



HAL
open science

Environnements et langages de programmation visuels pour le traitement de documents structurés

Emmanuel Pietriga

► **To cite this version:**

Emmanuel Pietriga. Environnements et langages de programmation visuels pour le traitement de documents structurés. Interface homme-machine [cs.HC]. Institut National Polytechnique de Grenoble - INPG, 2002. Français. NNT: . tel-00125472

HAL Id: tel-00125472

<https://theses.hal.science/tel-00125472>

Submitted on 19 Jan 2007

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.

Remerciements

Je tiens à remercier mon directeur de thèse Vincent Quint ainsi que Jean-Yves Vion-Dury qui m'ont offert la possibilité de réaliser ce travail et m'ont encadré durant ces trois années.

Je remercie également les membres du jury :

Christine Collet, professeur à l'Institut National Polytechnique de Grenoble de m'avoir fait l'honneur d'être président de ce jury ;

Paolo Bottoni, professeur à l'Université de Rome "La Sapienza" et Jacques Le Maître, professeur à l'Université de Toulon et du Var d'avoir évalué mon travail ;

Jean-Yves Vion-Dury, chercheur au Xerox Research Centre Europe d'avoir accepté de faire partie de ce jury.

Je remercie particulièrement Vincent Quint, Jean-Yves Vion-Dury et Veronika Lux pour les nombreuses discussions et conseils portant sur ce travail, pour les projets sur lesquels nous avons collaboré, mais aussi pour leurs relectures et commentaires.

Je voudrais adresser un remerciement à tous les membres du Xerox Research Centre Europe, en particulier, François Pacull, Christer Fernström, Damián Arregui, Jean-Pierre Chanod, Isabelle Pené, Tom Zell ainsi que Jérôme Bouat et Alice de Bignicourt qui ont utilisé la librairie XVTM développée dans le cadre de mon travail pour implémenter leurs applications, et enfin Lionel Balme et Sylvie Perrière dont j'ai co-encadré le stage de DESS ; aux membres du projet Opéra et de l'équipe W3C de l'INRIA Rhône-Alpes parmi lesquels se trouvent ou se trouvaient José Kahan, Nabil Layaïda, Cécile Roisin, Irène Vatton et Lionel Villard ; et enfin aux membres de l'équipe W3C du Massachusetts Institute of Technology, pour leur accueil chaleureux, pour m'avoir donné l'occasion de réaliser une partie de mon travail chez eux et pour leurs conseils : tout particulièrement Ralph Swick et Eric Miller, ainsi que Tim Berners-Lee, Art Barstow, Marja-Riitta Koivunen, Alan Kotok, Eric Prud'hommeaux et Daniel J. Weitzner.

Pour finir, je remercie ma compagne Élodie, mes parents, Julien et Julie, Guy et Annie ainsi que tout le reste de ma famille.

Table des matières

| | | |
|----------|--|-----------|
| 1 | Introduction | 11 |
| 1.1 | Motivations et objectifs | 14 |
| 1.2 | Cadre de travail | 16 |
| 1.3 | Plan du mémoire | 17 |
| I | Étude théorique | 19 |
| 2 | Documents structurés | 21 |
| 2.1 | Formats de documents structurés | 24 |
| 2.1.1 | Terminologie | 24 |
| 2.1.2 | Formats | 25 |
| 2.2 | Schémas pour les documents structurés | 29 |
| 2.2.1 | Validation | 29 |
| 2.2.2 | Langages de schéma | 29 |
| 2.2.3 | Modularisation | 33 |
| 2.3 | Les besoins de transformations | 33 |
| 2.4 | Les techniques et les outils de transformation | 35 |
| 2.4.1 | Le langage XSLT | 36 |
| 2.4.2 | Le modèle DOM | 40 |
| 2.4.3 | Le langage Circus | 41 |
| 2.5 | Conclusion | 45 |
| 3 | Langages visuels : État de l'art | 47 |
| 3.1 | Visualisation de structures | 49 |
| 3.1.1 | Principes généraux | 50 |
| 3.1.2 | Visualisation de données structurées | 51 |
| 3.2 | Généralités | 55 |
| 3.2.1 | Définition | 55 |
| 3.2.2 | Historique | 56 |

| | | |
|----------|---|-----------|
| 3.3 | Paradigmes | 57 |
| 3.3.1 | Flot de contrôle | 57 |
| 3.3.2 | Flot de données | 60 |
| 3.3.3 | Systèmes de règles | 64 |
| 3.3.4 | Langages de requête | 65 |
| 3.3.5 | Programmation par démonstration | 68 |
| 3.3.6 | Tableurs | 70 |
| 3.4 | Problèmes connus | 71 |
| 3.4.1 | Facteur d'échelle | 71 |
| 3.4.2 | Facilité d'utilisation | 73 |
| 3.5 | Synthèse | 75 |
| 4 | Programmation visuelle de transformations de documents XML | 77 |
| 4.1 | Présentation générale | 79 |
| 4.1.1 | Les différents types d'outils visuels | 79 |
| 4.1.2 | VXT : un langage visuel spécialisé | 81 |
| 4.1.3 | Des langages cibles multiples | 82 |
| 4.2 | Langage visuel pour la représentation de documents et de classes de documents | 83 |
| 4.2.1 | Intérêt de la représentation visuelle | 83 |
| 4.2.2 | Choix d'une méthode de représentation | 84 |
| 4.3 | Langage de transformation visuel à base de règles | 89 |
| 4.3.1 | Expressions de sélection et d'extraction (<i>VPME</i>) | 89 |
| 4.3.2 | Productions | 95 |
| 4.3.3 | Exemple de transformation | 99 |
| 4.4 | Définition formelle du langage de transformation | 102 |
| 4.4.1 | Grammaire visuelle et syntaxe abstraite | 102 |
| 4.4.2 | <i>VPMEs</i> et productions bien formées | 107 |
| 4.4.3 | Fonction de traduction | 111 |
| 4.4.4 | Exemple de traduction d'une règle de transformation | 123 |
| 4.4.5 | Propriétés de la fonction de traduction | 125 |
| 4.5 | Autres travaux | 127 |
| 4.5.1 | Environnements de développement intégrés | 128 |
| 4.5.2 | Xing | 129 |
| 4.5.3 | XML-GL | 131 |
| 4.5.4 | Approche basée sur les grammaires de graphes | 132 |
| 4.5.5 | Environnement d'édition et fonctionnalités proposées | 133 |
| 4.6 | Conclusion et perspectives | 134 |
| 4.6.1 | Synthèse | 134 |
| 4.6.2 | Évolutions | 135 |

| | |
|---|------------|
| II Applications | 137 |
| 5 Boîte à outils pour le développement d'interfaces graphiques | 139 |
| 5.1 Origines | 142 |
| 5.2 Principes | 143 |
| 5.2.1 Modèle d'objets graphiques | 143 |
| 5.2.2 Interface zoomable | 149 |
| 5.2.3 Animations et continuité perceptuelle | 152 |
| 5.2.4 Interaction contrainte | 155 |
| 5.3 Implémentation | 157 |
| 5.4 Conclusion | 164 |
| 6 Édition visuelle de méta-données | 167 |
| 6.1 Introduction à RDF | 170 |
| 6.1.1 Modèles RDF | 171 |
| 6.1.2 Syntaxe XML | 172 |
| 6.1.3 Définition de vocabulaires | 173 |
| 6.2 IsaViz | 175 |
| 6.2.1 Représentation et navigation dans les modèles | 176 |
| 6.2.2 Édition des modèles | 179 |
| 6.2.3 Architecture | 182 |
| 6.2.4 Évolutions futures | 183 |
| 7 Environnement interactif pour la spécification de programmes VXT | 185 |
| 7.1 Interface | 188 |
| 7.2 Aspects cognitifs | 188 |
| 7.2.1 Réduction de l'effort mental de mémorisation des structures | 188 |
| 7.2.2 Métaphore des filtres visuels | 191 |
| 7.3 Édition des règles | 192 |
| 7.3.1 Construction | 193 |
| 7.3.2 Interaction contrainte | 194 |
| 7.4 Mise au point des programmes | 194 |
| 7.4.1 Exécution des transformations | 194 |
| 7.4.2 Évaluation progressive | 195 |
| 7.5 Conclusion et perspectives | 196 |
| 8 Conclusion | 199 |
| 8.1 Rappel des objectifs | 201 |
| 8.2 Rappel du travail réalisé | 201 |
| 8.2.1 Démarche suivie | 201 |
| 8.2.2 Résultats théoriques | 202 |
| 8.2.3 Résultats pratiques | 204 |
| 8.3 Perspectives | 204 |
| 8.3.1 Développement d'interfaces graphiques | 204 |

| | | |
|----------------------|--|------------|
| 8.3.2 | Programmation visuelle de transformations de documents XML | 205 |
| 8.3.3 | Édition visuelle de modèles RDF | 206 |
| Annexes | | 209 |
| A | Exemples de programmes XSLT et Circus engendrés par VXT | 209 |
| B | Étude formelle d'un langage visuel pour la représentation de DTD | 213 |
| C | Introduction au formalisme des grammaires relationnelles | 223 |
| D | Exemple de vérification de partie droite de règle VXT | 229 |
| E | Démonstrations | 233 |
| E.1 | Lemmes préliminaires | 235 |
| E.2 | Correction syntaxique des expressions engendrées par \mathcal{T}_P | 236 |
| E.3 | Correction syntaxique des expressions engendrées par \mathcal{T}_M | 236 |
| E.4 | Correction syntaxique des expressions engendrées par \mathcal{T}_S | 237 |
| E.5 | Validité des corps de règle engendrés par \mathcal{T}_R | 241 |
| E.6 | Présentation alternative de la syntaxe abstraite \mathcal{AS}_V | 242 |
| E.7 | Complétude de la fonction \mathcal{T}_P | 242 |
| E.8 | Chemins reliant la racine d'une <i>VPME</i> à son nœud contextuel | 244 |
| E.9 | Complétude de la fonction \mathcal{T}_M | 245 |
| E.10 | Complétude de la fonction \mathcal{T}_S | 246 |
| E.11 | Complétude de la fonction \mathcal{T}_R | 247 |
| Bibliographie | | 249 |

Table des figures

| | | |
|------|--|----|
| 2.1 | <i>Exemple de document structuré</i> | 25 |
| 2.2 | <i>Exemple de fragment de document DocBook+MathML</i> | 27 |
| 2.3 | <i>Fragment de DTD DocBook</i> | 30 |
| 2.4 | <i>Fragment de XML Schema DocBook</i> | 31 |
| 2.5 | <i>Expression d'un choix dans XML Schema et dans Schematron</i> | 31 |
| 2.6 | <i>Processus de transformation</i> | 36 |
| 2.7 | <i>Fragment simplifié de la transformation MathMLc2p</i> | 39 |
| 2.8 | <i>Modélisation des structures XML en Circus</i> | 42 |
| | | |
| 3.1 | <i>Dimensions perceptuelles : prix du terrain dans la France de l'Est</i> | 50 |
| 3.2 | <i>Hyperbolic tree</i> | 52 |
| 3.3 | <i>Perspective wall : visualisation de fichiers</i> | 53 |
| 3.4 | <i>Cone tree : visualisation d'une hiérarchie de répertoires</i> | 53 |
| 3.5 | <i>La même scène tridimensionnelle vue sous deux angles différents</i> | 54 |
| 3.6 | <i>VIPR, Représentations statique et dynamique des instructions séquentielles s1, s2 et s3</i> | 58 |
| 3.7 | <i>VIPR, Structures de contrôle</i> | 59 |
| 3.8 | <i>Programme VIPR</i> | 59 |
| 3.9 | <i>Vampire : saisie de a puis b, calcul de c, affichage de c</i> | 60 |
| 3.10 | <i>Exemple de programme LabVIEW</i> | 61 |
| 3.11 | <i>Browser de classe Prograph</i> | 63 |
| 3.12 | <i>Exemple de méthodes Prograph</i> | 63 |
| 3.13 | <i>Représentation en Prograph des opérations d'accès à un attribut et d'appel de méthode</i> | 63 |
| 3.14 | <i>Cocoa wall-climber</i> | 64 |
| 3.15 | <i>Règles Vampire pour l'écoulement du flot de contrôle</i> | 66 |
| 3.16 | <i>Règles Vampire pour les opérations</i> | 67 |
| 3.17 | <i>Requête visuelle : nom et âge des personnes majeures</i> | 67 |
| 3.18 | <i>Toontalk : entraînement d'un robot</i> | 69 |
| 3.19 | <i>Script Pursuit</i> | 70 |
| | | |
| 4.1 | <i>Représentations textuelle et visuelle d'un document XML</i> | 83 |

| | | |
|------|---|-----|
| 4.2 | <i>Exemple de représentation d'une structure arborescente (système de fichiers) sous forme de treemap</i> | 85 |
| 4.3 | <i>Représentation des types de nœud des arbres XML</i> | 87 |
| 4.4 | <i>Visualisation d'une instance de document mail et de la DTD correspondante</i> | 88 |
| 4.5 | <i>Représentation visuelle des propriétés des nœuds de VPME</i> | 91 |
| 4.6 | <i>Exemples de VPMEs simples</i> | 93 |
| 4.7 | <i>Exemple de VPME plus complexe</i> | 93 |
| 4.8 | <i>VPME contenant des instructions d'extraction</i> | 94 |
| 4.9 | <i>Représentation visuelle des types des nœuds de production</i> | 96 |
| 4.10 | <i>Opérateurs de transformation VXT</i> | 96 |
| 4.11 | <i>Règle de transformation t0</i> | 100 |
| 4.12 | <i>Règle de transformation t1</i> | 100 |
| 4.13 | <i>Règle de transformation t2</i> | 101 |
| 4.14 | <i>Règle t0 modifiée (condition de sélection sur author)</i> | 101 |
| 4.15 | <i>Règle t1 modifiée (changement de nœud contextuel)</i> | 101 |
| 4.16 | <i>Exemples de phrases visuelles correctes</i> | 103 |
| 4.17 | <i>Exemples de phrases visuelles incorrectes</i> | 103 |
| 4.18 | <i>Preuve de correction d'une VPME</i> | 109 |
| 4.19 | <i>Opérateurs de transformation VXT</i> | 111 |
| 4.20 | <i>Combinaison des fonctions de traduction</i> | 113 |
| 4.21 | <i>Exemple de règle VXT à traduire</i> | 123 |
| 4.22 | <i>XSLDebugger : un environnement de développement intégré pour XSLT</i> | 128 |
| 4.23 | <i>Xing</i> | 130 |
| 4.24 | <i>XML-GL : représentation XML-GDM d'une DTD</i> | 131 |
| 4.25 | <i>Requête XML-GL</i> | 132 |
| 4.26 | <i>Exemple de règle de réécriture de graphe</i> | 133 |
| 5.1 | <i>Modèle d'objets VAM</i> | 144 |
| 5.2 | <i>Exemples de glyphes du modèle d'objets XVTM</i> | 144 |
| 5.3 | <i>Glyphes translucides</i> | 145 |
| 5.4 | <i>Formes rectangulaires</i> | 146 |
| 5.5 | <i>Courbe quadratique, courbe cubique</i> | 147 |
| 5.6 | <i>Espaces virtuels et caméras</i> | 149 |
| 5.7 | <i>Espaces virtuels multiples</i> | 151 |
| 5.8 | <i>Vue constituée de deux caméras</i> | 152 |
| 5.9 | <i>Schémas d'animation (progression en fonction du temps)</i> | 154 |
| 5.10 | <i>Événements générés par la VAM</i> | 155 |
| 5.11 | <i>Exemple de lexèmes d'interaction générés lors d'un déplacement de souris</i> | 156 |
| 5.12 | <i>Architecture globale de la XVTM</i> | 158 |
| 5.13 | <i>Classes représentant les glyphes</i> | 160 |
| 5.14 | <i>Classes représentant les animations de caméras et de glyphes</i> | 161 |
| 5.15 | <i>Méthode de clipping pour les VPath</i> | 162 |
| 5.16 | <i>Événements générés par les vues XVTM</i> | 163 |
| 5.17 | <i>Autres applications utilisant la XVTM</i> | 165 |

| | | |
|-----|--|-----|
| 6.1 | <i>Modèle RDF pour un roman</i> | 171 |
| 6.2 | <i>Fragment RDF pour la documentation XHTML de XVTM (Syntaxes RDF/XML et Notation³)</i> | 172 |
| 6.3 | <i>Exemple de RDF Schema</i> | 173 |
| 6.4 | <i>IsaViz 1.1 : Interface graphique</i> | 177 |
| 6.5 | <i>IsaViz : Property Browser</i> | 178 |
| 6.6 | <i>Représentation du même modèle par RDFAuthor et IsaViz</i> | 180 |
| 6.7 | <i>Librairies utilisées par IsaViz</i> | 182 |
| 6.8 | <i>Établissement de la correspondance entre modèle mémoire et représentation graphique</i> | 183 |
| 7.1 | <i>Visual XML Transformer : interface graphique</i> | 189 |
| 7.2 | <i>Filtrage visuel analogique : superposition d'un filtre et d'une instance</i> | 192 |
| 7.3 | <i>Règle de transformation VXT</i> | 193 |
| 7.4 | <i>Évaluation progressive d'une VPME</i> | 195 |
| A.1 | <i>Programme XSLT</i> | 211 |
| A.2 | <i>Programme Circus</i> | 212 |
| C.1 | <i>Règle simple d'arrangement</i> | 226 |
| C.2 | <i>Fragment de grammaire pour organigramme</i> | 227 |

Introduction

Les documents électroniques et les traitements qui leurs sont associés sont devenus, avec la démocratisation des ordinateurs dans le cadre du travail comme dans les foyers, un des pôles applicatifs les plus importants de l'informatique. La représentation numérique des documents facilite leur création, leur modification, leur archivage ainsi que leur échange à travers les réseaux de communication. C'est majoritairement dans ce cadre que se déroule l'accès à l'information, notamment à travers le World Wide Web, qui permet non seulement à toute personne de consulter de très nombreuses sources d'information, mais aussi de créer ses propres documents et de les rendre accessibles au public.

L'accès aux ressources du Web, auparavant limité aux ordinateurs de bureaux et aux stations de travail, peut désormais s'effectuer au moyen d'autres terminaux comme les assistants électroniques personnels (PDA, *Personal Digital Assistant*) et les téléphones cellulaires ; il est aussi étendu à d'autres types d'équipements comme les appareils électroménagers. Ces terminaux font des usages variés des ressources du Web en fonction de leurs besoins ; ils ont de plus des capacités différentes, que ce soit au niveau du stockage, du traitement et de l'affichage des documents et des données, ou encore de la connexion ponctuelle ou permanente à un réseau et de la vitesse de transfert. Cette hétérogénéité importante des terminaux s'ajoute à l'hétérogénéité des différents formats et contenus des documents eux-mêmes. Le contenu peut être très riche, intégrant dans ce qui est appelé document multimédia des structures et des données complexes telles que des images, des séquences vidéo, des sons et bien sûr du texte sous différentes formes (de la poésie aux expressions mathématiques). Un effort de standardisation des formats de documents a vu le jour avec SGML (*Standard Generalized Markup Language* [97]) et plus récemment avec XML (*Extensible Markup Language* [35]), plus spécifiquement dédié au Web. Ces formats permettent un échange plus aisé des documents entre applications, ainsi qu'une simplification des traitements. Ils fournissent en effet des modèles pour définir des classes de documents structurés ainsi qu'une syntaxe et des principes de structuration communs, qui permettent l'emploi d'outils génériques durant plusieurs phases de manipulation des documents (analyseur syntaxique, moteur de transformation, etc.). De nombreuses applications ont ainsi adopté XML, simplifiant les traitements appliqués aux documents, souvent spécifiés à l'aide de transformations. Cette simplification a fait apparaître de nouveaux besoins, liés notamment aux échanges de données sur les réseaux entre applications hétérogènes et à la possibilité de séparer dans les documents le contenu de la présentation.

Parallèlement, les évolutions en matière de performances et de capacités d'affichage des ordinateurs ces vingt dernières années ont rendu possible l'apparition des environnements graphiques communément utilisés aujourd'hui, qui proposent des interfaces à base d'icônes, de menus et de fenêtres. Elles ont aussi permis l'exploration de nouvelles voies de recherche dans le domaine des langages de programmation : les environnements et les langages de programmation visuels. Ces derniers sont destinés à la spécification de programmes au moyen d'expressions visuelles, comme par exemple des diagrammes ou des séquences d'icônes formant des phrases visuelles. Ils offrent également des possibilités intéressantes quant à la représentation, la spécification et la mise au point de programmes.

La thèse défendue ici est qu'un rapprochement est possible entre ces deux domaines, prenant la forme d'outils pour la création et la manipulation de documents structurés tirant parti des capacités de représentation et d'interaction mises à disposition par les environnements et les langages de programmation visuels.

1.1 Motivations et objectifs

Le format de documents structurés XML est devenu un standard populaire, adopté par de nombreuses applications dans le cadre de la représentation, du stockage et de l'échange de documents et de données, en particulier sur le Web. XML étant un méta-langage, les applications qui l'adoptent définissent et utilisent en réalité des langages fondés sur XML. Ces langages adoptent la syntaxe, la structure et les mécanismes définis par XML, apportant de leur côté un vocabulaire (fonction du domaine auquel est associé le langage) qui permettra de décrire le contenu des documents de manière exploitable par les applications. Les langages fondés sur XML appartiennent à des domaines variés et spécialisés (présentation de documents, commerce électronique, services Web, représentation de connaissances), et les documents doivent souvent être convertis d'un langage à un autre, ou bien combinés dans des documents utilisant plusieurs langages. Ces conversions et ces combinaisons sont réalisées par le biais de transformations, qui jouent donc un rôle central dans le processus de traitement des documents structurés. Elles sont notamment utilisées pour :

- produire des documents formatés qui peuvent être présentés à des utilisateurs humains, à partir de documents structurés de manière logique et qui ne contiennent pas d'information de présentation (séparation du contenu et de la présentation) ou bien à partir d'information provenant de bases de données, qui de plus en plus proposent une interface XML et, pour certaines, stockent leurs données directement en XML ;
- effectuer des conversions entre différents langages, dans le cadre de l'échange de données à travers le Web entre applications hétérogènes ;
- réorganiser la structure et le contenu des documents en fonction de leur utilisation, mais aussi dans le cadre de l'adaptation de documents en tenant compte des capacités du terminal sur lequel sont utilisés ces documents (problème d'hétérogénéité des terminaux mentionné précédemment) ;
- extraire automatiquement une partie de l'information contenue dans un document, par exemple dans le cadre de la création de méta-données qui permettront d'augmenter la pertinence des résultats de requête faites au moyen d'un moteur de recherche sur le Web.

Il existe de nombreux outils pour la spécification de transformations de documents XML. Ces outils offrent une expressivité et une facilité d'utilisation variant en fonction de leur niveau d'abstraction. Les approches de bas niveau, telles que les interfaces programmatiques associées à des langages de programmation généralistes, sont très expressives mais difficiles à utiliser. Les langages spécialisés, qui cachent une partie de la complexité inhérente aux problèmes de transformation (parcours de la structure, modèle d'exécution) en proposant des abstractions de plus haut niveau, représentent une alternative en général plus simple à utiliser mais moins expressive. Enfin, plusieurs outils proposent des interfaces graphiques au-dessus de ces langages spécialisés, fournissant des fonctionnalités intéressantes telles que des possibilités d'exécution pas-à-pas et une représentation colorée de certains éléments syntaxiques pour améliorer la lecture des documents et des programmes de transformation. Ces derniers sont cependant toujours spécifiés de manière textuelle en utilisant le langage sous-jacent.

Nous pensons qu'il est possible de tirer parti des techniques de visualisation de données et de programmation visuelle dans le cadre de la représentation de la structure des documents XML et de la spécification de transformations portant sur ces structures. Nous espérons ainsi obtenir une simplification

du processus de spécification des transformations. Cette idée de simplification est souvent considérée dans le domaine des langages de programmation visuels comme un moyen d'élargir la base des utilisateurs potentiels du langage (voir les efforts de recherche consacrés aux systèmes de programmation ne requérant pas de l'utilisateur de compétences particulières (*end-user programming*)). Des travaux ont déjà été réalisés dans cette direction, notamment par l'utilisation des techniques de programmation par démonstration pour la création de transformations de documents. Nous ne nous inscrivons pas dans ce courant, puisque nous avons choisi de nous intéresser aux véritables langages de programmation, plus expressifs, dans lesquels les transformations sont spécifiées de manière explicite par l'utilisateur, qui doit par conséquent avoir un minimum de compétences en programmation et de connaissances dans le domaine des documents structurés. Dans ce contexte, nous entendons par l'idée de "simplification" du processus de spécification une diminution des efforts mentaux requis de la part du programmeur et une amélioration du confort de programmation par l'emploi de représentations visuelles de qualité et la mise à disposition de fonctionnalités aidant l'utilisateur dans sa tâche. Ces propriétés peuvent aussi dans une certaine mesure être la source d'une plus grande facilité d'utilisation du langage, élargissant sa base d'utilisateurs potentiels, mais ce n'est pas là un de nos objectifs principaux. Un autre bénéfice attendu de ces propriétés est l'augmentation de la productivité du programmeur, fortement liée aux fonctionnalités évoquées précédemment.

Les manipulations de documents structurés ne se limitent pas aux transformations de documents XML, et nous nous intéressons aussi, dans le cadre des environnements interactifs pour la manipulation de documents, à la création et à l'édition de méta-données pour les documents Web en proposant un environnement d'édition visuel pour RDF (*Resource Description Framework* [165]). Les modèles RDF décrivent principalement des ressources du Web au moyen de propriétés reliant ces ressources entre elles ou à des valeurs. Les modèles représentent donc de l'information (c'est-à-dire des données) à propos des ressources Web, qui sont elles-mêmes des sources d'information (qui prennent la forme de documents ou de données) ; ces données décrivant d'autres données sont désignées par le terme de *méta-données*. Il existe différents types d'outils pour la génération automatique de méta-données, utilisant notamment des transformations de documents. Il est cependant utile de pouvoir créer manuellement des modèles RDF ou bien de pouvoir visualiser et modifier des modèles existants. Ces tâches peuvent être accomplies en éditant les modèles sous leur forme sérialisée, mais ceux-ci se prêtent mal à une représentation textuelle qui reflète difficilement leur structure de graphe et qui demande plus d'efforts de la part de l'utilisateur pour les appréhender. Nous proposons donc un environnement graphique pour la visualisation et l'édition de méta-données RDF représentées sous la forme de diagrammes constitués de nœuds et d'arcs. Il s'agit d'un éditeur spécialisé, capturant au niveau de l'interaction utilisateur les contraintes liées aux spécificités des modèles RDF et offrant ainsi des fonctionnalités qui nous permettront d'assister l'utilisateur dans sa tâche et de garantir une certaine validité des modèles manipulés dans l'environnement.

Pour atteindre ces buts, nous nous sommes fixés des moyens théoriques, mais aussi pratiques, qui prennent la forme de prototypes et d'applications qui nous serviront à expérimenter les idées développées durant l'étude théorique :

- étudier, dans le cadre d'un état de l'art, les techniques de programmation visuelle ainsi que les techniques de visualisation de données structurées ;
- définir un langage de programmation visuel spécialisé dans la transformation de documents XML

- en tenant compte des problèmes souvent rencontrés dans le cadre de la conception d'un langage visuel (ces problèmes seront en partie pris en compte par l'environnement de développement associé au langage). Ce langage, appelé VXT (*Visual XML Transformer*), devra tirer parti de sa spécialisation en proposant des constructions programmatiques et des métaphores fournissant une expressivité importante tout en conservant un niveau de complexité acceptable ;
- effectuer une étude théorique de ce langage en se basant sur des outils formels de manière à se reposer sur des bases solides ;
 - développer une boîte à outils offrant des fonctionnalités fondées sur les concepts étudiés dans l'état de l'art et qui permettra la conception rapide d'interfaces graphiques de qualité ;
 - étudier et implémenter un environnement de développement pour VXT utilisant la boîte à outils mentionnée précédemment, proposant des métaphores adaptées et offrant des fonctionnalités d'édition, d'exécution, d'exportation et de mise au point des programmes de transformation ;
 - développer une application pour la visualisation et l'édition de méta-données RDF utilisant la même boîte à outils et tirant parti de sa spécialisation pour assister l'utilisateur dans sa tâche. Cette application, distribuée publiquement, devra reposer sur une implémentation robuste et nous permettra d'évaluer le niveau de maturité atteint par la boîte à outils graphique.

1.2 Cadre de travail

Ce travail de thèse s'est déroulé au sein de l'équipe DMTT (*Document Models and Transformation Technologies*) du Xerox Research Centre Europe (XRCE) et dans le projet Opéra à l'INRIA Rhône-Alpes, ainsi qu'au W3C (World Wide Web Consortium) durant un séjour de quatre mois au Massachusetts Institute of Technology (MIT).

Le Xerox Research Centre Europe est constitué de plusieurs équipes dont les axes de recherche sont centrés sur ou en rapport avec les traitements documentaires (outils linguistiques, enrichissement automatique de documents en fonction de profils utilisateurs, informatique contextuelle) et la gestion de connaissances (extraction et traitement d'information à partir de textes en rapport avec la bio-informatique, méta-moteurs de recherche).

L'équipe DMTT s'intéresse aux modèles et aux transformations de documents. Elle est depuis peu intégrée à l'aire de recherche *Contextual Computing* et concentre ses efforts sur Circus, un langage de programmation spécialisé dans la transformation de structures et bien adapté au monde XML. Elle travaille aussi sur des composants utilisant ce langage pour l'analyse, la validation et la transformation des documents et des schémas modélisant des classes de documents. Plusieurs composants, à l'élaboration desquels j'ai participé, ont ainsi été développés : un analyseur syntaxique XML incluant un analyseur de DTD, une librairie de composants dédiés à la manipulation de schémas pour les documents structurés, et des transformations XML.

Le projet Opéra s'intéresse lui aussi aux documents électroniques mais avec une approche plus centrée sur les documents multimédia, leur édition et leur adaptation. Il étudie les modèles de documents, qui capturent leur organisation logique, leur contenu, leur présentation graphique, et leur organisation

temporelle. Le projet propose des techniques d'édition et de présentation basées sur ces modèles, et s'intéresse à l'adaptation des présentations multimédia en fonction du contexte (capacités du terminal utilisé, préférences et environnement de l'utilisateur). Ces techniques sont implémentées et expérimentées dans un ensemble de prototypes et d'applications, dont font partie :

- Thot, un outil prenant en compte les modèles pour l'édition interactive de documents structurés,
- Madeus, une application pour l'édition et la présentation de documents multimédia permettant l'agencement temporel des objets,
- LimSee, un éditeur permettant de modifier la structure temporelle des documents multimédia SMIL 1.0.

Le projet Opéra travaille en collaboration avec l'équipe du W3C basée à l'INRIA Rhône-Alpes, qui se concentre sur les formats de documents pour le Web et développe Amaya [75], un outil pour la visualisation et l'édition de documents Web supportant HTML, XHTML, CSS, MathML, SVG et utilisant RDF dans le cadre du système collaboratif d'annotations Annotea [137] (les langages mentionnés ici seront décrits dans le chapitre suivant).

Le World Wide Web Consortium (W3C) a été créé en 1994 ; il est constitué d'environ 500 organisations membres appartenant à l'industrie et à la recherche. Il s'est fixé pour but de conduire le Web à son plein potentiel (*Leading the Web to its Full Potential*), en développant une série de recommandations dans le but de favoriser son évolution et son interopérabilité. Le W3C a ainsi développé entre autres XML, HTML et plus récemment XHTML. Il contribue aux services Web, il s'intéresse aux aspects de sécurité (XML Signature), d'internationalisation et d'accessibilité ainsi qu'à d'autres domaines. C'est aussi l'un des initiateurs de l'effort autour du Web sémantique, et c'est au sein de l'équipe *Semantic Web Advanced Development* que j'ai développé IsaViz, un environnement visuel pour l'édition de modèles RDF.

1.3 Plan du mémoire

Ce mémoire est organisé en deux parties. La première regroupe les aspects théoriques de mon travail (état de l'art, définition et étude formelle du langage VXT), et la deuxième se concentre sur les applications qui résultent de la partie théorique.

Première partie : Étude théorique

Chapitre 2 Documents structurés. Ce premier chapitre traite de la partie du domaine du génie documentaire concernant les documents structurés et leur manipulation. Il commence par introduire les différents modèles et formats de documents électroniques, puis focalise sur XML et sur les technologies qui lui sont associées, comme les langages de schémas et les outils de transformation, après avoir traité des besoins de transformations documentaires. Le but n'est pas de dresser un état de l'art exhaustif des modèles de documents et des techniques de transformation, mais de familiariser le lecteur au domaine en définissant les concepts de base et en présentant les outils de manipulation existants.

Chapitre 3 Langages visuels : état de l'art. Ce chapitre s'intéresse à la visualisation d'information, puis aux langages de programmation visuels. Ces deux domaines présentent en effet des intersections et concernent la représentation et la manipulation de structures (et donc les documents structurés). Le chapitre commence par une introduction aux techniques de visualisation avant d'aborder l'état de l'art des langages de programmation visuels, qui traite des différentes approches, des paradigmes de programmation et des principaux problèmes liés au domaine.

Chapitre 4 Programmation visuelle de transformations de documents XML. Ce chapitre présente la partie théorique de ma contribution, à savoir la définition de VXT, un langage de programmation visuel spécialisé dans la transformation de documents XML. Le chapitre commence par une introduction générale au langage. Elle est suivie d'une étude formelle qui établit certaines propriétés relatives à la traduction de programmes VXT vers un langage textuel cible très utilisé dans le cadre des transformations XML, à savoir XSLT.

Deuxième partie : Applications

Chapitre 5 Boîte à outils pour le développement d'interfaces graphiques. XVTM (*Xerox Visual Transformation Machine*) est une boîte à outils expérimentale permettant la conception d'interfaces zoomables. Elle a été développée dans le cadre de cette thèse et sert de base aux outils décrits dans les deux chapitres suivants. Son but est de faciliter la conception de l'interface d'environnements de visualisation/édition dans lesquels l'utilisateur doit manipuler et animer de grandes quantités d'objets aux formes géométriques pouvant être complexes. Ce chapitre traite des fonctionnalités de la XVTM qui reposent en partie sur les principes étudiés dans le chapitre 3 «Langages visuels : État de l'art», ainsi que de certains détails d'implémentation.

Chapitre 6 Édition visuelle de méta-données. Après une introduction au Web sémantique, au concept de méta-données et à RDF, ce chapitre présente IsaViz, un environnement graphique pour la visualisation et l'édition de modèles RDF, qui repose sur la boîte à outils décrite au chapitre précédent. Ce chapitre montre comment nous avons tiré parti des fonctionnalités offertes par la XVTM pour créer une interface graphique conviviale permettant de manipuler des modèles de taille importante.

Chapitre 7 Environnement interactif pour la spécification de programmes VXT. Ce chapitre décrit l'environnement de développement associé à VXT, basé sur la XVTM, en se concentrant sur les aspects interactifs, c'est-à-dire les fonctionnalités d'édition, d'exécution et de mise au point des programmes, intimement liées au langage et qui représentent une composante importante de ma proposition pour la transformation de documents XML.

Chapitre 8 Conclusion. Ce dernier chapitre résume l'apport de ce travail de thèse et propose des perspectives d'évolution tant au niveau théorique qu'applicatif.

Première partie
Étude théorique

Documents structurés

Ce premier chapitre traite de la partie du domaine du génie documentaire concernant les documents structurés et leur manipulation. Notre but n'est pas de dresser un état de l'art exhaustif des modèles de documents et des techniques de transformation documentaire, mais de familiariser le lecteur au domaine en définissant les concepts de base et les outils de manipulation existants.

La question de ce qui définit un document de manière générale reste ouverte. Le terme a longtemps fait référence aux textes et aux graphiques imprimés. Mais il ne se limite en réalité pas à ceux-ci, qui ne représentent que des catégories de documents. P. Otlet, observe dans son *Traité de documentation* [175] que les documents peuvent être tridimensionnels, faisant ainsi des objets d'art et des artefacts archéologiques des documents potentiels, à condition que la personne les observant obtienne par ce processus des informations. S. Briet va plus loin en définissant un document comme "une preuve à l'appui d'un fait", ou encore comme "Tout indice concret ou symbolique, conservé ou enregistré, aux fins de représenter, de reconstituer ou de prouver un phénomène physique ou intellectuel." [38]. Le caractère documentaire d'un objet est donc fonction de sa place dans une collection organisée et de l'utilisation qui en est faite à un moment donné. La distinction entre document et non document se fonde donc plus sur l'aspect fonctionnel de l'objet que sur son support physique¹. La migration vers des représentations électroniques des documents tend à amplifier cette distinction.

M. Buckland pose la question "Qu'est-ce qu'un document numérique ?" [41] en la considérant comme une spécialisation de la question précédente. La représentation physique des documents électroniques étant à base de *bits*, la distinction ne peut pas s'effectuer sur la base du support (media). Elle a donc tendance à être fondée sur la fonctionnalité de l'entité (on ne peut plus vraiment parler d'objet) plutôt que sur son support physique. Malgré cela, la distinction reste toujours floue. M. Buckman étudie l'exemple des tables donnant des valeurs logarithmiques et des algorithmes permettant de calculer ces mêmes valeurs. Si l'on envisage une version électronique en ligne de ces tables, elles doivent être considérées comme un document. Mais dans ce cas, un algorithme calculant ces valeurs doit aussi être considéré comme un document puisqu'il remplit la même fonction que les tables, et cette classification de l'algorithme en tant que document est beaucoup moins évidente. V. Quint propose une définition plus pragmatique [188] : "Le document désigne un ensemble cohérent et fini d'informations plus ou moins structurées, perceptibles, à usage défini et représenté sur un support concret". Enfin, dans le contexte plus restreint du Web, une définition plus technique de ce qu'est un document est donnée par le *Technical Architecture Group* du W3C. Il s'agit d'un flot de *bits* accompagné d'un type MIME (sans lequel le flot est dépourvu de sens) indiquant au processeur comment interpréter ce flot pour le décomposer par exemple en une séquence de caractères ou une image *bitmap* [63]. En nous fondant sur les définitions précédentes, nous considérons ici comme document électronique tout ensemble fini de données numériques (fichier, flot de *bytes*, etc.) représentant une combinaison de textes, images, scripts, sons ou vidéo.

Notre travail étant centré sur les techniques de programmation visuelle pour la manipulation de documents structurés, nous nous intéressons dans ce chapitre plus particulièrement à ce type de documents. Nous allons étudier l'un des principaux formats de documents structurés, XML (*Extensible Markup Language* [35]), ainsi que les technologies qui lui sont attachées, comme les espaces de noms et les langages

¹Notons tout de même que certaines personnes comme Ranganathan [190] ne sont pas d'accord avec cette définition et limitent les documents aux seuls enregistrements sur des surfaces planes.

de schéma permettant la description de classes de documents. Nous décrirons ensuite les besoins de transformation liés aux documents structurés, avant de détailler les principales solutions permettant ces transformations et la manipulation des documents XML en général.

2.1 Formats de documents structurés

Dans son travail de thèse, S. Bonhomme classe les documents électroniques en fonction de leur modèle et de leur format [27]. Le format est le moyen de stockage physique du document. Il peut être soit binaire, soit textuel. C'est lui qui définit les aspects lexicaux, syntaxiques et sémantiques du document et de son contenu. Le modèle est quant à lui une représentation abstraite du document, qui définit la nature et l'organisation du contenu. Il peut s'agir d'un flot mêlant contenu et informations de présentation (modèle linéaire), ou bien d'une structure hiérarchique d'éléments typés (documents structurés).

Les modèles de documents linéaires regroupent le modèle texte pur (*plain text*) qui ne contient pas d'autre information que le texte brut (par exemple les fichiers ASCII), le modèle de document formaté (LaTeX [143], RTF [158]) dans lequel le contenu textuel est entrecoupé de macros et d'instructions de changement de format (par exemple un changement de taille de la police de caractères), et enfin Postscript. Ce dernier se démarque des précédents puisqu'il s'agit d'un langage de programmation proposant des instructions graphiques, le document étant donc un programme qui doit être interprété afin de produire une image du document.

Les modèles de documents structurés, qui nous intéressent plus spécifiquement ici, représentent quant à eux le contenu sous la forme d'éléments typés organisés en une structure hiérarchique. Cette structure est principalement un arbre, même si dans certains cas elle décrit aussi un graphe comme nous le verrons plus tard (attributs ID/IDREF de XML).

2.1.1 Terminologie

La figure 2.1 est une vue abstraite d'un document structuré DocBook [173] représentant un article parlant des animations dans la XVTM (la boîte à outils pour la conception d'interfaces graphiques zoomables décrite dans le chapitre 5). Les nœuds en italique représentent les nœuds terminaux de type texte. La structure du document, un arbre, est un ensemble de nœuds (*nodes*) connectés par des relations hiérarchiques (père-fils). L'arbre possède un élément racine (*root node*), dans notre exemple de type `article`. Les nœuds directement attachés à un élément père seront appelés ses fils, les fils de ces nœuds ainsi que tous les autres nœuds faisant partie du contenu d'un élément étant appelés ses descendants (les fils sont aussi considérés comme des descendants). Par exemple, l'élément `articleinfo` a deux fils (`author` et `date`) et sept descendants (les deux éléments précédents, ainsi que `firstname`, `surname` et les trois nœuds texte "*Emmanuel*", "*Pietriga*" et "*22 May 2002*"). L'ensemble de ces nœuds forme le contenu de l'élément `articleinfo`.

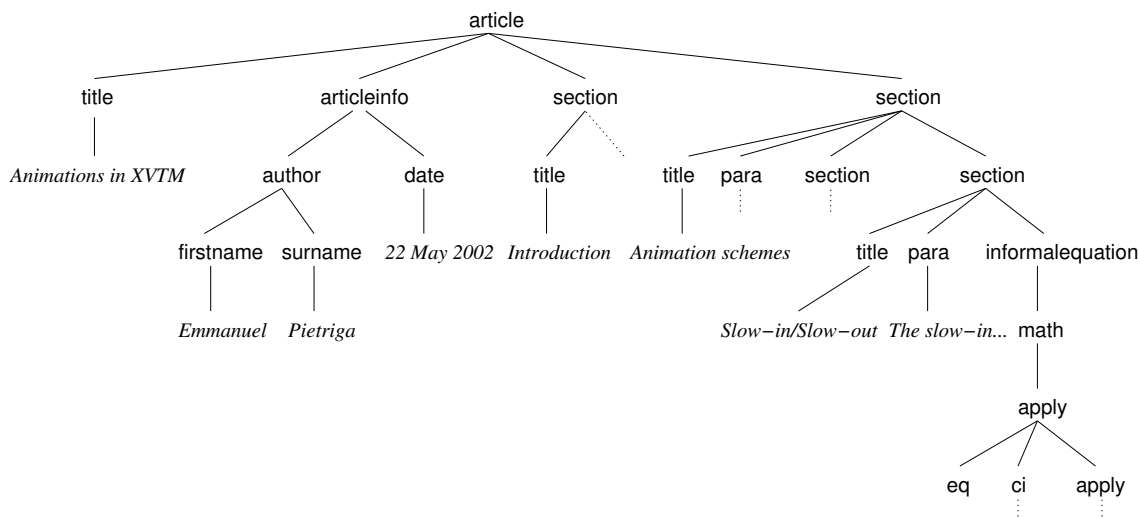


FIG. 2.1 : Exemple de document structuré

Il est possible d'ajouter des informations au niveau de chaque élément, sous la forme d'attributs. Il n'existe pas de règle absolue quant à l'utilisation des attributs ou des éléments², la ligne de conduite préconisée étant cependant de modéliser les données en tant que contenu, c'est-à-dire au moyen d'éléments fils, et les méta-données en tant qu'attributs attachés à l'élément qu'elles décrivent [117] (les méta-données sont des données à propos des données ; leur intérêt et leur utilisation sont abordés plus loin dans ce chapitre ainsi que dans le chapitre 6). Par exemple, nous pouvons attacher à l'élément `article` un attribut de nom `class` dont la valeur est `techreport` et qui décrit le type du document (un rapport technique) en fonction du vocabulaire DocBook.

2.1.2 Formats

Les formats associés à ces modèles structurés, de par leur nature linéaire, nécessitent l'emploi de balises (*tags*) qui représentent la structure logique. Les langages utilisant ces constructions sont appelés des langages de balisage (*markup languages*). Les deux formats standard les plus importants sont SGML (*Standard Generalized Markup Language* [97]), une norme ISO, et XML (*Extensible Markup Language* [35]), développé par le W3C (*World Wide Web Consortium* [226]) et présenté comme un sous-ensemble du précédent, simplifié et optimisé pour le Web (une comparaison très complète des deux langages est proposée par S. Ben Lagha [140]).

Applications et vocabulaires

Tout comme SGML, XML est un méta-langage : il ne propose pas un ensemble de balises à la sémantique bien définie comme c'est le cas pour HTML, mais fournit des normes pour la construction de

²L'information contenue dans un attribut peut aussi être représentée au moyen d'un élément supplémentaire dans le contenu de l'élément auquel l'attribut est attaché.

langages de balisage en imposant des contraintes syntaxiques sur la structure et le contenu des documents utilisant ces langages (syntaxe des balises, caractères autorisés dans le contenu textuel, attributs, etc . . .). Décrits au moyen de XML, ces langages sont appelés des applications XML, l'ensemble des balises et attributs définis par une application étant un vocabulaire. Le nombre d'applications XML est de plus en plus important. Nous donnons ici quelques unes des plus connues :

- DocBook [173], pour la description logique de documents (articles, livres, manuels d'utilisation),
- MathML (Mathematical Markup Language [79]), pour la description d'expressions mathématiques dans le cadre de la communication entre machines et de la présentation,
- RDF (Resource Description Framework [165]), pour la description de méta-données sur le Web,
- SMIL (Synchronized Multimedia Integration Language [80]), pour les présentations audiovisuelles interactives,
- SOAP (Simple Object Access Protocol [166]), pour la communication entre applications et systèmes d'information hétérogènes sur le Web,
- SVG (Scalable Vector Graphics [78]), un langage de dessin vectoriel,
- WSDL (Web Services Description Language [231]), un langage pour la description des services Web,
- XSL (Extensible Stylesheet Language [76]), composé de deux vocabulaires, l'un pour transformer des documents XML (XSLT) sur lequel nous reviendrons par la suite en détail, l'autre pour spécifier le formatage d'un document XML (*XSL Formatting Objects* [1]),
- XHTML [77], une reformulation de HTML en XML.

La figure 2.2 contient la représentation au format XML de l'article introduit dans la figure 2.1. Cet article utilise principalement DocBook, mais aussi MathML pour la description d'une expression mathématique. Les documents XML commencent par un prologue, composé d'un certain nombre de déclarations, comme la version de XML utilisée et le codage des caractères (ici UTF-8). Vient ensuite une déclaration optionnelle de type de document (*doctype declaration*, lignes 2 et 3), qui permet d'identifier à quelle classe appartient le document, et enfin l'arbre XML représentant la structure et le contenu du document (lignes 4 à 63). Les éléments sont délimités par une balise ouvrante et une balise fermante (*opening and closing tags*), leur contenu apparaissant entre ces deux balises. Par exemple, l'élément *author* est délimité par les balises `<author>` et `</author>` lignes 7 et 10 ; les nœuds faisant partie du contenu de cet élément sont déclarés entre ces deux balises. Les éléments vides sont autorisés, auquel cas il est possible de n'utiliser qu'une balise (*empty tag*), comme `<mm1: eq/>` à la ligne 33. Le texte du document apparaît sous la forme de nœuds texte de type #PCDATA pour *Parsed Character Data* (par exemple lignes 27 à 29). Le contenu d'un élément peut mélanger nœuds de type #PCDATA et éléments. Dans ce cas, il s'agit d'un contenu mixte. Enfin, les attributs attachés à un élément sont placés à l'intérieur de la balise ouvrante associée (attribut `class`, ligne 4). Contrairement au contenu des éléments, l'ordre d'apparition des attributs d'un même élément n'a pas d'importance.

Structure de graphe

Comme nous l'avons vu précédemment, les documents XML sont organisés de manière hiérarchique. Leur structure est donc un arbre. Cependant, XML propose des attributs spéciaux, de type ID et IDREF, qui transforment cette structure en un graphe lorsqu'ils sont utilisés. En effet, un attribut de type ID permet de désigner un nœud quelconque de l'arbre de manière unique. Il est alors possible de faire référence

```

1. <?xml version="1.0" encoding="UTF-8" ?>
2. <!DOCTYPE article PUBLIC "-//OASIS//DTD DocBook MathML Module V1.0//EN"
3.     "http://www.oasis-open.org/docbook/xml/mathml/1.0/dbmathml.dtd">
4. <article class="techreport">
5.   <title>Animations in XVTM</title>
6.   <articleinfo>
7.     <author>
8.       <firstname>Emmanuel</firstname>
9.       <surname>Pietriga</surname>
10.    </author>
11.    <date>22 May 2002</date>
12.  </articleinfo>
13.  <section>
14.    <title>Introduction</title>
15.    <para>...</para>
16.  </section>
17.  <section>
18.    <title>Animation schemes</title>
19.    <para>...</para>
20.    <section>
21.      <title>Linear</title>
22.      <para>...</para>
23.    </section>
24.    <section>
25.      <title>Slow-in/Slow-out</title>
26.      <para>
27.        The slow-in/slow-out animation scheme is a
28.        parameterable sigmoid described by the following
29.        equation where n controls the steepness.
30.      </para>
31.      <informalequation>
32.        <mml:math xmlns:mml="http://www.w3.org/1998/Math/MathML">
33.          <mml:apply><mml:eq/>
34.            <mml:ci>y</mml:ci>
35.            <mml:apply><mml:divide/>
36.              <mml:apply><mml:plus/>
37.                <mml:cn>1</mml:cn>
38.                <mml:apply><mml:divide/>
39.                  <mml:apply><mml:arctan/>
40.                    <mml:apply><mml:times/>
41.                      <mml:apply><mml:minus/>
42.                        <mml:apply><mml:times/>
43.                          <mml:ci>x</mml:ci>
44.                          <mml:cn>2</mml:cn>
45.                        </mml:apply>
46.                      <mml:cn>1</mml:cn>
47.                    </mml:apply>
48.                  <mml:ci>n</mml:ci>
49.                </mml:apply>
50.              </mml:apply>
51.            <mml:apply><mml:arctan/>
52.              <mml:ci>n</mml:ci>
53.            </mml:apply>
54.          </mml:math>
55.        </informalequation>
56.      </section>
57.    </section>
58.  </section>
59. </article>

```

FIG. 2.2 : Exemple de fragment de document DocBook+MathML

à ce nœud depuis n'importe quel autre nœud de la structure en utilisant un attribut IDREF (ou IDREFS si l'on veut faire référence à plusieurs nœuds) qui aura la même valeur que l'attribut ID précédent. La structure principale du document reste un arbre, mais ce mécanisme, autorisant l'adressage de n'importe quel nœud attribué d'une ID depuis un élément arbitraire de la structure, permet de l'envisager comme un graphe.

Espaces de noms

Les applications XML peuvent être développées par des individus ou des organisations différents sans concertation ni centralisation. Chaque application XML définit son propre vocabulaire, assignant des noms aux éléments et aux attributs qui le composent (ainsi que des domaines de valeurs pour les attributs). Ainsi, rien n'interdit à deux vocabulaires distincts d'utiliser par exemple un même nom d'élément, ce qui peut poser des problèmes d'ambiguïté lors de la composition dans un document de fragments utilisant plusieurs vocabulaires. Il est cependant important de pouvoir créer des documents composites (i.e. utilisant des vocabulaires XML différents) ; mais il est nécessaire pour cela de disposer d'un mécanisme de désambiguïsation, qui permettra par exemple d'intégrer des dessins SVG, des présentations SMIL et des expressions MathML dans un document XHTML sans créer d'ambiguïté quant à la sémantique des éléments et des attributs, qui est fonction du vocabulaire auquel ils appartiennent.

La sémantique associée à deux éléments de même nom mais provenant de vocabulaires différents n'est pas nécessairement la même. Par exemple, deux organisations séparées peuvent chacune définir un élément `address` dans leurs vocabulaires. Suivant l'organisation, cet élément pourra contenir soit une adresse de courrier électronique (*e-mail*) soit une adresse postale. Pour résoudre ce problème d'ambiguïté, le W3C propose les espaces de noms (*namespaces* [36]). Un espace de noms est défini comme une collection de noms d'éléments et de noms d'attributs associée à une URI (*Uniform Resource Identifier* [22]) qui identifie de manière unique le vocabulaire auxquels appartiennent ces noms. Plusieurs espaces de noms peuvent cohabiter dans un document, permettant ainsi l'utilisation de différents vocabulaires au sein de ce document. Les espaces de noms sont aussi utilisés dans le cadre de XSLT et de RDF comme nous le verrons dans le chapitre 6.

Les espaces de noms sont déclarés en tant qu'attributs d'éléments du document, au moyen du préfixe `xmlns`. Cette déclaration, souvent effectuée dans l'élément racine du document, permet de lier un espace de noms à un préfixe (*prefix binding*). Le préfixe sert de référence dans les éléments et attributs pour identifier à quel espace de noms ils appartiennent. Le document de la figure 2.2 donne un exemple d'utilisation d'espace de noms. Nous déclarons à la ligne 32 l'espace de noms MathML³ que nous lions au préfixe `mm1` (nous pourrions utiliser n'importe quel autre préfixe). La portée de la déclaration d'un espace de noms est limitée à l'élément dans lequel elle est effectuée et à ses descendants (ici de la ligne 32 à la ligne 59). Il est alors possible d'utiliser le préfixe dans l'élément `math` et dans son contenu pour spécifier que les éléments et attributs préfixés par cet espace doivent être interprétés suivant leur définition dans le langage MathML.

³i.e. [http : //www.w3.org/1998/Math/MathML](http://www.w3.org/1998/Math/MathML)

2.2 Schémas pour les documents structurés

2.2.1 Validation

Les documents se conformant aux règles syntaxiques définies par la recommandation XML du W3C [35] sont dit bien formés (*well-formed*). Cette propriété garantit qu'un analyseur (*parser*) XML pourra lire la forme sérialisée du document et en construire une représentation (DOM, SAX, voir section 2.4.2) sans rencontrer d'erreur au niveau syntaxique. La correction syntaxique des documents n'est cependant pas toujours suffisante, et il est intéressant de pouvoir exprimer des contraintes sur leur structure en fonction du vocabulaire utilisé. Ces contraintes peuvent par exemple porter sur le contenu d'un élément (existence d'au moins un élément d'un type donné, interdiction d'un autre type d'élément, ordre des fils, etc.), sur la valeur des attributs et des nœuds texte et sur l'intégrité du document (attributs ID/IDREF(S)). Elles sont exprimées au moyen de langages de schéma (*schema languages*), qui décrivent des classes de documents. Un document sera dit valide par rapport à un schéma s'il se conforme aux contraintes définies dans celui-ci. Dans ce cas, le document sera une instance de la classe de documents définie par le schéma.

Dans le cadre de SGML, la notion de *well-formedness* n'existe pas indépendamment de la validité ; un document est nécessairement associé à une classe de documents et donc à un schéma (une DTD dans la terminologie SGML). Avec XML, le mécanisme de validation d'un document est souvent intégré à l'analyseur syntaxique mentionné précédemment, mais l'étape de validation n'est pas obligatoire. Ainsi, il est possible de ne vérifier que la *well-formedness* du document sans se préoccuper du schéma potentiellement associé.

2.2.2 Langages de schéma

Les premiers travaux relatifs aux schémas datent des années 80, par exemple avec le langage S [188, 189] et les DTD (*Document Type Definition*) de la norme SGML. Les DTD ont été adaptées à XML, mais ne sont qu'une des solutions possibles pour créer un schéma XML. Notre but n'est pas ici de faire une étude comparative des différents langages de schéma, travail déjà effectué par ailleurs [145, 214]. Nous présentons simplement les principaux langages en tentant de les positionner les uns par rapport aux autres en fonction des critères de comparaison identifiés par Lee et Chu [145], parmi lesquels nous retenons :

- la syntaxe du langage de schéma,
- le support pour les espaces de noms,
- les fonctions d'inclusion et d'importation, très importantes dans le cadre de la modularisation (abordée dans la section 2.2.3),
- le support pour les types de données, les types énumérés, ainsi que les valeurs par défaut,
- l'expressivité au niveau du modèle de contenu (*content model*),
- la possibilité d'exprimer des contraintes contextuelles et co-occurentes.

Document Type Definition

Les DTD XML représentent un sous-ensemble des DTD SGML. Leur expressivité est limitée comparée aux autres langages, et ce sont les seules à ne pas employer une syntaxe XML, ce qui implique

| | | | |
|----|-----------|--------------|---|
| 1. | <!ELEMENT | figure | (title,titleabbrev?,(xref mediaobject informatable link ulink |
| 2. | | | blockquote programlisting literallayout)+> |
| 3. | <!ATTLIST | figure | label CDATA #IMPLIED |
| 4. | | float | CDATA "0" |
| 5. | | revisionflag | (changed off added deleted) #IMPLIED> |

FIG. 2.3 : *Fragment de DTD DocBook*

l'utilisation d'un analyseur syntaxique spécifique. Les DTD sont cependant très répandues du fait de l'antériorité de SGML, représentant le standard *de facto* pour la validation XML puisqu'elles sont définies dans la recommandation de ce dernier [35]. Elles sont d'autre part nécessaires pour la définition des entités générales⁴ utilisées dans les instances de documents. Elles fournissent un mécanisme d'inclusion (entités paramètres externes), mais ne gèrent pas les espaces de noms. Le système de types est pauvre (le contenu des nœuds texte (#PCDATA) ne peut pas être typé) mais il est possible d'assigner des valeurs par défaut aux attributs, et de définir les valeurs admissibles par une énumération. Enfin, les contraintes sur le modèle de contenu se limitent à la séquence (,) et au choix (|), sachant qu'il est possible de contraindre la cardinalité des éléments et des constructions précédentes (expressivité limitée à 0 ou 1 (?), exactement 1, 0 ou n (*), au moins 1 (+)).

La figure 2.3 contient un fragment de la DTD DocBook. Ce fragment spécifie (lignes 1 et 2) que l'élément `figure` doit contenir une séquence d'éléments constituée d'exactly un élément `title` suivi d'un élément `titleabbrev` optionnel, puis d'au moins un élément choisi parmi les options `xref` à `literallayout`. Les attributs associés à l'élément `figure` sont les suivants (lignes 3 à 5) : `label`, de type `CDATA`, est optionnel ; `float` a comme valeur par défaut 0 ; `revisionflag` est optionnel, et doit avoir pour valeur une des quatre chaînes de caractères énumérées.

XML Schema

Les *XML Schema* sont une recommandation du W3C depuis mai 2001. Ils sont influencés par les langages de schéma proposés par différentes compagnies, notamment SOX [68] et XDR [101], pour remplacer les DTD jugées trop peu expressives. Contrairement aux DTD, les *XML Schema* adoptent la syntaxe XML. Ils proposent un système de types riche, permettant à l'utilisateur de définir ses propres types de données (structurés ou non), ainsi que des mécanismes comme l'héritage et le sous-typage. Les *XML Schema* offrent donc un modèle de contenu beaucoup plus riche, tant au niveau de la structure qu'au niveau des données, puisque là où les DTD ne proposaient que des nœuds de type `#PCDATA`, les schémas offrent 37 types de données (allant de l'entier positif à la chaîne de caractères modélisée par une expression régulière) qui peuvent de plus servir de base à la définition de nouveaux types par l'utilisateur (par extension ou restriction). L'occurrence des éléments peut être contrôlée de manière très fine, les espaces de noms sont gérés, et des mécanismes d'importation et d'inclusion sont proposés.

⁴Il s'agit de macros d'expansion qui permettent de référencer dans les documents des chaînes de textes qui seront substituées aux références au moment de l'analyse syntaxique.

```

1. <xsd:complexType name='figure'>
2.   <xsd:sequence>
3.     <xsd:group ref='db:formalobject.title.content' minOccurs='0' maxOccurs='1'/>
4.     <xsd:choice minOccurs='1' maxOccurs='unbounded'>
5.       <xsd:group ref='db:figure.mix'/>
6.       <xsd:group ref='db:link.char.class'/>
7.     </xsd:choice>
8.   </xsd:sequence>
9.   <xsd:attribute name='float' type='db:yesorno.attvals' use='default' value='0'/>
10.  <xsd:attributeGroup ref='db:label.attrib'/>
11.  ...
12. </xsd:complexType>

```

FIG. 2.4 : *Fragment de XML Schema DocBook*

| | |
|--|--|
| <pre> 1. <xsd :element name="personne"> 2. <xsd :choice> 3. <xsd :element ref="homme"/> 4. <xsd :element ref="femme"/> 5. </xsd :choice> 6. </xsd :element> </pre> | <pre> 1. <rule context="personne"> 2. <assert test="homme or femme"/> 3. <assert test="count(*)=1"/> 4. </rule> </pre> |
|--|--|

FIG. 2.5 : *Expression d'un choix dans XML Schema et dans Schematron*

La figure 2.4 contient le fragment de *XML Schema* DocBook exprimant les mêmes contraintes que le fragment de DTD précédent. Figure est défini comme un type structuré (ligne 1) ; la cardinalité est exprimée au moyen de deux attributs (`minOccurs` et `maxOccurs`, lignes 3 et 4), et il est possible d'utiliser des groupes d'éléments et d'attributs (lignes 5, 6 et 10), qui d'une certaine manière remplacent les entités paramètres internes des DTD. Le type des attributs peut être modélisé au moyen de n'importe quel type de données de base (ou d'une de ses extensions/restrictions), et il est possible de définir des valeurs par défaut (ligne 9). Les *XML Schema* sont très riches du point de vue de l'expressivité, mais sont plus verbeux que les DTD et plus difficiles à maîtriser (la recommandation du W3C, séparée en deux documents, représente plus de 300 pages [209, 25]).

Schematron

Schematron [129] repose sur le concept de règles. Un schéma Schematron définit un ensemble de motifs structurels (*patterns*) qui décrivent des contraintes, sous forme de règles, à vérifier par les instances de document. Schematron utilise une syntaxe XML. Le processus de validation, illustré avec l'exemple de droite de la figure 2.5, est relativement simple et peut être implémenté en XSLT. Pour chaque règle, l'attribut `context` (une expression XPath, voir section 2.4.1) identifie un ensemble de nœuds dans la structure du document à valider (ici des éléments `personne`) sur lesquels des contraintes doivent être vérifiées. Ces contraintes sont représentées par des assertions exprimées au moyen d'autres expressions XPath, et sont vérifiées pour chaque élément de l'ensemble identifié précédemment par l'attribut `context`. Dans notre exemple, nous vérifions que les éléments `personne` (ligne 1) contiennent exactement un élément fils (ligne 3), et que cet élément fils est de type `homme` ou `femme` (ligne 2).

Cette approche fondée sur les règles est très différente de l'approche grammaticale sur laquelle reposent la plupart des langages de schéma (DTD, *XML Schema*, etc . . .), et permet d'exprimer des types de contraintes différents. L'approche grammaticale définit explicitement et complètement un modèle en restreignant le contenu des éléments, alors que l'approche à base de règles décrit des contraintes à respecter par les instances du modèle. D'une manière générale, les langages reposant sur l'approche grammaticale sont moins expressifs, ne permettant ni d'exprimer des contraintes contextuelles comme "si un élément a un parent de type T alors il doit posséder un attribut A" ni des contraintes co-occurentes (par exemple "si un élément possède un attribut A alors il doit aussi posséder un attribut B"), ce que peut faire Schematron. Cependant, en ce qui concerne les contraintes exprimables par les deux types de langages, les approches grammaticales peuvent être plus simples à utiliser, de par leur sémantique plus élevée. La figure 2.5 illustre la définition du contenu d'un élément comme un simple choix (l'élément personne doit contenir soit un élément homme soit un élément femme). Elle est donnée en *XML Schema* (approche grammaticale) et en Schematron (approche système de règles). Même si *XML Schema* a tendance à être plus verbeux, nous voyons que l'expression du choix, située à un niveau d'abstraction plus élevé, est plus simple dans ce langage. Elle ne nécessite la déclaration que d'un choix et de ses options, alors que la règle Schematron a besoin d'exprimer une contrainte sur le type des options et une contrainte sur leur occurrence.

Autres solutions et combinaison des approches

En réponse au manque d'expressivité des DTD et à la trop grande complexité des *XML Schema*, J. Clark et M. Murata ont développé RELAX-NG [56], produit de la fusion de TREX (*Tree Regular Expressions for XML* [55]) et RELAX (*REgular LAnguage description for XML* [168]). RELAX-NG se veut facile à apprendre et simple à utiliser. Il supporte les espaces de noms, propose des mécanismes d'importation et d'inclusion, et tente de traiter les éléments et les attributs de manière aussi uniforme que possible. RELAX-NG ne définit pas ses propres types de données, mais permet l'utilisation d'un système existant, comme les *datatypes* de *XML Schema*. D'autres propositions ont été faites, comme DCD (*Document Content Description* [34]) ou DSD (*Document Structure Description* [134]) ; celles-ci semblent être plus ou moins abandonnées mais ont influencé la conception des langages précédents.

L'expressivité et la facilité d'utilisation varient beaucoup en fonction du langage de schéma employé et du type de contraintes à modéliser. Comme nous l'avons vu précédemment, Schematron et les *XML Schema* permettent la formulation de contraintes très différentes. La complexité des *XML Schema*, due à leur grande expressivité qui n'est pas toujours nécessaire, rend les tâches de spécification de schéma et de validation plus difficiles, et il peut être plus intéressant d'utiliser RELAX-NG. Les différents langages sont donc complémentaires. Pour ces raisons, il semble intéressant de pouvoir utiliser différents langages de schéma en fonction des besoins. Ainsi, certains validateurs supportent l'incorporation de règles Schematron dans un *XML Schema*. Des discussions sont aussi en cours quant à la possibilité de supporter des règles Schematron dans RELAX-NG, qui rappelons le, permet déjà l'emploi d'un système de types de données externe. Une autre initiative dans cette direction est DSDL (*Document Schema Definition Languages* [120]), un projet en cours de développement qui permettra "d'appliquer dans un même cadre des tâches de validation différentes au même document XML pour parvenir à un résultat de validation plus complet en se reposant sur plusieurs technologies".

2.2.3 Modularisation

Les mécanismes d'inclusion et d'importation mentionnés précédemment sont très importants dans le cadre de la modularisation des schémas. La modularisation d'un schéma consiste à séparer sa définition en différentes parties plus ou moins autonomes, ce qui favorise la lisibilité, la maintenance et la réutilisation des composants. Les applications XML peuvent ainsi être combinées plus facilement, tout en conservant un schéma pour les classes de documents ainsi créées. La modularisation permet aussi de répondre en partie au problème d'adaptation [217] : les périphériques tels que les PDA (*Personal Digital Assistant*) et les téléphones mobiles embarquent, en général, du fait de leurs capacités limitées, des navigateurs légers ne supportant qu'un sous-ensemble des fonctionnalités proposées par un langage. Sont ainsi apparus des profils comme XHTML basic et SMIL basic, définis à partir de composants des applications XHTML 1.0 et SMIL 2.0, ou encore SVG tiny et SVG basic définis à partir de SVG. Le processus d'adaptation des documents nécessite souvent la manipulation des documents XML afin de les rendre compatibles avec le profil cible défini. Ce problème, qui peut être très complexe, est en général traité au moyen de transformations documentaires.

2.3 Les besoins de transformations

Comme nous l'avons vu précédemment, de nombreuses applications ont adopté XML comme format pour représenter, stocker et échanger les documents et les données qu'elles manipulent. La syntaxe et la structure communes à tous ces langages fondés sur XML, et le fait que ce dernier soit un standard non propriétaire, facilitent grandement les traitements appliqués à ces données, permettant l'utilisation d'outils génériques tels que les analyseurs syntaxiques (*parsers*), les validateurs (*validators*), les interfaces programmatiques (*API, Application Program Interface*) et les moteurs de transformation. Ces langages restent cependant différents quant au type d'information qu'ils représentent, qui dépend du domaine auquel est dédiée l'application, et aussi quant à leur manière de structurer cette information. Du fait de l'existence de ces différents langages apparaissent un certain nombre de besoins liés aux transformations de documents structurés. Ces transformations permettent la restructuration des documents, la conversion d'un vocabulaire à un autre, l'extraction d'information à partir de documents sources, ou encore l'enrichissement du contenu.

Les applications des transformations documentaires sont diverses, mais les plus fréquentes sont celles liées à la présentation des documents. XML permet en effet de dissocier le contenu des documents de leur présentation. Les avantages de cette séparation sont multiples : elle permet par exemple de structurer le document source de manière purement logique, sans que des contraintes liées à la présentation ne viennent influencer ni la structure ni le contenu, comme cela peut être le cas en HTML. Elle permet aussi d'associer de multiples présentations à un même document source, par exemple en fonction du contexte d'utilisation du document (filtrage de l'information), des capacités de traitement et d'affichage des machines sur lesquelles le document est visualisé (problème d'adaptation mentionné précédemment), ou encore en fonction du médium choisi par l'utilisateur pour recevoir le document (représentation graphique mais aussi sonore du document). Cependant, cette séparation implique que les éléments et les attributs d'un vocabulaire XML n'ont pas forcément une sémantique de présentation. Ainsi, pour fournir une représentation d'un document utilisant un tel vocabulaire, il est nécessaire de le transformer en un

document utilisant un autre vocabulaire qui pourra être interprété en vue de créer une représentation du document exploitable par un humain, que cette représentation soit visuelle (texte XHTML, graphique SVG, présentation multimédia SMIL, . . .) ou sonore (VoiceXML [81]).

Nous avons donné dans la figure 2.2 un exemple de document DocBook. Ce document ne peut pas être directement présenté à l'utilisateur, puisqu'il ne contient pas d'information de présentation⁵. Il peut par contre être transformé en un document utilisant un ou plusieurs vocabulaires de présentation qui pourra être interprété par des outils de visualisation comme les *browsers* (Amaya, Netscape Navigator, Mozilla, Opera, Internet Explorer) ou les applications de formatage (par exemple FOP [99], qui peut générer des documents PDF à partir de *XSL Formatting Objects*). Notre document utilise deux vocabulaires : DocBook et MathML2.0 Content. L'application MathML est constituée de deux vocabulaires : *MathML 2.0 Content* et *MathML 2.0 Presentation*. Les vocabulaires *Content* et *Presentation* servent à représenter les expressions mathématiques de deux manières différentes : le premier se focalise sur le contenu de l'expression mathématique, en associant aux éléments du vocabulaire une sémantique d'opérations mathématiques précise, compréhensible par des applications telles que Maple [152] ou Mathematica [194], alors que le deuxième vocabulaire se focalise sur leur représentation, en fournissant un ensemble de symboles et de fonctions typographiques permettant de placer des parties de l'expression en indice ou en exposant, de créer des barres de fraction, etc . . . sans donner à ces éléments une sémantique d'opération mathématique. Pour représenter le document de la figure 2.2, il est donc nécessaire de convertir les fragments DocBook, par exemple en XHTML, et les fragments *MathML Content* en *MathML Presentation*. Il existe pour cela des programmes de transformation, comme les *DocBook XSL Stylesheets* [230] et le projet *db2latex (DocBook to LaTeX [45])*, qui inclut notamment *MathMLc2p (MathML Content to Presentation transformation [178])*, une transformation XSLT écrite dans le cadre de ce travail de thèse. Cette transformation permet par exemple d'obtenir, à partir d'une modélisation *MathML Content* abstraite, orientée calcul, de l'expression mathématique utilisée dans la figure 2.2 (lignes 32 à 59), une version orientée vers la présentation (en LaTeX ou en *MathML Presentation*), qui une fois interprétée⁶, permettra d'obtenir un rendu de qualité tel que celui-ci :

$$y = \frac{1 + \frac{\arctan(n(2x - 1))}{\arctan(n)}}{2}$$

Mais la présentation des documents XML n'est pas le seul domaine faisant appel aux transformations. Ainsi, deux applications XML créées par deux organismes indépendants pourront représenter des données similaires en utilisant des jeux d'éléments et d'attributs (i.e. des vocabulaires) différents. Le besoin croissant d'échange des données, à travers le Web, entre applications implique de pouvoir convertir des données d'un langage à un autre. Ces conversions entre applications XML sont réalisées au moyen de transformations.

⁵Même si le vocabulaire DocBook contient quelques instructions liées à la présentation comme l'attribut `align`, il reste principalement un vocabulaire pour la structuration logique des documents.

⁶Le processus d'interprétation d'un document exprimé dans un vocabulaire de présentation et qui permet d'obtenir une représentation concrète du document s'appelle le formatage.

Les transformations sont aussi utilisées pour la création de méta-données pour le Web sémantique (voir le chapitre 6 pour plus de détails). Nous verrons par exemple dans la section 4.4.4 une transformation permettant d'extraire une partie des informations relatives à un document DocBook (lignes 6 à 12 dans la figure 2.2) pour générer des méta-données Dublin Core exprimées en RDF/XML ([69], voir aussi le chapitre 6) qui pourront être incorporées au document XHTML mentionné précédemment. Les méta-données et RDF vont jouer un rôle de plus en plus important dans le cadre des traitements documentaires, par exemple pour optimiser les recherches sur le Web ou encore dans le cadre de l'adaptation de documents avec CC/PP (*Composite Capabilities/Preference Profiles* [133]), un vocabulaire permettant de décrire le profil des appareils électroniques (capacités d'affichage, etc.) et les préférences des utilisateurs.

Enfin, nous verrons dans la section 2.4.1 que l'utilisation de XML dans les bases de données et l'émergence de langages de requête XML tels que XQuery [72] tendent à brouiller la frontière entre documents structurés et données, rendant accessibles de nouvelles sources d'informations XML nécessitant elles aussi des transformations lors de leur manipulation.

2.4 Les techniques et les outils de transformation

Le processus de transformation consiste à fournir un ou plusieurs documents sources à un processeur (ou moteur) de transformation (*transformation engine*) qui extrait une partie ou la totalité du contenu de ces documents, le modifie (recombinaison, enrichissement, filtrage, ...) afin de créer un ou plusieurs nouveaux documents appelés documents cibles (ou documents résultats). La transformation peut être spécifiée au moyen d'un langage de programmation généraliste associé à une API spécifique telle que DOM (voir section 2.4.2) pour faciliter la manipulation des structures, ou bien au moyen d'un langage de programmation spécialisé dans la transformation de documents structurés tel que XSLT [82] ou Omnimark [206].

Dans sa thèse, S. Bonhomme classe les langages selon leur méthode de transformation et de génération du résultat [27]. Il identifie tout d'abord deux classes de transformations :

- les transformations dirigées par la source, qui effectuent un parcours en profondeur d'abord (*depth first traversal*) de la structure source. La transformation est constituée d'un ensemble de règles qui sont évaluées par rapport aux nœuds de cette structure ; les règles qui correspondent aux nœuds sont déclenchées durant le parcours et produisent un fragment du résultat, éventuellement en extrayant et en transformant des parties de la structure source ;
- les transformations dirigées par la cible, ou transformations par requêtes, dans lesquelles la transformation prend la forme d'un document assez proche du document cible mais contenant des instructions d'extraction et de transformation de fragments du document source. La structure source n'est pas parcourue comme dans le cas précédent : au contraire, le processeur effectue un parcours de la structure cible, remplaçant les instructions de transformation rencontrées dans cette structure cible par les résultats associés.

Les approches précédentes, dans lesquelles l'utilisateur spécifie intégralement la transformation, sont dites explicites. Il existe aussi une autre approche des transformations, dite automatique, qui consiste à

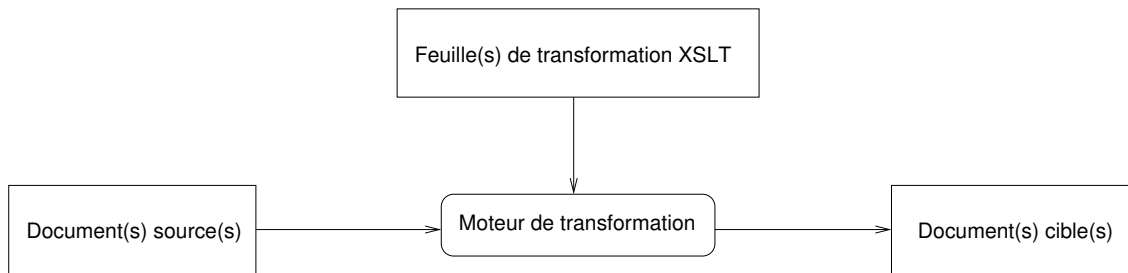


FIG. 2.6 : *Processus de transformation*

analyser des classes (ou instances) de documents sources et cibles, à identifier des relations structurales entre elles, et à en déduire des transformations permettant de convertir les instances de documents de la classe source en instances de la classe cible. Cependant, les règles structurales seules ne sont souvent pas suffisantes pour engendrer des transformations réellement intéressantes, et nécessitent l'intervention de l'utilisateur à certains moments. Il existe donc des approches mixtes, combinant transformations explicites et automatiques, comme celle proposée par S. Bonhomme. Les approches automatiques et mixtes seraient intéressantes dans le cadre d'un environnement pour la création de transformations fondé sur les techniques de programmation par démonstration (voir section 3.3.5). Notre but étant de développer un langage de programmation visuel pour la spécification de transformations de documents XML, dans lequel le programme est spécifié explicitement et non pas au moyen d'exemples, nous nous focalisons sur les approches explicites. Le lecteur intéressé par une description plus détaillée des différentes approches et des différents langages de transformation peut se référer au travail de S. Bonhomme [27]. Nous présentons maintenant trois des solutions textuelles existantes pour la transformation de documents XML, dont deux (XSLT et Circus) seront des cibles possibles pour l'exportation de programmes de transformation spécifiés avec VXT (décrit dans le chapitre 4).

2.4.1 Le langage XSLT

Conçu au départ comme le composant de transformation du langage de formatage XSL [76] du W3C, XSLT [82] est devenu un langage de transformation de documents XML indépendant. Il permet la transformation de documents XML en documents XSL-FO (*XSL Formatting Objects*) mais aussi vers n'importe quel autre vocabulaire XML (XHTML, SVG, etc . . .) ainsi que vers d'autres modèles comme le texte pur ou LaTeX. XSLT est principalement axé sur la transformation de la structure des documents XML, même s'il peut aussi, dans une moindre mesure, manipuler leur contenu. Nous présentons ici une introduction au langage en nous focalisant sur les constructions qui seront utilisées dans le cadre de notre langage visuel.

Modèle d'exécution

Comme le montre la figure 2.6, le processus de transformation consiste à engendrer un nouveau document à partir d'un document source et d'un programme (ou feuille de transformation) XSLT qui dirige

un moteur de transformation dans l'exécution de sa tâche⁷. Le principe général consiste à identifier des parties du document source, au moyen d'expressions de sélection (*pattern-matching expressions*) et de leurs faire correspondre des fragments de documents à engendrer et à insérer dans le document cible. Un programme XSLT est donc constitué d'un ensemble de règles de transformation appelées *template rules*. Ces règles contiennent en partie gauche une expression de sélection modélisant des contraintes sur le contexte (ancêtres et frères) et le contenu (descendants) des nœuds à sélectionner dans le document source ; et en partie droite un ensemble d'instructions (corps de règle) pour l'extraction de données du document source et pour la création de nouveaux éléments dans le document cible. XSLT étant une application XML, les règles de transformation sont représentées par des éléments et leurs attributs. Chaque règle XSLT est associée à un élément `template` dont le contenu représente la partie droite de la règle. La partie gauche est quant à elle représentée par un attribut attaché à l'élément `template`.

XSLT propose deux modes de transformation. Le mode dirigé par la cible se présente comme un squelette de document résultat contenant des instructions pour extraire des données provenant du document source (voir section 2.4). Le mode dirigé par la source consiste à parcourir la structure du document source en profondeur d'abord en essayant d'appliquer les règles de transformation aux nœuds rencontrés. Nous nous intéressons principalement à l'approche dirigée par la source puisque c'est celle retenue dans le cadre de VXT.

Dans les deux modes, le modèle d'exécution est caché à l'utilisateur, qui spécifie simplement les règles de transformation sans se préoccuper de la manière dont elles seront sélectionnées et exécutées. En cas de conflit (expressions de sélection de plusieurs règles reconnaissant un même nœud), le moteur de transformation sélectionne automatiquement la règle la plus spécifique. L'utilisateur peut dans certaines limites influencer sur ce parcours, en assignant manuellement des priorités aux règles ou en définissant des *modes* dans lesquels seules certaines règles peuvent être appliquées. Lorsqu'une règle est sélectionnée, les instructions associées sont exécutées et produisent un fragment de la structure constituant le document résultat. Le document cible est donc engendré par l'instanciation des fragments de résultats associés au corps des règles de transformation, et par le remplacement des instructions XSLT présentes dans ces fragments par le fragment qu'elles engendrent elles-mêmes. D'autres règles peuvent être appelées depuis le corps d'une règle, en général pour procéder à la transformation du contenu de l'élément en cours de manipulation (comme précédemment, le fragment de document produit par cet appel récursif aux règles de transformation remplace l'instruction elle-même). Là aussi il est possible de modifier le parcours par défaut puisque l'utilisateur a la possibilité de spécifier un sous-ensemble du contenu sur lequel seront appliquées les instructions de transformation.

Expressions de sélection

Les expressions de sélection correspondant aux parties gauches des règles sont spécifiées dans l'attribut `match` de cet élément, en utilisant le langage XPath [57]. Une expression XPath identifie un ensemble de nœuds du document source en modélisant des contraintes sur ce nœud (type, nom, attributs), sur son contenu et sur son contexte.

⁷L'instance de document source n'est donc pas transformée au sens propre, mais seulement parcourue et utilisée pour extraire des données.

Les expressions XPath sont construites au moyen des éléments suivants :

- `qname` et `@qname` indiquent respectivement un élément et un attribut dont le nom doit être `qname` ;
- `*` et `@*` indiquent respectivement un élément et un attribut dont le nom n'est pas contraint ;
- `text()` indique un nœud texte (`#PCDATA`) ;
- les axes, pour lesquels nous donnons la syntaxe abrégée quand elle existe (puisque c'est celle que nous retiendrons dans le cadre de l'exportation de programmes VXT vers XSLT) : `/` représente l'axe des fils (à la manière des chemins de fichiers UNIX), `//` représente l'axe des descendants, les axes `ancestor::`, `preceding-sibling::` et `following-sibling::` indiquent que le nœud qu'ils préfixent doit être recherché respectivement parmi les ancêtres, les frères précédents et les frères suivants du nœud courant ;
- et enfin les crochets (`[]`) qui permettent d'ajouter des prédicats aux nœuds de l'expression (les prédicats portant sur un même nœud sont combinés au moyen des opérateurs `and` et `or` et peuvent être rendus négatifs avec la fonction `not`). Ces prédicats doivent être vrais pour qu'un nœud du document source soit sélectionné.

Il existe de nombreuses autres constructions comme les fonctions `starts-with()`, `position()` ou `last()` ainsi que d'autres axes. Nous n'avons présenté ici que les constructions pertinentes par rapport à VXT.

La figure 2.7 est un fragment de programme XSLT contenant des versions simplifiées de certaines des règles de la transformation `MathMLc2p` [178] permettant de transformer l'équation mathématique de l'exemple de document de la figure 2.2 (lignes 31 à 60). Ce fragment contient trois règles de transformation. La première (ligne 5) sélectionne les éléments `math` de l'espace de nom `MathML` à condition qu'ils soient les fils d'un élément `informalequation`. La deuxième (ligne 9) sélectionne tous les éléments `ci`. Enfin, la troisième (ligne 13) sélectionne les éléments `apply` à condition qu'ils contiennent au moins un fils `eq`. Un autre exemple de transformation XSLT complète est donné dans la figure A.1 de l'annexe A.

Corps de règle et nœud contextuel

Le corps d'une règle (contenu de l'élément `xsl:template`) est constitué d'un ensemble d'instructions dont l'exécution génère un fragment de la structure résultat. Les éléments n'appartenant pas à l'espace de nom XSLT (c'est-à-dire ceux dont le nom ne commence pas par `xsl:` dans la figure 2.7) et les nœuds texte sont de simples instructions de création de nouveaux nœuds⁸. Le langage fournit aussi des instructions pour l'extraction et la copie de nœuds sources (`xsl:copy` et `xsl:copy-of`), l'extraction du seul contenu textuel d'un nœud (`xsl:value-of`) et l'appel des règles de transformation sur le contenu d'un nœud (`xsl:apply-templates` et `xsl:call-template`).

Ces instructions sont souvent accompagnées d'un attribut `select` dont la valeur est une expression XPath indiquant sur quel sous-ensemble du contenu (et parfois du contexte) du nœud courant doit être appliquée l'instruction. Cette expression XPath est interprétée de manière relative par rapport au nœud

⁸Le même résultat peut être obtenu au moyen d'instructions de l'espace de nom XSLT comme `xsl:element` et `xsl:attribute` qui sont plus puissantes, permettant notamment la sélection dynamique des valeurs d'attributs.

```

1. <?xml version="1.0" encoding="UTF-8"?>
2. <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
3.             xmlns:mml="http://www.w3.org/1998/Math/MathML">
4.
5.     <xsl:template match="informalequation/mml:math">
6.         <math><xsl:apply-templates select="*" /></math>
7.     </xsl:template>
8.
9.     <xsl:template match="mml:ci">
10.        <mi><xsl:value-of select="." /></mi>
11.    </xsl:template>
12.
13.    <xsl:template match="mml:apply[mml:eq]">
14.        <mrow>
15.            <xsl:for-each select="*[position() != 1 and position() != last()]">
16.                <xsl:apply-templates select="." />
17.                <mo>=</mo>
18.            </xsl:for-each>
19.            <xsl:apply-templates select="*[position()=last()]" />
20.        </mrow>
21.    </xsl:template>
22. </xsl:stylesheet>

```

FIG. 2.7 : *Fragment simplifié de la transformation MathMLc2p*

courant, c'est-à-dire par rapport au nœud source effectivement sélectionné par la règle. Ce nœud, appelé nœud contextuel, est référencé dans les expressions XPath par un point⁹ (.) ou au moyen de l'axe `self`.

Enfin, XSLT offre plusieurs primitives de contrôle, pour effectuer des branchements conditionnels (`xsl:if`, `xsl:choose`), itérer sur un ensemble (`xsl:for-each`), effectuer des classements (`xsl:sort`) et stocker des valeurs (`xsl:variable`, variables dont il n'est pas possible de modifier la valeur après initialisation (*one-time assignment variables*)).

Ainsi, dans la figure 2.7, la première règle, qui sélectionne les éléments `math` ayant comme parent un élément `informalequation`, produit (ligne 6) dans la structure cible, lorsqu'elle est déclenchée, un élément `math`. Le contenu de cet élément est alors engendré par l'application des règles de transformation dont l'expression retient les éléments fils du nœud contextuel (qui est un élément `math` du document source). La deuxième règle donne un exemple de référence au nœud contextuel : elle sélectionne les éléments `ci` et produit un élément `mi` dont le contenu est la valeur textuelle du nœud contextuel (élément `ci` sélectionné). Enfin, la troisième règle produit un élément `mrow` dont le contenu est défini par l'itération sur l'ensemble des éléments fils de `apply` (sauf le premier, qui correspond à l'élément `eq`, et le dernier qui est traité en dehors de la boucle de manière à ne pas avoir un signe = de trop). Les règles de transformation sont appelées sur chacun de ces éléments¹⁰ auxquels est ensuite ajouté un élément `mo`.

⁹Là aussi nous retrouvons la notation des chemins de fichiers UNIX, le double point (..) permettant quant à lui de référencer le nœud parent.

¹⁰Notons que dans le contexte d'une boucle `for-each`, le point (.) dans une expression XPath ne fait plus référence au nœud sélectionné par la règle mais pointe sur la variable d'itération de la boucle.

XSLT et les langages de requête XML

L'utilisation de XML dans le cadre des bases de données et la nécessité de pouvoir extraire et restructurer l'information aussi bien dans les bases de données que dans les bases documentaires ont fait naître le besoin de langages de requête pour XML. Plusieurs langages ont ainsi été proposés, comme Quilt [48], XML-QL [70], XQL [195] et plus récemment SgmlQL combiné à XGQL [150] ou encore DQL [40] (*Document Query Language* à ne pas confondre avec *Daml Query Language* [96]). Un comparatif de certaines de ces propositions a été établi par A. Bonifati et S. Ceri [28]. Le W3C concentre ses efforts sur XQuery [73], langage inspiré des précédents (notamment Quilt) et qui comme XSLT ou DQL utilise XPath pour désigner des fragments dans la structure à interroger.

Les capacités de restructuration de plus en plus puissantes de ces langages et l'utilisation de XPath pour l'identification des fragments, de par le recouvrement [146] qu'elles engendrent du point de vue de l'expressivité, tendent à rendre floue la frontière entre langage de requête et langage de transformation pour XML. De nombreux problèmes peuvent être résolus par les deux types de langages, et il est donc difficile d'établir une classification. Il semble cependant que les langages de requête offrent une expressivité et un degré d'optimisation plus grands au niveau de la requête, en proposant des constructions comme le *select ... from ... where* de SQL et des mécanismes d'agrégation, alors que les langages de transformation offrent des capacités de restructuration des données extraites plus poussées, permettant notamment l'appel des règles de transformation à partir du corps d'autres règles et la manipulation du contenu des documents. Nous positionnons ainsi clairement VXT comme un langage visuel de transformation XML, alors que XML-GL [47, 59] est considéré comme un langage visuel de requête XML (VXT sera comparé aux solutions existantes dans la section 4.5).

2.4.2 Le modèle DOM

Comme nous l'avons dit précédemment, la syntaxe commune à tous les langages fondés sur XML et le fait que ce dernier soit un standard non propriétaire permet la conception et l'utilisation par n'importe qui de composants de traitement comme les analyseurs syntaxiques (*parsers*). Cependant, pour que ces composants soient interchangeables et utilisables dans différents contextes applicatifs, il est nécessaire que leur interface de sortie soit elle-aussi standard, c'est-à-dire que le résultat engendré soit le même quel que soit l'analyseur utilisé. Il existe deux standards principaux définis indépendamment : DOM (*Document Object Model* [71]) et SAX (*Simple API for XML* [98]).

SAX est une interface fondée sur les événements : l'analyseur parcourt le document source et engendre des événements envoyés à l'application cliente. Ces événements sont relatifs à la structure du document source : par exemple, un événement est envoyé à chaque fois que l'analyseur rencontre une balise ouvrante ou une balise fermante, ou bien lorsqu'il rencontre un nœud de type texte. SAX ne fournit donc pas le document source sous la forme d'une structure, mais sous la forme d'un flot d'événements. C'est donc à l'application cliente de construire une représentation plus exploitable de la structure du document (qui peut être partielle) en fonction de ses besoins.

DOM est une interface programmatique normalisée permettant de manipuler la structure des documents XML (ajout, suppression, modification de la structure et du contenu de l'arbre). Contrairement à

SAX, DOM s'appuie sur une représentation complète de la structure du document. Cette approche est plus coûteuse au niveau de la consommation de ressources (mémoire), mais elle offre une représentation intégrale et structurée du document, dans laquelle il est plus facile de naviguer et qui est plus facile à manipuler que des événements SAX. Une représentation DOM est donc plus apte à servir de base pour la transformation des documents qu'un flot d'événements SAX. Il existe des implémentations de DOM dans différents langages de programmation généralistes, comme Java, Python, C ou ECMAScript. La combinaison de DOM avec un langage généraliste offre ainsi une solution très expressive puisqu'elle donne accès à toutes les fonctionnalités du langage sous-jacent et offre à l'utilisateur un contrôle très important sur la structure XML. Mais il s'agit d'une approche de très bas niveau comparée à XSLT, puisqu'elle ne définit pas de modèle de transformation *a priori*. Elle est donc beaucoup plus lourde à mettre en œuvre, puisque l'utilisateur doit prendre en charge le parcours des structures, la création et l'attachement des nouveaux nœuds dans la structure cible, ou encore les tests de sélection des nœuds (les sélecteurs XPath doivent être remplacés par ce qu'ils sont en réalité, une combinaison de parcours et de tests booléens sur le type, le contenu et le contexte (ancêtres et frères) des nœuds de l'arbre).

2.4.3 Le langage Circus

Circus [220, 223, 219, 183] est un langage de programmation spécialisé dans la manipulation de structures de données, et est en ce sens adapté à la manipulation de documents XML, même s'il ne se limite pas uniquement à ce type de structures. Le langage fournit les constructions et opérations communément proposées par les langages généralistes (opérations mathématiques, de manipulation de chaînes, abstractions procédurales, etc.), mais aussi des constructions programmatiques comme le filtrage structurel et un système de types très riche. Ainsi, Circus se positionne, dans le monde XML, à un niveau d'abstraction intermédiaire entre XSLT et des solutions de plus bas niveau telles que DOM combiné à un langage généraliste comme Java. Nous détaillons certaines constructions du langage, que nous utiliserons par la suite puisque Circus est une des deux cibles possibles pour l'exportation des programmes exprimés avec notre langage visuel (VXT).

Système de types

Circus propose un système de types incluant des types de données primitifs (booléens, chaînes de caractères, nombres entiers et flottants) ainsi que des types structurés (multi-ensembles, séquences, tuples, enregistrements et dictionnaires), des types énumérés et des types récursifs. Il est d'autre part possible de définir de nouveaux types par l'union de types existants (opérateur |).

La figure 2.8 contient les déclarations de types permettant de modéliser les structures XML. Le type principal est `XMLTree`, un type récursif formé par une union de types. Il permet de modéliser les éléments non vides attribués (ligne 5), les éléments non vides sans attribut (ligne 6), les éléments vides attribués (ligne 7), les éléments vides sans attributs (ligne 8) et les autres nœuds terminaux (ligne 9), à savoir les nœuds texte (`PCdata` et `Cdata`), les commentaires, et les *processing instructions*. Le champ `label` contient le nom de l'élément, le champ `attr` est un dictionnaire contenant les attributs (couples nom-valeur), et le champ `sub` est une séquence de nœuds représentant le contenu de l'élément. Il sera possible de déclarer des variables typées au moyen du mot-clé `var` (lignes 11 à 13, qui représentent un fragment de l'instance de document `DocBook` de la figure 2.2).

| | | | | |
|-----|-------------|-------------------|---|--|
| 1. | type | Cdata | = | < cdata: String > |
| 2. | type | XMLcomment | = | < comment: String > |
| 3. | type | XMLpi | = | < pi: < target: String , instructions: String > > |
| 4. | type | PCdata | = | String |
| 5. | type | XMLTree | = | < label: String , attr:{ String : String }, sub:[XMLTree] > |
| 6. | | | | < label: String , sub:[XMLTree] > |
| 7. | | | | < label: String , attr:{ String : String } > |
| 8. | | | | < label: String > |
| 9. | | | | Cdata XMLpi XMLcomment PCdata |
| 10. | | | | |
| 11. | var | aDocument:XMLTree | = | < label='article', attr={'class'='techreport'}, sub=[|
| 12. | | | | < label='title', sub=['Animations in XVTM'] >, |
| 13. | | | | < label='section', sub=[...] >]). |

FIG. 2.8 : Modélisation des structures XML en Circus

Ce modèle a fortement évolué depuis l'introduction des références dans le langage ; nous présentons cependant ici la première version car VXT repose pour l'instant sur celle-ci. Nous mentionnons simplement l'extension, dans la nouvelle version [220], du système de types, qui permet de modéliser les contraintes structurales des DTD simplement à l'aide de types Circus¹¹. La vérification de ceux-ci dans un programme se faisant de manière statique, cela signifie que l'utilisation, dans un programme de transformation, de sous-types de XMLTree modélisant les contraintes des DTD des documents source et cible, permet de garantir, s'il n'y a pas d'erreur de compilation du programme, la validité (au sens XML) des documents produits. Cette propriété est jugée de plus en plus importante, permettant de garantir la qualité des traitements appliqués aux documents et des résultats de ces traitements (voir l'introduction de J. Clark à la conférence XML 2001 [54] et le sujet de plusieurs articles présentés au *workshop* Plan-X [229]). Elle représente cependant un problème complexe et n'est offerte ni par XSLT, ni par les solutions utilisant DOM. À notre connaissance, les deux seuls autres langages proposant de modéliser les contraintes structurales des schémas afin de garantir la validité des documents engendrés par la transformation sont XDuce [121] et XSLT0 [210], qui n'est malheureusement qu'un sous-ensemble de XSLT et dont l'intérêt en tant que langage de transformation est limité. Le nouveau système de types Circus offre cette possibilité, dans le cadre d'un ensemble de composants de traitement de schémas appelé D-TaToo (*Document Type Analysis and Transformation Toolkit*), dont font aussi partie des composants de transformation de DTD réalisés dans le cadre de ce travail : l'analyseur de DTD Circus et DTD2HTML, un programme qui facilite la lecture d'une DTD par l'expansion des entités paramètres (internes et externes), l'ajout d'une vue inversée indiquant pour chaque élément la liste des éléments dans lesquels ils sont autorisés, et l'enrichissement du document ainsi obtenu par des liens hypertextes pour faciliter la navigation entre les éléments et attributs définis par la DTD.

Règles et opérations de filtrage

Circus autorise la combinaison d'instructions impératives et d'expressions déclaratives (*imperative and declarative statements*). Les instructions elles-mêmes sont évaluées lors de l'exécution, et retournent

¹¹Les contraintes structurales exprimées dans les DTD sont toutes capturées ; ce n'est par contre pas le cas pour les *XML Schema* qui offrent des contraintes beaucoup plus riches ; la nouvelle version de Circus offre cependant une couverture plus importante que la version initiale.

les valeurs *unit* ou *none* suivant qu'elles réussissent ou non. Ainsi, une règle, de la forme générale $b \rightarrow e$ avec b une expression booléenne et e n'importe quelle expression appartenant au langage, retournera soit *unit* si b est évaluée à vrai, soit *none* si b est évaluée à faux ; une instruction impérative retournera quant à elle toujours *unit*. La partie gauche des règles Circus peut contenir une expression de filtrage à la place de l'expression booléenne. Ces règles sont de la forme

$$e_1 \# f \rightarrow e_2$$

avec e_1 n'importe quelle expression du langage retournant une valeur, f un filtre et e_2 n'importe quelle expression appartenant au langage. Le filtre f est appliqué à e_1 , appelé le sujet de l'expression de filtrage. Si l'application du filtre sur la valeur retournée par e_1 réussit, e_2 est évaluée et sa valeur est retournée ; si elle échoue, la valeur *none* est retournée¹².

Les filtres sont des expressions modélisant des contraintes de sélection sur la structure et le contenu des sujets, mais aussi des instructions d'extraction d'une partie ou de la totalité du contenu de ces sujets. Les filtres adoptent une syntaxe proche de celle des constructeurs de valeurs. Nous adopterons une approche similaire dans le cadre de VXT, en proposant des expressions de filtrage (les *VPMEs*, *Visual Pattern Matching Expressions*) visuellement proches des structures qu'elles sélectionnent. La construction des filtres s'en trouve ainsi facilitée. Nous donnons ici les principales constructions utilisées dans les filtres. $?$ est un test d'existence, $?x$ vérifie l'existence dans le sujet d'une donnée du type de la variable x et stocke¹³ cette donnée dans x , $\%e$ évalue l'expression e et vérifie que le sujet a la même valeur. Les filtres structurés utilisent les mêmes constructions syntaxiques que les structures de données ($[]$ pour les séquences, $' '$ pour les chaînes de caractères, $\{ \}$ pour les multi-ensembles et $\{ = \}$ pour les dictionnaires). Ces filtres basiques et structurés peuvent être combinés au moyen de l'opérateur d'agrégation $++$ considéré comme le dual de l'opérateur de concaténation $+$ et qui s'applique comme lui aux chaînes de caractères, séquences, multi-ensembles et dictionnaires. L'opérateur $\&\&x$ a la même sémantique mais ne tient pas compte de l'ordre des éléments lors de la sélection. Différents filtres peuvent d'autre part être combinés dans une même règle au moyen des opérateurs booléens *and* et *or*. Nous donnons quelques exemples concrets d'expressions de filtrage :

- `'circus' # ?x++%'rc'++?` réussit à condition que la variable x soit de type chaîne (*String*) et lui assigne la valeur ci ,
- `[1, 2, 3] # [%1]++?x` réussit et assigne la séquence `[2, 3]` à x ,
- `<label = 'title', sub = ['Animations in XVTM']> # <label = '%title', sub = ?children>` réussit à condition que *children* soit de type (ou bien un sous-type de) `[XMLTree]` et stocke le contenu de l'élément `title` dans la variable *children*,
- `<label = 'section', sub = [(label = 'para', sub = ['sometext'])]> # <label = '%section', sub = ? + + [(label = '%title')] + + ?` échoue car l'élément `section` ne contient pas d'élément `title`,
- `<label = 'articleinfo', sub = [(label = 'date', sub = ['22 May 2002'])]> # <label = '%articleinfo'> and ?x` réussit si la variable x est de type (ou bien un sous-type de) `XMLTree` et stocke le sujet complet dans cette variable.

¹²Retourner la valeur de e_2 correspond donc informellement à retourner *unit*.

¹³Ces variables sont ensuite accessibles dans la partie droite de la règle.

Séquences d'instructions, systèmes d'action et boucles itératives

Les règles, comme les instructions impératives, peuvent être mises en séquence au moyen du séparateur d'instructions noté par un point-virgule (;). Elles peuvent aussi être cascadées :

$$e_1 \# f_1 \rightarrow e_2 \# f_2 \rightarrow e_3$$

ou être combinées dans des systèmes d'action, qui sont plus ou moins équivalents aux constructions *switch* des langages comme Java ou C. Les systèmes d'action sont de la forme

$$[[e_1, \dots, e_n]]$$

où e_i représente une expression appartenant au langage, et peut notamment être une règle de filtrage, de la forme

$$e_{i1} \# f \rightarrow e_{i2}$$

L'exécution d'un système d'actions $[[e_1, \dots, e_n]]$ consiste alors à évaluer les expressions e_i dans l'ordre jusqu'à la première ne retournant pas *none*, c'est-à-dire la première règle qui réussit.

Circus propose aussi plusieurs constructions permettant d'exprimer des boucles itératives. Nous utiliserons dans le cadre de VXT la boucle *for*, notée

$$\mathbf{for} \textit{el} \textit{ in } \textit{aSet} \mathbf{do} (\textit{insts})$$

où *el* désigne la variable d'itération sur l'ensemble *aSet* parcouru du premier au dernier élément (il peut s'agir d'une séquence ou d'un multi-ensemble), et *insts* un ensemble d'instructions impératives et déclaratives éventuellement combinées en un système d'actions qui est exécuté à chaque itération.

Machines abstraites polymorphes et λ -fonctions

Le langage offre deux sortes d'abstractions procédurales : les λ -fonctions et les PAM (*Polymorphic Abstract Machines*). Les λ -fonctions sont de la forme

$$\mathbf{const} \textit{aFunction} : t_1 \rightarrow t_2 = \mathbf{lambda} \textit{x} : t_1. \textit{insts}$$

où *aFunction* est le nom de la fonction, *x* le paramètre d'entrée de type t_1 et *insts* un ensemble d'instructions impératives et déclaratives. Cet ensemble d'instructions doit s'évaluer en une valeur de type t_2 , qui sera retournée par la fonction.

Enfin, les PAM sont des sortes de procédures ayant un paramètre d'entrée et un paramètre de sortie, tous les deux typés. Circus fournit un mécanisme innovant pour la composition de PAM que nous ne détaillons pas ici car nous n'en ferons pas usage dans le cadre de VXT (voir [223] pour une description des PAM et du mécanisme de composition).

Le fragment de code ci-dessous donne un équivalent en Circus de la deuxième règle de la transformation XSLT présentée dans la figure 2.7 (lignes 9 à 11) : *node* est une variable de type *XMLTree* contenant le nœud par rapport auquel sont évaluées les règles de transformation, et la variable *res* contient la structure résultat. La fonction *XValue* est équivalente à l’instruction *value-of* de XSLT. Un exemple de programme Circus complet, contenant la définition de la fonction *XValue*, est donné dans la figure A.2 de l’annexe A (cet exemple correspond à la transformation développée dans le chapitre 4).

```
node # ⟨ label=%'mml:ci' ⟩ → res := res + [⟨ label='mml:mi', sub=XValue(node.sub) ⟩]
```

2.5 Conclusion

Le langage XML est devenu un standard *de facto* pour la représentation de documents structurés et de manière plus générale pour la modélisation et l’échange de données à travers le Web. De nombreuses applications, telles que XHTML, RDF, SOAP, XSL-FO, RSS ou DocBook l’ont adopté, rendant ainsi possible leur manipulation au moyen d’outils génériques (analyse syntaxique et sérialisation, validation, transformation, présentation).

Les technologies associées améliorent la qualité du processus de traitement documentaire, par exemple grâce aux classes de documents définies par des schémas qui, combinées à des phases de vérification, garantissent la validité structurale (et dans une moindre mesure du contenu grâce au typage) des documents. Elles facilitent aussi la combinaison de fragments de documents appartenant à différents vocabulaires (espaces de noms), et la transformation de documents d’un vocabulaire à un autre, que ce soit pour convertir des données dans le cadre d’un échange entre applications hétérogènes, pour fournir une représentation formatée d’un document structuré de manière logique, ou encore pour extraire des informations et générer des méta-données qui seront utiles dans le cadre du Web sémantique (voir chapitre 6).

Il existe différents types de solutions pour la spécification de transformations de documents XML, dont l’expressivité et la facilité d’utilisation varient en fonction du niveau d’abstraction. Les approches de bas niveau telles que DOM sont très puissantes mais fastidieuses à mettre en œuvre ; au contraire, les langages spécialisés de haut niveau comme XSLT sont relativement simples d’utilisation, mais sont par contre beaucoup plus limités quant aux transformations qu’ils peuvent exprimer. Ainsi, le besoin de langages situés à un niveau d’abstraction intermédiaire, tels que Circus, se fait de plus en plus ressentir. Toutes ces solutions reposent cependant sur des langages de programmation textuels. Nous verrons dans le chapitre 4 qu’il existe des applications proposant un environnement de développement visuel au-dessus des langages de haut niveau (XSLT), mais aussi des langages de programmation visuels spécialisés dans la transformation de documents XML, dont VXT fait partie.

Langages visuels : État de l'art

Nous nous intéressons dans ce chapitre aux langages visuels, et plus particulièrement aux langages de programmation visuels. Les langages visuels se différencient des langages textuels par deux caractéristiques principales :

- Le système de représentation des signes de base autorise l'utilisation d'autres dimensions perceptuelles (paramètres physiques perçus directement par l'observateur humain) que la forme pour coder un élément de cet alphabet, telles que la couleur et la taille, contrairement aux langages textuels qui n'utilisent que la forme des glyphes pour différencier les éléments. Ces derniers peuvent donc être les glyphes de caractères mais aussi des formes géométriques comme les cercles, les triangles ou des entités plus complexes comme des icônes.
- La multi-dimensionnalité : dans les langages textuels, il existe des contraintes fortes sur la manière de placer les éléments de l'alphabet pour former les unités lexicales (mots) et les expressions (phrases) du langage. Ils doivent être juxtaposés suivant une direction donnée, formant ainsi une chaîne de caractères, donc un flot unidimensionnel. Les langages visuels s'affranchissent de cette contrainte (et il s'agit même, comme nous le verrons plus loin, de la différence essentielle entre langages visuels et textuels), permettant ainsi la création d'expressions utilisant plus d'une dimension pour coder la sémantique.

Il ne faut pas confondre les langages de programmation visuels avec les langages de programmation pour le traitement d'information visuelle, qui sont souvent des langages textuels permettant la manipulation d'information ayant une représentation visuelle inhérente. Un langage de programmation visuel se définit, de manière similaire à un langage de programmation textuel, comme un langage dont le vocabulaire, la sémantique et les abstractions permettent de spécifier des algorithmes.

Ce domaine de recherche présente des intersections avec la visualisation d'information, plus particulièrement de données structurées. Les programmes visuels représentent eux-mêmes des informations structurées, et certaines techniques de visualisation peuvent avoir un intérêt par rapport aux langages visuels. Nous nous intéressons donc dans un premier temps aux techniques de visualisation avant d'aborder, dans l'état de l'art des langages de programmation visuels proprement dit, les différents paradigmes de programmation et les principaux problèmes liés à la programmation visuelle.

3.1 Visualisation de structures

La visualisation d'information consiste à représenter des données sous forme graphique en utilisant au maximum les capacités de perception visuelle des utilisateurs et les différentes dimensions perceptuelles fournies par les représentations graphiques. Ceci de manière à afficher un maximum de données dans un espace restreint tout en conservant une bonne lisibilité.

La représentation graphique permet de refléter les propriétés structurales des données par l'arrangement spatial des objets, ou encore de refléter les propriétés intrinsèques de ces données par des attributs visuels comme la couleur, la taille ou la forme des objets graphiques. Nous nous intéressons ici plus particulièrement à la visualisation d'information structurée, sous forme de hiérarchie ou de graphe.

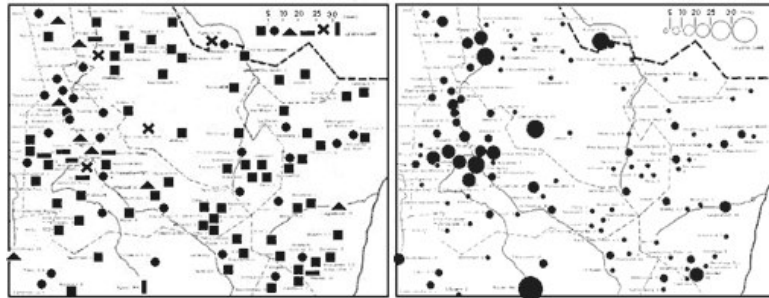


FIG. 3.1 : Dimensions perceptuelles : prix du terrain dans la France de l'Est

3.1.1 Principes généraux

Parmi les travaux importants reliés à la représentation graphique de données, on peut citer ceux de Tufte [211] et la sémiologie graphique de Bertin [24]. Cette dernière définit les propriétés spécifiques de la représentation graphique et permet de déterminer la meilleure transcription d'une information en faisant apparaître des relations de ressemblance et d'ordre. Elle s'applique à de nombreux domaines, comme la cartographie, la représentation de réseaux ou encore de données numériques sous formes de graphiques. Nous retenons dans le cadre de notre travail plusieurs choses. Tout d'abord, sa définition des fonctions de l'image : instrument de traitement de l'information (permettant de découvrir et de mesurer des corrélations), inventaire qui remplace la mémoire et message qui peut lui-même être mémorisé (plus facilement que les données qu'il représente). Nous retenons aussi l'identification de sept variables visuelles orthogonales : position, taille, orientation, forme, couleur, texture et intensité. Ces dimensions sont perçues simultanément par l'œil, mais traitées différemment. Il en résulte qu'en fonction de leurs propriétés, certaines variables sont mieux adaptées que d'autres pour représenter certaines données et leurs relations, en tenant aussi compte du fait que la représentation de ces données n'est pas une fin en soi, mais a une fonction, comme par exemple permettre à l'observateur d'effectuer des comparaisons. Les relations sont l'ordonnancement (*ceci est avant cela*), la proportionnalité (*ceci est n fois cela*), l'associativité (*ceci peut être vu semblable à cela*) et la sélectivité (*ceci est différent de cela*). Si l'on prend la figure 3.1, une carte représentant le prix des terrains dans l'Est de la France, il est évident que la version de droite, utilisant la taille comme variable associée au prix, est plus facilement exploitable, dans le cas où l'on désire comparer les valeurs (ordonnancement, proportionnalité), que la version de gauche, qui utilise la forme. Par contre, dans le cas où l'on voudrait uniquement identifier des groupes de prix, l'avantage de la représentation de droite n'est pas si évident.

Au choix des dimensions perceptuelles adéquates pour représenter les données vient s'ajouter le problème de la visualisation de grandes quantités d'information. D'après F. Vernier [215], une grande quantité d'information est constituée d'un grand nombre d'éléments, chacun possédant des attributs variés. Ces attributs peuvent être de nature différente, et les éléments peuvent constituer une simple collection ou faire partie d'un ensemble plus structuré. Plusieurs outils de visualisation sont proposés par Roberston [112] et Wittenburg [9].

3.1.2 Visualisation de données structurées

Toujours d'après Bertin, les données peuvent être représentées soit par leur valeur (numérique, catégorie, ...), soit par leur structure (relations, contraintes, ...). L. Tweedie [212] étend cette taxonomie en ajoutant la notion de méta-données dérivées des valeurs/structures de départ par transformation. Dans le cadre de ce travail, nous nous concentrons sur les données structurées sous forme d'arbre ou de graphe.

Nous passons rapidement sur les problèmes d'agencement (*layout*), domaine très complexe, pour nous concentrer sur l'aspect interface homme-machine de la visualisation (métaphores et techniques de rendu, navigation dans les structures). On retiendra simplement l'existence de bibliothèques telles que GraphViz (voir le Chapitre 6.2.1), qui fournissent à partir d'une représentation abstraite d'un graphe les informations géométriques permettant de le dessiner de manière à optimiser sa lecture, en minimisant par exemple l'effet spaghetti (croisements trop nombreux des chemins reliant les noeuds, voir [125]). On peut aussi citer les travaux de A. Quigley qui propose FADE [187]. Le graphe est vu comme un système physique virtuel dans lequel les noeuds sont considérés comme des corps sur lesquels s'appliquent des forces (les relations entre noeuds sont vues comme des ressorts). Enfin, il existe des techniques permettant de réduire la complexité visuelle des arbres et des graphes [132] :

- *ghosting* : modifier l'apparence de certains noeuds et arcs de manière à les rendre moins visibles (en leur assignant par exemple une couleur proche de celle de l'arrière-plan),
- *hiding* : ne pas afficher certains noeuds et arcs,
- *grouping* : grouper un sous-ensemble des noeuds du graphe en un méta-noeud.

Les deux dernières techniques impliquent une modification de la structure du graphe affiché et ne sont donc pas toujours appropriées. De plus, elles nécessitent une analyse sémantique du graphe afin d'être réellement pertinentes. Les techniques spécifiquement dédiées aux structures d'arbres (*node-link diagrams* et *treemaps*) sont étudiées au chapitre 4.

La taille et la résolution très limitées des écrans ne permet souvent pas d'afficher l'ensemble des éléments de la structure tout en gardant un niveau de détail adéquat pour les manipuler. Pourtant, les deux types de vues ont leur intérêt. La vue d'ensemble (contexte) permet d'avoir une compréhension globale de la structure et d'identifier rapidement les zones d'intérêt. La vue détaillée (focus) ne représente que la zone d'intérêt courant et affiche des informations plus complètes sur les éléments de cette zone. Différentes techniques permettent de représenter simultanément le focus et le contexte. La première solution consiste à offrir deux vues sur les données. La fenêtre principale affiche une vue détaillée d'un sous-ensemble des données (région) alors qu'une autre fenêtre, de taille réduite, présente une vue globale ainsi qu'un rectangle couplé à la vue principale délimitant la région observée dans la vue détaillée. Cette seconde fenêtre est parfois appelée *vue radar*. On la retrouve dans différents logiciels commerciaux, comme les jeux de stratégie ou encore dans le logiciel de manipulation d'images Adobe Photoshop. Une solution dérivée utilise la possibilité de superposer (*multi-layering*) des vues semi-transparentes dans une même fenêtre tout en conservant une bonne lisibilité [147]. L'utilisateur n'a qu'à changer la couche (*layer*) active suivant qu'il désire effectuer des actions dans la vue globale ou dans la vue détaillée. Cette solution a l'avantage d'offrir un plus grand espace d'affichage pour la vue globale, mais elle fait apparaître le problème d'interférence visuelle : ce phénomène se produit quand les objets des différentes couches se recouvrent partiellement et ont des représentations graphiques proches (par exemple au niveau de la taille des objets et des couleurs utilisées), ce qui rend difficile leur différenciation sur le plan visuel.

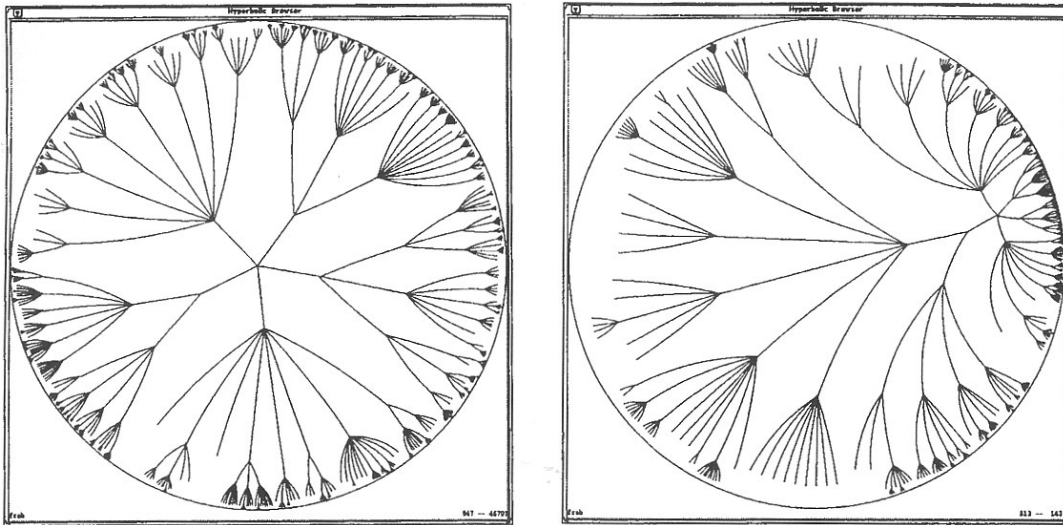


FIG. 3.2 : *Hyperbolic tree*

Des études [64] ont montré que les utilisateurs étaient capables de comprendre et d'utiliser avec succès ce genre de système dans les cas où l'interférence visuelle est faible. Nous verrons un moyen de minimiser ce phénomène dans la section 7.2.2.

Un défaut des approches précédentes est qu'elles n'offrent pas de continuité entre le focus et le contexte, et requièrent donc de l'utilisateur un effort mental pour faire le lien entre les deux vues. D'autres approches proposent d'agir sur la géométrie de la représentation pour intégrer le focus dans le contexte. On retiendra par exemple le *hyperbolic browser* de Lamping et Rao [142] qui permet d'afficher des structures d'arbre dans une vue hyperbolique (ou vue *fisheye* à la manière des objectifs du même nom pour appareils photographiques). Ces vues déformantes ont la propriété de grossir les noeuds au centre de la vue (focus) tout en conservant le contexte complet à la périphérie (voir figure 3.2). La déformation sphérique, variante de cette méthode, a des propriétés légèrement différentes qui la rend plus adaptée à des déformations locales (et potentiellement multiples) de l'espace (voir par exemple MultTab [215]). Enfin, on peut mentionner le *perspective wall* (figure 3.3) présenté dans l'*Information Visualizer* de Xerox PARC [112] qui fournit une zone de focus plane donc non déformante.

Une autre manière d'allier focus et contexte dans la représentation d'arbres, toujours proposée par Robertson dans [112], consiste à utiliser une vue pseudo 3D (figure 3.4), dans laquelle la zone d'intérêt est mise en avant par rapport au reste de l'arbre (l'utilisateur change le focus en effectuant une rotation du cône formé par les branches de l'arbre suivant son axe principal). Il est intéressant de noter que dans tous ces systèmes on retrouve la notion de transition animée. L'animation est un facteur important du point de vue de l'interaction homme-machine. Des transitions entre états et des déplacements animés correctement sont très utiles afin de ne pas désorienter l'utilisateur. Par exemple (figure 3.5), dans le cas où un déplacement d'un point à un autre est instantané, l'utilisateur doit fournir un effort pour identifier

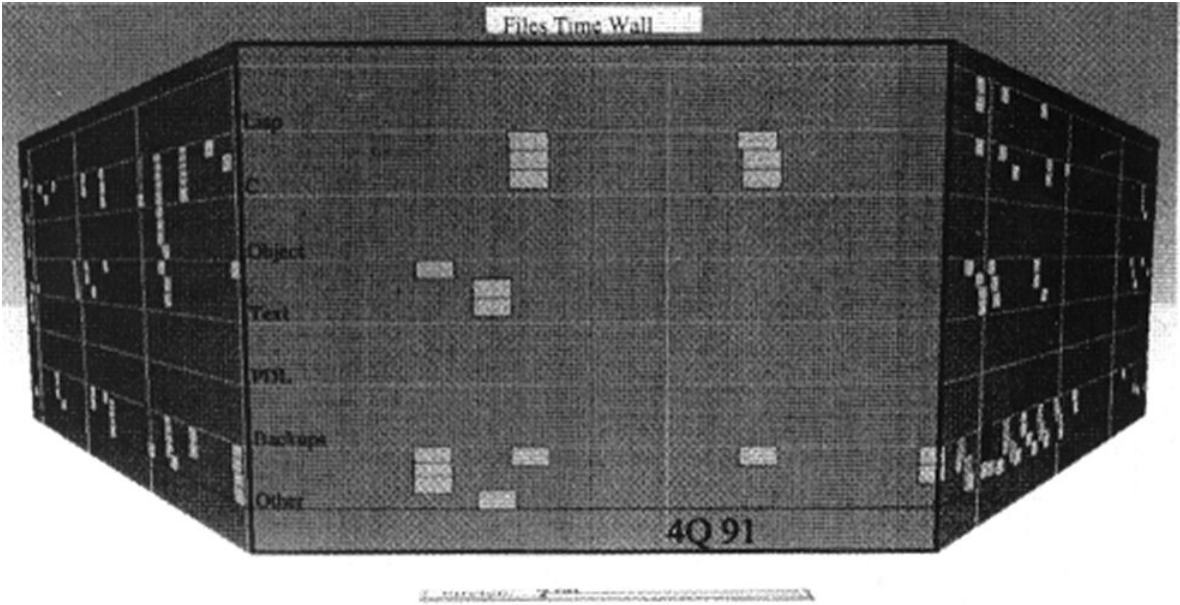


FIG. 3.3 : Perspective wall : visualisation de fichiers

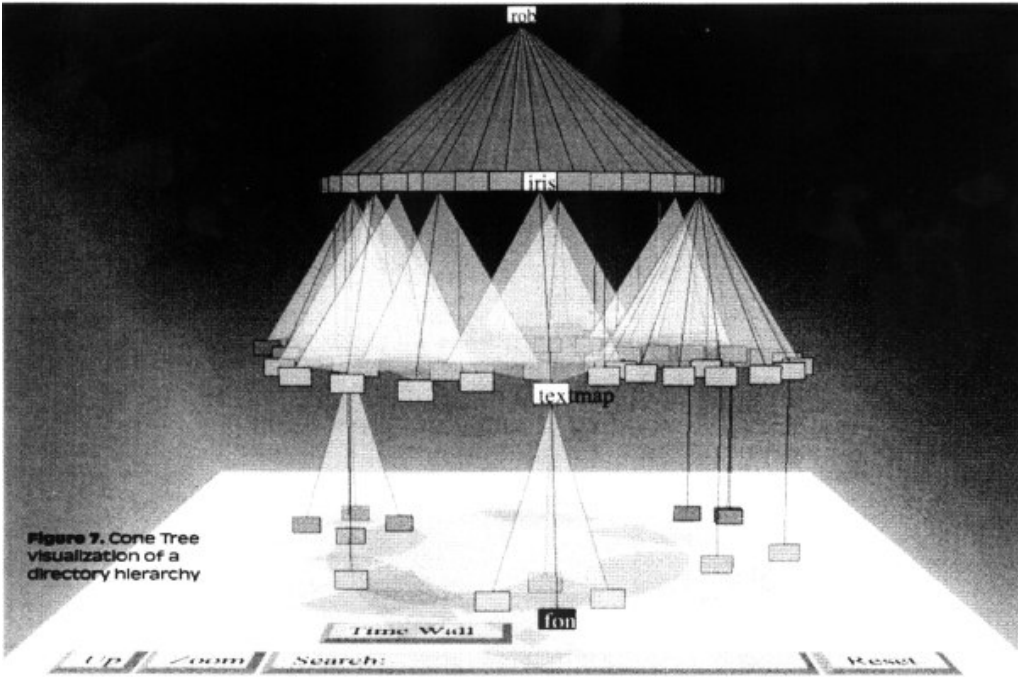


FIG. 3.4 : Cone tree : visualisation d'une hiérarchie de répertoires

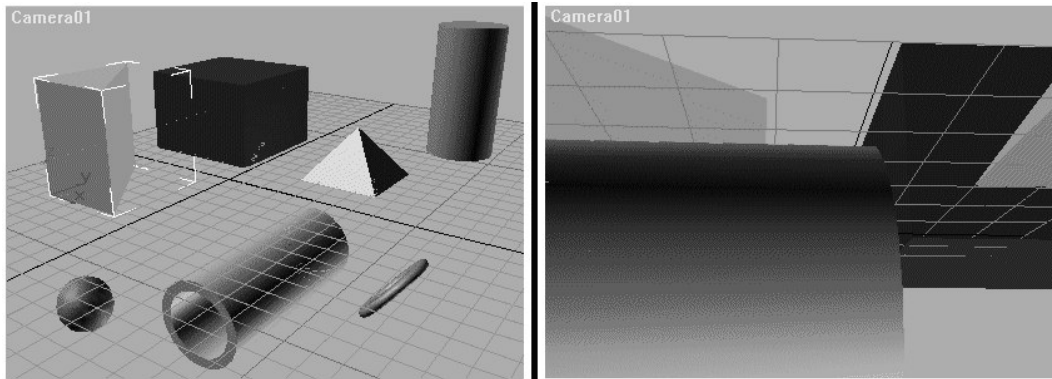


FIG. 3.5 : *La même scène tridimensionnelle vue sous deux angles différents*

la nouvelle vue et situer à nouveau le contexte dans lequel il se trouve. Au contraire, dans le cas où le déplacement est progressif (animé), l'utilisateur perçoit le mouvement et n'a aucun mal à se placer dans son nouveau contexte. On parle de continuité perceptuelle [112, 222], propriété très importante notamment pour la navigation dans l'espace de travail.

La navigation dans la structure visualisée peut être plus ou moins aisée suivant que l'espace est bidimensionnel ou tridimensionnel, et suivant les moyens de navigation mis à la disposition de l'utilisateur. Dans le cas d'espaces bidimensionnels, l'utilisateur n'aura à contrôler au maximum que trois paramètres : position (x,y) et orientation (rarement utilisée) de son point d'observation dans le plan. Les espaces tridimensionnels offrent quant à eux jusqu'à six degrés de liberté : position (x,y,z) et orientation (rotation autour des trois axes de l'espace). La navigation est rendue plus difficile par le nombre plus important de paramètres à contrôler, mais surtout par le fait que les outils d'interaction standard comme la souris et le clavier ne sont pas des moyens très intuitifs de se déplacer dans ces espaces. Il est d'autre part plus difficile de s'orienter dans un monde tridimensionnel projeté sur un écran. Pour pallier ces problèmes, Vion-Dury et Santana proposent dans [222] des techniques d'exploration automatique (à opposer aux techniques interactives) proposant par exemple des zooms intelligents (centrage de la vue sur un objet ou un groupe d'objets) et des trajectoires combinant zoom et rotation autour d'un centre d'intérêt.

Une solution intermédiaire consiste à construire une interface en 2.5 dimensions, c'est-à-dire une interface offrant la possibilité de zoomer de manière continue (ZUI : *Zoomable User Interface*). Les avantages par rapport aux interfaces 2D offrant un nombre fini de niveaux de zoom discret sont premièrement un contrôle plus fin de l'altitude d'observation (niveau de détails) et deuxièmement la possibilité d'avoir des transitions animées pour les déplacements impliquant un changement d'altitude. On améliore ainsi le confort d'utilisation sans rendre la navigation plus complexe. Les capacités de zoom sont importantes car elles permettent de modifier rapidement le point de vue de l'utilisateur (vue globale/vue détaillée de la zone d'intérêt), offrant ainsi une alternative aux techniques de focus+contexte décrites précédemment. Le chapitre 5 introduit XVTM (Xerox Visual Transformation Machine), une boîte à outils expérimentale basée sur les principes énoncés dans cette section et permettant la conception d'interfaces en 2.5 dimensions.

3.2 Généralités

3.2.1 Définition

Comme nous l'avons vu précédemment, les langages de programmation visuels (en abréviation *VPLs* : *Visual Programming Languages* par la suite) se différencient des langages de programmation textuels (en abréviation *TPLs*) principalement par le fait que la sémantique d'un programme peut être transmise en utilisant plus d'une dimension. Un programme écrit dans un langage textuel est constitué d'une chaîne de caractères ; il est donc unidimensionnel. Même si la syntaxe utilise souvent deux dimensions, les passages à la ligne suivante et les tabulations n'ont la plupart du temps pas de sémantique associée ; ils n'ont qu'un but documentaire, facilitant la relecture du code par le programmeur. Les langages de programmation visuels permettent d'utiliser d'autres dimensions, comme par exemple la couleur et la taille des objets formant les éléments de base du langage¹, les relations spatiales liant ces objets (inclusion, proximité, placement. . .) ou encore le temps. La combinaison de ces éléments de base forme des expressions visuelles, qui peuvent prendre la forme de diagrammes, d'icônes ou encore d'ébauches à main levée.

À notre connaissance, il n'existe pas de définition précise de ce qu'est un langage de programmation visuel. Tout d'abord, il est important de faire la différence entre les langages de programmation visuels (*VPLs*) et les environnements de programmation visuels (*VPEs*). Ces derniers fournissent une interface graphique aidant à la construction de programmes exprimés dans des langages textuels, facilitant notamment la spécification d'interfaces utilisateurs (GUI). On peut citer Microsoft Visual Basic, Visual C++ [160], JBuilder pour Java [29] ou encore VisualWorks pour SmallTalk. Les *VPEs* représentent aujourd'hui une part très importante des environnements de programmation commerciaux. Ils peuvent être considérés comme une étape intermédiaire entre les langages de programmation textuels et visuels, apportant aux langages textuels certains avantages des interfaces graphiques, comme l'exécution de programme pas à pas avec retour visuel immédiat, la construction de l'interface graphique du programme lui-même par manipulation directe des composants, ou encore la possibilité d'accéder facilement aux fichiers d'aide. Les *VPEs* font ainsi office de canal de transfert pour une partie de la recherche sur les langages visuels, mais ne sont pas considérés comme de véritables langages de programmation visuels. Différentes propositions de définition de ce qu'est un *VPL* ont été faites :

- Un langage de programmation visuel est une représentation graphique d'entités conceptuelles et d'opérations (Chang [92]).
- La programmation visuelle se réfère à tout système autorisant l'utilisateur à spécifier un programme au moyen de deux (ou plus) dimensions [...]. Les langages textuels conventionnels ne sont pas considérés comme bidimensionnels puisque les compilateurs ou interpréteurs les considèrent comme un unique flot de caractères unidimensionnel (Myers [169]).
- Un langage visuel manipule de l'information visuelle ou supporte l'interaction visuelle, ou bien encore autorise la programmation avec des expressions visuelles. Cette dernière possibilité est considérée comme étant la définition d'un langage de programmation visuel (Golin [106]).
- Les langages visuels ont une expression visuelle naturelle pour laquelle il n'y a pas d'équivalent textuel évident (Burnett [6]).
- La programmation visuelle est couramment définie comme l'utilisation d'expressions visuelles (comme des graphiques, dessins, animations ou icônes) dans le processus de programmation. Ces

¹Ce sont les équivalents des *mots* d'un *TPL*.

expressions visuelles peuvent être utilisées dans des environnements de programmation comme les interfaces graphiques pour langages de programmation textuels ; elles peuvent aussi être utilisées pour spécifier la syntaxe de nouveaux langages de programmation visuels conduisant à de nouveaux paradigmes comme la programmation par démonstration ; ou elles peuvent être utilisées dans des présentations graphiques du comportement ou de la structure d'un programme (Burnett et McIntyre).

- Enfin, F. Lakin [141] propose le terme “*executable graphics*” au lieu de “*Visual Programming Languages*” argumentant ainsi : “**Visual Programming Language** est une appellation impropre. Cela signifie soit langage de programmation que l'on peut voir, ce qui est trivial, soit langage pour programmer le comportement d'entités visuelles, ce qui est limitatif. **Executable graphics** exprime une orientation différente vers le domaine du problème : graphiques pouvant s'exécuter.”
- Un langage visuel est un ensemble d'arrangements spatiaux de symboles textuels et graphiques avec une interprétation sémantique utilisée pour exécuter des actions de communication.

Mais il est difficile de couvrir au moyen d'une seule définition l'ensemble des approches couramment répertoriées dans le domaine des *VPLs*. Les propositions précédentes restent vagues et incomplètes, d'autant plus que ce domaine de recherche tend à s'élargir comme en témoigne le changement de nom d'une des principales conférences qui lui est dédiée, à savoir l'*IEEE Symposium on Visual Languages* rebaptisée en 2001 *IEEE Symposium on Human Centric Computing* pour refléter son élargissement. Plus exactement, il s'agit d'une redéfinition de son thème : auparavant dédiée aux approches visuelles et multimédia, la conférence est maintenant centrée sur le but que ces dernières s'efforcent d'atteindre, à savoir améliorer et rendre plus simple la programmation et l'utilisation des machines. Les approches visuelles représentent une composante importante des moyens mis en œuvre, mais des techniques complémentaires telles que le son peuvent être utilisées afin de tirer au mieux parti des cinq sens humains.

Ainsi, il semble difficile de donner une définition générale de ce qu'est un langage de programmation visuel, ce terme étant parfois utilisé de manière abusive pour désigner des approches très différentes (voir les différents paradigmes introduits par la suite, par exemple le flot de contrôle comparé à la programmation par démonstration). Nous retiendrons la dernière proposition ainsi que celle de Burnett et McIntyre, les autres paraissant trop floues ou ne donnant pas de véritable définition de ce qu'est un langage de programmation visuel, mais des moyens de les différencier des langages de programmation textuels.

3.2.2 Historique

Les deux voies de recherche initialement explorées ont été d'une part une approche visuelle des langages de programmation traditionnels, et d'autre part des approches plus innovantes comme la programmation par démonstration. Ces systèmes se sont souvent révélés intéressants mais ont vite rencontré de gros problèmes lors de leur emploi dans des projets de taille réaliste, ce qui les a cantonnés au statut de “langages jouets”. Les *VPLs* ont ainsi été considérés comme impropres à effectuer des tâches de taille réaliste.

De nombreux travaux ont été effectués avec un certain succès pour essayer de résoudre ces problèmes, comme par exemple n'utiliser les langages visuels qu'à certains stades du développement d'un pro-

gramme ou encore spécialiser ces langages pour des domaines d'application bien définis. Il existe cependant toujours de nombreux problèmes comme le facteur d'échelle (*scalability*) ou la lisibilité des programmes et le défi original, c'est-à-dire proposer des *VPLs* assez expressifs et généraux pour aborder une gamme de problèmes de programmation toujours croissante, est toujours un sujet actif de recherche. On trouve néanmoins aujourd'hui des *VPLs* commerciaux spécialisés, comme LabVIEW [11], ou plus généraux, comme Prograph [65], qui ont une base conséquente d'utilisateurs.

3.3 Paradigmes

Les *VPLs* ayant moins de restrictions sur la façon d'exprimer les programmes d'un point de vue syntaxique, il devient possible d'explorer de nouveaux mécanismes de programmation qui n'étaient pas envisageables dans le passé mais qui sont rendus possibles aujourd'hui par la montée en puissance des processeurs et des cartes vidéo qui permet l'utilisation d'interfaces graphiques haute-résolution et d'environnements supportant le multi-fenêtrage (métaphore du bureau) [91].

Le but principal de la recherche sur les *VPLs* est d'améliorer le processus de création de programmes informatiques tout en réduisant les efforts mentaux du programmeur. Il s'agit de : 1^o rendre la programmation accessible à une certaine catégorie de personnes (qui ne sont pas forcément des programmeurs, mais par exemple des personnes ayant des connaissances dans le domaine du problème à résoudre) 2^o améliorer la justesse, au premier sens du terme, de la programmation, 3^o améliorer la vitesse à laquelle la tâche de programmation est accomplie (productivité).

Les stratégies mises en œuvre pour arriver à ces fins sont les suivantes [43] :

- être concret : c'est-à-dire essayer d'exprimer certains aspects du programme au moyen d'instances ;
- être direct : l'utilisateur doit avoir l'impression de manipuler directement des objets par ses actions ;
- être explicite : un aspect de la sémantique est explicite dans l'environnement s'il est déclaré directement (textuellement ou visuellement), sans que le programmeur n'ait à l'inférer ;
- fournir un retour visuel immédiat : afficher automatiquement les conséquences de l'édition de parties du programme. Le terme *liveness* [205] catégorise l'immédiateté avec laquelle un retour sémantique visuel est automatiquement fourni pendant l'édition. Les différents degrés de *liveness* sont décrits dans la section 3.4.2.

Les sections suivantes dressent un panorama des différents paradigmes existants, en illustrant chacun d'eux par un ou plusieurs exemples.

3.3.1 Flot de contrôle

L'approche flot de contrôle (*controlflow*) consiste à représenter graphiquement les structures de contrôle communément rencontrées dans les langages de programmation textuels, telles que les boucles, les appels

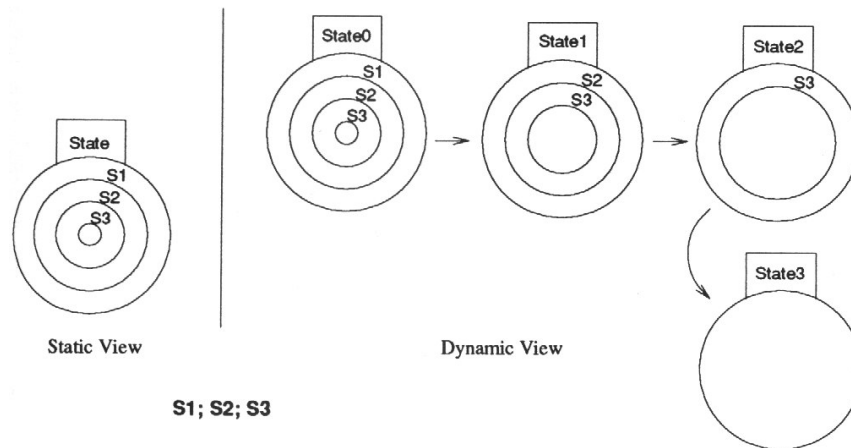


FIG. 3.6 : *VIPR, Représentations statique et dynamique des instructions séquentielles s_1 , s_2 et s_3*

de procédures, les branchements conditionnels et le parallélisme. Cette approche peut donc être considérée comme une transposition directe dans le domaine visuel de langages de programmation textuels conventionnels.

VIPR

VIPR [104, 52] est un langage de programmation visuel utilisant la topologie pour coder sa sémantique, les relations topologiques utilisées étant l'imbrication des cercles et la connection des objets. Des dimensions visuelles comme la taille ou la couleur des objets ne codent pas de sémantique. Il est inspiré de Pictorial Janus [131], mais modélise des constructions impératives conventionnelles. Une étape de computation est représentée (figure 3.6) par la fusion de deux cercles. Un anneau contenant un label optionnel représente une condition (figure 3.7a). Le résultat de la fusion de deux cercles est l'exécution de l'action associée au cercle le plus intérieur des deux fusionnés. Ces actions peuvent être des instructions d'affectation ou des déclarations d'entrée/sortie. La figure 3.7b illustre un appel de fonction.

Comme le montre la figure 3.8, les programmes VIPR deviennent rapidement complexes et difficiles à lire, ce qui rend le langage assez peu résistant au facteur d'échelle (section 3.4.1). L'environnement propose des mécanismes de navigation qui agrandissent (zoom) les parties du programme éditées ou s'exécutant. Il est aussi possible de paramétrer le niveau de détail d'une vue. Malgré cela, il semble difficile d'obtenir une vue globale du programme et de comprendre son but, les entités occupant beaucoup de place et la représentation des structures de contrôle étant peu intuitives.

VamPict

VamPict [12] est un langage très simple utilisé pour illustrer la spécification d'un langage basé sur le paradigme du flot de contrôle en VAMPIRE (section 3.3.3), un langage visuel pour la spécification d'autres langages de programmation visuels.

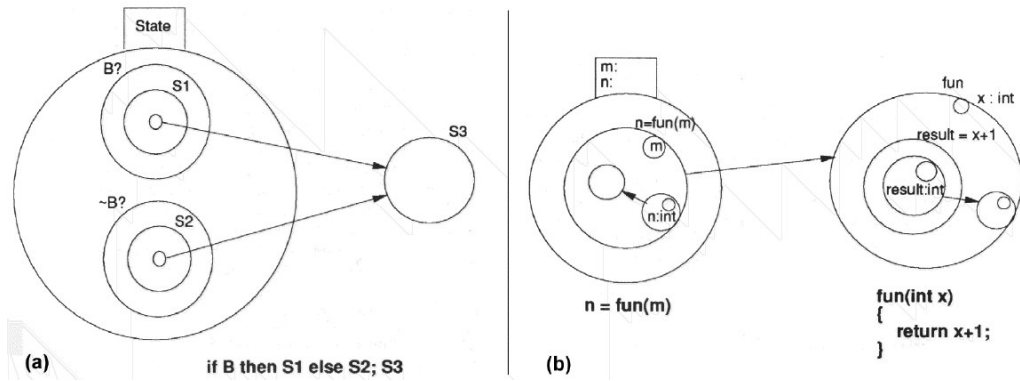


FIG. 3.7 : VIPR, Structures de contrôle

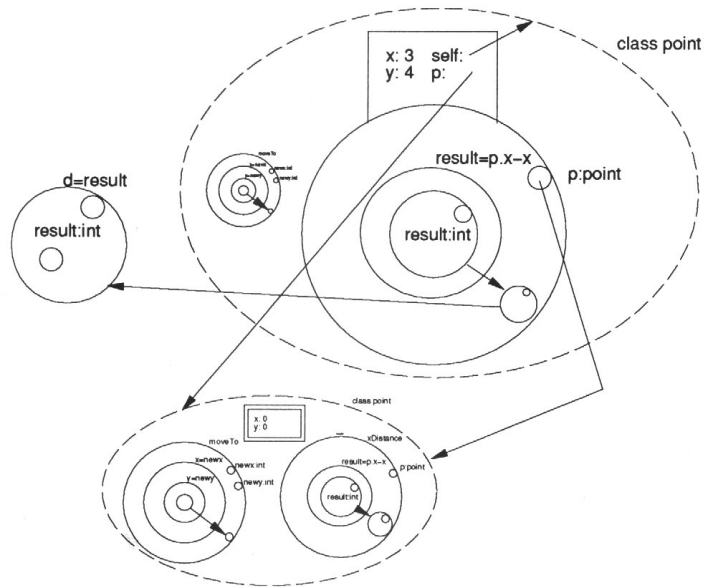


FIG. 3.8 : Programme VIPR

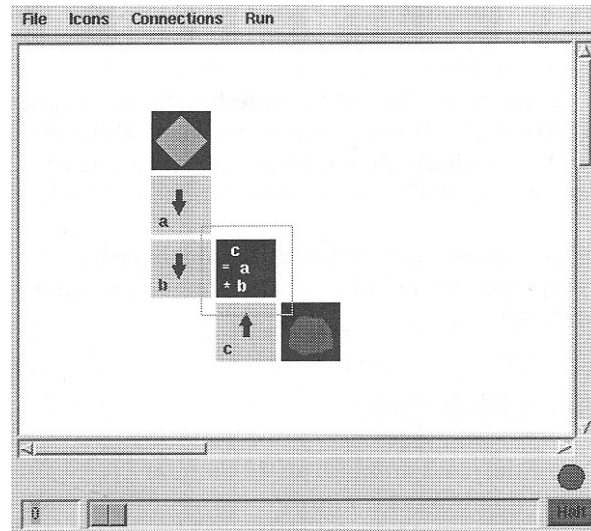


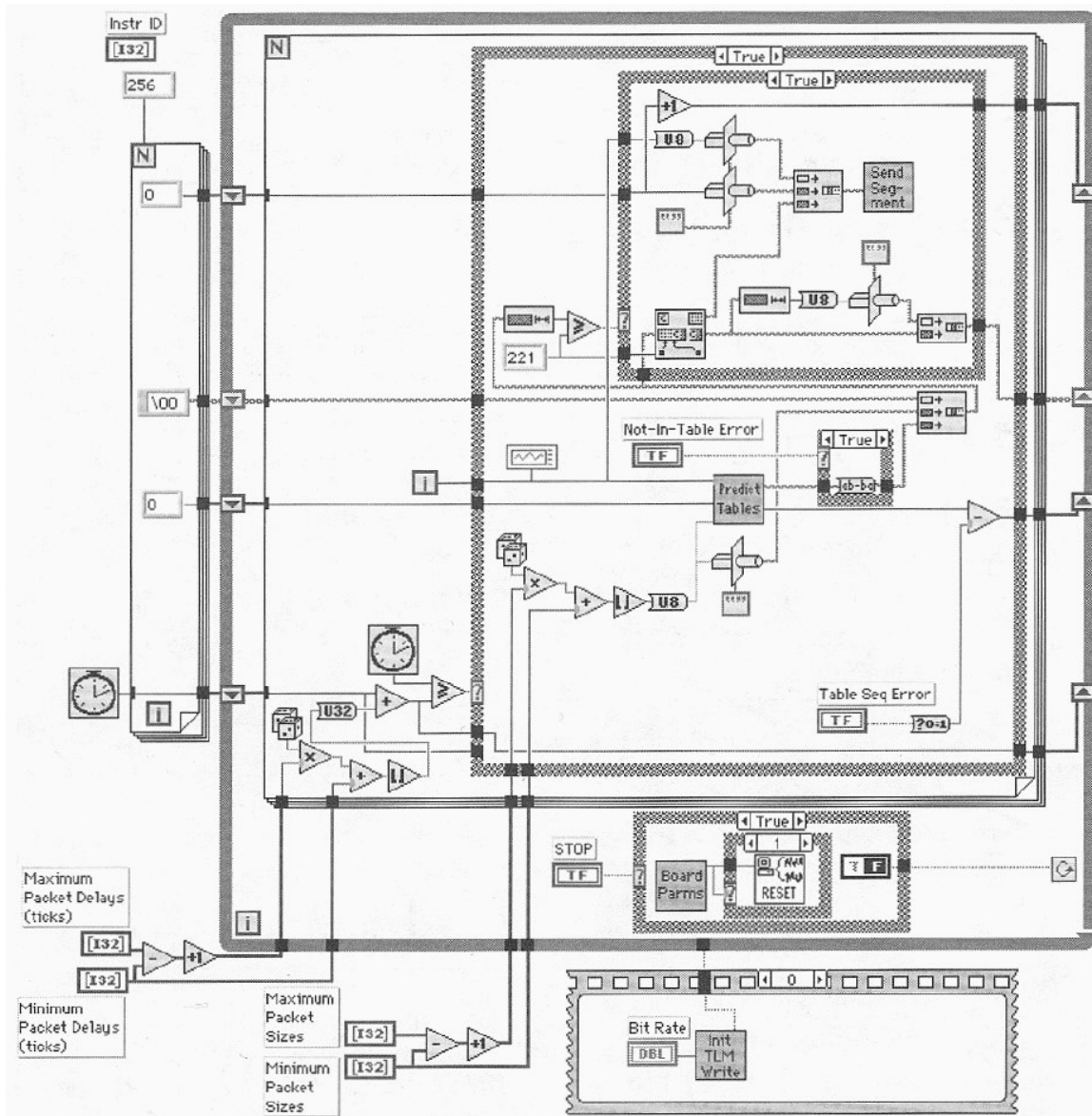
FIG. 3.9 : *Vampire* : saisie de a puis b , calcul de c , affichage de c

La figure 3.9 donne un exemple de programme VamPict. Ce programme demande à l'utilisateur d'entrer deux valeurs dont il calcule et affiche le produit. Une partie de la spécification VAMPIRE du langage VamPict est décrite dans la section 3.3.3. Le programme s'exécute en fonction des règles de réécriture visuelles exprimées en VAMPIRE. Dans le cas de VamPict, le flot est orienté du haut vers le bas, de la gauche vers la droite (le carré entourant l'instruction de calcul du produit représente le curseur d'exécution).

Les langages basés sur ce paradigme peuvent aussi adopter une représentation diagrammatique (boîtes reliées par des flèches). Le flot de contrôle est adapté aux problèmes de contrôle réactif (réaction à des événements). Les données sont souvent représentées et manipulées sous forme textuelle, limitant l'apport d'une représentation visuelle aux seules structures de contrôle ; mais celles-ci n'étant souvent pas très intuitives (la représentation visuelle d'entités abstraites n'est pas évidente et s'avère souvent plus lourde que l'équivalent textuel), l'intérêt de ce paradigme est limité. Il peut néanmoins être intéressant, par exemple dans le cadre de l'animation d'algorithme (but éducatif).

3.3.2 Flot de données

Le flot de données (*dataflow*) est l'approche la plus utilisée dans les produits commerciaux. Un programme *dataflow* est constitué d'un réseau d'objets représentant des opérations. Chaque objet possède un (ou plusieurs) port(s) par lequel les données arrivent (ports d'entrée). Les résultats des opérations sont envoyés par d'autres ports (ports de sortie) vers d'autres objets. Ceux-ci sont reliés par des segments qui symbolisent des conduits par lesquels transitent les données. Les états des différents composants sont donc modifiés au cours du temps. La figure 3.10 montre un exemple de programme dans le langage commercial LabVIEW détaillé plus loin. Avec le paradigme du flot de données, le programmeur se concentre sur les relations entre données, pas sur la manière d'effectuer les opérations. Dans une approche *dataflow*



Copyright © 1993, California Institute of Technology. U.S. Government Sponsorship under NASA Contract NAS7-918 is acknowledged.

FIG. 3.10 : Exemple de programme LabVIEW

pure, tous les aspects du flot de contrôle, c'est-à-dire la manière d'effectuer les opérations sont éliminés. C'est le système qui détermine *quand* les calculs doivent être effectués. Les fonctions (opérations) sont déclenchées par la présence de tous les paramètres d'entrée, c'est-à-dire quand des valeurs sont disponibles pour chaque port d'entrée, et non pas par des appels explicites du programmeur. S. Eisenbach propose dans [85] une méthode de traduction de diagrammes *dataflow* en programmes textuels exprimés dans un langage fonctionnel, en se basant sur le fait que les *VPLs dataflow* et les *TPLs* fonctionnels présentent des similarités dans le sens où ils sont pilotés par les données.

LabVIEW

LabVIEW [127, 136] est un langage visuel commercial de National Instruments Corp., fourni avec une importante librairie de fonctions prédéfinies et de composants d'interaction. Conçu initialement en 1986 pour l'acquisition et le traitement de données, il a évolué vers un langage plus général, permettant d'effectuer des simulations ou encore du traitement d'image grâce à des librairies spécialisées. Les programmes LabVIEW rappellent fortement des schémas de circuits électroniques, avec la possibilité d'associer des boutons et des afficheurs avec les différents composants du programme, ce qui rend le langage accessible à des électroniciens n'ayant pas forcément une grande expérience en programmation.

Les flots de données sont typés (nombres, booléens, chaînes de caractères, tableaux). Ces types sont reflétés visuellement par le motif et la couleur de la ligne représentant le flot. Le langage possède certaines structures de contrôle : branchements conditionnels, boucles et séquences, exprimées par l'imbrication des composants. La figure 3.10 représente un programme pour la génération de signaux. Le programme comprend notamment une boucle *for* et des branchements conditionnels. Dans un programme LabVIEW, une seule alternative est visible à la fois. La figure montre, pour les cinq tests présents, le code exécuté dans le cas où le booléen s'évalue à vrai (*true*).

Tous les composants et structures de contrôle peuvent être regroupés dans des abstractions appelées *VI*s (virtual instruments). Les *VI*s peuvent eux-mêmes être composés d'autres (*sub-)**VI*s, pouvant être sauvés et réutilisés dans d'autres programmes. Le langage est intuitif pour qui est habitué aux circuits électroniques et permet en général d'avoir une compréhension globale du programme, principalement grâce aux *virtual instruments*. La séparation entre composants du programme lui-même et leurs interfaces respectives apporte aussi de la clareté.

Prograph

Prograph [66, 65] est un langage de programmation visuel orienté objet. Les classes, les méthodes et les attributs sont créés et modifiés par manipulation directe d'icônes à l'écran. L'environnement de Prograph est composé d'un éditeur, d'un interpréteur, d'un débogueur et d'un compilateur.

Prograph propose des constructions pour spécifier des itérations, des branchements conditionnels, des opérations séquentielles ou parallèles. La figure 3.11 montre le *browser* de classes ; la figure 3.12 illustre deux méthodes de la classe *Index* : *Sort* et *BuildValue*. Le corps des méthodes est spécifié par un réseau d'icônes reliées entre elles reposant sur le paradigme du *dataflow* (le flux de données est orienté de haut en bas). L'accès au corps d'une méthode s'effectue par double-clic sur l'icône représentant son appel,

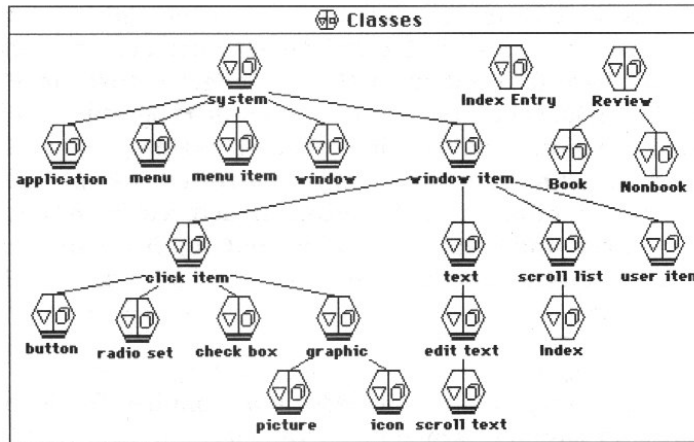


FIG. 3.11 : Browser de classe Prograph

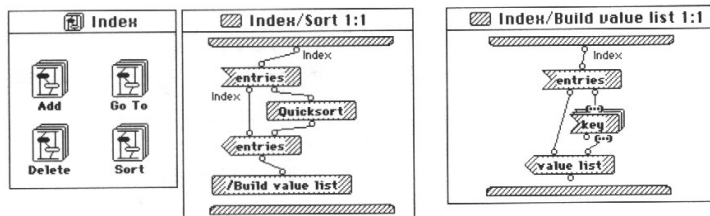


FIG. 3.12 : Exemple de méthodes Prograph

ou par le *browser* de classes. Toutes les méthodes possèdent une barre horizontale en haut de la fenêtre, représentant le flot d'entrée, et une barre similaire en bas de la fenêtre représentant le flot de sortie. La sémantique d'une opération varie en fonction de la forme de l'icône la représentant. Par exemple l'icône gauche de la figure 3.13 représente l'opération d'accès à un attribut, alors que la droite appelle une méthode. Pour plus de détails, voir [65].

L'éditeur fournit des mécanismes d'aide à la spécification qui limitent les erreurs syntaxiques. Il n'est par exemple pas possible de créer des connections incorrectes. L'exécution d'un programme est animée, surlignant les éléments responsables d'une erreur.



FIG. 3.13 : Représentation en Prograph des opérations d'accès à un attribut et d'appel de méthode

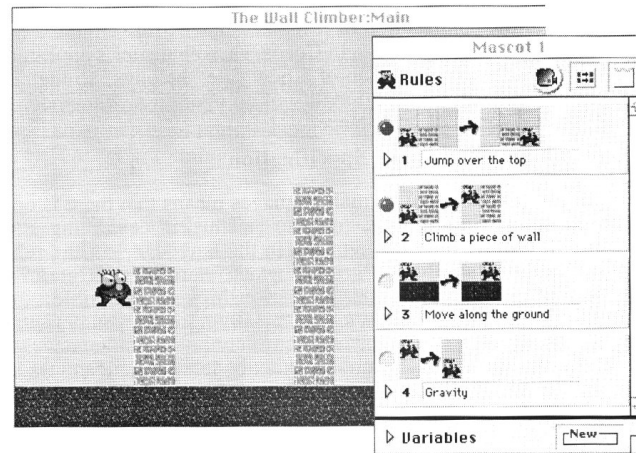


FIG. 3.14 : Cocoa wall-climber

Les langages basés sur ce paradigme peuvent être plus ou moins assimilés à des langages déclaratifs (contrairement aux langages basés sur le flot de contrôle qui sont impératifs), puisqu'ils ne requièrent pas que le programmeur se concentre sur les aspects contrôle du programme. Ces langages permettent de faire des abstractions au niveau des données et des procédures, comme les *VI*s (virtual instruments) de LabVIEW (voir section 3.3.2). Il est par contre difficile de gérer les événements et réactions associées dans des langages uniquement basés sur le *dataflow* puisqu'ils nécessitent un minimum de contrôle quant à l'ordonnancement des opérations.

3.3.3 Systèmes de règles

Le paradigme basé sur les systèmes de règles est très proche de son équivalent textuel. Le système est dans un certain état ; un ensemble de règles transforme cet état en un autre état. Chaque règle est composée d'une précondition (partie gauche) et d'une spécification de transformation (partie droite) à exécuter si la précondition est vérifiée. Les cas de conflits (plusieurs règles pouvant s'appliquer à un instant t) peuvent se résoudre de différentes manières. La plus simple consiste à appliquer la première règle rencontrée ; mais il est aussi possible d'assigner des priorités aux règles manuellement ou encore de calculer dynamiquement suivant un critère quelle règle doit être appliquée. On introduit alors un peu de contrôle dans une approche initialement déclarative.

Parmi les langages de programmation visuels basés sur les systèmes de règles, on peut citer Cocoa [197], destiné aux enfants, maintenant appelé Stagecast Creator [67]. Dans l'exemple de la figure 3.14, le personnage suit les règles définies dans la fenêtre *Mascot 1*. La partie gauche des règles représente la précondition, la partie droite représente la transformation. On peut aussi mentionner Altaira [90] et Roadsurf [233].

VAMPIRE (Visual Metatools for Programming Iconic Environments, [157]) est un outil de génération de systèmes de programmation icôniques lui-même implémenté comme un langage icônique² et basé sur un système de règles de réécriture graphiques.

La figure 3.15 présente les règles de réécriture qui permettent de définir l'évolution dynamique des constructions visuelles de VamPict, langage basé sur le paradigme du flot de contrôle et décrit précédemment. Ces règles sont respectivement *Orientation du flot du haut vers le bas*, *Orientation du flot de la gauche vers la droite*, *Condition de lancement du flot de contrôle* et *Condition d'arrêt du flot de contrôle*. La figure 3.16 illustre quant à elle *Saisie textuelle d'une valeur d'entrée*, *Affichage d'une variable de sortie* et *Réalisation d'un calcul numérique simple (produit)*. Le carré en surimpression symbolise l'icône active. Chaque membre d'une règle possède deux fenêtres : l'une pour les paramètres visuels, l'autre pour les attributs textuels et le code SmallTalk associé.

Il peut être difficile de comprendre un programme basé sur ce paradigme dans son ensemble, puisqu'il n'y a pas réellement de vision globale de ce qu'il fait. Le choix et l'ordre d'exécution des règles ne sont visibles que lors de son déroulement. Au niveau de la conception des langages visuels, P. Bottoni [33] propose une formalisation des sessions de dialogue homme/machine par un langage visuel et montre comment des systèmes de réécriture peuvent être utilisés pour spécifier les aspects computationnels et graphiques d'un langage visuel.

3.3.4 Langages de requête

Un paradigme assez proche du précédent est celui des langages visuels permettant d'exprimer des requêtes sur des bases de données. Le terme *langage de requête visuel* regroupe plusieurs choses différentes :

- des requêtes sur des bases de données pour retrouver des images ou des animations, que ce soit de manière textuelle ou visuelle,
- des interfaces graphiques aidant à la construction de requêtes textuelles (SQL,...) comme Microsoft Access [159],
- des requêtes exprimées au moyen de primitives graphiques (partie qui nous intéresse) pour l'interrogation de bases de données.

Parmi ces derniers, on peut citer les travaux de M. Andries et G. Engels [8] qui proposent un langage de requête mixant des primitives textuelles et graphiques. Le modèle sous-jacent est l'EER (modèle entité relation étendu), mais devrait pouvoir s'adapter à n'importe quel type de base de données. Le but de ce langage est de combiner les avantages des requêtes textuelles et des représentations graphiques. Ainsi, la structure d'une requête est toujours de la forme *select-from-where*, avec possibilité d'exprimer les clauses soit de manière textuelle soit de manière visuelle. La figure 3.17 illustre une requête simple permettant de retrouver les noms et âges des personnes ayant au moins 18 ans.

²La plupart des langages visuels sont implémentés au moyen de langages textuels conventionnels. VAMPIRE permet la spécification d'un langage visuel de manière visuelle.

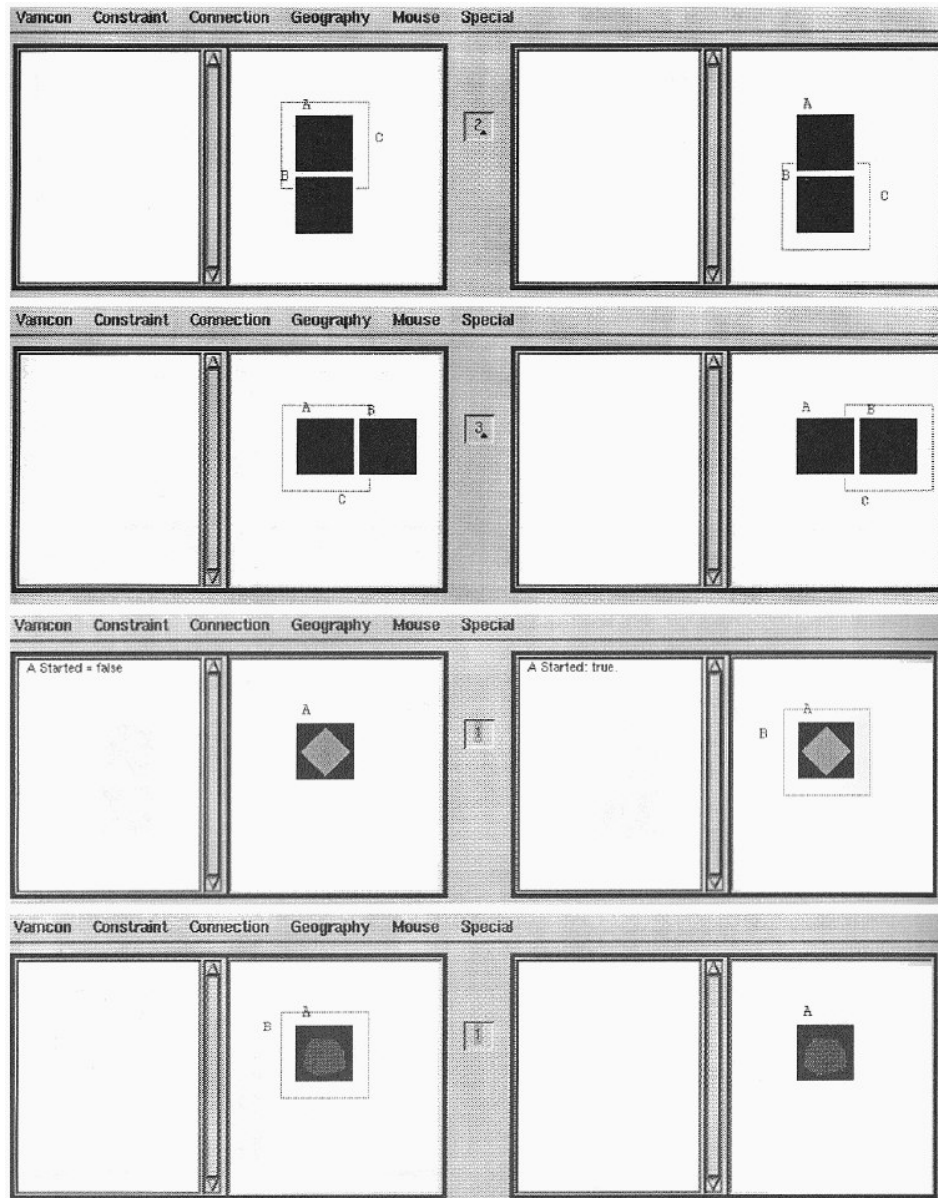


FIG. 3.15 : Règles Vampire pour l'écoulement du flot de contrôle

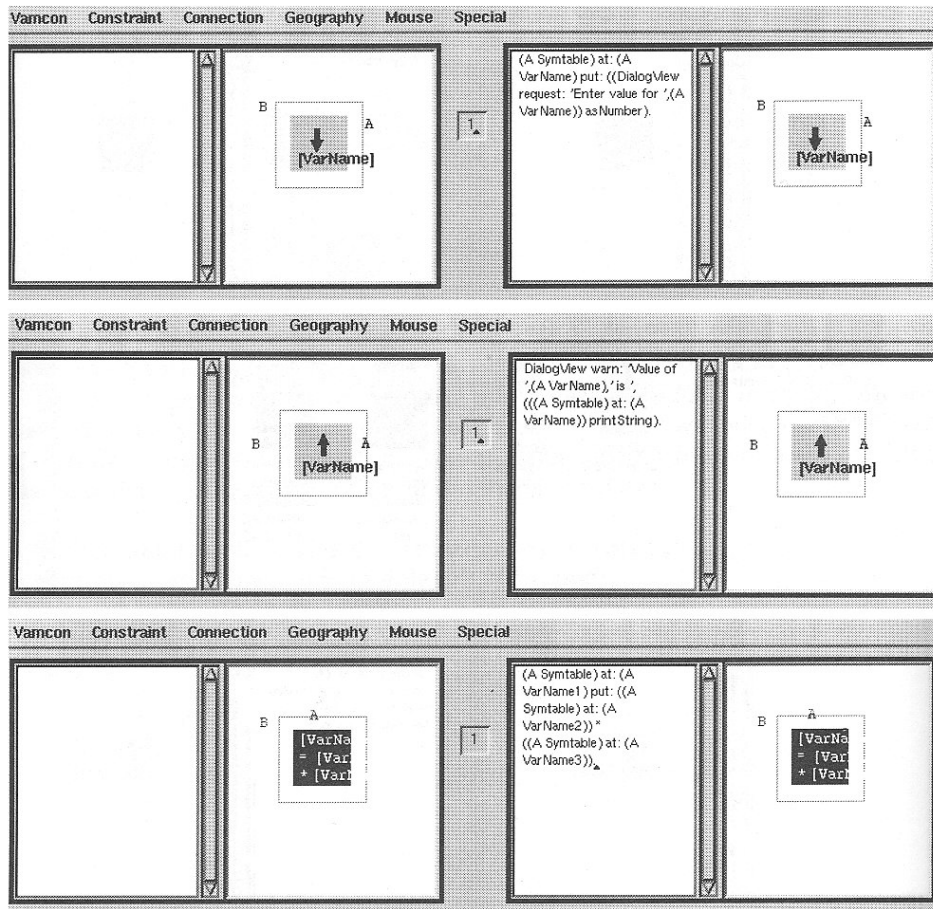


FIG. 3.16 : Règles Vampire pour les opérations



FIG. 3.17 : Requête visuelle : nom et âge des personnes majeures

Les bases de données décrivant des relations entre objets/données, les schémas de base de données sont souvent représentés sous forme visuelle (nous ne parlons pas ici des vues que l'utilisateur peut créer des données, mais bien du schéma et des instances de la base). Il semble donc naturel de pouvoir exprimer des requêtes de manière visuelle, les structures de graphe ou d'arbre étant mieux perçues sous forme visuelle que textuelle, et les requêtes exprimant des contraintes sur ces structures, rendues plus explicites par la représentation graphique.

Un des travaux les plus intéressants dans le domaine des langages de requêtes visuels est XML-GL[47] qui propose un modèle de représentation graphique des DTDs et documents XML, et un langage visuel permettant d'exprimer des requêtes sur des documents structurés et de réorganiser les résultats dans des structures telles que des listes ou des groupes³. XML-GL est étudié plus en détails dans le chapitre 4, ainsi que le langage de M. Erwig [86], et celui de K. Zhang [238].

Enfin, il existe plusieurs langages spécialisés dans certains types de requêtes comme ceux pour les systèmes d'information géographique. Dans [84], il est possible d'exprimer des requêtes en créant une ébauche des configurations topologiques voulues. Le système est capable de relâcher les contraintes de placement des objets ; il est possible d'utiliser les notions d'orientation, de relations spatiales entre objets (traverse, recouvre, proche de) et de distance pour exprimer des requêtes plus ou moins précises. Dans ce cas, l'utilisation d'un langage visuel réduit de manière très significative la complexité de la requête, puisque les contraintes exprimées dans celle-ci sont de nature visuelle, donc multidimensionnelles. D'autre part, la correspondance entre les éléments du langage visuel et les contraintes à exprimer est directe (*closeness of mapping*, cf section 3.4.2), ce qui rend le langage très intuitif.

3.3.5 Programmation par démonstration

L'exemple de programmation par démonstration le plus basique est le système de macro de Microsoft Word, qui permet à l'utilisateur d'enregistrer une séquence d'actions à répéter automatiquement. Un exemple un peu plus complexe est l'éditeur d'interface graphique, qui permet de spécifier l'interface d'un programme par manipulation directe de ses composants. Ce type de système est intéressant car il augmente la facilité, et par conséquent la vitesse avec laquelle l'interface est spécifiée. Le code source permettant de créer cette interface est généré automatiquement à partir de la spécification visuelle de l'utilisateur, par exemple en Java.

Mais il existe aussi des systèmes beaucoup plus complexes capables d'inférer des programmes en généralisant les actions de l'utilisateur effectuées sur des données réelles [171], en utilisant par exemple des techniques d'intelligence artificielle. Ces systèmes permettent à des utilisateurs de créer des procédures générales sans avoir à programmer.

Toontalk

Toontalk [148] est un environnement interactif décrivant un monde animé, rappelant un jeu vidéo, dans lequel les enfants peuvent créer des programmes en agissant sur des objets. Ces objets représentent

³Il est possible notamment d'exprimer des jointures et le modèle supporte les attributs ID/IDREF de XML.



FIG. 3.18 : Toontalk : entraînement d'un robot

les différentes abstractions computationnelles disponibles. Par exemple (figure 3.18), un robot représente une méthode, une balance représente une comparaison (test) et un oiseau représente un envoi de message. Les structures de données telles que les vecteurs et tableaux sont représentées par des boîtes compartimentées dans lesquelles peuvent être placés d'autres objets. La généralisation des programmes s'effectue en supprimant de manière explicite certains détails fournis lors de la démonstration.

La plupart des systèmes de programmation par démonstration ne proposent pas de représentation visuelle du programme inféré, ce qui empêche toute action d'édition, sachant que l'inférence peut se révéler incorrecte. Il est donc difficile de savoir si le programme est correct et de corriger les erreurs. Ils demandent d'autre part à l'utilisateur d'envisager le processus de "programmation" d'une manière très différente comparée aux méthodes de programmation conventionnelles. La programmation par démonstration peut être intéressante pour des utilisateurs ne voulant pas apprendre à programmer, mais risque de dérouter les personnes ayant un peu d'expérience, voir d'engendrer des frustrations notamment du fait que le contrôle qu'a l'utilisateur sur la spécification et donc l'expressivité est nettement plus limité.

Pursuit

Pursuit [167] est un environnement de programmation visuel orienté *shell* puisqu'il est dédié à la manipulation de fichiers. Pursuit fournit un retour à l'utilisateur quant au programme inféré en affichant les opérations par des changements sur les objets représentant les données (fichiers, répertoires, ...). Si le système a inféré des opérations de manière incorrecte, il est possible d'éditer le programme.

Le système est capable d'inférer des boucles et des branchements conditionnels. La dimension *couleur* des objets graphiques a la même fonction que les noms de variables dans les langages textuels. Dans ces derniers, les noms de variables représentent des données. Les opérations sur ces variables affectent les données, mais pas leur nom. Dans Pursuit, les opérations sont traduites visuellement par un changement de la représentation des icônes liées aux données. La couleur n'étant pas utilisée pour traduire les opérations, elle permet d'identifier de manière unique un objet graphique représentant une certaine donnée

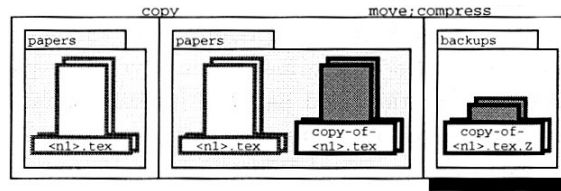


FIG. 3.19 : *Script Pursuit*

malgré le fait que son apparence change au cours du temps. Le fait que l'œil humain ne puisse distinguer qu'un nombre limité de couleurs n'est pas vraiment limitatif puisque les scripts Pursuit utilisent en général assez peu d'objets.

La figure 3.19 montre un exemple de script Pursuit dans lequel il a été inféré que tous les fichiers TeX du répertoire *papers* devaient être dupliqués puis compressés et déplacés dans le répertoire *backups* en leur ajoutant l'extension *.Z*. L'opération de compression est représentée par la réduction de taille de l'objet.

3.3.6 Tableurs

Le tableur (*form/spreadsheet* ou *equation-based*) est le paradigme qui a rencontré le plus grand succès commercial à ce jour. Parmi les produits commerciaux, Microsoft Excel est le plus connu. L'environnement est un tableau dont les cellules contiennent des valeurs ou des formules textuelles qui peuvent être complexes (expressions mathématiques, conditions, ...). La construction de ces formules peut s'effectuer visuellement en pointant les cellules impliquées et en utilisant des icônes représentant des opérations. Les formules forment un réseau unidirectionnel de contraintes : les changements effectués dans une cellule sont donc propagés automatiquement, fournissant un bon niveau de *liveness*.

Dans le domaine de la recherche, Forms/3 [42] propose de généraliser cette approche. Le langage est *Turing-complete*, son but étant d'étendre le concept du tableur pour en faire un langage de programmation complet. À cette fin, l'environnement supporte dans ses cellules des graphiques, des animations et la notion de récursivité. De plus les cellules ne sont pas contraintes par un tableau. Le programmeur crée un programme en plaçant directement des cellules sur des feuilles et en définissant une formule pour chaque cellule. Comme pour les systèmes de programmation par démonstration, la manière de spécifier des programmes peut être déroutante pour des personnes sachant déjà programmer. Et même si les personnes visées par Forms/3 sont les futurs programmeurs, c'est-à-dire les gens dont la tâche sera de programmer mais qui ne sont pas familiers avec les techniques conventionnelles, on peut se demander à quel point ce paradigme est approprié pour faire de la programmation générale. En effet, Forms/3 se base, entre autre, sur le fait que les tableurs sont largement utilisés, et ce par des personnes très diverses. Mais ce succès vient peut-être du fait qu'ils sont bien adaptés pour la résolution d'une certaine classe de problèmes.

Afin d'être assez expressifs, les *VPLs* reposent rarement sur un seul paradigme. Par exemple, LabVIEW, bien que basé sur le flot de données, propose des structures de contrôle comme les boucles *FOR*.

D'autres recherches sont plus explicitement multi-paradigmatiques. E. Ghittori essaye par exemple d'enrichir le langage VIPERS avec des structures de contrôle complexes et étudie leur utilisabilité [105]. Dans [89], une tentative est faite pour mélanger fortement programmation textuelle et programmation visuelle, les parties visuelles étant converties dans leur équivalent textuel avant la compilation. Une approche similaire apparaît dans Andrews [114]. Enfin, on peut citer les travaux de T. Catarci et al [46], qui proposent différents mécanismes d'interaction pour exprimer des requêtes (plus ou moins expressifs et simples à utiliser), dépendant du niveau d'expérience de l'utilisateur. Le modèle sous-jacent est commun aux différents modes de représentation, à savoir diagrammatique, icônique et orienté tableur.

3.4 Problèmes connus

Même si des progrès significatifs ont été faits depuis les premières tentatives de représentation visuelle de programmes, une partie de la communauté scientifique considère toujours que les langages de programmation visuels n'ont aucun avenir à cause de certaines faiblesses comme la résistance au facteur d'échelle. De nombreuses études empiriques ont été réalisées afin de mesurer l'impact de facteurs comme la facilité de compréhension, de spécification d'un programme visuel ou l'expressivité du langage.

3.4.1 Facteur d'échelle

Un des plus importants problèmes des langages visuels est la résistance au facteur d'échelle ou scalabilité (*scalability*), c'est-à-dire la possibilité d'utiliser le langage pour écrire des programmes de taille importante. La plupart des langages se comportent très bien avec de petits programmes de démonstration. Mais dès que la taille de ces programmes augmente, des problèmes de représentation, de navigation et de compréhension apparaissent.

Représentation statique

Le terme *limite de Deutsch* désigne une question posée par Peter Deutsch lors d'une présentation sur les langages visuels : "*Well, this is all fine and well, but the problem with visual programming languages is that you can't have more than 50 visual primitives on the screen at the same time. How are you going to write an operating system ?*"

Ce problème de représentation existe effectivement, mais l'on peut se demander combien de primitives sont vues simultanément dans un langage textuel. Ainsi, David McIntyre donne l'exemple d'une fenêtre Emacs de 50 lignes sur un moniteur 19 pouces, qui contient au maximum 80 primitives [156], ce qui ne représente pas une différence énorme, sachant qu'une primitive graphique contient souvent plus de sémantique qu'une primitive textuelle. Mais le problème ne se limite pas au nombre de primitives (graphiques ou textuelles) pouvant être représentées à l'écran. Les différences liées aux propriétés des représentations font que le programmeur ne les appréhende pas de la même manière suivant que les programmes sont représentés de manière textuelle ou visuelle. Ces deux types de représentation ne posent donc pas les mêmes problèmes. Par exemple, dans le cas d'une représentation textuelle, plus symbolique et plus abstraite, le programmeur n'attend pas⁴ que les liens de dépendance soient représentés

⁴Au-delà des attentes de l'utilisateur, c'est une partie des avantages supposés des représentations visuelles qui est en cause.

explicitement, contrairement au cas d'une représentation visuelle plus concrète. De même, les représentations textuelles ne permettent pas d'avoir une vision globale d'un programme. Plus précisément, elles n'offrent pas le même type de vision globale que permettent potentiellement les représentations graphiques couplées à des capacités de navigation sophistiquées. Sur ce plan, il est intéressant de proposer en plus des fonctions standard de translation des fonctions de zoom (discret ou continu) qui permettront d'avoir une vision globale d'un programme ou bien de se concentrer sur des détails (VIPR [104], outil de monitoring décrit dans [224]).

La navigation ne fournit qu'une solution partielle aux problèmes liés à la représentation statique. Il est par exemple difficile de représenter certains attributs, comme les références non textuelles sur des icônes sans nom qui vont rendre le programme difficile à lire si leur nombre est trop important (représentation surchargée). Dans ce cas, il est possible de cacher des détails excessifs en fonction du niveau d'abstraction en proposant différentes vues du même programme. Il est aussi possible d'utiliser les capacités de représentation dynamique des environnements visuels, par exemple en faisant apparaître les commentaires liés à un objet seulement quand celui-ci est pointé. Dans LabVIEW, les ports d'entrée/sortie non assignés n'apparaissent que lorsque l'objet est pointé et qu'une opération de lien est en cours ; il est aussi possible d'afficher un descriptif de l'objet en double-cliquant sur son icône. Enfin, il est à noter que certains paradigmes ont une syntaxe dynamique, comme la programmation par démonstration, ce qui rend la représentation statique difficile.

Abstraction

La scalabilité est facilitée par l'abstraction procédurale. Celle-ci est devenue tellement commune dans les *TPLs* que personne n'envisagerait un projet de programmation sans l'utiliser. Cette abstraction a été transposée dans les *VPLs*. Par exemple, dans les langages basés sur le *dataflow*, il est possible de regrouper un ensemble d'objets et leurs liens en une seule entité, qui pourra être sélectionnée et réutilisée dans d'autres parties du programme, voire stockée dans une librairie pour une utilisation ultérieure dans d'autres programmes. L'environnement peut alors utiliser les capacités de multi-fenêtrage des interfaces graphiques pour afficher les détails des différents composants, sachant qu'un trop grand nombre de fenêtres ouvertes peut désorienter et irriter l'utilisateur (Prograph alloue une fenêtre à chaque nouvelle méthode, LabVIEW fait la même chose pour chaque *Virtual Instrument (VI)*). Dans les langages de programmation par démonstration, les macros (séquences d'actions enregistrées dans le but d'être reproduites automatiquement par le système) représentent une autre forme d'abstraction procédurale.

L'abstraction sur les données est plus délicate si l'on veut conserver la propriété de concrétude (voir section 3.3) du langage. Il peut être difficile de préserver les propriétés de visibilité et d'interactivité avec des types de données abstraits. LabVIEW propose certaines abstractions, comme les séquences et tableaux de nombres, en modifiant l'aspect (épaisseur, couleur) des canaux de données. L'utilisateur est guidé dans la gestion de ces types par une aide visuelle au moment de connecter des ports entre eux, ces derniers ayant un aspect différent quand ils ont besoin de types abstraits comme les tableaux ou les séquences.

Parmi les travaux de recherche sur la scalabilité des langages visuels, on peut citer Gorlick et Quilici [108] qui proposent un environnement de programmation visuel dans lequel les programmes consistent

en des hiérarchies de réseaux de composants appelés *weaves*. L'environnement utilise le multi-fenêtrage, des fonctions de zoom, et est capable de proposer les composants appropriés au fur et à mesure que le programmeur construit son programme, ce qui est intéressant lorsque le nombre de composants à la disposition du programmeur est important. Jamal et Wenzel [128] ont étudié l'utilisabilité de LabVIEW dans le cas d'une application de taille importante et ont obtenu de bons résultats, LabVIEW proposant des abstractions intéressantes (*VI*s), une bonne lisibilité et permettant un prototypage rapide.

Il reste encore certains problèmes à résoudre, comme la possibilité d'effectuer des recherches dans le code visuel d'un programme : il est facile de spécifier un ensemble de chaînes de caractères au moyen d'une expression régulière, mais cela devient beaucoup plus difficile pour des objets graphiques. De même, les fonctions de couper-copier-coller ou de remplacement ne sont pas évidentes à traiter. Citrin propose dans [53] des fonctions de couper-copier-coller pour éditeurs graphiques basées sur une connaissance de la syntaxe et de la sémantique du langage.

3.4.2 Facilité d'utilisation

Plusieurs études ont été effectuées afin d'évaluer de manière empirique la facilité d'utilisation des langages visuels. Ces études se sont surtout intéressées à la compréhension des programmes, à leur création et au debuggage.

Afin de pouvoir évaluer les environnements de programmation visuelle, T. Green et M. Petre proposent une technique basée sur les dimensions cognitives [111]. Parmi ces dimensions, on trouve :

- *Closeness of mapping* : représente la distance entre le problème et le programme qui va le résoudre. Plus la distance est petite, plus la solution devrait être facile à exprimer. Cette distance est très petite dans LabVIEW lorsque celui est utilisé pour créer des éléments électroniques virtuels pour l'acquisition et le traitement de données (ce pour quoi il a été conçu). Ce concept est à mettre en relation avec celui de communication entre l'homme et la machine, modélisée par les processus que P. Bottoni appelle *interpretation et materialization* dans sa modélisation de l'interaction des langages visuels [33, 31]. Ces processus sont basés respectivement sur les critères cognitifs de l'utilisateur et les critères programmatiques de l'ordinateur. L'image est ainsi définie comme la composition de texte et de graphiques représentant le message échangé par les acteurs du dialogue : "L'image est la matérialisation du sens que donne l'expéditeur à son message et qui doit être interprété par le récepteur en y associant un sens possiblement différent."
- *Diffuseness/Terseness* : représente le nombre de lexèmes nécessaires dans un programme pour exprimer une idée et l'utilisation de l'espace. On peut parler de compacité du code. Il est évident que plus le code prend de place et plus le nombre de lexèmes est élevé, plus le programme sera difficile à comprendre.
- *Error-proneness* : représente l'équivalent des erreurs de syntaxe et les omissions, comme l'oubli d'un point-virgule (;) entre deux instructions. L'étude réalisée sur les langages LabVIEW et Prograph a montré que ce genre d'erreurs étaient moins fréquentes dans les langages visuels. Le programmeur est souvent guidé dans les actions qu'il doit entreprendre par des 'indices' visuels, des aides contextuelles, ou des contraintes sur l'interaction [224, 65].

- *Secondary notation* : représente la facilité à intégrer des commentaires ou encore la lisibilité des programmes par l'utilisation de techniques remplaçant l'indentation utilisée couramment dans les programmes textuels.
- *Viscosity* : représente la résistance à des changements locaux, c'est-à-dire la facilité d'édition d'éléments existants, leur copie, leur suppression. Cette dimension dépend beaucoup des capacités de l'éditeur, mais globalement les langages visuels sont beaucoup plus "visqueux" que les langages textuels (le ratio entre les deux extrêmes est de 8:1).

Une étude [111] basée sur les dimensions cognitives appliquées aux langages LabVIEW et Prograph a mis plusieurs points en évidence. La construction de programmes est plus aisée avec ces *VPLs* qu'avec des *TPLs* car il y a moins de choses à exprimer (pas de délimiteurs de blocs, de séparateurs d'instructions, ni d'initialisation de variable). Par contre, les notations secondaires, mis à part les commentaires, ne sont pas très développées. Enfin, la facilité d'édition des programmes, mesurée par la viscosité, est très élevée. Dans un autre article, les mêmes auteurs montrent pourquoi les programmes visuels sont plus difficiles à lire que les programmes textuels [110], en réalisant une étude empirique sur un groupe d'utilisateurs de LabVIEW. Ces résultats en défaveur des langages visuels seraient dûs au fait que les structures complexes comme les boucles et les branchements conditionnels (comme représentés dans LabVIEW) sont plus difficiles à comprendre quand elles sont exprimées visuellement. Dans le cas de LabVIEW, ces structures de contrôle sont, de plus, très éloignées du paradigme sous-jacent (flot de données) et ne s'intègrent donc pas naturellement avec les autres constructions du langage. Il est néanmoins nécessaire de modérer ces résultats. En effet, les programmeurs sont familiers avec des expressions telles que `while(!s[i++]){...}` ce qui les rend faciles à lire ; mais il n'est pas évident qu'elles soient plus lisibles que des constructions visuelles dans l'absolu, c'est-à-dire pour une personne n'ayant jamais programmé.

Whitley [232] propose une autre étude dans laquelle des paradigmes visuels sont comparés à des approches textuelles des mêmes problèmes. Il en ressort que les représentations visuelles peuvent améliorer les performances quand elles sont bien utilisées, mais qu'elles ne sont pas supérieures à des approches textuelles de manière absolue, l'essentiel pour les programmeurs/utilisateurs étant que l'information soit présentée de manière consistante, organisée et explicite, textuellement ou graphiquement.

L'expressivité des *VPLs* a aussi été étudiée, notamment par des études comparatives des langages existants pour résoudre des problèmes donnés [115, 5]. Il en ressort que les *VPLs* sont rarement capables de couvrir une large gamme de tâches, mais sont par contre très adaptés à la résolution d'une famille restreinte de problèmes. Par exemple, Forms/3 est très adapté à la résolution de problèmes liés à des calculs pour gérer des comptes en banque, alors que LabVIEW va être beaucoup plus à l'aise pour simuler la descente d'une roue sur une pente avec une vue graphique de l'exécution. LabVIEW, bien que spécialisé dans l'acquisition et le traitement de données, est particulièrement expressif, de même que Prograph qui se veut dès le départ plus généraliste.

D'une manière générale, les études sur la compréhension des programmes exprimés visuellement n'ont pas fourni de résultats très clairs, contrairement aux études sur leur création. Sur le plan de la création, les langages visuels peuvent avoir un net avantage, améliorant de manière conséquente les temps de développement. Par exemple, des tests d'implémentation de langages iconiques en VAMPIRE ont donné

de bons résultats [124] : l'implémentation du langage NOVIS a pris trois heures au lieu de six mois (pour l'implémentation textuelle), celle de SUNPICT prenant quinze heures au lieu d'un an⁵. Dans un autre cas [11], une application d'acquisition et de traitement de données est développée en parallèle par deux équipes, l'une programmant en C, l'autre avec LabVIEW. L'équipe LabVIEW a été considérablement plus performante, notamment grâce à la possibilité de mieux communiquer avec le client demandeur de l'application. En effet, celui-ci est un expert dans le domaine du problème à résoudre mais pas en informatique. La métaphore employée par LabVIEW lui permet de comprendre le fonctionnement général du programme et il peut participer de manière active au développement avec le programmeur, ce qui n'est pas vraiment possible en C. En ce qui concerne la compréhension des programmes, les études n'ont pas fourni de résultats concluants quant à l'apport de la représentation visuelle. Mais au-delà de la représentation statique du programme, qu'elle soit textuelle ou visuelle, il faut considérer les capacités d'animation des environnements. Ainsi, il existe plusieurs systèmes dédiés à l'animation d'algorithme. Ces systèmes ont souvent un but didactique (ils sont utilisés pour illustrer les concepts aux étudiants en informatique, voir par exemple [123, 122]), mais des variantes peuvent aussi être utilisées à des fins de debuggage. Parmi les techniques d'animation, on trouve un système spécialisé dans l'animation d'opérations de manipulation d'arbres [199], une méthode d'animation de graphes [196], et une technique utilisant des graphiques 3D interactifs, la troisième dimension étant utilisée non pas pour afficher des objets tridimensionnels mais pour améliorer la qualité de la représentation d'informations bidimensionnelles [39]. L'animation d'algorithmes permet d'approfondir la compréhension des traitements, facilitant les opérations de debuggage. Elle est aussi liée à la propriété de *liveness* des programmes visuels [205, 234], terme qui désigne l'immédiateté avec laquelle un retour sémantique visuel est automatiquement fourni pendant l'édition. Il existe différents degrés de *liveness* :

- Niveau 1 : pas de retour fourni au programmeur,
- Niveau 2 : le programmeur peut demander manuellement un retour sur une partie du programme,
- Niveau 3 : mise à jour et propagation automatique des changements dus à l'édition,
- Niveau 4 : niveau 3 avec réaction automatique aux événements extérieurs (clics de souris, etc.).

Cette propriété améliore le confort du programmeur et sa productivité, lui indiquant automatiquement les conséquences de ses actions d'édition sur les éléments du programme. Ce retour n'est souvent que partiel, l'environnement n'explorant pas les changements potentiels sur les branches inactives du programme (par exemple la branche *else* d'un branchement conditionnel évalué à vrai au moment de l'édition).

3.5 Synthèse

La programmation visuelle est un domaine relativement jeune par rapport à la programmation textuelle, rendue possible seulement depuis l'apparition des interfaces graphiques performantes, c'est-à-dire supportant des résolutions capables d'afficher un nombre minimum d'objets et différentes vues simultanément. Le but est d'explorer de nouveaux paradigmes afin de rendre plus accessible la tâche de programmation, mais aussi afin de construire de meilleurs langages, et aussi afin d'améliorer la vitesse de spécification des programmes en utilisant les avantages d'une représentation graphique (syntaxe moins lourde, sémantique plus riche des primitives visuelles, retour visuel immédiat par animation, . . .). Les langages de programmation visuels sont souvent confondus avec les environnements de programmation

⁵Ces résultats sont biaisés du fait que la version textuelle existait déjà au moment de l'implémentation en VAMPIRE ; mais les différences restent néanmoins significatives.

visuels (*VPEs*), qui fournissent une interface graphique et des éléments visuels pour la génération de programmes textuels. Les *VPEs* ne sont pas des langages de programmation visuels, bien qu'ils bénéficient des avancées venant de la recherche dans le domaine.

Malgré un certain attrait pour leur côté intuitif, les langages de programmation visuels ne rencontrent pas aujourd'hui de grand succès commercial, mis à part quelques exceptions comme LabVIEW (succès relatif) et les tableurs comme Microsoft Excel. Ceci est en partie dû au fait qu'ils rencontrent des problèmes importants et difficiles à résoudre, comme la résistance au facteur d'échelle, le fait que l'on ne soit parfois pas habitué aux paradigmes utilisés (et à la multi-dimensionnalité), ou encore les difficultés d'abstraction et de représentation de certaines données ou structures de contrôle. D'autre part, certains avantages prétendus des langages visuels sont fortement controversés, comme par exemple la lisibilité des programmes. Mais le problème ne vient pas tant de la représentation graphique en tant que telle que du paradigme utilisé. Ainsi, un programme LabVIEW est en général lisible, alors qu'un programme VIPR devient très vite incompréhensible, au moins pour un utilisateur non expérimenté.

Au-delà de la représentation statique, il faut aussi tenir compte de l'environnement d'édition et d'exécution accompagnant le langage. Comme nous l'avons vu dans la section sur la visualisation, il est important d'utiliser au mieux les variables visuelles, de tenir compte de concepts comme la continuité perceptuelle et de fournir des mécanismes de navigation évolués. C'est dans cette optique que nous proposons XVTM, une boîte à outils pour la construction d'interfaces graphiques en 2.5 dimensions (chapitre 5). Les fonctions d'édition textuelles telles que la recherche d'éléments au moyen d'expressions régulières, le remplacement, et les mécanismes de copier/coller sont simples. Mais fournir des fonctions équivalentes⁶ dans un environnement visuel en minimisant la viscosité n'est pas trivial. Par exemple, dans le cas d'un éditeur de graphe, comment doit se comporter la fonction de copier/coller par rapport aux arcs lors de la copie d'un noeud ? Une partie de ces problèmes est étudiée dans IsaViz, l'éditeur de graphes RDF basé sur la XVTM et présenté dans le chapitre 6.

La programmation visuelle remet en cause un grand nombre de choses établies par les langages textuels, et ce serait sans doute une erreur de vouloir simplement transférer les éléments et structures textuelles en versions visuelles plus ou moins équivalentes (comme c'est souvent le cas en *controlflow*). Il semble plus intéressant d'envisager des choses radicalement différentes, en réinventant si nécessaire les blocs de base afin de tirer pleinement partie des avantages que procure la programmation visuelle (possibilité d'animation, multi-dimensionnalité, sémantique riche, concrétude) tout en se préoccupant des problèmes tels que la scalabilité et la représentation des abstractions. Une solution intéressante réside probablement dans l'étude de langages mixtes mélangeant des sections de code textuelles et visuelles (par exemple [89]). Aussi, il semble assez difficile de créer un langage visuel généraliste ; tous les langages réellement utilisables le sont dans les domaines pour lesquels ils ont été créés au départ. Il paraît donc assez réaliste de concentrer la recherche sur des langages assez ciblés tout en essayant de conserver un maximum d'expressivité. C'est la voie que nous explorons avec VXT, notre langage de programmation visuel dédié à la spécification de transformations de documents XML (chapitre 4).

⁶SWYN [26] permet la spécification d'expressions régulières de manière visuelle, mais ces expressions régulières portent sur des données textuelles ; il serait intéressant de pouvoir exprimer des expressions régulières portant sur des expressions visuelles.

Programmation visuelle de transformations de documents XML

Dans les deux chapitres précédents, nous avons étudié deux domaines bien distincts, à savoir les transformations de documents structurés, et plus particulièrement de documents XML, puis les langages de programmation visuels, ainsi que les techniques de visualisation de données structurées. Nous présentons maintenant une tentative de rapprochement de ces deux domaines par l'application de techniques de programmation visuelle à la spécification de transformations de documents XML. Nous nous focalisons dans ce chapitre sur la définition théorique de VXT, un langage visuel pour la spécification de transformations de documents XML. Les problèmes relatifs à l'environnement d'édition de programmes VXT sont abordés dans la deuxième partie de ce mémoire (voir chapitre 7).

4.1 Présentation générale

4.1.1 Les différents types d'outils visuels

Les solutions de transformation existantes dans le cadre de la manipulation de documents XML sont pour la plupart textuelles. Il s'agit principalement du langage XSLT et des approches de plus bas niveau telles que DOM combiné avec un langage de programmation généraliste (par exemple Java). Sur la base de XSLT, certains outils proposent une interface graphique donnant accès à des fonctionnalités comme la représentation colorée des éléments du langage (*syntax highlighting*), des mécanismes d'exécution pas à pas pour la mise au point, ou encore une représentation graphique d'instance de document source sous la forme de structure hiérarchique qu'il est possible de développer à différents niveaux (typiquement à l'aide d'un *JTree* Java). Ces solutions apportent des fonctionnalités intéressantes du point de vue de l'utilisateur, mais ne sont qu'un premier pas dans la direction des techniques de programmation visuelle. En effet, dans tous ces outils, basés sur XSLT, la spécification de la transformation s'opère toujours de manière textuelle directement en XSLT et requiert donc de l'utilisateur une bonne connaissance du langage sous-jacent et de sa syntaxe. Ces outils s'apparentent donc plus à des *environnements* de programmation visuels (*VPEs* aussi appelés IDE pour Environnements de Développement Intégrés), qu'à des *langages* de programmation visuels (*VPLs*)¹. Nous pensons qu'il est intéressant d'explorer des voies plus spécifiquement visuelles quant à la représentation des structures à transformer et des programmes de transformation eux-mêmes. Les environnements de développement intégrés mis à part, il existe deux classes principales d'outils visuels pour la spécification de transformations XML : les environnements de programmation par démonstration et les langages de programmation. Nous commençons par présenter ces deux classes, avant d'introduire VXT (*Visual XML Transformer*, [183, 184]), notre langage de programmation visuel spécialisé dans la spécification de transformations de documents XML, qui appartient donc à la deuxième classe d'outils.

Programmation par démonstration et manipulation directe

La première classe est basée sur les techniques de programmation par démonstration (voir chapitre 3) : l'utilisateur, qui peut ne pas avoir de connaissances en programmation, donne des exemples de résultats désirés en se basant sur des instances de documents sources. L'outil infère alors un programme de transformation à partir de ces exemples. XSLWiz [126] permet la spécification de transformations simplement en mettant en correspondance des éléments de la structure source avec des éléments de la structure cible.

¹La différence entre un *VPE* tel que Visual C++ et un *VPL* a été établie dans le chapitre 3.

Ceci implique que la structure cible soit connue au préalable et relativement figée, c'est-à-dire variant peu par rapport à l'application de la transformation sur différentes instances sources. XSLbyDemo [138] est un autre système utilisant l'historique des actions utilisateur concernant l'édition d'un document pour créer une feuille de transformation XSLT comportant des règles généralisant ces actions. L'utilisateur édite ses documents non pas sous la forme d'une structure, mais sous une forme rendue, par exemple en HTML. Dans les deux cas, l'expressivité est limitée car ces outils ne permettent que l'expression de transformations basiques dans lesquelles n'interviennent pas d'opérations de restructuration complexes (au-delà du tri). Ce sont néanmoins des solutions intéressantes car accessibles à un grand nombre de personnes, y compris des non-programmeurs, puisque la spécification se fait en se concentrant sur le résultat désiré et non pas sur la manière d'obtenir ce résultat. On notera tout de même que ces solutions, basées sur des mécanismes d'inférence, ne sont pas infaillibles et peuvent parfois établir des généralisations incorrectes par rapport au souhait initial de l'utilisateur. Enfin, L. Villard propose dans ses travaux sur l'édition de documents adaptables [217] des éditeurs dédiés permettant de modifier des documents sources et les transformations qui leurs sont associées par manipulation directe du résultat formaté, qui peut être un document rendu ou une présentation multimédia [216] (il existe notamment un éditeur spécialisé dans l'édition de documents DocBook sous leur forme rendue en XHTML). Ces outils reposent sur le principe des transformations inverses et sur un moteur de transformation XSLT incrémental [218]. La technique incrémentale permet d'identifier les parties du document source et de la transformation affectées par les changements effectués par l'utilisateur, et de ne réexécuter en conséquence que certaines règles de transformation sur certains fragments du document source pour propager les modifications de l'utilisateur. Cette technique réduit ainsi le temps d'exécution de la transformation (comparé à une exécution complète de la transformation sur l'intégralité du document), chose très importante dans le cadre de l'édition interactive.

Langages de programmation visuels

La deuxième classe d'outils visuels pour la spécification de transformations de documents XML est composée des langages de programmation visuels à proprement parler, c'est-à-dire des langages représentant de manière explicite les programmes de transformation, qui doivent être spécifiés directement par l'utilisateur au moyen de constructions programmatiques (structures de contrôle telles que les branchements conditionnels et les boucle itératives, systèmes de règles, etc.). C'est dans cette classe que nous positionnons VXT puisqu'il s'agit d'un langage à base de règles de transformation exprimées visuellement. VXT est principalement de nature déclarative, même s'il propose aussi quelques structures de contrôle comme les itérations. Le modèle d'exécution est très proche de celui de XSLT, et, comme dans ce dernier, caché à l'utilisateur. Il consiste en un parcours de la structure arborescente source en profondeur d'abord, dans lequel le moteur de transformation essaye d'appliquer les règles définies par le programme aux nœuds rencontrés, sélectionnant la règle la plus spécifique dans le cas où plusieurs règles peuvent s'appliquer². Il existe à notre connaissance trois autres langages de programmation visuels spécialisés dans la transformation de documents XML, qui seront détaillés dans la section 4.5.

VXT se focalise principalement sur la représentation et la transformation de la *structure* des documents XML, à opposer au *contenu* de ces documents, même s'il existe quelques primitives pour le traitement

²Le modèle est en réalité plus subtil, le programmeur ayant un certain contrôle au niveau de la spécification des nœuds sur lesquels les règles doivent être évaluées. Pour plus de détails, voir la section 2.4.1.

de la valeur des nœuds de type #PCDATA. C'est en effet au niveau de la dimension "structure" du document que nous pensons pouvoir tirer le plus de bénéfices des techniques visuelles. Par rapport aux outils textuels standard, l'approche visuelle offre potentiellement :

- une amélioration de la perception de la structure par l'utilisateur en tirant parti des capacités graphiques des environnements visuels au niveau de la présentation mais aussi de la navigation dans cette structure. Cela permet de réduire l'effort mental que doit fournir l'utilisateur pour manipuler les différents concepts auxquels il est confronté ;
- une simplification du processus de spécification des transformations par une interface guidant ou contraignant l'utilisateur dans ses actions, et augmentant ainsi sa productivité par la prévention de certains types d'erreurs. Cet avantage, couplé au précédent, peut potentiellement mettre l'outil à la portée d'un plus grand public comparé aux environnements précédents, bien que ce ne soit pas là notre but principal³ ;
- l'utilisation des capacités des environnements visuels pour proposer des mécanismes d'aide à la mise au point comme l'évaluation progressive de parties du programme.

4.1.2 VXT : un langage visuel spécialisé

Il ressort de l'étude menée dans le chapitre 3 qu'une des démarches les plus efficaces concernant la conception de langages de programmation visuels est la création de langages spécialisés dans des domaines bien déterminés, à opposer aux langages généralistes manipulant des concepts abstraits (variables, procédures, structures de contrôle) qui se prêtent mal à une représentation visuelle. Nous avons vu quelques exemples de tels langages, comme LabView, spécialisé dans l'acquisition et le traitement de signaux et basé sur la métaphore des circuits électroniques, ou encore les langages de requête pour interroger des bases de données.

Nous nous inscrivons ici dans ce courant, en proposant un langage de programmation visuel spécifiquement dédié à la création de transformations de documents XML. Cette approche va nous permettre concrètement de rendre plus intuitive la tâche de programmation, et aussi de cacher une partie de la complexité inhérente aux problèmes de transformation de structure, en proposant à l'utilisateur des composants de programmation et une représentation de ces composants bien adaptée à la résolution de ce problème car offrant des métaphores en relation directe avec le domaine du problème à résoudre.

L'inconvénient majeur de ce type d'approche est la limitation de l'expressivité du langage au seul domaine identifié. Ceci peut être envisagé comme un faux problème si l'on considère le fait que, dès le départ, nous sommes conscients de cette limitation et que nous n'envisageons pas l'utilisation du langage dans un domaine autre que celui considéré initialement. Cependant, cette restriction implique souvent l'impossibilité d'exprimer certains traitements plus complexes liés au domaine mais non prévus dans le langage, cantonnant ainsi l'utilisateur aux seules fonctionnalités proposées. Il s'agit là d'un problème typique des langages de haut niveau comme XSLT, faciles à utiliser mais limités dans leur expressivité et non extensibles du fait qu'ils ne proposent pas de constructions programmatiques de bas niveau qui permettraient potentiellement d'exprimer ces traitements. Il n'existe pas de solution simple et

³VXT est principalement destiné à des utilisateurs ayant un minimum de connaissances dans le domaine des transformations XML.

générale à ce problème. Nous le prenons en compte dans VXT en proposant la spécification de certaines constructions absentes du langage directement sous forme textuelle : le but des langages visuels n'est pas d'éliminer complètement le texte [43], mais au contraire d'intégrer [8, 89, 114] au mieux les deux représentations (textuelle et visuelle).

4.1.3 Des langages cibles multiples

VXT a été conçu dans l'optique de pouvoir exécuter des transformations de documents XML directement depuis l'environnement d'édition associé au langage, mais aussi dans l'optique de pouvoir exprimer des transformations et de les exporter vers d'autres langages. Ainsi, il est possible de générer, pour un programme VXT donné, une transformation XSLT ou un programme Circus. Comme nous l'avons vu dans le chapitre 2, Circus se situe à un niveau d'abstraction intermédiaire entre XSLT et DOM combiné à un langage généraliste. Il en résulte que Circus a, d'une manière générale, un pouvoir expressif plus important que XSLT [185]. Cependant, VXT étant basé sur une première version de Circus et utilisant un modèle XML restreint (i.e. *XMLTree*, voir chapitre 2), l'expressivité est plus limitée du côté de Circus que de XSLT sur certains points, et plus particulièrement au niveau des axes tels que *parent*, *ancestor*, *descendant*.

La solution la plus simple permettant de proposer des fonctions d'exportation de VXT à la fois vers Circus et XSLT consiste à identifier, du point de vue de l'expressivité, l'intersection des deux langages et à proposer dans VXT des constructions programmatiques permettant d'atteindre un niveau d'expressivité moindre ou équivalent à cette intersection. Mais nous nous limiterions dans ce cas à un langage trop simple et peu expressif, donc sans véritable intérêt. Nous avons préféré proposer une sorte d'union entre les deux langages. Il ne s'agit pas d'une véritable union, puisque pour les deux langages nous ne retenons qu'un sous-ensemble des constructions existantes. Certaines constructions étant présentes uniquement dans un des langages, il en résulte que, en proposant cette union de sous-ensembles de constructions, un programme VXT peut ne pas être exportable vers l'un ou l'autre des langages cibles. Pour résoudre ce problème, nous proposons dans l'environnement d'édition associé à VXT deux modes de spécification ; l'un est dédié à Circus, l'autre à XSLT. Ces deux modes sont très proches, interdisant simplement certaines actions en fonction du mode sélectionné, de manière à garantir l'exportabilité du programme vers le langage cible choisi. Cette solution n'est cependant pas très satisfaisante (un mode unique nous aurait permis de nous abstraire complètement des langages d'exportation) et nous verrons dans la section 7.5 que le passage à la version 2 de Circus, offrant une plus grande expressivité, permettra de résoudre ce problème.

Nous proposons dans un premier temps un langage visuel pour la représentation de DTD et d'instances de documents XML, puis, dans un deuxième temps, le langage VXT pour la spécification de transformations, qui reprend les bases définies dans le premier langage. Nous commençons par introduire VXT de manière informelle, en illustrant les différentes constructions du langage par des exemples d'utilisation, puis nous proposons une définition formelle de celui-ci ainsi que du processus de traduction vers le langage XSLT. Cette étude formelle, déjà entamée dans le cadre de la définition du langage de représentation de DTD [221], servira de base à l'établissement de deux propriétés concernant la fonction de traduction de VXT vers XSLT : sa complétude par rapport au langage VXT, et sa correction syntaxique

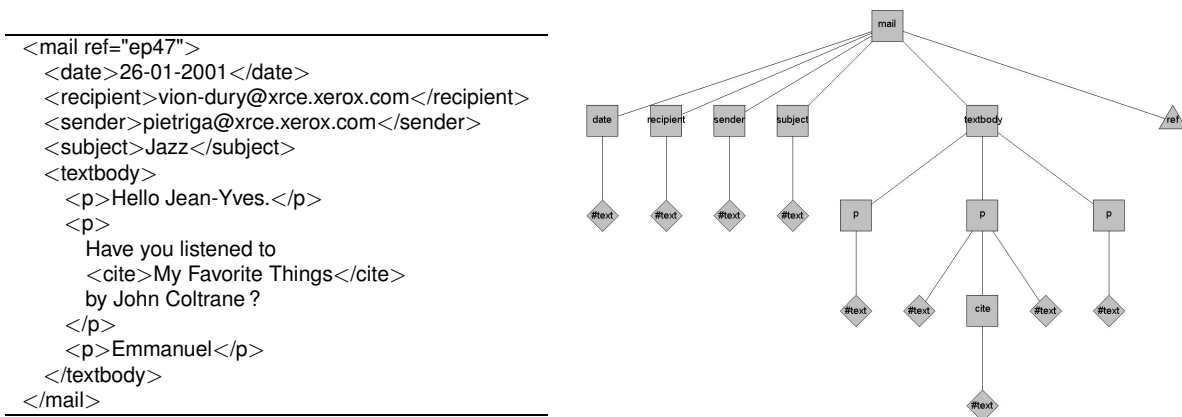


FIG. 4.1 : Représentations textuelle et visuelle d'un document XML

par rapport aux feuilles de transformation XSLT engendrées. Enfin, nous comparerons VXT aux autres travaux proposant une approche visuelle de la spécification de transformations de documents XML.

4.2 Langage visuel pour la représentation de documents et de classes de documents

En tirant parti des propriétés des représentations graphiques, un formalisme visuel unifié permet à la fois de représenter les documents XML, les classes de documents XML (DTD) ainsi que les programmes de transformation. Nous commençons par présenter ici un langage visuel pour la visualisation d'instances de documents XML et de DTD, avant de présenter dans la section suivante le langage de règles qui reprendra les bases définies ici.

4.2.1 Intérêt de la représentation visuelle

Même si les sauts de ligne et l'indentation amènent un peu de clarté dans les représentations textuelles de structures telles que les arbres et les graphes, les représentations graphiques de ces mêmes structures (figure 4.1) semblent être plus facilement appréhendées par les utilisateurs. Ceci peut être expliqué par plusieurs propriétés :

- Les relations structurales sont rendues plus explicites : par exemple, dans la figure 4.1, les relations parent-fils entre les nœuds sont représentées par des liens. Cette propriété est encore plus intéressante dans le cas des graphes, pour lesquels les notations secondaires des modes textuels ne sont pas suffisantes pour refléter même indirectement la structure (l'unidimensionnalité des modes textuels empêche de toute façon la représentation explicite des relations d'un graphe).
- La multidimensionalité permet de s'affranchir du besoin d'une balise ouvrante et d'une balise fermante pour les éléments non vides : un seul objet graphique est nécessaire pour représenter un élément. De plus, les constructions syntaxiques telles que les chevrons et les barres obliques indiquant une balise de fermeture sont éliminées, leur rôle étant pris en charge par des constructions graphiques (bordure et placement des objets), ce qui contribue à simplifier la représentation.

- L'utilisation de différentes dimensions perceptuelles pour la représentation et la classification des différents types de nœuds permet à l'utilisateur de traiter plusieurs informations en parallèle et augmente ainsi sa vitesse de compréhension (nous reviendrons plus en détails sur les choix de représentation dans la section suivante).
- Enfin, la multidimensionnalité de la représentation permet un placement plus fin des éléments graphiques comparé aux simples sauts de ligne et mécanismes d'indentation des modes textuels. Ce placement plus fin facilite la perception de la structure globale et détaillée du document.

Ce type de représentation visuelle, focalisé sur la structure XML, est par contre assez peu adapté à la représentation du texte contenu dans les documents. Ceci n'est pas un problème majeur dans notre cas puisque nous nous intéressons surtout aux transformations de structure et non pas de contenu, contenu qui reste de toute façon accessible dans une représentation parallèle.

4.2.2 Choix d'une méthode de représentation

Il existe différentes manières de représenter graphiquement des structures de données complexes. Il est possible d'utiliser des techniques de rendu 2D ou 3D, d'incorporer la dimension temporelle dans la représentation, ou encore de proposer des fonctionnalités pour développer ou rétracter des parties de la structure (exemple de la vue arborescente du système de fichiers dans *Windows Explorer* ou du *JTree* Java). Nous présentons ici les différentes possibilités que nous avons explorées et les raisons qui nous ont conduit à retenir une représentation basée sur une variante des *treemaps*.

La représentation doit permettre d'unifier de manière assez naturelle les différentes structures manipulées, malgré le fait qu'elles sont à des niveaux d'abstraction différents (les DTDs sont des grammaires décrivant des classes de documents XML). Elle doit de plus être assez résistante au facteur d'échelle (scalabilité de la représentation) puisque l'utilisateur pourra vouloir visualiser des instances de documents et des DTDs complexes. Nous avons assez naturellement opté pour une représentation bidimensionnelle non déformée, principalement pour la raison que nous n'allons pas nous contenter de visualiser la structure, mais allons utiliser la représentation de cette structure de différentes manières dans le cadre de la spécification des transformations. Les représentations tridimensionnelles ou déformées (arbres hyperboliques, ...) sont intéressantes surtout dans le cadre de la visualisation, mais se révèlent moins efficaces quand il s'agit de manipuler les structures (elles peuvent par contre servir de représentation secondaire aidant à l'orientation et à la navigation dans l'espace).

Agencement spatial

Les structures arborescentes sont souvent représentées au moyen de diagrammes composés de nœuds et d'arcs reliant ces nœuds, comme dans la figure 4.1. Mais il existe une autre méthode d'agencement appelée *treemap*. Cette méthode, proposée par Shneiderman [130] et reprise dans d'autres travaux [215], consiste en la représentation des relations père-fils par l'emboîtement des enfants à l'intérieur de leur parent. Cette méthode a principalement été utilisée pour la visualisation de structures arborescentes de grande taille, comme les systèmes de fichiers (voir figure 4.2). Contrairement aux méthodes standard à base de diagrammes nœuds/arcs, dans lesquelles plus de 50% des pixels ne sont pas directement utilisés pour la représentation de la structure et font partie du fond d'écran, les *treemaps* occupent la totalité de

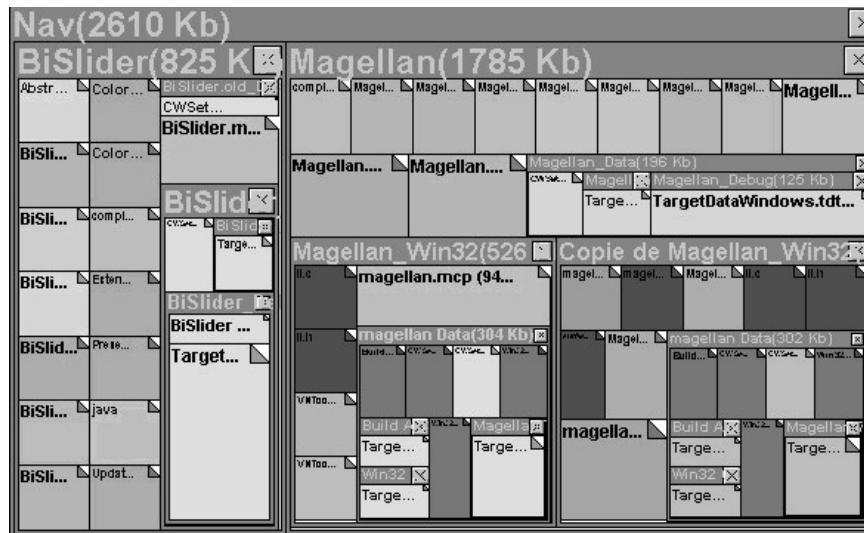


FIG. 4.2 : Exemple de représentation d'une structure arborescente (système de fichiers) sous forme de treemap

l'espace alloué, faisant ainsi meilleur usage d'une ressource limitée. Aussi, combinées à des fonctions de navigation évoluées, les *treemaps* permettent à l'utilisateur d'appréhender la structure de manière plus intuitive en spécifiant le niveau de détail souhaité simplement en se déplaçant dans l'espace (par exemple en modifiant son niveau de zoom, c'est-à-dire son altitude d'observation). Cette méthode présente cependant quelques inconvénients qui peuvent avoir un impact plus ou moins important suivant l'utilisation que fait l'utilisateur de la représentation (absence d'ordonnancement des éléments dans leur parent, taille variable des éléments de même profondeur, variations dans le flot de développement des composants d'un élément à un autre). Certains de ces inconvénients peuvent être contournés en modifiant la méthode d'agencement initiale. Il existe en effet différentes versions des *treemaps* en ce qui concerne la taille et l'agencement relatif des nœuds à l'intérieur de leur parent. Nous proposons ici une nouvelle variante qui tient compte de la spécificité de notre cas.

La technique standard ne rend pas très explicite l'ordonnancement des fils d'un élément, les plaçant sur plusieurs lignes à l'intérieur du rectangle parent. Cette politique permet d'optimiser l'agencement, et donc l'utilisation de l'espace, mais n'est pas satisfaisante dans notre cas. Nous avons en effet expérimenté cette solution dans un prototype de visualisation de structure XML, dans lequel le placement des fils s'effectue sur plusieurs lignes. De cette manière, il y a un nombre égal d'éléments alignés verticalement et horizontalement et la structure se développe uniformément dans les deux directions (par exemple, si un élément contient huit fils, l'algorithme de placement en positionnera trois sur une première ligne, trois sur une deuxième ligne, et les deux derniers sur une troisième ligne). Le résultat était satisfaisant du point de vue de l'occupation de l'espace mais présentait un problème majeur au niveau de la perception de la structure. Le placement des fils d'un élément sur différentes lignes introduit en effet une discontinuité dans le flot des nœuds qui est dans notre cas ordonné. De plus certains nœuds peuvent se retrouver

éloignés de leurs frères. Si l'on reprend l'exemple précédent où un élément contient huit fils, le troisième nœud, à la fin de la première ligne, est loin de son frère suivant, le quatrième nœud, qui se trouve au début de la deuxième ligne. À l'opposé, certains nœuds qui sont éloignés dans la structure logique du document sont rapprochés dans la représentation de cette structure. Ces différents problèmes contribuent à donner l'impression à l'utilisateur que le contenu d'un élément est un ensemble non ordonné, alors qu'au contraire l'ordonnancement est important dans les documents. Nous verrons aussi dans le chapitre 7 que la représentation graphique des instances et DTDs sert en particulier à évaluer visuellement les parties gauches de règles de transformation. La discontinuité dans le flot des éléments rendrait cette utilisation de la représentation du document source impossible, puisque les parties gauches de règles, représentées sans discontinuité en surimpression, ne s'adaptent pas visuellement à la structure source. Pour ces raisons, notre variante ne développe les fils d'un élément que sur une seule ligne, de gauche à droite, de manière à éliminer toute discontinuité et à illustrer le fait que ces fils sont ordonnés (voir figure 4.4).

L'autre modification concerne la taille des nœuds. Les représentations à base de *treemap* ont l'inconvénient de représenter l'information de profondeur d'un nœud de manière assez peu explicite, puisqu'elle n'est reflétée que par le niveau d'emboîtement du nœud dans la structure géométrique. Cette information peut être difficile à extraire de la représentation graphique, et requiert de la part de l'utilisateur un effort mental qui rend la perception globale de la structure moins intuitive. Dans le cas des représentations à base de diagrammes nœuds/arcs, les nœuds de même profondeur absolue sont souvent alignés (figure 4.1), ce qui rend l'extraction de cette information bien plus facile. Nous proposons ici d'adapter la taille des éléments en fonction de leur profondeur dans l'arbre. Cet indice visuel, même s'il est un peu moins évident que l'alignement mentionné précédemment, devrait améliorer la perception globale de la structure par l'utilisateur. L'algorithme en charge de calculer l'agencement des éléments assigne donc la même hauteur à tous les nœuds ayant la même profondeur absolue. La hauteur assignée à l'ensemble des nœuds de profondeur n est la hauteur de l'élément le plus haut à cette profondeur, la taille initiale des éléments étant calculée en fonction du contenu et donc de la taille des éléments de profondeur $n + 1$ (puisque les éléments doivent contenir complètement l'ensemble de leurs fils).

La figure 4.4 donne deux exemples d'utilisation de cette méthode, qui représente un compromis : elle fournit une représentation plus fidèle de la structure, mais consomme aussi plus d'espace d'affichage que les *treemaps* standard. Cet inconvénient est cependant compensé par le fait que nous proposons une représentation dynamique de la structure offrant de bonnes capacités de navigation et de zoom, qui sont aussi rendues nécessaires du fait que les nœuds situés aux niveaux les plus profonds peuvent être difficiles à percevoir depuis la racine. Ces fonctionnalités seront fournies par XVTM, la boîte à outils pour la conception d'interfaces zoomables décrite dans le chapitre 5.

Note : avant de retenir cette méthode d'agencement, nous avons envisagé une autre option consistant à appliquer la politique de taille égale pour tous les nœuds de même profondeur aussi bien à la hauteur qu'à la largeur. Cette méthode s'est rapidement révélée inutilisable, du fait que la représentation a déjà une assez forte tendance à se développer horizontalement. Ainsi, pour certaines configurations de structures XML, cette alternative utilisait parfois des quantités importantes d'espace horizontal supplémentaire sans


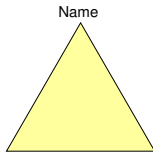
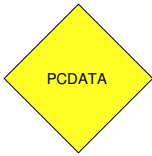
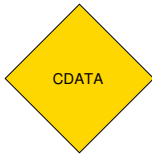
| Type de nœud | Élément | Attribut | #PCDATA | Section CDATA |
|--------------------------|---|---|--|---|
| Représentation graphique |  |  |  |  |

FIG. 4.3 : Représentation des types de nœud des arbres XML

nécessité du point de vue du contenu mais uniquement pour répondre à la contrainte d'égalité de largeur des éléments de même profondeur.

Dimensions perceptuelles

Nous avons vu dans le chapitre 3 l'importance du choix de dimensions perceptuelles adéquates pour représenter différents types de données en fonction de leur nature. Dans le cadre des documents XML, nous nous intéressons principalement à des structures arborescentes. La seule relation existant entre les nœuds d'un arbre XML est le lien père-fils, si l'on considère, comme c'est le cas dans DOM, que les attributs attachés à un élément ne font pas partie de la structure principale. Cette relation sera donc relativement facile à représenter puisqu'elle est unique. Il existe par contre différents types de nœuds : éléments, feuilles texte (section CDATA, #PCDATA), attributs, *processing-instructions*, auxquels il faut ajouter les constructions spécifiques aux DTD encodant la cardinalité des éléments, les séquences et les choix.

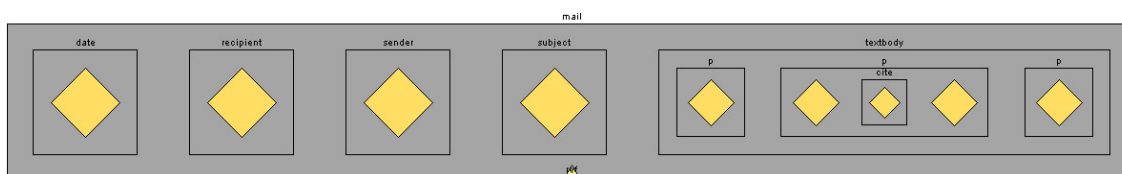
Pour représenter les différents types de nœuds et leurs caractéristiques, nous proposons l'utilisation des dimensions perceptuelles suivantes. Le type de nœud (élément, attribut, etc.), une information qualitative, est encodé par la forme géométrique de l'objet graphique associé au nœud. Des teintes différentes sont utilisées pour préciser⁴ ce type, en conservant la même forme. Comme nous l'avons vu dans la section précédente, les relations père-fils sont représentées par l'emboîtement des nœuds. La profondeur d'un nœud dans la structure (information quantitative) est alors représentée par sa hauteur. Le tableau de la figure 4.3 illustre les différentes constructions utilisées pour la représentation d'instances de documents XML, et la première *treemap* de la figure 4.4 donne un exemple d'instance de document XML.

Les DTD nécessitent des constructions abstraites additionnelles (séquence et choix) ainsi que l'ajout d'information sur la cardinalité des éléments (constructions ?, * et +). Les séquences sont représentées par des rectangles bleu contenant les éléments de la séquence ordonnés de gauche à droite ; les choix par un alignement vertical de rectangles verts contenant les différentes options du choix. La cardinalité est encodée par le style du contour des nœuds (continu (*solid*) ou discontinu (*dashed*)) et par l'utilisation dans certains cas d'un rectangle additionnel. Ces deux styles peuvent être assignés indifféremment à tous les types de nœuds, y compris les séquences et les choix puisque les DTD autorisent la spécification

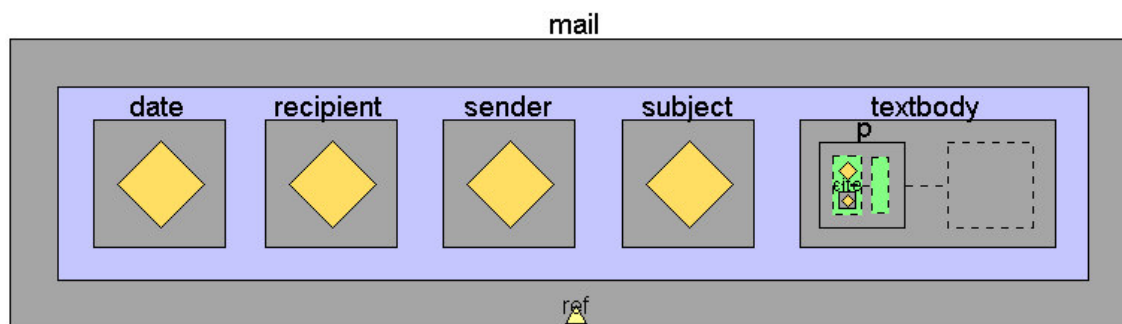
⁴Les feuilles de type texte peuvent être soit des sections CDATA soit des sections PCDATA, les DTD (et autres langages de schéma) peuvent aussi préciser le type du contenu des attributs (et des feuilles texte pour certains langages) ; il s'agit dans tous les cas d'une information qualitative.

| Cardinalité | Indicateur d'occurrence | Représentation | Structure | Représentation |
|-------------|-------------------------|----------------|-----------|----------------|
| 0..1 | ? | | Séquence | |
| 1 | néant | | | |
| 0..n | * | | Choix | |
| 1..n | + | | | |

Constructions additionnelles pour les DTD



Représentation visuelle d'une instance de document XML mail





Représentation visuelle de la DTD pour les documents mail

| | | | |
|----|------------|-----------|---|
| 1. | < IELEMENT | mail | (date,recipient,sender,subject,textbody)> |
| 2. | < IELEMENT | date | (#PCDATA)> |
| 3. | < IELEMENT | recipient | (#PCDATA)> |
| 4. | < IELEMENT | sender | (#PCDATA)> |
| 5. | < IELEMENT | subject | (#PCDATA)> |
| 6. | < IELEMENT | textbody | (p)+> |
| 7. | < IELEMENT | p | (#PCDATA cite)*> |
| 8. | < IELEMENT | cite | (#PCDATA)> |
| 9. | < !ATTLIST | mail ref | CDATA #REQUIRED> |

DTD pour les documents mail

FIG. 4.4 : Visualisation d'une instance de document mail et de la DTD correspondante

d'information de cardinalité au niveau de ces constructions. Le rectangle additionnel  symbolise la possibilité d'avoir entre 0 et n occurrences en plus de la première. Nous avons exploré dans un premier temps une autre alternative, utilisant la représentation proposée par Blackwell dans son langage visuel pour expressions régulières Perl [26] qui consistait à superposer avec un léger décalage trois icônes quasiment identiques comme suit : . Mais cette représentation avait l'inconvénient de donner l'impression que le flot d'éléments se développait suivant l'axe perpendiculaire au plan de l'écran. Nous avons donc opté pour la méthode précédente qui illustre mieux le fait que le flot se développe horizontalement. Le tableau de la figure 4.4 résume les différentes constructions spécifiques aux DTD, qui s'ajoutent à celles définies dans le tableau 4.3. Cette même figure donne un exemple d'instance de document *mail* et la DTD correspondante accompagnée de sa version textuelle.

Ce langage visuel sert de base au langage VXT pour la spécification des règles de transformation. Pour plus d'informations, le lecteur intéressé peut se référer à l'article [221] présenté en annexe B qui en propose une étude formelle. Cette étude servira par ailleurs de base partielle à la définition formelle de VXT, développée dans la section 4.4.

4.3 Langage de transformation visuel à base de règles

Nous venons de présenter un langage visuel pour la représentation d'instances de documents XML et de DTD qui sera utilisé, comme nous le verrons dans le chapitre 7, dans le but de proposer au concepteur de transformations une représentation des structures de documents sources et/ou cibles qu'il aura à manipuler. Cela le dispense de garder une représentation mentale de ces structures et réduit ainsi l'effort cognitif qu'il a à fournir. Nous présentons maintenant une introduction à VXT (*Visual XML Transformer*) qui est suivie d'une définition plus formelle du langage dans la section suivante.

VXT permet la spécification de transformations sous forme de règles visuelles. La partie gauche d'une règle, appelée *VPME* pour *Visual Pattern-Matching Expression*, exprime des conditions de sélection et d'extraction des nœuds du document source. La partie droite, que nous désignerons par le terme de *production*, représente le fragment de structure produit dans le document cible lorsque la règle est déclenchée.

4.3.1 Expressions de sélection et d'extraction (*VPME*)

Le modèle d'exécution de VXT est très proche de celui de XSLT. Il s'agit de parcourir la structure du document source en profondeur d'abord et d'évaluer les règles de transformation par rapport aux nœuds rencontrés, en n'exécutant finalement pour un nœud donné que la règle la plus spécifique quant aux conditions qu'elle exprime par rapport à la sélection des nœuds (voir la section «Priorité des règles»). En plus de ces conditions de sélection, la *VPME* identifie aussi, de manière similaire aux expressions de filtrage Circus, des données à extraire par rapport aux nœuds vérifiant les conditions. Ces données⁵ seront transformées et incorporées dans le document cible.

⁵Il peut s'agir simplement de #PCDATA mais aussi de sous-arbres XML.

Filtrage visuel

Les opérations de sélection et d'extraction ont été regroupées en une seule expression représentant l'opération de filtrage, qui combine les deux précédentes. Cette unique expression nous permet d'utiliser la métaphore des filtres (ou masques) visuels. En effet, le choix d'un formalisme unifié pour la représentation des instances, des classes de documents et des règles de transformation a pour conséquence que les *VPMEs*, qui modélisent des contraintes structurales sur le contexte et sur le contenu des nœuds à sélectionner, sont visuellement proches des structures qu'elles sélectionnent, et même parfois identiques. Cette propriété de correspondance visuelle entre les structures et les expressions de filtrage nous permet alors de représenter les expressions de sélection et d'extraction, c'est-à-dire les *VPMEs*, comme des constructions graphiques qui s'adaptent visuellement aux parties de la structure qu'elles sélectionnent effectivement dans une instance. Nous verrons dans le chapitre 7 comment cette métaphore de filtres visuels est concrètement mise à profit.

Dans le chapitre 2, nous avons vu que Circus offre une construction programmatique textuelle similaire : le filtre, qui permet dans une même expression de spécifier des conditions (structurales) de sélection et d'identifier des données à extraire (qui seront stockées dans des variables typées). Les valeurs de ces variables peuvent alors être récupérées et modifiées dans la partie droite de la règle (la partie gauche correspond au filtre⁶). Nous avons vu dans le même chapitre que les règles XSLT (*template rules*) sont structurées de manière différente, puisque les opérations de sélection et d'extraction sont dissociées. La partie gauche d'une règle, c'est-à-dire l'attribut `match`, n'exprime que les conditions de sélection des nœuds (au moyen d'une expression XPath), alors que les expressions identifiant les fragments à extraire de l'arbre source sont spécifiées dans le corps de la règle (la partie droite) par une ou plusieurs expression(s) XPath associée(s) à des instructions de transformation.

Le regroupement des opérations de sélection et d'extraction peut présenter l'inconvénient de rendre plus complexe l'expression représentant le filtre. Par exemple, en XSLT, il serait possible, en modifiant la syntaxe XPath de manière significative, de combiner les expressions liées aux attributs `match` et `select`. Ceci rendrait cependant encore plus obscures des expressions déjà difficiles à lire et à spécifier. En Circus, une partie du problème est rejeté à l'extérieur de l'expression de filtrage puisque les conditions portant sur l'identification des données à extraire sont spécifiées en partie dans la déclaration de type de la variable servant de réceptacle à ces données et en partie dans des filtres additionnels cascades en partie droite de la règle principale. En ce qui concerne VXT, la multidimensionnalité de la représentation visuelle nous permet d'utiliser différentes dimensions perceptuelles pour spécifier les diverses caractéristiques des éléments composant une *VPME*. La syntaxe visuelle est ainsi moins chargée et les expressions restent faciles à comprendre, malgré la complexité supplémentaire apportée par la combinaison des opérations de sélection et d'extraction.

Terminologie et constructions de base des *VPMEs*

Cette partie utilise largement la notion de *nœud contextuel* définie par XPath/XSLT. Nous rappelons rapidement que ce terme désigne à son tour chaque nœud effectivement sélectionné par une expression

⁶Notons que Circus offre des propriétés de symétrie syntaxique entre structures et filtres, à rapprocher de notre propriété de correspondance visuelle entre structures XML et *VPMEs*.









| | | | |
|--|---|--------------------------------|--|
| Nœud contextuel | → | bordure bleu clair continue |  |
| Condition d'existence pour la sélection | → | bordure bleu foncé continue |  |
| Condition d'absence pour la sélection | → | bordure rouge continue |  |
| Nœud à extraire (s'il existe) mais dont l'existence n'est pas une condition de sélection | → | bordure bleu foncé discontinue |  |
| Père/fils | → | bordure fine |  |
| Ancêtre/descendant | → | bordure épaisse |  |
| Nœud à extraire | → | intérieur bleu translucide |  |
| Nœud non extrait | → | intérieur vide |  |

FIG. 4.5 : Représentation visuelle des propriétés des nœuds de VPME

XPath. Les nœuds descendant du nœud contextuel font partie du *contenu* alors que les nœuds ancêtres du nœud contextuel ainsi que les nœuds appartenant à d'autres parties de l'arbre forment quant à eux le *contexte* du nœud contextuel. Le lecteur désirant plus de précisions peut se reporter à la section 2.4.1.

Les VPMEs représentent des opérations de filtrage structurel appliquées à des arbres. Elles sont elles-mêmes structurées de manière arborescente, et par conséquent constituées de nœuds terminaux (feuilles) et non terminaux. Les opérations de sélection et d'extraction faisant partie de la même expression, certains nœuds de VPME représentent soit des conditions de sélection, soit des identifiants de données à extraire, ou encore les deux à la fois⁷. Notons que les identifiants de données à extraire sont aussi exprimés sous la forme de conditions de sélection, à la différence que ces conditions ne sont pas prises en compte dans le processus de sélection du nœud contextuel (sauf si cela est spécifié explicitement, par exemple quand un nœud est à la fois condition de sélection et identifiant de données à extraire).

⁷Ceci est rendu possible par la multidimensionnalité de la représentation qui permet d'utiliser un unique objet graphique pour représenter des fonctions différentes du même nœud.

La figure 4.5 résume les différentes propriétés qui peuvent être assignées aux nœuds d'une *VPME* et leur représentation graphique. Certaines de ces propriétés peuvent être combinées dans un même nœud. Elles ne sont cependant pas orthogonales, puisque les propriétés assignées à un nœud contraignent les autres propriétés qui peuvent être assignées à ce nœud ainsi qu'à ses descendants. Par exemple, il n'est pas possible d'exprimer une condition d'absence et un identifiant de données à extraire sur le même nœud, ces deux instructions étant incompatibles. Les combinaisons autorisées sont décrites précisément dans la section 4.4 par les quadruplets de style. Les restrictions imposées sur les propriétés des nœuds et de leurs descendants en fonction des propriétés déjà assignées sont quant à elles capturées par les critères de validation détaillés dans cette même section.

Sémantique des expressions

VXT reprend la notion de nœud contextuel de XPath/XSLT. Ainsi, une *VPME* est au minimum constituée d'un nœud contextuel. Le nœud contextuel est unique, donc, comme nous le verrons dans la formalisation du langage, toute *VPME* bien formée contient exactement un nœud contextuel, identifié par sa couleur bleu clair. Notons que le nœud contextuel peut être de n'importe quel type : élément, attribut ou nœud texte. Les autres nœuds au contour solide représentent alors des prédicats sur le nœud contextuel, c'est-à-dire des conditions à remplir par ce nœud en ce qui concerne son contexte et son contenu pour être sélectionné par la règle. Ces conditions peuvent porter sur l'existence (couleur bleu) ou l'absence (couleur rouge) de nœuds d'un certain type dans le contenu ou parmi les ancêtres du nœud contextuel.

Dans la représentation graphique, le nœud contextuel joue un rôle central. C'est en fonction de ce nœud que tous les autres sont interprétés. Ainsi, les nœuds au contour continu contenus dans le nœud contextuel représentent des prédicats sur ses descendants. Au contraire, les nœuds au contour continu englobant le nœud contextuel représentent des prédicats sur ses ancêtres. L'épaisseur de la bordure du nœud indique si la condition porte sur l'entourage direct du nœud (c'est-à-dire ses fils et son père), ou bien sur ses descendants et ses ancêtres. Le tableau de la figure 4.6 donne quelques exemples de *VPMEs* exprimant des conditions de sélection simples sur des documents DocBook tels que notre fragment de documentation décrivant les animations dans XVTM introduit dans le chapitre 2.

Il est important de souligner plusieurs subtilités quant à l'interprétation des *VPMEs*. La première concerne les axes ancêtre et descendant. Comme nous l'avons mentionné précédemment, le nœud contextuel joue un rôle central au sein de la *VPME* dans le sens où les autres nœuds sont interprétés en fonction de leur position par rapport à ce nœud. Ainsi, les prédicats portant sur le nœud contextuel, c'est-à-dire les autres nœuds au contour solide, sont interprétés suivant les axes ancêtre et descendant avec comme origine le nœud contextuel. Une autre solution aurait pu être d'interpréter tous les nœuds sur l'axe descendant avec comme origine la racine de la *VPME*. Cette différence a un impact sur l'interprétation du nœud contextuel et sur les nœuds qui l'englobent (i.e. ses ancêtres dans l'arbre de la *VPME*). Concrètement, l'interprétation de la *VPME* (7) de la figure 4.6 ne changerait pas puisque dans les deux cas le nœud `title` se trouve sur l'axe descendant. Par contre, l'interprétation de la *VPME* (8) est différente. Dans le cas non retenu (axe descendant ayant pour origine le nœud racine de la *VPME*), la traduction XSLT serait `//section/title` alors que la véritable traduction est `section//title`. Dans le premier cas, la bordure épaisse du nœud `section` est interprétée suivant l'axe descendant, alors que dans le deuxième cas elle est interprétée suivant l'axe ancêtre puisque le nœud `section` englobe le nœud contextuel. Cette





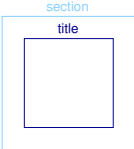
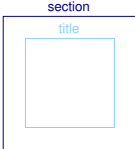
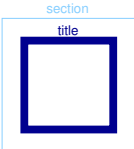
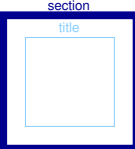
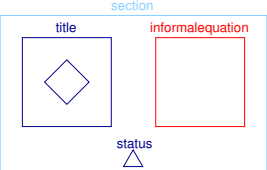
| VPME | Attribut match | | VPME | Attribut match | |
|---|--|-----|--|----------------|-----|
|  | section | (1) |  | * | (2) |
|  | @status | (3) |  | @* | (4) |
|  | section[title] | (5) |  | section/title | (6) |
|  | section[descendant::title] | (7) |  | section//title | (8) |
|  | section[title[text()] and @status and not(informalequation)] | (9) | | | |

FIG. 4.6 : Exemples de VPMEs simples

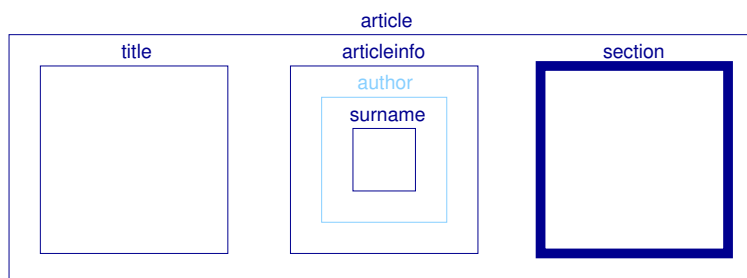


FIG. 4.7 : Exemple de VPME plus complexe

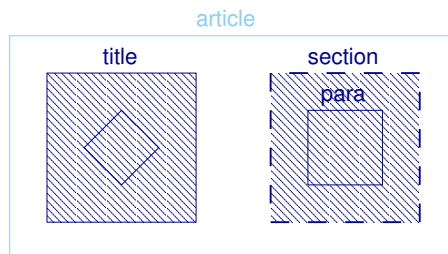


FIG. 4.8 : VPME contenant des instructions d'extraction

interprétation des VPMEs plaçant le nœud contextuel dans une position centrale nous a semblée la plus judicieuse pour fournir une expressivité élevée et garder un maximum de cohérence avec l'interprétation de la position des nœuds translucides, représentant les données à extraire.

Lié au point précédent, il existe une deuxième subtilité concernant l'interprétation des nœuds qui ne sont ni ancêtres ni descendants du nœud contextuel dans la VPME, c'est-à-dire les nœuds qui sont descendants de la racine de la VPME mais ne se trouvent ni sur le chemin reliant la racine au nœud contextuel, ni dans le contenu du nœud contextuel. Ces nœuds sont considérés comme des prédicats sur les ancêtres du nœud contextuel. Ainsi, leur interprétation ne doit pas se faire en fonction des axes ancêtre et descendant définis précédemment et ayant comme origine le nœud contextuel, mais seulement en tant que descendants de l'ancêtre du nœud contextuel auquel ils sont rattachés. Par exemple, si l'on considère la VPME de la figure 4.7 et plus particulièrement les nœuds `title` et `section` qui sont des instances du cas décrit ici, sa traduction en expression XPath est

```
article[title and descendant::section]/articleinfo/author[surname]
```

Nous invitons le lecteur désirant une définition précise et non ambiguë à se reporter à la section 4.4 portant sur la définition formelle du langage et notamment des VPMEs. Notons aussi que dans le cas de Circus ces subtilités d'interprétation n'existent pas, l'expressivité des filtres de Circus v1.0 nous restreignant à la création de VPMEs dont le nœud contextuel est toujours le nœud racine et ne contenant pas de nœud sur les axes ancêtre et descendant (seul l'axe *fils* est supporté).

Enfin, les nœuds translucides, qui correspondent à des données à extraire, peuvent aussi être des conditions de sélection. C'est le cas quand la bordure du nœud translucide est continue. Pour le cas où un nœud représentant des données à extraire ne doit pas être pris en compte comme condition de sélection, il existe des nœuds au contour discontinu. Ces nœuds ont obligatoirement un intérieur translucide, qui garantit leur raison d'être (un nœud au contour discontinu et à l'intérieur vide ne participerait pas à la sélection et ne serait pas extrait ; sa présence dans le filtre n'aurait pas de sens). Comme nous le verrons dans la section sur les productions de règles, la position des nœuds représentant des données à extraire, que nous appellerons instructions d'extraction, sera interprétée en fonction de leur localisation par rapport au nœud contextuel de la VPME, comme c'est le cas pour les expressions XPath associées aux attributs `select` des instructions XSLT. La figure 4.8 donne un exemple de VPME contenant deux instructions d'extraction. La première est aussi une condition de sélection car le contour du nœud à extraire (`title`)

est continu. La seconde (section) ne sera pas prise en compte dans la sélection car son contour est discontinu, de même que tous ses descendants qui n'agissent que comme des prédicats de sélection des données à extraire. Ainsi, la traduction XSLT de cette *VPME* est

```
article[title[text()]]
```

Priorité des règles

La recommandation XSLT [82] définit dans la section 5.5 une méthode de calcul de la priorité des règles dans le cas où plusieurs peuvent s'appliquer à un nœud de la structure source. Cette méthode tient compte des priorités assignées aux règles par le concepteur de la transformation et de la complexité de l'expression XPath dans le cas où les règles en conflit ont la même priorité ou n'ont pas de priorité explicitement spécifiée par le programmeur. En mode XSLT, ce calcul est pris en charge par le moteur XSLT sous-jacent selon la méthode préconisée et nous n'avons pas à nous en soucier. Par contre, il n'existe pas en Circus de notion équivalente. Nous devons donc nous-mêmes calculer les priorités des règles avant d'engendrer le code source Circus de la transformation. Ces priorités, calculées suivant la même méthode que celle préconisée par la recommandation XSLT afin de rester cohérent dans les deux modes, nous permettront de classer les règles dans l'ordre décroissant à l'intérieur du système d'action principal de la transformation lors de la traduction Circus.

4.3.2 Productions

La partie droite des règles de transformation VXT est appelée la *production*. Cette partie prend la forme d'une forêt XML qui représente les sous-arbres produits par le déclenchement de la règle. Il s'agit d'une séquence ordonnée d'arbres toujours représentés en utilisant notre variante des *treemaps*. Les composants de cette forêt sont agencés suivant un flot horizontal orienté de gauche à droite, ou vertical orienté de haut en bas. L'agencement horizontal est en théorie le meilleur puisqu'il est en accord avec la méthode de représentation standard, qui aligne horizontalement les nœuds frères (ce que sont les racines des sous-arbres composant la forêt). Cependant, d'un point de vue pratique, l'agencement vertical peut être intéressant pour la lisibilité de la règle si les sous-arbres ont une trop forte tendance à se développer horizontalement par eux-mêmes. Notons que les règles produisant assez souvent une forêt ne contenant qu'un seul arbre, ce problème est peu important.

Les arbres formant cette forêt correspondent à des fragments de la structure résultat et sont insérés dans celle-ci à chaque fois que la règle est déclenchée. L'insertion dans la structure cible se fait de manière similaire à XSLT (voir section 2.4.1), c'est-à-dire que les fragments produits par les instructions telles que `xsl:apply-templates` ou `xsl:value-of` remplacent, dans le fragment qui la contient, l'instruction qui les a engendrés. Les fragments sont composés de trois types de nœuds :

- les nœuds correspondant à de nouveaux éléments, attributs et textes produits dans le document résultat. Ce sont les nœuds de couleur gris clair, toujours associés à une étiquette (*label*) représentant le nom de l'élément, de l'attribut, ou le début de la valeur des nœuds texte.
- les nœuds correspondant au résultat de la transformation de nœuds extraits du document source. Ces nœuds sont colorés avec le même bleu que celui utilisé pour les nœuds représentant les instructions d'extraction des *VPMEs*. Ils ne sont pas labélisés puisqu'ils ne représentent que les em-

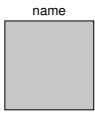






| Type | Représentation | | Type | Représentation | |
|-----------------|----------------|---|--|----------------|---|
| nouvel élément | gris |  | élément obtenu par transformation de données source | bleu |  |
| nouvel attribut | gris |  | attribut obtenu par transformation de données source | bleu |  |
| nouveau texte | gris |  | texte obtenu par transformation de données source | bleu |  |
| itération | orange hachuré |  | | | |

FIG. 4.9 : Représentation visuelle des types des nœuds de production




| symbole | type d'opération | équivalent XSLT |
|--|---|-----------------------|
|  | application des règles | xsl:apply-templates |
|  | extraction des nœuds texte (character data) | xsl:value-of |
|  | copie du nœud ou du sous-arbre | xsl:copy, xsl:copy-of |

FIG. 4.10 : Opérateurs de transformation VXT

placements où seront insérés les résultats des transformations appliquées aux nœuds extraits du document source.

- les structures de contrôle, comme les boucles itératives. Ces nœuds sont représentés par des rectangles hachurés pour bien les différencier des autres nœuds : il s'agit de structures de contrôle programmatiques qui ne feront pas elles-mêmes partie du résultat.

Le tableau de la figure 4.9 présente les différents types de nœuds pouvant apparaître dans la production.

Extraction et transformation de données du document source

Tous les nœuds appartenant à la deuxième catégorie sont reliés à exactement un nœud de la VPME en partie gauche de la règle par une opération de transformation. Il existe trois opérations principales qui peuvent être appliquées sur les données extraites de l'arbre source. Elles sont résumées dans le tableau de la figure 4.10 avec leur correspondance en XSLT. Nous ne présentons pas les équivalents Circus car ceux-ci n'existent pas en tant que tels. Circus est en effet un langage de plus bas niveau et les opérations de transformation se traduisent par des opérations de filtrage et de manipulation des structures qui n'ont pas de représentation abstraite.

Le lien, représenté par une ligne brisée attribuée d'une icône représentant le type d'opération à effectuer, permet d'associer un nœud extrait de l'arbre source à une opération de transformation à effectuer sur ces données et à un emplacement où mettre le résultat de cette transformation dans le fragment d'arbre produit par la règle. Seuls les nœuds bleus translucides de la *VPME*, qui représentent des instructions d'extraction, peuvent être reliés à la production. Des exemples de règles complètes contenant des liens sont détaillés dans la section suivante (figures 4.11 à 4.15). Les trois opérations de transformation sont définies pour tous les types de nœuds de l'arbre d'entrée, et produisent des résultats dont il est parfois possible d'inférer le type (texte, attribut, ou élément). Ainsi, le nœud identifiant dans la production l'emplacement où devra être inséré le résultat de l'opération de transformation associée se voit assigné une forme géométrique reflétant le type inféré⁸. Quand le type du résultat ne peut pas être inféré, le nœud est représenté par un cercle.

Identification des données à extraire

La correspondance entre la *VPME* et XSLT n'est pas directe, puisque là où (dans VXT) nous avons une seule expression pour spécifier le filtrage de la structure source, nous avons en XSLT une expression XPath pour la sélection du nœud contextuel et n expressions XPath pour les n instructions de transformation. Dans le cas de Circus, la correspondance est plus évidente, puisque comme nous l'avons vu précédemment ce langage offre aussi des primitives de filtrage qui permettent de spécifier dans la même expression conditions de sélection et données à extraire.

Les instructions d'extraction et de transformation génèrent des fragments du résultat à partir de données extraites du document source. Ces données sont identifiées par les nœuds bleus translucides des *VPMEs*. Leur position dans la structure est interprétée par rapport au nœud contextuel. Ces nœuds peuvent eux-mêmes contenir des nœuds au contour continu, qui représentent alors des prédicats portant sur la sélection des nœuds à extraire par l'instruction. Ainsi, dans la *VPME* de la figure 4.8, nous avons deux instructions d'extraction. La première sélectionne les éléments `title` fils d'éléments `article` et contenant obligatoirement un nœud texte (cette condition est symbolisée par la présence d'un losange dans le nœud à extraire). Sa traduction XPath sera, en prenant pour origine le nœud contextuel, `title[text()]`. Toujours dans la figure 4.8, la deuxième instruction sélectionne les éléments `section` fils de `article` à la condition qu'ils contiennent au moins un élément `para`. La traduction XPath de ceci est `section[para]`.

Note : à ce stade, la bordure continue ou discontinue des nœuds de la *VPME* n'a plus d'importance. Nous nous occupons ici d'identifier les nœuds à extraire, et le fait qu'ils aient été ou non requis pour la sélection du nœud contextuel n'a pas de conséquence. Ainsi, tous les nœuds sont pris en compte dans la génération du chemin reliant le nœud contextuel au nœud à extraire, que leur bordure soit continue ou discontinue.

La traduction en XSLT des instructions d'extraction et transformation va donc consister à générer une instruction XSLT comme `xsl:apply-templates` qui sera placée dans le fragment de résultat à l'endroit indiqué par le nœud associé au lien (ligne brisée) dans la production. L'attribut `select` de cette

⁸Un carré/rectangle pour les éléments, un triangle pour les attributs et un losange pour les nœuds texte.

instruction sera la valeur du chemin XPath reliant le nœud contextuel de la *VPME* au nœud identifiant les données à extraire, c'est-à-dire le nœud bleu translucide associé au lien dans la *VPME*.

Complexité des programmes

D'une manière générale, nous avons essayé dans VXT de cacher au maximum la complexité liée à certaines structures de contrôle, qui doivent néanmoins être proposées sans quoi l'expressivité du langage s'en trouverait fortement réduite. Nous proposons donc un mécanisme d'itération similaire au `for – each` de XSLT et un mécanisme de branchement conditionnel.

Les expressions XPath associées aux attributs `select` des instructions XSLT peuvent retourner un ensemble de nœuds (*node-set*) vérifiant les conditions exprimées. L'instruction de transformation est alors appliquée à tous les éléments de cet ensemble, le résultat étant intégré à l'arbre de sortie. La sémantique des instructions d'extraction VXT est la même. C'est-à-dire qu'un nœud bleu translucide identifiera et retournera potentiellement tout un ensemble de nœuds du document source qui seront transformés et placés à l'endroit indiqué dans la production, dans l'ordre qui était le leur dans la structure source. Il s'agit là d'un premier mécanisme d'itération complètement invisible du point de vue de l'utilisateur. Il est par contre parfois nécessaire de rendre explicite le mécanisme d'itération, par exemple quand l'utilisateur veut produire des éléments additionnels pour chaque nœud de l'ensemble de départ. Il est alors nécessaire d'identifier quels sont les nœuds produits en plus du résultat de la transformation des nœuds source à chaque itération. En XSLT, ceci est réalisé par l'emploi de la structure de contrôle `xsl : for – each` qui itère sur l'ensemble des nœuds retournés par l'expression XPath de l'attribut `select` associé. Le contenu de cet élément `xsl : for – each` est alors produit à chaque itération. VXT utilise pour sa part un rectangle orange hachuré qui délimite ce qui est produit à chaque itération sur l'ensemble des nœuds retournés par l'instruction d'extraction de la *VPME* associée à l'unique nœud coloré (bleu) fils du rectangle représentant la boucle (voir le tableau de la figure 4.9 et l'exemple de règle de la figure 4.11). L'équivalent Circus utilise une construction programmatique récemment proposée dans le langage qui permet de s'affranchir des structures de contrôle génériques qu'il aurait autrement fallu utiliser. Elle est de la forme :

$$seq[f \Rightarrow e]$$

où *seq* identifie une séquence, *f* un filtre Circus et *e* une expression qui peut faire référence à des variables du filtre. Cette construction retourne une séquence, et sa sémantique est la suivante : *f* est appliqué à chaque élément de la séquence *seq* ; dans les cas où le filtre sélectionne l'élément, *e* est ajouté à la séquence retournée. Elle est utilisée plusieurs fois dans l'exemple de transformation de la section 4.3.3 dont le code Circus complet est donné en annexe A.

Le mécanisme d'itération VXT cache au maximum les structures de contrôle liées à l'itération sur un ensemble et aide ainsi à réduire la complexité de la transformation et de sa représentation visuelle. Ceci est rendu possible par le fait que VXT est un langage spécialisé dans un domaine bien précis, pour lequel il n'y a pas de réel besoin de mécanisme d'itération général ; contrairement à LabView, qui doit proposer des boucles génériques dont la représentation visuelle n'est pas du tout intuitive (voir section 3.4.2).

Notons que, dans le même esprit, l'opération de copie proposée par VXT est plus abstraite que ce que propose XSLT. Ce dernier offre en effet deux instructions : `xsl:copy` et `xsl:copy - of`. La première ne copie que le nœud courant, sans tenir compte de son contenu et de ses attributs, et permet l'ajout d'un nouveau contenu dans l'élément copié. Ce contenu peut être formé de nouveaux nœuds et d'appels à d'autres instructions comme `xsl:apply - templates`. Au contraire, la deuxième instruction (`xsl:copy - of`) copie l'intégralité du sous-arbre dont la racine est le nœud identifié par l'expression XPath associée et n'autorise pas l'insertion de nouveau contenu dans l'élément copié. La différence entre les deux instructions peut être assez subtile. Cependant, elles s'utilisent pour des raisons différentes, et il est possible de savoir quelle est l'instruction appropriée simplement en tenant compte des contraintes exprimées ci-dessus⁹. Ainsi, nous avons choisi dans VXT d'exploiter cette propriété pour ne proposer qu'une seule opération de copie plus abstraite qui sera traduite en XSLT soit par `xsl:copy`, soit par `xsl:copy - of` suivant les cas, de manière transparente pour l'utilisateur.

4.3.3 Exemple de transformation

Nous développons maintenant un exemple complet de transformation illustrant l'utilisation des différentes constructions étudiées précédemment. Cette transformation produit une page XHTML contenant des informations sur l'auteur et la structure logique d'un article au format DocBook. Les figures 4.11, 4.12 et 4.13 associent les règles de transformation VXT et leurs équivalents XSLT.

La règle `t0` sélectionne les éléments `article` à la condition qu'ils contiennent un fils `title` et un fils `articleinfo`. Elle produit le squelette du document résultat (éléments `html`, `head` et `body`), copie le titre (dans `head` et `body`) et applique les règles de transformation aux éléments `author` extraits de l'élément `articleinfo`. Enfin, pour chaque `section` extraite du document source (à n'importe quel niveau de profondeur par rapport à l'élément `article`), elle demande l'application des règles de transformation après l'insertion d'un nouvel élément `hr`. La règle `t1` sélectionne les éléments `author` fils de `articleinfo`. Le fragment de résultat qu'elle produit est constitué d'un élément `p` dans lequel sont insérés les nom et prénom de l'auteur extraits du document source (s'ils existent). Enfin, la règle `t2` sélectionne les éléments `section` contenant un titre. Elle produit un élément `p` dans lequel elle place le contenu textuel de l'élément `title` (à l'intérieur d'un élément `h2`), un nouvel élément `br`, le contenu textuel du premier paragraphe s'il existe et enfin le texte «(Continued)».

Nous avons vu précédemment que dans le mode Circus il n'est pas possible de créer des VPMEs dont le nœud racine n'est pas le nœud contextuel. Ceci implique que la règle `t1` ne peut pas être traduite dans ce mode. Ainsi, pour pouvoir créer une transformation VXT équivalente mais exportable en code Circus, il faut légèrement modifier les règles `t0` et `t1` de manière à n'avoir que des règles compatibles avec le mode Circus. Ces deux règles et le code Circus correspondant à `t1` sont représentées par les figures 4.14 et 4.15. La transformation complète est fournie dans l'annexe A (y compris la fonction `XValue` qui retourne la concaténation sous forme de chaîne de tous les nœuds texte contenus dans une forêt XML¹⁰). `v5` à `v8` font référence à des variables de type `XMLTree`.

⁹C'est-à-dire la spécification ou non de nouveaux nœuds ou d'instructions à l'intérieur du nœud copié en partie droite.

¹⁰Cette fonction est équivalente à l'instruction XSLT `value-of`.

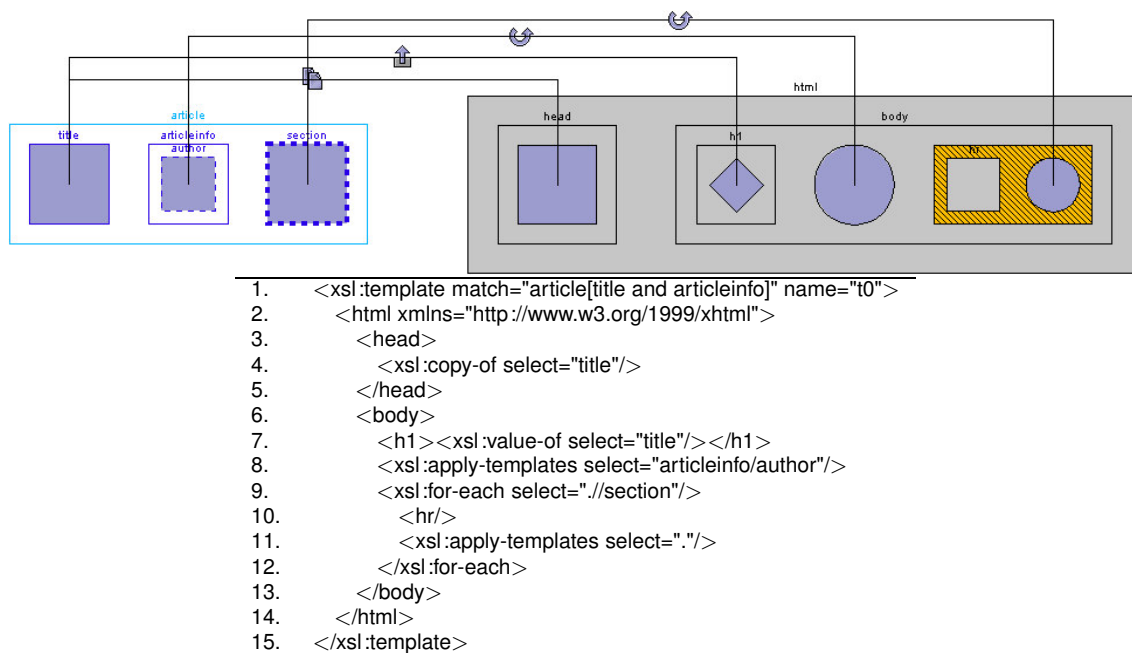


FIG. 4.11 : Règle de transformation t0

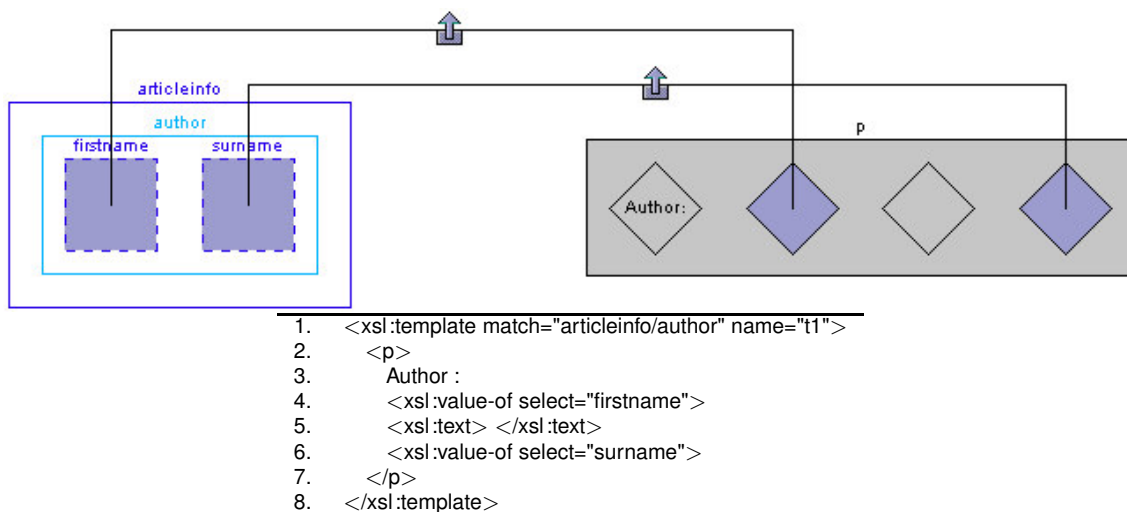
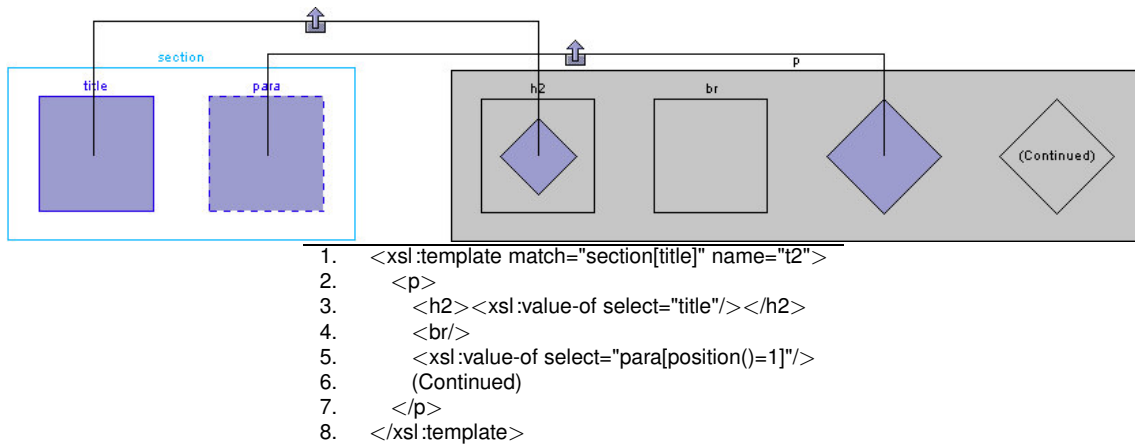
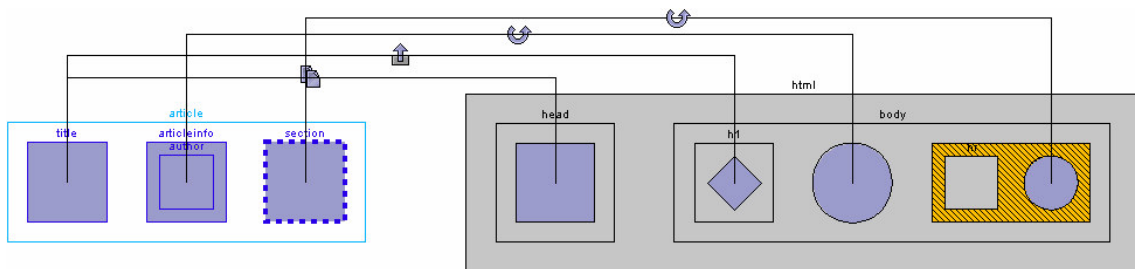
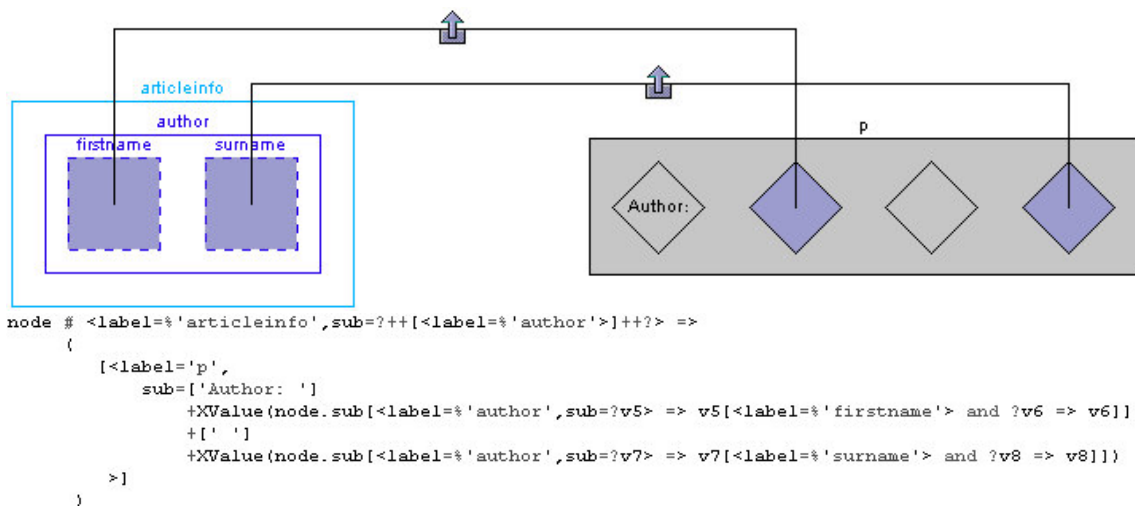


FIG. 4.12 : Règle de transformation t1

FIG. 4.13 : Règle de transformation t_2 FIG. 4.14 : Règle t_0 modifiée (condition de sélection sur *author*)FIG. 4.15 : Règle t_1 modifiée (changement de nœud contextuel)

4.4 Définition formelle du langage de transformation

Nous nous attachons maintenant à définir plus formellement le langage VXT. Nous allons commencer par définir la syntaxe du langage, puis nous spécifierons la fonction de traduction permettant de générer des feuilles de transformation XSLT à partir d'un programme VXT.

D'une manière générale, l'emploi d'outils formels dans la définition d'un langage visuel procure plusieurs avantages. Il est par exemple possible de spécifier de manière non ambiguë la syntaxe du langage. Cette spécification est souvent plus compacte et plus compréhensible que ne le serait l'équivalent en langue naturelle, moyen de communication imprécis et verbeux dans le cadre de la description de concepts et processus complexes. La spécification formelle de la syntaxe peut être utilisée pour générer un analyseur grammatical (*parser*) qui pourra reconnaître et lire les phrases du langage. Elle offre aussi une base concrète à l'utilisateur débutant, qui n'a alors plus à se reposer seulement sur une généralisation des exemples de code existant pour apprendre les constructions autorisées. Cependant, peu de langages de programmation visuels ont une syntaxe formellement définie. Ceci est en partie dû à la lourdeur et à la difficulté de manipulation des formalismes utilisés, tels que les grammaires visuelles pour la spécification de la syntaxe. Les formalismes grammaticaux utilisés pour les langages visuels sont en effet beaucoup plus complexes comparés par exemple aux grammaires BNF qui n'ont pas à tenir compte entre autres de la multi-dimensionnalité potentielle des expressions du langage. Ce point est développé dans l'annexe C «Introduction aux grammaires relationnelles».

Dans le cadre de ce travail, l'emploi d'outils formels va nous permettre de définir la syntaxe du langage de manière non ambiguë. De même pour la fonction de traduction de programmes VXT en feuilles de transformation XSLT, sur laquelle nous allons établir deux propriétés. Nous allons vérifier la complétude de la fonction de traduction, c'est-à-dire que toute *VPME* bien formée peut être traduite. Nous allons aussi vérifier que la traduction produit des feuilles de transformation XSLT correctes, dans le sens où elles sont bien formées (*XML well-formedness*) et où toutes les expressions XPath engendrées sont légales (nous verrons dans la section 4.4.5 pourquoi nous ne pouvons pas prouver la validité des feuilles de transformation XSLT par rapport à un schéma en général).

4.4.1 Grammaire visuelle et syntaxe abstraite

Nous avons défini, dans le cadre de l'étude formelle du langage visuel pour la représentation de DTD ([221] et Annexe B), une grammaire basée sur une version légèrement étendue du formalisme des grammaires relationnelles [236, 95]. Cette grammaire (voir la Définition 1 de l'article en Annexe B) définit un langage visuel représentant un sur-ensemble des phrases visuelles pouvant être engendrées à partir de la traduction de DTD. Informellement, cette grammaire autorise l'emboîtement (potentiellement infini) d'objets graphiques et de chaînes de caractères aux conditions suivantes :

- les objets contenant d'autres objets doivent être rectangulaires,
- l'ensemble des fils d'un objet est disposé soit horizontalement soit verticalement (alignement des centres géométriques), avec la condition supplémentaire que ces objets ne doivent pas se chevaucher même partiellement,
- le rectangle englobant l'ensemble des fils (*bounding box*) est centré par rapport à l'objet parent, dont la taille est définie de manière à contenir l'ensemble de ses fils.

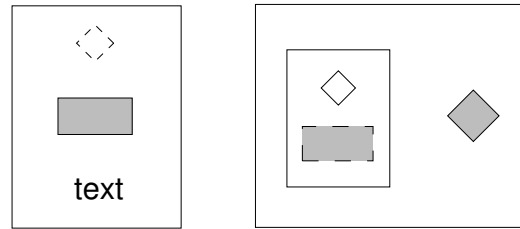


FIG. 4.16 : *Exemples de phrases visuelles correctes*

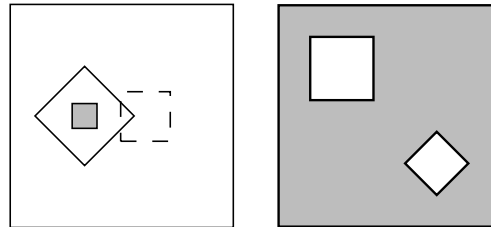


FIG. 4.17 : *Exemples de phrases visuelles incorrectes*

Nous donnons quelques exemples de phrases visuelles correctes et incorrectes. La figure 4.16 montre deux exemples de phrases correctes ; les objets fils d'un même élément sont alignés entre eux et centrés par rapport à leur parent. La figure 4.17 illustre deux phrases n'appartenant pas au langage : la première est incorrecte car une forme autre qu'un rectangle (le losange¹¹) contient un autre objet et parce que deux objets se chevauchent partiellement ; la seconde est incorrecte car les fils du rectangle gris ne sont alignés ni verticalement ni horizontalement.

La définition formelle de cette grammaire est donnée dans l'annexe B. Une introduction au formalisme des grammaires relationnelles est proposée en annexe C. Sa lecture n'est cependant pas requise pour la compréhension de la suite de ce chapitre puisque nous travaillons directement sur des variantes de la syntaxe abstraite \mathcal{AS} définie dans l'article mentionné précédemment (annexe B) et reprise ici.

Les propriétés du langage visuel décrit par cette grammaire ont été capturées dans une syntaxe abstraite nommée \mathcal{AS} afin de simplifier les traitements effectués lors de la manipulation des structures visuelles. Il est en effet difficile de raisonner sur la grammaire, qui représente le langage de manière concrète et fournit par conséquent un niveau de détails (et donc de complexité) trop important par rapport aux traitements envisagés. La syntaxe \mathcal{AS} simplifie les traitements formels en s'abstrayant de certaines questions spatiales de bas niveau, tout en conservant une sémantique visuelle fournie par la fonction de traduction \mathcal{T} (annexe B) qui s'applique aux phrases appartenant à $\mathcal{L}(\mathcal{AS})$ (le langage généré par \mathcal{AS}) et produit un ensemble d'objets graphiques. Dans l'article, une syntaxe abstraite plus spécifique \mathcal{DS}

¹¹Il s'agit en fait d'un carré orienté à 45 degrés ; nous désignons par la suite de telles formes par le terme de losange pour des raisons de commodité.

capturant les propriétés du langage de représentation de DTD¹² a été obtenue par spécialisation de \mathcal{AS} . Nous adoptons ici la même démarche, en nous basant sur une version légèrement modifiée de la syntaxe abstraite \mathcal{AS} ci-dessous. Afin de ne pas nous encombrer de données superflues, nous nous affranchissons des attributs de largeur w , hauteur h et d'espacement d .

Définition 1 (Syntaxe abstraite \mathcal{AS})

| | |
|------------------------------------|--|
| $G \rightarrow \mathbf{Box}(G, s)$ | <i>objet rectangulaire contenant d'autres objets</i> |
| $G \rightarrow \mathbf{Va}(G, G)$ | <i>alignement vertical</i> |
| $G \rightarrow \mathbf{Ha}(G, G)$ | <i>alignement horizontal</i> |
| $G \rightarrow \mathbf{Shp}(f, s)$ | <i>forme géométrique terminale</i> |
| $G \rightarrow \mathbf{Txt}(t)$ | <i>texte terminal</i> |

avec t une chaîne de texte et f une variable codant la forme et pouvant prendre les valeurs *triangle*, *square*, *lozenge*, *star* ou *circle*. s est un quadruplet dont les éléments représentent les attributs de style des objets graphiques. Le premier élément représente le style du contour de l'objet (continu ou discontinu) ; le deuxième la couleur de ce contour ; le troisième l'épaisseur du contour et enfin le dernier la transparence¹³ de l'intérieur. Dans le cadre de VXT, seules certaines combinaisons seront autorisées :

Définition 2 (Valeurs autorisées des quadruplets de style)

s : $\langle v_1 \in \{solid, dashed\}, v_2 \in Color, v_3 \in \{thin, thick\}, v_4 \in \{notFilled, translucent\} \rangle$

$s_1 = \langle solid, lightblue, thin, translucent \rangle$

$s_2 = \langle solid, lightblue, thin, notFilled \rangle$

$s_3 = \langle solid, lightblue, thick, translucent \rangle$

$s_4 = \langle solid, lightblue, thick, notFilled \rangle$

$s_5 = \langle solid, blue, thick, translucent \rangle$

$s_6 = \langle solid, blue, thick, notFilled \rangle$

$s_7 = \langle solid, blue, thin, translucent \rangle$

$s_8 = \langle solid, blue, thin, notFilled \rangle$

$s_9 = \langle solid, red, thick, notFilled \rangle$

$s_{10} = \langle solid, red, thin, notFilled \rangle$

$s_{11} = \langle dashed, blue, thick, translucent \rangle$

$s_{12} = \langle dashed, blue, thin, translucent \rangle$



Nous définissons alors deux syntaxes abstraites \mathcal{AS}_V et \mathcal{AS}_R , spécialisations de \mathcal{AS} capturant les contraintes structurales liées respectivement aux *VPMEs* et aux productions de règles VXT.

¹²Ce langage est plus contraint que $\mathcal{L}(\mathcal{AS})$; il s'agit donc d'un sous-langage de $\mathcal{L}(\mathcal{AS})$.

¹³Il s'agit en fait de semi-transparence : l'objet est coloré et translucide.

Définition 3 (Syntaxe abstraite \mathcal{AS}_V pour les VPMEs)

$$\begin{aligned}
G_V &\rightarrow B_V \mid S_{VE} \mid S_{VT} \mid S_{VZ} \mid S_{VA} \\
H_{VE} &\rightarrow B_V \mid S_{VE} \mid S_{VT} \mid S_{VZ} \\
H_{VE} &\rightarrow \mathbf{Ha}(B_V \mid S_{VE} \mid S_{VT} \mid S_{VZ}, H_{VE}) \\
H_{VA} &\rightarrow S_{VA} \mid \mathbf{Ha}(S_{VA}, H_{VA}) \\
B_V &\rightarrow \mathbf{Box}(H_{VE} \mid H_{VA}, s) \\
B_V &\rightarrow \mathbf{Va}(\mathbf{Txt}(t), \mathbf{Box}(H_{VE} \mid H_{VA}, s)) \\
B_V &\rightarrow \mathbf{Box}(\mathbf{Va}(H_{VE}, H_{VA}), s) \\
B_V &\rightarrow \mathbf{Va}(\mathbf{Txt}(t), \mathbf{Box}(\mathbf{Va}(H_{VE}, H_{VA}), s)) \\
S_{VE} &\rightarrow \mathbf{Shp}(\mathbf{square}, s) \mid \mathbf{Va}(\mathbf{Txt}(t), \mathbf{Shp}(\mathbf{square}, s)) \\
S_{VT} &\rightarrow \mathbf{Shp}(\mathbf{lozenge}, s) \\
S_{VZ} &\rightarrow \mathbf{Shp}(\mathbf{star}, s) \\
S_{VA} &\rightarrow \mathbf{Shp}(\mathbf{triangle}, s) \mid \mathbf{Va}(\mathbf{Txt}(t), \mathbf{Shp}(\mathbf{triangle}, s))
\end{aligned}$$

Nous notons alors $\mathcal{L}(\mathcal{AS}_V)$ le langage visuel des VPMEs généré par \mathcal{AS}_V . De la même façon, nous définissons le langage visuel $\mathcal{L}(\mathcal{AS}_R)$ représentant les parties droites correspondant aux fragments d'arbre produits par les règles. Ce langage est généré par la grammaire \mathcal{AS}_R suivante (le style associé aux objets graphiques est beaucoup plus simple pour les parties droites ; nous remplaçons donc pour des raisons de concision le quadruplet de style par une chaîne de caractères indiquant si l'objet est coloré en bleu (*blue*), s'il est gris (*gray*) ou bien hachuré (*hatched*)).

Définition 4 (Syntaxe abstraite \mathcal{AS}_R pour les productions)

$$\begin{aligned}
G_R &\rightarrow F_R \mid \mathbf{Va}(F_R, G_R) \\
F_R &\rightarrow B_R \mid S_{REF} \mid S_{REW} \mid S_{RT} \mid S_{RC} \mid S_{RAF} \mid S_{RAW} \\
H_{RE} &\rightarrow B_R \mid S_{REF} \mid S_{REW} \mid S_{RT} \mid S_{RC} \\
H_{RE} &\rightarrow \mathbf{Ha}(B_R \mid S_{REF} \mid S_{REW} \mid S_{RT} \mid S_{RC}, H_{RE}) \\
H_{RA} &\rightarrow S_{RAF} \mid S_{RAW} \\
H_{RA} &\rightarrow \mathbf{Ha}(S_{RAF} \mid S_{RAW}, H_{RA}) \\
B_R &\rightarrow \mathbf{Va}(\mathbf{Txt}(t), \mathbf{Box}(H_{RE} \mid H_{RA}, \mathbf{gray})) \\
B_R &\rightarrow \mathbf{Va}(\mathbf{Txt}(t), \mathbf{Box}(\mathbf{Va}(H_{RE}, H_{RA}), \mathbf{gray})) \\
B_R &\rightarrow \mathbf{Box}(H_{RE} \mid H_{RA}, \mathbf{blue}) \\
B_R &\rightarrow \mathbf{Box}(\mathbf{Va}(H_{RE}, H_{RA}), \mathbf{blue}) \\
B_R &\rightarrow \mathbf{Box}(H_{RE} \mid H_{RA}, \mathbf{hatched}) \\
B_R &\rightarrow \mathbf{Box}(\mathbf{Va}(H_{RE}, H_{RA}), \mathbf{hatched}) \\
S_{REF} &\rightarrow \mathbf{Shp}(\mathbf{square}, \mathbf{blue}) \\
S_{REW} &\rightarrow \mathbf{Va}(\mathbf{Txt}(t), \mathbf{Shp}(\mathbf{square}, \mathbf{gray})) \\
S_{RT} &\rightarrow \mathbf{Shp}(\mathbf{lozenge}, \mathbf{gray}) \mid \mathbf{Shp}(\mathbf{lozenge}, \mathbf{blue}) \\
S_{RC} &\rightarrow \mathbf{Shp}(\mathbf{circle}, \mathbf{blue}) \\
S_{RAF} &\rightarrow \mathbf{Shp}(\mathbf{triangle}, \mathbf{blue}) \\
S_{RAW} &\rightarrow \mathbf{Va}(\mathbf{Txt}(t), \mathbf{Shp}(\mathbf{triangle}, \mathbf{gray}))
\end{aligned}$$

Notation des arbres

Par la suite, nous notons X , X_1 et X_2 des variables prenant valeur dans $\mathcal{L}(\mathcal{AS}_V)$ ou $\mathcal{L}(\mathcal{AS}_R)$. f désignera une forme géométrique et prendra valeur dans $\{\text{triangle}, \text{square}, \text{lozenge}, \text{star}, \text{circle}\}$.

Nous aurons besoin de désigner certains nœuds spécifiques des arbres engendrés par les grammaires précédentes. Pour cela, nous introduisons une notation supplémentaire pour les constructions $\mathbf{Box}(X, s)$ et $\mathbf{Shp}(f, s)$ en utilisant des index pour chacun de ces nœuds qui seront notés respectivement

$$\mathbf{Box}_i(X, s)$$

et

$$\mathbf{Shp}_i(f, s)$$

Chaque règle VXT possède deux ensembles d'index disjoints référant respectivement les nœuds de la VPME et les nœuds de la production (partie droite). Dans chacun de ces ensembles, l'index d'un nœud est unique et référence donc exactement un nœud dans l'un des arbres (VPME ou production). Par exemple,

$$\mathbf{Shp}_3(\text{square}, \langle \text{solid}, \text{blue}, \text{thin}, \text{notFilled} \rangle)$$

indique que ce nœud de type élément est référencé par l'index 3. Afin d'alléger la notation dans les traitements suivants, nous ne faisons apparaître ces indices (notés i , j et k) que quand ils sont nécessaires.

Correspondance entre index et sous-arbres

Nous allons avoir besoin dans les définitions suivantes de plusieurs fonctions retournant des sous-arbres de VPME en fonction de l'index et de la nature des nœuds. Nous avons vu précédemment que chaque nœud de type $\mathbf{Shp}(f, s)$ ou $\mathbf{Box}(X, s)$ est associé à un index unique dans l'ensemble des index associé à la VPME à laquelle le nœud appartient. Nous fixons, par convention, une contrainte supplémentaire : pour chaque VPME, le nœud contextuel est associé à l'index 0. Il n'y a par contre aucune contrainte sur la manière d'indexer les autres nœuds, hormis l'unicité des index.

Nous appelons alors $i2v$ la fonction qui, étant donné une VPME et un index, retourne le sous-arbre dont le nœud racine a pour index la valeur passée en paramètre. Nous notons I l'ensemble des index.

Définition 5 ($i2v$: Fonction d'association index/sous-arbre)

$$i2v : \mathcal{L}(\mathcal{AS}_V) \times I \rightarrow \mathcal{L}(\mathcal{AS}_V)$$

Nous allons aussi avoir besoin d'une fonction utilisée lors de la manipulation d'arbres, notée gp pour *get path* qui, étant donné deux index associés à deux nœuds d'une même VPME, retourne le chemin à parcourir dans l'arbre pour atteindre le second nœud en partant du premier. Le résultat est fourni sous la forme d'une séquence contenant les index associés aux nœuds rencontrés sur ce chemin, les index du nœud origine et du nœud final inclus. La fonction est définie uniquement dans le cas où le deuxième nœud est un descendant du premier nœud ou bien égal à ce nœud. Elle fournit par conséquent uniquement des séquences représentant des chemins orientés de la racine vers les feuilles de l'arbre. Dans le cas où cette condition n'est pas vérifiée, c'est-à-dire que le premier nœud n'est pas un ancêtre du (ou égal au) second, la fonction retourne une séquence vide.

Définition 6 (gp : Génération de chemins entre deux nœuds)

$$gp : \mathcal{L}(\mathcal{AS}_V) \times I \times I \rightarrow I^*$$

Nous notons I^* le monoïde libre obtenu par la concaténation de l'ensemble des index. Nous utilisons pour l'accès aux membres la notation standard suivante :

$$s_k$$

où k est un entier positif indiquant la position du membre dans le monoïde. Par convention, le premier élément est à l'index 0, donc s_1 désigne le deuxième élément.

Remarque : comme nous l'avons défini plus haut, seuls les nœuds de type $\mathbf{Shp}(f, s)$ ou $\mathbf{Box}(X, s)$ sont indexés. Il en résulte que les index présents dans une séquence retournée par gp sont uniquement associés à des nœuds de ces deux types.

Enfin, toujours dans le but d'alléger les notations, nous introduisons la fonction $gscn$ pour *Get Subtree Containing Node*, qui étant donné une $VPME$ et deux index i et j retourne le sous-arbre dont la racine est un fils du nœud d'indice i et qui contient le nœud d'indice j :

Définition 7 (gscn : Génération de chemins entre deux nœuds)

$$\begin{aligned} gscn : \mathcal{L}(\mathcal{AS}_V) \times I \times I &\rightarrow \mathcal{L}(\mathcal{AS}_V) \\ gscn(v, i, j) &= i2v(v, s_1) \quad \text{avec} \quad gp(v, i, j) = s_0 s_1 \dots s_n \end{aligned}$$

Le contexte d'utilisation de cette fonction nous garantit que la séquence retournée par $gp(v, i, j)$ contient au moins deux éléments, donc s_1 sera toujours défini dans ce contexte. Nous utiliserons souvent cette fonction en assignant la valeur 0 à j , cette index étant par convention assigné au nœud contextuel de chaque $VPME$.

4.4.2 VPMEs et productions bien formées**Vérification des VPMEs**

Le langage $\mathcal{L}(\mathcal{AS}_V)$ ne capture cependant pas toutes les contraintes structurales des $VPMEs$. Modéliser ces contraintes dans la grammaire aurait rendu la définition de \mathcal{AS}_V beaucoup plus complexe ; nous avons donc choisi de les traiter au moyen de relations logiques vérifiant la correction des $VPMEs$ appartenant à $\mathcal{L}(\mathcal{AS}_V)$ par rapport aux contraintes suivantes.

Premièrement, les nœuds d'une expression de sélection exprimant une condition négative (absence du nœud) ne doivent avoir aucun descendant correspondant à une instruction d'extraction. Ne pas respecter cette règle n'entraîne ni de problème du point de vue de la traduction des $VPMEs$ ni du point de vue de l'exécution de la transformation ; les expressions XPath résultantes pour les attributs *match* et *select* sont syntaxiquement correctes. Cependant, les instructions d'extraction ne seront jamais exécutées. En effet, elles correspondent à des nœuds descendants d'un nœud qui par définition n'existe pas dans le contenu de l'élément source sélectionné par la $VPME$. Elles retournent donc toujours un ensemble vide sur lequel aucune opération ne peut être effectuée.

Une contrainte similaire interdit au nœud contextuel d’être un descendant d’une condition négative. Cette contrainte vient de la sémantique visuelle des *VPMEs*, qui considère tous les nœuds descendants d’une condition négative comme des prédicats portant sur cette condition. Notons que cette contrainte n’interdit pas la spécification de *VPMEs* sélectionnant des nœuds ayant pour ancêtre un ou des nœuds de type(s) spécifique(s)¹⁴. Il suffit pour cela de déclarer chaque nœud ancêtre comme une condition de sélection standard en lui ajoutant un prédicat textuel lui interdisant d’être du type spécifié.

Enfin, une *VPME* doit contenir exactement un nœud contextuel. Le non-respect de cette règle produira des expressions XPath incorrectes du point de vue syntaxique. Du point de vue sémantique, il serait possible d’associer différentes significations à une *VPME* contenant de multiples nœuds contextuels. Dans le cadre de la traduction XSLT, il serait par exemple possible de générer n règles de transformation (*template rules*) correspondant chacune à un de ces n nœuds contextuels¹⁵. Dans chaque cas, les autres nœuds contextuels de la *VPME* seraient soit ignorés soit considérés comme des nœuds standard (conditions existentielles). Mais la sémantique visuelle de ces expressions n’est pas intuitive. Les cas d’utilisation intéressants sont très limités et il est probable que la spécification de telles expressions par le programmeur résulterait souvent d’une mauvaise compréhension de cette sémantique. C’est la raison pour laquelle nous nous restreignons à des *VPMEs* contenant exactement un nœud contextuel, plus faciles à comprendre et plus proches de leurs traductions Circus et XSLT.

Pour exprimer les relations logiques de “*VPME* bien formée”, nous introduisons deux critères de validation. VF1 vérifie que les nœuds exprimant des conditions négatives ne contiennent ni d’instruction d’extraction (symbolisée par un intérieur semi-transparent¹⁶), ni le nœud contextuel (symbolisé par la couleur bleu clair, c’est-à-dire **lightblue**). La variable f indique le type de forme géométrique, le quadruplet $\langle S, C, B_1, B_2 \rangle$ représente l’information de style et prend une des valeurs définies précédemment. Enfin, X et X_i sont des variables prenant des valeurs dans $\mathcal{L}(\mathcal{AS}_V)$.

Définition 8 (Critère de validation VF1) $i \in \{1, 2\}$

$$\begin{array}{c}
\frac{}{\vdash_{A_1} \mathbf{Shp}(f, \langle S, C, B_1, B_2 \rangle)} \text{ [VF1a]} \qquad \frac{C \neq \mathbf{lightblue}}{\vdash_{A_2} \mathbf{Shp}(f, \langle S, C, B_1, \mathbf{notFilled} \rangle)} \text{ [VF1b]} \\
\frac{\vdash_{A_2} X}{\vdash_{A_1} \mathbf{Box}(X, \langle S, \mathbf{red}, B_1, B_2 \rangle)} \text{ [VF1c]} \qquad \frac{\vdash_{A_1} X \quad C \neq \mathbf{red}}{\vdash_{A_1} \mathbf{Box}(X, \langle S, C, B_1, B_2 \rangle)} \text{ [VF1d]} \\
\frac{\vdash_{A_2} X \quad C \neq \mathbf{lightblue}}{\vdash_{A_2} \mathbf{Box}(X, \langle S, C, B_1, \mathbf{notFilled} \rangle)} \text{ [VF1e]} \qquad \frac{\vdash_{A_i} X_1 \quad \vdash_{A_i} X_2}{\vdash_{A_i} \mathbf{Ha}(X_1, X_2)} \text{ [VF1f]} \\
\frac{\vdash_{A_i} X}{\vdash_{A_i} \mathbf{Va}(\mathbf{Txt}(t), X)} \text{ [VF1g]} \qquad \frac{\vdash_{A_i} X_1 \quad \vdash_{A_i} X_2}{\vdash_{A_i} \mathbf{Va}(X_1, X_2)} \text{ [VF1h]}
\end{array}$$

¹⁴Ce qui pourrait s’exprimer visuellement en positionnant le nœud contextuel à l’intérieur d’une condition négative, sachant que cela serait en conflit avec la sémantique visuelle des conditions négatives.

¹⁵A noter qu’il serait incorrect de créer une seule règle de transformation ayant un attribut *match* regroupant les différentes expressions XPath en un ensemble d’alternatives puisque l’interprétation des expressions XPath de sélection (attribut *select*) liées aux opérations dans la partie droite de la règle dépendent directement de l’expression XPath de l’attribut *match* et sont donc différentes pour un même ensemble de nœuds de l’arbre source suivant l’élément à partir duquel elles sont interprétées (nœud contextuel) dans ce même arbre source.

¹⁶i.e. quatrième élément du quadruplet de style associé ayant pour valeur **translucent**.

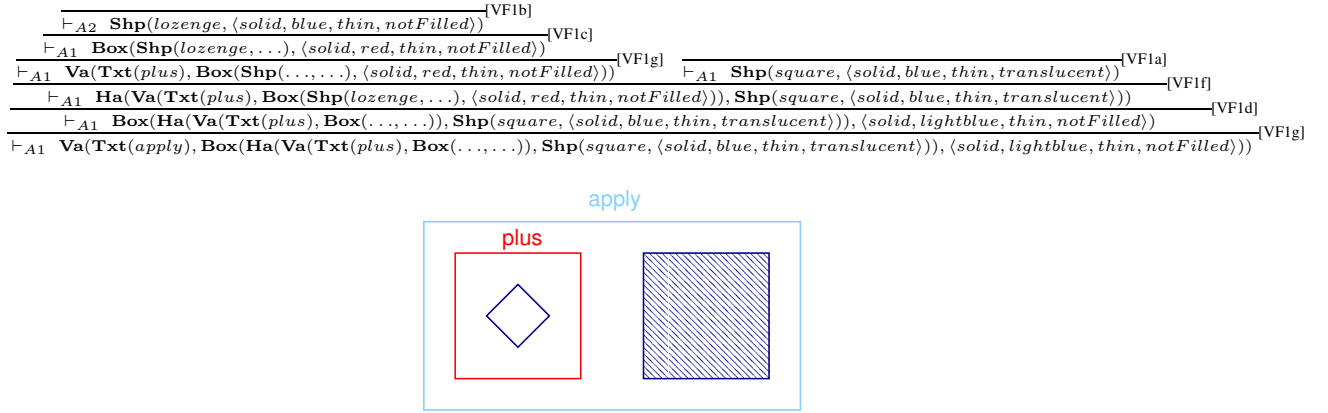


FIG. 4.18 : Preuve de correction d'une VPME

La relation VF2 exprime quant à elle le fait que la VPME doit contenir exactement un nœud contextuel.

Définition 9 (Critère de validation VF2) $i \in \{1, 2\}$

$$\begin{array}{c}
\frac{}{\vdash_{B_1} \mathbf{Shp}(f, \langle S, \mathbf{lightblue}, B_1, B_2 \rangle)} \text{[VF2a]} \quad \frac{C \neq \mathbf{lightblue}}{\vdash_{B_2} \mathbf{Shp}(f, \langle S, C, B_1, B_2 \rangle)} \text{[VF2b]} \\
\frac{\vdash_{B_2} X}{\vdash_{B_1} \mathbf{Box}(X, \langle S, \mathbf{lightblue}, B_1, B_2 \rangle)} \text{[VF2c]} \quad \frac{\vdash_{B_1} X \quad C \neq \mathbf{lightblue}}{\vdash_{B_1} \mathbf{Box}(X, \langle S, C, B_1, B_2 \rangle)} \text{[VF2d]} \\
\frac{\vdash_{B_2} X \quad C \neq \mathbf{lightblue}}{\vdash_{B_2} \mathbf{Box}(X, \langle S, C, B_1, B_2 \rangle)} \text{[VF2e]} \quad \frac{\vdash_{B_i} X_1 \quad \vdash_{B_i} X_2}{\vdash_{B_i} \mathbf{Ha}(X_1, X_2)} \text{[VF2f]} \\
\frac{\vdash_{B_i} X}{\vdash_{B_i} \mathbf{Va}(\mathbf{Txt}(t), X)} \text{[VF2g]} \quad \frac{\vdash_{B_i} X_1 \quad \vdash_{B_i} X_2}{\vdash_{B_i} \mathbf{Va}(X_1, X_2)} \text{[VF2h]}
\end{array}$$

Une VPME v sera alors considérée correcte si et seulement si

$$\vdash_{A_1} v \wedge \vdash_{B_1} v$$

La figure 4.18 développe une preuve de correction de VPME par VF1. Nous faisons par la suite l'hypothèse que les traitements et transformations sont appliqués sur des expressions correctes. Pour cela, nous définissons le langage visuel \mathcal{V} auquel appartiennent les phrases du langage $\mathcal{L}(\mathcal{AS}_V)$ vérifiant les contraintes imposées par VF1 et VF2.

Définition 10 (Langage visuel \mathcal{V} pour les VPMEs)

$$v \in \mathcal{V} \text{ ssi } (v \in \mathcal{L}(\mathcal{AS}_V) \wedge \vdash_{A_1} v \wedge \vdash_{B_1} v)$$

Vérification des productions

De manière similaire, nous devons introduire un critère de validation VF3 pour le langage $\mathcal{L}(\mathcal{AS}_R)$, la grammaire \mathcal{AS}_R ne capturant pas la contrainte suivante : chaque nœud de type `for – each` doit contenir comme fils exactement un nœud résultant d’une opération d’extraction et de transformation, c’est-à-dire ayant comme style *blue*. Ce critère est un peu plus complexe que les précédentes et requiert trois relations. Informellement, \vdash_{C_1} traite les parties extérieures aux boucles `for – each`, \vdash_{C_2} capture le fait qu’il doit exister un nœud coloré (bleu) à l’intérieur du rectangle hachuré représentant la boucle `for – each` et \vdash_{C_3} indique les autres constructions autorisées dans cette même boucle. Un exemple concret d’application du système logique VF3 est donné en Annexe D.

Définition 11 (Critère de validation VF3)

$$\begin{array}{c}
\frac{}{\vdash_{C_1} \mathbf{Shp}(f, \mathbf{gray})} \text{ [VF3a]} \quad \frac{}{\vdash_{C_1} \mathbf{Shp}(f, \mathbf{blue})} \text{ [VF3b]} \quad \frac{\vdash_{C_1} X}{\vdash_{C_1} \mathbf{Box}(X, \mathbf{gray})} \text{ [VF3c]} \\
\frac{\vdash_{C_1} X}{\vdash_{C_1} \mathbf{Box}(X, \mathbf{blue})} \text{ [VF3d]} \quad \frac{\vdash_{C_2} X}{\vdash_{C_1} \mathbf{Box}(X, \mathbf{hatched})} \text{ [VF3e]} \quad \frac{\vdash_{C_1} X}{\vdash_{C_1} \mathbf{Box}(X, \mathbf{blue})} \text{ [VF3f]} \\
\frac{\vdash_{C_1} X}{\vdash_{C_1} \mathbf{Va}(\mathbf{Txt}(t), X)} \text{ [VF3g]} \quad \frac{\vdash_{C_1} X_1 \quad \vdash_{C_1} X_2}{\vdash_{C_1} \mathbf{Va}(X_1, X_2)} \text{ [VF3h]} \quad \frac{\vdash_{C_1} X_1 \quad \vdash_{C_1} X_2}{\vdash_{C_1} \mathbf{Ha}(X_1, X_2)} \text{ [VF3i]} \\
\frac{}{\vdash_{C_2} \mathbf{Shp}(f, \mathbf{blue})} \text{ [VF3j]} \quad \frac{\vdash_{C_1} X}{\vdash_{C_2} \mathbf{Box}(X, \mathbf{blue})} \text{ [VF3k]} \quad \frac{\vdash_{C_2} X}{\vdash_{C_2} \mathbf{Va}(\mathbf{Txt}(t), X)} \text{ [VF3l]} \\
\frac{\vdash_{C_2} X_1 \quad \vdash_{C_3} X_2}{\vdash_{C_2} \mathbf{Va}(X_1, X_2)} \text{ [VF3m]} \quad \frac{\vdash_{C_3} X_1 \quad \vdash_{C_2} X_2}{\vdash_{C_2} \mathbf{Va}(X_1, X_2)} \text{ [VF3n]} \\
\frac{\vdash_{C_2} X_1 \quad \vdash_{C_3} X_2}{\vdash_{C_2} \mathbf{Ha}(X_1, X_2)} \text{ [VF3o]} \quad \frac{\vdash_{C_3} X_1 \quad \vdash_{C_2} X_2}{\vdash_{C_2} \mathbf{Ha}(X_1, X_2)} \text{ [VF3p]} \\
\frac{}{\vdash_{C_3} \mathbf{Shp}(f, \mathbf{gray})} \text{ [VF3q]} \quad \frac{\vdash_{C_1} X}{\vdash_{C_3} \mathbf{Box}(X, \mathbf{gray})} \text{ [VF3r]} \quad \frac{\vdash_{C_3} X}{\vdash_{C_3} \mathbf{Va}(\mathbf{Txt}(t), X)} \text{ [VF3s]} \\
\frac{\vdash_{C_3} X_1 \quad \vdash_{C_3} X_2}{\vdash_{C_3} \mathbf{Ha}(X_1, X_2)} \text{ [VF3t]} \quad \frac{\vdash_{C_3} X_1 \quad \vdash_{C_3} X_2}{\vdash_{C_3} \mathbf{Ha}(X_1, X_2)} \text{ [VF3u]}
\end{array}$$

Nous définissons alors le langage visuel \mathcal{R} auquel appartiennent les phrases du langage $\mathcal{L}(\mathcal{AS}_R)$ vérifiant les contraintes imposées par la fonction VF3.

Définition 12 (Langage visuel \mathcal{R} pour les productions)

$$r \in \mathcal{R} \text{ ssi } (r \in \mathcal{L}(\mathcal{AS}_R) \wedge \vdash_{C_1} r)$$




| <i>op</i> | symbole | type d'opération |
|----------------------|---|--|
| <i>apply – rules</i> |  | application des règles |
| <i>text</i> |  | extraction des noeuds texte (character data) |
| <i>copy</i> |  | copie du nœud ou du sous-arbre |

FIG. 4.19 : Opérateurs de transformation VXT

4.4.3 Fonction de traduction

Les programmes VXT peuvent être exécutés directement depuis l'environnement d'édition, ou bien exportés comme feuilles de transformation XSLT ou comme code source Circus. Dans tous les cas, il est nécessaire de traduire l'ensemble des règles visuelles VXT en équivalents XSLT ou Circus. Nous allons ici définir formellement la fonction de traduction des programmes VXT en feuilles de transformation XSLT.

Nous commençons par définir les règles de transformation VXT. Une règle de transformation est un triplet

$$TR = \langle v, r, l \rangle$$

avec

$$v \in \mathcal{V}, r \in \mathcal{R}, l = \{ \langle N_1, N_2, op \rangle \}$$

où l représente l'ensemble des instructions d'extraction et de transformation d'une règle VXT. Chaque instruction est représentée par un triplet $\langle N_1, N_2, op \rangle$, où N_1 est un nœud de v , N_2 est un nœud de r et op est une chaîne représentant le type d'opération appliquée à N_1 . Les différentes valeurs de op sont regroupées dans le tableau de la figure 4.19.

Un programme VXT est alors défini comme un ensemble de règles de transformation :

$$\mathcal{P} = \{TR\}$$

Le processus de traduction en feuille de transformation XSLT consiste à appliquer une fonction (de traduction) \mathcal{T} à chaque règle TR de \mathcal{P} .

Définition 13 (\mathcal{T} , Fonction de traduction de programmes VXT en XSLT)

$$\mathcal{T}[\] : TR \rightarrow \mathcal{XS}$$

$$\mathcal{T}[\langle v, r, l \rangle] = \langle \text{xsl:template match} = \text{"}\mathcal{T}_M[v]_v\text{"} \rangle \\ \mathcal{T}_R[r]_v^\Gamma \\ \langle \text{/xsl:template} \rangle$$

avec $\Gamma / \varphi_1 = l$

Nous appelons ici \mathcal{XS} tout langage XML décrit par une instanciation du fragment de DTD relatif au langage XSLT (voir section 4.4.5). Pratiquement, nous vérifierons simplement que les feuilles de transformation engendrées par la traduction de programmes VXT sont des documents XML bien formés. Nous ferons une vérification syntaxique des expressions XPath associées aux attributs *match* et *select*, en utilisant la syntaxe abstraite définie par Wadler [228] que nous étendons légèrement ici afin de refléter certains changements entre la *working draft* XSLT de Décembre 1998 [58] et la recommandation finale [82] sur laquelle est basée la traduction VXT. Nous ajoutons deux *patterns* : *parent(p)* et *node()*. D'après la recommandation du W3C, *node()* sélectionne n'importe quel nœud à l'exclusion de la racine et des attributs. L'objet *star* de VXT a la même sémantique. Sa forme géométrique est en fait la composition d'un nœud élément (*square*) et d'un nœud texte (*lozenge*), qui se traduit par `*|text()`, c'est-à-dire les deux principales alternatives de *node()*. La version étendue de la syntaxe abstraite XPath proposée par Wadler est la suivante :

$$\begin{array}{ll}
XPath & \rightarrow p \\
XQual & \rightarrow q \\
n & \rightarrow Name \\
s & \rightarrow String \\
p & \rightarrow Pattern \quad p ::= p_1 \mid p_2 \mid /p \mid //p \mid p_1/p_2 \mid p_1//p_2 \mid p[q] \mid \\
& \quad n \mid * \mid @n \mid @* \mid \text{text}() \mid \text{comment}() \mid \text{pi}(n) \mid \text{pi}() \mid \text{id}(p) \mid \text{id}(s) \mid \\
& \quad \text{parent}(p) \mid \text{ancestor}(p) \mid \text{ancestor-or-self}(p) \mid . \mid .. \mid \text{node}() \\
q & \rightarrow Qualifier \quad q ::= q_1 \text{ and } q_2 \mid q_1 \text{ or } q_2 \mid \text{not}(q) \mid (q) \mid \\
& \quad p=s \mid p
\end{array}$$

Nous notons alors respectivement $\mathcal{L}(XPath)$ et $\mathcal{L}(XQual)$ les langages décrivant les *Patterns* XPath et les *Qualifiers* XPath. Nous aurons aussi besoin plus tard d'introduire l'élément vide noté ε qui sera retourné par certains cas des fonctions \mathcal{T}_P et \mathcal{T}_S . Nous notons alors $\mathcal{L}(eXPath)$ et $\mathcal{L}(eXQual)$ les langages engendrés par la grammaire ci-dessus augmentée des productions suivantes :

$$\begin{array}{ll}
eXPath & \rightarrow XPath \mid \varepsilon \\
eXQual & \rightarrow XQual \mid \varepsilon
\end{array}$$

La version de Wadler définit certains éléments obsolètes comme *first-of-type* que nous ne repreneons pas ici. D'autre part, cette syntaxe ne couvre pas l'ensemble de la spécification XPath [57] du W3C. En particulier, elle ne permet pas de modéliser toutes les expressions XPath associées aux attributs *select* des instructions XSLT, mais seulement celles, plus restreintes, associées aux attributs *match* des règles de transformation. Par exemple, certains axes comme *preceding-sibling::* et *namespace::* sont absents. Cette limitation ne pose pas de problème ici puisque, comme nous l'avons vu précédemment, VXT restreint lui-même les axes pouvant être utilisés dans les *VPMEs* et ne requiert pas de construction n'apparaissant pas dans la syntaxe abstraite de Wadler.

Nous allons maintenant définir les fonctions \mathcal{T}_R , \mathcal{T}_M , \mathcal{T}_P et \mathcal{T}_S composant la fonction de traduction \mathcal{T} . Le schéma de la figure 4.20 illustre les différentes composantes de la fonction de traduction et leur domaine d'application.

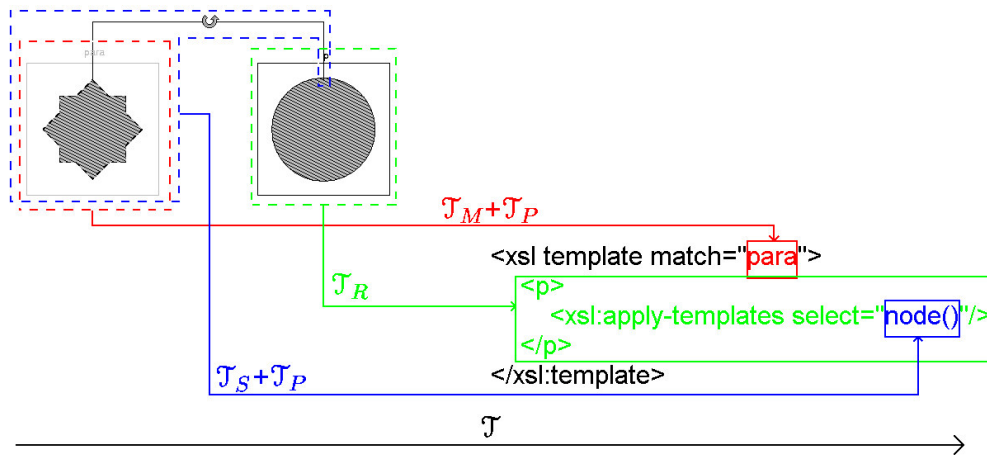


FIG. 4.20 : Combinaison des fonctions de traduction

Contexte de traduction

La traduction utilise un contexte $\Gamma = \varphi_1, \varphi_2, \varphi_3, \varphi_4$. Les index i et j correspondent à des références sur des racines de sous-arbres des *VPMEs* et des productions. Ces références ont été introduites dans la section 4.4.1.

L'élément φ_1 établit les relations entre les nœuds extraits de la *VPME* et la production. Il contient l'ensemble des instructions d'extraction et de transformation associées aux deux nœuds correspondants dans la *VPME* et dans la production (désignés par leurs index) :

$$\varphi_1 = \{ \langle i, j, op \rangle \} \text{ où } i \text{ et } j \text{ sont des indices désignant respectivement un nœud de } v \text{ et un nœud de } r \text{ et } op \text{ un opérateur de transformation VXT.}$$

φ_2 permet de stocker la valeur d'un nœud de type attribut dans la production, cette information n'apparaissant pas directement dans la représentation visuelle des règles de transformation (seul le nom des attributs est représenté) :

$$\varphi_2 = \{ \langle i, value \rangle \} \text{ où } i \text{ est un indice désignant un nœud de } r \text{ et } value \text{ une chaîne de caractères.}$$

De manière similaire, φ_3 stocke les valeurs associées aux nœuds #PCDATA de la production :

$$\varphi_3 = \{ \langle i, text \rangle \} \text{ où } i \text{ est un indice désignant un nœud de } r \text{ et } text \text{ une chaîne de caractères.}$$

φ_4 permet de relier les nœuds de type boucle *for-each* au nœud principal de l'itération, c'est-à-dire le nœud associé à l'instruction extrayant un ensemble d'éléments sources sur lequel l'itération est effectuée :

$$\varphi_4 = \{ \langle i, j \rangle \} \text{ où } i \text{ et } j \text{ sont des indices désignant des nœuds de } r.$$

Afin de rendre les définitions plus compactes, nous notons $\varphi, \langle i, j \rangle$ l'expression $\varphi \cup \{\langle i, j \rangle\}$ pour détailler un élément de φ .

Nous pouvons maintenant définir les différentes composantes de la fonction de traduction. Nous commençons par \mathcal{T}_R , fonction générant le corps des *template rules* XSLT.

Définition 14 (\mathcal{T}_R , Fonction de traduction des productions (parties droites des règles VXT))

On suppose t une chaîne de caractères conforme à la définition des noms qualifiés XML ([35], production [5] valable pour les noms d'éléments et les noms d'attributs).

Cette fonction est définie par deux composantes : \mathcal{T}_{R1} et \mathcal{T}_{R2} . Nous utilisons la notation \mathcal{T}_{Rc} avec $c \in \{1, 2\}$ pour désigner les cas d'applications communs aux deux composantes.

$$\begin{aligned}
\mathcal{T}_R \llbracket \cdot \rrbracket_v^\Gamma &: \mathcal{L}(AS_R) \rightarrow \mathcal{XS} \\
\mathcal{T}_R \llbracket X \rrbracket_v^\Gamma &= \mathcal{T}_{R1} \llbracket X \rrbracket_v^\Gamma & [R_0] \\
\mathcal{T}_{Rc} \llbracket \mathbf{Va}(X_1, X_2) \rrbracket_v^\Gamma &= \mathcal{T}_{Rc} \llbracket X_1 \rrbracket_v^\Gamma \mathcal{T}_{Rc} \llbracket X_2 \rrbracket_v^\Gamma & [R_1] \\
\mathcal{T}_{Rc} \llbracket \mathbf{Ha}(X_1, X_2) \rrbracket_v^\Gamma &= \mathcal{T}_{Rc} \llbracket X_1 \rrbracket_v^\Gamma \mathcal{T}_{Rc} \llbracket X_2 \rrbracket_v^\Gamma & [R_2] \\
\mathcal{T}_{Rc} \llbracket \mathbf{Va}(\mathbf{Txt}(t), \mathbf{Shp}_j(\mathbf{triangle}, \mathbf{gray})) \rrbracket_v^{\Gamma \subset \varphi_2, \langle j, value \rangle} &= \langle \text{xsl:attribute name}="t" > & [R_3] \\
&\quad \text{value} \\
&\quad \langle \text{/xsl:attribute} > \\
\mathcal{T}_{Rc} \llbracket \mathbf{Shp}_j(\mathbf{lozenge}, \mathbf{gray}) \rrbracket_v^{\Gamma \subset \varphi_3, \langle j, text \rangle} &= \text{text} & [R_4] \\
\mathcal{T}_{Rc} \llbracket \mathbf{Va}(\mathbf{Txt}(t), \mathbf{Shp}_j(\mathbf{square}, \mathbf{gray})) \rrbracket_v^\Gamma &= \langle t / > & [R_5] \\
\mathcal{T}_{Rc} \llbracket \mathbf{Va}(\mathbf{Txt}(t), \mathbf{Box}_j(X, \mathbf{gray})) \rrbracket_v^\Gamma &= \langle t > & [R_6] \\
&\quad \mathcal{T}_{R1} \llbracket X \rrbracket_v^\Gamma \\
&\quad \langle \text{/t} > \\
\mathcal{T}_{Rc} \llbracket \mathbf{Box}_j(X, \mathbf{hatched}) \rrbracket_v^{\Gamma \subset \varphi_4, \langle j, k \rangle} &= \langle \text{xsl:for-each select}=" \mathcal{T}_S \llbracket v \rrbracket^k " > & [R_7] \\
&\quad \mathcal{T}_{R2} \llbracket X \rrbracket_v^\Gamma \\
&\quad \langle \text{/xsl:for-each} > \\
\mathcal{T}_{Rc} \llbracket \mathbf{Box}_j(X, \mathbf{blue}) \rrbracket_v^{\Gamma \subset \varphi_1, \langle i, j, copy \rangle} &= \langle \text{xsl:copy} > & [R_8] \\
&\quad \mathcal{T}_{R1} \llbracket X \rrbracket_v^\Gamma \\
&\quad \langle \text{/xsl:copy} > \\
\mathcal{T}_{R1} \llbracket \mathbf{Shp}_j(\mathbf{square}, \mathbf{blue}) \rrbracket_v^{\Gamma \subset \varphi_1, \langle i, j, copy \rangle} &= \langle \text{xsl:copy-of select}=" \mathcal{T}_S \llbracket v \rrbracket^i " / > & [R_9] \\
\mathcal{T}_{R1} \llbracket \mathbf{Shp}_j(\mathbf{triangle}, \mathbf{blue}) \rrbracket_v^{\Gamma \subset \varphi_1, \langle i, j, copy \rangle} &= \langle \text{xsl:copy-of select}=" \mathcal{T}_S \llbracket v \rrbracket^i " / > & [R_{10}] \\
\mathcal{T}_{R1} \llbracket \mathbf{Shp}_j(\mathbf{lozenge}, \mathbf{blue}) \rrbracket_v^{\Gamma \subset \varphi_1, \langle i, j, copy \rangle} &= \langle \text{xsl:copy-of select}=" \mathcal{T}_S \llbracket v \rrbracket^i " / > & [R_{11}] \\
\mathcal{T}_{R1} \llbracket \mathbf{Shp}_j(\mathbf{lozenge}, \mathbf{blue}) \rrbracket_v^{\Gamma \subset \varphi_1, \langle i, j, text \rangle} &= \langle \text{xsl:value-of select}=" \mathcal{T}_S \llbracket v \rrbracket^i " / > & [R_{12}] \\
\mathcal{T}_{R1} \llbracket \mathbf{Shp}_j(\mathbf{circle}, \mathbf{blue}) \rrbracket_v^{\Gamma \subset \varphi_1, \langle i, j, copy \rangle} &= \langle \text{xsl:copy-of select}=" \mathcal{T}_S \llbracket v \rrbracket^i " / > & [R_{13}] \\
\mathcal{T}_{R1} \llbracket \mathbf{Shp}_j(\mathbf{circle}, \mathbf{blue}) \rrbracket_v^{\Gamma \subset \varphi_1, \langle i, j, text \rangle} &= \langle \text{xsl:value-of select}=" \mathcal{T}_S \llbracket v \rrbracket^i " / > & [R_{14}]
\end{aligned}$$

$$\mathcal{T}_{R1}[\text{Shp}_j(\text{circle}, \text{blue})]_v^{\Gamma^C \varphi_1, \langle i, j, \text{apply-rules} \rangle} = \langle \text{xsl:apply-templates select}=\mathcal{T}_S[v]^i \text{"/} \rangle \quad [R_{15}]$$

$$\mathcal{T}_{R2}[\text{Shp}_j(\text{square}, \text{blue})]_v^{\Gamma^C \varphi_1, \langle i, j, \text{copy} \rangle} = \langle \text{xsl:copy-of select}="." \text{/} \rangle \quad [R_{16}]$$

$$\mathcal{T}_{R2}[\text{Shp}_j(\text{triangle}, \text{blue})]_v^{\Gamma^C \varphi_1, \langle i, j, \text{copy} \rangle} = \langle \text{xsl:copy-of select}="." \text{/} \rangle \quad [R_{17}]$$

$$\mathcal{T}_{R2}[\text{Shp}_j(\text{lozenge}, \text{blue})]_v^{\Gamma^C \varphi_1, \langle i, j, \text{copy} \rangle} = \langle \text{xsl:copy-of select}="." \text{/} \rangle \quad [R_{18}]$$

$$\mathcal{T}_{R2}[\text{Shp}_j(\text{lozenge}, \text{blue})]_v^{\Gamma^C \varphi_1, \langle i, j, \text{text} \rangle} = \langle \text{xsl:value-of select}="." \text{/} \rangle \quad [R_{19}]$$

$$\mathcal{T}_{R2}[\text{Shp}_j(\text{circle}, \text{blue})]_v^{\Gamma^C \varphi_1, \langle i, j, \text{copy} \rangle} = \langle \text{xsl:copy-of select}="." \text{/} \rangle \quad [R_{20}]$$

$$\mathcal{T}_{R2}[\text{Shp}_j(\text{circle}, \text{blue})]_v^{\Gamma^C \varphi_1, \langle i, j, \text{text} \rangle} = \langle \text{xsl:value-of select}="." \text{/} \rangle \quad [R_{21}]$$

$$\mathcal{T}_{R2}[\text{Shp}_j(\text{circle}, \text{blue})]_v^{\Gamma^C \varphi_1, \langle i, j, \text{apply-rules} \rangle} = \langle \text{xsl:apply-templates select}="." \text{/} \rangle \quad [R_{22}]$$

Nous définissons ensuite la fonction \mathcal{T}_P , qui génère les prédicats représentant les conditions de sélection d'un nœud. Cette fonction n'est pas appelée directement par \mathcal{T} mais est utilisée dans la définition de \mathcal{T}_M et \mathcal{T}_S . Cette fonction, ainsi que les suivantes, utilise deux autres fonctions nommées *Pred* et *And*. *Pred* génère les délimiteurs de prédicats (crochets) quand ceux-ci sont nécessaires et *And* détermine s'il est nécessaire d'insérer le connecteur XPath and pour lier les prédicats passés en paramètre.

Définition 15 (*Pred*, Fonction de génération de délimiteurs de prédicats XPath)

$$\text{Pred} : \mathcal{L}(\text{XPath}), \mathcal{L}(\text{eXQual}) \rightarrow \mathcal{L}(\text{XPath})$$

$$\text{Pred}(a, b) = \begin{cases} a[b] & \text{si } (b \neq \varepsilon) \\ a & \text{si } (b = \varepsilon) \end{cases}$$

Définition 16 (*And*, Fonction de liaison de prédicats XPath)

$$\text{And} : \mathcal{L}(\text{eXQual}), \mathcal{L}(\text{eXQual}) \rightarrow \mathcal{L}(\text{eXQual})$$

$$\text{And}(a, b) = \begin{cases} a \text{ and } b & \text{si } (a \neq \varepsilon \wedge b \neq \varepsilon) \\ a & \text{si } (a \neq \varepsilon \wedge b = \varepsilon) \\ b & \text{si } (a = \varepsilon \wedge b \neq \varepsilon) \\ \varepsilon & \text{si } (a = \varepsilon \wedge b = \varepsilon) \end{cases}$$

Nous définissons alors la fonction \mathcal{T}_P .

Définition 17 (\mathcal{T}_P , **Fonction de traduction de prédicats**)

On suppose t une chaîne de caractères conforme à la définition des noms qualifiés XML ([35], production [5] valable pour les noms d'éléments et les noms d'attributs).

| | | |
|---|-----------------|--------------------|
| $\mathcal{T}_P[\] : \mathcal{L}(AS_V) \rightarrow \mathcal{L}(eXQual)$ | | |
| $\mathcal{T}_P[\text{Shp}(\text{square}, s_1 s_2 s_3 s_4 s_7 s_8)]$ | = * | [P ₁] |
| $\mathcal{T}_P[\text{Shp}(\text{square}, s_{10})]$ | = not(*) | [P ₂] |
| $\mathcal{T}_P[\text{Shp}(\text{square}, s_5 s_6)]$ | = //* | [P ₃] |
| $\mathcal{T}_P[\text{Shp}(\text{square}, s_9)]$ | = not(//*) | [P ₄] |
| $\mathcal{T}_P[\text{Shp}(\text{square}, s_{11} s_{12})]$ | = ε | [P ₅] |
| $\mathcal{T}_P[\text{Va}(\text{Txt}(n), \text{Shp}(\text{square}, s_1 s_2 s_3 s_4 s_7 s_8))]$ | = n | [P ₆] |
| $\mathcal{T}_P[\text{Va}(\text{Txt}(n), \text{Shp}(\text{square}, s_{10}))]$ | = not(n) | [P ₇] |
| $\mathcal{T}_P[\text{Va}(\text{Txt}(n), \text{Shp}(\text{square}, s_5 s_6))]$ | = // n | [P ₈] |
| $\mathcal{T}_P[\text{Va}(\text{Txt}(n), \text{Shp}(\text{square}, s_9))]$ | = not(// n) | [P ₉] |
| $\mathcal{T}_P[\text{Va}(\text{Txt}(n), \text{Shp}(\text{square}, s_{11} s_{12}))]$ | = ε | [P ₁₀] |
| $\mathcal{T}_P[\text{Shp}(\text{triangle}, s_1 s_2 s_3 s_4 s_7 s_8)]$ | = @* | [P ₁₁] |
| $\mathcal{T}_P[\text{Shp}(\text{triangle}, s_{10})]$ | = not(@*) | [P ₁₂] |
| $\mathcal{T}_P[\text{Shp}(\text{triangle}, s_5 s_6)]$ | = //@* | [P ₁₃] |
| $\mathcal{T}_P[\text{Shp}(\text{triangle}, s_9)]$ | = not(//@*) | [P ₁₄] |
| $\mathcal{T}_P[\text{Shp}(\text{triangle}, s_{11} s_{12})]$ | = ε | [P ₁₅] |
| $\mathcal{T}_P[\text{Va}(\text{Txt}(n), \text{Shp}(\text{triangle}, s_1 s_2 s_3 s_4 s_7 s_8))]$ | = @ n | [P ₁₆] |
| $\mathcal{T}_P[\text{Va}(\text{Txt}(n), \text{Shp}(\text{triangle}, s_{10}))]$ | = not(@ n) | [P ₁₇] |
| $\mathcal{T}_P[\text{Va}(\text{Txt}(n), \text{Shp}(\text{triangle}, s_5 s_6))]$ | = //@ n | [P ₁₈] |
| $\mathcal{T}_P[\text{Va}(\text{Txt}(n), \text{Shp}(\text{triangle}, s_9))]$ | = not(//@ n) | [P ₁₉] |
| $\mathcal{T}_P[\text{Va}(\text{Txt}(n), \text{Shp}(\text{triangle}, s_{11} s_{12}))]$ | = ε | [P ₂₀] |
| $\mathcal{T}_P[\text{Shp}(\text{lozenge}, s_1 s_2 s_3 s_4 s_7 s_8)]$ | = text() | [P ₂₁] |
| $\mathcal{T}_P[\text{Shp}(\text{lozenge}, s_{10})]$ | = not(text()) | [P ₂₂] |
| $\mathcal{T}_P[\text{Shp}(\text{lozenge}, s_5 s_6)]$ | = //text() | [P ₂₃] |

| | | |
|---|--|--------------------|
| $\mathcal{T}_P[\text{Shp}(\text{lozenge}, s_9)]$ | $= \text{not}(\text{//text}())$ | [P ₂₄] |
| $\mathcal{T}_P[\text{Shp}(\text{lozenge}, s_{11} s_{12})]$ | $= \varepsilon$ | [P ₂₅] |
| $\mathcal{T}_P[\text{Shp}(\text{star}, s_1 s_2 s_3 s_4 s_7 s_8)]$ | $= \text{node}()$ | [P ₂₆] |
| $\mathcal{T}_P[\text{Shp}(\text{star}, s_{10})]$ | $= \text{not}(\text{node}())$ | [P ₂₇] |
| $\mathcal{T}_P[\text{Shp}(\text{star}, s_5 s_6)]$ | $= \text{//node}()$ | [P ₂₈] |
| $\mathcal{T}_P[\text{Shp}(\text{star}, s_9)]$ | $= \text{not}(\text{//node}())$ | [P ₂₉] |
| $\mathcal{T}_P[\text{Shp}(\text{star}, s_{11} s_{12})]$ | $= \varepsilon$ | [P ₃₀] |
| $\mathcal{T}_P[\text{Box}(X, s_1 s_2 s_3 s_4 s_7 s_8)]$ <i>avec</i> $X \neq \mathbf{Va}(X_1, X_2)$ <i>sauf</i> $X = \mathbf{Va}(\mathbf{Txt}(n), X_2)$ | $= \text{Pred}(*, \mathcal{T}_P[X])$ | [P ₃₁] |
| $\mathcal{T}_P[\text{Box}(X, s_{10})]$ <i>avec</i> $X \neq \mathbf{Va}(X_1, X_2)$ <i>sauf</i> $X = \mathbf{Va}(\mathbf{Txt}(n), X_2)$ | $= \text{not}(\text{Pred}(*, \mathcal{T}_P[X]))$ | [P ₃₂] |
| $\mathcal{T}_P[\text{Box}(X, s_5 s_6)]$ <i>avec</i> $X \neq \mathbf{Va}(X_1, X_2)$ <i>sauf</i> $X = \mathbf{Va}(\mathbf{Txt}(n), X_2)$ | $= \text{Pred}(\text{//}, \mathcal{T}_P[X])$ | [P ₃₃] |
| $\mathcal{T}_P[\text{Box}(X, s_9)]$ <i>avec</i> $X \neq \mathbf{Va}(X_1, X_2)$ <i>sauf</i> $X = \mathbf{Va}(\mathbf{Txt}(n), X_2)$ | $= \text{not}(\text{Pred}(\text{//}, \mathcal{T}_P[X]))$ | [P ₃₄] |
| $\mathcal{T}_P[\text{Box}(X, s_{11} s_{12})]$ <i>avec</i> $X \neq \mathbf{Va}(X_1, X_2)$ <i>sauf</i> $X = \mathbf{Va}(\mathbf{Txt}(n), X_2)$ | $= \varepsilon$ | [P ₃₅] |
| $\mathcal{T}_P[\mathbf{Va}(\mathbf{Txt}(n), \text{Box}(X, s_1 s_2 s_3 s_4 s_7 s_8))]$ <i>avec</i> $X \neq \mathbf{Va}(X_1, X_2)$ <i>sauf</i> $X = \mathbf{Va}(\mathbf{Txt}(n), X_2)$ | $= \text{Pred}(\mathbf{n}, \mathcal{T}_P[X])$ | [P ₃₆] |
| $\mathcal{T}_P[\mathbf{Va}(\mathbf{Txt}(n), \text{Box}(X, s_{10}))]$ <i>avec</i> $X \neq \mathbf{Va}(X_1, X_2)$ <i>sauf</i> $X = \mathbf{Va}(\mathbf{Txt}(n), X_2)$ | $= \text{not}(\text{Pred}(\mathbf{n}, \mathcal{T}_P[X]))$ | [P ₃₇] |
| $\mathcal{T}_P[\mathbf{Va}(\mathbf{Txt}(n), \text{Box}(X, s_5 s_6))]$ <i>avec</i> $X \neq \mathbf{Va}(X_1, X_2)$ <i>sauf</i> $X = \mathbf{Va}(\mathbf{Txt}(n), X_2)$ | $= \text{Pred}(\text{//}\mathbf{n}, \mathcal{T}_P[X])$ | [P ₃₈] |
| $\mathcal{T}_P[\mathbf{Va}(\mathbf{Txt}(n), \text{Box}(X, s_9))]$ <i>avec</i> $X \neq \mathbf{Va}(X_1, X_2)$ <i>sauf</i> $X = \mathbf{Va}(\mathbf{Txt}(n), X_2)$ | $= \text{not}(\text{Pred}(\text{//}\mathbf{n}, \mathcal{T}_P[X]))$ | [P ₃₉] |
| $\mathcal{T}_P[\mathbf{Va}(\mathbf{Txt}(n), \text{Box}(X, s_{11} s_{12}))]$ <i>avec</i> $X \neq \mathbf{Va}(X_1, X_2)$ <i>sauf</i> $X = \mathbf{Va}(\mathbf{Txt}(n), X_2)$ | $= \varepsilon$ | [P ₄₀] |
| $\mathcal{T}_P[\text{Box}(\mathbf{Va}(X_1, X_2), s_1 s_2 s_3 s_4 s_7 s_8)]$ <i>avec</i> $X_1 \neq \mathbf{Txt}(n)$ | $= \text{Pred}(*, \text{And}(\mathcal{T}_P[X_1], \mathcal{T}_P[X_2]))$ | [P ₄₁] |
| $\mathcal{T}_P[\text{Box}(\mathbf{Va}(X_1, X_2), s_{10})]$ <i>avec</i> $X_1 \neq \mathbf{Txt}(n)$ | $= \text{not}(\text{Pred}(*, \text{And}(\mathcal{T}_P[X_1], \mathcal{T}_P[X_2])))$ | [P ₄₂] |
| $\mathcal{T}_P[\text{Box}(\mathbf{Va}(X_1, X_2), s_5 s_6)]$ <i>avec</i> $X_1 \neq \mathbf{Txt}(n)$ | $= \text{Pred}(\text{//}, \text{And}(\mathcal{T}_P[X_1], \mathcal{T}_P[X_2]))$ | [P ₄₃] |
| $\mathcal{T}_P[\text{Box}(\mathbf{Va}(X_1, X_2), s_9)]$ <i>avec</i> $X_1 \neq \mathbf{Txt}(n)$ | $= \text{not}(\text{Pred}(\text{//}, \text{And}(\mathcal{T}_P[X_1], \mathcal{T}_P[X_2])))$ | [P ₄₄] |

$$\begin{aligned}
\mathcal{J}_P[\mathbf{Box}(\mathbf{Va}(X_1, X_2), s_{11}|s_{12})] &= \varepsilon & [P_{45}] \\
\text{avec } X_1 \neq \mathbf{Txt}(n) & & \\
\mathcal{J}_P[\mathbf{Va}(\mathbf{Txt}(n), \mathbf{Box}(\mathbf{Va}(X_1, X_2), s_1|s_2|s_3|s_4|s_7|s_8))] &= \mathit{Pred}(\mathbf{n}, \mathit{And}(\mathcal{J}_P[X_1], \mathcal{J}_P[X_2])) & [P_{46}] \\
\text{avec } X_1 \neq \mathbf{Txt}(n) & & \\
\mathcal{J}_P[\mathbf{Va}(\mathbf{Txt}(n), \mathbf{Box}(\mathbf{Va}(X_1, X_2), s_{10}))] &= \mathit{not}(\mathit{Pred}(\mathbf{n}, \mathit{And}(\mathcal{J}_P[X_1], \mathcal{J}_P[X_2]))) & [P_{47}] \\
\text{avec } X_1 \neq \mathbf{Txt}(n) & & \\
\mathcal{J}_P[\mathbf{Va}(\mathbf{Txt}(n), \mathbf{Box}(\mathbf{Va}(X_1, X_2), s_5|s_6))] &= \mathit{Pred}(\mathbf{/n}, \mathit{And}(\mathcal{J}_P[X_1], \mathcal{J}_P[X_2])) & [P_{48}] \\
\text{avec } X_1 \neq \mathbf{Txt}(n) & & \\
\mathcal{J}_P[\mathbf{Va}(\mathbf{Txt}(n), \mathbf{Box}(\mathbf{Va}(X_1, X_2), s_9))] &= \mathit{not}(\mathit{Pred}(\mathbf{/n}, \mathit{And}(\mathcal{J}_P[X_1], \mathcal{J}_P[X_2]))) & [P_{49}] \\
\text{avec } X_1 \neq \mathbf{Txt}(n) & & \\
\mathcal{J}_P[\mathbf{Va}(\mathbf{Txt}(n), \mathbf{Box}(\mathbf{Va}(X_1, X_2), s_{11}|s_{12}))] &= \varepsilon & [P_{50}] \\
\text{avec } X_1 \neq \mathbf{Txt}(n) & & \\
\mathcal{J}_P[\mathbf{Ha}(X_1, X_2)] &= \mathit{And}(\mathcal{J}_P[X_1], \mathcal{J}_P[X_2]) & [P_{51}]
\end{aligned}$$

Nous définissons maintenant \mathcal{J}_M , fonction de traduction de *VPMEs* en attributs *match* de règles XSLT.

Définition 18 (\mathcal{J}_M , Fonction de traduction de *VPME* en attribut *match*)

On suppose t une chaîne de caractères conforme à la définition des noms qualifiés XML ([35], production [5] valable pour les noms d'éléments et les noms d'attributs).

$$\begin{aligned}
\mathcal{J}_M[\] : \mathcal{V} &\rightarrow \mathcal{L}(XPath) \\
\mathcal{J}_M[\mathbf{Shp}(\mathbf{square}, s_1|s_2|s_3|s_4)]_v &= * & [M_1] \\
\mathcal{J}_M[\mathbf{Va}(\mathbf{Txt}(n), \mathbf{Shp}(\mathbf{square}, s_1|s_2|s_3|s_4))]_v &= \mathbf{n} & [M_2] \\
\mathcal{J}_M[\mathbf{Shp}(\mathbf{triangle}, s_1|s_2|s_3|s_4)]_v &= @* & [M_3] \\
\mathcal{J}_M[\mathbf{Va}(\mathbf{Txt}(n), \mathbf{Shp}(\mathbf{triangle}, s_1|s_2|s_3|s_4))]_v &= @\mathbf{n} & [M_4] \\
\mathcal{J}_M[\mathbf{Shp}(\mathbf{lozenge}, s_1|s_2|s_3|s_4)]_v &= \mathbf{text}() & [M_5] \\
\mathcal{J}_M[\mathbf{Shp}(\mathbf{star}, s_1|s_2|s_3|s_4)]_v &= \mathbf{node}() & [M_6] \\
\mathcal{J}_M[\mathbf{Box}(X, s_1|s_2|s_3|s_4)]_v &= \mathcal{J}_P[\mathbf{Box}(X, s)] & [M_7] \\
\text{avec } X \neq \mathbf{Va}(X_1, X_2) \text{ sauf } X = \mathbf{Va}(\mathbf{Txt}(n), X_2) & & \\
\mathcal{J}_M[\mathbf{Box}_i(X, s_7|s_8)]_v &= \mathit{Pred}(*, \mathcal{J}_P[X]) / \mathcal{J}_M[\mathit{gscn}(v, i, 0)]_v & [M_8] \\
\text{avec } X \neq \mathbf{Va}(X_1, X_2) \text{ sauf } X = \mathbf{Va}(\mathbf{Txt}(n), X_2) & & \\
\mathcal{J}_M[\mathbf{Box}_i(X, s_5|s_6|s_{11}|s_{12})]_v &= \mathit{Pred}(*, \mathcal{J}_P[X]) // \mathcal{J}_M[\mathit{gscn}(v, i, 0)]_v & [M_9] \\
\text{avec } X \neq \mathbf{Va}(X_1, X_2) \text{ sauf } X = \mathbf{Va}(\mathbf{Txt}(n), X_2) & & \\
\mathcal{J}_M[\mathbf{Va}(\mathbf{Txt}(n), \mathbf{Box}(X, s_1|s_2|s_3|s_4))]_v &= \mathcal{J}_P[\mathbf{Va}(\mathbf{Txt}(n), \mathbf{Box}(X, s))] & [M_{10}] \\
\text{avec } X \neq \mathbf{Va}(X_1, X_2) \text{ sauf } X = \mathbf{Va}(\mathbf{Txt}(n), X_2) & &
\end{aligned}$$

$$\mathcal{T}_M \llbracket \mathbf{Va}(\mathbf{Txt}(n), \mathbf{Box}_i(X, s_7|s_8)) \rrbracket_v \quad \text{avec } X \neq \mathbf{Va}(X_1, X_2) \text{ sauf } X = \mathbf{Va}(\mathbf{Txt}(n), X_2) = \text{Pred}(n, \mathcal{T}_P \llbracket X \rrbracket) // \mathcal{T}_M \llbracket gscn(v, i, 0) \rrbracket_v \quad [M_{11}]$$

$$\mathcal{T}_M \llbracket \mathbf{Va}(\mathbf{Txt}(n), \mathbf{Box}_i(X, s_5|s_6|s_{11}|s_{12})) \rrbracket_v \quad \text{avec } X \neq \mathbf{Va}(X_1, X_2) \text{ sauf } X = \mathbf{Va}(\mathbf{Txt}(n), X_2) = \text{Pred}(n, \mathcal{T}_P \llbracket X \rrbracket) // \mathcal{T}_M \llbracket gscn(v, i, 0) \rrbracket_v \quad [M_{12}]$$

$$\mathcal{T}_M \llbracket \mathbf{Box}(\mathbf{Va}(X_1, X_2), s_1|s_2|s_3|s_4) \rrbracket_v = \mathcal{T}_P \llbracket \mathbf{Box}(\mathbf{Va}(X_1, X_2), s) \rrbracket \quad [M_{13}]$$

$$\mathcal{T}_M \llbracket \mathbf{Box}_i(\mathbf{Va}(X_1, X_2), s_7|s_8) \rrbracket_v \quad \text{avec } X_1 \neq \mathbf{Txt}(n) = \text{Pred}(*, \text{And}(\mathcal{T}_P \llbracket X_1 \rrbracket, \mathcal{T}_P \llbracket X_2 \rrbracket)) // \mathcal{T}_M \llbracket gscn(v, i, 0) \rrbracket_v \quad [M_{14}]$$

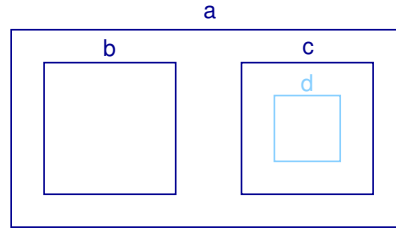
$$\mathcal{T}_M \llbracket \mathbf{Box}_i(\mathbf{Va}(X_1, X_2), s_5|s_6|s_{11}|s_{12}) \rrbracket_v \quad \text{avec } X_1 \neq \mathbf{Txt}(n) = \text{Pred}(*, \text{And}(\mathcal{T}_P \llbracket X_1 \rrbracket, \mathcal{T}_P \llbracket X_2 \rrbracket)) // \mathcal{T}_M \llbracket gscn(v, i, 0) \rrbracket_v \quad [M_{15}]$$

$$\mathcal{T}_M \llbracket \mathbf{Va}(\mathbf{Txt}(n), \mathbf{Box}(\mathbf{Va}(X_1, X_2), s_1|s_2|s_3|s_4)) \rrbracket_v = \mathcal{T}_P \llbracket \mathbf{Va}(\mathbf{Txt}(n), \mathbf{Box}(\mathbf{Va}(X_1, X_2), s)) \rrbracket \quad [M_{16}]$$

$$\mathcal{T}_M \llbracket \mathbf{Va}(\mathbf{Txt}(n), \mathbf{Box}_i(\mathbf{Va}(X_1, X_2), s_7|s_8)) \rrbracket_v \quad \text{avec } X_1 \neq \mathbf{Txt}(n) = \text{Pred}(n, \text{And}(\mathcal{T}_P \llbracket X_1 \rrbracket, \mathcal{T}_P \llbracket X_2 \rrbracket)) // \mathcal{T}_M \llbracket gscn(v, i, 0) \rrbracket_v \quad [M_{17}]$$

$$\mathcal{T}_M \llbracket \mathbf{Va}(\mathbf{Txt}(n), \mathbf{Box}_i(\mathbf{Va}(X_1, X_2), s_5|s_6|s_{11}|s_{12})) \rrbracket_v \quad \text{avec } X_1 \neq \mathbf{Txt}(n) = \text{Pred}(n, \text{And}(\mathcal{T}_P \llbracket X_1 \rrbracket, \mathcal{T}_P \llbracket X_2 \rrbracket)) // \mathcal{T}_M \llbracket gscn(v, i, 0) \rrbracket_v \quad [M_{18}]$$

En appelant la fonction \mathcal{T}_P pour générer les prédicats associés aux ancêtres du nœud contextuel, nous obtenons une expression XPath non optimale car contenant des conditions de sélection redondantes, puisque les nœuds sur l'axe principal se trouvent aussi exprimés en tant que prédicats de leur parents respectifs. Par exemple, pour la VPME suivante où d est le nœud contextuel :



\mathcal{T}_M générera

$$a[b \text{ and } c]/c[d]/d$$

alors que l'expression optimale est

$$a[b]/c/d$$

Les deux expressions sont équivalentes et sélectionnent donc le même ensemble de nœuds. La version non optimisée peut cependant être plus coûteuse en temps puisque le processeur XSLT aura parfois à vérifier deux fois la même condition, exprimée de deux manières différentes. L'implémentation Java de la fonction de traduction tient compte de ce problème et génère des expressions XPath optimisées.

Note : la fonction \mathcal{T}_M n'est pas définie pour certains styles comme s_9 et s_{10} car ces cas ne peuvent pas se produire si l'on fait l'hypothèse que les VPMEs traduites appartiennent à $\mathcal{L}(\mathcal{AS}_V)$ et qu'elles vérifient VF1 et VF2. La fonction \mathcal{T}_M est donc complète par rapport aux VPMEs vérifiant VF1 et VF2, et non pas par rapport à toute expression appartenant à $\mathcal{L}(\mathcal{AS}_V)$ (la propriété de complétude de la fonction \mathcal{T}_M est établie dans la section 4.4.5).

La fonction précédente nous permet de traduire les expressions XPath associées aux attributs *match* des éléments `xsl:template`. Nous devons encore définir la fonction de traduction \mathcal{T}_S permettant d'obtenir l'expression XPath équivalente à un nœud de *VPME* devant être extrait du document source (que nous nommons par la suite *nœud sélectionné*). Cette expression sera la valeur de l'attribut *select* de l'instruction XSLT associée à cette opération d'extraction/transformation. La fonction \mathcal{T}_S est composée de trois fonctions \mathcal{T}_{S1} , \mathcal{T}_{S2} et \mathcal{T}_{S3} . Suivant la position du nœud sélectionné par rapport au nœud contextuel dans la *VPME*, une seule de ces fonctions est utilisée pour générer l'expression XPath adressant le nœud sélectionné à partir du nœud contextuel.

Nous allons dans la suite utiliser la fonction *gcca* (pour *Get Closest Common Ancestor*) qui, étant donné une *VPME* et un nœud sélectionné (désigné par son index), retourne l'index du nœud racine du plus petit sous-arbre de la *VPME* contenant à la fois le nœud contextuel et le nœud sélectionné, c'est-à-dire le sous-arbre dont la racine est le nœud le plus proche possible du nœud sélectionné et ancêtre à la fois du nœud contextuel et du nœud sélectionné. L'opération consistant à identifier le plus proche ancêtre commun de deux nœuds d'un arbre est une opération commune et bien définie dans le cadre général de la manipulation d'arbres. Nous nous contentons donc ici de définir la signature de notre fonction.

Définition 19 (*gcca* : Fonction d'extraction du plus petit sous-arbre contenant le nœud d'indice i et le nœud contextuel)

$$gcca : \mathcal{V} \times index \rightarrow index$$

Définition 20 (\mathcal{T}_S , Fonction de traduction de *VPME* en attribut *select*)

$$\mathcal{T}_S[\]^i : \mathcal{L}(\mathcal{AS}_V) \rightarrow \mathcal{L}(XPath)$$

$$\mathcal{T}_S[v]^i \quad \text{avec } i = 0 \quad = \quad . \quad (S_{0a})$$

$$\mathcal{T}_S[v]^i \quad \text{avec } card(gp(v, 0, i)) \geq 2 \quad = \quad . \mathcal{T}_{S1}[\text{gscn}(v, 0, i)]_v^i \quad (S_{0b})$$

$$\mathcal{T}_S[v]^i \quad \text{avec } card(gp(v, i, 0)) \geq 2 \quad = \quad . \mathcal{T}_{S2}[\text{i2v}(v, i)]_v \quad (S_{0c})$$

$$\mathcal{T}_S[v]^i \quad \text{avec } card(gp(v, i, 0)) = card(gp(v, 0, i)) = 0 \wedge i \neq 0 \quad = \quad . \mathcal{T}_{S3}[\text{i2v}(v, gcca(v, i))]_v^i \quad (S_{0d})$$

card est la fonction standard retournant la cardinalité d'un ensemble ou d'une séquence. La première option ($i = 0$) définit le cas où le nœud sélectionné est aussi le nœud contextuel (qui a toujours par convention l'index 0). La seconde ($card(gp(v, 0, i)) \geq 2$), le cas où le nœud sélectionné est un descendant du nœud contextuel. La troisième ($card(gp(v, i, 0)) \geq 2$) représente le cas où le nœud sélectionné est un ancêtre du nœud contextuel. Enfin, la dernière option représente le reste des solutions, c'est-à-dire les cas où le nœud contextuel et le nœud sélectionné sont dans des branches différentes mais ont un ancêtre commun.

Le point (.) inséré devant chaque appel aux fonctions \mathcal{T}_{S_i} sert à référencer le nœud contextuel dans l'expression XPath. Toutes les expressions retournées par \mathcal{T}_{S_i} ont donc la forme $.X$ où X commence par le séparateur slash (/).

Définition 21 (\mathcal{T}_{S_1} , 1er cas de traduction de VPME en attribut *select*)

On suppose t une chaîne de caractères conforme à la définition des noms qualifiés XML ([35], production [5] valable pour les noms d'éléments et les noms d'attributs).

$$\mathcal{T}_{S_1}[\] : \mathcal{L}(AS_V) \rightarrow \mathcal{L}(XPath)$$

$$\mathcal{T}_{S_1}[\mathbf{Shp}_i(\mathbf{square}, s_7|s_{12})]_v^i = /* \quad [S_{1a}]$$

$$\mathcal{T}_{S_1}[\mathbf{Shp}_i(\mathbf{square}, s_5|s_{11})]_v^i = // * \quad [S_{1b}]$$

$$\mathcal{T}_{S_1}[\mathbf{Va}(\mathbf{Txt}(n), \mathbf{Shp}_i(\mathbf{square}, s_7|s_{12}))]_v^i = /n \quad [S_{1c}]$$

$$\mathcal{T}_{S_1}[\mathbf{Va}(\mathbf{Txt}(n), \mathbf{Shp}_i(\mathbf{square}, s_5|s_{11}))]_v^i = //n \quad [S_{1d}]$$

$$\mathcal{T}_{S_1}[\mathbf{Shp}_i(\mathbf{triangle}, s_7|s_{12})]_v^i = /@ * \quad [S_{1e}]$$

$$\mathcal{T}_{S_1}[\mathbf{Shp}_i(\mathbf{triangle}, s_5|s_{11})]_v^i = // @ * \quad [S_{1f}]$$

$$\mathcal{T}_{S_1}[\mathbf{Va}(\mathbf{Txt}(n), \mathbf{Shp}_i(\mathbf{triangle}, s_7|s_{12}))]_v^i = /@n \quad [S_{1g}]$$

$$\mathcal{T}_{S_1}[\mathbf{Va}(\mathbf{Txt}(n), \mathbf{Shp}_i(\mathbf{triangle}, s_5|s_{11}))]_v^i = // @n \quad [S_{1h}]$$

$$\mathcal{T}_{S_1}[\mathbf{Shp}_i(\mathbf{lozenge}, s_7|s_{12})]_v^i = /text() \quad [S_{1i}]$$

$$\mathcal{T}_{S_1}[\mathbf{Shp}_i(\mathbf{lozenge}, s_5|s_{11})]_v^i = //text() \quad [S_{1j}]$$

$$\mathcal{T}_{S_1}[\mathbf{Shp}_i(\mathbf{star}, s_7|s_{12})]_v^i = /node() \quad [S_{1k}]$$

$$\mathcal{T}_{S_1}[\mathbf{Shp}_i(\mathbf{star}, s_5|s_{11})]_v^i = //node() \quad [S_{1l}]$$

$$\mathcal{T}_{S_1}[\mathbf{Box}_j(X, s_7|s_8|s_{12})]_v^i = \text{Pred}(/*, \mathcal{T}_P[X]) \mathcal{T}_{S_1}[\text{gscn}(v, j, i)]^i \quad [S_{1m}]$$

avec $X \neq \mathbf{Va}(X_1, X_2)$ et $i \neq j$

$$\mathcal{T}_{S_1}[\mathbf{Box}_j(X, s_5|s_6|s_{11})]_v^i = \text{Pred}(// *, \mathcal{T}_P[X]) \mathcal{T}_{S_1}[\text{gscn}(v, j, i)]^i \quad [S_{1n}]$$

avec $X \neq \mathbf{Va}(X_1, X_2)$ et $i \neq j$

$$\mathcal{T}_{S_1}[\mathbf{Box}_i(X, s_7|s_{12})]_v^i = \text{Pred}(/*, \mathcal{T}_P[X]) \quad [S_{1o}]$$

avec $X \neq \mathbf{Va}(X_1, X_2)$

$$\mathcal{T}_{S_1}[\mathbf{Box}_i(X, s_5|s_{11})]_v^i = \text{Pred}(// *, \mathcal{T}_P[X]) \quad [S_{1p}]$$

avec $X \neq \mathbf{Va}(X_1, X_2)$

$$\mathcal{T}_{S_1}[\mathbf{Va}(\mathbf{Txt}(n), \mathbf{Box}_j(X, s_7|s_8|s_{12}))]_v^i = \text{Pred}(/n, \mathcal{T}_P[X]) \mathcal{T}_{S_1}[\text{gscn}(v, j, i)]^i \quad [S_{1q}]$$

avec $X \neq \mathbf{Va}(X_1, X_2)$ et $i \neq j$

$$\mathcal{T}_{S_1}[\mathbf{Va}(\mathbf{Txt}(n), \mathbf{Box}_j(X, s_5|s_6|s_{11}))]_v^i = \text{Pred}(//n, \mathcal{T}_P[X]) \mathcal{T}_{S_1}[\text{gscn}(v, j, i)]^i \quad [S_{1r}]$$

avec $X \neq \mathbf{Va}(X_1, X_2)$ et $i \neq j$

| | | | |
|--|---|--|--------------------|
| $\mathcal{T}_{S1}[\mathbf{Va}(\mathbf{Txt}(n), \mathbf{Box}_i(X, s_7 s_{12}))]_v^i$ avec $X \neq \mathbf{Va}(X_1, X_2)$ | = | $Pred(/n, \mathcal{T}_P[X])$ | [S _{1s}] |
| $\mathcal{T}_{S1}[\mathbf{Va}(\mathbf{Txt}(n), \mathbf{Box}_i(X, s_5 s_{11}))]_v^i$ avec $X \neq \mathbf{Va}(X_1, X_2)$ | = | $Pred(//n, \mathcal{T}_P[X])$ | [S _{1t}] |
| $\mathcal{T}_{S1}[\mathbf{Box}_j(\mathbf{Va}(X_1, X_2), s_7 s_8 s_{12}))]_v^i$ avec $i \neq j$ | = | $Pred(/*, And(\mathcal{T}_P[X_1], \mathcal{T}_P[X_2])) \mathcal{T}_{S1}[gscn(v, j, i)]^i$ | [S _{1u}] |
| $\mathcal{T}_{S1}[\mathbf{Box}_j(\mathbf{Va}(X_1, X_2), s_5 s_6 s_{11}))]_v^i$ avec $i \neq j$ | = | $Pred(//*, And(\mathcal{T}_P[X_1], \mathcal{T}_P[X_2])) \mathcal{T}_{S1}[gscn(v, j, i)]^i$ | [S _{1v}] |
| $\mathcal{T}_{S1}[\mathbf{Box}_i(\mathbf{Va}(X_1, X_2), s_7 s_{12}))]_v^i$ | = | $Pred(/*, And(\mathcal{T}_P[X_1], \mathcal{T}_P[X_2]))$ | [S _{1w}] |
| $\mathcal{T}_{S1}[\mathbf{Box}_i(\mathbf{Va}(X_1, X_2), s_5 s_{11}))]_v^i$ | = | $Pred(//*, And(\mathcal{T}_P[X_1], \mathcal{T}_P[X_2]))$ | [S _{1x}] |
| $\mathcal{T}_{S1}[\mathbf{Va}(\mathbf{Txt}(n), \mathbf{Box}_j(\mathbf{Va}(X_1, X_2), s_7 s_8 s_{12}))]_v^i$ avec $i \neq j$ | = | $Pred(/n, And(\mathcal{T}_P[X_1], \mathcal{T}_P[X_2])) \mathcal{T}_{S1}[gscn(v, j, i)]^i$ | [S _{1y}] |
| $\mathcal{T}_{S1}[\mathbf{Va}(\mathbf{Txt}(n), \mathbf{Box}_j(\mathbf{Va}(X_1, X_2), s_5 s_6 s_{11}))]_v^i$ avec $i \neq j$ | = | $Pred(//n, And(\mathcal{T}_P[X_1], \mathcal{T}_P[X_2])) \mathcal{T}_{S1}[gscn(v, j, i)]^i$ | [S _{1z}] |
| $\mathcal{T}_{S1}[\mathbf{Va}(\mathbf{Txt}(n), \mathbf{Box}_i(\mathbf{Va}(X_1, X_2), s_7 s_{12}))]_v^i$ | = | $Pred(/n, And(\mathcal{T}_P[X_1], \mathcal{T}_P[X_2]))$ | [S _{1α}] |
| $\mathcal{T}_{S1}[\mathbf{Va}(\mathbf{Txt}(n), \mathbf{Box}_i(\mathbf{Va}(X_1, X_2), s_5 s_{11}))]_v^i$ | = | $Pred(//n, And(\mathcal{T}_P[X_1], \mathcal{T}_P[X_2]))$ | [S _{1β}] |

Définition 22 (\mathcal{T}_{S2} , 2ème cas de traduction de VPME en attribut *select*)

On suppose t une chaîne de caractères conforme à la définition des noms qualifiés XML ([35], production [5] valable pour les noms d'éléments et les noms d'attributs).

| | | |
|--|---|--|
| $\mathcal{T}_{S2}[\] : \mathcal{L}(AS_V) \rightarrow \mathcal{L}(eXPath)$ | | |
| $\mathcal{T}_{S2}[\mathbf{Shp}_i(f, s_1 s_2 s_3 s_4)]_v$ | = | ε [S _{2a}] |
| $\mathcal{T}_{S2}[\mathbf{Box}_i(X, s_1 s_2 s_3 s_4)]_v$ | = | ε [S _{2b}] |
| $\mathcal{T}_{S2}[\mathbf{Va}(\mathbf{Txt}(n), \mathbf{Shp}_i(f, s_1 s_2 s_3 s_4))]_v$ | = | ε [S _{2c}] |
| $\mathcal{T}_{S2}[\mathbf{Va}(\mathbf{Txt}(n), \mathbf{Box}_i(X, s_1 s_2 s_3 s_4))]_v$ | = | ε [S _{2d}] |
| $\mathcal{T}_{S2}[\mathbf{Box}_i(X, s_7 s_8 s_{12})]_v$ avec $X \neq \mathbf{Va}(X_1, X_2)$ | = | $\mathcal{T}_{S2}[gscn(v, i, 0)]_v / \mathbf{parent}(Pred(*, \mathcal{T}_P[X]))$ [S _{2e}] |
| $\mathcal{T}_{S2}[\mathbf{Box}_i(X, s_5 s_6 s_{11})]_v$ avec $X \neq \mathbf{Va}(X_1, X_2)$ | = | $\mathcal{T}_{S2}[gscn(v, i, 0)]_v / \mathbf{ancestor}(Pred(*, \mathcal{T}_P[X]))$ [S _{2f}] |
| $\mathcal{T}_{S2}[\mathbf{Box}_i(\mathbf{Va}(X_1, X_2), s_7 s_8 s_{12})]_v$ | = | $\mathcal{T}_{S2}[gscn(v, i, 0)]_v / \mathbf{parent}(Pred(*, And(\mathcal{T}_P[X_1], \mathcal{T}_P[X_2])))$ [S _{2g}] |
| $\mathcal{T}_{S2}[\mathbf{Box}_i(\mathbf{Va}(X_1, X_2), s_5 s_6 s_{11})]_v$ | = | $\mathcal{T}_{S2}[gscn(v, i, 0)]_v / \mathbf{ancestor}(Pred(*, And(\mathcal{T}_P[X_1], \mathcal{T}_P[X_2])))$ [S _{2i}] |

$$\mathcal{T}_{S2}[\mathbf{Va}(\mathbf{Txt}(n), \mathbf{Box}_i(X, s_7|s_8|s_{12}))]_v \quad \text{avec } X \neq \mathbf{Va}(X_1, X_2) = \mathcal{T}_{S2}[\mathit{gscn}(v, i, 0)]_v / \mathit{parent}(\mathit{Pred}(n, \mathcal{T}_P[X])) \quad [S_{2k}]$$

$$\mathcal{T}_{S2}[\mathbf{Va}(\mathbf{Txt}(n), \mathbf{Box}_i(X, s_5|s_6|s_{11}))]_v \quad \text{avec } X \neq \mathbf{Va}(X_1, X_2) = \mathcal{T}_{S2}[\mathit{gscn}(v, i, 0)]_v / \mathit{ancestor}(\mathit{Pred}(n, \mathcal{T}_P[X])) \quad [S_{2l}]$$

$$\mathcal{T}_{S2}[\mathbf{Va}(\mathbf{Txt}(n), \mathbf{Box}_i(\mathbf{Va}(X_1, X_2), s_7|s_8|s_{12}))]_v = \mathcal{T}_{S2}[\mathit{gscn}(v, i, 0)]_v / \mathit{parent}(\mathit{Pred}(n, \mathit{And}(\mathcal{T}_P[X_1], \mathcal{T}_P[X_2]))) \quad [S_{2m}]$$

$$\mathcal{T}_{S2}[\mathbf{Va}(\mathbf{Txt}(n), \mathbf{Box}_i(\mathbf{Va}(X_1, X_2), s_5|s_6|s_{11}))]_v = \mathcal{T}_{S2}[\mathit{gscn}(v, i, 0)]_v / \mathit{ancestor}(\mathit{Pred}(n, \mathit{And}(\mathcal{T}_P[X_1], \mathcal{T}_P[X_2]))) \quad [S_{2o}]$$

Définition 23 (\mathcal{T}_{S3} , 3ème cas de traduction de VPME en attribut *select*)

On suppose t une chaîne de caractères conforme à la définition des noms qualifiés XML ([35], production [5] valable pour les noms d'éléments et les noms d'attributs).

$$\mathcal{T}_{S3}[\] : \mathcal{L}(AS_V) \rightarrow \mathcal{L}(XPath)$$

$$\mathcal{T}_{S3}[\mathbf{Box}_j(X, s_7|s_8|s_{12})]_v^i = \mathcal{T}_{S2}[\mathit{gscn}(v, j, 0)]_v / * \mathcal{T}_{S1}[\mathit{gscn}(v, j, i)]_v^i \quad [S_{3a}]$$

$$\mathcal{T}_{S3}[\mathbf{Box}_j(X, s_5|s_6|s_{11})]_v^i = \mathcal{T}_{S2}[\mathit{gscn}(v, j, 0)]_v / / * \mathcal{T}_{S1}[\mathit{gscn}(v, j, i)]_v^i \quad [S_{3b}]$$

$$\mathcal{T}_{S3}[\mathbf{Va}(\mathbf{Txt}(n), \mathbf{Box}_j(X, s_7|s_8|s_{12}))]_v^i = \mathcal{T}_{S2}[\mathit{gscn}(v, j, 0)]_v / \mathbf{n} \mathcal{T}_{S1}[\mathit{gscn}(v, j, i)]_v^i \quad [S_{3c}]$$

$$\mathcal{T}_{S3}[\mathbf{Va}(\mathbf{Txt}(n), \mathbf{Box}_j(X, s_5|s_6|s_{11}))]_v^i = \mathcal{T}_{S2}[\mathit{gscn}(v, j, 0)]_v / \mathbf{n} \mathcal{T}_{S1}[\mathit{gscn}(v, j, i)]_v^i \quad [S_{3d}]$$

4.4.4 Exemple de traduction d'une règle de transformation

Nous illustrons ici le processus de traduction par l'exemple de règle VXT de la figure 4.21, qui extrait des informations sur l'auteur d'un article au format DocBook et génère des méta-données Dublin Core [69] exprimées au moyen de RDF. La règle VXT est décrite de la manière suivante :

$$TR = \langle v, r, l \rangle$$

avec

$$\begin{aligned} v &= \mathbf{Va}(\mathbf{Txt}(\mathit{articleinfo}), \mathbf{Box}_1(\mathbf{Va}(\mathbf{Txt}(\mathit{author}), \mathbf{Box}_0(\mathbf{Shp}_2(\mathit{star}, s_{12}), s_2)), s_8)) \\ r &= \mathbf{Va}(\mathbf{Txt}(\mathit{rdf:Description}), \mathbf{Box}_1(\mathbf{Va}(\mathbf{Txt}(\mathit{dc:Creator}), \mathbf{Box}_2(\mathbf{Shp}_3(\mathit{circle}, \mathit{blue}), \mathit{gray})), \mathit{gray})) \\ l &= \{(2, 3, \mathit{text})\} \end{aligned}$$

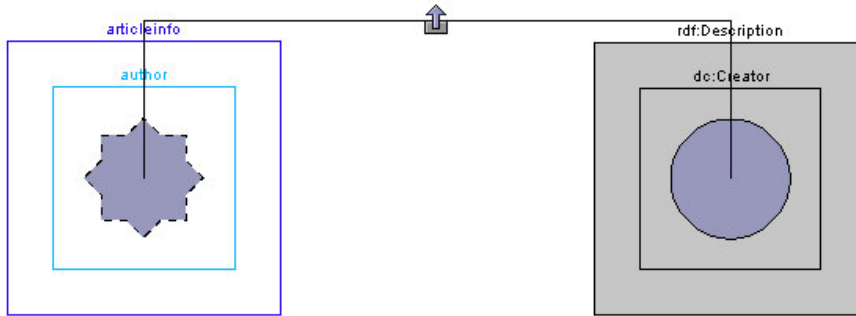


FIG. 4.21 : Exemple de règle VXT à traduire

Appels aux fonctions de traduction

$$\mathcal{T}[\langle v, r, l \rangle] = \langle \text{xsl:template match}=\text{"}\mathcal{T}_M[v]\text{"} \rangle \\ \mathcal{T}_R[r]_v^\Gamma \\ \langle \text{/xsl:template} \rangle$$

avec $\Gamma/\varphi_1 = l, \varphi_2 = \emptyset, \varphi_3 = \emptyset, \varphi_4 = \emptyset$

Traduction de l'expression XPath associée à l'attribut *match*

$$\begin{aligned} \mathcal{T}_M[v] &= \text{Pred}(\text{articleinfo}, \mathcal{T}_P[v_1]) / \mathcal{T}_M[\text{gscn}(v, 1, 0)] && (M_{11}) \\ &\text{avec } v_1 = \text{gscn}(v, 1, 0) = \mathbf{Va}(\mathbf{Txt}(\text{author}), \mathbf{Box}_0(\mathbf{Shp}_2(\mathbf{star}, s_{12}), s_2)) \\ &= \text{Pred}(\text{articleinfo}, \mathcal{T}_P[v_1]) / \mathcal{T}_P[v_1] && (M_{10}) \\ &= \text{Pred}(\text{articleinfo}, \mathcal{T}_P[v_1]) / \text{Pred}(\text{author}, \mathcal{T}_P[\mathbf{Shp}_2(\mathbf{star}, s_{12})]) && (P_{36}) \\ &= \text{Pred}(\text{articleinfo}, \mathcal{T}_P[v_1]) / \text{Pred}(\text{author}, \varepsilon) && (P_{30}) \\ &= \text{Pred}(\text{articleinfo}, \text{Pred}(\text{author}, \mathcal{T}_P[\mathbf{Shp}_2(\mathbf{star}, s_{12})])) / \text{Pred}(\text{author}, \varepsilon) && (P_{36}) \\ &= \text{Pred}(\text{articleinfo}, \text{Pred}(\text{author}, \varepsilon)) / \text{Pred}(\text{author}, \varepsilon) && (P_{30}) \\ &= \text{Pred}(\text{articleinfo}, \text{Pred}(\text{author}, \varepsilon)) / \text{author} && (\text{pred}) \\ &= \text{Pred}(\text{articleinfo}, \text{author}) / \text{author} && (\text{pred}) \\ &= \text{articleinfo}[\text{author}] / \text{author} && (\text{pred}) \end{aligned}$$

Traduction du corps de la règle

$$\begin{aligned} \mathcal{T}_R[r]_v^\Gamma &= \mathcal{T}_{R_1}[r]_v^\Gamma && (R_0) \\ &\text{avec } \Gamma/\varphi_1 = \{\langle 2, 3, \text{text} \rangle\}, \varphi_2 = \emptyset, \varphi_3 = \emptyset, \varphi_4 = \emptyset \\ \mathcal{T}_{R_1}[r]_v^\Gamma &= \langle \text{rdf:Description} \rangle && (R_6) \\ &\quad \mathcal{T}_{R_1}[\mathbf{Va}(\mathbf{Txt}(\text{dc:Creator}), \mathbf{Box}_2(\mathbf{Shp}_3(\mathbf{circle}, \mathbf{blue}), \mathbf{gray}))]_v^\Gamma \\ &\quad \langle \text{/rdf:Description} \rangle \\ &= \langle \text{rdf:Description} \rangle && (R_6) \\ &\quad \langle \text{dc:Creator} \rangle \\ &\quad \quad \mathcal{T}_{R_1}[\mathbf{Shp}_3(\mathbf{circle}, \mathbf{blue})]_v^{\Gamma \subset \varphi_1, \langle 2, 3, \text{text} \rangle} \\ &\quad \quad \langle \text{/dc:Creator} \rangle \\ &\quad \langle \text{/rdf:Description} \rangle \\ &= \langle \text{rdf:Description} \rangle && (R_{14}) \\ &\quad \langle \text{dc:Creator} \rangle \\ &\quad \quad \langle \text{xsl:value-of select}=\text{"}\mathcal{T}_S[v]\text{"} \rangle \\ &\quad \quad \langle \text{/dc:Creator} \rangle \\ &\quad \langle \text{/rdf:Description} \rangle \end{aligned}$$

Traduction de l'expression XPath associée à l'attribut *select*

Nous avons :

$$gp(\mathbf{Va}(\mathbf{Txt}(\text{articleinfo}), \mathbf{Box}_1(\mathbf{Va}(\mathbf{Txt}(\text{author}), \mathbf{Box}_0(\mathbf{Shp}_2(\mathbf{star}, s_{12}), s_2)), s_8)), 0, 2) = [0, 2]$$

donc :

$$\text{card}(gp(v, 0, 2)) \geq 2$$

Alors :

$$\begin{aligned}
 \mathcal{T}_S[v]^2 &= . \mathcal{T}_{S_1}[\text{gscn}(v, 0, 2)]_v^2 && (S_{0b}) \\
 &= . \mathcal{T}_{S_1}[\text{Shp}_2(\text{star}, s_{12})]_v^2 && (\text{gscn}) \\
 &= ./\text{node}() && (S_{1k})
 \end{aligned}$$

Résultat final

```

 $\mathcal{T}[\langle v, r, l \rangle] =$  <xsl:template match="articleinfo[author]/author">
  <rdf:Description>
    <dc:Creator>
      <xsl:value-of select="./node()" />
    </dc:Creator>
  </rdf:Description>
</xsl:template>

```

4.4.5 Propriétés de la fonction de traduction

Nous venons de définir formellement la syntaxe visuelle des règles de transformation VXT ainsi que la fonction de traduction générant des feuilles de transformation XSLT à partir de programmes VXT. Nous allons maintenant utiliser ces définitions pour établir deux propriétés de la fonction de traduction : sa validité (*correctness*) et sa complétude (*completeness*).

Validité des feuilles de transformation produites

Il n'est pas possible de prouver la validité¹⁷ des feuilles de transformation XSLT en général car celles-ci mélangent des instructions XSLT avec des fragments d'arbre résultat. Le W3C fournit un fragment de DTD non normatif pour les feuilles de transformation XSLT ([82], en particulier les annexes C et D). Il ne s'agit que d'un fragment de DTD, une DTD complète ne pouvant être obtenue que par l'association de fragments de la DTD des documents cibles (la feuille de transformation mêlant dans les règles instructions XSLT et fragments de résultat). On parle alors d'instanciation d'une DTD résultat spécifique propre à la feuille de transformation. Dans notre cas, ne connaissant *a priori* rien sur la classe des documents produits par la transformation, nous ne pouvons pas vérifier que la fonction de traduction génère des feuilles de transformation valides, faute d'avoir une DTD générale pour les feuilles de transformation XSLT. Nous pouvons par contre montrer que la fonction de traduction produit des feuilles de transformation XSLT bien formées au sens XML et que les expressions XPath associées aux attributs *match* et *select* des instructions de transformation sont syntaxiquement correctes.

Nous énonçons ici les propriétés qui doivent être établies sur les différentes fonctions composant la fonction de traduction \mathcal{T} . Le détail des démonstrations est donné en annexe E.

Propriété 1 (Correction syntaxique des expressions engendrées par \mathcal{T}_P)

$$\mathcal{T}_P[v] = x \implies x \in \mathcal{L}(eXQual)$$

Preuve par induction sur la structure. Voir Annexe E.2

¹⁷Au sens XML du terme, c'est-à-dire sa conformité à un schéma.

Propriété 2 (Correction syntaxique des expressions engendrées par \mathcal{T}_M)

$$\mathcal{T}_M[[v]] = x \implies x \in \mathcal{L}(XPath)$$

Preuve par induction sur la structure. Voir Annexe E.3

Propriété 3 (Correction syntaxique des expressions engendrées par \mathcal{T}_S)

$$\mathcal{T}_S[[v]] = x \implies x \in \mathcal{L}(XPath)$$

Preuve par induction sur la structure. Voir Annexe E.4

Propriété 4 (Validité des corps de règles engendrés par \mathcal{T}_R)

$$\mathcal{T}_R[[r]] = x \implies x \in \mathcal{XS}$$

Nous rappelons que \mathcal{XS} désigne l'ensemble des documents XML bien formés décrits par une instantiation du fragment de DTD relatif au langage XSLT (voir section 4.4.3). Preuve par induction sur la structure. Voir Annexe E.5

Complétude de la fonction de traduction

Nous venons de montrer que les différentes fonctions composant la fonction de traduction engendrent toujours des expressions XPath syntaxiquement correctes et que le contenu des règles `xsl:template` est bien formé au sens XML. Nous allons maintenant prouver la complétude de chaque fonction de transformation, c'est-à-dire que pour toute règle VXT bien formée (vérifiant les fonctions VFi), il existe une traduction en règle XSLT. Nous combinerons ensuite pour chaque fonction les deux propriétés établies et nous énoncerons la propriété générale sur la fonction complète \mathcal{T} .

Pour cela, nous devons prouver les quatre propriétés suivantes :

Propriété 5 (Complétude de \mathcal{T}_P)

$$\forall v \in \mathcal{L}(\mathcal{AS}_V), \exists x \mid (x = \mathcal{T}_P[[v]])$$

Voir Annexe E.7

Propriété 6 (Complétude de \mathcal{T}_M)

$$\forall v \in \mathcal{V} \wedge \vdash_D v, \exists x \mid (x = \mathcal{T}_M[[v]])$$

Voir Annexe E.9

Propriété 7 (Complétude de \mathcal{T}_S)

$$\forall v \in \mathcal{L}(\mathcal{AS}_V) \wedge \forall i \mid (i \geq 0 \wedge i \in I), \exists x \mid (x = \mathcal{T}_S[[v]]^i)$$

Nous rappelons que I est l'ensemble des index de la VPME (voir la Définition 5).

Voir Annexe E.10

Propriété 8 (Complétude de \mathcal{T}_R)

$$\forall r \in \mathcal{R}, \exists x \mid (x = \mathcal{T}_R[[r]])$$

Voir Annexe E.11

Nous pouvons alors exprimer pour chaque fonction de transformation une propriété combinant les propriétés de complétude et de correction syntaxique.

Propriété 9 (Par combinaison des propriétés 1 et 5)

$$\forall v \in \mathcal{L}(\mathcal{AS}_V), \exists x \mid (\mathcal{T}_P[v] = x \wedge x \in \mathcal{L}(eXQual))$$

Propriété 10 (Par combinaison des propriétés 2 et 6)

$$\forall v \in \mathcal{V}, \exists x \mid (\mathcal{T}_M[v] = x \wedge x \in \mathcal{L}(XPath))$$

Propriété 11 (Par combinaison des propriétés 3 et 7)

$$\forall v \in \mathcal{L}(\mathcal{AS}_V), \exists x \mid (\mathcal{T}_S[v] = x \wedge x \in \mathcal{L}(XPath))$$

Propriété 12 (Par combinaison des propriétés 4 et 8)

$$\forall r \in \mathcal{R}, \exists x \mid (\mathcal{T}_R[r] = x \wedge x \in \mathcal{XS})$$

Nous pouvons alors écrire pour la fonction générale \mathcal{T} , en examinant la définition 13 et en utilisant les propriétés ci-dessus :

$$\forall tr \in TR \mid (tr = \langle v, r, l \rangle \wedge v \in \mathcal{V} \wedge r \in \mathcal{R}), \exists xs \in \mathcal{XS} \mid (xs = \mathcal{T}[\langle v, r, l \rangle])$$

C'est-à-dire que pour toute règle de transformation VXT bien formée, il existe une traduction en règle XSLT bien formée au sens XML et correcte du point de vue de la syntaxe XPath.

4.5 Autres travaux

Par rapport aux langages de transformation textuels, VXT, langage spécialisé dans la transformation de documents XML, se positionne à un niveau d'abstraction assez élevé puisqu'au dessus de XSLT. Son expressivité est cependant proche de celle de ce dernier si l'on tient compte de la possibilité d'exprimer la plupart des prédicats et paramètres non capturés directement par le langage visuel sous forme textuelle (la spécification textuelle se fait dans une fenêtre résumant les propriétés de l'entité sélectionnée comme nous le verrons dans le chapitre 7). Il est ainsi relativement simple de générer des feuilles de transformation XSLT à partir de programmes VXT. La distance entre VXT et Circus est plus grande, ce dernier se trouvant à un niveau d'abstraction inférieur à celui de XSLT. Proposer un mode Circus dans VXT est cependant intéressant pour plusieurs raisons. Premièrement, VXT représente une interface visuelle conviviale pour l'utilisateur débutant non familier avec Circus qui lui permet de créer des transformations sans avoir à apprendre la syntaxe du langage et sans devoir spécifier ou choisir une technique de transformation. Le mode Circus est aussi un moyen pour l'utilisateur familier avec XSLT de faire la transition entre les deux langages, puisque VXT peut aussi être utilisé pour générer des transformations partielles, c'est-à-dire des squelettes de code que l'utilisateur finira de remplir manuellement.

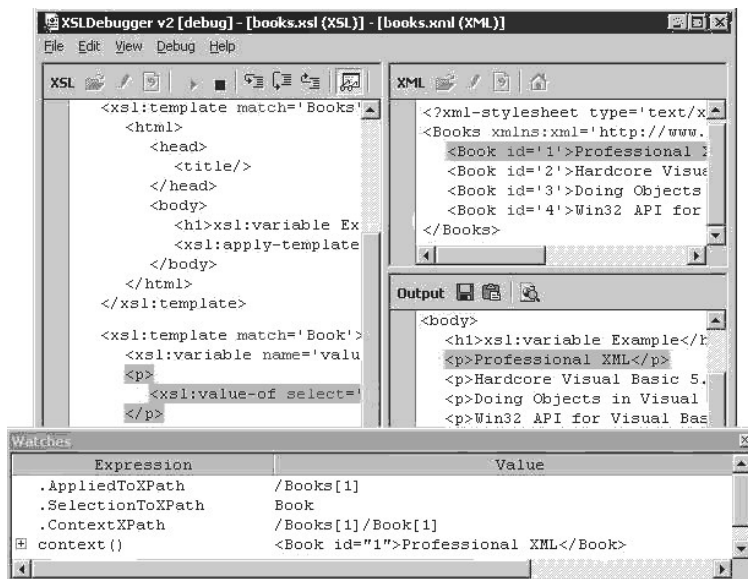


FIG. 4.22 : XSLDebugger : un environnement de développement intégré pour XSLT

Ce n'est cependant pas au niveau de l'expressivité que nous pensons pouvoir bénéficier des techniques de programmation visuelle, mais plutôt par rapport aux aspects cognitifs et à la facilité d'utilisation et de compréhension du langage. Il semble donc plus judicieux de comparer VXT aux autres approches visuelles pour la transformation de documents XML. Comme nous allons le voir, il existe un très grand nombre d'environnements visuels offrant une interface graphique au-dessus de XSLT mais dans lesquels la transformation est toujours exprimée directement au moyen de ce dernier. Ces environnements, commerciaux pour la plupart, sont intéressants mais assez éloignés de la solution que nous proposons et difficiles à comparer. Nous allons donc dans cette section surtout nous concentrer sur les autres langages de programmation visuels permettant la spécification de transformations de documents XML.

4.5.1 Environnements de développement intégrés

Nous avons vu dans l'introduction de ce chapitre que les environnements de développement intégrés pour XML sont équivalents à ce que nous avons nommé *VPE (Visual Programming Environments)* dans le chapitre 3. Ces outils sont donc à rapprocher de Visual C++ [160] ou JBuilder [29] pour Java qui proposent un environnement de développement graphique au-dessus d'un langage de programmation textuel avec des fonctionnalités évoluées pour la mise au point et la création de l'interface utilisateur du programme lui-même. Comme dans ces environnements, le programme est représenté et doit être spécifié au moyen du langage sous-jacent. Dans le cadre de la transformation de documents XML, il s'agit la plupart du temps de XSLT (certains outils comme Omnimark [206] proposent leur propre langage de transformation). Ainsi, même si l'environnement graphique améliore la lisibilité du code en colorant les mots-clés du langage, nous sommes avec ces outils encore loin des approches de programmation visuelle.

On retiendra cependant quelques éléments intéressants, présents dans un ou plusieurs des environnements principaux dont nous citons quelques exemples : XML Spy [4], eXcelon Stylus Studio [94], XSLDebugger [201] et Near & Far Designer [149].

- Un mécanisme qui, lors de la création d’une transformation XSLT, propose uniquement les éléments autorisés dans une liste de constructeurs en fonction de l’endroit où l’utilisateur veut insérer son nouvel élément dans la feuille de transformation. Ce mécanisme est à rapprocher de l’interaction contrainte proposée par VXT et qui sera détaillée dans le chapitre 7 traitant de l’environnement d’édition associé à notre langage.
- Une représentation graphique des structures source et cible, utilisant des arbres standard comme le *JTree* Java qui peuvent être développés, ou des formalismes visuels spécifiquement conçus pour la représentation de schémas de documents (DTD dans le cas de Near & Far, DTD et *XML Schema* dans le cas de eXcelon Stylus Studio). Ces représentations utilisent des diagrammes nœuds/arcs plus ou moins évolués. Comparés à notre langage visuel pour la représentation d’instances de documents et de DTD, ces formalismes semblent moins unifiés (la distance entre une instance et la classe de document semble plus grande). Ils semblent aussi plus complexes et difficiles à lire, ne reposant souvent que sur une simple translation en objets graphiques des constructions syntaxiques des versions textuelles, ce qui peut aussi poser des problèmes de résistance au facteur d’échelle, surtout si l’environnement ne propose pas de fonctions de navigation avancées (zoom et mécanismes de déplacement automatique).
- Des fonctionnalités de mise au point des transformations, comme les mécanismes d’exécution pas à pas avec identification de la règle sélectionnée pour le nœud courant dans l’arbre source et du fragment produit dans l’arbre résultat (figure 4.22). Nous verrons que l’environnement associé à VXT propose un mécanisme similaire pour l’évaluation progressive des transformations.
- Un mode de transformation dirigé par la cible dans lequel l’instance cible est représentée dans sa version formatée (quand il s’agit par exemple de documents XHTML) avec des références à des instructions d’extraction et de transformation qui peuvent être engendrées à partir d’actions de *drag & drop* de l’utilisateur.
- Des fonctions de génération de transformation à partir d’exemples d’association entre éléments source et cible fournis par l’utilisateur, à rapprocher des techniques de programmation par démonstration abordées dans l’introduction de ce chapitre. Les environnements de programmation par démonstration ne sont pas détaillés ici car ils sont très éloignés de VXT. Destinés avant tout à un public ayant des compétences très limitées en programmation, ils tendent à cacher les aspects programmatiques de la transformation et sont peu expressifs. Au contraire, VXT représente explicitement la transformation et est destiné à un public de programmeurs.

4.5.2 Xing

Xing [86, 87, 88] de M. Erwig, est présenté comme un langage visuel destiné aux non-programmeurs (*end-users*) pour l’expression de requêtes et de transformations sur des documents XML. L’outil propose une représentation des documents appelée *document metaphor* qui consiste à emboîter les éléments les uns dans les autres comme pour les *treemaps* mais où seuls les nœuds de type élément sont représentés graphiquement (voir figure 4.23 (a)). Ce formalisme fait un usage très limité du potentiel des représenta-

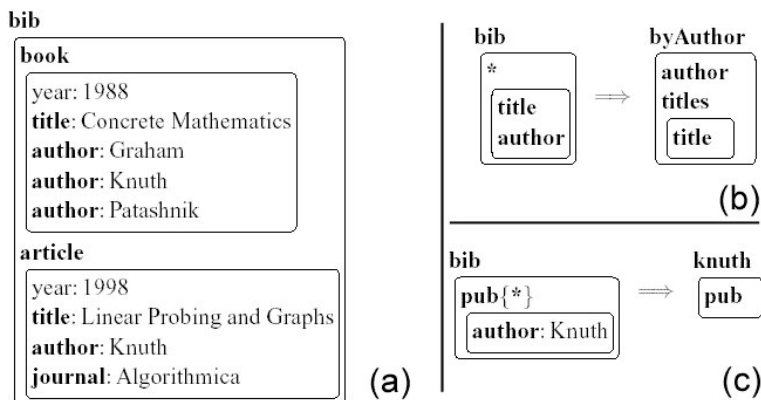


FIG. 4.23 : Xing

tions graphiques¹⁸ et ne permet pas de représenter les classes de documents comme les DTD, qui peuvent tout de même être utilisées, lors de la construction des requêtes, afin de ne proposer lors de l’insertion de nouveaux éléments que ceux autorisés par la DTD.

Les requêtes sont exprimées par des règles visuelles reprenant le formalisme précédent étendu par de nouvelles constructions textuelles et visuelles. La requête présentée dans la figure 4.23 (b) utilise par exemple le symbole *** dans la partie gauche pour exprimer le fait que le nom de l’élément est indifférent, de manière similaire à XPath. L’absence de liens entre partie gauche et partie droite implique l’utilisation de références (appelées ici *alias*) pour identifier dans la partie droite les données extraites de la partie gauche. Ainsi, comme le montre la figure 4.23 (c), il est parfois nécessaire d’utiliser des labels supplémentaires (*pub*) ce qui alourdit la représentation.

Une différence majeure par rapport aux règles de transformation VXT est que les *patterns* Xing représentant des conditions de sélection dans la partie gauche des règles sont par défaut copiés dans le résultat. Aussi, la première version de Xing ne permet pas d’exprimer des sous-*patterns* représentant des conditions de sélection mais qui ne doivent pas être transférés dans le résultat. Cette restriction importante implique une limitation sévère de l’expressivité et semble avoir été relevée dans un article à paraître [88]. Ce dernier fait mention d’une nouvelle construction qui permettra d’exprimer des sous-*patterns* qui ne doivent être considérés que comme des conditions de sélection. Leur représentation est cependant assez lourde du fait de la non-utilisation des couleurs, impliquant l’ajout d’une nouvelle notation.

Enfin, Xing propose des fonctionnalités pour la restructuration des données extraites du document source, à savoir la possibilité de grouper des éléments et un mécanisme d’agrégation. Aussi, le langage offre la possibilité d’utiliser les références basées sur les attributs XML *ID* et *IDREF* dans les requêtes, qu’il représente alors par des flèches reliant les nœuds impliqués. Les règles ne peuvent par contre pas être combinées (il n’est par exemple pas possible d’appeler une autre règle sur une partie des données

¹⁸L’environnement est par exemple restreint pour une raison inconnue à des représentations en noir et blanc.

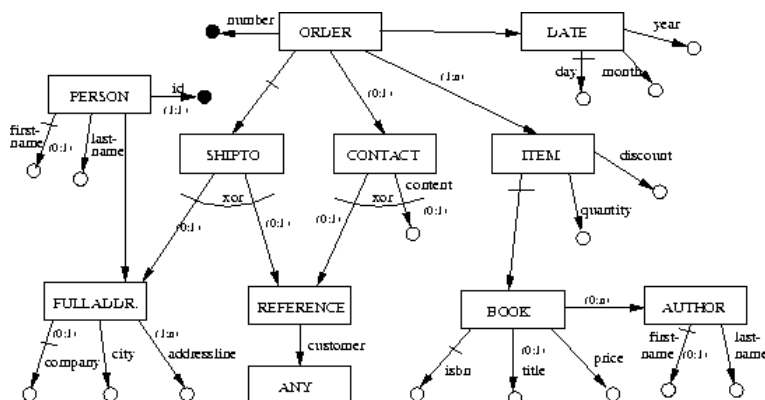


FIG. 4.24 : XML-GL : représentation XML-GDM d'une DTD

extraites avant de placer le résultat en partie droite). Xing est présenté comme un langage de *requête* pour XML, ce qui le rend proche de langages textuels comme XML Query, contrairement à VXT qui est présenté comme un langage de *transformation* pour XML, donc proche de XSLT. Les capacités de restructuration des langages de requête tendent cependant à rendre floue la frontière entre langage de requête et langage de transformation. Ainsi, la constatation faite dans le cadre de la comparaison entre XSLT et les langages de requête XML (section 2.4.1), reste vraie si l'on compare VXT à Xing ; c'est-à-dire que l'expressivité de VXT est plus grande que celle de Xing au niveau des opérations de transformation alors que c'est le contraire au niveau des opérations de sélection et d'extraction.

4.5.3 XML-GL

XML-GL [47, 59], développé par S. Comai, est un autre langage visuel de requête pour XML. Les requêtes sont formulées visuellement au moyen d'un formalisme de représentation basé sur les graphes, la syntaxe et la sémantique des requêtes étant définies par des structures et des opérations de graphes.

XML-GL se base sur XML-GDM (*XML Graphical Data Model*), un modèle de données visuel basé sur les diagrammes nœuds/arcs pour la représentation de la structure d'instances de documents XML et de DTD (figure 4.24). XML-GL est alors défini comme un langage de requête sur des données XML-GDM. Ainsi, comme dans VXT, l'unification du formalisme pour la représentation des instances, des DTD et des requêtes rend la création de ces dernières plus facile du fait de la ressemblance visuelle des différentes entités manipulées. Comparé au langage pour la visualisation de documents XML et de DTD proposé dans VXT, ce type de représentation semble un peu plus complexe à lire et plus sensible au facteur d'échelle. L'utilisation de diagrammes nœuds/arcs, plus courants que les *treemaps*, peut cependant rendre la représentation plus naturelle pour un certain nombre d'utilisateurs.

Comme Xing, XML-GL est plus un langage de requête qu'un langage de transformation. Il offre cependant une expressivité plus élevée. Les requêtes XML-GL sont visuellement représentées par deux graphes XML-GDM côte à côte séparés par une ligne verticale. La figure 4.25 donne un exemple de

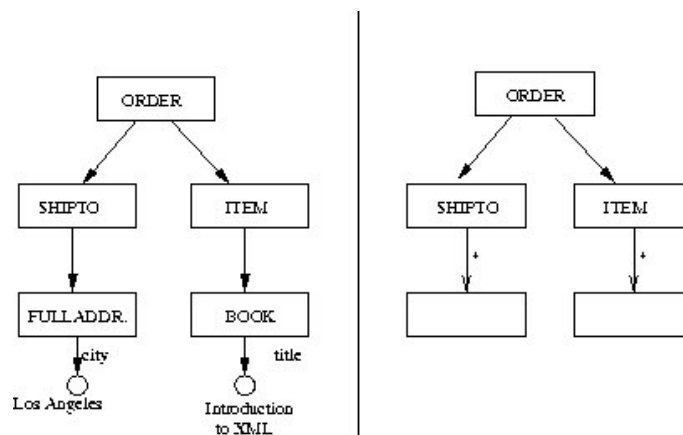


FIG. 4.25 : Requête XML-GL

requête qui sélectionne les éléments *order* contenant un livre dont le titre est «Introduction to XML» et qui doivent être envoyés à une adresse dans Los Angeles. La partie droite indique que pour chaque élément de type *order* vérifiant ces conditions, la requête produit un élément *order* avec seulement les informations précédentes. Le graphe de gauche représente les opérations de sélection et d'extraction, alors que le graphe de droite correspond aux opérations de construction du résultat (création de nouveaux éléments et insertion des données extraites). La liaison entre partie gauche et droite se fait aussi de manière similaire aux *aliases* de Xing par l'emploi du même label de chaque côté. Les cas d'ambiguïté sont résolus de manière différente, en liant graphiquement par un segment les nœuds des parties gauche et droite.

Comme précédemment, il n'est pas possible de faire appel à d'autres règles dans la partie droite, ce qui limite l'expressivité du point de vue de la transformation des données¹⁹. Par contre, XML-GL offre des possibilités de restructuration comme le tri, l'agrégation et le groupement, et gère aussi les attributs ID/IDREF. Comparé à VXT, XML-GL est plus puissant du point de vue de l'expression des conditions de sélection, mais il ne permet pas d'exprimer des transformations (restructurations) aussi complexes. XML-GL, comme Xing, semble donc plus adapté au traitement de données XML provenant de bases de données, alors que VXT est plus intéressant dans le cadre de documents XML exprimés dans des vocabulaires tels que DocBook, MathML, ou XHTML.

4.5.4 Approche basée sur les grammaires de graphes

Zhang et Deng proposent une approche visuelle pour la conception et la transformation de documents multimédia XML [238] basée sur un formalisme appartenant aux grammaires de graphes : les RGG (*Reserved Graph Grammar*) [237]. Ce formalisme permet d'exprimer une grammaire pour la structure XML et peut donc remplacer en partie les DTD (il semble cependant moins expressif). La validation se fait par l'application récursive d'un ensemble de règles de réécriture décrivant les contraintes sur la

¹⁹Une nouvelle version en cours de développement appelée XML-GL^{rec} semble proposer cette possibilité.

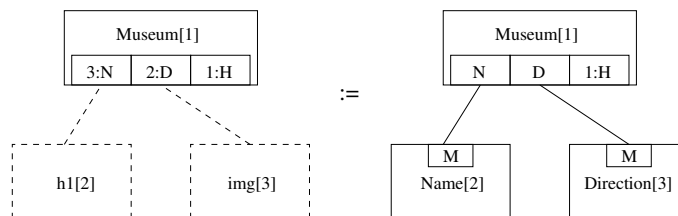


FIG. 4.26 : Exemple de règle de réécriture de graphe

structure (figure 4.26 en ne tenant pas compte des constructions pointillées). La partie droite représente un motif (*pattern*) exprimant des conditions de sélection de nœuds du document source ; la partie gauche représente la production de la règle (le sens de lecture est inversé par rapport aux autres langages). Les transformations sont quant à elles exprimées par une version étendue de ces mêmes règles de réécriture, qui appliquées sur la structure source, transforment celle-ci en un document résultat (figure 4.26 en tenant compte des constructions pointillées).

Cette approche est intéressante d'un point de vue théorique, car elle propose une méthode innovante et purement visuelle pour la validation et la transformation de structures XML. Elle semble par contre assez peu exploitable sur le plan pratique. L'utilisateur doit en effet manipuler des concepts assez peu intuitifs propres aux grammaires de graphes et plus particulièrement aux RGG, comme le mécanisme de marquage des nœuds (nombres préfixant les *vertices* dans les règles). D'autre part, un certain nombre d'opérations comme la création de nouveaux attributs dans le résultat ou le réarrangement de contenu textuel sont spécifiées par des *actions* directement en langage Java, ce qui rend la spécification de transformations réalistes extrêmement lourde. Enfin, si l'on considère la validation et la transformation de documents XML en général, les solutions standard à base d'analyseurs XML validants et de moteurs XSLT sont probablement bien plus performantes que l'approche décrite ici.

4.5.5 Environnement d'édition et fonctionnalités proposées

Dans les trois langages visuels décrits précédemment, il est assez peu fait mention de l'environnement d'édition et des fonctionnalités associées. Pourtant, dans le cadre des langages de programmation visuels, l'environnement d'édition est une partie importante de la solution, puisque c'est de lui que va dépendre par exemple la viscosité du langage. L'environnement peut aussi proposer des fonctionnalités aidant le programmeur dans sa tâche, comme des notations secondaires dynamiques, des fonctions de navigation adaptées ou des mécanismes guidant l'utilisateur dans la spécification des programmes afin de prévenir certains types d'erreur. C'est aussi dans cet environnement qu'a lieu la mise au point (*debugging*) et souvent l'exécution des programmes.

Comme nous le verrons dans la partie «Applications» de ce mémoire, nous avons conçu un environnement d'édition et d'exécution pour VXT. Cet environnement intègre des fonctionnalités qui aident l'utilisateur dans sa tâche au niveau de la spécification et de la mise au point des transformations. Par exemple, l'interaction est contrainte au niveau de l'édition des *VPMEs* afin d'empêcher l'utilisateur de créer des

expressions incorrectes. Nous verrons aussi qu'il existe un mécanisme pour l'évaluation progressive des règles de transformation, et que l'environnement propose des fonctions de navigation évoluées combinées à des choix de représentation qui rendent le langage moins sensible au facteur d'échelle, c'est-à-dire apte à travailler sur des documents et transformations de taille conséquente.

Ce type de fonctionnalités et les aspects cognitifs en général semblent assez peu développés dans les solutions précédentes. Un article sur Xing évoque l'utilisation des informations de la DTD pour guider l'utilisateur dans la création des parties gauches de règles (en ne proposant, par l'intermédiaire d'un menu déroulant, que les éléments autorisés en fonction du contexte). XML-GL fait aussi mention de l'utilisation de la DTD pour faciliter la création des requêtes, sans donner de détails concrets. Les environnements de développement intégrés fournissent quant à eux de nombreuses fonctionnalités puisque c'est ce qui représente leur principale valeur ajoutée par rapport à XSLT.

4.6 Conclusion et perspectives

Nous avons présenté dans ce chapitre la définition théorique de VXT, un langage de programmation visuel spécialisé dans la transformation de documents XML. Après une introduction au langage suivie d'un exemple complet de transformation, nous avons formellement défini sa syntaxe visuelle, ainsi que la fonction de traduction de VXT vers XSLT, sur laquelle nous avons établi les propriétés de complétude et de correction syntaxique des expressions engendrées. Nous avons ensuite essayé de positionner VXT par rapport aux solutions de transformation existantes, en nous concentrant sur les approches visuelles dédiées aux transformations XML. Nous avons vu que les programmes VXT peuvent être traduits en feuilles de transformation XSLT ou en code source Circus et qu'ils peuvent aussi être exécutés directement depuis l'environnement de développement associé à VXT (abordé dans le chapitre 7), principalement pour la mise au point des transformations.

4.6.1 Synthèse

Une évaluation partielle du langage a été effectuée dans le cadre de l'implémentation et du test de l'environnement de développement pour VXT, et par la création de transformations au moyen de ce prototype. Il ne s'agit cependant pas d'une évaluation objective, puisqu'elle ne résulte que de notre propre expérimentation et se trouve donc biaisée par notre bonne connaissance du langage ainsi que des métaphores et des fonctionnalités proposées pour l'édition et la mise au point. Cette évaluation nous a permis d'apprécier la puissance expressive de VXT, mais nous n'avons pas encore pu tirer de conclusion quant aux aspects cognitifs et fonctionnels liés plus spécifiquement à l'environnement. Ce sont en effet des points plus subjectifs qu'il est difficile d'évaluer en dehors d'un cadre d'expérimentation rigoureux. Nous reviendrons plus en détails sur ce point dans le chapitre 7.

Le langage semble offrir une expressivité permettant de spécifier des transformations relativement complexes, même si toutes les constructions programmatiques présentes dans XSLT ne sont pas proposées par VXT²⁰. Certaines constructions simples, comme la fonction de tri des éléments, pourront être

²⁰Circus étant situé à un niveau d'abstraction inférieur, et n'étant pas spécifiquement dédié à XML, il n'a pas été envisagé d'offrir un niveau d'expressivité équivalent à celui-ci au niveau de VXT.

ajoutées facilement puisqu'elles ne nécessiteront pas de modification significative du langage visuel lui-même. D'autres, comme les branchements conditionnels et les variables, vont par contre nécessiter une étude plus poussée et demanderont de revoir l'étude formelle du langage.

Nous avons déjà présenté une première solution pour la représentation des branchements conditionnels dans VXT [183]. Celle-ci est inspirée de la représentation des choix dans les DTD et utilise un empilement de rectangles verts pour délimiter les différentes options. Chacun de ces rectangles est séparé en deux zones. La zone de gauche contient l'expression booléenne exprimant le cas d'application. Il peut s'agir d'une expression textuelle représentant un test (par exemple pour vérifier si le contenu d'un nœud textuel contient la chaîne de caractères spécifiée) ou d'une *VPME* restreinte (dans le sens où elle ne peut utiliser qu'un sous-ensemble du langage des *VPMEs*). La zone de droite contient le fragment d'arbre produit par l'option. Cette solution n'a pas encore été implémentée car elle ne nous satisfait pas entièrement du point de vue de la lisibilité de la représentation et de la complexité des expressions²¹. Par exemple, dans le cas où le test est exprimé par une *VPME* restreinte, il est nécessaire de faire apparaître le nœud contextuel dans cette expression car il sert de référence pour l'interprétation des autres nœuds (sans lui, il est impossible de savoir si la *VPME* qui sert de test exprime des conditions sur le contexte ou sur le contenu de l'élément). D'une manière plus générale, cette solution pour les branchements conditionnels semble relativement complexe par rapport aux autres constructions du langage et soulève le problème de la représentation des structures de contrôle dans les langages de programmation visuels. Celles-ci ont tendance à être complexes (voir par exemple LabVIEW, section 3.3.2), alors que nous voudrions offrir une représentation simple, à la manière de nos boucles itératives, qui font abstraction des détails (variable d'itération, condition d'arrêt) liés à ces structures de contrôle, cachant ainsi une partie de la complexité sous-jacente. Cette simplification est rendue possible par le fait que le langage est spécialisé et n'a par conséquent pas besoin de proposer des structures de contrôle de bas niveau. Une approche similaire est envisageable pour les branchements conditionnels, mais elle présente une plus grande difficulté étant donné la diversité des constructions qui leur sont associées (différents types de tests et instructions).

Sur le plan visuel, il semble assez simple de comprendre les règles de transformation, même si l'interprétation des *VPMEs* complexes présente quelques subtilités (voir section «Sémantique des expressions» dans ce chapitre). Nous proposons pour cette raison un mécanisme d'évaluation progressive (voir chapitre 7) qui permet de tester rapidement une *VPME* sur les nœuds d'une instance de document source. Du point de vue de la spécification, nous verrons dans le même chapitre que l'utilisateur est assisté par l'environnement d'édition qui l'empêche de créer des expressions incorrectes au niveau syntaxique mais aussi dans certaines limites au niveau sémantique (dans une *VPME*, il est par exemple impossible de créer une instruction d'extraction à l'intérieur d'une condition d'absence d'élément).

4.6.2 Évolutions

En ce qui concerne l'évolution de VXT, il est dans un premier temps nécessaire de finaliser la représentation des branchements conditionnels, même si leur absence (temporaire) dans l'environnement de développement ne nuit pas de manière importante au langage du strict point de vue de l'expressivité, puisqu'il est souvent possible d'exprimer les branchements conditionnels de manière détournée, en

²¹Elle semble par contre fournir une expressivité proche de l'équivalent XSLT et serait de ce point de vue satisfaisante.

créant des règles de transformation plus spécifiques (ce qui contraint par contre l'utilisateur à adopter un certain style de programmation).

Dans un deuxième temps, il faudrait pouvoir proposer un moyen de représenter et d'utiliser des variables. La représentation visuelle de ce genre de construction abstraite est souvent un point épineux, et trouver une méthode adaptée demande une certaine réflexion afin de ne pas proposer un formalisme trop lourd ou une expressivité trop faible. Nous n'avons pas pour l'instant de solution satisfaisante, étant donné la diversité des formes que peut prendre le contenu qu'elles référencent. Une piste intéressante consiste peut-être à exploiter la spécialisation du langage pour proposer plusieurs représentations et moyens de référencement adaptés aux différents contenus et modes d'utilisation de ces variables.

Enfin, il faudrait prendre en compte les évolutions récentes des schémas de documents et proposer une extension du langage visuel pour la représentation d'instances de documents XML et de DTD. L'extension à des langages comme TREX, qui apportent assez peu d'expressivité par rapport aux DTD en ce qui concerne les contraintes structurales, ne devrait pas poser de gros problèmes. Il n'en va pas de même pour les *XML Schema* recommandés par le W3C qui proposent de nombreuses évolutions, comme le sous-typage, l'héritage et les types de données primitifs. Il serait possible d'utiliser comme solution temporaire les travaux récents autour de la gestion de schéma XML en Circus pour proposer une représentation approximative des contraintes structurales d'un *XML Schema* en passant par une conversion vers le format pivot exprimé en Circus. Mais il faudrait aussi à terme adapter l'outil aux nouvelles propositions du W3C, à savoir XPath 2.0 [18] et XSLT 2.0 [83]. XPath 2.0 utilise par exemple les types de données primitifs définis par *XML Schema*. Avoir une représentation visuelle de ces nouvelles contraintes dans le langage pour la visualisation de (classes de) documents nous permettrait d'enrichir les *VPMEs* tout en conservant un formalisme unifié. Les nouveaux concepts proposés par les *XML Schema* sont cependant très riches et très complexes, et il n'est pas évident de trouver un formalisme visuel capable de capturer cette complexité tout en conservant les propriétés que nous avons tenté de donner à notre proposition, c'est-à-dire une relative simplicité des constructions visuelles et un formalisme unifié pour la représentation des instances de documents, des classes de documents, et des règles de transformation.

Deuxième partie

Applications

Boîte à outils pour le développement d'interfaces graphiques

Le code associé aux interfaces utilisateur graphiques (*GUI : Graphical User Interface*) représente une part très importante des applications. Une étude [170] réalisée en 1992 montre qu'en moyenne 48% du volume de code et 50% du temps d'implémentation étaient alors dédiés au développement de l'interface. Ces proportions ont probablement encore augmenté, les interfaces devenant de plus en plus complexes du point de vue de la programmation afin de fournir une interaction de qualité toujours croissante. De nombreuses fonctionnalités liées à l'interface sont désormais considérées comme naturelles et indispensables : fonctions de copier/coller, barres de progression reflétant l'état d'avancement d'une tâche, raccourcis clavier et d'une manière générale fonctions d'accessibilité, constructions facilitant l'internationalisation et la localisation. Elles doivent donc être proposées par l'environnement. Sur un plan plus subjectif, l'apparence visuelle (le *look and feel*) de l'interface est aussi important. Ainsi, des critères purement esthétiques, comme le soin apporté au dessin des icônes, ayant *a priori* peu à voir avec les aspects cognitifs (utilisabilité) de l'interface graphique, entrent en ligne de compte et modifient la perception qu'a l'utilisateur du logiciel quant à sa qualité globale. Il faut tout de même faire attention à ne pas se baser sur des critères uniquement esthétiques, ce qui peut amener à la création d'interfaces parfois confuses (voir par exemple l'interface du logiciel de création de paysages tridimensionnels Bryce [61]).

La complexité inhérente du code lié à la gestion de l'interface utilisateur, due aux différents aspects à traiter (gestion de l'affichage, gestion des événements utilisateurs, liens avec les autres parties du programme) et au nombre de composants ne cesse d'augmenter. On assiste cependant depuis plusieurs années à une normalisation des interfaces graphiques autour du modèle WIMP (*Windows, Icons, Menus, and Pointers*) initié par l'Alto puis le Xerox Star et utilisé dans la plupart des environnements graphiques actuels : Microsoft Windows, Mac OS, Motif, et les environnements tels que KDE pour Linux. Cette normalisation permet de proposer des systèmes de gestion d'interfaces utilisateurs (*UIMS : User Interface Management System*) aidant à la conception de l'interface graphique des applications. Il existe plusieurs types de systèmes positionnés à différents niveaux d'abstraction, sachant qu'il devient difficile de les classer du fait qu'ils couvrent souvent différents niveaux :

- Les serveurs d'affichage et les systèmes de fenêtrage : positionnés au-dessus de la couche système d'exploitation, ils permettent la manipulation de fenêtres (création, déplacement, etc.) dans l'environnement graphique, offrant ainsi la possibilité de séparer l'espace d'affichage en différentes zones indépendantes. Ils offrent aussi des primitives de dessin de bas niveau pour dessiner le contenu des fenêtres (rectangle, cercle, polygone, texte) et gère les événements de la souris et du clavier (touche du clavier ou de la souris pressée, déplacement de la souris, etc...). Dans le cas de X-Windows, le système de fenêtrage est une application indépendante et peut être changé à tout instant. Même si son cas est un peu spécial¹, l'API Java 2D [162] se trouve à ce niveau d'abstraction.
- Les boîtes à outils (*toolkits*) : elles offrent un ensemble de composants graphiques (*widgets*) comme les menus, barres de défilement et champs textuels ainsi qu'une API pour les manipuler. Cette API inclue des événements de plus haut niveau que dans le cas précédent, comme par exemple *press* et *release* dans le cas d'un bouton de commande ou la sélection d'un item dans un menu. Ces systèmes, fournissant seulement une interface programmatique, ne sont utilisables

¹Faisant partie de la machine virtuelle Java (multi-plateformes), cette API offre un grand nombre de primitives graphiques de bas niveau utilisables à l'intérieur de composants Java spécifiques mais ne permet pas de gérer les fenêtres qui restent à la charge du système de fenêtrage natif.

que par des programmeurs. Dans le monde Java, les deux principales boîtes à outils sont AWT et Swing [163].

- Les outils de haut niveau : ils sont en général basés sur les boîtes à outils et permettent la création d'interface graphique par manipulation directe (donc visuelle) des composants tels que les boutons et les menus. Le code source équivalent est automatiquement généré par l'outil (dans certains cas, l'outil ne génère qu'un squelette de code à compléter). Exemple : les environnements de développement intégré tels que JBuilder [29] ou Forte pour Java [161].

Il existe un grand nombre de boîtes à outils pour la construction d'interfaces purement 2D ou 3D. Dans le cadre de la visualisation de grandes quantités d'objets, les interfaces 3D présentent des avantages par rapport aux interfaces 2D (compacité de la représentation, perception du contexte) comme le montrent les mesures effectuées pour les *Virtual Images* [222]. Les représentations tridimensionnelles ne sont cependant pas adaptées à toutes les tâches ni à toutes les structures de données². Le nombre de degrés de liberté plus élevé peut être encombrant pour l'utilisateur ; la navigation est plus complexe et il peut être plus difficile de s'orienter. Il semble alors intéressant d'explorer une voie intermédiaire entre l'interface purement 2D et l'interface 3D : l'interface 2.5D (*ZUI : Zoomable User Interface*). Celle-ci offre un environnement 2D combiné à une fonction de zoom continu permettant un contrôle plus fin et plus rapide de l'altitude d'observation comparé aux possibilités limitées de zoom discret de certains environnements 2D. Comme nous l'avons vu dans le chapitre 3, les interfaces zoomables sont une alternative aux techniques de focus+contexte comme les vues hyperboliques et les vues multiples (radar). Elles offrent des mécanismes adaptés à la représentation de grandes quantités d'information. Nous pensons aussi qu'elles rendent possible ou tout du moins plus aisée l'utilisation de certaines techniques de visualisation comme les *treemaps* [130] qui représentent les structures d'arbres au moyen de formes emboîtées et nécessitent donc des fonctions de zoom pour explorer les branches profondes de la structure. Une variante de cette technique est utilisée pour la représentation des structures dans VXT (chapitre 4).

Les interfaces 2.5D ont fait leur apparition dans les années 90 et il existe encore peu de boîtes à outils dédiées. On peut citer Pad++ [15, 16], Jazz [17] (une évolution de Pad++ en Java) et Zomit [186]. Dans ce chapitre, nous présentons XVTM (*Xerox Visual Transformation Machine*), une boîte à outils expérimentale que nous avons développée dans le cadre de ce travail de thèse. XVTM est implémentée en Java et permet notamment la conception d'interfaces zoomables. Son but est de faciliter la conception de l'interface d'environnements de visualisation/édition dans lesquels l'utilisateur doit manipuler et animer de grandes quantités d'objets aux formes géométriques pouvant être complexes. Les fonctionnalités de la XVTM reposent sur les principes étudiés dans le chapitre 3 «Langages visuels : État de l'art», dans le but d'aider les programmeurs à créer plus facilement des interfaces visuelles de bonne qualité.

5.1 Origines

La XVTM est l'évolution d'un premier prototype implémenté au centre de recherche XRCE appelé VAM *Visual Abstract Machine* [223, 224, 119]. La VAM a été développée pour expérimenter différentes

²Certaines structures sont mieux représentées dans un plan, auquel cas la troisième dimension spatiale n'apporte rien du strict point de vue de la représentation statique.

idées de recherches : 1^o les interfaces zoomables, 2^o un modèle d'objets graphiques permettant de modéliser un grand nombre de formes géométriques, 3^o l'expression de contraintes portant sur l'interaction de l'utilisateur au moyen de grammaires ainsi qu'une interface programmatique basée sur un jeu de codes opératoires (instructions graphiques) combiné à une pile de données permettant de gérer l'espace de représentation.

Bien qu'intéressants d'un point de vue scientifique, certains concepts de la VAM ont semblé trop restrictifs dans le cadre de la conception d'une boîte à outils destinée à un grand nombre de programmeurs et devant répondre à une large gamme de besoins. Par exemple, la spécification des actions autorisées au moyen d'une grammaire (interaction contrainte) n'est pas forcément évidente (même pour un programmeur) ni appropriée et peut rapidement devenir lourde. Sur ce point précis, il serait d'ailleurs intéressant de voir si la grammaire d'interaction pourrait être inférée à partir d'une démonstration par le programmeur des actions autorisées. Sur un autre plan, le modèle d'objets graphiques est riche mais ne permet pas de représenter n'importe quelle forme géométrique et demande à être enrichi.

La XVTM reprend plusieurs idées de la VAM mais propose aussi des alternatives plus conventionnelles, par exemple pour l'interface programmatique et la gestion de l'interaction. Elle propose aussi un modèle d'objets étendu, moins pur que celui de la VAM mais plus expressif. Ces différents changements devraient rendre la boîte à outils plus fonctionnelle car plus générale, plus facile à utiliser et capable de résoudre un plus grand nombre de problèmes.

5.2 Principes

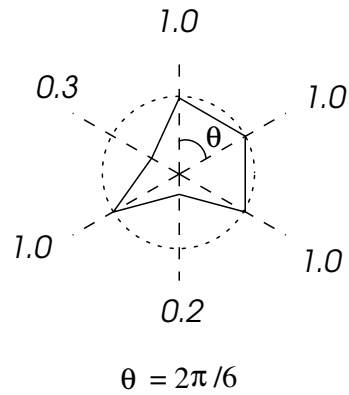
Comme nous l'avons dit précédemment, les fonctionnalités offertes par la XVTM sont censées aider le programmeur à créer plus facilement des interfaces graphiques se basant sur les principes étudiés dans les travaux de recherche récents sur la visualisation d'information et les langages de programmation visuels. Cette section présente les principales fonctionnalités de la XVTM et les changements par rapport à la VAM en justifiant les choix faits par rapport aux principes précédents.

5.2.1 Modèle d'objets graphiques

Le modèle d'objets graphiques de la XVTM est directement inspiré du système de types visuels [223] de la VAM, qui définit tous les objets graphiques (appelés glyphes), et ce quelle que soit leur forme, au moyen des mêmes attributs perceptuels. Le système de types visuels de la VAM est défini de la manière suivante :

| | | | | |
|--------------------|---|---|------------------------|-----------------------------------|
| <i>type pos</i> | = | $\langle x : num, y : num \rangle$ | $num \in \mathbb{R}$ | coordonnées du centre géométrique |
| <i>type size</i> | = | num | $num \in \mathbb{R}^+$ | rayon du cercle englobant |
| <i>type orient</i> | = | num | $num \in \mathbb{R}$ | orientation |
| <i>type shape</i> | = | $\langle offset : orient, xi : [num] \rangle$ | $num \in [0.0, 1.0]$ | forme normalisée |
| <i>type color</i> | = | $\langle chr : num, lum : num, sat : num \rangle$ | $num \in [0.0, 1.0]$ | couleur dans le système HSV |

Shape=(1, 1, 1, 0.2, 1, 0.3)



Star=(1, 0.2, 1, 0.2, 1, 0.2, 1, 0.2)

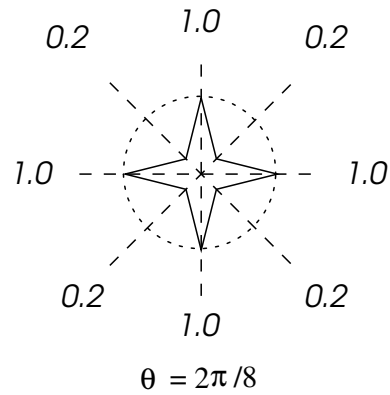


FIG. 5.1 : Modèle d'objets VAM

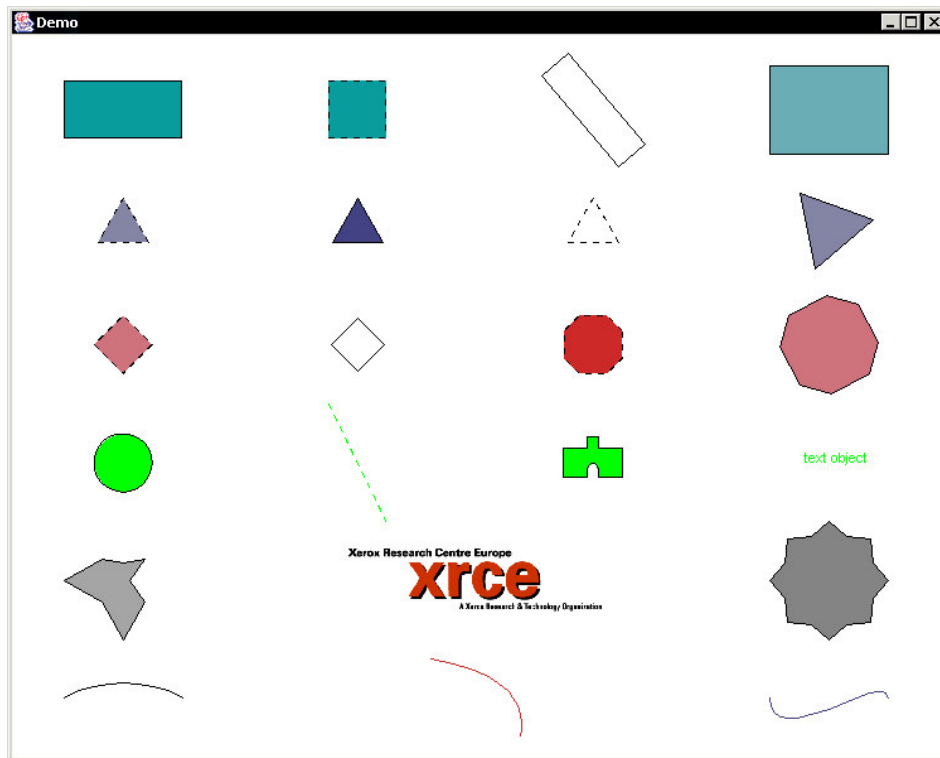


FIG. 5.2 : Exemples de glyphes du modèle d'objets XVTM



FIG. 5.3 : Glyphes translucides

La figure 5.1 contient deux objets définis suivant ce modèle. Chaque glyphe est défini par une variable de chaque type, c'est-à-dire : la position de son centre géométrique, le rayon du cercle englobant, l'orientation du glyphe par rapport à la verticale dans le sens trigonométrique, et une séquence de réels normalisés définissant la forme du glyphe indépendamment de sa taille. Les valeurs de cette séquence représentent la distance des sommets de la forme par rapport au centre géométrique, qui est aussi le centre du cercle englobant. Les n sommets d'une forme sont distribués de manière uniforme : l'angle entre deux sommets consécutifs est donc constant ($2\pi/n$).

L'orthogonalité des attributs perceptuels choisis et le polymorphisme induit par leur utilisation quelle que soit la forme à représenter simplifient la manipulation des glyphes : par exemple, pour déplacer un objet, quel que soit son type, il suffit de modifier les coordonnées de son centre géométrique. De même, pour modifier sa taille, il suffit de modifier le rayon du cercle englobant. Ce choix a été fait dans le but de proposer un modèle d'objets graphiques plus conviviale, c'est-à-dire plus intuitif et plus simple à manipuler par l'utilisateur (comparé aux modèles classiques, plus orientés vers la machine, mais qui ont pour cette raison l'avantage d'être plus performants). En conservant la même philosophie, la couleur est représentée dans le système HSV, dont les composantes (teinte, saturation et brillance) sont plus intuitives à manipuler que celles du système RGB, là encore très proche de la machine.

Ce premier modèle ne permet malheureusement pas de représenter n'importe quel type de forme géométrique. Il ne permet pas de définir les rectangles, les cercles, les ellipses, les courbes de béziers ainsi que d'autres formes couramment utilisées dans les interfaces graphiques. Nous en proposons donc une version enrichie dans la XVTM. Cette dernière, implémentée en Java et bénéficiant donc du modèle de programmation objet, définit une classe *Glyph* encapsulant les attributs précédents ainsi que les méthodes associées, et propose des sous-classes permettant de capturer les formes telles que les rectangles et les courbes de béziers en s'éloignant le moins possible du modèle de la VAM (figure 5.2). Des modifications sont tout de même nécessaires (l'attribut *shape* va disparaître dans certains cas), mais sont cachées au maximum par l'encapsulation, offrant ainsi au programmeur un ensemble de méthodes pour les opérations standard³ commun à tous les types de glyphes. La XVTM fournit aussi un attribut pour contrôler l'opacité (transparence) des glyphes (figure 5.3), qui peut être intéressant dans le cadre d'interfaces proposant plusieurs couches indépendantes (calques, *layers*) superposées dans la même fenêtre.

³Translation, réorientation, changement de taille et de couleur.

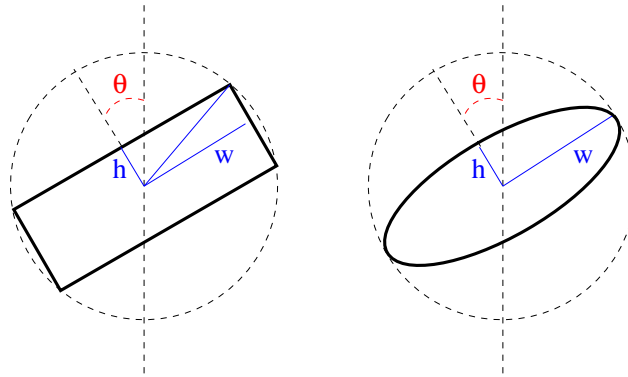


FIG. 5.4 : Formes rectangulaires

Le modèle d'objet original implémenté dans la VAM est repris dans la classe *VShape* (voir section 5.3). Les glyphes ne pouvant pas être capturés par ce modèle sont implémentés dans leurs propres classes. Ainsi, les rectangles (*VRectangle* et sous-classes) sont définis de la manière suivante (figure 5.4) : position du centre géométrique, orientation (θ), couleur, taille (qui correspond toujours au rayon du cercle englobant), et enfin le rapport hauteur/largeur (h/w). Les ellipses (*VEllipse* et sous-classes) sont basées sur le même modèle, le rayon du cercle englobant étant égal à $\max(w, h)$. Il est aussi possible de créer des glyphes à partir d'images bitmap, qui sont gérées de manière similaire aux rectangles. Les courbes de béziers s'éloignent plus du modèle. La figure 5.5 représente les modèles pour courbes quadratiques et cubiques. Le centre géométrique du glyphe est défini comme le centre du segment imaginaire reliant les deux extrémités de la courbe. Le ou les points de contrôle de la courbe sont alors exprimés en coordonnées polaires (ρ_2, θ_2) et (ρ_3, θ_3) dans le repère orthogonale dont l'axe x est lié au segment imaginaire précédent. Il est aussi possible de définir des *VPath*, constitués d'un ensemble de segments et courbes (quadratiques et cubiques) formant un chemin (*spline curve*).

Les variables visuelles du modèle d'objet correspondent aux dimensions perceptuelles de la sémiologie graphique, la texture exceptée (voir chapitre 3). Le choix des dimensions perceptuelles adéquates en fonction de la nature des données à représenter conditionne la qualité de la représentation. La correspondance directe entre les dimensions perceptuelles de la sémiologie graphique et le modèle d'objets offert ici permet de relier facilement les variables visuelles aux données représentées. Le programmeur peut ainsi se concentrer sur les problèmes de représentation puisque la transformation des glyphes dans le modèle géométrique Java 2D, hétérogène et moins intuitif, est prise en charge par la XVTM.

Solveur de contraintes

Déjà explorée dans la VAM, la possibilité d'exprimer des contraintes sur les différents attributs perceptuels des objets a été reprise dans la XVTM. Une contrainte est une relation mathématique entre un ensemble de variables. Elle est dite *satisfaite* quand ces variables ont des valeurs qui rendent vraie la relation. Un solveur de contraintes est un programme qui a pour tâche d'assurer qu'un ensemble de contraintes sont satisfaites. L'intégration d'un solveur de contraintes dans la XVTM permet donc d'ex-

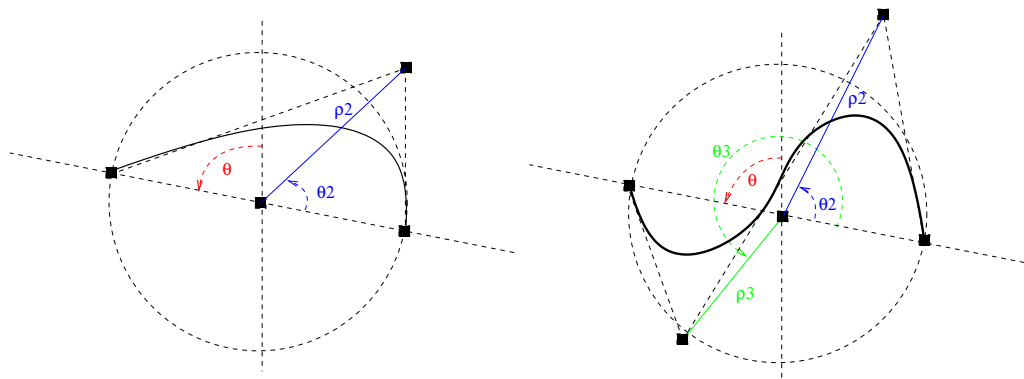
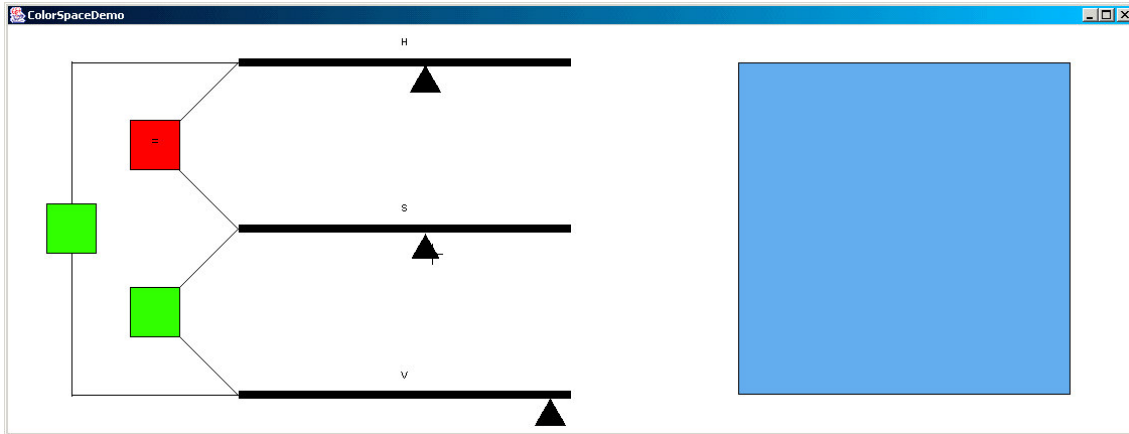


FIG. 5.5 : Courbe quadratique, courbe cubique

primer des relations entre les attributs perceptuels des glyphes et de maintenir automatiquement ces relations. Il est par exemple possible de contraindre la taille d'un glyphe par rapport à la position d'un autre ou encore la composante de brillance de la couleur d'un glyphe par rapport à la somme de la taille d'un autre glyphe et de son orientation. Les solveurs de contraintes permettent donc d'exprimer de manière directe des relations (mathématiques) entre variables et libèrent le programmeur du poids associé à la maintenance et à la propagation des valeurs de ces variables afin de satisfaire les contraintes, la complexité de ces opérations étant prise en charge par le solveur.

Il est à noter que le modèle d'objets proposé dans la XVTM, de par l'orthogonalité et la correspondance directe des variables visuelles avec les dimensions perceptuelles de la sémiologie graphique, rend la spécification de contraintes plus naturelle que dans le cas de modèles de plus bas niveau comme celui de Java 2D ; en effet les variables sur lesquelles le programmeur veut poser des contraintes sont en générale les dimensions perceptuelles. Dans ces modèles de bas niveau, proches de la machine (i.e. des instructions de dessin au plus bas niveau), les variables utilisées pour représenter les objets ne correspondent pas aux dimensions perceptuelles et varient en fonction de l'objet à représenter. Par exemple, les rectangles sont souvent codés par les coordonnées du coin supérieur gauche et la largeur/hauteur. Les cercles le sont par les coordonnées du centre géométrique et le rayon. Les formes plus générales, ainsi que les rectangles dont les côtés ne sont pas horizontaux et verticaux, sont simplement représentés par une séquence de coordonnées absolues ou relatives par rapport au point précédent. Cette distance par rapport aux dimensions perceptuelles oblige soit à créer des contraintes plus complexes soit à gérer manuellement des structures intermédiaires représentant les objets par des variables plus proches des grandeurs à contraindre (les contraintes sont alors exprimées sur ces variables qui propagent les mises à jour au modèle d'objet de bas niveau). Au-delà de la correspondance directe entre variables visuelles et grandeurs à contraindre offerte par la XVTM, le polymorphisme du modèle permet en plus d'exprimer des contraintes sur des glyphes de types différents de manière uniforme, puisque pour poser une contrainte d'égalité entre les positions de deux glyphes (par exemple), il suffira d'exprimer une relation sur la position des centres géométriques des deux glyphes.



$$\left\{ \begin{array}{ll}
 Hslider.pos.y & = Hruler.pos.y - 25 & (1) \\
 Hslider.pos.x & < Hruler.pos.x + Hruler.size & (2) \\
 Hslider.pos.x & > Hruler.pos.x - Hruler.size & (3) \\
 Sslider.pos.y & = Sruler.pos.y - 25 & (4) \\
 Sslider.pos.x & < Sruler.pos.x + Sruler.size & (5) \\
 Sslider.pos.x & > Sruler.pos.x - Sruler.size & (6) \\
 Vslider.pos.y & = Vruler.pos.y - 25 & (7) \\
 Vslider.pos.x & < Vruler.pos.x + Vruler.size & (8) \\
 Vslider.pos.x & > Vruler.pos.x - Vruler.size & (9) \\
 Selector.color.h & = \frac{Hslider.pos.x+400}{400} & (10) \\
 Selector.color.s & = \frac{Sslider.pos.x+400}{400} & (11) \\
 Selector.color.v & = \frac{Vslider.pos.x+400}{400} & (12) \\
 Hslider.pos.x & = Sslider.pos.x & (13) \\
 Sslider.pos.x & = Vslider.pos.x & (14) \\
 Hslider.pos.x & = Vslider.pos.x & (15)
 \end{array} \right.$$

La figure ci-dessus illustre l'utilisation de contraintes pour la création d'un sélecteur de couleur dans le système HSV (*Hue*, *Saturation*, *Value*). En plus des instructions de création des glyphes, il suffit d'ajouter les 12 premières contraintes du système d'équations pour coder le sélecteur de couleur. Le programmeur n'a pas à ajouter de code relatif à la mise à jour des différentes variables : une fois les contraintes activées, la XVTM propage automatiquement les changements dus aux manipulations des curseurs H, S et V par l'utilisateur aux variables du système d'équations de manière à rendre toujours vraies les relations de ce système. Les équations 1 à 9 contraignent la position des trois curseurs (*slider*) par rapport aux réglettes associées (*ruler*). Les équations 10 à 12 contraignent quant à elles respectivement les composantes teinte (*Hue*), saturation (*Saturation*), et brillance (*Value*) de la couleur de l'objet Selector (carré à droite) par rapport aux positions des curseurs sur les réglettes. Les contraintes peuvent être ajoutées et rétractées dynamiquement pendant l'exécution d'un programme. Les équations 13 à 15 sont des contraintes liant les positions horizontales des curseurs deux à deux. Leur activation/désactivation est commandée par les trois carrés à gauche.

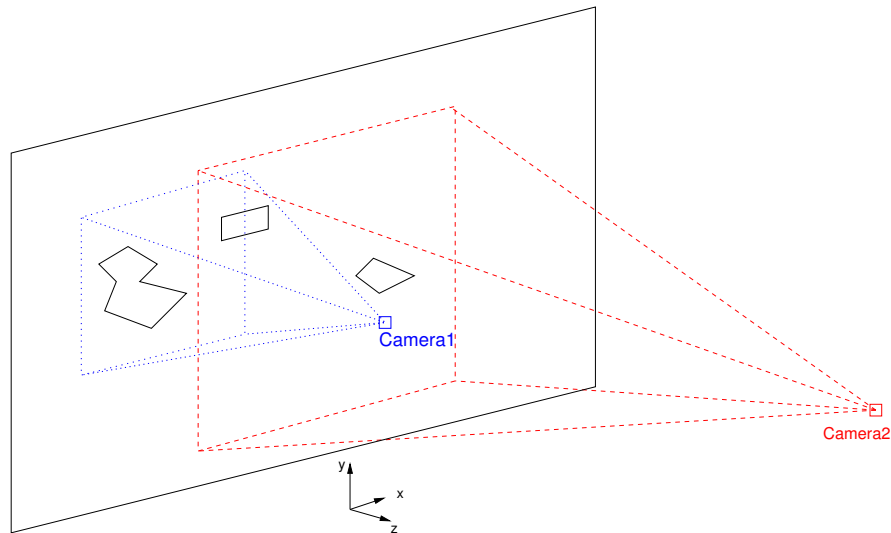


FIG. 5.6 : *Espaces virtuels et caméras*

Plusieurs outils pour interfaces 2D permettent l'utilisation de contraintes (Sketchpad [202], un des premiers systèmes, Thinglab [30] et Garnet [172]). Dans la famille des solveurs de contraintes graphiques⁴, il existe différents types de solveurs, permettant l'expression de graphes de contraintes non directionnelles, directionnelles, cycliques ou acycliques. Notre choix s'est porté sur Cassowary [10], un solveur de contraintes arithmétiques linéaires basé sur une version étendue de l'algorithme du simplexe autorisant les variables à prendre des valeurs négatives : toutes les variables visuelles, y compris la couleur, ayant une représentation numérique, la seule limitation vient de la linéarité des expressions mathématiques représentant les contraintes. Cette limitation a en partie été contournée par l'ajout de méthodes externes au solveur permettant l'expression de certaines contraintes non linéaires mais faisant perdre au graphe de contraintes sa propriété d'acyclisme. Le lecteur intéressé trouvera un état de l'art des solveurs de contraintes graphiques ainsi que des détails concernant l'intégration du solveur dans la VAM dans [177].

5.2.2 Interface zoomable

La XVTM est basée sur la métaphore d'espaces virtuels 2D infinis (*VirtualSpace*) contenant des glyphes et des caméras (figure 5.6). Une caméra est attachée à un espace virtuel et permet d'observer une région de cet espace. Elle peut être déplacée dans le plan (dimensions x et y dans la figure 5.6), mais aussi suivant son axe d'observation (z), ce qui correspond à modifier son altitude, c'est-à-dire à effectuer une opération de zoom avant ou arrière. L'axe d'observation reste par contre toujours perpendiculaire au plan de l'espace virtuel ; l'interface zoomable offre donc un degré de liberté supplémentaire par rapport aux interfaces 2D.

⁴Nous ne nous sommes intéressés qu'aux solveurs de contraintes graphiques car ce sont les seuls à proposer des temps de réponse faibles permettant un taux de rafraîchissement acceptable.

La différence avec une fenêtre typique d'interface 2D disposant de barres de défilement est assez subtile. Les capacités de zoom continu des interfaces 2.5D mises à part, la différence vient surtout de la représentation mentale que construit l'utilisateur de l'espace d'information et de la navigation dans cet espace. La fenêtre de visualisation associée à une caméra XVTM affiche une région de l'espace. La région observée est modifiée par les déplacements (translation, zoom) de la caméra. L'espace est de plus considéré comme infini⁵. Ces éléments contribuent à renforcer l'idée que les objets ont une position géographique constante dans l'espace, et incitent par conséquent l'utilisateur à faire usage (inconsciemment) de ses capacités cognitives relatives à l'orientation spatiale [204, 16], rendant ainsi l'orientation et la navigation dans l'espace plus aisée.

Zoom sémantique

La fonction de zoom standard modifie seulement la taille des glyphes par une projection géométrique tenant compte de l'altitude du point d'observation. Elle ne modifie donc pas leur aspect. Le zoom sémantique permet quant à lui de modifier automatiquement la représentation d'un objet graphique en fonction de sa taille apparente à l'écran, c'est-à-dire en fonction du niveau de zoom. Un exemple simple d'utilisation est l'horloge digitale implémentée avec Pad++ [16]. Cette horloge, au niveau de zoom le plus haut, n'affiche que les heures et les minutes. Si l'utilisateur effectue un zoom avant, l'horloge affiche aussi les secondes et éventuellement la date au lieu de se contenter de rendre le texte heures/minutes plus gros. Le zoom sémantique peut aussi servir à afficher un résumé ou les premières lignes d'un texte dans l'espace qui lui est alloué au lieu de présenter l'ensemble du texte avec une police illisible car de taille trop petite. Enfin, il peut être utilisé pour donner plus de détails ou au contraire diminuer la complexité des objets graphiques représentés à l'écran quand ces détails deviennent illisibles ou s'ils ne sont pas considérés importants compte tenu du contexte (niveau de zoom, lié à la notion de vue globale/vue détaillée). C'est le cas de l'outil de visualisation pour la base de données génétiques HuGeMap construit au moyen de Zomit [186], une boîte à outils pour interfaces zoomables orientée vers le zoom sémantique. Ce dernier a aussi un intérêt par rapport aux performances de l'interface graphique puisqu'une réduction de la complexité des objets à afficher peut entraîner une augmentation du taux de rafraîchissement et par conséquent une amélioration de la qualité de l'interaction utilisateur.

La XVTM supporte la notion de zoom sémantique mais ne fournit pas de glyphes offrant cette fonctionnalité, la définition des différentes apparences étant très dépendante de l'application cliente. La bibliothèque de glyphes est cependant extensible à volonté : pour ajouter un nouveau type de glyphe, il suffit de sous-classer la classe *Glyph* en définissant quelques méthodes abstraites. L'une de ces méthodes doit définir la projection du glyphe (calcul de sa taille apparente) en fonction de l'altitude d'observation de la caméra, c'est-à-dire du niveau de zoom. Cette méthode est appelée juste avant la méthode ayant pour fonction d'effectivement afficher le glyphe. Dans le cas standard, la première méthode se contente de calculer et de projeter la forme du glyphe en fonction de paramètres géométriques. Mais rien n'empêche, dans le cadre de la définition de nouveaux glyphes, de créer une méthode plus complexe qui, en fonction de la taille apparente calculée par la projection, sélectionnera une option parmi un choix d'apparences visuelles.

⁵En réalité, il est limité par le format de représentation sur 64 bits des entiers longs qui autorise des valeurs allant de -9 223 372 036 854 775 808 à +9 223 372 036 854 775 807.

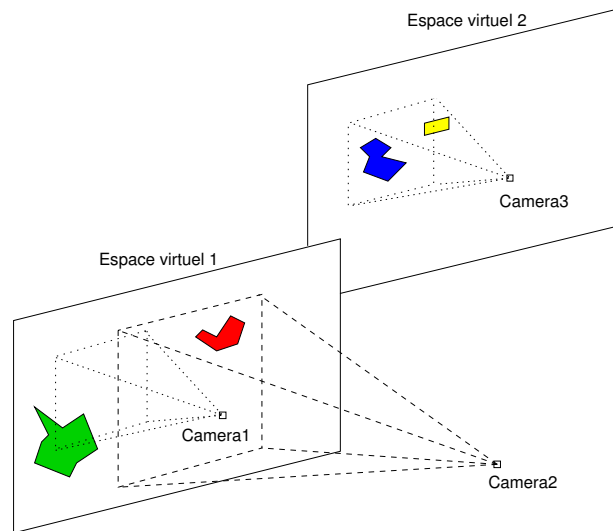


FIG. 5.7 : *Espaces virtuels multiples*

Espaces virtuels, vues et couches multiples

Il est possible d'associer plusieurs caméras indépendantes à un espace virtuel, ces caméras observant différentes régions de l'espace virtuel. Il est aussi possible de définir plusieurs espaces virtuels existant simultanément (figure 5.7). Cette idée d'espaces virtuels multiples était absente de la VAM. On la retrouve par contre dans *Jazz*, sous la forme de *scene graphs* indépendants. Elle est intéressante car elle offre une séparation logique de l'espace de travail, inexistante dans le cas d'un espace virtuel unique qui n'offre qu'une séparation géométrique des objets et groupes d'objets. Cette séparation logique est possible dans le cas où l'interface gère différents groupes d'objets graphiques qui n'auront jamais à interagir, c'est-à-dire qui n'ont pas besoin de se trouver dans le même espace géométrique. Elle offre une abstraction supplémentaire au programmeur, lui permettant une gestion plus fine des entités présentes dans chaque espace virtuel. D'un point de vue technique, elle permet aussi une gestion plus localisée des entités présentes dans les espaces virtuels et contribue à l'amélioration des performances de l'interface : par exemple, une caméra appartenant à un seul espace virtuel, la liste des glyphes à inspecter au cours de la phase de *clipping*⁶ est plus petite si les glyphes sont répartis dans différents espaces virtuels.

La XVTM offre la possibilité d'afficher plusieurs vues simultanément. Le concept de vue XVTM (*View*) est différent du concept de caméra. Une vue affiche la région observée par une ou plusieurs caméras. Dans le cas où plusieurs caméras sont associées à une même vue, les différentes régions observées sont affichées sur des couches (*layers*) superposées à fond transparent. Ces différentes couches sont chacune reliées à une caméra et sont donc indépendantes : il est possible de naviguer (translation et zoom) dans une couche sans modifier les autres. Il est possible de combiner dans la même vue des caméras appartenant à des espaces virtuels différents, ou bien au même espace virtuel. Cette dernière possibilité facilite la création des vues radars étudiées dans le chapitre 3 qui offrent dans une fenêtre de taille très

⁶Phase de détermination des glyphes effectivement visibles dans la région observée.

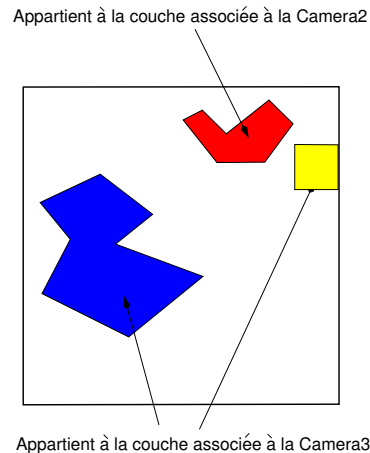


FIG. 5.8 : *Vue constituée de deux caméras*

réduite une vue d'ensemble de l'espace d'information tout en identifiant la région couramment observée par la vue principale. La figure 5.8 montre une vue constituée de deux couches associées aux caméras *Camera2* et *Camera3* de la figure 5.7.

Le concept de couches indépendantes est utilisé dans des applications comme les programmes de dessin et de retouche d'image (Adobe Photoshop [2]), mais aussi dans des travaux de recherche. Lieberman [147] propose un outil pour la visualisation d'information (par exemple des cartes géographiques) dans lequel la vue radar, au lieu d'être affichée dans une petite fenêtre, est représentée à l'arrière plan dans la fenêtre principale, la couche supérieure étant semi-transparente. VXT (chapitre 7) utilise des couches superposées pour représenter graphiquement la notion de filtrage structurel au moyen de masques visuels appartenant à la couche supérieure et adoptant la forme des structures qu'ils sélectionnent dans la couche inférieure.

5.2.3 Animations et continuité perceptuelle

Le chapitre 3 a mis en évidence l'intérêt de la continuité perceptuelle. Des changements brusques et inattendus au niveau de l'interface forcent l'utilisateur à fournir un effort mental important afin de relier l'ancienne représentation à la nouvelle, c'est-à-dire pour comprendre ce qui s'est passé. Au contraire, si le système propose une transition entre les états sous forme d'animation, une partie de la charge cognitive de l'utilisateur est transférée au système perceptuel (Robertson [112], Chang [49]). L'animation utilise la dimension temporelle pour fournir un lien entre l'ancien et le nouvel état du système. Elle donne des informations sur ce qui vient de se passer, ce qui est en train de se passer, et ce qui va se passer.

Les animations peuvent concerner toutes les variables du modèle d'objet. La XVTM fournit donc des primitives d'animation pour la position, la taille, l'orientation et la couleur des glyphes. Ces animations tiennent compte des contraintes pouvant être appliquées sur les glyphes par le solveur de contrainte. La continuité perceptuelle est aussi importante dans le cadre des déplacements du point d'observation,

surtout dans le cas des interfaces zoomables. Il est donc possible d'animer les déplacements (translation et zoom) des caméras.

L'animation aide l'utilisateur à comprendre ce qui se passe, mais elle augmente aussi dans une certaine mesure le plaisir d'utilisation de l'interface. Nous entrons ici dans des considérations esthétiques, donc plus arbitraires. Il est tout de même possible d'identifier certains principes influant sur la qualité et l'utilité de l'animation. Il semble tout d'abord assez évident que toute animation doit avoir un but. Une animation attire l'oeil ; c'est donc une manière de capter l'attention de l'utilisateur, au même titre qu'un objet aux couleurs vives se démarque d'un ensemble en tons de gris. Si un grand nombre d'éléments sont animés en même temps, cette propriété est en grande partie perdue, chaque animation se retrouvant noyée dans l'ensemble des animations. Des animations répétitives et sans véritable intérêt peuvent aussi être fatigantes. Chang et Ungar [49] détaillent certains principes influant sur la qualité des animations en s'inspirant des techniques venant des dessins animés. Le critère de réalisme des animations est important. Pour cela, il faut simuler certains principes physiques qui donneront une impression de solidité des objets à l'utilisateur. Le taux de rafraîchissement ne doit pas être trop inférieur à 25 images par seconde (fréquence correspondant à la perception visuelle humaine). La XVTM est conçue pour fournir un taux de rafraîchissement aussi proche de 25 Hz que possible. Elle utilise la technique du *double-buffering*⁷ pour fournir une animation sans scintillement, ainsi que des méthodes de *clipping* manuel. Afin de garantir un taux de rafraîchissement acceptable y compris pour des vues contenant un grand nombre de glyphes, et compte tenu de la nature de l'interface (zoom, espace virtuel infini), les fonctions de *clipping* automatique de Java 2D n'étaient pas assez performantes car arrivant trop tard dans la chaîne de rendu (*rendering*). Nous avons donc implémenté une méthode de *clipping* de plus bas niveau appliquée plus en amont dans le processus de rendu (voir section 5.3. Implémentation).

Le déroulement des animations suit souvent un schéma linéaire qui donne l'impression d'objets irréels. Dans le monde réel, les lois physiques font que les objets se déplaçant passent par une phase d'accélération avant d'atteindre une vitesse constante, puis passent par une phase de décélération avant de s'arrêter ; ils ne s'arrêtent pas abruptement, à moins de rencontrer un obstacle. La VAM offrait déjà une alternative au schéma d'animation linéaire en proposant un schéma dans lequel l'animation s'accélérait au cours du temps (exponentiel, *slow-in/fast-out*). La XVTM supporte aussi le schéma *slow-in/slow-out* [49], dans lequel le déroulement de l'animation s'accélère jusqu'à la moitié puis décélère (forme d'une sigmoïde). Ce schéma est intéressant car il consacre plus de temps au début et à la fin de l'animation qui sont les moments les plus importants du processus : la fonction principale de l'animation est d'établir un lien entre deux états, et les étapes se trouvant au milieu du déroulement sont moins importantes car éloignées de ces deux états. Ce schéma prépare l'utilisateur à ce qui va se passer, fournissant d'une certaine manière des transitions à l'intérieur de la transition que représente l'animation elle-même. Les différents schémas d'animation proposés par la XVTM sont présentés par la figure 5.9. Chaque schéma peut s'appliquer à n'importe quelle variable visuelle du modèle d'objet.

⁷Au lieu de peindre directement sur le canevas affiché à l'écran, les glyphes sont peints sur un canevas virtuel qui est ensuite transféré vers le canevas affiché à l'écran.

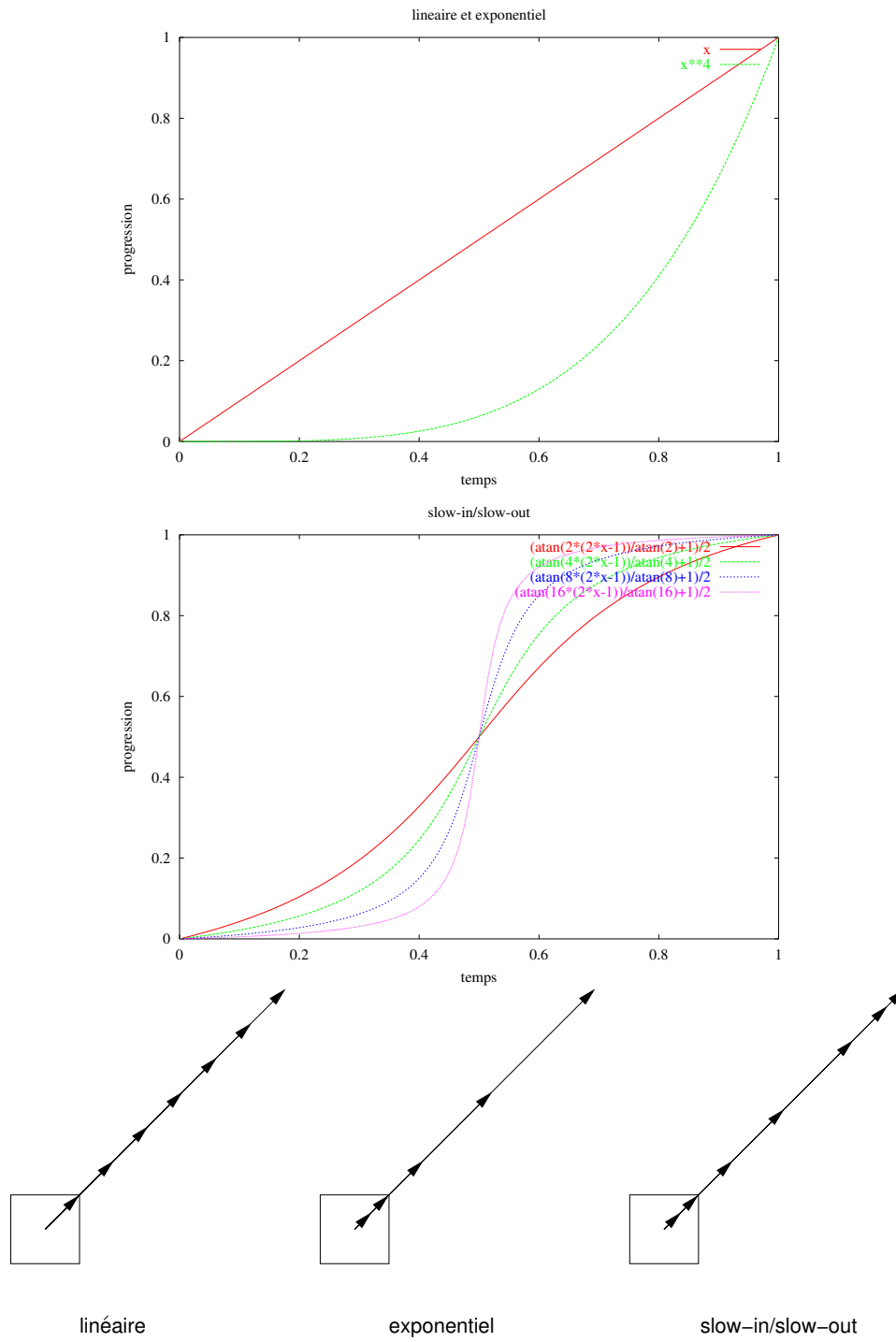


FIG. 5.9 : Schémas d'animation (progression en fonction du temps)

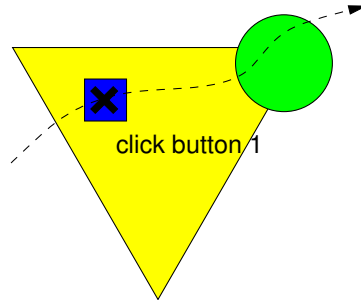
| Action utilisateur | Unité lexicale | Attributs |
|----------------------------------|------------------|---------------|
| Entrée dans un objet de type t | $+t$ | ID de l'objet |
| Sortie d'un objet de type t | $-t$ | ID de l'objet |
| Bouton souris n enfoncé | <i>press</i> | n |
| Bouton souris n relâché | <i>release</i> | n |
| Touche clavier k enfoncé | <i>keypress</i> | k |
| Touche clavier k relâchée | <i>keyrelase</i> | k |

FIG. 5.10 : Événements générés par la VAM

5.2.4 Interaction contrainte

L'idée de guider l'interaction utilisateur a été émise entre autre par P. Bottoni [32] et S. Levialdi [51]. Les actions de l'utilisateur devraient être contraintes afin de l'empêcher d'effectuer des actions illégales dans le sens où elles rendraient la spécification incorrecte (par ex. dans le cadre d'un langage visuel). Ces contraintes ne doivent cependant pas être trop rigides, afin que l'utilisateur ait assez de liberté et ne soit pas obligé de se conformer à des séquences d'actions qui interféreraient avec sa façon naturelle de spécifier le programme. J.-Y. Vion-Dury propose dans [223] un modèle d'interaction en remarquant que "la structure des actions de l'utilisateur est linéaire dans le temps [...] et [les actions] peuvent être considérées comme unidimensionnelles et ordonnées, donc traitables par les techniques classiques de l'analyse syntaxique". Ce modèle d'interaction est basé sur une grammaire associée à un analyseur traitant les actions de l'utilisateur. Il permet de fortement les contraindre, fournissant un retour visuel évident et suggérant des corrections lorsque l'utilisateur effectue des actions interdites par la grammaire. Ce modèle permet donc de contrôler l'interaction utilisateur en spécifiant de manière déclarative les actions autorisées. Ce contrôle est surtout intéressant lorsque l'utilisateur doit effectuer des actions précises dans un ordre bien défini. L'interaction contrainte peut aussi être utilisée dans un but didactique, guidant les actions de l'utilisateur novice mais laissant plus de liberté à l'utilisateur expert.

Un modèle d'interaction contrainte a été implémenté dans la VAM et utilisé pour la création d'un outil de monitoring et de configuration d'applications distribuées pour la plateforme *middleware* CLF [224]. Les actions utilisateur se traduisent par des événements générés par la VAM, résumés dans le tableau de la figure 5.10. Ces événements sont considérés comme des unités lexicales appelées lexèmes d'interaction. Ils sont agencés séquentiellement dans le temps et forment un flot envoyé par le moteur graphique de la VAM à un analyseur basé sur une grammaire d'interaction (voir exemple de la figure 5.11). Cette grammaire (de type LR(1), hors-contexte, non ambiguë, voir [3]) construite par le programmeur de l'application permet de spécifier les actions utilisateurs autorisées. Les éléments terminaux en sont les événements de la figure 5.10. En fonction de ses actions et de leur agencement temporel, l'analyseur envoie des instructions graphiques à la VAM et déclenche des opérations du côté de l'application cliente. Il peut par exemple à la suite de l'analyse de la séquence d'actions $+triangle, +square, press1$ envoyer des instructions pour changer la couleur de l'objet de type *triangle* (cette séquence d'actions doit bien entendu être autorisée par la grammaire). La correspondance entre séquences d'actions et instructions à déclencher en réponse à ces actions est spécifiée de manière déclarative.



+triangle +square press1 release1 -square +circle -triangle -circle

FIG. 5.11 : Exemple de lexèmes d'interaction générés lors d'un déplacement de souris

L'analyseur gère les erreurs (c'est-à-dire les actions non autorisées par la grammaire) en stoppant l'analyse et en passant dans un mode d'erreur qui stocke les actions non autorisées dans une pile FILO (*First-In/Last-Out*). Chaque événement de base est associé à un événement symétrique (+*t*/*-t*, *press n* / *release n*, *keypress/keyrelease*). En mode d'erreur, l'occurrence de l'événement associé à l'événement en haut de la pile a pour effet de supprimer ce dernier de la pile d'erreur. Il est donc possible de sortir du mode d'erreur en effectuant les actions "inverses" de celles ayant conduit au mode d'erreur⁸. Si l'on reprend l'exemple de la figure 5.11 et la grammaire ci-dessous :

```

All    → Tr | Ci;
Tr     → '+triangle' '-triangle' | '+triangle' Tr '-triangle' | '+triangle' Ci '-triangle';
Ci     → '+circle' '-circle' | '+circle' click2 '-circle';
click2 → 'press2' 'release2';

```

la séquence d'actions décrite dans la figure est interdite. Ainsi, l'analyseur passe en mode d'erreur lorsque le curseur entre dans le carré bleu. Si l'utilisateur effectue les actions *+triangle*, *+square*, *press1*, l'analyseur l'invite à effectuer les actions *release1*, *-square* dans cet ordre de manière à sortir du mode d'erreur.

Le modèle d'interaction de la VAM dirige très fortement l'interaction utilisateur et requiert de la grammaire qu'elle décrive toutes les séquences d'actions autorisées. Ainsi, si elle enlève au niveau de la programmation une partie de la complexité liée à la vérification des actions de l'utilisateur, elle oblige par contre le programmeur à construire une grammaire qui peut vite devenir complexe et difficile à modifier, surtout si la combinatoire des actions est importante. Suivant le type d'application, ces contraintes fortes sur l'interaction peuvent d'autre part présenter un intérêt minimal auquel cas la grammaire devient un poids pour le programmeur. L'approche est néanmoins très intéressante et pourrait être utilisée ponctuellement, par exemple dans les phases critiques de l'interaction requérant de l'utilisateur qu'il effectue des actions bien déterminées et ordonnancées, mais offrant à l'utilisateur une plus grande liberté le reste du

⁸Ces actions correctives sont suggérées à l'utilisateur par l'analyseur.

temps. Pour cela il est nécessaire de proposer un modèle mixte autorisant la spécification de l'interface par une grammaire (les événements étant envoyés à un analyseur) et des opérations associées à certaines séquences d'actions, ou en suivant un modèle plus standard déclenchant les opérations directement à partir des méthodes liées aux événements souris/clavier définies par le programmeur et appelées par *callback*.

Comme la VAM, la XVTM fournit les événements listés dans le tableau de la figure 5.10. Mais elle fournit ces événements directement à l'application cliente de la XVTM, celle-ci ayant à charge d'implémenter l'interface de gestion des événements définie par la XVTM et basée sur un mécanisme de *callbacks*. L'implémentation de cette interface côté application peut alors utiliser un analyseur similaire à celui de la VAM pour traiter les événements en fonction d'une grammaire d'interaction, ou bien les traiter de manière plus standard, associant directement les événements à des instructions. Elle offre par ailleurs d'autres événements de plus bas niveau liés à la gestion des fenêtres d'affichage et au déplacement de la souris de manière à rendre la boîte à outils vraiment fonctionnelle. La XVTM permet donc de combiner dans une même interface les deux modèles de gestion de l'interaction, autorisant ainsi le programmeur à faire usage de l'interaction contrainte ponctuellement.

5.3 Implémentation

Nous étudions maintenant certaines parties de l'implémentation sur lesquelles sont basées les fonctionnalités décrites précédemment. Nous avons choisi d'implémenter XVTM en Java pour des raisons de portabilité. Les performances de Java 2D se sont nettement améliorées avec la génération 1.3 des machines virtuelles, la génération 1.4 offrant encore de meilleurs résultats ainsi que des capacités d'accélération matérielle sur certaines plateformes comme Windows. La XVTM, basée sur les primitives offertes par l'API de Java 2D, fonctionne avec n'importe quelle JVM (*JVM : Java Virtual Machine*) version 1.3.0 ou ultérieure.

La figure 5.12 décrit l'architecture globale de la boîte à outils. L'application cliente de la XVTM envoie ses instructions par l'intermédiaire du *VirtualSpaceManager* dans lequel sont regroupées la plupart des fonctionnalités offertes par l'API pour la manipulation des espaces virtuels (*VirtualSpace*), des vues (*View*), des contraintes graphiques et des animations. Un *thread* d'exécution est assigné à chaque vue et est configuré pour fournir un taux de rafraîchissement proche de 25 images par seconde. Les vues sont automatiquement rafraîchies sans que le programmeur ait à intervenir par des appels explicites. Elles ne sont par contre rafraîchies que si cela est effectivement nécessaire, c'est-à-dire si le rendu graphique à l'instant $t+1$ est différent du rendu à l'instant t de manière à consommer le moins de temps processeur possible⁹. Le programmeur peut donc manipuler ses glyphes et caméras facilement sans avoir à se soucier des problèmes de rafraîchissement. Le module chargé de la gestion des animations est lui aussi exécuté dans son propre *thread* puisque les animations sont indépendantes des vues : les animations travaillent

⁹D'une manière générale, il n'est pas évident que cette technique soit la plus performante. La difficulté liée à la détection des changements entre deux images peut, suivant la technique utilisée, prendre plus de temps qu'un rafraîchissement périodique complet de l'écran n'impliquant aucun test pour la détection de changement. Nous avons évalué les deux méthodes dans la XVTM, et c'est la méthode de détection des changements qui s'est révélée la plus performante (nous avons essayé d'utiliser les tests les plus simples possibles).

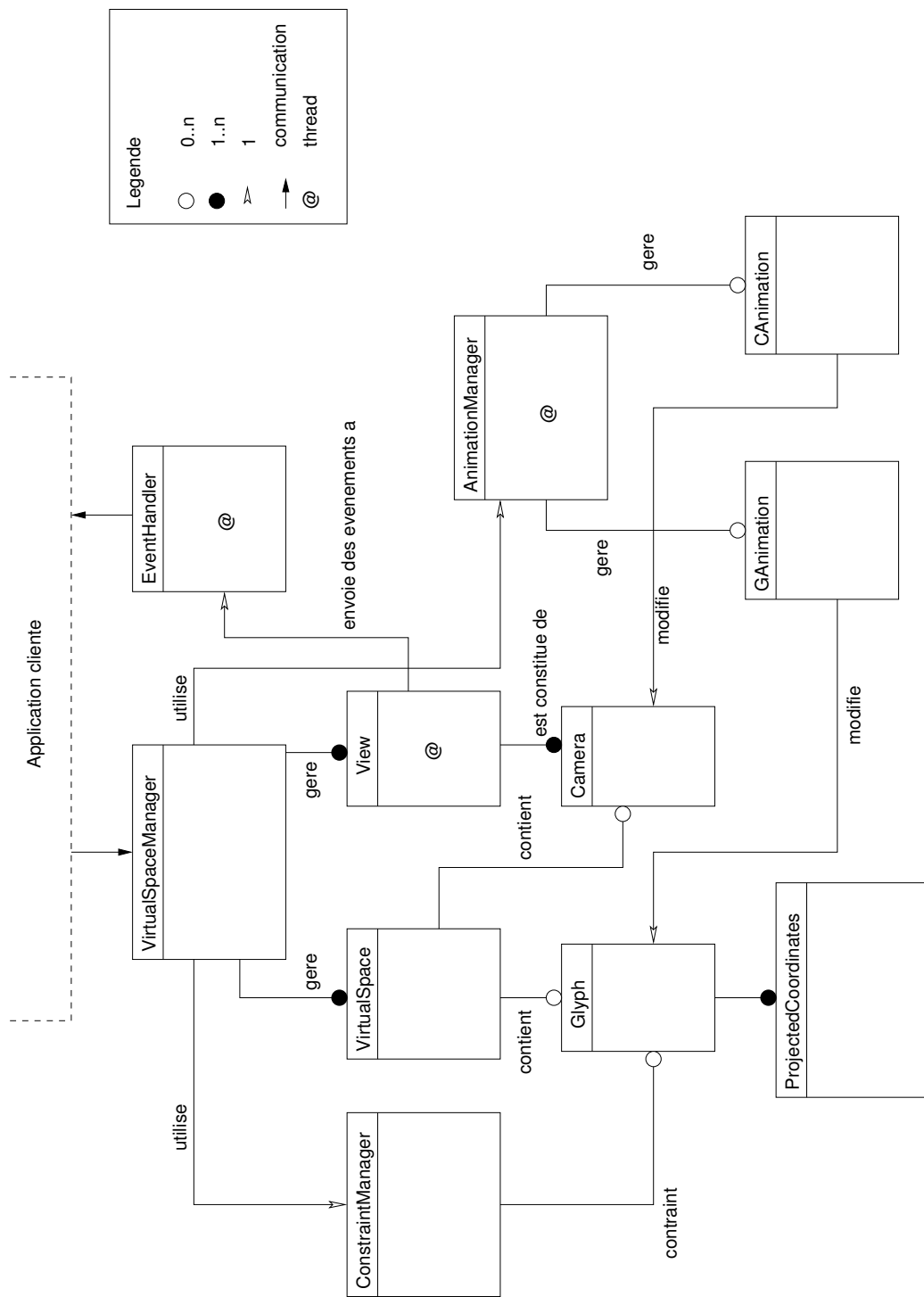


FIG. 5.12 : Architecture globale de la XVTM

directement sur les glyphes et les caméras dans l'espace virtuel et peuvent être observées dans différentes vues. Les glyphes pouvant apparaître dans plusieurs vues simultanément, il est nécessaire de définir pour chaque glyphe d'un espace virtuel donné autant de jeux de coordonnées projetées (*projected coordinates*) qu'il y a de caméras dans cet espace (ce qui correspond au nombre maximal de représentations différentes d'un même objet pouvant exister simultanément). Enfin, les événements de bas niveau déclenchés par les actions de l'utilisateur dans une vue (manipulation de la souris et du clavier) sont envoyés au module de gestion d'événements associé à la vue (*EventHandler*) qui les convertit en événements de plus haut niveau (entrée/sortie de la souris dans un glyphe, clic souris avec informations explicites par rapport aux modificateurs actifs) avant de les envoyer à l'application cliente. Ces différents mécanismes permettent à la XVTM d'offrir une *API* de haut niveau. Ainsi déchargé des problèmes techniques de bas niveau comme le rafraîchissement des vues et le maintien d'un bon niveau de performances, le programmeur peut se concentrer sur les aspects cognitifs de son interface graphique : qualité de la représentation et de l'interaction.

Manipulation des glyphes

Comme le montre le graphe d'héritage de la figure 5.13, toutes les opérations standard de manipulation des glyphes sont définies dans la classe abstraite *Glyph*. Afin d'améliorer les performances, le programmeur a le choix pour certains glyphes d'utilisation courante entre différentes classes définissant chacune des algorithmes de projection et de dessin optimisés en fonction des propriétés du glyphe. Par exemple, il est possible de représenter un rectangle en utilisant quatre classes différentes :

- *VRectangle* permet le dessin de rectangles ne pouvant pas être réorientés (côtés horizontaux et verticaux). Étant données ces propriétés, il est possible d'utiliser directement les primitives de dessin de rectangle de Java 2D sans passer par la génération d'une forme (*Shape*) plus coûteuse ;
- *VRectangleST* gère en plus un canal alpha pour l'information de transparence (l'effet de transparence demande un changement au niveau du contexte graphique avant le rendu puis une restauration de l'ancien contexte) ;
- *VRectangleOr* permet au glyphe d'avoir une orientation arbitraire ;
- *VRectangleOrST* ajoute le canal alpha au glyphe réorientable.

En fonction de ses besoins, le programmeur peut ainsi optimiser les performances de son application en choisissant les glyphes les plus performants offrant les fonctionnalités dont il a besoin. Le gain de temps obtenu pour un glyphe est négligeable mais le cumul des gains devient sensible dans le contexte de la visualisation de grandes quantités d'information.

Animations

L'interface pour la spécification d'animations est sensiblement la même pour les glyphes et les caméras. Le programmeur désigne l'objet (glyphe ou caméra) à animer, quelle dimension perceptuelle doit être animée, le schéma et la durée en millisecondes de l'animation, et en fonction de la dimension il fournit une structure de données contenant des informations sur l'animation elle-même (distance à parcourir dans le cas d'une translation, angle dans le cas d'une réorientation, etc.). Comme le montre la figure 5.14, chaque animation concerne une unique dimension perceptuelle d'un glyphe. Ce contrôle fin

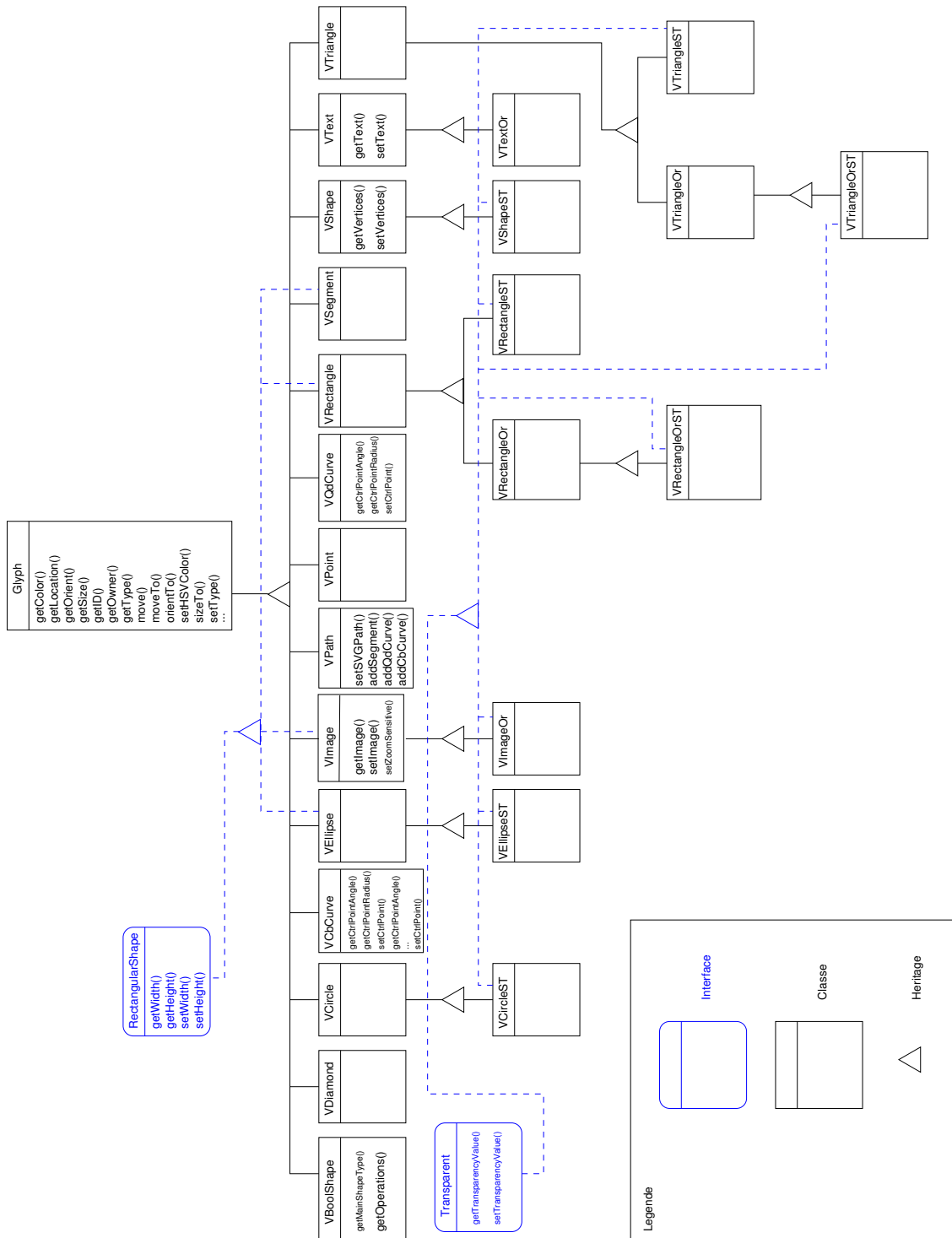


FIG. 5.13 : Classes représentant les glyphes

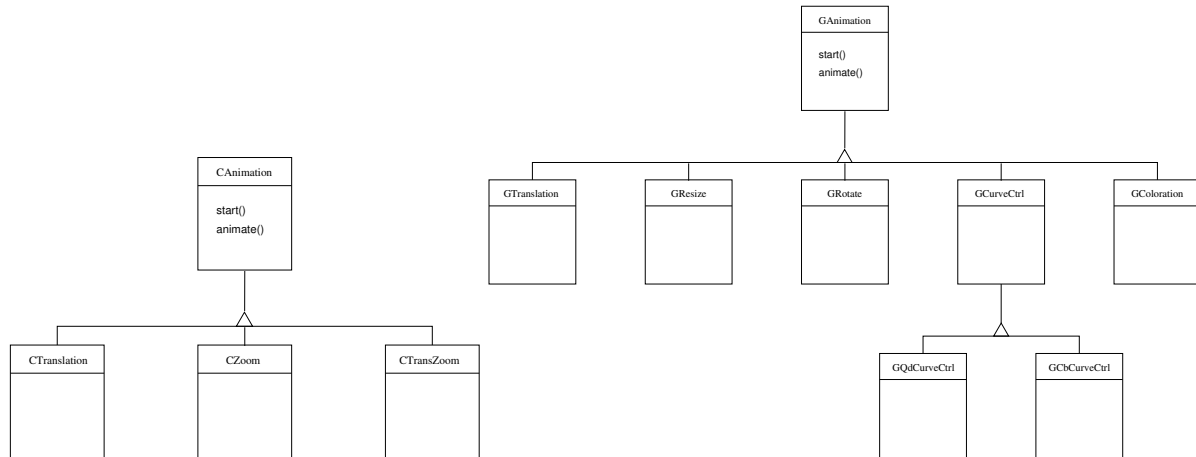


FIG. 5.14 : Classes représentant les animations de caméras et de glyphes

permet de déclencher simultanément (mais aussi avec un décalage) plusieurs animations concernant différentes dimensions perceptuelles d'un même glyphe. Un verrou est par contre posé sur les dimensions perceptuelles en cours d'animation, les nouvelles animations concernant cette dimension pour le même glyphe étant alors mises dans une file d'attente en attendant la fin de l'animation en cours.

Le temps d'exécution des animations est très important d'un point de vue cognitif et doit être respecté indépendamment des performances de l'ordinateur sur lequel s'exécute l'application. Le module de gestion des animations (*AnimationManager*) calcule donc au moment de la création d'une animation sa trajectoire avec une résolution assez fine, c'est-à-dire un nombre d'étapes intermédiaires assez important pour avoir une animation fluide à 25 images par seconde. Si les performances de l'ordinateur ne permettent pas d'afficher une nouvelle image toutes les 40 millisecondes, l'animation saute certaines étapes¹⁰ de manière à garantir que l'animation dure effectivement le temps spécifié par le programmeur.

Performances

Accélération matérielle. Les *JVMs* 1.4.0 et ultérieures offrent des possibilités d'accélération matérielle sur certaines plateformes (Windows) qui améliorent plusieurs primitives graphiques de base. Cette accélération n'étant pas supportée par toutes les *JVMs* et tous les systèmes d'exploitation, la *XVTM* fournit des vues accélérées ainsi que des vues standard qui peuvent être utilisées dans n'importe quel environnement. Toutes ces vues sont manipulées de la même manière et les différences de gestion des tampons image (*offscreen image buffers*) sont cachées au programmeur qui n'a qu'à choisir le type de vue qu'il désire utiliser.

¹⁰Pour cela il suffit de garder en mémoire l'instant t auquel l'animation a démarré et de comparer la durée écoulée à l'instant $t+i$ par rapport à la durée de l'animation pour en déduire la progression de l'animation et ainsi choisir l'étape intermédiaire correspondante calculée précédemment.

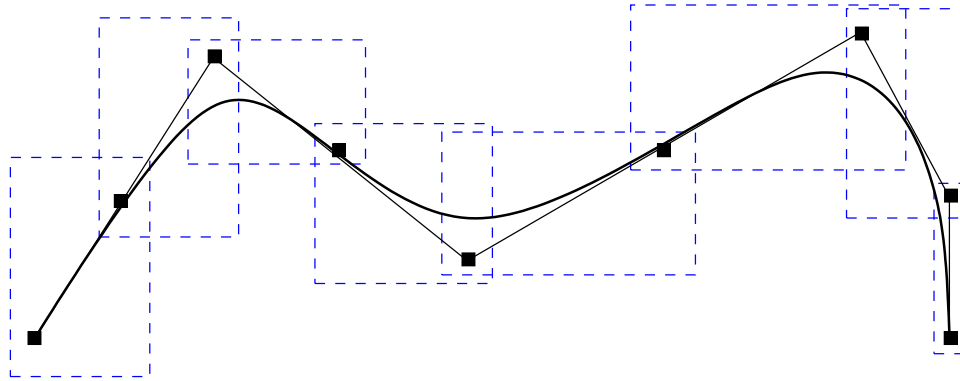


FIG. 5.15 : Méthode de clipping pour les VPath

Clipping. La XVTM implémente des méthodes de *clipping* manuel exécutées en amont du *clipping* Java 2D qui arrive assez tard dans le processus de rendu. Ces méthodes additionnelles permettent de déterminer au plus tôt (avant projection) les glyphes à traiter en fonction de la région observée par la caméra. Ainsi, les glyphes non visibles dans une vue ne seront pas projetés et il n'y aura pas d'appel à leur méthode de dessin (le *clipping* Java 2D est encore en aval de cet appel ; on supprime ainsi plusieurs opérations intermédiaires). Le taux de rafraîchissement d'une vue est donc fonction du nombre d'objets effectivement observés par les caméras qui la composent. Le mécanisme de *clipping* de la XVTM est simplement basé sur la boîte englobant le glyphe (*bounding box*) afin d'être très rapide. Le principe général est le suivant :

Si le centre géométrique se trouve dans la région observée **alors**
le glyphe doit être peint

Sinon

Si la boîte englobante est au moins partiellement visible dans la région observée **alors**
le glyphe doit être peint

Sinon

le glyphe est ignoré

Dans le cas des *VPath*, chemins (*spline curves*) constitués d'un ensemble de segments et courbes (cubiques ou quadratiques), l'algorithme a été adapté pour tenir compte de la spécificité de ces objets. L'enveloppe de *clipping* n'est plus la boîte englobant le chemin car elle n'est pas représentative de l'espace occupé par celui-ci. Afin d'en être plus proche, l'enveloppe est définie comme le contour extérieur de la forme composée des boîtes englobant chaque segment reliant un point de contrôle définissant la courbe au suivant (voir figure 5.15). Les dimensions de ces boîtes sont légèrement augmentées de manière à ce que la zone de recouvrement ne soit pas nulle à la jonction des boîtes.

La XVTM utilise aussi un algorithme inspectant la liste des glyphes dans l'ordre inverse de l'ordre de peinture. Sa tâche est de détecter les objets remplissant la vue, c'est-à-dire contenant complètement la région rectangulaire observée par la caméra. Si de tels glyphes existent, les glyphes suivant le premier

| Événement | Attributs |
|-----------------|--------------------------|
| press n | n, mod, x, y |
| release n | n, mod, x, y |
| click n | n, mod, x, y,clickNumber |
| mouseMoved | x,y |
| mouseDragged | mod, buttonNumber, x, y |
| enterGlyph | g |
| exitGlyph | g |
| Ktype | code, mod |
| Kpress | code, mod |
| Krelease | code, mod |
| viewActivated | |
| viewDeactivated | |
| viewIconified | |
| viewDeiconified | |
| viewClosing | |

avec n=bouton souris (1 à 3), mod=modificateurs (Ctrl/Shift),
(x,y)=coordonnées du curseur dans l'espace virtuel,
clickNumber=simple clic/double clic, g=référence sur le glyphe,
code=code hexadécimal associé au caractère.

FIG. 5.16 : Événements générés par les vues XVTM

glyphe dans la pile d'affichage et répondant à ce critère sont ignorés (ils ne sont ni projetés¹¹ ni peints). Cet algorithme travaille bien entendu sur la pile ne contenant que les éléments retenus par l'algorithme de *clipping* décrit précédemment.

Gestion des événements

Les événements Java fournis par l'AWT (souris, clavier, fenêtre) sont capturés au niveau de chaque vue et convertis en des événements de plus haut niveau, plus facilement utilisables par le programmeur qui n'a pas à gérer de masques, avant d'être envoyés à l'application cliente. Celle-ci a la charge d'assigner à chaque vue un gestionnaire d'événements qui doit implémenter les méthodes définies par *EventHandler*, à la manière des écouteurs (*Event Listener*) de l'AWT. Les événements fournis par un *EventHandler* sont listés dans le tableau 5.16. Pour générer les événements *enterGlyph/exitGlyph*, la XVTM maintient pour chaque vue une pile contenant les objets actuellement sous le curseur souris et déduit ces événements de la différence entre la pile à l'instant t et l'instant $t+1$.

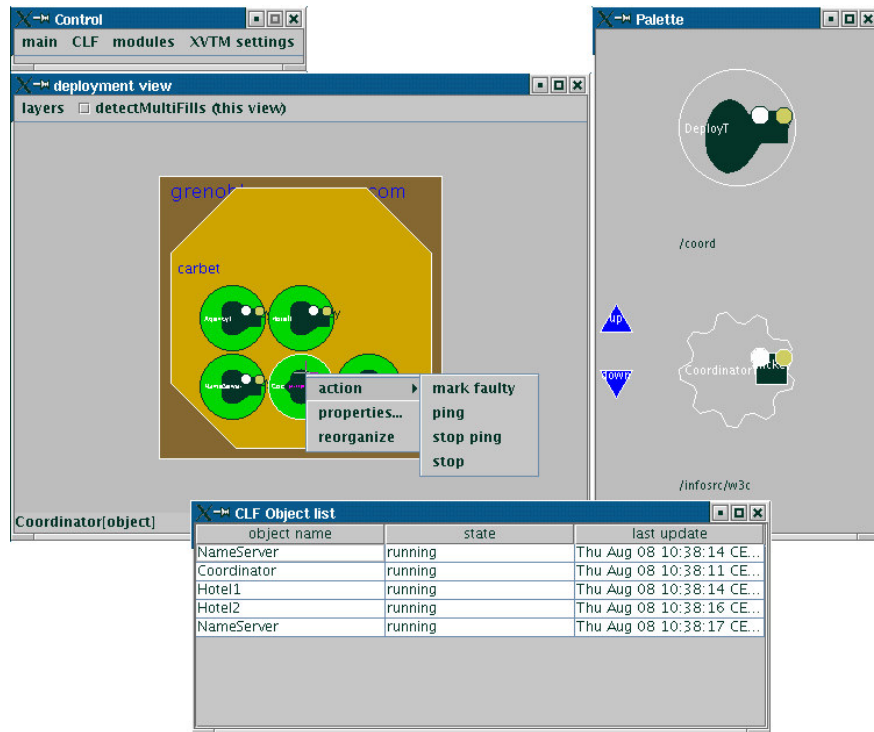
¹¹Le fait de ne pas projeter les glyphes non visibles n'a pas d'impact sur la génération des événements d'entrée/sortie du curseur souris dans un glyphe car l'algorithme de génération de ces événements (voir section suivante) travaille au niveau de l'espace virtuel et non pas au niveau de la vue (il effectue la projection inverse du curseur souris de manière à connaître ses coordonnées dans l'espace virtuel).

5.4 Conclusion

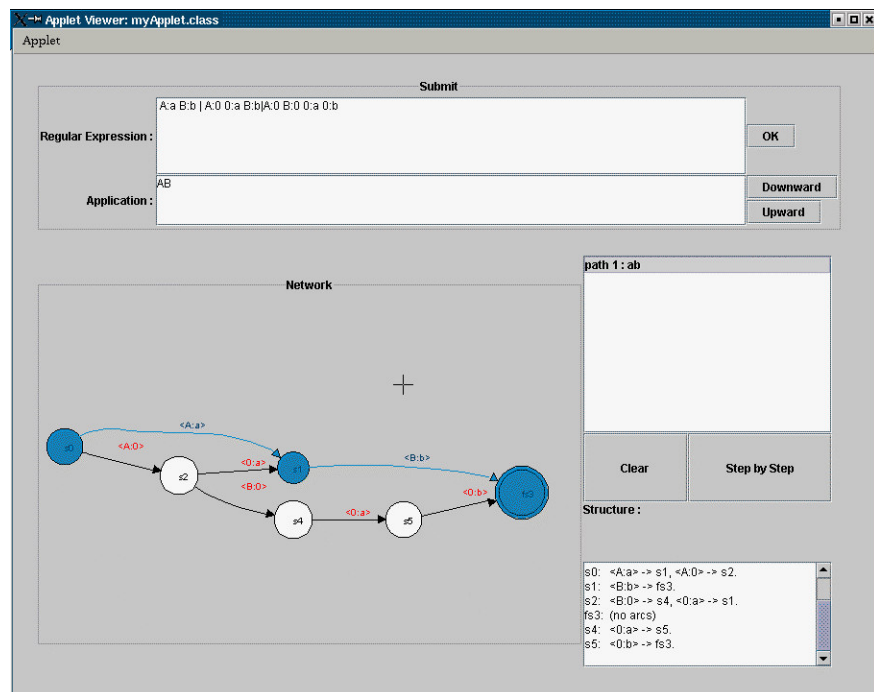
La XVTM est une boîte à outils Java pour la création d'interfaces graphiques en 2,5 dimensions. Elle est plus spécialement dédiée à la création d'éditeurs visuels ayant à manipuler de grandes quantités d'objets graphiques pouvant avoir des formes complexes. Il peut s'agir par exemple d'éditeurs de graphes ou d'environnements d'édition et d'exécution (très interactifs) pour langages de programmation visuels. La XVTM offre des fonctionnalités basées sur les principes étudiés dans les travaux de recherche sur la visualisation et la manipulation d'information, ces fonctionnalités ayant pour but de fournir des mécanismes et des composants "prêts à l'emploi" pour l'implémentation d'interfaces de qualité.

Les fonctionnalités principales de la XVTM sont :

- Capacités de zoom continu : le contrôle fin de l'altitude d'observation permet de rapidement placer la caméra à la position permettant d'observer la région désirée. Il est d'autre part possible d'animer les déplacements de la caméra impliquant un changement d'altitude.
- Gestion automatique et performante du rafraîchissement des fenêtres de visualisation : le programmeur n'a pas à se soucier de la mise à jour des vues suite aux changements liés à la manipulation des glyphes et des caméras.
- Fonctions de *clipping* spécifiques : deux algorithmes tenant compte de la spécificité du modèle XVTM sont appliqués en amont des fonctions de *clipping* automatique de Java 2D. Ce mécanisme rend le taux de rafraîchissement de chaque vue proportionnel au nombre d'objets effectivement visibles dans la vue et non pas proportionnel au nombre d'objets de l'espace virtuel.
- Modèle d'objets graphiques basé sur les dimensions perceptuelles de la sémiologie graphique : les variables visuelles du modèle correspondent aux dimensions perceptuelles identifiées par Jacques Bertin [24]. Le programmeur peut ainsi lier plus facilement les grandeurs à représenter aux attributs des glyphes, sachant qu'il est de plus possible d'exprimer des contraintes graphiques entre les glyphes. Le modèle fournit d'autre part une interface unifiée pour la manipulation des glyphes quel que soit leur type, facilitant ainsi la gestion des opérations de modification, qu'elles soient ponctuelles ou animées.
- Primitives et schémas d'animation : le programmeur peut animer les modifications portant sur les attributs perceptuels des glyphes en utilisant différents schémas d'animation ; ces derniers ont un impact sur la perception de l'utilisateur, c'est-à-dire sur les aspects cognitifs de l'interface.
- Vues multiples et vues multi-couches : les espaces virtuels peuvent contenir plusieurs caméras observant différentes régions. Chaque vue correspond à une fenêtre de l'interface. Une vue peut être composée d'une ou plusieurs caméras, auquel cas les régions observées par les différentes caméras sont superposées dans la même fenêtre sur des couches indépendantes.
- Interaction contrainte : les actions de l'utilisateur peuvent être fortement dirigées, ponctuellement ou bien tout au long de la session. Ce mécanisme permet de s'assurer que l'utilisateur ne commet pas d'erreur pendant les phases critiques de l'interaction.
- Fonction d'exportation du contenu des espaces virtuels au format SVG ([78], *Scalable Vector Graphics*). Un interpréteur SVG est aussi en cours de développement, qui ne couvre pour l'instant que partiellement la recommandation SVG 1.0 (support limité aux objets graphiques générés par la bibliothèque GraphViz [193] spécialisée dans le calcul de représentations visuelles optimisées pour les structures de graphe).



Outil pour le *monitoring* et la configuration d'applications CLF



Outil pour la visualisation d'automates à états finis

FIG. 5.17 : Autres applications utilisant la XVTM

La boîte à outils est distribuée publiquement [182] en tant que librairie *open-source* et a déjà été utilisée dans diverses applications par différentes équipes du Xerox Research Centre Europe et à l'extérieur :

- VXT : le langage de programmation visuel pour la spécification de transformations de documents XML étudié dans le chapitre 4. L'environnement interactif associé, basé sur la XVTM, fait l'objet du chapitre 7.
- IsaViz : un outil de visualisation et d'édition de modèles RDF (méta-données) représentés sous forme de graphes. IsaViz est étudié dans le chapitre 6.
- Panoramix : cet outil (figure 5.17) permet de visualiser l'état et de configurer des applications basées sur CLF [7], une plateforme *middleware* permettant la coordination de composants logiciels distribués, développée par l'équipe *Coordination Technologies* du Xerox Research Centre Europe. L'outil offre différentes vues de l'application (structure logique, charge réseau, etc.). L'utilisateur peut ainsi surveiller l'état de l'application, créer de nouveaux composants et interagir avec les composants existants.
- Outil pour la visualisation de machines à états finis : cette application (figure 5.17) permet de visualiser et d'éditer des automates à états finis (*Finite State Automata*) utilisés dans le domaine linguistique. Elle a été développée par l'équipe de recherche *Content Analysis* [93].

Ces différentes applications ont permis de valider plusieurs concepts. Les capacités de zoom continu et d'animation des caméras se sont révélées très utiles par exemple dans IsaViz pour la navigation dans les graphes de grande taille. Dans VXT, la possibilité de combiner plusieurs couches dans une même fenêtre et l'opacité paramétrable des glyphes ont permis de représenter les expressions de sélection et d'extraction d'éléments XML (*VPMEs*) sous la forme de masques (ou filtres) visuels s'adaptant aux structures sélectionnées. Sur le plan des performances et de la robustesse, la XVTM a maintenant atteint un niveau de maturité suffisant et peut être employée pour la conception d'applications de taille importante et disponibles publiquement telles que IsaViz.

Édition visuelle de méta-données

Dans le chapitre précédent, nous avons détaillé les fonctionnalités offertes par la boîte à outils XVTM pour le développement d'interfaces graphiques. Nous allons maintenant voir comment nous avons tiré parti de ces fonctionnalités pour créer une interface graphique conviviale dans un exemple d'application. Nous présentons dans ce chapitre un environnement graphique dédié à la visualisation et à l'édition de modèles RDF (*Resource Description Framework*) [164, 165, 151], une des briques de base du Web sémantique [203, 174].

Tim Berners-Lee définit le Web sémantique comme "*une extension du Web d'aujourd'hui dans lequel un sens bien défini est donné à l'information, rendant les machines et les personnes plus aptes à travailler en coopération*" [23]. Le *World Wide Web* fournit pour l'instant des documents adaptés aux personnes : le format de ces documents (HTML, SVG, SMIL), et les extensions telles que Javascript et les *applets* Java, sont orientés vers la présentation de l'information ; le contenu est peu structuré, même si certaines balises HTML comme h1 et les tables sont souvent en rapport avec la structure logique du document. Ces constructions conservent néanmoins une sémantique de présentation et ne peuvent pas être considérées comme un moyen sûr de récupérer des informations sur la structure et le contenu du document (certains types de balise comme les tables sont en effet employés à dessein de manière détournée pour obtenir des formatages de documents complexes). Les documents disponibles sur le Web sont donc principalement structurés en fonction de leur présentation et sont conçus pour le lecteur (humain). Cette présentation peut être très élaborée et inclure des éléments dynamiques (animations, composants pour l'interaction utilisateur comme des menus). Le contenu de ces documents est par contre difficilement exploitable par les machines, celles-ci n'ayant pas d'information sémantique sur ce contenu, telle que le sujet ou l'auteur d'une page Web donnée.

Le Web sémantique, souvent comparé à une base de connaissances globale [19, 176], fournit une nouvelle forme de contenu structuré que la machine est capable en partie de "comprendre", permettant ainsi à des programmes sur le Web (des *agents*) d'effectuer des tâches sophistiquées tirant parti de cette information. Il s'agit donc, par rapport aux documents Web tels que nous les connaissons aujourd'hui et que l'ordinateur se contente de transmettre et d'afficher, de fournir des informations supplémentaires ayant un sens bien défini, que ce même ordinateur sera capable d'exploiter automatiquement. La machine ne "comprendra" pas l'information dans le même sens qu'un humain peut la comprendre, mais elle sera capable de manipuler les données de manière plus efficace et pourra ainsi fournir des résultats plus intéressants (il sera par exemple possible d'effectuer des inférences sur ces données, ou encore d'établir des relations d'équivalence entre des ressources et propriétés décrites par différents vocabulaires). Le Web sémantique permettra par exemple à différents types d'appareils et de programmes de communiquer plus efficacement en partageant les mêmes définitions des concepts qu'ils utilisent. Il permettra aussi aux moteurs de recherche de fournir des résultats mieux ciblés en réponse à des requêtes utilisateur. En effet, les moteurs de recherche courants fournissent souvent une grande quantité de résultats dont certains n'ont que peu ou pas de rapport avec la requête initiale de l'utilisateur, le principe de base de la sélection d'une page étant l'occurrence des mots clés de la requête dans son contenu et dans la balise *meta* associée quand celle-ci existe¹. Les résultats des recherches peuvent être affinés en tenant compte des informations

¹Des moteurs plus évolués comme Google [107] utilisent des techniques plus complexes, comme la mesure de popularité de la page par comptage du nombre de références extérieures sur cette page, mais fournissent cependant des résultats toujours imprécis.

supplémentaires concernant la requête. Par exemple, un moteur de recherche pourra prendre en compte le fait qu'une requête concerne un article dont l'auteur se nomme "Fontaine" pour ne sélectionner que les articles dont l'auteur se nomme ainsi et non pas se contenter de retourner tous les articles dans lesquels le mot "fontaine" apparaît.

L'un des composants de base du Web sémantique est RDF, développé par le W3C pour exprimer des méta-données structurées décrivant des ressources du Web. Comme nous allons le voir plus en détail par la suite, un modèle RDF est un ensemble de triplets représentant un graphe orienté (*directed labelled graph*). Ce graphe peut être sérialisé sous différentes formes comme la syntaxe RDF/XML [14] ou encore Notation3 [20]. Ces sérialisations, notamment RDF/XML, sont très utiles pour l'échange des méta-données entre machines et leur utilisation par des agents. Elles sont par contre peu appropriées pour la lecture, la compréhension et l'édition des modèles par un utilisateur. Les représentations textuelles (unidimensionnelles) sont, comme nous l'avons vu dans les chapitres précédents, assez peu adaptées aux représentations de structures telles que les arbres et les graphes, dans le contexte de la visualisation et de l'édition de ces structures par un utilisateur humain. Au contraire, les représentations graphiques de ces modèles fournissent des versions plus lisibles des graphes pour lesquelles l'utilisateur a moins d'effort mental à fournir pour appréhender le modèle représenté. Nous décrivons dans ce chapitre IsaViz [179, 180], un outil pour la visualisation et l'édition de modèles RDF représentés sous forme de graphes. Après une brève introduction à RDF, nous présentons les fonctionnalités principales de l'outil en nous focalisant sur l'aspect IHM (Interface Homme-Machine). Nous détaillons ensuite quelques points cruciaux de l'architecture avant de conclure par quelques perspectives d'évolution.

6.1 Introduction à RDF

Les méta-données, c'est-à-dire les données à propos de données, représentent un moyen d'améliorer l'accès à de grandes collections d'information par la description de ces collections et des éléments qui les composent. Un exemple typique de méta-données est la fiche de classement associée à chaque livre d'une bibliothèque, contenant des informations telles que l'auteur, la date de parution, ou le code ISBN de l'ouvrage, sachant que la distinction entre données et méta-données n'est pas absolue, mais dépend du contexte d'utilisation de ces données. Le Web représente une gigantesque source d'information, qui par conséquent peut bénéficier de l'utilisation de méta-données.

RDF est un langage développé par le W3C pour la représentation d'information au sein du Web, et plus particulièrement dédié aux méta-données structurées décrivant des ressources reliées au Web, comme l'auteur d'une page XHTML, son titre ou le sujet abordé, les préférences d'un utilisateur ou encore les capacités d'affichage d'un PDA (*Personal Digital Assistant*). RDF peut être assimilé à un mécanisme de représentation de connaissances pour le Web, permettant à différentes communautés de créer des vocabulaires ayant une sémantique bien définie et exploitable par les machines, et permettant aussi de partager ces vocabulaires, c'est-à-dire de les mettre en commun pour décrire les ressources du Web et faciliter leur traitement automatique ; ceci grâce à un ensemble de contraintes exprimant des conventions à respecter sur la syntaxe, la sémantique et la structure des descriptions, afin de faciliter l'échange des méta-données.

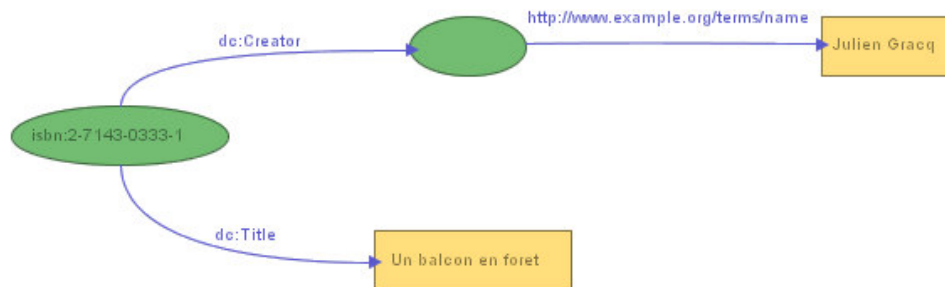


FIG. 6.1 : Modèle RDF pour un roman

6.1.1 Modèles RDF

RDF fournit un modèle pour la description de *ressources*. Dans le cadre de ce modèle, une ressource est définie comme n'importe quel objet identifiable de manière unique par une URI (*Uniform Resource Identifier*, [22]). Ces ressources ont des caractéristiques (ou attributs) appelées *propriétés*, chaque propriété reliant une ressource à une valeur qui peut être soit une autre ressource soit une valeur littérale (nombre, chaîne de caractères, ...). Un modèle est alors composé d'*affirmations*, ou déclarations (*statements*) à propos des ressources qui le composent. La terminologie utilisée pour parler de ces affirmations est la suivante :

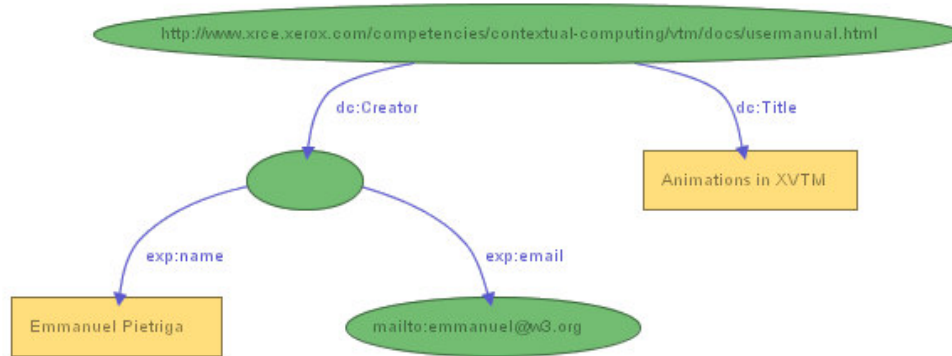
- une *affirmation* a la forme d'un triplet $\langle \text{ sujet } \rangle \langle \text{ prédicat } \rangle \langle \text{ objet } \rangle$,
- le *sujet* représente l'entité sur laquelle l'affirmation porte (il s'agit d'une *ressource*),
- le *prédicat* identifie la *propriété* définie sur le sujet par l'affirmation,
- l'*objet* représente la valeur de cette propriété. Il peut s'agir soit d'une ressource (qui peut être le sujet de cette même affirmation ou bien une autre ressource du modèle) soit d'une valeur *littérale*.

Un exemple simple d'affirmation est donné ci-dessous :

Julien Gracq est l'auteur de "Un balcon en forêt".

Le sujet de cette affirmation est *Julien Gracq*, le prédicat est *auteur* et l'objet "*Un balcon en forêt*". Il est bien entendu possible d'associer plusieurs propriétés à une même ressource. Un modèle RDF est donc un ensemble de triplets du type $\langle \text{ sujet } \rangle \langle \text{ prédicat } \rangle \langle \text{ objet } \rangle$. Cet ensemble de triplets forme un graphe constitué de nœuds et d'arcs orientés et labélisés (*nodes and arcs diagram* ou *directed, labelled graph*). Les nœuds correspondent aux ressources et aux valeurs littérales, les arcs aux propriétés associées aux ressources. Les nœuds sont labélisés soit, dans le cas des ressources, par l'URI associée, soit, dans le cas des littéraux, par la valeur du littéral. Les propriétés sont labélisées par le type de la propriété.

Il est possible de définir dans le modèle des nœuds anonymes sans URI (récemment renommés *bNodes* pour *blank nodes*). Ces nœuds ne peuvent pas être adressés directement, faute d'identifiant. Cela pose



```

1. <rdf:RDF xmlns:dc='http://purl.org/metadata/dublin_core#'
2.   xmlns:rdf='http://www.w3.org/1999/02/22-rdf-syntax-ns#'
3.   xmlns:exp='http://www.example.org/terms/' >
4. <rdf:Description rdf:about='http://www.xrce.xerox.com/competencies/contextual-computing/vtm/docs/usermanual.html'>
5.   <dc:Creator rdf:parseType='Resource'>
6.     <exp:email rdf:resource='mailto:emmanuel@w3.org' />
7.     <exp:name>Emmanuel Pietriga</exp:name>
8.   </dc:Creator>
9.   <dc:Title xml:lang='en'>Animations in XVTM</dc:Title>
10. </rdf:Description>
11. </rdf:RDF>

```

```

20. @prefix : <http://purl.org/metadata/dublin_core#> .
21. @prefix exp: <http://www.example.org/terms/> .
22. @prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
23. <http://www.xrce.xerox.com/competencies/contextual-computing/vtm/docs/usermanual.html> :Creator [
24.   <http://www.example.org/terms/email> <mailto:emmanuel@w3.org> ;
25.   <http://www.example.org/terms/name> "Emmanuel Pietriga" ] ;
26. :Title "Animations in XVTM" .

```

FIG. 6.2 : Fragment RDF pour la documentation XHTML de XVTM (Syntaxes RDF/XML et Notation3)

certain problèmes, notamment au niveau de la sérialisation (suivant la syntaxe utilisée, il peut être nécessaire d'engendrer des identifiants uniques temporaires pour désigner ces nœuds). Dans l'exemple de la figure 6.1, un modèle RDF décrivant un livre, nous utilisons l'espace de noms *isbn* appartenant à l'espace des URNs² mais pas encore normalisé. Nous utilisons aussi un *bNode*. Le modèle peut être interprété de la manière suivante : "la ressource identifiée par le numéro ISBN 2-7143-0333-1 a pour titre *Un balcon en forêt* et un créateur dont le nom est *Julien Gracq*". La ressource symbolisant le créateur n'a pas de label et ne peut être désignée que par les propriétés qui la définissent.

6.1.2 Syntaxe XML

Il existe plusieurs sérialisations pour les modèles RDF. La syntaxe la plus utilisée et favorisant l'échange de modèles est RDF/XML [144], récemment révisée [14]. Une syntaxe plus simple mais en théorie limitée aux tests d'analyseurs RDF est NTriples. Tim Berners-Lee propose aussi Notation3 [20], un sur-ensemble de NTriples gérant entre autre les préfixes pour les espaces de noms et des quantificateurs.

²Le type le plus connu d'URI est l'URL (*Uniform Resource Locator*), mais ne se limite pas à cet ensemble [213].

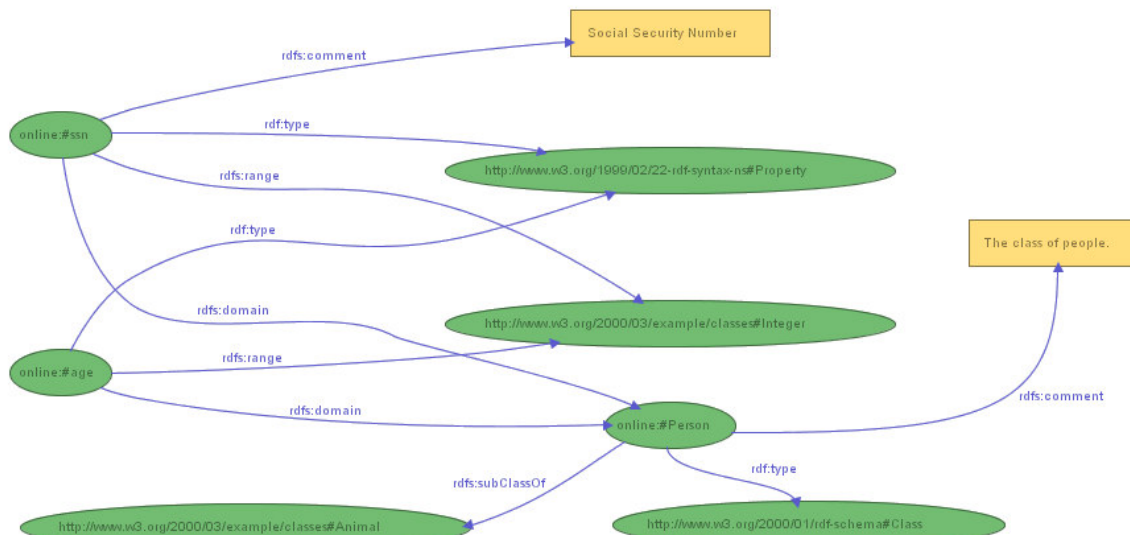


FIG. 6.3 : Exemple de RDF Schema

La figure 6.2 présente la sérialisation RDF/XML (syntaxe abrégée), d'un modèle que nous utilisons pour enrichir le document qui nous sert d'exemple depuis le chapitre 2, et l'équivalent Notation3 généré par *cwm* (*Closed World Machine*) [21]. L'élément RDF contient les déclarations d'espaces de noms et toutes les descriptions. Dans notre cas, nous décrivons la page *usermanual.html* de la documentation XVTM. Nous spécifions que la page *usermanual.html* (ligne 4) dont le titre est *Animations in XVTM* (ligne 9), a été créée (ligne 5) par une personne dont le nom est *Emmanuel Pietriga* (ligne 7) et l'adresse e-mail *emmanuel@w3.org* (ligne 6). La syntaxe abrégée autorise l'emboîtement des descriptions les unes dans les autres (lignes 5 à 8), ce qui permet ici de ne pas avoir à désigner explicitement le *bNode* symbolisant l'auteur de la page. La sérialisation XML des modèles RDF présente beaucoup de subtilités [44] que nous ne développons pas ici, notre but étant simplement d'introduire les notions de base de RDF pour une meilleure compréhension de l'outil décrit dans ce chapitre.

6.1.3 Définition de vocabulaires

Dans les exemples développés précédemment, nous utilisons des propriétés telles que *Creator* ou *Title*. Ces propriétés appartiennent à des vocabulaires RDF et ont une sémantique bien définie. Un exemple de vocabulaire très utilisé est Dublin Core [69], un ensemble d'éléments (auteur, titre, date, format, ...) pour décrire des documents. Un autre exemple est CC/PP (*Composite Capabilities/Preference Profiles* [133]), un vocabulaire permettant de décrire le profil des appareils électroniques (capacités d'affichage, etc.) et les préférences des utilisateurs dans le contexte de l'adaptation de contenu au cours de la visualisation de documents Web. Chaque vocabulaire est associé à un espace de noms afin de pouvoir gérer les conflits qui apparaissent dans les cas où différentes organisations définissent des éléments de même nom n'ayant pas la même sémantique. Par exemple, l'espace de noms associé aux éléments Dublin Core est "http://purl.org/metadata/dublin_core#", souvent abrégé "dc:".

Pour décrire ces vocabulaires, le W3C propose *RDF Schema* (RDFS) [37], une recommandation qui accompagne RDF. Ce dernier fournit un modèle pour décrire des ressources, des propriétés sur ces ressources ainsi que des relations entre ces ressources. Mais la description de vocabulaires RDF nécessite des mécanismes pour décrire les classes (ou types) de ressources, ainsi que les propriétés et les types de ressources sur lesquelles ces propriétés peuvent s'appliquer. Ces mécanismes sont fournis par *RDF Schema*, qui utilise le modèle RDF pour définir son propre système de types. Pour cela, RDFS définit un vocabulaire consistant en un ensemble de ressources et de propriétés prédéfinies comme `rdfs:Class`, `rdfs:Resource`, ou `rdfs:subClassOf`. La figure 6.3 montre un exemple de fragment de modèle RDFS pour un vocabulaire décrivant des personnes. Dans ce schéma, `Person` est une classe qui est elle-même sous-classe de `Animal`. Deux propriétés sont définies : `age` et `ssn` (*Social Security Number*). Pour chacune de ces propriétés, le domaine (i.e. la classe dont les instances peuvent être le sujet d'une affirmation dont le prédicat est du type de la propriété en cours de définition) est défini comme la classe `Person`, `rdfs:range` définissant quels types d'objets peuvent être associés à ces mêmes prédicats (dans ce cas un nombre entier).

Les schémas RDF jouent un rôle différent des schémas XML. Les schémas XML contraignent la structure des documents XML au moyen de types de données primitifs et structurés, incorporant par exemple la notion d'héritage. Les schémas RDF sont plus généraux, en ce sens qu'ils décrivent des contraintes sur les modèles appartenant à un vocabulaire donné. Ces contraintes³ étant exprimées sur les ressources et les propriétés du modèle, elles peuvent se traduire par des contraintes sur la structure de la sérialisation RDF/XML du modèle. La différence est donc assez subtile et génère des discussions quant à l'intérêt de RDFS sur les listes de distributions relatives à RDF [227]. De notre point de vue, l'utilisation de RDFS semble légitime dans le sens où un schéma RDF porte sur les entités du modèle RDF et non pas sur une sérialisation XML de ce modèle. De manière plus générale, RDFS est spécifiquement conçu pour la description de vocabulaires RDF et semble donc mieux adapté que ne le seraient les schémas XML qui, même s'il existe effectivement des recouvrements entre les deux langages, répondent à une classe de problèmes différente. La critique qui consiste à affirmer que les spécifications RDFS pourraient être exprimées par des *XML Schema* n'est qu'une instance du problème classique de niveau d'abstraction, d'équilibre entre complexité et expressivité d'un langage : du point de vue de l'expressivité, tout programme pourrait être exprimé en assembleur, ou même en code binaire. Mais une telle solution aurait des conséquences à de nombreux points de vue, par exemple au niveau de la facilité de spécification et de correction des programmes. RDFS propose un ensemble restreint de concepts simples et assez puissants pour spécifier des vocabulaires RDF en manipulant directement des entités du monde RDF, ce qui n'est pas le cas pour *XML Schema* qui travaille sur la structure des documents XML. La plupart des contraintes RDFS pourraient être exprimées au moyen de *XML Schema*, mais avec un niveau de complexité supérieur, et avec le défaut supplémentaire que les contraintes ne porteraient pas directement sur les modèles RDF eux-mêmes, mais sur leur sérialisation RDF/XML, ce qui n'est pas la même chose, surtout si l'on tient compte du fait que RDF/XML n'est qu'une des syntaxes existantes pour RDF.

Pour clore cette introduction à RDF, nous mentionnerons aussi l'existence de DAML+OIL (*DARPA Agent Markup Language, Ontology Inference Language*) [60], un langage pour la description d'ontolo-

³Nous ne parlons pas ici du typage des littéraux qui est défini par RDF Datatyping (en cours de développement) mais des contraintes portant sur le typage des ressources et les domaines d'application des propriétés.

gies (définies comme *une description formelle des concepts et relations existant pour un agent ou une communauté d'agents* [113]). Les ontologies sont des manières de décrire le sens des termes et les relations existant entre ces termes. Les ontologies permettent donc aux machines de manipuler les termes plus facilement et de décider comment établir des correspondances entre elles. DAML+OIL étend RDFS avec de nouvelles primitives de modélisation permettant d'exprimer des relations telles que l'équivalence, l'inversion, des restrictions sur les propriétés (cardinalité, . . .), et des contraintes sur les types de données utilisant *XML Schema Datatypes* [25]. Cet ensemble de propriétés sert de base au mécanisme d'inférence, qui consiste à dériver de nouvelles données à partir de données déjà existantes. Dans le cas du Web sémantique, les inférences sont effectuées automatiquement par des agents autonomes, c'est-à-dire par les machines.

6.2 IsaViz

RDF a été défini comme un langage pour la représentation de méta-données pouvant être lues par des personnes et traitées par des machines. L'une des idées du Web sémantique est que ces méta-données doivent, pour une grande partie, être engendrées automatiquement par les machines, soit par conversion à partir d'autres formats de représentation de connaissances (par exemple des bases de données), soit par inférence à partir de connaissances déjà exprimées (utilisation des ontologies). Il est néanmoins fréquent que des utilisateurs humains aient à créer des quantités plus ou moins importantes de méta-données, c'est-à-dire à créer des modèles RDF ainsi que des vocabulaires en utilisant RDFS et/ou DAML+OIL. Pour cela, la méthode la plus grossière consiste à éditer les modèles sous leur forme sérialisée, que ce soit RDF/XML ou Notation3. Comme nous l'avons vu précédemment, ces représentations textuelles de structures de graphes ne sont pas les plus adaptées dans le cadre de la visualisation et de l'édition par des humains. Une des preuves en est que la plupart des exemples de modèles fournis dans les diverses recommandations et notes du W3C relatives à RDF sont représentés sous forme de graphe et non pas en utilisant une syntaxe textuelle.

Il semble donc naturel d'envisager un outil pour la visualisation et l'édition de modèles RDF représentés sous forme visuelle. C'est en tout cas le souhait émis par l'équipe du W3C et les membres de la communauté RDF. Une des premières réponses à ce souhait est le validateur RDF du W3C (*RDF Validator* [225]). Ce service disponible sur le Web fournit une représentation graphique d'un modèle sous forme d'image bitmap ou de dessin vectoriel (SVG) à partir de sa sérialisation RDF/XML. Les images bitmap sont malheureusement peu exploitables quand il s'agit de visualiser des modèles de taille importante. D'autre part, *RDF Validator* fournit une représentation statique et découplée du modèle qu'il n'est pas possible d'éditer : il ne s'agit que d'un service de visualisation.

Nous présentons ici IsaViz [179, 180], un outil pour la visualisation et l'édition de modèles RDF représentés graphiquement. Notre but n'est pas ici de dresser une liste exhaustive des fonctionnalités proposées par l'environnement ni d'écrire son manuel d'utilisation, disponible par ailleurs [181]. Nous nous concentrons sur certains points relatifs à l'interface homme-machine proposée et fondés sur les principes abordés dans le chapitre 3.

IsaViz repose sur la XVTM pour fournir une interface zoomable adaptée à la navigation dans les gros modèles, dont la représentation initiale est calculée par une bibliothèque spécialisée dans l'agencement spatial (*layout*) de graphes. IsaViz permet l'importation et l'exportation de modèles utilisant une sérialisation RDF/XML ou NTriples, ainsi que l'exportation de la représentation graphique du modèle vers les formats PNG [74] (images bitmap) et SVG [78] (dessins vectoriels). La figure 6.4 est une capture d'écran de l'interface graphique d'IsaViz. Cette interface est composée de quatre fenêtres principales :

- une palette d'icônes regroupant l'ensemble des commandes de navigation et d'édition,
- une vue XVTM dans laquelle est représenté le modèle,
- une fenêtre permettant d'éditer les attributs de l'entité sélectionnée dans le graphe (ressource, littéral ou propriété),
- une fenêtre regroupant des définitions générales, comme les déclarations des espaces de noms utilisés par le modèle, une liste des types de propriétés, ainsi qu'un *browser* textuel récapitulant toutes les propriétés attachées à la ressource sélectionnée, qui peut aussi être affiché dans une fenêtre indépendante.

6.2.1 Représentation et navigation dans les modèles

Agencement spatial

Les modèles RDF ont une structure de graphe dont les arcs sont étiquetés et orientés (*DLG : Directed, Labelled Graph*). Ces graphes sont des structures abstraites, et les sérialisations RDF ne contiennent pas d'information géométrique relative à l'aspect visuel du graphe. Il n'existe de toute façon pas de représentation unique d'un graphe donné, mais une infinité de solutions, qui fournissent une représentation plus ou moins lisible et exploitable par l'utilisateur, la solution optimale variant en fonction de la raison pour laquelle le graphe est visualisé. Comme nous l'avons vu dans l'état de l'art, il existe différents algorithmes dédiés à la représentation visuelle de graphes et capables de gérer différents types de graphes. L'idée générale de ces algorithmes est d'améliorer la lisibilité des représentations, par exemple en évitant la superposition d'objets et en minimisant les coupures entre arcs (*edge crossing*) tout en conservant une géométrie simple (les chemins trop sinueux sont difficiles à suivre). La lisibilité peut aussi être améliorée en mettant en évidence des motifs (symétrie, répétition) dans la structure du graphe. Ces techniques sont utilisées par GraphViz [102, 103] de AT&T Research, et plus particulièrement dans le programme *dot*, spécialisé dans la représentation de graphes orientés. Ce programme fournit une représentation visuelle d'un graphe à partir de sa description abstraite (utilisant une syntaxe spécifique à *dot*). La représentation visuelle est engendrée soit sous la forme d'image bitmap, soit sous la forme de fichier *dot* enrichi d'information de placement, ou encore sous la forme de fichier SVG. C'est cette dernière option que nous utilisons, la XVTM disposant d'un interpréteur SVG. Cet interpréteur génère un espace virtuel et y place les glyphes XVTM correspondant aux déclarations du fichier SVG. Le mécanisme d'association de ces glyphes avec les entités du modèle RDF obtenu par une voie parallèle est décrit dans la section 6.2.3.

Navigation

La navigation dans le graphe se fait alors suivant les principes des vues XVTM. Une caméra est créée dans l'espace virtuel ; cette caméra peut être déplacée dans l'espace et il est possible de changer son altitude, ce qui revient à effectuer des opérations de zoom dans le graphe. Cette faculté se révèle utile lors de la visualisation de modèles de taille importante (voir par exemple la figure 6.4), permettant de

The screenshot displays the IsaViz 1.1 graphical interface. The main window shows a dense network graph of RDF triples. The interface includes a toolbar with various editing tools, a search bar, and a definitions panel on the right. The definitions panel is currently showing the 'Properties of Resource' for the URI <http://www.daml.org/2001/103/football-on#Team>.

Properties of Resource <http://www.daml.org/2001/103/football-on#Team> (Team) (Team)

- `rdfs:subClassOf` (AR)
- `rdsl:label` (L) Team
- `rdfs:subClassOf` (AR)
- `oil:creationDate` (L) 12:59:26.08.2001
- `rd:type` (R) <http://www.w3.org/200001/rdf-schema#Class>
- `rdfs:subClassOf` (AR)
- `rdsc:comment` (L)
- `rdfs:subClassOf` (AR)

Definitions

| Namespace | Property Types | Property Browser | Prefix | Property name |
|---|----------------|------------------|--------|---|
| http://www.daml.org/2001/103/damh-ol# | | | daml | onProperty |
| http://www.daml.org/2001/103/damh-ol# | | | daml | toClass |
| http://www.daml.org/2001/103/damh-ol# | | | daml | toClassQ |
| http://www.daml.org/2001/103/damh-ol# | | | daml | versionInfo |
| http://www.w3.org/1999/02/22-rdf-syntax-ns# | | | rdf | _?? (Membership property auto-numbering: _1, _2, ...) |
| http://www.w3.org/1999/02/22-rdf-syntax-ns# | | | rdf | li |
| http://www.w3.org/1999/02/22-rdf-syntax-ns# | | | rdf | object |
| http://www.w3.org/1999/02/22-rdf-syntax-ns# | | | rdf | predicate |
| http://www.w3.org/1999/02/22-rdf-syntax-ns# | | | rdf | subject |

FIG. 6.4 : IsaViz 1.1 : Interface graphique

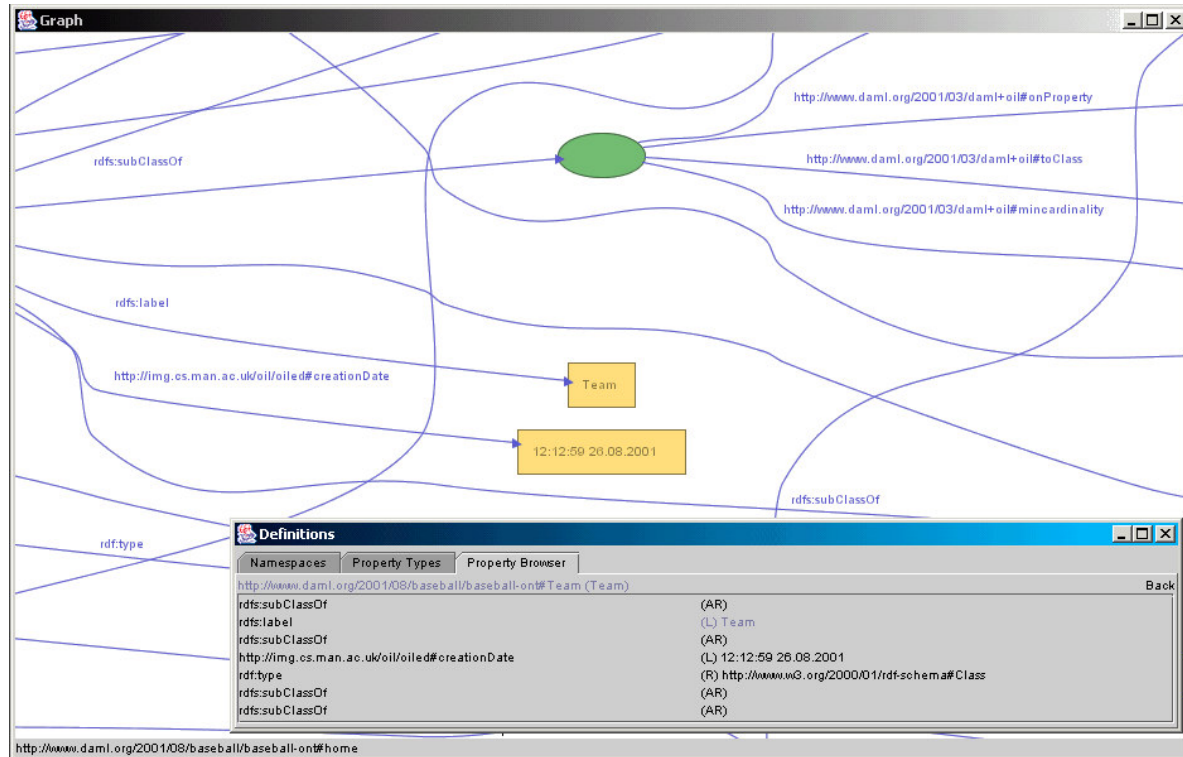


FIG. 6.5 : IsaViz : Property Browser

passer rapidement d'une vue d'ensemble du graphe à des vues plus détaillées de régions spécifiques. L'utilisateur peut délimiter une région d'intérêt ou cliquer sur un nœud spécifique, et la caméra se déplace automatiquement (translation et zoom) pour adapter au mieux la vue par rapport à ce nouveau centre d'intérêt. Ces déplacements sont animés, de manière à prendre en compte le principe de continuité perceptuelle. Le schéma d'animation retenu est *slow-in/slow-out* qui fournit les déplacements les plus plaisants sur le plan esthétique et sur le plan cognitif (réalisme du mouvement et concentration sur les phases importantes de l'animation, voir section 5.2.3).

Pour certaines configurations de graphes, il n'est pas possible d'obtenir automatiquement une représentation optimale pour tous les nœuds. En particulier, pour certaines ressources sujets d'un grand nombre d'affirmations dont les objets sont éloignés sur le plan géométrique, il peut être difficile d'obtenir une vue complète des propriétés reliées à la ressource sans avoir à placer la caméra à une altitude très haute, configuration dans laquelle n'apparaissent pas les détails de ces propriétés. Pour pallier ce problème, IsaViz offre un *Property Browser* (figure 6.5), qui fournit de manière synthétique la liste complète des propriétés faisant partie d'affirmations dont le nœud sélectionné dans le graphe est le sujet. Cette liste présente pour chacune de ces affirmations le type de la propriété ainsi que le nom et la nature de l'objet associé. Chaque membre de cette liste fonctionne comme un hyperlien, la traversée de cet hyperlien ayant pour effet d'afficher la liste des affirmations dont l'objet précédent est le sujet (le lien n'est donc

actif que pour les objets de type ressource). Il est aussi possible de synchroniser cette vue avec la vue graphique puisqu'un clic droit sur l'un des éléments de la liste affichée par le *Property Browser* a pour effet de positionner la caméra de la vue XVTM de manière optimale par rapport à l'objet ainsi désigné. Cette fonctionnalité est importante car les deux types de représentations sont utilisés pour des tâches différentes et l'utilisateur doit pouvoir passer de l'une à l'autre et continuer son travail sans fournir d'effort supplémentaire quant à son orientation dans le modèle.

Enfin, l'environnement fournit des fonctions de recherche et de sélection. La fonction de recherche rapide permet de trouver dans le graphe tous les nœuds et tous les arcs dont le texte⁴ contient la chaîne de caractères donnée. La fonction de recherche avancée permet de spécifier des critères de sélection plus fins, comme le type de l'entité à sélectionner : ressource, littéral, propriété ou indifférent. Les fonctions de recherche fournies par l'environnement sont très utiles, que la représentation soit textuelle ou visuelle. Elles sont cependant relativement peu présentes dans les langages de programmation visuels [111] et pourraient être améliorées, par exemple en ajoutant un mode de spécification visuel des critères de recherche. Dans le cadre d'IsaViz, la fonction de recherche avancée semble suffisante étant donné le nombre peu élevé de types d'objets graphiques et leur correspondance directe avec les entités RDF. Une fonction de recherche visuelle plus complexe pourrait permettre à l'utilisateur d'ébaucher des fragments incomplets de modèles qui seraient convertis en critères de recherche, ou plus exactement en requêtes. Nous nous rapprochons alors en effet d'un langage de requêtes visuel, une fonctionnalité considérée dans les évolutions futures de l'outil.

6.2.2 Édition des modèles

L'intérêt principal d'IsaViz par rapport aux autres outils visuels existants dédiés à RDF est la possibilité de créer et d'éditer les modèles à partir de leur représentation visuelle. Il existe un autre outil offrant lui aussi des possibilités d'édition, appelé RDFAuthor [200]. Cet outil est de notre point de vue plus limité, notamment au niveau de la résistance au facteur d'échelle (*scalability*) : l'algorithme d'agencement spatial du graphe est très basique et les arcs représentant les propriétés sont limités à des flèches droites. La figure 6.6 montre le même modèle RDF après importation et agencement automatique, dans RDFAuthor (à gauche), et dans IsaViz (à droite), qui fait appel à GraphViz pour calculer la position des nœuds et la forme des arcs. RDFAuthor fournit d'autre part des fonctions de navigation très limitées et peu ergonomiques (translation par barre de défilement et zoom discret contrôlé par une règlette). RDFAuthor présente néanmoins les intérêts suivants : il supporte la représentation Unicode (modèles utilisant par exemple des caractères asiatiques) et il est possible d'exprimer des requêtes visuelles à partir de graphes incomplets.

Interaction contrainte

IsaViz n'est pas un simple outil d'édition de graphes orientés, mais a été conçu spécialement pour RDF. Les contraintes spécifiques aux graphes RDF ont été modélisées dans l'environnement (on parle d'un *RDF-aware environment*), ce qui se traduit concrètement par une limitation des actions que peut entreprendre l'utilisateur au cours de l'édition, ainsi que par le déclenchement automatique de commandes

⁴L'URI dans le cas des ressources et des propriétés, la valeur de la chaîne de texte dans le cas des littéraux.

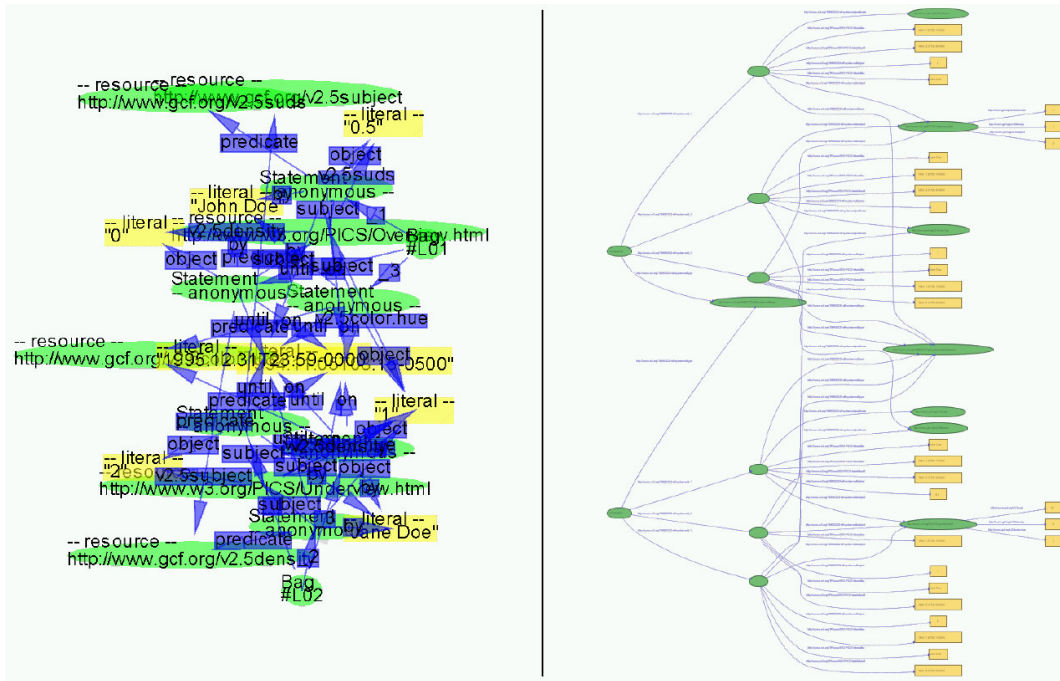


FIG. 6.6 : Représentation du même modèle par RDFAuthor et IsaViz

en réponse à certaines actions, ceci afin de garantir la validité du modèle par rapport à la recommandation RDF du W3C [144]. L'interaction ne doit cependant pas être trop contrainte de manière à ne pas frustrer l'utilisateur en l'empêchant de réaliser des séquences d'actions qui pourraient rendre le modèle temporairement invalide mais qui sont plus pratiques du point de vue du processus d'édition (voir section 5.2.4, le concept d'interaction visuelle légitime de Bottoni [32] et la notion de *Premature commitment* de Green [111]). Ainsi, nous n'empêchons pas la création de ressources et de littéraux sans valeur associée, cette valeur pouvant être spécifiée plus tard. Par contre, nous interdisons les actions d'édition présentant un caractère d'erreur évident, comme la création d'une affirmation ayant pour sujet un nœud de type littéral.

Dans le même esprit, et aussi afin de limiter la viscosité (effort requis pour effectuer un changement, voir section 3.4.2), l'environnement détruit automatiquement les propriétés attachées aux nœuds supprimés par l'utilisateur, ceci afin de ne pas se retrouver avec des arcs flottants (*pending edges*). IsaViz fournit aussi des fonctions de couper/copier/coller ainsi qu'une fonction permettant d'annuler les dernières actions. Ces fonctions sont plus complexes à implémenter que leurs équivalents textuels car elles doivent gérer des structures de données plus complexes : la copie d'une ressource engendre une nouvelle URI unique afin de préserver la validité du graphe ; lorsqu'un ensemble de nœuds et d'arcs sont copiés, leur collage entraîne une duplication de l'ensemble des entités et leur ajout au modèle en mémoire⁵.

⁵Les arcs sélectionnés pour la copie ayant au moins un des deux nœuds associés non sélectionnés ne sont pas collés afin, encore une fois, de ne pas avoir d'arc flottant.

Géométrie

Lors de l'importation des modèles RDF, IsaViz fait appel au programme *dot* pour calculer l'agencement spatial initial du graphe. L'utilisateur peut ensuite modifier la disposition et la taille des nœuds ainsi que le chemin emprunté par les arcs. Les différentes sérialisations ne conservent pas l'information géométrique associée aux différentes entités. Les modifications effectuées par l'utilisateur ne sont donc pas conservées d'une session à l'autre. Il est cependant intéressant de pouvoir restituer les modèles en fonction de l'apparence que leur a donné l'utilisateur. En effet, comme nous l'avons vu dans le chapitre 5, les vues XVTM et le mode de navigation renforcent l'idée que les objets ont une localisation géométrique constante rendant ainsi l'orientation de l'utilisateur dans le modèle plus aisée. Permettre à l'utilisateur de modifier la position des objets à sa convenance contribue aussi à ce résultat en lui fournissant potentiellement des repères visuels plus intuitifs et familiers. L'utilisateur peut plus facilement retrouver une ressource dans le graphe s'il se rappelle, même vaguement, où il l'a placée dans l'espace virtuel.

Afin de pouvoir enregistrer l'information géométrique, nous avons défini un format XML spécifique pour les projets IsaViz. L'utilisateur peut utiliser ce format pour sauvegarder les projets sur lesquels il travaille. Les fichiers projets ISV sauvegardent le modèle augmenté de l'information géométrique associée à chaque nœud et chaque arc du graphe ainsi que quelques attributs supplémentaires abordés plus loin dans cette section. Ces fichiers sont plus volumineux que les sérialisations RDF/XML ou NTriples des modèles car ils enregistrent plus d'information et ne peuvent pas bénéficier des techniques d'abréviation utilisées dans la syntaxe RDF/XML. Nous avons envisagé au départ d'enrichir la sérialisation RDF/XML par l'information géométrique représentée en SVG. Cette solution rencontre malheureusement certains obstacles, comme l'absence d'élément représentant les nœuds anonymes (*bNodes*). Cette possibilité n'est cependant pas écartée, mais demande à être étudiée en profondeur en tenant compte de la nouvelle spécification du W3C relative à la syntaxe [14] afin de fournir une solution élégante et respectant cette nouvelle spécification.

Notations secondaires

Le format ISV permet de plus de proposer dans l'environnement une fonction de désactivation des nœuds et des arcs du modèle. Cette fonction peut être comparée à la mise en commentaire d'éléments dans les fichiers XML. Ces éléments existent toujours dans la représentation visuelle du modèle, mais ils ne font pas partie du modèle mémoire et ne sont pas exportés lors de la sérialisation en RDF/XML ou NTriples. Ils sont affichés en utilisant une couleur grise peu contrastée par rapport à la couleur de fond d'écran de manière à ne pas interférer avec les éléments actifs du graphe, améliorant ainsi la perception qu'en a l'utilisateur. Comme précédemment, le fait de désactiver un nœud du graphe a pour conséquence la désactivation automatique de toutes les propriétés qui lui sont attachées, afin de ne pas engendrer un modèle invalide.

La désactivation des nœuds est aussi un moyen de créer des notations secondaires [111]. En effet, comme avec les fonctions de commentaire des langages textuels, il est possible de créer des littéraux désactivés qui serviront de boîtes de texte dans lesquelles l'utilisateur pourra écrire des commentaires textuels pour annoter le modèle. Les possibilités d'annotation et surtout de désactivation sont encore

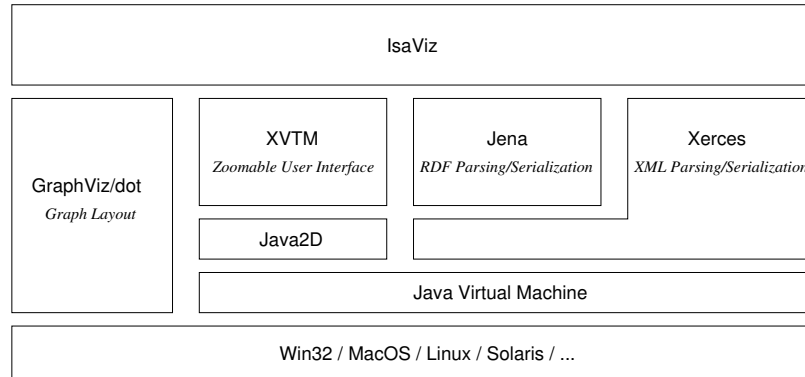


FIG. 6.7 : Bibliothèques utilisées par IsaViz

rare dans les environnements visuels, alors que le fait d’avoir une représentation visuelle ne modifie en rien leur besoin.

6.2.3 Architecture

IsaViz a été implémenté en Java 2, ce qui nous permet d’utiliser les bibliothèques (figure 6.7) suivantes, aussi implémentées en Java :

- XVTM pour l’interface graphique zoomable (détaillée dans le chapitre précédent),
- Jena [139], une boîte à outils dédiée à la manipulation de modèles RDF, permettant leur analyse, sérialisation et modification,
- Xerces [100], un parseur XML utilisé par Jena pour l’analyse syntaxique des fichiers RDF/XML et leur sérialisation, ainsi que par IsaViz et la XVTM pour l’analyse et la sérialisation des fichiers ISV et SVG.

La dernière bibliothèque, GraphViz [193], utilisée pour l’agencement spatial des graphes, n’est pas implémentée en Java. Il existe cependant des implémentations natives pour un grand nombre de systèmes d’exploitation tels que Windows, Solaris, Linux ou encore Mac OS. IsaViz peut par conséquent être utilisé dans la plupart des environnements, même si ce dernier point complique un peu la procédure d’installation.

Association des glyphes aux entités de la représentation abstraite du modèle

IsaViz conserve en mémoire tout au long de l’exécution une représentation abstraite du modèle. Ce modèle mémoire est obtenu par conversion du modèle fourni par Jena, auquel il faut associer le modèle graphique engendré par l’interpréteur SVG de la XVTM. La figure 6.8 représente le processus d’association : Jena analyse le modèle RDF sérialisé et génère un ensemble d’affirmations envoyé à IsaViz. À partir de cet ensemble d’affirmations, qui représentent le modèle, IsaViz construit une représentation abstraite du modèle en mémoire, mieux adaptée à sa manipulation que l’ensemble des triplets fourni par Jena. En parallèle, IsaViz génère un fichier temporaire au format *dot* avant d’appeler le programme éponyme qui engendrera une représentation graphique du modèle au format SVG. La XVTM peut alors interpréter ce

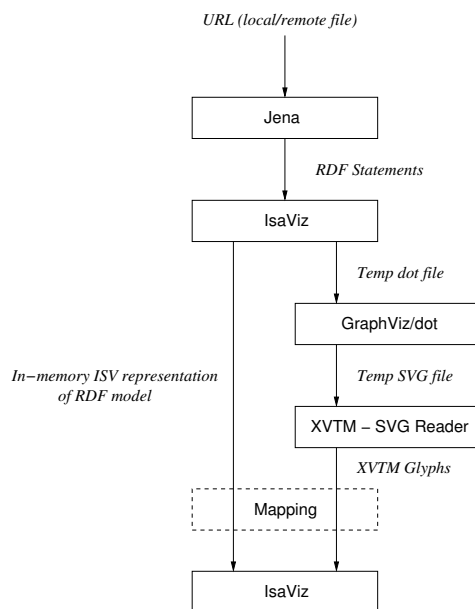


FIG. 6.8 : Établissement de la correspondance entre modèle mémoire et représentation graphique

fichier et créer les glyphes correspondants. Il reste à établir la correspondance des ressources, des littéraux et des propriétés du modèle abstrait, avec les glyphes XVTM. Une première méthode pour établir cette correspondance utilisait le fait que chaque ressource peut être identifiée de manière unique par son URI, information que l'on retrouve sous la forme du label attaché aux ellipses dans la représentation graphique. Cette méthode ne nous permettait malheureusement pas de discriminer ni entre les littéraux de même valeur (qui peuvent co-exister), ni entre les propriétés de même type, à moins de procéder à une analyse topologique du graphe en établissant des relations spatiales entre les entités graphiques. Nous avons donc opté pour une solution plus simple et plus robuste consistant à engendrer des identifiants uniques pour chaque entité du graphe abstrait, ces identifiants étant préservés par *dot* jusque dans les objets graphiques décrits par le fichier SVG.

6.2.4 Évolutions futures

IsaViz est distribué publiquement par le W3C depuis le mois de mars 2002. La version actuelle (1.1) est basée sur une version améliorée de XVTM utilisant la technique de dessin *XOR mode* [135] qui améliore de façon considérable le taux de rafraîchissement dans certains cas particuliers. Cette amélioration a été apportée en réponse à la demande de certains utilisateurs qui avaient à manipuler de gros modèles, comme le schéma RDF pour P3P [155], constitué de 852 ressources, 1517 littéraux et 4346 propriétés, ce qui correspond en termes de glyphes XVTM à 6715 objets de type *VText*, 852 ellipses, 1517 rectangles et 4346 chemins constitués de courbes de Bézier (soit 13430 glyphes indépendants), prouvant ainsi que notre boîte à outils est capable de gérer un grand nombre d'objets graphiques tout en maintenant un taux de rafraîchissement acceptable. La mise à disposition publique de l'outil nous a permis de tester la robustesse de l'implémentation et d'expérimenter certains concepts offerts par la XVTM dans une application

réelle. Les retours informels fournis par la communauté des utilisateurs ont été positifs. IsaViz, dans sa version 1.1, a atteint un niveau de maturité assez élevé, et nous pouvons maintenant nous concentrer sur l'ajout des fonctionnalités évoquées ci-dessous.

Représentation et navigation

Nous n'avons pas effectué d'évaluation formelle de l'interface, mais le retour informel fourni par les utilisateurs de l'outil semble indiquer que le modèle de navigation (continuité perceptuelle fournie par les animations, modes de navigation) est apprécié. Les utilisateurs semblent plus critiques par rapport à la représentation du graphe, surtout lorsqu'ils ont à manipuler de gros modèles. GraphViz fournit des représentations de très bonne qualité compte tenu du fait qu'il s'agit d'un agencement automatique, mais les utilisateurs aimeraient une représentation du graphe plus avancée, qui tienne en quelque sorte compte de la sémantique du graphe, par exemple de manière à mettre mieux en évidence les éléments centraux du modèle.

Pour arriver à cette fin, nous envisageons, dans le cadre des nouvelles fonctionnalités à incorporer dans l'environnement, un mécanisme plus ou moins équivalent aux feuilles de style. L'utilisateur pourrait spécifier un ensemble de règles de présentation (couleur, forme du glyphe, épaisseur du trait) à appliquer aux nœuds et aux arcs du graphe en fonction par exemple du type de propriété ou de la classe à laquelle appartient la ressource. Comme nous l'avons vu dans le chapitre 3, cette différenciation visuelle des éléments du modèle permettrait à l'utilisateur d'appréhender plus facilement et plus rapidement le modèle. Des utilisateurs ont aussi suggéré l'utilisation d'agencements alternatifs groupant certains couples propriétés/valeurs associés à une ressource dans un tableau. Cette représentation faciliterait la lecture du modèle en effectuant des regroupements et en minimisant le nombre d'arcs dans le diagramme. Les littéraux et les ressources multimédia (images, vidéo) pourraient aussi être affichés sous leur forme naturelle plutôt que d'être symbolisés par une ellipse étiquetée par une URI.

Édition

Du point de vue de l'édition, nous envisageons dans un premier temps deux nouvelles fonctionnalités. La première proposerait à l'utilisateur une forme d'interaction contrainte plus poussée utilisant les informations du schéma RDF associé au modèle édité quand un tel schéma existe. Il serait par exemple possible d'utiliser les propriétés *rdfs:domain* et *rdfs:range* définies pour un type de propriété afin d'empêcher l'utilisateur de désigner, comme sujet ou objet des affirmations utilisant cette propriété, une ressource appartenant à une classe en dehors du domaine sur lequel est définie la propriété. Afin encore une fois de ne pas trop fortement contraindre l'interaction et ainsi frustrer l'utilisateur, cette fonctionnalité pourrait prendre la forme de simples suggestions faites dans un mode non intrusif.

La seconde fonctionnalité est un générateur de fragments de modèle. Quand un modèle contient beaucoup de fragments ayant la même structure, il est intéressant de pouvoir engendrer les parties statiques de cette structure automatiquement (à l'aide de *templates*) de manière à ce que l'utilisateur n'ait qu'à spécifier les parties variant d'une instance à l'autre. Les fonctions de couper/copier/coller visuelles sont un premier pas dans cette direction, mais nécessitent tout de même plus d'opérations d'édition que si l'utilisateur pouvait engendrer d'un simple clic un ensemble de ressources et de propriétés.

Environnement interactif pour la spécification de programmes VXT

Nous avons proposé dans le chapitre 4 un langage de programmation visuel pour la spécification de transformations de documents XML. Nous avons présenté les constructions du langage, et nous avons donné une définition formelle de sa syntaxe et de sa traduction vers le langage XSLT. Nous n'avons par contre pas développé les aspects interactifs du langage, c'est-à-dire l'édition, l'exécution et la mise au point des programmes VXT. Or, encore plus que pour les langages de programmation textuels, l'environnement de développement associé à un langage visuel représente une composante importante de la solution et est souvent intimement lié au langage lui-même.

Le code source de la plupart des langages textuels peut en effet être modifié avec n'importe quel éditeur de texte et ensuite compilé par une ligne de commande ou depuis un environnement de développement. Ce n'est pas le cas des programmes visuels, qui nécessitent souvent l'utilisation d'un éditeur spécifique. Il serait possible d'envisager conceptuellement l'édition de programmes visuels dans un éditeur de dessin vectoriel (voire d'images *bitmap* même si cela semble assez peu réaliste), c'est-à-dire n'ayant aucune connaissance de la sémantique du contenu des images. Mais, au-delà de la plus grande difficulté d'édition des programmes, il serait nécessaire de procéder à une analyse très poussée en plusieurs étapes de la représentation du programme afin de pouvoir l'exécuter. Il faudrait réaliser une analyse lexicale, puis syntaxique de l'image, nécessitant la définition d'une grammaire visuelle complète et d'algorithmes de reconnaissance des structures graphiques employées. Il s'agit là d'une méthode envisageable mais qui nécessite un effort très important de formalisation et d'implémentation et n'est possible que pour certains langages. En effet, même s'il existe des formalismes grammaticaux assez puissants pour modéliser des contraintes syntaxiques multidimensionnelles très complexes (par exemple les *Constraint Multiset Grammars* [118]), ces formalismes restent difficiles à manipuler et d'un point de vue plus pratique le coût d'exécution de l'analyse grammaticale est souvent élevé.

La relative simplicité de VXT, comparé par exemple à LabView, et le fait que nous avons déjà défini formellement sa syntaxe, en utilisant un formalisme relativement simple (les grammaires relationnelles de Wittenburg [235]), nous permettraient d'implémenter un analyseur de règles de transformation VXT représentées sous forme d'image ou de dessin vectoriel. Nous avons cependant préféré concevoir un environnement d'édition dédié à VXT dans la lignée des éditeurs spécialisés associés à LabView [11, 127] ou Prograph [65]. Ces éditeurs ont une certaine connaissance du langage, et construisent une représentation du programme en mémoire au fur et à mesure que l'utilisateur ajoute et modifie des composants. Grâce à cette connaissance de VXT par l'environnement, nous allons pouvoir définir un mode d'édition spécialisé réduisant la viscosité (voir chapitre 3) et capturant les contraintes syntaxiques définies par la grammaire visuelle directement au niveau de l'interaction, empêchant ainsi la création de règles mal formées, à la manière des éditeurs syntaxiques pour langages de programmation textuels (*syntax directed editors*) tels que le Cornell Program Synthesizer [207] et Alice Pascal [208]. La construction dynamique du programme en mémoire au court de l'édition (en rapport avec la *liveness* abordée dans l'état de l'art, chapitre 3) va quant à elle nous permettre de proposer des fonctionnalités d'aide à la mise au point, comme l'exécution des transformations depuis l'environnement et l'évaluation progressive de certaines parties des programmes.

7.1 Interface

La ligne directrice que nous avons suivie pour la conception de cette interface repose sur une approche différente du problème de représentation des transformations XML. Cette approche est basée sur la décomposition de l'espace de travail en fonction des opérations à accomplir et non pas en fonction des entités (hétérogènes) manipulées durant le processus de spécification, ce qui nous permet d'adapter spécifiquement la représentation en fonction des métaphores utilisées, comme nous le verrons dans la section 7.2.2.

La figure 7.1 est une capture d'écran de l'environnement VXT. La fenêtre principale, une vue XVTM appelée ici *VPMEs*, affiche sur deux couches indépendantes la structure source et l'ensemble des *VPMEs* (ici trois) de la transformation (comme nous l'avons vu dans le chapitre 4, les opérations de sélection et d'extraction ont été regroupées pour chaque règle en une seule expression représentant la notion de filtrage : la *VPME* : *Visual Pattern-Matching Expression*). Chaque règle de transformation est entièrement représentée dans sa propre fenêtre (par exemple *t#0*), en reprenant la *VPME* en partie gauche et en affichant la production en partie droite. Trois fenêtres regroupent les fonctions de recherche et de navigation dans la structure, la palette d'icônes pour la construction des *VPMEs* et des productions, et les propriétés (modifiables) du nœud sélectionné (fenêtres *VXT*, *Template* et *Node Properties*). Enfin, la dernière fenêtre (*Result*) est utilisée lors de la mise au point des transformations et affiche une vue structurale du résultat obtenu. Nous allons voir dans ce chapitre quelles sont les raisons qui nous ont poussé à choisir cette représentation peu orthodoxe des règles de transformation.

7.2 Aspects cognitifs

7.2.1 Réduction de l'effort mental de mémorisation des structures

Les environnements de développement intégrés comme XML Spy proposent souvent une vue graphique d'une instance de document à transformer sous la forme d'un arbre qui peut être développé (par exemple un *JTree* Java). Cette représentation graphique est très importante car elle permet à l'utilisateur d'avoir sous les yeux en permanence un exemple des structures qu'il doit manipuler au cours des transformations qu'il spécifie. Peu d'environnements proposent un équivalent pour les DTD ou les autres langages de schéma. Il n'est en effet pas possible de les représenter au moyen de simples arbres comme les *JTree*, et les représentations proposées sont complexes et gourmandes en espace d'affichage (*screen real-estate*). La majeure partie de cet espace devant être allouée à la représentation de la transformation, il n'est donc pas possible dans ces environnements de représenter simultanément la DTD (ou le schéma) et la transformation. Grâce à la représentation permanente d'une instance source dans son espace de travail, l'utilisateur n'a pas à maintenir en mémoire une représentation abstraite de la structure du document, ce qui contribue à la réduction de l'effort mental qu'il doit fournir. Cependant, la représentation graphique des instances est assez éloignée de la représentation (qui reste textuelle) de la feuille de transformation XSLT. Nous avons donc voulu dans VXT expérimenter une approche permettant de réduire encore plus l'effort mental à fournir, en tirant parti du fait que la transformation est elle-même exprimée visuellement.

Le premier moyen pour arriver à ce résultat est l'utilisation d'un formalisme de représentation unifié pour les instances de documents, les DTD, et les règles de transformation. Ce formalisme a été étudié

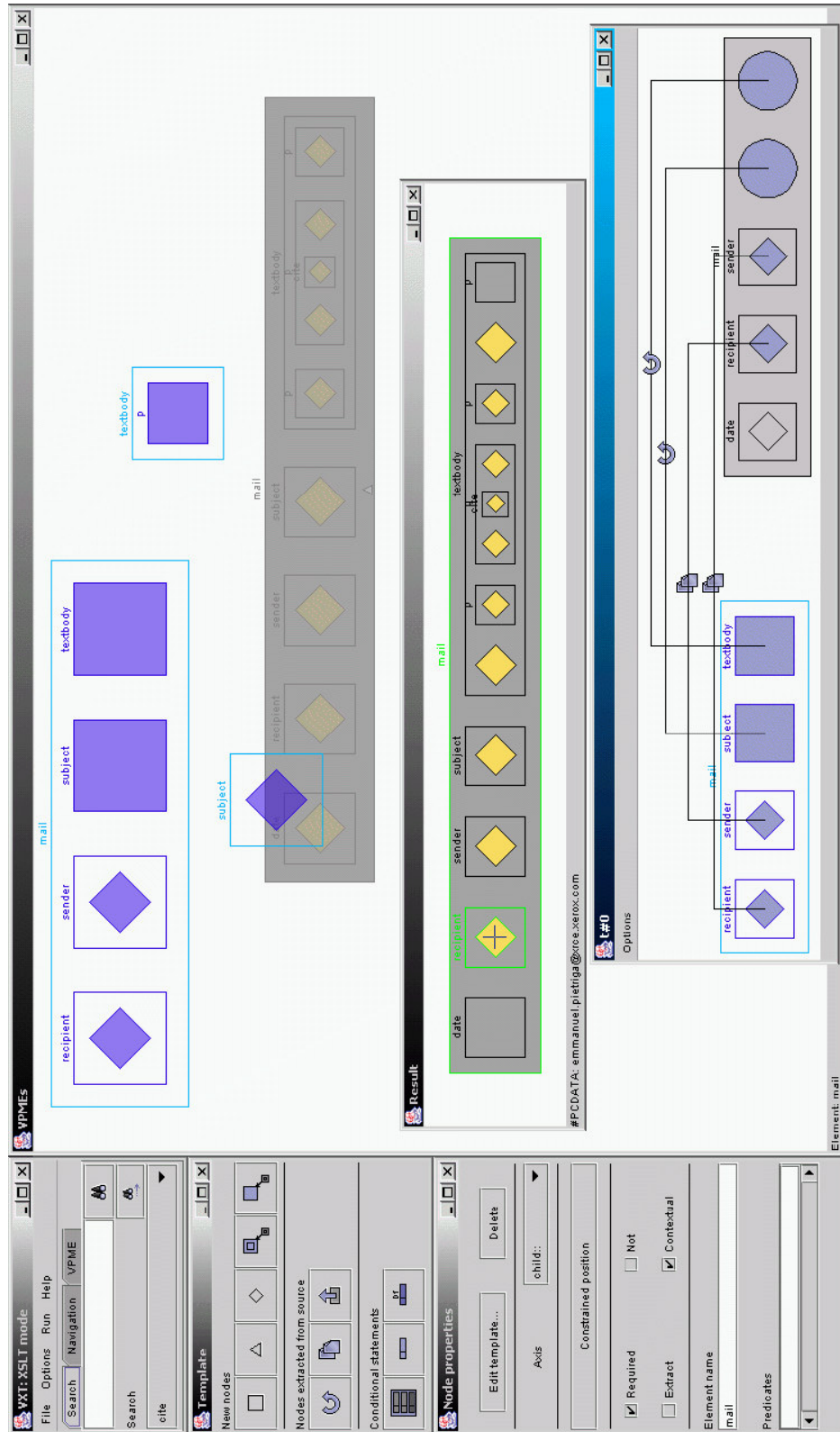


FIG. 7.1 : Visual XML Transformer : interface graphique

dans le chapitre 4. Comparé aux arbres précédents (*JTree*, etc.), il permet de réduire la distance entre la représentation des structures à transformer et les règles de transformation, et contribue ainsi à diminuer l'effort de l'utilisateur relatif à l'établissement de la correspondance entre structures à transformer et règles de transformation.

Le deuxième moyen est lié au formalisme précédent et à la technique de visualisation. La représentation des DTD dans notre formalisme est plus complexe mais relativement proche de celle des instances. Ainsi, la distance entre la représentation d'une DTD et celle d'un document XML instance de cette DTD, du point de vue de l'aspect visuel, semble plus faible¹ que dans le cas des formalismes tels que celui de Near & Far Designer [149]. Ceci nous permet de proposer à l'utilisateur la visualisation de documents sources mais aussi des DTD décrivant ces documents. Les deux types de structures sont visualisés en utilisant la variante des *treemaps* décrite dans le chapitre 4. La structure de graphe des DTD due à leur récursivité potentielle est cassée dans notre représentation afin de pouvoir représenter la DTD comme un arbre. Le problème de cette cassure dans la représentation statique est résolu en ne définissant le contenu d'un élément donné qu'une seule fois, lors de sa première occurrence. Ses autres occurrences sont alors vides² mais pointent dynamiquement sur la définition initiale, c'est-à-dire la première occurrence dans la structure.

Les DTD et instances de documents pouvant être de taille très importante (voir par exemple les DTD pour SVG, MathML ou XHTML, ou encore celle qui réunit les trois [154]), nous avons choisi d'utiliser les capacités de zoom continu de la XVTM pour leur représentation. Les fonctions de zoom sont rendues nécessaires par l'utilisation d'une variante des *treemaps*, qui, par nature, assignent des tailles très petites aux nœuds situés profondément dans la structure, rendant leur visualisation difficile voire impossible si l'environnement n'a pas de capacité d'agrandissement local. Mais combiné à des fonctions de navigation et de recherche faciles à utiliser, le zoom continu permet à l'utilisateur de se déplacer rapidement dans la structure sans effort. De plus, la combinaison d'une représentation à base de *treemap* avec des fonctions de zoom continu représente d'une certaine manière un moyen intuitif de filtrer l'information visuellement : là où dans un *JTree* Java l'utilisateur devrait manuellement développer et réduire une par une des parties de l'arbre pour n'afficher que le contenu des nœuds qui l'intéressent, il suffit dans VXT de changer l'altitude d'observation de la caméra. Par exemple, lors d'un zoom arrière, les nœuds très profonds disparaissent automatiquement mais la représentation révèle plus d'information sur les nœuds frères du nœud au centre de la vue, c'est-à-dire sur son contexte. Au contraire, un zoom avant aura tendance à développer le contenu du nœud au centre de la vue, le contexte de ce nœud disparaissant petit à petit (là aussi, les capacités de zoom *continu* sont très importantes car elles permettent d'ajuster de manière fine la région observée en fonction des besoins).

Enfin, le troisième moyen utilise les vues multi-couches de la XVTM (section 5.2.2) pour expérimenter une nouvelle manière d'utiliser l'espace de représentation. Sachant que nous désirons offrir à l'utilisateur

¹Nous basons cette affirmation sur notre observation des différents formalismes et sur des éléments plus concrets comme le fait que notre représentation a besoin d'un nombre plus limité d'objets graphiques additionnels pour les DTD par rapport aux instances (nous avons surtout essayé de jouer sur les attributs visuels des objets existants). Il s'agit néanmoins d'une évaluation subjective puisque nous n'avons pas défini de métrique permettant de mesurer cette distance.

²Ceci est rendu nécessaire du fait de la récursivité qui nous obligerait autrement à représenter une structure infinie. Notons que cela serait possible grâce aux fonctions de zoom dans le cadre d'une représentation dynamique.

la représentation simultanée de la structure source et des règles de transformation, nous avons tenté de les combiner dans le même espace sur des couches différentes (idée de calques (*layers*)). La fenêtre VPMEs de la figure 7.1 illustre notre approche : une instance de document XML *mail* est représentée dans la couche inférieure (structure en niveaux de gris), alors que la couche supérieure contient l'ensemble des VPMEs composant la transformation (structures bleues). Ainsi, au lieu d'assigner deux fenêtres différentes pour la représentation de la structure source et de la transformation, nous proposons une approche différente du problème de représentation des transformations : l'espace de travail n'est plus décomposé en fonction des entités manipulées, à savoir les documents sources et les règles de transformation, mais en fonction des opérations à accomplir, c'est-à-dire le filtrage d'une structure source (sélection et extraction de données) et la production d'un résultat. Nous allons maintenant voir comment cette représentation est mise à profit, au-delà de la réduction de l'effort mental de l'utilisateur, pour soutenir la métaphore des filtres visuels.

7.2.2 Métaphore des filtres visuels

Le modèle mental associé aux règles de transformation consiste en l'application sur un arbre de filtres dont la tâche est de sélectionner des parties de cet arbre en fonction de contraintes principalement structurales et d'extraire des données par rapport aux éléments sélectionnés. Ces données sont alors réorganisées et enrichies afin de produire les fragments d'une structure qui, une fois combinés, représentent l'arbre du document résultat de la transformation.

Nous avons essayé dans VXT d'utiliser des métaphores proches de ce modèle. La séparation de l'espace de travail en fonction des différentes opérations (filtrage et production) nous permet d'adapter la représentation pour ces métaphores. Nous nous concentrons ici sur la partie filtrage, spécifiée dans la fenêtre VPMEs (figure 7.1). Cette fenêtre est composée de deux couches dans lesquelles l'utilisateur peut naviguer de manière indépendante (i.e. un déplacement ou un zoom dans une couche n'entraîne pas de déplacement ou de zoom dans l'autre couche). La couche inférieure contient la représentation sous forme de *treemap* d'une instance de document source ou de la DTD correspondante. L'utilisateur peut aussi décider de ne rien afficher dans cette couche, puisque la présence d'une structure source n'est pas nécessaire pour la *spécification* de la transformation. La représentation de l'instance ou de la DTD dans la couche inférieure peut donc s'apparenter à un fond d'écran (*background*) dynamique qui assiste l'utilisateur dans sa tâche en l'affranchissant du poids cognitif lié à la représentation mentale de la structure à transformer.

Toutes les VPMEs (*Visual Pattern-Matching Expression*) composant la transformation sont regroupées dans la couche supérieure. Grâce à notre formalisme unifié pour la représentation des documents et des règles de transformation, les VPMEs sont visuellement proches des structures qu'elles sélectionnent, et même parfois identiques. Cette propriété, combinée à notre méthode d'affichage en couches superposées, invite alors l'utilisateur à considérer les VPMEs comme des filtres (ou masques) visuels qu'il applique sur une structure. Il peut en effet sélectionner une VPME particulière et la déplacer au-dessus de la structure source. Elle donne alors l'impression de s'adapter à certaines parties de la structure, qui correspondent à ce que la VPME va effectivement sélectionner dans le document (figure 7.2).

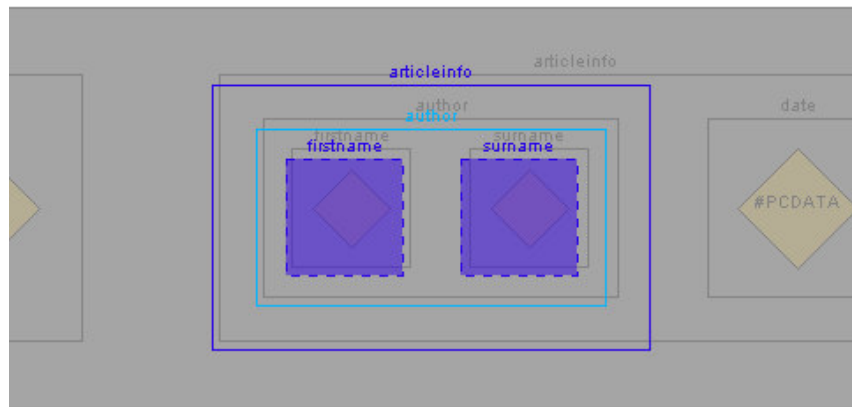


FIG. 7.2 : Filtrage visuel analogique : superposition d'un filtre et d'une instance

La représentation en couches superposées pose cependant le problème de l'interférence visuelle qui peut se produire entre les deux représentations (difficulté de différenciation, sur le plan visuel, des objets des différentes couches, voir section 3.1.2). A. Blackwell propose avec SWYN [26] un environnement dans lequel une expression régulière textuelle est représentée au-dessus du texte sur lequel elle est appliquée. L'expression régulière utilise des couleurs vives alors que le texte source est noir. Cette technique permet de différencier assez facilement les deux entités. Nous proposons ici de minimiser encore plus l'interférence visuelle en appliquant le principe des contrastes minimaux de Tufte [211] : les objets de la couche active sont représentés au moyen de couleurs vives très contrastées alors que ceux de la couche inactive (à l'arrière-plan) utilisent des tons de gris peu contrastés. La figure 7.2 illustre ce principe : lors de l'édition des filtres, les VPMEs (dans la couche active) utilisent des tons de bleu vifs, alors que le document source est représenté par des niveaux de gris discernables mais assez peu contrastés pour ne pas interférer avec les structures au premier plan (*foreground*). Si l'utilisateur passe en mode de navigation dans le document source, celui-ci est rendu avec des couleurs plus contrastées, les VPMEs étant soit cachées soit rendues avec un contraste minimal (elles restent cependant dans ce cas dans la couche supérieure). Cette technique contribue de manière importante à la minimisation du phénomène d'interférence visuelle. D'autre part, la possibilité de naviguer dans les couches de manière indépendante et à tout moment³ offre une autre manière de différencier les structures appartenant aux deux couches.

7.3 Édition des règles

L'édition des VPMEs, c'est-à-dire des parties gauches de règles VXT, a lieu dans la couche supérieure de la fenêtre VPMEs décrite précédemment. Les productions, c'est-à-dire les parties droites de ces règles, ne sont quant à elles ni représentées ni spécifiées dans cette fenêtre, puisque celle-ci est dédiée au filtrage.

³Les fonctions de déplacement dans l'espace, jugées importantes, sont à tout moment accessibles au moyen du bouton droit de la souris.

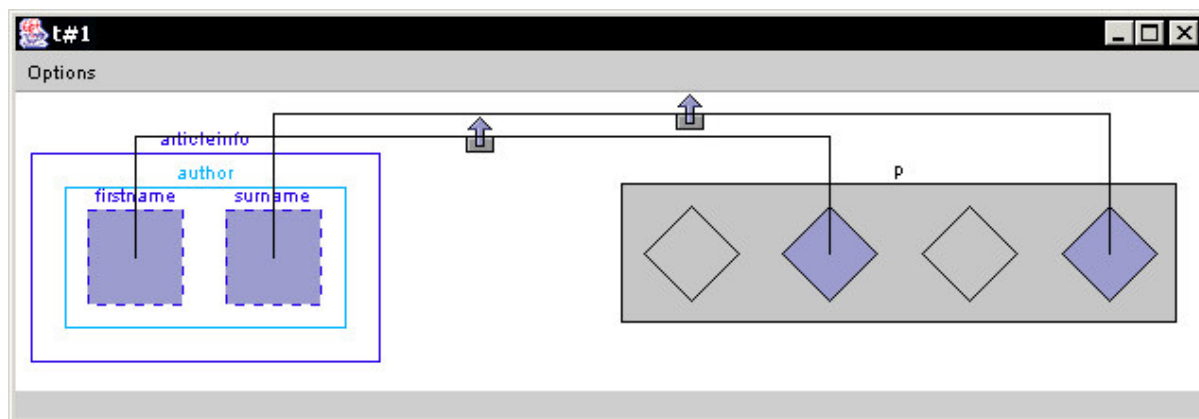


FIG. 7.3 : Règle de transformation VXT

Nous avons essayé de tirer parti de l'indépendance des règles du point de vue de leur spécification⁴ pour proposer une modularisation de la transformation dans l'environnement. Ainsi, chaque règle est représentée dans sa propre fenêtre, qui peut être affichée ou cachée par l'utilisateur suivant ses besoins. Chaque fenêtre contient une règle complète, c'est-à-dire une copie synchronisée de la *VPME* (qui a été spécifiée dans la fenêtre *VPMEs*) en partie gauche, et la production en partie droite (voir la figure 7.3 et la fenêtre *t#0* de la figure 7.1). La représentation de la *VPME* dans la fenêtre permet d'établir le lien entre les données extraites par le filtre et la production. Les figures 4.11 à 4.15 du chapitre 4 donnent d'autres exemples de règles VXT telles qu'elles apparaissent dans les fenêtres de l'environnement.

7.3.1 Construction

Les *VPMEs* et les productions se construisent de manière similaire. L'utilisateur choisit un type d'objet dans la palette d'icônes (figure 7.1) et clique ensuite à l'endroit où il veut insérer le nouveau nœud. Ensuite, suivant le type de nœud, des informations supplémentaires lui sont demandées, comme un nom quand il s'agit d'un élément. Les *VPMEs* sont construites dans la fenêtre *VPMEs*, les productions dans la fenêtre associée à chaque règle, de même que les opérations de transformation qui sont spécifiées en sélectionnant dans la palette d'icônes un type d'opération, puis en sélectionnant dans l'ordre un nœud translucide de la *VPME* et un emplacement dans la production (un nouvel objet graphique représentant le type inféré du résultat de l'opération est alors ajouté à cet endroit).

L'environnement propose aussi des fonctions de génération automatique de *VPME*. Au lieu de spécifier nœud par nœud une *VPME*, l'utilisateur peut sélectionner un nœud dans l'instance de document source représentée en arrière-plan. VXT génère alors automatiquement la *VPME* qui sélectionne ce nœud en considérant les fils du nœud dans l'instance comme des prédicats existentiels. De nouveaux nœuds peuvent ensuite être insérés ou supprimés, et il est possible de modifier leurs propriétés .

⁴Comme en XSLT, les règles peuvent s'appeler entre elles grâce à l'opération *apply-rules*. Mais nous considérons qu'elles sont indépendantes du point de vue de la spécification puisqu'il n'y a pas de lien explicite à établir entre elles.

Étant donné la grande variété des nœuds, et de manière à ne pas trop encombrer l'interface, seuls les types de nœud les plus courants ont leur propre constructeur dans la palette d'icônes. Les autres types peuvent être obtenus en modifiant les propriétés des nœuds existants par l'intermédiaire de la fenêtre *Node properties*. Nous avons cependant essayé de faciliter la tâche de spécification en assignant des propriétés par défaut en fonction du contexte. Ainsi, avec le même constructeur, les propriétés par défaut d'un nœud seront différentes suivant qu'il est inséré dans une *VPME* existante ou qu'il est le premier nœud d'une nouvelle *VPME* : dans le premier cas, l'environnement créera un simple prédicat, alors que dans le deuxième cas il créera un nœud contextuel.

7.3.2 Interaction contrainte

Avant d'exécuter les transformations ou de les exporter vers XSLT ou Circus, l'environnement doit faire un certain nombre de vérifications pour s'assurer que les règles de transformation spécifiées par l'utilisateur sont correctes, ceci afin de garantir que les programmes exportés seront eux-mêmes corrects au niveau syntaxique. Pour cela, une solution consiste à examiner les *VPMEs* après leur création en leur appliquant les fonctions de vérification définies dans le chapitre 4, par exemple au moment de l'exécution de la transformation, ou bien suite à une demande explicite de la part de l'utilisateur. Mais il semble plus intéressant de faire remonter les erreurs de spécification à l'utilisateur au plus tôt dans le but d'améliorer sa productivité.

Pour cela, nous utilisons la notion d'interaction contrainte introduite dans la section 5.2.4. L'environnement d'édition empêche l'utilisateur de créer des *VPMEs* incorrectes en contraignant ses actions. Par exemple, il est impossible de placer une instruction d'extraction comme descendant d'une condition de non existence. De même, le fait de déclarer un nœud comme étant le nœud contextuel enlève automatiquement cette propriété à tout autre nœud de la *VPME* afin de garantir l'existence d'un unique nœud contextuel dans chaque *VPME*. Nous avons préféré la solution d'un environnement d'édition imposant des contraintes sur les actions, car cette méthode permet un retour utilisateur plus précis en cas d'erreur⁵ et prévient certaines erreurs plutôt que de les signaler dans un futur plus ou moins proche. Dans la pratique, nous n'appliquons donc pas les fonctions de vérification VF_i introduites lors de la définition formelle du langage (section 4.4), puisque les contraintes qu'elles imposent sur la structure des *VPMEs* sont capturées par les contraintes sur l'interaction utilisateur.

7.4 Mise au point des programmes

7.4.1 Exécution des transformations

Les programmes VXT peuvent être exécutés directement dans l'environnement d'édition, le résultat de la transformation étant automatiquement présenté dans une nouvelle fenêtre. L'utilisateur peut ainsi rapidement voir les conséquences des modifications qu'il apporte au programme sur le résultat, toujours représenté sous forme de *treemap* (voir la figure 7.1, fenêtre *Result*). Bien entendu, il est aussi possible d'accéder à des représentations textuelles des documents source et cible de la transformation, ainsi qu'à un journal contenant les messages associés aux erreurs survenues lors de l'exécution.

⁵Les actions d'édition interdites sont sans effet sur la *VPME* mais font l'objet d'un retour à l'utilisateur sous forme de message non intrusif expliquant pourquoi cette action est interdite.

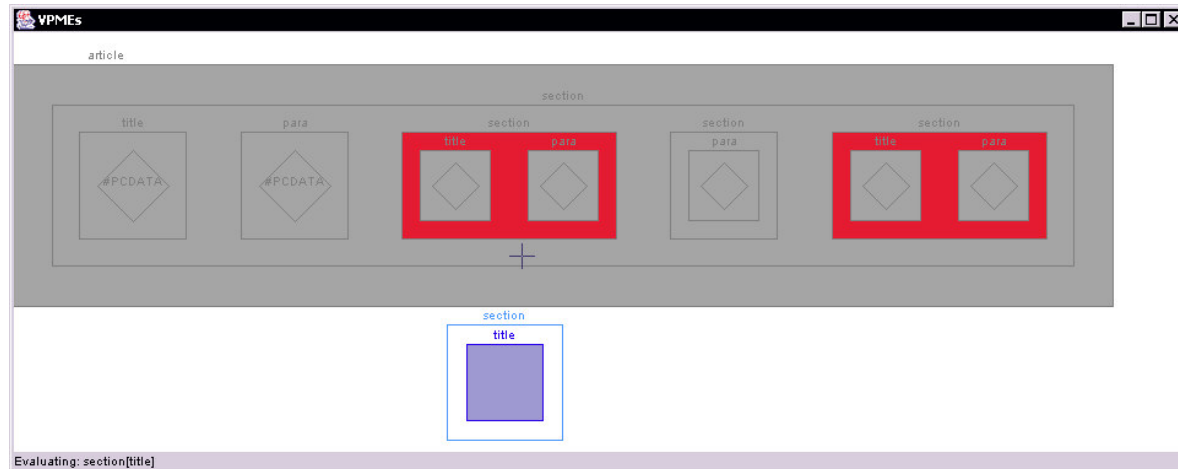


FIG. 7.4 : Évaluation progressive d'une VPME

Cependant, il est aussi intéressant de pouvoir exporter les programmes VXT vers d'autres langages, qui permettront d'exécuter ces transformations dans d'autres contextes (par une ligne de commande, dans un fichier de script, sur un *proxy* ou un serveur, ...). Nous proposons pour cela deux fonctions d'exportation, l'une qui génère des feuilles de transformation XSLT, l'autre du code source Circus qu'il faut ensuite compiler. Comme IsaViz, VXT offre son propre vocabulaire XML pour sauvegarder et restaurer les programmes d'une session à l'autre. Cette fonctionnalité est d'autant plus utile qu'il n'existe pas de fonction d'importation de code Circus ou de feuille de transformation XSLT : les deux langages étant plus expressifs que VXT, nous ne pourrions pas assurer l'interprétation de n'importe quel programme écrit dans l'un de ces langages. Il serait possible de créer un interpréteur restreint faisant l'hypothèse que le code importé ne contient que des constructions supportées par VXT, mais cette solution ne nous permettrait par exemple pas de restaurer l'information géométrique associée au programme (placement de la caméra et des *VPMEs* dans l'espace, etc.).

7.4.2 Évaluation progressive

La possibilité d'exécuter les transformations directement depuis l'environnement représente une première aide à la mise au point puisqu'elle permet d'effectuer des tests rapidement et sans avoir à changer d'environnement. Cette fonctionnalité nécessite cependant l'exécution de la transformation complète, ce qui peut être lourd et fastidieux. Nous proposons dans l'environnement un mécanisme d'évaluation progressive qui permet de tester rapidement et sans exécuter la transformation de certains composants, à savoir les *VPMEs*. Ces dernières, de même que les expressions XPath et les filtres Circus, sont parfois complexes à comprendre et il n'est pas évident de prédire les nœuds du document source qu'elles sélectionneront effectivement. En utilisant le mécanisme d'évaluation progressive, l'utilisateur peut tester à tout instant une *VPME* sur l'instance de document XML chargée dans l'environnement. Pour cela, il suffit de sélectionner une *VPME* et ensuite de déplacer le curseur de la souris dans la structure source à l'arrière-plan. L'environnement colorera alors en rouge les éléments sélectionnés par la *VPME* dans le contexte du nœud où se trouve le curseur souris.

La figure 7.4 illustre l'évaluation progressive sur un exemple très simple de *VPME* sélectionnant les éléments `section` qui ont au moins un fils `title`. Ici, l'évaluation se passe dans le contexte du nœud `section` fils de `article`, dans lequel se trouve le curseur souris. Nous voyons que la *VPME* sélectionne deux des trois éléments `section`, le second étant mis de côté car il ne contient pas de fils `title`. L'évaluation progressive est surtout intéressante quand l'utilisateur doit manipuler des *VPMEs* complexes. Elle pourra aussi servir à l'utilisateur novice désirant se familiariser avec le mécanisme de filtrage VXT.

7.5 Conclusion et perspectives

Nous avons présenté dans ce chapitre l'environnement de développement associé à VXT implémenté en Java et reposant sur la XVTM. Cet environnement offre sa propre décomposition de la représentation des programmes de transformation, en fonction des tâches à accomplir, ceci afin de fournir des représentations visuelles adaptées aux métaphores utilisées (filtres visuels). Il permet l'édition des programmes VXT et offre aussi des fonctionnalités assistant l'utilisateur dans sa tâche, comme l'interaction contrainte, la génération automatique et l'évaluation progressive des *VPMEs*.

Une évaluation partielle du langage a été effectuée dans le cadre de l'implémentation du prototype et de la création de transformations. Cette évaluation, qui ne résulte que de notre propre expérimentation et présente par conséquent un caractère subjectif, s'est principalement concentrée sur l'expressivité du langage (voir chapitre 4); nous n'avons pas encore mené d'étude sur les aspects cognitifs et fonctionnels liés à l'environnement. Ce sont en effet des points plus subjectifs qu'il est difficile d'évaluer en dehors d'un cadre d'expérimentation rigoureux, et qui nécessitent d'entreprendre une évaluation plus approfondie à l'aide de personnes extérieures et suivant une certaine méthodologie. Même s'il existe des cadres de travail fournissant des méthodes pour effectuer correctement de telles expérimentations [109], ce type d'étude est généralement mené par des équipes ayant une certaine expertise dans ce domaine et qui n'était pas immédiatement disponible dans notre cadre de travail. Cette étude nous permettrait par exemple d'évaluer la lisibilité des structures représentées sur des couches superposées et, d'une manière générale, la pertinence de la métaphore des filtres visuels : ceux-ci sont-ils effectivement perçus en tant que tels par l'utilisateur ? cette métaphore et les fonctionnalités associées (adaptation des *VPMEs* aux structures sélectionnées) sont-elles considérées utiles ? Elle permettrait aussi d'apprécier, au-delà de son expressivité, la facilité de compréhension du formalisme de représentation proposé par le langage par un utilisateur novice. Par exemple, nous avons vu dans la section «Sémantique des expressions» du chapitre 4 qu'il existe des subtilités concernant certaines constructions visuelles (rôle central du nœud contextuel et interprétation des axes par rapport à ce nœud); nous pensons avoir choisi la meilleure option en ce qui concerne l'expressivité du langage et le caractère intuitif de la représentation, mais une évaluation impliquant des personnes n'ayant jamais utilisé VXT permettrait de valider ce choix.

Évolutions

L'environnement est capable d'exécuter les programmes VXT mais aussi de les exporter vers deux langages textuels : XSLT et Circus v1.0. Idéalement, nous aurions voulu ne proposer qu'un seul mode d'édition, faisant ainsi complètement abstraction des langages d'exportation. La différence d'expressivité entre Circus v1.0 et XSLT nous oblige malheureusement à proposer pour l'instant deux modes d'édition,

qui imposent certaines contraintes en fonction du langage cible choisi⁶. Cependant, Circus ayant beaucoup évolué ces derniers mois, il serait intéressant d'adapter VXT en fonction du nouveau modèle de données XML de la version 2 du langage. Ce modèle, qui gère les références, permettrait d'atteindre la même expressivité que XSLT du point de vue des expressions de filtrage et ainsi de proposer un mode d'édition unique, toute *VPME* du mode XSLT devenant exprimable en Circus. Il faudrait tout de même dans ce cas tenir compte des prédicats textuels que l'utilisateur a la possibilité d'ajouter aux nœuds des *VPMEs* ; ceux-ci sont en effet spécifiés pour l'instant en utilisant la syntaxe du langage cible et insérés dans la sérialisation tels quels. Un mode d'édition unique nécessiterait de pouvoir convertir ces prédicats d'un langage à l'autre, en fonction du langage cible choisi, ou bien de définir une syntaxe unifiée, cette solution ayant l'inconvénient d'obliger l'utilisateur à apprendre une nouvelle syntaxe mais étant plus facile à implémenter.

Une autre amélioration concerne les fonctions de navigation dans le document source et dans l'espace des *VPMEs*. La représentation à base de *treemap* et de zoom continu ne permet pas à l'utilisateur de conserver une vue d'ensemble de la structure et peut ainsi le désorienter, sachant qu'il est très simple de revenir à une vue globale de manière automatique et de retourner à la dernière région plus détaillée observée. Ce va-et-vient peut cependant être frustrant, et nous avons pensé ajouter une vue secondaire représentant la structure de manière globale ainsi qu'un rectangle délimitant la région couramment observée par la vue principale, à la manière des vues radar décrites dans le chapitre 3. Ces vues pourraient aussi bénéficier des techniques déformantes comme les vues hyperboliques permettant de combiner focus et contexte dans la même représentation (leur intérêt par rapport aux *treemaps* reste cependant à étudier, et il faudrait sans doute adapter la méthode).

Enfin, la dernière amélioration prévue mais pas encore implémentée concerne l'évaluation progressive des *VPMEs* et la métaphore des filtres visuels. Pour l'instant, lors du test d'une *VPME* sur une structure source, l'utilisateur déplace seulement le curseur de la souris. Afin que l'utilisateur perçoive encore plus les *VPMEs* comme des filtres visuels, nous voulons lui proposer de déplacer la *VPME* elle-même plutôt que la croix représentant le curseur. La *VPME* s'adaptera alors visuellement (sur le plan géométrique) aux nœuds qu'elle sélectionne dans la structure source, rappelant ainsi l'idée de filtre visuel. La *VPME* n'exprimant pas de contrainte sur l'ordonnement des prédicats, ceux-ci pourront être réordonnés dans la *VPME* afin de s'adapter au nœud source sélectionné en fonction de l'ordre de ses fils. Enfin, cette adaptation visuelle pourra éventuellement servir à identifier finement les données extraites par les instructions d'extraction de la *VPME* en mettant en correspondance les nœuds translucides de cette dernière avec les nœuds de la structure source. D'un point de vue technique, cette amélioration peut poser un problème de performances : l'évaluation de la *VPME* par rapport au nœud indiqué par l'utilisateur doit se faire rapidement, de manière à fournir un retour visuel immédiat. Nous utilisons pour l'instant une technique d'évaluation locale performante proposée par le moteur de transformation sous-jacent. Mais dans le cas d'une adaptation (sur le plan visuel) de la *VPME* au nœud sélectionné, il faudra aussi évaluer les instructions d'extraction de la *VPME* par rapport au contenu de ce nœud, puis en fonction des résultats de ces multiples évaluations modifier et peut-être recombinaison la *VPME* de manière à ce qu'elle s'adapte à la

⁶La solution consistant à ne proposer que les fonctionnalités communes aux deux langages n'est pas satisfaisante car elle limite significativement l'expressivité de VXT.

structure source. Suivant la quantité et la complexité des instructions d'extraction et suivant la taille du contenu de l'élément sélectionné, ces opérations peuvent avoir un coût non négligeable.

Conclusion

8.1 Rappel des objectifs

Le format de documents structurés XML est devenu un standard utilisé par de nombreuses applications pour représenter, stocker et échanger leurs documents et leurs données. Cette popularité, associée à l'évolution du World Wide Web, est due à la simplification des traitements documentaires apportée par le partage de la même syntaxe, des mêmes principes de structuration et des mêmes mécanismes entre tous les langages fondés sur XML. Ces caractéristiques communes rendent possible l'emploi d'outils génériques durant plusieurs phases du processus de traitement des documents, comme l'analyse syntaxique et les transformations. Les langages fondés sur XML sont utilisés pour décrire des documents et représenter des données dans de nombreux domaines ; les documents et les données ont souvent besoin d'être combinés ou convertis d'un langage à un autre, opérations effectuées par l'intermédiaire de transformations, qui sont rendues nécessaires par les besoins suivants :

- échanger des données entre applications hétérogènes par la conversion des documents d'un langage à un autre,
- produire des documents formatés à partir de documents structurés de manière logique et ne contenant pas d'information de présentation,
- filtrer ou réorganiser la structure et le contenu d'un document en fonction des besoins de l'utilisateur, mais aussi combiner des fragments de documents utilisant différents langages,
- extraire automatiquement des informations contenues dans des documents afin de créer des méta-données décrivant ces documents.

Il existe différents types de langages et d'outils pour la création et la transformation de documents XML, positionnés à différents niveaux d'abstraction et offrant par conséquent des niveaux d'expressivité et de difficulté d'utilisation variables. La plupart de ces langages sont cependant textuels, et même si des outils tels que les environnements de développement intégrés offrent une interface graphique et des fonctionnalités aidant le programmeur dans la spécification des transformations, celles-ci restent spécifiées au moyen du langage textuel sous-jacent, nécessitant de la part de l'utilisateur une bonne connaissance de ce langage.

L'objectif de ce travail de thèse était d'aller plus loin dans la direction des environnements de développement intégrés en proposant des outils pour le traitement de documents structurés qui tirent pleinement parti des capacités de représentation et d'interaction des environnements et des langages de programmation visuels, en fournissant des représentations et des méthodes de spécification visuelles de la structure des documents, des méta-données associées, et des programmes de transformation de ces documents.

8.2 Rappel du travail réalisé

8.2.1 Démarche suivie

Ce travail s'est déroulé en trois phases. Tout d'abord deux études théoriques menées en parallèle, l'une portant sur les solutions existantes pour la création et la transformation des documents XML et des technologies associées, l'autre portant sur les langages de programmation visuels. Ces deux études, effectuées dans le but de se familiariser avec les deux domaines, ont été accompagnées de phases d'expérimentation ponctuelles, comme la création de la transformation MathMLc2p (voir chapitre 2) dans

le cadre d'une comparaison entre les langages XSLT et Circus, la participation à l'implémentation d'un analyseur syntaxique XML en Circus, et de petits prototypes destinés à évaluer les performances des différents environnements et langages de programmation dans le cadre de la réalisation, par la suite, d'une boîte à outils pour la conception d'interfaces graphiques zoomables.

La deuxième phase a consisté principalement en la définition théorique d'un langage de programmation visuel spécialisé dans la transformation de documents XML, en son étude formelle et en l'étude des fonctionnalités à intégrer dans l'environnement de développement associé et devant supporter (c'est-à-dire exploiter) les caractéristiques du langage pour proposer des métaphores adaptées et intuitives pour le programmeur.

Enfin, la troisième phase a débuté par l'implémentation des fonctionnalités de base de la XVTM, la boîte à outils pour la conception d'interfaces graphiques zoomables. Elle s'est poursuivie par l'implémentation de l'environnement visuel associé à VXT, menée en parallèle avec l'ajout de nouvelles fonctionnalités à la boîte à outils. La conception et l'implémentation de l'éditeur visuel pour RDF a été effectuée à la fin de cette phase, alors que la XVTM avait atteint un niveau de maturité assez élevé, même s'il a fallu lui apporter quelques modifications mineures, notamment pour qu'elle permette l'utilisation de courbes cubiques et quadratiques.

Les deuxième et troisième phases ont pendant un certain temps été menées en parallèle, certains problèmes d'ordre théorique, au niveau de l'expressivité du langage ou de l'ambiguïté de certaines constructions visuelles, n'apparaissant qu'au moment de l'implémentation du prototype et de l'expérimentation.

8.2.2 Résultats théoriques

L'étude des techniques de transformation de documents structurés et des environnements de développement intégrés pour XML a montré que l'usage fait des capacités des environnements visuels, du point de vue de la représentation et de l'interaction avec les structures et les programmes, était limité. Elle a permis d'identifier certaines fonctionnalités importantes, et d'évaluer le degré d'expressivité nécessaire pour avoir un langage intéressant ne se limitant pas à des transformations simples et peu pertinentes. L'étude des techniques de visualisation de données et des langages de programmation visuels existants a quant à elle permis d'identifier les avantages de ces approches, mais aussi les limitations et les problèmes couramment rencontrés lors de leur emploi. Ainsi, la conception des outils proposés dans cette thèse se base sur l'étude des langages existants mais aussi sur les recommandations données dans le cadre d'études telles que les dimensions cognitives de T. Green [111].

La contribution principale de cette thèse, sur le plan théorique, est la conception d'un langage de programmation visuel spécialisé dans la transformation de documents XML. Certains choix effectués par rapport au langage et à son environnement ont été guidés par les études précédentes :

- le choix de concevoir un langage de programmation spécialisé (à opposer à un langage généraliste), permettant de proposer des métaphores et des constructions programmatiques bien adaptées au domaine du problème (les transformations de documents XML) ;

- l’emploi d’une interface en 2,5 dimensions, de représentations à base de *treemaps* et de mécanismes facilitant la navigation ; ceci afin de permettre la visualisation de documents, classes de documents et programmes de transformation de taille importante (problème de la résistance au facteur d’échelle) ;
- l’importance accordée aux fonctionnalités associées à l’environnement d’édition des programmes visuels, qui représente à notre avis une composante très importante de toute solution de programmation visuelle. Il ne s’agit pas simplement des fonctionnalités propres à l’édition, qui permettent par exemple de réduire la viscosité, mais aussi des choix de représentation (séparation de l’espace de travail en fonction des tâches à réaliser, superposition des *VPMEs* au-dessus des structures sources) et des aides à la mise au point.

La solution proposée est un langage de programmation visuel reposant sur une étude formelle qui a permis de définir précisément sa syntaxe au moyen d’une grammaire visuelle et d’établir les propriétés de complétude et de correction syntaxique de la fonction de traduction des programmes VXT en feuilles de transformations XSLT. Cette étude formelle présente à notre sens un grand intérêt, puisqu’au-delà des bases théoriques solides qu’elle établit, sa réalisation et sa prise en compte dans la conception du langage fait de VXT un exemple d’utilisation de techniques appartenant à la fois aux branches théorique et pratique de la recherche dans le domaine des langages de programmation visuels. VXT représente donc un lien entre ces deux branches, qui semblent relativement cloisonnées, alors que des interactions plus développées leurs seraient sans doute bénéfiques. VXT offre une solution de transformation complète, reposant à la fois sur un langage dont l’expressivité permet de spécifier des transformations relativement complexes et sur un environnement de développement fournissant des métaphores et des fonctionnalités destinées à réduire la charge cognitive de l’utilisateur et à l’assister au niveau de la spécification et de la mise au point des transformations.

Une évaluation partielle du langage et des fonctionnalités de l’environnement a été effectuée dans le cadre de l’implémentation et du test du prototype et par la création de transformations. Il ne s’agit cependant pas d’une évaluation objective, puisqu’elle ne résulte que de notre propre expérimentation et se trouve donc biaisée par notre bonne connaissance du langage ainsi que des métaphores et des fonctionnalités proposées par l’environnement. Cette évaluation nous a donc permis d’apprécier la puissance expressive de VXT, mais nous n’avons pas encore pu tirer de conclusion quant aux aspects cognitifs et fonctionnels liés à l’environnement. Ce sont en effet des points plus subjectifs qu’il est difficile d’évaluer en dehors d’un cadre d’expérimentation rigoureux. Ainsi, il serait intéressant d’entreprendre une évaluation plus approfondie de ces aspects. Cette étude nous permettrait par exemple d’évaluer la lisibilité des structures représentées sur des couches superposées et, d’une manière générale, la pertinence de la métaphore des filtres visuels (est-elle effectivement perçue par les utilisateurs ?, la considèrent-ils utile ?). Elle permettrait aussi d’apprécier, au-delà de son expressivité, la facilité de compréhension du formalisme de représentation proposé par le langage par un utilisateur novice (la sémantique des constructions du langage, liée aux choix de représentation, est-elle toujours bien perçue ou bien au contraire les utilisateurs ont-ils parfois des difficultés d’interprétation ?). Ce type d’étude nécessite cependant une certaine expertise qui n’était pas disponible dans notre cadre de travail, et nous avons préféré concentrer nos efforts sur l’étude formelle du langage.

8.2.3 Résultats pratiques

Nous avons apporté au cours de ce travail une contribution sur le plan pratique à travers trois réalisations principales, sous la forme d'une bibliothèque et d'applications écrites en Java, qui représentent environ 50000 lignes de code.

Xerox Visual Transformation Machine (XVTM). La XVTM est une boîte à outils pour la conception d'interfaces graphiques zoomables. Elle est plus spécialement dédiée à la création d'éditeurs visuels ayant à manipuler de grandes quantités d'objets graphiques pouvant avoir des formes complexes. Il peut s'agir par exemple d'éditeurs de graphes ou d'environnements d'édition et d'exécution pour langages de programmation visuels. La XVTM a atteint un bon niveau de robustesse et offre un nombre important de fonctionnalités qui la rendent utilisable dans le contexte de projets de taille conséquente. Elle est déjà utilisée dans quatre applications : VXT, IsaViz, Panoramix (un outil de visualisation et de configuration pour la plateforme *middleware* CLF [7]) et l'outil de visualisation de machines à états finis de l'équipe *Content Analysis* [93]. Elle est aussi considérée dans le cadre de la réalisation d'un logiciel dans le domaine de la musique acousmatique [13], dont la conception devrait débiter à l'automne 2002.

Visual XML Transformer (VXT). La deuxième contribution pratique est un environnement de développement intégré pour l'édition, la mise au point et l'exécution de programmes VXT. Cette application est basée sur la boîte à outils précédente et implémente la majorité des fonctionnalités prévues dans l'étude théorique. Le prototype nous a permis de créer un certain nombre de transformations, de tester la fonction de traduction de VXT vers XSLT et il pourra être utilisé dans le cadre d'une évaluation plus rigoureuse du langage et des fonctionnalités proposées dans l'environnement.

IsaViz. Cet éditeur visuel de méta-données RDF représente la dernière contribution pratique de notre travail. Distribuée publiquement par le W3C depuis mars 2002, cette application rencontre un certain succès [191, 198] et va continuer d'évoluer pour intégrer de nouvelles méthodes de représentation des modèles RDF (voir la section Perspectives). Dans sa version actuelle, l'environnement semble offrir un bon niveau de stabilité et les fonctionnalités offertes, surtout au niveau de la représentation et de la navigation dans les graphes ont été bien accueillies (fonctions de déplacement, capacités de zoom, continuité perceptuelle, etc. qui utilisent les capacités de la XVTM). Aucune évaluation n'a été effectuée, mais une telle étude ne semble pas essentielle, étant donné la nature moins exploratoire d'IsaViz comparé à VXT. Les remarques informelles des utilisateurs nous permettent cependant de considérer notre approche comme valide et de continuer d'explorer cette voie.

8.3 Perspectives

8.3.1 Développement d'interfaces graphiques

Bibliothèque de glyphes. La bibliothèque de glyphes XVTM peut être étendue à volonté, la seule contrainte à respecter par les classes modélisant des glyphes étant d'hériter de la classe abstraite *Glyph*. Un très grand nombre d'objets graphiques est déjà capturé par les classes existantes. Nous aimerions cependant ajouter la possibilité de créer des glyphes composites, c'est-à-dire des glyphes (abstraits) composés d'une hiérarchie de glyphes. La difficulté réside dans la gestion des contraintes associées aux

composants mais aussi dans l'implémentation d'un algorithme d'animation générique et performant de manière à conserver le polymorphisme des opérateurs de transformation visuelle (translation, rotation, etc.).

Performances. Le niveau de performances atteint par la XVTM, en terme de taux de rafraîchissement des vues, est satisfaisant dans la plupart des cas. Il existe cependant une situation dans laquelle les performances peuvent se dégrader de manière assez importante : lorsqu'il faut afficher, même partiellement, un très grand nombre d'objets de type *VPath*¹ (chemins constitués de segments et de courbes quadratiques ou cubiques). Nous avons décrit dans la section 5.3 l'algorithme de *clipping* associé à ces objets, qui permet déjà d'améliorer sensiblement les performances. Nous envisageons d'implémenter un nouvel algorithme qui permettra une sélection plus fine de fragments des chemins à afficher en fonction des résultats de la phase de *clipping*. Les bénéfices de cette technique ne sont cependant pas assurés, et devront être validés par des mesures de performances.

8.3.2 Programmation visuelle de transformations de documents XML

Extension du langage. VXT permet de spécifier des transformations assez complexes, mais il serait intéressant d'intégrer certaines des constructions programmatiques manquantes proposées par XSLT 1.0, de manière à lui donner encore plus d'expressivité. À plus long terme, il faudrait prendre en compte les évolutions récentes des schémas et des langages de transformation et proposer en conséquence une extension du langage visuel. Nous n'avons pas encore étudié en détails les nouvelles constructions de XSLT 2.0 [83] et de XPath 2.0 [18], qui ne sont pour l'instant que des travaux en cours (au stade de *working drafts*). Mais une des extensions possibles pourrait être de supporter des contraintes de sélection basées sur les types de données proposés par les *XML Schema* et repris dans XPath 2.0. Cette extension est liée à celle du formalisme de représentation des instances et des classes de documents, qui devrait quant à lui évoluer pour supporter au moins partiellement des langages de schéma autres que les DTD, comme par exemple les *XML Schema*. Les nouveaux concepts proposés par ces derniers sont cependant très riches et très complexes, et il n'est pas évident de trouver un formalisme visuel capable de capturer cette complexité tout en conservant les propriétés que nous avons tenté de donner à notre proposition, c'est-à-dire une relative simplicité des constructions visuelles et un formalisme unifié pour la représentation des instances de documents, des classes de documents, et des règles de transformation.

Métaphore des filtres visuels. Nous avons déjà développé dans la section 8.2.2 la nécessité d'effectuer une évaluation rigoureuse de VXT à l'aide du prototype. Il serait intéressant avant cela d'implémenter le mécanisme permettant d'adapter visuellement et automatiquement les *VPMEs* aux structures qu'elles sélectionnent lors des phases d'évaluation progressive, de manière à proposer à l'utilisateur un environnement dans lequel la métaphore des filtres visuels est complètement supportée.

Contrôle statique de type. Comme nous l'avons vu dans le chapitre 2, le système de types Circus et la bibliothèque D-TaToo permettent de créer des transformations pour lesquelles la validité (au sens XML) des documents produits peut être vérifiée de façon statique (c'est-à-dire au moment de la compilation du

¹Pour l'instant, seule IsaViz utilise ce type d'objets ; ce problème n'affecte donc ni VXT ni les autres applications en cours de développement.

programme de transformation). Il serait intéressant d'étudier la possibilité d'intégrer cette fonctionnalité à l'environnement d'édition de VXT, de manière à bénéficier de ce contrôle (*Static Type Checking*).

Exportation vers Circus et suppression des modes d'édition. La fonction de traduction des programmes VXT vers Circus a été définie pour la version 1.0 de ce dernier. La version 2.0, disponible depuis peu, propose un nouveau modèle XML et de nouvelles constructions programmatiques, notamment de nouveaux opérateurs de filtrage et le support des références. Le passage à la version 2.0 permettrait de s'affranchir, dans l'environnement de développement, des deux modes d'édition distincts (l'un associé à XSLT, l'autre à Circus), qui sont pour l'instant nécessaires du fait de la différence d'expressivité entre XSLT et la version 1.0 de Circus, qui interdit par exemple au programmeur désirant exporter son programme de transformation vers Circus d'exprimer des contraintes sur les ancêtres du nœud contextuel.

8.3.3 Édition visuelle de modèles RDF

La version 1.1 d'IsaViz est stable et implémente la plupart des fonctionnalités simples dont l'ajout a été demandé par les utilisateurs de la version 1.0. Nous avons décrit dans le chapitre 6 un certain nombre de perspectives, que nous résumons ici. Celles-ci représentent des évolutions importantes, qui feront partie de la version 2.0 de l'application, dont l'implémentation devrait commencer au début de l'année 2003, après une phase d'étude liée au caractère plus exploratoire des nouvelles fonctionnalités envisagées.

Représentation des modèles. Le programme *dot* de la librairie GraphViz fournit des agencements de graphes de bonne qualité. Les utilisateurs aimeraient cependant pouvoir obtenir des représentations plus avancées, offrant par exemple des possibilités pour mettre en avant les éléments centraux du modèle. Nous proposons pour cela un mécanisme équivalent aux feuilles de style qui permettraient de modifier l'apparence visuelle des nœuds et arcs du diagramme en fonction de leur type et de leurs relations. Des méthodes d'agencement alternatives sont aussi envisagées, comme le regroupement dans un tableau de certaines propriétés/valeurs associées à une ressource donnée, de manière à optimiser et simplifier la représentation visuelle.

Édition des modèles. Deux nouvelles fonctionnalités sont envisagées dans le cadre de l'édition. Premièrement, un mécanisme d'interaction contrainte ou de suggestions faites à l'utilisateur, tenant compte de l'information fournie par le schéma RDF associé au modèle (s'il existe). Deuxièmement, un générateur automatique de fragments incomplets de modèles (*templates*) que l'utilisateur pourrait ensuite remplir. Ces deux mécanismes sont destinés à augmenter la productivité de l'utilisateur.

Annexes

Exemples de programmes XSLT et Circus engendrés par VXT

```

<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output indent="yes" method="xml" omit-xml-declaration="no"/>

  <xsl:template match="article[title and articleinfo]">
    <html xmlns="http://www.w3.org/1999/xhtml">
      <head>
        <xsl:copy-of select="title"/>
      </head>
      <body>
        <h1><xsl:value-of select="title"/></h1>
        <xsl:apply-templates select="articleinfo/author"/>
        <xsl:for-each select="//section">
          <hr/>
          <xsl:apply-templates select="."/>
        </xsl:for-each>
      </body>
    </html>
  </xsl:template>

  <xsl:template match="articleinfo/author">
    <p>
      Author :
      <xsl:value-of select="firstname"/>
      <xsl:text> </xsl:text>
      <xsl:value-of select="surname"/>
    </p>
  </xsl:template>

  <xsl:template match="section[title]">
    <p>
      <h2><xsl:value-of select="title"/></h2>
      <br/>
      <xsl:value-of select="para[position()=1]" />
      (Continued)
    </p>
  </xsl:template>

</xsl:stylesheet>

```

FIG. A.1 : *Programme XSLT*

Nous présentons dans cette annexe les programmes complets en XSLT et Circus correspondant à ce que l'environnement VXT génère à partir du programme de transformation dont la version VXT a servi d'exemple d'introduction au langage (section 4.3.3). Ce programme produit une page XHTML contenant des informations sur l'auteur et la structure logique d'un article au format DocBook. La figure A.1 contient la feuille de transformation XSLT. La figure A.2 contient le code source Circus de la transformation. On notera la présence de la fonction *XValue*, présente dans tous les programmes de transformation Circus générés par VXT. Cette fonction générique parcourt une forêt XML et retourne la concaténation sous forme de chaîne de caractères de tous les nœuds texte contenus dans les arbres composant la forêt, traversés en profondeur d'abord. On remarquera aussi la présence de la fonction *forEach_1*. Une fonction de ce type est générée pour chaque nœud *for-each* des productions de règles VXT. Il aurait été possible d'insérer ce code dans la partie droite des règles Circus, mais le code résultant aurait été plus difficile à lire. Nous avons donc fait le choix de traiter chaque boucle *for-each* de VXT dans une fonction séparée appelée dans la partie droite de la règle correspondante.

```

module DB2HTML {
  const db2html : [XMLTree]->[XMLTree]=lambda x: [XMLTree].
  var res: [XMLTree].
  var v1: XMLTree. var v2: XMLTree. var v3: XMLTree. var v4: XMLTree. var v5: XMLTree. //temp variables
  var v6: XMLTree. var v7: XMLTree. var v8: XMLTree. var v9: XMLTree. var v10: XMLTree. var v11: XMLTree.
  {
    for node in x do {
      res:=res+ //append result of each iteration to res, which will be returned at the end as the result
      [
        node # <label=%'article', //try to match node against rule t#0
          sub=? && [<label=%'title'>] && ? && [<label=%'articleinfo',sub=?+ [<label=%'author'>]++?>] && ?> =>
          (//production !a n html element)
          [<label='html',
            sub=[<label='head',sub=node.sub [<label=%'title'> and ?v1 => v1]>]
              + [<label='body',
                sub=[<label='h1',sub=XValue(node.sub [<label=%'title'> and ?v2 => v2])>]
                  + db2html(node.sub [<label=%'articleinfo',sub=?+ [<label=%'author'>]++?> and ?v3 => v3])
                  + forEach_1(node.sub [<label=%'section'> and ?v4 => v4])>]
              ]
            ]
          ],
        node # <label=%'articleinfo',sub=?+ [<label=%'author'>]++?> => //try to match node against rule t#1
          (//production !a p element containing the name of the author)
          [<label='p',
            sub=['Author: '
              + XValue(node.sub [<label=%'author',sub=?v5 => v5 [<label=%'firstname'> and ?v6 => v6]])
              + [ ' ' ]
              + XValue(node.sub [<label=%'author',sub=?v7 => v7 [<label=%'surname'> and ?v8 => v8]])
            ]
          ],
        node # <label=%'section',sub=?+ [<label=%'title'>]++?> => //try to match node against rule t#2
          (//production !a p element containing the section titles in h2 tags)
          [<label='p',
            sub=[<label='h2',
              sub=XValue(node.sub [<label=%'title',sub=?v9 => v9])>]
              + [<label='br'>]
              + XValue(node.sub [<label=%'para',sub=?v10 => v10])
              + [ '(Continued)' ]
            ]
          ],
      ]
    ]
  };
  res // when iteration has been completed, return the result
}

const forEach_1 : [XMLTree]->[XMLTree]=lambda x: [XMLTree].
var res: [XMLTree].
{
  for node in x do {
    res:=res
    + [<label='hr'>]
    + db2html(node)
  };
  res
}

const XValue : [XMLTree]->[XMLTree]=lambda x: [XMLTree].
var res: [XMLTree].
var t: [XMLTree]. var s: String. //temp variables
{
  for node in x do {
    res:=res+[
      node # ?s => [s],
      node # <sub=?t> => XValue(t)
    ]
  };
  res
}
}

```

FIG. A.2 : Programme Circus

Étude formelle d'un langage visuel pour la représentation de DTD

A Formal Study of a Visual Language for the Visualization of Document Type Definition

Jean-Yves Vion-Dury Emmanuel Pietriga*
Xerox Research Centre Europe
6 Chemin de Maupertuis 38240 Meylan - France
{jean-yves.vion-dury, emmanuel.pietriga}@xrce.xerox.com

Abstract

This formal study proposes a transformational approach to the definition of general purpose visual languages based on hierarchical structures, addressing more specifically DTD visualization as its application area. We show that such visual languages can be constructed through progressive refinement of a syntax based on nested/juxtaposed rectangles. Several transformation stages, which can all be formally characterized, produce a high quality visual representation which expresses the fundamental properties of the original structure. Moreover, this approach opens some perspectives in proving visual properties through standard mathematical tools such as inductive proofs, thus establishing some practical links between visual language theory and classical language theory.

1. Introduction

Document Type Definitions (DTDs) are used to constrain the structure of XML documents (trees). This paper describes formally a visual language for the graphical representation of DTDs, which is used in VXT [8], a visual programming language specialized in the specification of XML transformations. Several visual representations of DTDs have been proposed, both in industrial products and research prototypes. However, they all use node-link diagrams (Near and Far[7], XML-GL[2]), and none of these proposals seems to have been specified or studied using formal methods.

Section 2 presents the basic graphical object model we consider, and a visual syntax \mathcal{V} adapted to the representation of hierarchical structures, such as structured documents (SGML, XML, HTML) and DTDs. The required properties of this syntax are identified and captured in a pivotal ab-

* in partnership with INRIA Rhone-Alpes (Projet OPERA), 655 Av de l'Europe, 38330 Monbonnot Saint-Martin - France

stract syntax \mathcal{AS} which simplifies formal treatment by abstracting over some spatial issues while still having a visual semantics. The latter is defined through a translation function from the language of \mathcal{AS} into the language generated by \mathcal{V} . Section 3 proposes a formal definition of Document Type Definitions and a first visual semantics by the means of a translation that transforms any valid DTD into a sentence of \mathcal{AS} . Section 4 describes how such representations can be simplified and made more effective through an additional transformation based on a term rewriting system. This one reduces the number of graphical objects while preserving all the information. A subtle pretty-printing function is then presented in section 5, which enforces the perception of structural information thanks to spatial analogies. The discussion section analyses briefly other solutions for the representation and treatment of visual abstract syntaxes, showing how our work differentiates from them. The conclusion synthesizes the paper and discusses global results.

2. Visual Syntax

2.1. Representation model

graphical object model. We first define Gl as an abstract graphical type which does not have any representation. Objects of type Gl have five attributes, namely x, y which represent the coordinates of the object's center, z for the depth of the object (this information is used to determine the order in which to paint objects) and w, h for the width and height. Renderable objects can be of type Sh for shapes or Tx for text strings. Both are subtypes of Gl , the first one defining a shape to draw, the second one a text value. Sh also has a subtype Re for a particular kind of shape : rectangles. Type Sh also defines two additional attributes for color and border style (solid or dashed). For objects of type Sh (which are not of type Re), attributes w and h are computed by respectively projecting them on horizontal and vertical axes. Graphical objects being rendered in an infinite virtual space

[8], they can have negative coordinates and their width and height is not limited.

bounding box function. We then define function BB , which returns the smallest rectangle bounding two objects :

$$BB : Gl \times Gl \rightarrow Re$$

and which is defined as follows :

$$BB(A_1, A_2) = R$$

with

$$\begin{aligned} R.x &= (\min X + \max X)/2 \\ R.y &= (\min Y + \max Y)/2 \\ R.w &= \max X - \min X \\ R.h &= \max Y - \min Y \end{aligned}$$

where

$$\begin{aligned} \max X &= \max(A_1.x + A_1.w/2, A_2.x + A_2.w/2) \\ \max Y &= \max(A_1.y + A_1.h/2, A_2.y + A_2.h/2) \\ \min X &= \min(A_1.x - A_1.w/2, A_2.x - A_2.w/2) \\ \min Y &= \min(A_1.y - A_1.h/2, A_2.y - A_2.h/2) \end{aligned}$$

spatial relations. Using this relation, we introduce the following relational constraints, all defined by a 3-tuple $(r \ i \ j)$ where r is a name for the relation, i and j being objects of type Gl or one of its subtypes :

| Relation | Arg. types | Definition |
|-------------------|----------------|--|
| <i>CenteredOn</i> | $Gl \times Gl$ | $(i.x = j.x) \wedge (i.y = j.y)$ |
| <i>LeftOf</i> | $Gl \times Gl$ | $(i.x < j.x) \wedge (i.y = j.y)$ |
| <i>Above</i> | $Gl \times Gl$ | $(i.y > j.y) \wedge (i.x = j.x)$ |
| <i>Over</i> | $Gl \times Gl$ | $(i.z > j.z)$ |
| <i>Intersects</i> | $Gl \times Gl$ | $(BB(i, j).w < BB(i, i).w + BB(j, j).w) \wedge (BB(i, j).h < BB(i, i).h + BB(j, j).h)$ |
| <i>Contains</i> | $Gl \times Gl$ | $BB(i, j) = BB(i, i)$ |

2.2. Syntax definition

We can now introduce our visual language, defined by a grammar which is based on a slightly extended version¹ of the Relational Grammar formalism described by Wittenburg et al [14].

Definition 1 The grammar is a 5-tuple $\mathcal{V} = (\mathcal{N}, \Sigma, S, \mathcal{R}, P)$ where :

- \mathcal{N} is the following set of nonterminals : $\mathcal{N} = \{N_a, N_b, N_c\}$.
- Σ is a set of graphical terminal symbols of type Sh , Re or Tx , disjoint from \mathcal{N} .
- S is the start symbol in \mathcal{N} : $S = N_a$.
- \mathcal{R} is the set of relation symbols defined above : $\mathcal{R} = \{Contains, CenteredOn, Above, LeftOf, Intersect\}$.

¹Some relational constraints reference bounding boxes defined on objects belonging to α . This liberty, with respect to the formalism, is of no consequence since we are not interested in the properties associated with the formalism : we rely on it only as a descriptive tool and do not intend to use it for parsing.

- P is the following set of productions of the form $A \rightarrow \alpha/\beta/F$, where $A \in \mathcal{N}$, $\alpha \in (\mathcal{N}|\pm)^+$, β is a set of relational constraints of the form $(r \ i \ j)$ where $r \in \mathcal{R}$ and i, j are integers referencing a member of α (the left-hand-side of a rule is conventionally referenced as 0, the one or more right-hand-side elements are referenced 1..n in the order in which they appear in the definition. F is a set of assignment statements of the form $(a \ 0) = (a \ i)$, i referencing a member of α (see [14]):

$$\begin{aligned} N_a &\rightarrow Gl \ Sh && 1.0 \\ & \quad (CenteredOn \ 2 \ 1) \ (Contains \ 1 \ 2) \ (Over \ 2 \ 1) \\ & \quad (x \ 0) = (x \ 1), (y \ 0) = (y \ 1), ((z + 1) \ 0) = (z \ 1), \\ & \quad (w \ 0) = (w \ 1), (h \ 0) = (h \ 1) \\ N_a &\rightarrow Gl \ Tx && 1.1 \\ & \quad (CenteredOn \ 2 \ 1) \ (Contains \ 1 \ 2) \ (Over \ 2 \ 1) \\ & \quad (x \ 0) = (x \ 1), (y \ 0) = (y \ 1), ((z + 1) \ 0) = (z \ 1), \\ & \quad (w \ 0) = (w \ 1), (h \ 0) = (h \ 1) \\ N_a &\rightarrow Gl \ Re && 1.2 \\ & \quad (CenteredOn \ 2 \ 1) \ (Contains \ 1 \ 2) \ (Over \ 2 \ 1) \\ & \quad (x \ 0) = (x \ 1), (y \ 0) = (y \ 1), ((z + 1) \ 0) = (z \ 1), \\ & \quad (w \ 0) = (w \ 1), (h \ 0) = (h \ 1) \\ N_a &\rightarrow Gl \ Re \ N_a && 1.3 \\ & \quad (CenteredOn \ 2 \ 1) \ (Contains \ 1 \ 2) \ (Over \ 2 \ 1) \\ & \quad (CenteredOn \ 3 \ 2) \ (Contains \ 2 \ 3) \ (Over \ 3 \ 2) \\ & \quad (x \ 0) = (x \ 1), (y \ 0) = (y \ 1), ((z + 1) \ 0) = (z \ 1), \\ & \quad (w \ 0) = (w \ 1), (h \ 0) = (h \ 1) \\ N_a &\rightarrow Gl \ N_b && 1.4 \\ & \quad (CenteredOn \ 2 \ 1) \ (Contains \ 1 \ 2) \ (Over \ 2 \ 1) \\ & \quad (x \ 0) = (x \ 1), (y \ 0) = (y \ 1), ((z + 1) \ 0) = (z \ 1), \\ & \quad (w \ 0) = (w \ 1), (h \ 0) = (h \ 1) \\ N_a &\rightarrow Gl \ N_a \ N_c && 1.5 \\ & \quad (Contains \ 1 \ BB(2, 3)) \ (Over \ 2 \ 1) \ (Above \ 2 \ 3) \\ & \quad (CenteredOn \ BB(2, 3) \ 1) \ (Over \ 3 \ 1) \\ & \quad (not \ Intersect \ 2 \ 3) \\ & \quad (x \ 0) = (x \ 1), (y \ 0) = (y \ 1), ((z + 1) \ 0) = (z \ 1), \\ & \quad (w \ 0) = (w \ 1), (h \ 0) = (h \ 1) \\ N_b &\rightarrow Gl \ N_a \ N_b && 1.6 \\ & \quad (Contains \ 1 \ BB(2, 3)) \ (not \ Intersect \ 2 \ 3) \\ & \quad (CenteredOn \ BB(2, 3) \ 1) \ (Over \ 3 \ 1) \\ & \quad (LeftOf \ 2 \ 3) \ (Over \ 2 \ 1) \\ & \quad (x \ 0) = (x \ 1), (y \ 0) = (y \ 1), ((z + 1) \ 0) = (z \ 1), \\ & \quad (w \ 0) = (w \ 1), (h \ 0) = (h \ 1) \\ N_b &\rightarrow Gl \ N_a && 1.7 \\ & \quad (CenteredOn \ 2 \ 1) \ (Contains \ 1 \ 2) \ (Over \ 2 \ 1) \\ & \quad (x \ 0) = (x \ 1), (y \ 0) = (y \ 1), ((z + 1) \ 0) = (z \ 1), \\ & \quad (w \ 0) = (w \ 1), (h \ 0) = (h \ 1) \\ N_c &\rightarrow Gl \ N_a \ N_c && 1.8 \\ & \quad (Contains \ 1 \ BB(2, 3)) \ (Over \ 2 \ 1) \\ & \quad (CenteredOn \ BB(2, 3) \ 1) \ (Over \ 3 \ 1) \\ & \quad (Above \ 2 \ 3) \ (not \ Intersect \ 2 \ 3) \\ & \quad (x \ 0) = (x \ 1), (y \ 0) = (y \ 1), ((z + 1) \ 0) = (z \ 1), \\ & \quad (w \ 0) = (w \ 1), (h \ 0) = (h \ 1) \\ N_c &\rightarrow Gl \ N_a && 1.9 \\ & \quad (CenteredOn \ 2 \ 1) \ (Contains \ 1 \ 2) \ (Over \ 2 \ 1) \\ & \quad (x \ 0) = (x \ 1), (y \ 0) = (y \ 1), ((z + 1) \ 0) = (z \ 1), \\ & \quad (w \ 0) = (w \ 1), (h \ 0) = (h \ 1) \end{aligned}$$

2.3. Visual properties of the syntax

The language defined by this grammar is a superset of the sentences that can be generated from the translation of DTDs. Basically, it allows the nesting at an arbitrary level of graphical objects, including text strings, provided that objects containing other objects are of type Re (i.e. rectangles). When a rectangle contains a set of objects, elements of this set can be laid out either horizontally or vertically but cannot overlap even partially, and the set is always centered with respect to the parent rectangle, whose size is defined so as to fully contain its children.

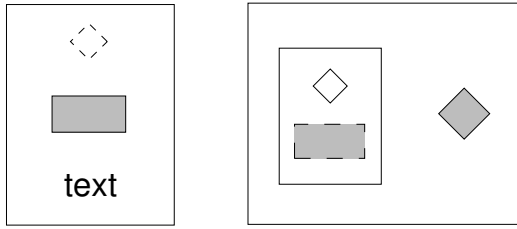


Figure 1. Examples of correct visual sentences

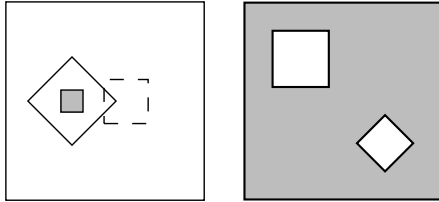


Figure 2. Examples of incorrect visual sentences

Figure 1 shows two examples of visual sentences allowed by the grammar : all objects and sets of objects are centered with respect to their parent and are fully contained within them. Figure 2 illustrates two sentences that do not belong to the language. The left-hand one is incorrect because shapes other than rectangles are not allowed to contain objects and because two sibling shapes partially overlap. In the right-hand one, components of the outmost rectangle are not aligned vertically or horizontally, thus making the sentence incorrect.

Example 1 *the full derivation path of the leftmost sentence in Figure 1*^{2 3}.

$$\begin{aligned}
& (\{N_a\}, \emptyset) \\
\stackrel{(1.3)}{\longrightarrow} & (\{Gl_1, Re_2, N_a\} \models \mathcal{R}_1 \cup \{(Contains\ Gl_1\ Re_2), \\
& (CenteredOn\ Re_2\ Gl_1), (CenteredOn\ N_a\ Re_2), \\
& (Over\ N_a\ Re_2), (Contains\ Re_2\ N_a), (Over\ Re_2\ Gl_1)\}) \\
\stackrel{(1.5)}{\longrightarrow} & (\{Gl_1, Re_2, Gl_3, N_a, N_c\} \models \mathcal{R}_2 \cup \{(Above\ N_a\ N_c), \\
& (Contains\ Gl_3\ BB(N_a, N_c)), (Over\ N_a\ Gl_3), \\
& (not\ Intersect\ N_a\ N_c), (Over\ N_a\ Gl_3), \\
& (CenteredOn\ BB(N_a, N_c)\ Gl_3)\}) \\
\stackrel{(1.0)}{\longrightarrow} & (\{Gl_1, Re_2, Gl_3, Gl_4, Sh_5, N_c\} \models \\
& \mathcal{R}_3 \cup \{(Over\ Sh_5\ Gl_4), (CenteredOn\ Sh_5\ Gl_4), \\
& (Contains\ Gl_4\ Sh_5)\}) \\
\stackrel{(1.8)}{\longrightarrow} & (\{Gl_1, Re_2, Gl_3, Gl_4, Sh_5, Gl_6, N_a, N_c\} \models \\
& \mathcal{R}_4 \cup \{(Over\ N_a\ Gl_6), (Contains\ Gl_6\ BB(N_a, N_c)), \\
& (Over\ N_c\ Gl_6), (CenteredOn\ BB(N_a, N_c)\ Gl_6), \\
& (Above\ N_a\ N_c), (not\ Intersect\ N_a\ N_c)\}) \\
\stackrel{(1.2)}{\longrightarrow} & (\{Gl_1, Re_2, Gl_3, Gl_4, Sh_5, Gl_6, Gl_7, Sh_8, N_c\} \models \\
& \mathcal{R}_5 \cup \{(CenteredOn\ Sh_8\ Gl_7), (Contains\ Gl_7\ Sh_8), \\
& (Over\ Sh_8\ Gl_7)\})
\end{aligned}$$

² \mathcal{R}_i represents the union of \mathcal{R}_{i-1} with the set of relations specified in the previous derivation step.

³We note $S \models R$ the couple (S, R) such that elements in S satisfy all relations in R .

$$\begin{aligned}
\stackrel{(1.9)}{\longrightarrow} & (\{Gl_1, Re_2, Gl_3, Gl_4, Sh_5, Gl_6, Gl_7, Sh_8, \\
& Gl_9, N_a\} \models \mathcal{R}_6 \cup \{(CenteredOn\ N_a\ Gl_9), \\
& (Contains\ Gl_9\ N_a), (Over\ N_a\ Gl_9)\}) \\
\stackrel{(1.1)}{\longrightarrow} & (\{Gl_1, Re_2, Gl_3, Gl_4, Sh_5, Gl_6, Gl_7, Sh_8, \\
& Gl_9, Gl_{10}, Tx_{11}\} \models \mathcal{R}_7 \cup \{(CenteredOn\ Tx_{11}\ Gl_{10}), \\
& (Contains\ Gl_{10}\ Tx_{11}), (Over\ Tx_{11}\ Gl_{10})\})
\end{aligned}$$

2.4. Abstract syntax

We now introduce a formal abstraction which will ease the subsequent formal treatments and transformations (with underlying visual soundness).

definition. We propose to capture the previous visual properties into an abstract syntax which is not purely visual but rather structural [5]. The tree grammar formalism [10] is a natural extension of CFG grammars allowing the definition of languages as sets of trees instead of sets of strings. This formalism, while powerful in capturing tree-like structures, is also clear and simple enough to understand and supports inductive proof approaches naturally.

Definition 2 (Abstract Syntax \mathcal{AS})

| | | | | |
|-----|---------------|-------------------------------|-------------------------|------|
| G | \rightarrow | $\mathbf{Box}(G, s, w, h, c)$ | <i>container object</i> | 2.10 |
| G | \rightarrow | $\mathbf{VA}(G, G, d)$ | <i>Ver. alignment</i> | 2.11 |
| G | \rightarrow | $\mathbf{HA}(G, G, d)$ | <i>Hor. alignment</i> | 2.12 |
| G | \rightarrow | $\mathbf{Shp}(f, s, w, h, c)$ | <i>terminal shape</i> | 2.13 |
| G | \rightarrow | $\mathbf{Txt}(v)$ | <i>terminal text</i> | 2.14 |

where f is a meta-variable for various shapes such as (**triangle, square, ...**), s denotes style (**dashed, solid**), w, h, d are positive numerical variables for width, height and spacing and c represents color. Meta-variable v designates the text value. We note $\mathcal{L}(\mathcal{AS})$ the (tree) language generated by \mathcal{AS} .

Example 2 (an element of $\mathcal{L}(\mathcal{AS})$) *(captures the structure of the leftmost sentence in Fig. 1)*

$$\begin{aligned}
& \mathbf{Box}(\\
& \quad \mathbf{VA}(\\
& \quad \quad \mathbf{Shp}(f_2, s_2, w_2, h_2, c_2), \\
& \quad \quad \mathbf{VA}(\mathbf{Txt}(v_4), \mathbf{Shp}(f_5, s_5, w_5, h_5, c_5), d_3) \\
& \quad \quad , d_1), s_0, w_0, h_0, c_0)
\end{aligned}$$

We now demonstrate the membership of the previous tree by providing a full derivation path in \mathcal{AS} .

Example 3 (The derivation path of Example 2) *the previous tree belongs to $\mathcal{L}(\mathcal{AS})$, as shown by the following derivation path*

$$G \rightarrow \mathbf{Box}(G, s_0, w_0, h_0, c_0) \quad (2.10)$$

$$\rightarrow \mathbf{Box}(\mathbf{VA}(G, G, d_1), s_0, w_0, h_0, c_0) \quad (2.11)$$

$$\rightarrow \mathbf{Box}(\mathbf{VA}(\mathbf{Shp}(f_2, s_2, w_2, h_2, c_2), G, d_1), s_0, w_0, h_0, c_0) \quad (2.13)$$

$$\rightarrow \mathbf{Box}(\mathbf{VA}(\mathbf{Shp}(f_2, s_2, w_2, h_2, c_2), \mathbf{VA}(G, G, d_3), d_1), s_0, w_0, h_0, c_0) \quad (2.11)$$

$$\rightarrow \mathbf{Box}(\mathbf{VA}(\mathbf{Shp}(f_2, s_2, w_2, h_2, c_2), \mathbf{VA}(\mathbf{Txt}(v_4), G, d_3), d_1), s_0, w_0, h_0, c_0) \quad (2.14)$$

$$\rightarrow \mathbf{Box}(\mathbf{VA}(\mathbf{Shp}(f_2, s_2, w_2, h_2, c_2), \mathbf{VA}(\mathbf{Txt}(v_4), \mathbf{Shp}(f_5, s_5, w_5, h_5, c_5), d_3), d_1), s_0, w_0, h_0, c_0) \quad (2.13)$$

visual semantics. The visual semantics of this abstract syntax is defined by a translation function which produces as output a set of indexed visual objects compatible with the representation model introduced in section 2.

The function accepts items of $\mathcal{L}(\mathcal{AS})$, and returns a set of indexed graphical objects Γ . It computes the value of visual attributes associated to each object through an ordered sequence of a basic linear algebra instruction set. Elements of Γ are indexed sets of graphical objects noted $\gamma = \{O_1, \dots, O_n\}$ where the meta-variable O ranges over the syntactic categories $\{Gl, Re, Tx\}$. The function also propagates a numerical index i which corresponds to the depth of the current node in the source tree. We use the notation $\gamma[G_k.attr := \dots]$ for the set computed from γ by applying the operation $O_k.attr := \dots$ on every element O_k of γ . For instance, if $\gamma = \{Re_1, Re_2\}$ then $\gamma[O_k.x := O_k.x + 10]$ will compute a new set $\gamma' = \{Re_2, Re_3\}$ where all x coordinates are shifted by 10. We consider also $\gamma[O_k.attr := \dots, O_k.attr := \dots]$ as a shorthand for $(\gamma[O_k.attr := \dots])[O_k.attr := \dots]$. Finally, $BB(\gamma)$ denotes the bounding box obtained from all objects in γ .

Definition 3 (\mathcal{T} , Visual translation of $\mathcal{L}(\mathcal{AS})$)

$$\mathcal{T}[] : \mathcal{L}(\mathcal{AS}) \times N \rightarrow \Gamma$$

$$\mathcal{T}[\text{Txt}(v)]_i = \{Tx_i, Gl_i\} \quad 3.15$$

such that $\begin{cases} Tx_i.z = i \\ Tx_i.y = 0 \\ Gl_i = BB(Tx_i, Tx_i) \end{cases} \quad \begin{matrix} Tx_i.x = 0 \\ Tx_i.value = v \end{matrix}$

$$\mathcal{T}[\text{Shp}(f, s, w, h, c)]_i = \{Sh_i, Gl_i\} \quad 3.16$$

such that $\begin{cases} Sh_i.x = 0, & Sh_i.y = 0, & Sh_i.w = w, \\ Sh_i.h = h, & Sh_i.shape = f, & Sh_i.border = s, \\ Sh_i.z = i & Sh_i.color = c & Sh_i.value = v \end{cases}$

$$\text{and } Gl_i = BB(Sh_i, Sh_i)$$

$$\mathcal{T}[\text{Box}(G_1, s, w, h, c)]_i = \gamma_1 \cup \{Re_i, Gl_i\} \quad 3.17$$

with $\begin{cases} \mathcal{T}[G_1]_j = \gamma_1, & B_1 = BB(\gamma_1), \\ Re_i.x = B_1.x, & Re_i.y = B_1.y, \\ Re_i.w = \max(w, B_1.w + 2), \\ Re_i.h = \max(h, B_1.h + 2) \\ Re_i.border = s, & Re_i.color = c \\ Re_i.z = i, & Gl_i = BB(Re_i, Re_i) \end{cases}$

$$\mathcal{T}[\text{HA}(G_1, G_2, d)]_i = \gamma_1 \cup \gamma_2 \cup \{Gl_i\} \quad 3.18$$

with $\begin{cases} \mathcal{T}[G_1]_j = \gamma_1, & B_1 = BB(\gamma_1), \\ \mathcal{T}[G_2]_j = \gamma_2, & B_2 = BB(\gamma_2), & Gl_i = BB(B_1, \gamma_2) \end{cases}$

and $\gamma_2 = \gamma_2[G_k.y := B_1.y, G_k.x := B_1.w + B_2.w + d]$

$$\mathcal{T}[\text{VA}(G_1, G_2, d)]_i = \gamma_1 \cup \gamma_2 \cup \{Gl_i\} \quad 3.19$$

with $\begin{cases} \mathcal{T}[G_1]_j = \gamma_1, & B_1 = BB(\gamma_1), \\ \mathcal{T}[G_2]_j = \gamma_2, & B_2 = BB(\gamma_2), & Gl_i = BB(B_1, \gamma_2) \end{cases}$

and $\gamma_2 = \gamma_2[G_k.x := B_1.x, G_k.x := B_1.h + B_2.h + d]$

The following property states that the visual translation produces visual sentences which belong to the language generated by the visual syntax \mathcal{V} .

Property 1 (membership of $\mathcal{T}[\mathcal{AS}]$.) for all tree t belonging to $\mathcal{L}(\mathcal{AS})$, $\mathcal{T}[t]_0$ belongs to $\mathcal{L}(\mathcal{V})$

Proof 1 (by induction on the length of derivation paths)

Sketch: the induction hypothesis $\mathcal{H}_{i \geq 1}$ is

$$\begin{array}{l} \text{for all derivation} \\ \text{there exist } j \geq 1 \\ \text{and a mapping} \\ \text{such that} \\ \text{where} \end{array} \quad \begin{array}{l} G \xrightarrow{j} t \\ N_1 \models \emptyset \xrightarrow{j} \gamma \models \mathcal{R} \\ \phi : \gamma \rightarrow \mathcal{T}[t]_0 \\ \mathcal{T}[t]_0 \models \mathcal{R}_\phi, \\ \mathcal{R}_\phi \text{ denotes the transposition of relations in } \mathcal{R} \text{ through } \phi \end{array}$$

We then demonstrate that $\forall i \geq 1, \mathcal{H}_1 \wedge \dots \wedge \mathcal{H}_i \implies \mathcal{H}_{i+1}$. We just produce one of the 3 cases (**Box**):

3. Visual representation of DTDs

The interested reader will find the precise specification of DTDs in [1]. For the clarity of our presentation, we use hereafter a simplified version which does not handle attributes even if our implementation does represent them [8].

3.1. Document Type Definitions

Definition 4 (Document Type Definition) A DTD D_l is a set of rules indexed by a unique name, noted $D_{name_i} = \{name_1 \rightarrow N_1, \dots, name_n \rightarrow N_n\}$ in which the rule named $name_i$ is considered as the root. The structure N of each rule is defined through the following (abstract) syntax $\mathcal{R}\mathcal{H}$:

$$R \rightarrow R? \mid R^* \mid R^+ \mid (R_1, \dots, R_n) \mid (R_1 \mid \dots \mid R_n) \mid \# \mid \text{EMPTY} \mid \text{ANY} \mid \text{name}$$

Definition 5 (Valid DTD) A DTD is valid if all rule names are defined and if no pathological recursive rules are defined, i.e. rules that imply infinite derivation paths. We call $\mathcal{D}\mathcal{T}\mathcal{D}$ the set of all valid DTDs.

The XML DTD described here after

```
<!ELEMENT mail (sender,subject,textbody)>
<!ELEMENT sender (#PCDATA)>
<!ELEMENT subject (#PCDATA)>
<!ELEMENT textbody (p)+>
<!ELEMENT p (#PCDATA | cite)*>
<!ELEMENT cite (#PCDATA)>
```

is thus noted

$$D_{\text{mail}} = \{ \text{mail} \rightarrow (\text{sender,subject,textbody}), \\ \text{sender} \rightarrow \#, \\ \text{subject} \rightarrow \#, \\ \text{textbody} \rightarrow p^+, \\ p \rightarrow (\# | \text{cite})^*, \\ \text{cite} \rightarrow \# \}$$

In order to simplify notations we consider DTDs as mappings, and thus for instance, the functional notation $D_{\text{mail}}(p)$ corresponds to $(\# | \text{cite})^*$.

3.2. Translation Semantics

informal description. We propose a recursive function which computes a visual representation of any legal DTD. Possible recursion and cross-references in rule definitions are handled thanks to contextual information. Each element is processed normally the first time it is encountered in the tree, subsequent references to this element being symbolized by a special graphical object with the element's name as a decoration but no content.

The function is defined over DTDs, sets of labels (rule names) Ψ and $\mathcal{L}(\mathcal{AS})$. Elements of Ψ are sets of labels

noted $\psi = \{l_1, \dots, l_n\}$, memorizing which rules l_i have already been processed (to avoid endless processing of recursive rule definition and to allow the factorization of graphical object translation).

The full signature of the main function is formally defined by:

$$\mathcal{VF}[\cdot] : \mathcal{L}(\mathcal{R}\mathcal{H}) \times \mathcal{D}\mathcal{T}\mathcal{D} \times \Psi \rightarrow \mathcal{L}(\mathcal{A}\mathcal{S}) \times \Psi$$

and the translation of a DTD D_ℓ , noted $\mathcal{VF}[\![D_\ell]\!]$ is recursively defined.

Definition 6 (translation function \mathcal{VF})

$$\mathcal{VF}(D_\ell) = t \text{ with } t, \psi = \mathcal{VF}[\![D_\ell(t)]\!]_{D_\ell, \emptyset}$$

$$\mathcal{VF}[\![\#]\!]_{D_\ell, \psi} = \text{Shp}(\text{lozenge, solid, 10, 10, yellow}), \psi \quad 6.20$$

$$\mathcal{VF}[\![ANY]\!]_{D_\ell, \psi} = \text{Box}(\text{Txt}(ANY), \text{solid, 10, 10, white}), \psi \quad 6.21$$

$$\mathcal{VF}[\![EMPTY]\!]_{D_\ell, \psi} = \text{Shp}(\text{rectangle, solid, 0, 0, void}), \psi \quad 6.22$$

$$\mathcal{VF}[\![R_1, \dots, R_n]\!]_{D_\ell, \psi} = \text{Box}(t, \text{solid, 10, 10, blue}), \psi_n \quad 6.23$$

$$\text{with } \begin{cases} t = \text{HA}(t_1, t_2, 2) \\ t_1, \psi_1 = \mathcal{VF}[\![R_1]\!]_{D_\ell, \psi} \\ \vdots \\ t_i, \psi_i = \text{HA}(t'_i, t_{i+1}, 2), \\ \text{and } t'_i, \psi_i = \mathcal{VF}[\![R_i]\!]_{D_\ell, \psi_{i-1}} \\ \vdots \\ t_n = t'_n, t'_n, \psi_n = \mathcal{VF}[\![R_n]\!]_{D_\ell, \psi_{n-1}} \end{cases}$$

$$\mathcal{VF}[\![R_1 | \dots | R_n]\!]_{D_\ell, \psi} = \text{VA}(t, t_2, 2), \psi_2 \quad 6.24$$

$$\text{with } \begin{cases} t = \text{Box}(t_1, \text{solid, 10, 10, green}) \\ t_1, \psi_1 = \mathcal{VF}[\![R_1]\!]_{D_\ell, \psi} \\ t_2, \psi_2 = \mathcal{VF}[\![R_2 | \dots | R_n]\!]_{D_\ell, \psi_1} \end{cases}$$

$$\mathcal{VF}[\![R]\!]_{D_\ell, \psi} = \mathcal{VF}[\![R]\!]_{D_\ell, \psi} \quad 6.25$$

$$\mathcal{VF}[\![R?]\!]_{D_\ell, \psi} = \text{HA}(t, \text{Shp}(\text{rectangle, dashed, 10, 10, white}), 2), \psi_1$$

$$\text{with } t = \text{Box}(t_1, \text{solid, 10, 10, white}) \quad 6.26$$

$$\text{and } t_1, \psi_1 = \mathcal{VF}[\![R]\!]_{D_\ell, \psi}$$

$$\mathcal{VF}[\![R^*]\!]_{D_\ell, \psi} = \text{HA}(t, \text{Shp}(\text{rectangle, dashed, 10, 10, c}), 2), \psi_1$$

$$\text{with } t = \text{Box}(t_1, \text{dashed, 10, 10, white}) \quad 6.27$$

$$\text{and } t_1, \psi_1 = \mathcal{VF}[\![R]\!]_{D_\ell, \psi}$$

$$\mathcal{VF}[\![R^+]\!]_{D_\ell, \psi} = \text{HA}(t, \text{Shp}(\text{rectangle, dashed, 10, 10, white}), 2), \psi_1$$

$$\text{with } t = \text{Box}(t_1, \text{solid, 10, 10, white}) \quad 6.28$$

$$\text{and } t_1, \psi_1 = \mathcal{VF}[\![R]\!]_{D_\ell, \psi}$$

$$\mathcal{VF}[\![name]\!]_{D_\ell, \psi} = \begin{cases} \text{VA}(\text{Txt}(name), t_1, 2), \psi & \text{if } name \in \psi \\ \text{VA}(\text{Txt}(name), t_2, 2), \psi_1 & \text{otherwise} \end{cases} \quad 6.29$$

$$\text{with } t_1 = \text{Shp}(\text{rectangle, solid, 10, 10, grey})$$

$$\text{and } t_2 = \text{Box}(t, \text{solid, 10, 10, white})$$

$$\text{and } t, \psi_1 = \mathcal{VF}[\![D_\ell(name)]\!]_{D_\ell, \psi \cup \{name\}}$$

Property 2 $\forall d_m \in \mathcal{D}\mathcal{T}\mathcal{D}, \mathcal{VF}[\![d_m]\!] \in \mathcal{L}(\mathcal{A}\mathcal{S})$

Proof 2 We first prove that the function computes a result for every finite DTD (the difficulty is for recursive DTDs), and then we demonstrate that the result belongs to $\mathcal{L}(\mathcal{A}\mathcal{S})$.

Note that the terms generated by \mathcal{VF} are more specific than terms of $\mathcal{A}\mathcal{S}$. For instance, no term having the form $\text{VA}(\text{VA}(G_1, G_2), G_3)$ is produced. We propose to capture these specific structural properties through a tree grammar $\mathcal{D}\mathcal{S}$ which is a refinement of $\mathcal{A}\mathcal{S}$.

Definition 7 (A more specific abstract syntax $\mathcal{D}\mathcal{S}$)

| | | | |
|-----|---------------|---|------|
| D | \rightarrow | $\text{Box}(\text{Txt}(v), s, w, h, c)$ | 7.30 |
| D | \rightarrow | $\text{Box}(H, s, w, h, c)$ | 7.31 |
| H | \rightarrow | $\text{HA}(D, H, d) \mid \text{HA}(D, D, d)$ | 7.32 |
| D | \rightarrow | $\text{VA}(\text{Txt}(v), \text{Shp}(f, s, w, h, c), d)$ | 7.33 |
| D | \rightarrow | $\text{VA}(\text{Txt}(v), \text{Box}(D, s, w, h, c), d)$ | 7.34 |
| D | \rightarrow | $\text{VA}(\text{Box}(D, s, w, h, c), d, V)$ | 7.35 |
| V | \rightarrow | $\text{VA}(\text{Box}(D, s, w, h, c), d, V) \mid D$ | 7.36 |
| D | \rightarrow | $\text{HA}(\text{Box}(D, s, w, h, c), \text{Shp}(f, s, w, h, c))$ | 7.37 |

We now prove that $\mathcal{D}\mathcal{S}$ is a specialization of $\mathcal{A}\mathcal{S}$

Property 3 $\mathcal{L}(\mathcal{D}\mathcal{S}) \subseteq \mathcal{L}(\mathcal{A}\mathcal{S})$

Proof 3 We prove for all term t that

$$D \xrightarrow{*} t \implies G \xrightarrow{*} t$$

(by induction on the length of derivation paths)

And last, we state that \mathcal{T} produces terms of $\mathcal{L}(\mathcal{D}\mathcal{S})$

Property 4 for all valid DTD $D_\ell, \mathcal{T}[\![D_\ell]\!] \in \mathcal{L}(\mathcal{D}\mathcal{S})$

Proof 4 By structural induction on terms returned by \mathcal{T}

Example 4 (Translations of the mail DTD D_{mail}) See Figure 3 for $\mathcal{VF}(D_{mail})$ and Figure 4 for $\mathcal{T}[\![\mathcal{VF}(D_{mail})]\!]$.

4. Simplification

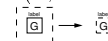
overview. A term rewriting system [3], named \mathcal{O} , is used to simplify the visual sentences obtained after translation. It consists of a set of unordered orthogonal rewriting rules, which can for instance be applied using a depth-first, innermost-first strategy. Basically, the simplification erases graphical objects representing cardinality and transfers this information to their children by updating attributes such as the border style (solid or dashed).

Definition 8 Rewriting system \mathcal{O} over $\mathcal{L}(\mathcal{A}\mathcal{S})$

$$\text{Box}(\text{VA}(\text{Txt}(v), \text{Shp}(\text{rectangle, solid, } w_1, h_1, c_1), d), \text{dashed, } w_2, h_2, c_2) \rightarrow \text{VA}(\text{Txt}(v), \text{Shp}(\text{rectangle, dashed, } w_1, h_1, c_1), d) \quad 8.38$$



$$\text{Box}(\text{VA}(\text{Txt}(v), \text{Box}(G, \text{solid, } w_1, h_1, c_1), d), \text{dashed, } w_2, h_2, c_2) \rightarrow \text{VA}(\text{Txt}(v), \text{Box}(G, \text{dashed, } w_1, h_1, c_1), d) \quad 8.39$$



$$\text{Box}(\text{Box}(H, \text{solid, } w_1, h_1, \text{blue}), \text{dashed, } w_2, h_2, c_2) \rightarrow \text{Box}(H, \text{dashed, } w_1, h_1, \text{blue}) \quad 8.40$$



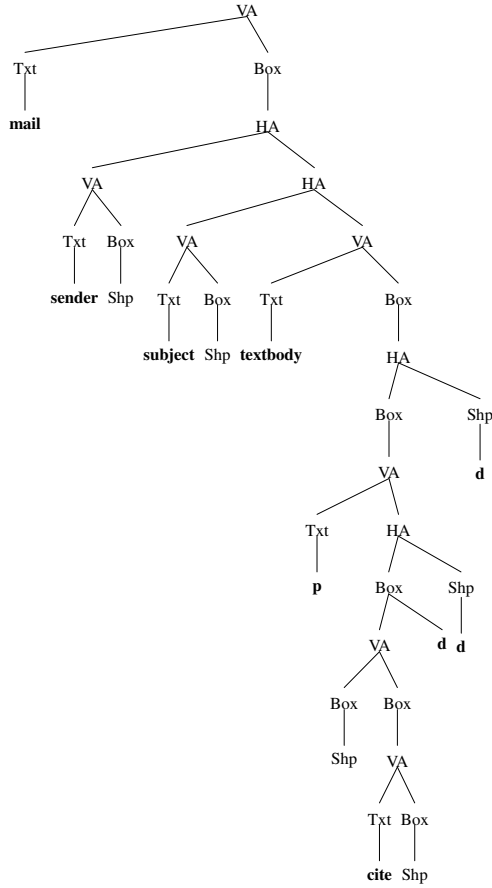


Figure 3. the translation $\mathcal{VF}(D_{mail})$ (simplified leaves, with $d = \text{dashed}$)

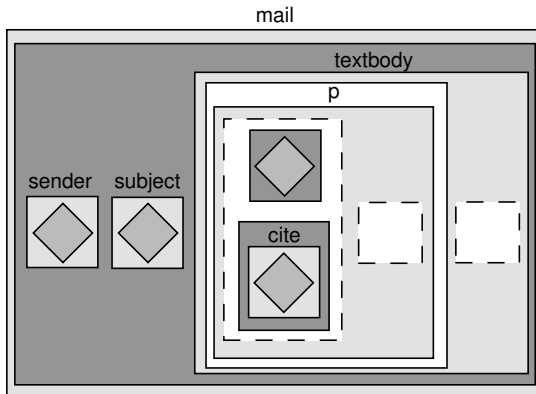
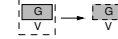
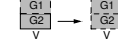


Figure 4. Raw translation of the mail DTD (no optimization, no pretty printing)

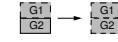
$$\text{Box}(\text{VA}(\text{Box}(G, \text{solid}, w_1, h_1, \text{green}), V, d), \text{dashed}, w_2, h_2, c_2) \rightarrow \text{VA}(\text{Box}(G, \text{dashed}, w_1, h_1, \text{green}), V, d) \quad 8.41$$



$$\text{VA}(\text{Box}(G_1, \text{dashed}, w_1, h_1, \text{green}), \text{VA}(\text{Box}(G_2, \text{solid}, w_2, h_2, \text{green}), V, d_2), d_1) \rightarrow \text{VA}(\text{Box}(G_1, \text{dashed}, w_1, h_1, \text{green}), \text{VA}(\text{Box}(G_2, \text{dashed}, w_2, h_2, \text{green}), V, d_2), d_1) \quad 8.42$$



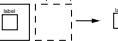
$$\text{VA}(\text{Box}(G_1, \text{dashed}, w_1, h_1, \text{green}), \text{Box}(G_2, \text{solid}, w_2, h_2, \text{green}), d) \rightarrow \text{VA}(\text{Box}(G_1, \text{dashed}, w_1, h_1, \text{green}), \text{Box}(G_2, \text{dashed}, w_2, h_2, \text{green}), d) \quad 8.43$$



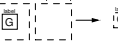
$$\text{HA}(\text{Box}(\text{VA}(\text{Txt}(v), \text{Shp}(\text{rectangle}, \text{solid}, w_1, h_1, c_1), d_1), \text{dashed}, w_2, h_2, c_2), \text{Shp}(\text{rectangle}, \text{dashed}, w_2, h_2, c_2), 2) \rightarrow \text{HA}(\text{VA}(\text{Txt}(v), \text{Shp}(\text{rectangle}, \text{dashed}, w_1, h_1, c_1), d_1), \text{Shp}(\text{rectangle}, \text{dashed}, h_1, h_1, c_1), 2) \quad 8.44$$



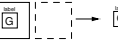
$$\text{HA}(\text{Box}(\text{VA}(\text{Txt}(v), \text{Shp}(\text{rectangle}, \text{solid}, w_1, h_1, c_1), d_1), \text{solid}, w_2, h_2, c_2), \text{Shp}(\text{rectangle}, \text{dashed}, w_2, h_2, c_2), 2) \rightarrow \text{HA}(\text{VA}(\text{Txt}(v), \text{Shp}(\text{rectangle}, \text{solid}, w_1, h_1, c_1), d_1), \text{Shp}(\text{rectangle}, \text{dashed}, h_1, h_1, c_1), 2) \quad 8.45$$



$$\text{HA}(\text{Box}(\text{VA}(\text{Txt}(v), \text{Box}(G, \text{solid}, w_1, h_1, c_1), d_1), \text{dashed}, w_2, h_2, c_2), \text{Shp}(\text{rectangle}, \text{dashed}, w_3, h_3, c_3), 2) \rightarrow \text{HA}(\text{VA}(\text{Txt}(v), \text{Box}(G, \text{dashed}, w_1, h_1, c_1), d_1), \text{Shp}(\text{rectangle}, \text{dashed}, h_1, h_1, c_1), 2) \quad 8.46$$



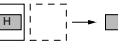
$$\text{HA}(\text{Box}(\text{VA}(\text{Txt}(v), \text{Box}(G, \text{solid}, w_1, h_1, c_1), d_1), \text{solid}, w_2, h_2, c_2), \text{Shp}(\text{rectangle}, \text{dashed}, w_3, h_3, c_3), 2) \rightarrow \text{HA}(\text{VA}(\text{Txt}(v), \text{Box}(G, \text{solid}, w_1, h_1, c_1), d_1), \text{Shp}(\text{rectangle}, \text{dashed}, h_1, h_1, c_1), 2) \quad 8.47$$



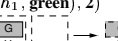
$$\text{HA}(\text{Box}(\text{Box}(H, \text{solid}, w_1, h_1, \text{blue}), \text{dashed}, w_2, h_2, c_2), \text{Shp}(\text{rectangle}, \text{dashed}, w_3, h_3, c_3), 2) \rightarrow \text{HA}(\text{Box}(H, \text{dashed}, w_1, h_1, \text{blue}), \text{Shp}(\text{rectangle}, \text{dashed}, h_1, h_1, \text{blue}), 2) \quad 8.48$$



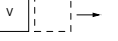
$$\text{HA}(\text{Box}(\text{Box}(H, \text{solid}, w_1, h_1, \text{blue}), \text{solid}, w_2, h_2, c_2), \text{Shp}(\text{rectangle}, \text{dashed}, w_3, h_3, c_3), 2) \rightarrow \text{HA}(\text{Box}(H, \text{solid}, w_1, h_1, \text{blue}), \text{Shp}(\text{rectangle}, \text{dashed}, h_1, h_1, \text{blue}), 2) \quad 8.49$$



$$\text{HA}(\text{Box}(\text{VA}(\text{Box}(G, \text{solid}, w_1, h_1, \text{green}), V, d_1), \text{dashed}, w_2, h_2, c_2), \text{Shp}(\text{rectangle}, \text{dashed}, w_3, h_3, c_3), 2) \rightarrow \text{HA}(\text{VA}(\text{Box}(G, \text{dashed}, w_1, h_1, \text{green}), V, d_1), \text{Shp}(\text{rectangle}, \text{dashed}, w_1, h_1, \text{green}), 2) \quad 8.50$$



$$\text{HA}(\text{Box}(V, \text{solid}, w_1, h_1, c_1), \text{Shp}(\text{rectangle}, \text{dashed}, w_2, h_2, c_2), 2) \rightarrow \text{HA}(V, \text{Shp}(\text{rectangle}, \text{dashed}, w_2, h_2, c_2), 2) \quad 8.51$$



Property 5 (closure over \mathcal{DS}) for all trees t belonging to $\mathcal{L}(\mathcal{DS})$, $\mathcal{O}(t)$ belongs to $\mathcal{L}(\mathcal{DS})$

Proof 5 We prove the termination over $\mathcal{L}(\mathcal{AS})$ through a measure of the complexity of any tree $t \in \mathcal{L}(\mathcal{AS})$. We then demonstrate for each rule that if the left-hand side belongs to $\mathcal{L}(\mathcal{DS})$, then the right-hand side belongs to $\mathcal{L}(\mathcal{DS})$

Example 5 The optimized mail DTD (see Figure 5)

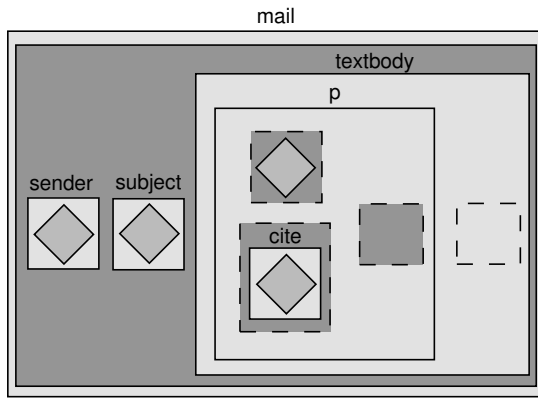


Figure 5. mail DTD, (simplification, but no pretty printing)

5. Pretty-Printing

overview. The principle of the pretty-printing we propose is (i) to establish visual analogies between nodes having the same depth in the tree structure, (ii) to make explicit node types with respect to the DTD semantics (e.g sequences are blue, alternations green). The first point is addressed through a fine tuning of block sizes that avoids pathological cases (e.g too thin rectangles). Figure 6 gives an intuition of the result for the mail DTD. Note that our representation space is based on a camera metaphor, which allows the user to zoom on any part of the observed sentence ([8, 12, 13]).

Definition 9 *Pretty-Printing over simplified terms*

$$\begin{aligned}
& \mathcal{P}[\cdot] : \mathcal{L}(AS) \times Int \times Int \rightarrow \mathcal{L}(AS) \times Int \times Int \\
& \mathcal{P}[\mathbf{Shp}(f, s, w, h, c)]_{W, H} = \mathbf{Shp}(f, s, w, h, c), \max(W, w), \max(H, h) \\
& \mathcal{P}[\mathbf{Box}(G, s, w, h, c)]_{W, H} = \mathbf{Box}(G', s, W', H', c), W', H' \\
& \text{with } G', w', h' = \mathcal{P}[G]_{W/1.2, H/1.2} \\
& \text{and } H' = \max(h' * 1.2, H), \quad W' = \max(w' * 1.2, W, H') \\
& \mathcal{P}[\mathbf{VA}(\mathbf{Box}(G_1, s_1, w_1, h_1, c_1), \mathbf{Box}(G_2, s_2, w_2, h_2, c_2), \mathbf{d}))]_{W, H} = \\
& \mathbf{VA}(G'_1, G'_2, \mathbf{0}), w, h \\
& \text{with } G'_1, w'_1, h'_1 = \mathcal{P}[\mathbf{Box}(G_1, s_1, w_1, h_1, c_1)]_{W, H} \\
& \text{and } G'_2, w'_2, h'_2 = \mathcal{P}[\mathbf{Box}(G_2, s_2, w_2, h_2, c_2)]_{w'_1, h'_1} \\
& \text{and } G'_1, w, h = \mathcal{P}[G'_1]_{w'_2, h'_2} \\
& \mathcal{P}[\mathbf{VA}(\mathbf{Box}(G_1, s_1, w_1, h_1, c_1), \mathbf{VA}(G_2, G_3, \mathbf{d}'), \mathbf{d}))]_{W, H} = \\
& \mathbf{VA}(G'_1, G_4, \mathbf{0}), w, h \\
& \text{with } G'_1, w'_1, h'_1 = \mathcal{P}[\mathbf{Box}(G_1, s_1, w_1, h_1, \mathbf{green})]_{W, H} \\
& \text{and } G_4, w_4, h_4 = \mathcal{P}[\mathbf{VA}(G_2, G_3, \mathbf{d}')]_{w'_1, h'_1} \\
& \text{and } G'_1, w, h = \mathcal{P}[G'_1]_{w_4, h_4} \\
& \mathcal{P}[\mathbf{VA}(\mathbf{Txt}(v), \mathbf{Box}(G_1, s_1, w_1, h_1, c_1), \mathbf{d}))]_{W, H} = \\
& \mathbf{VA}(\mathbf{Txt}(v), \mathbf{Box}(G'_1, s_1, w, h, c_1), h * 0.1), w, h \\
& \text{with } G'_1, w', h' = \mathcal{P}[G_1]_{W, H} \\
& \text{and } w = \max(\mathbf{BB}(\mathbf{Txt}(v)), w, w' * 1.2) \\
& \text{and } h = \max(\mathbf{BB}(\mathbf{Txt}(v)), h, h' * 1.2)
\end{aligned}$$

$$\begin{aligned}
& \mathcal{P}[\mathbf{HA}(G_1, G_2, \mathbf{d}))]_{W, H} = \mathbf{HA}(G'_1, G'_2, \mathbf{d}), w, h \\
& \text{with } G'_1, w'_1, h'_1 = \mathcal{P}[G_1]_{W, H} \\
& \text{and } G'_2, w'_2, h'_2 = \mathcal{P}[G_2]_{w'_1, h'_1} \\
& \text{and } G'_1, w, h = \mathcal{P}[G'_1]_{w'_2, h'_2}
\end{aligned}$$

$$\begin{aligned}
& \mathcal{P}[\mathbf{HA}(G_1, \mathbf{Shp}(\mathbf{rectangle}, \mathbf{dashed}, w, h, c), \mathbf{d}))]_{W, H} = \\
& \mathbf{HA}(G'_1, \mathbf{Shp}(\mathbf{rectangle}, \mathbf{dashed}, w', h'_1, c'), \mathbf{d}), w, h \\
& \text{with } G'_1, w'_1, h'_1 = \mathcal{P}[G_1]_{W, H} \\
& \text{and} \\
& \left\{ \begin{array}{l} \text{if } G'_1 = \mathbf{VA}(\mathbf{Box}(G, s, w_1, h_1, c_1), G, d_1) \\ \quad w' = \min(h_1, w_1), c' = c_1 \\ \text{if } G'_1 = \mathbf{Box}(G, s, w_1, h_1, c_1) \\ \quad w' = h'_1, c' = c_1 \\ \text{if } G'_1 = \mathbf{VA}(\mathbf{Txt}(v), \mathbf{Box}(G, s, w_1, h_1, c_1), d_1) \\ \quad w' = h'_1, c' = c_1 \\ \text{otherwise} \\ \quad w' = h'_1, c' = c \end{array} \right.
\end{aligned}$$

Property 6 *The pretty-printing of all term in $\mathcal{L}(\mathcal{DS})$ is a term in $\mathcal{L}(\mathcal{DS})$*

Proof 6 *By induction on derivation paths $D \xrightarrow{i} t$*

implementation. The representation obtained thanks to the pretty-printing function is not completely satisfactory. The layout algorithm implemented in VXT [8] is based on another function, too complex to be formally described in this paper. This function further enhances the user's perception of the structure, by assigning the same height to all nodes at the same absolute depth in the DTD tree, without taking into account the choice and sequence nodes, considered as constructors rather than true nodes of the DTD. Choice and sequence nodes are simply assigned a width and a height slightly bigger than their content. The result of this enhanced pretty-printing function is illustrated in Figure 6.

6. Discussion

Several authors [11, 5] pointed out the importance of visual abstract syntaxes, just as textual abstract syntaxes are enablers of formal treatments in "standard" language theory. Martin Erwig focuses on (visual) abstract syntaxes in [4], and proposes a graph-based approach (as many others, e.g [6]) to abstract over the spatial complexity inherent to visual formalisms.

We consider that for a subclass of VLs such as nested box languages, a tree formalism is well-suited to abstract over spatial information, and is more compact than a graph-based formalism. [4] provides an example of an abstract syntactic representation of a VEX expression (Figures 4 and 5) where the abstraction is *quantitatively* more complex than the concrete representation (10 graphical objects for the VEX expression versus 19+11 visual/textual objects). Of course the graph-based solution is useful because abstract over *qualitative* complexity. When tree-based approaches are applicable, they provide both *qualitative* and *quantitative* abstraction while also being grounded on a strong theoretical basis. An interesting question remains the expressiveness (and the limits) of tree-based approaches.

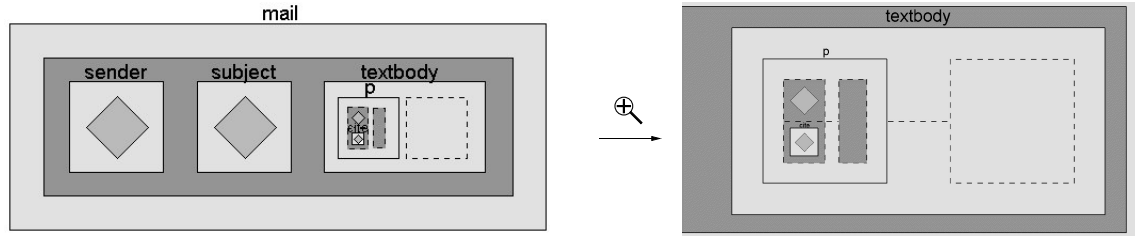


Figure 6. The final pretty-printed mail DTD

7. Conclusion

Martin Erwig underlined the importance of visual abstract syntaxes [5], and remarked that few papers deal with abstract syntaxes. We add that most of them rely on simplified graph-based syntactic formalisms [9]. In that respect, this paper shows that (i) tree-based visual abstract syntaxes can deal with a general class of visual language syntaxes through well-know proof technics, and (ii) that transformational approaches can help in finding the right abstraction level. The following diagram sums up the transformation architecture developed in this paper. The pivot abstract syntax \mathcal{AS} and its associated translation semantics \mathcal{VF} is quite general and can be used in other studies. We demonstrated through the simplification transformation \mathcal{O} and a pretty-printing transformation \mathcal{P} that the proposed approach produces realistic outputs while enabling formal treatments.

$$\begin{array}{ccccc}
 \mathcal{DTD} & \xrightarrow{\mathcal{T}} & \mathcal{L}(\mathcal{DS}) & \xrightarrow{\subseteq} & \mathcal{L}(\mathcal{AS}) & \xrightarrow{\mathcal{VF}} & \mathcal{L}(\mathcal{V}) \\
 & & \mathcal{O} \downarrow & & & & \\
 & & \mathcal{L}(\mathcal{DS}) & \xrightarrow{\subseteq} & \mathcal{L}(\mathcal{AS}) & \xrightarrow{\mathcal{VF}} & \mathcal{L}(\mathcal{V}) \\
 & & \mathcal{P} \downarrow & & & & \\
 & & \mathcal{L}(\mathcal{DS}) & \xrightarrow{\subseteq} & \mathcal{L}(\mathcal{AS}) & \xrightarrow{\mathcal{VF}} & \mathcal{L}(\mathcal{V})
 \end{array}$$

This work could be extended by proposing *visual soundness properties*, establishing relations between structural properties of DTDs and their effective perception by observers of the visual translation. For instance, showing that all element names of a DTD are visible, and that the parent-child relationship is actually translated by a nesting relation. Another soundness property could rely on operational view of DTDs, showing that if a (document) tree conforms to a DTD, its visual translation (to be defined through the pivot syntax \mathcal{AS}) conforms in some sense to the visual translation of its DTD. This should reveal that this essential property is captured by the visual language.

References

- [1] T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler. Extensible markup language (xml) 1.0 (second edition), October 2000. <http://www.w3.org/TR/2000/REC-xml-20001006>.
- [2] S. Ceri, S. Comai, E. Damiani, P. Fraternali, S. Paraboschi, and L. Tanca. Xml-gl: a graphical language for querying and restructuring xml documents. *8th WWW conference*, 1999.
- [3] N. Dershowitz and J.-P. Jouannaud. *Handbook of Theoretical computer Science - Rewrite Systems*, pages 243–320. Elsevier Science Publishers B.V., j. van leeuwen edition, 1990.
- [4] M. Erwig. Abstract visual syntax. In *2nd IEEE Int. Workshop on Theory of Visual Languages*, 1997.
- [5] M. Erwig. Abstract syntax and semantics of visual languages. *Journal of Visual Languages and Computing*, 1998.
- [6] D. Harel. On visual formalisms. *Commun. ACM* 31, 5:514–530, 1988.
- [7] OpenText. Near & far designer, March 2001. http://www.opentext.com/near_and_far/.
- [8] E. Pietriga and J.-Y. Vion-Dury. Vxt: Visual xml transformer. *IEEE Symposium on Visual/Multimedia Approaches to Programming and Software Engineering (Human Centric Computing Languages and Environments)*, 2001.
- [9] J. Reckers and A. Schürr. Defining and parsing visual languages with layered graph grammars. *Journal of Visual Languages and Computing - Academic Press*, 8:27–55, 1996.
- [10] J. W. Thatcher. *Tree Automata: An Informal Survey*, chapter 4, pages 143–172. Prentice-Hall, 1973.
- [11] S. Üsküdarlı and T. B. Dinesh. The vas formalism in vase. In *IEEE symposium on Visual Languages*, 1996.
- [12] J.-Y. Vion-Dury and F. Pacull. A structured workspace for a visual configuration language. In *Proceedings of VL'97*, Capri, Italy, 1997. IEEE symposium on Visual Languages.
- [13] J.-Y. Vion-Dury and M. Santana. Virtual images: Interactive visualization of distributed object-oriented systems. In *OOPSLA'94*, Portland, Oregon, USA, October 1994.
- [14] K. Wittenburg and L. Weitzman. Relational grammars: Theory and practice in a visual language interface for process modeling. *AVI '96 International Workshop on Theory of Visual Languages*, May 1996.

Introduction au formalisme des grammaires relationnelles

La théorie des langages visuels s’efforce de construire une base théorique pour expliquer les moyens de communication que sont les langages : comment un langage est formé (la syntaxe) et comment un langage transmet une signification (la sémantique) [50]. Elle s’intéresse donc aux formalismes visuels (diagrammes de Venn, cercles d’Euler, diagrammes d’état, hypergraphes, voir [116]), aux méthodes d’analyse (*parsing*, grammaires) et à l’interprétation des expressions visuelles.

Différents formalismes pour la spécification de langages visuels ont été étudiés, dont les grammaires. Dans le cas des langages textuels, uni-dimensionnels, il n’existe qu’une seule relation entre les symboles (*mots*) composant le langage : *suivant*. Décrire la syntaxe d’un langage textuel, par exemple en utilisant une grammaire BNF, ne nécessite pas de décrire les relations entre les symboles, puisque étant unique, la relation est implicite (dans un langage textuel, un élément est toujours écrit après le mot précédent). Les langages visuels étant multi-dimensionnels, les relations entre symboles doivent être spécifiées¹, et des méthodes de spécification comme les grammaires BNF ne sont plus assez puissantes. On a alors recours à d’autres formalismes comme les grammaires de formes, les grammaires positionnelles [62], relationnelles [236, 95], de multi-ensembles contraints (*CMGs : Constraint Multiset Grammars*) [118] et les grammaires de graphes [192].

La puissance expressive et la complexité des formalismes varient beaucoup de l’un à l’autre. Pour une grammaire positionnelle (système de réécriture travaillant sur un ensemble de symboles positionnés précisément dans l’espace), une production est de la forme

$$A \rightarrow x_1 R_1 x_2 R_2 \dots R_{m-1} x_m$$

avec A un non-terminal, x_i un terminal ou un non-terminal et R_i une relation spatiale constituée d’un couple de valeurs (dx, dy) indiquant que x_{i+1} doit se trouver aux coordonnées $(a + dx, b + dy)$ si x_i se trouve aux coordonnées (a, b) . Ce formalisme est donc relativement simple, mais aussi très limité quant aux langages qu’il est capable de capturer. Au contraire, le formalisme des *CCMGs* (*Copy-restricted Constraint Multiset Grammars*) a été utilisé comme base de la hiérarchie des formalismes grammaticaux de [153]², qui fournit pour chacun de ces formalismes une correspondance avec une restriction des *CCMGs*. La forme générale d’une production est

$$x \rightarrow X_1, \dots, X_n \text{ where exists } X'_1, \dots, X'_m \text{ where } C \text{ then } \vec{v} = E$$

et signifie que le non-terminal x peut être réécrit comme le multi-ensemble X_1, \dots, X_n si la phrase contient les symboles X'_1, \dots, X'_m (il s’agit du contexte) dont les attributs doivent satisfaire la contrainte C . \vec{v} est un vecteur composé d’attributs de x dont les valeurs sont égales aux valeurs d’attributs (E) d’autres éléments de la production.

D’une manière générale, la complexité du formalisme pose le problème de l’analysabilité grammaticale (*parsability*), et notamment du coût en temps de cette analyse (complexité algorithmique). Cependant, dans le cadre de l’étude théorique de VXT (voir Chapitre 4.4), nous ne tenons pas compte de ce

¹Parmi les autres relations, on trouve : *chevauche, est au dessus de, touche*.

²Une taxonomie des langages visuels basée sur la grammaire utilisée, présentée comme l’équivalent *VPL* de la hiérarchie de Chomsky pour les langages de programmation textuels.

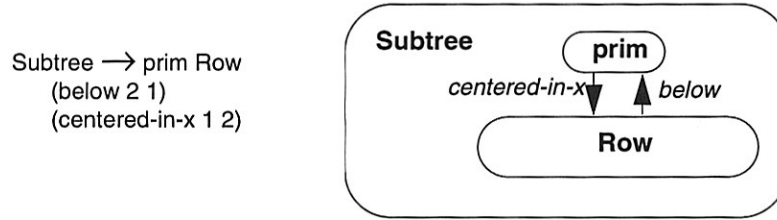


FIG. C.1 : Règle simple d'arrangement

critère puisque le formalisme utilisé n'a qu'un but descriptif : la représentation interne des programmes VXT est construite dynamiquement au cours de la spécification par le programmeur et ne requiert pas de phase d'analyse (reconnaissance) des expressions visuelles. Notre choix s'est porté sur les grammaires relationnelles qui fournissent le niveau d'expressivité nécessaire pour capturer les contraintes de notre langage tout en gardant un niveau de complexité assez bas.

Grammaires relationnelles

Les grammaires relationnelles [235, 236, 95] sont des systèmes de réécriture travaillant sur deux ensembles interdépendants : un ensemble de symboles et un ensemble de relations portant sur ces symboles. Elles ont par exemple été utilisées pour l'analyse d'expressions visuelles (expressions mathématiques, diagrammes). Plusieurs classes ont été définies par restriction des grammaires relationnelles, comme le formalisme des ARGs (*Atomic Relational Grammars*). Ces variations ont surtout un intérêt du point de vue de l'analyse³. Nous décrivons donc ici le formalisme général en l'illustrant par des exemples tirés de [236].

Définition 24 Une grammaire relationnelle est un tuple $G = (N, \Sigma, S, R, Attr, P)$ avec :

1. N un ensemble fini de symboles non-terminaux
2. Σ un ensemble fini de symboles terminaux, disjoint de N
3. S un symbole de N appelé symbole de départ (start symbol)
4. R un ensemble fini de relations
5. $Attr$ un ensemble fini d'attributs
6. P est un ensemble fini de productions de la forme $A \rightarrow \alpha/\beta/F$, avec
 - $A \in N$;
 - $\alpha \in (n|\sigma)^+$ avec $n \in N, \sigma \in \Sigma$;
 - β un ensemble de contraintes relationnelles de la forme $(r \ x \ y)$ avec $r \in R$ et x, y des entiers référencant un membre de α ou bien une expression de la forme $(a \ i)$ où $a \in Attr$ et i est un entier référencant un membre de α .
 - F un ensemble de déclarations d'assignation de la forme $(a \ 0) = x$ où $a \in Attr$ et x un entier référencant un membre de α ou bien une expression de la forme $(a \ i)$ décrite précédemment.

³Un algorithme basé sur les ARGs permet par exemple de commencer l'analyse d'une expression visuelle à partir de n'importe quel élément de cette expression.

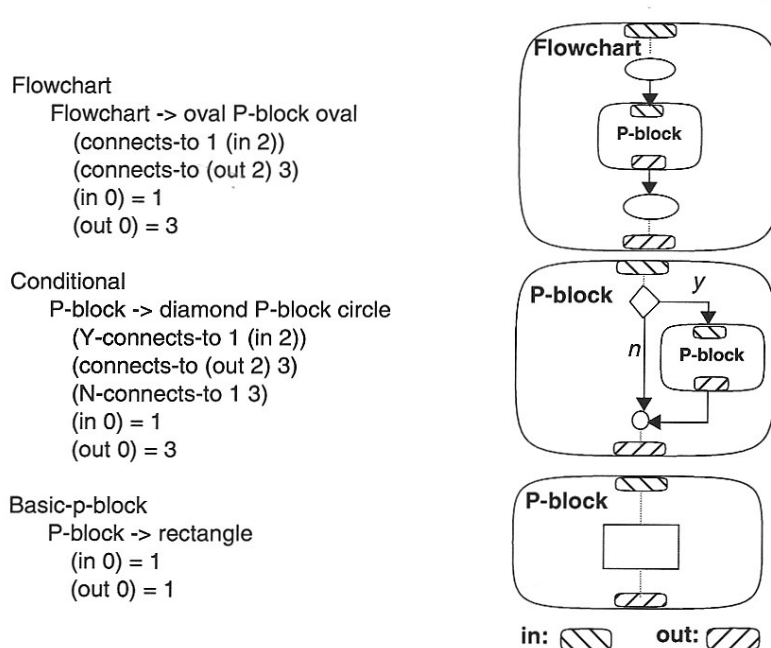


FIG. C.2 : Fragment de grammaire pour organigramme

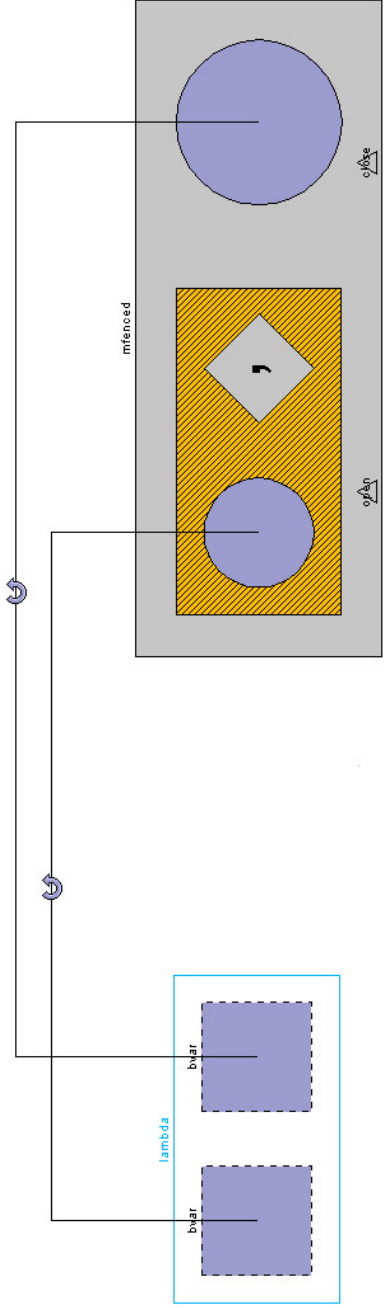
La figure C.1 est un exemple de production simple qui n'utilise pas d'attributs (l'illustration de droite est une représentation graphique de cette production). Les nombres 1 et 2 sont les index désignant respectivement *prim* et *Row* (l'index 0 est réservé au non-terminal $A \in N$ de la partie gauche de la règle). Celle-ci indique donc que *Subtree* peut être remplacé par les objets *prim* et *Row* avec les contraintes suivantes : *Row* doit se trouver sous (*below*) *prim*, *prim* doit être horizontalement centré (*centered-in-x*) par rapport à *Row*.

Un fragment de grammaire pour organigramme est représenté par la figure C.2 (la grammaire complète est fournie en appendice de [236]). Cet exemple utilise les attributs *in* et *out* (éléments hachurés dans la représentation graphique ; il ne s'agit pas d'objets graphiques du langage décrit par cette grammaire). La première production indique que *Flowchart* peut se réécrire comme un *oval* suivi d'un *P-Block* suivi d'un second *oval*. Le premier *oval* doit être connecté à l'attribut *in* du *P-Block*, l'attribut *out* au second *oval*. L'attribut *in* de *Flowchart* (partie gauche de la production) se voit assigné le premier *oval*, et l'attribut *out* le second *oval*.

Les attributs et les déclarations d'assignation (*statement assignement*) rendent ce formalisme assez expressif pour capturer les contraintes syntaxiques du langage VXT (Chapitre 4). La grammaire relationnelle de VXT a été spécifiée dans l'article présenté en Annexe B.

Exemple de vérification de partie droite de règle VXT

Nous développons ici un exemple de vérification de partie droite de règle VXT au moyen de la fonction VF3 (Chapitre 4, Définition 11). La règle de transformation utilisée est inspirée¹ de la transformation MathMLc2p [178]. Elle permet d'extraire tous les éléments `bvar` d'une lambda-fonction et de les présenter dans un élément `mfenced` en ajoutant une virgule entre chaque élément (nous ne donnons pas ici le détail des prédicats additionnels de la *VPME* permettant de faire la différence entre les n premiers éléments `bvar` et le dernier, après lequel il ne faut pas ajouter de virgule et qui est par conséquent traité en dehors de la boucle `for-each`).



| | | | | | | | | |
|---------------|---|-----------------|--|-----------------|--|---------------|---|-----------------|
| \vdash_{C2} | $\text{Shp}(\text{circle}, \text{colored})$ | \vdash_{C3} | $\text{Shp}(\text{lozenge}, \text{wired})$ | \vdash_{VF3a} | | \vdash_{C1} | $\text{Shp}(\text{triangle}, \text{wired})$ | \vdash_{VF3a} |
| | $\text{Ha}(\text{Shp}(\text{circle}, \text{colored}), \text{Shp}(\text{lozenge}, \text{wired}))$ | | $\text{Shp}(\text{lozenge}, \text{wired})$ | | | | $\text{Va}(\text{Txt}(\text{close}), \text{Shp}(\text{triangle}, \text{wired}))$ | \vdash_{VF3g} |
| \vdash_{C1} | $\text{Box}(\text{Ha}(\text{Shp}(\text{circle}, \text{colored}), \text{Shp}(\text{lozenge}, \text{wired})), \text{hatched})$ | \vdash_{VF3e} | $\text{Shp}(\text{circle}, \text{colored})$ | \vdash_{VF3b} | | \vdash_{C1} | $\text{Va}(\text{Txt}(\text{open}), \text{Shp}(\text{triangle}, \text{wired}))$ | \vdash_{VF3g} |
| | $\text{Ha}(\text{Box}(\text{Ha}(\text{Shp}(\text{circle}, \text{colored}), \text{Shp}(\text{lozenge}, \text{wired})), \text{hatched}), \text{Shp}(\text{circle}, \text{colored}))$ | | $\text{Shp}(\text{circle}, \text{colored})$ | | | | $\text{Ha}(\text{Va}(\text{Txt}(\text{close}), \text{Shp}(\text{triangle}, \text{wired})), \text{Va}(\text{Txt}(\text{close}), \text{Shp}(\text{triangle}, \text{wired})))$ | \vdash_{VF3h} |
| \vdash_{C1} | $\text{Box}(\text{Va}(\text{Ha}(\text{Box}(\text{Ha}(\text{Shp}(\text{circle}, \text{colored}), \text{Shp}(\text{lozenge}, \text{wired})), \text{hatched}), \text{Shp}(\text{circle}, \text{colored})), \text{hatched})$ | \vdash_{C1} | $\text{Ha}(\text{Shp}(\text{circle}, \text{colored}), \text{hatched})$ | \vdash_{VF3i} | | \vdash_{C1} | $\text{Ha}(\text{Va}(\text{Txt}(\text{open}), \text{Shp}(\text{triangle}, \text{wired})), \text{Va}(\text{Txt}(\text{close}), \text{Shp}(\text{triangle}, \text{wired})))$ | \vdash_{VF3h} |
| | $\text{Va}(\text{Box}(\text{Ha}(\text{Shp}(\text{circle}, \text{colored}), \text{Shp}(\text{lozenge}, \text{wired})), \text{hatched}), \text{Ha}(\text{Va}(\text{Txt}(\text{open}), \text{Shp}(\text{triangle}, \text{wired})), \text{Va}(\text{Txt}(\text{close}), \text{Shp}(\text{triangle}, \text{wired}))))$ | | $\text{Shp}(\text{circle}, \text{colored})$ | | | | $\text{Va}(\text{Va}(\text{Txt}(\text{open}), \text{Shp}(\text{triangle}, \text{wired})), \text{Va}(\text{Txt}(\text{close}), \text{Shp}(\text{triangle}, \text{wired}))))$ | \vdash_{VF3g} |

¹Le lecteur familier avec MathML notera sûrement qu'il existe une manière plus élégante de traiter ce problème en utilisant l'attribut `separators` des éléments `mfenced`; la solution retenue ici sert uniquement à illustrer l'emploi simple d'une boucle `for-each`.

Démonstrations

Cette annexe contient les démonstrations des propriétés de validité et de complétude établies sur la fonction de traduction \mathcal{T} générant des feuilles de transformation XSLT à partir de programmes VXT. Au cours des démonstrations, nous utilisons la notation :

$$a \xrightarrow{*} b$$

pour désigner un chemin de dérivation complet permettant d'atteindre b à partir de a .

E.1 Lemmes préliminaires

Nous introduisons les propriétés suivantes sur les fonctions *Pred* et *And*.

Lemme 1 (*Pred* engendre des expressions appartenant à $\mathcal{L}(XPath)$)

La fonction $Pred(a, b)$ peut produire soit a , soit $a[b]$

Comme la syntaxe abstraite $XPath$ définit

$$p \rightarrow p[q]$$

nous pouvons établir que si

$$XPath \xrightarrow{*} a \wedge eXQual \xrightarrow{*} b$$

alors

$$XPath \xrightarrow{*} Pred(a, b)$$

donc

$$(XPath \xrightarrow{*} a \wedge eXQual \xrightarrow{*} b) \implies XPath \xrightarrow{*} Pred(a, b)$$

Lemme 2 (*And* engendre des expressions appartenant à $\mathcal{L}(eXQual)$)

La fonction $And(a, b)$ peut produire a ou b ou $a \text{ and } b$ ou ε

Comme la syntaxe abstraite de $XPath$ définit

$$q \rightarrow q_1 \text{ and } q_2$$

nous pouvons établir que si

$$eXQual \xrightarrow{*} a \wedge eXQual \xrightarrow{*} b$$

alors

$$eXQual \xrightarrow{*} And(a, b)$$

donc

$$(eXQual \xrightarrow{*} a \wedge eXQual \xrightarrow{*} b) \implies eXQual \xrightarrow{*} And(a, b)$$

E.2 Correction syntaxique des expressions engendrées par \mathcal{T}_P

Preuve 1 (Par induction sur la structure)

$$\mathcal{T}_P[[v]] = x \implies x \in \mathcal{L}(eXQual)$$

Hypothèse d'induction $\mathcal{H}_k : \mathcal{T}_P[[X_k]] = x \wedge eXQual \xrightarrow{*} x$

Démonstration de \mathcal{H}_0 : nous développons l'exemple de la règle P_{14} . Les autres cas sont similaires.

$$\mathcal{T}_P[[\text{Shp}(\text{triangle}, s_9)]] = \text{not}(//@*)$$

et

$$eXQual \rightarrow XQual \rightarrow q \rightarrow \text{not}(q) \rightarrow \text{not}(p) \rightarrow \text{not}(/@p) \rightarrow \text{not}(//@*)$$

Induction $\mathcal{H}_k \implies \mathcal{H}_{k+1}$ pour $k \geq 0$ (comme précédemment, nous ne développons qu'un exemple représentatif, la règle P_{38}) :

$$\mathcal{T}_P[[\text{Va}(\text{Txt}(n), \text{Box}(X_k, s_5|s_6))]] = \text{Pred}(/@n, \mathcal{T}_P[[X_k]])$$

Par \mathcal{H}_k , nous savons que

$$eXQual \xrightarrow{*} \mathcal{T}_P[[X_k]]$$

Comme

$$XPath \rightarrow p \rightarrow /@p \rightarrow /@n$$

et que le lemme 1 établit que :

$$(XPath \xrightarrow{*} a \wedge eXQual \xrightarrow{*} b) \implies XPath \xrightarrow{*} \text{Pred}(a, b)$$

nous avons donc

$$eXQual \xrightarrow{*} \text{Pred}(/@n, \mathcal{T}_P[[X_k]])$$

puisque

$$eXQual \rightarrow eXPath \rightarrow XPath$$

E.3 Correction syntaxique des expressions engendrées par \mathcal{T}_M

Preuve 2 (Par induction sur la structure)

$$\mathcal{T}_M[[v]]_v = x \implies x \in \mathcal{L}(XPath)$$

Hypothèse d'induction $\mathcal{H}_k : \mathcal{T}_M[[X_k]]_v = x \wedge XPath \xrightarrow{*} x$

Démonstration de \mathcal{H}_0 : nous développons l'exemple de la règle M_5 . Les autres cas sont similaires.

$$\mathcal{T}_M[[\text{Shp}(\text{lozenge}, s_1|s_2|s_3|s_4)]]_v = \text{text}()$$

et

$$XPath \rightarrow p \rightarrow \text{text}()$$

Induction $\mathcal{H}_k \implies \mathcal{H}_{k+1}$ pour $k \geq 0$ (comme précédemment, nous ne développons qu'un exemple représentatif, la règle M_{17}) :

$$\mathcal{T}_M[\mathbf{Va}(\mathbf{Txt}(n), \mathbf{Box}_i(\mathbf{Va}(X_k, Y_k), s_7|s_8))]_v = \text{Pred}(n, \text{And}(\mathcal{T}_P[[X_k]], \mathcal{T}_P[[Y_k]])) / \mathcal{T}_M[[gscn(v, i, 0)]]_v$$

La preuve 1 et le lemme 2 nous permettent d'affirmer que

$$eXQual \xrightarrow{*} \text{And}(\mathcal{T}_P[[X_k]], \mathcal{T}_P[[Y_k]])$$

Alors, comme

$$XPath \rightarrow p \rightarrow n$$

nous obtenons par le lemme 1

$$XPath \xrightarrow{*} \text{Pred}(n, \text{And}(\mathcal{T}_P[[X_k]], \mathcal{T}_P[[Y_k]]))$$

Par définition de la fonction $gscn$ (Définition 7), et compte tenu du fait que dans l'expression

$$\mathbf{Va}(\mathbf{Txt}(n), \mathbf{Box}_i(\mathbf{Va}(X_k, Y_k), s))$$

l'index i référence le nœud qui précède immédiatement les nœuds X_k et Y_k sur les chemins engendrés par la fonction gp (Définition 6), nous avons

$$gscn(v, i, 0) = X_k \vee gscn(v, i, 0) = Y_k$$

Hors, par \mathcal{H}_k , nous savons que

$$XPath \xrightarrow{*} \mathcal{T}_M[[X_k]]_v \wedge XPath \xrightarrow{*} \mathcal{T}_M[[Y_k]]_v$$

ce qui nous permet d'affirmer que

$$XPath \xrightarrow{*} \mathcal{T}_M[[gscn(v, i, 0)]]_v$$

Alors, comme la syntaxe abstraite de $XPath$ définit

$$p \rightarrow p_1/p_2$$

nous avons

$$XPath \xrightarrow{*} \text{Pred}(n, \text{And}(\mathcal{T}_P[[X_k]], \mathcal{T}_P[[Y_k]])) / \mathcal{T}_M[[gscn(v, i, 0)]]_v$$

E.4 Correction syntaxique des expressions engendrées par \mathcal{T}_S

Pour prouver cette propriété, il faut prouver la correction syntaxique des trois fonctions \mathcal{T}_{S_m} (avec $m \in \{1, 2, 3\}$). Les expressions produites par ces fonctions doivent de plus être de la forme $/p$ ou $//p$, puisque la fonction \mathcal{T}_S produit des expressions de la forme :

$$\cdot \mathcal{T}_{S_m}[[v]]$$

Preuve 3 (Correction de \mathcal{T}_{S1} , par induction sur la structure)

$$\mathcal{T}_{S1}[[v]]_v^i = x \implies (XPath \rightarrow p \rightarrow /p \overset{*}{\rightarrow} x \vee XPath \rightarrow p \rightarrow //p \overset{*}{\rightarrow} x)$$

Hypothèse d'induction $\mathcal{H}_k : \mathcal{T}_{S1}[[X_k]]_v^i = x \wedge (XPath \rightarrow p \rightarrow /p \overset{*}{\rightarrow} x \vee XPath \rightarrow p \rightarrow //p \overset{*}{\rightarrow} x)$

Démonstration de \mathcal{H}_0 : nous développons l'exemple de la règle S_{1l} . Les autres cas sont similaires.

$$\mathcal{T}_{S1}[[\mathbf{Shp}_i(\mathbf{star}, s_5 \mid s_{11})]]_v^i = //\mathbf{node}()$$

et

$$XPath \rightarrow p \rightarrow //p \rightarrow //\mathbf{node}()$$

Induction $\mathcal{H}_k \implies \mathcal{H}_{k+1}$ pour $k \geq 0$ (comme précédemment, nous ne développons qu'un exemple représentatif, la règle S_{1u}) :

$$\mathcal{T}_{S1}[[\mathbf{Box}_j(\mathbf{Va}(X_k, Y_k), s_7 \mid s_8 \mid s_{12})]]_v^i = \mathit{Pred}(/*, \mathit{And}(\mathcal{T}_P[[X_k]], \mathcal{T}_P[[Y_k]])) \mathcal{T}_{S1}[[gscn(v, j, i)]]_v^i$$

La preuve 1 et le lemme 2 nous permettent d'affirmer que

$$eXQual \overset{*}{\rightarrow} \mathit{And}(\mathcal{T}_P[[X_k]], \mathcal{T}_P[[Y_k]])$$

Alors, comme

$$XPath \rightarrow p \rightarrow /p \rightarrow /*$$

nous obtenons par le lemme 1

$$XPath \rightarrow p \rightarrow /p \overset{*}{\rightarrow} \mathit{Pred}(/*, \mathit{And}(\mathcal{T}_P[[X_k]], \mathcal{T}_P[[Y_k]]))$$

Par définition de la fonction $gscn$ (Définition 7), et compte tenu du fait que dans l'expression

$$\mathbf{Box}_j(\mathbf{Va}(X_k, Y_k), s)$$

l'index j référence le nœud qui précède immédiatement les nœuds X_k et Y_k sur les chemins engendrés par la fonction gp (Définition 6), nous avons

$$gscn(v, j, i) = X_k \vee gscn(v, j, i) = Y_k$$

Hors, par \mathcal{H}_k , nous savons que

$$\begin{aligned} & (XPath \rightarrow p \rightarrow /p \overset{*}{\rightarrow} \mathcal{T}_{S1}[[X_k]]_v^i \vee XPath \rightarrow p \rightarrow //p \overset{*}{\rightarrow} \mathcal{T}_{S1}[[X_k]]_v^i) \\ \wedge & (XPath \rightarrow p \rightarrow /p \overset{*}{\rightarrow} \mathcal{T}_{S1}[[Y_k]]_v^i \vee XPath \rightarrow p \rightarrow //p \overset{*}{\rightarrow} \mathcal{T}_{S1}[[Y_k]]_v^i) \end{aligned}$$

ce qui nous permet d'affirmer que

$$XPath \rightarrow p \rightarrow /p \overset{*}{\rightarrow} \mathcal{T}_{S1}[[gscn(v, j, i)]]_v^i \vee XPath \rightarrow p \rightarrow //p \overset{*}{\rightarrow} \mathcal{T}_{S1}[[gscn(v, j, i)]]_v^i$$

et comme la syntaxe abstraite de $XPath$ définit

$$p \rightarrow p_1/p_2 \mid p_1//p_2$$

nous avons

$$XPath \rightarrow p \rightarrow /p \overset{*}{\rightarrow} \mathit{Pred}(/*, \mathit{And}(\mathcal{T}_P[[X_k]], \mathcal{T}_P[[Y_k]])) \mathcal{T}_{S1}[[gscn(v, j, i)]]_v^i$$

Preuve 4 (Correction de \mathcal{T}_{S_2} , par induction sur la structure)

$$\mathcal{T}_{S_2}[[v]]_v = x \implies (eXPath \rightarrow XPath \rightarrow p \rightarrow /p \xrightarrow{*} x \vee x = \varepsilon)$$

Hypothèse d'induction $\mathcal{H}_k : \mathcal{T}_{S_2}[[X_k]]_v = \varepsilon \vee (\mathcal{T}_{S_2}[[X_k]]_v = x \wedge eXPath \rightarrow XPath \rightarrow p \rightarrow /p \xrightarrow{*} x)$

Démonstration de \mathcal{H}_0 : nous développons l'exemple de la règle S_{2b} . Les autres cas sont similaires.

$$\mathcal{T}_{S_2}[[\mathbf{Box}_i(X, s_1 | s_2 | s_3 | s_4)]]_v = \varepsilon$$

Induction $\mathcal{H}_k \implies \mathcal{H}_{k+1}$ pour $k \geq 0$ (comme précédemment, nous ne développons qu'un exemple représentatif, la règle S_{2e}) :

$$\mathcal{T}_{S_2}[[\mathbf{Box}_j(X_k, s_7 | s_8 | s_{12})]]_v = \mathcal{T}_{S_2}[[gscn(v, j, 0)]]_v / \text{parent}(\text{Pred}(*, \mathcal{T}_P[[X_k]]))$$

La preuve 1 et le lemme 1 nous permettent d'affirmer que

$$XPath \xrightarrow{*} \text{Pred}(*, \mathcal{T}_P[[X_k]])$$

Alors, la syntaxe abstraite de XPath définissant

$$p \rightarrow /p$$

et

$$p \rightarrow \text{parent}(p)$$

nous avons

$$XPath \rightarrow p \rightarrow /p \xrightarrow{*} / \text{parent}(\text{Pred}(*, \mathcal{T}_P[[X_k]]))$$

Par définition de la fonction $gscn$ (Définition 7), et compte tenu du fait que dans l'expression

$$\mathbf{Box}_j(X_k, s)$$

l'index j référence le nœud qui précède immédiatement le nœud X_k sur les chemins engendrés par la fonction gp (Définition 6), nous avons

$$gscn(v, j, 0) = X_k$$

Hors, par \mathcal{H}_k , nous savons que

$$(\mathcal{T}_{S_2}[[X_k]]_v = \varepsilon) \vee (eXPath \rightarrow XPath \rightarrow p \rightarrow /p \xrightarrow{*} \mathcal{T}_{S_2}[[X_k]]_v)$$

ce qui nous permet d'affirmer que

$$(\mathcal{T}_{S_2}[[gscn(v, j, 0)]]_v = \varepsilon) \vee (eXPath \rightarrow XPath \rightarrow p \rightarrow /p \xrightarrow{*} \mathcal{T}_{S_2}[[gscn(v, j, 0)]]_v)$$

Finalement, nous avons bien

$$eXPath \rightarrow XPath \rightarrow p \rightarrow /p \xrightarrow{*} \mathcal{T}_{S_2}[[gscn(v, j, 0)]]_v / \text{parent}(\text{Pred}(*, \mathcal{T}_P[[X_k]]))$$

Preuve 5 (Correction de \mathcal{T}_{S_3})

$$\mathcal{T}_{S_3}[[v]]_v^i = x \implies (XPath \rightarrow p \rightarrow /p \xrightarrow{*} x \vee XPath \rightarrow p \rightarrow //p \xrightarrow{*} x)$$

Nous développons l'exemple de la règle S_{3a} . Les autres cas sont similaires.

$$\mathcal{T}_{S_3}[\mathbf{Box}_j(X, s_7|s_8|s_{12})]_v^i = \mathcal{T}_{S_2}[\mathit{gscn}(v, j, 0)]_v / * \mathcal{T}_{S_1}[\mathit{gscn}(v, j, i)]_v^i$$

Les preuves 3 et 4 nous permettent d'affirmer respectivement que

$$\mathcal{T}_{S_1}[[v]]_v^i = x \implies (XPath \rightarrow p \rightarrow /p \xrightarrow{*} x \vee XPath \rightarrow p \rightarrow //p \xrightarrow{*} x)$$

et

$$\mathcal{T}_{S_2}[[v]]_v = x \implies (eXPath \rightarrow XPath \rightarrow p \rightarrow /p \xrightarrow{*} x \vee x = \varepsilon)$$

Alors, comme la syntaxe abstraite de XPath nous donne

$$p \rightarrow /p \rightarrow /*$$

ainsi que

$$p \rightarrow p_1/p_2 \mid p_1//p_2$$

nous pouvons affirmer que

$$XPath \rightarrow p \rightarrow /p \xrightarrow{*} \mathcal{T}_{S_3}[\mathbf{Box}_j(X, s_7|s_8|s_{12})]_v^i$$

Preuve 6 (Correction de \mathcal{T}_S)

$$\mathcal{T}_S[[v]]^i = x \implies x \in \mathcal{L}(XPath)$$

Les expressions produites par \mathcal{T}_S sont soit . soit de la forme . $\mathcal{T}_{S_m}[[v]]$ avec $m = \{1, 2, 3\}$

Des preuves 3, 4 et 5 nous pouvons alors déduire que

$$XPath \rightarrow p \rightarrow . \mathcal{T}_{S_m}[[v]] \text{ avec } (XPath \rightarrow p \rightarrow /p \xrightarrow{*} \mathcal{T}_{S_m}[[v]] \vee XPath \rightarrow p \rightarrow //p \xrightarrow{*} \mathcal{T}_{S_m}[[v]])$$

Nous avons aussi

$$XPath \rightarrow p \rightarrow .$$

Or, comme

$$XPath \rightarrow p \rightarrow p_1/p_2$$

et

$$XPath \rightarrow p \rightarrow p_1//p_2$$

nous pouvons écrire

$$(XPath \rightarrow p \rightarrow p_1/p_2 \rightarrow ./p_2 \xrightarrow{*} \mathcal{T}_S[[v]]^i) \vee (XPath \rightarrow p \rightarrow p_1//p_2 \rightarrow ./p_2 \xrightarrow{*} \mathcal{T}_S[[v]]^i) \vee (\mathcal{T}_S[[v]]^i = .)$$

E.5 Validité des corps de règle engendrés par \mathcal{T}_R

Preuve 7 (Par induction sur la structure)

$$\mathcal{T}_R[[r]]_v^\Gamma = x \implies x \in \mathcal{XS}$$

Hypothèse d'induction $\mathcal{H}_k : \mathcal{T}_R[[X_k]]_v^\Gamma = x \wedge x \in \mathcal{XS}$

Démonstration de \mathcal{H}_0 : nous développons l'exemple de la règle R_{15} .

$$\mathcal{T}_{R1}[[\mathbf{Shp}_j(\mathbf{circle}, \mathbf{blue})]]_v^{\Gamma \subset \varphi_1, \langle i, j, \mathbf{apply-rules} \rangle} = \langle \mathbf{xsl:apply-templates select = "}\mathcal{T}_S[[v]]^i\text{"} / \rangle$$

Nous avons de manière évidente

$$\langle \mathbf{xsl:apply-templates}/ \rangle \in \mathcal{XS}$$

et la preuve 6 établit que

$$\mathcal{T}_S[[v]]^i \in \mathcal{L}(XPath)$$

donc

$$\langle \mathbf{xsl:apply-templates select = "}\mathcal{T}_S[[v]]^i\text{"} / \rangle \in \mathcal{XS}$$

Induction $\mathcal{H}_k \implies \mathcal{H}_{k+1}$ pour $k \geq 0$ (comme précédemment, nous ne développons qu'un exemple représentatif, la règle R_7) :

$$\mathcal{T}_{Rc}[[\mathbf{Box}_j(X_k, \mathbf{hatched})]]_v^{\Gamma \subset \varphi_4, \langle j, l \rangle} = \langle \mathbf{xsl:for-each select = "}\mathcal{T}_S[[v]]^l\text{"} / \rangle \mathcal{T}_{R2}[[X_k]]_v^\Gamma \langle / \mathbf{xsl:for-each} \rangle$$

Par \mathcal{H}_k , nous savons que

$$\mathcal{T}_R[[X_k]]_v^\Gamma \in \mathcal{XS}$$

et comme la preuve 6 établit que

$$\mathcal{T}_S[[v]]^l \in \mathcal{L}(XPath)$$

nous avons

$$\mathcal{T}_R[[\mathbf{Box}_j(X_k, \mathbf{hatched})]]_v^{\Gamma \subset \varphi_4, \langle j, l \rangle} \in \mathcal{XS}$$

puisque

$$\langle \mathbf{xsl:for-each} \rangle \langle / \mathbf{xsl:for-each} \rangle \in \mathcal{XS}$$

E.6 Présentation alternative de la syntaxe abstraite \mathcal{AS}_V

Nous présentons ici la syntaxe \mathcal{AS}_V sous une forme équivalente mais plus développée que dans le chapitre 4 et qui rendra plus évidente la preuve établissant la complétude de la fonction de traduction \mathcal{T}_P .

Définition 25 (Présentation alternative de la syntaxe abstraite \mathcal{AS}_V)

$$G_V \rightarrow B_V \mid \text{Shp}(\text{square}, s) \mid \text{Va}(\text{Txt}(t), \text{Shp}(\text{square}, s)) \mid \text{Shp}(\text{lozenge}, s) \mid \text{Shp}(\text{star}, s) \quad (25.1)$$

$$\mid \text{Shp}(\text{triangle}, s) \mid \text{Va}(\text{Txt}(t), \text{Shp}(\text{triangle}, s))$$

$$H_{VE} \rightarrow \text{Shp}(\text{square}, s) \mid \text{Va}(\text{Txt}(t), \text{Shp}(\text{square}, s)) \mid \text{Shp}(\text{lozenge}, s) \mid \text{Shp}(\text{star}, s) \quad (25.2)$$

$$H_{VE} \rightarrow \text{Ha}(B_V, H_{VE}) \quad (25.3)$$

$$H_{VE} \rightarrow \text{Ha}(B_V, B_V) \quad (25.4)$$

$$H_{VE} \rightarrow \text{Ha}(\text{Shp}(\text{square}, s), H_{VE}) \quad (25.5)$$

$$H_{VE} \rightarrow \text{Ha}(\text{Shp}(\text{square}, s), B_V) \quad (25.6)$$

$$H_{VE} \rightarrow \text{Ha}(\text{Va}(\text{Txt}(t), \text{Shp}(\text{square}, s)), H_{VE}) \quad (25.7)$$

$$H_{VE} \rightarrow \text{Ha}(\text{Va}(\text{Txt}(t), \text{Shp}(\text{square}, s)), B_V) \quad (25.8)$$

$$H_{VE} \rightarrow \text{Ha}(\text{Shp}(\text{lozenge}, s), H_{VE}) \quad (25.9)$$

$$H_{VE} \rightarrow \text{Ha}(\text{Shp}(\text{lozenge}, s), B_V) \quad (25.10)$$

$$H_{VE} \rightarrow \text{Ha}(\text{Shp}(\text{star}, s), H_{VE}) \quad (25.11)$$

$$H_{VE} \rightarrow \text{Ha}(\text{Shp}(\text{star}, s), B_V) \quad (25.12)$$

$$H_{VA} \rightarrow \text{Shp}(\text{triangle}, s) \mid \text{Va}(\text{Txt}(t), \text{Shp}(\text{triangle}, s)) \quad (25.13)$$

$$H_{VA} \rightarrow \text{Ha}(\text{Shp}(\text{triangle}, s), H_{VA}) \quad (25.14)$$

$$H_{VA} \rightarrow \text{Ha}(\text{Va}(\text{Txt}(t), \text{Shp}(\text{triangle}, s)), H_{VA}) \quad (25.15)$$

$$B_V \rightarrow \text{Box}(H_{VE}, s) \quad (25.16)$$

$$B_V \rightarrow \text{Box}(B_V, s) \quad (25.17)$$

$$B_V \rightarrow \text{Box}(H_{VA}, s) \quad (25.18)$$

$$B_V \rightarrow \text{Va}(\text{Txt}(t), \text{Box}(H_{VE}, s)) \quad (25.19)$$

$$B_V \rightarrow \text{Va}(\text{Txt}(t), \text{Box}(B_V, s)) \quad (25.20)$$

$$B_V \rightarrow \text{Va}(\text{Txt}(t), \text{Box}(H_{VA}, s)) \quad (25.21)$$

$$B_V \rightarrow \text{Box}(\text{Va}(H_{VE}, H_{VA}), s) \quad (25.22)$$

$$B_V \rightarrow \text{Box}(\text{Va}(B_V, H_{VA}), s) \quad (25.23)$$

$$B_V \rightarrow \text{Va}(\text{Txt}(t), \text{Box}(\text{Va}(H_{VE}, H_{VA}), s)) \quad (25.24)$$

$$B_V \rightarrow \text{Va}(\text{Txt}(t), \text{Box}(\text{Va}(B_V, H_{VA}), s)) \quad (25.25)$$

E.7 Complétude de la fonction \mathcal{T}_P

Preuve 8 (Par induction sur la structure)

$$\forall v \in \mathcal{L}(\mathcal{AS}_V), \exists x \mid (x = \mathcal{T}_P[v])$$

Hypothèse d'induction $\mathcal{H}_k : X_k \in \mathcal{L}(\mathcal{AS}_V) \implies \mathcal{T}_P[X_k]$ est défini

Démonstration de \mathcal{H}_0 : nous développons deux options de la première production (25.1) de la présentation alternative de la syntaxe abstraite $\mathcal{L}(\mathcal{AS}_V)$ (Définition 25).

| | | |
|------|---|----------------------------------|
| | $\mathcal{T}_P[\mathbf{Shp}(\mathbf{square}, s_1 s_2 s_3 s_4 s_7 s_8)]$ | est défini par [P ₁] |
| | $\mathcal{T}_P[\mathbf{Shp}(\mathbf{square}, s_5 s_6)]$ | est défini par [P ₃] |
| | $\mathcal{T}_P[\mathbf{Shp}(\mathbf{square}, s_9)]$ | est défini par [P ₄] |
| | $\mathcal{T}_P[\mathbf{Shp}(\mathbf{square}, s_{10})]$ | est défini par [P ₂] |
| | $\mathcal{T}_P[\mathbf{Shp}(\mathbf{square}, s_{11} s_{12})]$ | est défini par [P ₅] |
| donc | $\mathcal{T}_P[\mathbf{Shp}(\mathbf{square}, s)]$ | est défini |

| | | |
|------|---|-----------------------------------|
| | $\mathcal{T}_P[\mathbf{Va}(\mathbf{Txt}(t), \mathbf{Shp}(\mathbf{triangle}, s_1 s_2 s_3 s_4 s_7 s_8))]$ | est défini par [P ₁₆] |
| | $\mathcal{T}_P[\mathbf{Va}(\mathbf{Txt}(t), \mathbf{Shp}(\mathbf{triangle}, s_{10}))]$ | est défini par [P ₁₇] |
| | $\mathcal{T}_P[\mathbf{Va}(\mathbf{Txt}(t), \mathbf{Shp}(\mathbf{triangle}, s_5 s_6))]$ | est défini par [P ₁₈] |
| | $\mathcal{T}_P[\mathbf{Va}(\mathbf{Txt}(t), \mathbf{Shp}(\mathbf{triangle}, s_9))]$ | est défini par [P ₁₉] |
| | $\mathcal{T}_P[\mathbf{Va}(\mathbf{Txt}(t), \mathbf{Shp}(\mathbf{triangle}, s_{11} s_{12}))]$ | est défini par [P ₂₀] |
| donc | $\mathcal{T}_P[\mathbf{Va}(\mathbf{Txt}(t), \mathbf{Shp}(\mathbf{triangle}, s))]$ | est défini |

et ainsi de suite pour tous les terminaux de la grammaire \mathcal{AS}_V .

Induction $\mathcal{H}_k \implies \mathcal{H}_{k+1}$ pour $k \geq 0$ (nous ne développons que quelques exemples représentatifs) :

Par \mathcal{H}_k nous savons que $\mathcal{T}_P[X_k]$ et $\mathcal{T}_P[Y_k]$ sont définis. Alors

| | | |
|------|--|-----------------------------------|
| | $\mathcal{T}_P[\mathbf{Box}(X_k, s_1 s_2 s_3 s_4 s_7 s_8)]$ | est défini par [P ₃₁] |
| | avec $X_k \neq \mathbf{Va}(X_{k-1}, Y_{k-1})$ sauf $X_k = \mathbf{Va}(\mathbf{Txt}(n), X_{k-1})$ | |
| | $\mathcal{T}_P[\mathbf{Box}(X_k, s_{10})]$ | est défini par [P ₃₂] |
| | avec $X_k \neq \mathbf{Va}(X_{k-1}, Y_{k-1})$ sauf $X_k = \mathbf{Va}(\mathbf{Txt}(n), X_{k-1})$ | |
| | $\mathcal{T}_P[\mathbf{Box}(X_k, s_5 s_6)]$ | est défini par [P ₃₃] |
| | avec $X_k \neq \mathbf{Va}(X_{k-1}, Y_{k-1})$ sauf $X_k = \mathbf{Va}(\mathbf{Txt}(n), X_{k-1})$ | |
| | $\mathcal{T}_P[\mathbf{Box}(X_k, s_9)]$ | est défini par [P ₃₄] |
| | avec $X_k \neq \mathbf{Va}(X_{k-1}, Y_{k-1})$ sauf $X_k = \mathbf{Va}(\mathbf{Txt}(n), X_{k-1})$ | |
| | $\mathcal{T}_P[\mathbf{Box}(X_k, s_{11} s_{12})]$ | est défini par [P ₃₅] |
| | avec $X_k \neq \mathbf{Va}(X_{k-1}, Y_{k-1})$ sauf $X_k = \mathbf{Va}(\mathbf{Txt}(n), X_{k-1})$ | |
| donc | $\mathcal{T}_P[\mathbf{Box}(X_k, s)]$ | est défini |
| | avec $X_k \neq \mathbf{Va}(X_{k-1}, Y_{k-1})$ sauf $X_k = \mathbf{Va}(\mathbf{Txt}(n), X_{k-1})$ | |

et

| | | |
|------|---|-----------------------------------|
| | $\mathcal{T}_P[\mathbf{Box}(\mathbf{Va}(X_k, Y_k), s_1 s_2 s_3 s_4 s_7 s_8)]$ | est défini par [P ₄₁] |
| | avec $X_k \neq \mathbf{Txt}(n)$ | |
| | $\mathcal{T}_P[\mathbf{Box}(\mathbf{Va}(X_k, Y_k), s_{10})]$ | est défini par [P ₄₂] |
| | avec $X_k \neq \mathbf{Txt}(n)$ | |
| | $\mathcal{T}_P[\mathbf{Box}(\mathbf{Va}(X_k, Y_k), s_5 s_6)]$ | est défini par [P ₄₃] |
| | avec $X_k \neq \mathbf{Txt}(n)$ | |
| | $\mathcal{T}_P[\mathbf{Box}(\mathbf{Va}(X_k, Y_k), s_9)]$ | est défini par [P ₄₄] |
| | avec $X_k \neq \mathbf{Txt}(n)$ | |
| | $\mathcal{T}_P[\mathbf{Box}(\mathbf{Va}(X_k, Y_k), s_{11} s_{12})]$ | est défini par [P ₄₅] |
| | avec $X_k \neq \mathbf{Txt}(n)$ | |
| donc | $\mathcal{T}_P[\mathbf{Box}(\mathbf{Va}(X_k, Y_k), s)]$ | est défini |
| | avec $X_k \neq \mathbf{Txt}(n)$ | |

Nous venons de montrer que

$\mathcal{T}_P[\mathbf{Box}(X_k, s)]$ avec $X_k \neq \mathbf{Va}(X_{k-1}, Y_{k-1})$ sauf $X_k = \mathbf{Va}(\mathbf{Txt}(n), Y_{k-1})$ est défini

et que

$$\mathcal{J}_P \llbracket \mathbf{Box}(\mathbf{Va}(X_k, Y_k), s) \rrbracket \text{ avec } X_k \neq \mathbf{Txt}(n) \quad \text{est défini}$$

Nous pouvons en déduire que

$$\forall X_k, \mathcal{J}_P \llbracket \mathbf{Box}(X_k, s) \rrbracket \quad \text{est défini} \quad (\text{couvre les productions 25.16 à 25.18, 25.22 et 25.23})$$

Par \mathcal{H}_k nous savons que $\mathcal{J}_P \llbracket X_k \rrbracket$ et $\mathcal{J}_P \llbracket Y_k \rrbracket$ sont définis donc

$$\mathcal{J}_P \llbracket \mathbf{Ha}(X_k, Y_k) \rrbracket \quad \text{est défini par [P}_{51}] \quad (\text{couvre les productions 25.3 à 25.12, 25.14 et 25.15})$$

E.8 Chemins reliant la racine d'une VPME à son nœud contextuel

Nous présentons ici un système logique décrivant le chemin allant de la racine d'une VPME à son nœud contextuel. Ce système sera utilisé par la suite pour prouver la complétude de \mathcal{J}_M .

Définition 26 (Description du chemin reliant la racine d'une VPME à son nœud contextuel)

$$\begin{array}{ll} \frac{}{\vdash_D \mathbf{Shp}(f, \langle S, \mathbf{lightblue}, B_1, B_2 \rangle)} \text{ [VF4a]} & \frac{}{\vdash_D \mathbf{Va}(\mathbf{Txt}(t), \mathbf{Shp}(f, \langle S, \mathbf{lightblue}, B_1, B_2 \rangle))} \text{ [VF4b]} \\ \frac{}{\vdash_D \mathbf{Box}(X, \langle S, \mathbf{lightblue}, B_1, B_2 \rangle)} \text{ [VF4c]} & \frac{}{\vdash_D \mathbf{Va}(\mathbf{Txt}(t), \mathbf{Box}(X, \langle S, \mathbf{lightblue}, B_1, B_2 \rangle))} \text{ [VF4d]} \\ \frac{\vdash_D X \quad C \neq \mathbf{lightblue} \quad C \neq \mathbf{red}}{\vdash_D \mathbf{Box}(X, \langle S, C, B_1, B_2 \rangle)} \text{ [VF4e]} & \frac{\vdash_D X \quad C \neq \mathbf{lightblue} \quad C \neq \mathbf{red}}{\vdash_D \mathbf{Va}(\mathbf{Txt}(t), \mathbf{Box}(X, \langle S, C, B_1, B_2 \rangle))} \text{ [VF4f]} \end{array}$$

Avant de pouvoir utiliser ce système logique pour montrer la complétude de \mathcal{J}_P , il est nécessaire de prouver la propriété suivante :

$$\vdash_{A1} v \wedge \vdash_{B1} v \implies \vdash_D v$$

Pour cela, nous présentons une autre forme du critère VF2 de manière à faire apparaître les constituants du système précédent (Définition 26).

Définition 27 (Critère de validation VF2A (forme alternative de VF2)) $i \in \{1, 2\}$

$$\begin{array}{c}
\frac{}{\vdash_{B_1} \mathbf{Shp}(f, \langle S, \mathbf{lightblue}, B_1, B_2 \rangle)} \text{ [VF2Aa]} \quad \frac{}{\vdash_{B_1} \mathbf{Va}(\mathbf{Txt}(t), \mathbf{Shp}(f, \langle S, \mathbf{lightblue}, B_1, B_2 \rangle))} \text{ [VF2Ab]} \\
\frac{C \neq \mathbf{lightblue}}{\vdash_{B_2} \mathbf{Shp}(f, \langle S, C, B_1, B_2 \rangle)} \text{ [VF2Ac]} \quad \frac{C \neq \mathbf{lightblue}}{\vdash_{B_2} \mathbf{Va}(\mathbf{Txt}(t), \mathbf{Shp}(f, \langle S, C, B_1, B_2 \rangle))} \text{ [VF2Ad]} \\
\frac{\vdash_{B_2} X}{\vdash_{B_1} \mathbf{Box}(X, \langle S, \mathbf{lightblue}, B_1, B_2 \rangle)} \text{ [VF2Ae]} \quad \frac{\vdash_{B_2} X}{\vdash_{B_1} \mathbf{Va}(\mathbf{Txt}(t), \mathbf{Box}(X, \langle S, \mathbf{lightblue}, B_1, B_2 \rangle))} \text{ [VF2Af]} \\
\frac{\vdash_{B_1} X \quad C \neq \mathbf{lightblue}}{\vdash_{B_1} \mathbf{Box}(X, \langle S, C, B_1, B_2 \rangle)} \text{ [VF2Ag]} \quad \frac{\vdash_{B_1} X \quad C \neq \mathbf{lightblue}}{\vdash_{B_1} \mathbf{Va}(\mathbf{Txt}(t), \mathbf{Box}(X, \langle S, C, B_1, B_2 \rangle))} \text{ [VF2Ah]} \\
\frac{\vdash_{B_2} X \quad C \neq \mathbf{lightblue}}{\vdash_{B_2} \mathbf{Box}(X, \langle S, C, B_1, B_2 \rangle)} \text{ [VF2Ai]} \quad \frac{\vdash_{B_2} X \quad C \neq \mathbf{lightblue}}{\vdash_{B_2} \mathbf{Va}(\mathbf{Txt}(t), \mathbf{Box}(X, \langle S, C, B_1, B_2 \rangle))} \text{ [VF2Aj]} \\
\frac{\vdash_{B_i} X_1 \quad \vdash_{B_i} X_2}{\vdash_{B_i} \mathbf{Ha}(X_1, X_2)} \text{ [VF2Ak]} \quad \frac{\vdash_{B_i} X_1 \quad \vdash_{B_i} X_2}{\vdash_{B_i} \mathbf{Va}(X_1, X_2)} \text{ [VF2Al]}
\end{array}$$

Nous pouvons alors facilement mettre en correspondance VF4 et VF2A :

$$\begin{array}{l}
\text{[VF4a]} \leftrightarrow \text{[VF2Aa]} \\
\text{[VF4b]} \leftrightarrow \text{[VF2Ab]} \\
\text{[VF4c]} \leftrightarrow \text{[VF2Ac]} \\
\text{[VF4d]} \leftrightarrow \text{[VF2Ad]} \\
\text{[VF4e]} \leftrightarrow \text{[VF2Ae]} \\
\text{[VF4f]} \leftrightarrow \text{[VF2Af]}
\end{array}$$

sachant que les expressions $C \neq \mathbf{red}$ dans les numérateurs de [VF4e] et [VF4f] proviennent des constituants [VF1d] et [VF1g] du critère VF1, qui peuvent être réécrits comme suit (nous ne développons que les deux qui nous intéressent) :

$$\frac{\vdash_{A_1} X \quad C \neq \mathbf{red}}{\vdash_{A_1} \mathbf{Box}(X, \langle S, C, B_1, B_2 \rangle)} \text{ [VF1Ag]} \quad \frac{\vdash_{A_1} X \quad C \neq \mathbf{red}}{\vdash_{A_1} \mathbf{Va}(\mathbf{Txt}(t), \mathbf{Box}(X, \langle S, C, B_1, B_2 \rangle))} \text{ [VF1Ah]}$$

Nous avons donc bien

$$\vdash_{A_1} v \wedge \vdash_{B_1} v \implies \vdash_D v$$

E.9 Complétude de la fonction \mathcal{T}_M

Contrairement à la fonction \mathcal{T}_P étudiée précédemment, la fonction \mathcal{T}_M n'est pas définie pour toutes les constructions autorisées dans une *VPME* bien formée (suivant les critères de validation \vdash_{A_1} et \vdash_{B_1}), puisqu'elle est spécialisée dans la traduction des nœuds se trouvant sur le chemin allant de la racine de la *VPME* au nœud contextuel. Ces chemins ont été capturés par le système logique introduit dans la

définition 26 (noté \vdash_D). Nous montrons maintenant la complétude de la fonction \mathcal{T}_M par rapport à ce système.

Preuve 9 (Par induction sur la structure)

$$\forall c \mid \vdash_D c, \exists x \mid (x = \mathcal{T}_M[[c]])$$

Hypothèse d'induction $\mathcal{H}_k : \vdash_D X_k \implies \mathcal{T}_M[[X_k]]$ est défini

Démonstration de \mathcal{H}_0 :

| | | |
|--|--------|--|
| Shp ($f, \langle S, \text{lightblue}, B_1, B_2 \rangle$) | [VF4a] | est défini par $[M_1],[M_3],[M_5],[M_6]$ |
| Va (Txt (t), Shp ($f, \langle S, \text{lightblue}, B_1, B_2 \rangle$)) | [VF4b] | est défini par $[M_2],[M_4]$ |
| Box ($X, \langle S, \text{lightblue}, B_1, B_2 \rangle$) | [VF4c] | est défini par $[M_7],[M_{13}]$ |
| Va (Txt (t), Box ($X, \langle S, \text{lightblue}, B_1, B_2 \rangle$)) | [VF4d] | est défini par $[M_{10}],[M_{16}]$ |

(les cas $[M_7],[M_{10}],[M_{13}],[M_{16}]$ de la Définition 18 montrent que les sous-arbres désignés ici par X sont traduits par la fonction \mathcal{T}_P)

Induction $\mathcal{H}_k \implies \mathcal{H}_{k+1}$ pour $k \geq 0$:

Par \mathcal{H}_k nous savons que $\mathcal{T}_M[[X_k]]$ est défini. Alors

| | | |
|---|--------|--|
| Box ($X_k, \langle S, C, B_1, B_2 \rangle$) | [VF4e] | est défini par $[M_8],[M_9],[M_{14}],[M_{15}]$ |
| Va (Txt (t), Box ($X_k, \langle S, C, B_1, B_2 \rangle$)) | [VF4f] | est défini par $[M_{11}],[M_{12}],[M_{17}],[M_{18}]$ |

E.10 Complétude de la fonction \mathcal{T}_S

Nous devons d'abord prouver la complétude des fonctions \mathcal{T}_{S_m} .

Preuve 10 (Complétude des fonctions $\mathcal{T}_{S_1}, \mathcal{T}_{S_2}$ et \mathcal{T}_{S_3})

$$\forall v \in \mathcal{L}(\mathcal{AS}_V), \exists x \mid (x = \mathcal{T}_{S_m}[[v]]) \text{ avec } m \in \{1, 2, 3\}$$

La méthode de démonstration est la même que celle employée pour la fonction \mathcal{T}_M : par induction sur la structure, après avoir défini des systèmes logiques décrivant les fragments de VPME traduits par chacune des fonctions. Ces preuves ne sont pas détaillées ici pour des raisons de place.

En utilisant la preuve précédente (non développée) établissant la complétude des fonctions \mathcal{T}_{S_m} avec $m \in \{1, 2, 3\}$, et en se basant sur la définition de la fonction \mathcal{T}_S (Définition 20) reprise ci-dessous, nous pouvons établir la complétude de cette dernière.

$$\mathcal{T}_S[\]^i : \mathcal{L}(\mathcal{AS}_V) \rightarrow \mathcal{L}(XPath)$$

$$\mathcal{T}_S[v]^i \text{ avec } i = 0 = . \quad (S_{0a})$$

$$\mathcal{T}_S[v]^i \text{ avec } \text{card}(gp(v, 0, i)) \geq 2 = . \mathcal{T}_{S1}[gscn(v, 0, i)]_v^i \quad (S_{0b})$$

$$\mathcal{T}_S[v]^i \text{ avec } \text{card}(gp(v, i, 0)) \geq 2 = . \mathcal{T}_{S2}[i2v(v, i)]_v \quad (S_{0c})$$

$$\mathcal{T}_S[v]^i \text{ avec } \text{card}(gp(v, i, 0)) = \text{card}(gp(v, 0, i)) = 0 \wedge i \neq 0 = . \mathcal{T}_{S3}[i2v(v, gcca(v, i))]_v^i \quad (S_{0d})$$

Preuve 11 (Complétude de \mathcal{T}_S)

$$\forall v \in \mathcal{L}(\mathcal{AS}_V) \wedge \forall i \mid (i \geq 0 \wedge i \in I), \exists x \mid (x = \mathcal{T}_S[v]^i)$$

Dans les expressions $\mathcal{T}_S[v]^i$, i représente l'index du nœud à extraire et il doit appartenir à l'ensemble des index de la VPME noté I (voir la Définition 5). Alors, en utilisant les définitions des fonctions card et gp (Définitions 6 et 20), nous avons :

(S_{0a}) traite le cas où le nœud à extraire est le nœud contextuel ($i = 0$),

(S_{0b}) traite le cas où le nœud à extraire est un descendant du nœud contextuel ($\text{card}(gp(v, 0, i)) \geq 2$),

(S_{0c}) traite le cas où le nœud à extraire est un ancêtre du nœud contextuel ($\text{card}(gp(v, i, 0)) \geq 2$),

(S_{0d}) traite les autres solutions, c'est-à-dire les cas où le nœud contextuel et le nœud sélectionné appartiennent à des branches différentes mais ont un ancêtre commun ($\text{card}(gp(v, i, 0)) = \text{card}(gp(v, 0, i)) = 0 \wedge i \neq 0$).

Puisque $i = 0 \implies \text{card}(gp(v, i, 0)) = \text{card}(gp(v, 0, i)) = 1$, l'ensemble des chemins possibles du nœud contextuel au nœud sélectionné est couvert par les quatre cas S_{0a} à S_{0d} . Nous en déduisons que

$$\forall v \in \mathcal{L}(\mathcal{AS}_V) \wedge \forall i \mid (i \geq 0 \wedge i \in I), \exists x \mid (x = \mathcal{T}_S[v]^i)$$

Nous rappelons que I est l'ensemble des index de la VPME (voir la Définition 5).

E.11 Complétude de la fonction \mathcal{T}_R

Preuve 12 (O) Nous devons ici raisonner sur l'ensemble des productions appartenant à $\mathcal{L}(\mathcal{AS}_R)$ et vérifiant VF3. Il est nécessaire pour établir la preuve de réécrire VF3 avec des dénominateurs plus proches des cas de \mathcal{T}_R . Nous ne donnons ici que quelques exemples, la forme alternative de VF3 et la preuve complète étant très longues.

$$\forall r \mid (\vdash_{C1} r), \exists x \mid (x = \mathcal{T}_R[r])$$

Hypothèse d'induction $\mathcal{H}_k : \vdash_{C_1} X_k \implies \mathcal{T}_R[X_k]$ est défini

Démonstration de \mathcal{H}_0 :

| | |
|---|---|
| $\mathcal{T}_R[\mathbf{Shp}(\mathbf{lozenge}, \mathbf{gray})]$ | est défini par $[R_4]$ |
| $\mathcal{T}_R[\mathbf{Va}(\mathbf{Txt}(t), \mathbf{Shp}(\mathbf{triangle}, \mathbf{gray}))]$ | est défini par $[R_3]$ |
| $\mathcal{T}_R[\mathbf{Shp}(\mathbf{lozenge}, \mathbf{blue})]$ | est défini par $[R_{11}], [R_{12}], [R_{18}], [R_{19}]$ |

et ainsi de suite pour tous les terminaux.

Induction $\mathcal{H}_k \implies \mathcal{H}_{k+1}$ pour $k \geq 0$ (nous ne développons que quelques exemples représentatifs) :

| | |
|--|--|
| $\mathcal{T}_R[\mathbf{Va}(X_k, Y_k)]$ | est défini par $[R_1]$ puisque par \mathcal{H}_k nous savons que $\mathcal{T}_R[X_k]$ et $\mathcal{T}_R[Y_k]$ est défini |
| $\mathcal{T}_R[\mathbf{Box}(X_k, \mathbf{hatched})]$ | est défini par $[R_7]$ puisque par \mathcal{H}_k nous savons que $\mathcal{T}_R[X_k]$ est défini |

Bibliographie

- [1] Sharon Adler, Anders Berglund, Jeff Caruso, Stephen Deach, Tony Graham, Paul Grosso, Eduardo Gutentag, Alex Milowski, Scott Parnell, Jeremy Richman, and Steve Zilles. Extensible stylesheet language (XSL) version 1.0, 15 October 2001. <http://www.w3.org/TR/xsl/>
- [2] Adobe Photoshop. <http://www.adobe.com/products/photoshop/main.html>, March 2002.
- [3] A. Aho, R. Sethi, and J. Ullman. *Compilers*. Addison-Wesley, 1986.
- [4] Altova. XML Spy, July 2002. <http://www.xmlspy.com>
- [5] Allen L. Ambler, Thomas Green, Takayuki Dan Kumura, Alexander Repenning, and Trevor Smedley. 1997 Visual Programming Challenge Summary. In *Proceedings of Visual Languages 1997*, Capri, Italy, 1997.
- [6] A. L. Ambler and M. M. Burnett. Influence of visual technology on the evolution of language environments. *IEEE Computer*, 6(2) :9–22, October 1989.
- [7] Jean-Marc Andreoli, François Pacull, Daniele Pagani, and Remo Pareschi. Multiparty Negotiation for Dynamic Distributed Object Services. *Journal of Science of Computer Programming*, 31 :179–203, 1998.
- [8] M. Andries and G. Engels. A Hybrid Query Language for an Extended Entity-Relationship Model. *Journal of Visual Languages and Computing*, 7(3) :321, 1996.
- [9] M. Apperley, R. Spence, and K. Wittenburg. Selecting one from many : The development of a scalable visualization tool. In IEEE Computer Society, editor, *IEEE Symposia on Human-Centric Computing (HCC)*, pages 366–372, Stresa, Italy, September 2001.
- [10] Greg J. Badros and Alan Borning. The Cassowary linear arithmetic constraint solving algorithm : Interface and implementation. Technical Report UW-CSE-98-06-04, University of Washington, June 1998.
- [11] Ed Baroth and Chris Hartsough. *Visual Object Oriented Programming*, chapter 2, pages 21–42. Manning, 1995.
- [12] Ed Baroth and Chris Hartsough. *Visual Object Oriented Programming*, chapter 7, pages 129–159. Manning, 1995.
- [13] François Bayle. *Musique acousmatique, propositions... ..positions*. Number 2-7020-1584-0 in ISBN. Buchet/Chastel/INA-GRM, 1993.

- [14] Dave Beckett. RDF/XML syntax specification (revised). W3C Working Draft, March 2002. <http://www.w3.org/TR/rdf-syntax-grammar/>
- [15] B.B. Bederson and J.D. Hollan. Pad++ : A zooming graphical interface for exploring alternate interface physics. In *Proceedings of the ACM symposium on User Interface software and technology (UIST)*, 1994.
- [16] B.B. Bederson and J.D. Hollan. Advances in the Pad++ zoomable graphics widget. In *CHI'95, ACM Conference on Human Factors in Computing Systems*, 1995.
- [17] B.B. Bederson, J. Meyer, and L. Good. Jazz : An extensible zoomable user interface graphics toolkit in Java. In *Proceedings of the ACM symposium on User Interface software and technology (UIST)*, pages 171–180, 2000.
- [18] Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernandez, Michael Kay, Jonathan Robie, and Jérôme Siméon (Editors). Xml path language (xpath) 2.0. W3C Working Draft, 30 April 2002. <http://www.w3.org/TR/xpath20/>
- [19] Tim Berners-Lee. Web architecture from 50,000 feet, October 1999. <http://www.w3.org/DesignIssues/Architecture.html>
- [20] Tim Berners-Lee. Notation3 (ideas about web architecture - yet another notation), November 2001. <http://www.w3.org/DesignIssues/Notation3.html>
- [21] Tim Berners-Lee. CWM - a general purpose data processor for the semantic web, May 2002. <http://www.w3.org/2000/10/swap/doc/cwm.html>
- [22] Tim Berners-Lee, R. Fielding, and L. Massinter. RFC 2396 - Uniform Resource Identifiers (URI) : Generic syntax. IETF, August 1998. <http://www.isi.edu/in-notes/rfc2396.txt>
- [23] Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web. *Scientific American*, May 2001.
- [24] J. Bertin. *Sémiologie graphique*. Number 2-7132-1277-4 in ISBN. Mouton, Paris, 1973.
- [25] P. V. Biron and A. Malhotra. XML Schema part 2 : Datatypes. W3C Recommendation, May 2001. <http://www.w3.org/TR/xmlschema-2/>
- [26] A. Blackwell. *Your Wish is My Command : Giving Users the Power to Instruct their Software*, chapter SWYN : a visual representation for regular expressions. M. Kaufmann, 2000.
- [27] S. Bonhomme. *Transformation de documents structurés : une combinaison des approches explicite et automatique*. PhD thesis, Université Joseph Fourier-Grenoble I, Décembre 1998.
- [28] A. Bonifati and S. Ceri. Comparative analysis of five XML query languages. *SIGMOD Record*, 29(1) :68–79, 2000.
- [29] Borland. Jbuilder, March 2002. <http://www.borland.com/jbuilder/>
- [30] A. Borning. Defining constraints graphically. In *SIGCHI*, pages 137–143, Boston, MA, 1986.
- [31] P. Bottoni, M.F. Costabile, D. Fogli, S. Levialdi, and P. Mussio. Multilevel modelling and design of visual interactive systems. In IEEE Computer Society, editor, *IEEE Symposia on Human-Centric Computing (HCC), Symposium on Visual/Multimedia Approaches to Programming and Software Engineering*, pages 256–263, Stresa, Italy, September 2001.

- [32] P. Bottoni, M.F. Costabile, S. Levialdi, and P. Mussio. From visual language specification to legal visual interaction. In *Proceedings of Visual Languages 1997*, Capri, Italy, 1997.
- [33] P. Bottoni, M.F. Costabile, and P. Mussio. Specification and dialogue control of visual interaction through visual rewriting systems. *ACM Transactions on Programming Languages and Systems*, 21(6) :1077–1136, November 1999.
- [34] T. Bray, C. Frankston, and A. Malhotra (editors). Document content description for XML. W3C Note, July 1998. <http://www.w3.org/TR/NOTE-dcd>
- [35] T. Bray, J. Paoli, C. M. Sperberg-McQueen, and Eve Maler. Extensible markup language (XML) 1.0 (second edition), October 2000. <http://www.w3.org/TR/2000/REC-xml-20001006>
- [36] Tim Bray, Dave Hollander, and Andrew Layman. Namespaces in XML. W3C Recommendation, 14 January 1999. <http://www.w3.org/TR/1999/REC-xml-names-19990114/>
- [37] D. Brickley and R. V. Guha. RDF vocabulary description language 1.0 : RDF Schema. W3C Working Draft, April 2002. <http://www.w3.org/TR/rdf-schema/>
- [38] S. Briet. *Qu'est-ce que la documentation*. EDIT, Paris, 1951.
- [39] M. H. Brown and M. A. Najork. Algorithm animation using 3D interactive graphics. In *Proceedings of the 6th annual ACM symposium on User Interface software and technology (UIST)*, pages 93–100, Atlanta, USA, 1993.
- [40] E. Bruno, J. Le Maître, and E. Murisasco. Querying XML data : the DQL language. In *10th WWW conference*, Hong Kong, 2001.
- [41] M. Buckland. What is a digital document. *Document numérique*, 23(11), November 1990.
- [42] M. Burnett and H. Gottfried. Graphical definitions : expanding spreadsheet languages through direct manipulation and gestures. *ACM Transaction on Computer-Human Interaction*, 5(1), March 1998.
- [43] Margaret M. Burnett. What is visual programming. In *Encyclopedia of Electrical and Electronics Engineering (John G. Webster ed.)*, New York, 1999.
- [44] J. Carroll. Unparsing RDF/XML. In *Proceedings of the 11th World Wide Web Conference*, page 184, Honolulu, Hawaii, 2002.
- [45] R. Casellas. db2latex, January 2002. <http://db2latex.sourceforge.net/>
- [46] T. Catarci, S.K. Chang, M.F. Costabile, S. Levialdi, and G. Santucci. A multiparadigmatic visual environment for adaptative access to databases. In *Adjunct proceedings of the ACM conference on Human Factors in Computing Systems, INTERACT'93 and CHI'93*, 1993.
- [47] S. Ceri, S. Comai, E. Damiani, P. Fraternali, S. Paraboschi, and L. Tanca. XML-GL : a graphical language for querying and restructuring XML documents. In *8th WWW conference*, Toronto, Canada, 1999.
- [48] Don Chamberlin, Jonathan Robie, and Daniela Florescu. Quilt : An (XML) query language for heterogeneous data sources. *Lecture Notes in Computer Science*, 1997 :1–??, 2001.
- [49] B.W. Chang and D. Ungar. Animation : from cartoons to the user interface. In *UIST'93 Conference Proceedings*, pages 45–55, Atlanta, USA, 1993.

- [50] S.K. Chang. Ten years of visual languages research. In IEEE Computer Society Press, editor, *Proceedings of the IEEE symposium on Visual Languages*, pages 196–205, St Louis, Missouri, USA, 1994.
- [51] S.K. Chang, M. Burnett, S. Levialdi, K. Marriott, J. J. Pfeiffer, and S. Tanimoto. The future of visual languages. In IEEE, editor, *Proceedings of the symposium on visual languages*, pages 58–61, Tokyo, Japan, 1999.
- [52] W. Citrin, M. Doherty, and B. Zorn. Control constructs in a completely visual imperative programming language. Technical Report CU-CS-673-93, University of Colorado, Boulder, September 1993.
- [53] Wayne Citrin, Daniel Broodsky, and Jeffrey McWhirter. Style-based cut-and-paste in graphical editors. In ACM, editor, *Proceedings of the workshop on Advanced visual interfaces*, page 105, Bari Italy, 1994.
- [54] J. Clark. Five challenges facing the XML community. In *XML Conference and Exposition*, Orlando (Florida), USA, December 2001.
- [55] J. Clark. TREX - Tree Regular Expressions for XML language specification, February 2001. <http://www.thaiopensource.com/trex/spec.html>
- [56] J. Clark and M. Murata. Relax-NG, July 2002. <http://xml.coverpages.org/relax-ng.html>
- [57] James Clark and Steve DeRose. XML Path language (XPath) version 1.0, November 1999. <http://www.w3.org/TR/xpath>.
- [58] James Clark and Stephen Deach (Editor). Extensible Stylesheet Language (XSL) version 1.0, 16 December 1998. <http://www.w3.org/TR/1998/WD-xsl-19981216>
- [59] S. Comai. *Graphical Query Language for Semi-structured Information*. PhD thesis, Politecnico di Milano, Dipartimento di Elettronica e Informazione, 1999.
- [60] Dan Connolly, Frank van Harmelen and Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider, and Lynn Andrea Stein. DAML+OIL reference description. W3C Note, December 2001. <http://www.w3.org/TR/daml+oil-reference>
- [61] Corel. Bryce 5. <http://www.corel.com>, March 2002.
- [62] Gennaro Costagliola, Andrea De Lucia, and Sergio Orefice. Towards efficient parsing of diagrammatic languages. In *Proceedings of the workshop on Advanced visual interfaces*, Bari, Italy, 1994.
- [63] P. Cotton and N. Walsh (Editors). What does a document mean? W3C TAG Draft, 25 March 2002. <http://www.w3.org/2001/tag/doc/docmeaning.html>
- [64] D. Cox, J.S. Chugh, C. Gutwin, and S. Greenberg. The usability of transparent overview layers. In *CHI'98, ACM Conference on Human Factors in Computing Systems*, pages 301–302, April 1998.
- [65] P.T. Cox, F.R. Giles, and T. Pietrzykowski. *Visual Object Oriented Programming*, chapter 3, pages 45–66. Manning, 1995.
- [66] P.T. Cox and T. Pietrzykowski. Using a pictorial representation to combine dataflow and object-orientation in a language-independent programming mechanism. In *Proceedings of the fifth international workshop on Software specification and design*, pages 695–704, Hong Kong, 1988.

- [67] A. Cypher and D. Smith. Stagecast creator, 2002. <http://www.stagecast.com>
- [68] Andrew Davidson, Matthew Fuchs, Mette Hedin, Mudita Jain, Jari Koistinen, Chris Lloyd, Murray Maloney, and Kelly Schwarzhof. Schema for object-oriented XML 2.0. W3C Note, July 1999. <http://www.w3.org/TR/NOTE-SOX/>
- [69] DCMI. Dublin Core metadata element set, June 2002. <http://dublincore.org>
- [70] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. XML-QL : A query language for XML. In *8th WWW conference*, Toronto, Canada, 1999.
- [71] W3C Architecture Domain. Document Object Model (dom), June 2002. <http://www.w3.org/DOM/>
- [72] W3C Architecture Domain. XML Query, August 2002. <http://www.w3.org/XML/Query>
- [73] W3C Architecture Domain. XML Query, September 2002. <http://www.w3.org/XML/Query>
- [74] W3C Document Formats Domain. PNG (Portable Network Graphics), July 2001. <http://www.w3.org/Graphics/PNG/>
- [75] W3C Document Formats Domain. Amaya - W3C's editor/browser, July 2002. <http://www.w3.org/Amaya/>
- [76] W3C Document Formats Domain. The extensible stylesheet language (XSL), July 2002. <http://www.w3.org/Style/XSL/>
- [77] W3C Document Formats Domain. Hypertext markup language (HTML) home page, July 2002. <http://www.w3.org/MarkUp/>
- [78] W3C Document Formats Domain. Scalable vector graphics (svg), June 2002. <http://www.w3.org/Graphics/SVG/Overview.htm8>
- [79] W3C Document Formats Domain. W3C Math Home, July 2002. <http://www.w3.org/Math/>
- [80] W3C Interaction Domain. Synchronized Multimedia, July 2002. <http://www.w3.org/AudioVideo/>
- [81] W3C Interaction Domain. Voice browser activity - voice enabling the web!, August 2002. <http://www.w3.org/Voice/>
- [82] James Clark (Editor). XSL Transformations (XSLT) version 1.0 w3c recommendation, 16 November 1999. <http://www.w3.org/TR/1999/REC-xslt-19991116>
- [83] Michael Kay (Editor). XSL Transformations (XSLT) version 2.0. W3C Working Draft, 30 April 2002. <http://www.w3.org/TR/xslt20/>
- [84] M J Egenhofer. Query processing in spatial-query-by-sketch. *Journal of Visual Languages and Computing*, 8(4) :403–424, 1997.
- [85] S. Eisenbach, L. McLoughlin, and C. Sadler. Data-flow design as a visual programming language. In ACM, editor, *Proceedings of the fifth international workshop on Software specification and design*, pages 281–283, Pittsburgh, PA USA, 1989.
- [86] M. Erwig. A visual language for XML. In *IEEE Symposium on Visual Languages*, pages 47–54, Seattle, USA, 2000.
- [87] M. Erwig. XML queries and transformations for end users. In *XML 2000*, pages 259–269, 2000.

- [88] M. Erwig. Xing : A visual XML query language. *Journal of Visual Languages and Computing*, 13, 2002. to appear.
- [89] M. Erwig and B. Meyer. Heterogeneous visual languages - integrating visual and textual programming. In *IEEE Symposium on Visual Languages*, pages 318–325, Darmstadt, Germany, 1995.
- [90] J. Pfeiffer et al. Altaira : reactive robot control meets visual languages. Visual Programming Challenge submission, 1997.
- [91] Myers et al. Strategic directions in human-computer interaction. *ACM Computing Surveys*, 28(4), 1996.
- [92] S-K. Chang et al. Visual language system for user interface. *IEEE Software Issn 0740-7459*, 12(2) :33–44, 1995.
- [93] Xerox Research Centre Europe. Content analysis, September 2002. <http://www.xrce.xerox.com/competencies/content-analysis/>
- [94] eXcelon Corporation. Excelon stylus studio, July 2002. <http://www.exceloncorp.com>
- [95] Filomena Ferucci, Genny Tortora, and Maurizio Tucci. Semantics of visual languages. In *Proceedings of the workshop on Advanced visual interfaces*, pages 219–221, 1994.
- [96] R. Fikes, P. Hayes, and I. Horrocks. DAML Query Language, August 2002. <http://www.daml.org/2002/08/dql/>
- [97] International Organization for Standardisation. Information processing - text and office systems - standard generalized markup language (sgml), ISO 8879 :1986. Genève.
- [98] SAX (Simple API for XML). David megginson, August 2002. <http://www.saxproject.org/>.
- [99] Apache Software Foundation. FOP : Formatting objects processor, August 2002. <http://xml.apache.org/fop/index.html>
- [100] Apache Software Foundation. Xerces java parser, June 2002. <http://xml.apache.org/xerces-j/index.html>
- [101] Charles Frankston and Henry S. Thompson. XML-data reduced. Draft, July 1998. <http://www.ltg.ed.ac.uk/ht/XMLData-Reduced.htm>.
- [102] Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Kiem-Phong Vo. A technique for drawing directed graphs. *Software Engineering*, 19(3) :214–230, 1993.
- [103] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Software - Practice and Experience*, 11(30) :1203–1233, 2000.
- [104] S. Ghiasi. Vipr and the visual programming challenge. Visual Programming Challenge submission, 1997. University of Colorado.
- [105] Elena Ghittori, Mauro Mosconi, and Marco Porta. Designing and testing new programming constructs in data flow VL. Technical report, Università di Pavia, Italy, 1998.
- [106] E. J. Golin and S. P. Reiss. The specification of visual language syntax. *Journal of Visual Languages and Computing*, 1(2) :141–157, 1990.
- [107] Google web search engine, June 2002. <http://www.google.com>
- [108] Michael Gorlick and Alex Quilici. Visual programming-in-the-large versus visual programming-in-the-small. In *10th Annual IEEE Conference on Visual Languages*, St. Louis, MI, USA, 1994.

- [109] T. R. G. Green, M. M. Burnett, A. J. Ko, K. J. Rothermel, C. R. Cook, and J. Schonfeld. Using the cognitive walkthrough to improve design of a visual programming experiment. In *Proceedings of Visual Languages 2000*, Seattle, USA, 2000.
- [110] T. R. G. Green and M. Petre. When visual programs are harder to read than textual programs. In *Proceedings of Sixth European Conference on Cognitive Ergonomics*, pages 167–180, 1992.
- [111] T.R.G Green and M. Petre. Usability analysis of visual programming environments : a ‘cognitive dimensions’ framework. *Journal of Visual Languages and Computing*, 7(2) :131–174, January 1996.
- [112] George G.Robertson, Stuart K.Card, and Jack D.Mackinlay. Information visualization using 3d interactive animation. *Communication of the ACM*, 36(4) :56–71, 1993.
- [113] Tom Gruber. What is an ontology ?, June 2002. <http://www-ksl.stanford.edu/kst/what-is-an-ontology.html>
- [114] W.J. Hansen. Andrew as a multiparadigm environment for visual languages. In *IEEE Symposium on Visual Languages*, pages 256–260, Bergen, Norway, 1993.
- [115] W.J. Hansen. The 1994 visual languages comparison. In IEEE Computer Society Press, editor, *Proceedings 1994 IEEE Symposium on Visual Languages*, St. Louis, Missouri, USA, 1994.
- [116] David Harel. On visual formalisms. *Communication of the ACM*, 31(5) :514–530, May 1988.
- [117] Elliotte Rusty Harold. *XML Bible*. IDG Books Worldwide, 1999.
- [118] Richard Helm, Kim Marriott, and Martin Odersky. Building visual language parsers, 1991.
- [119] Yannick Hereus. Spécification, analyse et implémentation d’une machine graphique abstraite. DESS Génie Informatique, Université de Nantes, 1998.
- [120] K. Holman. Document schema definition languages. ISO/IEC 19757, June 2002. <http://www.dsdl.org/>
- [121] H. Hosoya and B. C. Pierce. XDuce : A typed XML processing language (preliminary report). In *WebDB (Informal Proceedings)*, pages 111–116, 2000.
- [122] C. Hundhausen and S. Douglas. Salsa and alvis : A language and system for constructing and presenting low fidelity algorithm visualizations. In *Proceedings of the IEEE Symposium on Visual Languages*, pages 67–68, Seattle, USA, 2000.
- [123] C. Hundhausen and S. Douglas. Using visualizations to learn algorithms : Should students construct their own, or view an expert’s. In *Proceedings of the IEEE Symposium on Visual Languages*, pages 21–28, Seattle, USA, 2000.
- [124] K. Hunn. The design of visual languages. http://home.eunet.no/khunn/papers/2650_2.html, 2002.
- [125] B. Ibrahim, H. Randriamparany, and H. Yoshizumi. Heuristics for edge drawing in a graph-based visual language. In IEEE Computer Society, editor, *IEEE Symposia on Human-Centric Computing (HCC)*, pages 334–337, Stresa, Italy, September 2001.
- [126] Induslogic. Xslwiz, 2001. <http://www.induslogic.com/products/xslwiz.html>
- [127] J. Jagadeesh and Y. Wang. «LabView» product review. *Computer*, February 1993.

- [128] Rahman Jamal and Lothar Wenzel. The applicability of the visual programming language labview to large real-world applications.
- [129] R. Jelliffe. The schematron : An xml structure validation language using patterns in trees. ISO/IEC 19757 - DSDL Document Schema Definition Language, June 2002. <http://www.ascc.net/xml/resource/schematron/schematron.html> (Academia Sinica Computing Centre).
- [130] B. Johnson and B. Shneiderman. Treemaps : a space-filling approach to the visualization of hierarchical information structures. *Proceedings of the 2nd International IEEE Visualization Conference, San Diego*, pages 284–291, 1991.
- [131] K.M Khan and V.A. Saraswat. Complete visualizations of concurrent programs and their executions. In *Proceedings of the IEEE Workshop on Visual Languages*, pages 7–15, Skokie, IL (USA), 1990.
- [132] D. Kimelman, B. Leban, T. Roth, and D. Zernik. Reduction of visual complexity in dynamic graphs. In R. Tamassia and I. G. Tollis, editors, *Graph Drawing : DIMACS International Workshop*, number 894 in Lecture notes in computer science, pages 218–225, Princeton, USA, October 1994.
- [133] Kazuhiro Kitagawa. CC/PP working group public homepage, July 2002. <http://www.w3.org/Mobile/CCPP/>
- [134] N. Klarlund, A. Moller, and M.I. Schwartzbach. Dsd : A schema language for xml. *ACM SIGSOFT Workshop on formal methods in software practices*, August 2000.
- [135] Jonathan Knudsen. *Java 2D Graphics*, chapter XOR mode, pages 101–103. Number 1-56592-484-3 in isbn. O'Reilly, May 1999.
- [136] J. Kodosky, J. MacCrisken, and G. Rymar. Visual programming using structured data flow. In *Proceedings of the IEEE Workshop on Visual Languages*, 1991.
- [137] Marja-Ritta Koivunen. Annotea project, August 2002. <http://www.w3.org/2001/Annotea/>
- [138] T. Koyanagi, K. Ono, and M. Hori. Demonstrational interface for xslt stylesheet generation. *Extreme Markup Languages*, 2000.
- [139] HP Laboratories. The jena semantic web toolkit, June 2002. <http://www.hpl.hp.com/semweb/jena-top.html>
- [140] S. Ben Lagha, W. Sadfi, and B. Ahmed. *Actes de la journée XML au Congrès GUT99*, chapter Comparaison SGML-XML, pages 127–154. Number 1140-9304 in ISSN. Cahiers GUTenberg (33-34), Irista/INRIA Rennes, Campus Universitaire de Beaulieu F-35042 Rennes Cedex, France, November 1999.
- [141] F. Lakin. Spatial parsing for visual languages. In S.-K. Chang, T. Ichikawa, and P. Ligomenides, editors, *Visual Languages*, pages 35–85. Plenum Press, New York, 1986.
- [142] J. Lamping and R. Rao. The hyperbolic browser : a focus+context technique for visualizing large hierarchies. *Journal of Visual Languages and Computing*, 7(1) :33–56, 1996.
- [143] L. Lamport. *LaTeX : A Document Preparation System*. Addison-Wesley, Massachusetts, 1985.
- [144] Ora Lassila and Ralph Swick. Resource description framework (RDF) model and syntax specification. W3C Recommendation, February 1999. <http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/>

- [145] D. Lee and Wesley W. Chu. Comparative analysis of six xml schema languages. *ACM SIGMOD Record*, 29(3), September 2000.
- [146] E. Lenz. XQuery : Reinventing the wheel ?, 2001. <http://www.xmlportfolio.com/xquery.html>
- [147] H. Lieberman. Powers of ten thousand : navigating in large information spaces. In *Conference on User Interface Software Technology*, Marina del Rey, California, USA, November 1994.
- [148] H. Lieberman, editor. *Your Wish is My Command : Giving Users the Power to Instruct their Software*, chapter Generalizing by Removing Detail : How Any Program Can Be Created by Working with Examples. M. Kaufmann, 2000.
- [149] Vervet Logic. Near & Far designer, July 2002. <http://www.vervet.com/bundle.html>
- [150] J. Le Maître, Y. Marcoux, and E. Murisasco. SgmlQL + XGQL = powerful pattern-matching and data-manipulation in a single language. In *RIAO' 2000 : Content-Based Multimedia Information Access*, pages 1346–1362, Paris, France, 2000.
- [151] Frank Manola and Eric Miller. RDF Primer, April 2002. <http://www.w3.org/TR/rdf-primer/>
- [152] Waterloo Maple. Maple, September 2002. <http://www.maplesoft.com/main.shtml>
- [153] K Marriott and B Meyer. On the classification of visual languages by grammar hierarchies. *Journal of Visual Languages and Computing*, 8(4) :375–402, April 1997.
- [154] Ishikawa Masayasu. An XHTML + MathML + SVG profile, 9 August 2002. <http://www.w3.org/TR/XHTMLplusMathMLplusSVG/>
- [155] B. McBride, R. Wenning, and L. Cranor. An RDF Schema for P3P. W3C Note, January 2002. <http://www.w3.org/TR/p3p-rdfschema/>
- [156] David McIntyre. comp.lang.visual frequently asked questions, 1998. newsgroup comp.lang.visual.
- [157] D.W. McIntyre and E.P. Glinert. Visual tools for generating iconic programming environments. In *Visual Languages*, pages 162–169, 1992.
- [158] Microsoft. Rich Text Format (RTF) specification, version 1.6, 1999. <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnrtf/spec/html/rftspec.asp>
- [159] Microsoft. Access. <http://www.microsoft.com/office/access/default.htm>, 2000.
- [160] Microsoft. Visual studio, 2000. <http://msdn.microsoft.com/>
- [161] SUN microsystems. Forte, March 2002. <http://forte.sun.com>
- [162] SUN microsystems. Java 2D API. <http://java.sun.com/products/java-media/2D/index.html>, March 2002.
- [163] SUN microsystems. The swing connection. <http://java.sun.com/products/jfc/tsc/index.html>, March 2002.
- [164] E. Miller. An introduction to the resource description framework. *D-Lib Magazine*, issn 1082-9873, May 1998. <http://www.dlib.org/dlib/may98/miller/miller05.html>
- [165] E. Miller, R. Swick, and D. Brickley. Resource description framework (RDF), May 2002. <http://www.w3.org/RDF/>
- [166] Nilo Mitra. Soap version 1.2 part 0 : Primer. W3c Working Draft, 26 June 2002. <http://www.w3.org/TR/2002/WD-soap12-part0-20020626/>

- [167] F. Modugno and B. Myers. Control constructs in a completely visual imperative programming language. Technical Report CMU-CS-94-109, Carnegie Mellon University, January 1994.
- [168] M. Murata, D. Lee, and M. Mani. Taxonomy of XML schema languages using formal language theory. In *Extreme Markup Languages*, Montreal, Canada, August 2001.
- [169] B. A. Myers. Taxonomies of visual programming and program visualization. *Journal of Visual Languages and Computing*, 1(1) :97–123, 1990.
- [170] B.A. Myers and M.B. Rosson. Survey on user interface programming. In *Human Factors in Computing Systems - Proceedings of SIGCHI*, pages 195–202, Monterey, CA, 1992.
- [171] Brad A. Myers. Demonstrational interfaces : A step beyond direct manipulation. *IEEE Computer*, 25(8) :61–73, 1992.
- [172] Brad A. Myers, Dario Giuse, Roger B. Dannenberg, Brad Vander Zanden, David Kosbie, Ed Pervin, Andrew Mickish, , and Philippe Marchal. Garnet : Comprehensive support for graphical, highly-interactive user interfaces. *IEEE Computer*, 23(11), November 1990.
- [173] Oasis. Docbook, June 2002. <http://www.docbook.org>
- [174] Uche Ogbuji. The languages of the semantic web. *New Architect*, June 2002.
- [175] P. Otlet. *Traité de documentation*. Editions Mundaneum (Bruxelles), Liège, Centre de lecture publique de la communauté française, 1934 (Reprinted 1989).
- [176] Sean B. Palmer. The semantic web : An introduction, 2001. <http://infomesh.net/2001/swintro/>
- [177] E. Pietriga. Intégration d'un solveur de contraintes dans une machine abstraite visuelle. DEA Informatique Coordination et coopération dans les systèmes à agents - Université de Savoie, Septembre 1999.
- [178] E. Pietriga. MathMLc2p : Content to Presentation transformation, December 2000. <http://www.inrialpes.fr/opera/people/Emmanuel.Pietriga/mathmlc2p.html>
- [179] E. Pietriga. IsaViz : A visual authoring tool for RDF, March 2002. <http://www.w3.org/2001/11/IsaViz/>
- [180] E. Pietriga. IsaViz (Semantic Web track). Developer's day at WWW 2002, the 11th World Wide Web Conference, May 2002. <http://www2002.org/devday.html>
- [181] E. Pietriga. Isaviz user manual, April 2002. <http://www.w3.org/2001/11/IsaViz/usermanual.html>
- [182] E. Pietriga. Xerox Visual Transformation Machine, 2002. <http://www.xrce.xerox.com/competencies/contextual-computing/vtm/>
- [183] E. Pietriga, V. Quint, and J.-Y. Vion-Dury. VXT : A Visual Approach to XML Transformations. In *ACM Symposium on Document Engineering*, pages 1–10, Atlanta, USA, November 2001.
- [184] E. Pietriga and J.-Y. Vion-Dury. VXT : Visual XML Transformer. In IEEE Computer Society, editor, *IEEE Symposia on Human-Centric Computing (HCC), Symposium on Visual/Multimedia Approaches to Programming and Software Engineering*, pages 404–405, Stresa, Italy, September 2001.
- [185] E. Pietriga, J.-Y. Vion-Dury, and V. Lux. Circus vs XSLT (technical report), June 2001. <http://www.xrce.xerox.com/competencies/contextual-computing/circus/resources/circusVSxslt.pdf>.

- [186] S. Pook, E. Lecolinet, G. Vaysseix, and E. Barillot. Context and interaction in zoomable user interfaces. In *AVI 2000 Conference*, pages 227–231, Palermo, Italy, 2000.
- [187] Aaron Quigley. *Large Scale Relational Information Visualization, Clustering, and Abstraction*. PhD thesis, Department of Computer Science and Software Engineering, University of Newcastle, Callaghan 2308, NSW, Australia, August 2001.
- [188] V. Quint. *Une approche de l'édition structurée des documents*. PhD thesis, Université Scientifique, Technologique et Médicale de Grenoble, Mai 1987.
- [189] V. Quint. The languages of thot, 1997. <http://www.inrialpes.fr/opera/Thot/Doc/languages.toc.html>
- [190] S. R. Ranganathan. *Documentation and its facets*. Asia publishing house, London, 1963.
- [191] J. Rapoza. Rdf standard, illustrated. *PC Magazine*, April 15th 2002.
<http://www.pcmag.com/article2/0,4149,97136,00.asp>
- [192] J. Rekers and A. Schurr. A graph grammar approach to graphical parsing, 1995.
- [193] AT&T Research. Graphviz - open source graph drawing software, June 2002.
<http://www.research.att.com/sw/tools/graphviz/>
- [194] Wolfram Research. Mathematica, September 2002. <http://www.wolfram.com/>
- [195] J. Robie, J. Lapp, and D. Schach. XML Query Language (XQL), September 1998.
<http://www.w3.org/TandS/QL/QL98/pp/xql.html>
- [196] Peter J. Rodgers and Natalia Vidal. Graph algorithm animation with Grrr, 1998.
- [197] D. Smith, A. Cypher, and J. Spohrer. Kidsim : programming agents without a programming language. *Communications of the ACM*, 37(7) :54–67, July 1994.
- [198] P. Snively. Hack the planet - java desktop apps, March 27 2002.
[http://wmf.editthispage.com/discuss/msgReader\\$7310?mode=day](http://wmf.editthispage.com/discuss/msgReader$7310?mode=day)
- [199] John T. Stasko and Carlton Reid Turner. Tidy animations of tree algorithms. Technical Report GIT-GVU-92-11, Georgia Institute of Technology, Atlanta, USA, 1992.
- [200] D. Steer, L. Miller, and D. Brickley. Rdfauthor (semantic web track). Developer's day at WWW 2002, the 11th World Wide Web Conference, May 2002. <http://www2002.org/devday.html>
- [201] Chris Stefano. Xsldebugger, July 2002. <http://www.vbxml.com/xsldebugger/>
- [202] I. Sutherland. Sketchpad : A man-machine graphical communication system. In IFIPS, editor, *Spring Joint Computer Conference*, pages 329–346, 1963.
- [203] A. Swartz and J. Hendler. The semantic web : A network of content for the digital city. In *Proceedings Second Annual Digital Cities Workshop*, Kyoto, Japan, 2001.
- [204] Rashmi Tambe. Zooming user interfaces.
<http://jupiter.eecs.utoledo.edu/rashmi/research/zui.html>, March 2002.
- [205] S. Tanimoto. Viva : a visual language for image processing. *Journal of Visual Languages and Computing*, 2(2) :127–139, June 1990.
- [206] Omnimark Technologies. Omnimark, July 2002.
<http://www.omnimark.com/products/products.html>
- [207] T. Teitelbaum and T. Reps. The cornell program synthesizer : A syntax-directed programming environment. *Communications of the ACM*, 24(9) :563–573, September 1981.

- [208] B. Templeton. Alice pascal, 1985. <http://www.templetons.com/brad/alice.html>
- [209] Henry S. Thompson, David Beech, Murray Maloney, and Noah Mendelsohn. XML Schema part 1 : Structures. W3C Recommendation, May 2001. <http://www.w3.org/TR/xmlschema-1/>
- [210] A. Tozawa. Towards static type checking for xslt. In *ACM Symposium on Document Engineering*, pages 18–27, Atlanta, USA, November 2001.
- [211] Edward R. Tufte. *Visual Explanations*. Number 0-9613921-2-6 in ISBN. Graphics Press, 1997.
- [212] L. Tweedie. Characterizing interactive externalizations. In *CHI'97, ACM Conference on Human Factors in Computing Systems*, pages 375–382, Atlanta, Georgia, USA, March 1997. Academic Press.
- [213] W3C/IETF URI Planning Interest Group. Uris, urls, and urns : Clarifications and recommendations 1.0 - report from the joint w3c/ietf uri planning interest group. W3C Note, September 2001. <http://www.w3.org/TR/2001/NOTE-uri-clarification-20010921/>
- [214] E. van der Vlist. Comparing xml schema languages. *XML.com*, December 2001. <http://www.xml.com/pub/a/2001/12/12/schemacompare.html>
- [215] F. Vernier. *La multimodalité en sortie et son application à la visualisation de grandes quantités d'information*. PhD thesis, Université Grenoble 1 Joseph Fourier, 2001.
- [216] L. Villard. Authoring transformations by direct manipulation for adaptable multimedia presentations. In *ACM Symposium on Document Engineering*, pages 125–134, Atlanta, USA, November 2001.
- [217] L. Villard. *Modèles de documents pour l'édition et l'adaptation de présentations multimédias*. PhD thesis, Institut National Polytechnique de Grenoble, Mars 2002.
- [218] L. Villard and N. Layaida. An incremental xslt transformation processor for xml document manipulation. In *Proceedings of the 11th World Wide Web Conference*, page 321, Honolulu, Hawaii, 2002.
- [219] J.-Y. Vion-Dury. Circus web page, June 2002. <http://www.xrce.xerox.com/competencies/contextual-computing/circus/>
- [220] J.-Y. Vion-Dury, V. Lux, and E. Pietriga. Experimenting with the circus language for XML modeling and transformation. In *ACM Symposium on Document Engineering*, Mc Lean (VA), USA, November 2002.
- [221] J.-Y. Vion-Dury and E. Pietriga. A formal study of a visual language for the visualization of document type definition. In IEEE Computer Society, editor, *IEEE Symposia on Human-Centric Computing (HCC), Symposium on Visual Languages and Formal Methods*, pages 52–59, Stresa, Italy, September 2001.
- [222] J.-Y. Vion-Dury and M. Santana. Virtual images : Interactive visualization of distributed object-oriented systems. In ACM press, editor, *OOPSLA'94 Conference Proceedings*, volume 29, pages 65–84, Portland, Oregon, USA, October 1994.
- [223] Jean-Yves Vion-Dury. *CIRCUS : Un générateur de composants pour le traitement des langages visuels et textuels*. PhD thesis, Université Joseph Fourier - Grenoble 1, Domaine Universitaire, Saint Martin d'Hères, France, Juin 1999.

- [224] Jean-Yves Vion-Dury and François Pacull. A structured interactive workspace for a visual configuration language. In *Proceedings of Visual Languages 1997*, pages 132–139, Capri, Italy, 1997.
- [225] W3C. RDF validation service. On-line service, December 2001.
<http://www.w3.org/RDF/Validator/>
- [226] W3C. The world wide web consortium, July 2002. <http://www.w3.org>
- [227] W3C. www-rdf-interest@w3.org mail archives, September 2002.
<http://lists.w3.org/Archives/Public/www-rdf-interest/>
- [228] P. Wadler. A formal semantics of patterns in xslt. Markup Technologies, 1999. 16, 1999.
- [229] P. Wadler. Plan-X : Programming language technologies for xml, October 2002. Pittsburgh (USA).
- [230] N. Walsh. Docbook XSL stylesheets, July 2002. <http://docbook.sourceforge.net/projects/xsl/>
- [231] Sanjiva Weerawarana, Roberto Chinnici, Martin Gudgin, and Jean-Jacques Moreau. Web services description language (wsdl) version 1.2. W3C Working Draft, 9 July 2002.
<http://www.w3.org/TR/wsdl12/>
- [232] K N Whitley. Visual programming languages and the empirical evidence for and against. *Journal of Visual Languages and Computing*, 8(1) :109–142, 1996.
- [233] R. Widell and R. Daniel. Roadsurf 1.0. Visual Programming Challenge submission, 1997. Center for Advanced Technologies, Florida.
- [234] E.M. Wilcox, J.W. Atwood, M.M. Burnett, J.J. Cadiz, and C.R. Cook. Does continuous visual feedback aid debugging in direct-manipulation programming systems ? In ACM, editor, *Conference proceedings on Human factors in computing systems*, pages 258–265, 1997.
- [235] K. Wittenburg. Earley-style parsing for relational grammars. In *Proceedings of the IEEE workshop on Visual Languages*, pages 192–199, Seattle, USA, 1992.
- [236] K. Wittenburg and L. Weitzman. Relational grammars : Theory and practice in a visual language interface for process modeling. In *AVI '96 International Workshop on Theory of Visual Languages*, Gubbio, Italy, 1996.
- [237] D.Q. Zhang and K. Zhang. Reserved graph grammar : a specification tool for diagrammatic vpls. In IEEE Computer Society, editor, *IEEE Symposium on Visual Languages*, pages 341–348, Capri, Italy, September 1997.
- [238] K. Zhang, D.Q. Zhang, and Y. Deng. A visual approach to xml document design and transformation. In IEEE Computer Society, editor, *IEEE Symposia on Human-Centric Computing (HCC)*, pages 312–319, Stresa, Italy, September 2001.

Résumé

L'adoption du langage XML dans de nombreux domaines pour la représentation des documents et des données a simplifié les manipulations associées aux documents du World Wide Web en offrant des solutions d'analyse et de traitement génériques. Ces manipulations se traduisent souvent par des opérations de transformation de la structure et du contenu des documents et jouent un rôle essentiel dans la chaîne de traitement documentaire.

Les solutions existantes pour la transformation de documents XML sont pour la plupart basées sur un langage textuel, et même si certains outils proposent une interface graphique au-dessus de langages tels que XSLT, les transformations sont toujours spécifiées textuellement. L'objectif de ce travail est d'étudier l'intérêt des représentations graphiques de structures logiques et des techniques de programmation visuelle pour la visualisation et la transformation des documents et classes de documents XML. C'est dans ce cadre qu'a été conçu VXT, un langage de programmation visuel pour la spécification de transformations de documents XML.

Mots-clés

Langages de programmation visuels - Documents structurés - XML - RDF - Transformations de documents - XSLT - Visualisation de structures - Interfaces graphiques zoomables.

Abstract

The adoption of XML in a wide variety of domains for the representation of structured documents and data has made the processing of World Wide Web documents easier, by offering generic solutions for parsing and manipulating them. These operations are often achieved through document structure and content transformations, and play a key role in the document manipulation process.

Most existing solutions for the transformation of XML documents rely on a textual language, and even if some tools offer a graphical user interface on top of languages such as XSLT, transformations are still specified textually. The goal of this work is to study the applicability of visual programming techniques with respect to the visualisation and transformation of XML documents. It has led to the design of VXT (Visual XML Transformer), a visual programming language for the specification of XML document transformations.

Keywords

Visual Programming Languages - Structured documents - XML - RDF - Document transformations - XSLT - Structure visualization - Zoomable User Interfaces.

Thèse préparée à l'INRIA Rhône-Alpes dans le projet Opéra, 655 avenue de l'Europe - Montbonnot Saint Martin - 38334 Saint Ismier Cedex