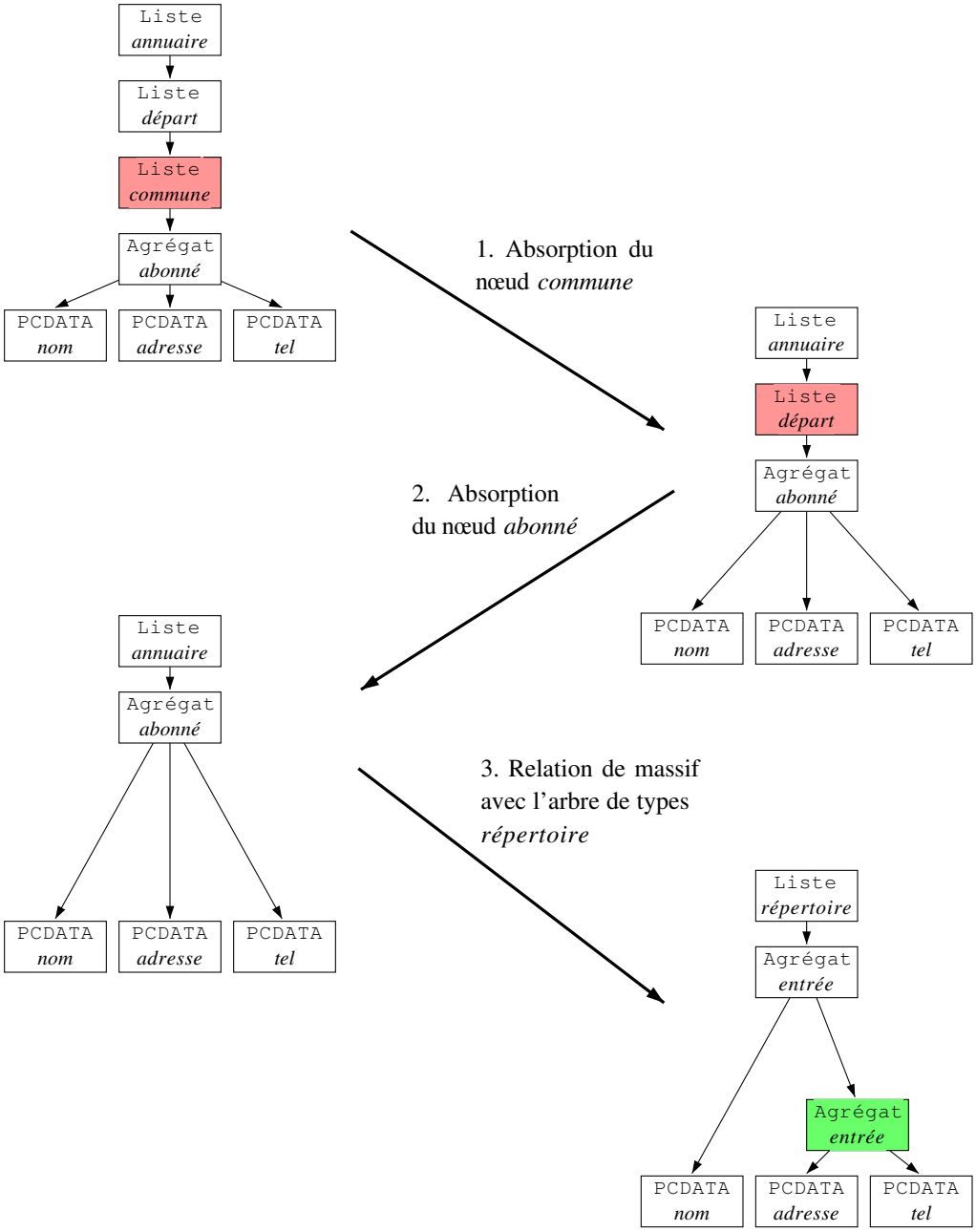


Figure 35 : Composition de relations d'absorption et de massif pour la transformation d'annuaire en répertoire

IV.4 Transformations fondées sur les systèmes de types

Quelques systèmes de transformation s'appuient sur les définitions génériques des types pour spécifier les transformations en terme de schémas de traduction entre grammaires. Comme dans les systèmes présentés dans le chapitre précédent, ces systèmes peuvent utiliser une approche dirigée par la source ou implémenter une sémantique de transformation qui leur est propre (contraintes, actions explicites).

Parmi les travaux que nous avons étudiés, trois proposent une modélisation des systèmes de types pour la transformation, et s'apparentent en cela de la proposition du système de types faite dans la section précédente ([Furuta 88b], [Akpotsui 93] et [Hirvonnen 95]) et deux proposent des systèmes plus achevés (SIMON et SYNDOC).

Modèles de systèmes de types

Dans [Furuta 88b], le problème de la relation entre opérations élémentaires de transformation et conformité du document avec sa structure générique est soulevé. Les auteurs posent les fondements d'un langage de transformation basé sur des opérateurs de modification des règles de production des grammaires définissant les structures génériques.

Dans sa thèse, [Akpotsui 93], [Akpotsui 97] propose un modèle fonctionnel pour représenter les types d'éléments et formalise la notion d'adoption d'éléments. Ce modèle se fonde sur une relation d'isomorphisme dont nous avons montré les limites dans la section IV.3.

[Hirvonnen 95] présente des études de cas qui mène à la définition de relations élémentaires de transformation de façon similaire à [Furuta 88b] : l'extension (ajout d'éléments), la suppression, le renommage, l'application d'un ordre sur les éléments, la recherche d'éléments spécifiques et l'adoption d'éléments par un nouvel élément père. Il a également défini un langage de transformation explicite permettant de combiner ces transformations élémentaires. Ce langage est la base d'une méthode de transformation dirigée par la source avec une génération linéaire.

Systèmes de transformation expérimentaux

Le système SIMON [Feng 93] définit des schémas, nommés Higher–order Attribute Grammar (HAG) qui sont des extensions des grammaires attribuées. Les HAG spécifient à la fois des traductions dirigées par la source et des actions sémantiques incluses dans les parties droites des règles de la grammaire attribuée.

La transformation est effectuée en trois phases (figure 36) :

1. Analyse syntaxique des arbres source : construction de l'arbre syntaxique correspondant au schéma HAG.
2. Évaluation des attributs : construction de l'arbre décoré (évaluation des attributs et de leur valeur conformément aux règles sémantiques).
3. Génération de l'arbre résultant de la transformation à partir de l'arbre décoré, en respectant la grammaire cible.

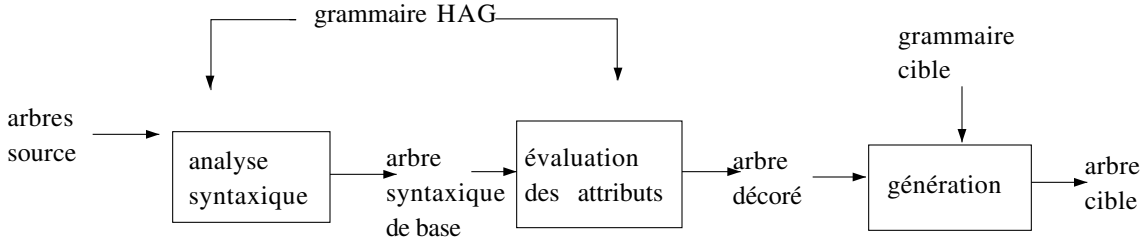


Figure 36 : Transformation de structure basée sur les HAG

SYNDOC [Kuikka 93] permet des transformations par l'intermédiaire de grammaires à clauses définies (Definite Clause Grammar ou DCG) qui sont également une généralisation des grammaires hors–contexte, mais qui sont exécutables car elles sont une variante rationnelle d'une classe de programmes Prolog.

Ces systèmes sont proches de ceux étudiés dans le chapitre précédent car ils se fondent sur une déclaration explicite des transformations. Nous avons choisi de faire figurer ces systèmes dans cette partie car les expressions de transformation ne s'appuient pas sur les structures d'éléments mais sur les structures génériques définissant les documents.

La principale critique que l'on peut faire à ces systèmes est qu'ils réclament une connaissance experte des structures des documents manipulés et de leur représentation sous forme de grammaire. D'autre part, bien que ces systèmes permettent une spécification des transformations au niveau des structures génériques, la spécification d'un nombre important de transformations est nécessaire dès lors que de multiples classes de documents sont impliquées.

IV.5 Synthèse

Dans ce chapitre nous avons défini une notion de type au sens des langages de programmation (types de données) appliquée au cas des documents structurés (types d'éléments).

Les méthodes de conversion de variables entre types de données (coercition) et de typage des données s'appuient sur une notion d'équivalence calculée par l'algorithme d'unification qui est présenté dans l'annexe A.

Une comparaison des systèmes types de données et des types d'éléments de document a permis d'identifier les spécificités de ces derniers. Ces spécificités se situent au niveau de la définition d'un type :

- les constructeurs de types sont identiques quel que soit la structure générique ;
- des attributs typés peuvent être portés par les éléments,

et au niveau de l'utilisation des données typées que font les applications comme :

- le traitement des documents incomplets ;
- la modularité et l'inclusion d'éléments appartenant à des structures génériques différentes dans un même document.

Ces différences entre ces systèmes de types font que l'équivalence telle qu'elle est définie pour la vérification et l'interprétation de programmes est une notion trop stricte pour être appliquée à la transformation de documents. De plus, au cours d'une transformation de documents, des éléments de structure peuvent être créés ou disparaître, il est donc nécessaire de traiter des relations mettant en correspondance des types ayant des structures différentes.

C'est pourquoi nous avons proposé dans ce chapitre une définition d'un système de types adaptée aux documents et nous avons proposé des relations entre types qui prennent en compte les spécificités des documents. Ces définitions ont pour objet de permettre la recherche automatique de similarités de structures syntaxiques (au sens des systèmes de types), représentées par les relations de massif et d'absorption, dans le but de les transformer. En ce sens, elles vont plus loin que les relations de transformation proposées dans [Hirvonnen 95] : ces dernières portent une sémantique plus importante que les relations de facteur et de massif mais ne peuvent être identifiées par un algorithme de comparaison automatique. C'est pour cela qu'elles sont exprimées en utilisant un langage de transformation explicite.

Les relations antisymétriques de massif et d'absorption permettent d'associer un type source avec un type cible dont les structures diffèrent par le nombre de nœuds. Cependant des similitudes persistent entre les types ayant de telles relations :

- L'ordre préfixe des nœuds concordant dans les arbres de types est le même.
- Les types de base sont conservés par les relations.

Ces propriétés impliquent qu'un système de transformation basé sur ces relations conserve l'ordre des éléments dans le document et garantisse que l'intégralité du contenu des éléments source soient conservés.

Dans le chapitre suivant, nous définissons un processus de transformation de documents structurés basé sur la comparaison des structures génériques et la recherche de relations de massif et d'absorption.

Chapitre V

Transformations automatiques

Ce chapitre présente le processus de transformation automatique que nous avons développé dans l'éditeur Thot. Ce système de transformation se base sur la recherche des relations de massif et d'absorption entre le type des éléments à transformer et le type dans lequel ils doivent être transformés.

Le système de transformation présenté ici s'inscrit dans la continuité des travaux présentés dans [Claves 95].

L'objectif de la transformation est de trouver une relation entre les types source et cible préservant le plus possible la structure du type source.

Le système de transformation de Thot est composé de deux modules (*cf.* figure 37) : le premier intègre l'algorithme de recherche des relations entre les structures génériques source et cible et produit un ensemble de relations entre types d'éléments ; le second exploite ces relations pour transformer les éléments source de façon à produire l'instance cible de la transformation.

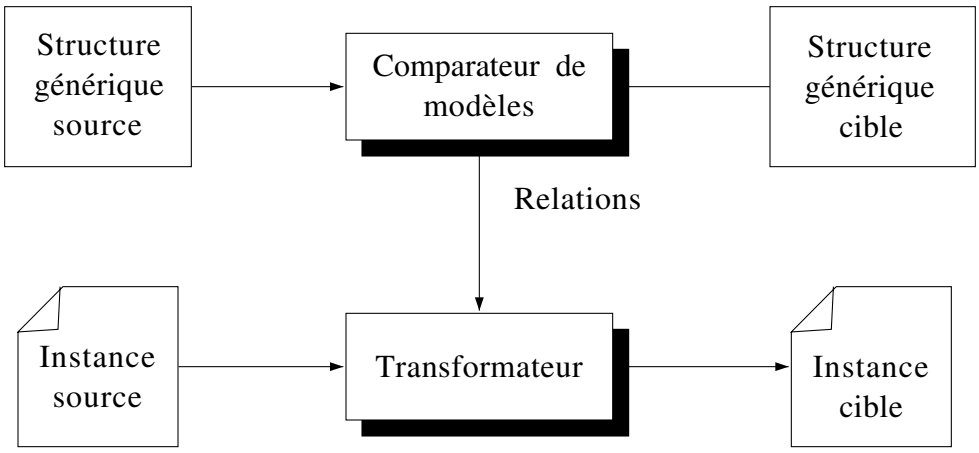


Figure 37 : Transformation automatique de document

Les algorithmes de comparaison d'arbres ont une complexité élevée et nécessitent une place en mémoire importante. Ils font d'autre part appel à des primitives d'accès dans les arbres entraînant un surcoût de temps d'exécution. C'est pour ces raisons que nous représentons les arbres sous forme de chaînes parenthésées qui permettent une comparaison plus efficaces par l'application d'algorithmes de pattern matching éprouvés.

La première section de ce chapitre donne quelques préliminaires à la recherche de massifs. Nous donnons ensuite l'algorithme de comparaison de types utilisé pour la recherche de relations. Nous présentons ensuite le module de transformation. Ces deux sections se basent sur une définition arborescente des types d'éléments. Pour prendre en compte la récursivité dans les définitions de types d'éléments, nous proposons dans la section V.4.4 une extension de l'algorithme de comparaison. Une extension pour prendre en compte les absorptions spécifiques aux éléments transformés est également présentée. Nous présentons ensuite l'intégration du système de transformations dans l'éditeur Thot et les commandes d'édition qu'il permet d'implémenter ou d'améliorer. Nous terminons ce chapitre sur une discussion de la pertinence des transformations réalisées avec ce système et sur les moyens d'améliorer ce mode de transformation.

V.1 Préliminaires

Nous présentons ici l'algorithme donné par Frilley dans [Frilley 89]. Cet algorithme implémente un automate de reconnaissance de massifs entre deux arborescences. Il se base sur une forme linéaire des arborescences étiquetées : *les mots de Dyck*. Le langage des mots de Dyck sur un alphabet S, noté $\mathbf{D}(S)$ est un langage algébrique sur l'alphabet $\Sigma = S \cup \overline{S}$, où $\overline{S}$ est l'ensemble des inverses des éléments de S. Notons que chaque élément de S à un élémnt inverse unique dams $\overline{S}$. Ce langage est défini récursivement de la façon suivante :

1. $w = \varepsilon$
2. $w = x\ w'\ \overline{x}\ w''$ où $x \in S$ et w' et w'' sont eux–mêmes des mots de Dyck.

Frilley montre qu'il existe une bijection entre les arbres ordonnés d'une forêt d'arbres étiquetés et le mot de Dyck correspondant.

Le langage des mots de Dyck est un langage parenthésé donc un langage algébrique. Par conséquent, il est reconnu par un automate à pile. Le problème de reconnaissance de massifs peut s'écrire comme un problème d'intersection de langages algébriques.

L'automate proposé par Frilley permet la reconnaissance d'un massif (U) dans une forêt d'arbres ordonnés (V). Les données de l'automate sont deux mots de Dyck u et v, représentant respectivement les forêts U et V. Il est démontré que le langage reconnu par cet automate est non vide si et seulement si la forêt U est isomorphe à un massif de V.

L'automate de Frilley ne permet pas de reconnaître tous les massifs d'arborescence. Ainsi si l'on considère l'arborescence ordonnée représentée par le mot de dyck $aabb\overline{baa}cc\overline{aa}$, l'automate ne permet pas la reconnaissance de $ab\overline{baa}c\overline{ca}$ qui est de toute évidence un massif de l'arborescence précédente. Plus généralement l'automate proposé par Frilley ne reconnaît pas les massifs de la forme $ad\overline{aa}d'\overline{a}$ dans une arborescence $a..ad\overline{aa}d'\overline{a}..\overline{a}$ où d et d' sont des mots de Dyck. Ceci est dû au fait que le retour sur une erreur partielle ne remet pas en cause la progression de l'indice dans la chaîne cible définissant les états de l'automate.

Pour résoudre ce problème nous proposons une modification de l'automate de Frilley permettant de rechercher une solution dans la descendance d'un nœud non reconnu en cas d'erreur partielle de l'algorithme (voir section V.2.2).

Nous adaptons ce principe de reconnaissance à la comparaison des structures génériques des documents. Nous proposons une forme linéaire des structures génériques appelées *empreintes* de types. Les empreintes appartiennent au langage de Dyck sur un alphabet représentant les constructeurs de types (section V.2.1).

V.2 Comparaison des arbres de types

L'algorithme de recherche de relation entre types d'éléments prend en entrée un type d'élément source et un type cible pour produire un ensemble de relations entre ces types appelé *couplage*.

Nous commençons par définir la représentation linéaire sous forme d'*empreinte* des types d'éléments. Nous donnons ensuite l'automate puis l'algorithme de comparaison d'empreintes pour la recherche de relations de massif.

Nous adaptons l'algorithme de Frilley [Frilley 89] aux spécificités des arbres de types d'éléments de documents : l'ensemble des types représenté par les nœuds terminaux (les types de base) est différent de l'ensemble des types représenté par les nœuds intermédiaires (les types construits).

De plus, nous proposons une extension de l'automate pour prendre en compte la recherche de solutions dans un sous–arbre lorsque le couplage d'un nœud de l'arbre cible a été tenté mais n'a pas pu être réalisé.

V.2.1 Génération des empreintes de type

Les étapes nécessaires à la génération de l'empreinte d'un type d'élément sont :

1. la mise sous *forme canonique* des graphes de types. Cette opération consiste à redéfinir un type sous une forme qui ne fait pas usage d'identificateurs déjà utilisés, de manière à ce que chaque type soit identifié de manière unique par son identificateur ;
2. les arbres canoniques sont ensuite ordonnés car l'algorithme de comparaison travaille sur des arbres ordonnées (cf. section V.2.2) ;
3. enfin, la *linéarisation* des arbres ordonnés permet de produire des chaînes de caractères représentant leur structure. Durant cette étape, on fait abstraction des noms des types d'éléments, seuls sont conservés :
   - les constructeurs ;
   - les liens de parenté ;

- l'ordre entre les nœuds frères ;
- les types des feuilles.

V.2.1.1 Mise sous forme canonique

Le type d'élément tel qu'il est défini dans la DTD est d'abord mis sous forme canonique. La mise sous forme canonique d'un type se fonde sur la représentation sous forme de graphe de type et produit un *arbre de types canonique*.

Les opérations nécessaires à la mise sous forme canonique d'un graphe de types sont :

- L'élimination des types identité.
- La définition d'arbres finis : le graphe de types est transformé en graphe connexes sans cycles. Pour cela, les règles récursives sont traitées séparément (voir section V.4.4). Le graphe de types est exploré et chaque branche est répliquée si elle a déjà été parcourue précédemment.
- La représentation systématique des éléments optionnels de la DTD. Ces derniers seront éventuellement ignorés par la relation de massif. Par contre, les éléments exclus ne sont pas représentés dans la branche d'arbre concernée par l'exclusion.

Les nœuds non terminaux des arbres canoniques sont étiquetés par les constructeurs choix, liste et agrégat. Les feuilles de ces arbres sont étiquetées par les types de base.

V.2.1.2 Ordre pour les arbres de types

L'algorithme de recherche de massifs nécessite la définition d'un ordre sur les arbres de types sans quoi le problème de la détermination de l'existence d'une relation de massif entre deux types est indécidable [Kilpeläinen 92].

Plusieurs ordres peuvent être proposés pour prendre en compte :

- les constructeurs des fils de la racine du sous−arbre de types,
- la profondeur du sous−arbre de type,
- le nombre de types compris dans le sous−arbre.

La relation d'ordre sur les constructeurs est notée $<_c$.

Le choix de la relation d'ordre associée à l'arbre de types est déterminant pour la pertinence des relations de massif trouvées par l'algorithme de comparaison d'empreintes (voir section V.6). Pour présenter l'algorithme, nous choisissons la relation d'ordre suivante sur les types canoniques d'une structure générique S :

1. si $t \in \mathbf{B}_S$ (type de base) et $t' \in \mathbf{C}_S$ (type construit) alors $t < t'$.
2. si t et $t' \in \mathbf{C}_S$ avec $t = c\,(t_1,..., t_m)$ et $t' = c'(t'_1,...,t'_n)$ alors

   $t < t'$ si et seulement si :

   $c <_c c'$

   ou

   $c = c'$, $m < n$ et $\forall\ i \in [1,.., m]$, $t_i = t'_i$.

   ou

   $c = c'$, $\exists i \in [1,.., \min(m, n)]$ tel que $\forall j \in [1,.., i-1]$, $t_j = t'_j$ et $t_i < t'_i$.

Cette définition implique le choix d'un ordre sur les types de base et sur les constructeurs. Pour les types de base, nous choisissons arbitrairement :
`T (texte) < G (graphique) < S (symbole) < P (image)`
l'ordre des feuilles n'ayant pas de conséquence pour la comparaison (c.f. figure 38). Les constructeurs sont ordonnés selon deux critères :

- le critère principal est l'arité des types qu'ils sont susceptibles d'engendrer : les types de constructeur liste ont une cardinalité égale à un, les types de constructeur choix et agrégat ont une arité supérieure à un.

  liste $<_c$ *choix*, *agrégat*.
- un critère secondaire est l'arité des instances des éléments des types concernés : les éléments dont le constructeur de type est le choix ont un unique fils, tandis que les éléments dont le constructeur de type est agrégat ont plusieurs fils.

  choix $<_c$ *agrégat*.

Cet ordre a pour but de comparer prioritairement les sous−arbres comprenant le plus petit nombre de nœuds, ceci pour trouver les relations de massifs les plus « compactes » possible.

V.2.1.3 Linéarisation des arbres de types canoniques

Une empreinte de type est ensuite construite à partir de l'arbre de types produit par la canonisation (voir section IV.2.3). Une empreinte d'un type est une chaîne de caractères contenant des parenthèses pour représenter les constructeurs de type (différentes sortes de parenthèses sont utilisées pour représenter les différents constructeurs) et des lettres pour représenter les types de base. Le parenthésage des empreintes reflète la structure générique du

type. La figure 39 représente la DTD définissant le type *exercice*, l'arbre de type canonique correspondant et l'empreinte du type.

Dans les empreintes (figure 38), le constructeur choix est représenté par des crochets, le constructeur liste par des parenthèses, le constructeur agrégat par des accolades. Les types de base sont identifiés par les lettres `T` pour le type *texte*, `G` pour le type *graphique*, `S` pour le type *symbole mathématique*, `P` pour le type *image* et R pour le type *référence*. Les autres types de base sont représentés par des lettres minuscules pour les types définis comme vides et par le caractère @ pour les pseudo–types de base *récurrence*.

Constructeur	*Liste*	*Choix*	*Agrégat*
Caractère	**(**	**[**	**{**

Type de base	*texte*	*graphique*	*symbole*	*image*
Caractère	**T**	**G**	**S**	**P**

Type de base	*référence*	*autres*	*récurrence*
Caractère	**R**	**a, b, c...**	**@**

Figure 38 : Correspondance entre types et caractères des empreintes

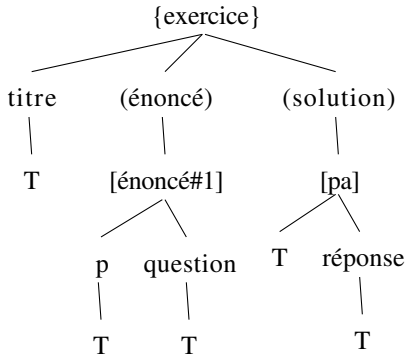
Un alphabet des constructeurs et des types de base est ainsi établi : S = {'{', '[', '(', 'T', 'G', 'S', 'P','a',...,'z'}, L'ensemble $\overline{S}$ des inverses est {'}', ']', ')', '$\overline{\text{T}}$', '$\overline{\text{G}}$', '$\overline{\text{S}}$', '$\overline{\text{P}}$','$\overline{\text{a}}$',..., '$\overline{\text{z}}$'}. Les applications associant un nœud de l'arbre de types au caractère représentant son constructeur dans S sont notées e pour les types construits et b pour les types base.

Dans les empreintes, nous confondons les caractères représentant les types de base et leur inverses. En effet, les types de base étant des feuilles des arbres de types, l'opération de réduction effectuée par les transitions M_1 et $M_{1'}$ de l'automate présenté dans la section V.2.2 est unifiée pour être réalisée par la transition M_t.

DTD :

```
<!ELEMENT exercice    (titre, énoncé, solution) >
<!ELEMENT énoncé      (p | question)+           >
<!ELEMENT solution    (pa)+                     >
<!ELEMENT titre       (#PCDATA)                 >
<!ELEMENT p           (#PCDATA)                 >
<!ELEMENT question    (#PCDATA)                 >
<!ELEMENT pa          (#PCDATA | réponse)       >
<!ELEMENT réponse     (#PCDATA)                 >
<!ATTRLIST question   ident    ID     #REQUIRED >
<!ATTRLIST réponse    ident    ID     #REQUIRED >
<!ATTRLIST exercice   niveau   CDATA  "1"       >
```

Arbre de types :



Empreinte :

```
{T([TT])([TT])}
```

Figure 39 : DTD, arbre de types et empreinte du type exercice

La définition donnée des empreintes de types est proche de celle des mots de Dyck. La différence réside dans la représentation des nœuds terminaux par un caractère unique.

Définition

L'*empreinte générique* d'une forêt d'arbres de types est un mot défini par les règles suivantes :

1. $w = b(x)$ si $x \in \mathbf{B}_S$
2. $w = e(x)\ w'\ \overline{e(x)}\ w''$ *si* $x \in \mathbf{C}_S$ et w' et w'' sont les empreintes respectives des enfants et des successeurs de x.

Comme pour les mots de Dyck, une bijection ϕ est définie entre les arbres canoniques et les empreintes. Remarquons que :

- À tout arbre canonique A du système de types des documents structurés, on associe un mot unique $\Phi(A) \in \mathbf{D}(S)$ définie par : $\Phi(A) = \Phi_0(r(A))$

où r(A) est la racine de l'arbre A et Φ_0 l'application de A dans D(S), elle–même définie récursivement par :

$$\Phi_0(x) = e(x) \sum_{y \in Fils(x)} \Phi_0(y)\ \overline{e(x)}$$

(si x est une feuille : $\Phi_0(x) = b(x)$).

- L'application Φ est une bijection de $A°$, l'ensemble des arbres canoniques de types sur l'ensemble **D**(S).
- À tout nœud x d'un arbre canonique A on peut associer un unique couple d'entiers (i, j) indices respectifs dans Φ(A) de $e(x)$ et $\overline{e(x)}$ (pour les nœuds terminaux, on associe le couple (i, i) où i est l'indice de $b(x)$). Ces indices sont ceux permettant la construction de Φ^{-1}. On notera $i = \varphi_A(x)$ et $j = \overline{\varphi}_A(x)$ (et lorsqu'il n'y a pas ambiguïté $i = \varphi(x)$ et $j = \overline{\varphi}(x)$. Ces deux applications sont injectives.
- L'application $\alpha = \overline{\varphi} \circ \varphi^{-1}$ qui pour un nœud x associe l'indice de $e(x)$ à celui de $\overline{e(x)}$ dans Φ(A) est bijective. α associe le caractère « parenthèse ouvrante » associé à un nœud avec le caractère « parenthèse fermante » associé au même nœud.

Un tableau de références vers les nœuds de l'arbre canonique de types est construit simultanément à l'empreinte. Ce tableau est utilisé lors de la génération des éléments transformés pour lier un indice de l'empreinte au type canonique qu'il représente. Il implémente Φ_0^{-1} (voir section V.3).

V.2.2 Comparaison des empreintes

V.2.2.1 Principe de la comparaison

La comparaison des empreintes de types pour la recherche de massif se fait à l'aide d'un l'automate à pile.

À chaque couple d'indices (i, j) des empreintes source et destination est associé un état de l'automate. À chaque état, l'automate compare les caractères u_i et v_j et effectue une transition vers un nouvel état. Une pile permet de mémoriser les caractères de S rencontrés dans l'empreinte cible alors que leur image par α n'a pas encore été rencontrée.

L'état initial (1, 1) est celui associant les indices représentant deux types de plus haut niveau (deux premiers caractères des empreintes). Les états terminaux sont ceux associant l'indice de la parenthèse fermante du type source de plus haut niveau (|u|+1) avec un indice quelconque de l'empreinte destination

(l'ensemble des indices de l'empreinte source à été parcouru si et seulement un massif à été trouvé). La pile contient des triplets (caractère, état de retour, compteur) définis comme suit :

- Le caractère est la parenthèse ouvrante (v_j) correspondant à l'état courant lorsque le triplet à été empilé.
- L'état de retour (s) est utilisé pour les retours arrière en cas d'échec d'une solution partielle.
- Le compteur (k) totalise les parenthèses ouvertes identiques au symbole de sommet de pile depuis qu'il à été empilé.

V.2.2.2 Automate de comparaison

La comparaison des empreintes de types pour la recherche de massif se fait à l'aide de l'automate à pile $\{Q, A, Y, q_{init}, Q_{fin}, M\}$:

- Ensemble des états : $Q = [1, |u| + 1]_N \times [1, |v| + 1]_N$
- Alphabet d'entrée : $A = \Sigma(u) \cup \Sigma(v)$
- Alphabet de pile :
  $Y = \Sigma(v) \times ([0, |u| + 1]_N \times [0, |v| + 1]_N) \times [0, |v| + 1]_N$
- État initial : $q_{init} = (1, 1)$
- Symbole initial de la pile : ε
- Ensemble des états terminaux : $Q_{fin} = \{(|u| +1, j) \mid j \in [1, |v| +1]_N\}$
- Mouvements élémentaires :
  $M = M_t \cup M_1 \cup M_{1'} \cup M_2 \cup M_3 \cup M'_3 \cup M_4 \cup M_5$.

  Chaque transition M_i est définie par un tuple {état initial, caractère d'entrée dans l'empreinte cible, sommet de pile initial, état suivant la transition, sommet de pile suivant la transition} et par un ensemble de conditions sur l'état initial et le sommet de pile :

  – La transition M_t est activée lorsque les symboles source et cible correspondant à l'état courant sont des symboles de base. On progresse sur les deux empreintes à la fois.

    $M_t = \{(i, j),\ v_j,\ (y, s, k),\ (i+1, j+1),\ (y, s, k)\}$

    conditions : $u_i = v_j$, $u_i \in \mathbf{B}$

  – La transition M_1 est activée lorsque les symboles source et cible coïncident entre eux et avec l'inverse du symbole en sommet de pile, le compteur étant égal a 0. Cette transition est activé lorsqu'un type

et sa descendance a été reconnu. L'automate passe alors de l'état (i, j) à l'état $(i+1, j+1)$ en dépilant. On progresse sur u et v.

$M_1 = \{(i, j),\ v_j,\ (v_j^{-1}, s, 0),\ (i+1, j+1),\ \varepsilon\}$

condition : $u_i = v_j$

- La transition M'_1 est déclenchée lorsque les symboles source et cible coïncident entre eux et sont des parenthèses ouvrantes. Cette transition est activée lors de la reconnaissance du constructeur d'un type, avant d'inspecter la descendance. On passe alors dans l'état $(i+1, j+1)$ en empilant le symbole reconnu. On progresse sur u et v.

  $M'_1 = \{(i, j),\ v_j,\ (y, s, k),\ (i+1, j+1),\ (y, s, k)(v_j,\ (i,\ j+1),\ 0)\}$

  conditions : $u_i = v_j, y \neq v_j^{-1}$

- La transition M_2 est activée lorsque les symboles source et cible coïncident entre eux et sont des parenthèses fermantes. Le compteur de pile n'étant pas nul, la parenthèse ouvrante correspondante dans la cible n'a pas été reconnue (transition M'_1). On passe dans l'état $(i, j+1)$ en décrémentant le compteur de pile. On progresse sur v.

  $M_2=\{(i, j),\ v_j,\ (v_j^{-1}, s, k),\ (i, j+1),\ (v_j^{-1}, s, k-1)\}$

  condition : $k \neq 0$

- Les transitions M_3 et M'_3 sont activées lors que le symbole cible est l'inverse du symbole de sommet de pile, les symboles source et cible étant différents. Ces transitions sont déclenchées lorsque l'algorithme n'a pas pu trouver de massif dans une sous−arborescence d'un type.

  La transition M'_3 mène dans l'état défini comme l'état suivant (par incrémentation de l'indice dans l'empreinte source) l'état empilé. L'indice cible de l'état présent en sommet de pile est incrémenté. On progresse sur u par rapport à l'état dans lequel on a engagé la comparaison (état empilé). Cet état différencie notre automate de celui proposé par Frilley. Nous verrons dans la démonstration du théorème 1 comment cet état permet de reconnaître le massif que l'automate de Frilley ne reconnaît pas (cf. section V.1).

$M'_3=\{(i,j),\ v_j,\ (v_j^{-1},\ (i',\ j'),\ 0),\ (i'+1,\ j'),\ (v_j^{-1},\ (i',\ j'+1),\ 0)\}$

condition : $u_i \neq v_j$

Le transition M_3 est activée lorsque les transitions M'_3 successives n'ont pas pu mener à une solution (l'indice j est égal à l'indice de l'état de retour empilé). Cette transition mène à l'état suivant (en j) l'état empilé. Si (i,j) est l'état courant et (i',j') l'état de retour, le nouvel état est $(i, \alpha^{-1}(j'+1)+1)$. Le sommet de pile est dépilé. On progresse sur v par rapport à l'état dans lequel on a engagé la comparaison (état empilé).

$M_3=\{(i,j),\ v_j,\ (v_j^{-1},\ (i',\ j'),\ 0),\ (i',\ \alpha^{-1}(j'+1)+1),\ \varepsilon\ \}$

- La transition M_4 est activée lorsque les symboles source et cible ne coïncident pas et que le symbole cible est le même que le symbole stocké sur le sommet de la pile (ouvrant). Le compteur de pile est alors incrémenté et on passe à l'état $(i, j+1)$. On progresse sur v.

  $M_4=\{(i,j),\ v_j,\ (v_j,\ s,\ k),\ (i,j+1),\ (v_j,\ s,\ k+1)\}$

  condition : $u_i \neq v_j$

- La transition M_5 est activée dans les autre cas, le symbole source ne correspond ni au symbole cible, ni au sommet de pile, ni à son inverse. On progresse sur v.

  $M_5=\{(i,j),\ v_j,\ (y,\ s,\ k),\ (i,j+1),\ (y,\ s,\ k)\}$

  condition : $u_i \neq v_j, y \neq v_j^{-1}, y \neq v_j$

V.2.2.3 Un exemple de comparaison

Pour illustrer les transitions de l'automate prenons l'exemple de la comparaison de l'empreinte du type *exercice* avec celle du type *paragraphe* donnée dans la figure 40.

DTD :

```
<!ELEMENT paragraphe
          (simple|cite|groupe|liste|titré)>
<!ELEMENT simple            (#PCDATA)      >
<!ELEMENT cite              (#PCDATA)      >
<!ELEMENT groupe            (paragraphe)+  >
<!ELEMENT liste             (item)+        >
<!ELEMENT item              (paragraphe)+  >
<!ELEMENT titré             (titre, corps)>
<!ELEMENT titre             (#PCDATA)      >
<!ELEMENT corps             (paragraph)+   >
```

Empreinte :

```
[TT(@)((@)){T(@)}]
```

Figure 40 : DTD et empreinte du type Paragraphe

L'empreinte du type *paragraphe* contient des symboles '@' marquant les définitions récursives de types. Dans un premier temps nous considérerons des développements récursifs nécessaires à la correspondance, la section V.4.4 présente une méthode pour traiter les types récursifs. En développant les récursions nécessaires, l'empreinte du type *paragraphe* devient :

```
[TT(@)((@)){T([TT([TT(@)((@)){T(@)}])(([TT(@)((@)){T(@)}])){T(@)}])}]
```

Le déroulement pas à pas de cet exemple est présenté sous la forme des 2 chaînes placées l'une sous l'autre de telle sorte que les symboles reconnus de la source soient alignés verticalement avec le symbole correspondant de la cible. Les symboles * représentent les indices de progression dans chaque empreinte. À chaque étape nous donnons également l'état de la pile.

La comparaison commence sur l'indice 1 des deux empreintes avec une pile vide :

```
*
{T([TT])([TT])}
[TT(@)((@)){T([TT([TT(@)((@)){T(@)}])(([TT(@)((@)){T(@)}])){T(@)}])}]
*
Pile : ε
```

Une suite de transitions M_5 mène à l'état (1,12).

```
          *
           {T([TT])([TT])}
[TT(@)((@)){T([TT([TT(@)((@)){T(@)}])(([TT(@)((@)){T(@)}])){T(@)}])}]
          *
Pile : ε
```

Ensuite, la suite de transitions M'_1, M_t, M'_1, M'_1, M_t, M_t conduit à l'état (7, 18) en empilant les parenthèses ouvrantes rencontrées avec les états correspondant :

```
                *
           {T([TT])([TT])}
[TT(@)((@)){T([TT([TT(@)((@)){T(@)}])(([TT(@)((@)){T(@)}])){T(@)}])}]
                *
Pile : ('{',2,13,0) ('(',4,15,0) ('[',5,16,0)
```

Une suite de transitions M_5 et une transition M_4 (état 7, 19) mènent à l'état (7,35), le compteur du sommet de pile n'étant pas nul, il est décrémenté (transition M_2).

```
                                        *
            {T([TT                      ])([TT])}
[TT(@)((@)){T([TT([TT(@)((@)){T(@)}])(([TT(@)((@)){T(@)}])){T(@)}])}]
                                        *
Pile : ('{',2,13,0) ('(',4,15,0) ('[',5,16,1)
```

Une suite de transitions M_5, M_4 et M_2 mènent à l'état (7, 66).

```
                                                                    *
...{T([TT                                                  ])([TT])}
...{T([TT([TT(@)((@)){T(@)}])(([TT(@)((@)){T(@)}])){T(@)}])}]
                                                                    *
Pile : ('{',2,13,0) ('(',4,15,0) ('[',5,16,0)
```

Dans l'état (7, 68) le symbole cible est identique à l'inverse du symbole de sommet de pile et le compteur de pile est nul, la concordance démarrée à l'état (1, 12) est complète. Mais la transition M_3 déclenchée dans l'état suivant provoque un retour à l'état (2, 13)

```
                                                                      *
...{T([TT                                                  ])([TT])}
...{T([TT([TT(@)((@)){T(@)}])(([TT(@)((@)){T(@)}])){T(@)}])}]
                                                                      *
Pile : ('{',2,13,0)
```

L'état de retour est alors changé en (2, 14) et la comparaison se poursuit.

```
              *
            { T(     [TT])([TT])}
[TT(@)((@)){T([TT([TT(@)((@)){T(@)}])(([TT(@)((@)){T(@)}])){T(@)}])}]
              *
Pile : ('{',2,14,0)
```

Après avoir reconnu une autre parenthèse et un autre crochet, la transition M_1 valide la reconnaissance du type choix.

```
                                        *
            {   T ([TT                  ])([TT])}
[TT(@)((@)){T([TT([TT(@)((@)){T(@)}])(([TT(@)((@)){T(@)}])){T(@)}])}]
                                        *
Pile : ('{',2,14,0) ('(',4,20,0) ('[',5,21,0)
```

L'algorithme se poursuit jusqu'a ce que l'intégralité de l'empreinte source ait été parcourue (succès de la reconnaissance) ou que la pile soit vide alors que l'instance source n'a pas été entièrement parcourue (échec de la reconnaissance).

```
                                                                      *
            {   T ([TT                  ])( [TT                 ] )       }
[TT(@)((@)){T([TT([TT(@)((@)){T(@)}])(([TT(@)((@)){T(@)}])){T(@)}])}]
                                                                      *
```

Ici la comparaison s'est terminée avec succès et une relation de massif à été trouvée entre les structures des deux types d'éléments. La figure 40 montre la mise en correspondance entre les nœuds de l'arbre de types source et l'arbre de

types cible. Les nœuds de l'arbre cible présentés en rouge sont les nœuds image de la relation de massif.

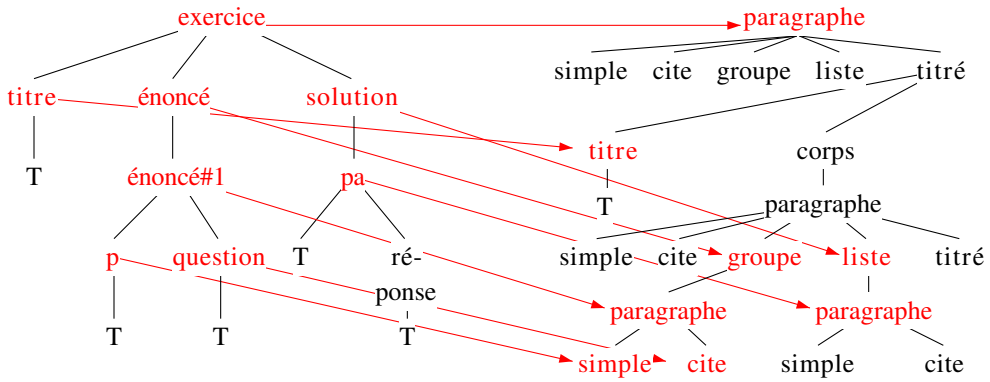


Figure 41 : Types mis en correpondance par la transformation automatique

V.2.2.4 Validité de l'automate

Nous montrons maintenant la consistance et la complétude de l'automate de comparaison d'empreintes de types non récursifs.

Théorème 1

> Désignons par $L(u, v)$ le langage reconnu par l'automate à pile précédemment défini, Alors $L(u, v) \neq \emptyset$ si et seulement si U $\subseteq$ V.

Preuve

> Nous allons faire la démonstration par récurrence sur le cardinal $|v|$ à u fixé.
>
> Remarquons qu'à cause des mouvements de M_5, $L(u,v)$ est non vide si et seulement s'il contient le mot v lui–même.
>
> Remarquons également qu'en tout état (i, j) atteignable de l'automate, nécessairement $i \leq j$. Ceci entraîne d'une part que les mouvements de M_3 mènent toujours en état (h, k) tel que $h < i$ (l'état précédent étant (i, j)) et d'autre part que si $|v| < |u|$, il vient $|v| < |u| - 1$ et par suite aucun état terminal ne peut être atteint (puisque l'automate se bloque sur un état (i, j) et que, dans ce cas, $i \leq j \leq |v| + 1 \leq |u| + 1$).
>
> Nous supposerons donc que $|v| \geq |u|$, ce qui permet de commencer la récurrence en $|u| = |v|$.

♦ Si $|u| = |v|$, Supposons d'abord que $U \subseteq V$. Dans ce cas $U \equiv V$, ce qui entraîne la succession de mouvements suivants (qui sont tous dans $M_1 \cup M_{1'} \cup M_t$) :

$(v_i\, R_{|v|-i+1}(v),\ (i, i),\ (y\ (c, s, 0))) \rightarrow (R_{|v|-i+1}(i+1, i+1),\ (y\ \Theta\ ((c, s, 0)\ (v_i, (i, j),\ 0)))$

Les empreintes étant des chaînes parenthésées et par la relation d'absorption Θ définie par les transitions M_1, $M_{1'}$ et M_t, l'automate arrive donc après $|u|$ mouvements, dans la configuration $(\varepsilon, (|u|+1, |u|+1), \varepsilon)$.

Il s'agit d'un état terminal et le mot v a été reconnu donc $L(u,v) \neq \emptyset$.

Réciproquement, si l'automate reconnaît au moins un mot w, alors il reconnaît le mot v et il est parvenu en un état terminal, celui–ci ne pouvant être que de la forme : $(|u|+1, |u|+1)$ puisque $|u| = |v|$. Nous en déduisons aussi que $|v| = |u|$.

D'après le fonctionnement de l'automate ceci ne peut se produire que si l'on a exécuté une suite de mouvements élémentaires de la forme :

$(v_i\, R_{|v|-i+1}(v),\ (i, i),\ (y\ (c, s, 0))) \rightarrow (R_{|v|-i+1}(v),\ (i+1, i+1),\ (y\ \Theta\ ((c, s, 0)(v_i, (i, j),\ 0))$

$R_n(v)$ étant le suffixe d'ordre n de l'empreinte v.

Cette suite n'est réalisable que si, pour tout indice i, $u_i = v_j$. Ce qui permet de conclure que $u = v = w$ et par suite que $U \equiv V$ (seuls les mouvements élémentaires de $M_1 \cup M_{1'} \cup M_t$ font avancer simultanément i et j) et donc $U \subseteq V$.

♦ Supposons le résultat établi à un rang $|v| = k \geq |u|$ et montrons–le au rang supérieur :

Remarquons d'abord que si $L(u, v) \neq \emptyset$, alors l'automate a réussi à atteindre un état terminal par une suite d'états $q_1 = (i_1, j_1),...,q_k = (i_k, j_k)$ en reconnaissant le mot v qui est aussi dans le langage reconnu.

On construit alors l'ensemble E :

$\mathbf{E} = \{q_h = (i_h, j_h) \mid i_{h+1} = i_h + 1 \text{ et } \forall\ h' \neq h,\ i_{h'} = i_h \Rightarrow j_{h'} < j_h\}$

Supposons qu'il existe deux couples (i_h, j_h) et $(i_{h'}, j_{h'})$ dans E tels que $i_h = i_{h'}$ et $j_h \neq j_{h'}$. Alors, par définition de E $j_{h'} < j_h$ et par croissance des indices $j_h < j_{h'}$ ce qui est absurde. D'autre part, pour tout indice $i \in [1, |u|]$, il est clair qu'il existe un indice j tel que $(i, j) \in \mathbf{E}$ puisque l'on atteint un état terminal.

Donc **E** définit une application ϕ de $\{1,..., |u|\}$ dans $\{1,..., |v|\}$ telle que $v_{\phi(i)} = u_i$ pour tout i (en effet le mouvement effectué en l'état $(i, \phi(i))$ ne peut conduire qu'en l'état $(i+1, \phi(i)+1)$, par définition de **E**, ce qui implique que ce mouvement est dans $M_1 \cup M_{1'} \cup M_t$).

Remarquons que la suite $j_1,..., j_k$ est strictement croissante par construction de l'automate, en effet le retour sur j occasionné par la transition $M_{3'}$ implique qu'aucun élément de **E** ne comprenne un indice j qui soit compris entre l'indice de départ et l'indice résultant de la transition. Donc la sous–suite des indices j_h pour lesquels il existe i_h tel que $(i_h, j_h) \in \mathbf{E}$ est elle–même croissante (strictement).

Supposons alors qu'il existe dans **E** deux couples (i_h, j_h) et $(i_{h'}, j_{h'})$ tels que $i_h > i_{h'}$ et h < h'. Alors comme l'automate parvient en état terminal (l'état q_k) il existe nécessairement un indice $l >$ h' tel que $i_h = i_l$. Comme $l >$ h', $l >$ h et donc nécessairement $j_l > j_h$, la suite $(j_1, j_2,..., j_k)$ étant croissante. Mais ceci est contradictoire avec le fait que $(i_h, j_h) \in \mathbf{E}$. Donc la sous–suite des indices i_h pour lesquels il existe j_h tel que $(i_h, j_h) \in \mathbf{E}$ est croissante. Ce qui nous permet de conclure que l'application ϕ est strictement croissante (ce qui implique d'ailleurs qu'elle est bijective sur son image).

Posons $m = \overline{\varphi}_V^{-1} \circ \phi \circ \overline{\varphi}_U$. Par construction de ϕ, $u_i = v_{\phi(i)}$ pour tout i, donc, par définition des applications φ_U et φ_V, $e'(m(x)) = e(x)$ pour tout $x \in U$.

Soit $\psi = \varphi_V \circ \overline{\varphi}_V^{-1} \circ \phi \circ \overline{\varphi}_U$ et $\mu = \overline{\varphi}_V^{-1} \circ \psi$. Par construction $m = \mu$.

Analysons maintenant les mouvements de M_1, M_2 et M_4 :

Si $(i_h, j_h) \in E$, avec $u_{ih} \in \overline{C}$, ce couple est associé à un autre couple $(i_{h'}, j_{h'})$ $\in \mathbf{E}$ dans un mouvement de M_1 (l'appartenance de ce couple à **E** résulte de celle de (i_h, j_h)). Par construction de l'automate, il vient : $i_{h'} = \alpha^{-1}(i_h)$. Si le compteur de pile est nul, on a aussi exactement $j_{h'} = \alpha^{-1}(j_h)$. Si le compteur n'est pas nul, cette dernière égalité n'est pas vérifié et on a seulement : $j_{h'} \leq \alpha^{-1}(j_h)$. Mais, si $q_l = (i_l, j_l)$ désigne l'état qui suit exactement $q_{h'}$ dans la liste d'états de E, il vient aussi : $j_{h'} \leq \alpha^{-1}(j_h) \leq j_l$ (sinon il y aurait eu empilement et non incrémentation du compteur).

Or, par définition de ϕ et de ψ, $j_{h'} = \phi(i_{h'})$ et

$\alpha^{-1}(j_h) = \varphi_V \circ \overline{\varphi}_V^{-1}(j_h) = \psi\ (i_h)$

Donc comme ϕ est croissante, ψ l'est aussi. Et, par suite μ est aussi croissante,

Si x et y sont deux nœuds de l'arbre U comparables pour l'ordre partiel de cet arbre, alors (avec $x \leq y$) :

$\varphi_U(x) \leq \varphi_U(y) \leq \overline{\varphi}_U(y) \leq \overline{\varphi}_U(x)$

ce qui implique, par croissance de l'application ψ, que :

$\psi(\varphi_U(x)) \leq \psi(\varphi_U(y)) \leq \psi(\overline{\varphi}_U(y)) \leq \psi(\overline{\varphi}_U(x))$

que l'on peut écrire sous la forme

$\varphi_V(\mu(x)) \leq \varphi_V(\mu(y)) \leq \overline{\varphi}_V(m(y)) \leq \overline{\varphi}_V(m(x))$

et, comme $\mu = m$:

$\varphi_V(\mu(x)) \leq \varphi_V(\mu(y)) \leq \overline{\varphi}_V(\mu(y)) \leq \overline{\varphi}_V(\mu(x))$

Donc $\mu(x)$ et $\mu(y)$ sont aussi comparables dans V et se comparent dans le même ordre.

On vérifie de même que si x et y sont incomparables dans U, alors ils restent incomparables dans V, ceci résultant de l'inégalité :

$\varphi_U(x) \leq \overline{\varphi}_U(y) \leq \varphi_U(y) \leq \overline{\varphi}_U(x)$

Donc μ est bien un isomorphisme de U sur $\mu(U)$ et donc $U \subseteq V$ •

Réciproquement montrons que si $U \subseteq V$ alors langage L (u, v) n'est pas vide.

Il existe au moins un nœud $x_0 \in V$ tel que $U \subseteq V - \{x_0\}$. On choisit alors un tel nœud maximisant la fonction $\varphi(x_0)$ et l'on considère l'automate construit sur les deu arbres U et V' = V − $\{x_0\}$. La maimisation de la fonction $\varphi(x_0)$ peut conduire à le détermination de x_0 comme racine de l'arbre V. C'est précisément le cas dans le contre exemple de l'automate de Frilley donné en V.1. Nous expliquons dans la suite de cette démonstration comment la transition M'_3 introduite permet de traiter le cas où le $_0$ déterminé par maximisation est égal à la racine de V.

Nous considérons donc dans un premier temps que l'ensemble V' est bien un massif de V puisque x_0 est différent de la racine.

Comme $U \subseteq V'$, le langage reconnu par cet automate contient au moins Φ (V'), celui–ci étant reconnu par une suite d'états $q^0_h = (i^0_h, j^0_h)$,

$1 \leq h \leq k$. Nous considérons alors, dans l'automate construit sur U et V, la suite d'états $q_1 = (i_1, j_1)$, ... , $q_k = (i_k, j_k)$ définie par :

- Pour tout indice h, $1 \leq h \leq k$, $i_h = i'_h$.
- Pour tout indice h tel que : $1 \leq j'_h < \varphi(x_0)$, $j_h = j'_h$.
- Pour tout indice h tel que : $\varphi(x_0) \leq j'_h < \overline{\varphi}(x_0) - 1$, $j_h = j'_h + 1$.
- Pour tout indice h tel que : $\overline{\varphi}(x_0) - 1 \leq j'_h \leq | \Phi(V') | + 1$, $j_h = j'_h + 2$.

Nous ajoutons les deux états q et q' définis par : si h est l'indice tel que $j'_h = \varphi(x_0)$ alors $q = (i_h, j'_{h'})$ et de même, si h' est l'indice tel que $j'_h = \overline{\varphi}(x_0) - 1$ alors $q = (i_h, j'_{h'} + 1)$.

Il ne reste alors plus qu'à trier cette suite d'états sur l'ordre des j_h.

Remarquons maintenant que, jusqu'en l'état q, tous les mouvements sont possibles car ils se produisent dans l'automate reconnaissant L (U, V'). Désignons par (i, j) l'état qui précède immédiatement l'état q dans notre liste. Le mouvement exécuté à partir de cet état ne peut pas être dans M_3. S'il était dans $M_1 \cup M'_1 \cup M_t$, cela impliquerait que x_0 est « indispensable » pour la reconnaissance de U dans V', i.e. il n'existe pas d'autre nœud x'_0 pouvant jouer le rôle avant x_0 lui–même au sens de l'ordre préfixe.

Mais comme on a choisi x_0 en maximisant la valeur de $\varphi(x_0)$, cela signifierait exactement que U $\not\subset$ V', ce qui est contradictoire avec l'hypothèse ou encore que x_0 est racine de l'arbre V.

Dans ce dernier cas, la transition M'_3 joue le rôle de « maximisation » de la fonction $\varphi(x_0)$. Lorsque l'automate ne permet pas de trouver une solution pour un sous–arbre ($v_j \in \overline{S}$, le compteur du sommet de pile état nul et $i \neq \overline{\varphi}(q_t)$, q_t étant l'état placé en sommet de pile) la transition M'_3 simule le fait que $\varphi^{-1}(v_j)$ soit racine de l'arbre en provoquant la comparaison de u_i avec chacun de ses fils.

On parvient donc bien en l'état q et ce, par un mouvement de $M_4 \cup M_5 \cup M'_3$ qui ne change pas le caractère de dessus de pile. Tout au plus, ce mouvement incrémente le compteur associé à ce caractère.

Par rapport à l'automate reconnaissant L(U, V') seule la valeur associée au caractère de sommet de pile est donc susceptible d'avoir changé. Mais ceci n'aucune influence sur les mouvements suivants car V est un arbre : le « compteur » sera nécessairement décrémenté en q'. On vérifie alors sans difficulté que l'ensemble des états que nous avons construits sont effectivement atteints. Par suite L (U, V) ≠ ∅, ce qui assure le résultat. ♦

V.2.2.5 Algorithme de comparaison

L'automate de reconnaissance de massif est implémenté par l'algorithme donné dans la figure 42.

Cet algorithme prend en entrée deux chaînes bien parenthésées appelées respectivement *u* pour la chaîne représentant le type de l'élément source et *v* pour la chaîne représentant le type dans lequel l'élément doit être converti. L'algorithme utilise une pile pour mémoriser les parenthèses ouvrantes rencontrées au cours du déroulement de l'algorithme. Chaque élément de la pile contient trois informations : *pile.symb* est le symbole représentant le constructeur du type de l'élément comparé ('{', '[' ou '(') ; *pile.état* est l'état de retour sur erreur correspondant à l'état courant quand le type à été empilé ; *pile.compte* est le nombre de parenthèses identiques à pile.symb rencontrées depuis que le type a été empilé.

L'algorithme utilise également les fonctions `Inverse`, `AjoutCouple`, `SupprCouple` et `Alpha`. `Inverse` renvoie la parenthèse fermante opposée à celle qui lui est passée en paramètre (c'est à dire ')' pour '(', ']' pour '[' et '}' pour '{'). La fonction `AjoutCouple(i,j)` enregistre une relation entre le type d'indice `i` dans la source et le type d'indice `j` dans la cible. La fonction `SupprCouple(i1, i2)` efface les relations concernant l'ensemble des types dont les indices dans l'empreinte source sont compris entre `i1` et `i2`. La fonction `Alpha` implémente α et α^{-1}. Elle retourne l'indice de la parenthèse opposée au caractère dont l'indice est passé en paramètre.

Le résultat est un ensemble de couples (type source, type cible), appelé *couplage* de *u* dans *v*, qui sont utilisés pour la conversion de l'instance source *u*.

```
i := 1;              /* indice source */
j := 1;              /* indice cible */
empiler (pile, ε); /* initialisation de la pile */

tantque i <= |u| et j <= |v| faire   /*boucle principale*/
  si u[i] = v[j] alors
    si v[j] = 'T' alors

      /* cas 1 transition Mt : deux terminaux égaux */
```

```
    AjoutCouple (i,j);
    i := i + 1;

  sinon /* v[j] != 'T'*/
    si v[j] = Inverse (pile.symb) alors
     /* v[j] est une parenthese fermante inverse
        du symbole en sommet de pile */
      si pile.compte = 0 alors

        /* cas 2, transition M1 :
           compte = 0 signifie que v(j) est la
           parenthèse fermante associée a celle du sommet
           de pile :
           le type u[i] est en relation avec v[j]
           transition M1 */
        AjoutCouple (i, j);
        i := i + 1;
        dépiler (pile);

      sinon /* pile.compte != 0 */

        /* cas 3, transition M2 : v[j] n'est pas au même
           niveau de parenthésage que le sommet de pile */
        pile.compte := pile.compte - 1;

      finsi
    sinon /* v[j] != inverse (pile.symb) */
      /* v[j] n'est pas l'inverse du somment de pile */
      si v[j] ∈ ('{','[','(') alors

        /* cas 4, transition M1' : v[j] est une parenthèse
 ouvrante,
           on empile le type correspondant */
        empiler (pile, (v[j],(i, j),0));
        i := i + 1;

      finsi
      /* l'autre cas ne peut arriver car les chaînes
         sont bien parenthésées */
    finsi
  finsi
  i := i + 1;
sinon   /* u[i] != v[j] */
  si v[j] = Inverse (pile.symb) alors
     /* v[j] est une parenthese fermante inverse
        du symbole en sommet de pile */
    si pile.compte > 0 alors

      /* cas 5, transition M2:
         v[j] correspond à un autre niveau de
         parenthésage que le type en sommet de pile */
      pile.compte := pile.compte - 1;

    sinon /* pile.compte = 0 */

      /* cas 6 :
         le compteur ne peut plus être décrémenté :
         la correspondance de types echoue partiellement.
         les couplages de tous les types de l'instance
         source compris entre pile.ind et i sont effacées
 */
      si (pile.état.j < j - 1) alors
        /* transition M3 */
         SupprCouple (pile.état.i + 1, i);
```

```
            i := pile.état.i + 1;
              /* j est incrémenté en fin de boucle */
            j := pile.état.j - 1;
            pile.état.j := pile.état.j + 1;
        sinon /* transition M3' */
            SupprCouple (pile.état.i, i);
            j := Alpha ( pile.état.j + 1);
            i := pile.état.i;
            dépiler (pile);
        finsi
      finsi
    finsi
    si v[j] = pile.symb et u[i] ∉ ('}',']',')')
      alors

       /* cas 7, transition M4: v[j] est une parenthèse
    identique à la
          derniere empilée */
       pile.compte := pile.compte + 1;

    finsi

     /* cas 8,  transition M5:
        v[j] n'est ni identique ni inverse au symbole
        de sommet de pile, on ne fait qu'avancer dans la
        chaîne cible */

  finsi
  j := j + 1; /* progression dans la chaîne cible */
fintantque

si i > |u| alors
  retourner succes;
sinon
  retourner echec;
finsi
```

Figure 42 : Algorithme de comparaison d'empreintes

Théorème 2

La complexité de l'algorithme est polynômiale en $O(|v|^2)$.

Preuve

Si l'on ne considère que les états $M_t \cup M_1 \cup M_1, \cup M_2 \cup M_3 \cup M_4 \cup M_5$, l'indice j est systématiquement incrémenté par ces état, ils peuvent donc engendrer au plus $|v|$ itérations de la boucle de l'algorithme.

Considérons maintenant la transition M'_3 cette transition ne peut entraîner le parcours supplémentaire que des fils du nœud de V correspondant à l'état (i, j) dans lequel est déclenchée la transition. Par conséquent, au plus $|v|$ transitions M'_3 sont déclenchées lors de la comparaison. La comparaison des

fils entraînant au plus $\overline{\varphi_j} - \varphi_j \leq |v|$ transitions. Le surcoût dû à la transition M'_3 est donc $|v|^2$. Le nombre d'itérations de la boucle est donc limité par $|v|+|v|^2$. Par conséquent la complexité de l'algorithme est $O(|v|^2)$.

Cet algorithme calcule un massif de l'arbre de types source U dans l'arbre de types cible V s'il en existe au moins un. Dans le cas de la comparaison de types de documents structurés, il peut exister un grand nombre de solutions – ceci est dû au nombre limité de constructeurs du système de types et à la forte composante textuelle des documents traités, qui implique de nombreuses correspondances possibles entre les types de base intervenant dans les structures génériques des documents.

Les relations trouvées par cet algorithme sont des relations de massif entre types non définis récursivement. Aussi, pour prendre en compte les relations d'absorption et trouver des relations entre types récursifs, qui sont fréquents dans les structures considérées (les paragraphes, les sections d'articles, les arbres sont définis récursivement), nous étendons la méthode de comparaison de types effectuée par l'algorithme de recherche de massif.

Pour optimiser le processus de transformation, nous évitons certaines comparaisons superflues. Lorsque la comparaison est effectuée entre deux types appartenant à une même structure générique, les éléments comparés peuvent être du même type. Il est, dans ce cas superflu de comparer les images de ces descendants dans les empreintes. Grâce à la table associative construite lors de la génération de l'empreinte, l'utilisation de la fonction ϕ_u^{-1} qui associe le nœud de l'arbre de type U à l'indice d'un symbole de l'empreinte appartenant à l'ensemble S est immédiate, on ajoute l'état M''_1 à l'automate de comparaison :

$M''_1=\{((i,j), v_j, (y,h,k), ((\phi_u \circ \phi_u^{-1}(i)) + 1, j + 1), (y,h,k))\}$

avec la condition $u_i = v_j, y \neq v_j - 1$

Cet état permet d'éviter la comparaison de la descendance de deux types égaux.

V.3 Création des éléments cible

S'il existe une relation de massif M entre l'arbre canonique représentant le type d'un élément et un type cible, il est possible de convertir l'élément dans le nouveau type.

La relation de massif étant une relation injective, tous les nœuds de l'arbre de types source ont une image dans l'arbre de type cible. D'autre part, l'ascendance des éléments et les constructeurs étant préservés par cette relation, si le type t est un fils du type t' alors $M(t)$ est un descendant de $M(t')$.

Par conséquent, lors d'un parcours dans l'ordre préfixe de l'élément à transformer, l'image de chaque élément rencontré peut être inséré comme un descendant de l'image du parent de l'élément original.

L'algorithme de génération de l'élément cible présenté dans la figure 44 utilise les fonctions `CoupleAvec(t)` qui retourne le type couplé avec `t`, `TypeElement(elem)` qui retourne le type canonique l'élément `elem`, `NouvelElem(t)` qui crée un élément du type `t`, `PremierFils(elem)` qui renvoie le premier fils de l'élément `elem`, `Suivant (elem)` qui revoie l'élément suivant `elem`, `InsérerFils(e1, e2)` qui insère l'élément `e1` comme fils de `e2` et `InsérerDernier(e1, e2)` qui insère l'élément `e1` comme dernier fils de `e2`.

La fonction récursive `Générer` parcourt les fils de l'élément source. Le type cible couplé avec le type de chacun des éléments source est recherché et la descendance d'éléments depuis l'élément `parentSource` jusqu'à un élément du type couplé. La fonction `Générer` est ainsi appliquée récursivement sur tous les éléments fils et leurs images.

Nous illustrons ce processus dans la figure 43 qui montre comment est réalisée la transformation d'un élément *répertoire* en *annuaire*. Le type *répertoire* est couplé avec le type *annuaire* et le type *entrée* est couplé avec le type *abonné* (en vert). Lors de la transformation de l'élément de type *entrée*, l'élément de type *annuaire* a déjà été créé. L'algorithme parcourt en remontant les nœuds de l'arbre de types *annuaire* depuis le nœud couplé avec *entrée* (*abonné*) et crée successivement les éléments correspondants, soit *abonné*, *commune* puis *départ*. Le dernier nœud créé est finalement rattaché au parent qui existait préalablement (ici *annuaire*).

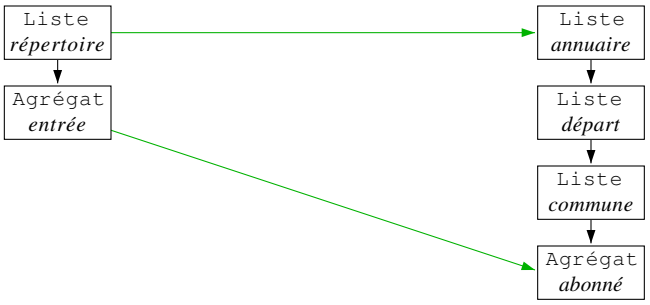


Figure 43 : Transformation d'un élément répertoire en annuaire.

```
fonction Générer (élément parentSource,
                  élément parentCible)
{
  typeCibleParent := TypeElement (parentCible);
  elemSource := PremierFils (parentSource);
  /* traite successivement chacun des fils du parent source */
  tantque (elemSource)
    typeSource := TypeElement (elemSource);
    /* recherche le noeud couplé avec le fils courant */
    typeCible = CoupleAvec (typeSource);
    type := typeCible;
    precElem := ∅;
    /* génère les ascendants jusqu'au type de parentCible */
    tantque (type != typeCibleParent) faire
       elem := NouveauElem (type);
       si (precElem) alors
         InsérerFils (precElem, elem);
       sinon
         elemCible := elem;
       finsi
       precElem := elem;
    fintantque
    /* insère la descandance comme dernier fils de parentCible
 */
    InsérerDernier (elem, parentCible);
    Générer (elemSource, elemCible);
    elemSource := Suivant (elemSource);
  fintantque
}
```

Figure 44 : Algorithme de génération des éléments cible

Cette méthode de génération s'appuie sur le fait que, par construction des arbres de types, il n'existe qu'une branche entre un type canonique et un de ses ancêtres à quelque niveau que ce soit. Cependant, à cause des constructeurs liste, un élément n'est pas associé de façon unique à un type canonique. Par conséquent, la descendance entre un élément et l'un de ses descendants peut être crée de plusieurs manières, comme le montre la figure 45.

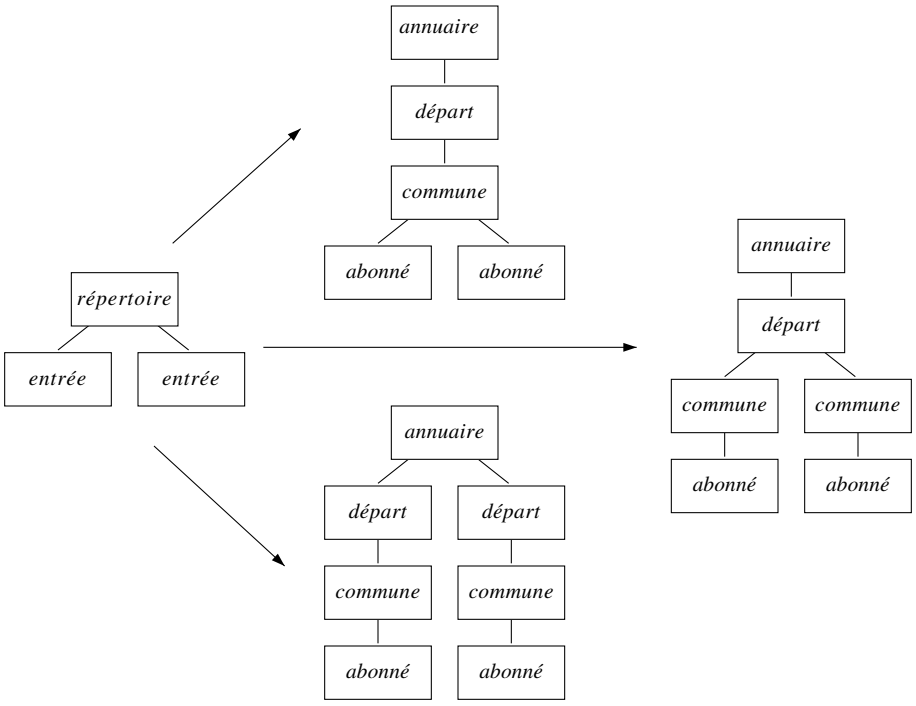


Figure 45 : Trois transformations possibles de répertoire en annuaire

Ces possibilités multiples de génération des éléments issus de la transformation démontre les limites de l'approche de la transformation automatique. Lorsque plusieurs solutions sont possibles, seule une connaissance additionnelle portant sur la sémantique de chaque type d'élément peut permettre un choix pertinent entre les différentes solutions. L'exemple ci–dessus montre également que les noms de types d'éléments ne permettent pas de lever l'ambiguïté qui apparaît lors de la comparaison des structures. Dans le chapitre suivant nous proposons d'intégrer un minimum de spécifications de l'utilisateur pour trancher entre les différentes possibilités qui se présentent lors de la transformation.

V.4 Extensions de l'algorithme de comparaison

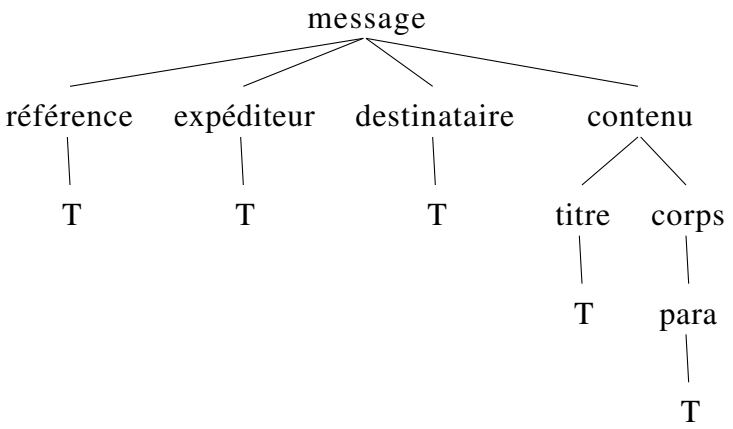
L'algorithme de comparaison présenté dans la section V.2.2 est étendu pour prendre en compte les relations d'absorption et pour pouvoir traiter les types récursifs. Les deux premières extensions utilisent des formes réduites de l'empreinte source, permettant de combiner recherche de massif et recherche d'absorption. La troisième est une extension de l'ensemble des transitions de l'automate pour la prise en compte des définitions récursives des types.

V.4.1 Forme réduite de l'empreinte

La réduction de l'empreinte qui consiste à ignorer les nœuds qui ont un constructeur identique à celui de leur père ne préserve pas l'isomorphisme d'un arbre de type ni l'empreinte qui le représente. La question est de savoir si cet isomorphisme est essentiel pour pouvoir effectuer une transformation entre le type source et le type cible avec perte de structure.

Pour satisfaire la clause 2 de la définition de la relation d'absorption énoncée en IV.3.2, les nœuds de l'arbre de type source ayant un constructeur identique à celui de leur parent ne sont pas représentés dans l'empreinte. Ainsi, l'empreinte réduite du type message (figure 46) ne représente pas le type contenu car ce type à le même constructeur que son parent.

```
<!ELEMENT message        (référence, expéditeur,
                          destinataire, contenu)  >
<!ELEMENT référence      (#PCDATA)                >
<!ELEMENT expéditeur     (#PCDATA)                >
<!ELEMENT destinataire   (#PCDATA)                >
<!ELEMENT contenu        (titre, corps)           >
<!ELEMENT titre          (#PCDATA)                >
<!ELEMENT corps          (para)+                  >
<!ELEMENT para           (#PCDATA)                >
<!ATTLIST message      date   CDATA   #REQUIRED   >
```



Empreinte générique `{TTT{T(T)}}`

Empreinte réduite `{TTTT(T)}`

Figure 46 : DTD, arbre de types et empreintes du type message

V.4.2 Comparaison avec les empreintes réduites

La recherche des relations d'absorption utilise la réduction de l'empreinte source pour la comparaison. Ce procédé consiste à construire une empreinte par linéarisation de l'arbre de types en ignorant certains nœuds susceptibles d'être absorbés.

La transformation devant préserver le plus possible la structure du document source, la relation de massif est d'abord recherchée en utilisant l'empreinte générique du type source. En cas d'échec, l'empreinte réduite est utilisée.

La comparaison utilisant la forme réduite de l'empreinte source est effectuée avec le même algorithme que la comparaison des empreintes génériques présenté en V.2.2.5.

Théorème 3

> Si l'algorithme de comparaison conclut à une relation d'équivalence entre l'empreinte réduite d'un type source t et l'empreinte d'un type cible t', alors il existe une relation d'absorption A telle que $t' = A(t)$

Preuve

> La preuve est une conséquence immédiate de l'existence d'un isomorphisme entre l'arbre de types source dans lequel les nœuds absorbables ont été supprimés (t_r) et l'arbre de types cible (t'). cet isomorphisme est la conséquence de l'équivalence calculée par l'automate.
>
> Par définition de l'empreinte réduite, t_r est équivalent à t aux nœuds absorbés près, par conséquent t' est également équivalent à t aux nœuds absorbés près et les fils des parents des nœuds absorbés dans t_r sont tous équivalents à leur image dans t'. Donc la condition 2 de la définition de la relation d'absorption est vérifiée.

V.4.3 Spécialisation de l'empreinte du type source

Une spécificité des applications permettant l'édition de documents structurés est la possibilité de manipuler des documents dont la structure n'est pas entièrement instanciée (c.f. section IV.1.1). Ainsi, il est possible de définir une relation partielle de l'arbre de types canonique vers l'arbre de structure du

document. La relation inverse, qui associe chaque élément avec un type canonique, est une application injective associant à chaque élément son type canonique.

Cette relation inverse est suffisante pour être utilisée par le processus de génération décrit précédemment (section V.3). Utilisée transitivement avec la relation définie par le couplage, elle permet d'associer un type canonique de l'arbre de types cible à chaque élément de l'instance source. Nous illustrons cette idée dans la figure 47, ou les types d'éléments instanciés sont représentés en rouge dans l'arbre de types source et leurs types associés par couplage sont également représentés en rouge dans l'arbre de types cible.

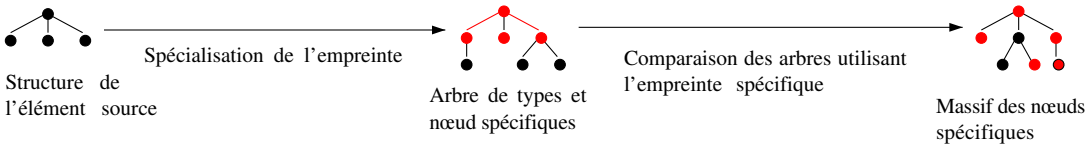


Figure 47 : Principe de comparaison utilisant l'empreinte spécifique

Par transitivité, nous définissons une application injective de l'ensemble des éléments de l'instance source dans l'arbre des types cible. La transformation peut alors avoir lieu. On constate qu'aucun couplage n'est possible en utilisant simplement les empreintes générique et réduite du type source.

La figure 48 montre une instance de document *message* et l'empreinte spécifique relative à cette instance. Les types *expéditeur* et *titre* ne sont pas instanciés dans ce document, cela peut se produit lorsque au cours de l'édition l'utilisateur n'a pas encore inséré ces éléments dans la structure. Une autre situation dans laquelle certains types de la structure générique ne sont pas instanciés est la création d'un élément dont le constructeur de type est le choix. Dans ce cas, une alternative du choix est instanciée et les types définissant les autres alternatives peuvent ne pas être représentées.

```
<message  date="30-07-1998">
  <référence>sb332</référence>
  <destinataire>
    Cecile.Roisin@inrialpes.fr
  </destinataire>
  <contenu>                                          {TT{(T)}}
   <corps>
     <para>Bonjour Cécile,</para>
   </corps>
  </contenu>
</message>
```

Figure 48 : Instance et empreinte spécifique du type message

Comme l'empreinte générique, l'empreinte spécifique peut être simplifiée pour la recherche de relations d'absorption en cas d'échec de l'algorithme de comparaison.

V.4.4 Comparaison des types récursifs

Les types d'éléments définis de manière récursive ne peuvent pas être pris en compte par l'algorithme de comparaison défini précédemment. Ces types sont très répandus, les définitions des paragraphes et des sections d'article en sont un exemple, et un système de transformation doit permettre la transformation d'éléments entre types définis récursivement.

Dans [Claves 95] il est proposé de considérer les éléments récursifs comme des types de base, ne permettant le couplage d'un tel type qu'avec un autre type récursif.

Cette proposition a l'inconvénient de ne permettre le couplage de types définis récursivement qu'avec des types également définis récursivement. Avec cette approche, dans l'exemple de la section V.2 les types *énoncé#1* et *pa* n'auraient pas pu être couplés au type *paragraphe*, qui est défini de manière récursive (voir figure 41).

Pour résoudre cette limitation, nous proposons de développer les types récursifs lors de la génération des empreintes. Le problème de cette approche est la détermination de nombre de récursions à développer. Ce nombre peut être déterminé pour le type source en se basant sur l'instanciation des types canoniques. Par contre, la profondeur du développement de l'instance cible ne peut être déterminé facilement car la relation de massif peut impliquer des nœuds de l'arbre de types cible de profondeur supérieure à la structure source.

Nous utilisons donc ces développements pour supprimer la récursion lors de la génération de l'empreinte source : les types canoniques définis récursivement sont développés tant qu'ils sont instanciés dans la structure source du document.

En ce qui concerne l'empreinte cible, il n'est pas possible de décider *a priori* quelle la profondeur maximale des développements récursifs, en effet, cette profondeur dépend du massif trouvé. Pour résoudre ce problème nous décidons de représenter une occurrence de type récursif comme un type de base et nous l'associons au caractère @ dans les empreintes. Lorsque l'automate rencontre ce caractère dans l'empreinte cible, des transitions particulières lui permettent de rechercher un massif dans le développement de la récursion.

L'automate de reconnaissance est modifié pour prendre en compte ce nouveau type de base. Nous étendons la relation α qui associe à chaque valeur de l'intervalle $[1, |v|]$ une valeur dans ce même intervalle. Soit $i \in [1, |v|]$, et $e(i) = @$, alors $\alpha(i)$ est égal à l'indice du symbole ouvrant correspondant au type semblable dans les ascendants du type récursif.

Les transitions suivantes sont ajoutées à l'automate :

- Les transitions M_6 et $M_{6'}$ permettent d'ignorer les types récursifs dans la cible, ceci pour ne pas explorer une récursion si aucun élément n'a été reconnu dans la récursion couramment explorée. Ces transitions garantissent que le nombre de récursions développées est inférieur à $|S_U|$ qui est le nombre de caractères de parenthésage ouvrant présent dans l'empreinte source. La transition $M_{6'}$ permet de ne pas développer de récursion au niveau le plus haut tant qu'aucun élément n'a été reconnu.

  $M_6 = \{ ((i, j), @, (@, s, 0), (i, j+1), (@, s, 0))\}$

  $M_{6'} = \{ ((i, j), @, \varepsilon, (i, j+1), \varepsilon)\}$

- La transition M_7 engage l'exploration d'une branche récursive dans l'empreinte cible. A l'issue de cette transition, la pile contient le symbole de récursion l'état de retour qui est celui dans lequel la transition a été activée. La troisième valeur du triplet utilisé n'est dans ce cas plus utilisée comme compteur de parenthèses, mais il contient l'indice de la fin du développement de la récursion dans l'empreinte cible qui est égal à $\alpha(\alpha(j))$.

  $M_7 = \{((i, j),@,(y, s, k),(i, \alpha^{-1}(j)),\ (y, s, k)(@,(i, j),\alpha(j))\ ) \mid\ y\ != @\ \}$

- La transition M_8 permet le retour des développement des branches récursives.

  $M_8 = \{ ((i, j), v_j, (@, (i', j'), k), (i, j'+1), \varepsilon) \mid j > k \}$

Il reste à montrer la validité de cet automate, c'est–à–dire que le langage qu'il reconnaît est non vide, ce qui implique qu'il existe une relation de massif entre le type source et le type récursif cible.

Théorème 4

Le langage L(u, v) reconnu par l'automate à pile {Q, A, Y, q_{init}, Q_{fin}, M} avec : M = $M_t \cup M_1 \cup M_1, \cup M_2 \cup M_3 \cup M'_3 \cup M_4 \cup M_5 \cup M_6 \cup M'_6 \cup M_7 \cup M_8$ est non vide si et seulement si U $\subseteq$ V.

Preuve

Nous montrons que les états $M_6 \cup M'_6 \cup M_7 \cup M_8$ conserve la validité de l'automate de comparaison en montrant que ces transitions permettent de construire un arbre V'' par développement d'un nombre limité de récursions tel qu'il existe une relation de massif entre U et V'' reconnue par l'automate de la section V.2.2.

Les transitions M_6 et $M_{6'}$ impliquent que le nombre de développements de chaque type récursif est limité à $|u|$. Si R est le nombre de symboles récursifs présents dans l'empreinte cible, la complexité de l'algorithme est donc $O(|v|^2 \times R|u|)$ soit $O(|u| \times |v|^2)$.

V.5 Application des transformations automatiques

D'un point de vue global, le processus de transformation automatique permet de trouver une correspondance entre les éléments d'un type source et les types composant un type cible. Cette comparaison se fonde sur les structures génériques et ne réclame aucune intervention de l'utilisateur.

Ces caractéristiques font que la transformation automatique de types est adaptée pour la réalisation de certaines fonctions d'édition qui nécessitent la transformation des éléments manipulés. Ces opérations peuvent ainsi être effectuées d'une façon simple nécessitant une interaction minimale avec l'utilisateur. Ainsi, dans l'éditeur Thot, la transformation automatique intervient lorsque les commandes suivantes sont effectuées :

- Coller : lorsque le type des éléments contenus dans le presse−papier ne permet pas leur insertion à l'emplacement de la sélection, ces éléments sont transformés dans un type compatible avec le contexte dans lequel ils sont collés.
- Changer : cette commande permet de changer sur place le type d'un élément de structure. La transformation est possible lorsque l'élément

sélectionné est un descendant unique d'un ancêtre dont le constructeur est un choix. Cette descendance peut contenir des éléments dont le constructeur est l'identité ou des éléments de liste unique.

La transformation se déroule en deux phases : l'élément source est d'abord confronté à l'ensemble des autres options proposées par le type choix. Les types dans lesquels cet élément peut être converti sont proposés à l'utilisateur sous forme de pop–up menu. Ainsi, seuls les types vers lesquels le système peut effectuer la transformation sont proposés à l'utilisateur. La transformation correspondant au choix de l'utilisateur est alors effectuée.

L'application du processus de transformation automatique à la conversion et la structuration de documents dans leur intégralité est plus délicate. La transformation automatique est d'autant plus pertinente que les structures des types d'éléments mises en jeu sont proches ce qui est souvent le cas pour les commandes mentionnées ci–dessus. La comparaison de structures génériques de documents entiers pose des problèmes d'un autre ordre : les structures de document au niveau global sont généralement très différentes et les relations de massif et d'absorption ne permettent pas de représenter les correspondances adaptées à la transformation.

V.6 Discussion

Le processus de transformation automatique présenté ici permet de réaliser des restructurations d'éléments de documents sans intervention de l'utilisateur ni de spécification préalable de la méthode de transformation. Contrairement aux transformations explicites, cette méthode s'appuie sur le système de types des documents structurés et non sur la structure particulière d'un élément. Cette méthode est donc intéressante pour la manipulation de documents structurés quelle que soit leur structure générique.

Ce jugement doit cependant être nuancé car la méthode transformation automatique présente un nombre non négligeable d'inconvénients et n'est pas assez complète :

- Le premier de ces inconvénients est le manque de pertinence des transformations effectuées : lorsqu'une relation de massif est trouvée entre deux structures, elle n'est pas forcément pertinente par rapport à la sémantique des structures mises en jeu. La transformation ne peut que s'appuyer sur des similarités structurales mais ignore la sémantique des

types d'éléments comme l'a montré l'exemple de la section V.3 (figure 45).

L'ambiguïté impliquée par la transformation automatique est due au mode de génération de l'instance cible et au choix de la relation d'ordre définissant le placement des fils d'un nœud dans les arbres de types préalablement à la transformation.

Une relation d'ordre sera valide pour un ensemble de transformations mais ne le sera pas pour un autre. Le résultat attendu d'une transformation dépend de la sémantique attachée à la structure et de l'utilisation que font les applications de celle–ci, plutôt que de la structure elle–même.

Le seul critère absolu qui puisse être retenu pour évaluer les transformations automatiques est la similarité entre la structure cible et le massif trouvé qui peut se mesurer par le nombre de nœuds « sautés » par cette relation. C'est pour maximiser cette similarité que nous avons choisi un ordre mettant en priorité les constructeurs des types susceptibles d'avoir une descendance importante (constructeurs liste, choix, puis agrégat).

D'autres solutions auraient été de choisir comme critère d'ordonnancement la profondeur ou le nombre de nœuds du sous–arbre de types canonique. Ces solutions auraient conduit à augmenter le nombre de types non reconnus dans les fils d'un nœud donné et, ainsi à trouver un massif plus « éparpillé » que celui retrouvé avec la relation d'ordre sur les constructeurs.

Le problème de la pertinence se pose également pour la relation d'absorption ; l'élimination de nœuds dans l'arbre de types canonique source conduit à de nombreuses combinaisons de relations de massif entre un arbre source et un arbre cible. En conséquence, la pertinence des relations trouvées souffrent de l'« aplatissement » de la structure qui introduit de nombreuses combinaisons de couplages possibles.

- La deuxième limite de la méthode automatique est de nature algorithmique. La définition récursive des types d'éléments nécessite des algorithmes de complexité élevée. L'algorithme de comparaison de structures présenté dans ce chapitre est applicable à la transformations d'ensembles d'éléments de cardinal limité (de l'ordre de la dizaine d'éléments). L'expérience montre que l'utilisation de cette méthode peut mener à des temps de comparaison élevés lorsqu'elle est appliquée à l'intégralité de documents pouvant comporter plusieurs milliers d'éléments.

Cependant cette méthode de transformation est adaptée au contexte de l'édition interactive et répond aux besoins exprimés pour les commandes d'édition nécessitant des transformations, car ces commandes portent généralement sur un nombre réduit d'éléments.

- Les attributs ne sont pas inclus dans le modèle de type, alors qu'ils pourraient aider à établir des correspondances pertinentes entre types, puisque les attributs sont porteurs d'une sémantique additionnelle à celle qui est inhérente à la structure et au type des éléments.
- Un autre problème est le manque de souplesse. Si la comparaison des empreintes échoue, la transformation ne peut simplement pas être effectuée. Nous avons proposé dans ce chapitre une solution de repli avec la comparaison des empreintes spécifiques. Toutefois cette solution de permet pas de résoudre tous les cas d'échec. Une autre solution de repli peut se fonder sur une spécification explicite de la façon dont les éléments qui empêchent la comparaison de donner une solution sont transformés. C'est une des raisons pour lesquelles nous proposons dans le chapitre suivant de combiner les approches automatique et explicite de la transformation de documents structurés.

Chapitre VI

Une combinaison des approches automatique et explicite

Après avoir présenté dans les chapitres précédents les différentes approches de transformation explicite, et après avoir proposé une méthode de transformation automatique, nous proposons maintenant un système de transformation permettant de répondre aux inconvénients et limitations des méthodes précédentes.

Le système de transformation que nous présentons dans ce chapitre combine la transformation automatique et les déclarations explicites.

VI.1 Rappel des objectifs

Dans les chapitres III et V nous avons identifié les limites des approches explicite et automatique pour la conversion de documents. Nous rappelons maintenant les trois catégories d'applications identifiées au chapitre II en synthétisant les problèmes non résolus et en montrant le besoin d'une approche qui combine les techniques explicite et automatique.

Conversion de documents non structurés

La méthode explicite nécessite une déclaration extensive de la transformation des documents non structurés. Cette solution peut être envisagée dans certaines applications très spécifiques, mais ne peut pas être adaptée au cas général car les formats de documents non structurés sont très diversifiés.

Par ailleurs, les documents source n'ayant pas de structure générique définie, la méthode de transformation automatique proposée dans le chapitre V ne peut être utilisée telle quelle pour ces transformations.

Pour résoudre ce problème, nous proposons dans la section VI.5 une application de conversion de documents qui se fonde sur la structure spécifique de l'instance pour engendrer un arbre de types spécifique au document cible. Les informations de formatage des documents doivent être préalablement traduites au format XML afin de disposer d'une syntaxe unique quel que soit le format d'origine du document. Nous disposons ainsi d'une structure spécifique au document source qui est utilisée par l'algorithme de transformation automatique.

Le résultat mène à des solutions très peu pertinentes principalement à cause du mode de génération automatique de l'arbre de types source.

Ainsi dans le cas de cette application, il est nécessaire de compléter la technique de transformation automatique par des informations supplémentaires données par l'utilisateur qui mettent en relation les éléments du document source avec les types de la structure cible.

Transformations de documents structurés au cours de l'édition

C'est la classe de transformations pour laquelle la méthode automatique donne les résultats les plus pertinents. Ces transformations portant sur un nombre réduit d'éléments, les structures génériques mises en jeu sont de taille réduite et le nombre de correspondances potentielles entre les arbres de types source et cible est plus faible. De plus, lors de transformations effectuées « sur place », les types communs aux arbres source et cible limitent l'ambiguïté induite par la comparaison.

Cependant, les transformations effectuées ne sont pas toujours satisfaisantes, c'est en particulier le cas lorsqu'un élément « peu structuré » doit être transformé dans un type dont la structure est plus profonde, par exemple lors de la transformation d'une liste d'éléments (qui ne comporte qu'un niveau d'éléments) en tableau (qui est une structure à deux niveaux : lignes et colonnes). Il peut alors exister plusieurs relations de massif entre les structures source et cible.

Il serait souhaitable de pouvoir spécifier par un ensemble de règles quelle relation sera privilégiée lors de la comparaison des structures.

Conversions de documents entre structures génériques

La méthode de transformation automatique est peu performante pour la transformation de l'intégralité d'un document : la taille de l'arbre de types représentant la structure du document source peut s'avérer importante. Ce facteur fait croître le nombre de relations de massif possibles entre les structures génériques source et cible. En conséquence, la probabilité de trouver la transformation pertinente avec l'algorithme de transformation automatique est faible.

L'approche explicite semble plus adaptée à ce type de transformation, cependant elle nécessite des déclarations à la mesure de la taille des structures mises en jeu qui peuvent se révéler importantes.

Là encore, une approche mixte paraît souhaitable : utiliser l'algorithme de transformation automatique et permettre à l'utilisateur de corriger de manière interactive les imperfections de cette méthode. En mémorisant ces corrections, on peut espérer obtenir une transformation pertinente de l'ensemble des documents source appartenant à une même structure générique.

VI.2 Principe de la méthode proposée

La méthode de transformation que nous présentons dans ce chapitre est fondée sur les transformations automatiques décrites dans le chapitre précédent. Son originalité par rapport aux méthodes de transformation explicites présentées dans le chapitre III réside dans l'importance prépondérante accordée à la part automatique de la transformation. Dans le système de transformation explicite d'Amaya présenté dans la section III.4, nous avons introduit une part d'automatisme pour adapter localement les éléments dont la transformation n'est pas exprimée. Ces adaptations consistent principalement à créer ou supprimer un niveau de structure quand celui–ci est imposé ou interdit par le contexte dans lequel doit être inséré l'élément.

Le système de transformation que nous proposons ici est fondamentalement différent : la transformation n'est pas conduite par les expressions de transformations mais par l'algorithme présenté dans le chapitre précédent. Les déclarations ne font que modifier localement le comportement de l'algorithme pour diriger la comparaison sur une relation différente de la relation qu'il aurait trouvée par défaut.

Cette approche différente implique bien sûr une algorithmique adaptée qui sera présentée dans la section VI.4 mais également une méthode de

spécification particulière. Les expressions de transformation associent un ensemble d'éléments de la structure du document source avec un nœud de l'arbre de types cible. Un ensemble de telles associations est appelé pré–couplage, il est défini dans la section VI.3.

La méthode de transformation présentée dans ce chapitre est fortement dépendante de la transformation automatique. Il est donc nécessaire de proposer une méthode de spécification intégrée au processus de transformation automatique. Nous développons ce point dans la section VI.5. Nous terminons ce chapitre sur les perspectives d'utilisation de cette technique de transformation dans des applications documentaires et donnons un exemple de structuration de documents XML pour l'éditeur Thot.

VI.3 Pré–couplage

Un pré–couplage est un ensemble d'associations entre les nœuds des arbres de types source et cible utilisés pour la transformation. Le but d'un pré–couplage est de diriger l'algorithme de transformation automatique sur une relation de massif qui mette en relation les nœuds pré–couplés.

Dans cette section nous définissons précisément la notion de pré–couplage et proposons un langage permettant de spécifier des pré–couples.

VI.3.1 Définitions

Pré–couple

Une relation liant un nœud d'un arbre de types source avec un nœud d'un arbre de types cible est appelée *pré–couple*.

La relation associant un élément avec le nœud représentant son type dans l'arbre canonique n'étant pas bijective. Un pré–couple a la propriété de spécifier le type cible d'un ensemble d'éléments de l'instance source.

Un nœud x de l'arbre de types source et un nœud x' de l'arbre de type cible sont *pré–couplés* si et seulement si il existe une relation de pré–couple p entre x et x' ; $x = \theta(p)$ est alors appelé *origine* du pré–couple et $x' = \rho(p)$ est appelé *cible* du pré–couple.

Pré–couplage

Un ensemble de pré–couples $P = \{p_1, ..., p_n)$ est un *pré–couplage* si et seulement si $\forall\ (i, j) \in [1,n]\times[1,n],\ \theta(p_i) \neq \theta(p_j)$. Les nœuds origine des pré–couples composant un pré–couplage sont tous différents.

Un pré–couplage P_γ d'un arbre de types A dans un arbre de types A' définit une fonction de pré–couplage notée γ associant à chaque nœud origine de l'arbre de type source le nœud cible correspondant dans l'arbre de types cible. L'ensemble des nœuds origine d'un pré–couplage (le domaine de γ) est noté $\Theta(P_\gamma)$.

VI.3.2 Expression d'un pré–couplage

Pour permettre la ré–utilisation des pré–couples définis pour une transformation, ceux–ci doivent être mémorisés.

Pour permettre la ré–utilisation des pré–couples définis pour une transformation, ceux–ci doivent être mémorisés.Il doivent également pouvoir être exprimés quels que soient les arbres de types considérés. C'est dans ce but que nous proposons un langage d'expression de pré–couplages. Ce langage est basé sur la syntaxe XML. Cette syntaxe permet de décrire facilement les structures de documents (dans section III.3.3.7 nous avons vu comment XSL se sert de XML pour décrire des fragments de structures de documents, avec les *templates*).

Nous décrivons la syntaxe de ce langage en l'illustrant à l'aide de la transformation d'un élément exercice en paragraphe présentée dans le chapitre précédent. Le couplage calculé par l'algorithme automatique est représenté dans la figure 49, en particulier le nœud *énoncé* de l'arbre source et couplé avec un nœud *liste* et le nœud *solution* est couplé avec un nœud *groupe*.

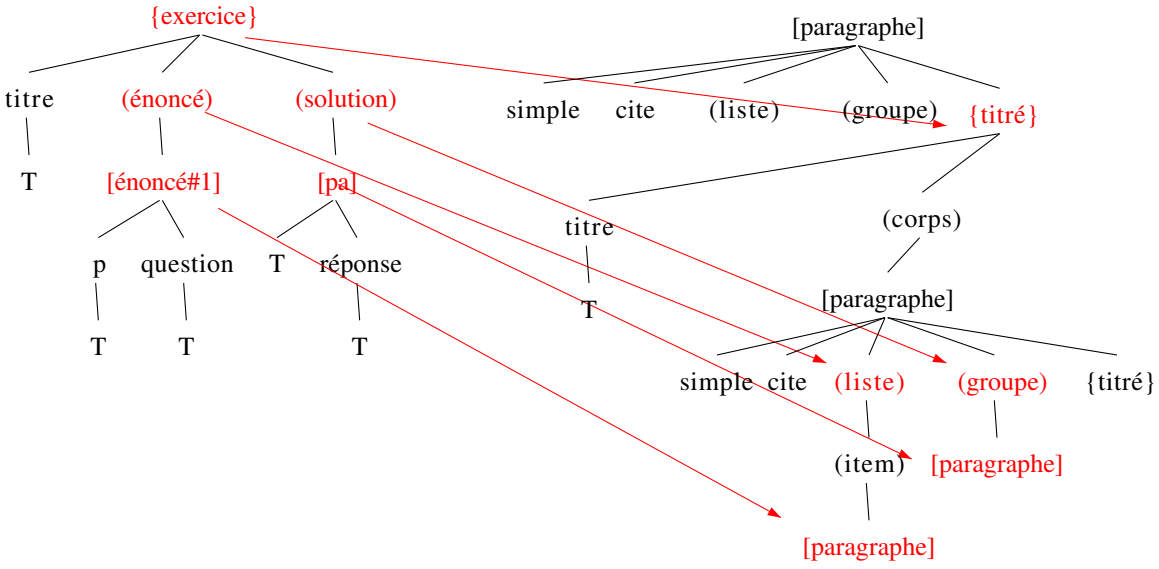


Figure 49 : Couplage calculé par l'algorithme de comparaison automatique

Supposons maintenant que l'on veuille que l'*énoncé* et la *solution* soient transformés en *groupes*, il faut alors pré–coupler le type *énoncé* et le type *solution* avec le type *groupe*. Le pré–couplage est représenté par les flèches rouges sur la figure 50.

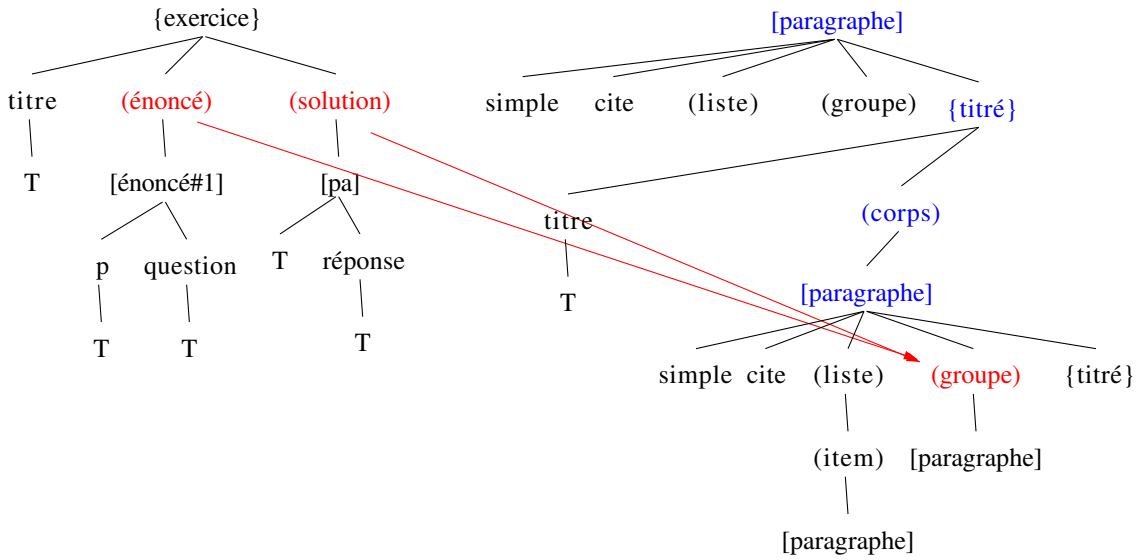


Figure 50 : Pré–couples entre les arbre de type exercice et paragraphe.

Un pré–couplage est décrit par une structure d'éléments XML *node*. Cette structure est une projection de l'arbre de types source ne conservant que les nœuds appartenant aux branches comprenant un nœud de $\Theta(P)$. Dans l'exemple cette structure comprend les nœuds exercice, énoncé et solution.

Les éléments correspondant aux nœuds source des pré–couples ont un ensemble d'éléments *destnode* parmi leurs fils qui définit le nœud cible du pré–couplage. Cet ensemble d'éléments est composé des nœuds ancêtres successifs compris entre la racine de l'arbre de types cible et le nœud cible du pré–couple. Ces nœuds sont présentés en bleu dans la figure 50.

La figure 51 présente l'expression du pré–couplage. Un élément XML *precouples* contient les éléments *node* décrivant les nœuds pré–couplés de l'arbre de type source. Cet élément porte impérativement un attribut *target–type* donnant le nom du type racine de l'arbre cible. L'élément *precouples* est composé d'un ou plusieurs éléments de type *node* correspondant aux racines des arborescences des types source. Chaque élément *node* porte impérativement un attribut *type* contenant le nom du type représenté par le nœud. S'il représente un nœud origine d'un pré–couple, un élément *node* contient une séquence d'éléments *targetnode* identifiant le nœud cible en donnant la descendance des

éléments depuis la racine de l'arbre de types cible jusqu'au nœud cible du pré–couple. Les éléments *node* peuvent également contenir d'autre éléments *node* représentant leurs fils dans l'arbre de types source, permettant ainsi de définir des pré–couplages impliquant des descendants de nœuds pré–couplés.

```
<precouples targettype="paragraphe">
<node type="exercice">
 <node type="énoncé">
  <targetnode type="paragraphe"/>
  <targetnode type="titré"/>
  <targetnode type="corps"/>
  <targetnode type="paragraphe"/>
  <targetnode type="groupe"/>
 </node>
 <node type="solution">
  <targetnode type="paragraphe"/>
  <targetnode type="titré"/>
  <targetnode type="corps"/>
  <targetnode type="paragraphe"/>
  <targetnode type="groupe"/>
 </node>
</node>
</precouples>
```

Figure 51 : Déclaration d'un pré–couplage

La syntaxe du langage présenté ici reprend l'idée de XSL qui consiste à décrire les structures de documents en utilisant le marquage. Mais à la différence de ce langage de transformation, nous décrivons ici des arbres de types et non des structures d'éléments car la localisation des types pré–couplés est relative aux arbres de types alors que les transformations XSL se fondent sur la structure spécifique des documents. De plus, nous n'associons pas à ces structures des règles de génération, mais l'identification d'un nœud dans l'arbre de types cible. Pour cela une séquence d'éléments est suffisante.

Les pré–couples pouvant être spécifiés de façon interactive, les expressions de ce langage doivent pouvoir être construites à partir des arbres de types. Contrairement aux *patterns* du langage XSL qui décrivent un sous–arbre spécifique, la syntaxe XML présentée ici permet de représenter directement une partie de l'arbre de types cible.

Notons que l'arbre de types source n'est pas représenté intégralement dans l'expression d'un pré–couplage, seules les branches contenant des nœuds origines d'un pré–couple sont nécessaires. Ainsi dans l'exemple précédent, le nœud *titre* n'est pas représenté dans l'expression du pré–couplage.

La syntaxe du langage des expressions de pré–couplage est donnée sous la forme d'une DTD XML :

```
<!ELEMENT  precouples     (node+)                      >
<!ATTRLIST precouples
           target-type     CDATA      #REQUIRED       >
<!ELEMENT  node             ((targetnode)*, (node)*)  >
<!ATTRLIST node
           type             CDATA      #REQUIRED       >
<!ELEMENT  targetnode         EMPTY                    >
<!ATTRLIST targetnode
           type             CDATA      #REQUIRED       >
```

L'exemple de la figure 51 montre que la syntaxe des spécifications de pré–couplage peut être complexe et réclamer des spécifications assez longues. Cela va à l'encontre des objectifs que nous nous sommes fixés dans le chapitre II.

Il est également difficile de prévoir quels pré–couplages seront nécessaires pour la correction d'une relation de massif retrouvée par la comparaison d'arbres de types. Le pré–couplage permettant de compléter ou de préciser la relation trouvée par cette comparaison, il est logique que la spécification d'un pré–couplage intervienne suite à une première transformation automatique. Le mode d'utilisation de la méthode proposée dans ce chapitre est donc le plus souvent un processus cyclique impliquant la transformation automatique et la spécification explicite.

Pour ces deux raisons, nous proposons de n'utiliser ce langage que pour la mémorisation des pré–couples et d'intégrer la spécification de pré–couplage à l'application de transformation. L'utilisateur ne verra donc pas ce langage, mais pourra spécifier les pré–couples à travers une interface simple.

VI.4 Comparaison de types avec pré–couplage

Nous proposons maintenant un algorithme de comparaison d'arbres de types canoniques prenant en compte un ensemble de pré–couples entre les nœuds de l'arbre source et les nœuds de l'arbre de types cible.

VI.4.1 Principe de la comparaison

L'algorithme de comparaison doit prendre en compte une relation partielle de couples nœud source – nœud cible dans un processus de comparaison des arbres de types.

Pour cela, l'algorithme reprend l'automate présenté dans le chapitre V, mais effectue un traitement particulier lorsque l'état courant contient l'indice d'un nœud pré–couplé. Ce traitement diffère selon la position du nœud cible du

pré–couplage dans l'empreinte cible par rapport à l'indice définissant l'état courant. Il est également affecté par la relation entre le constructeur du type source et le constructeur du type cible du pré–couplage.

Ces deux facteurs interviennent de manière indépendante sur l'algorithme de comparaison : l'indice du type pré–couplé met en cause les caractères précédant ceux représentant les nœuds pré–couplés dans la source et la cible ; la relation entre les constructeurs influence la façon dont est comparée la descendance des types pré–couplés. Ces descendances sont représentées par les sous–chaînes délimitées par les caractères représentant le début et la fin des types pré–couplés dans les empreintes. Nous désignerons ces deux étapes de l'opération de pré–couplage par les deux termes :

- détermination du *contexte* d'un pré–couple et
- comparaison de *types hétérogènes*. Deux types sont hétérogènes s'ils sont définis par des constructeurs différents.

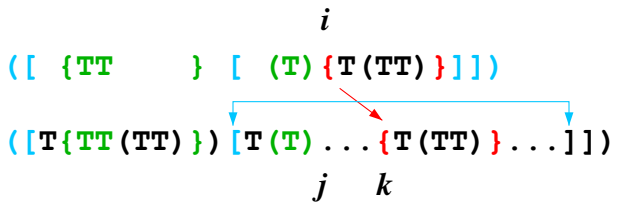
VI.4.1.1 Contexte d'un pré–couple

Le contexte d'un pré–couple p entre les nœuds u_i et v_j est calculé lorsque l'automate de comparaison traite le caractère u_i de la source qui représente le nœud origine du pré–couple. L'automate est alors dans un état (i, k).

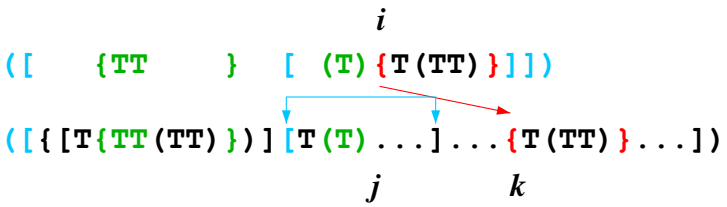
Cet état ne peut être atteint que suite à une transition M_t, M_1, M'_1 (qui implique la reconnaissance du caractère d'indice $i-1$), ou M'_1 (qui implique un retour en arrière après l'échec de la reconnaissance de la sous–chaîne contenant le caractère pré–couplé).

Examinons la comparaison effectuée par l'automate présenté dans le chapitre V relativement à l'indice du nœud pré–couplé dans l'empreinte cible. Dans la source, les caractères correspondant aux nœuds précédant le nœud pré–couplé (en vert) ont déjà été reconnus dans leur intégralité. Seule l'image par φ des nœuds ancêtres (en bleu) a été traitée. Les images du nœud pré–couplé sont présentées en rouge. Plusieurs cas de figure peuvent se produire. Nous illustrons chacun d'entre eux en montrant la comparaison des empreintes source et cible avec un pré–couple correspondant au cas présenté :

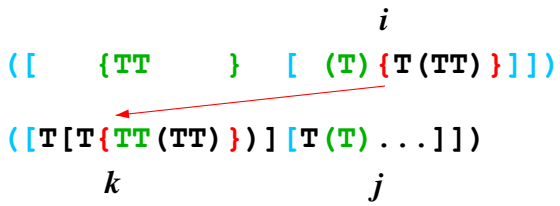
1. L'indice k du nœud cible est compris entre j et l'indice du caractère de $\overline{S}_V$ représentant le nœud couplé avec le parent du nœud origine du pré–couple.



2. L'indice k du nœud cible est supérieur à celui du caractère de $\overline{S}_V$ représentant le nœud couplé avec le parent du nœud origine du pré–couple.



3. Le nœud cible du pré–couplage a déjà été exploré par l'automate, c'est à dire qu'il existe un indice i' tel que 0<i'≤i et l'état (i', γ(i)) est dans la suite des états menant à (i, j). Ce cas est illustré par la configuration suivante :



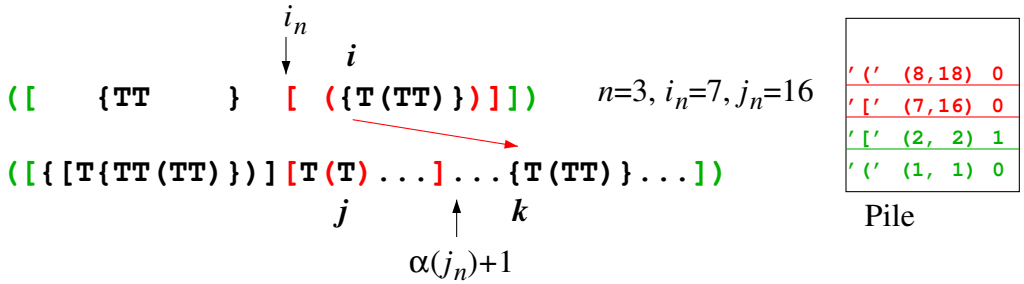
Ces trois cas de figure sont traités de la façon suivante :

1. Le premier cas est le plus simple. L'indice dans l'empreinte cible doit simplement être incrémenté jusqu'à $\gamma(i)$. Le compteur de sommet de pile doit être incrémenté chaque fois que le caractère de sommet de pile est rencontré ; il doit être décrémenté chaque fois que le caractère inverse est rencontré.

2. Dans le second cas, il n'est pas possible d'avancer l'indice de l'empreinte cible jusqu'à l'indice du nœud cible du pré–couple car cela conduirait à ne pas respecter la hiérarchie du couplage et donc ne conduirait pas à une relation de massif.

   Lorsqu'un nœud pré–couplé est rencontré dans ce contexte, supposons que la pile contienne $(v_1, (i_1, j_1), k_1)(v_2, (i_2, j_2), k_2)...(v_p, (i_p, j_p), k_p)$. Soit n le plus petit indice compris dans $[1..p]$ tel que $\alpha(j_n) < \varphi(\gamma(i))$, les nœuds correspondant aux éléments de la pile compris entre n et p compris (représentés en rouge ci–dessous) ne peuvent pas participer à un massif contenant $\gamma(i)$. En effet la bijection entre l'empreinte et l'arbre

de type implique que ces nœuds ne soient pas des ancêtres du nœud cible du couplage. La comparaison reprend donc dans l'état $(i_n,\alpha(j_n)+1)$ et les $p-n$ éléments du sommet de la pile sont éliminés.

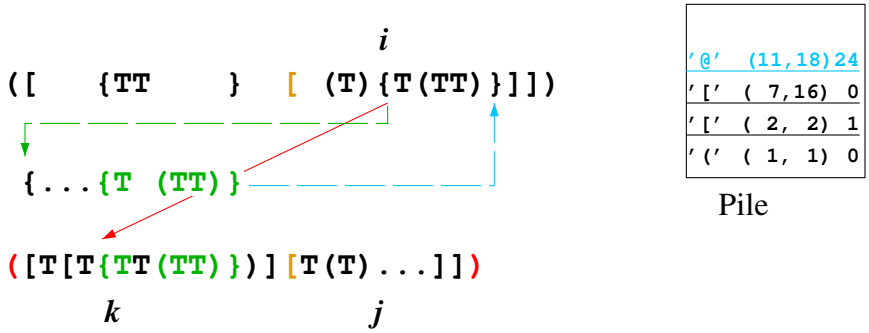


Ce saut dans l'empreinte source est similaire à celui effectué par l'automate de comparaison automatique lorsqu'un massif n'a pas pu être trouvé dans une partie de l'empreinte (transition M'_3).

3. Dans le troisième cas, si le dernier nœud couplé (en orange) de l'arbre cible et le nœud cible du pré–couple ont un ancêtre commun dont le constructeur est liste (dont l'image est représentée en rouge ci–dessous, le pré–couple étant symbolisé par la flèche rouge), la comparaison peut localement transgresser la règle de conservation de l'ordre des nœuds car une instance de liste pourra être engendrée pour chaque branche de l'arbre source.

   Pour cela, tous les nœuds ascendants du nœud origine du pré–couplage qui ne sont pas ancêtres du nœud précédemment couplé dans l'arbre source doivent être reconsidérés. La comparaison reprend donc dans un état (i', j') où i' est l'indice associé à l'ascendant le plus éloigné du nœud source. Ce retour en arrière est symbolisé par la flèche verte.

   Ainsi, dans l'exemple, il est possible de conserver le couplage de l'agrégat **{TT}** et de la liste **(T)** précédant le nœud origine du pré–couple :



L'exemple ci–dessus montre que l'état dans lequel à été rencontré le pré–couple doit être mémorisé dans la pile pour être rétabli après la

comparaison. Cette mémorisation permet la reconnaissance des nœuds ascendants du nœud origine du pré–couple ou le retour dans un état cohérent si la comparaison ne peut s'effectuer (flèche bleue). Ce retour en arrière dans l'empreinte cible est similaire au développement d'une récursion dans l'algorithme de comparaison du chapitre V.

Si la condition d'existence d'un nœud de type liste dans les ancêtres du nœud pré–couplé n'est pas remplie, alors le pré–couple fait échouer la comparaison. Suivant la nature de l'application, il faut soit éliminer ce pré–couple et continuer la comparaison, soit arrêter la transformation avec un cas d'échec.

VI.4.1.2 Comparaison de types hétérogènes

Lorsque le contexte de couplage a pu être déterminé et pris en compte dans l'algorithme, la descendance des nœuds pré–couplés doit également être comparée pour que la transformation puisse s'effectuer. Les pré–couples pouvant mettre en relation des types de constructeurs différents, la technique de comparaison diffère selon le constructeur du type source et le constructeur du type cible du pré–couple.

Dans le tableau 52, nous analysons les différents cas de figure pouvant se présenter lors de la spécification de pré–couples. Dans chaque cellule, les deux nombres représentent les arités respectives des types source et cible sur la première ligne, et des instances de ces types sur la seconde. Ainsi, les types définis par le constructeur liste ont une arité de 1 (un seul fils dans l'arbre de type), mais les instances sont des éléments dont l'arité est supérieure à 1. Inversement, les types définis par le constructeur choix ont une arité multiple (les alternatives du choix), mais leurs instances ont une arité égale à 1.

Constructeur du type cible		Constructeur du type source		
		Liste	Choix	Agrégat
Liste	générique	1, n	1, n	1, n
	spécifique	n, n	n, n	n, n
Choix	générique	n, n	n, n	n, n
	spécifique	1, n	1, n	1, n
Agrégat	générique	n, n	n, n	n, n
	spécifique	n, n	n, n	n, n

Figure 52 : Relation entre les arités des types et des instances en fonction des constructeurs des types source et cible d'un pré−couple.

Dans le cas de la comparaison de types, l'arité des instance du type source doit être inférieure ou égale à celle du type cible. Ce critère élimine la possibilité de pré−couple entre un type défini par le constructeur liste ou agrégat avec un type défini par un constructeur choix.

Un type liste ne peut être pré−couplé avec un type agrégat qu'à la condition que toutes les instances du type liste aient une cardinalité inférieure à l'arité du type cible. Le modèle de types ne permettant pas de représenter la cardinalité des listes, ces pré−couple ne peuvent être utilisés que pour les comparaisons utilisant la forme réduite de l'empreinte source (cf. section V.4.1). Si toutes les instances du type liste ont une cardinalité inférieure à l'arité du type agrégat, alors le nœud fils du type liste est comparé avec chacun des nœuds fils de l'agrégat. Si toutes ces comparaisons successives aboutissent, la transformation peut être réalisée.

Des pré−couples peuvent être définis entre un type source de constructeur choix et un type de constructeur liste. Une solution est trouvée à la condition que chaque nœud fils du nœud de l'arbre de type source soit la racine d'un massif du fils du nœud liste dans l'arbre de type cible. La comparaison avec l'empreinte source spécifique est plus adéquate car elle ne nécessite pas la reconnaissance d'un massif pour l'ensemble des alternatives d'un type choix, mais seulement de ceux qui sont effectivement instanciés.

Le dernier cas est celui de la comparaison d'un type source de constructeur choix et d'un type de cible de constructeur agrégat. Dans ce cas, chaque alternative du choix doit être un massif d'au moins un type composant l'agrégat. Comme dans le cas de la transformation en liste, l'utilisation des empreintes

spécifiques permet de retrouver des solutions supplémentaires dans le cas où des alternatives de choix non instanciées font échouer la comparaison des empreintes génériques.

VI.4.1.3 Algorithme

L'algorithme de comparaison avec pré–couplage `TraiterPreCouple` utilise la fonction `Comparer(i,j)` qui implémente l'automate de comparaison automatique sur les sous–empreintes délimitées respectivement par i, $\overline{\varphi_U}(\varphi_U{}^{-1}(i))$ et j, $\overline{\varphi_V}(\varphi_V{}^{-1}(j))$. L'algorithme prend en entrée une liste de pré–couples, accessibles par la fonction `PCouple(i)` qui retourne $\gamma(i)$ et un état (i, j) tel $\gamma(i)$ soit défini, c'est à dire que $\varphi_U{}^{-1}(i)$ est un nœud origine d'un pré–couple.

La fonction `TraiterPreCouple` est appelé par l'algorithme de comparaison automatique lorsque l'indice i de l'état courant correspond à un type pré–couplé. Dans cet algorithme, la présence d'un pré–couple ayant pour nœud origine $\varphi_U{}^{-1}(i)$ est testée à chaque itération de la boucle. La fonction `TraiterPreCouple` retourne l'état dans lequel doit reprendre la comparaison automatique après détermination du contexte ou un état d'échec de l'algorithme de comparaison qui appliquera une transition permettant de traiter l'échec partiel. Cette fonction utilise également les fonction Alpha et Inverse définies en V.2.2.5.

Cet algorithme est composé de deux parties : la détermination du contexte de pré–couple implémente les trois cas présentés dans la section VI.4.1.1, la comparaison de type hétérogènes implémente l'analyse de cas de la section VI.4.1.2.

```
Fonction TraiterPreCouple (i, j) -> état
/*
   traite un pré-couple U[i] -> V[PCouple(i)]
   (i, j) est l'état de départ.
*/
 {
  /*
     Détermination du contexte du pré-couple
  */

  /* cas 3 : le noeud cible à déjà été exploré              */
  si PCouple (i) ≤ j alors
    /*  recherche si il existe un type liste dans les      */
    /*  ancêtres du dernier noeud couplé                   */
    n := pile.etat.j;
    tantque (n > 0 et n ≠ '(')
      n := n - 1;
    fintantque
    si n ≠ 0 alors
      empiler ('@', (i,j), Alpha (i));
      j := n;
    sinon
     retourner echec;
    finsi

  /* cas 1 : la cible du pré-couple est avant le caractère */
  /*         fermant image du noeud en sommet de pile      */
  sinon si PCouple (i) < Alpha (pile.etat.j) alors
    /* avance en j jusqu'à la cible du pré-couple          */
    tant que j < PCouple (i)
      si v[j] = pile.symb alors
        pile.compte := pile.compte + 1;
      sinon si v[j] = Inverse (pile.symb) alors
        pile.compte := pile.compte - 1;
      finsi
    fintantque

  /* cas 2 : la cible du pré-couple est après le caractère */
  /*         fermant image du noeud en sommet de pile      */
  sinon
    /* dépile les noeuds ne pouvant participer au massif   */
    tantque Alpha (pile.etat.j) < PCouple (i)
      i := pile.etat.i;
      j := Alpha (pile.etat.j) + 1;
      dépiler (pile);
    fintantque
  finsi

/*
   Comparaison de types hétérogènes
*/
  si j = PCouple (i) alors

    si u[i] = v[j] alors
      /* les deux types ont le même constructeur            */
      Comparer (i, j);

    sinon si u[i] = '(' alors
```

```
  /* le constructeur du type origine est la liste       */
  si v[j] = '[' alors
    /* le constructeur du type cible est le choix       */
    retourner echec;
  sinon si v[j] = '{' alors
    /* le constructeur du type cible est l'agrégat      */
    i' := i;
    j' := j;
    succes := vrai;
    /* compare le type fils du type liste avec chaque   */
    /* fils du type agrégat                             */
    tant que sucess et j' ≠ Alpha (j)
      succes := Comparer (i', j');
      si succes alors
        i' := i;
    fintantque
  finsi

sinon si u[i] = '['
  /* le constructeur du type origine est le choix       */
  si v[j] = '(' alors
    /* le constructeur du type cible est la liste       */
    i' := i;
    j' := j;
    succes := vrai;
    /* compare chaque fils du type choix avec le type   */
    /* fils du type liste                               */
    tant que succes et i' ≠ Alpha (i)
      succes = Comparer (i', j');
      si succes alors
        j' := j;
    fintantque
  sinon si v[j] = '{'
    /* le constructeur du type cible est l'agrégat      */
    /* existe-t-il un massif de chaque alternative du   */
    /* avec un type fils du type agrégat                */
    i'  := i;
    i'' := i;
    j'  := j;
    succes := vrai;
    tant que sucess et i' ≠ Alpha (i)
      trouvé := Faux;
      tantque ¬trouvé et j' ≠ Alpha (j)
        trouvé := Comparer (i'', j');
        si ¬trouvé alors
          i'' := i';
        finsi
      fintantque
      succes := trouve;
      i' := i'';
      j' := j;
    fintantque

  sinon si u[i] = '{'
    /* le constructeur du type source est l'agrégat     */
    si v[j]='[' alors
      /* le constructeur du type cible est le choix     */
```

```
          echec
        sinon si v[j] = '(' alors
          /* le constructeur du type cible est la liste      */
          /* existe-t-il une relation de massif entre chaque */
          /* type fils de l'agrégat et le type de la liste   */
          i' := i;
          j' := j;
          succes := vrai;
          tant que sucess et i' ≠ Alpha (i)
          sucess := Comparer (i', j');
          si sucess alors
            j' := j;
          fintantque
        finsi
      finsi
    finsi
}
```

Figure 53 : fonction de transition associée à un nœud pré–couplé

L'algorithme de transformation semi–automatique intégrant la fonction de transition pour les nœuds pré–couplés construit ainsi un ensemble de couples entre l'arbre de types source et les nœuds du massif image dans l'arbre de types cible. Ce couplage peut être utilisé pour conduire la transformation des éléments source dans le type cible selon la méthode exposée dans le chapitre précédent (cf. section V.3).

VI.4.2 Pré–couplage et types récursifs

La méthode de transformation avec pré–couplage présentée dans la section VI.4.1 permet la spécification de pré–couples sur des arbres de types. Elle ne prend donc pas en compte le cas de la définition récursive du type cible. Nous proposons ici une adaptation de l'algorithme précédent pour prendre en compte ce cas de figure.

Pour prendre en compte des pré–couples de types définis récursivement il faut reconsidérer le cas 3 exposé plus tôt car le pré–couplage d'un nœud de l'arbre de types source avec un nœud déjà exploré de l'arbre de type cible ne conduit pas forcément à un échec. En effet, ce nœud est susceptible de participer à un couplage dans une branche récursive de la cible.

Ainsi, si dans un état (i, j) le nœud $\overline{\varphi}(i)$ est couplé avec un nœud $x_{j'}$ tel que $\varphi(x_{j'}) \leq j$, alors l'algorithme avance dans l'empreinte du type cible jusqu'à rencontrer un caractère de récursion @ d'indice r tel que $\alpha(r) \leq \varphi(x_{j'})$. Cette récursion est développée en empilant l'état (i, r). On peut alors continuer depuis le cas 1 exposé en VI.4.1.1.

Si la comparaison des types hétérogènes pré−couplés échoue, le sommet de pile est dépilé et la stratégie de retour de l'algorithme non récursif est appliquée.

La comparaison avec pré−couplage et type récursif n'est pas complète : en effet l'algorithme ne retrouve pas tous les couplages possibles. Ceci se produit dans le cas où il faudrait développer plusieurs niveaux de récursion pour retrouver un indice j inférieur à $\varphi(x_{j'})$. Pour prendre en compte ce cas, l'algorithme doit explorer tous les développements récursifs possibles à partir de l'indice j dans l'empreinte du type cible. Cette recherche implique une complexité exponentielle de l'algorithme de comparaison. Par conséquent, nous limitons le développement à un seul niveau de récursion conformément à la règle énoncée dans la section V.4.4 disant qu'on ne développe pas de types récursifs tant que l'on a pas progressé dans la reconnaissance de l'empreinte source.

VI.5 Applications de la transformation avec pré−couplage

Cette section présente l'intégration dans les applications documentaires de la méthode de transformation avec pré−couplage. Dans un premier temps nous décrivons le module d'import des documents XML qui ne sont pas associés à une DTD connue par Thot. Dans une deuxième partie, nous discutons plus généralement de l'application de cette méthode pour répondre aux autres besoins de transformation.

L'application présentée dans la section suivante montre que l'information contenue dans le document source et exploitée pour la transformation peut être de deux natures :

- si la DTD du document source est connue, le système de transformation construit une empreinte générique à partir de la DTD.
- si la DTD du document source n'est pas connue, la système engendre une DTD « *ad−hoc* » à partir de la structure spécifique du document source.

Dans l'exemple d'application présenté ici, nous montrons l'adaptation d'un document de type exercice dont la DTD n'est pas connue. Ce exemple de transformation se fonde donc sur une empreinte spécifique engendrée à partir de la structure du document.

VI.5.1 Importation de documents XML dans Thot

L'éditeur Thot permet de réutiliser des documents quelle que soit leur DTD d'origine, ou même des documents venant sans DTD connue. Pour réaliser cette fonction d'import, nous avons développé un module fondé sur la transformation avec pré–couplage. Cette application montre l'aptitude de cette méthode à remplir l'objectif de conversion de documents que nous nous étions fixés au début de ce travail.

Nous illustrons la description de cette application avec l'exemple de la transformation d'un document de type *exercice* en *paragraphe* défini dans la figure 39.

VI.5.1.1 Principe de l'application

Thot permet de manipuler des documents XML conformes à un ensemble de DTD connues par le système (Rapport, Paragraphe, Exposé, etc.). Lorsque l'utilisateur ouvre un document XML qui référence une DTD inconnue, ou lorsqu'il ne référence pas de DTD, Thot propose à l'utilisateur de transformer le document pour le conformer à une DTD connue du système. Cette fonction de l'éditeur utilise la transformation avec pré–couplage et intègre une méthode spécification interactive de pré–couples.

Le processus de transformation du document XML source comporte plusieurs étapes :

- une empreinte est d'abord générée à partir de la structure du document source (étape représentée en bleu dans la figure 54) ;
- les empreintes sont ensuite comparées et la transformation automatique est effectuée (en vert) ;
- La solution proposée est désambiguisée par la spécification d'un pré–couple. Une nouvelle comparaison avec pré–couplage est alors effectuée et une nouvelle solution à la transformation est proposée (en rouge).

La troisième étape du processus est répétée jusqu'a ce que l'utilisateur juge la transformation pertinente, il peut alors éditer le document comme les autres documents Thot.

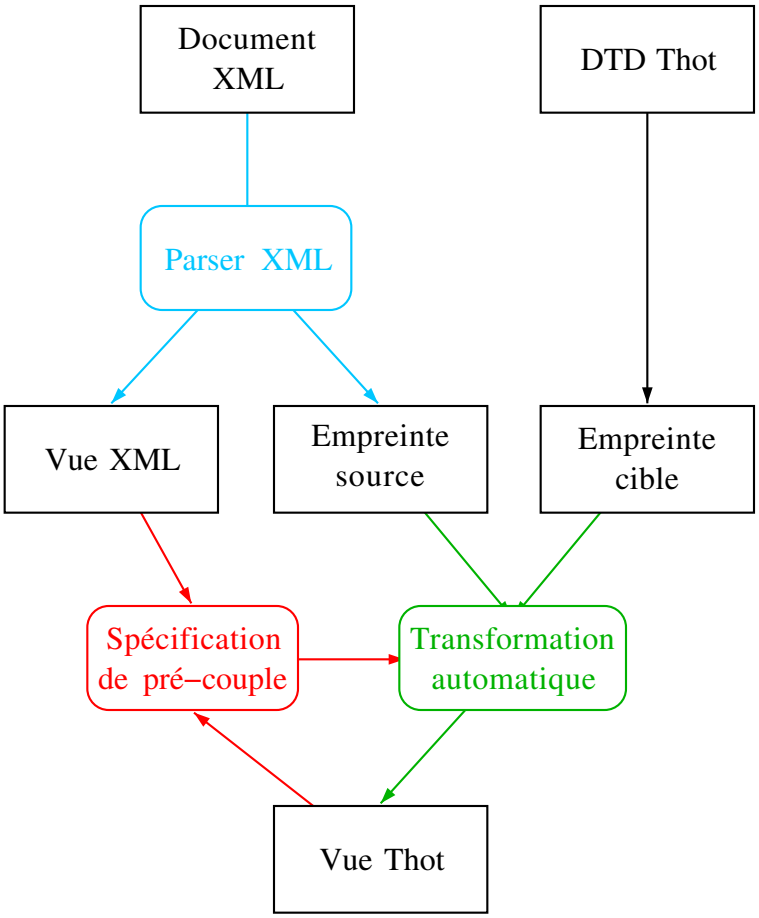


Figure 54 : Processus de conversion de documents XML en documents Thot

VI.5.1.2 Étapes de la transformation

Le document XML est initialement affiché dans une vue qui présente la hiérarchie des éléments composant ce document (Figure 55). Le nom du type de chaque élément XML est présenté en gras et un filet vertical délimite son contenu qui peut être composé de texte ou d'autres éléments.

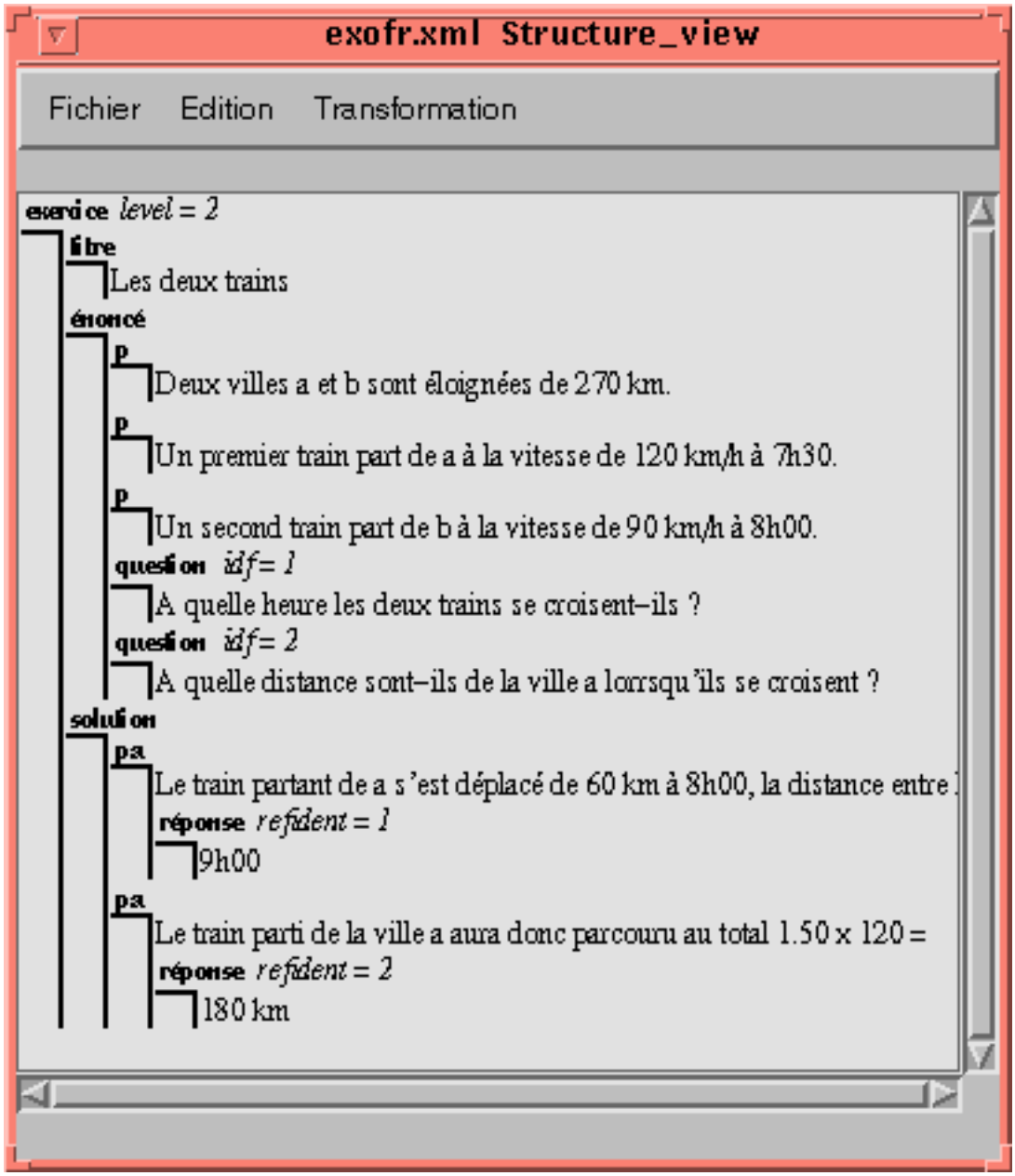


Figure 55 : vue de la hiérarchie des éléments XML

Le menu *Transformation* contient une entrée *DTD cible...* qui permet d'afficher un formulaire proposant de sélectionner la DTD dans laquelle le document doit être converti.

Une fois la DTD cible choisie, une première transformation automatique est effectuée. Pour cela, un arbre de types spécifique est créé à partir de la structure du document source. Cet arbre de type ne correspond pas toujours à une DTD (celle–ci n'étant pas connue par le système) et ne reflète que partiellement la structure du document, celle qui peut être automatiquement déduite et qui dépend de la source de l'information. Cette structure peut être fondée sur les

caractéristiques physiques du document s'il provient d'une analyse d'image ou d'un traitement de texte. Elle peut aussi refléter une composante logique si le document est produit à partir de requêtes de bases de données ou de l'interrogation d'un moteur de recherche sur le Web.

Pour générer l'arbre de type, la procédure suivie consiste à parcourir la structure du document source et à générer les nœuds correspondants dans l'arbre de types en déterminant leur constructeur en fonction du nombre d'occurrences de l'élément correspondant. Chaque fois qu'un nœud de la structure source est parcouru, le traitement suivant est effectué :

- Si le nœud courant de l'arbre de type est nul :
  - Si l'élément courant n'a qu'un seul fils, alors un nœud de constructeur choix est créé dans l'arbre de types comme fils du nœud père. Le fils de l'élément courant est parcouru avec le nouveau nœud comme nœud père et un nœud courant nul.
  - Si l'élément à plusieurs fils de même type, alors un nœud de constructeur liste est créé dans l'arbre de types. Le premier fils de l'élément courant est d'abord parcouru avec un nœud courant nul. Les fils suivants sont ensuite parcourus avec le nœud de l'abre de type créé par le parcours du premier fils comme nœud courant.
  - Si l'élément courant à plusieurs fils de types différents, alors un nœud de constructeur agrégat est créé. Les fils de l'élément courant sont parcourus avec un nœud courant nul.
- Si le nœud courant de l'arbre de types n'est pas nul alors :
  - Si le constructeur du nœud courant est le choix :
    + Si l'élément courant n'a qu'un seul fils et si le type de ce fils est identique au type de l'un des fils du nœud courant, alors le fils de l'élément courant est parcouru avec le fils du nœud de même type comme nœud courant.
    + Si l'élément courant n'a qu'un seul fils de type différent de tous les fils du nœud courant, alors une nouvelle alternative du choix est créée et le fils de l'élément courant est parcouru avec un nœud courant nul.
    + Si l'élément courant à plusieurs fils alors un nœud liste est inséré comme père du nœud courant et tous les fils sont traités comme s'ils étaient fils uniques (les deux cas précédents).

- Si le constructeur du nœud courant est la liste :
  + Si les fils de l'élément courant ont tous un type identique à celui du fils du nœud liste, chcun de ces fils sont parcourus avec le nœud fils du nœud liste comme nœud courant.
  + Sinon un nœud choix est inséré comme fils du nœud liste et chaque fils de l'élément courant est parcouru avec le nœud choix comme nœud courant.
- Si le constructeur du nœud courant est l'agrégat, alors :
  + Si tous les fils du nœud courant respectent l'ordre de l'agrégat tel qu'il est défini dans l'arbre de types, alors chacun des fils est parcouru avec le nœud de l'arbre du même type.
  + Sinon un nœud choix est inséré comme fils du nœud agrégat et chaque fils de l'élément courant est parcouru avec le nœud choix comme nœud courant.

Cette analyse par cas permet d'implémenter une fonction qui produit un arbre de types correspondant à la structure du document source, mais dont toutes les occurences des fils des éléments liste sont représentés par le même nœud.

Après que cet arbre de types ait été créé, la transformation s'effectue selon la méthode automatique donnée dans le chapitre V. Le document résultant de cette transformation est affichée dans une vue formatée Thot (figure 56).

Le document résultant de la transformation peut ne pas convenir à l'utilisateur. Ainsi dans l'exemple, l'élément source *énoncé* a été transformé en élément *liste*, chaque *paragraphe* composant l'*énoncé* ayant été transformé en *item* de *liste*. Un autre facteur de non pertinence est la transformation des éléments *réponse* (9h00 et 180 km) en éléments *cite* alors qu'il aurait été plus pertinent de les intégrer aux paragraphes des réponses.

L'utilisateur peut alors procéder à la correction de la transformation proposée. Pour cela, il spécifie des pré–couples de manière incrémentale pour raffiner ce résultat. La spécification de chaque pré–couple déclenche une nouvelle comparaison et le résultat est immédiatement affiché dans la vue du document cible.

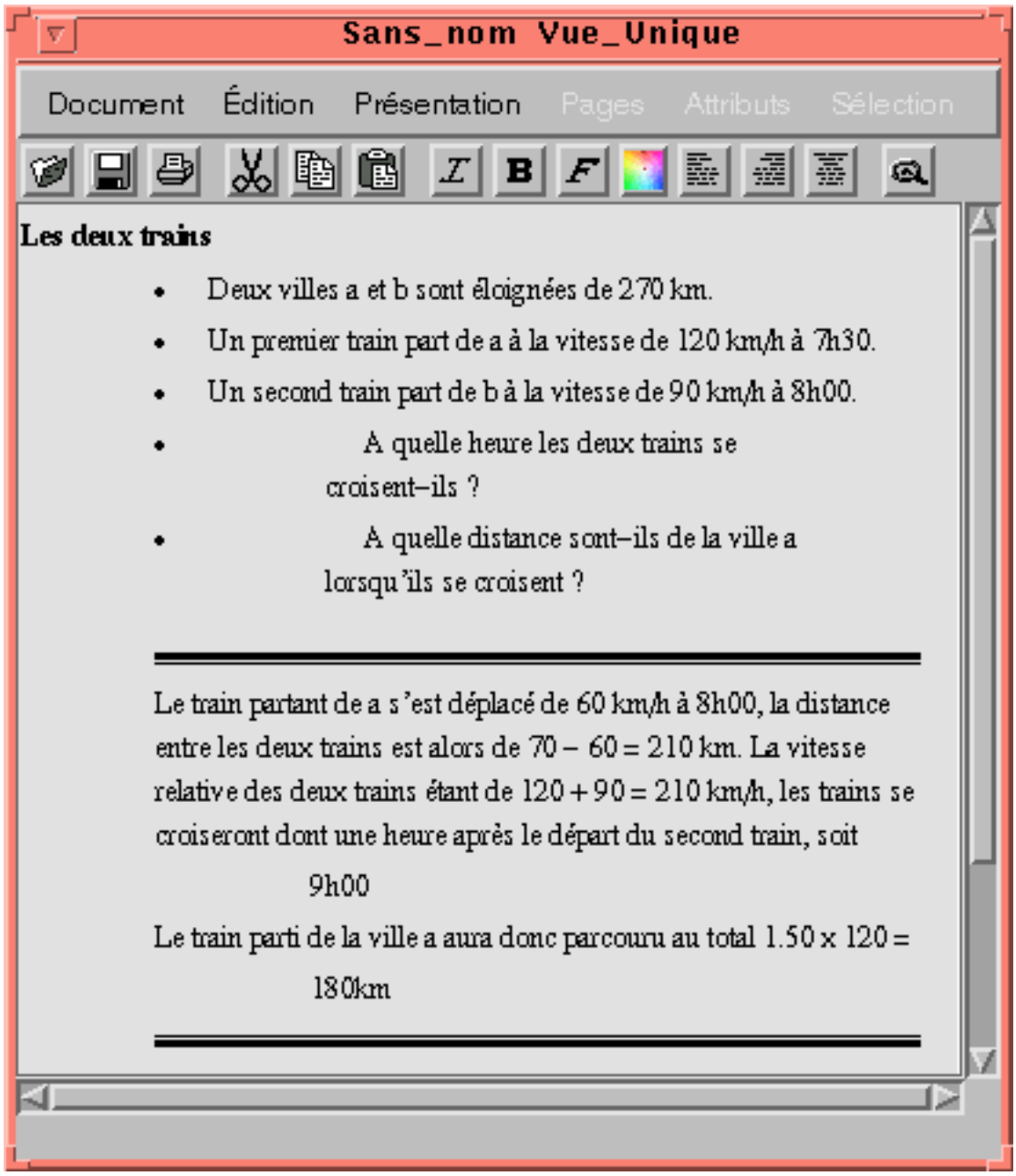


Figure 56 : Document résultant de la transformation automatique

La spécification d'un pré–couple se fait en deux temps :

- Dans un premier temps, le nœud source est sélectionné « par l'exemple ». Pour cela l'utilisateur sélectionne un élément particulier dans la vue du document source. Cet élément est affiché en rouge vif et tous les éléments du même type (plus précisément dont le type porte le même nom) sont affichés en rouge foncé. L'utilisateur a alors une perception de la sélection et de l'ensemble d'éléments du même type canonique qui seront impliqués par la transformation utilisant le pré–couple

Ainsi, pour spécifier le pré–couple entre les éléments *réponse* et le texte des *item* de *liste*, l'utilisateur sélectionne un des éléments du type *réponse* qui s'affiche en rouge vif (ici 9h00) et l'autre élément de même type (180 km) est affiché en rouge foncé. La désignation des éléments *réponse* est illustrée par la figure 57.

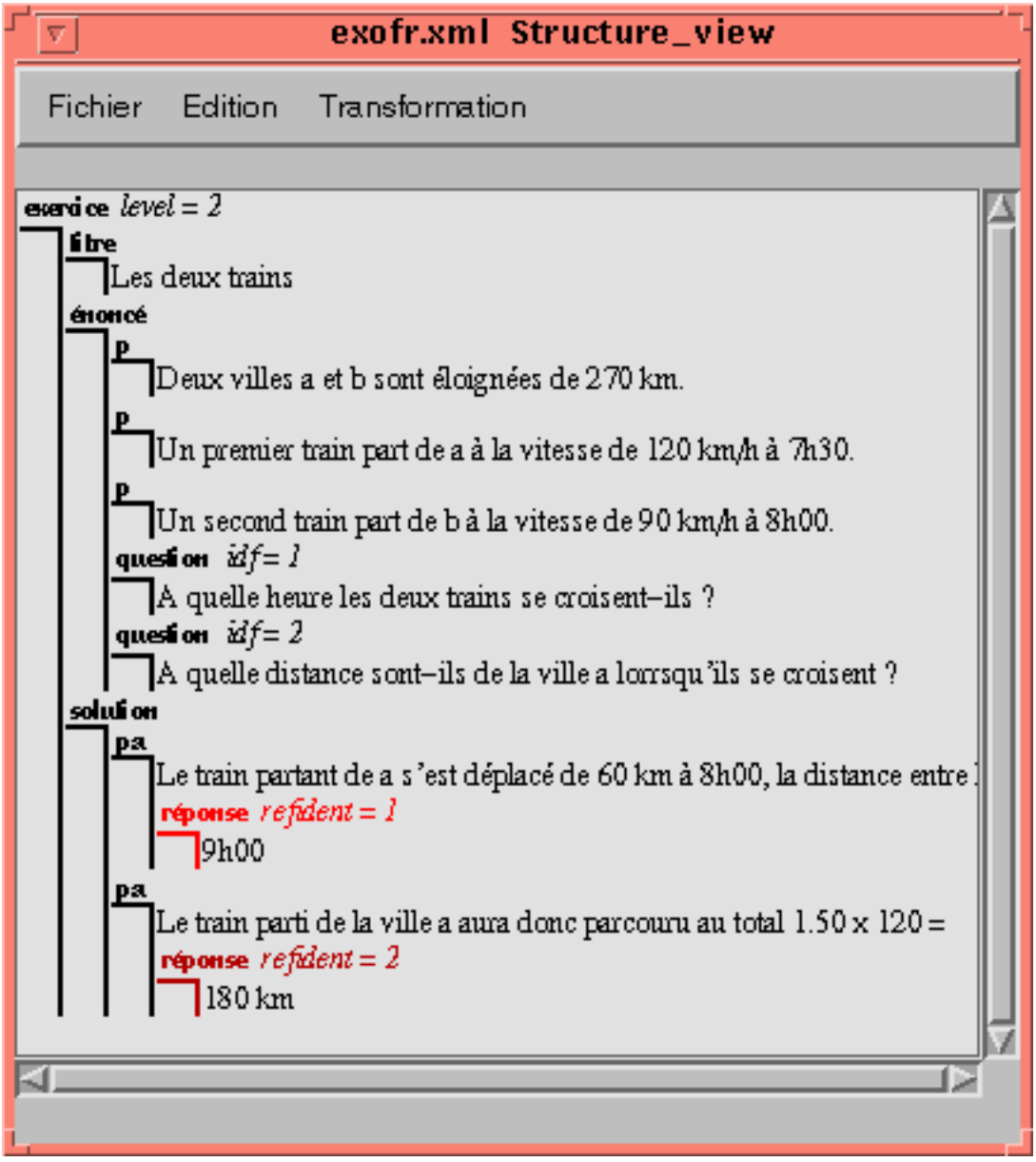


Figure 57 : Détermination du type origine d'un pré–couple

- Le second temps consiste à déterminer le(s) nœuds cible du pré–couplage. Pour cela, l'utilisateur peut soit désigner un élément déjà présent dans le document cible, soit utiliser le menu d'insertion de Thot pour créer de nouveaux éléments du type cible du pré–couple.

Dans l'exemple (figure 58), l'utilisateur désigne un paragraphe groupé pour spécifier la cible du pré–couple dont l'origine est le type *p* (flèche verte sur la figure). Ce pré–couple se substitue à la relation trouvée par l'algorithme de comparison automatique (flèche rouge).

Lorsqu'un pré–couple est spécifié, une nouvelle transformation intégrant cette spécification est effectuée, le résultat est immédiatement affiché dans la vue du document cible Thot. Lorsque l'utilisateur juge la transformation pertinente, il peut fermer la vue du document source et éditer le document Thot de façon classique.

Pour accroître la souplesse et la facilité d'utilisation du système de transformation, nous avons ajouté deux fonctions permettant de modifier des pré–couples et de mémoriser les pré–couplages utilisés pour la conversion du document.

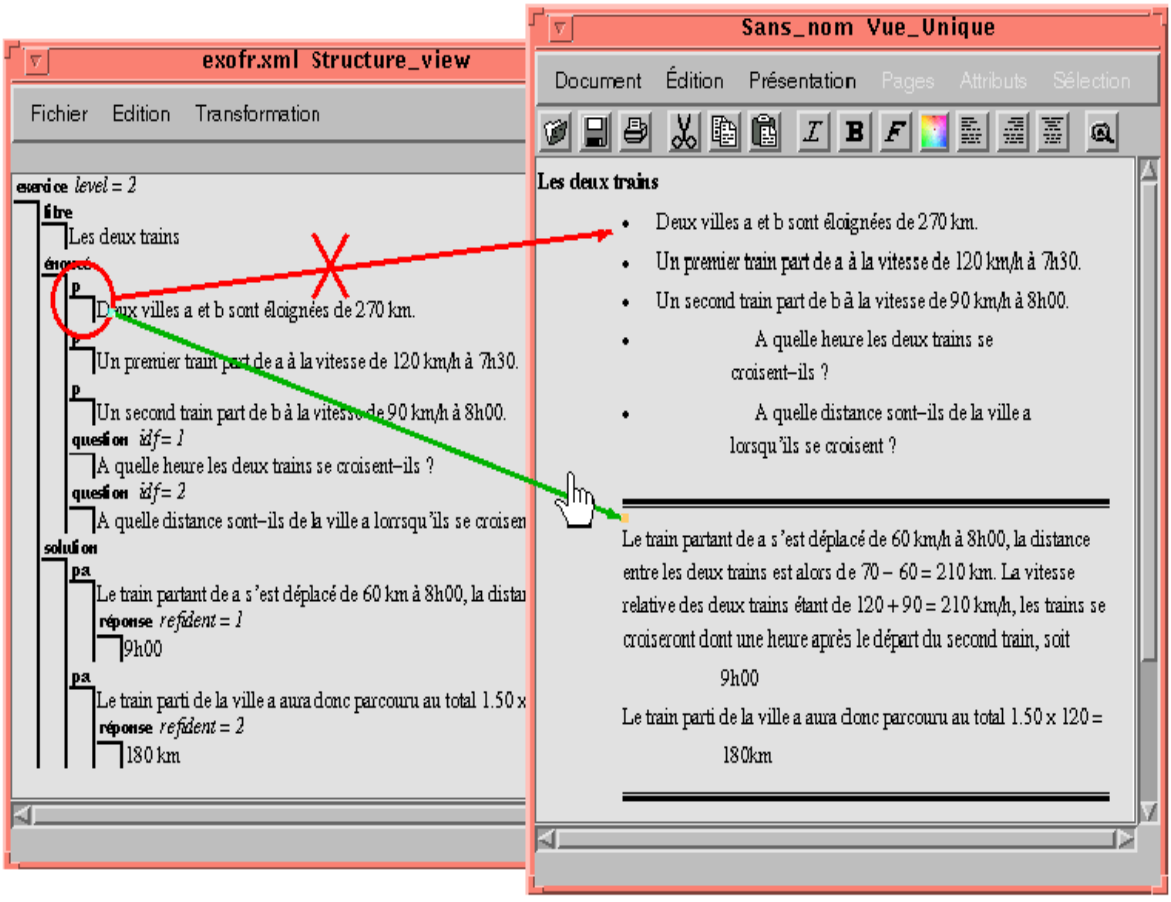


Figure 58 : Désignation interactive de la cible d'un pré–couple

VI.5.1.3 Modification des pré–couples

Au cours du processus de correction, il se peut qu'un pré–couple ne mène pas à une solution adéquate. Ceci peut être causé par une erreur de manipulation de l'utilisateur ou par un effet de bord non prévu lors de la spécification impliquant une mauvaise transformation d'éléments autres que ceux qui ont servi à spécifier le pré–couple.

Pour supprimer ces erreurs, il est nécessaire que l'utilisateur puisse visualiser, et éventuellement supprimer ou modifier un pré–couple entré précédemment.

Par définition du système de type, un élément est défini par un unique type canonique. Par conséquent, il ne peut exister qu'un unique pré–couple impliquant un élément du document source. Le pré–couple associé au type canonique de l'élément peut donc être affiché lors de la sélection de l'élément source. La perception d'un pré–couple est rendue par deux moyens :

1. Le nom du type du nœud cible affiché à côté du nœud origine du pré–couple sélectionné.
2. Les éléments définis par le type canonique cible du pré–couple sont affichés sur fond de couleur.

Le menu *Transformation* de la vue source permet de supprimer ou de définir un pré–couple. Lorsqu'un pré–couple est modifié ou supprimé, la transformation est immédiatement relancée, donnant ainsi à l'utilisateur un retour instantané de la commande effectuée.

VI.5.1.4 Mémorisation des pré–couples

Lorsqu'une transformation est jugée pertinente par l'utilisateur, l'ensemble des pré–couples spécifiés durant la phase de spécification est mémorisée dans la syntaxe précédemment définie (cf. section VI.3.2). Le langage de pré–couplage permet de mémoriser le contexte dans lequel un pré–couple a été défini. Dans cette application, il permet de mémoriser le type de l'élément source et la DTD cible.

Lorsqu'un nouveau document sera ouvert, les expressions des pré–couplages mémorisées seront évaluées et pourront être utilisées lors de la comparaison initiale.

VI.5.2 Pré–couplage et applications des transformations

La technique de transformation présentée dans les deux derniers chapitres repose sur le modèle d'arbre de types. Cette représentation des structures génériques des données ne sont pas liées à un environnement spécifique tel que la boîte à outils Thot.

Ainsi nous pouvons envisager de proposer un mécanisme de transformations de structure s'appuyant sur l'interface DOM [Wood 98] lui permettant de s'intégrer dans de nombreuses applications XML. La norme XML est de plus en plus utilisée et est en passe de devenir un format de représentation de données de référence. La prise en compte de cette norme est non seulement une priorité dans le développement des logiciels d'édition mais aussi une préoccupation dans d'autres domaines (bases de données, recherche d'information, publication Web). XML est la garantie de l'utilisation d'une syntaxe commune facilitant l'échange de documents entre applications. Si XML permet une compatibilité au niveau syntaxique, les structures sont spécifiques aux documents. Par exemple, les balises employées dans un document provenant d'un traitement de texte seront certainement différentes de celles employées dans un document résultant d'une requête sur une base de données.

VI.6 Conclusion

Nous avons proposé dans ce chapitre une méthode de transformation combinant la transformation automatique et la méthode déclarative. La transformation automatique est basée sur des similarités entre la structure générique définissant les éléments source et la structure du type cible. Ce mode de transformation est uniquement fondé sur la structure. Les expressions de transformation spécifient des correspondances explicites entre certains types de la structure source et un sous ensemble des types cible. Ces correspondances permettent d'exprimer la sémantique de la transformation par la liaison des types d'éléments jouant des rôles similaires dans les structures des documents source et cible.

L'originalité de notre méthode est de donner une part prépondérante à la transformation automatique, des spécifications de relations appelées pré–couplage venant compléter et corriger cette transformation.

Nous avons défini un langage de spécification permettant de décrire ces pré–couplages. Nous avons ensuite proposé une méthode interactive et incrémentale qui complète le processus de transformation. En effet, l'intervention de

l'utilisateur permet d'établir des correspondances localisées qui permettent de réduire les possibilités de couplage incorrect.

Nous avons finalement proposé plusieurs champs d'application de cette méthode de transformation répondant aux besoins que nous avons identifiés au début de ce travail.

Chapitre VII

Conclusion

VII.1 Rappel des objectifs

L'utilisation des documents structurés apporte de nombreux avantages aux applications de traitement de documents. Elle permet en particulier d'automatiser une partie des traitements effectués sur les documents. Les documents structurés ont également contribué à améliorer la portabilité en fournissant un formalisme commun pour la représentation de ces documents.

Les applications de documents structurés souffrent néanmoins de limites liées aux contraintes apportées par la description des structures par des grammaires hors–contexte. Ces contraintes portent principalement sur les fonctions d'édition qui doivent garantir la conservation de la conformité des documents après leur application. Une autre limite importante concerne la ré–utilisation de documents existants, qui doivent être adaptés lorsque leur structure générique d'origine est différente de celle exploitée par la nouvelle application.

L'objectif de ce travail de thèse est de répondre à ces limites par des techniques de transformation permettant d'adapter tout ou partie d'un document à une structure cible donnée. Les solutions proposées et les expérimentations réalisées dans ce travail ont pour cadre applicatif l'édition de document structurés.

La transformation de structure n'est pas une problématique récente dans le domaine des documents structurés, mais aucune réponse satisfaisante n'a été donnée par les travaux menés jusqu'à ce jour.

- Les systèmes de transformation explicites permettent de répondre à des besoins précis et ne peuvent être généralisés qu'au prix de déclarations complexes.
- Les techniques de transformation automatiques ne permettent d'effectuer que des transformations élémentaires, ou peu pertinentes.

Notre objectif était donc de concevoir un système de transformation combinant ces deux approches pour profiter des avantages de chacune, tout en apportant une réponse à leurs limites. En termes de spécifications fonctionnelles, cet objectif se traduit par la nécessité de proposer un module de transformation permettant de :

- mettre en œuvre les transformations utiles au cours de l'édition ;
- assurer la validité du document issu de la transformation par rapport à la DTD cible ;
- s'appuyer sur une expression minimale des transformations ;
- utiliser un algorithme qui ait des performances satisfaisantes pour permettre des transformations interactives ou utilisées par les commandes d'édition.

VII.2 Rappel du travail réalisé

VII.2.1 Démarche suivie

Cette thèse a été effectuée selon une démarche comportant trois phases principales : étude des applications de traitement de documents et des besoins de transformation, évaluation et expérimentation des techniques de transformation existantes et enfin spécification et mise en œuvre de nouvelle approche pour la transformation des documents structurés. Ces trois phases sont développées ci–dessous :

1. Étude des besoins

Pour réaliser nos objectifs nous avons commencé par faire une étude des applications traitant des documents pour identifier leurs besoins. Ce tour d'horizon nous à permis de situer le problème dans le contexte plus large de l'évolution du document tout au long de son existence. A l'issue de cette étude nous avons identifié les applications importantes de la transformation de structure pour l'édition de documents structurés.

2. Évaluation et expérimentation des techniques de transformation

Dans une seconde partie, nous avons classé et évalué les systèmes de transformation existants dans deux catégories de méthodes : les transformations explicites et les transformations automatiques fondées sur la similarité des structures génériques source et cible.

Pour chaque catégorie nous avons procédé en trois temps :

1. **Analyse des principes généraux de la méthode** : les classes de langages utilisés, les modèles de représentation et les techniques algorithmiques ont été présentés et comparés.

   Cette analyse nous a permis d'isoler les filtres des transformations explicites. Nous avons également identifié deux approches algorithmiques de la transformation explicite : les transformations dirigées par la source et les transformations par requêtes.

   Nous avons également présenté le système de types sur lequel s'appuient les techniques de transformation automatique.
2. **Évaluation des techniques et outils existants** : en particulier, pour l'étude des outils de transformation explicite, nous avons utilisé un exemple de transformation concret pour comparer les différents systèmes au niveau de l'expression et des capacités de ces systèmes.
3. **Expérimentation des techniques dans un système d'édition** : nous avons au cours de ce travail développé des systèmes de transformation pour chacune des techniques évaluées permettant d'évaluer leur capacité à s'adapter au domaine de l'édition.

3. Proposition d'une méthode de transformation originale

Dans un troisième temps nous avons proposé une méthode de transformation originale répondant aux objectifs fixés dans la première partie et dont nous rappelons les principaux résultats ci–dessous.

VII.2.2 Résultats théoriques

Nous avons identifié dans le chapitre III les situations dans lesquelles la transformation de documents est utilisée pour :

- importer des documents existants et permettre leur utilisation dans les applications de documents structurés ;
- permettre une modification dynamique de la structure du document ;

- convertir de documents entre modèles et les mettre à jour lors de l'évolution d'une DTD.

L'évaluation des techniques de conversion nous a permis d'identifier quelles techniques adaptées à ces différentes tâches et éventuellement quelles adaptations doivent être apportées. La synthèse de cet état de l'art nous a permis de spécifier une méthode de transformation bénéficiant des avantages des deux approches de la transformation.

L'originalité de la méthode présentée dans le chapitre VI est d'intégrer dans un même processus deux processus de transformation qui semblaient *a priori* totalement opposés. Cette combinaison d'approches permet de réaliser des transformations plus pertinentes que les transformations basées sur la comparaison des structures générique, tout en permettant une spécification plus aisée que celle requise par les systèmes explicites.

L'idée principale sur laquelle repose la méthode de transformation présentée est de réaliser une transformation automatique qui prend en compte la dimension syntaxique de la comparaison de documents. Notre expérience a montré que la dimension syntaxique n'est pas suffisante pour réaliser des transformations satisfaisantes. Avec les pré–couplages nous permettons d'introduire une part de sémantique pour diriger la transformation automatique.

VII.2.3 Résultats pratiques

Cette étude n'offrait un réel intérêt que si elle était concrétisée par des réalisations. La réalisation des systèmes de transformation dans les outils développés dans le projet Opéra à permis de valider par la pratique les idées qui ont été proposées dans ce mémoire.

Ce travail a conduit au développement des systèmes de transformation aujourd'hui intégrés dans les logiciels du projet Opéra et du W3C :

- Un filtre de conversion permettant de transformer les documents Thot de type *article* en documents de type *rapport* (évolution de DTD).
- Un système de transformation dynamique utilisant la méthode déclarative intégré au logiciel Amaya. Ce système permet à Amaya de proposer une édition facilitée des documents HTML et XML : l'auteur peut ainsi typer *a posteriori* les éléments du document (transformations en cours d'édition).
- Un prototype de transformation automatique utilisant les transformations automatiques et permettant d'améliorer les commandes d'édition

copier/coller et *changer en* dans l'éditeur Thot (transformations en cours d'édition).

- Un module permettant d'importer des documents XML dans l'éditeur Thot utilisant la transformation avec pré–couplage pour combiner les approches automatique et explicite (conversion et structuration *a posteriori*).

VII.3 Bilan et évaluation

Le problème de la transformation de documents est très vaste. Les méthodes de transformation sont diversifiées et leur pertinence est dépendante du domaine auquel elles sont appliquées. Par exemple, les filtres que nous avons présentés comme les systèmes les moins adaptables peuvent se révéler adéquats pour des applications précises comme la conversion de grandes quantités de documents. Ce travail nous a permis de prendre conscience qu'il n'existe pas de solution générale au problème de la transformation, mais plutôt différentes techniques adaptées à des situations particulières.

Parmi les applications de transformation de documents, le domaine des applications interactives est celui qui avait été le moins étudié et pour lequel les techniques de transformation existantes étaient les moins efficaces. C'est pourquoi nous avons concentré notre travail sur ce domaine. Les résultats obtenus sont également adpatés à d'autres applications et permettent de répondre à certaines limites des outils de transformation plus traditionnels principalement en évitant un mode de spécification des transformations exhaustif.

De part la diversité des besoins de transformation et des modèles de transformation, il est difficile de quantifier les apports d'une méthode de transformation par rapport à une autre. Si notre centre d'intérêt n'avait pas été l'édition de documents, mais l'interrogation de bases de données ou la gestion de fonds documentaires, l'évaluation des systèmes et les propositions que nous aurions formulées auraient certainement été très différentes.

L'évaluation de la pertinence des techniques que nous proposons est rendue difficile également par le fait que les transformations se déroulent dans un cadre interactif. De nombreux facteurs entrent en compte dans une telle évaluation : les types de documents en jeu, la complexité de l'instance source ainsi que le comportement de l'utilisateur (niveau de sélection, connaissance de la structure logique du document).

VII.4 Perspectives

La contribution apportée dans ce travail peut être appliquée à de nombreux domaines. En particulier, la norme XML ouvre des champs d'application qui s'étendent au–delà du domaine de l'édition de documents. La méthode de transformation présentée dans le chapitre VI demande également à être améliorée. Nous pouvons dégager les perspectives suivantes :

Amélioration de la méthode de transformation avec pré–couplage

La méthode de transformation avec pré–couplage peut être améliorée sur deux points :

- La génération des arbres de types spécifiques lorsqu'un document venant sans DTD est importé demande à être améliorée. En particulier, en présence d'une suite d'éléments de types différents, nous créons de façon arbitraire une liste de choix dans l'arbre de types. Dans certains cas, qui peuvent être identifiés lorsqu'il existe plusieurs occurrences de ces éléments dans le document source, des nœuds *agrégat* seraient plus adaptés.
- Une information d'ordre sémantique peut être extraite du document pour spécifier des pré–couplages grâce à des méthodes d'analyse linguistique du contenu du document. En effet, la catégorisation des mots composant les éléments du document, associée à une analyse probabiliste permet de déterminer le rôle des éléments du document. Il est ainsi possible, par exemple, de distinguer un résumé d'un paragraphe général ou d'une explication technique.

  Nous pouvons envisager de nous appuyer sur de telles techniques pour permettre la détermination du type d'éléments du document source et de leur correspondance dans la structure cible.

Architecture des systèmes de transformation

Les techniques de transformation élaborées dans ce travail et les expérimentations sont liées à l'environnement d'édition Thot. Une architecture à base de composants fondée sur les interfaces standards telles que DOM permettrait à de nombreuses applications d'intégrer un service de transformation de structure. Il existe un besoin réel de transformation dans les applications industrielles d'édition de documents, mais aussi pour toutes les applications manipulant des données structurées. Les actions industrielles menées en partenariat avec le projet Opéra en témoignent comme les collaborations avec les sociétés Dassault–Aviation et Aérospatiale.

Vers d'autres outils de transformation

Le travail de prospection mené durant cette thèse permet d'envisager d'autres méthodes de transformation permettant de mieux intégrer la processus de transformation dans l'édition de documents. Les travaux récents sur ce thème [Trombettoni 97] permettent d'envisager un système d'édition de documents structurés basé sur les contraintes. Pour cela les règles de structure des documents doivent être exprimées sous forme de contraintes. Ainsi, étant donné un ensemble d'éléments, un résolveur de contraintes pourrait donner un ensemble de structurations possibles de cet ensemble d'éléments en fonction du contexte dans lequel il se trouve.

De nouvelles applications de la transformation

La norme XML est très récente, l'implication de cette norme sur les applications informatiques de nombreux domaines est difficilement prévisible, notamment comme format de représentation de données structurées. On constate toutefois un grand intérêt de la part des concepteurs de bases de données pour cette nouvelle norme. Les techniques de conversion fournissent un moyen de composer le résultat des requêtes sur des bases des données ou sur des moteurs de recherche pour construire des documents homogènes [Baru 98].

Annexe A

Systèmes de types des langages de programmation

A.1 Langages de programmation et systèmes de types

A.1.1 Langages typés et non typés

Les langages de programmation sont dépendants des applications qu'ils permettent de programmer : les opérations définies dans un langage permettent de réaliser des tâches particulières. Par exemple, le langage Lisp permet d'implémenter une algorithmique fonctionnelle adaptée à l'intelligence artificielle et à la déduction automatique. Le langage C, qui permet de manipuler les adresses mémoire, est couramment utilisé pour implémenter des fonctions du système d'exploitation. Certains langages à objets tels que Java permettent de concevoir des interfaces utilisateurs à un moindre coût et d'utiliser, de façon intégrée, des protocoles réseau de haut niveau. La nature des types de données proposés par chacun de ces langages diffère selon la nature des données traitées : le langage C permet la manipulation de variables de type pointeur, tandis que le langage Java l'interdit. Les structures de données complexes dans le langage Lisp sont toutes construites à base de tuples, tandis que les autres langages proposent des constructions de types plus élaborées (structures, enregistrements).

L'affectation de types aux variables et aux expressions d'un programme permet au programmeur de ne pas avoir le souci de la représentation en mémoire des données manipulées. Lorsqu'il utilise un langage non typé, tel que l'assembleur, le programmeur doit contrôler que le domaine des variables et les paramètres des instructions qu'il écrit sont corrects. De même, il doit gérer la

réservation et la libération des espaces mémoire nécessaires pour le stockage de ces variables.

Par contre, Le λ–calcul, qui peut également être considéré comme un langage non typé (ce n'est pas un langage de programmation), garantit l'absence d'erreur car toute donnée est une fonction et retourne donc forcément un résultat valide, de même tout opérateur est une fonction et retourne donc un résultat. Cependant les expressions fonctionnelles sont difficiles à comprendre. Cela présente un inconvénient pour la maintenance des programmes écrits dans des langages proches du λ–calcul (Lisp, Scheme).

Les types de données apportent un supplément d'information permettant au système de prendre en charge les tâches qui incombent au programmeur dans les langages non typés (réservation de mémoire, contrôle des domaines de valeur), tout en apportant une meilleure lisibilité des programmes [Cardelli 85], ce qui se traduit par :

- une économie de temps d'exécution,
- une économie de temps de développement,
- une économie de compilation,
- une simplicité d'expression des langages.

A.1.2 Systèmes de types

Nous donnons ici la définition des éléments d'un langage de programmation typé, ainsi que deux approches permettant l'affectation de types aux variables d'un programme.

Définitions

L'ensemble des types pouvant être exprimés dans un langage, les règles qui permettent de les définir et les règles qui définissent le type d'une expression sont communément appelés *système de types* du langage.

L'ensemble des types valides d'un langage est composé de *types de base* représentant les unités d'information atomique pouvant être manipulées par le langage. Par exemple, les types entier, booléen, caractère, chaîne ou réel sont des types de base couramment définis dans les langages de programmation. Ces types de base peuvent être combinés entre eux pour définir des *types construits*. Deux approches permettent de définir ou d'utiliser les types construits :

1. Définition *explicite* des types de données : une *expression de type* décrit un type construit, ses opérandes sont des types de base ou

d'autres types construits et ses opérateurs sont appelés *constructeurs* de type.

2. Définition *implicite* des types de données : le type d'une expression est calculé lors de son évaluation. Le type des variables utilisées dans un programme est déduit lors de leur utilisation.

Chacune de ces deux approches implique des politiques différentes pour la compilation et l'interprétation des langages qui les utilisent. Les techniques mises en œuvre pour garantir la correction d'un programme vis–à–vis du système de types du langage sont également différentes.

D'une manière générale nous pouvons dire que les langages explicitement typés permettent une meilleure compréhension des programmes et une économie du temps de compilation et d'exécution. En contrepartie, ils imposent des contraintes sur l'application des traitements à des variables d'un type donné : les variables utilisées dans un calcul doivent être d'un type acceptable pour l'opérande auquel elles sont appliqués.

La plupart des langages explicitement typés permettent de nommer des expressions de type pour les ré–utiliser ultérieurement. Le nommage d'une expression de type est introduit par un mot–clé spécifique (typedef en C, type en Pascal) et définit un *nom de type*. Si un nom de type est utilisé dans l'expression qui le définit, le type est alors un type *récursif*.

Le calcul des types des expressions et des variables des programmes écrits dans des langages implicitement typés implique un surcoût de temps à l'exécution du programme. Le type d'une variable ne peut pas toujours être déduit d'une seule expression, et le système doit faire des hypothèses sur le type de la variable qui pourront être confirmées ou infirmées par la suite. Dans l'exemple suivant,

```
a = 1 + 3;
```

Le type de a peut être entier ou réel, le système doit conserver tous les types potentiels de la variable a, pour éventuellement choisir ultérieurement. L'expression

```
a = sin (b/2);
```

détermine le type réel pour a. Certains types peuvent être incomplets du fait qu'une expression utilise des variables dont le type n'a pas encore été déterminé. Dans ce cas des *variables de type* doivent être introduites pour paramétrer le type cherché. Une expression est dite *entièrement typée* lorsque l'ensemble des variables de type utilisées pour définir son type sont valuées (voir section A.4).

A.1.3 Contrôle de type et sécurité à l'exécution

Les types de données permettent une plus grande lisibilité des programmes et un temps de développement réduit. L'utilisation de systèmes de types permettent également d'écarter les erreurs à l'exécution. En effet, lors de la compilation ou de l'exécution d'un programme, le système de types est un moyen de contrôler que les paramètres appliqués à un opérateur ou à une fonction sont corrects. Il est ainsi possible de prévoir si une opération pourra aboutir et donnera un résultat (type et domaine des paramètres corrects) ou si elle déclenchera à coup sûr une erreur (paramètres incorrects).

La sécurité des programmes à l'exécution n'est pas l'intérêt premier de ce travail, mais cette propriété de sécurité s'appuie d'une part sur le fait qu'un programme est bien typé, d'autre part que le système de type du langage permet d'assurer cette propriété, c'est à dire que les opérations permises par les langages ne produisent que des variables de valeurs correctes par rapport à leur type. Ce n'est pas le cas, par exemple, de l'opération de coercition (ou *casting*) du langage C qui permet de changer le type d'une valeur sans contrôler sa validité pour le type dans lequel elle est transformée.

Les systèmes de types assurant la propriété de sécurité à l'exécution restreignent en général la souplesse de programmation en interdisant les opérations de changement de type pouvant engendrer des erreurs de type. L'arithmétique de pointeurs est généralement exclue des langages utilisant un système de type conservant la propriété de sécurité.

La conception d'un langage de programmation se heurte donc au défi de proposer un système de type sûr sans restreindre les opérations proposées à l'utilisateur.

A.2 Un contrôleur de types simple

Dans cette section nous décrivons un langage permettant de spécifier le système de types d'un langage et de l'exploiter pour décrire les algorithmes de contrôle et d'inférence de types.

Un système de types s'appuie sur la syntaxe du langage pour lequel il est défini, associant aux nœuds de l'arbre syntaxique (construit par l'interpréteur ou le compilateur) une information de type sous la forme d'attributs. Nous donnons ici une représentation permettant de décrire la façon dont ces attributs sont calculés et/ou vérifiés.

Nous appuyons notre description sur un exemple de langage simple que nous nommerons L. Ce langage permet de décrire des programmes constitués d'un ensemble de variables typées et d'une expression unique. Sa grammaire est donnée par la figure 59. Les programmes décrits par le langage L (`P`) sont composés d'une partie déclaration (`D`) où sont déclarées les variables et les types (`T`) auxquelles elles appartiennent et d'une partie expression (`E`) combinant des littéraux et des variables (`id`) à l'aide des opérateurs d'addition d'entiers (`+`), de concaténation de chaînes (`&`), d'accès à un tableau, à une structure et d'application de fonctions.

```
P → D ; E
D → D ; D | id : T
T → 'entier' | 'chaîne' | 'tableau' '[' entier ']' 'de' T |
     'structure' '{' D '}' |
     'fonction' '(' id ':' T ')' ':' T '{' E '}'
E → littéral | nb | id | E '+' E | E '&' E |
     E '[' E ']' | E '.' id | E '(' E ')'
```

Figure 59 : Grammaire du langage L.

Le langage L est un langage fonctionnel ne contenant pas de structures de contrôle telles que les boucles ou les expressions conditionnelles. Les structures de contrôle ne posant pas de problèmes particulier pour le contrôle de type et nous les avons ignorées pour simplifier la syntaxe du langage.

Un programme est engendré par le non–terminal P de la grammaire. Les déclarations de variables typées (D) sont composées d'un identificateur et d'une expression de type. Voici un exemple de programme écrit avec le langage L qui calcule le double de la variable a :

```
a : entier;
double : fonction ( c : entier ) : entier
  {
    c + c
  };
double (a)
```

Les expressions de types de ce programme sont : `entier` (pour définir le type de la variable a) et `fonction (entier) : entier` (pour définir le type de la variable double).

Le langage L comme tous les langages de programmation définissent deux catégories de types, les types de base et les types construits :

- Les types de base du langage L sont les types entier et chaîne.
- Les types construits sont les entités composites décrites par une expression de type. Les constructeurs du langage L sont : tableau, pointeur, structure et fonction.

Les expressions de type ne sont pas appropriées aux algorithmes de contrôle de type. Il est plus commode de représenter les types sous forme de graphes. La figure 60 montre la représentation sous forme de graphe d'une expression de type. Dans un *graphe de types*, les nœud terminaux représentent les types de base et les nœuds non terminaux sont étiquetés avec les constructeurs des types qu'ils représentent (en police courrier), les noms de types appartenant à chacun des nœuds étant donnés à titre indicatif (en italique).

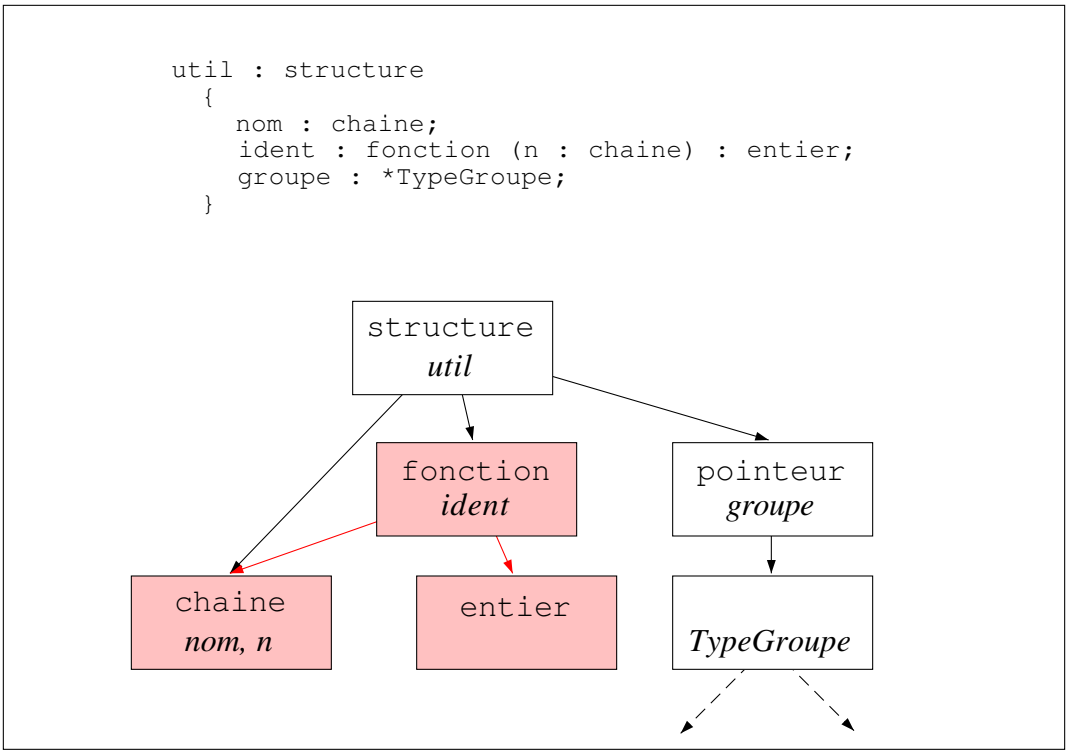


Figure 60 : Un exemple de type construit et son graphe de types

Par la suite nous assimilerons un type et le graphe engendré par l'ensemble des sommets pouvant être atteints depuis le sommet représentant le type. Dans la figure 60, nous avons représenté le type *ident* en rouge.

Vérification du type d'une expression

Le contrôle de type d'une expression s'appuie sur un ensemble de règles sémantiques donnant le type de cette expression en fonction du type de ses opérandes. Pour les unités syntaxiques représentant les types de base, ces règles sont :

E → littéral	chaîne
E → nb	entier

La fonction TypeSymb recherche dans la table des symboles le type d'un identificateur id. Cette fonction est utilisée lorsqu'un identificateur est rencontré dans une expression :

$E \rightarrow id$	TypeSymb (id)

Le type d'une expression construite dépend du type de ses opérandes :

$E \rightarrow E_1 + E_2$	si Type (E_1) = entier et Type (E_2) = entier alors entier sinon erreur de type.
$E \rightarrow E_1$ & E_2	si Type (E_1) = chaîne et Type (E_2) = chaîne alors chaîne sinon erreur de type.

La vérification de type lors de l'accès à une cellule de tableau dépend du type des éléments du tableau. Pour désigner ces éléments, on utilise une variable de type t, permettant de faire abstraction du type de données stockés dans le tableau.

$E \rightarrow E_1$[E_2]	si Type(E_1) = tableau de t et Type(E_2) = entier alors t sinon erreur de type.
$E \rightarrow \uparrow E_1$	si Type(E_1)= pointeur vers t alors t sinon erreur de type.

Lors de l'accès à un champ de structure il faut vérifier que l'identificateur de champ représente réellement un champ de la structure, pour cela, on utilise la fonction champs, retournant l'ensemble des identificateurs des champs d'une structure. La fonction ChercherType recherche le type d'un champ particulier d'une structure.

$E \rightarrow E_1$.id	si Type (E_1) = structure et id $\subset$ Champs (E_1) alors ChercherType (E_1.id) sinon erreur de type.
$E \rightarrow E_1(E_2)$	si Type (E_1) = fonction (t1) : t2 et Type (E_2) = t1 alors t2 sinon erreur de type.

Un algorithme simple de contrôle de type consiste à parcourir l'arbre syntaxique de l'expression en calculant le type de chaque nœud en fonction des

types de ses composants. Le type de l'expression est correct si l'algorithme ne retourne pas d'erreur de type.

Les fonctionalités décrites ci–dessus sont proposées par de nombreux langages de programmation et nécessitent la mise en œuvre de techniques de contrôle de type plus complexes pour déterminer si un programme est bien typé et donc exempt d'erreurs non désirables à l'exécution (propriété de sûreté). Ces techniques avancées que sont l'inférence de type, la surcharge d'opérateur, la coercition, l'équivalence de types ou encore les fonctions polymorphes sont exposées dans la section suivante.

A.3 Limites et extensions du contrôle de types simple

Le langage et l'algorithme de vérification de type présentés ci–dessus sont basiques et ne supportent pas la plupart des caractéristiques de langages de programmation de haut niveau. Les caractéristiques suivantes ne sont pas proposées par le langage L et ne sont pas supportées :

- *Utilisation de variables de types* : les critères d'égalité entre types ne se fondent que sur l'égalité des constructeurs. L'algorithme ne permet pas de déterminer si une variable du programme est compatible avec une variable de type. Le contrôle de type doit pouvoir déterminer si une expression du langage appartient à un type contenant des variables de types dans sa définition.
- *Types récursifs* : l'utilisation des variables de types permet la définition de types intervenant dans leur propre définition. Cette propriété introduit des cycles dans le graphe de représentation des types de données, l'algorithme de contrôle de types doit prendre en compte la gestion des cycles.
- *Surcharge des opérateurs* : dans le langage L chaque opérateur agit sur des opérandes dont le type est bien défini. Dans certains langages, le même opérateur peut représenter des opérations différentes quand il est appliqué à des types des données différentes. Un exemple simple est l'opérateur + qui peut être indifféremment appliqué à des opérandes entiers ou réels. La comparaison de types de données ne doit pas simplement se faire sur l'identification des types, mais sur des critères plus généraux.
- *Types implicites* : le langage ML [Milner 90] admet l'utilisation d'identificateurs dont le type n'est pas déclaré. Lors de l'interprétation de

programmes écrits dans ce langage, le système de types doit calculer le type de ces identificateurs lors de leur utilisation.

- *Abstraction des types de données* : l'implémentation de fonctions indépendantes du type des objets qu'elles manipulent permet leur réutilisation dans des environnement différents.

Pour permettre ces fonctionalités dans un langage de programmation, tout en conservant la propriété de sécurité à l'exécution, un contrôleur de types doit intégrer :

- une méthode d'*inférence de type* pour calculer le type des variables implicitement typées du programme, permettant ainsi la manipulation de variables implicitement typées ;
- une méthode de calcul d'*équivalence de types* ne se basant pas seulement sur l'identification des types, mais sur leur construction structurelle (notion d'équivalence structurelle) et permettant de prendre en compte les types récursifs ;
- une méthode de *conversion* de données entre types ayant des structures équivalentes.

A.4 Inférence de type

L'inférence de type a pour but de calculer, s'il existe, le type d'une expression non typée. Lorsqu'une expression est évaluée, les types de chacune des variables et de ses opérandes doivent être évalués pour savoir s'il sont compatibles avec l'opérateur.

Formellement, le problème de l'inférence de type est exprimé de la façon suivante : étant donnée une expression M, existe–t–il un type A parmi les types pouvant être engendrés par les règles du système de types tel que Type(M) = A ?

L'algorithme d'inférence de type d'une expression exploite la même structure de données que l'algorithme de vérification : Pour typer l'expression M, on parcours l'arbre syntaxique de l'expression M en assignant à chaque nœud un attribut de type calculé conformément aux règles du système de types associé au langage.

Contrairement à l'algorithme de vérification de type, l'algorithme d'inférence parcourt l'arbre syntaxique d'une expression de bas en haut, en calculant un attribut type pour chacun des nœuds de l'expression (en rouge sur l'exemple de la figure 61) en fonction des attributs des fils du nœud.

Pour calculer le type de la variable e, l'algorithme calcule d'abord le type des opérandes de l'expression associée. Le premier opérande est de type entier (2), l'autre est le résultat de l'accès au tableau n, si le type du tableau n est connu alors le système peut décider si l'expression à un type valide (tableau d'entier ou de réels) et affecter un type à la variable e.

Dans le cas où le type de n n'est pas connu, alors une nouvelle variable de type ν est crée dans l'environnement, et le type tableau de ν est affecté à la variable n. Le type de la variable e pourra être effectivement déterminé si l'opérateur + peut être appliqué à un entier et à ν.

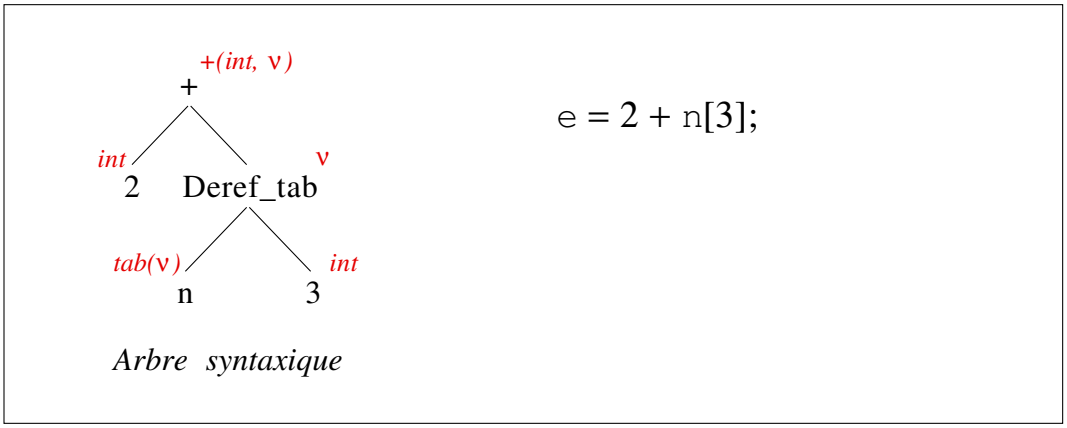


Figure 61 : Inférence du type d'une expression

Le problème de l'inférence se généralise à la reconstruction de type en calculant un type pour chacune des variables d'un programme. Pour cela toutes les variables de types non libres (ne pouvant être renommées) introduites lors de l'inférence doivent être valuées par des expressions de type. Il est montré dans [Cardelli 85] que ce problème est décidable pour les systèmes de types n'incluant pas de quantificateurs.

Lors de la reconstruction de types d'un programme il est fréquent de tester l'équivalence de deux expressions de types. Par exemple lorsqu'une variable à laquelle est associé une expression de type est modifiée, il faut comparer le type de la valeur affectée avec l'expression de type. Comme pour l'abstraction de types de données et pour la surcharge des opérateurs, un système de types permettant le typage implicite s'appuie sur l'équivalence des expressions de types.

A.5 Équivalence de types

Les systèmes de types des langages calculant la validité des types lors de l'abstraction de types de données, du nommage des expressions de type ou de la surcharge des opérateurs utilisent la notion d'équivalence de types de données. L'équivalence peut exister à plusieurs niveaux :

- équivalence de noms : deux types sont équivalents si et seulement si ils portent le même nom. Les compilateurs des langages de programmation interdisant généralement les définitions multiples, deux variables appartenant à des types ayant le même identificateur appartiennent au même type.

  Cette notion d'équivalence est très restrictive car elle implique que les types des variables comparées soient nommés. L'équivalence de deux types « égaux », mais de noms différents ne sera pas relevée.

- équivalence structurale : deux types sont équivalents s'ils ont le même constructeur et les types qui les définissent sont équivalents deux à deux.

  Cette définition permet de mettre en équivalence des types portant des noms différents en se basant sur la correspondance des constructeurs et l'équivalence des composantes du type :

  Soient deux types α et β de constructions respectives $C(\alpha_1,\ldots,\alpha_n)$ et $C'(\beta_1,\ldots,\beta_m)$, où C et C' sont des constructeurs et α_i, β_j les types composants respectivement les types α et β. α et β sont structuralement équivalents si et seulement si :

  1. C est le même constructeur que C'.
  2. $n = m$ et $\forall i \in [1,n]$, α_i est structuralement équivalent à β_i.

L'équivalence structurale peut être implémentée par un algorithme de parcours de graphes dirigés sans cycles, mais cette implémentation ne permet pas de déterminer l'équivalence de types contenant des variables de types ou des définitions récursives.

L'algorithme d'unification permet de calculer l'équivalence structurale d'expressions contenant des variables de types. En conséquence, la récursion dans la définition des types est prise en compte.

Nous détaillons dans la suite un algorithme d'unification utilisant des classes d'équivalence de types pour prendre en compte la circularité du graphe de type.

A.5.1 Algorithme d'unification

L'algorithme d'unification permet de comparer deux types pour trouver une relation d'équivalence. Cet algorithme est une généralisation de l'algorithme d'unification donné dans [Aho 91] aux graphes de types n–aires.

Les types sont définis par des graphes orientés dont les nœuds représentent soit des types soit des variables de types. Le nœuds représentant des types sont étiquetés par le constructeur du type représenté. Les arcs représentent la relation « entre dans le définition de ».

Le principe de l'algorithme repose sur la construction de classes d'équivalence regroupant des nœuds en cours d'unification. Le rôle de ces classes d'équivalence est de ne pas répéter l'unification sur des nœuds participant aux cycles des graphes de types. Chaque classe est représentée de façon unique par un de ses nœuds.

La fonction Fusionner_Classes (n1, n2) permet de créer une nouvelle classe d'équivalence en fusionnant les classes auxquelles appartiennent les nœuds n1 et n2. Si une seule des deux classes est représentée par un nœud qui n'est pas une variable, Fusionner_Classes choisit ce nœud comme représentant de la nouvelle classe. Ainsi, un nœud correspondant à une variable ne peut être représentant d'une classe comprenant des nœuds correspondant à un type construit ou un type de base, ce qui mènerait à unifier à travers cette variable des nœuds non équivalents.

La fonction Trouver_Classe (n) retourne le représentant de la classe d'équivalence à laquelle appartient le nœud n.

Au début de l'unification chaque nœud représente une classe dont il est l'unique élément. Les conditions d'unifications des deux nœuds sont :

- l'appartenance à la même classe,
- la représentation d'un même type de base,
- la représentation de deux types de même constructeur, à la condition de pouvoir unifier les types participant à leur construction (nœuds fils),
- la représentation par l'un des nœuds d'une variable de type.

Algorithme :

```
Unifier (m, n : noeud) : booléen;
début
 s := Trouver_Classe (m);
 t := Trouver_Classe (n);

 si s = t alors retourner vrai;

 sinon si s et t sont des noeuds représentant le même type de
```

```
    base
    alors  retourner vrai;

  sinon si
   s est un noeud de constructeur α et de fils s1...sp et
   t est un noeud de constructeur α et de fils t1...tp
    alors
     Fusionner_Classes (s, t);
     retourner (Unifier (s1, t1) et ... et Unifier (sp, tp));

  sinon si s ou t représente une variable alors
    Fusionner_Classes (s, t);
    retourner vrai;

  sinon retourner faux;
  finsi

fin
```

Avec cet algorithme, on détermine si deux types sont équivalents. Si c'est le cas, toute variable d'un des deux types est utilisable à la place d'une variable de l'autre type tout en assurant la conformité de sa valeur, ce qui préserve la propriété de sécurité lors des surcharges d'opérateurs. Les variables relevant d'un type unifié avec un type cible peuvent également être converties dans ce type cible pour permettre une cœrcition sûre.

L'équivalence de type déterminée par l'algorithme d'unification est également utilisée pour l'inférence de type dans les langages implicitement typés (ML, Lisp).

Bibliographie

[Adobe 91] Adobe Systems Incorporated, *PostScript Language*, Addison−Wesley, Menlo Park, California, novembre 1991.

[Adobe 95] Adobe Systems Incorporated, *Manuel d'utilisaton de FrameMaker, version 5*, Adobe, juin 1995.

[Adobe] Adobe Systems Incorporated, *Portable Document Format (PDF)*, en ligne [URI] http://www.adobe.com.

[Akpotsui 93] E. Akpotsui, *Transformations de types dans les systèmes d'édition de documents structurés*, Thèse de doctorat, Institut National Polytechnique de Grenoble, octobre 1993.

[Akpotsui 97] E. Akpotsui, V. Quint et C. Roisin, ''Type Modelling for Document Transformation in Structured Editing Systems'', *Mathematical and Computer Modelling*, vol. 25(4), pp. 1−19, 1997.

[Aho 91] A. Aho, R. Sethi et J. Ullman, *Compilateurs − Principes, techniques et outils*, Addison−Wesley, Reading, Massachussets USA, 1991.

[AIS 96] *Balise 3 Reference Manual*, AIS S.A, 1996.

[Arnon 93] D.S. Arnon, ''Scrimshaw: A Language for Document Queries and Transformations'', *Electronic Publishing − Origination, Dissemination and Design, proceedings of the EP'94 Conference,* vol. 6, num. 4, pp. 385−396, décembre 1993.

[ATA] ATA, *Air Transport Association of America*, en ligne [URI] http://www.air−transport.org.

[Baru 98] C. Baru, B. Ludäscher, Y. Papakonstantinou, P. Velikhov et V. Vianu, ''Features and Requirements for an XML View Definition Language:Lessons from XML Information Mediation'', *QL'98 − The Query Languages Workshop*, décembre 1998.

[Belaïd 97] A. Belaïd, ''Analyse du document : de l'image à la représentation par les normes de codage'', *Document numérique*, 1(1), pp. 21−38, 1997.

[Bonhomme 96] S. Bonhomme et C. Roisin, ''Interactively Restructuring HTML Documents'', *Computer Network and ISDN Systems*, 28(7−11), pp. 1075−1084, mai 1996.

[Bonhomme 97] S. Bonhomme, V. Quint, H. Richy, C. Roisin et I. Vatton, *Thot – Manuel utilisateur*, 1997,
en ligne [URI] http://www.inrialpes.fr/opera/Thot/Doc/Thotman–F.html.

[Bos 98] B. Bos, H. W. Lie, C. Lilley, I. Jacobs, *Cascading Style Sheets, level 2 – CSS2 Specification*, (REC–CSS2–19980512), World Wide Web Consortium, mai 1998,
en ligne [URI] http://www.w3.org/TR/REC–CSS2.

[Bray 98a] Tim Bray et al., *Extensible Markup Language (XML) 1.0 – W3C Recommendation*, (REC–xml–19980210), World Wide Web Consortium, février 1998,
en ligne [URI] http://www.w3.org/TR/REC–xml.

[Bray 98b] T. Braw et al., *Namespaces in XML – W3C Working Draft*, (WD–xml–names–19980802), World Wide Web Consortium, août 1998,
en ligne [URI] http://www.w3.org/TR/WD–xml–names.

[Burnard 95] L. Burnard, *Text Encoding for Information Interchange – An Introduction to the Text Encoding Initiative*, (TEI J31), Oxford University Computing Services, juillet 1995.

[Cai 90] J. Cai, R. Page et R. Trajan, ''More Efficient Bottom Up Pattern matching in Trees'', *Proc. CAAP 90*, ed. A. Arnold, Lecture Notes in Computer Science (vol. 431), pp. 72–86, Springer–Verlag, 1990.

[Cardelli 85] L. Cardelli et P. Wegner, ''On Understanding Type, Data Abstraction, and Polymorphism'', *Computing Surveys*, 17(4), pp. 471–522, décembre 1985.

[Chahuneau 97] F. Chahuneau, ''XML une voie de convergence entre SGML et HTML'', *Document numérique*, 1(1), pp. 69–74, 1997.

[Clark] J. Clark, *The SP parser*, en ligne [URI] http://www.jclark.com .

[Clark 98] J. Clark et S. Deach, *Extensible Stylesheet Language (XSL) Version 1.0 – W3C Working Draft*, (WD–xsl–19980818), World Wide Web Consortium, August 1998,
en ligne [URI] http://www.w3.org/TR/WD–xsl.

[Claves 95] P. Claves, *Restructuration dynamique de documents structurés*, Mémoire CNAM, CNAM centre régional associé de Grenoble, mars 1995.

[Cole 90] F. Cole et H. Brown, *Editing a Structured Document with Classes*, (report n.73), Computing Laboratory, University of Kent, 1990.

[Cole 92] F. Cole et H. Brown, ''Editing structured Documents–problems and solutions'', *Electronic Publishing––Origination Dissemination and Design*, 5(4), pp. 209–216, décembre 1992.

[Cover] R. Cover, *The SGML/XML Web Page*, Oasis,
[URI] http://www.oasis–open.org/cover.

[Digithome] *Introduction to IDM (Intelligent Document Manager)*, Digithome Electronic Publishing, Dublin, Ireland, 1996,
En ligne [URI] http://www.digithome.com.

[Dougherty 92] D. Dougherty , *Sed & Awk*, O'eilly & Associates, 1992.

[English 96a] J. English, *Cost 2 Reference Manual*, Janvier 1996,
En ligne [URI] http://www.art.com/cost/manual.html.

[English 96b] J. English, *RATFINK A library of RTF output utilities for Tcl Version 0.8*, Mars 1996,
En ligne [URI] http://www.art.com/cost/ratfink/ratfink.html.

[Feng 93] A. Feng et T.Wakayama, ''SIMON: A Grammar–based Transformation System for Structured Documents'', *Electronic Publishing – Origination, Dissemination and Design, proceedings of the EP'94 Conference*, vol. 6 (4), pp. 361 – 372, décembre 1993.

[Fielding 98] R. Fielding et al., *Hypertext Transfer Protocol –– HTTP/1.1*, (draft–ietf–http–v11–spec–rev–04), World Wide Web Consortium, août 1998,
en ligne [adressse URL] http://www.w3.org/Protocols.

[Frilley 89] F. Frilley, *Différenciation d'ensembles structurés*, Doctorat informatique, Université de Paris VII, mars 1989.

[Furuta 88a] R. Furuta, V. Quint et J. André, ''Interactively Editing Structured Documents'', *Electronic Publishing –– Origination, Dissemination and Design*, 1(1), pp. 19–44, avril 1988.

[Furuta 88b] R. Furuta et P.D. Stotts, ''Specifying Structured Document Transformations'', *Document Manipulation and typography*, édité par Cambridge University Press, Proceedings of the International Conference on Electronic Publishing, Document Manipulation and Typography, Nice (France), 20–22 avril 1988.

[Hirvonnen 95] M. Hirvonen et H. Toivonen, *Automatic Transformation of structured Documents*, Rapport de recherche, University of Kent at Canterbury, 1995.

[Hoffmann 82] C.M. Hoffmann et M.J. O'Donnel, ''Pattern Matching in Trees'', *Journal of the Association for Computing Machinery*, 29(1), pp. 68–95, janvier 1982.

[Inso] *Dyna Tag*, Inso Corporation,
en ligne [URI] http://www.inso.com/dynatext/dtagbrief.htm.

[Ion 98] P. Ion et al., *Mathematical Markup Language (MathML) 1.0 Specification – W3C Recommendation*, (REC–MathML–19980407), World Wide Web Consortium, avril 1998,
en ligne [URI] http://www.w3.org/TR/REC–MathML .

[ISO] ISO, *Organisation internationale de normalisation*, Genève,
En ligne [URI] http://www.iso.ch.

[ISO 86] ISO, *Information Processing – Text and Office Systems – Standard Generalized Markup Language (SGML)*, (ISO 8879:1986), International Organization for Standardization, Genève, 1986.

[ISO 94] ISO, *Information Technology – Text and Office Systems – Document Style Semantics and Specification Language (DSSSL)*, (ISO/IEC DIS 10179.2:1994), International Organization for Standardization, Genève, 1994.

[ISO 98a] ISO, *Information technology – Hypermedia/Time–based Structuring Language (HyTime)*, (ISO/IEC JTC1/SC18/WG8 N1920), Organisation internationale de normalisation, Genève, 1998.

[ISO 98b] ISO, *Technologies de l'information –– Jeux de caractères graphiques codés sur un seul octet –– Partie 1: Alphabet latin no. 1*, (ISO/IEC 8859–1:1998), Organisation internationale de normalisation, Genève, 1998.

[Kilpeläinen 92] P. Kilpeläinen, *Tree Matching Problems With Application to Structed Text Databases*, (A–1992–6), Department of computer science, University of Helsinki, Finlande, 1992.

[Kuikka 93] E. Kuikka et M. Pentonnen, ‘‘Transformation of structured documents with the use of grammar’’, *Electronic Publishing – Origination, Dissemination and Design, proceedings of the EP’94 conference*, 6(4), pp. 373–383, décembre 1993.

[Lambolez 95] P–Y. Lambolez, J–P. Queille, J–F.Voidrot et Claude Chrisment, ‘‘EXREP: a generic rewriting tool for textual information extraction’’, *Ingénerie des systèmes d’information*, Volume 3(4), pp. 471 à 487, 1995.

[Lamport 85] L. Lamport, *LaTeX : A document preparation system*, Addisson–Wesley, Reading, Massachussets, 1985.

[Lassila 98] O. Lassila et R.R. Swick, *Resource Description Framework (RDF) Model and Syntax Specification*, (WD–rdf–syntax–19981008), World Wide Web Consortium, octobre 1998,
en ligne [Adresse URL] http://www.w3.org/TR/WD–rdf–syntax.

[Layaida 97] N. Layaïda, *Madeus : système d’édition et de présentation de documents structurés multimédia*, Thèse de doctorat, Université Joseph Fourier – Grenoble, juin 1997.

[Le Maître 95] J. Le Maître, E. Murisasco et M. Rolbert, ‘‘SgmlQL, un langage d’interrogation de documents structurés’’, *Actes des 11èmes journées BDA’95*, Nancy, août 1995.

[Le Maître 98] J. Le Maître, E. Murisasco et M. Rolbert, ‘‘From annotated Corpora to Databases : the SgmlQL language’’, *CSLI lectures notes*, n. 77, janvier 1998.

[Levine 92] J.R. Levine, T. Mason et D. Brown, *Lex & Yacc*, O’Reilly & Associates, 1992.

[Maler 98] E. Maler et S. DeRose, *XML Linking Language (XLink) – W3C Working Draft*, (WD–xlink–19980303), World Wide Web Consortium, Mars 1998,
en ligne [URI] http://www.w3.org/TR/WD–xlink .

[Mamrak 89] 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 1995.

[Megginson 96] D. Megginson, *SGMLS.pm: a perl5 class library for use with the SGMLS and NSGMLS parsers*, University of Ottawa, en ligne [URI] ftp://ftp.funet.fi/pub/languages/perl/CPAN/modules/by−module/SGMLS.

[Microsoft 92] Microsoft Incorporation, *Microsoft Word : Guide de l'utilisateur*, 1992.

[Microsoft 98] Microsoft Incorporation, *Rich Text Format (RTF) Specification*, en ligne, [URI] http://premium.microsoft.com/msdn/library/specs/f15/d11/s6cd3.htm.

[Milner 90] R. Milner, M. tofte et R. Harper, *The Definition of standard ML*, The MIT Press, 1990.

[Pepper] S. Pepper, *The Whirlwind Guide to SGML & XML Tools and Vendors*, Infotek, [URI] http://www.infotek.no/sgmltool/guide.htm .

[Quint 97] V. Quint et I. Vatton, *La boite à outils de Thot*, INRIA, [en ligne] http://www.inrialpes.fr/opera/Thot/Doc/APIman.toc.html.

[Quint 94] V. Quint et I. Vatton, ''Making Structured Documents Active'', *Electronic Pulishing −− Origination, Dissemination and Design*, vol. 7(2), pp. 55−74, juin 1994.

[Quint 98] V. Quint et I. Vatton, ''An Introduction to Amaya'', *World Wide Web Journal*, 2(2), pp. 39−46, Printemps 1997.

[Raggett 98] D. Raggett, A. Le Hors et Ian Jacobs, *HTML 4.0 Specification − W3C Recommendation*, (REC−html40−19980424), World Wide Web Consortium, avril 1998, en ligne [URI] http://www.w3.org/TR/REC−html40.

[Rus 94] D. Rus et K. Summers, ''Using White Space for Automated Document Structuring'', *Proceedings of the Workshop on Principles of Document Processing*, 1994.

[Unicode 96] The Unicode Consortium, *The unicode Standard, Version 2.0*, Addison−Wesley, juillet 1996.

[Schrod 95] J. Schrod et Christine Detig, *STIL − SGML Transformation in Lisp*, août 1995, en ligne [URI] ftp://ftp.th−darmstadt.de/pub/text/sgml/stil/stil.ps.

[Sklar 94] David Sklar, ‘‘Accelerating Conversion to SGML via the Rainbow Format’’, <*TAG*>, 7(1), pp. 4–5, janvier 1994.

[Trombettoni 97] G. Trombettoni, *Algorithmes de maintien de solution par propagation locale pour les systèmes de contraintes*, Thèse de doctorat, Université de Nice–Sophia Antipolis, juin 1997.

[Wood 98] L. Wood et al., *Document Object Model (DOM) Level 1 Specification – Version 1.0 – W3C Proposed Recommendation*, (PR–DOM–Level–1–19980818), World Wide Web Consortium, août 1998, en ligne [URI] http://www.w3.org/TR/PR–DOM–Level–1.

[W3C] The World Wide Web Consortium,
en ligne [URI] http://www.w3.org.

[W3C a] *W3C HTML Validation Service*, World Wide Web Consortium,
en ligne [URI] http://validator.w3.org.

[W3C b] W3C, *W3C XML Software*, World Wide Web Consortium,
en ligne [URI] http://www.w3.org/XML/#software.