No d'Ordre :

THÈSE

présentée à

L'UNIVERSITÉ DE NICE - SOPHIA ANTIPOLIS
Ecole Doctorale - Sciences pour l'ingénieur

pour obtenir

LE TITRE DE DOCTEUR EN SCIENCES
Spécialité INFORMATIQUE

par

Claude PASQUIER

Sujet de la thèse :

GESTION ET CONCEPTION DE DOCUMENTS STRUCTURÉS PAR LE CONTEXTE

Soutenue le 5 Juillet 1994 devant un jury composé de :

MM.	**M. Rueher**	Président
	P. Cointe	Rapporteurs
	V. Quint	
	P. Franchi-Zannettacci	Examinateurs
	J.M. Sézérat	

No d'Ordre :

THÈSE

présentée à

L'UNIVERSITÉ DE NICE - SOPHIA ANTIPOLIS
Ecole Doctorale - Sciences pour l'ingénieur

pour obtenir

LE TITRE DE DOCTEUR EN SCIENCES
Spécialité INFORMATIQUE

par

Claude PASQUIER

Sujet de la thèse :

GESTION ET CONCEPTION DE DOCUMENTS STRUCTURÉS PAR LE CONTEXTE

Soutenue le 5 Juillet 1994 devant un jury composé de :

MM.	**M. Rueher**	Président
	P. Cointe	Rapporteurs
	V. Quint	
	P. Franchi-Zannettacci	Examinateurs
	J.M. Sézérat	

A mes parents

Je tiens à remercier

Michel Rueher pour l'honneur qu'il me fait en acceptant de présider le jury,

Vincent Quint pour avoir accepté de rapporter sur cette thèse et pour l'intérêt qu'il a porté à mon travail. Ses conseils avisés et les voies qu'il m'a suggéré d'explorer ont été d'une grande aide pour l'aboutissement de mes recherches,

Pierre Cointe pour avoir accepté de rapporter sur cette thèse,

Paul Franchi-Zannettacci, mon directeur de thèse. Ses idées, ses conseils et ses relectures rigoureuses ont été décisifs dans la réalisation de ce projet. Je lui exprime toute ma gratitude,

Jean-Marie Sézérat pour m'avoir donné la chance d'effectuer cette thèse. Je le remercie de l'intérêt constant qu'il a porté à mes recherches et pour tout le temps et les efforts qu'il m'a consacrés. Je lui suis très reconnaissant d'avoir accepté de participer au jury,

Les membres du service RDT et d'Informatique CDC en général. J'ai trouvé auprès d'eux un environnement de travail d'une exceptionnelle qualité. Mes pensées vont plus particulièrement à André Blavier, André Candio, Kent Hudson, Marielle Riffault, Jacques Sadoun et Francis Wolinski qui se sont beaucoup investis dans les relectures et corrections des différentes parties de ce mémoire. Je tiens également à remercier Bruno Dillet qui, pendant mon stage de DEA, m'a familiarisé au vaste domaine des documents structurés et m'a incité à poursuivre des recherches dans cette voie,

Enfin tous les autres, parents et amis, qui par leur soutien, leurs encouragements et leur affection, surtout dans les moments difficiles, ont beaucoup contribués à l'achèvement de ce travail. Je remercie tout particulièrement Marie-Hélène, et Sylvie pour ses relectures scrupuleuses.

Sommaire

Contexte de travail 11

Introduction 13

La structuration. Les documents structurés. Similitudes entre documents et programmes. Différences entre documents et programmes. Focalisation sur les problèmes quantitatifs. Personnalisation des documents par une édition multicouches. Définition des modèles par copies différentielles. Plan de l'étude.

PREMIÈRE PARTIE
Modélisation et manipulation des documents structurés

1 La modélisation des documents 29

1.1 L'évolution des concepts 29
1.1.1 Le balisage procédural 29
1.1.2 Le balisage déclaratif 30
1.1.3 Les boîtes 31
1.1.4 Les styles 32
1.1.5 What You See Is What You Get 33
1.1.6 Les limites de l'édition libre 33

1.2 La représentation de la structure abstraite 34
1.2.1 Représentation arborescente 35
1.2.2 Représentations par graphe orienté 38
1.2.3 Représentation mixte 38

1.3 Les modèles de structuration logique 40
1.3.1 Structure syntaxique 40
1.3.2 Sémantique statique 41
1.3.3 Vision orientée-objet 43
1.3.4 La modularité dans la représentation des documents 50

1.4 La représentation concrète 52
1.4.1 Une dérivation de la structure logique 52
1.4.2 Une structure physique définie par un modèle 53
1.5 Les normes de représentation actuelles 54
1.5.1 SGML 54
1.5.2 ODA 58
1.5.3 SGML vs ODA 60
1.6 Conclusion 63

2 La manipulation des documents 65

2.1 Les problèmes induits par la structure 65
2.1.1 Les transformations dynamiques 66
2.1.2 Les transformations statiques 69
2.1.3 L'élaboration des représentations graphiques 69
2.1.4 Le problème central des transformations de structure 70
2.2 Les techniques de décompilation 71
2.2.1 Visualisation textuelle 71
2.2.2 Décompilation libre de contexte 72
2.2.3 Décompilation contextuelle déterministe 73
2.2.4 Décompilation contextuelle non déterministe 73
2.3 Les transformations de structures 76
2.3.1 Utilisation d'une sémantique translationnelle 76
2.3.2 Les transformations en SGML grâce à DSSSL 77
2.3.3 Automatisation du processus de transformation 78
2.3.4 La prise en considération du contexte 80
2.4 L'adéquation des documents aux données 81
2.4.1 Les documents actifs 82
2.4.2 La génération de la structure des documents 85
2.4.3 L'utilisation de parties variables dans SGML 86
2.5 Les fondements du projet BIBLE 87
2.5.1 La génération vue comme une transformation statique 87
2.5.2 La représentation des modèles par des prototypes 90

SECONDE PARTIE
Le système BIBLE

3 Description conceptuelle du système BIBLE 93

3.1 Conception de documents dirigée par les données 93
3.1.1 L'EDTD : extension sémantique d'une DTD 94
3.1.2 Transformation d'une DTD en EDTD 95
3.1.3 Expression de la variabilité dans les gabarits 99

3.2 L'héritage de la composition dans une hiérarchie de prototypes 101
3.2.1 Partage de sous-structures 102
3.2.2 Clonage de sous-structures 103
3.2.3 Clonage-lié de sous-structures 106
3.2.4 L'organisation des données dans BIBLE 107
3.2.5 Répercussion des transformations sur les instances 109

3.3 Les apports des concepts utilisés 111

4 Les composants logiciels du système 113

4.1 Représentation orientée-objet de la base de modèles 116
4.1.1 L'héritage de la composition des modèles 118

4.2 L'éditeur de modèles 122

4.3 L'éditeur de présentations 126
4.3.1 L'éditeur de boîtes 127
4.3.2 Le placement des boîtes sur les pages 130
4.3.3 La génération du langage 'P' 132

4.4 Transformation d'une DTD en EDTD grâce à un parser Haskell 137

4.5 L'édition des gabarits sur l'éditeur Grif 140

4.6 La génération des documents 144
4.6.1 L'assemblage des documents 144
4.6.2 Le formatage du document final 146

Conclusion 147

Références bibliographiques 151

Liste des figures 161

Index 165

Annexes 169

Annexe A : Les développements en CLOS 171
A.1 : Les classes utilisées 173
A.2 : Une partie des sources écrites en CLOS 175

Annexe B : Les développements en langage Haskell 217
B.1 : Les fonctions de parsing utilisées 219
B.2 : Sources du module de transformation DTD -->EDTD 223
B.3 : Sources du module d'assemblage 229

Annexe C : Exemple d'un fichier 'P' généré par BIBLE 233

Annexe D : La table de personnalisation utilisée dans BIBLE 239
D.1 : La DTD Condition 241
D.2 : Le fichier de présentation 243

Contexte de travail

Ce travail de recherche a été effectué dans le cadre d'un contrat CIFRE passé entre l'ANRT et INFORMATIQUE CDC, la branche informatique du groupe Caisse des Dépôts et Consignations. Il s'est déroulé pour une grande partie, au sein du service Recherche Développement et Techniques Avancées d'INFORMATIQUE CDC dirigé par Jean-Marie Sézérat, et pour une autre partie au sein de l'équipe de Paul Franchi-Zannettacci au laboratoire I3S de l'université de Sophia-Antipolis.

Ce travail vise à apporter une solution à un problème particulier ressenti par le groupe CDC en ce qui concerne la génération à grande échelle de documents personnalisés. Les hypothèses retenues ainsi que l'environnement technique utilisé ont bien sûr été influencés par l'entreprise d'accueil. Cependant, nous nous sommes efforcés dans la première partie de ce mémoire de nous détacher de ce cadre particulier pour réaliser une étude aussi générique que possible.

Introduction

> A document is a representational, primarily textual object constructed for the purpose of causing transformations in the cognitive/knowledge state of readers.
>
> C. Nicholas, J. Mayfield et J. Sasaki [103]

Le travail présenté ici s'inscrit dans le cadre général de la manipulation des documents. Nous nous intéressons aux documents représentés par des structures abstraites de données dont les règles de composition sont définies dans des modèles : un type de document qui est désigné par le terme de "document structuré".

Les documents structurés ne sont pas uniquement destinés à être imprimés. Ils peuvent être traités automatiquement par des applications, stockés dans une base de données, organisés de manière à optimiser les recherches d'informations, imprimés sur différents supports, etc. A chaque usage correspond un type de document particulier dont les règles de structuration sont données par un modèle. L'inconvénient de cette situation est qu'un document décrit suivant les spécifications d'un modèle donné n'est pas directement exploitable par des applications utilisant d'autres formalismes de représentation. Cela introduit un partitionnement structurel qui est néfaste à une bonne gestion des documents. Le problème est encore accentué par le fait que les modèles ne sont pas complètement figés : ils sont susceptibles d'évoluer après que des documents aient été conçus suivant leurs spécifications. Elaborer une organisation cohérente des modèles qui permette de faciliter la migration des documents constitue la première facette de notre étude.

La seconde facette concerne plus spécifiquement les types de documents destinés à être lus par des lecteurs humains. Une théorie actuelle énoncée par Philip Johnson-Laird en 1983 considère l'opération de lecture comme la conversion d'une information écrite en une représentation mentale [70]. Cette représentation est un modèle interne du monde propre à chaque lecteur ; d'où l'importance que revêt la prise en considération du lecteur lors de l'élaboration des documents. Dans notre étude, nous partons de l'hypothèse selon laquelle les informations concernant les destinataires des documents sont disponibles dans une base de données. Nous utilisons ces informations pour élaborer un système de conception de documents structurés dirigé par des données.

La structuration

On appelle structuration, l'organisation d'un ensemble de données en fonction de règles contenues dans un modèle de structure [26]. On distingue deux types de règles :

- des propriétés structurelles qui définissent les type de données modélisées et les relations entre ces données,
- des propriétés sémantiques qui spécifient les conditions de validité d'un ensemble de données structurellement correct.

L'utilisation d'une structure, et surtout d'un modèle de structure, permet une définition formelle des données manipulées et de leurs interactions. On dispose de cette manière d'un support algébrique propre à la mise en place de règles de représentation et de manipulation des données [21, 22, 46]. On peut prendre comme exemple le modèle relationnel des bases de données : il est fondé sur le concept mathématique de relation et permet la définition d'une algèbre relationnelle propre à la manipulation des données [37].

La structure influence également l'architecture des outils. Plus les données sont typées et structurées, plus les outils qui les manipulent peuvent être ciblés et appropriés [14, 23, 46, 56, 143]. Dans un éditeur de texte qui ne connaît que les caractères alphanumériques, il est par exemple possible de représenter un tableau, mais sa manipulation ne sera réellement efficace que si le logiciel connaît la structure de l'élément tableau car cela autorise des traitements sur des cellules ou des groupes de cellules.

Il est même possible de générer automatiquement des outils de manipulation d'une structure de données à partir d'une description formelle de sa syntaxe et de sa sémantique [11, 13, 18, 39, 40, 97, 133, 139]. En génie logiciel, cette technique est très utilisée pour, par exemple, générer automatiquement des parsers ou des compilateurs en fonction des spécifications d'un langage.

Une structure permet de représenter des relations conceptuelles entre des éléments. En attribuant une sémantique aux liens entre les données, il est possible de modéliser des relations de composition, de spécialisation ou d'autres qui dépendent du domaine traité [86]. L'ensemble des relations entre les éléments permet d'élaborer une structure conceptuelle de données qui est bien adaptée à une modélisation par objets.

Les modèles de structures peuvent enfin être utilisés pour guider le travail des utilisateurs. Sur un éditeur structurel, par exemple, seuls sont autorisées (ou proposées) les opérations valides sur la structure de donnée[1] [118, 144].

1. Ces contrôles peuvent cependant être ressentis comme des contraintes par les utilisateurs habitués à des systèmes d'édition libre [4].

Les documents structurés

L'application, vers la fin des années soixante-dix, de la structuration à la modélisation des documents, a donné naissance aux documents structurés. Ce concept relativement récent est cependant à la base de toutes les recherches actuellement effectuées dans le domaine des documents [36, 50, 58, 68, 69, 117, 140].

Un document structuré est appréhendé non plus comme une suite de caractères, mais comme un objet composite représenté sous la forme d'une structure arborescente [8, 71], d'un graphe orienté [7, 64] ou d'une structure hétérogène [47, 50, 116]. Les règles régissant la structuration d'un type de document particulier sont contenues dans un modèle de document [6] également appelé structure générique [68, 118]. Dans un document structuré, la structuration logique est clairement séparée de la présentation. On parle ainsi de structure logique (manipulée par un auteur) et de structure physique (manipulée par un typographe).

Documents et programmes : les similitudes

Beaucoup d'auteurs établissent un parallèle entre les documents et les programmes informatiques [8, 53, 54, 92, 118, 120]. Cette analogie a permis d'utiliser les acquis du génie logiciel pour élaborer les premiers systèmes de manipulation de documents structurés [11, 90, 91, 142]. Dans les systèmes récents, la similitude entre les deux domaines est mise en avant, tant en ce qui concerne la modélisation des données que leur manipulation[1] [120].

Documents et programmes sont représentées par une structure logique définie par des règles syntaxiques et sémantiques. En génie logiciel, cette structure est appelée un arbre de syntaxe abstraite et les règles sont contenues dans une grammaire abstraite[2]. Une grammaire abstraite est composée d'opérateurs et de phyla. Les opérateurs sont des règles qui permettent la création de noeuds étiquetés dans l'arbre de syntaxe abstraite. Les phyla représentent des choix entre différents noeuds possibles. La figure suivante illustre :

- la grammaire abstraite représentant une instruction qui peut être soit un choix (*IF*), soit une répétition (*WHILE*),
- l'arbre de syntaxe abstraite correspondant au *IF*,
- la syntaxe concrète correspondante.

1. La société canadienne Microstar Software Ltd. propose un environnement de développement intégré dédié aux documents structurés. Elle le désigne par CADE (Computer Aided Document Engineering) par analogie au CASE (Computer Aided Software Engineering) utilisé pour le développement de produits logiciels.
2. Arbre de syntaxe abstraite et grammaire abstraite sont des termes introduits dans le projet MENTOR [40].

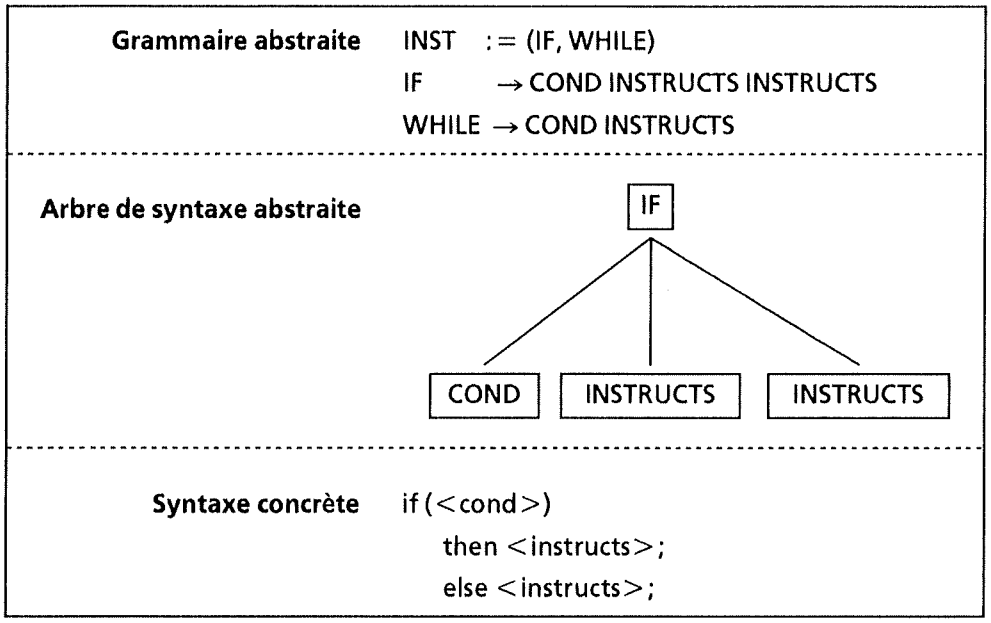


Figure I : Représentation structurée d'une instruction *IF*.

Dans le cas des documents, on a une organisation très similaire. La grammaire abstraite est remplacée par le modèle logique et la syntaxe concrète est nommée modèle physique (figure II).

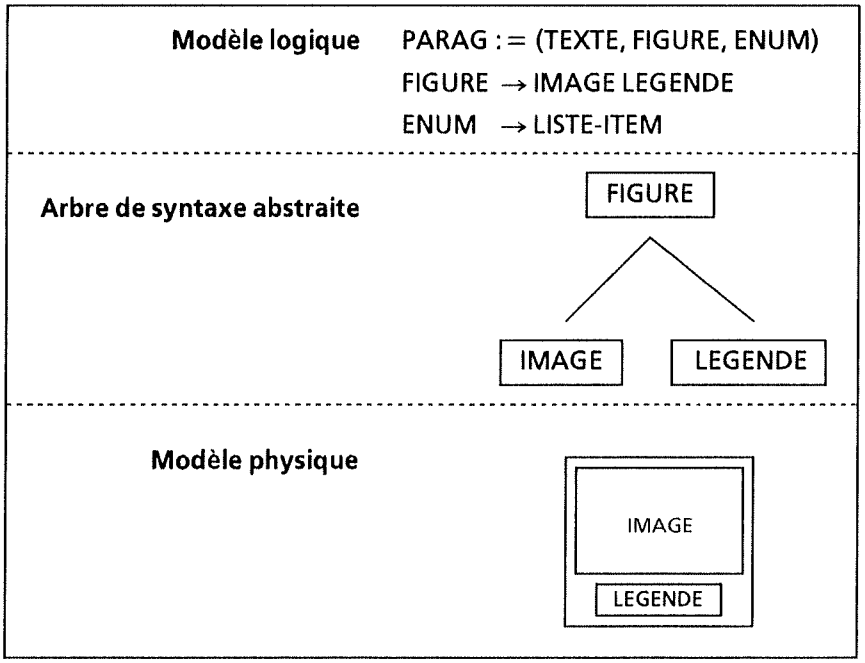


Figure II : Représentation structurée d'une figure d'un document

Modèle logique et grammaire abstraite définissent des relations syntaxiques entre les éléments. Ces déclarations sont appelées grammaires hors contexte car les règles de production s'appliquent de la même façon, quel que soit l'emplacement dans la structure de l'élément concerné. Par exemple, un noeud *FIGURE* est toujours représenté par l'arbre illustré dans la figure II.

Certaines relations ne peuvent pas être modélisées par des grammaires hors contexte. Ce sont celles qui concernent la sémantique. Dans son livre intitulé "Introduction à la théorie des langages de programmation", Bertrand Meyer donne de la sémantique statique la définition suivante [92] :

> *« Static semantics is the description of the structural constraints that cannot be adequately captured by syntax descriptions. »*

Cette définition s'applique aussi bien aux programmes qu'aux documents. Pour les programmes, la sémantique statique permet par exemple de contrôler que toutes les variables utilisées sont correctement déclarées. Dans le cas des documents, cela permet de vérifier, par exemple, que toutes les références utilisées dans un texte font effectivement partie de la bibliographie.

La manipulation des documents, comme celle des programmes, se divise en quatre opérations distinctes :

- la compilation concerne la transformation d'une représentation concrète en une structure abstraite de données,
- la décompilation est l'opération inverse de la précédente : elle consiste à élaborer une représentation concrète à partir d'une structure abstraite,
- l'édition permet de transformer une structure abstraite de données en fonction d'opération effectuées par un utilisateur,
- l'exécution s'applique surtout aux programmes ; un équivalent dans le domaine des documents est représenté par les documents actifs qui ont la faculté d'intéragir avec leur environnement [42, 134].

Documents et programmes : les différences

Malgré les ressemblances qui viennent d'être exposées, les documents diffèrent des programmes sur de nombreux aspects.

a) représentation logique hétérogène

La structure d'un programme est correctement représentée par une arborescence qui se déduit de la grammaire de syntaxe abstraite. Une structure arborescente est également bien adaptée à la représentation de la partie principale d'un document. Certaines données composant un document ne peuvent cependant être représentées de cette façon : pour les tableaux une matrice convient mieux, pour le texte une représentation linéaire est préférable, pour les

images digitalisées une représentation spécifique est également plus appropriée, etc [48, 51].

Un document se distingue également d'un programme du fait de l'importance de ses liens non hiérarchiques. Dans les hypertextes, ces liens deviennent même plus importants que la partie purement arborescente [120]

b) multi-représentations logiques

La structure abstraite doit être adaptée à l'usage que l'on veut en faire. Pour les programmes, la situation est simple puisque ceux-ci sont principalement destinés à être exécutés. Les documents, au contraire, sont destinés à de multiples usages : stockage, lecture, échange, etc. De nombreux modèles de représentation sont donc utilisés et le problème qui se pose fréquemment est : comment passer d'un mode de représentation à un autre ? Ce problème est commun aux documents et aux programmes. La différence vient du nombre de modèles existants : faibles pour les programmes, ils est très important dans le cas des documents car chaque organisation est susceptible de créer des modèles qui lui sont propres.

c) représentation physique multi-dimensionnelle

Pour la majeure partie des programmes, l'aspect physique est représenté par la disposition sur un support d'une succession de lignes avec quelques variations de présentation[1] (changement de type de caractères, de taille, d'indentation, etc). Pour les documents au contraire, l'aspect physique est la représentation multi-dimensionnelle de la structure abstraite de données. Pour l'affichage de beaucoup d'éléments la prise en compte de deux dimensions est nécessaire (tableaux, multi-colonnes, graphiques, images, formules mathématiques) [125]. D'autres éléments, comme les images animées ou le son, nécessitent de considérer également une troisième dimension pour effectuer l'affichage[2].

d) multi-représentations physiques

Une autre différence entre documents et programmes vient également de la diversité des représentations physiques. Pour les programmes l'aspect physique est peu important puisqu'il n'intervient pas dans l'opération d'exécution. Pour les documents, l'aspect physique a au contraire une importance prépondérante. Celui-ci influe en effet très fortement sur la manière dont un lecteur va percevoir les informations. Ce n'est pas un hasard si dans le monde de l'édition, la mise en page et la présentation des documents sont confiés à des personnes spécialisées.

1. Certains langages possèdent une syntaxe graphique (G-LOTOS par exemple [144]), mais il s'agit le plus souvent d'une manière différente de présenter des spécifications qui peuvent s'exprimer de manière linéaire.
2. Les documents hypermédias qui contiennent des données temporelles (image animée ou son) peuvent être modélisés sous la forme d'une structure parallélépipédique constituée d'un empilement de vues successives [105].

e) interdépendance logique physique

Documents et programmes diffèrent également par les liens qui existent entre les structures logiques et physiques. Pour les programmes, seule compte la représentation logique. L'aspect physique est simplement une présentation particulière de la structure logique. Dans le cas des documents, l'aspect physique importe beaucoup plus et il arrive que des contraintes de présentation influent sur la structure logique [120, 127, 128]. Un auteur peut par exemple être contraint de modifier un graphique ou d'éclater un tableau pour que les éléments puissent tenir sur une page. Il est également quelquefois nécessaire d'éclater ou de regrouper des paragraphes en fonction de la taille des caractères utilisés ou de la largeur des colonnes.

f) flexibilité de la structure logique

Les modèles de documents autorisent une flexibilité dans la création des structures que ne possèdent pas les grammaires des langages de programmation. Pour illustrer ce point comparons un programme qui contient une expression *IF* et un document composé de trois blocs de symboles alphanumériques.

Dans le cas du programme, l'expression *IF* correspond à une structure unique qui est donnée par la syntaxe abstraite du langage. Le programmeur ne peut pas changer la structure sans changer l'expression. La grammaire de syntaxe abstraite des langages de programmation possède une sémantique très liée à la syntaxe.

Dans le cas du document, plusieurs structures différentes peuvent être associés aux trois éléments : une liste de trois paragraphes ; un titre et deux paragraphes ; un titre, un paragraphe et un sous titre, une énumeration, etc. C'est l'auteur du document qui, dans son travail de rédaction, spécifie la structure qui est appropriée. Il faut remarquer que la structure regroupant les trois éléments peut être changée sans aucune modification du contenu de ces éléments. Les modèles de documents possèdent une sémantique liée au contexte.

g) prise en compte de la sémantique naturelle

La flexibilité dans la structure des documents est nécessaire car les modèles logiques décrivent une structuration qui ne prend pas en compte la sémantique du langage naturel utilisé pour la rédaction.

Alors qu'un programme est défini par une seule grammaire, un document doit être compatible avec les règles définies par trois formalismes : un modèle logique, un modèle physique (nous avons vu que la structure physique n'est pas simplement obtenue à partir de la structure logique mais qu'il existe des interactions entre les deux) et les règles d'une sémantique naturelle [120].

Les systèmes de manipulation de documents existants permettent de travailler sur la structure logique et/ou la structure physique des documents en

tenant compte des spécifications définies dans les modèles. Par contre, les formalismes de définition de la sémantique naturelle utilisée ne sont accessibles que par les utilisateurs. Certaines opérations ne concernent que les structures logiques ou physiques et d'autres opérations n'entraînent des modifications que dans la représentation sémantique des documents.

Une modification uniquement logique consiste par exemple à identifier de manière spéciale un mot pour qu'il figure dans un index. Cette opération n'induit aucune modification au niveau physique (l'apparence du mot est la même que tous les autres mots du paragraphe) ni dans la sémantique du document. Une modification physique consiste, par exemple, à changer la taille des caractères utilisés dans un document. Une modification sémantique peut être illustrée par le changement de place d'un mot dans une phrase[1].

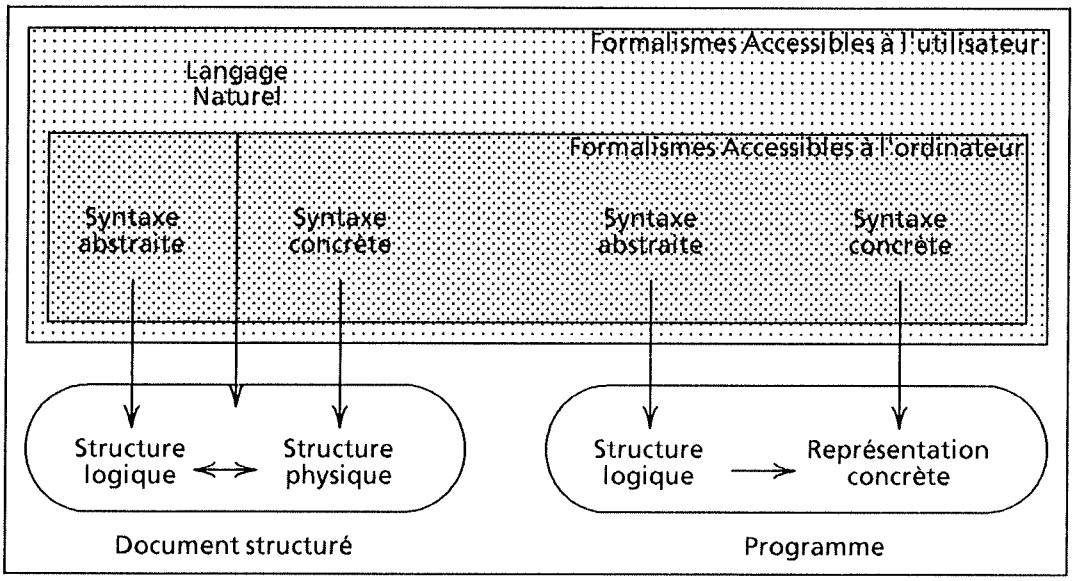


Figure III : Comparaison des formalismes définissant documents structurés et programmes

h) Adaptation nécessaire aux lecteurs

Il existe, dans le domaine des documents, une opération qui n'a pas d'équivallent pour les programmes. Cette opération, c'est la lecture. Comme le notent Vincent Quint et al. dans un article intitulé "Towards Document Engineering" [120] : « *a program is intended to be executed by a computer; a document is intended to be read by a human reader, and that makes an important difference.* »

1. On ne s'occupe pas ici de l'image physique du document telle qu'un lecteur la perçoit, mais telle qu'elle est traitée par une machine. Un changement dans l'ordre des mots influe bien entendu sur l'image du document, mais pour un programme de formatage ou de visualisation le document est inchangé.

On pourrait être tenté de rapprocher la lecture d'un document de l'exécution d'un programme [54]. Ces opérations se traduisent en effet dans les deux cas par une action ou un changement d'état chez un récepteur[1]. La lecture et l'exécution se différencient cependant de par le nombre et la diversité des récepteurs. Alors que le résultat de l'exécution d'un programme est le même sur toutes les machines compatibles avec le langage utilisé, le résultat de la lecture d'un document diffère en fonction des lecteurs (même si tous comprennent la langue utilisée). Certains trouveront le message adapté à leur cas et se formeront un certain modèle mental du contenu, d'autres n'adhèreront pas au message transmis et quelques uns se construiront des représentations erronées du contenu du messsage.

Pour éviter les mauvaises interprétations, le contenu des documents doit être adapté aux lecteurs. Pour certains documents, les lecteurs sont divisés en types auxquels correspondent des documents au contenu différent. On peut par exemple avoir, sur un même sujet technique, un document de vulgarisation, un document destiné aux lecteurs avertis et un autre à destination des spécialistes. D'autres documents impliquent trop fortement les lecteurs pour qu'une telle subdivision soit possible ; nous pensons aux documents administratifs, commerciaux et de correspondance qui doivent impérativement avoir un contenu adapté aux lecteurs.

Le projet BIBLE : une focalisation sur les problèmes quantitatifs

Ces différences ne remettent pas en cause les ressemblances conceptuelles importantes qui existent entre documents et programmes. Pour beaucoup d'auteurs, un programme constitue un type de document particulier[2] [92, 120]. Certains systèmes de manipulation de documents structurés ont d'ailleurs été dérivés de produits initialement conçus pour le génie logiciel. Dans cette catégorie, on trouve, par exemple, Mentor-Rapport [90, 91], bâti au-dessus de Mentor [40] et une recherche [11] utilisant Centaur [18]. Même les systèmes qui ont été créés spécifiquement pour les documents utilisent, en les adaptant, les concepts appliqués en génie logiciel. Parmi les éditeurs structurels développés dans le milieu universitaire on peut citer Grif [117], Lilac [20], pedtn [47], Sif [19] et Speed [32].

1. Nous utilisons ici le mot récepteur dans son acception linguistique, c'est à dire : « celui qui reçoit et décode un message réalisé selon un code spécifique. »
2. Dans le but de se rattacher à des techniques connues, le parallèle entre document et programme est souvent évoqué par les personnes qui travaillent dans le domaine de la manipulation de document. Il est intéressant de noter que ce parallèle est également fait par les personnes du génie logiciel. Dans son livre consacré à la théorie des langages de programmation [92], Bertrand Meyer présente les éditeurs structurés de la façon suivante : « *A structural editor [...] is a system for constructing and manipulating structured documents such as programs, program designs or formal specifications.* »

De nombreux produits commerciaux ont également vu le jour. Des éditeurs, bien sûr (Author/Editor, CheckMark, DynaText, Grif, SGML-Editor, TextWrite Write-It, WriterStation), mais aussi des parseurs (Mark-it, ParseStation, SGML Translator, XGML), des convertisseurs (EasyTag, FastTag, TextTagger, XTran) et des formateurs (Scribe, FrameMaker, Interleaf).

Pour nous, les différences entre documents et programmes proviennent de deux causes : un accroissement de la complexité et un accroissement de la quantité. L'accroissement de la complexité se retrouve dans la modélisation logique des données, la représentation graphique, les relations logique/physique et la sémantique utilisée. Les problèmes quantitatifs sont relatifs au nombre important de représentations logiques possibles, au nombre de vues physiques à gérer, et au nombre de récepteur qui chacun appliquent un traitement différent sur les documents.

Le projet BIBLE est concerné par deux problèmes qui touchent aux aspect quantitatifs : l'adaptation des documents aux lecteurs et l'organisation des multiples représentations logiques.

Personnalisation des documents par une édition multicouches

L'édition consiste, pour un auteur, à élaborer ou à modifier la représentation d'un document [68]. Cette définition sous entend que l'opération d'édition ne porte que sur un seul document à la fois. Il faut procéder à autant d'éditions que l'on veut de documents différents. Certains documents sont des produits uniques obtenus à la suite d'un important travail de rédaction fourni par un ou plusieurs auteurs. Dans ce groupe, nous plaçons les documentations, les articles, les comptes-rendus, les livres, les rapports, etc. D'autres documents ne nécessitent pas une rédaction complète à chaque création. Ils sont élaborés par l'adaptation de principes généraux à un contexte particulier. Nous plaçons dans cette catégorie les lettres basées sur un modèle standard, les contrats d'assurance, les relevés se situation périodique, etc.

Nous décomposons la production de ces derniers documents en deux opérations distinctes : une phase manuelle d'élaboration de ce que nous avons appelé "les principes généraux", et une phase automatique de génération des documents. Nous appelons "gabarit", l'ensemble des principes communs à l'élaboration d'un groupe de documents. Un gabarit peut être vu comme une structure abstraite de données dans laquelle certaines des parties peuvent être choisies parmi un ensemble de structures candidates. Les différentes manières d'orienter les choix dans un gabarit permettent de générer des documents différents mais bâtis sur une base commune.

La figure IV illustre une telle organisation. Dans ce schéma, les triangles représentent des structures ou des parties de structures de documents. Chaque

triangle est, soit un élément de base, soit composé d'autres triangles (qui constituent des sous-structures). Dans la structure adaptable qui figure en amont de la phase de génération, les liens représentent des choix entre plusieurs sous-structures. Les choix effectués dans la phase de génération permettent d'obtenir, de manière automatique, des documents structurés adaptés aux données et conformes à un modèle de document. La conformité des documents avec des modèles existants est effectuée lors de l'élaboration des gabarits. Cette manière de procéder permet de générer plusieurs documents en n'effectuant qu'une seule fois les contrôles liés à sa structure. La phase de génération, qui est exécutée à chaque élaboration de documents, consiste alors simplement à "exécuter", en fonction de données externes, les structures de contrôle contenues dans un gabarit.

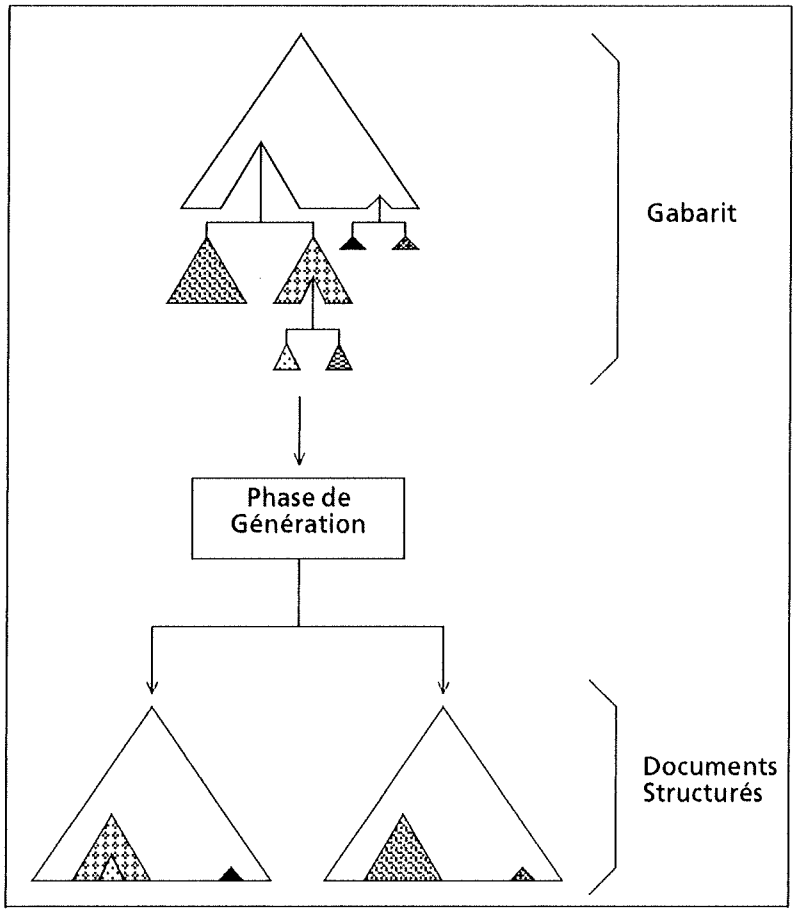


Figure IV : La génération des documents dans BIBLE.

Pour vérifier la validité d'un gabarit, nous utilisons un modèle de gabarit. Ce modèle prend en compte, d'une part les règles contenues dans un modèle de document, et d'autre part la syntaxe des structures de contrôle ajoutées.

Nous définissons un modèle de gabarit comme une extension sémantique d'un modèle de document. Les modèles de gabarit sont obtenus automatiquement par un processus de transformation à partir des modèles de documents. La génération d'un document consiste alors simplement à supprimer les structures de contrôle contenues dans un gabarit et permet d'obtenir des documents dont on est assuré de la correction structurelle.

Définition des modèles par copies différentielles

En génie logiciel, il existe une séparation entre les programmes écrits dans différents langages. Cependant, le nombre réduit de langages existants (comparé aux nombreux modèles de documents) et leur stabilité importante rend possible le développement de convertisseurs. Cette solution n'est évidemment pas transposable dans le cas des documents. Du fait de la multiplicité des modèles et de leurs interactions, il est impensable de procéder aux transformations au cas par cas. Il faut obligatoirement disposer d'un processus de transformation automatique ou tout au moins léger à mettre en oeuvre. Le même problème survient lorque l'on modifie un modèle, à ceci près que dans ce cas, la transformation des documents ne doit pas être systématique. Dans certains cas (par exemple si l'on désire conserver les représentations fidèles de documents qui ont été envoyés), il est préférable de conserver plusieurs versions d'un même modèle.

Le problème de la transformation des modèles de documents (et par là même, des documents qui leur correspondent) est prise en compte dans des systèmes ou des normes récentes [2, 4, 21, 67]. Cependant, cette problématique est abordée d'une façon statique qui peut se résumer en ces termes : on dispose de données qui sont compatibles avec un certain modèle de structuration et on veut les transformer pour les rendre compatibles avec un autre modèle. Cette manière de procéder ne permet pas, à l'heure actuelle, de définir un processus de transformation général réellement efficace [12].

Les modèles de documents ont la particularité d'être beaucoup plus proches deux à deux que les langages utilisés en programmation. On peut facilement isoler des parties communes à plusieurs modèles, comme par exemple la manière de définir un en-tête, un corps, un paragraphe ou une figure.

Dans BIBLE, nous utilisons cette caractéristique des modèles de documents pour faciliter les opérations de transformations de structure. Nous définissons les modèles de documents par copies différentielles de modèles existants. Cette opération consiste à définir un objet en terme de différences par rapport à un autre. L'objet *chaise* peut par exemple être présenté comme un objet *tabouret* auquel on ajoute un élément *dossier*. Nous avons adapté l'opération de copie

différentielle classique aux objet très structurés que sont les modèles de documents.

L'organisation que nous obtenons permet de faciliter les transformations statiques car les modèles ne sont plus des entités isolées : tous possèdent au moins un lien avec un autre modèle de la base. Par chaînages successifs, on peut définir la suite des transformations à apporter à un modèle pour en obtenir un autre et en déduire les modifications à effectuer sur les instances.

Plan de l'étude

La première partie de cette étude est consacrée à une description de l'état de l'art en ce qui concerne la modélisation et la manipulation des documents structurés.

Le premier chapitre concerne la modélisation des documents. Nous présentons les concepts de base ainsi que deux normes de représentation des documents : ODA et SGML. Nous positionnons BIBLE sur la représentation de l'aspect logique des documents.

Le second chapitre est consacré à la manipulation des documents. Nous présentons les problèmes posés par l'utilisation d'une structure et les techniques utilisées pour les résoudre. Nous concluons en posant les fondements de notre système.

La deuxième partie du mémoire est consacrée à une description du système BIBLE.

Le troisième chapitre concerne la description conceptuelle du système.

Le quatrième chapitre décrit la réalisation des différents modules composant BIBLE.

Nous concluons ce mémoire en montrant que les problèmes auxquels nous nous sommes attachés sont relativement peu étudiés à l'heure actuelle. Nous proposons quelques prolongements possibles au travail effectué et présentons l'apport de ce projet pour le groupe CDC.

Première partie

Modélisation et manipulation des documents structurés

1
La modélisation des documents

> [...] this structural viewpoint does not only allow us to lay out documents to page and print them, but also provides real processing possibilities: given the logical structure of a document, we can build up a table of contents or an index, we can automatically number sections or notes, we can scan through the document and therefore order it in different ways, we can set up different styles of page layout without changing the text itself, we can send it to other people, etc. and this is all possible without the tiresome use of *search and replace* operations that generally have to be carried out in non-structured texts.
>
> J. André, R. Furuta & V. Quint [8]

1.1 L'évolution des concepts

La représentation des documents dans le domaine de l'informatique a beaucoup évolué. Ainsi, en quelques années, nous sommes passés de représentations linéaires à des structures de données complexes définies par des modèles. Cette histoire a été marquée par plusieurs évolutions fondamentales que nous allons détailler.

1.1.1 Le balisage procédural

Dans les premiers systèmes, les documents étaient simplement représentés par des flux de caractères dans lesquels étaient insérées des commandes destinées à l'imprimante (soulignement, justification, italisation). Très vite, la nécessité de disposer d'une représentation indépendante du périphérique d'impression se fait sentir. On a donc créé des représentations révisables, plus faciles à manipuler par les utilisateurs, qui sont transformées en représentations imprimables grâce à une opération appelée formatage (figure 1.1). DCF [85], Script [135] et TROFF/NROFF [106] font partie de cette catégorie que l'on nomme systèmes à balisage.

Figure 1.1 : Représentation révisable et imprimable [71].

On range ces représentations dans les langages de balisage de type procéduraux car c'est l'auteur qui doit spécifier lui même l'opération à effectuer par le formateur. En TROFF, par exemple, les ordres destinés à l'imprimante sont précédés d'un point. La figure 1.2 donne un exemple d'un document écrit en TROFF et du résultat correspondant.

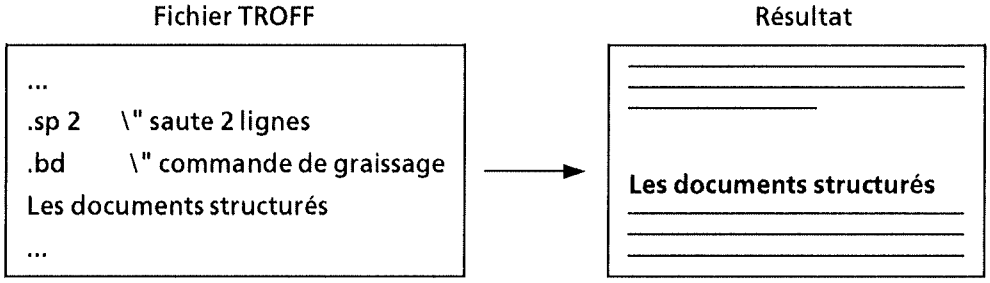


Figure 1.2 : Un document TROFF et le résultat correspondant.

1.1.2 Le balisage déclaratif

Du fait de l'étroit mélange entre contenu et présentation, il était alors très fastidieux de changer l'aspect d'un document. Une opération comme celle consistant à souligner tous les titres devait obligatoirement s'effectuer manuellement, car rien dans le document ne permettait d'identifier les suites de caractères correspondant à des titres. Ce problème a été résolu en utilisant le concept de macro-instruction déjà bien connu dans les langages de programmation. On a remplacé des suites de commandes ayant une cohérence logique par des macros, appelées balisages déclaratifs.

Dans ce type de représentation, l'auteur identifie les différents éléments qui composent un document et non la manière de les manipuler. Dans le document, chaque entité logique est repérée par un marqueur et chaque marqueur est associé à une ou plusieurs commandes de présentation. On associe par exemple au marqueur générique *Titre*, les opérations centrage et soulignement, qui doivent s'effectuer avant tout affichage d'un titre. Les modifications concernant la présentation se font ainsi très facilement en changeant quelques macros. Dans les systèmes UNIX, TROFF est par exemple livré avec un ensemble de macros élaborées par les laboratoires Bell [81]. L'utilisateur a toutefois la possibilité de déclarer des macros qui lui sont propres, tout comme dans GML [52], une extension de DCF.

Dans TROFF, la déclaration d'une macro se fait avec la commande *.de*. Les commandes de formatage utilisées dans la figure 1.2 peuvent, par exemple, être regroupées dans une macro *TI* permettant d'identifier les titres (figure 1.3). Dans un document TROFF, la macro s'utilise ensuite grêce à la commande *.TI*.

```
.de TI    \"définition de la macro
.sp 2     \" saute 2 lignes
.bd       \" commande de graissage
```

Figure 1.3 : Déclaration d'une macro TROFF.

1.1.3 Les boîtes

L'arrivée des imprimantes à haute résolution au milieu des années 70, entraîna de nouvelles recherches. Celles-ci furent destinées à exploiter au maximum les nouvelles possibilités des périphériques d'impression. L'aboutissement dans ce domaine est certainement le système TEX [77]. Basé sur le concept de boîtes, il s'adapte parfaitement à la représentation de constructions complexes comme les formules mathématiques ou chimiques.

Une boîte est une figure bi-dimensionnelle de forme regtangulaire. Elle est caractérisée par trois mesures qui sont la hauteur, la profondeur et la largeur. Deux autres caractéristiques découlent des mesures précédentes : la ligne de base et le point de référence (figure 1.4).

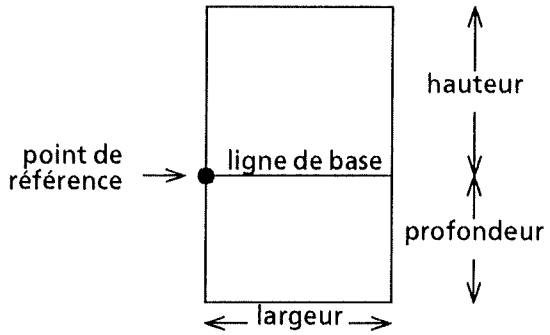


Figure 1.4 : Caractéristiques d'une boîte dans TEX [77].

Des boîtes élémentaires sont associées aux caractères. D'autres boîtes sont associées aux mots, aux lignes, aux paragraphes, etc. Un document est ainsi modélisé par une arborescence de boîtes. Cette représentation permet un formatage très précis (puisqu'il prend en compte le positionnement de chaque caractère) sur un espace à deux dimensions. De plus, la description arborescente autorise un traitement contextuel des éléments.

On peut par exemple associer une boîte particulière aux paragraphes d'un document dans laquelle on spécifie une règle de maximisation de la largeur. Lors

du formatage, la largeur attribuée à cette boîte dépendra de son emplacement dans le document (figure 1.5). TEX autorise l'utilisation de macros qui sont appelées *définitions*.

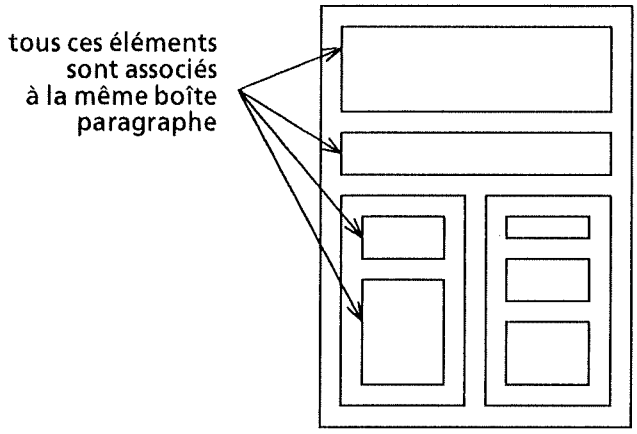


Figure 1.5 : L'emboîtement des éléments propre à un formatage contextuel.

1.1.4 Les styles

Un style est un regroupement de macros qui concerne un document particulier. Le système LATEX [79] a été conçu pour pouvoir définir des styles utilisables par les documents TEX. Avec LATEX, pour utiliser un style, il suffit de le mentionner en tête du document avec la commande :

\documentstyle{<*nom-du-style*}

Dans le document, les différents éléments sont identifiés par les macros définies dans le style (figure 1.6)

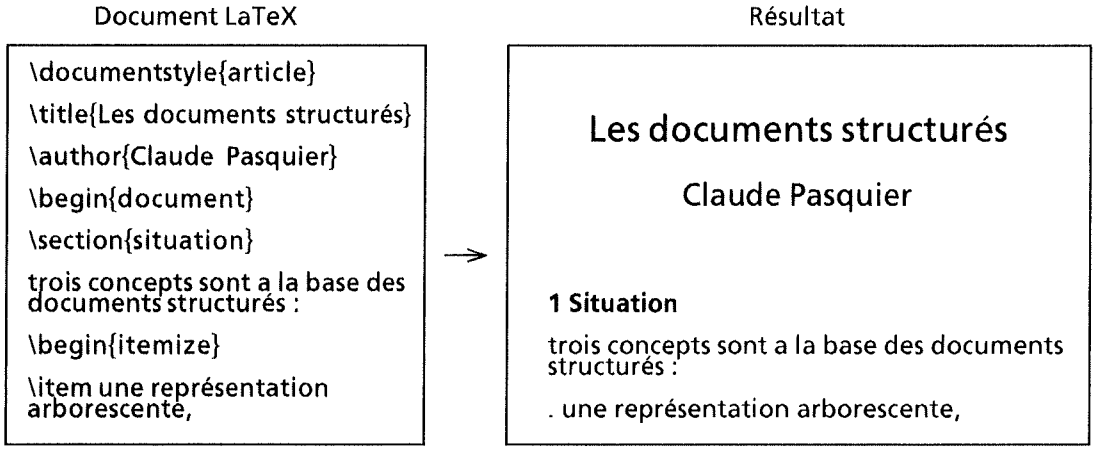


Figure 1.6 : Un document LATEX et le résultat correspondant.

1.1.5 What You See Is What You Get

Les systèmes décrits dans les sections précédentes permettent de réaliser des impressions d'une qualité remarquable. Cependant, ils sont assez difficiles à utiliser et surtout, nécessitent une abstraction de la part de l'utilisateur qui doit imaginer le résultat produit sur l'imprimante[1] par telle ou telle commande figurant dans le texte. Actuellement, TEX et LATEX sont encore beaucoup utilisés, mais presque exclusivement dans le milieu universitaire où ils constituent un standard.

Parallèlement à cette évolution, guidée par l'accroissement des performances des périphériques d'impression, une avancée a été liée à l'arrivée des micros ordinateurs. Avec l'informatique personnelle, le but n'était plus d'effectuer des impressions de grande qualité, mais plutôt, de mettre le traitement de texte à la portée du plus grand nombre. Sur les écrans graphiques des micros ordinateurs, il a été possible de représenter les documents tels qu'ils allaient être imprimés. Disparus les marqueurs et autres commandes exotiques, désormais, l'utilisateur voit directement sur l'écran, l'effet de ses actions : un titre en gras apparaît en gras, un paragraphe justifié est effectivement justifié, etc. On appelle ces systèmes WYSIWYG , pour « *What You See Is What You Get.* »

On oppose souvent les systèmes WYSIWYG, d'usage intuitif et agréable aux outils lourds et peu conviviaux que sont les systèmes de balisage. En réalité, le terme WYSIWYG regroupe des fonctionnalités que possèdent certains éditeurs mais ne constitue pas un concept permettant la catégorisation d'un système. Certains éditeurs, comme Grif [116], Interleaf [96], ou Quill [30] peuvent, du point de vue de leur utilisation, être considérés comme des éditeurs WYSIWYG. Cependant, cette apparence cache une modélisation structurelle des documents, à la différence des systèmes WYSIWYG "purs" qui eux ne prennent en compte que la présentation des documents [119]. Du point de vue conceptuel, il est plus correct de distinguer les éditeurs structurels des éditeurs dirigés par la présentation [66].

1.1.6 Les limites de l'édition libre

Les traitements de texte sur micro-ordinateurs les plus connus et les plus utilisés entrent dans la catégorie des éditeurs dirigés par la présentation[2]. Ces systèmes intègrent le concept de style qui permet de regrouper des commandes de formatages.

1. Il existe aujourd'hui des programmes de visualisation sur écran [5], ce qui accélère la conception des documents. Mais la critique reste valable puisque cette facilité ne permet pas une visualisation immédiate de l'effet des commandes utilisées.
2. Actuellement, deux systèmes se partagent la plus grande part du marché. Il s'agit des traitements de texte Word, qui en est à sa version 6.0 et Wordperfect, également en version 6.0.

Dans les versions récentes de ces systèmes, des informations sur la composition logique des documents sont déduites de l'utilisation des styles. Un style est donc tout à la fois un identifiant logique (un titre est un élément qui nomme une section) et une commande de formatage (un titre doit être affiché en gras, en taille 14, etc).

Non seulement les aspects logiques et physiques sont mélangés, mais en plus, le "logique" est dépendant du "physique". Ceci est source de nombreuses difficultés lorsque l'on désire, par exemple, changer de support d'impression.

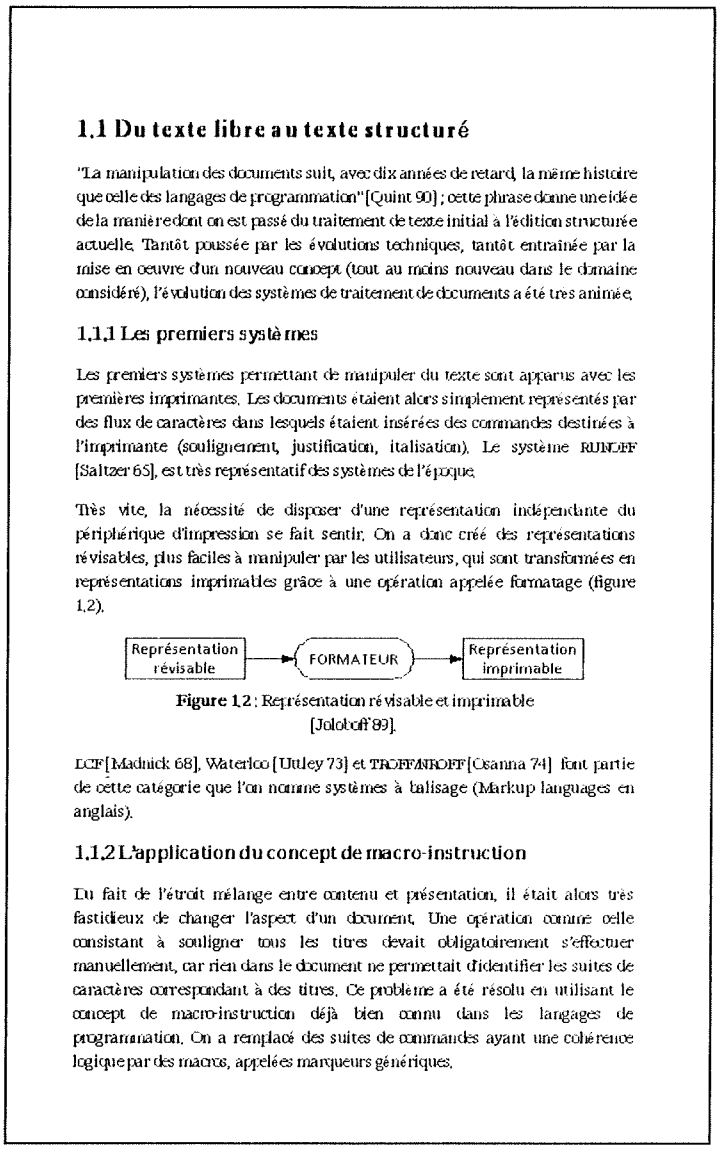
La prise en compte de la présentation et en particulier la volonté d'offrir sur un écran une image papier, limite la visualisation des documents aux seuls éléments imprimables. Or, les documents contiennent également des éléments logiques ou structurants. Comme nous l'avons vu précédemment, certains systèmes profitent des styles pour baliser logiquement les documents. Cette utilisation permet d'identifier et de différencier les éléments (imprimables) qui apparaissent dans les documents. Au lieu d'être vu comme une simple chaîne de caractères, un document est vu comme une suite d'éléments étiquetés : des titres, des paragraphes, des sous-titres, des têtes de chapitres, des figures, etc.

Cette description ne correspond pas à la vision que nous avons des documents. Pour un auteur, un document est composé de chapitres, de sections, de titres qui permettent d'identifier les chapitres, etc. Nous avons même des informations sur la manière d'agencer ces différents éléments : les sections sont incluses dans les chapitres, les légendes sont rattachées à des figures, etc. Autant de données abstraites qui ne sont pas directement liées à la présentation.

Les inconvénients d'une modélisation des documents axée sur la présentation ont été mis en évidence par Robert A. Morris dans un article de 1985 ou il nommait ces systèmes « *What You See Is All You Get* » [96]. Actuellement, la tendance est clairement à une modélisation structurée des données contenues dans un document. Ceci n'empêche pas de disposer à l'écran d'une visualisation WYSIWYG des documents, comme par exemple dans le systéme Grif, qualifié par ses auteurs d'éditeur WYSIWYG de documents structurés [116].

1.2 La représentation de la structure abstraite

Une structure abstraite est une représentation des données et des relations conceptuelles qui existent entre ces données. Elle se focalise sur la logique de structuration plutôt que sur l'aspect [87]. Les trois principaux types de modélisation abstraite de données qui sont utilisés dans le cas des documents sont les arbres, les graphes orientés et les représentations qui mixent plusieurs approches. Pour illustrer nos propos, nous basons nos exemples sur un extrait de document dont nous donnons une vision concrète dans la figure 1.7.



1.1 Du texte libre au texte structuré

"La manipulation des documents suit, avec dix années de retard, la même histoire que celle des langages de programmation" [Quint 90] ; cette phrase donne une idée de la manière dont on est passé du traitement de texte initial à l'édition structurée actuelle. Tantôt poussée par les évolutions techniques, tantôt entraînée par la mise en œuvre d'un nouveau concept (tout au moins nouveau dans le domaine considéré), l'évolution des systèmes de traitement de documents a été très animée.

1.1.1 Les premiers systèmes

Les premiers systèmes permettant de manipuler du texte sont apparus avec les premières imprimantes. Les documents étaient alors simplement représentés par des flux de caractères dans lesquels étaient insérées des commandes destinées à l'imprimante (soulignement, justification, italisation). Le système RUNOFF [Saltzer 65], est très représentatif des systèmes de l'époque.

Très vite, la nécessité de disposer d'une représentation indépendante du périphérique d'impression se fait sentir. On a donc créé des représentations révisables, plus faciles à manipuler par les utilisateurs, qui sont transformées en représentations imprimables grâce à une opération appelée formatage (figure 1.2).

Figure 1.2 : Représentation révisable et imprimable [Joloboff 89].

DCF [Madnick 68], Waterloo [Uttley 73] et TROFFANROFF [Osanna 74] font partie de cette catégorie que l'on nomme systèmes à balisage (Markup languages en anglais).

1.1.2 L'application du concept de macro-instruction

Du fait de l'étroit mélange entre contenu et présentation, il était alors très fastidieux de changer l'aspect d'un document. Une opération comme celle consistant à souligner tous les titres devait obligatoirement s'effectuer manuellement, car rien dans le document ne permettait d'identifier les suites de caractères correspondant à des titres. Ce problème a été résolu en utilisant le concept de macro-instruction déjà bien connu dans les langages de programmation. On a remplacé des suites de commandes ayant une cohérence logique par des macros, appelées marqueurs génériques.

Figure 1.7 : L'image concrète d'une page de document.

1.2.1 Représentation arborescente

Dans ce type de modélisation, un document est représenté par un arbre [18, 41]. La racine de l'arbre représente le document dans son ensemble et chaque branche de la hiérarchie correspond à une relation de type ComposéDe [7] (figure 1.8). On fait la distinction entre deux types d'objets :

- les objets composés, correspondant aux noeuds de l'arbre, qui contiennent la référence à un ou plusieurs sous-objets,
- les objets de base, correspondant aux feuilles de l'arbre, qui font référence à des contenus élémentaires.

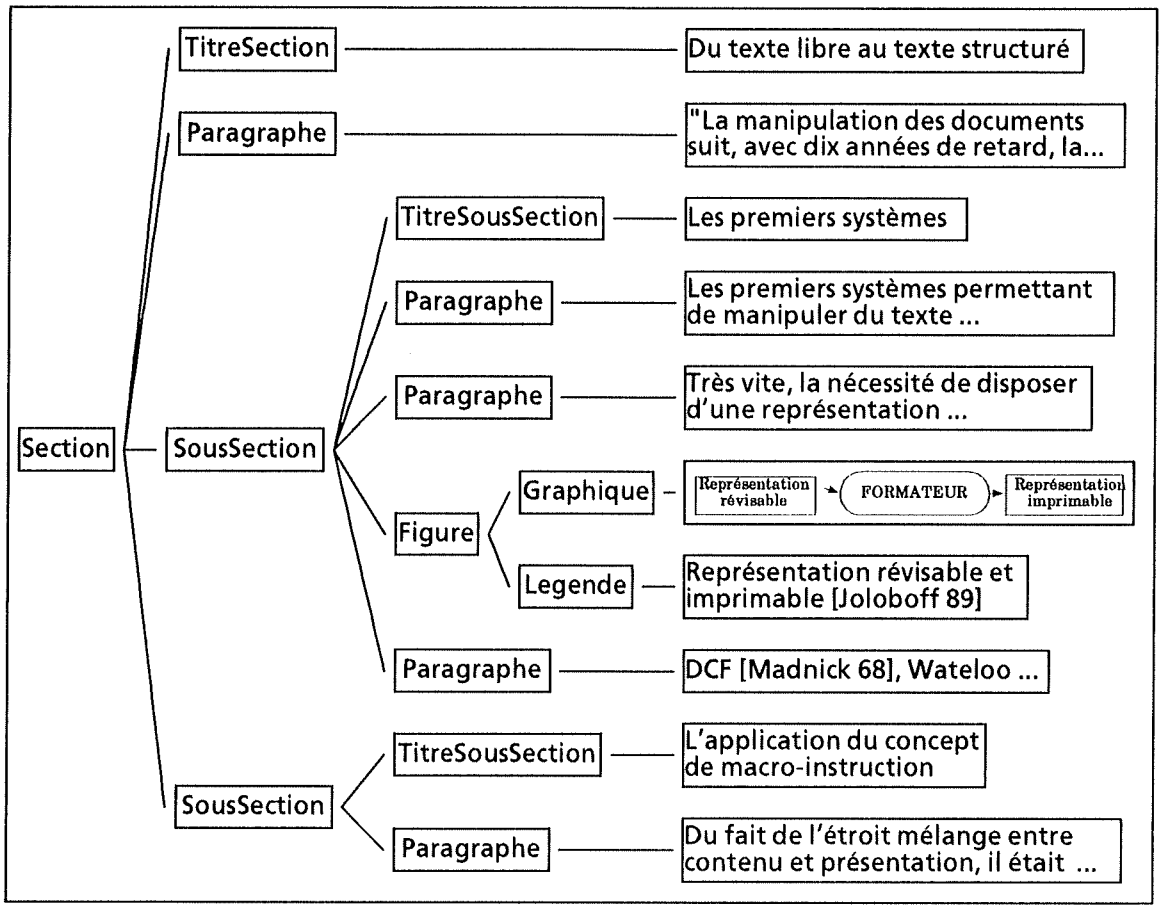


Figure 1.8 : Représentation arborescente d'un document.

La représentation sous forme arborescente est facile à manipuler et peut être décrite par des règles de syntaxe abstraite (voir § 1.3.1). Cependant, ce type de modélisation ne permet pas la prise en compte de relations autres que celles de composition (utiles pour exprimer des relations entre éléments) ni d'utiliser des sous-structures partagées. Dans la représentation arborescente de la figure 1.8, nous n'avons pas mentionné les définitions des références bibliographiques, mais il est bien évident que celles-ci doivent être déclarées de manière unique dans le document.

Certaines informations, qui apparaissent sur la vision papier du document (figure 1.7) n'ont pas à être explicitement mentionnées dans la structure logique. Les numéros de section, de sous-section ou de figure sont, par exemple, obtenus par un calcul effectué sur la structure.

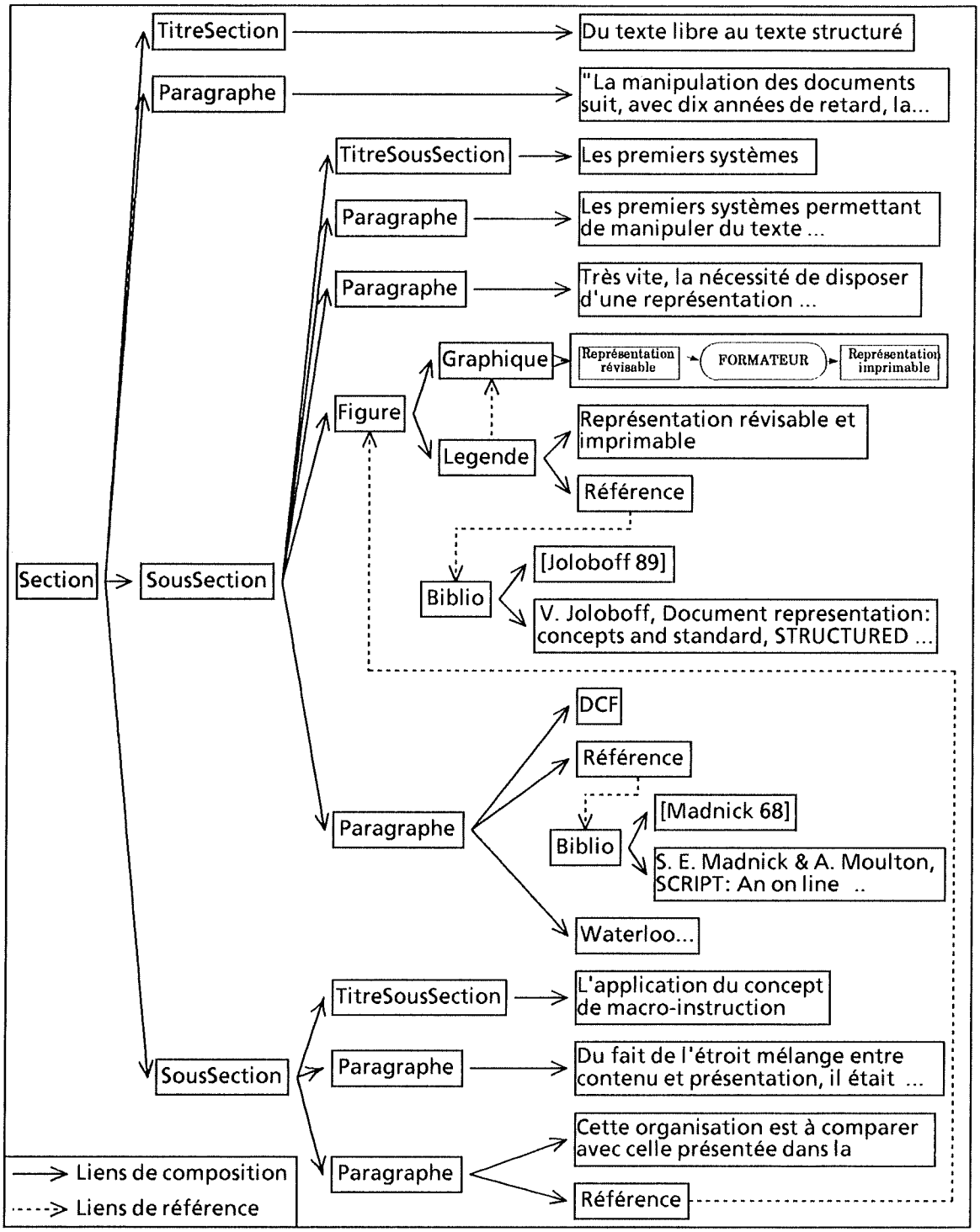


Figure 1.9 : Représentation d'un document sous forme de graphe.

1.2.2 Représentation par graphe orienté

Pour toutes les raisons évoquées précédemment, la représentation sous forme de graphe orienté apparaît mieux adaptée à la modélisation des documents structurés [64]. Alors que dans la représentation arborescente on ne considère que la relation de composition, un graphe orienté utilise des relations typées. En plus de la relation de composition, il est possible de définir des liens quelconques entre un élément source et un élément cible. La figure 1.9 est une représentation sous forme de graphe orienté de notre document de référence[1].

Si l'utilisation de graphe permet une meilleure modélisation des documents, elle conduit souvent à des performances désastreuses [26]. C'est la raison pour laquelle, dans beaucoup de systèmes, on utilise une représentation mixte qui bénéficie des avantages des arbres et des graphes.

1.2.3 Représentation mixte

Les problèmes, relevés précédemment au sujet des graphes orientés, viennent du fait que toutes les relations sont du même niveau. Or, il est clair, en regardant la figure 1.9, que la relation la plus importante est la relation de composition, les autres n'étant que secondaires. Cette constatation a donné naissance à des systèmes mixtes qui utilisent la structure arborescente comme structure principale. Dans de tels systèmes, un document est représenté par une forêt d'arbres, chacun des arbres étant bâti sur une relation de composition ou d'inclusion. Ceci constitue la structure principale d'un document. La structure secondaire est constituée de relations non compositionnelles (figure 1.10).

La plupart des opérations de manipulation du document sont réalisées en utilisant la structure arborescente : scrolling, numérotation des sections, déplacement rapide dans le document, etc. La structure secondaire est utilisée principalement pour exprimer des références entre éléments. Ceci permet tout à la fois d'obtenir de bonnes performances lors de la manipulation, et de pouvoir exprimer des relations non hiérarchiques entre les éléments. Une telle modélisation est à la base de systèmes comme Grif [116, 117] ou tnt [50, 51].

1. Bien qu'elle puisse être utilisée pour modéliser tout document structuré, la structure de graphe est plus particulièrement adaptée à la représentation des hypertextes pour lesquels la structure hiérarchique n'est pas prédominante. Dans ces systèmes, une information est représentée par un noeud et une relation par un arc orienté [31, 50].

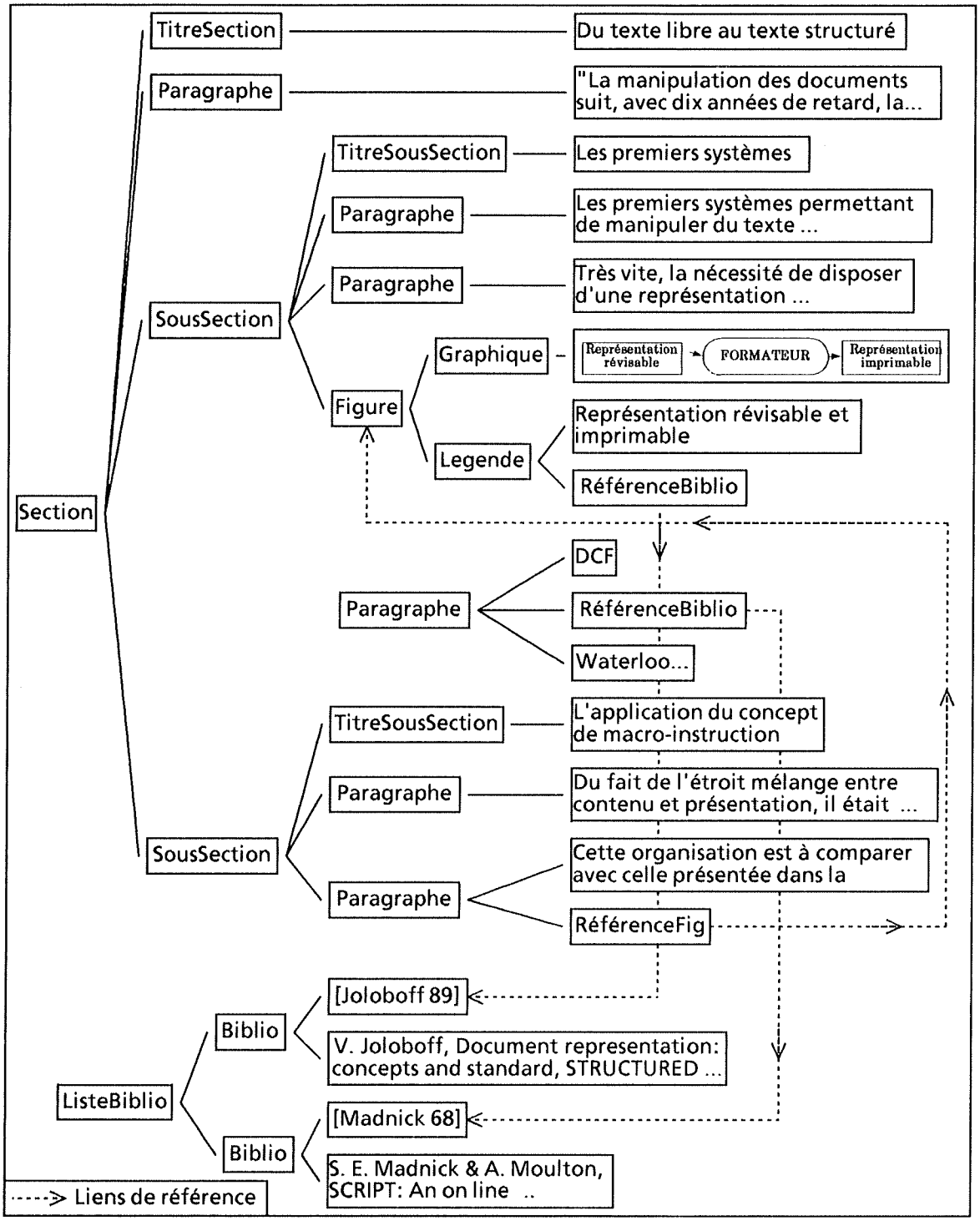


Figure 1.10 : Représentation d'un document possédant une structure arborescente principale.

Dans tous les exemples que nous avons présentés, nous avons limité la description des documents à des éléments que nous avons considérés comme élémentaires tels un paragraphe ou une figure. En réalité, ces éléments sont modélisés en utilisant d'autres structures de données plus appropriées qu'une description hiérarchique : une liste pour mémoriser les caractères d'un paragraphe, une suite d'opérations graphiques pour les illustrations vectorielles, une matrice pour les tableaux, etc. Un document est donc représenté par une structure principale arborescente, une structure secondaire constituée de liens entre les éléments et de structures de données spécifiques pour les éléments terminaux.

1.3 Les modèles de structuration logique

En génie logiciel, deux approches sont possibles pour rendre les composants logiciels plus extensibles et plus réutilisables : la généricité et l'héritage [93]. La généricité est la faculté de définir des modules paramétrés. Elle découle d'une approche statique orientée « grammaire » alors que l'héritage est une vision dynamique fondé sur le concept de la modélisation par objets. Dans le domaine des documents structurés, on a également cette alternative. L'approche orientée grammaire est utilisée pour décrire des structure purement syntaxiques mais permet également de représenter les relations non hiérarchiques d'une sémantique statique.

1.3.1 Structure syntaxique

Une grammaire de syntaxe abstraite (GSA), ou grammaire à contexte libre est un moyen d'exprimer la syntaxe d'un langage [40]. Une grammaire de syntaxe abstraite G est définie par un tuple G = (N,T,P,S) avec :

N = l'ensemble des symboles non terminaux du langage,

T = l'ensemble des symboles terminaux du langage,

P = un ensemble de règles de production de la forme $A \rightarrow \alpha$ dans laquelle $A \in N$ et α est une expression régulière dans $N \cup T$,

S = le symbole de départ, $S \in N$.

En utilisant cette notation, on définit la syntaxe de notre document de référence par la GSA suivante [46] :

G = (N,T,P,S) avec :
N = {Doc, Entête, Titre, Auteur, Corps, Section, TitreS, Parag, Illustr, IdFig, Figure, Légende, EltTextuel, RéfFig, RéfBib, SSection, TitreSS, Biblio, UneBiblio, IdBib, ContenuBib}
T = {TEXTE, GRAPHIQUE}
S = Doc
P = l'ensemble composé des règles suivantes :

Doc	→	Entête, Corps, Biblio
Entête	→	Titre, Auteur
Titre	→	TEXTE
Auteur	→	TEXTE
Corps	→	Section+
Section	→	TitreS, Parag*, SSection*
TitreS	→	TEXTE
Parag	→	Illustr | EltTextuel+
Illustr	→	IdFig, Figure, Légende
IdFig	→	TEXTE
Figure	→	GRAPHIQUE
Légende	→	EltTextuel+
EltTextuel	→	TEXTE | RéfFig | RéfBib
RéfFig	→	TEXTE
RéfBib	→	TEXTE
SSection	→	TitreSS, Parag*
TitreSS	→	TEXTE
Biblio	→	UneBiblio*
UneBiblio	→	IdBib, ContenuBib
IdBib	→	TEXTE
ContenuBib	→	TEXTE

Figure 1.11 : Représentation d'un modèle de document par une GSA..

1.3.2 Sémantique statique

Une GSA est indépendante de l'aspect physique et de la manière dont l'objet est manipulé. Elle permet de construire des arbres de composition qui respectent une syntaxe donnée.

Cependant, une GSA ne permet ni de définir des relations non hiérarchiques (comme celles qui existent entre une référence et l'objet référencé) ni des règles de

sémantique statique (spécifier par exemple qu'une même bibliographie ne doit pas être déclarée plus d'une fois).

Une sémantique statique est définie en ajoutant des contraintes sur la GSA. Plusieurs moyens peuvent être utilisés pour définir une sémantique statique : utiliser une grammaire attribuée [18, 26, 133] ou définir une sémantique dénotationnelle sur la grammaire libre [92]. Le formalisme de déclaration des deux approches n'est guère différent. Nous allons présenter une sémantique dénotationnelle de la GSA utilisée. Les grammaires attribuées seront présentées dans la section consacrée au formatage.

La sémantique dénotationnelle associe à chaque construction T, deux fonctions:

$V_T : T \rightarrow B$

$M_T : T \rightarrow D_T$

V_T est une fonction de validité qui retourne un booléen. Elle est aussi nommée condition contextuelle ou contrainte. M_T est une fonction de signification qui permet d'exprimer la sémantique dynamique de l'expression (son effet).

La sémantique statique peut être définie en utilisant uniquement les fonctions de validité. Voici comment se définissent les règles de sémantique statique concernant les références bibliographiques[1] :

```
--
-- Une référence bibliographique ne doit être déclarée qu'une fois
--
V Biblio [b: Biblio] :: = (∀i,j ∈ 1 .. b.LENGTH • i ≠ j ⊃ b(i).IdBib ≠ Ib(j).IdBib)
--
-- Une référence bibliographique utilisée doit exister.
--
-- On définit une variable globale qui contient la liste des références :
ListingBiblio: Biblio → LBiblio
ListingBiblio (b: Biblio) :: = {b(i).IdBib | i ∈ 1 .. b.LENGTH}
-- La fonction de validité d'une référence bibliographique est alors :
V RefBib [rb: RefBib] :: = (∃ i ∈ 1 .. LBiblio.LENGTH • rb = LBiblio(i))
```

Figure 1.12 : Exemple de règles de sémantique statique.

1. Pour définir les fonctions, nous utilisons les conventions suivantes :
   - LENGTH est un opérateur qui retourne la longueur d'une liste,
   - x.LENGTH retourne le nombre d'éléments de x,
   - ⊃ représente l'implication,
   - a ⊃ b signifie 'a implique b'.

Pour les références aux figures, le principe est le même. La déclaration est simplement un peu plus longue puisque l'on doit parcourir tout le corps du document pour collecter toutes les figures utilisées. Nous ne présentons pas ici les règles concernant les références aux figures. Des exemples sur la manière de procéder figurent dans le livre de Bertrand Meyer : « *Introduction to the Theory of Programming Languages* » [92].

1.3.3 Vision orientée-objet

Le principe de base d'une modélisation orientée-objet est de représenter les entités du monde réel par des structures abstraites de données que l'on appelle objet [7, 93].

L'instanciation et la composition

Dans le domaine des documents structurés, les objets manipulés sont les paragraphes, les chapitres, les titres, les figures, etc. Comme en génie logiciel, un objet particulier est obtenu par instanciation d'un objet générique. Par exemple, l'élément représentant un paragraphe particulier d'un document constitue une instance d'un objet générique *PARAGRAPHE*[1]. Dans chaque objet générique sont définis les attributs et les méthodes qui permettent de représenter et de manipuler[2] ses propres instances. L'objet générique *PARAGRAPHE* est par exemple muni d'un attribut *contenu* et d'une méthode d'affichage. On retrouve ici les notions de classe et d'instance de l'approche objet [6, 86, 93].

Certains objets ne sont pas terminaux mais sont au contraire composés d'autres objets. On les appelle objets composés ou composites. Gérald Masini et al. [86] définissent un objet composite comme « l'assemblage d'autres objets appelés ses parties ou ses composants. » Ils ajoutent que « la création d'un objet composite provoque la création d'une *instance* de chacun de ses composants. » Si l'on dispose par exemple des objets *carrosserie*, *roue* et *moteur*, on peut définir un objet *voiture* comme étant composé de quatre objets *roue*, d'un objet *carrosserie* et d'un objet *moteur*. Une voiture particulière est alors représentée par une instance de la classe *voiture* pointant sur quatre instances de la classe *roue*, sur une instance de la classe *carrosserie* et sur une instance de la classe *moteur*.

1. Dans le but de différencier les objets spécifiques des objets génériques, nous désignerons dans cette section les objets génériques par des noms écrits en majuscules.
2. Dans cette partie, nous nous contentons d'évoquer l'aspect manipulation des documents que nous étudierons en détail dans le second chapitre.

Pour la modélisation des documents, le principe est le même mais comme dans ce cas, la composition est beaucoup plus variable, on utilise des règles de production dans les objets de la structure générique [7, 68]. Un objet générique contient une fonction qui exprime les compositions possibles des instances de cet objet. Un objet générique *SECTION* est par exemple défini comme étant composé d'un objet de type *TITRES* (titre section) suivi d'une suite d'objets de type *PARAGRAPHE* et d'une suite d'objets de type *SSECTION* (sous section). Les objets spécifiques *Section1*, composé d'un titre et d'un paragraphe, ou *Section2*, composé d'un titre et de deux sous sections sont des instances possibles de l'objet générique *SECTION* (figure 1.13).

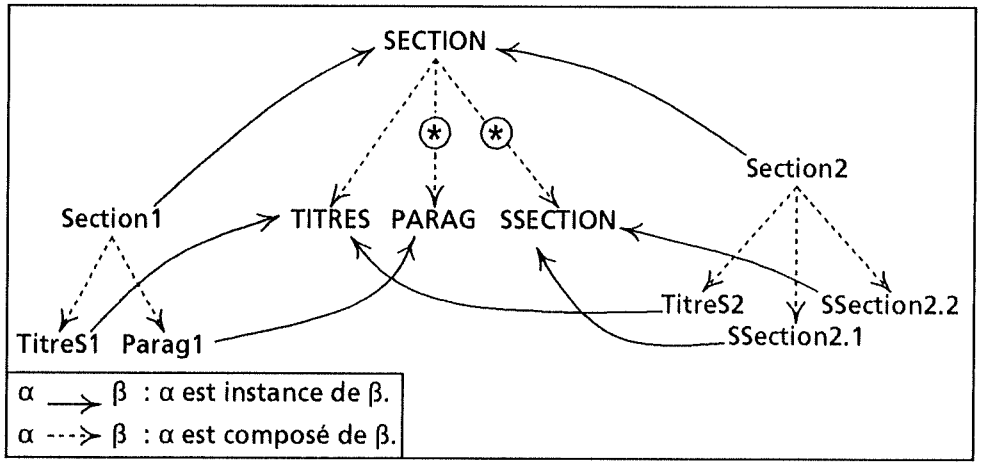


Figure 1.13 : L'instanciation d'un objet générique composé.

La spécialisation

Une relation d'héritage peut être véhiculée par le lien *composé-de*. Une taille de caractères par défaut définie dans l'objet *SECTION*, peut par exemple être héritée par les objets *TITRES*, *PARAG* et *SSECTION*. L'héritage dont nous parlons ici, et qui est quelquefois nommé héritage horizontal [45], ne doit pas être confondu avec la notion d'héritage communément admise dans les systèmes à objets. En génie logiciel l'héritage est basé sur la spécialisation d'objet. Cette relation, qui est nommée *Est-un*, permet de définir rapidement des objets qui se comportent comme d'autres objets.

Comme l'ont montré Giovanni Coray et al., il est possible d'utiliser cette relation pour modéliser les documents structurés [33]. On peut en effet regrouper les caractéristiques communes des objets génériques dans de nouveaux objets. On regroupe par exemple dans *OBJET-COMPOSITE*, les méthodes qui permettent d'accéder ou de modifier un composant, l'attribut dans lequel est mémorisée la règle de production des sous-objets, etc. De la même manière, on définit *OBJET-DE-BASE* pour regrouper les caractéristiques communes à tous les objets de

base, et *OBJET-DOC* pour mémoriser les caractéristiques communes à tous les objets utilisés dans la modélisation des documents (nom, description, méthodes de création, de suppression, etc). On peut même tirer parti de la puissance de l'héritage multiple en déclarant, par exemple, *OBJET-COMPOSITE* en tant que spécialisation d'une hypothétique classe *ARBRE* où sont regroupés tous les éléments caractérisant un arbre (figure 1.14).

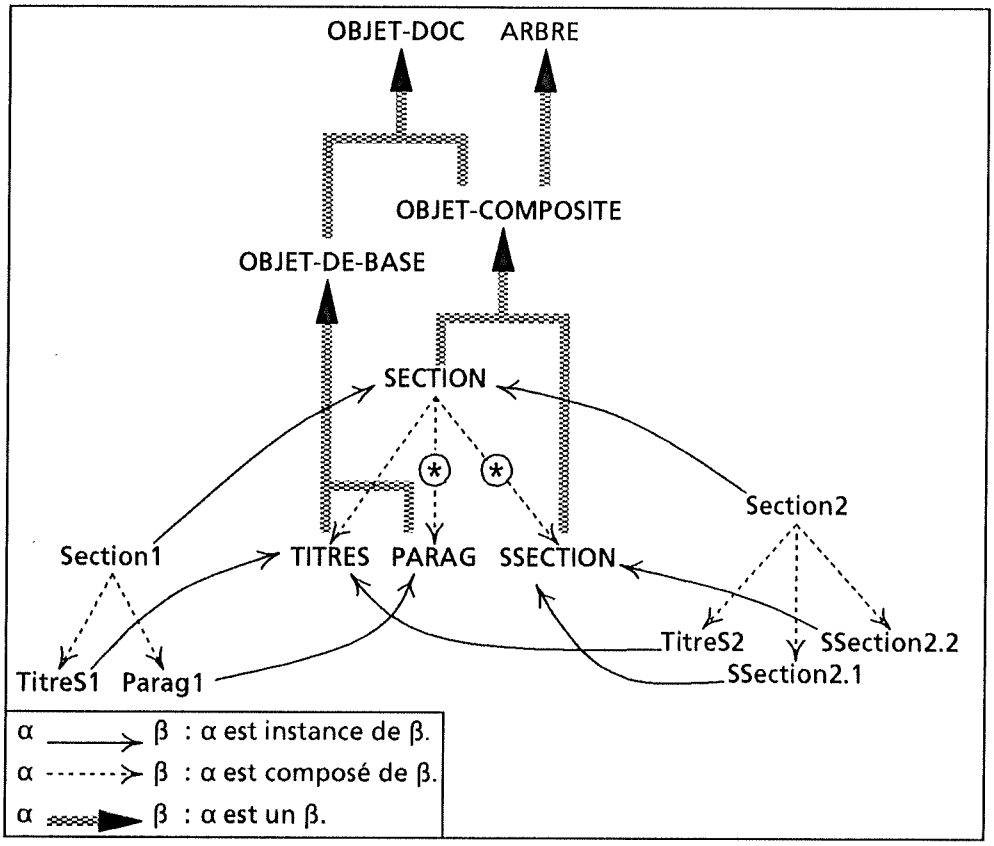


Figure 1.14 : Les trois relations utilisées pour la modélisation d'un document.

Les problèmes de conflits

En procédant de la sorte, on doit traiter les éventuels conflits provoqués par l'héritage multiple. Si, par exemple la même donnée est définie dans les objets *ARBRE* et *OBJET-DOC*, quelle est celle qui doit être utilisée dans *OBJET-COMPOSITE* ?

Dans les systèmes orientés-objet, ce problème est réglé par différents moyens. Il n'est pas dans notre but de les développer ici, mais nous les citons pour mémoire :

- en C++ [131], c'est le programmeur qui doit indiquer, lorsqu'il y a conflit, quelle est la super-classe à utiliser,

- en CLOS [130], le problème est résolu en utilisant un parcours arbitraire pour déterminer le graphe d'héritage,
- en Eiffel [93], le programmeur doit renommer explicitement l'une des deux données pour éliminer le conflit.

D'autres conflits sont générés pas la multiplicité des liens susceptibles de véhiculer l'héritage. Imaginons, par exemple, que chacun des objets de la figure 1.9 possède un attribut *ctype*[1] chargé de mémoriser le type de caractère à utiliser pour l'affichage. Plusieurs types de conflits peuvent se produire :

a) conflit lié à l'héritage horizontal

G. Masini relève que « la relation qui traduit le partage de propriétés n'est pas orientée, contrairement à l'héritage » [86]. Un composant peut hériter de l'objet composite : le type de caractère utilisé dans *Parag1* est "hérité" de celui utilisé dans *Section1*. Inversement, l'objet composite peut "hériter" d'un de ses composants : le type de caractère de *Section1* est celui utilisé pour son unique paragraphe. La résolution de ce type de conflit demeure un problème ouvert. Il n'existe pas de solution globale satisfaisante car cela dépend très fortement de la sémantique donnée aux attributs.

b) conflit entre composition et instanciation

Si une valeur par défaut de *ctype* est définie dans *TITRES*, de quoi va hériter *Titre1* ? De l'objet composite dans lequel il se trouve (*Section1*) ou de l'objet générique dont il est une instance (*TITRES*) ?

c) conflit entre spécialisation et composition

Si aucune valeur pour *ctype* n'est spécifiée dans *TITRES*, où va-t-elle être recherchée ? Dans *SECTION*, en utilisant le lien de composition ou dans *OBJET-DE-BASE* en utilisant le lien de spécialisation ?

Des méthodes pour résoudre ces conflits doivent être utilisées, en s'inspirant des solutions élaborées pour traiter les conflits liés à l'héritage multiple dans les langages de classes.

Le prototypage

La spécialisation consiste à définir une nouvelle classe d'objets en prenant pour base une classe déjà existante dont le comportement est proche. Une autre façon de représenter une hiérarchie d'abstraction repose sur la théorie des prototypes.

1. Bien que nous nous occupions de la modélisation de l'aspect logique des documents, nous avons choisi de baser nos exemples sur un attribut concernant la taille des caractères. Cet attribut physique est en effet beaucoup plus parlant qu'une caractéristique logique.

Le principe du prototypage consiste à créer un nouvel objet en prenant pour base un objet déja existant appelé son prototype [86]. Le prototypage est à la base des langages de frame comme KRL[1] [17], FRL[2] [123] ou SHIRKA [122]. L'opération qui consiste à créer un nouvel objet à partir d'un ancien est également appelée *copie-différentielle* car seules les différences entre les deux objets sont mentionnées.

Les trois langages cités reposent sur la notion de frame. Un frame est composé d'attributs ayant différentes facettes auxquelles sont associées une ou plusieurs valeurs. Un frame doit obligatoirement contenir un attribut, souvent appelé *Sorte-de*, qui établit une relation avec le frame servant de modèle. Par exemple, définissons un frame *RUY-BLAS*, spécialisation du frame *OEUVRE*. En utilisant la syntaxe FRL, cela donne :

```
(FASSERT RUY-BLAS
   (AKO ($VALUE (OEUVRE) ) )
   (TYPE ($VALUE (DRAME) ) )
   (AUTEUR ($VALUE (VICTOR-HUGO) ) ) )
```

En FRL, *FASSERT* est une fonction permettant de créer un frame et *AKO*, une abréviation de A-Kind-Of. L'expression précédente peut se traduire par : créer un frame nommé *RUY-BLAS*, qui est une sorte d'*OEUVRE* de type *DRAME* et dont l'auteur est *VICTOR-HUGO*.

Maintenant, il est possible de créer d'autres frames décrivant des oeuvres sans les définir entièrement à partir du frame *OEUVRE*. Par exemple :

```
(FASSERT LES-MISERABLES
   (AKO ($VALUE (RUY-BLAS) ) )
   (TYPE ($VALUE (ROMAN) ) )
```

Dans le frame *LES-MISERABLES*, l'attribut *AUTEUR* n'est pas spécifié. Il sera recherché en cas de besoin dans les frames supérieurs (obtenus grâce à l'attribut *AKO*). Avec ce principe, la modification d'un frame est automatiquement prise en compte dans les sous-frames. Par exemple, si l'on rajoute l'attribut *SIECLE* dans *RUY-BLAS* :

```
(FASSERT RUY-BLAS
   (SIECLE ($VALUE (XIXe) ) )
```

on peut immédiatement y accéder dans le frame *LES-MISERABLES* avec la fonction *FNEED* :

```
? (FNEED 'LES-MISERABLES 'SIECLE)
= (XIXe)
```

1. *Knowledge Representation Language.*
2. *Frame Representation Language.*

L'utilisation des prototypes dans la modélisation des documents pose des problèmes non traités par les langages de frame. Tout d'abord, parce que ces représentation ne permettent pas de représenter des attributs dont les valeurs sont des listes ordonnées. Dans les trois langages présentés, on peut modéliser un objet composé d'autres objets. Il est possible d'ajouter de nouveaux objets ou d'en supprimer mais pas dans insérer à un emplacement précis dans une liste.

L'autre limite est que ces systèmes ne permettent pas de manipuler de manière efficace des objets possédant une structure très « profonde ». Nous qualifions de profonde, la structure d'un objet qui est composé d'autres objets qui sont eux-mêmes composés d'autres objets, etc.

Pour illustrer les difficultés provoqués par la structure profonde des documents, prenons un exemple :

Soit un objet *Automobile* décrit sous la forme d'un prototype composé de deux éléments : un objet *Moteur* et un objet *Carrosserie* (figure 1.15). Supposons que l'on crée, à partir de l'objet *Automobile*, un nouvel objet que nous appellons *NouvAuto* qui est composé des deux éléments figurant dans *Automobile* et d'un objet *Chassis*. L'objet *NouvAuto* utilise les mêmes objets *Moteur* et *Carrosserie* que *Automobile* (figure 1.16). Cette organisation permet de prendre automatiquement en compte dans l'objet *NouvAuto* toutes les modifications apportées aux éléments *Moteur* et *Carrosserie* de *Automobile*.

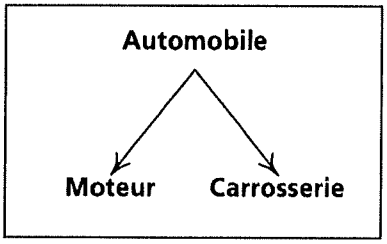


Figure 1.15 : Représentation d'un objet *Automobile*.

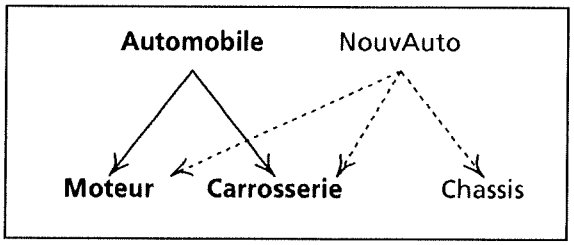


Figure 1.16 : Représentation des objets *Automobile* et *NouvAuto*.

Regardons maintenant ce que donne cette organisation dans le cas de la représentation d'un document. La figure 1.17 est l'illustration d'un prototype *Document* qui est composé des éléments *Section1* et *Section2*. *Section1* est composé de *TitreS1* et de *Parag1*. *Section2* est composé de *TitreS2*, *SSection2.1* (élément qui est à son tour composé de *TitreSS2.1* et de *Parag2*) et de *SSection2.2*. Supposons que l'on crée, à partir de ce prototype, un nouveau document, appelé *Document2*, dans lequel est l'ajouté un élément *Parag3* dans *SSection2.1*. Cette simple modification nécessite de créer un nouvel élément *SSection2.1'* qui contient les objets *TitreSS2.1*, *Parag2* et *Parag3*. Dans *Document2* l'objet *Section2* doit être composé de *TitreS2*, de la nouvelle *SSection2.1* (appelée *SSection2.1'*) et

de *SSection2.2*. On doit donc créer un nouvel objet *Section2'* pour matérialiser cette composition (figure 1.18).

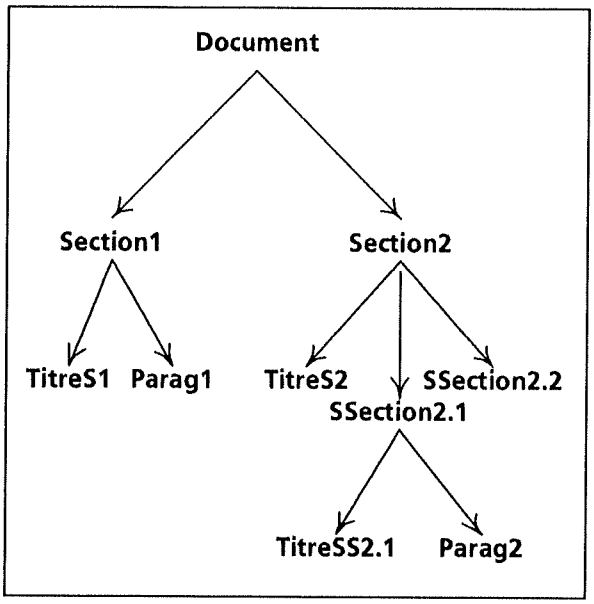


Figure 1.17 : Représentation d'un prototype *Document*.

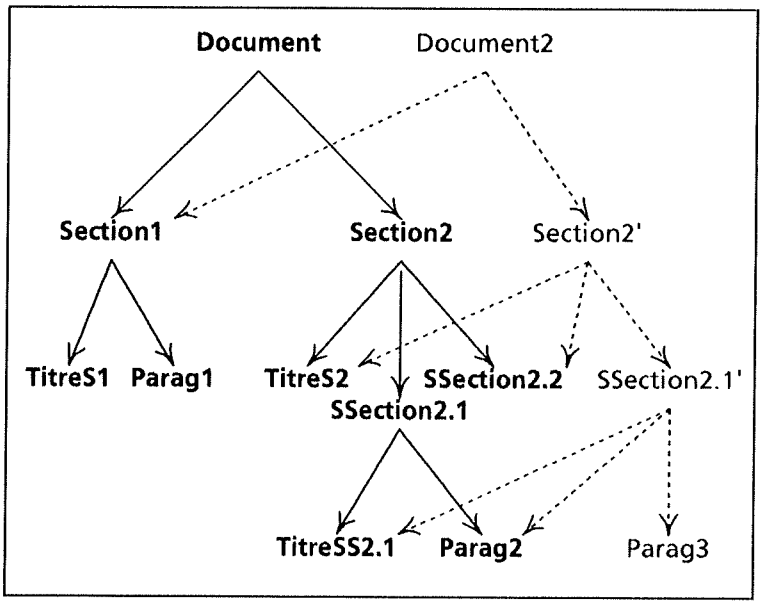


Figure 1.18 : Représentation des prototypes *Document* et *Document2*.

En résumé, avec ce type de modélisation, une modification apportée à un objet dans une hiérarchie nécessite de créer autant de nouveaux objets qu'il y a de niveaux d'emboîtement jusqu'à la racine. La gestion de ces multiples objets est

très difficile et n'est pas correctement traitée dans les systèmes de frame existants. Supposons par exemple que l'on ajoute un paragraphe dans la deuxième section de *Document*. Cette opération va provoquer un changement dans l'objet *Section2*, mais cette modification ne sera pas prise en compte dans *Document2* puisque dans ce dernier, *Section2* est remplacé par *Section2'*.

1.3.4 La modularité dans la représentation des documents

La modularité est un concept très important en génie logiciel. Elle permet d'opérer une structuration propre à une bonne réutilisation des composants. Il est tout naturel de vouloir appliquer ce concept au cas des documents [82, 120, 126]. Une description modulaire peut être utilisée pour structurer les documents, mais aussi les modèles [118].

Un document structuré peut être modélisé sous la forme d'un assemblage de sous-structure stockées de manière séparée. Ceci permet d'utiliser une même sous-structure dans plusieurs documents différents (figure 1.19). Le problème lié à cette organisation a déjà été évoqué lorque nous avons présenté le prototypage. Tout d'abord, la structure partagée doit être suffisament indépendante pour ne pas nécessiter de modifications liées à son utilisation dans un contexte donné. Dans la figure 1.19, par exemple, il n'est pas possible d'effectuer des modifications sur la structure commune qui ne soient prisent en compte que dans la *Structure1*. D'autre part, la modularité n'est pas très appropriée à la représentation de structures très proches ; comme, par exemple, celles qui sont illustrées dans la figure 1.18.

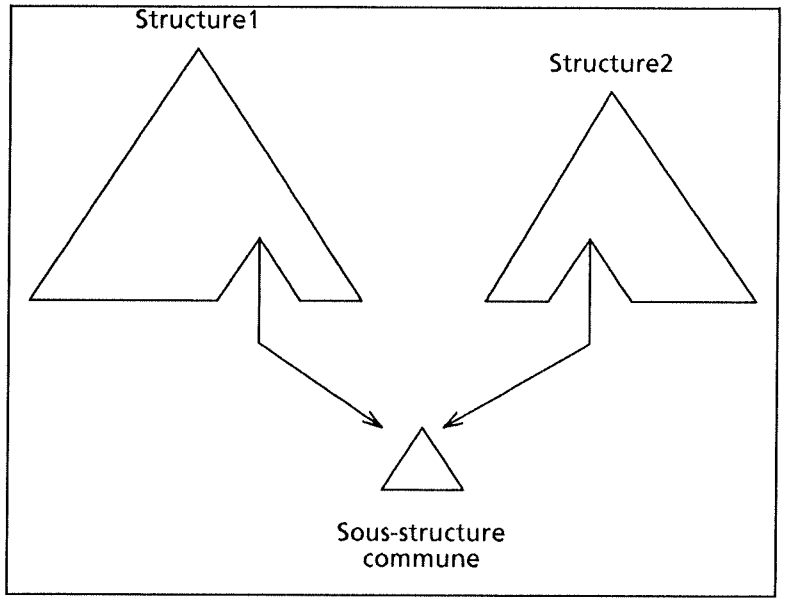


Figure 1.19 : Le partage d'une sous-structure commune.

L'utilisation de liens typés

Pour résoudre ces problèmes, il est possible d'opter pour une vision proche de celle utilisée dans les bases de données orientées objet [28]. Le principe est de représenter les relations de composition entre les éléments par des liens typés. Plusieurs structures peuvent être représentées conjointement en optimisant la réutilisation des parties identiques. La reconstitution des différentes structures est effectuée en parcourant la base d'une certaine manière. Dans la figure 1.20, ce concept est utilisé pour représenter les deux document de la figure 1.18. Dans ce schéma aucun élément identique n'est dupliqué. La structure des deux documents se retrouve en parcourant, à partir du noeud commun *Racine*, soit les liens étiquetés (1) pour obtenir la structure de *Document*, soit les liens étiquetés (2) ce qui permet d'obtenir la structure de *Document2*. La représentation de la figure 1.20 est à comparer avec celle équivalente de la figure 1.18.

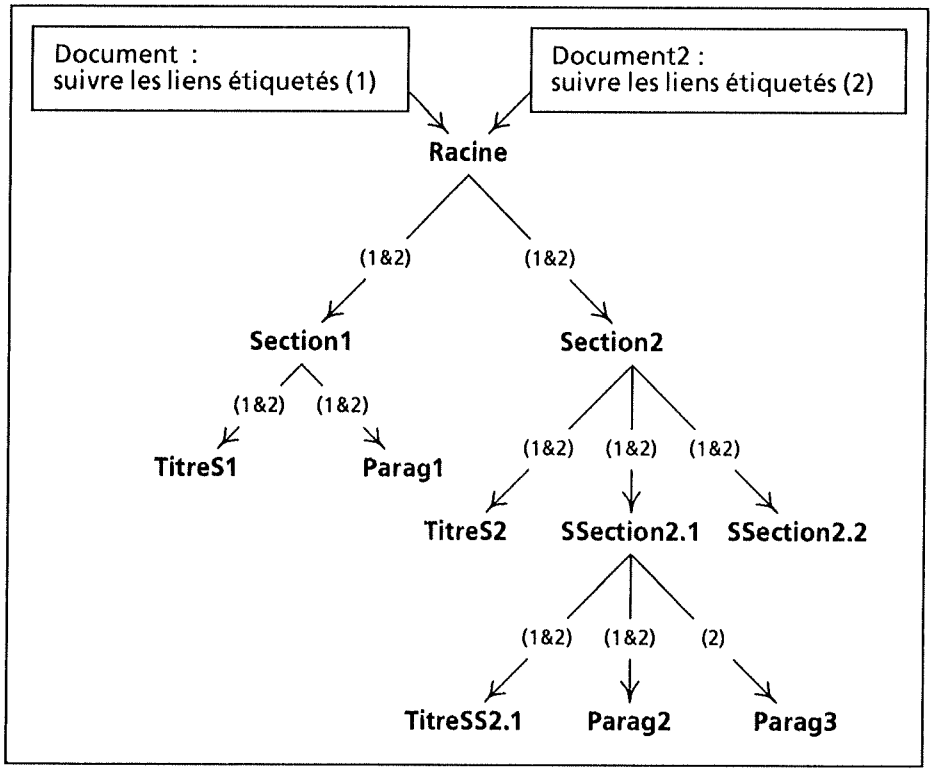


Figure 1.20 : Représentation des structures *Document* et *Document2* en utilisant des liens typés.

1.4 La représentation concrète

Nous avons vu que pour les documents, l'obtention d'une présentation découle d'un processus beaucoup plus complexe que pour les programmes. Mais l'accroissement de la complexité n'est pas la seule différence entre programmes et documents. Nous nous intéressons dans cette section à la manière dont est décrite une présentation. Soit elle est vue comme une décoration de la structure logique, soit elle est modélisée par une structure générique.

1.4.1 Une dérivation de la structure logique

La structure physique des programmes est obtenue en effectuant des traitements sur la structure logique. Cette situation est illustrée par le schéma de la figure 1.21.

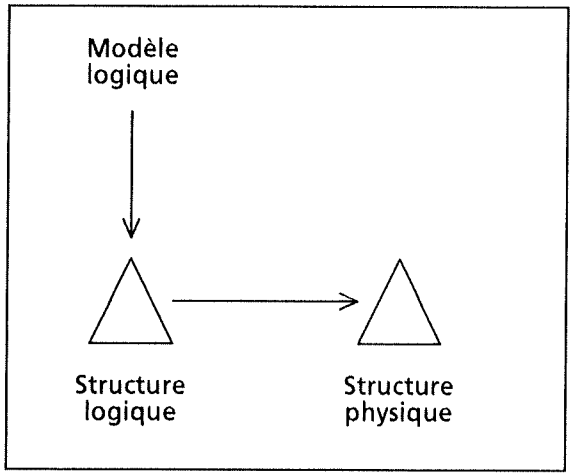


Figure 1.21 : Structure physique obtenue par dérivation d'une structure logique.

Beaucoup de systèmes de manipulation des documents utilisent cette même technique. Le principe est d'associer une présentation à chacun des éléments de la structure logique. La structure physique est obtenue en parcourant la structure logique et en lui appliquant des règles de présentation.

Cette manière de procéder n'est cependant pas assez puissante pour résoudre certains cas qui se posent dans le traitement des documents. Un bon exemple qui a souvent été cité, est celui de l'affichage des documents multilingues : une structure logique, composée de deux parties (par exemple anglais et français), doit être disposée en deux colonnes sur un document, les paragraphes correspondants des deux langues étant situés en regard l'un de l'autre (figure 1.22). Il est évident que dans un tel cas, la structure physique ne peut être obtenue par simple dérivation de la structure logique. La manière dont est constituée la

représentation concrète dépend en effet de règles qui sont indépendantes de la structuration logique.

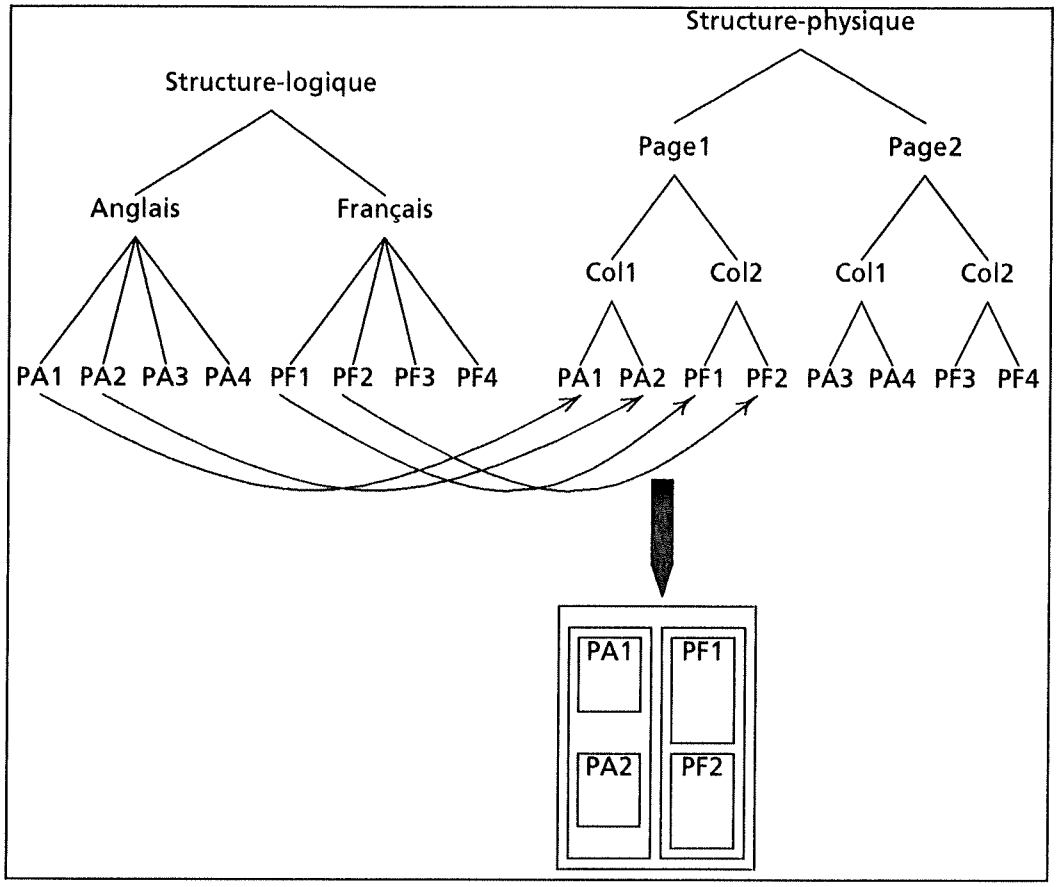


Figure 1.22 : Un cas où la structure physique ne peut être obtenue par dérivation de la structure logique.

1.4.2 Une structure physique définie par un modèle

Pour résoudre le cas précédent, il faut associer un modèle à la structure physique. La présentation n'est alors plus obtenue par transformation de la structure logique mais grâce à l'adoption de la structure logique par un autre modèle (figure 1.23).

Nous reviendrons sur les différentes techniques utilisées pour passer d'une structure logique à une structure physique dans une prochaine section consacrée à la décompilation (§ 2.2).

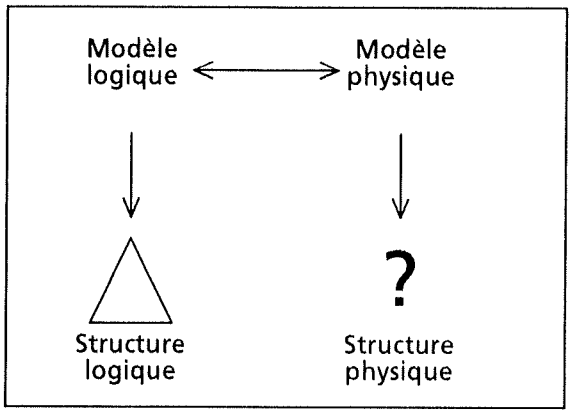


Figure 1.23 : Elaboration de la présentation en utilisant un modèle physique.

1.5 Les normes de représentation actuelles

Les concepts qui sont à la base des documents structurés sont utilisés dans deux normes ISO : SGML [69] et ODA [68]. Ces deux normes illustrent bien les deux facettes de la modélisation des documents : SGML découle d'une approche procédurale et ODA d'une approche objet.

1.5.1 SGML

SGML (Standard Generalized Markup Language) est une norme internationale de représentation des documents codifiée par l'ISO en 1986. C'est un méta-langage qui permet aux utilisateurs de définir leurs propres syntaxes pour décrire la structure logique des documents.

SGML est conçu pour être le plus universel possible. De ce fait, il est indépendant des périphériques utilisés et ne prend en compte que la codification du texte. Des formats spécifiques permettant de codifier d'autres types de contenus peuvent cependant être utilisés conjointement à SGML.

Un exemple est donné par le projet CALS (Continuous Acquisition and Lifecycle Support[1]) du département de la défense américaine dans lequel SGML est utilisé ou est prévu d'être utilisé avec IGES (Initial Graphics Exchanged Specification) pour les graphismes en 3D, CCITT groupe 4 pour les images numérisées, CGM (Computer Graphics Metafile) pour les graphismes en 2D, PDES (Product Data Exchange Specification) pour la représentation des données CAO, DSSSL (Document Style Semantic and Specification Language) pour la mise en page et SPDL (Standard Page Description Language) pour le formatage.

1. La signification de cette acronyme a été modifié en 1993. Avant cette date, CALS était utilisé pour « Computer-Aided Acquisition and Logistics Support. »

La DTD

```
01 <!ENTITY     % doctype        "Doc"                          >
02 <!ENTITY     % EltTextuel     "#PCDATA | RefFig | RefBib"    >
03 <!ENTITY     % refs           "RefFig | RefBib"              >
04 <!ELEMENT    Doc          --  (Entete, Corps, Biblio)        >
05 <!ATTLIST    Doc
06              type (final, draft) draft                       >
07 <!ELEMENT    Entete       --  (Titre, Auteur)                >
08 <!ELEMENT    Corps        --  (Section+)                     >
09 <!ELEMENT    Section      --  (TitreS, Parag*, SSection*)    >
10 <!ELEMENT    Parag        --  (Illustr | (%EltTextuel;)+)    >
11 <!ELEMENT    Illustr      --  (Figure, Legende)              >
12 <!ATTLIST    Illustr
13              id ID #IMPLIED                                  >
14 <!ELEMENT    Legende      --  (%EltTextuel;)+                >
15 <!ELEMENT    SSection     --  (TitreSS, Parag+)              >
16 <!ELEMENT    Biblio       --  (UneBiblio*)                   >
17 <!ELEMENT    UneBiblio    --  (IdBib, ContenuBib)            >
18 <!ATTLIST    UneBiblio
19              id ID #IMPLIED                                  >
20 <!ELEMENT    Figure       -O  EMPTY                          >
21 <!ATTLIST    Figure
22              ref ENTITY #IMPLIED                             >
23 <!ELEMENT    (%refs;)     -O  EMPTY                          >
24 <!ATTLIST    (%refs;)
25              rid IDREF #REQUIRED                             >
26 <!ELEMENT    (Titre, Auteur, TitreS, TitreSS, IdBib, ContenuBib)
27                           --  (#PCDATA)                      >
```

Figure 1.24 : La DTD *Doc*.

La structure logique des documents est décrite par une DTD (Document Type Definition) (figure 1.24). Une DTD est composée de lignes identifiées par les symboles de début "<!" et de fin ">". La définition de la structure est identifiée par le mot clé ELEMENT placé en début de ligne. Elle est composée schématiquement d'une partie gauche qui contient le nom d'un élément et d'une partie droite qui indique son contenu. La grammaire comprend six opérateurs relationnels divisés en deux catégories [59] : les connecteurs, qui indiquent les

relations existantes entre plusieurs éléments, et les indicateurs d'occurence qui spécifient le nombre de fois qu'un élément peut apparaître dans un document.

Dans la catégorie des connecteurs, on a :

- la séquence, représentée par une virgule (","), qui indique que les éléments apparaissant de part et d'autre de la virgule doivent figurer dans le même ordre dans le document instance,
- l'agrégation, représentée par un et commercial ("&"), qui indique que les éléments apparaissant de part et d'autre du connecteur doivent figurer dans le document instance, mais dans un ordre quelconque,
- le choix, symbolisé par une barre verticale ("|"), qui indique que l'un (et un seul) des éléments apparaissant de part et d'autre du connecteur doit figurer dans le document instance.

Parmi les indicateurs d'occurence on distingue :

- l'option, symbolisée par un point d'interrogation ("?"), qui indique qu'un élément est optionnel,
- la répétition, symbolisée par un plus ("+"), qui indique qu'un élément peut apparaître une ou plusieurs fois,
- la répétition optionnelle, représentée par un astérisque ("*"), qui indique qu'un élément peut apparaître zéro ou plusieurs fois.

SGML ne traite que les contenus de type textuel. Ceci est exprimé par le mot clé #PCDATA pour Parsed Character DATA. Dans un élément de type #PCDATA ne peuvent figurer que des caractères alphanumériques différents de ceux utilisés pour représenter les tags ('<', '>' et '/'). D'autres types de contenus existent suivant la manière dont ils doivent être traités par un analyseur SGML (CDATA, RCDATA, SDATA, NDATA, EMPTY, ANY) [59]. Dans notre définition de DTD, nous avons utilisé le type EMPTY (lignes 20 et 23) qui permet de définir des éléments non saisis par l'utilisateur. Des suites de caractères fréquemment utilisées dans la DTD peuvent être définies avec la commande ENTITY (lignes 2 et 3). Les entités ainsi définies sont remplacées dans le corps du document avant l'analyse. La commande ENTITY utilisée dans la ligne 1 permet de donner un nom au modèle de document. Ce nom est celui qui est utilisé pour faire la correspondance entre un document et son modèle.

Des attributs peuvent être attachés à n'importe quel élément. La déclaration des attributs se fait avec la commande ATTLIST. Dans les lignes 5 et 6, on déclare un attribut *type*, associé à l'élément *Doc* qui peut prendre deux valeurs, *final* ou *draft*, la valeur par défaut étant *draft*. Les attributs sont également utilisés pour identifier des éléments. Dans ce cas, l'attribut est de type ID, ce qui lui impose d'être unique dans le document (lignes 13 et 19). Le type IDREF, également utilisé, permet de définir des attributs qui font référence à d'autres attributs de type ID

(ligne 25). #IMPLIED signifie que l'attribut est déterminé par le système et #REQUIRED indique qu'il doit être obligatoirement spécifié.

Le document SGML

```
<!DOCTYPE Doc SYSTEM>
<!ENTITY FIG1 SYSTEM "dessin">
<Doc type=final>
<Entete>
<Titre>La génération de documents personnalisés</Titre>
<Auteur>C. Pasquier</Auteur></Entete>
<Corps><Section><TitreS>Du texte libre au texte structure</TitreS>
<Parag>La manipulation des documents suit, avec dix années de retard,
...</Parag>
<SSection><TitreSS>Les premiers systèmes</TitreSS>
<Parag>Les premiers systèmes permettant de manipuler du texte ...</Parag>
<Parag>Très vite, la nécessité de disposer d'une représentation ... </Parag>
<Parag><Illustr id=f01><Figure ref=FIG1>
<Legende>représentation révisable et imprimable<RefBib rid=b01>.
</Legende></Illustr></Parag>
<Parag>DCF <RefBib rid=b02> Waterloo ...</Parag></SSection>
<SSection><TitreSS>L'application du concept de macro-instruction</TitreSS>
<Parag>Du fait de l'étroit mélange entre contenu et présentation, il ...</Parag>
<Parag>Cette organisation est à comparer avec celle présentée dans la
<RefFig rid=f01>...</Parag></SSection></Section></Corps>
<Bilio><UneBilio id=b01><IdBib>[Joloboff 89]</IdBib>
<ContenuBib>V. Joloboff, Document representation: concepts and standards,
STRUCTURED ...</ContenuBib></UneBiblio>
<UneBilio id=b02><IdBib>[Madnick 68]</IdBib>
<ContenuBib>S. E. Madnick & A. Moulton, SCRIPT: An on line ...</ContenuBib>
</UneBiblio></Bilio></Doc>
```

Figure 1.25 : Exemple d'un document balisé[1].

1. En SGML, seuls sont pris en compte les caractères alphanumériques ASCII. Pour représenter des caractères spéciaux comme les caractères accentués, il faut définir dans la DTD des entités qui sont ensuite utilisées dans le document. On appelle par exemple *acirc*, le caractère français « a circonflexe » et on y fait référence par *â* dans le document. Le mot « tâche » sera par exemple codifié par *tâche*. Dans les exemples que nous mentionnons, nous ne tenons pas compte de cette particularité de SGML.

Un document SGML commence toujours par le mot clé DOCTYPE qui permet de spécifier la DTD auquel il est rattaché. Le contenu du document est balisé avec le nom des éléments définis dans la DTD encadrés par les délimiteurs "<" et ">". Pour indiquer la fin d'un élément, on utilise une balise de fin repérée par un "/" avant le nom de l'élément (figure 1.25). La valeur des attributs est spécifiée dans la balise de début. Un document peut utiliser des parties définies de manière externe comme l'illustration par exemple pour laquelle on associe une entité que l'on nomme FIG1 au fichier "dessin". Dans un document SGML, les références sont uniquement spécifiées de manière déclarative. C'est au système de manipulation qu'il incombe d'effectuer les traitements nécessaires.

1.5.2 ODA

ODA (Open Document Architecture) [68] est un standard international permettant d'échanger des documents multimédias entre des systèmes informatiques hétérogènes.

Les travaux d'élaboration d'un standard permettant de manipuler et d'échanger des documents composites ont commencé au début des années 80. La première version de ODA a été publiée en 1985 par l'European Computer Manufacturers Association sous le nom de ECMA-101. Son acceptation par l'ISO, en tant que standard international, a eu lieu en 1988.

Le but d'un standard tel que ODA, est de prendre en compte tous les types de documents possibles. La norme a donc été conçue comme un noyau définissant les différents concepts employés. Autour de cette base, différents profils ont été définis pour permettre la modélisation des documents à plusieurs niveaux de complexité.

Actuellement, quatre profils (ou DAP : Document Application Profile) sont reconnus au niveau international :

- Q111 [112] permet de modéliser les documents ne contenant que du texte,
- Q112 [113] supporte le texte, les graphiques géométriques et les images digitalisées,
- Q113 [114] autorise la manipulation de documents de type PAO,
- Q121 [115] permet l'échange de données textuelles simples par des systèmes de messagerie.

Q111, Q112 et Q113 (mais pas Q121) respectent une compatibilité ascendante. Ainsi, un système utilisant le profil Q111 peut être compris par des systèmes travaillant en Q112 ou en Q113. D'autres profils prenant en compte la modélisation des tableaux, du son ou des images animées sont prévus.

La norme ODA constitue une architecture abstraite pouvant être implantée sur divers matériels. Pour pouvoir communiquer, un format d'échange a

également été normalisé. Il s'agit de ODIF (ODA Interchange Format) qui utilise la notation ASN.1.

Les concepts de base

Dans ODA, la structure logique et la structure physique d'un document sont deux structures indépendantes. Elles peuvent être manipulées séparément, mais elles pointent toutes les deux sur les mêmes contenus [68, 88].

Contrairement à SGML, la norme fait la distinction entre les trois types de contenus suivants [25] : les caractères, les graphiques géométriques et les bitmaps.

Les structures logiques et physiques sont respectivement nommées dans ODA structure logique spécifique et structure physique spécifique. On utilise le qualificatif 'spécifique' pour les différencier des modèles, eux mêmes représentés par des structures appelées structures génériques. Dans ODA, les structures spécifiques sont considérées comme des instances des structures génériques.

On touche ici à l'une des spécificités de ODA qui en fait toute sa puissance. Dans beaucoup de systèmes, la structure physique d'un document est obtenue par l'application de règles sur la structure logique. Dans ODA, l'aspect physique dispose d'une structure propre qui peut être totalement indépendante de l'aspect logique. Les liens entre les deux structures s'effectuent dans les deux sens.

La grammaire utilisée dans ODA pour définir les modèles comporte 5 opérateurs de base [7] :

- la séquence (SEQ),
- le choix (CHO),
- la répétition (REP),
- l'option (OPT),
- et la répétition optionnelle (OPTREP).

Les styles

Les styles sont divisés en styles de présentation et styles de mise en page (layout style).

Les styles de mise en page sont associés à la structure logique. Ils permettent de spécifier, par exemple, qu'un paragraphe particulier ne doit pas être coupé ou que les figures doivent apparaître sur les mêmes pages que leurs légendes. Ils sont également utilisés pour spécifier les liens entre une structure logique et une structure physique. Par exemple : l'objet logique *paragraphe* utilise le style de mise-en-page *P1*, et le style *P1* consiste à dire que l'objet est disposé dans l'élément physique *bloc-para*. De cette manière, les structures logiques et physiques peuvent être indépendantes.

Les styles de présentation s'appliquent aux objets de base, tant logiques que physiques. Ils permettent de contrôler l'affichage des contenus en indiquant, par exemple, pour un contenu de type texte, la police de caractère utilisée, l'espacement entre les lignes, etc.

Les trois classes d'architecture de documents

ODA prévoit d'échanger les documents sous trois formes différentes qui sont appelées classes d'architecture de document [124]. Le choix de la classe d'architecture est fonction des informations disponibles et de l'usage qui va être fait du document transmis. Les trois formes d'échange sont : retraitable, formaté et formaté retraitable.

Un document retraitable comprend une structure logique et des contenus. Il peut être retravaillé sur un éditeur.

Un document formaté contient une structure physique et les contenus associés. C'est le format utilisé pour réaliser l'impression ou pour obtenir une visualisation WYSIWYG du document sur un terminal. La modification du contenu n'est pas possible.

Un document formaté retraitable est composé d'une structure logique, d'une structure physique et de contenus. Ce format permet de visualiser et d'éditer le document.

Le profil document

Le profil de document contient les informations relatives au document dans son ensemble. On distingue :

- les données concernant la gestion du document comme le nom de l'auteur ou la liste des mots clés (ces informations sont utilisées par les applications de sauvegarde et de recherche),
- les données techniques comme la classe d'architecture, les standards utilisés pour le codage des contenus et toutes les informations qui permettent à un système de connaître le document sans avoir à le parcourir entièrement.

1.5.3 SGML vs ODA

On oppose souvent SGML à ODA par le type de document que ces normes permettent de manipuler. SGML est destiné à la modélisation de documents longs et complexes alors qu'ODA est plus particulièrement adapté à la représentation des documents commerciaux ou de travail. On pourrait schématiser cette différence en disant que le premier type de document est conçu par des imprimeurs alors que le second type peut être produit par toutes les entreprises.

Sémantique implicite contre sémantique explicite

SGML fournit une syntaxe pour baliser les documents, mais n'associe aux éléments qu'elle permet d'identifier ni sens ni traitement. Cette situation est à la fois un avantage et un inconvénient. Un avantage, parce qu'il est ainsi possible de définir de nouvelles applications sur SGML en associant des traitements spécifiques aux éléments logiques définis dans la DTD. Un inconvénient, parce que des individus ne peuvent traiter de manière uniforme un document que s'ils échangent également la DTD et les traitements associés.

Avec SGML, les échanges ne peuvent pas se faire en aveugle, contrairement à ODA où les informations concernant le comportement des objets sont transportées avec chaque document.

Représentation objet contre approche orientée grammaire

Dans SGML, la définition de la structure logique s'effectue grâce à des DTD facilement modifiables. Dans ODA, la notion de classe de document existe [99], mais elle est restreinte par l'utilisation de profils de documents. Les profils Q111, Q112 et Q113 définissent des classes de documents relativement générales, mais aucune possibilité n'est laissée pour les étendre. Le profil Q114, actuellement en cours d'élaboration, permettra aux utilisateurs de définir leurs propres classes de documents.

ODA est basé sur une représentation objet [98, 111] alors que SGML répond à une approche orientée grammaire.

Représentation logique contre représentation globale

La norme ODA modélise conjointement l'aspect logique et physique des documents. Ceci permet d'imposer le respect de normes de présentation, notamment lors des opérations de transmission. Un document ODA aura le même aspect sur tous les sites. L'inconvénient de cette situation est qu'il est assez délicat de changer l'aspect physique d'un document. Cela impose en particulier de changer de classe de document.

SGML ne considère que l'aspect logique des documents mais il peut être couplé à d'autres systèmes traitant les aspects présentation et transformation (DSSSL par exemple).

Texte balisé contre séparation structure/contenu

Un document SGML est représenté par un texte balisé qui ne contient que des caractères éditables. Ceci permet de le visualiser et de le modifier sur des outils standards. L'inconvénient de ce type de représentation est qu'il mélange structure et contenu et qu'il impose de parcourir séquentiellement la structure pour détecter les objets logiques.

Dans ODA, la structure est représentée indépendamment des contenus, l'aspect édition de la structure et édition des contenus sont donc clairement différenciés. L'inconvénient est que la manipulation de documents ODA nécessite des outils dédiés. Le format ODIF utilisé pour l'échange est un format binaire relativement encombrant [62].

Synthèse de la comparaison

SGML est un système à la fois souple (il permet aux utilisateurs de définir leurs propres DTD) et rigoureux (les DTD garantissent le respect des règles de structuration des documents) qui s'intègre facilement aux environnements de travail. Du fait de son approche orientée grammaire, les documents représentés en SGML sont facilement exploitables.

ODA permet de traiter des documents de manière uniforme. Il assure l'intégrité des documents échangés, mais le développement d'applications ODA est complexe et coûteux. Une comparaison des deux normes, sous forme de tableau, est donnée dans la figure 1.26 [89] :

Caractéristiques	SGML	ODA
Universalité/Interchangeabilité	+ +	+ + +
Richesse de la structure	+ + +	+
Adaptabilité aux applications	+ + +	+
Manipulabilité	+ + +	-
Support de contenus graphiques	*	OUI
Support de contenus images	*	OUI
Support de contenus tableaux	*	NON
Support d'autres contenus	*	NON
Formatage - Mise en page	* DSSSL	+
Format d'échange réseau	* SDIF	OUI
Existence de produits	+ +	-

* non défini par la norme mais possible

Figure 1.26 : Tableau comparatif SGML/ODA.

L'affirmation de Vincent Quint [120] selon laquelle le domaine de la manipulation des documents avait dix années de retard sur celui des langages de programmation se trouve vérifiée ici si l'on regarde l'état actuel des normes de représentation des documents. Malgré l'utilisation de concepts empruntés aux langages orientés objets, les deux normes les plus abouties à l'heure actuelle, SGML et ODA, comportent encore de nombreuses insuffisances.

Comme le note Bertrand Meyer, un programme est un type de document particulier [92]. Pourtant, d'importants concepts issus de la programmation structurée, comme la modularité ou la réutilisabilité, ne sont pas pris en compte par les normes.

Ainsi, des langages à objets, les auteurs de ces normes n'ont pas retenu la notion d'encapsulation, pourtant l'un des concepts de base de la programmation par objets. D'après Vincent Quint [118], c'est parce que ces normes ne considèrent les documents que d'un point de vue statique : elles décrivent la représentation des documents mais ne s'intéressent pas à leur aspect dynamique.

Pourtant, cet aspect dynamique est de plus en plus considéré comme prédominant. Une phrase de Paul M. English prononcée lors de la conférence EP'90 [42] illustre bien cette évolution dans la façon de considérer les documents : « *a theme has emerged that the document pieces might be described not by what they* are, *but rather by what they* do. »

1.6 Conclusion

Pour notre projet, nous adhérons entièrement à la modélisation conventionnelle des documents structurés basée sur les trois concepts suivants :

- représentation structurée de données abstraites,
- compatibilité avec un modèle,
- séparation des aspects physiques et logiques.

Notre problème se trouve centré sur la représentation de l'aspect logique des documents. Dans BIBLE, le concept de WYSIWYG n'est pas traité car notre but est de permettre une conception de document dirigée par les données. Dans tous les cas, les auteurs travailleront sur des vues des documents qui seront ensuite modifiées par les données. Une visualisation fidèle du document papier n'est dans ce cas d'aucune utilité. Nous nous contenterons de proposer des vues agréables sur les documents pour faciliter le travail des auteurs.

Nous privilégions une modélisation abstraite des données qui autorise la représentation des relations purement arborescentes, mais aussi des relations "transversales" entre certains éléments.

La représentation interne des données que nous utiliserons dans BIBLE sera conçue selon une approche orientée objet qui nous permettra, en particulier, d'utiliser le concept d'héritage. Nous ne nous attarderons pas sur les problèmes de conflits qui ne manqueront pas de se poser car c'est un problème difficile qui est encore ouvert en génie logiciel.

Nous appliquerons la théorie des prototypes pour représenter les relations qui existent entre documents mais surtout entre modèles de documents. Nous

proposerons une solution permettant de remédier aux problèmes relatifs à la délégation entre objets très structurés.

Nous adoptons SGML comme format pivot de base pour BIBLE. Ce formalisme permet grâce a son balisage déclaratif, une correspondance facile avec une structure de données arborescente. Choisir une norme comme SGML permet d'intégrer dans le projet des maillons logiciels existants pour réaliser par exemple des formatages, des impressions ou des éditions. Autant de fonctions qui ne font pas partie de la plus value apportée par le projet.

2
La manipulation des documents

> ODA and DSSSL has in common the desire to describe all the typographical conventions of the world [...]. There is a major problem with the typographical conventions they try to describe: There is an incredible amount of intelligence in the processing of a document for production of printed pages. This should come as no surprise to people who do it manually. Not only have we had 500 years to practice on inconsistencies that wind up as aesthetically pleasing but remain elusive to explain. (For instance, you cannot simply compute kerning values; a "correct" value may "look wrong".) We are inventing new things (that may or may not wind up as aesthetically pleasing) with the new technology available to us that put very high demands on the level of abstraction of the specification. It is sort of as in non-Euclidian geometry where your axioms are free to be redefined.
>
> Erik Naggum.

La modélisation de la structure des documents, dont les différents aspects ont été abordés dans le précédent chapitre, est une composante importante du domaine des documents structurés. Cependant, les principales difficultés proviennent non pas de la représentation, mais de la manipulation de ces structures de données. Ce chapitre présente les problèmes liés à la prise en compte de la structure et les différentes techniques utilisées pour les résoudre. Nous utilisons cet état de l'art pour poser les fondements du projet BIBLE.

2.1 Les problèmes induits par la structure

Les premiers problèmes qui viennent à l'esprit sont ceux qui sont directement liés au processus d'édition. La puissance apportée par les éditeurs structurels est en effet accompagnée de nouvelles contraintes liées aux opérations de restructuration dynamique [2, 3, 4, 100].

2.1.1 Les transformations dynamiques

Les restructurations dynamiques consistent à répercuter les opérations d'édition effectuées par un utilisateur dans la structure abstraite sous-jacente. Cette opération n'est pas triviale. La réalisation d'une simple opération comme le "couper/coller[1]" pose déjà, comme nous allons le voir, de gros problèmes. Elle requiert en effet des traitements importants destinés à assurer l'intégrité de la structure abstraite lorsqu'une partie de cette structure est déplacée.

Imaginons, par exemple, qu'un élément *Auteur*, décomposé en *Nom* et *Prénom*, figure en tête d'un document et que l'utilisateur veuille copier cet élément à l'intérieur d'une énumération située dans une autre partie du même document. Dans un traitement de texte non structuré, cette opération ne pose aucun problème. Le texte représentant le nom et le prénon de l'auteur est simplement copié dans une zone tampon, puis collé à l'endroit approprié (figure 2.1).

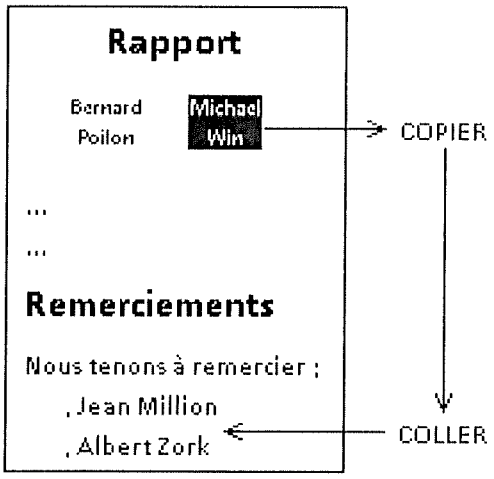


Figure 2.1 : L'opération de copier/coller.

Dans un éditeur structuré, cette simple opération nécessite d'effectuer une transformation sur la structure de l'élément copié. Dans notre exemple, la structure de données *Auteur* composée des sous-éléments *Nom* et *Prénom*, doit être transformée en une structure *Item* avant de pouvoir être insérée dans l'élément *Enumération* (figure 2.2).

1. L'opération de couper/coller consiste à extraire un bloc d'information d'un document source pour l'intégrer, soit dans un document cible, soit à un autre emplacement dans le même document. L'opération de copier/coller est similaire mais ne provoque pas la destruction du bloc d'information source.

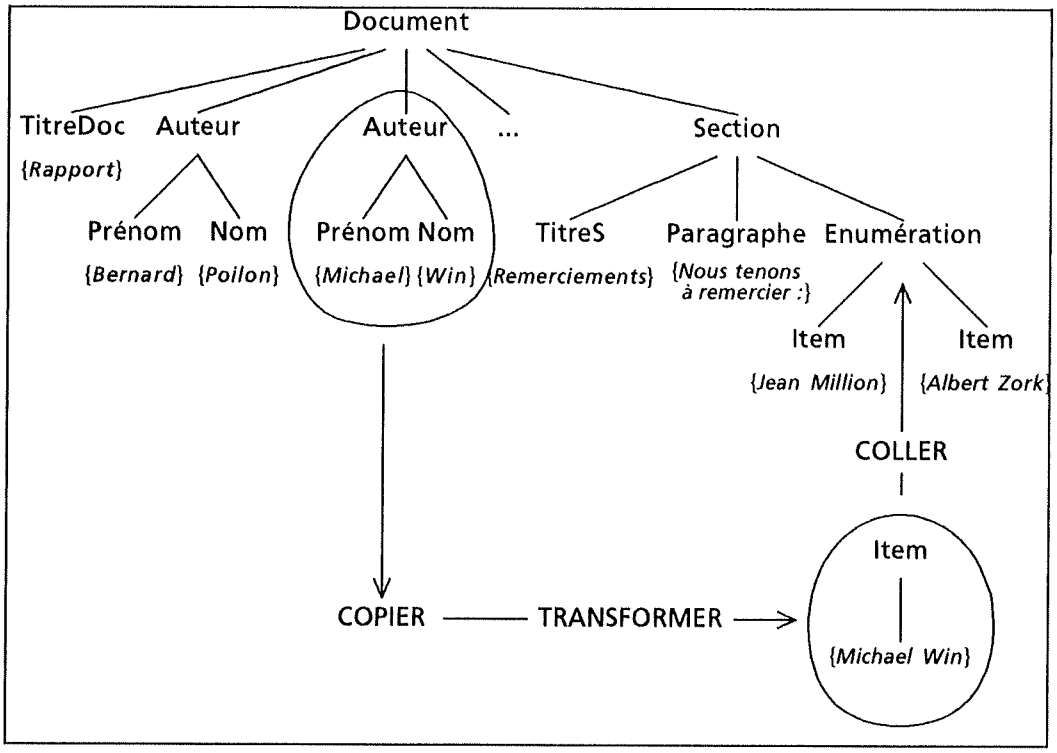


Figure 2.2 : Les transformations nécessaires à une opération de copier/coller.

Un éditeur structurel doit également permettre aux utilisateurs d'agir sur la structure sous-jacente des documents en proposant de nouvelles fonctionnalités d'édition. La transformation de type est l'une de ces nouvelles fonctionnalités. Elle consiste à modifier le type d'un élément composant un document, par exemple, transformer une section en sous-section, une liste en une suite de paragraphes, etc.

Pour la transformation de type, on a une structure donnée et on veut la transformer en une structure cible. Pour l'opération de couper/coller, on a une structure et on veut la greffer à un endroit précis à l'intérieur d'une autre structure. Cette dernière opération se décompose en :

- l'identification des différentes structures susceptibles d'être placées à l'endroit cible désigné,
- l'évaluation des différences entre les différentes structures potentielles et la structure à déplacer,
- le choix d'une structure cible qui va constituer la structure d'accueil de l'élément à copier,
- une transformation de type entre la structure de départ et la structure cible.

On remarque que la transformation de type est incluse dans l'opération de couper/coller. Elle en constitue cependant le problème central. La résolution de la

transformation de type implique la résolution de l'opération de couper/coller. En effet, si on sait transformer une structure en une autre, on sait évaluer le coût de cette transformation et on est capable de juger de l'éloignement entre deux structures en prenant en compte ce coût. Choisir la structure cible la plus proche de la structure à copier ne pose donc plus aucun problème. Quant à l'identification des structures possibles à un endroit donné, il suffit de regarder les possibilités prévues dans le modèle de document. Le problème qui doit retenir toute l'attention est bien celui la transformation de type.

L'exemple que nous avons choisi est volontairement simple, mais dans beaucoup de cas, la transformation de type pose des problèmes plus complexes. Transformer par exemple un élément *Auteur* illustré dans la figure 2.2 pour le rendre compatible avec les structures *Personne* ou *Author* des figures 2.3 ou 2.4 est un problème très complexe dont on ne peut pas garantir a priori une exécution correcte du point de vue de l'utilisateur.

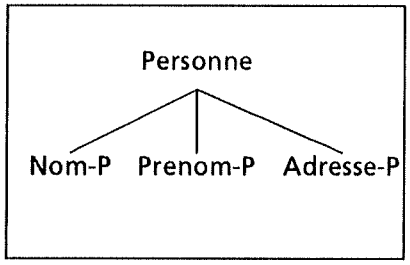


Figure 2.3 : Structuration d'un élément *Personne.*

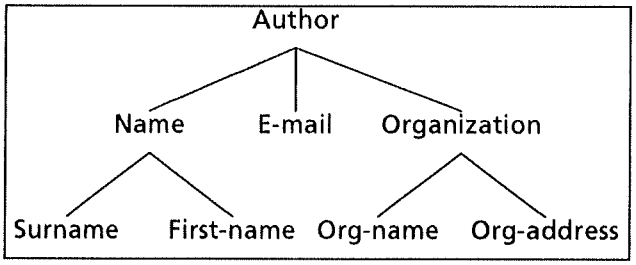


Figure 2.4 : Structuration d'un élément *Author.*

Dans sa thèse consacrée à la transformation de type, Extase Akpotsui note à plusieurs reprises l'importance que revêtent les intentions de l'utilisateur [4]. Il note en particulier que :

> « *La compréhension des intentions de l'utilisateur à travers les requêtes de transformation de structure qu'il exprime semble indispensable à l'élaboration d'une politique globale de recherche de solutions.* »

Malgré cet écueil à la transformation automatique, il est possible de définir des règles élémentaires de transformation (ajout d'un élément, suppression, modification, réordonnancement, etc). Ces règles servent, d'une part, à pouvoir décider si deux structures sont compatibles (c'est-à-dire s'il est possible de passer de l'une à l'autre en un nombre fini de transformations élémentaires), et d'autre part, à effectuer la transformation en appliquant une suite d'opérations élémentaires.

Ces problèmes de transformation de type sont d'autant plus importants dans l'édition des documents que l'éditeur ne travaille pas sur la même modélisation

des documents que l'utilisateur. Quand un utilisateur effectue une opération comme le déplacement d'un paragraphe, il vérifie que le paragraphe déplacé est sémantiquement compatible avec le reste du texte. Pour l'éditeur structurel, cette même opération nécessite d'effectuer des contrôles de cohérence de la structure ainsi obtenue (l'élément paragraphe peut il être inséré à cet endroit ? Dans la négative, est-il possible de le transformer pour le rendre compatible avec le modèle logique ? etc).

La tâche assignée à un éditeur structurel est donc très complexe puisqu'elle nécessite de traduire en modifications structurelles, les transformations sémantiques effectuées par un utilisateur.

2.1.2 Les transformations statiques

Certains systèmes de manipulation de documents structurés comme Mentor-report [90], Andra [54] ou Lara [55] n'utilisent qu'un seul modèle logique. La gestion de l'ensemble des documents produits est donc grandement simplifiée. Par contre les systèmes qui permettent d'utiliser plusieurs modèles comme pedtn [47], Speed [32] ou Grif [116] génèrent de nouvelles difficultés. En effet, dès lors qu'un document structuré n'existe que par son modèle, il est nécessaire de toujours maintenir une correspondance et une cohérence entre les deux. Dans les systèmes actuels, il est extrêmement difficile de changer le modèle d'un document déjà créé [140] et quasiment impossible de modifier un modèle sans perdre les documents qui lui sont associés.

Le problème est voisin de celui déjà évoqué, relatif au déplacement de structures dans un document. Dans un cas, on a une structure et on veut la greffer sur une autre structure en faisant en sorte que la nouvelle structure obtenue soit compatible avec la grammaire de la structure d'accueil. Dans l'autre cas, on modifie une grammaire et on veut modifier les structures qui lui sont rattachées de manière à les rendre compatibles avec la nouvelle version.

2.1.3 L'élaboration des représentations graphiques

Le calcul des images graphiques associées aux structures abstraites qui représentent les documents est un problème qui peut également être très difficile. Dans sa version la plus simple, l'image graphique est obtenue en associant des changements graphiques aux éléments composant une structure abstraite. La structure est parcourue suivant un algorithme déterministe (par exemple en profondeur d'abord) et la manière de représenter les contenus est mémorisé dans des règles (figure 2.5).

Le problème se complique si l'on ne se contente pas d'un formatage occasionnel du document, mais que l'on souhaite un mapping dynamique entre la structure abstraite et l'image graphique ; ceci pour disposer, par exemple, de

fonctionnalités WYSIWYG. Dans les cas les plus complexes, l'image du document est modélisée par une structure qui ne peut être obtenue par un simple parcours de la structure logique (la figure 1.22 constitue une bonne illustration de ce dernier cas).

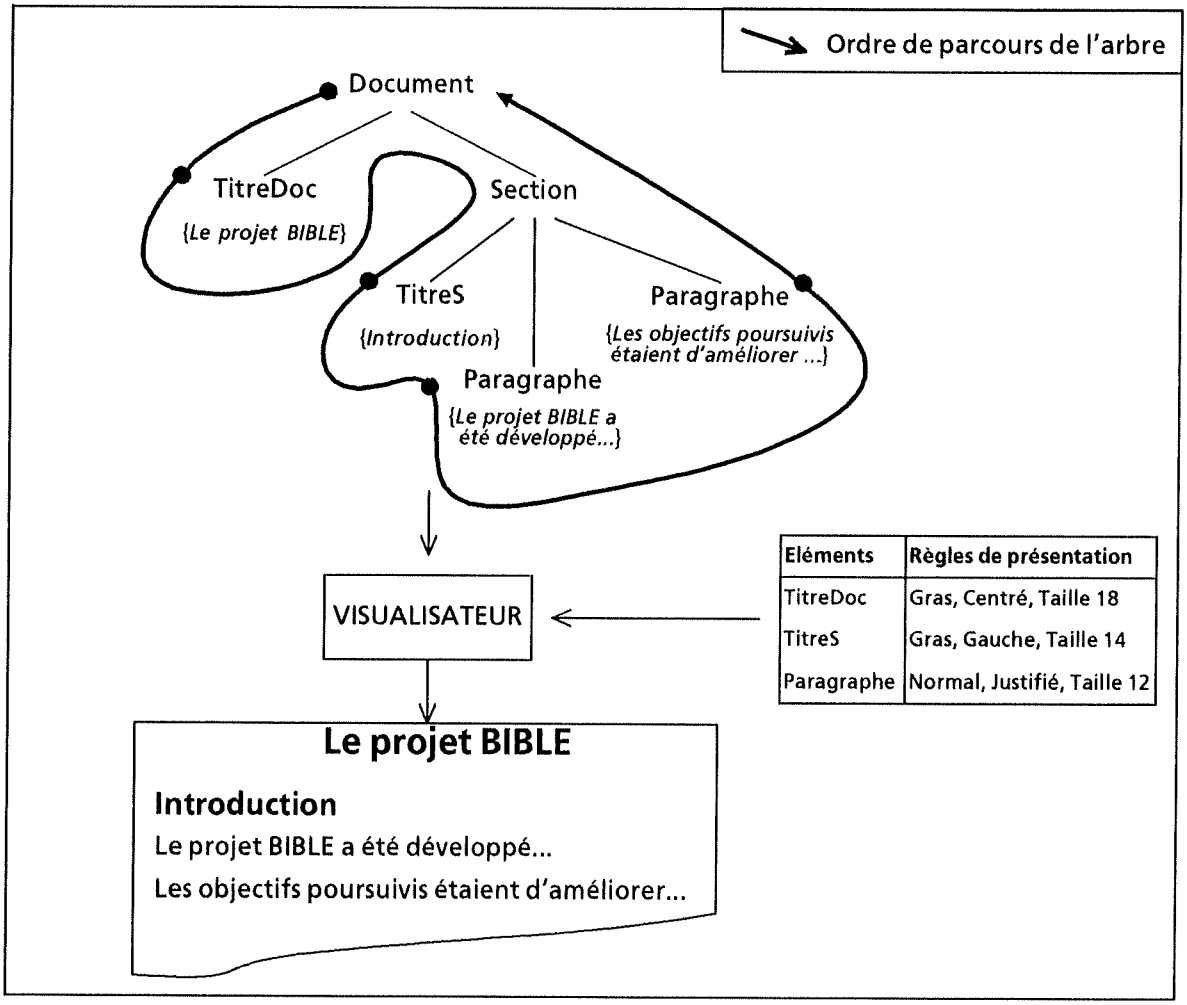


Figure 2.5 : Visualisation par application de règles sur la structure abstraite

2.1.4 Le problème central des transformations de structures

La transformation de structure constitue le problème central de la manipulation des documents. C'est un problème ouvert qui est très étudié actuellement [4, 11, 12, 44, 49, 78]. Ce processus intervient dans chacun des trois problèmes qui viennent d'être relevé.

La transformation de structure lors d'une restructuration dynamique

Dans une restructuration dynamique, une transformation de structure est nécessaire pour transformer, lors d'un couper/coller, une sous-structure quelconque en une autre structure propre à être insérée à un endroit précis d'un document. L'autre opération de restructuration dynamique qu'est la transformation de type nécessite également un transformation de structure.

La transformation de structure lors d'une transformation statique

Les deux opérations de transformation statiques (la transformation d'un document suite à une modification de son modèle et le transfert d'un document vers un autre modèle) nécessitent également des transformations de structures.

La transformation de structure lors d'une visualisation

Typol et PPML que nous décrivons dans la section suivante sont deux composant du générateur d'environnement de programmation CENTAUR [18]. Typol est utilisé pour spécifier les propriétés sémantiques d'un programme, PPML sert à décrire le processus de compilation et un troisième langage Métal permet de spécifier les syntaxes abstraites et concrètes d'un langage.

Cette séparation entre les différents composants d'un langage est très intéressante et permet de séparer l'opération de génération d'une structure graphique en deux phases :

- une phase de transformation de la structure logique en une structure compatible avec un modèle physique,
- une phase de calcul de l'image graphique du document à partir de sa structure physique.

2.2 Les techniques de décompilation

Par analogie avec les langages de programmation, l'opération consistant à associer une image graphique à un structure abstraite est appelée décompilation.

2.2.1 Visualisation textuelle

La méthode la plus simple pour effectuer la visualisation d'une structure abstraite est d'appliquer à chaque élément qui la compose une même opération de formatage définie statiquement [121]. Dans le cas d'un programme, la visualisation est obtenue en ajoutant une syntaxe concrète sur la syntaxe abstraite.

Soit par exemple l'instruction suivante représentant une alternative :

```
Alternative      → Condition, Instruction1, Instruction2
```

On peut définir deux représentations concrètes de cette instruction. L'une respectant la syntaxe CLOS, et l'autre, la syntaxe C.

```
Concrete-CLOS_Alternative  (c:Condition, i1:Instruction1, i2:Instruction2) →
    "(IF " c i1 i2 ")"

Concrete-C_Alternative  (c:Condition, i1:Instruction1, i2:Instruction2) →
    "if " c " then " i1 ";"
    "    else" i2 ";"
```

Appliquée au cas des documents, cette technique permet par exemple d'afficher sur un support le contenu de chaque élément de base et d'indenter la représentation chaque fois que l'on descend d'un niveau dans la hiérarchie.

2.2.2 Décompilation libre de contexte

La manière de procéder qui est décrite précédemment ne permet d'obtenir que des représentations textuelles très simples. Pour pouvoir générer des vues concrètes en deux dimensions, il faut procéder d'une autre manière. L'utilisation du concept de boîtes, introduit par Donald E. Knuth [76, 77] est une manière commode d'exprimer la disposition d'objets sur une surface plane. Une vue sur une structure abstraite est ainsi matérialisée par une arborescence de boîtes.

Chaque type d'objet logique est associé à des règles de décompilation qui indiquent, grâce à des opérateurs, la représentation concrète de chacun de ses fils.

Un exemple de ce type de systèmes est donnés par PPML (Pretty-Printing Meta-Language) [95]. Une spécification PPML est un ensemble de règles de la forme :

```
schéma --> format
```

où *schéma* est un arbre de syntaxe abstraite contenant des variables, et *format*, une spécification de formatage dans un langage de boîtes.

La syntaxe de PPML est très simple, elle comprend uniquement 4 opérateurs :

- **h**, qui définit un alignement horizontal,
- **v**, qui définit un alignement vertical,
- **hv**, qui définit un alignement horizontal, mais qui génère un saut à la ligne chaque fois qu'une boîte ne peut pas tenir sur une ligne,
- **hov**, qui définit un alignement horizontal pouvant se transformer en alignement vertical si tous les éléments ne tiennent pas sur la même ligne.

On dispose ainsi d'une puissance d'expression qui permet de décrire des opérations de formatage plus sophistiquées. Voici par exemple la règle qui permet d'afficher l'élément figure composé d'une image, d'un numéro de figure et d'une légende :

```
figure(*image, *noFigure, *legende)
--> [v *image
     [h "Figure No " *noFigure ":" *legende] ];
```

Figure 2.6 : Exemple d'une règle de décompilation PPML.

Cette règle signifie que la représentation physique d'une figure est constituée de deux boîtes placées verticalement. La première contient le sous-élément image et la seconde, une liste de 4 boîtes disposées horizontalement dont le contenu est respectivement : "Figure No", *noFigure*, ":" et *legende* (*noFigure* et *legende* sont deux sous objets de figure).

2.2.3 Décompilation contextuelle déterministe

Nous avons présenté PPML comme un système de décompilation libre de contexte. Cela n'est pas tout à fait vrai. PPML dispose en effet d'un mécanisme pour effectuer des choix entre plusieurs visualisation possibles. La règle *hov* indique en effet un alignement horizontal qui peut se transformer en un alignement vertical si la place disponible n'est pas suffisante. Ce mécanisme étant le seul traitement contextuel intégré dans PPML, il ne justifie pas la classification de PPML dans les systèmes de décompilation contextuelles.

Nous intégrons dans cette dernière catégorie les systèmes qui permettent des traitements contextuels multiples. Le système GIGAS (Graphical Interfaces Generation by Attribute Specifications) [46, 144], par exemple, permet véritablement de calculer de manière contextuelle et de plus incrémentale l'aspect physique des documents. GIGAS regroupe les idées de PPML, pour l'agencement des boîtes graphiques, et du Cornell Synthesizer Generator [133] pour le traitement de grammaires attribuées permettant d'obtenir l'incrémentalité. GIGAS assure également la coordination entre structure logique et structure physique ce qui rend le système très approprié à la génération d'éditeurs graphiques structurels.

2.2.4 Décompilation contextuelle non déterministe

Dans les systèmes décrits précédemment, la structure physique s'obtient à partir de la structure logique. De ce fait, la structure abstraite se projette dans l'arbre des boîtes, ce qui signifie qu'une boîte est toujours associée à un noeud de l'arborescence logique et qu'une relation hiérarchique entre deux boîtes correspond toujours à une relation hiérarchique entre deux noeuds abstraits [26].

Ces limitations ne sont pas très pénalisantes car souvent les représentations physiques sont assez proches des représentations logiques. Mais il existe certains cas qui ne peuvent être correctement résolus avec les systèmes présentés (par

exemple l'affichage de documents multilingues présenté dans la figure 1.22). Dans ces cas, la structuration physique du document correspond aussi à un modèle.

Le langage Typol, une implémentation de la sémantique naturelle [72], a été défini dans le cadre du système CENTAUR [18] pour exprimer des propriétés sémantiques dans un environnement de programmation. En Typol, la définition des contraintes est entrée sous la forme d'un ensemble non ordonné d'axiomes ou de règles d'inférence [38]. Ces règles sont directement exécutables par traduction en clauses Prolog ce qui permet de disposer d'un mécanisme de rétro parcours [144]. Le principe général est de trouver un but à partir d'un certain nombre d'hypothèses. Typol est très adapté à la description de vérificateurs, de traducteurs d'interprètes et même de formateurs [10].

Une règle Typol est de la forme :

$$\frac{H1 \vdash T1 : S1 \quad \ldots \quad Hi \vdash Ti : Si \quad \ldots \quad Hn \vdash Tn : Sn}{H \vdash T : S}$$

Dans cette formulation, chacune des expressions $Hi \vdash Ti : Si$ constitue une prémisse, et $H \vdash T : S$, la conclusion de la règle. Les termes T sont des expressions d'une syntaxe abstraite, les termes H représentent les attributs hérités, et les termes S, les attributs synthétisés.

Par exemple, la spécification de la sémantique dynamique d'une instruction *IF* est donnée par les deux règles Typol suivantes :

$$\frac{s \vdash EXP : true \qquad s \vdash STM1 : s1}{s \vdash \text{if EXP then STM1 else STM2} : s1}$$

$$\frac{s \vdash EXP : false \qquad s \vdash STM2 : s1}{s \vdash \text{if EXP then STM1 else STM2} : s1}$$

La conclusion des règles se lit ainsi : soit un état courant *s*, on doit évaluer l'arbre de syntaxe abstraite défini par *if EXP then STM1 else STM2* et le résultat de cette évaluation est donné par le nouvel état *s1*.

Regardons maintenant la première règle. La première prémisse dit que l'évaluation de *EXP* retourne *TRUE*. Si cette prémisse est respectée, on passe à la seconde, sinon la règle échoue. La seconde prémisse dit que *s1*, l'état de sortie de la règle prend pour valeur l'évaluation de *STM1*. Cette première règle est donc la spécification de l'expression *if* quand *EXP* retourne *TRUE*. La seconde règle provoque le traitement de la partie *else* de la règle.

Ce formalisme permet d'exprimer des règles de mise en page complexes. Prenons par exemple le cas de l'affichage d'une figure composée d'une image et d'une légende. On exprime grâce à des règles Typol que l'image composant la figure doit être affichée sur la même page que la légende (figure 2.7).

$$\frac{s \vdash \text{IMAGE} : s1 \qquad s1 \vdash \text{LEGENDE} : s2}{s \vdash \text{FIGURE} = \text{IMAGE LEGENDE} : s2} \quad (1)$$

$$\frac{s \vdash \text{nouvelle-page} : s1 \qquad s1 \vdash \text{IMAGE} : s2 \qquad s2 \vdash \text{LEGENDE} : s3}{s \vdash \text{FIGURE} = \text{IMAGE LEGENDE} : s3} \quad (2)$$

$$\frac{\begin{array}{c} s \vdash \text{TiensDansPage?(BITMAP)} : \text{true} \\ s \vdash \text{AjouteBloc(BITMAP)} : s1 \end{array}}{s \vdash \text{IMAGE} = \text{BITMAP} : s1} \quad (3)$$

$$\frac{\begin{array}{c} s \vdash \text{TiensDansPage?(TEXT)} : \text{true} \\ s \vdash \text{AjouteBloc(TEXT)} : s1 \end{array}}{s \vdash \text{LEGENDE} = \text{TEXT} : s1} \quad (4)$$

Figure 2.7 : Les règles Typol permettant de disposer une image sur une page.

La première règle spécifie la séquence d'affichage d'un objet *FIGURE* (composé d'une image et d'une légende) : ainsi, on commence par afficher l'image, puis on affiche la légende.

L'affichage de l'image est décrit dans la règle (3). On commence par calculer si l'image tient dans la page et dans l'affirmative, on ajoute un bloc image. Si cette règle échoue, elle fait échouer la règle (1), et la règle (2), qui impose de commencer une nouvelle page avant l'affichage de la figure, est essayée.

Si la règle (3) n'échoue pas, la règle (4) est activée pour l'affichage de la légende. Si cette règle échoue, alors la règle (3) est annulée grâce au rétro-parcours et la règle (2) est déclenchée.

Le gros problème de cette solution est l'absence d'incrémentalité due à l'utilisation du mécanisme de rétro-parcours.

2.3 Les transformations de structures

2.3.1 Utilisation d'une sémantique translationnelle

Une définition formelle des relations qui existent entre deux langages est définie par une sémantique translationnelle qui est ajoutée à la grammaire de syntaxe abstraite du langage source. Comme pour la sémantique dénotationnelle présentée dans la section 1.3.1, la sémantique translationnelle associe à chaque construction T du langage source, une équivalence dans le langage cible qui est notée T_T [92].

Cette technique peut être utilisée dans le cas des documents. Soient par exemple deux modèles dans lesquels la définition auteur est donnée, dans le modèle source par :

Auteur-s → Nom-s, Coord-s
Nom-s → TEXTE
Coord-s → Rue-s, Ville-s
Rue-s → TEXTE
Ville-s → TEXTE

et dans le modèle cible par :

Auteur-c → Nom-c, Prenom-c, Ville-c
Nom-c → TEXTE
Prenom-c → TEXTE
Ville-c → TEXTE

Les deux éléments diffèrent par trois points :

- la suppression de l'élément *Rue*,
- l'ajout de l'élément *Prénom* (qui sera initialisé par défaut à NULL),
- la montée d'un niveau de l'élément *Ville*.

Une sémantique translationnelle permettant de transformer une structure *Auteur* compatible avec le modèle source pour la rendre compatible avec le modèle cible est définie par :

$T_{Auteur\text{-}s}$ [n: Nom-s, c: Coord-s] ::= n, NULL, T_c

$T_{Coord\text{-}s}$ [r: Rue-s, v: Ville-s] ::= r, v

On constate que des transformations complexes peuvent être exprimées de manière simple. Cependant, le gros défaut de cette méthode est qu'elle nécessite un développement « à la main » et au cas par cas. Ceci est inadapté dans le cas des documents où les modèles sont nombreux et susceptibles d'être définis par les utilisateurs.

2.3.2 Les transformations en SGML grâce à DSSSL

DSSSL (Document Style Semantics and Specification Language) [67] est une norme qui offre les ressources nécessaires à l'expression de sémantiques de traitement comme le formatage ou la transformation de structure. DSSSL opte pour une décomposition du processus de visualisation en deux phases :

- une phase de transformation de structure qui permet de passer d'une structure abstraite compatible avec un modèle logique en une autre structure compatible avec un modèle physique. Cette phase permet d'effectuer des transformations telles que groupements, suppressions, ajouts ou réarrangements sur les structures de documents.
- une phase de formatage dans laquelle sont prises en compte les caractéristiques géométriques des éléments destinés à être affichés.

DSSSL contient en outre un langage d'interrogation qui permet de naviguer dans la structure arborescente des documents. Les requêtes peuvent porter sur des types d'éléments, des attributs, mais aussi sur les relations qui existent entre plusieurs éléments.

Si l'on reprend l'exemple de notre document bilingue (voir figure 1.22), la phase de formatage consiste à passer de la structure physique du document à la vision papier. Cette opération consiste à résoudre des contraintes d'agencement d'éléments rectangulaires dans une page. La structuration des documents n'est pas modifiée.

La phase de transformation est quant à elle beaucoup plus générique. Elle n'intervient pas uniquement dans le formatage du document, puisqu'elle consiste en la transformation d'une structure abstraite en une autre. La phase de transformation de DSSSL peut donc tout à fait être utilisée pour faire passer un document d'un modèle à un autre, c'est-à-dire pour réaliser les transformations statiques.

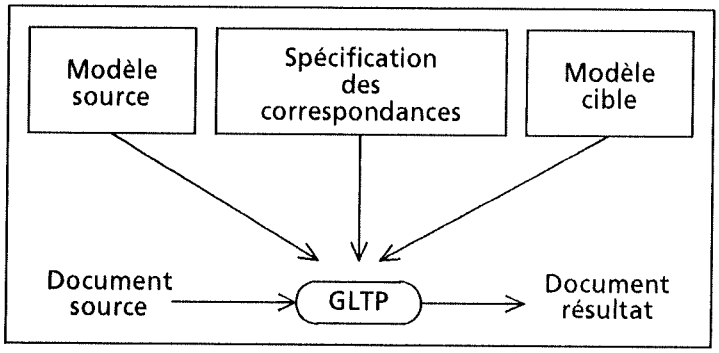


Figure 2.8 : La phase de transformation de DSSSL.

Dans la norme, cette phase est appelée GLTP (General Language Transformation Process). Son but est de transformer un document source pour le

rendre compatible avec un modèle donné. Pour ce faire, le GLTP a besoin, en plus du document source, de trois types de ressources :

- le modèle duquel dépend le document source,
- le modèle destiné à accueillir le document résultat,
- une liste de correspondance (appelée *Association Specification*) entre, d'une part les éléments utilisés dans le modèle source, et d'autre part ceux utilisés dans le modèle cible (figure 2.8).

L'inconvénient majeur de cette architecture est le même que celui lié à l'utilisation d'une sémantique translationnelle : des développements doivent être effectués au cas par cas.

2.3.3 Automatisation du processus de transformation

La solution au problème qui vient d'être évoqué consiste à comparer le modèle source et le modèle cible pour en déduire les transformations à effectuer sur les instances.

Dans le système Grif [116, 117], la phase de comparaison donne lieu à l'établissement de règles de transformation. Ces règles sont ensuite appliquées une à une sur l'instance correspondant au modèle source ce qui permet d'obtenir une nouvelle instance de document compatible avec le modèle donné comme cible (figure 2.9) [3, 4].

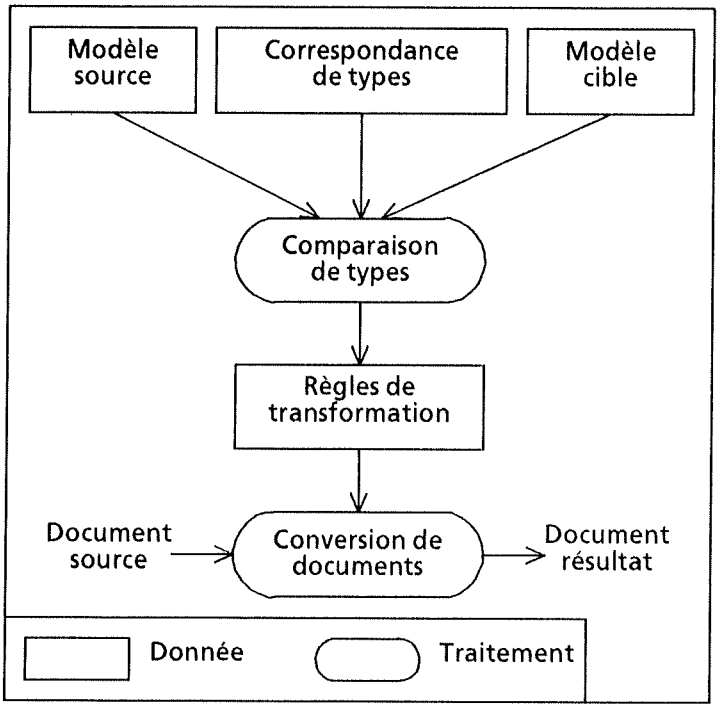


Figure 2.9 : La transformation statique dans Grif [4].

Dans SYNDOC, la comparaison entre une grammaire source et une grammaire cible permet l'établissement d'une grammaire translationnelle. Cette grammaire permet la génération d'un compilateur qui est utilisé pour convertir tous les documents écrits avec la grammaire source (figure 2.10) [78].

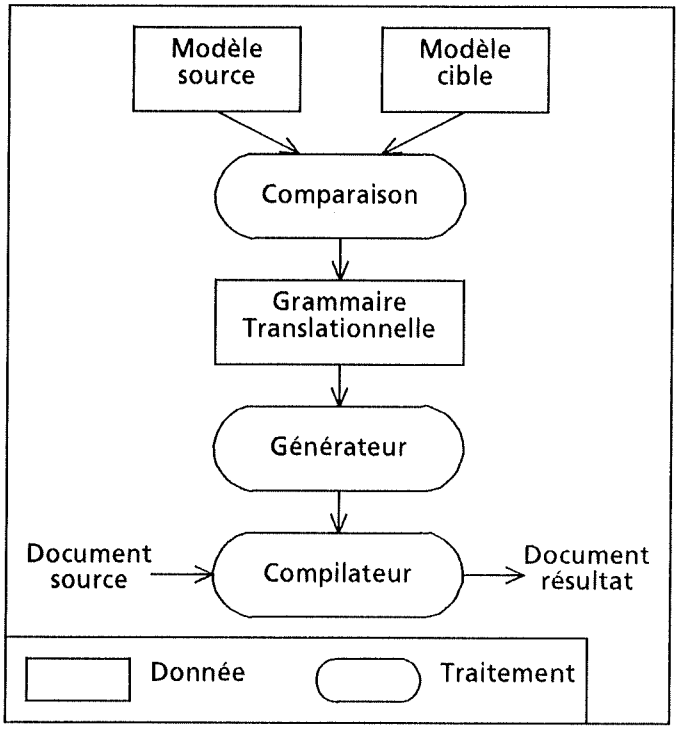


Figure 2.10 : La transformation statique dans SYNDOC [78].

Cette architecture permet potentiellement d'automatiser l'établissement des règles de transformation. Aujourd'hui, l'automatisation complète n'est possible que pour des cas simples. Sans autres informations que la connaissance des grammaires de départ et d'arrivée, il semble en effet utopique d'espérer automatiser complètement la migration de documents d'un modèle à un autre ; par exemple transformer un document écrit avec la DTD CALS 20081 en un document compatible avec la DTD d'IBM IBMIDDoc. Même en génie logiciel, ou l'on travaille depuis longtemps sur ce problème, on est bien loin d'une solution générale [12].

Le point faible des solutions qui viennent d'être présentées est que l'on ne peut pas, à partir de deux états d'un même objet, déduire la suite des informations qui permettent de passer de l'un à l'autre. L'approche adoptée par R. Furuta et D. Scotts permet de remédier à ce problème [49]. Elle consiste à décrire les modifications effectuées sur les modèles de documents à l'aide d'un langage de transformation simple. La modification des documents peut alors être effectuée en répercutant sur les instances les transformations apportées aux modèles.

L'inconvénient de cette solution vient de l'écriture des commandes de transformation qui peut vite se révéler longue et fastidieuse. Cette technique ne permet en outre de ne prendre en compte que les évolutions d'un modèle et pas la migration entre deux modèles distincts.

2.3.4 La prise en considération du contexte

Dans leur article présenté à la conférence EP'92, A. Brown et al. annoncent deux principes de base de la transformation statique de documents [22] :

- l'abstraction de traitement, qui permet de séparer la représentation source de la manière dont elle est traitée,
- l'abstraction structurelle, qui permet de spécifier un traitement pour une classe de documents, et pas uniquement pour un document isolé.

Dans leur modélisation d'un processus de transformation qui respecte ces deux principes de base, un document source est représenté par une structuration abstraite de données compatible avec une grammaire source. Un traitement effectué sur la structure de données source produit une structure résultat dont les propriétés structurelles sont données par une grammaire résultat (figure 2.11).

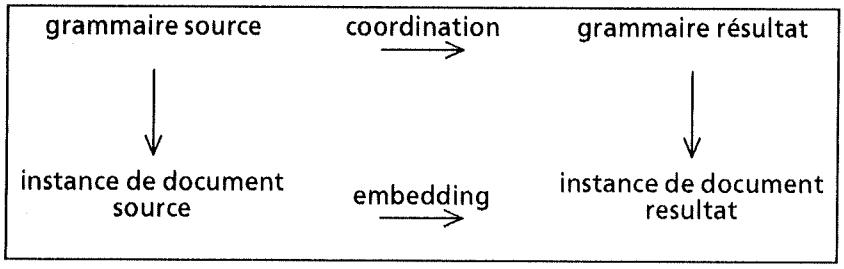


Figure 2.11 : La phase de transformation définie par A. Brown et al.

L'utilisation unique des deux principes d'abstraction présentés plus haut peut conduire à la définition d'un système insensibles aux particularités de chaque document. Or, à la différence d'une grammaire à contexte libre, le traitement d'un document dépend fortement du contexte. Dans tous les modèles de représentation des documents (y compris dans SGML et dans ODA), un document est en effet une structure abstraite munie d'attributs. La manipulation des noeuds de cette structure peut nécessiter des traitements sur les attributs de ce noeud, mais aussi, sur les attributs de noeuds proches.

Le travail effectué par A. L. Brown et al. consiste justement à définir un traitement générique qui prenne en compte les particularités de chacune des instances de documents. Les auteurs introduisent la notion de traitement de document dépendant du contexte :

> « *Our notion of context-dependant document processing allows the specification of generic document processing in such a way as to be sensitive to individual differences of source instances.* »

En SGML, on dispose de la possibilité de spécifier la valeur des attributs d'un élément lors de la phase de traitement. Ce processus permet d'introduire une variabilité du document liée à la manière dont il est manipulé. Mais ceci ne constitue pas un véritable traitement contextuel des documents. Ce dernier peut être schématisé de la façon suivante : un même traitement appliqué à deux éléments identiques peut générer deux résultats différents. Cette variabilité est produite par le contexte dans lequel se trouve l'élément, c'est-à-dire, la position occupée par l'élément dans la structure.

Pour qu'une telle variabilité soit possible, il faut associer les attributs non pas à des éléments, mais à des règles de production. Ainsi, il devient possible de déclarer des attributs dont la valeur dépende de la règle utilisée. Ce qui revient à tenir compte des objets englobants.

Un élément paragraphe peut, par exemple, être défini de façon différente suivant s'il a été produit par la règle :

Section → Paragraphe

ou par la règle :

Note → Paragraphe

La valeur des attributs peut également être calculée en fonction de valeurs héritées ou synthétisées ; comme dans les grammaires attribuées.

Ce principe de traitement contextuel des documents est très intéressant et apporte une puissance supplémentaire aux opérations de transformation statiques. Ceci permet en particulier, d'éliminer des ambiguïtés en limitant l'espace des solutions admissibles, grâce à des clauses associées aux règles de production. Dans le cas d'un passage d'une structure logique à une structure physique, il est par exemple possible de tenir compte du nombre maximum de lignes par pages, du nombre de caractères par lignes, etc.

2.4 L'adéquation des documents aux données

Dans le projet BIBLE nous devons faire appel à des traitements contextuels. Le terme contextuel est utilisé ici dans un sens différent de celui qui a été exposé dans la section précédente.

Dans la section 2.3.4, un contexte est constitué de l'ensemble d'un document traité. Cela permet d'effectuer sur des éléments identiques des traitements qui varient selon l'emplacement ou le voisinage de ces éléments. Dans BIBLE, nous

utilisons ce type de traitement contextuel, mais nous étendons la notion de contexte à des données qui peuvent être extérieures au document. Nous considérons qu'un document est une vue sur un ensemble de données. Des changements dans ces données de base influent donc soit sur la structure abstraite représentant le document, soit sur son apparence physique. Par ce biais, nous rejoignons un concept récent qui est celui de document actif.

2.4.1 Les documents actifs

Le terme de document actif est très récent puisqu'il a été prononcé pour la première fois par Brian Reid lors de la conférence EP-88. Le principe de base est qu'un document est appréhendé non pas en fonction de ce qu'il *est*, mais en fonction de ce qu'il *fait*. A l'heure actuelle, aucun consensus n'existe sur une définition précise des documents actifs. Nous retiendrons pour notre part la définition qu'en ont donné Paul M. English et al. lors de la conférence EP'90 [42] :

> « *[Active documents are ...] structured documents in which the objects in the documents can be acted upon by, and can themselves act upon, other objects in the document or the outside world.* »

Avant son évocation à EP-88, des développements avaient pourtant déjà été effectués sur ce concept. Dès 1981, cette notion était appliquée dans Emacs, le célèbre éditeur [129, 141]. Puis, les années qui suivirent virent de nombreuses études s'effectuer. George Towner [134] décrit des documents qui supportent une auto-mise à jour à partir d'une base de données externe. Donald Chamberlin et al. [29] définissent un mécanisme grâce auquel les éléments d'un document utilisent des modules de formatage externes (dont la nature dépend du type d'élément) pour être présentés. Dans le système *Camino-Real* [9], les formules mathématiques contenues dans un document sont envoyées à un module de manipulation symbolique qui se charge d'effectuer le formatage. Paul English et al. [42] rapportent une utilisation conjointe d'Interleaf avec le logiciel de CAO AutoCad et le tableur Lotus 1-2-3. Ian Macleod et al. [84] proposent une extension de SGML pour y attacher des applications. R. Furuta et P. David Stotts [50] remarquent l'intérêt d'inclure dans les documents hypertextuels des spécifications sur la manière dont le document est lu. Ils nomment cela une ***browsing semantics***.

Une intéressante généralisation du concept de document actif a été effectuée dans la version 19 de GnuEmacs qui considère un document comme un flux d'objets lisp qui peuvent représenter des caractères textuels, des commandes de formatage, des événements comme les actions souris, ou toute autre fonction représentable en Lisp [136].

Interleaf est actuellement l'éditeur le plus connu qui utilise le concept de documents actifs. Dans le système Interleaf, il est possible de greffer des

procédures lisp à n'importe quel niveau d'un document. Ces procédures sont invoquées soit automatiquement soit en réponse à certaines actions effectuées par les utilisateurs. Il est ainsi possible d'effectuer des contrôles de saisie, d'aller récupérer dans une base de données des informations relatives à un paramètre entré, de faire figurer sur le document des boutons qui activent des applications, ou d'afficher des indicateurs sur des événements quelconques. L'éditeur de documents Grif suit également cette évolution puisque sa nouvelle version dénommée GATE (Grif Application Toolkit Environment) permet de manipuler les documents actifs.

Dans un article de 1992 [75], Takamune Kitazawa et Mitsutoshi Hayata décrivent un système qu'ils qualifient eux-mêmes de génération intelligente de documents. Leur application contient un module qui permet à un document d'effectuer des requêtes à des systèmes experts et d'intégrer les réponses, mais également une procédure qui maintient la cohérence du document en effectuant des contrôles sur les informations recueillies.

De plus en plus de fonctionnalités jusqu'alors du domaine de l'éditeur sont prises en compte dans les documents. Dans la conférence Usenix de 1991, Mattew Hodges et Russel Sasnett ont présenté un projet dont le but est de mettre fin à la distinction entre l'éditeur (le programme) et le document (les données) [61]. Cela a amené la notion d'éditeurs plastiques (Plastic Editor) dans lesquels, l'auteur d'un document crée non seulement des informations, mais indique également la manière de les manipuler. Automatiquement reconfigurables, les documents actifs permettent même de simplifier le travail laborieux de personnalisation des documents et des outils qui les manipulent [15].

En poursuivant un peu plus loin cette idée, on peut sans peine imaginer un système informatique où il n'existerait plus qu'un seul type d'objet : le document. Ces documents contiendraient toutes les procédures nécessaires à leur propre manipulation. Ils seraient le reflet de certains aspects du monde extérieur et serviraient d'interface pour déclencher des actions ou des requêtes [16]. Cette vision permet d'englober la plupart des outils actuellement disponibles. On peut très bien avoir, par exemple, un document qui s'appelle 'état des fichiers sur disque' dans lequel figure une liste des fichiers existants. Cette liste est automatiquement mise-à-jour dès qu'une opération est effectuée sur l'un des fichiers du disque. Réciproquement, toute modification effectuée sur le document est immédiatement répercutée sur les fichiers physiques.

Modélisation des documents actifs en SGML

L'élaboration automatique de documents adaptés à un contexte nécessite soit d'intégrer aux documents des traitements, soit de faire appel à des constructions spéciales permettant de modéliser leurs comportements. Dans tout

les cas, cela nécessite de dépasser la simple description syntaxique des documents pour aller vers des représentations sémantiques.

SGML qui est une norme passive [84] est consacrée presque exclusivement à la modélisation de l'aspect syntaxique des documents [104, 137]. La possibilité d'inclure des données sémantiques est bien évoquée dans la norme mais aucune normalisation n'est proposée en ce qui concerne l'interface avec les éléments extérieurs. Des appels à des commandes externes peuvent être incorporés à un document SGML en les balisant avec <? et >. On peut par exemple grâce à l'instruction <?|*newpage*> spécifier un saut de page avec une syntaxe TEX. On peut également imaginer d'incorporer des ordres destinés à un système d'exploitation, par exemple <?*rm fichier*> ou <?*ls*>.

Représenter des traitements à l'intérieur d'un document est donc possible sans sortir du cadre de la norme, mais rien n'est dit quant à leur exécution. Celle-ci peut être déclenchée par un parser, par un formateur ou même au niveau d'un éditeur. Dans tous les cas, ces traitements seront dépendants de l'application. Si l'on veut que ces commandes puissent tester ou agir sur des parties du document, il est indispensable de coupler l'exécution des traitements à une opération de parsing de manière à pouvoir disposer de l'arbre d'analyse.

Un exemple : l'architecture proposée par Ian A. Macleod et al.

Ian A. Macleod et al. [84] proposent une architecture composée d'APIs (Application Programmer Interface) capables de communiquer avec un parser et avec l'arbre d'analyse courant (figure 2.12). Cette organisation comprend un parser, l'arbre d'analyse qu'il utilise, des APIs contrôlées par un driver qui sont capables de communiquer avec le parser et avec l'arbre d'analyse, une application qui contient le corps des procédures utilisées dans le document analysé et enfin une table d'actions où sont spécifiées les correspondances entre les éléments d'une DTD et les traitements à effectuer.

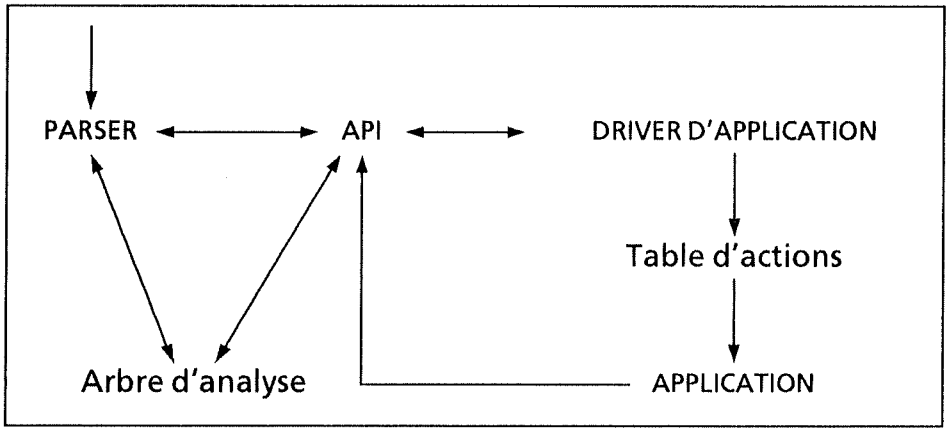


Figure 2.12 : Exemple d'une interface applicative à SGML [84].

La syntaxe utilisée pour décrire la table d'actions est volontairement conçue selon un look SGML. Avec le mot clé *ELEMENT*, des actions sont associées aux tags utilisés dans une DTD. Ces actions peuvent être conditionnées grâce à des prédicats qui retournent soit *TRUE* soit *FALSE*. Il est ainsi possible de tester le contenu d'un attribut du document ou de faire des tests sur l'état courant du parser. Dans la figure 2.13 présentant un extrait d'une table d'actions, les prédicats des deux premières expressions testent l'attribut status associé à un article. En fonction du contenu de l'attribut status (final ou draft), une API permettant de changer le style d'impression est invoquée avec le paramètre 1 ou 2. Le prédicat de la troisième expression est un test contextuel effectué sur l'élément analysé. Il retourne *TRUE* si l'on est positionné sur le premier paragraphe d'une section. Si tel est le cas, alors une commande destinée à changer l'indentation du paragraphe est activée.

```
<!DATTYPE     exemple    article [
<!ELEMENT     article    (status=final)     SetStyle (1) >
<!ELEMENT     article    (status=draft)     SetStyle (0) >
<!ELEMENT     parag      section.parag/1    SetIndent (10) >
...
]>
```

Figure 2.13 : Exemple d'une table d'actions [84]

L'organisation qui vient d'être décrite est très intéressante car elle est performante et surtout, elle ne requière aucune modification du standard SGML. Ceci constitue un exemple dont nous nous inspirerons pour la conception de l'architecture du système BIBLE. Nous devons également nous inspirer de l'état de l'art en ce qui concerne l'activité consistant à élaborer de toutes pièces une structure de documents à partir de données.

2.4.2 La génération de la structure des documents

Les études menées sur ce sujet visent à élaborer, à la demande, des documents structurés. Elle partent de l'hypothèse selon laquelle toute l'information présentée dans les documents préexiste indépendamment de ceux-ci [27]. Un document est alors une vue sur un ensemble d'informations [101].

Mac Web [101], est un outil pour élaborer la structure des documents. Il utilise un langage d'interrogation de base de données qui permet d'obtenir un document structuré répondant à une grammaire. Une requête a la forme suivante:

```
listePubli :
for each Publi of PubliRécentes (Auteur) do
UnePubli : Publi;
```

La notation *A:B* signifie que l'on veut créer une instance de la classe *A* dont le contenu est *B*. *PubliRécentes (Auteur)* permet d'obtenir successivement toutes les publications d'un auteur dans la variable *Publi*. Donc, *listePubli*, instance de la classe du même nom contient une liste de *UnePubli* dont chacune des valeurs est donnée par *PubliRécentes (Auteur)*. On obtient donc un document structuré défini par la grammaire :

<ListePubli> --> <UnePubli>*
<UnePubli> --> TEXTE

François Chahuneau adopte la même démarche, mais s'appuie sur la norme SGML [27]. Une requête sur une base de données produit donc un document structuré sous la forme d'un fichier SGML, et sa grammaire correspondante (DTD).

Les documents produits sont plutôt des documents de travail, résultat d'une requête à une base de données. Ils sont très synthétiques puisqu'ils ne contiennent que des informations à l'état brut. A la différence des lettres ou les documents commerciaux qui nécessitent un minimum de présentation et de mise en forme des informations. De plus, chaque document produit à partir d'une requête correspond à un modèle qui lui est propre. Dans notre application, nous voulons que les documents automatiquement générés correspondent à une grammaire connue à l'avance et susceptible d'être partagée avec d'autres documents.

2.4.3 L'utilisation de parties variables dans SGML

Dans SGML, où chaque document possède une DTD, il est également possible de modéliser des contenus variables.

```
<!ENTITY % confiden "INCLUDE">
<!ENTITY % public "IGNORE">
...
<![ %confiden; [We are happy to announce that this year
the managers will receive a 30% pay-rise. ]]
<![ %public; [Unfortunately the depressed economic
situation forces us to limit this year's pay rise to 2%. ]]
```

Figure 2.14 : un document SGML avec des parties variables [59].

La norme permet en effet d'étiqueter certaines parties du texte pour les traiter de manière spécifique. On peut par exemple indiquer que des parties d'un document ne doivent pas être prises en compte par le parser avec le mot clé *IGNORE* ou au contraire qu'elles doivent être traitées avec le mot clé *INCLUDE*. Grâce à la commande *ENTITY*, il est possible de donner un nom symbolique à des parties d'un document, et de les valoriser, ensuite, avec *IGNORE* ou *INCLUDE*.

Dans l'exemple de la figure 2.14 [59], les parties du document sont étiquetées avec *confiden* ou *public*. Suivant la valorisation de ces deux étiquettes, on obtient un document destiné soit à la masse des salariés, soit aux managers.

Cette même technique peut être utilisée pour conditionner des parties structurées comme par exemple un chapitre particulier. Cependant, nous notons deux inconvénients. En premier lieu, l'utilisation des étiquettes ne permet pas d'exprimer des traitements complexes comme la répétition ou le choix entre plusieurs alternatives. De plus, cette manière de procéder ne permet pas de garantir la conformité d'un document produit avec sa DTD. Rien n'interdit en effet de conditionner un élément déclaré obligatoire dans la DTD en spécifiant que l'une des alternatives est vide.

Nous pensons qu'il est tout à fait possible de définir une extension de SGML pour prendre en compte l'utilisation de variables externes et le conditionnement de parties d'un document. La norme SGML est suffisamment ouverte pour permettre cela. De nombreuses extensions, permettant d'étendre la puissance du langage ou de normaliser certaines possibilités existantes sont à l'étude [22].

2.5 Les fondements du projet BIBLE

2.5.1 La génération vue comme une transformation statique

L'objectif principal de BIBLE est de générer automatiquement des documents adaptés aux lecteurs [108]. Nous nous distinguons de la génération de documents présentée dans la section 2.4.2 [27, 101] car le but du système BIBLE n'est pas d'élaborer une vue sur un ensemble de données, mais d'utiliser les données disponibles pour générer des documents pertinents. De plus, les documents produits par notre système doivent être compatibles avec des modèles de documents existants. BIBLE ne doit pas générer une grammaire associée aux documents, mais assurer la compatibilité des documents produits avec une grammaire donnée (figure 2.15).

La génération de documents dans BIBLE ne s'effectue pas suite à des requêtes libres (sinon il serait bien difficile de faire en sorte que les résultats respectent une grammaire). Les données utilisées sont préalablement structurées en fonction des modèles. Dans BIBLE, nous répartissons les données en deux groupes [107] :

- le premier groupe est constitué de données générales. Celles-ci sont stockées dans un système d'information et existent indépendamment des applications. Ce sont des informations factuelles qui servent à orienter la personnalisation des documents.
- le second groupe de données, qui est lié à l'application BIBLE, regroupe toutes les autres informations nécessaires à l'élaboration des documents. Lors de la phase

de génération, ces données sont modelées en fonction des informations provenant du premier groupe.

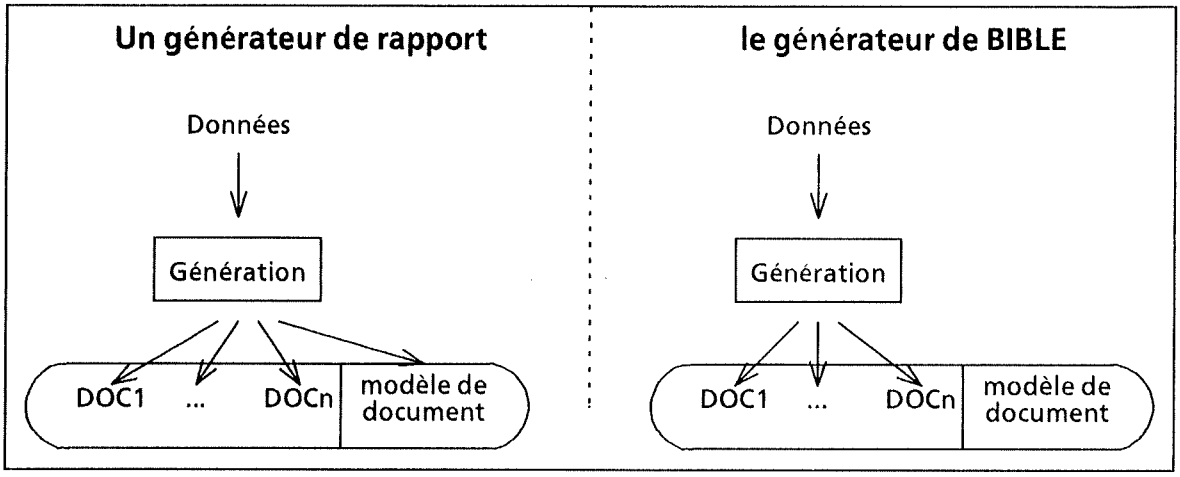


Figure 2.15 : Comparaison entre BIBLE et un générateur de rapport.

Le processus de génération de BIBLE peut être rapproché d'une opération d'édition : les documents sont créés par des apports successifs de données et la conformité de la structure obtenue avec un modèle est contrôlée. L'unique différence avec une opération d'édition classique est que les modifications effectuées sur les documents proviennent non pas d'un opérateur humain mais d'un flux de données. Cette solution est coûteuse et difficile à mettre en œuvre car elle nécessite d'effectuer des transformations dynamiques sur un grand nombre de documents.

Nous préférons considérer le processus de génération comme une opération de transformation statique d'une structure de données en une autre représentant le document généré. Notre processus de génération est à rapprocher de l'opération de conversion de document utilisée dans Grif (figure 2.9) sauf que dans BIBLE, les règles de transformation sont remplacées par des données. La manière d'utiliser ces données dans la phase de génération est incluse dans la structure de départ.

Des gabarits pour modéliser les structures variables

Nous introduisons le concept de gabarit pour représenter la structure à configurations multiples utilisée par le processus de génération [109]. Un gabarit peut être vu comme une structure abstraite de données dans laquelle certaines des parties peuvent être choisies parmi un ensemble de structures candidates. Notre vision se rapproche de la représentation conjointe de plusieurs documents en utilisant le principe des liens typés (figure 1.20).

Dans un gabarit les liens ne sont pas typés statiquement, mais des aiguillages, placés à certains endroits de la structure permettent d'orienter

dynamiquement le parcours en fonction de variables externes. La figure 2.16 illustre l'utilisation d'un tel aiguillage : la composition de l'élément *A* est conditionné par la valeur de la variable externe *var*.

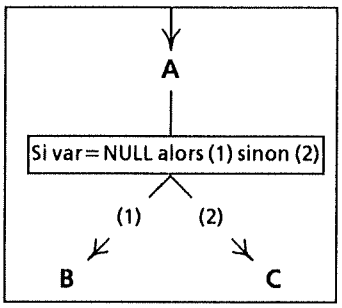


Figure 2.16 : Représentation d'une composition variable.

L'opération de génération peut se définir comme un processus permettant d'orienter les choix dans un gabarit. L'utilisation d'un modèle pour vérifier la validité des documents obtenus n'est pas nécessaire si l'on est capable d'assurer que, quelques soient les aiguillages empruntés, le document obtenu est structurellement correct.

Nous utilisons également les gabarits pour spécifier des choix en fonction d'informations sur la structure du document lui même. Les conditions portent dans ce cas sur les attributs des éléments ou la position des éléments dans la structure. Ceci permet d'effectuer des traitements contextuels à la manière du système de Ian A. Macleod [84].

Gabariage + génération : une alternative à l'édition

Le gabarit constitue une structure intermédiaire entre le modèle et le document. Il est créé sur un éditeur structurel grâce à une opération que nous nommons gabariage pour la distinguer de l'édition classique. Chaque gabarit regroupe les caractéristiques de plusieurs documents qui sont générés grâce à l'apport de données externes. Si le gabarit ne contient que des données fixes, alors, il ne se différencie pas d'un document et l'opération qui lui a donné naissance est une édition classique (figure 2.17).

D'autres structures de contrôle, comme la répétition ou l'option, doivent pouvoir être utilisées dans les gabarits. Nous détaillons cela, dans la seconde partie de ce mémoire consacrée au système.

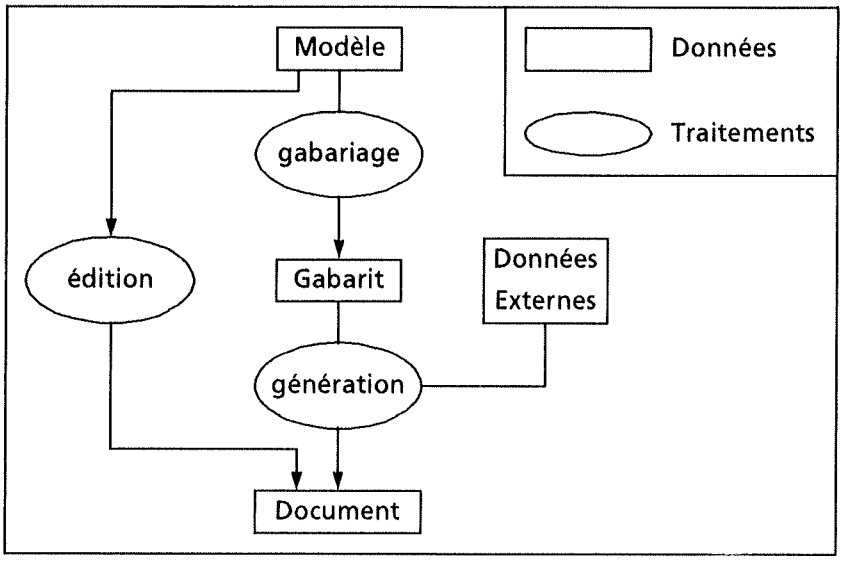


Figure 2.17 : Le parallèle entre édition classique et gabariage + génération.

2.5.2 La représentation des modèles par des prototypes

Nous avons vu qu'il n'existe pas à l'heure actuelle, de solution générale permettant d'assurer la conversion automatique de documents [12]. A notre sens, cela est du au fait que tous les systèmes travaillent sur des représentations statiques des modèles. Il est en effet extrêmement difficile de déduire, à partir de deux structures différentes, la suite d'opérations qui permet de passer de l'une à l'autre.

Dans BIBLE, nous optons pour une représentation dynamique dans laquelle chaque modèle de document est défini par un ensemble de transformations à appliquer à un modèle de référence. Nous utilisons en ce sens l'approche de R. Furuta et D. Scotts [49].

Nous utilisons une représentation basée sur le prototypage pour mémoriser l'état des différents modèles utilisés. Chaque modèle de document constitue un prototype qui est créé par copie différentielle d'un autre modèle. Les transformations à effectuer sur un modèle ne sont pas écrites manuellement mais sont générées à partir des opérations effectuées par un utilisateur sur un éditeur de modèle.

Nous utilisons la puissance des prototypes pour élaborer une base qui prenne en compte de manière dynamique les modifications apportées aux modèles. Si, par exemple, le modèle M_1 est défini par une suite de transformations à appliquer au modèle M_0, il est nécessaire que toute les modifications postérieurement effectuées sur M_0 soient répercutées dynamiquement sur M_1.

Seconde partie

Le système BIBLE

3
Description conceptuelle du système BIBLE

> L'information présentée dans un document préexiste en général à l'élaboration de celui-ci sous une forme plus ou moins élicitée. Elle peut même intervenir dans plusieurs documents. Elaborer un document revient à définir une application entre la structure sémantique initiale d'un ensemble d'informations et la structure du document final. Un document est alors une vue de cet ensemble d'informations.
>
> Marc Nanard et Jocelyne Nanard [101]

Ce troisième chapitre est consacré à la description conceptuelle des aspects novateurs du système BIBLE [108] : un éclatement en deux phases de l'opération d'édition et une application du concept d'héritage à la représentation d'un ensemble de modèles de documents.

Le système BIBLE a été développé en utilisant l'éditeur SGML Grif. Les exemples que nous mentionnons sont écrits selon la syntaxe SGML, mais les concepts exposés sont suffisamment généraux pour pouvoir être appliqués à n'importe quelle modélisation des documents.

3.1 Conception de documents dirigée par les données

Pour inclure des éléments contextuels dans un langage de programmation, la solution habituelle est d'ajouter des macro-commandes qui ne font pas partie du langage lui-même. Ceci est également possible dans le cas des documents. Nous pouvons inclure dans un document des commandes utilisant des données externes qui seront interprétées par le module de génération. Les figures 3.1 et 3.2 donnent deux exemples possibles de conditionnement de la formule d'appel d'une lettre dans un document SGML. Dans le premier exemple, c'est l'élément *Appel* qui est conditionné alors que dans le second, c'est le contenu qui varie.

```
...
<Corps>
SI SEXE='M'<Appel>Monsieur,</Appel>
SI SEXE='F'<Appel>Madame,</Appel>
<Parag>
...
```

Figure 3.1 : Exemple de conditionnement d'un élément SGML.

```
...
<Corps>
<Appel>
SI SEXE='M' Monsieur,
SI SEXE='F'Madame,
</Appel>
<Parag>
...
```

Figure 3.2 : Exemple de conditionnement d'un contenu.

3.1.1 L'EDTD : extension sémantique d'une DTD

Chacune des deux solutions présentées est possible et correspond à ce qui se fait pour les langages de programmation. Cependant, ce n'est pas de cette manière que nous avons procédé. Une DTD impose en effet des contraintes sur la présence ou l'absence de certains éléments. En ajoutant des constructions pilotées par des données, il se peut très bien que le document généré ne corresponde plus à la DTD originelle, ce qui enlève tout l'intérêt d'utiliser des modèles.

Quelles que soient les données utilisées, tout document issu de la phase de génération doit être compatible avec la DTD initiale. Le modèle utilisé pour représenter les documents variables doit donc contenir toutes les contraintes structurelles de la DTD de départ.

Pour le projet, nous modélisons les gabarits grâce à des extensions sémantiques des DTD des documents. Ces extensions, que nous nommons EDTD (pour Extended DTD), permettront d'inclure des structures de contrôle à l'intérieur des documents. D'un point de vue syntaxique, il n'y a cependant aucune différence entre des DTD SGML standard et les EDTD que nous utilisons. Ceci nous permet d'utiliser n'importe quel éditeur SGML supportant plusieurs DTD pour élaborer les gabarits.

Soit par exemple l'élément B qui est déclaré optionnel dans un élément A :

```
<!ELEMENT A        --(B?)>                    (1)
```

En transformant cette expression en :

```
<!ELEMENT A          --(cond?, B?)>          (2)
<!ELEMENT cond       --CDATA>
```

l'auteur peut créer ou pas un élément *B* dans l'élément *A* comme avec (1), mais il peut aussi conditionner la présence de l'élément. L'élément Cond que nous avons défini contient soit une expression, soit la référence à une procédure dont l'évaluation retourne une valeur booléenne. Si la valeur retournée est *TRUE*, alors, l'élément qui suit apparaît sur le document, sinon, il n'apparaît pas. *Cond* est déclaré en CDATA, ce qui signifie qu'il peut contenir n'importe quel caractère alphanumérique.

Comme expression booléenne, nous pouvons utiliser soit le test d'un attribut, soit un test contextuel comme l'ont défini Macleod et al. [84]. Nous reviendrons de manière plus détaillée sur la définition des conditions et des variables dans la section 3.1.3.

Avec (1), on a une alternative pour la composition de l'élément *A*. Soit il contient l'élément *B* (3), soit il ne le contient pas (4).

```
<A><B>contenu de B</B>...</A>          (3)
<A></A>                                (4)
```

Avec (2), on a une autre alternative qui est le conditionnement de l'élément *B*. Dans l'extrait de document (5), l'élément *B* est conditionné par la fonction *proc1*.

```
<A><cond>proc1</cond><B>contenu de B</B></A>          (5)
```

La DTD (2) offre la possibilité de spécifier une condition sans mentionner l'élément auquel elle s'applique (6). Ce cas se traduit par une non-apparition de l'élément *B*, quel que soit le résultat de l'évaluation de *Cond*.

```
<A><Cond>proc1</Cond></A>          (6)
```

L'utilisation d'une EDTD permet d'éditer les gabarits sur un éditeur syntaxique standard.

3.1.2 Transformation d'une DTD en EDTD

L'EDTD que nous définissons est une extension d'une DTD initiale qui ne doit pas remettre en cause la manière dont sont structurés les documents. Nous nous imposons les règles suivantes (figure 3.3) :

- tout document SGML compatible avec une DTD doit également être compatible avec l'EDTD,
- il doit être possible d'éditer un document avec une EDTD sans utiliser les apports de celle-ci,
- après évaluation du gabarit, le document produit doit être compatible avec la DTD initiale.

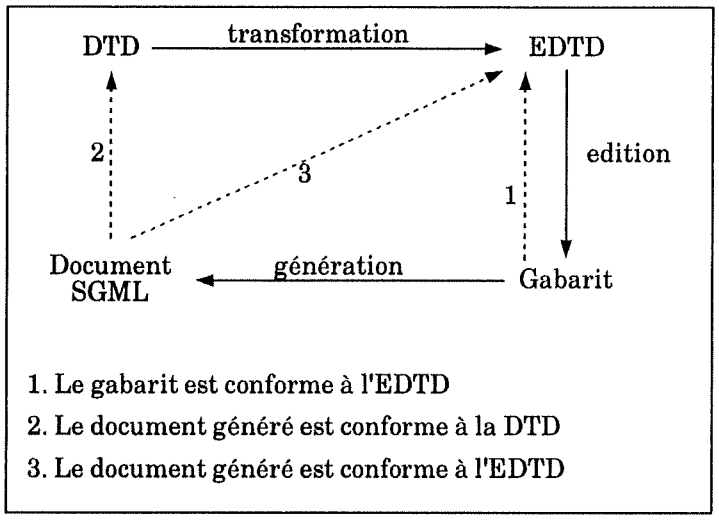


Figure 3.3 : DTD, EDTD et Gabarit.

L'opération consistant à transformer une DTD en EDTD varie suivant les opérateurs utilisés dans la DTD initiale. Nous allons passer en revue les différentes possibilités.

L'inclusion de parties variables

L'inclusion de parties variables est très simple en SGML puisqu'il suffit de baliser par un marqueur le début et la fin de la variable. Ces parties variables peuvent être utilisées dans tous les éléments contenant du texte et ceci, un nombre indéterminé de fois. Ceci peut être exprimé en transformant tous les éléments textuels de la manière suivante :

La prise en compte de parties variables dans l'élément A défini par :

```
<!ELEMENT A          --(#PCDATA)>
```

sera autorisée par :

```
<!ELEMENT A          --((#PCDATA|var)+>
<!ELEMENT var        --CDATA>
```

Ceci permet d'exprimer des combinaisons comme :

```
<A>Monsieur <var>Nom</var>, vous êtes conviés à la visite médicale annuelle le
<var>DateVisite</var> à <var>HeureVisite</var>.</A>
```

Le conditionnement des éléments obligatoires

Un élément obligatoire peut être conditionné en spécifiant un choix entre plusieurs alternatives.

```
<!ELEMENT A          --(B)>
<!ELEMENT B          --(...)>
```

est transformé en :

```
<!ELEMENT    A        --(B)>
<!ELEMENT    B        --(...|B-case)>
<!ELEMENT    B-case   --((cond?, B)+)>
```

L'élément *B-case* est une construction de type case bien connue dans les langages de programmation. Chaque élément de l'alternative est associé à une condition. Le premier élément associé à une condition dont l'évaluation retourne *TRUE* est sélectionné. Si aucune condition n'est vérifiée, c'est par défaut le premier élément spécifié qui est choisi. Si l'on n'indique pas de condition associée à un élément, celle-ci est considérée comme retournant TRUE. Ceci permet de spécifier un élément par défaut à la fin d'une alternative.

La construction que nous utilisons permet d'emboîter les alternatives. Un exemple type est la formule d'appel d'une lettre qui, bien que devant nécessairement figurer dans un document, peut prendre plusieurs valeurs différentes. Dans un gabarit, ceci peut être exprimé par :

```
<Corps>
    <Appel>
        <Appel-case>
            <cond>Masculin?</cond>
            <Appel>Monsieur,</Appel>
            <cond>Feminin?</cond>
            <Appel-case>
                <cond>marié?</cond>
                <Appel>Madame,</Appel>
                <Appel>Mademoiselle,</Appel>
            </Appel-case>
        </Appel-case>
    </Appel>
</Corps>
```

qui correspond à l'extrait d'EDTD suivant :

```
<!ELEMENT Corps          --(Appel, Paragraphe+)>
<!ELEMENT Appel          --(#PCDATA|Appel-case)>
<!ELEMENT Appel-case     --((cond?, Appel)+)>
```

La représentation des éléments optionnels

Pour les éléments optionnels, nous utilisons la même construction que pour les éléments obligatoires car plusieurs choix sont possibles. La seule différence entre les deux constructions est la méthode de résolution. Pour un élément obligatoire, une valeur par défaut sera toujours retournée alors qu'un élément optionnel pourra retourner une composition vide.

La modélisation des répétitions optionnelles

Pour la répétition optionnelle, un élément doit pouvoir être répété un certain nombre de fois en fonction d'une condition. Soit cette condition est une fonction

booléenne qui est appelée à chaque itération, soit elle retourne un nombre qui représente le nombre de fois que l'élément doit être répété.

```
<!ELEMENT A          --(B*)>
<!ELEMENT B          --(...)>
```

est transformé en :

```
<!ELEMENT A          --(B*|B-rep)>
<!ELEMENT B          --(...)>
<!ELEMENT B-rep      --(cond, B)>
```

La représentation des répétitions

Nous utilisons une construction de type *jusqu'à-ce-que* alors que la construction précédente était du style *tant-que*. Une occurence de *B* est créée, ensuite, la condition est testée pour déterminer la présence d'autres éléments.

```
<!ELEMENT A          --(B+)>
<!ELEMENT B          --(...)>
```

est transformé en :

```
<!ELEMENT A          --(B+|B-rep)>
<!ELEMENT B          --(...)>
<!ELEMENT B-rep      --(B, cond)>
```

Le conditionnement des choix

Pour permettre à un choix d'être effectué automatiquement, il faut transformer la structure de choix en une séquence dans laquelle chaque alternative est munie d'une condition. Cette nouvelle structure ne remplace pas la structure de choix, elle y est simplement ajoutée.

```
<!ELEMENT A          --(B|C)>
```

est transformé en :

```
<!ELEMENT A          --(B|C|A-choix)>
<!ELEMENT A-choix    --(cond?,B,cond?,C)>
```

Lors de la saisie de l'élément A, l'auteur pourra choisir comme avant l'une des alternatives B, ou C ou conditionner le choix en sélectionnant l'élément A-choix. Dans A-choix, il aura la possibilité d'associer une condition à chacun des sous-éléments. Si aucune condition ne retourne vrai, on sélectionne le premier élément. Si plusieurs conditions retournent vrai, on sélectionne l'élément correspondant à la première condition évaluée.

3.1.3 Expression de la variabilité dans les gabarits

Dans l'EDTD, les variables sont déclarées en CDATA, ce qui permet à l'élément var de contenir n'importe quel caractère alphanumérique. En fait, nous allons pouvoir utiliser deux types de variables différents :

- des variables externes qui sont repérées en faisant précéder le nom de la variable par data.
- des variables qui retournent la valeur d'un attribut figurant dans le document SGML. Dans ce cas, le nom de l'attribut est préfixé par le nom de l'élément dans lequel il se trouve.

Soit par exemple l'extrait d'EDTD suivant :

```
...
<!ELEMENT     Corps      - - (Sect+|Sect-rep)>
<!ATTLIST     Corps
              status     (PUBLIC, PRIVE) PUBLIC>
<!ELEMENT     Sect       - - (Par+|Par-rep)>
<!ATTLIST     Sect
              status     (PUBLIC, PRIVE) PUBLIC>
<!ELEMENT     Par        - - ((#PCDATA|var)+)>
<!ATTLIST     Par
              status     (PUBLIC, PRIVE) PUBLIC>
<!ELEMENT     Sect-rep   - - (Parag|cond)>
<!ELEMENT     Par-rep    - - (#PCDATA|cond)>
<!ELEMENT     cond       - - CDATA>
<!ELEMENT     var        - - CDATA>
...
```

et le fichier de données qui suit :

```
nom           Georges
grade         07
status        CONFIDENTIEL
```

Les conditionnements syntaxiques

Dans les paragraphes du document SGML, on peut faire référence à des données qui figurent dans le fichier externe et qui sont considérées comme des informations syntaxiques :

- *<var>data.nom</var>* fait référence à la variable *nom* qui se trouve dans le fichier de données,
- *<var>data.status</var>* fait référence à la variable *status* qui se trouve dans le fichier de données.

Les conditionnements sémantiques

On peut également accéder à des informations sémantiques en faisant référence à des attributs associés à certains éléments du document :

- *<var>Par.status</var>* fait référence à l'attribut *status* de l'élément *Par* dans lequel on se trouve,
- *<var>Sect.status</var>* fait référence à l'attribut *status* de l'élément *Sect* dans lequel on se trouve,

- *<var>.status</var>* fait référence à l'attribut *status* de l'élément courant.

Les conditionnements contextuels

On peut également effectuer des accès contextuels aux attributs des éléments :

- *<var>Sect.Par/1.status</var>* fait référence à l'attribut *status* du premier paragraphe de la section courante,
- *<var>Sect/1.Par/1.status</var>* fait référence à l'attribut *status* du premier paragraphe de la première section du document. Pour ce dernier cas, on peut également écrire de manière équivalente : *<var>Par/1.status</var>*

L'expression des conditions par des prédicats

En ce qui concerne les conditions, on utilise des prédicats qui retournent soit TRUE soit FALSE. On peut faire des tests sur n'importe quelle variable. Par exemple :

- le prédicat *data.grade > 5* retourne TRUE si la variable *grade* figurant dans les données externes est supérieure à 5,
- le prédicat *Sect.status=PRIVE* retourne TRUE si l'attribut *status* associé à la section courante a pour valeur PRIVE,
- le prédicat *Sect/1.status=PRIVE* retourne TRUE si l'attribut *status* associé à la première section a pour valeur PRIVE.

On peut également effectuer des tests contextuels. Par exemple :

- le prédicat *Par/1* retourne TRUE si l'on se trouve dans le premier paragraphe du document,
- le prédicat Sect.*Par/1* retourne TRUE si l'on se trouve dans le premier paragraphe d'une section.

Pour éviter d'avoir à redéfinir plusieurs fois un même prédicat, nous faisons référence à un prédicat par un nom et nous associons chaque nom avec le prédicat qui lui correspond dans un fichier séparé que nous appelons *table de personnalisation*. Pour définir un prédicat, nous nous autorisons l'emploi du *'et'* logique que nous notons *'&'*. Le *'ou'* logique est exprimé en associant plusieurs prédicats au même nom. Par exemple, la condition *Cond1* définie par :

```
Cond1        data.status=CONFIDENTIEL
Cond1        .status=PRIVE
Cond1        data.grade < 5 & data.grade !=0
```

retourne TRUE si le status figurant dans le fichier des données a pour valeur CONFIDENTIEL ou si l'attribut status de l'élément courant a pour valeur PRIVE ou si le grade de la personne figurant dans le fichier externe est inférieur à 5 tout en étant différent de 0.

Dans la table de personnalisation, il est également possible et même souhaitable d'associer des noms de variables à leur définition.

3.2 L'héritage de la composition dans une hiérarchie de prototypes

V. Quint et I. Vatton établissent un parallèle entre la description d'une structure de document et un programme. Ceci les pousse à appréhender la conception des modèles de documents de manière modulaire [118]. La solution qu'ils ont adoptée est de définir des petites entités autonomes qui sont assemblées pour concevoir des structures plus complexes. Par exemple, la structure *Corps* peut être définie de manière séparée et être utilisée ensuite dans la description des modèles d'article, de chapitre ou de rapport, le modèle de chapitre pouvant à son tour être réutilisé dans un modèle de livre. Ceci fonctionne parfaitement et est appliqué dans l'éditeur Grif.

Cependant, cette vision modulaire a ses limites. Supposons qu'une structure de corps soit définie et qu'elle soit utilisée pour composer les modèles de rapport, de compte-rendu et de lettre. Supposons maintenant que les modèles de chapitre et d'article utilisent un corps qui ne se différencie du corps défini précédemment que par la présence de références bibliographiques dans les paragraphes. On ne peut pas réutiliser tel quel le corps qui a été défini précédemment[1], alors, on doit choisir entre deux solutions :

- soit on redéfinit une structure de corps différente qui sera utilisée pour décrire les modèles chapitre et article,
- soit on modifie la définition précédente de corps et, en partant du principe que « *qui peut le plus peut le moins* », on l'utilise pour définir tous les documents.

Ces deux solutions ont chacune leurs inconvénients. La première nous oblige à mémoriser et à maintenir deux versions différentes de corps qui sont, non seulement très semblables, mais en plus très dépendantes l'une de l'autre. La plupart des modifications susceptibles d'être effectuées sur une version du corps devra en effet "vraisemblablement" être répercutée sur l'autre.

1. Si les différents modèles sont conçus conjointement ou si l'on est capable de prévoir les types d'éléments susceptibles d'être modifiés lors d'une réutilisation future, alors il est possible d'utiliser le concept d'unité introduit dans Grif. Les unités représentent un ensemble d'éléments dons la composition varie en fonction des modèles utilisés. Dans l'exemple qui a été mentionné, il suffit de définir l'élément paragraphe comme étant composé d'unités. Dans les modèles de rapport, de compte-rendu et de lettre, l'unité contient uniquement du texte. Dans les modèles de chapitre et d'article, la même structure de corps est conservée mais les références bibliographiques sont ajoutées à l'unité.

La seconde solution pose également un problème car, s'il est certain que l'on peut définir un modèle très laxiste qui permette de prendre en compte tous les documents, on perd tous les avantages procurés par les modèles.

Nous avons parlé plus haut de répercussion "vraisemblable" des modifications apportées à l'un des deux corps. Il n'est pas certain en effet que toutes les modifications faites sur la structure du corps concernent de la même manière les différents documents qui l'utilisent. Prenons deux modèles qui utilisent une même structure de corps. Nous devons distinguer trois influences possibles des modifications apportées au corps :

- la modification du corps dans un modèle entraîne la modification du corps dans un autre modèle,
- la modification du corps dans l'un ou l'autre des deux modèles laisse inchangé le corps de l'autre modèle,
- la modification du corps dans un modèle entraîne la modification du corps de l'autre modèle, par contre la modification du corps du second modèle laisse inchangé le corps du premier.

Ces trois cas font appel respectivement aux notions de partage, de clonage et de clonage-lié de sous-structures.

3.2.1 Partage de sous-structures

Le partage consiste à ne définir qu'une fois les éléments communs à plusieurs modèles (figure 3.4). Ainsi, toute modification effectuée sur un objet partagé ou sur l'un des objets qui le composent est prise en compte dans tous les modèles qui l'utilisent. La figure 3.5 représente la situation après la spécialisation de *Parag* en un choix entre un dessin ou un texte.

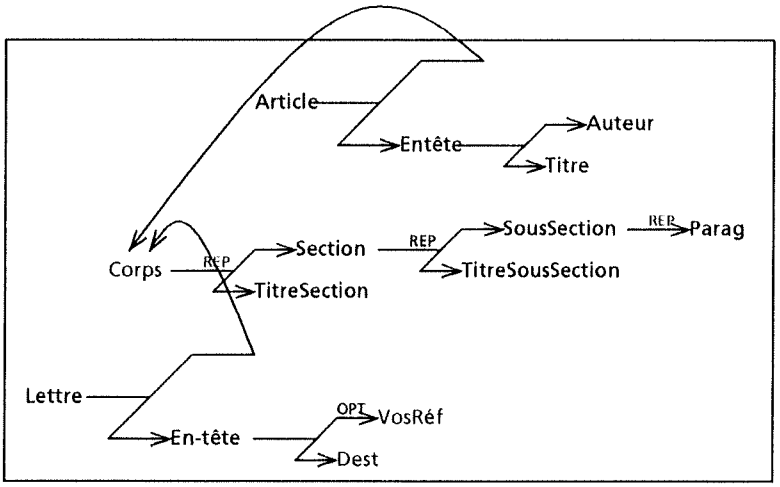


Figure 3.4 : Le partage du corps entre deux modèles.

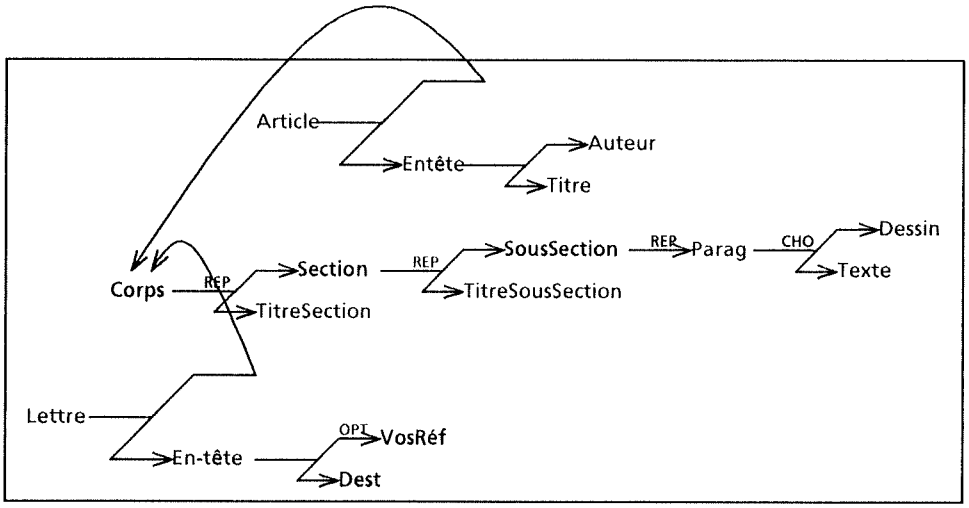


Figure 3.5 : Situation après la modification du paragraphe.

3.2.2 Clonage de sous-structures

Dans ce que nous appelons clonage, les éléments communs à plusieurs modèles sont décrits séparément et peuvent donc évoluer de manière indépendante. Si, pour définir le modèle d'article, on clone le corps de *Lettre* et on lui applique la modification décrite dans la section précédente, on obtient la situation présentée dans la figure 3.6. Dans ce schéma, nous identifions tout élément cloné a par a'.

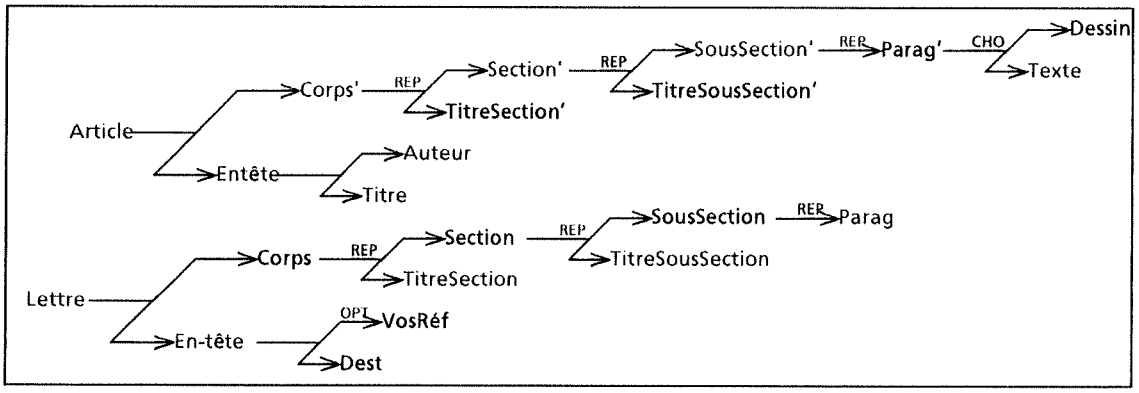


Figure 3.6 : Clonage puis modification du corps.

Cette situation ne nous satisfait pas car elle nous oblige à définir deux fois une structure de corps qui est quasiment la même. Cela est dû à l'utilisation des liens de composition qui font directement référence aux sous-éléments utilisés. Si l'on définit deux objets *Parag* et *Parag'* (qui sont les deux seuls éléments qui diffèrent), nous sommes obligés de définir deux objets *SousSection*, l'un étant

composé d'une répétition de *Parag*, et l'autre d'une répétition de *Parag'*. De manière récursive, on devra définir deux objets *Section*, puis deux objets *Corps*.

Si l'on regarde la définition SGML du corps de *Lettre* (figure 3.7) et celle du corps d'*Article* (figure 3.8), on constate que seul un élément est modifié et que deux éléments sont ajoutés dans la définition d'*Article* (les différences apparaissent en gras dans les deux DTD).

```
...
<!ELEMENT Corps              --  ((Section, TitreSection)+)              >
<!ELEMENT Section            --  ((SousSection, TitreSousSection)+)      >
<!ELEMENT SousSection        --  (Parag+)                                >
<!ELEMENT Parag              --  (#PCDATA)                               >
<!ELEMENT TitreSection       --  (#PCDATA)                               >
<!ELEMENT TitreSousSection   --  (#PCDATA)                               >
...
```

Figure 3.7 : Description SGML du corps de *Lettre*.

```
...
<!ELEMENT Corps              --  ((Section, TitreSection)+)              >
<!ELEMENT Section            --  ((SousSection, TitreSousSection)+)      >
<!ELEMENT SousSection        --  (Parag+)                                >
<!ELEMENT Parag              --  (Dessin | Texte)                        >
<!ELEMENT Dessin             --  (#PCDATA)                               >
<!ELEMENT Texte              --  (#PCDATA)                               >
<!ELEMENT TitreSection       --  (#PCDATA)                               >
<!ELEMENT TitreSousSection   --  (#PCDATA)                               >
...
```

Figure 3.8 : Description SGML du corps d'*Article*.

Pour permettre un clonage minimal des seuls éléments qui sont effectivement différents, nous utilisons un lien de composition indirect. Au lieu de spécifier que la sous-section d'une lettre est composée de *Parag* et que la sous-section d'un article est composée de *Parag'*, nous allons dire que dans les deux cas, l'élément *SousSection* est composé d'une répétition d'un élément de type *Parag* que nous nommons *T-Parag*. Suivant le modèle dans lequel on se trouve, *T-Parag* sera associé à *Parag* ou à *Parag'*. Cela nous oblige à maintenir au niveau de chaque modèle une liste de correspondance entre les types utilisés et les éléments qui leur sont associés.

On peut définir des types pour tous les objets, par exemple *T-Entete* qui regroupe les en-têtes des différents modèles existants, *T-Corps* qui regroupe les corps, etc. Les racines des modèles elles-mêmes peuvent être regroupées dans un type. Nous nommerons *T-Modèle* le type des racines des modèles.

On dispose donc désormais de deux sortes d'éléments : ceux qui se décomposent en d'autres éléments et ceux qui représentent des choix entre différents éléments.

Ces deux types d'éléments sont exactement ceux utilisés pour définir une grammaire de syntaxe abstraite (GSA) [144]. La composition de tous les éléments des modèles est exprimée par des types (figure 3.9). La description d'un modèle particulier est obtenue grâce à une liste de correspondance associée à chaque modèle qui permet de faire correspondre un type à un élément (figure 3.10).

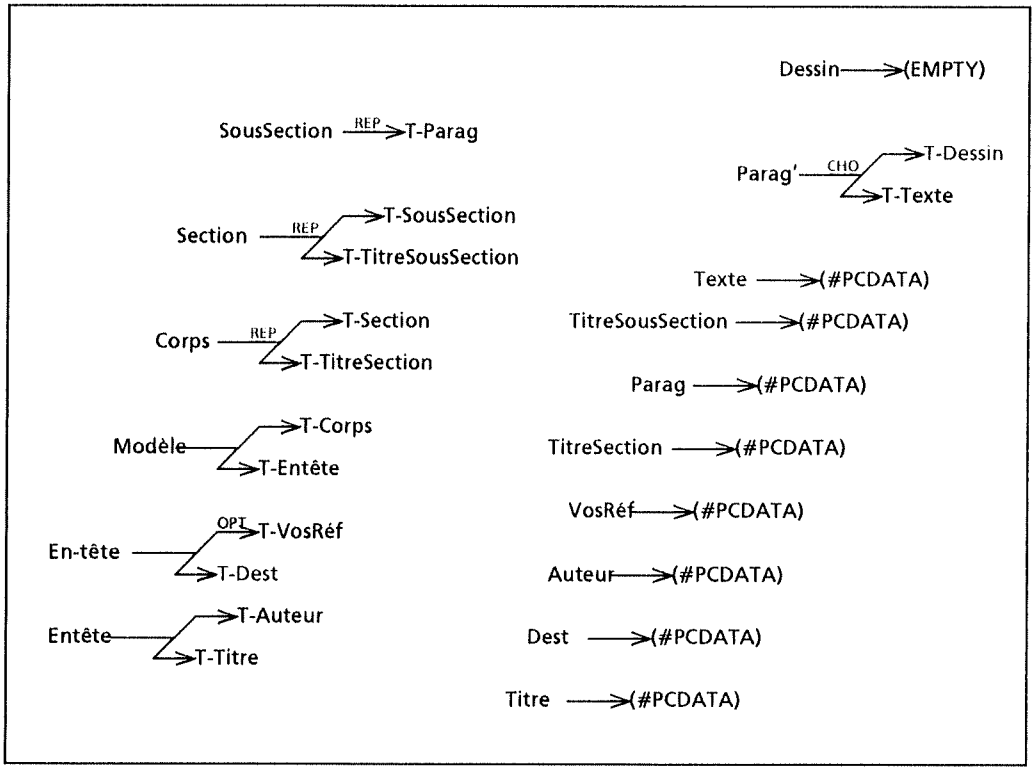


Figure 3.9 : Tous les éléments utilisés dans *Article* & *Lettre* définis par des types.

Avec cette organisation, on peut modifier un élément particulier d'une structure en ne touchant qu'à cet élément. Il suffit pour cela de créer un nouvel élément et de modifier la liste de correspondance associée au modèle concerné.

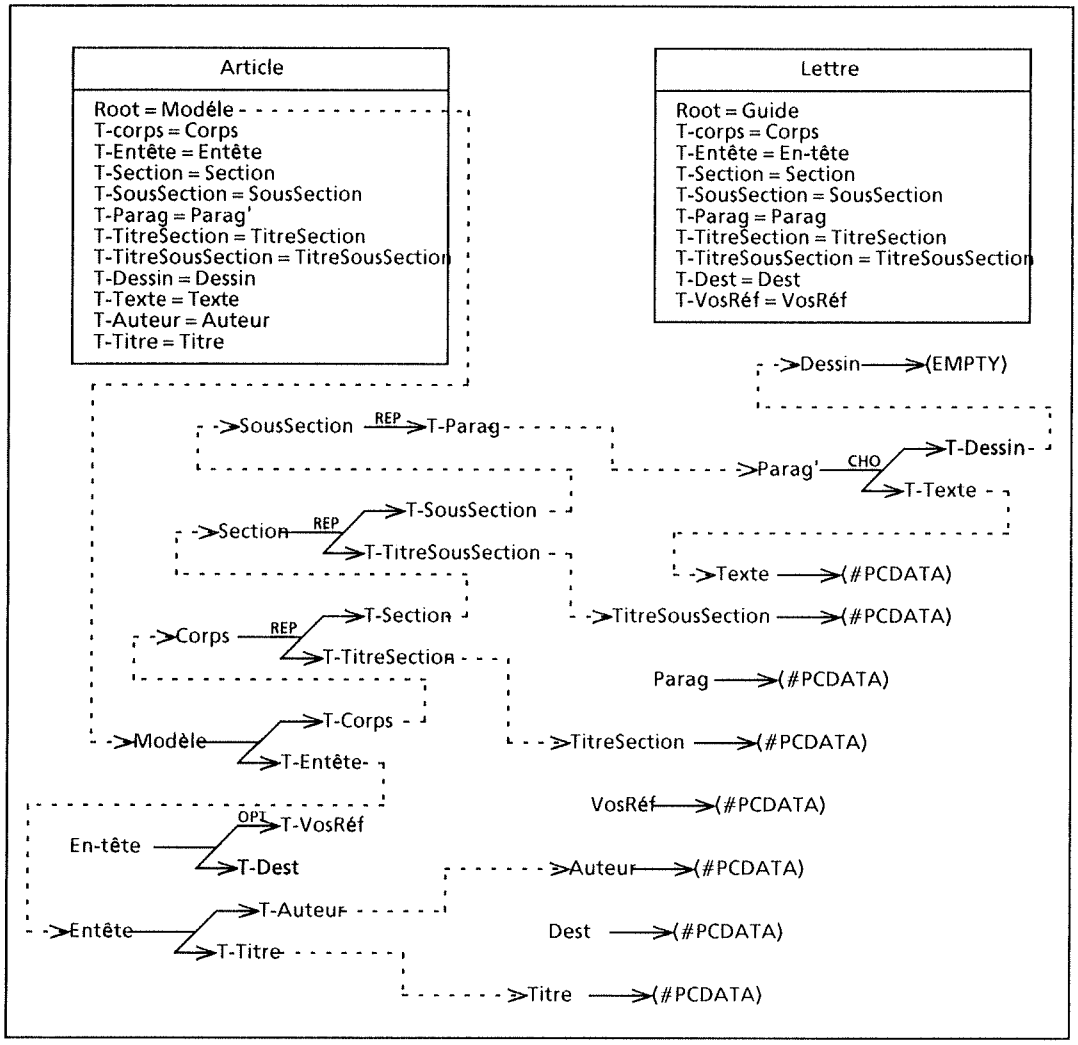


Figure 3.10 : Reconstitution de la définition d'*Article* grâce aux liens indirects.

3.2.3 Clonage-lié de sous-structures

Le clonage-lié[1] est un mélange des deux modes précédents. Il suppose une différence de niveau entre les deux éléments identiques.

Le clone-lié d'un élément possède un lien de **prototypage** avec l'élément original que nous appelons son **prototype**. Ce lien institue une relation de dépendance entre les deux objets.

1. Nous reprenons la même terminologie que celle utilisée par E. Cournarié dans son travail sur l'héritage des contraintes graphiques [34].

Un clone-lié peut être considéré comme une spécialisation où ne figurent que les caractéristiques qui diffèrent de son prototype. De cette manière, un clone-lié "hérite" de toutes les modifications apportées à son prototype. Cette manière de représenter l'héritage est fondée sur la théorie des prototypes où le partage des connaissances se fait par copie différentielle [86].

Dans l'exemple que nous utilisons, si nous définissons le corps d'un article en effectuant un clonage-lié du corps de *Lettre*, alors, la structure du corps d'*Article* sera dépendante de la structure du corps de *Lettre*. Toute modification effectuée sur le corps de *Lettre* affectera le corps d'*Article* alors que les changements faits sur le corps d'*Article* seront sans effet sur la composition du corps de *Lettre*. On est dans une situation mixte : partage quand on modifie le prototype et clonage lorsque l'on agit sur le clone-lié.

Une relation de prototypage liant deux éléments s'applique récursivement pour tous leurs composants. Si nous modifions, par exemple, l'élément *Parag* dans *Lettre*, alors *Parag* d'*Article* sera également changé.

3.2.4 L'organisation des données dans BIBLE

Les concepts développés dans la partie précédente sont très puissants, mais malheureusement assez difficiles à appréhender par un utilisateur. Nous allons nous imposer quelques restrictions :

- le clonage-lié ne s'effectuera qu'entre les modèles où cette notion est assez facile à comprendre et à mettre en oeuvre. Si par exemple, *Lettre* est le prototype d'*Article*, on sait clairement et sans ambiguité, que toutes les modifications faites dans *Lettre* se trouveront répercutées dans *Article*. Par contre, si l'on autorise le clonage-lié entre n'importe quels éléments, on court le risque de se retrouver confronté à des situations très complexes voire impossibles à gérer, du style : *Corps* de *Lettre* prototype du *Corps* d'*Article* alors que *Entête* d'*Article* est le prototype de *En-tête* de *Lettre*, etc.
- le partage des éléments d'un modèle sera interdit pour une meilleure visualisation des relations entre objets. En autorisant uniquement le clonage-lié entre les modèles, les liens de prototypage formeront une hiérarchie unique que l'on pourra facilement visualiser par un graphe (figure 3.11). Par contre, le partage d'éléments générerait des liaisons secondaires entre les branches qui seraient difficiles à représenter et à gérer par l'utilisateur.

La composition de chacun des éléments qui entrent dans la composition d'un modèle est obtenue en appliquant un certain nombre de modifications locales à la composition de l'élément de même nom dans le modèle supérieur. Ainsi, dans l'organisation de la figure 3.11, la structuration de l'en-tête du modèle *ArticleBilingue* (en supposant qu'il contienne un élément *En-tête*) est basé sur la

définition de l'en-tête dans le modèle *Article* qui elle-même est exprimée par rapport à l'en-tête du modèle *Lettre*.

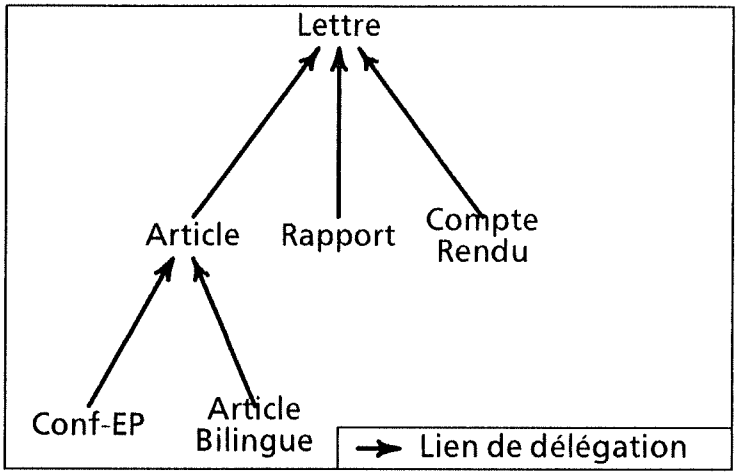


Figure 3.11 : Exemple d'un graphe représentant les relations de prototypage entre les modèles.

Nous utilisons des fonctions pour exprimer, dans un élément, les transformations à effectuer sur la composition héritée de son prototype. Ces fonctions permettent d'exprimer cinq opérations de base :

- **l'ajout** d'un composant dans un élément,
- **la suppression** d'un composant dans un élément,
- **le changement d'indicateur d'occurence** d'un composant (répétition, option, etc),
- **le changement de connecteur** (séquence, aggrégat ou choix),
- **le réordonnancement** des composants.

Regardons sur un exemple comment s'effectue, dans BIBLE, la détermination de la structure d'un modèle de document. La figure 3.12 illustre la représentation d'un modèle *M2* et de son prototype, le modèle *M1*. Les composants utilisables par les modèles sont placés sur la droite du graphique. La structure du modèle *M2* s'obtient de la façon suivante :

On commence en premier lieu par déterminer la racine de la structure. La variable *Root* n'est pas valorisée dans *M2* donc on recherche sa valeur dans le modèle prototype. Dans *M1*, *Root* pointe sur *OB#01* qui constitue une racine commune aux deux modèles. *OB#01* est composé des éléments *En-tête* et *Corps*. On recherche, grâce à la liste de correspondance de *M2*, les objets qui correspondent respectivement à *En-tête* et à *Corps*. On trouve *OB#05* pour Entête et *OB#04* pour *Corps* (valeur récupérée dans le modèle *M1*). On recommence l'opération pour déterminer la structure du premier composant. La composition de

OB#05 est définie par l'opération *+date*. Cela signifie que la composition de l'élément *En-tête* du modèle *M2* est obtenue en ajoutant l'élément *Date* à la composition de l'élément *En-tête* du modèle *M1*. Comme l'en-tête du modèle *M1*, représenté par *OB#02*, est composé de l'élément *Dest*, on en déduit que *OB#05* est composé d'une liste de deux éléments : *Dest* et *Date*. Le modèle M2 est donc défini par la grammaire suivante :

M2 ← En-tête, Corps
En-Tête ← Dest, Date
Dest ← ...
Date ← ...
Corps ← ...

Grâce à cette organisation, les modification effectuées sur le modèle *M1* sont automatiquement prises en compte à chaque évaluation de la structure de *M2*.

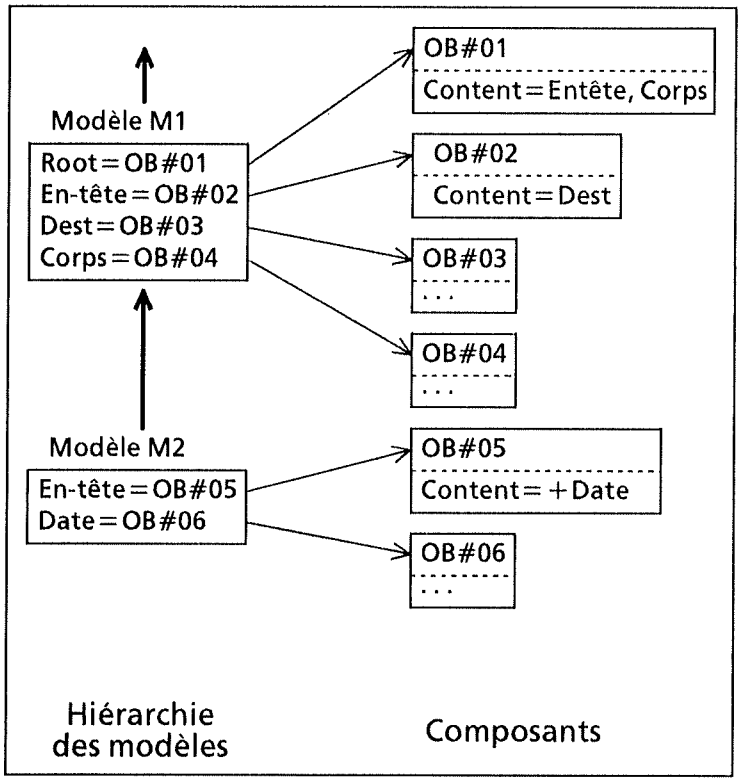


Figure 3.12 : Exemple de représentation de deux modèles

3.2.5 Répercussion des transformations sur les instances

Grâce à la mémorisation des transformations effectuées sur les modèles, il est facile de transformer leur instances ou de déplacer des instances d'un modèle à un autre. Si, en parcourant les relations de prototypage, il est possible de relier le modèle d'origine au modèle de destination, on peut, par concaténation des

opérations effectuées dans les modèles, en déduire la liste des transformations à appliquer aux instances.

Notre but n'est pas d'effectuer une étude exhaustive des transformations possibles. Nous donnons simplement ci-dessous l'influence sur les instances de trois transformations courantes.

En cas d'ajout d'un élément dans un modèle, les modifications à effectuer sur les instances dépendent à la fois de l'indicateur d'occurence de l'objet ajouté et du connecteur utilisé. L'ajout d'un élément optionnel dans une séquence ou d'une alternative dans un choix ne nécessite aucune modification des instances. Au contraire, l'ajout d'un élément obligatoire dans une séquence implique la création d'un élément vide dans les instances. Le tableau de la figure 3.13 récapitule les différents cas :

<table>
<tr><th>Indicateur d'occurence de l'objet \ Connecteur du père de l'objet</th><th>Séquence</th><th>Agrégat</th><th>Choix</th></tr>
<tr><td>Une fois</td><td colspan="2" rowspan="2">Création d'un nouvel élément vide</td><td rowspan="4">RIEN</td></tr>
<tr><td>Une ou plusieurs foix</td></tr>
<tr><td>Zéro ou une fois</td><td colspan="2" rowspan="2"></td></tr>
<tr><td>Zéro ou plusieurs fois</td></tr>
</table>

Figure 3.13 : Transformations sur les instances suite à un ajout dans un modèle.

Pour l'opération de suppression, deux cas peuvent se présenter en fonction du connecteur utilisé. Si c'est une séquence ou un agrégat, on supprime dans les instances le ou les élément(s) correspondant(s). Si c'est un choix, on remplace dans les instances l'élément par la première alternative figurant dans le choix modifié.

Le changement de l'indicateur d'occurence d'un objet consiste, par exemple, à transformer un élément optionnel en un élément obligatoire, une répétition en une répétition optionnelle, etc. L'influence de cette opération sur les instances est fonction de la transformation effectuée et du nombre d'éléments de ce type existant dans les instances. La transformation d'une répétition en une option, ne nécessite par exemple aucune modification des instances si l'élément considéré ne figure qu'une seule fois. S'il apparait plus d'une fois, il est nécessaire de supprimer les éléments en trop. La figure 3.14 récapitule les différents cas :

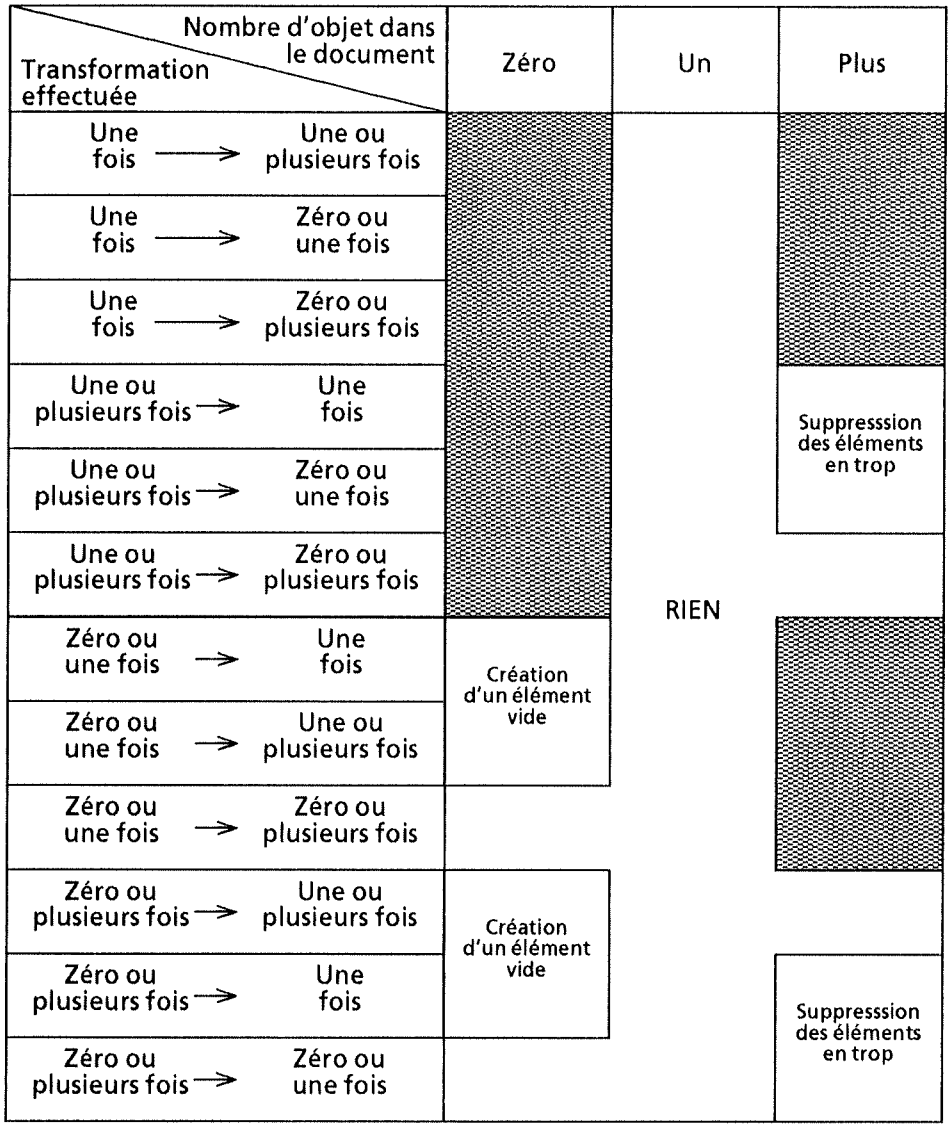


Figure 3.14 : Transformations sur les instances suite à un changement d'indicateur d'occurence.

3.3 Les apports des concepts utilisés

Dans BIBLE, nous utilisons conjointement les concepts de prototypage et de liaison indirecte pour représenter un ensemble de modèles de documents.

L'utilisation de liens indirects permet de définir des modèles qui partagent des éléments communs. Contrairement à la modularité, qui exige que les structures communes soient en tous points identiques, ce concept permet d'effectuer des modifications sur les structures réutilisées. Il est ainsi possible de

modifier n'importe quel élément d'une structure sans avoir à redéfinir les éléments englobants. Ceci permet de ne redéfinir que les éléments qui diffèrent d'un modèle à un autre.

Le concept de prototypage, pour les modèles de documents, permet de représenter ces derniers par des prototypes. Chaque modèle mémorisé est défini par une liste de transformations élémentaires à appliquer à un autre modèle de la base. Ceci permet de prendre en compte dans tous les clones-liés d'un modèle les modifications qui lui sont apportées. Ces modifications peuvent ensuite être répercutées à toutes les instances des modèles modifiés. Le principe de prototypage est appliqué, de manière indirecte, à tous les éléments entrant dans la composition d'un modèle. Ceci se traduit par la relation suivante : si le modèle M1 est un prototype du modèle M2, alors, tous les éléments de M1 sont des prototypes des éléments de même nom entrant dans la composition de M2.

Comme chaque modèle possède un lien avec un autre modèle de la base, il existe une relation entre tous les modèles définis. Nous mémorisons, pour chaque modèle, les opérations de transformation qui sont à l'origine de sa création. Ceci permet de faciliter les opérations de transformation statiques puisque les différences entre deux modèles quelconques sont obtenues directement.

Les gabarits

Dans le cas d'une production de documents en grande quantité, l'utilisation des gabarits comporte deux avantages :

- elle permet de décomposer l'édition des documents en deux phases : une phase d'édition structurelle où la compatibilité de la structure éditée avec le modèle doit être assurée et une phase de génération qui peut être réalisée de manière très simple en résolvant une à une les structures de contrôle utilisées,
- elle permet de factoriser la structure commune à un ensemble de documents et donc de minimiser le stockage. Au lieu de stocker l'ensemble des documents produits, il est en effet possible de ne mémoriser que les gabarits (qui sont forcément en nombre moins important) et les différentes données qui ont été utilisées pour la génération des documents. Un document est complètement défini par la référence de son gabarit, son modèle de présentation et les données utilisées pour sa génération.

4
Les composants logiciels du système

> Compiler construction is today guided by an impressive body of theory: formal language and finite automata theory, programming language semantics, and so on. Textbooks are written and courses taught presenting the whys and wherefores of compiler construction; significantly, this theory exists independently of the implementation details of particular compilers. But this was not always the case. There was a pre-theoretical stage in the development of compilers when they were simply implemented by the seat of the pants, without a clear understanding of the foundation that was needed, or perhaps even an awareness that a theoretical foundation was desirable or achievable.
>
> We believe that document manipulation too is in its pre-theoretical stage.
>
> David M. Levy et al. [83]

Ce quatrième chapitre traite de notre modélisation du système BIBLE, le but de notre travail étant d'élaborer un prototype qui permette d'illustrer le fonctionnement d'un futur produit opérationnel. Nous avons beaucoup mis l'accent sur l'aspect convivialité et facilité d'utilisation. Nous voulions par exemple pouvoir spécifier la présentation d'un type de document de manière totalement graphique. D'autre part, il n'était pas dans notre intention de tout réécrire. Nous avons cherché à réutiliser au maximum des produits disponibles sur le marché. L'éditeur de documents structurés ainsi que le formateur sont deux produits standards qui peuvent être réutilisés. Nous avons choisi d'utiliser le produit Grif qui dispose d'un mode édition WYSIWYG et d'un module de formatage en batch.

Nous avons développé en CLOS [73] un système permettant l'héritage de la composition des modèles ainsi qu'un éditeur de modèles. Les modèles sont représentés selon un format spécifique à l'application qui permet de tirer parti des possibilités des langages orientés objets. Nous avons cependant prévu une sortie

au format SGML. La DTD SGML obtenue passe ensuite dans un parser que nous avons développé dans le langage Haskell [63] pour être transformée en EDTD.

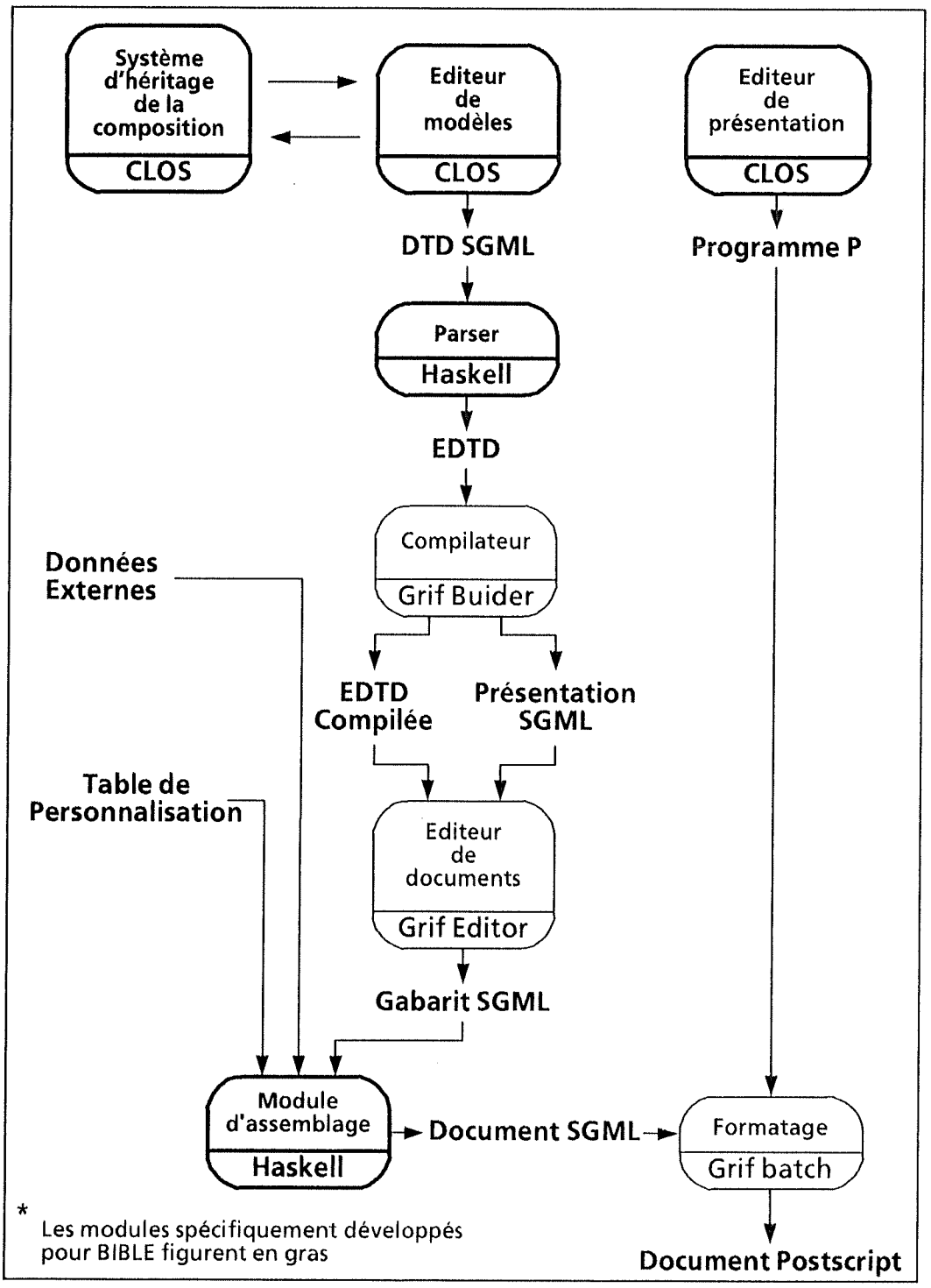


Figure 4.1 : Les différents modules composant l'application BIBLE.

Ce que nous appelons EDTD est une DTD modifiée (étendue), mais cette description est toujours compatible avec la syntaxe SGML. Ceci nous permet de compiler l'EDTD avec le compilateur de DTD de Grif de manière à pouvoir saisir les gabarits sur l'éditeur Grif. Lors de cette opération, une image WYSIWYG du document n'est pas nécessaire, on peut même dire qu'elle n'est d'aucune utilité car les gabarits saisis sont ensuite modifiés pour donner les documents finals. L'édition des gabarits s'effectue donc avec une présentation purement SGML qui est produite en standard par Grif lors de la phase de compilation.

Le passage d'un gabarit qui contient des parties variables à un document prêt à être formaté est réalisé par un module d'assemblage que nous avons également développé en Haskell. Outre le gabarit au format SGML, ce module utilise en entrée une table contenant les données externes nécessaires et une table de personnalisation qui indique le contenu des conditions utilisées dans le gabarit.

Parallèlement, nous avons élaboré un éditeur graphique de boîtes qui prend en compte un grand nombre de caractéristiques et un éditeur de modèles physiques dans lequel sont disposées les boîtes créées. La présentation conçue grâce à ces éditeurs est convertie en langage P. Le langage P est le langage utilisé par Grif pour spécifier la présentation des documents. Cela nous permet d'utiliser un module de formatage de Grif qui, à partir d'un document SGML élaboré par le module d'assemblage et d'un fichier en langage P, génère un fichier PostScript prêt à être imprimé. Nos développements en Haskell sont présentés en annexe B et nos développements en CLOS figurent en annexe A.

Nous avons développé BIBLE sous deux environnements différents :

- sous l'environnement XAIE (Xerox Artificial Intelligence Environment), nous avons développé en CLOS le système d'héritage des modèles, l'éditeur de modèle et l'éditeur de présentation,
- sous X-WINDOW, nous avons écrit en Haskell le module de transformation de DTD et le module d'assemblage de documents. Nous avons également utilisé l'éditeur Grif pour l'édition des gabarits.

Sous XAIE, nous avons utilisé une représentation orientée objet des modèles de documents. Sous X-WINDOW, EDTD, gabarits et documents sont stockés sous la forme de fichiers indépendants. Nous n'avons pas maintenu de liaison directe entre les modèles et les documents. Nous récupérons sous X-WINDOW les modèles exportés au format SGML mais nous n'avons pas accès à l'état de la base de modèles. Ceci ne nous a pas permis d'implémenter un module réalisant la transformation statique des documents.

Les différents développements que nous avons réalisés apparaissent dans la figure 4.1 qui représente l'organisation générale du système BIBLE.

4.1 Représentation orientée-objet de la base de modèles[1]

Un modèle est une structure arborescente basée sur une relation de composition. Nous distinguons les éléments non terminaux de la structure arborescente, qui contiennent d'autres éléments, et les éléments terminaux qui font référence à un type de contenu. Les attributs et les méthodes permettant de représenter ces deux types d'éléments figurent dans deux classes nommées respectivement *COMPOSED-ELEMENT* et *BASIC-ELEMENT*, toutes deux sous-classes de la classe *OBJECT* (figure 4.2).

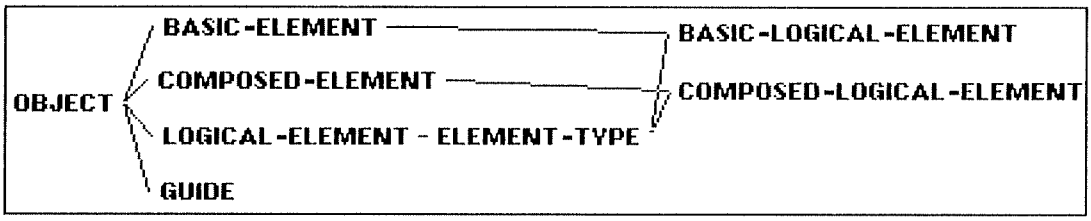


Figure 4.2 : Les classes utilisées pour représenter les modèles.

Comme nous avons également à représenter des objets physiques pour prendre en compte l'aspect présentation, nous avons créé une classe *LOGICAL-ELEMENT* qui constitue l'ancêtre commun de tous les objets logiques. Un élément logique particulier est celui défini dans la section 3.3, que nous avons nommé Type d'élément. Ces éléments sont regroupés dans une classe appelée *ELEMENT-TYPE* qui est une sous-classe de *LOGICAL-ELEMENT*. Une classe *BASIC-LOGICAL-ELEMENT* est destinée à représenter les objets de base logique. Par héritage multiple, elle utilise les attributs et les méthodes des classes *BASIC-ELEMENT* et *ELEMENT-TYPE*. De la même manière, la classe *COMPOSED-LOGICAL-ELEMENT* destinée à représenter les objets composés logiques est créée par combinaison des classes *COMPOSED-ELEMENT* et *ELEMENT-TYPE*.

BASIC-ELEMENT contient une fonction *CONTENT-TYPE* qui retourne le type de contenu associé à l'élément représenté (TEXTE, GRAPHIQUE ou IMAGE). *COMPOSED-ELEMENT* utilise une fonction *COMPOSED-OF* qui retourne la composition de l'élément représenté. *TYPE-OF* est un attribut de la classe *ELEMENT-TYPE* dans lequel est spécifié le type d'élément auquel est rattaché une instance. Dans la classe *ELEMENT-TYPE*, l'attribut *TYPE-OF* d'une instance retourne cette même instance.

1. Dans les exemples tirés de notre prototype, nous faisons référence aux modèles par le terme "guide". Nous avons en effet préféré utiliser cette dénomination pour présenter le concept de modèle aux utilisateurs. Dans les extraits de code que nous présentons, le terme « guide » est également employé.

Tous les éléments non terminaux, y compris la racine d'un modèle, sont représentés par des instances de la classe *COMPOSED-LOGICAL-ELEMENT*. Pour différencier la racine d'un guide représentant une structure complète des autres sous-structures, nous définissons également une classe *GUIDE*.

```
(DEFCLASS GUIDE (OBJECT)
    ;; Slot specifications
    ((ROOT :INITARG :ROOT :INITFORM NIL :ACCESSOR ROOT)
                                        ; Nom de l'élément racine
                                        ; de la structure
     (CONTEXT :INITFORM NIL :ACCESSOR CONTEXT)
                                        ; Liste de
                                        ; correspondance entre
                                        ; les noms des éléments
                                        ; utilisés dans le guide et
                                        ; leur allocation physique
     (KIND-OF :INITARG :KIND-OF :INITFORM NIL
              :ACCESSOR KIND-OF)        ; Pointeur sur le guide
                                        ; utilisé comme prototype
     (FO-DNIK :INITFORM NIL :ACCESSOR FO-DNIK)
                                        ; Liste des guides qui ont
                                        ; comme prototype le
                                        ; guide courant (inverse
                                        ; de kind-of)
     (OLD-VERSION :INITFORM NIL :ACCESSOR OLD-VERSION)
                                        ; mémorisation de
                                        ; l'ancienne version d'un
                                        ; guide modifié
     (ICON :ACCESSOR ICON)              ; Illustration iconifiée du
                                        ; modèle
     ))
```

Figure 4.3: Définition de la classe *GUIDE*.

Dans Bible, la composition d'un modèle est toujours définie par rapport à celle d'un autre modèle. Pour représenter cette hiérarchie, chaque modèle possède une fonction appelée *KIND-OF* qui retourne la référence du modèle utilisé comme prototype (celui dont il hérite la composition). Les modèles possèdent également une fonction inverse appelée *FO-DNIK* qui retourne, pour un modèle, la liste des modèles qui utilisent sa composition. L'attribut *ROOT* contient la référence du type d'élément utilisé comme racine de la structure du modèle. Le lien entre le nom mentionné et la référence de l'instance de la classe

COMPOSED-LOGICAL-ELEMENT qui lui correspond est obtenu grâce à l'attribut *CONTEXT* de la classe *GUIDE*. *CONTEXT* est une liste de correspondance entre les types d'éléments utilisés dans un modèle et leur allocation physique. En effet, un élément *ENTETE*, utilisé dans un modèle, n'est pas forcément défini par la même instance que l'élément *ENTETE* d'un autre modèle. Nous avons également défini un attribut *OLD-VERSION* pour mémoriser l'ancienne version d'un modèle modifié (figure 4.3).

4.1.1 L'héritage de la composition des modèles

Le type de contenu d'un élément de base aussi bien que la composition d'un élément composé sont donnés par une fonction. Nous avons utilisé une fonction parce que ces données sont calculées en fonction du contexte (du modèle) dans lequel les éléments sont utilisés.

C'est la fonction *REFERENCE* qui est utilisée pour donner l'instance qui est associée à un type d'élément dans le contexte d'un modèle (figure 4.4). Comme les modèles sont organisés sous forme d'une hiérarchie représentant une relation d'héritage, si l'association d'un type d'élément avec l'élément qui lui correspond n'est pas mentionnée dans un modèle, alors, il faut aller chercher dans le modèle supérieur.

```
(DEFMETHOD REFERENCE
           ((ELEMENT-TYPE ELEMENT-TYPE)
            (GUIDE GUIDE)
            &OPTIONAL RECURSEFLG)
;;; Retourne la référence d'ELEMENT, dans le contexte de GUIDE
;;; Si la référence n'est pas trouvée et que RECURSEFLG est différent
;;; de NIL, on va chercher la référence de l'objet dans le supérieur du
;;; modèle
  (LET ((REF (SECOND (ASSOC ELEMENT-TYPE (CONTEXT GUIDE)))))
    (COND (REF)
          ((NOT RECURSEFLG) NIL)
          ((NOT (KIND-OF GUIDE)) ELEMENT-TYPE)
          (T (REFERENCE ELEMENT-TYPE (KIND-OF GUIDE) T)))))
```

Figure 4.4 : Définition de la fonction *REFERENCE*.

La composition et le contenu d'un élément particulier d'un modèle sont obtenus par modification de l'élément de même type figurant dans le modèle prototype. Les modifications à apporter à un élément figurent de manière locale à cet élément sous la forme de suites de commandes.

Pour les éléments de base, la seule modification possible concerne le changement de type de contenu. Ceci est spécifié par la commande *CHANGE-CONTENT* suivi du nouveau type de contenu. La figure 4.5 illustre la définition de la fonction *CONTENT-TYPE* qui retourne le type de contenu associé à un élément de base.

Pour les éléments composés, plusieurs modifications sont possibles. On distingue :

- **l'ajout d'un élément**, spécifié par la commande :
  (ADD ELEMENT OPERATEUR) qui a pour effet d'ajouter le couple (ELEMENT OPERATEUR) à la fin de la liste de composition,
- **la suppression d'un élément**, indiquée par la commande :
  (REMOVE ELEMENT) qui a pour effet de supprimer l'élément de la liste de composition,

```
(DEFMETHOD CONTENT-TYPE
           ((ELEMENT BASIC-ELEMENT) (GUIDE GUIDE))
;;; retourne le type de contenu associé à l'objet
  (LET ((INHERITED-CONTENT (CONTENT-TYPE
                            (REFERENCE
                             (TYPE-OF ELEMENT)
                             GUIDE)
                            (KIND-OF GUIDE)))
                                        ; Contenu hérité
       (CDE (CAR (MODIF-OF-CONTENT ELEMENT)))
       (NEW-TYPE (SECOND (MODIF-OF-CONTENT ELEMENT))))
    (IF (EQUAL CDE 'CHANGE-CONTENT)
        NEW-TYPE
      INHERITED-CONTENT)
    ;; Si le type de contenu est modifié, on retourne le nouveau type
    ;; de contenu, sinon on retourne le type hérité
    ))
```

Figure 4.5 : Définition de la fonction *CONTENT-TYPE*.

- **le changement de connecteur**, spécifiée par la commande :
  (CHANGE-CONNECTOR NEW-CONNECTOR) qui permet de remplacer le connecteur courant par un nouveau (SEQ, AGR ou CHO),
- **le changement d'un indicateur d'occurence**, spécifié par la commande :
  (CHANGE-OCCURENCE ELEMENT NEW-OCCURENCE) qui permet de remplacer l'ancien indicateur d'occurence de l'élément spécifié par un nouveau (REP, OPT, OPTREP ou MAN),

- **le réordonnancement** de la liste de composition. Cette opération intervient toujours en dernier, lorsque la composition de l'élément est déterminée. C'est pourquoi nous mémorisons cette commande dans un attribut spécial que nous appelons *REORDER-LIST*. Chaque élément de cette liste de réordonnancement a pour valeur soit 'H', qui signifie que l'on prend l'élément de tête de la liste héritée, soit un nombre n, qui signifie que l'on prend le nième objet défini localement. Avec cette commande, on ne peut modifier que l'ordre des éléments définis localement et les disposer autour des éléments composant la liste héritée. Si la liste de réordonnancement n'est pas spécifiée, on retourne simplement la composition obtenue après application des différentes opérations de modification. Par exemple :

  compositionHéritée = (SEQ (A MAN) (B MAN))
  modificationsLocales = ((ADD (C OPT)))
  réordonnancement = (H 1 H)
  permet d'obtenir comme composition locale : (SEQ (A MAN) (C OPT) (B MAN))
  (l'ordre des éléments A et B ne peut être changé que dans les modèles prototypes)

La fonction *COMPOSED-OF* qui permet d'obtenir la composition d'un élément composé est donnée dans la figure 4.6.

```
(DEFMETHOD COMPOSED-OF
           ((ELEMENT COMPOSED-ELEMENT) (GUIDE GUIDE))
;;; Retourne la composition de l'objet
  (LET ((INHERITED-ELTS (COMPOSED-OF
                         (REFERENCE (TYPE-OF ELEMENT) GUIDE)
                         (KIND-OF GUIDE)))
                                    ; composition héritée
       (LOCAL-ELTS)
       (REORDER-LIST (REORDER-LIST ELEMENT)))
    (LOOP FOR X IN
          (MODIF-OF-COMPONENTS ELEMENT)
          DO
          (LET ((CDE (CAR X))
                (ELT (SECOND X))
                (ARG (CDR X)))
            (COND ((EQUAL CDE 'ADD)
                   (SETQ LOCAL-ELTS
                         (APPEND LOCAL-ELTS (LIST ARG))))
                  ((EQUAL CDE 'REMOVE)
                   (IF (MEMBER ELT INHERITED-ELTS :KEY #'CAR)
```

Figure 4.6 : Définition de la fonction *COMPOSED-OF*.

```
                    (SETQ INHERITED-ELTS
                          (REMOVE ELT INHERITED-ELTS :KEY
                                  #'CAR :COUNT 1))
                (SETQ LOCAL-ELTS
                      (REMOVE ELT LOCAL-ELTS :KEY #'CAR
                              :COUNT 1))))
            ;; On supprime l'élément soit dans la liste héritéé,
            ;; soit dans la liste d'éléments définie localement
            ((EQUAL CDE 'CHANGE-CONNECTOR)
             (SETQ INHERITED-ELTS
                   (CONS ELT (CDR INHERITED-ELTS))))
            ((EQUAL CDE 'CHANGE-OCCURENCE)
             (IF (MEMBER ELT INHERITED-ELTS :KEY #'CAR)
                 (SETQ INHERITED-ELTS
                       (SUBST ARG ELT
                              (INHERITED-ELTS :KEY
                                              #'CAR)))
                (SETQ LOCAL-ELTS
                      (SUBST ARG ELT LOCAL-ELTS :KEY
                             #'CAR))))))
(IF REORDER-LIST
    (LET ((ORDERED-ELTS (CONS
                         (POP INHERITED-ELTS)
                         (LOOP FOR X IN REORDER-LIST
                               COLLECT
                               (IF (EQUAL X 'H)
                                   (POP INHERITED-ELTS)
                                 (NTH (-X 1) LOCAL-ELTS)))))
          (NON-ORDERED-ELTS (APPEND INHERITED-ELTS
                                    (SET-DIFFERENCE
                                     LOCAL-ELTS
                                     ORDERED-SUBORDINATES)))
          ;; NON-ORDERED-ELTS contient les éléments non
          ;; traités par la fonction
          )
      (APPEND ORDERED-ELTS NON-ORDERED-ELTS))
  (APPEND INHERITED-ELTS LOCAL-ELTS))))
```

Figure 4.6 (suite) : Définition de la fonction *COMPOSED-OF*.

4.2 L'éditeur de modèles

Pour tous nos développements graphiques, dont l'éditeur de modèles fait partie, nous avons utilisé CLOS. Comme aucune classe graphique n'était encore définie, nous les avons créées en encapsulant les fonctions graphiques d'INTERLISP. Nous avons défini 7 classes graphiques de base : *WINDOW*, *GRAPHIC-WINDOW*, *BROWSER*, *FREE-MENU*, *GRAPHER*, *MENU* et *MENU-WINDOW* à partir desquelles sont créées les classes spécifiques à l'application BIBLE (figure 4.7).

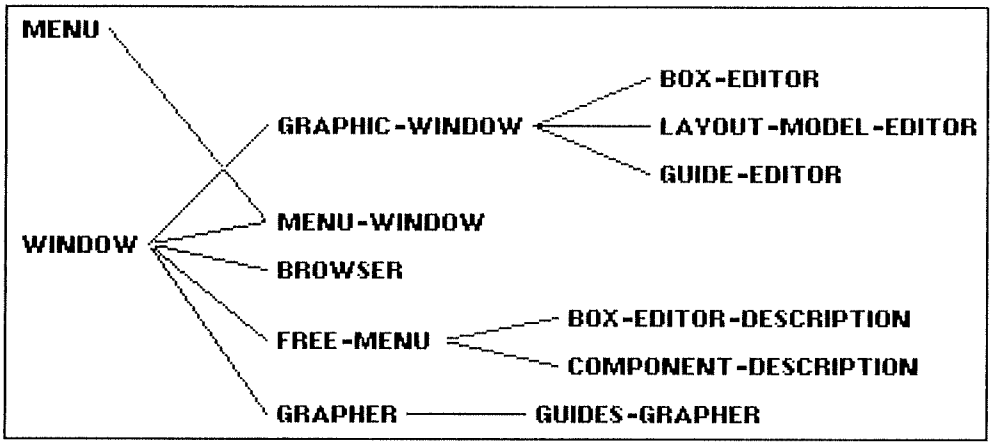


Figure 4.7 : Les classes graphiques définies en CLOS.

Ceci a permis de développer des applications graphiques en n'utilisant que des ordres CLOS, ce qui permet une plus grande cohérence du code développé.

Les spécifications décrites dans la section précédente sont établies automatiquement à partir des actions effectuées par un utilisateur sur l'éditeur de modèles. Un browser représentant la hiérarchie des modèles existants (figure 4.8) permet soit de créer un nouveau modèle, soit de le supprimer, soit de modifier la définition d'un modèle existant, soit encore de générer une DTD correspondant à un modèle :

- la création est une opération de spécialisation. Elle permet de créer le squelette d'un nouveau modèle qui hérite de toutes les spécifications de son délégué. Dans le chapitre 3, nous avons appelé cette opération : clonage-lié,
- la suppression permet d'éliminer un modèle. Deux options sont proposées : soit seul le modèle concerné est supprimé, soit tous ses descendants sont également détruits. Dans ce dernier cas, les transformations locales contenues dans le modèle supprimé sont recopiées dans tous ses descendants.

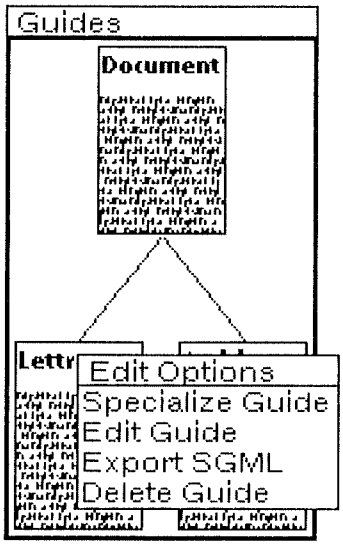


Figure 4.8 : Représentation des modèles par une hiérarchie.

- l'export SGML permet de générer dans un fichier, une DTD SGML qui correspond au modèle sélectionné,
- la modification d'un modèle provoque l'ouverture d'une fenêtre contenant un éditeur de modèle.

L'édition se fait sur une représentation BNF du modèle. Les différentes modifications possibles sont proposées par un menu (figure 4.9).

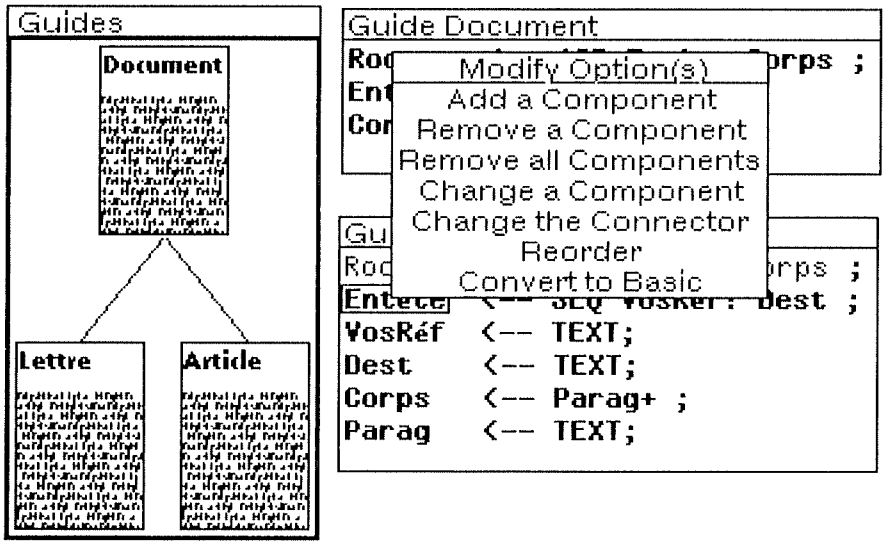


Figure 4.9 : Les différentes opérations possibles sur les éléments d'un modèle.

Chacune des lignes du menu est traduite par une commande comme celles exposées dans la section précédente. Par exemple, l'ajout d'un composant à un

élément (figure 4.10) appelle une méthode qui prend en entrée le type d'objet modifié, le modèle courant et le composant ajouté. La méthode *ADD-SUBORDINATE* dont il est question (figure 4.13 & 4.14) identifie l'élément qui correspond au type et le clone si cela est nécessaire (i.e. si le modèle courant utilise la même instance de l'élément que son délégué).

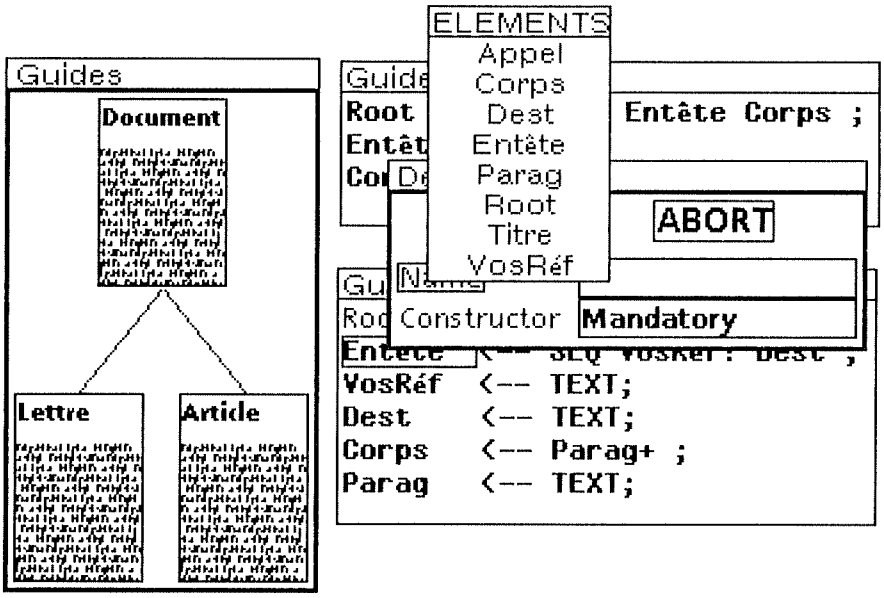


Figure 4.10 : L'ajout d'un composant dans un élément.

```
Guide Lettre
Root     <-- SEQ Entete Corps ;
Entete   <-- AGR Dest VosRef? ;
Dest     <-- CHO Dest1 Dest2 ;
Dest1    <-- SEQ Nom Adr ;
Nom      <-- TEXT;
Adr      <-- TEXT;
Dest2    <-- Nom+ ;
VosRef   <-- TEXT;
Corps    <-- Parag+ ;
Parag    <-- TEXT;
```

Figure 4.11 : Le modèle *Lettre* sur l'éditeur.

L'export SGML permet d'obtenir une définition SGML minimale du modèle réalisé. Nous disons que notre sortie SGML est minimale car elle contient uniquement une description de la hiérarchie des éléments utilisés pour définir le modèle. Les figures 4.11 et 4.12 représentent respectivement la définition d'un modèle de lettre sur notre notre éditeur et la DTD de Lettre qui est produite en sortie.

```
<!DOCTYPE  LETTRE [
<!ELEMENT Lettre        - - (Entete, Corps)>
<!ELEMENT Entete        - - (Dest & VosRéf?)>
<!ELEMENT Dest          - - (Dest1 | Dest2)>
<!ELEMENT Dest1         - - (Nom, Adr)>
<!ELEMENT Nom           - - (#PCDATA)>
<!ELEMENT Adr           - - (#PCDATA)>
<!ELEMENT Dest2         - - (Nom+)>
<!ELEMENT VosRef        - - (#PCDATA)>
<!ELEMENT Corps         - - (Parag+)>
<!ELEMENT Parag         - - (#PCDATA)>
]>
```

Figure 4.12 : La sortie SGML du modèle *Lettre.*

```
(DEFMETHOD ADD-SUBORDINATE
           ((ELEMENT-TYPE ELEMENT-TYPE)
            (GUIDE GUIDE)
            &OPTIONAL COMPONENT)
;;; Ajoute dans la composition de l'élément
;;; correspondant à ELEMENT-TYPE dans GUIDE, le
;;; composant ELEMENT
  (LET ((CURRENT-OBJECT (CLONE-IF-NECESSARY
                         ELEMENT-TYPE GUIDE)))
    (ADD-SUBORDINATE CURRENT-OBJECT COMPONENT)))
```

Figure 4.13 : Définition de la méthode *ADD-SUBORDINATE* pour un type d'élément.

```
(DEFMETHOD ADD-SUBORDINATE
           ((ELEMENT COMPOSED-ELEMENT)
            COMPONENT
            &OPTIONAL OTHER)
;;; Ajoute le composant COMPONENT dans la composition de
;;; ELEMENT
  (SETF
   (MODIF-OF-COMPONENTS ELEMENT)
   (APPEND
    (MODIF-OF-COMPONENTS ELEMENT)
    `((ADD ,@COMPONENT))))
  (LET ((POS (1+ (LENGTH (REORDER-LIST ELEMENT)))))
    (LOOP FOR X IN
          (CONS ELEMENT (FO-DNIK* ELEMENT))
          DO
          (WHEN (REORDER-LIST X)
                (SETF
                 (REORDER-LIST X)
                 (LOOP FOR Y IN
                       (REORDER-LIST X)
                       COLLECT
                       (IF (>= Y POS)
                           (1+ Y) Y)))
                (SETF
                 (REORDER-LIST X)
                 (APPEND (REORDER-LIST X) (LIST POS)))))))
```

Figure 4.14 : Définition de la méthode *ADD-SUBORDINATE* pour un élément composé.

4.3 L'éditeur de présentations

Nous exprimons la présentation des documents par une arborescence de boîtes. Comme pour l'aspect logique, nous distinguons les boîtes de base et les boîtes composées. Ces deux types de boîtes sont représentées par deux classes CLOS : *BASIC-BOX* et *COMPOSED-BOX*. Nous avons défini cinq sous-classes à la classe *COMPOSED-BOX* pour représenter des boîtes composées particulières. Il s'agit des classes *LAYOUT-MODEL*, *PAGE*, *BODY*, *HEADER* et *FOOTER*. Ce qui est spécifique à la représentation des boîtes est contenu dans une classe nommée *BOX*. Tout ce qui a déjà été défini pour la manipulation des objets composés (accès aux sous-objets, opérations sur la hiérarchie, etc.) ou des objets de base (accès au

contenu) est réutilisé grâce à un héritage multiple entre la classe *BOX* et les classes *BASIC-ELEMENT* ou *COMPOSED-ELEMENT* (figure 4.15).

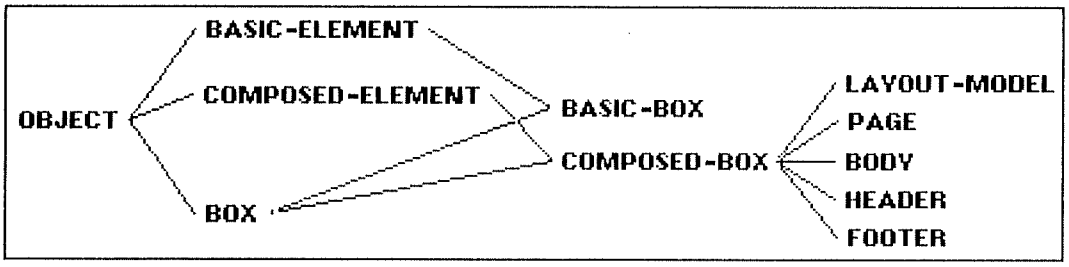


Figure 4.15 : Les classes utilisées pour représenter les éléments physiques.

Pour pouvoir spécifier une présentation de manière graphique, nous avons développé deux éditeurs :

- un éditeur nommé *Layout objects Editor*, où l'utilisateur indique les différentes caractéristiques des boîtes utilisées,
- un éditeur nommé *Layout model Editor*, où l'utilisateur place sur des pages, les boîtes qui figurent avec une position fixe.

4.3.1 L'éditeur de boîtes

Lors de l'édition d'une boîte avec le *Layout objects Editor* (figure 4.16), on doit spécifier, soit graphiquement, soit en entrant des valeurs dans la première partie de l'éditeur, la largeur et la hauteur standard de la boîte. On doit également indiquer si ces dimensions sont fixes, à minimaliser ou à maximaliser. Dans le cas d'un dimensionnement variable, on peut indiquer un maximum à l'agrandissement de la boîte. La minimalisation correspond à l'opération *adjust-to-children* et la maximalisation à l'opération *adjust-to-parent* [57]. On peut également spécifier des marges qui réduisent l'espace disponible pour la disposition de boîtes filles ou de contenus. Quatre autres attributs sont utilisés pour le layout des boîtes à position variable : deux paramètres de séparation et de quatre paramètres d'offset. La séparation est l'espace minimal à conserver entre la boîte courante et les boîtes filles alors que l'offset est l'espace entre la boîte courante et la boîte mère.

Pour résumer, une boîte est décrite avec six attributs de dimensionnement :

- une largeur : *W* (pour Width),
- une hauteur : *H* (pour Height),
- une contrainte de dimensionnement horizontal qui peut prendre pour valeur *FIXED*, *MINSIZE* ou *MAXSIZE* : *WC* (pour Width Constraint),

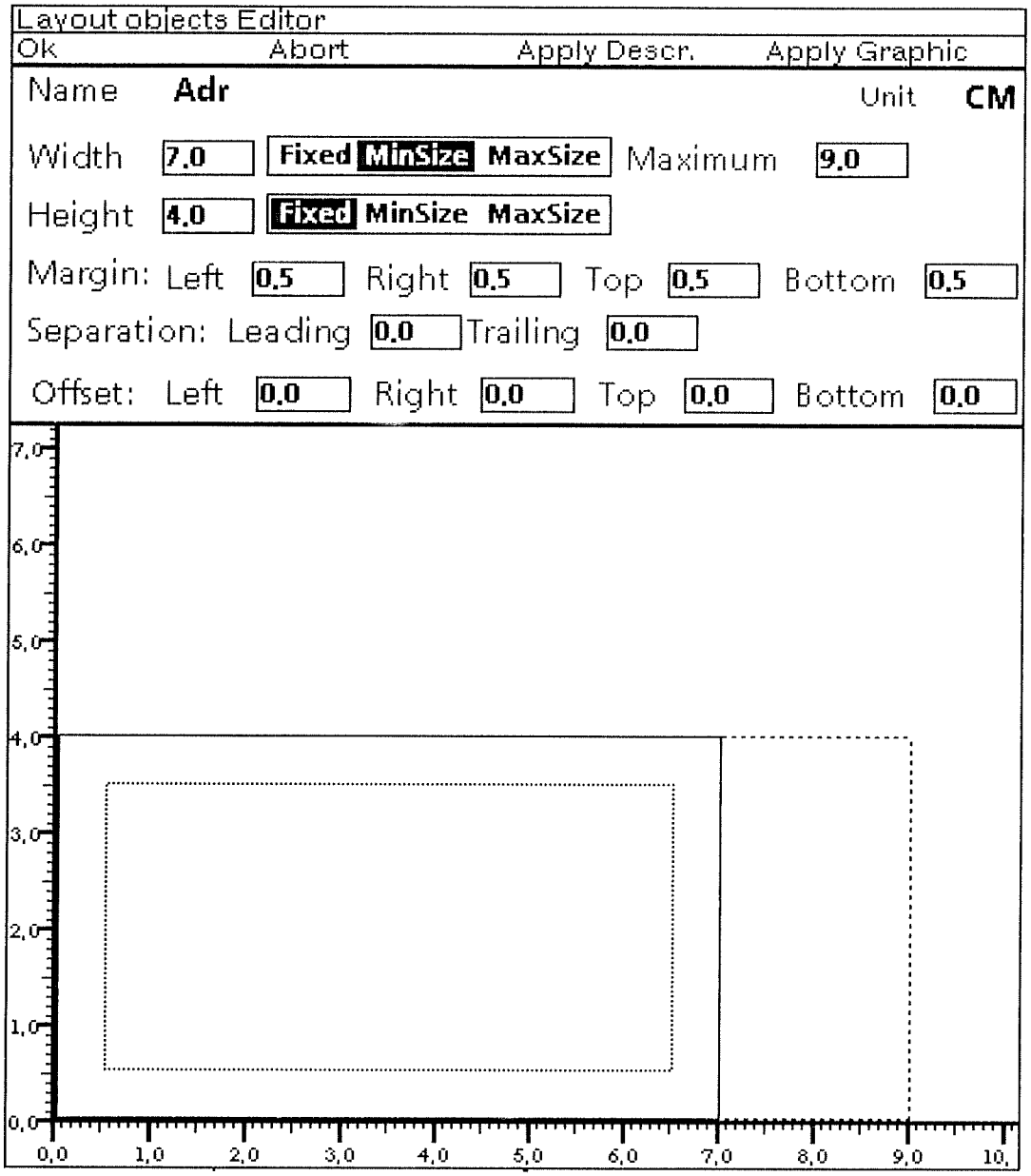


Figure 4.16 : L'éditeur de boîtes.

- une contrainte de dimensionnement vertical qui peut prendre pour valeur *FIXED*, *MINSIZE* ou *MAXSIZE* : *HC* (pour Height Constraint),
- une largeur maximale : *MaxW* (pour Maximum Width),
- une hauteur maximale : *MaxH* (pour Maximum Height),

Pour le positionnement, on utilise 10 Attributs (figure 4.17):

Quatre attributs représentent des distances minimales à respecter entre la boîte et chacune des boîtes inférieures :

- une marge gauche : *LhE* (pour Left-hand Edge),
- une marge droite : *RhE* (pour Right-hand Edge),
- une marge supérieure : *LE* (pour Leading Edge),
- et une marge inférieure : *TE* (pour Trailing Edge).

Quatre attributs expriment les distances minimales entre la boîte courante et la boîte englobante :

- un offset gauche : *LhO* (pour Left-hand Offset),
- un offset droit : *RhO* (pour Right-hand Offset),
- un offset supérieur : *LO* (pour Leading Offset),
- et un offset inférieur : *TO* (pour Trailing Offset).

Deux attributs spécifient les distances minimales à respecter avec les boîtes de même niveau :

- une séparation supérieure : *LS* (pour Leading Separation)
- et une séparation inférieure : *TS* (pour Trailing Separation).

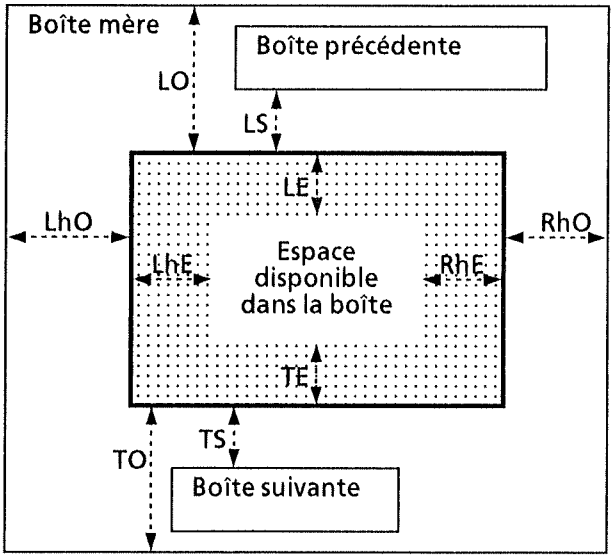


Figure 4.17 : Illustration des attributs de positionnement.

Le *Layout objects Editor* comprend également une zone permettant de spécifier la typographie des éléments textuels se trouvant dans l'objet (figure 4.18). L'utilisateur choisit les attributs typographiques grâce à des menus déroulants. Il faut noter que la spécification des attributs typographiques peut

intervenir au niveau de boîtes composées, ces attributs sont ainsi valables pour toutes les boîtes de niveau inférieur tant que leur valeur n'est pas modifiée.

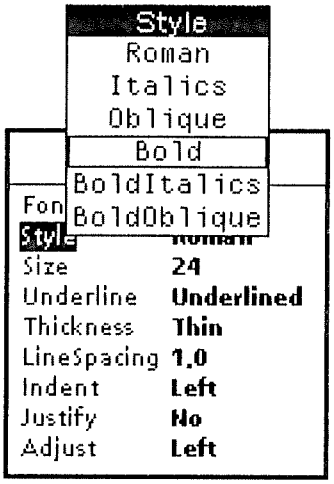


Figure 4.18 : Le choix des attributs typographiques.

4.3.2 Le placement des boîtes sur les pages

La hiérarchie des boîtes dépend de la hiérarchie spécifiée dans le modèle logique. La position effective des boîtes à positionnement variable est calculée lors du formatage du document. Le positionnement des boîtes fixes est réalisé directement par l'utilisateur sur l'éditeur de modèles physiques (figure 4.19). En nous inspirant des spécifications du profil Q112 d'ODA, nous avons simplifié les possibilités de mise en page. Pour nous, un document est constitué d'une première page puis d'une suite d'autres pages toutes identiques (ceci peut être changé pour différencier, par exemple, les pages recto et verso). Chaque page est composée d'un en-tête et d'un pied de page, le corps d'un document est pour sa part partagé entre toutes les pages. On peut utiliser sur cet éditeur toutes les boîtes disponibles qui sont présentées dans le browser de boîtes (figure 4.20). On peut également changer la hauteur de l'en-tête et du pied-de-page.

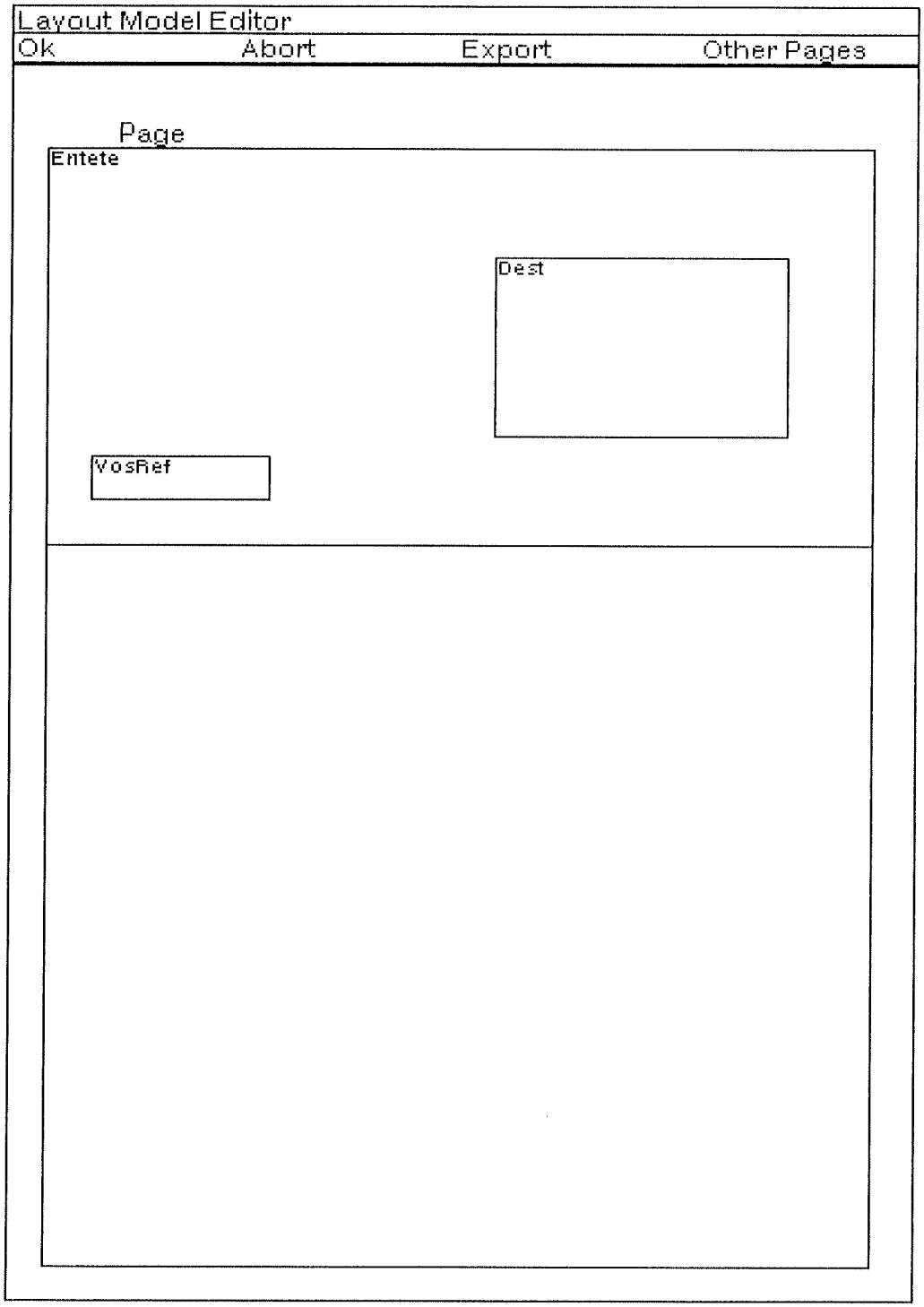


Figure 4.19 : L'éditeur de modèle physique.

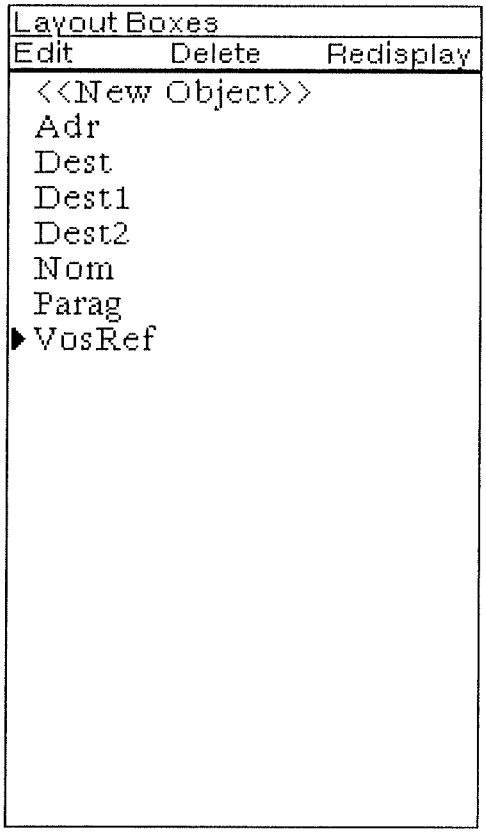


Figure 4.20 : Le browser de boîtes.

4.3.3 La génération du langage 'P'

Comme nous avons choisi d'utiliser Grif pour réaliser le formatage des documents, il faut que les présentations élaborées soient converties dans un format utilisable par cet éditeur. Le langage utilisé par Grif est appelé P. Il consiste à définir des boîtes qui sont associées aux éléments logiques déclarés dans une DTD. Bien que basé sur le principe de boîtes, le langage P diffère par certains points de la présentation exprimable par notre éditeur. Grâce à quelques aménagements, il nous a été cependant possible de générer du P en sortie de notre éditeur de modèle.

Les spécificités du système Grif

La différence principale entre Grif et les notions dont nous nous sommes inspirées pour bâtir notre éditeur est que dans Grif, la présentation n'est pas associée à la description SGML du document. Dans le système Grif en effet, le fichier SGML est transformé en une description spécifique à Grif qui est appelée

langage S. L'une des particularités du langage S est de permettre de décrire de façon très précise les répétitions. Grâce à la syntaxe :

```
ELEMENT = LIST [n .. m] of (SOUS-ELEMENT) ;
```

on peut quantifier le nombre d'occurences des sous-éléments. Par exemple :

```
CORPS = LIST [1 .. 3] of (SECTION) ;
```

permet de limiter à trois le nombre de sections pouvant figurer dans l'élément CORPS.

A l'origine, Grif n'a pas été conçu pour être utilisé avec SGML, la description d'un document était directement écrite avec le langage S. Maintenant, la description S est obtenue par conversion d'un fichier SGML. La description bornée d'une répétition, qui n'est pas possible en SGML, n'est donc jamais utilisée. La répétition SGML est toujours transformée en :

```
CORPS = LIST [1 .. *] of (SECTION) ;
```

Comme le langage S est obtenu par transformation d'une DTD SGML, toutes les particularités du langage S qui ne sont pas exprimables avec la syntaxe SGML sont perdues. Par contre, certaines possibilitées de SGML ne sont pas exprimables directement avec le langage S. Par exemple, dans le langage S, il n'est pas possible d'exprimer des compositions complexes qui ne sont par directement représentables par un arbre. Or, en SGML, ce type de déclarations est très courant. C'est le cas par exemple des trois descriptions suivantes :

```
<!ELEMENT    Auteurs   - - ((Nom, Coord)+)>
<!ELEMENT    Expr      - - (Var, (Comp, Var)?)>
<!ELEMENT    Corps     - - (Tsect, Section+)>
```

Dans le langage S, un élément intermédiaire est créé à chaque fois que l'on rencontre ce type de construction. Par exemple, la description de l'élément *Corps* présentée ci-dessus, nécessite avec le langage S de créer un élément intermédiaire appelé LISTOF-SECTION pour représenter la liste :

```
CORPS       =  BEGIN
               TSECT ;
               LISTOF-SECTION ;
               END
LISTOF-SECT =  LIST [1 .. *] OF (SECTION) ;
```

Le fait de créer de nouveaux noeuds de manière à constituer une structure arborescente plus facilement manipulable est une opération qui se justifie. D'ailleurs, tous les éditeurs SGML effectuent ce style de transformation, même si cela n'est pas visible. Mais le gros problème avec Grif c'est que dans le langage P, les boîtes qui sont créées sont directement associées aux éléments du fichier S. L'utilisateur doit donc, non seulement spécifier la présentation de tous les éléments qui figurent dans une DTD (ce qui semble normal), mais en plus il doit

élaborer une présentation pour tous les éléments qui ont été conçus automatiquement. Il est dommage, pour créer une présentation, d'être obligé de tenir compte d'un formalisme spécifique au produit qui normalement devrait être transparent.

Ce problème lié au système Grif nous a très sérieusement handicapé dans notre développement du module de génération automatique de P. Nous devons en effet, non seulement créer les mêmes boîtes (avec les mêmes noms) que celles qui sont produites lors de la compilation par Grif d'une DTD, mais en plus, nous devons tenir compte de l'influence de ces nouvelles boîtes sur la dimension et la position des boîtes englobées. Dans la description du layout tel que nous l'avons prévue, le positionnement de chaque boîte se fait en fonction de plusieurs paramètres : les marges de la boîte englobante, les offsets de la boîte courante par rapport à la boîte englobante et les séparations entre la boîte courante et les autres boîtes de même niveau (figure 4.21).

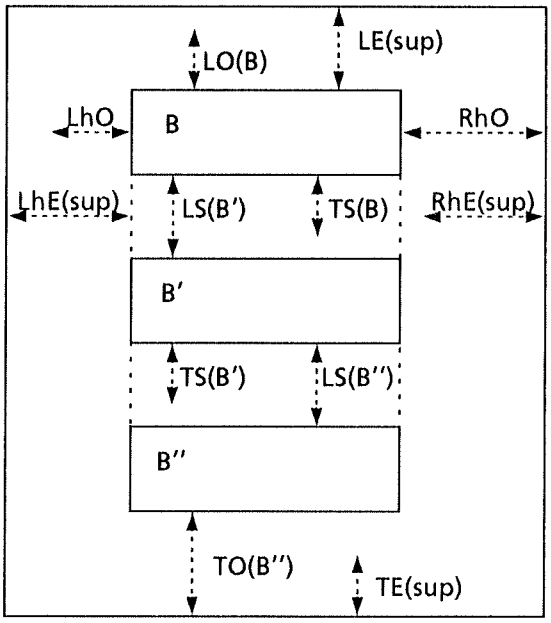


Figure 4.21 : Détermination de la position des boîtes.

Pour transformer cela en P, il faut procéder en deux étapes : d'abord positionner la boîte . . . fictive, en fonction des marges et des offsets, puis placer les boîtes composant l'énumération en ne se souciant que des séparations.

Sur notre éditeur, il est possible de saisir des positionnements complexes qui nécessitent un traitement pour être résolus. Par exemple, si l'on spécifie un offset haut de 1 cm pour une boîte, cela n'implique pas forcément que cette boîte sera séparée de 1cm de sa boîte mère. Il se peut en effet que la boîte englobante possède une marge haute supérieure à cette valeur. Cela n'est pas possible en Grif où un

positionnement n'est exprimé que par rapport à un seul élément. La génération à partir de notre éditeur d'une présentation en langage P n'est donc pas immédiate et nécessite d'effectuer des calculs.

Transformation d'un système de contraintes en langage P

L'appel du générateur de P se fait grâce à une option figurant sur le *Layout Model Editor* (voir figure 4.19). Comme sur cet éditeur ne sont mentionnées que les boîtes ayant une position fixe, il faut, pour pouvoir générer le fichier P, indiquer également le nom du modèle logique concerné de manière à pouvoir accéder aux noms de toutes les boîtes variables (un exemple de fichier 'P' obtenu à partir de la présentation de la figure 4.19 est donné en annexe C).

En P, chaque boîte est munie de trois groupes d'attributs : ceux relatifs à la typographie, ceux exprimant la position et ceux concernant la dimension. De plus, des attributs par défaut peuvent être spécifiés.

1) les attributs par défaut

La spécification de tous les attributs pour chacune des boîtes est facultative. Des règles, données au début du fichier P, indiquent la manière de procéder quand un attribut est absent. Pour tous les attributs typographiques, nous prenons par défaut l'attribut correspondant de la boîte englobante. Ceci est spécifié en valorisant chaque attribut avec *Enclosing* =. En l'absence de toute indication contraire, les dimensions d'une boîte sont calculées pour contenir exactement les boîtes englobées. Ceci est indiqué par les règles :

```
Width : Enclosed . Width;
Height : Enclosed . Height;
```

Par défaut également, la position horizontale d'une boîte est égale à la position gauche de la boîte englobante, par contre, sa position verticale est égale à la position inférieure de la boîte précédente, d'où la génération des règles :

```
HorizPos : Left = Enclosing . Left;
VertPos : Top = Previous . Bottom;
```

2) les attributs typographiques

Au niveau de chaque boîte, comme les attributs typographiques sont identiques à ceux qui sont saisis sur l'éditeur de boîtes, il suffit de les recopier dans chacune des boîtes créées.

3) les attributs de dimensionnement

En ce qui concerne les dimensions, les possibilités du langage P sont moindres que ce qui est permis sur notre éditeur. En particulier, la spécification

de dimensions maximales n'est pas supportée. Nous ne tiendrons donc pas compte de ce paramètre. Il suffit simplement de distinguer deux cas :

- soit la boîte est de dimension fixe, dans ce cas il suffit d'indiquer sa valeur en centimètres par :

  Width = <W> cm;
  ou
  Height = <H> cm;

- soit *MAXSIZE* est spécifié, dans ce cas la dimension de la boîte est égale à la dimension de la boîte englobante :

  Width = Enclosing . Width;
  ou
  Height = Enclosing . Height;

- soit *MINSIZE* est spécifié, dans ce cas la dimension de la boîte est égale à l'espace occupé par ses boîtes filles majoré des marges éventuelles :

  Width = Enclosed . Width + <RhE+LhE> cm;
  ou
  Height = Enclosed . Height + <LE+TE> cm;

4) les attributs de positionnement

La spécification de la position d'une boîte est le cas le plus compliqué. Il convient de distinguer deux cas :

- soit la boîte est positionnée à emplacement fixe sur l'éditeur de modèles physiques. Dans ce cas, on indique sa position en cm par rapport à l'angle supérieur gauche de la boîte englobante :

  HorizPos = Enclosing . Left + <Hpos> cm;
  ou
  VertPos = Enclosing . Top + <Vpos> cm;

- soit la position de la boîte est variable. La position horizontale par rapport au coté gauche de la boîte englobante est égale au maximum entre l'offset gauche de la boîte (*LhO*) et la marge gauche de la boîte englobante (*LhE(sup)*) (voir figure 4.21) :

  HorizPos = Enclosing . Left + <MAX(LhO,LhE(sup))> cm;

  Pour la position verticale, il faut distinguer plusieurs cas :

  - si la boîte est la première d'une énumération, on doit tenir compte de l'offset haut (*LO*) et de la marge haute de la boîte englobante (*Le(sup)*):

    VertPos = Enclosing . Top + <MAX(LO,LE(sup))> cm;

  - si la boîte est, soit au milieu, soit à la dernière place d'une énumération, on doit tenir compte de la séparation haute de la boîte (*LS*) et de la séparation basse de la boîte précédente (*TS(prec)*) :

- si la boîte est dans une boîte fictive, les choses sont plus délicates car les positions des boîtes inférieures sont exprimées par rapport à celle-ci. Pour que cette boîte n'influe pas sur la position des boîtes englobées, nous la créons à la dimension maximale disponible dans la boîte mère.
- si la boîte fait partie d'une répétition, on se trouve dans une situation que l'on ne peut pas traduire en langage P. Le langage P ne permet pas en effet de traiter différemment la première boîte d'une répétition. Mais avec notre éditeur, il est possible d'indiquer pour une même boîte un offset avec la boîte englobante et une séparation avec la boîte qui précède. Nous choisissons, pour traiter toutes les boîtes de manière uniforme, de ne tenir compte que de l'attribut de séparation[1].

4.4 Transformation d'une DTD en EDTD grâce à un parser Haskell

Le module que nous développons ici concerne la transformation en EDTDs des DTDs qui sont produites en sortie de notre éditeur de guide. Au lieu de développer un module séparé, nous aurions très bien pu prévoir directement une sortie du guide saisi en EDTD. Ce n'est pas la solution que nous avons retenue car cela nous obligerait à n'utiliser dans BIBLE que les DTDs qui sont saisies dans l'éditeur de guide. Il est préférable de constituer un module indépendant capable d'analyser une DTD quelconque et de générer l'EDTD correspondante.

Pour développer ce module, nous avons utilisé un langage fonctionnel très bien adapté à l'écriture de parsers. Il s'agit d'Haskell [63]. Le choix de ce langage a été conforté par le fait que le module développé ne nécessite pas de développements graphiques et n'a pas à être connecté avec l'extérieur. Autour du noyau de base d'Haskell, des fonctions de parsing de haut-niveau sont disponibles [65]. Ces fonctions de parsing (voir annexe B.1) définissent un type Parser qui, à partir d'une chaîne de caractères en entrée, retourne les différents tokens reconnus et le reste de la chaîne non analysé. Le principe de base est de concevoir un parser par regroupement de petits parsers. C'est pour cela que des fonctions comme l'alternative (notée |||) ou la séquence (notée +.+) sont définies.

1. Dans le cas où la séparation haute des boîtes répétées est inférieure à leur offset haut et si la boîte englobante est une boîte fictive (ce qui a de grandes chances d'être le cas), nous aurions pu positionner verticalement la boîte LISTOF (créée par le compilateur Grif) à une position égale à LO-LE. Ceci aurait permis, en ne tenant compte que de la séparation des boîtes répétées, de positionner correctement la première boîte à une distance égale à LO du haut de la boîte englobante. Une distance de LO par rapport à la boîte englobante est en effet égale à une distance de LE par rapport à une boîte LISTOF dont la position est de LO-LE. Mais ceci complique le traitement et ne résoud le problème que pour un cas bien particulier. Nous n'avons pas traité ce cas.

Quelques fonctions très utiles sont pourtant absentes. Nous avons donc défini dans le module Parse1 (voir annexe B.1), de nouvelles fonctions générales comme la répétition, la reconnaissance d'un token, et d'autres plus spécialement adaptées à l'analyse de fichiers SGML (reconnaissance des tags de début et de fin, définition des caractères autorisés, etc).

A partir de cette matière première, l'écriture de n'importe quel parser est très naturelle. Par exemple, le parser chargé de reconnaître un flux SGML est décrit comme étant la répétition d'un parser reconnaissant une phrase :

```
pSgml = (some pLigne)
```

pLigne est un parser qui est défini comme une séquence d'autres parsers. Un parser reconnaissant le token "<!" situé en début de ligne, un parser reconnaissant une commande, un parser reconnaissant un commentaire pouvant être optionnel et un parser reconnaissant le token de fin de phrase ">".

```
pLigne = token "<!" ..+ pCommande +.+ (pComment ||| succeed []) +.. spaces +.. token
">" +.. spaces
```

Suivant le même principe, le parser pCommande est défini comme suit :

```
pCommande = pElement ||| pDocType ||| pOtherCdes ||| pEmptyCde
```

Le parser pElement est plus intéressant puisqu'il retourne un type Element :

```
pElement = token1 "ELEMENT" +.. sep ..+ name +.. sep +.+ pTag +.. sep +.+ pTag +..
sep +.+ pModelGroup +.. spaces >> build
                    where build (((s,t1),t2),expr) = Element s t1 t2 expr
```

Les différents types que nous avons définis sont les suivants :

```
data Commande = Element String Bool Bool ModelGroup |
                pDocType String |
                EmptyCde |
                OtherCdes String
                deriving Text
```

Une fois tout le fichier SGML analysé et traduit en types, il ne reste plus qu'à effectuer les actions inhérentes à chacun des types. Grâce au pattern matching de Haskell, cette opération est très vite réalisée. Il suffit d'écrire des fonctions *transcris* chargées de transformer chacune des parcelles du texte analysé. La procédure appropriée est automatiquement sélectionnée en fonction des types d'éléments à traiter. Il ne reste plus ensuite qu'à reconvertir l'arbre d'analyse en ASCII pour effectuer la sortie dans un fichier.

Le parser que nous avons écrit en Haskell est très court (Voir annexe B.2) et effectue convenablement les transformations souhaitées. La figure 4.22 donne le résultat obtenu après le traitement de la DTD *Lettre* présentée dans la figure 4.12.

```
<!DOCTYPE LETTRE [
<!ELEMENT Lettre              --   ((Entete, Corps)|case-Lettre)>
<!ELEMENT case-lettre         --   ((cond?, Lettre)+)>
<!ELEMENT Entete              --   ((Dest & VosRef?) | case-Entete)>
<!ELEMENT case-Entete         --   ((cond?, Entete)+)>
<!ELEMENT Dest                --   (Dest1 | Dest2 | choice-Dest)>
<!ELEMENT choice-Dest         --   (cond?, Dest1, cond?, Dest2)>
<!ELEMENT Dest1               --   ((Nom, Adr) | case-Dest1)>
<!ELEMENT case-Dest1          --   ((cond?, Dest1)+)>
<!ELEMENT Nom                 --   ((#PCDATA|var)+|case-Nom)>
<!ELEMENT case-Nom            --   ((cond?, Nom)+)>
<!ELEMENT Adr                 --   ((#PCDATA|var)+|case-Adr)>
<!ELEMENT case-Adr            --   ((cond?, Adr)+)>
<!ELEMENT Dest2               --   ((Nom+ | rep-Nom)|case-Dest2>
<!ELEMENT case-Dest2          --   ((cond?, Dest2)+)>
<!ELEMENT rep-Nom             --   (Nom, cond)>
<!ELEMENT VosRef              --   ((#PCDATA|var)+|case-VosRef)>
<!ELEMENT case-VosRef         --   ((cond?, VosRef)+)>
<!ELEMENT Corps               --   ((Parag+|rep-Parag)|case-Corps)>
<!ELEMENT case-Corps          --   ((cond?, Corps)+)>
<!ELEMENT rep-Parag           --   (cond, Parag)>
<!ELEMENT Parag               --   ((#PCDATA|var)+|case-Parag)>
<!ELEMENT case-Parag          --   ((cond?, Parag)+)>
<!ELEMENT cond                --   CDATA>
<!ELEMENT var                 --   CDATA>
]>
```

Figure 4.22 : L'EDTD correspondant à la DTD de la figure 4.12.

4.5 L'édition des gabarits sur l'éditeur Grif

L'EDTD que nous avons obtenue en résulat du parsing est compilée pour être utilisée par Grif. Nous effectuons la saisie des gabarits avec la présentation standard SGML qui est générée par le compilateur de Grif. La saisie, sans être le summum de la convivialité, se fait de manière assez intuitive ; tout au moins pour un utilisateur familier avec la documentation structurée. Il ne fait aucun doute que cette caractéristique s'appliquera à l'administrateur de document qui aura à effectuer cette tâche. La seule différence par rapport à la saisie d'un document classique est la possibilité d'utiliser des variables, des conditions et des structures de contrôle pour exprimer la variabilité de certaines parties du gabarit.

Les exemples qui suivent sont tirés de l'édition avec Grif d'une lettre en utilisant l'EDTD données à la figure 4.21. Pour illustrer un conditionnement simple, nous prenons comme exemple la zone *VosRef*. Dans la DTD (figure 4.12), cette zone est déclarée optionnelle. Lors de la saisie du gabarit, nous pouvons conditionner l'apparition ou la non-apparition de l'élément Vos-ref en fonction d'une condition que nous nommons *REFERENCE?*. Cette condition retourne soit TRUE soit FALSE suivant l'existence de références concernant le destinataire de la lettre. Si des références existent, le contenu textuel de l'élément VosRef est donné par la concaténation entre la chaîne de caractères "Ref : " et la variable DATA.REF (figure 4.23).

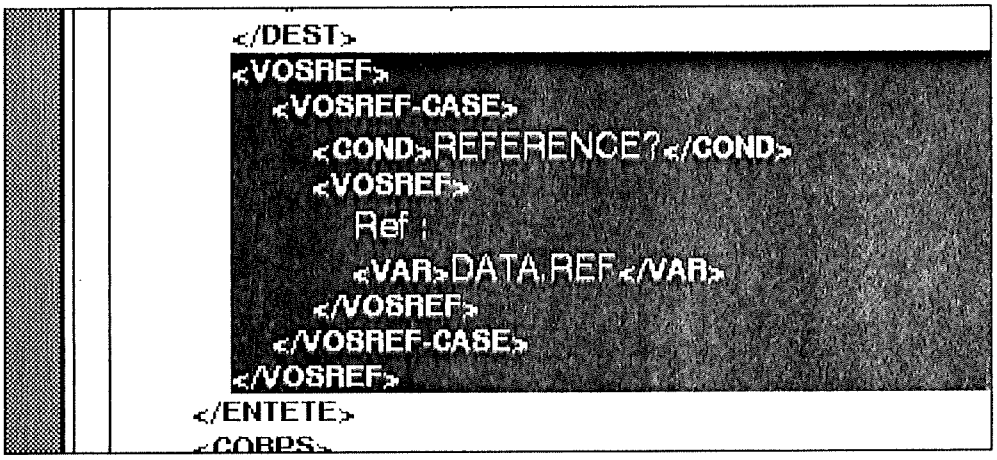


Figure 4.23 : Le conditionnement de l'élément *VosRef*.

La description de l'élément *Dest* fait appel à d'autres constructions. Dans la DTD, il est défini par un choix entre l'élément *Dest1* et l'élément *Dest2* ; *Dest1* étant défini par un nom et une adresse et *Dest2* par une liste de noms. Nous conditionnons ce choix en fonction d'une condition appelée *UN-DEST?* qui retourne TRUE si la lettre n'a qu'un destinataire. Si tel est le cas, *Dest1* est créé, sinon on

conditionne la répétition des noms grâce à une condition *NBDEST* qui retourne le nombre de destinataires (figure 4.24).

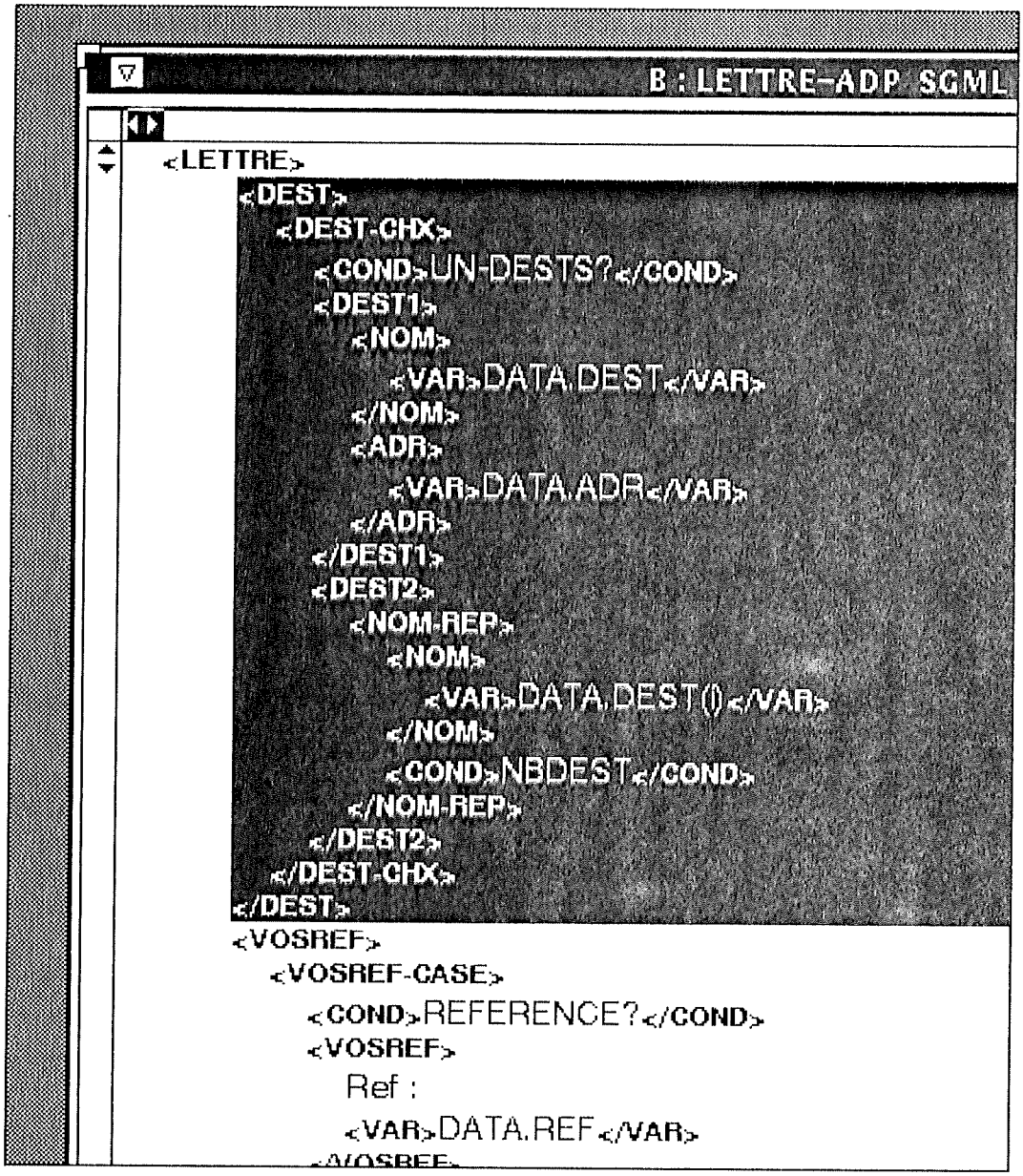


Figure 4.24 : Le conditionnement de l'élément *Dest.*

Grâce à l'EDTD, les structures de contrôle utilisables dans un gabarit sont très souples d'utilisation. On peut en particulier effectuer des imbrications de conditions qu'illustre très bien la définition de la formule d'appel de notre gabarit de lettre (figure 4.25).

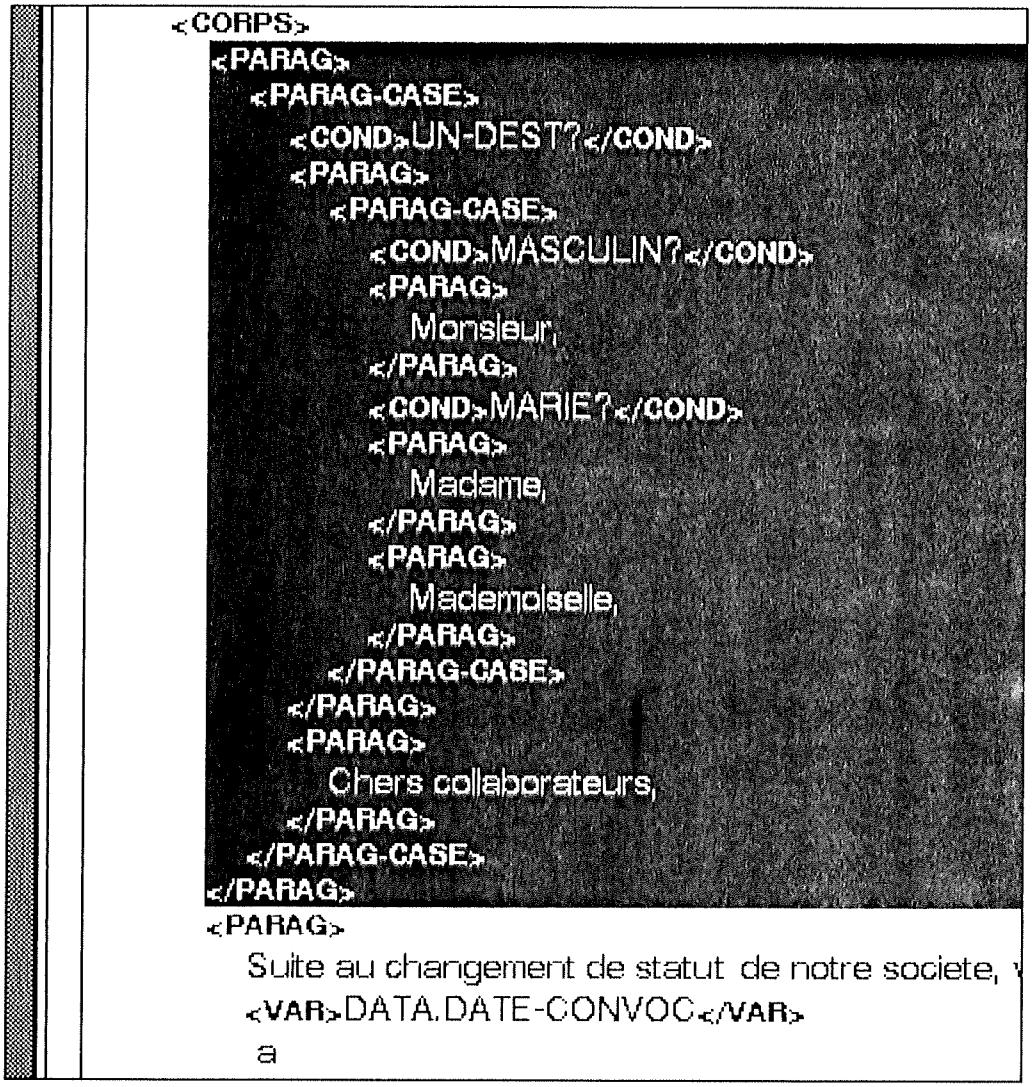


Figure 4.25 : La définition de la formule d'appel.

Les variables externes que l'on a utilisées jusqu'à présent et qui sont identifiées par le préfixe *DATA.* figurent dans un fichier sous forme d'une liste de correspondance entre nom et valeur. A la différence de ces variables qui existent indépendamment de tout document, on utilise dans le gabarit des conditions qui sont spécifiques au document. Les prédicats associés à ces conditions sont également saisis avec l'éditeur Grif en même temps que le gabarit (figure 4.26).

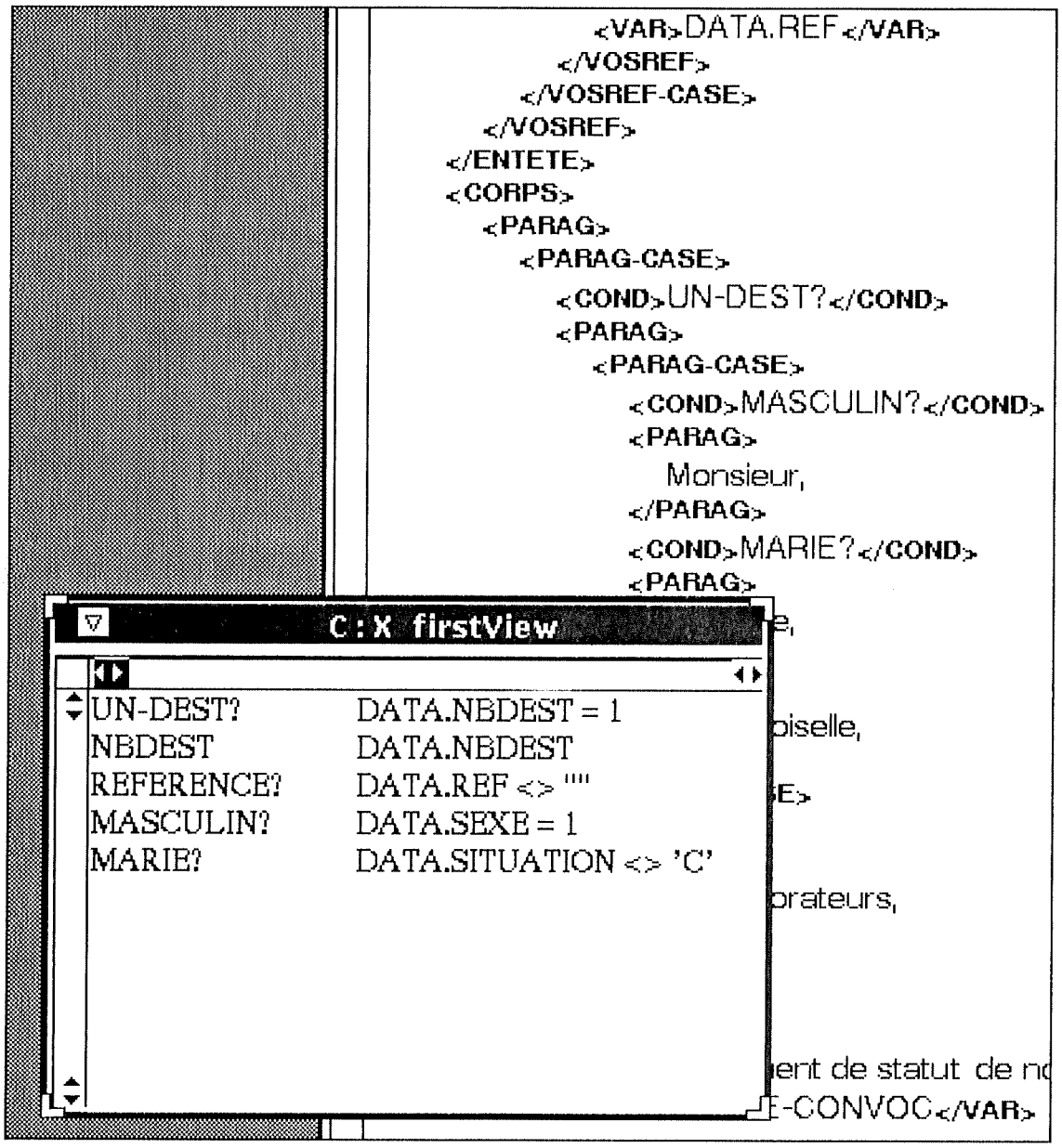


Figure 4.26 : Edition des conditions associées à un gabarit.

Pour permettre d'éditer les conditions avec Grif, nous avons créé une DTD pour le fichier CONDITION ainsi qu'une description de la présentation en langage P (voir Annexe D). La DTD que nous avons définie permet d'associer des combinaisons de prédicats à une condition (figure 4.27).

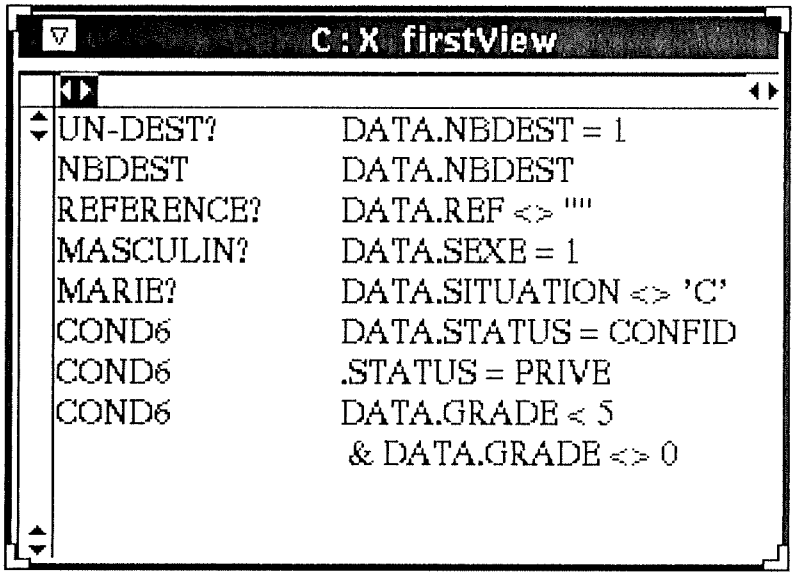


Figure 4.27 : Exemple d'une condition définie par une combinaison de prédicats.

4.6 La génération des documents

A partir du gabarit que nous avons saisi, des conditions qui lui sont associées et d'un fichier de données, nous disposons de toute la matière première nécessaire à la génération d'un document final. Cette opération comprend deux phases :

- une phase d'assemblage des trois groupes d'informations donnés précédemment, qui permet d'obtenir un document balisé SGML,
- et une phase de formatage qui produit un fichier imprimable (en l'occurence en PostScript) à partir d'un document SGML et de directives de présentation.

4.6.1 L'assemblage des documents

L'opération d'assemblage est réalisée grâce à un module que nous avons développé en langage Haskell (voir annexe B.3). Le programme consiste à analyser le gabarit SGML fourni, puis, à le transformer en fonction des conditions et des variables qui lui sont associées. Ensuite, une sortie en ASCII du document SGML est réalisée. Pour illustrer nos propos, nous présentons dans la figure 4.29, le document obtenu par l'assemblage du gabarit défini dans la section précédente (figures 4.23 à 4.25), des conditions de la figure 4.26 et des variables suivantes :

```
DATA.DEST              "Jean Pilon"
DATA.ADR               "Paris XIV"
DATA.REF               "XW09"
DATA.DATE.CONVOC       "8 Janvier 1993"
DATA.HEURE.CONVOC      "10h30"
DATA.NBDEST            1
DATA.SEXE              "1"
DATA.SITUATION         "M"
```

Figure 4.28 : Exemple d'un fichier de données.

```
<!DOCTYPE LETTRE SYSTEM "LETTRE.dcl" [
<!NOTATION GrifImg SYSTEM "GrifImg">
<!NOTATION GrifIDoc SYSTEM "GrifIncludeDoc">
]>
<LETTRE>
<ENTETE>
<DEST>
<DEST1>
<NOM>Jean Pilon</NOM>
<ADR>Paris XIV</ADR>
</DEST1>
</DEST>
<VOSREF>Ref : XW09</VOSREF>
</ENTETE>
<CORPS>
<PARAG>Monsieur,</PARAG>

<PARAG>Suite au changement de statut de notre societe, vous etes convie a une
reunion d'information qui se deroulera le 8 Janvier 1993 a 10h30.</PARAG>

<PARAG>En esperant vous rencontrer a cette occasion, je vous prie d'accepter,
Monsieur, mes sinceres salutations.</PARAG>
</CORPS>
</LETTRE>
```

Figure 4.29 : Un document SGML généré par le module d'assemblage.

4.6.2 Le formatage du document final

Le formatage d'un document est possible grâce aux utilitaires fournis avec l'éditeur Grif. Pour réaliser le formatage d'un document SGML, nous exécutons successivement les modules sgml2PIV et grifbatch. Par exemple, à partir du document de la figure 4.29 et grâce à la présentation définie dans la figure 4.20 (et dont la transcription en langage 'P' est donnée en annexe C), nous obtenons le document de la figure 4.30.

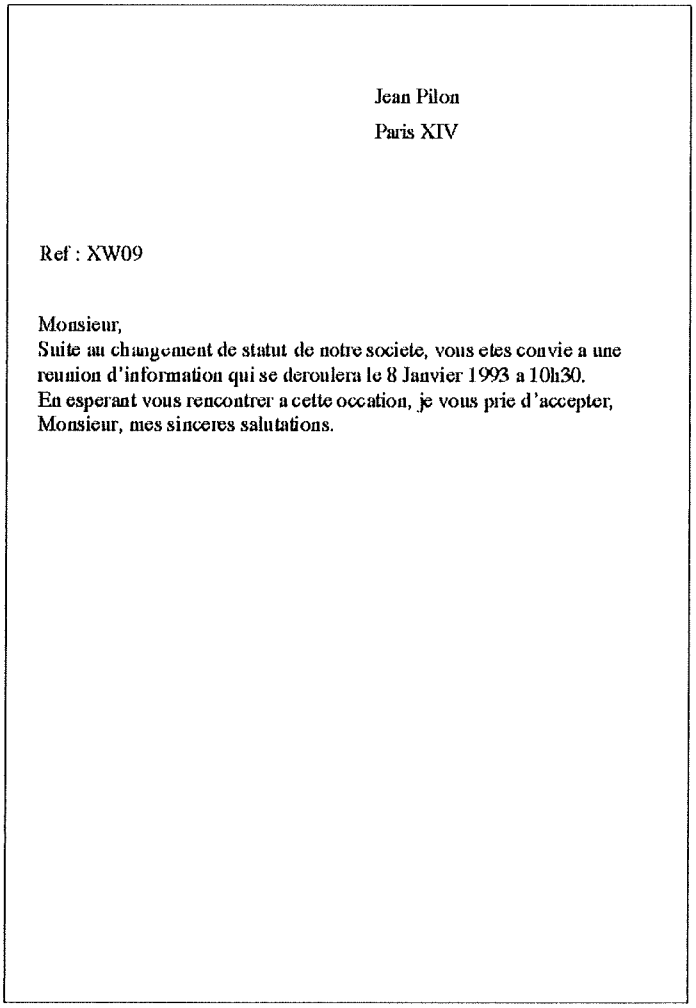


Figure 4.30 : Exemple d'un document final après formatage.

Conclusion

> I'm starting to see more and more DTDs that are taking a more database-oriented approach, using SGML to express real-world structures and relationships rather than composition system data structures.
>
> Eliot Kimber.

Le travail présenté dans ce mémoire comprend deux parties :

- la première traite d'une décomposition de l'opération d'édition, permettant de factoriser les caractéristiques structurelles communes à un groupe de documents. Cette recherche permet de répondre à un besoin du secteur tertiaire où sont produites des quantités énormes de documents de correspondance,
- la seconde consiste en l'étude d'une gestion cohérente d'un ensemble de modèles de documents. Les acteurs susceptibles d'être intéressés par ce sujet sont ici beaucoup plus nombreux puisque sont concernés tous ceux qui ont à manipuler plusieurs modèles de documents.

Malgré l'intérêt évident de ces deux sujets, ils ne sont que peu traités dans la littérature. Concernant plus spécifiquement les opérations relatives à la transformation de type, E. Akpotsui et V. Quint constatent [2] :

> « *It seems that until now, no much attention has been paid to the problem of the type transformation in structured documents. As far as we know, very few papers have been published on that issue [...]. This is probably due to the fact that even in other fields, namely in object-oriented systems and data bases, the problem of type transformation has not yet been solved completely.* »

En plus de cette raison, nous voyons à ce fait plusieurs explications :

- la quasi-totalité des développements effectués dans le domaine des documents structurés concerne l'édition et la génération de documents longs et composites (avec des figures, des tableaux, des formules mathématiques, etc). Les documents de correspondance qui sont simples, relativement courts et associés à un destinataire ne correspondent pas à ces caractéristiques.
- vu les difficultés qu'il y a à passer d'un modèle à un autre, la stratégie actuelle consiste plutôt à minimiser le nombre de modèles et à essayer d'établir des modèles communs utilisés par le maximum de personnes. Autour de SGML, on

voit par exemple se développer des modèles de livres utilisés par les imprimeurs ou les éditeurs (la DTD Docbook ou les DTDs de l'American Association of Publisher), un modèle de documentation diffusé par le département de la défense américain (la DTD CALS 20081), une DTD promue par IBM (la DTD IBMIDDoc), et beaucoup d'autres conçues pour des secteurs économiques particuliers (la DTD J2008 pour l'industrie automobile, la DTD ATA pour l'industrie aéronautique, etc) . Des études visant au passage d'un modèle à un autre sont en cours, mais la règle veut qu'un modèle de document soit fixe ou n'ait à subir des modifications que de manière extrêmement rare.
Cette hypothèse là encore ne convient pas au domaine tertiaire. Souvent, les documents utilisés répondent à des règles imposées qui peuvent être modifiées à tout moment. Des documents tels que les contrats d'assurance ou tous les documents légaux comprennent un certain nombre de clauses ou d'obligations imposées par la législation. Les modèles sont donc susceptibles de changer très fréquemment.

Pour toutes ces raisons, il se trouve que les problèmes que nous avons traités dans cette étude sont relativement nouveaux bien que d'autres travaux se rapprochent de notre problématique.

Nous pensons que des progrès importants dans la manipulation des documents ne pourront être réalisés que dans la mesure où la sémantique des informations contenues dans les documents sera accessible à l'ordinateur [132]. Des domaines de recherche comme les bases de données orientées-objet [1, 24, 36, 58, 60, 74, 138], la compréhension du langage naturel [91, 110], sa génération [35, 94] ou les réseaux sémantiques [43, 102] sont donc porteurs d'évolution pour le traitement des documents. Et, comme l'indique la phrase mise en exergue en tête de cette conclusion, l'évolution a déjà commencé.

Prolongements possibles

Le prototype que nous avons développé est une illustration de ce qu'il est possible de faire en ce qui concerne la manipulation des documents dans le tertiaire. Il permet de traiter entièrement un cas de génération automatisée, mais il reste de gros développements à effectuer pour en faire un produit opérationnel, notamment en ce qui concerne la fiabilité et la facilité d'utilisation.

Pour ce qui est de l'édition des modèles, il est nécessaire de prendre en compte toutes les possibilités de SGML de manière à pouvoir générer une « vraie » DTD, c'est-à-dire comprenant au minimum la définition d'attributs et de liens. Il faudra également étudier les influences de ces ajouts sur le processus d'héritage (devrons-nous prévoir un héritage des attributs ?). Il serait également souhaitable de pouvoir utiliser des structures prédéfinies telles que tableaux ou formules

mathématiques de manière à ne pas avoir à tout reconstruire à partir d'éléments de base.

Dans l'éditeur de présentation, il faudrait prendre en compte plus de paramètres pour coller au plus près au langage P. Ce point n'est cependant pas prioritaire car des éditeurs interactifs de présentation commencent à être développés par des entreprises qui distribuent les éditeurs structurés. Par contre, il faudrait prévoir la possibilité d'héritage entre différentes présentations, de même que la possibilité de conditionner la présentation d'un document.

Dans le prototype, l'édition des gabarits se fait sur une vue purement SGML. Il faudrait rendre cela plus convivial en produisant automatiquement un mode d'édition des gabarits qui soit agréable, par exemple en faisant bien apparaître les conditions, les variables ou les structures de contrôle.

Toujours en ce qui concerne l'édition des gabarits, on peut envisager de tirer partie de la future version de Grif appelée GATE (Grif Application Toolkit Environment) pour rendre la saisie plus aisée. Par exemple, on pourrait stocker dans un fichier les noms des différentes conditions utilisées avec leur définition. Lors de l'utilisation d'une condition dans un document, l'éditeur pourrait alors vérifier automatiquement si cette condition existe et proposer éventuellement sa saisie. On serait ainsi certain d'avoir une concordance entre les conditions utilisées dans un document et les conditions qui sont disponibles.

Pour l'assemblage, toutes les données utiles sont définies localement. Pour que ce produit soit réellement utilisable, il faudrait étudier tout ce qui tourne autour des bases de données, de l'accès aux informations et de l'intégration du système aux chaînes de gestion existantes.

L'apport de ce travail pour le groupe CDC

Le développement d'un produit à partir des concepts utilisés pour bâtir le prototype ne sera pas effectué dans l'immédiat, la demande d'un tel système n'étant pas encore assez forte. Cependant, des besoins localisés relevant de la même problématique apparaissent de plus en plus fréquemment. Les concepts mis en exergue dans ce projet seront donc utilisés au coup par coup pour résoudre des problèmes très précis.

L'expertise acquise durant ces 3 années de travail permettra de répondre à d'autres besoins relatifs aux documents structurés ; domaine destiné à un fort développement dans le secteur tertiaire.

Références Bibliographiques

[1] M. Agosti, G. Gradenigo & P. G. Marchetti, *A Hypertext Environment for Interacting with Large Textual Databases*, Information Processing & Management, vol. 28, no 3, pages 371-387, 1992.

[2] Extase Akpotsui & Vincent Quint, *Type transformation in Structured Editing Systems*, EP'92, Proceeding of the fourth International Conference on Electronic Publishing, Document Manipulation, and Typography, pages 27-41, Lauzanne, Switzerland, 1992.

[3] Extase Akpotsui, Vincent Quint & Cécile Roisin, *Type Modelling for Document Transformation in Structured Editing Systems*, PODP'92, Proceeding of the first international conference on Principles of Document Processing, Washington, October 1992.

[4] Extase Akpotsui, *Transformation de types dans les systèmes d'édition de documents structurés*, Thèse présentée à l'institut polytechnique de Grenoble, 26 Octobre 1993.

[5] Graham Allport & Peter Jarratt, *The old and the new in document processing*, The Electronic Library, vol. 10, no 1, pages 41-46, February 1992.

[6] Y. Amghar & J.M. Pinon, *L'approche objet : une application aux documents ODA*, Journée de l'AFCET, Lyon, 26 Janvier 1989.

[7] Y. Amghar & J.M. Pinon, *Un éditeur de documents ODA : une application de l'approche objet*, WOODMAN'89, Workshop on Object Oriented Document Manipulation, pages 52-67, Rennes, France, Mai 1989.

[8] J. André, R. Furuta & V. Quint, *By Way of an Introduction. Structured Documents: What and why ?*, STRUCTURED DOCUMENTS, Cambridge University Press, pages 1-6, 1989.

[9] Dennis S. Arnon, Richard Beach, Kevin McIsaac & Carl Waldspurger, *Camino-Real: An Interactive Mathematical Notebook*, Document Manipulation and Typography, Proceedings of the International Conference on Electronic Publishing, Document Manipulation and Typography, Nice (France) April 20-22, 1988, Cambridge University Press, pages 1-18, 1988.

[10] Dennis S. Arnon, Isabelle Attali & Paul Franchi-Zannettacci, *Layout of Cedar documents in Typol*, Rapport de recherche Xerox, 1990.

[11] Dennis S. Arnon, Isabelle Attali & Paul Franchi-Zannettacci, *Language-based Document Processing*, PODP'92, Proceeding of the first international conference on Principles of Document Processing, Washington, October 1992.

[12] Dennis S. Arnon, *Scrimshaw: A language for document queries and transformations*, EP'94, Proceeding of the fifth International Conference on Electronic Publishing, Document Manipulation, and Typography, pages 385-396, Darmstadt, Germany, 13-15 April 1994.

[13] Rolf Bahlke & Gregor Snelting, *The PSG System: From Formal Language Definitions to Interactive Programming Environments*, ACM Transactions on Programming Languages and Systems, Vol. 8, No 4, pages 547-576, October 1986.

[14] P. Baudelaire, *Design models for document preparation systems*, Proceeding of International Conference on research and trends in document preparation systems, Lausanne, Switzerland, page 19-21, February 1981

[15] Ellen Beal, *Smart Documents*, Computer Graphics World, vol. 14, no 5, pages 53-58, May 1991.

[16] Eric A. Bier & Aaron Goodisman, *Documents as User Interfaces*,EP'90, Proceeding of the international conference on Electronic Publishing, Document Manipulation & Typography, pages 249-262, Gaithersburg, Maryland, September 1990.

[17] D. G. Bobrow & T. Winograd, *An overview of KRL, a Knowledge Representation Language*, Cognitive Science, pages 3-46, 1977.

[18] P. Borras, D. Clement, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang & V. Pascual, *CENTAUR: the system*, INRIA Research Report 777, 1987 & SIGSOFT'88, Third Annual Symposium on Software Development Environments, Boston, November 1988.

[19] K. Borup & E. Sandvad, *Users and Programmars Guide for Sif - a Syntax-directed Editor*, Project Mjolner Working Note, DK-SYS 29.1, Sysware/Aarhus University, July 1988.

[20] Kenneth P. Brooks, *Liliac: A two-view Document Editor*, IEEE Software, June 1991, pages 7-19.

[21] Allen L. Brown, Jr. & Howard A. Blair, A *logic Grammar for Document Representation and Document Layout*, EP'90, Proceeding of the international conference on Electronic Publishing, Document Manipulation & Typography, pages 47-64, Gaithersburg, Maryland, September 1990.

[22] Allen L. Brown, Jr., Toshiro Wakayama & Howard A. Blair, *A Reconstruction of Context-Dependent Document Processing in SGML*, EP'92, Proceeding of the fourth International Conference on Electronic Publishing, Document Manipulation, and Typography, pages 1-25, Lauzanne, Switzerland, 1992.

[23] H. Burkhart & J. Stelovsky, *Towards an integration of editors*, Proceeding of International Conference on Research and Trends in Document Preparation Systems, Lausanne, Switzerland, page 9-11, February 1981.

[24] Forbes J. Burkowski, An Algebra for Hierchically Organized Text-Dominated Databases, Information Processing & Management, vol. 28, no 3, pages 333-348, 1992.

[25] CCITT Recommendations T.410 Serie : *Open Document Architecture (ODA) and Interchange Format*, 1988.

[26] Xavier Ceugniet & Vincent Lextrait, *Génération de serveurs de vues*, Thèse présentée à l'université de Nice - Sophia Antipolis, 18 Décembre 1992.

[27] François Chahuneau, *SGML and Databases*, Tutorial at the International Conference on Electronic Publishing, Swiss Federal Institute of Technology, Lausanne, April 1992.

[28] François Chahuneau, *SGML and Databases*, Tutorial at the fifth International Conference on Electronic Publishing, Document Manipulation, and Typography, Darmstadt, Germany, 13-15 April 1994.

[29] Donald D. Chamberlin, Helmut F. Hasselmeier & Dieter P. Paris, *Defining Document Styles For WYSIWYG Processing*, Document Manipulation and Typography, Proceedings of the International Conference on Electronic Publishing, Document Manipulation and Typography, Nice (France) April 20-22, 1988, Cambridge University Press, pages 121-138, 1988.

[30] Donald D. Chamberlin, *An Adaptation of Dataflow Methods for WYSIWYG Document Processing*, Proceeding of the ACM Conference on Document Processing Systems, Santa Fe, New Mexico, ACM Press, pages 101-109, December 5-9 1988.

[31] J. Conklin, *Hypertext: An Introduction and Survey*, IEEE Computer, vol. 9, no 20, pages 17-41, September 1987.

[32] Giovanni Coray, Rolf Ingold & Christine Vanoirbeek, *Formatting structured documents: Batch versus interactive*, Text Processing and Document Manipulation, pages 154-170, Cambridge University Press, Proceeding of the international conference, University of Nottingham, April 1986.

[33] Giovanni Coray, Karen Lemone & Christine Vanoirbeek, *The Use of Inheritance in Document Specifications*, WOODMAN'89, Workshop on Object Oriented Document Manipulation, pages 165-169, Rennes, France, Mai 1989.

[34] Eric Cournarié, *Contraintes et Prototypes pour la Description du Comportement des Objets dans l'Interaction Homme-Machine*, Thèse soutenue à l'université d'Orsay, no 2006, Mars 1992.

[35] L. Danlos, *The linguistic basis of automatic generation*, Cambridge University Press, 1986.

[36] José Valdeni De Lima, *Gestion d'objets composés dans un SGBD : cas particulier des documents structurés*, Thèse soutenue à l'université de grenoble, Mars 1990.

[37] Claude Delobel, Christophe Lécluse & philippe Richard, *Bases de données : des systèmes relationnels aux systèmes à objets*, InterEditions, 1991.

[38] Thierry Despeyroux, *TYPOL : a Formalism to Implement Natural Semantics*, INRIA research report 94, 1988.

[39] Prasun Dewan, *Object-Oriented Editor Generation*, Journal of Object Oriented Programming (JOOP), pages 35-49, July/August 1990.

[40] V. Donzeau-Gouge, G. Huet, G. Kahn & B. Lang, *Programming environments based on structured editors : The Mentor experience*, Rapport de recherche INRIA no 26, Juillet 1984.

[41] V. Donzeau-Gouge, G. Kahn, B. Lang & B. Mélèse, *Document structure and modulatiry in Mentor*, SIGPLAN notices 19, page 141-148. 1984.

[42] Paul M. English, Ethan S. Jacobson, Robert A. Morris, Kimbo B. Mundy, Stephen D. Pelletier, Thomas A. Polucci & H. David Scarbro, *An Extensible, Object-Oriented System for Active Documents*, EP'90, Proceeding of the international conference on Electronic

Publishing, Document Manipulation & Typography, pages 263-276, Gaithersburg, Maryland, September 1990.

[43] J. Fargues, M. Landau, A. Dugourd & L. Catach, *Conceptual Graphs for Semantics and Knowledge Processings*, IBM Journal of R&D, 1986.

[44] An Feng & Toshiro Wakayama, *SIMON: A grammar-based transformation system for structured documents*, EP'94, Proceeding of the fifth International Conference on Electronic Publishing, Document Manipulation, and Typography, pages 361-372, Darmstadt, Germany, 13-15 April 1994.

[45] J. Ferber, *MERING : un langage d'acteur pour la représentation et la manipulation des connaissances*, Thèse de Docteur-Ingénieur, Université de Paris 6, 1983.

[46] Paul Franchi-Zannettacci & Dennis S. Arnon, *Context-sensitive semantics as a basis for processing structured documents*, WOODMAN'89, Workshop on Object Oriented Document Manipulation, pages 135-146, Rennes, France, Mai 1989.

[47] Richard Furuta, *An integrated, but not Exact-Representation, Editor/Formatter*, Text Processing and Document Manipulation, pages 246-259, Cambridge University Press, Proceeding of the international conference, University of Nottingham, April 1986.

[48] Richard Furuta, Vincent quint & Jacques André, *Interactively editing structured documents*, Electronic Publishing, vol. 1, pages 19-44, April 1988.

[49] Richard Furuta & P. David Scotts, *Specifying Structured Document Transformations*, EP'88, Proceeding of the International Conference on Electronic Publishing, Document Manipulation, and Typography, pages 109-120, Nice, France, 1988.

[50] Richard Furuta & P. David Scotts, *Object Structures in Paper Documents and Hypertexts*, WOODMAN'89, Workshop on Object Oriented Document Manipulation, pages 147-151, Rennes, France, Mai 1989.

[51] Richard Furuta, *Concepts and models for structured documents*, STRUCTURED DOCUMENTS, Cambridge University Press, pages 7-38, 1989.

[52] Charles Goldfarb, *Document Composition Facility: Generalized Markup Language (GML" Users Guide*, Technical Report SH20-9160-0, IBM, 1978.

[53] Michael J. Groves & David F. Brailsford, *Separate compilation of structured documents*, EP'94, Proceeding of the fifth International Conference on Electronic Publishing, Document Manipulation, and Typography, pages 315-326, Darmstadt, Germany, 13-15 April 1994.

[54] J. Gutknecht & W. Winiger, *Andra: The document preparation system of the personal workstation Lilith*, Software - Practice and Experience, No 14, pages 73-100, 1984.

[55] J. Gutknecht, *Concepts of the text editor Lara*, Communications of the ACM, No 28, pages 942-960, September 1985.

[56] Richard Hamlet, *A Disciplined Text Environment*, Text Processing and Document Manipulation, Cambridge University Press, Proceeding of the international conference, University of Nottingham, pages 78-89, April 1986.

[57] N.Hayashi, K. Saito, H. Nakatsuyama, Y. Suzuki, M. Murata, *Some ODA layout control problems revealed by a prototype editor*, WOODMAN'89, Workshop on Object Oriented Document Manipulation, pages 79-90, Rennes, France, Mai 1989.

[58] M. A. Heater & B. N. Rossiter, *Theoretical Structures for Object-based Text*, WOODMAN'89, Workshop on Object Oriented Document Manipulation, pages 178-192, Rennes, France, Mai 1989.

[59] Eric van Erwijnen, *Practical SGML*, Kluwer Academic Publishers, 1990.

[60] Wolfgang Herzner & Erwin Hocevar, *CDAM - Compound Document Access and Management : An Object-Oriented Approach*, SIGOIS Bulletin, ACM Press, vol. 12, no 1, July 1991.

[61] Mattew E. Hodges & Russel M. Sasnett, *Plastic Editors for Multimedia Documents*, Proceeding of the summer 1991 Usenix Conference, Nashville, Tennessee, USA, pages 463-473, June 1991.

[62] W. Horak, *Office Document Architecture and Office Document Interchange Formats : Currents Status and International standardization*, Computer, vol. 18, no 10, October 1985.

[63] P. Hudak, S. Peyton Jones & P. Walder, *Report on the Programming Language Haskell, A Non-strict Purely Functional Language (version 1.2)*, ACM SIGPLAN Notices, 27(5), May 1992.

[64] S. E. Hudson, *UIMS Support for Direct Manipulation Interface*, Computer Graphics, vol. 21, no 2, pages 120-124, Avril 1987.

[65] Graham Hutton, *Higher-Order Functions for Parsing*, To appear in the Journal of Functional Programming, March 1992.

[66] Richard Ilson, *Interactive Effectivity Control: Design and Applications*, Proceeding of the ACM Conference on Document Processing Systems, Santa Fe, New Mexico, ACM Press, pages 85-92, December 5-9 1988.

[67] ISO 10179 (Draft), *Information Technology - Text and Office Systems - Document Style, Semantic and Specification Language (DSSSL)*, 1991.

[68] ISO 8613, *Information Processing - Text and Office Systems - Office Document Architecture (ODA) and Interchange Format*, 1988.

[69] ISO 8879, *Text and Office Systems - Standard Generalized Markup Language (SGML)*, 1986.

[70] Philip N. Johnson-laird, *Mental models: toward a cognitive science of language, inference and consciousness*, 1983.

[71] V. Joloboff, *Document representation : concepts and standards*, STRUCTURED DOCUMENTS, Cambridge University Press, pages 75-106, 1989.

[72] G. Kahn, *Natural Semantics*, Proceeding of Symposium on Theoretical Aspects of Computer Science, Passau, Germany, Lecture Notes in Computer Science No 247, 1987.

[73] S.E. Keene, *Object-oriented programming in COMMON LISP : a programmer's guide to CLOS.*

[74] Pekka Kilpelainen, Greger Linden, Heikki Mannila & Erja Nikunen, *A structured document database system*, EP'90, Proceeding of the international conference on Electronic Publishing, Document Manipulation & Typography, pages 139-152, Gaithersburg, Maryland, September 1990.

[75] Takamune Kitazawa & Mitsutoshi Hayata, *Intelligent Document Generation System for Construction Planning*, IEEE Software, September 1992.

[76] Donald E. Knuth, *Mathematical Typography*, TEX and METAFONT: New Directions in Typesetting, Chap. 1, Digital Press and the American Mathematical Society, December 1979.

[77] Donald E. Knuth, *The TEXbook*, Addison-Wesley, 1984.

[78] Eila Kuikka & Martti Penttonen, *Transformation of structured documents with the use of grammar*, EP'94, Proceeding of the fifth International Conference on Electronic Publishing, Document Manipulation, and Typography, pages 373-383, Darmstadt, Germany, 13-15 April 1994.

[79] Leslie Lamport, *LATEX: A Document Preparation System*, Addison-Wesley, 1985.

[80] M-C. Landau, F. Sillion & F. Vichot, *Exoseme: A Document Filtering System Based on Conceptual Graphs*, ICCS'93, First International Conference on Conceptual Structures, Quebec City, Canada, August 1993.

[81] M.E. Lesk, *Typing documents on the UNIX system: Using the -ms macros with TROFF and NROFF*, Internal Memorandum, Bell Laboratories, October 1976.

[82] David M. Levy, *Document reuse and document systems*, EP'94, Proceeding of the fifth International Conference on Electronic Publishing, Document Manipulation, and Typography, pages 339-348, Darmstadt, Germany, 13-15 April 1994.

[83] David M. Levy, Daniel C. Brotsky& Kenneth R. Olson, *Formalizing the Figural: Aspects of a Foundation for Document Manipulation*, Proceeding of the ACM Conference on Document Processing Systems, Santa Fe, New Mexico, ACM Press, pages 145-151, December 5-9, 1988.

[84] Ian A. Macleod, Brent Nordin, David T. Barnard & Doug Hamilton, *A Framework for Developping SGML Applications*, EP'92, Proceeding of the fourth International Conference on Electronic Publishing, Document Manipulation, and Typography, pages 53-63, Lauzanne, Switzerland, 1992.

[85] S.E. Madnick & A. Moulton, *SCRIPT: An on-line manuscript processing system*, IEEE Transactions on Engineering Writing and Speech, Institute of Electronic and Electrical Engineers, 1968.

[86] Gérald Masini, Amedeo Napoli, Dominique Colnet, Daniel Léonard & Karl Tombre, *Les langages à objets*, InterEdition, 1989.

[87] Philippe Maurice, *L'architecture d'un document électronique: concepts et applications*, L'Echo des RECHERCHES, no 130, pages 15-24, 1987.

[88] Philippe Maurice, *Architecture de documents ODA, éléments de stratégie bureautique*, Publication du CNET, 1989.

[89] Claude Maymard & Eric Meynieux, *Les normes de documents : l'état de l'art*, Génie logiciel & systèmes experts, no 21, Décembre 1990.

[90] B. Mélèse, *MENTOR-RAPPORT, Manipulation de textes structurés sous MENTOR*, Actes de cours et séminaires INRIA, AUSSOIS, pages 205-236, Avril 1983.

[91] B. Mélèse, *Edition de documents multi-langages sous Mentor-Rapport*, T.S.I., vol. 4, no 5, pages 267-277, 1986.

[92] Bertrand Meyer, *Introduction to the Theory of Programming Languages*, PRENTICE Hall, 1990.

[93] Bertrand Meyer, *Conception et programmation par objets*, InterEditions, 1991.

[94] V. Mital & T. D. Gedeon, *A Neural Network Integrated with Hypertext for Legal Document Assembly*, Proceeding of the 55e Hawai International Conference on System Science, vol. 4: Information Systems, pages 533-539, Hawai, 1992.

[95] E. Morcos-Chounet & A. Conchon, *PPML: A general formalism to specify pretty-printing*, Proceeding of IFIP Congress, Dublin, pages 583-590, 1986.

[96] Robert A. Morris, *Is what you see enough to get ? A description of the Interleaf publishing system*, PROTEXT II: Proceeding of the Second International Conference on Text Processing Systems, pages 56-81, Boole Press, October 1985.

[97] T. G. Muchnik, *A language-adjustable structure editor with means of semantic control*, Programming and computer software, vol. 16, no 2, page 39-46, 1991.

[98] Makoto Murata, *An object-odiented interpretation of ODA*, WOODMAN'89, Workshop on Object Oriented Document Manipulation, pages 91-100, Rennes, France, Mai 1989.

[99] H.A. Muscate, *ODA Document editing in office systems*, WOODMAN'89, Workshop on Object Oriented Document Manipulation, pages 68-78, Rennes, France, Mai 1989.

[100] M. Nanard, J. Nanard & J Falgueirettes, *Top Down or Bottom Up Approach for Document Structuration*, PROTEXT I, Dublin, October 1985.

[101] Marc Nanard & Jocelyne Nanard, *MacWeb, un outil pour élaborer les documents*,WOODMAN'89, Workshop on Object Oriented Document Manipulation, pages 18-29, Rennes, France, Mai 1989.

[102] A. Nazarenko, *Representing natural language causality in conceptual graphs: the higher order conceptual relation problem*, ICCS'93, First International Conference on Conceptual Structures, Quebec City, Canada, August 1993.

[103] C. Nicholas, J. Mayfield & J. Sasaki, *Category Theory and Natural Language Understanding as Models for Document Processing*, PODP'92, Proceeding of the first international conference on Principles of Document Processing, Washington, October 1992.

[104] B. Nordin, D. T. Barnard & I. A. Macleod, *A Critique of the Standard Generalized Markup Language (SGML)*, Technical Report 91-308, Department of Computing and Information Science, Queen's University, 1991.

[105] Ryuichi Ogawa, Hiroaki Harada & Asao Kaneko, *Scenario-based Hypermedia : A Model and a System*, Hypertext : Concepts, Systems and Applications, Proceeding of the First European Conference on Hypertext, pages 38-51, INRIA, France, November 1990.

[106] Joe F. Osanna, *NROFF User's Book - Second Edition*, Bell Laboratories, 1974.

[107] Claude Pasquier, *Génération de documents personnalisés*, Rapport annuel d'activité 1990, pages 54-58, document interne Informatique CDC, 1990.

[108] Claude Pasquier, *La génération de documents personnalisés*, publication interne dans CYCLOPE, le journal d'Informatique CDC, Juin 1991.

[109] Claude Pasquier, *BIBLE : a system for Design and Management of Context-Controlled Documents*, PODP'92, Proceeding of the first international conference on Principles of Document Processing, Washington, October 1992.

[110] M. T. Pazienza & P. Velardi, *Knowledge acquisitionfor Natural Language Processing: Tools and Methods*, Proceedings of the International Conference on Current Issues in Computational Linguistics, Penang, University of Malaysia, June 1991.

[111] M. Peterlongo, *Object oriented environment of ODA editing*, WOODMAN'89, Workshop on Object Oriented Document Manipulation, pages 123-134, Rennes, France, Mai 1989.

[112] *ODA Document Application Profile Q111 (ENV 41509) - Processable and Formatted Documents - Basic Character Content*, August 1989.

[113] *ODA Document Application Profile Q112 (ENV 41510) - Processable and Formatted Documents - Extended Mixed Mode*, August 1989.

[114] *ODA Document Application Profile Q113 - Processable and Formatted Documents - Enhanced Mixed Mode*, August 1989.

[115] *Information systems interconnection; Office Document Architecture (ODA); Processable and Layout Independent Documents; Simple Messaging Profile*, March 1989.

[116] V. Quint, I. Vatton & H. Bedor, *Le système Grif*, T.S.I., pages 337-341, 1986.

[117] Vincent Quint, Irène Vatton, *Grif: an interactive system for structured document manipulation*, Text processing and document manipulation (proceedings of the international conference, university of Nottingham, Cambridge University Press, April 1986.

[118] Vincent Quint, Irène Vatton, *Modularity in structured documents*, WOODMAN'89, Workshop on Object Oriented Document Manipulation, pages 170-177, Rennes, France, Mai 1989.

[119] Vincent Quint, *Systems for the manipulation of structured documents*, STRUCTURED DOCUMENTS, Cambridge University Press, pages 39-74, 1989.

[120] Vincent Quint, Marc Nanard & Jacques André, *Towards Documents Engineering*, Publication Interne no 536, IRISA, Mai 1990.

[121] Darrell R. Raymond, *Flexible Text Display with Lector*, Computer, vol. 25, no. 8, pages 49-60, August 1992.

[122] F. Rechenmann, *SHIRKA : système de gestion de bases de connaissances centrées-objet, manuel de référence*, INRIA/ARTEMIS, Grenoble, 1988.

[123] R. B. Roberts & I. P. Goldstein, *The FRL Manual*, AI Memo 409, AI Lab, MIT, Cambridge, Massachusetts, 1977.

[124] Peter J. Robinson, *Open Document Architecture (ODA)*, ELEDIS Journal, Bi-monthly Report on Electronic Data Interchange Systems, no 5, pages 4-7, Nov/Dec 1990.

[125] Cécile Roisin & Irène Vatton, Merging logical and physical structures in documents, EP'94, Proceeding of the fifth International Conference on Electronic Publishing, Document Manipulation, and Typography, pages 327-337, Darmstadt, Germany, 13-15 April 1994.

[126] Elmer Sandvad, Hypertext in an Object-Oriented Programming Environment, WOODMAN'89, Workshop on Object Oriented Document Manipulation, pages 30-41, Rennes, France, Mai 1989.

[127] Richard Southall, *Interface between the designer and the document*, STRUCTURED DOCUMENTS, Cambridge University Press, pages 119-132, 1989.

[128] Richard Southall, *Presentation Rules and Rules of Composition in the formatting of Complex Text*, EP'92, Proceeding of the fourth International Conference on Electronic Publishing, Document Manipulation, and Typography, pages 275-290, Lauzanne, Switzerland, 1992.

[129] Richard M. Stallman, *Emacs, The Extensible, Customizable Self-Documenting Display Editor*, Proceeding of the ACM SIGPLAN SIGOA Symposium on Text Manipulation, Portland Oregon, June 8-10, 1981, SIGPLAN Notices ver. 16, no 6, pages 147-156, 1981.

[130] G. L. Steel Jr, *COMMON LISP : The Language*, Digital Press, 1984.

[131] B. Stroustrup, *The C++ Programming Language*, Addison-Wesley Series in Computer Science, 1986.

[132] Said TAZI, *Representation de connaissances structurelles du texte naturel pour la conception d'un editeur intelligent*, Thèse soutenue à l'université Paul-Sabatier de Toulouse, 1985.

[133] T. Teitelbaum & T. Reps, *The Cornell Program Synthesizer : a syntax directed programming environment*, Communication of the ACM, Vol. 24, No 9, pages 563-573, September 1981.

[134] George Towner, *Auto-Updating as a Technical Documentation Tool*, Proceeding of the ACM Conference on Document Processing Systems, December 5-9, 1988, Santa FE, New Mexico", ACM Press, pages 31-36, 1988.

[135] B. Uttley, *Waterloo Script*, Private Communication of Waterloo University, 1973.

[136] Ray Valdès, *Text Editors: Algorithms and Architectures*, Dr. Doob's journal, pages 38-43, April 1993

[137] L. van Dam & E. van Loenen, *A Programmer's Interface to SGML Applications*, Technical Report, CERN, Geneva, 1989.

[138] F. Velez, *La prise en compte de documents structurés dans un SGBD : aspects modèle, langage et architecture du système*, Nouvelles perspectives des bases de données, éditions Eyrolles, pages 35-56, 1986.

[139] Anne-Marie Vercoustre, *Le méta-décompilateur Pretty sous Mentor*, Rapport Technique INRIA No 62, Décembre 1985.

[140] Anne-Marie Vercoustre, *Structured Editing - Hypertext Approach: Cooperation and Complementarity*, EP'90, Proceeding of the international conference on Electronic Publishing, Document Manipulation & Typography, pages 65-78, Gaithersburg, Maryland, September 1990.

[141] Janet H. Walker, *The Document Editor : A Support Environment for Preparing Technical Documents*, Proceeding of the ACM SIGPLAN SIGOA Symposium on Text Manipulation, Portland Oregon, June 8-10, 1981, SIGPLAN Notices ver. 16, no 6, pages 44-50, 1981.

[142] Janet H. Walker, The Role of Modularity in Document Authoring Systems, Proceeding of the ACM Conference on Document Processing Systems, Santa Fe, New Mexico, ACM Press, pages 117-124, December 5-9 1988.

[143] J. Weidert, *Structuring tools for the design of man-machine dialogs*, Proceeding of International Conference on research and trends in document preparation systems, Lausanne, Switzerland, page 5-8, February 1981.

[144] Alain Zarli, *Spécifications par attributs sémantiques pour la génération d'éditeurs structurés graphiques incrémentaux*, Thèse soutenue à l'université de Nice Sophia-Antipolis, 14 Décembre 1990.

Liste des figures

I Représentation structurée d'une instruction *IF* 16

II Représentation structurée d'une figure d'un document 16

III Comparaison des formalismes définissant documents structurés et programmes 20

IV La génération des documents dans BIBLE 23

1.1 Représentation révisable et imprimable 30

1.2 Un document TROFF et le résultat correspondant 30

1.3 Déclaration d'une macro TROFF 31

1.4 Caractéristiques d'un boîte dans T_EX 31

1.5 L'emboîtement des éléments propre à un formatage contextuel 32

1.6 Un document L^AT_EX et le résultat correspondant 32

1.7 L'mage concrète d'une page d'un document 35

1.8 Représentation arborescente d'un document 36

1.9 Représentation d'un document sous forme de graphe 37

1.10 Représentation d'un document possédant une structure arborescente principale 39

1.11 Représentation d'un modèle de document par une GSA 41

1.12 Exemple de règles de sémantique statique 42

1.13 L'instanciation d'un objet générique composé 44

1.14 Les trois relations utilisées pour la modélisation d'un document 45

1.15 Représentation d'un objet *Automobile* 48

1.16 Représentation des objets *Automobile* et *NouvAuto* 48

1.17 Représentation d'un prototype *Document* 49

1.18 Représentation des prototypes *Document* et *Document2* 49

1.19 Le partage d'une sous-structure commune 50

1.20 Représentation des structures *Document* et *Document2* en utilisant des liens typés 51

1.21 Structure physique obtenue par dérivation d'une structure logique 52
1.22 Un cas où la structure physique ne peut être obtenue par dérivation de la structure logique 53
1.23 Elaboration de la présentation en utilisant un modèle physique 54
1.24 La DTD *Doc* 55
1.25 Exemple d'un document balisé 57
1.26 Tableau comparatif SGML/ODA 62

2.1 L'opération de copier/coller 66
2.2 Les transformations nécessaires à une opération de copier/coller 67
2.3 Structuration d'un élément *Personne* 68
2.4 Structuration d'un élément *Author* 68
2.5 Visualisation par application de règles sur la structure abstraite 70
2.6 Exemple d'une règle de décompilation PPML 73
2.7 Les règles Typol permettant de disposer une image sur une page 75
2.8 La phase de transformation de DSSSL 77
2.9 La transformation statique dans Grif 78
2.10 La transformation statique dans SYNDOC 79
2.11 La phase de transformation définie par A. Brown 80
2.12 Exemple d'une interface applicative à SGML 84
2.13 Exemple d'une table d'actions 85
2.14 Un document SGML avec des parties variables 86
2.15 Comparaison entre BIBLE et un générateur de rapport 88
2.16 Représentation d'une composition variable 89
2.17 Le parallèle entre édition classique et gabariage + génération 90

3.1 Exemple de conditionnement d'un élément SGML 94
3.2 Exemple de conditionnement d'un contenu 94
3.3 DTD, EDTD et Gabarit 96
3.4 Le partage du corps entre deux modèles 102
3.5 Situation après la modification du paragraphe 103
3.6 Clonage puis modification du corps 103

3.7 Description SGML du corps de *Lettre* 104
3.8 Description SGML du corps d'*Article* 104
3.9 Tous les éléments utilisés dans *Article* & *Lettre* définis par des types 105
3.10 Reconstitution de la définition d'*Article* grâce aux liens indirects 106
3.11 Exemple d'un graphe représentant les relations de délégation entre les modèles 108
3.12 Exemple de représentation de deux modèles 109
3.13 Transformations sur les instances suite à un ajout dans un modèle 110
3.14 Transformation sur les instances suite à un changement d'indicateur d'occurence 111

4.1 Les différents modules composant l'application BIBLE 114
4.2 Les classes utilisées pour représenter les modèles 116
4.3 Définition de la classe *GUIDE* 117
4.4 Définition de la fonction *REFERENCE* 118
4.5 Définition de la fonction *CONTENT-TYPE* 119
4.6 Définition de la fonction *COMPOSED-OF* 120
4.7 Les classes graphiques définies en CLOS 122
4.8 Représentation des modèles par une hiérarchie 123
4.9 Les différentes opérations possibles sur les éléments d'un modèle 123
4.10 L'ajout d'un composant dans un élément 124
4.11 Le modèle *Lettre* sur l'éditeur 124
4.12 La sortie SGML du modèle *Lettre* 125
4.13 Définition de la méthode *ADD-SUBORDINATE* pour un type d'élément 125
4.14 Définition de la méthode *ADD-SUBORDINATE* pour un élément composé 126
4.15 Les classes utilisées pour représenter les éléments physiques 127
4.16 L'éditeur de boîtes 128
4.17 Illustration des attributs de positionnement 129
4.18 Le choix des attributs typographiques 130
4.19 L'éditeur de modèle physique 131
4.20 Le browser de boîtes 132
4.21 Détermination de la position des boîtes 134

4.22 L'EDTD correspondant à la DTD de la figure 4.12 139
4.23 Le conditionnement de l'élément *VosRef* 140
4.24 Le conditionnement de l'élément *Dest* 141
4.25 La définition de la formule d'appel 142
4.26 Edition des conditions associées à un gabarit 143
4.27 Exemple d'une condition définie par une combinaison de prédicats 144
4.28 Exemple d'un fichier de données 145
4.29 Un document SGML généré par le module d'assemblage 145
4.30 Exemple d'un document final après formatage 146

Index

#PCDATA, 55

A

ATTLIST, 55
attribut, 135
  typographique, 135
  de dimensionnement, 135
  de positionnement, 136

B

balisage,
  déclaratif, 30
  procédural, 29
  en SGML, 56
Boîte, 31, 127, 130, 134
BIBLE, 21, 87, 93, 107, 115

C

clonage, 103
clonage-lié, 106
CLOS, 46, 113
composition, 43
  héritage, 44, 45, 90, 101
  lien typés, 51
condition, 99, 143
conditionnement, 89, 96, 99
contexte, 80
contraintes, 127, 135
copie différentielle, 24
couper/coller, 66

D

DAP, 58
décompilation, 71
  avec modèle physique, 53
  contextuelle, 73
  libre, 72
délégation, 46, 90, 106, 112
document
  actif, 82
  assemblage, 144
  formatage, 146
  structuré, 15
donnée
  conception dirigée, 13, 93, 144
DSSSL, 54, 77
  GLTP, 77
DTD, 55, 94, 137

E

éditeur
  de boîte, 127
  de gabarit, 140
  de modèle, 122
  de présentation, 126
  structurel, 14
édition, 66, 89
  multicouches, 22
EDTD, 94, 137
ELEMENT, 55
ENTITY, 55

F

formatage, 146
frame, 47

G

gabariage, 89
gabarit, 88, 96, 99, 112
  editeur, 140
génération, 23, 85, 87, 89, 144
générique
  structure, 59
GIGAS, 73
GLTP, 77
grammaire
  attribuée, 42, 74
  de syntaxe abstraite, 15, 40
Grif, 78, 83, 115, 132, 140, 146

H

Haskell, 115, 137
héritage, 44
  conflit, 45
  de la composition, 90, 101, 118, 123

I

instance, 44, 109
instanciation, 43

L

langage
  de frame, 47
  de programmation, 15, 17
  naturel, 19
  P, 95, 132
lien typé, 51, 111

M

macro-instruction, 30
modèle, 24, 40, 90, 116
  éditeur, 122
  héritage, 118
  logique, 40
  physique, 53
modularité, 50

N

norme
  DSSSL, 54, 77
  ODA, 58, 60
  SGML, 54, 60, 77, 81, 83, 86

O

ODA, 58, 60
  DAP, 58
orienté-objet, 43, 116

P

P
  langage, 115, 132
partage, 102
PPML, 72
prédicat, 100
présentation, 18, 52
  éditeur, 126
profil
  DAP, 58
  document, 60
programme, 15, 23
prototype, 47, 90, 101

R

représentation
  abstraite, 34
  arborescente, 35
  concrète, 52
  globale, 61
  graphique, 69
  par graphe orienté, 38
  logique, 17, 61

mixte, 38
objet, 61
imprimable, 30
physique, 18
révisable, 30

S

sémantique
naturelle, 19
statique, 41
translationnelle, 76
SGML, 54, 60, 77, 81, 83, 86
#PCDATA, 56
DTD, 55, 94, 137
ATTLIST, 55
ELEMENT, 55
ENTITY, 55
spécialisation, 44
spécifique
structure, 59
structure, 14, 65, 85
abstraite, 34
générique, 59
logique, 40, 59
physique, 52, 59
spécifique, 59
syntaxique, 40
style, 32
dans ODA, 59

T

TEX, 31
transformation, 24, 109
dynamique, 66, 70
structurelle, 67, 70, 76
statique, 69, 71, 78, 87
TROFF, 30
Typol, 74

V

variable, 86, 88, 96
visualisation, 71

W

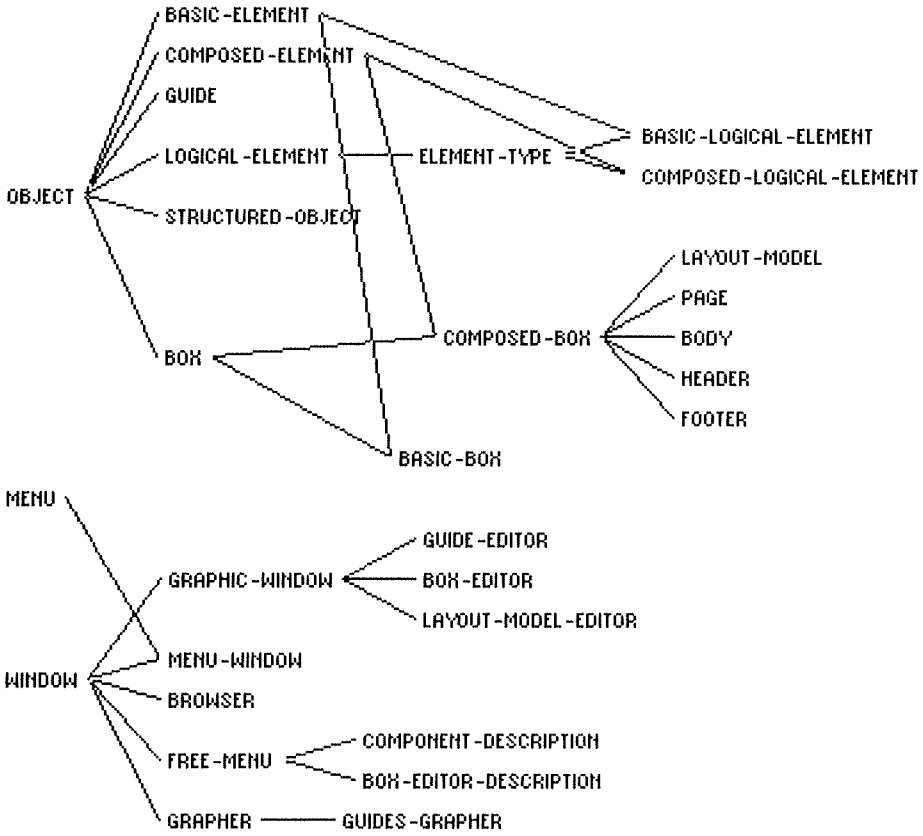
WYSIWYG, 33, 60, 63, 113

Annexes

Annexe A

Les développements en CLOS

Annexe A.1
Les classes utilisées



Annexe A.2
Une partie des sources CLOS

```
(DEFMETHOD ADD-SUBOBJECT ((OBJECT COMPOSED-BOX)
                          (WINDOW LAYOUT-MODEL-EDITOR)
                          REGION)

;;; Ajout d'un sous objet dans l'objet structuré

   (DECLARE (SPECIAL *COPY-BITMAP* *COPIED-OBJECT* IL:BACKGROUNDFNS))
   (PROG* ((INITIAL-POS (GET-CURSOR-POSITION WINDOW))
           (OLD-POS (COPY-LIST INITIAL-POS))
           (NEW-POS (COPY-LIST INITIAL-POS)))
          (WHEN (EQUAL (CAR IL:BACKGROUNDFNS)
                       'WAIT-FOR-COPY)
                (RETURN))                                    ; On ne fait rien si on est en
attente de copie
          BITBLT *COPY-BITMAP* 0 0 (SLOT-VALUE WINDOW 'IL-WINDOW)
                 (- (CAR INITIAL-POS)
                    8)
                 (- (CDR INITIAL-POS)
                    2)
                 NIL NIL 'IL:INPUT 'IL:INVERT)               ; Affiche le symbole COPY
          (SETQ *COPIED-OBJECT* NIL)                         ; Efface les objets du
presse-papier
          (LOOP WHILE KEYDOWNP 'IL:LEFT))                  ; Attend que le bouton gauche soit
relaché
          (EVAL
           'PUTD
             'WAIT-FOR-COPY
             '(LAMBDA
               NIL
               (WHEN MOUSESTATE (OR IL:LEFT IL:MIDDLE IL:RIGHT))
                                                           ; Attend qu'un bouton soit
actionné
                   PUTD 'WAIT-FOR-COPY
                        '(LAMBDA NIL (UNLESS KEYDOWNP 'IL:COPY)
                                                           ; Attend que copy soit relaché

                                                ;; Ici, on attend un cycle, pour être sur que
l'objet sélectionné dans le browser est disponible dans *COPIED-OBJECT*

                                               PUTD 'WAIT-FOR-COPY
                                                    '(LAMBDA NIL (POP IL:BACKGROUNDFNS)
                                                           ; Enlève la fonction de
BACKGROUNDFNS
                                                      (APPLY #'ADD-SUBOBJECT2
                                                             '(,OBJECT ,WINDOW ,REGION
                                                                       ,OLD-POS))
                                                           ; Et lance la fonction qui va
réaliser la copie sur la fenêtre
                                                      )))))))))
          (PUSH 'WAIT-FOR-COPY IL:BACKGROUNDFNS)           ; On met la fonction dans la
liste BACKGROUNDFNS
      ))

(DEFMETHOD ADD-SUBOBJECT2 ((OBJECT COMPOSED-BOX)
                           (WINDOW LAYOUT-MODEL-EDITOR)
                           REGION POSITION)

;;; Réalise la copie de l'objet situé dans *COPIED-OBJECT*, à l'intérieur de la fenêtre, à la
position POSITION

   (DECLARE (SPECIAL *COPIED-OBJECT* *COPY-BITMAP*))
   BITBLT *COPY-BITMAP* 0 0 (SLOT-VALUE WINDOW 'IL-WINDOW)
          (- (CAR POSITION)
             8)
```

```
          (- (CDR POSITION)
             2)
          NIL NIL 'IL:INPUT 'IL:INVERT)                    ; Efface le symbole COPY
    (WHEN (EQUAL (CLASS-OF *COPIED-OBJECT*)
                 (FIND-CLASS 'BOX))
        (PROG* ((SCALE (GETWINDOWPROP WINDOW 'PAGE-SCALE))
                (NEW-OBJECT-REGION (MAKE-REGION :WIDTH (ROUND (SLOT-VALUE *COPIED-OBJECT*
'WIDTH)
                                                              SCALE)
                                                :HEIGHT
                                                (ROUND (SLOT-VALUE *COPIED-OBJECT* 'HEIGHT)
                                                       SCALE)))
                (REGIONS (LOOP FOR REG IN (LOOP FOR REG-INFO IN (GETWINDOWPROP WINDOW
'REGIONS)
                                                COLLECT
                                                (THIRD REG-INFO))
                               WHEN
                               SUBREGIONP REGION REG)
                               UNLESS
                               (EQUAL REGION REG)
                               COLLECT REG)                ; Liste des régions incluses
dans l'objet
                )
                (FREE-REGION (REGION-FREE-SPACE REGION POSITION REGIONS))
                                                           ; FREE-REGION contient la région
maximale disponible autour de POSITION
                (FREE-REGION-CENTER (REGION-CENTER FREE-REGION))
                                                           ; Position centrale de la région
libre
                (NEW-OBJECT-CENTER (REGION-CENTER NEW-OBJECT-REGION))
                                                           ; Centre
                )
               (SETF (REGION-LEFT NEW-OBJECT-REGION)
                     (- (CAR FREE-REGION-CENTER)
                        (ROUND (REGION-WIDTH NEW-OBJECT-REGION)
                               2)))
               (SETF (REGION-BOTTOM NEW-OBJECT-REGION)
                     (- (CDR FREE-REGION-CENTER)
                        (ROUND (REGION-HEIGHT NEW-OBJECT-REGION)
                               2)))                        ; On centre l'objet à
l'intérieur de la zone libre
               (LET* ((TEST-REGION (COPY-LIST NEW-OBJECT-REGION))
                      (DELTA-POS (POSITION-SUBTRACT POSITION FREE-REGION-CENTER))
                      (INCX (IF (PLUSP (CAR DELTA-POS))
                                1
                                -1))
                      (INCY (IF (PLUSP (CDR DELTA-POS))
                                1
                                -1)))
                     (LOOP FOR I FROM 1 TO (ABS (CAR DELTA-POS))
                           DO
                           (SETQ NEW-OBJECT-REGION (COPY-LIST TEST-REGION))
                           (INCF (REGION-LEFT TEST-REGION)
                                 INCX)
                           ALWAYS
                           SUBREGIONP FREE-REGION TEST-REGION))
                     (SETQ TEST-REGION (COPY-LIST NEW-OBJECT-REGION))
                     (LOOP FOR I FROM 1 TO (ABS (CDR DELTA-POS))
                           DO
                           (SETQ NEW-OBJECT-REGION (COPY-LIST TEST-REGION))
                           (INCF (REGION-BOTTOM TEST-REGION)
                                 INCY)
                           ALWAYS
                           SUBREGIONP FREE-REGION TEST-REGION)))
                                                           ; NEW-OBJECT-REGION contient
maintenant la région la plus proche possible de position
               (UNLESS SUBREGIONP FREE-REGION NEW-OBJECT-REGION)
                   (PRINT-ERRORS '("The object cannot be placed at the specified position"))
                   (RETURN))
               (LET ((ABSOLUTE-POSITION (POSITION-MULTIPLY (POSITION-SUBTRACT (REGION-POSITION
NEW-OBJECT-REGION
                                                                              )
                                                              (REGION-POSITION REGION))
                                                       SCALE))        ; Position du nouvel objet par
rapport à l'origine de l'objet courant
```

```
                    )
                   (PUSH '(ADD ,*COPIED-OBJECT* ,ABSOLUTE-POSITION)
                         (SLOT-VALUE OBJECT 'SUBORDINATES)))
             (DRAW-REGION WINDOW NEW-OBJECT-REGION :OPERATION 'IL:PAINT)
             (REDISPLAY (GETWINDOWPROP WINDOW 'MODEL)
                    WINDOW))))

(DEFMETHOD ADD-SUBORDINATE ((OBJECT COMPOSED-LOGICAL-ELEMENT)
                            COMPONENT &OPTIONAL OTHER)

;;; Ajoute le composant COMPONENT dans la composition de OBJECT

   (SETF (SUBORDINATES OBJECT)
         (APPEND (SUBORDINATES OBJECT)
                 '((ADD ,@COMPONENT))))
   (LET ((POS (1+ (LENGTH (REORDER-LIST OBJECT)))))
        (LOOP FOR X IN (CONS OBJECT (FO-DNIK* OBJECT))
              DO
              (WHEN (REORDER-LIST X)
                  (SETF (REORDER-LIST X)
                        (LOOP FOR Y IN (REORDER-LIST X)
                              COLLECT
                              (IF (>= Y POS)
                                  (1+ Y)
                                  Y)))
                  (SETF (REORDER-LIST X)
                        (APPEND (REORDER-LIST X)
                                (LIST POS)))))))

(DEFMETHOD ADD-SUBORDINATE ((OBJECT ELEMENT-TYPE)
                            (MODEL GUIDE)
                            &OPTIONAL COMPONENT)

;;; Ajoute dans la composition de OBJECT, utilisé dans le contexte de MODEL, le composant
COMPONENT

   (LET ((CURRENT-OBJECT (CLONE-IF-NECESSARY OBJECT MODEL)))
        (ADD-SUBORDINATE CURRENT-OBJECT COMPONENT)))

(DEFMETHOD BUTTON-EVENT ((WINDOW LAYOUT-MODEL-EDITOR))

;;; Réagit au clic souris en activant la fonction correspondant à la région ou se trouve le
curseur

   (LET* ((SELECTED-REGION (WHERE-IS-CURSOR? WINDOW))
          (SELECTED-OBJECT (SECOND SELECTED-REGION)))
         (FORMAT IL:PROMPTWINDOW "~S" SELECTED-OBJECT)
         (COND
            (MOUSESTATE IL:LEFT)
             (CASE SELECTED-OBJECT
                 ((NIL) (LOOP WHILE KEYDOWNP 'IL:LEFT)))
                 (T (APPLY (FIFTH SELECTED-REGION)
                           '(,SELECTED-OBJECT ,WINDOW ,(THIRD SELECTED-REGION)
                                              ,WINDOW)))))
            (MOUSESTATE IL:MIDDLE)
             (WHEN (PCL::STANDARD-CLASS-P (CLASS-OF 'BOX))
                   (DISPLAY-MENU SELECTED-OBJECT WINDOW))))))

(DEFMETHOD BUTTON-EVENT ((WINDOW BOX-EDITOR))

;;; Réagit au clic souris en activant la fonction correspondant à la région ou se trouve le
curseur

   (LET* ((SELECTED-REGION (WHERE-IS-CURSOR? WINDOW))
          (SELECTED-OBJECT (SECOND SELECTED-REGION)))
         (FORMAT IL:PROMPTWINDOW "~S" SELECTED-OBJECT)
         (COND
            (MOUSESTATE IL:LEFT)
             (CASE SELECTED-OBJECT
                 ((NIL) (LOOP WHILE KEYDOWNP 'IL:LEFT)))
                 (HORIZONTAL-SCALE (CHANGE-H-SCALE WINDOW))
                 (VERTICAL-SCALE (CHANGE-V-SCALE WINDOW))
                 ((TOP-MAX TOP-MAIN TOP-MARGIN BOTTOM-MARGIN) (CHANGE-HORIZONTAL-LINES WINDOW
                                                                SELECTED-OBJECT))
                 ((RIGHT-MAX RIGHT-MAIN RIGHT-MARGIN LEFT-MARGIN) (CHANGE-VERTICAL-LINES
WINDOW
```

```
                                                                    SELECTED-OBJECT))))
          (MOUSESTATE IL:MIDDLE)
           (WHEN (PCL::STANDARD-CLASS-P (CLASS-OF SELECTED-OBJECT))
                 (DISPLAY-MENU SELECTED-OBJECT WINDOW))))))

(DEFMETHOD CHANGE-CONSTRUCTOR ((OBJECT COMPOSED-LOGICAL-ELEMENT)
                               CONSTRUCTOR &OPTIONAL OTHER)

;;; Remplace le constructeur de OBJECT par CONSTRUCTOR

   (SETF (CONSTRUCTOR OBJECT)
         CONSTRUCTOR))

(DEFMETHOD CHANGE-CONSTRUCTOR ((OBJECT ELEMENT-TYPE)
                               (MODEL GUIDE)
                               &OPTIONAL CONSTRUCTOR)

;;; remplace le constructeur de OBJECT utilisé dans le contexte de MODEL,  par CONSTRUCTOR

   (CHANGE-CONSTRUCTOR (CLONE-IF-NECESSARY OBJECT MODEL)
          CONSTRUCTOR)
   (WHEN (EQUAL CONSTRUCTOR 'SEQUENCE)
         (REORDER-SUBORDINATES OBJECT MODEL)))

(DEFMETHOD CHANGE-CONTENT ((OBJECT BASIC-LOGICAL-ELEMENT)
                           CONTENT &OPTIONAL OTHER)

;;; Remplace le contenu de OBJECT par CONTENT

   (SETF (CONTENT OBJECT)
         CONTENT))

(DEFMETHOD CHANGE-CONTENT ((OBJECT ELEMENT-TYPE)
                           (MODEL GUIDE)
                           &OPTIONAL CONTENT)

;;; remplace le contenu de OBJECT utilisé dans le contexte de MODEL,  par CONTENT

   (CHANGE-CONTENT (CLONE-IF-NECESSARY OBJECT MODEL)
          CONTENT))

(DEFMETHOD CHANGE-H-SCALE ((WINDOW BOX-EDITOR))

;;; Permet de changer interactivement l'échelle horizontale

   (DECLARE (SPECIAL GRAYSHADE1))
   (WITH-SLOTS (IL-WINDOW)
       WINDOW
     (LET* ((WINDOW-WIDTH (INTERIOR-WIDTH WINDOW))
            (XPOS (POSITION-XCOORD (GET-CURSOR-POSITION WINDOW)))
            (OLD-XPOS XPOS)
            (NEW-XPOS XPOS)
            (H-SCALE-IMAGE BITMAPCREATE WINDOW-WIDTH 18))
            (H-SCALE (GETWINDOWPROP WINDOW 'H-SCALE)))
           (DRAW-LINE WINDOW (CONS XPOS 0)
                  (CONS XPOS 18)
                  :WIDTH 3 :OPERATION 'IL:PAINT)
           BITBLT IL-WINDOW 0 0 H-SCALE-IMAGE 0 0 WINDOW-WIDTH 18 'IL:INPUT 'IL:PAINT)
           (LOOP WHILE KEYDOWNP 'IL:LEFT)
                 DO
                 (SETQ NEW-XPOS (KEEP-IN-LIMITS (POSITION-XCOORD (GET-CURSOR-POSITION
WINDOW))
                                                19 WINDOW-WIDTH))
                 (COND
                    ((NOT (= OLD-XPOS NEW-XPOS))
                     BITBLT H-SCALE-IMAGE 18 0 IL-WINDOW 18 0 (- (MIN XPOS NEW-XPOS)
                                                                 18)
                            18
                            'IL:INPUT
                            'IL:REPLACE)
                     (FILL-REGION WINDOW (LIST (MIN XPOS NEW-XPOS)
                                               3
                                               (ABS (- XPOS NEW-XPOS))
                                               12)
                            :TEXTURE GRAYSHADE1 :OPERATION 'IL:PAINT)
                     BITBLT H-SCALE-IMAGE (MAX XPOS NEW-XPOS)
```

```
                              0 IL-WINDOW (MAX XPOS NEW-XPOS)
                              0
                              (- WINDOW-WIDTH (MAX XPOS NEW-XPOS))
                              18
                              'IL:INPUT
                              'IL:REPLACE)
                        (SETQ OLD-XPOS NEW-XPOS))))
          (PUTWINDOWPROP WINDOW 'H-SCALE (* H-SCALE (/ (- XPOS 18)
                                                       (- NEW-XPOS 18))))
          (COMPUTE-REGIONS (GETWINDOWPROP WINDOW 'LAYOUT-OBJECT)
                 WINDOW)
          (DRAW-H-SCALE WINDOW)
          (DRAW-REGIONS WINDOW))))

(DEFMETHOD CHANGE-HORIZONTAL-LINES ((WINDOW BOX-EDITOR)
                                    POINTED-OBJECT)
   (DECLARE (SPECIAL GRAYSHADE1 GRAYSHADE2))
   (LET* ((YPOS (POSITION-YCOORD (GET-CURSOR-POSITION WINDOW)))
          (OLD-YPOS YPOS)
          (NEW-YPOS YPOS)
          (MAX-REGION (GETWINDOWPROP WINDOW 'MAX-REGION))
          (MAIN-REGION (GETWINDOWPROP WINDOW 'MAIN-REGION))
          (MARGIN-REGION (GETWINDOWPROP WINDOW 'MARGIN-REGION))
          (TOP-MAX-YPOS (REGION-TOP MAX-REGION))
          (TOP-MAIN-YPOS (REGION-TOP MAIN-REGION))
          (TOP-MARGIN-YPOS (REGION-TOP MARGIN-REGION))
          (BOTTOM-MARGIN-YPOS (REGION-BOTTOM MARGIN-REGION))
          (LIMITS)
          (LINE-XPOS)
          (LINE-LENGTH))
         (SETQ LIMITS (CASE POINTED-OBJECT
                            (TOP-MAX (LIST TOP-MAIN-YPOS (INTERIOR-HEIGHT WINDOW)))
                            (TOP-MAIN (LIST TOP-MARGIN-YPOS TOP-MAX-YPOS))
                            (TOP-MARGIN (LIST BOTTOM-MARGIN-YPOS TOP-MAIN-YPOS))
                            (BOTTOM-MARGIN (LIST 20 TOP-MARGIN-YPOS))))
         (SETQ LINE-XPOS (CASE POINTED-OBJECT
                               ((TOP-MAX TOP-MAIN) 20)
                               ((TOP-MARGIN BOTTOM-MARGIN) (REGION-LEFT MARGIN-REGION))))
         (SETQ LINE-LENGTH (1- (CASE POINTED-OBJECT
                                     (TOP-MAX (REGION-WIDTH MAX-REGION))
                                     (TOP-MAIN (REGION-WIDTH MAIN-REGION))
                                     ((TOP-MARGIN BOTTOM-MARGIN) (REGION-WIDTH
MARGIN-REGION)))))
         (SETQ OLD-YPOS (CASE POINTED-OBJECT
                              (TOP-MAX (REGION-TOP MAX-REGION))
                              (TOP-MAIN (REGION-TOP MAIN-REGION))
                              (TOP-MARGIN (REGION-TOP MARGIN-REGION))
                              (BOTTOM-MARGIN (REGION-BOTTOM MARGIN-REGION))))
         (PUT-CURSOR-POSITION WINDOW (MAKE-POSITION :XCOORD (CAR (GET-CURSOR-POSITION
WINDOW))
                                                    :YCOORD OLD-YPOS))
                                                    ; Place le curseur sur la ligne
à bouger
         (LOOP WHILE KEYDOWNP 'IL:LEFT)
               DO
               (SETQ NEW-YPOS (KEEP-IN-LIMITS (POSITION-YCOORD (GET-CURSOR-POSITION WINDOW))
                                              (FIRST LIMITS)
                                              (SECOND LIMITS)))
               (UNLESS (= OLD-YPOS NEW-YPOS)                ; A chaque déplacement du
curseur
                   (MOVE-TO WINDOW (CONS LINE-XPOS OLD-YPOS))
                   (RELDRAW-LINE WINDOW LINE-LENGTH 0 :WIDTH 1 :OPERATION 'IL:ERASE)
                                                    ; Efface la ligne précédente
                   (WHEN (OR (AND (< OLD-YPOS NEW-YPOS)
                                  (EQ POINTED-OBJECT 'BOTTOM-MARGIN))
                             (AND (> OLD-YPOS NEW-YPOS)
                                  (MEMBER POINTED-OBJECT '(TOP-MAX TOP-MAIN TOP-MARGIN))))
                                                    ; Si le déplacement du curseur
se rapproche du centre de la région
                         (DRAW-LINE WINDOW (CONS LINE-XPOS OLD-YPOS)
                                (CONS LINE-XPOS NEW-YPOS)
                                :WIDTH 1 :OPERATION 'IL:ERASE)
                         (DRAW-LINE WINDOW (CONS (+ LINE-XPOS LINE-LENGTH)
                                                 OLD-YPOS)
                                (CONS (+ LINE-XPOS LINE-LENGTH)
                                      NEW-YPOS)
```

```
                                :WIDTH 1 :OPERATION 'IL:ERASE) ; Alors, il faut supprimer les
barres verticales
                        )
                    (CASE POINTED-OBJECT
                        (TOP-MAX (SETF (REGION-HEIGHT MAX-REGION)
                                       (- NEW-YPOS 19)))
                        (TOP-MAIN (SETF (REGION-HEIGHT MAIN-REGION)
                                        (- NEW-YPOS 19)))
                        (TOP-MARGIN (SETF (REGION-HEIGHT MARGIN-REGION)
                                          (- NEW-YPOS (REGION-BOTTOM MARGIN-REGION)
                                             -1)))
                        (BOTTOM-MARGIN
                           (SETF (REGION-HEIGHT MARGIN-REGION)
                                 (- (REGION-TOP MARGIN-REGION)
                                    NEW-YPOS -1))
                           (SETF (REGION-BOTTOM MARGIN-REGION)
                                 NEW-YPOS)))
                    (DRAW-REGION WINDOW MAX-REGION :BORDER 1 :OPERATION 'IL:PAINT :TEXTURE
GRAYSHADE2)
                    (DRAW-REGION WINDOW MAIN-REGION :BORDER 1 :OPERATION 'IL:PAINT)
                    (DRAW-REGION WINDOW MARGIN-REGION :BORDER 1 :OPERATION 'IL:PAINT :TEXTURE
                        GRAYSHADE1)
                    (SETQ OLD-YPOS NEW-YPOS)))
        (PUTWINDOWPROP WINDOW 'MAX-REGION MAX-REGION)
        (PUTWINDOWPROP WINDOW 'MAIN-REGION MAIN-REGION)
        (PUTWINDOWPROP WINDOW 'MARGIN-REGION MARGIN-REGION)
        (PUTWINDOWPROP WINDOW 'REGIONS (COMPUTE-ACTIVE-REGIONS WINDOW))))

(DEFMETHOD CHANGE-PAGE ((WINDOW LAYOUT-MODEL-EDITOR))

;;; Echange la page affichée sur l'éditeur

   (LET* ((MENU-WINDOW (CAR (ATTACHEDWINDOWS WINDOW)))
          (CURRENT-PAGE (GETWINDOWPROP WINDOW 'CURRENT-PAGE))
          (ITEMS (APPEND '(("Ok" OK)
                           ("Abort" ABORT)
                           ("Export P" EXPORT-P))
                         (IF (ZEROP CURRENT-PAGE)
                             '(("    First Page   " CHANGE-PAGE))
                             '(("    Other Pages   " CHANGE-PAGE))))))
        (SETF (IL:FETCH IL:ITEMS IL:OF (SLOT-VALUE MENU-WINDOW 'IL-MENU))
              ITEMS)                                        ; Change la page
        MENUWRESHAPEFN (SLOT-VALUE MENU-WINDOW 'IL-WINDOW))
        (PUTWINDOWPROP WINDOW 'CURRENT-PAGE (IF (ZEROP CURRENT-PAGE)
                                                1
                                                0))
        (CLEARW WINDOW)
        (REDISPLAY (GETWINDOWPROP WINDOW 'MODEL)
               WINDOW)))

(DEFMETHOD CHANGE-SUBORDINATE ((OBJECT COMPOSED-LOGICAL-ELEMENT)
                               COMPONENT NEW-COMPONENT &OPTIONAL OTHER)

;;; remplace dans la composition de OBJECT, le composant COMPONENT par NEW-COMPONENT

   (SETF (SUBORDINATES OBJECT)
         (APPEND (SUBORDINATES OBJECT)
                 '((REPLACE ,COMPONENT ,@NEW-COMPONENT)))))

(DEFMETHOD CHANGE-SUBORDINATE ((OBJECT ELEMENT-TYPE)
                               (MODEL GUIDE)
                               COMPONENT &OPTIONAL NEW-COMPONENT)

;;; remplace dans la composition de OBJECT utilisé dans le contexte de MODEL, le composant
COMPONENT par NEW-COMPONENT

   (CHANGE-SUBORDINATE (CLONE-IF-NECESSARY OBJECT MODEL)
          COMPONENT NEW-COMPONENT))

(DEFMETHOD CHANGE-V-SCALE ((WINDOW BOX-EDITOR))

;;; Permet de changer interactivement l'échelle verticale

   (DECLARE (SPECIAL GRAYSHADE1))
   (WITH-SLOTS (IL-WINDOW)
       WINDOW
```

```
    (LET* ((WINDOW-HEIGHT (INTERIOR-HEIGHT WINDOW))
           (YPOS (POSITION-YCOORD (GET-CURSOR-POSITION WINDOW)))
           (OLD-YPOS YPOS)
           (NEW-YPOS YPOS)
           (V-SCALE-IMAGE BITMAPCREATE 18 310))
           (V-SCALE (GETWINDOWPROP WINDOW 'V-SCALE)))
          (DRAW-LINE WINDOW (CONS 0 YPOS)
                 (CONS 18 YPOS)
                 :WIDTH 3 :OPERATION 'IL:PAINT)
          BITBLT IL-WINDOW 0 0 V-SCALE-IMAGE 0 0 18 WINDOW-HEIGHT 'IL:INPUT 'IL:PAINT)
          (LOOP WHILE KEYDOWNP 'IL:LEFT)
                DO
                (SETQ NEW-YPOS (KEEP-IN-LIMITS (POSITION-YCOORD (GET-CURSOR-POSITION
WINDOW))
                                               19 WINDOW-HEIGHT))
                (COND
                   ((NOT (= OLD-YPOS NEW-YPOS))
                    BITBLT V-SCALE-IMAGE 0 18 IL-WINDOW 0 18 18 (- (MIN YPOS NEW-YPOS)
                                                                   18)
                           'IL:INPUT
                           'IL:REPLACE)
                    (FILL-REGION WINDOW (LIST 3 (MIN YPOS NEW-YPOS)
                                              12
                                              (ABS (- YPOS NEW-YPOS)))
                           :TEXTURE GRAYSHADE1 :OPERATION 'IL:PAINT)
                    BITBLT V-SCALE-IMAGE 0 (MAX YPOS NEW-YPOS)
                           IL-WINDOW 0 (MAX YPOS NEW-YPOS)
                           18
                           (- WINDOW-HEIGHT (MAX YPOS NEW-YPOS))
                           'IL:INPUT
                           'IL:REPLACE)
                    (SETQ OLD-YPOS NEW-YPOS))))
          (PUTWINDOWPROP WINDOW 'V-SCALE (* V-SCALE (/ (- YPOS 18)
                                                       (- NEW-YPOS 18))))
          (COMPUTE-REGIONS (GETWINDOWPROP WINDOW 'LAYOUT-OBJECT)
                 WINDOW)
          (DRAW-V-SCALE WINDOW)
          (DRAW-REGIONS WINDOW))))

(DEFMETHOD CHANGE-VERTICAL-LINES ((WINDOW BOX-EDITOR)
                                  POINTED-OBJECT)
   (DECLARE (SPECIAL GRAYSHADE1 GRAYSHADE2))
   (WITH-SLOTS (IL-WINDOW)
      WINDOW
     (LET* ((XPOS (POSITION-XCOORD (GET-CURSOR-POSITION WINDOW)))
            (OLD-XPOS XPOS)
            (NEW-XPOS XPOS)
            (MAX-REGION (GETWINDOWPROP WINDOW 'MAX-REGION))
            (MAIN-REGION (GETWINDOWPROP WINDOW 'MAIN-REGION))
            (MARGIN-REGION (GETWINDOWPROP WINDOW 'MARGIN-REGION))
            (RIGHT-MAX-XPOS (REGION-RIGHT MAX-REGION))
            (RIGHT-MAIN-XPOS (REGION-RIGHT MAIN-REGION))
            (RIGHT-MARGIN-XPOS (REGION-RIGHT MARGIN-REGION))
            (LEFT-MARGIN-XPOS (REGION-LEFT MARGIN-REGION))
            (LIMITS)
            (LINE-YPOS)
            (LINE-LENGTH))
           (SETQ LIMITS (CASE POINTED-OBJECT
                              (RIGHT-MAX (LIST RIGHT-MAIN-XPOS (INTERIOR-WIDTH WINDOW)))
                              (RIGHT-MAIN (LIST RIGHT-MARGIN-XPOS RIGHT-MAX-XPOS))
                              (RIGHT-MARGIN (LIST LEFT-MARGIN-XPOS RIGHT-MAIN-XPOS))
                              (LEFT-MARGIN (LIST 20 RIGHT-MARGIN-XPOS))))
           (SETQ LINE-YPOS (CASE POINTED-OBJECT
                                 ((RIGHT-MAX RIGHT-MAIN) 20)
                                 ((RIGHT-MARGIN LEFT-MARGIN) (REGION-BOTTOM MARGIN-REGION))))
           (SETQ LINE-LENGTH (1- (CASE POINTED-OBJECT
                                       (RIGHT-MAX (REGION-HEIGHT MAX-REGION))
                                       (RIGHT-MAIN (REGION-HEIGHT MAIN-REGION))
                                       ((RIGHT-MARGIN LEFT-MARGIN) (REGION-HEIGHT
MARGIN-REGION)))))
           (SETQ OLD-XPOS (CASE POINTED-OBJECT
                                (RIGHT-MAX (REGION-RIGHT MAX-REGION))
                                (RIGHT-MAIN (REGION-RIGHT MAIN-REGION))
                                (RIGHT-MARGIN (REGION-RIGHT MARGIN-REGION))
                                (LEFT-MARGIN (REGION-LEFT MARGIN-REGION))))
```

```
          (PUT-CURSOR-POSITION WINDOW (MAKE-POSITION :XCOORD OLD-XPOS :YCOORD
(POSITION-YCOORD
(GET-CURSOR-POSITION
                                                                    WINDOW))))
          (LOOP WHILE KEYDOWNP 'IL:LEFT)
                DO
                (SETQ NEW-XPOS (KEEP-IN-LIMITS (POSITION-XCOORD (GET-CURSOR-POSITION
WINDOW))
                                       (FIRST LIMITS)
                                       (SECOND LIMITS)))
                (UNLESS (= OLD-XPOS NEW-XPOS)              ; A chaque déplacement du
curseur
                    (MOVE-TO WINDOW (CONS OLD-XPOS LINE-YPOS))
                    (RELDRAW-LINE WINDOW 0 LINE-LENGTH :WIDTH 1 :OPERATION 'IL:ERASE)
                                                           ; Efface la ligne précédente
                    (WHEN (OR (AND (< OLD-XPOS NEW-XPOS)
                                   (EQ POINTED-OBJECT 'LEFT-MARGIN))
                              (AND (> OLD-XPOS NEW-XPOS)
                                   (MEMBER POINTED-OBJECT '(RIGHT-MAX RIGHT-MAIN
RIGHT-MARGIN))))
                                                           ; Si le déplacement du curseur
se rapproche du centre de la région
                        (DRAW-LINE WINDOW (CONS OLD-XPOS LINE-YPOS)
                               (CONS NEW-XPOS LINE-YPOS)
                               :WIDTH 1 :OPERATION 'IL:ERASE)
                        (DRAW-LINE WINDOW (CONS OLD-XPOS (+ LINE-YPOS LINE-LENGTH))
                               (CONS NEW-XPOS (+ LINE-YPOS LINE-LENGTH))
                               :WIDTH 1 :OPERATION 'IL:ERASE)
                                                           ; Alors, il faut supprimer les
barres verticales
                        )
                    (CASE POINTED-OBJECT
                        (RIGHT-MAX (SETF (REGION-WIDTH MAX-REGION)
                                         (- NEW-XPOS 19)))
                        (RIGHT-MAIN (SETF (REGION-WIDTH MAIN-REGION)
                                          (- NEW-XPOS 19)))
                        (RIGHT-MARGIN (SETF (REGION-WIDTH MARGIN-REGION)
                                            (- NEW-XPOS (REGION-LEFT MARGIN-REGION)
                                               -1)))
                        (LEFT-MARGIN
                           (SETF (REGION-WIDTH MARGIN-REGION)
                                 (- (REGION-RIGHT MARGIN-REGION)
                                    NEW-XPOS -1))
                           (SETF (REGION-LEFT MARGIN-REGION)
                                 NEW-XPOS)))
                    (DRAW-REGION WINDOW MAX-REGION :OPERATION 'IL:PAINT :TEXTURE GRAYSHADE2)
                    (DRAW-REGION WINDOW MAIN-REGION :OPERATION 'IL:PAINT)
                    (DRAW-REGION WINDOW MARGIN-REGION :OPERATION 'IL:PAINT :TEXTURE
GRAYSHADE1)
                    (SETQ OLD-XPOS NEW-XPOS)))
          (PUTWINDOWPROP WINDOW 'MAX-REGION MAX-REGION)
          (PUTWINDOWPROP WINDOW 'MAIN-REGION MAIN-REGION)
          (PUTWINDOWPROP WINDOW 'MARGIN-REGION MARGIN-REGION)
          (PUTWINDOWPROP WINDOW 'REGIONS (COMPUTE-ACTIVE-REGIONS WINDOW)))))

(DEFMETHOD CLONE ((OBJECT ELEMENT-TYPE)
                  &OPTIONAL NAME)

;;; Effectue une copie de l'objet qui va venir s'intercaler entre l'objet et ses descendants

   (LET* ((NEW-OBJECT (MAKE-INSTANCE (CLASS-NAME (CLASS-OF OBJECT))
                                     :NAME NAME)))
         (SETF (KIND-OF NEW-OBJECT)
               OBJECT)
         (SETF (FO-DNIK NEW-OBJECT)
               (FO-DNIK OBJECT))
         (SETF (FO-DNIK OBJECT)
               (LIST NEW-OBJECT))
         (LOOP FOR X IN (FO-DNIK NEW-OBJECT)
               DO
               (SETF (KIND-OF X)
                     NEW-OBJECT))
         NEW-OBJECT))

(DEFMETHOD CLONE-IF-NECESSARY ((OBJECT ELEMENT-TYPE)
```

```
                                    (MODEL GUIDE))

;;; Clone l'objet si cela est nécessaire

   (LET ((REF (REFERENCE OBJECT MODEL T)))
        (UNLESS (REFERENCE OBJECT MODEL)                    ; Si le modéle n'est pas
propriétaire de l'objet
           (SETQ REF (CLONE REF))                           ; On le Clone
           (PUSH (LIST OBJECT REF)
                 (CONTEXT MODEL)))
        REF))

(DEFMETHOD CLOSEW ((WINDOW GUIDE-EDITOR))
   (DECLARE (SPECIAL *EDITED-GUIDES*))
   (SETQ *EDITED-GUIDES* (LOOP FOR X IN *EDITED-GUIDES* UNLESS (EQUAL WINDOW (SECOND X))
                               COLLECT X))

   ;; Suprime la fenêtre de la liste des éditeurs de modèle ouverts

   (CALL-NEXT-METHOD))

(DEFMETHOD COMPUTE-ACTIVE-REGIONS ((WINDOW BOX-EDITOR))
   (LET ((WINDOW-WIDTH (INTERIOR-WIDTH WINDOW))
         (WINDOW-HEIGHT (INTERIOR-HEIGHT WINDOW))
         (MAIN-REGION (GETWINDOWPROP WINDOW 'MAIN-REGION))
         (MAX-REGION (GETWINDOWPROP WINDOW 'MAX-REGION))
         (MARGIN-REGION (GETWINDOWPROP WINDOW 'MARGIN-REGION)))
        '((,(MAKE-REGION :LEFT 18 :BOTTOM 0 :WIDTH (- WINDOW-WIDTH 18)
                  :HEIGHT 18)
           HORIZONTAL-SCALE)
          (,(MAKE-REGION :LEFT 0 :BOTTOM 18 :WIDTH 18 :HEIGHT (- WINDOW-HEIGHT 18))
           VERTICAL-SCALE)
          (,(MAKE-REGION :LEFT (REGION-LEFT MARGIN-REGION)
                  :BOTTOM
                  (- (REGION-BOTTOM MARGIN-REGION)
                     2)
                  :WIDTH
                  (REGION-WIDTH MARGIN-REGION)
                  :HEIGHT 5)
           BOTTOM-MARGIN)
          (,(MAKE-REGION :LEFT (REGION-LEFT MARGIN-REGION)
                  :BOTTOM
                  (- (REGION-TOP MARGIN-REGION)
                     2)
                  :WIDTH
                  (REGION-WIDTH MARGIN-REGION)
                  :HEIGHT 5)
           TOP-MARGIN)
          (,(MAKE-REGION :LEFT (REGION-LEFT MAIN-REGION)
                  :BOTTOM
                  (- (- (REGION-TOP MAIN-REGION)
                        2)
                     2)
                  :WIDTH
                  (REGION-WIDTH MAIN-REGION)
                  :HEIGHT 5)
           TOP-MAIN)
          (,(MAKE-REGION :LEFT (REGION-LEFT MAX-REGION)
                  :BOTTOM
                  (- (REGION-TOP MAX-REGION)
                     2)
                  :WIDTH
                  (REGION-WIDTH MAX-REGION)
                  :HEIGHT 5)
           TOP-MAX)
          (,(MAKE-REGION :LEFT (- (REGION-LEFT MARGIN-REGION)
                                  2)
                  :BOTTOM
                  (REGION-BOTTOM MARGIN-REGION)
                  :WIDTH 5 :HEIGHT (REGION-HEIGHT MARGIN-REGION))
           LEFT-MARGIN)
          (,(MAKE-REGION :LEFT (- (REGION-RIGHT MARGIN-REGION)
                                  2)
                  :BOTTOM
                  (REGION-BOTTOM MARGIN-REGION)
                  :WIDTH 5 :HEIGHT (REGION-HEIGHT MARGIN-REGION))
```

```
          RIGHT-MARGIN)
         (,(MAKE-REGION :LEFT (- (REGION-RIGHT MAIN-REGION)
                                 2)
                 :BOTTOM
                 (REGION-BOTTOM MAIN-REGION)
                 :WIDTH 5 :HEIGHT (REGION-HEIGHT MAIN-REGION))
          RIGHT-MAIN)
         (,(MAKE-REGION :LEFT (- (REGION-RIGHT MAX-REGION)
                                 2)
                 :BOTTOM
                 (REGION-BOTTOM MAX-REGION)
                 :WIDTH 5 :HEIGHT (REGION-HEIGHT MAX-REGION))
          RIGHT-MAX))))

(DEFMETHOD COMPUTE-REGIONS ((OBJECT BOX)
                            (WINDOW BOX-EDITOR))

;;; Détermine les différentes régions occupées par l'objet et initialise les propriétés
correspondantes de la window

   (WITH-SLOTS (WIDTH MAX-WIDTH HEIGHT MAX-HEIGHT LEFT-MARGIN RIGHT-MARGIN BOTTOM-MARGIN
TOP-MARGIN)
       OBJECT
     (LET* ((WINDOW-WIDTH (INTERIOR-WIDTH WINDOW))
            (WINDOW-HEIGHT (INTERIOR-HEIGHT WINDOW))
            (V-SCALE (GETWINDOWPROP WINDOW 'V-SCALE))
            (H-SCALE (GETWINDOWPROP WINDOW 'H-SCALE))
            (MAIN-REGION (MAKE-REGION :LEFT 20 :BOTTOM 20 :WIDTH (ROUND WIDTH H-SCALE)
                                      :HEIGHT
                                      (ROUND HEIGHT V-SCALE)))
            (MAX-REGION (MAKE-REGION :LEFT 20 :BOTTOM 20 :WIDTH (ROUND MAX-WIDTH H-SCALE)
                                     :HEIGHT
                                     (ROUND MAX-HEIGHT V-SCALE)))
            (MARGIN-REGION (MAKE-REGION :LEFT (+ 20 (ROUND LEFT-MARGIN H-SCALE))
                                        :BOTTOM
                                        (+ 20 (ROUND BOTTOM-MARGIN V-SCALE))
                                        :WIDTH
                                        (- (REGION-RIGHT MAIN-REGION)
                                           20
                                           (ROUND RIGHT-MARGIN H-SCALE)
                                           (ROUND LEFT-MARGIN H-SCALE))
                                        :HEIGHT
                                        (- (REGION-TOP MAIN-REGION)
                                           20
                                           (ROUND TOP-MARGIN V-SCALE)
                                           (ROUND BOTTOM-MARGIN V-SCALE)))))
           (PUTWINDOWPROP WINDOW 'MAIN-REGION MAIN-REGION)
           (PUTWINDOWPROP WINDOW 'MAX-REGION MAX-REGION)
           (PUTWINDOWPROP WINDOW 'MARGIN-REGION MARGIN-REGION)
           (PUTWINDOWPROP WINDOW 'REGIONS (COMPUTE-ACTIVE-REGIONS WINDOW)))))

(DEFMETHOD DELETE-OBJECT :BEFORE
   ((MODEL GUIDE))
   (LOOP FOR X IN (CONTEXT MODEL)
         DO
         (DELETE-OBJECT (SECOND X)))

   ;; On supprime tous les objets définis dans le modèle

   (SETF (FO-DNIK (KIND-OF MODEL))
         (REMOVE MODEL (APPEND (FO-DNIK (KIND-OF MODEL))
                               (FO-DNIK MODEL))))
   (LOOP FOR X IN (FO-DNIK MODEL)
         DO
         (SETF (KIND-OF X)
               (KIND-OF MODEL))))

(DEFMETHOD DELETE-OBJECT :BEFORE
   ((OBJECT ELEMENT-TYPE))
   (SETF (FO-DNIK (KIND-OF OBJECT))
         (REMOVE OBJECT (APPEND (FO-DNIK (KIND-OF OBJECT))
                                (FO-DNIK OBJECT))))
   (LOOP FOR X IN (FO-DNIK OBJECT)
         DO
         (SETF (KIND-OF X)
               (KIND-OF OBJECT))))
```

```
(DEFMETHOD DELETE-OBJECT-ON-WINDOW ((OBJECT BOX)
                                    (WINDOW LAYOUT-MODEL-EDITOR))

;;; Efface l'objet sélectionné à l'intérieur du modèle édité

   (LET* ((REGION (THIRD (GETWINDOWPROP WINDOW 'SELECTED-REGION)))
          (SUPERORDINATE (LOOP FOR REG-INFO IN (GETWINDOWPROP WINDOW 'REGIONS)
                               THEREIS
                               (AND SUBREGIONP (THIRD REG-INFO)
                                    REGION)
                                    (NOT (EQUAL (THIRD REG-INFO)
                                                REGION))
                                    (SECOND REG-INFO)))))
         (DRAW-REGION WINDOW REGION :OPERATION 'IL:ERASE)    ; Efface l'objet sur la fenêtre
         (PRINT-STRING WINDOW (SLOT-VALUE OBJECT 'NAME)
                :PLACEMENT REGION :H-JUSTIFICATION 'LEFT :V-JUSTIFICATION 'TOP :OPERATION
                'IL:ERASE)
         (SETF (SLOT-VALUE SUPERORDINATE 'SUBORDINATES)
               (LOOP FOR X IN (SLOT-VALUE SUPERORDINATE 'SUBORDINATES)
                     UNLESS
                     (AND (EQUAL OBJECT (THIRD X))
                          (EQUAL (FOURTH X)
                                 (FOURTH (GETWINDOWPROP WINDOW 'SELECTED-REGION))))
                     COLLECT X))
         (REDISPLAY (GETWINDOWPROP WINDOW 'MODEL)
                WINDOW)))

(DEFMETHOD DISPLAY :AFTER
   ((OBJECT BODY)
    (WINDOW LAYOUT-MODEL-EDITOR)
    &KEY POSITION FREE-REGION))

(DEFMETHOD DISPLAY ((OBJECT BODY)
                    (WINDOW LAYOUT-MODEL-EDITOR)
                    &KEY POSITION FREE-REGION))

(DEFMETHOD DISPLAY :AFTER
   ((OBJECT COMPOSED-BOX)
    (WINDOW LAYOUT-MODEL-EDITOR)
    &KEY POSITION FREE-REGION)

;;; Cette fonction provoque l'affichage des sous-objets de l'objet courant

   (LOOP FOR X IN (GET-SUBORDINATES OBJECT)
         DO
         (DISPLAY (FIRST X)
                WINDOW :POSITION (SECOND X)
                :FREE-REGION
                (MAKE-REGION :LEFT (+ (REGION-LEFT FREE-REGION)
                                      (CAR POSITION)
                                      (SLOT-VALUE OBJECT 'LEFT-MARGIN))
                             :BOTTOM
                             (+ (REGION-BOTTOM FREE-REGION)
                                (CDR POSITION)
                                (SLOT-VALUE OBJECT 'BOTTOM-MARGIN))
                             :WIDTH
                             (- (SLOT-VALUE OBJECT 'WIDTH)
                                (SLOT-VALUE OBJECT 'RIGHT-MARGIN)
                                (SLOT-VALUE OBJECT 'LEFT-MARGIN))
                             :HEIGHT
                             (- (SLOT-VALUE OBJECT 'HEIGHT)
                                (SLOT-VALUE OBJECT 'TOP-MARGIN)
                                (SLOT-VALUE OBJECT 'BOTTOM-MARGIN))))))

(DEFMETHOD DISPLAY- :AFTER
   ((BODY BODY)
    (WINDOW LAYOUT-MODEL-EDITOR)
    &KEY POSITION FREE-REGION))

(DEFMETHOD DISPLAY- ((BODY BODY)
                     (WINDOW LAYOUT-MODEL-EDITOR)
                     &KEY POSITION FREE-REGION))

(DEFMETHOD DISPLAY- :AFTER
   ((OBJECT COMPOSED-BOX)
```

```
    (WINDOW LAYOUT-MODEL-EDITOR)
    &KEY POSITION FREE-REGION)

;;; Cette fonction provoque l'affichage des sous-objets de l'objet courant

   (LOOP FOR X IN (GET-SUBORDINATES OBJECT)
         DO
         (DISPLAY- (FIRST X)
                WINDOW :POSITION (SECOND X)
                :FREE-REGION
                '(,(+ (PCL::REGION-LEFT FREE-REGION)
                      (CAR POSITION)
                      (SLOT-VALUE OBJECT 'LEFT-MARGIN))
                  ,(+ (PCL::REGION-BOTTOM FREE-REGION)
                      (CDR POSITION)
                      (SLOT-VALUE OBJECT 'BOTTOM-MARGIN))
                  ,(- (SLOT-VALUE OBJECT 'WIDTH)
                      (SLOT-VALUE OBJECT 'RIGHT-MARGIN)
                      (SLOT-VALUE OBJECT 'LEFT-MARGIN))
                  ,(- (SLOT-VALUE OBJECT 'HEIGHT)
                      (SLOT-VALUE OBJECT 'TOP-MARGIN)
                      (SLOT-VALUE OBJECT 'BOTTOM-MARGIN))))))

(DEFMETHOD DISPLAY- ((OBJECT BOX)
                     (WINDOW LAYOUT-MODEL-EDITOR)
                     &KEY POSITION FREE-REGION)
   (LET* ((SCALE (GETWINDOWPROP WINDOW 'PAGE-SCALE))
          (OBJECT-REGION (MAKE-REGION :LEFT (+ (REGION-LEFT FREE-REGION)
                                               (CAR POSITION))
                                      :BOTTOM
                                      (+ (REGION-BOTTOM FREE-REGION)
                                         (CDR POSITION))
                                      :WIDTH
                                      (SLOT-VALUE OBJECT 'WIDTH)
                                      :HEIGHT
                                      (SLOT-VALUE OBJECT 'HEIGHT))); Région occupée par l'objet en
unité courante
          (OBJECT-WINDOW-REGION (MAKE-REGION :LEFT (ROUND (REGION-LEFT OBJECT-REGION)
                                                          SCALE)
                                             :BOTTOM
                                             (ROUND (REGION-BOTTOM OBJECT-REGION)
                                                    SCALE)
                                             :WIDTH
                                             (ROUND (REGION-WIDTH OBJECT-REGION)
                                                    SCALE)
                                             :HEIGHT
                                             (ROUND (REGION-HEIGHT OBJECT-REGION)
                                                    SCALE)))       ; Région occupée par l'objet en
pixels
          )
         (MEMORIZE-ACTIVE-REGION OBJECT WINDOW OBJECT-WINDOW-REGION POSITION)
                                                                   ; Mémorise la région occupée par
l'objet sur la fenêtre
         (DRAW-REGION WINDOW OBJECT-WINDOW-REGION :BORDER 1 :OPERATION 'IL:PAINT)
         (PRINT-STRING WINDOW (SLOT-VALUE OBJECT 'NAME)
                :PLACEMENT OBJECT-WINDOW-REGION :H-JUSTIFICATION 'LEFT :V-JUSTIFICATION 'TOP
                :OPERATION 'IL:REPLACE)))

(DEFMETHOD DISPLAY-DESCRIPTION ((OBJECT BOX)
                                (WINDOW BOX-EDITOR-DESCRIPTION))

;;; Affiche la description de l'objet OBJECT dans la fenêtre FreeMenu WINDOW

   (WITH-SLOTS (NAME WIDTH MAX-WIDTH OP-WIDTH HEIGHT MAX-HEIGHT OP-HEIGHT LEFT-MARGIN
RIGHT-MARGIN
                     BOTTOM-MARGIN TOP-MARGIN LEADING-SEPARATION TRAILING-SEPARATION
LEADING-OFFSET
                     TRAILING-OFFSET)
       OBJECT
     (LET ((IL-WINDOW (SLOT-VALUE WINDOW 'IL-WINDOW))
           (SCALE-FACTOR (GETWINDOWPROP (MAINWINDOW WINDOW)
                                        'SCALE-FACTOR))
           (ROUND-UNIT (GETWINDOWPROP (MAINWINDOW WINDOW)
                                      'ROUND-UNIT)))
          (CHANGE-STATE WINDOW :EDITMINWIDTH (DROUND (* WIDTH SCALE-FACTOR)
                                                     ROUND-UNIT))
```

```
          (CHANGE-STATE WINDOW :EDITMAXWIDTH (DROUND (* MAX-WIDTH SCALE-FACTOR)
                                                     ROUND-UNIT))
          (CHANGE-STATE WINDOW :|(WIDTHFIXED WIDTHMINSIZE WIDTHMAXSIZE)| (CASE OP-WIDTH
                                                                            (FIXED
:WIDTHFIXED)
                                                                            (MINSIZE
:WIDTHMINSIZE)
                                                                            (MAXSIZE
:WIDTHMAXSIZE))
                )
          (COND
             ((EQUAL OP-WIDTH 'FIXED)
              (INHIBIT-ITEM WINDOW FM.GETITEM :MAXWIDTH NIL IL-WINDOW))
              (INHIBIT-ITEM WINDOW FM.GETITEM :EDITMAXWIDTH NIL IL-WINDOW)))
             (T (RESTORE-ITEM WINDOW FM.GETITEM :MAXWIDTH NIL IL-WINDOW)
                      'IL:DISPLAY)
                (RESTORE-ITEM WINDOW FM.GETITEM :EDITMAXWIDTH NIL IL-WINDOW)
                      'IL:NUMBER)))
          (CHANGE-STATE WINDOW :EDITMINHEIGHT (DROUND (* HEIGHT SCALE-FACTOR)
                                                      ROUND-UNIT))
          (CHANGE-STATE WINDOW :EDITMAXHEIGHT (DROUND (* MAX-HEIGHT SCALE-FACTOR)
                                                      ROUND-UNIT))
          (CHANGE-STATE WINDOW :|(HEIGHTFIXED HEIGHTMINSIZE HEIGHTMAXSIZE)| (CASE OP-HEIGHT
                                                                              (FIXED
:HEIGHTFIXED)
                                                                              (MINSIZE
:HEIGHTMINSIZE)
                                                                              (MAXSIZE
:HEIGHTMAXSIZE)))
          (COND
             ((EQUAL OP-HEIGHT 'FIXED)
              (INHIBIT-ITEM WINDOW FM.GETITEM :MAXHEIGHT NIL IL-WINDOW))
              (INHIBIT-ITEM WINDOW FM.GETITEM :EDITMAXHEIGHT NIL IL-WINDOW)))
             (T (RESTORE-ITEM WINDOW FM.GETITEM :MAXHEIGHT NIL IL-WINDOW)
                      'IL:DISPLAY)
                (RESTORE-ITEM WINDOW FM.GETITEM :EDITMAXHEIGHT NIL IL-WINDOW)
                      'IL:NUMBER)))
          (CHANGE-STATE WINDOW :EDITLEFTMARGIN (DROUND (* LEFT-MARGIN SCALE-FACTOR)
                                                       ROUND-UNIT))
          (CHANGE-STATE WINDOW :EDITRIGHTMARGIN (DROUND (* RIGHT-MARGIN SCALE-FACTOR)
                                                        ROUND-UNIT))
          (CHANGE-STATE WINDOW :EDITTOPMARGIN (DROUND (* TOP-MARGIN SCALE-FACTOR)
                                                      ROUND-UNIT))
          (CHANGE-STATE WINDOW :EDITBOTTOMMARGIN (DROUND (* BOTTOM-MARGIN SCALE-FACTOR)
                                                         ROUND-UNIT))
          (CHANGE-STATE WINDOW :EDIT-LEADING-SEPARATION (DROUND (* LEADING-SEPARATION
SCALE-FACTOR)
                                                                ROUND-UNIT))
          (CHANGE-STATE WINDOW :EDIT-TRAILING-SEPARATION (DROUND (* TRAILING-SEPARATION
SCALE-FACTOR)
                                                                 ROUND-UNIT))
          (CHANGE-STATE WINDOW :EDIT-LEADING-OFFSET (DROUND (* LEADING-OFFSET SCALE-FACTOR)
                                                            ROUND-UNIT))
          (CHANGE-STATE WINDOW :EDIT-TRAILING-OFFSET (DROUND (* TRAILING-OFFSET SCALE-FACTOR)
                                                             ROUND-UNIT)))))

(DEFMETHOD DISPLAY-MENU ((OBJECT COMPOSED-BOX)
                         (WINDOW LAYOUT-MODEL-EDITOR)))

(DEFMETHOD DISPLAY-MENU ((OBJECT BOX)
                         (WINDOW LAYOUT-MODEL-EDITOR))

;;; Affiche un menu contenant les actions possibles sur l'objet

   (LET ((CHOICE (EXECUTE (MAKE-INSTANCE 'MENU :ITEMS '(("Delete" 'DELETE))))))
        (CASE CHOICE
           ('DELETE (DELETE-OBJECT-ON-WINDOW OBJECT WINDOW)))))

(DEFMETHOD DRAW-H-SCALE ((WINDOW BOX-EDITOR))

   ;; Dessine ou redessine la graduation horizontale

   (LET* ((WINDOW-WIDTH (INTERIOR-WIDTH WINDOW))
          (H-SCALE (GETWINDOWPROP WINDOW 'H-SCALE))
```

```
          (ROUND-UNIT (GETWINDOWPROP WINDOW 'ROUND-UNIT))
          (SCALE-FACTOR (GETWINDOWPROP WINDOW 'SCALE-FACTOR))
          (UNIT ROUND-UNIT))
         (LOOP WHILE (< (/ UNIT H-SCALE SCALE-FACTOR)
                        2)
               DO
               (SETQ UNIT (* 10 UNIT)))
         (FILL-REGION WINDOW '(18 0 ,(- WINDOW-WIDTH 18)
                                 18)
                :TEXTURE IL:WHITESHADE :OPERATION 'IL:REPLACE)
         (DRAW-LINE WINDOW '(18 . 18)
                '(,WINDOW-WIDTH . 18)
                :WIDTH 2 :OPERATION 'IL:PAINT)
         (LET ((CTR 10)
               (NB-UNIT 0))
              (LOOP FOR X FROM 18 TO WINDOW-WIDTH BY (/ UNIT H-SCALE SCALE-FACTOR)
                    DO
                    (COND
                       ((= CTR 10)
                        (SETQ CTR 1)
                        (DRAW-LINE WINDOW (CONS X 18)
                               (CONS X 10)
                               :WIDTH 2 :OPERATION 'IL:PAINT)
                        (RELMOVE-TO WINDOW 0 -8)
                        (PRINT-STRING WINDOW (FORMAT NIL "~D" (DROUND NB-UNIT ROUND-UNIT))
                               :H-JUSTIFICATION
                               'CENTER)
                        (INCF NB-UNIT (* UNIT 10))          ; tracage tous les 10 UNITs
                        )
                       ((= CTR 5)
                        (INCF CTR)
                        (DRAW-LINE WINDOW (CONS X 18)
                               (CONS X 14)
                               :WIDTH 1 :OPERATION 'IL:PAINT); tracage tous les 5 UNITs
                        )
                       (T (INCF CTR)
                          (DRAW-LINE WINDOW (CONS X 18)
                                 (CONS X 16)
                                 :WIDTH 1 :OPERATION 'IL:PAINT)
                                                            ; Tracage tous les UNIT
                        ))))))

(DEFMETHOD DRAW-REGIONS ((WINDOW BOX-EDITOR))
   (DECLARE (SPECIAL GRAYSHADE1 GRAYSHADE2))
   (LET* ((WINDOW-WIDTH (INTERIOR-WIDTH WINDOW))
          (WINDOW-HEIGHT (INTERIOR-HEIGHT WINDOW))
          (MAIN-REGION (GETWINDOWPROP WINDOW 'MAIN-REGION))
          (MAX-REGION (GETWINDOWPROP WINDOW 'MAX-REGION))
          (MARGIN-REGION (GETWINDOWPROP WINDOW 'MARGIN-REGION)))
         (FILL-REGION WINDOW '(20 20 ,(- WINDOW-WIDTH 20)
                                  ,(- WINDOW-HEIGHT 20))
                :TEXTURE IL:WHITESHADE :OPERATION 'IL:REPLACE)
         (DRAW-REGION WINDOW MAIN-REGION :OPERATION 'IL:PAINT)
         (DRAW-REGION WINDOW MAX-REGION :OPERATION 'IL:PAINT :TEXTURE GRAYSHADE2)
         (DRAW-REGION WINDOW MARGIN-REGION :OPERATION 'IL:PAINT :TEXTURE GRAYSHADE1)))

(DEFMETHOD DRAW-V-SCALE ((WINDOW BOX-EDITOR))

   ;; Dessine ou redessine la graduation horizontale

   (LET* ((WINDOW-HEIGHT (INTERIOR-HEIGHT WINDOW))
          (V-SCALE (GETWINDOWPROP WINDOW 'V-SCALE))
          (ROUND-UNIT (GETWINDOWPROP WINDOW 'ROUND-UNIT))
          (SCALE-FACTOR (GETWINDOWPROP WINDOW 'SCALE-FACTOR))
          (UNIT ROUND-UNIT))
         (LOOP WHILE (< (/ UNIT V-SCALE SCALE-FACTOR)
                        2)
               DO
               (SETQ UNIT (* 10 UNIT)))
         (FILL-REGION WINDOW '(0 18 18 ,(- WINDOW-HEIGHT 18))
                :TEXTURE IL:WHITESHADE :OPERATION 'IL:REPLACE)
         (DRAW-LINE WINDOW (CONS 18 18)
                (CONS 18 WINDOW-HEIGHT)
                :WIDTH 2 :OPERATION 'IL:PAINT)
         (LET ((CTR 10)
               (NB-UNIT 0))
```

```
            (LOOP FOR Y FROM 18 TO WINDOW-HEIGHT BY (/ UNIT V-SCALE SCALE-FACTOR)
                  DO
                  (COND
                     ((= CTR 10)
                      (SETQ CTR 1)
                      (DRAW-LINE WINDOW (CONS 18 Y)
                             (CONS 10 Y)
                             :WIDTH 2 :OPERATION 'IL:PAINT)
                      (RELMOVE-TO WINDOW -10 0)
                      (PRINT-STRING WINDOW (FORMAT NIL "~D" (DROUND NB-UNIT ROUND-UNIT))
                             :V-JUSTIFICATION
                             'MIDDLE :OPERATION 'IL:REPLACE)
                      (INCF NB-UNIT (* UNIT 10))          ; tracage tous les 10 UNITs
                      )
                     ((= CTR 5)
                      (INCF CTR)
                      (DRAW-LINE WINDOW (CONS 18 Y)
                             (CONS 14 Y)
                             :WIDTH 1 :OPERATION 'IL:PAINT); tracage tous les 5 UNITs
                      )
                     (T (INCF CTR)
                        (DRAW-LINE WINDOW (CONS 18 Y)
                               (CONS 16 Y)
                               :WIDTH 1 :OPERATION 'IL:PAINT)
                                                          ; Tracage tous les UNIT
                      ))))))

(DEFMETHOD EDIT ((MODEL GUIDE))

;;; Ouvre une fenêtre et lance l'édition du guide

   (DECLARE (SPECIAL *EDITED-GUIDES*))
   (LET ((WINDOW (MAKE-INSTANCE 'GUIDE-EDITOR :TITLE (FORMAT NIL "Guide ~A" (NAME MODEL)))))
        (SETF (EDITED-MODEL WINDOW)
              MODEL)
        (PUSH (LIST MODEL WINDOW)
              *EDITED-GUIDES*)
        (CLEARW WINDOW)
        (PRINT-DESCRIPTION (REFERENCE (ROOT MODEL)
                                      MODEL T)
               WINDOW)))

(DEFMETHOD EDIT ((OBJECT ELEMENT-TYPE))
   (DECLARE (SPECIAL *GENERIC-OBJECT-BROWSER*))
   (LET ((OBJECT-COPY (MAKE-INSTANCE (CLASS-OF OBJECT)))
         (WINDOW CREATEW '(300 100 300 300)
                 NIL NIL T)))
        (WHEN (EQUAL (SLOT-VALUE OBJECT 'NAME)
                     "<<New Object>>")
            (SETF (SLOT-VALUE OBJECT-COPY 'NAME)
                  (PROMPT-FOR-STRING :TITLE "Enter the Name of the new Object" :PROMPT-STRING
"Name:"
                         :CANDIDATE-STRING "NoName"))
            (SETQ OBJECT (MAKE-INSTANCE (CLASS-OF OBJECT)
                                :MEMORIZE T))
            (COPY OBJECT-COPY OBJECT)                     ; On recopie tous ses slots dans
'OBJECT
            (REPLACE-ITEMS *GENERIC-OBJECT-BROWSER* (GET-BROWSER-ITEMS 'ELEMENT-TYPE))
                                                          ; On met à jour le browser
            )
        CLOSEW WINDOW)))

(DEFMETHOD EDIT ((OBJECT LAYOUT-MODEL))
   (LET* ((OBJECT-COPY (MAKE-INSTANCE (CLASS-OF OBJECT)))    ; On travaille sur une copie de
l'objet
          (PAGE-SCALE (GETF (SLOT-VALUE OBJECT 'DISPLAY-PARAMETERS)
                            'PAGE-SCALE))
          (MENU-WINDOW (MAKE-INSTANCE 'MENU-WINDOW :ITEMS '(("Ok" OK)
                                                            ("Abort" ABORT)
                                                            ("Export P" EXPORT-P)
                                                            ("    Other Pages    "
CHANGE-PAGE))
                                :TITLE "Layout Model Editor" :MENUROWS 1 :WHENSELECTEDFN
                                #'(LAMBDA (ITEM MENU MOUSE)
                                          (LET* ((MENU-WINDOW GETMENUPROP MENU 'CLOS-MENU))
                                                 (MAIN-WINDOW (MAINWINDOW MENU-WINDOW)))
```

```
                                          (CASE (SECOND ITEM)
                                              ('OK (OK MAIN-WINDOW))
                                              ('ABORT (CLOSEW MAIN-WINDOW))
                                              ('CHANGE-PAGE (CHANGE-PAGE
MAIN-WINDOW)))))))
          (MAIN-WINDOW (MAKE-INSTANCE 'LAYOUT-MODEL-EDITOR :REGION
                                      '(400 50 ,WIDTHIFWINDOW (ROUND (SLOT-VALUE OBJECT 'WIDTH)
                                                                     PAGE-SCALE))
                                           ,HEIGHTIFWINDOW (ROUND (SLOT-VALUE OBJECT 'HEIGHT)
                                                                  PAGE-SCALE)
                                                  T))))
          (PAGES)
          (FIRST-PAGE)                                        ; FIRST-PAGE est une liste du
type (Objet, Position)
          )
         CURSOR IL:WAITINGCURSOR)                          ; Affiche un sablier
         (COPY OBJECT OBJECT-COPY)
         (SETQ PAGES (GET-SUBORDINATES OBJECT-COPY))
         (SETQ FIRST-PAGE (CAR PAGES))                      ; Sauvegarde l'objet
         (ATTACH MAIN-WINDOW MENU-WINDOW :EDGE 'IL:TOP :POSITIONEDGE 'IL:LEFT)
         (PUT-FONT MAIN-WINDOW FONTCREATE 'HELVETICA 8 'MRR))
         (PUTWINDOWPROP MAIN-WINDOW 'MODEL-MEMO OBJECT)
         (PUTWINDOWPROP MAIN-WINDOW 'MODEL OBJECT-COPY)      ; Mémorise le Modèle affiché
dans l'éditeur
         (PUTWINDOWPROP MAIN-WINDOW 'PAGE-SCALE PAGE-SCALE)  ; Mémorise l'échelle utilisée
         (PUTWINDOWPROP MAIN-WINDOW 'CURRENT-PAGE 0)         ; Mémorise la page affichée sur
l'éditeur.
                                                             ; On accéde à la page par
(NTH(SUBORDINATES OBJECT))
         (PUTWINDOWPROP MAIN-WINDOW 'MENU-WINDOW MENU-WINDOW)
         (REDISPLAY OBJECT-COPY MAIN-WINDOW)                 ; Affiche le modèle sur la
fenêtre
         CURSOR T)))

(DEFMETHOD EDIT ((OBJECT BOX))
   (DECLARE (SPECIAL *MEASURE-UNITS* *DEFAULT-UNIT* *DISPLAY-PARAMETERS*
                     IL:FM-LAYOUT-OBJECT-DESCR IL:FM-LAYOUT-OBJECT-CDE))
   CURSOR IL:WAITINGCURSOR)                                  ; Affiche un sablier
   (LET* ((OBJECT-COPY (MAKE-INSTANCE (CLASS-OF OBJECT)))    ; On travaille sur une copie de
l'objet
          (FREE-MENU (MAKE-INSTANCE 'BOX-EDITOR-DESCRIPTION :DESCRIPTION
IL:FM-LAYOUT-OBJECT-DESCR
                                    :BORDER 3))              ; WINDOW contenant la
description détaillée de l'objet édité
          (MENU-WINDOW (MAKE-INSTANCE 'MENU-WINDOW :ITEMS '(("Ok" OK)
                                                            ("Abort" ABORT)
                                                            ("Apply Descr.        "
APPLY-DESCRIPTION
                                                                  )
                                                            ("Apply Graphic" APPLY-GRAPHIC))
                                    :TITLE "Layout objects Editor" :MENUROWS 1 :WHENSELECTEDFN
                                    #'(LAMBDA (ITEM MENU MOUSE)
                                              (LET* ((CLOS-WINDOW (MAINWINDOW GETMENUPROP MENU
                                                                              'CLOS-MENU)))
                                                     (OBJECT (GETWINDOWPROP CLOS-WINDOW
'LAYOUT-OBJECT)))
                                                    (CASE (SECOND ITEM)
                                                        ('OK (OK CLOS-WINDOW))
                                                        ('ABORT (CLOSEW CLOS-WINDOW))
                                                        ('APPLY-DESCRIPTION (UPDATE-OBJECT
                                                                             OBJECT
                                                                             (CAR (ATTACHEDWINDOWS

CLOS-WINDOW))))
                                                        ('APPLY-GRAPHIC (UPDATE-OBJECT OBJECT

CLOS-WINDOW)))
                                                    ))))
          (MAIN-WINDOW (MAKE-INSTANCE 'BOX-EDITOR :REGION '(100 100 ,(REGION-WIDTH
(WINDOWREGION

FREE-MENU))
                                                                 310)
                                    :BORDERSIZE 1 :NOOPENFLG T))   ; Fenêtre ou est affiché l'objet
édité
          )
         (IF (EQUAL (SLOT-VALUE OBJECT 'NAME)
```

```
                "<<New Object>>")
            (SETF (SLOT-VALUE OBJECT-COPY 'NAME)
                  (PROMPT-FOR-STRING :TITLE "Enter the Name of the new Object"
:PROMPT-STRING
                                     "Name:" :CANDIDATE-STRING "NoName"))
            (COPY OBJECT OBJECT-COPY))                          ; Si l'objet est un nouvel
objet, alors, on lui change simplement le nom , sinon, on recopie les slots de l'ancien objet
        (PUTWINDOWPROP MAIN-WINDOW 'LAYOUT-OBJECT OBJECT-COPY)
                                                                ; Objet sur lequel on travaille
        (PUTWINDOWPROP MAIN-WINDOW 'LAYOUT-OBJECT-MEMO OBJECT)
                                                                ; Mémorisation de l'objet
original
        (ATTACH MAIN-WINDOW FREE-MENU :EDGE 'IL:TOP :POSITIONEDGE 'IL:LEFT)
                                                                ; Attache la fenêtre contenant
la description détaillée de l'objet édité
        (ATTACH MAIN-WINDOW MENU-WINDOW :EDGE 'IL:TOP :POSITIONEDGE 'IL:LEFT)
                                                                ; MENU-WINDOW est attaché à
MAIN-WINDOW
        (FIX-SIZE MENU-WINDOW)
        (FIX-SIZE FREE-MENU)
        (SET-DISPLAY-FONT MAIN-WINDOW FONTCREATE '(TIMESROMAN 8 MRR)))
        (LET* ((DISPLAY-PARAMETERS (CASE *DISPLAY-PARAMETERS*
                                         (PREVIOUS (SLOT-VALUE OBJECT-COPY
'DISPLAY-PARAMETERS))
                                         (DEFAULT (GETF *MEASURE-UNITS* *DEFAULT-UNIT*))))
               (SCALE-FACTOR (GETF DISPLAY-PARAMETERS 'SCALE-FACTOR)))
              (SETF (SLOT-VALUE OBJECT-COPY 'DISPLAY-PARAMETERS)
                    DISPLAY-PARAMETERS)
              (PUTWINDOWPROP MAIN-WINDOW 'ROUND-UNIT (GETF DISPLAY-PARAMETERS 'ROUND-UNIT))
              (PUTWINDOWPROP MAIN-WINDOW 'SCALE-FACTOR SCALE-FACTOR)
              (PUTWINDOWPROP MAIN-WINDOW 'H-SCALE (/ (GETF DISPLAY-PARAMETERS 'H-SCALE)
                                                     SCALE-FACTOR))
              (PUTWINDOWPROP MAIN-WINDOW 'V-SCALE (/ (GETF DISPLAY-PARAMETERS 'V-SCALE)
                                                     SCALE-FACTOR))
              (PUTWINDOWPROP MAIN-WINDOW 'MEASURE-UNIT (GETF DISPLAY-PARAMETERS 'UNIT)))
        (CHANGE-LABEL FREE-MENU :NAME (SLOT-VALUE OBJECT-COPY 'NAME))
                                                                ; Affiche le nom de l'objet
        (CHANGE-LABEL FREE-MENU :UNIT (CASE (GETWINDOWPROP MAIN-WINDOW 'MEASURE-UNIT)
                                            (CM :CM)
                                            (INCH :INCH)
                                            (PIXEL :PIXEL)))  ; Affiche l'unité
        (DISPLAY-DESCRIPTION OBJECT-COPY FREE-MENU)             ; Affiche la description de
l'objet dans la fenêtre correspondante
        (COMPUTE-REGIONS OBJECT-COPY MAIN-WINDOW)               ; Détermine les différentes
régions occupées par l'objet et initialise les propriétés correspondantes de la window
        (REPAINT MAIN-WINDOW)                                   ; Dessine le contenu de
l'éditeur
        CURSOR T)))

(DEFMETHOD EDIT-NODE ((GRAPHER GUIDES-GRAPHER)
                      NODE)

;;; Propose les options d'édition sur un noeud

   (CASE (EXECUTE (MAKE-INSTANCE 'MENU :TITLE " Edit Options " :ITEMS '(("Specialize Guide"
                                                                         'SPECIALIZE)
                                                                        ("Edit Guide" 'EDIT)
                                                                        ("Export SGML"
'EXPORT-SGML)
                                                                        ("Delete Guide"
'DELETE))))
      (SPECIALIZE
         (SPECIALIZE (CAR NODE)
                (PROMPT-FOR-STRING :TITLE "Enter the Name of the New Guide" :PROMPT-STRING
"Name:"))
         (COMPUTE-GRAPH GRAPHER)
         (FIT-TO-REGION GRAPHER GRAPHREGION (IL-GRAPH GRAPHER))))
      (EDIT (EDIT (CAR NODE)))
      (EXPORT-SGML (EXPORT-BIBLE (CAR NODE)))
      (DELETE (WHEN MOUSECONFIRM (FORMAT NIL
                                         "Deleting ~A...~%All Data defined in the Guide will be
lost !"
                                         (NAME (CAR NODE))))
              (DELETE-OBJECT (CAR NODE))
              (COMPUTE-GRAPH GRAPHER)
              (FIT-TO-REGION GRAPHER GRAPHREGION (IL-GRAPH GRAPHER)))))))
```

```
(DEFMETHOD EDIT-OK ((OBJECT LAYOUT-MODEL)
                    WINDOW)

;;; Recopie l'objet édité dans l'objet mémorisé

   CLOSEW WINDOW)
   CURSOR IL:WAITINGCURSOR)                                    ; Affiche un sablier
   (COPY OBJECT WINDOWPROP WINDOW 'MODEL-MEMO))
   CURSOR T))

(DEFMETHOD EDIT-REDISPLAY ((OBJECT LAYOUT-MODEL)
                           WINDOW)

;;; Réaffiche les données initiales de l'objet lorsque le bouton REDISPLAY est sélectionné

   CURSOR IL:WAITINGCURSOR)                                    ; Affiche un sablier
   (LET ((NEW-MODEL (MAKE-INSTANCE 'LAYOUT-MODEL)))
        WINDOWPROP WINDOW 'MODEL NEW-MODEL)
        (COPY WINDOWPROP WINDOW 'MODEL-MEMO)
              NEW-MODEL)                                          ; Reprend les données de l'objet
mémorisé
        (REDISPLAY NEW-MODEL WINDOW))                             ; Et réaffiche
   CURSOR T))

(DEFMETHOD EXPORT-BIBLE ((MODEL GUIDE))

;;; Crée un fichier contenant la description SGML du modèle

   (DECLARE (SPECIAL *PRINTED-OBJECTS*))
   (LET ((STREAM OPENSTREAM (FORMAT NIL "/u/pasquier/testbible/~A.dtd" (STRING-UPCASE
                                                                        (NAME MODEL)))
                            'IL:OUTPUT)))
        (SETQ *PRINTED-OBJECTS* NIL)
        (FORMAT STREAM "<!DOCTYPE ~A [~%" (STRING-UPCASE (NAME MODEL)))
        (EXPORT-SGML (REFERENCE (ROOT MODEL)
                                MODEL T)
              MODEL STREAM)
        (FORMAT STREAM "]>~%")
        (CLOSE STREAM)))

(DEFMETHOD EXPORT-P ((LAYOUT-MODEL LAYOUT-MODEL)
                     (GUIDE GUIDE)
                     &OPTIONAL ARG)

;;; Crée un fichier contenant la description P  du modèle physique

   (DECLARE (SPECIAL *PRINTED-OBJECTS*))
   (SETQ *PRINTED-OBJECTS* NIL)
   (LET* ((STREAM OPENSTREAM (FORMAT NIL "/u/pasquier/testbible/~AP.P" (STRING-UPCASE
                                                                        (NAME GUIDE)))
                             'IL:OUTPUT))
          (PAGE (CAAR (GET-SUBORDINATES LAYOUT-MODEL)))
          (P-WIDTH (SLOT-VALUE PAGE 'WIDTH))
          (P-HEIGHT (SLOT-VALUE PAGE 'HEIGHT))
          (SCALE (SECOND (MEMBER 'SCALE-FACTOR (SLOT-VALUE PAGE 'DISPLAY-PARAMETERS)))))
        (SETQ *PRINTED-OBJECTS* NIL)
        (FORMAT STREAM "PRESENTATION ~A;~%" (STRING-UPCASE (NAME GUIDE)))
        (FORMAT STREAM "VIEWS~%")
        (FORMAT STREAM "~6tfirstView;~2%")
        (FORMAT STREAM "DEFAULT~%")
        (FORMAT STREAM "~6tBEGIN~%")
        (FORMAT STREAM "~6tHorizRef: Enclosed . HRef;~%")
        (FORMAT STREAM "~6tVertRef: * . Left;~%")
        (FORMAT STREAM "~6tWidth: Enclosed . Width;~%")
        (FORMAT STREAM "~6tHeight: Enclosed . Height;~%")
        (FORMAT STREAM "~6tVertPos: Top = Previous . Bottom;~%")
        (FORMAT STREAM "~6tHorizPos: Left = Enclosing . Left;~%")
        (FORMAT STREAM "~6tJustify: Enclosing =;~%")
        (FORMAT STREAM "~6tLineSpacing: Enclosing =;~%")
        (FORMAT STREAM "~6tBreak: Yes;~%")
        (FORMAT STREAM "~6tVisibility: Enclosing =;~%")
        (FORMAT STREAM "~6tFont: Enclosing =;~%")
        (FORMAT STREAM "~6tStyle: Enclosing =;~%")
        (FORMAT STREAM "~6tUnderline : Enclosing =;~%")
        (FORMAT STREAM "~6tthickness : Enclosing =;~%")
```

```
        (FORMAT STREAM "~6tSize: Enclosing =;~%")
        (FORMAT STREAM "~6tAdjust: Enclosing =;~%")
        (FORMAT STREAM "~6tIndent: Enclosing =;~%")
        (FORMAT STREAM "~6tDepth : 3;~%")
        (FORMAT STREAM "~6tEND;~2%")
        (FORMAT STREAM "BOXES~2%")
        (FORMAT STREAM "{----- Definition des pages -----}~2%")
        (EXPORT-P PAGE STREAM)
        (FORMAT STREAM "RULES~2%")
        (FORMAT STREAM "~6t~A:~%" (STRING-UPCASE (NAME GUIDE)))
        (FORMAT STREAM "~8tBEGIN~%")
        (FORMAT STREAM "~8tPage(~A);~%" (STRING-UPCASE (NAME PAGE)))
        (FORMAT STREAM "~8tSize: ~A;~%" (SLOT-VALUE LAYOUT-MODEL 'SIZE))
        (FORMAT STREAM "~8tFont: ~A;~%" (SLOT-VALUE LAYOUT-MODEL 'FONT))
        (FORMAT STREAM "~8tStyle: ~A;~%" (SLOT-VALUE LAYOUT-MODEL 'STYLE))
        (FORMAT STREAM "~8tIndent: ~A;~%" (SLOT-VALUE LAYOUT-MODEL 'INDENT))
        (FORMAT STREAM "~8tJustify: ~A;~%" (SLOT-VALUE LAYOUT-MODEL 'JUSTIFY))
        (FORMAT STREAM "~8tLineSpacing: ~A;~%" (SLOT-VALUE LAYOUT-MODEL 'LINESPACING))
        (FORMAT STREAM "~8tAdjust: ~A;~%" (SLOT-VALUE LAYOUT-MODEL 'ADJUST))
        (FORMAT STREAM "~8tUnderline: ~A;~%" (SLOT-VALUE LAYOUT-MODEL 'UNDERLINE))
        (FORMAT STREAM "~8tThickness: ~A;~%" (SLOT-VALUE LAYOUT-MODEL 'THICKNESS))
        (FORMAT STREAM "~8tEND;~2%")
        (LET (POSX POSY)
             (LOOP FOR X IN (GET-SUBORDINATES PAGE)
                   DO
                   (UNLESS (EQUAL (CLASS-NAME (CLASS-OF (CAR X)))
                                  'BODY)
                       (SETQ POSX (* (CAADR X)
                                     SCALE))
                       (SETQ POSY (* (- P-HEIGHT (CDADR X))
                                     SCALE))
                       (EXPORT-P (CAR X)
                              STREAM
                              (CONS POSX POSY)))))
        (LOOP FOR X IN (GET-ORDERED-SUBORDINATES (REFERENCE (ROOT GUIDE)
                                                            GUIDE T))
              DO
              (EXPORT-P (REFERENCE (CAR X)
                                   GUIDE T)
                        GUIDE STREAM))
        (FORMAT STREAM "END~%")
        (CLOSE STREAM)))

(DEFMETHOD EXPORT-P ((COMPOSED-BOX COMPOSED-BOX)
                     STREAM &OPTIONAL POS)
   (DECLARE (SPECIAL *PRINTED-OBJECTS*))
   (UNLESS (FIND (NAME COMPOSED-BOX)
                 *PRINTED-OBJECTS*)
       (PUSH (NAME COMPOSED-BOX)
             *PRINTED-OBJECTS*)
       (LET ((SCALE (SECOND (MEMBER 'SCALE-FACTOR (SLOT-VALUE COMPOSED-BOX
'DISPLAY-PARAMETERS)))))
            (P-HEIGHT (SLOT-VALUE COMPOSED-BOX 'HEIGHT))
            POSX POSY)
           (FORMAT STREAM "~6t~A:~%" (STRING-UPCASE (NAME COMPOSED-BOX)))
           (FORMAT STREAM "~8tBEGIN~%")
           (FORMAT STREAM "~8tWidth: ~A;~%" (CASE (SLOT-VALUE COMPOSED-BOX 'OP-WIDTH)
                                                  (FIXED (FORMAT NIL "~,2F cm"
                                                                 (* (SLOT-VALUE COMPOSED-BOX
                                                                                'WIDTH)
                                                                    SCALE)))
                                                  (MAXSIZE "Enclosing . Width")
                                                  (MINSIZE "Enclosed . Width")))
           (FORMAT STREAM "~8tHeight: ~A;~%" (CASE (SLOT-VALUE COMPOSED-BOX 'OP-HEIGHT)
                                                   (FIXED (FORMAT NIL "~,2F cm"
                                                                  (* (SLOT-VALUE COMPOSED-BOX
                                                                                 'HEIGHT)
                                                                     SCALE)))
                                                   (MAXSIZE "Enclosing . Height")
                                                   (MINSIZE "Enclosed . Height")))
           (FORMAT STREAM "~8tHorizPos: ~A;~%" (COND
                                                 ((EQUAL 'VARIABLE POS)
                                                  (FORMAT NIL "Left = Enclosing . Left +
~,2F cm"
                                                          (SLOT-VALUE COMPOSED-BOX
                                                                 'LEFT-HAND-OFFSET)))
```

```
                                                    (T (FORMAT NIL "Left = Enclosing . Left +
~,2F cm"
                                                       (CAR POS)))))
            (FORMAT STREAM "~8tVertPos: ~A;~%" (COND
                                                 ((EQUAL 'VARIABLE POS)
                                                  (FORMAT NIL "Top = Previous . Bottom +
~,2F cm"
                                                          (SLOT-VALUE COMPOSED-BOX
                                                                      'LEADING-SEPARATION)))
                                                 (T (FORMAT NIL "Top = Enclosing . Top +
~,2F cm"
                                                            (- (CDR POS)
                                                               (* (SLOT-VALUE COMPOSED-BOX
                                                                              'HEIGHT)
                                                                  SCALE))))))
            (FORMAT STREAM "~8tSize: ~A;~%" (SLOT-VALUE COMPOSED-BOX 'SIZE))
            (FORMAT STREAM "~8tFont: ~A;~%" (SLOT-VALUE COMPOSED-BOX 'FONT))
            (FORMAT STREAM "~8tStyle: ~A;~%" (SLOT-VALUE COMPOSED-BOX 'STYLE))
            (FORMAT STREAM "~8tIndent: ~A;~%" (SLOT-VALUE COMPOSED-BOX 'INDENT))
            (FORMAT STREAM "~8tJustify: ~A;~%" (SLOT-VALUE COMPOSED-BOX 'JUSTIFY))
            (FORMAT STREAM "~8tLineSpacing: ~A;~%" (SLOT-VALUE COMPOSED-BOX 'LINESPACING))
            (FORMAT STREAM "~8tAdjust: ~A;~%" (SLOT-VALUE COMPOSED-BOX 'ADJUST))
            (FORMAT STREAM "~8tUnderline: ~A;~%" (SLOT-VALUE COMPOSED-BOX 'UNDERLINE))
            (FORMAT STREAM "~8tThickness: ~A;~%" (SLOT-VALUE COMPOSED-BOX 'THICKNESS))
            (FORMAT STREAM "~8tEND;~2%")
            (LOOP FOR X IN (GET-SUBORDINATES COMPOSED-BOX)
                  DO
                  (SETQ POSX (* (CAADR X)
                                SCALE))
                  (SETQ POSY (* (- P-HEIGHT (CDADR X))
                                SCALE))
                  (EXPORT-P (CAR X)
                            STREAM
                            (CONS POSX POSY))))))

(DEFMETHOD EXPORT-P ((BOX BOX)
                     STREAM &OPTIONAL POS)
   (DECLARE (SPECIAL *PRINTED-OBJECTS*))
   (UNLESS (FIND (NAME BOX)
                 *PRINTED-OBJECTS*)
       (PUSH (NAME BOX)
             *PRINTED-OBJECTS*)
       (LET ((SCALE (SECOND (MEMBER 'SCALE-FACTOR (SLOT-VALUE BOX 'DISPLAY-PARAMETERS)))))
            (FORMAT STREAM "~6t~A:~%" (STRING-UPCASE (NAME BOX)))
            (FORMAT STREAM "~8tBEGIN~%")
            (FORMAT STREAM "~8tWidth: ~A;~%" (CASE (SLOT-VALUE BOX 'OP-WIDTH)
                                                   (FIXED (FORMAT NIL "~,2F cm"
                                                                  (* (SLOT-VALUE BOX 'WIDTH)
                                                                     SCALE)))
                                                   (MAXSIZE "Enclosing . Width")
                                                   (MINSIZE "Enclosed . Width")))
            (FORMAT STREAM "~8tHeight: ~A;~%" (CASE (SLOT-VALUE BOX 'OP-HEIGHT)
                                                    (FIXED (FORMAT NIL "~,2F cm"
                                                                   (* (SLOT-VALUE BOX 'HEIGHT)
                                                                      SCALE)))
                                                    (MAXSIZE "Enclosing . Height")
                                                    (MINSIZE "Enclosed . Height")))
            (FORMAT STREAM "~8tHorizPos: ~A;~%" (COND
                                                  ((EQUAL 'VARIABLE POS)
                                                   (FORMAT NIL "Left = Enclosing . Left +
~,2F cm"
                                                           (SLOT-VALUE BOX
'LEFT-HAND-OFFSET)))
                                                  (T (FORMAT NIL "Left = Enclosing . Left +
~,2F cm"
                                                     (CAR POS)))))
            (FORMAT STREAM "~8tVertPos: ~A;~%" (COND
                                                 ((EQUAL 'VARIABLE POS)
                                                  (FORMAT NIL "Top = Previous . Bottom +
~,2F cm"
                                                          (SLOT-VALUE BOX
'LEADING-SEPARATION)))
                                                 (T (FORMAT NIL "Top = Enclosing . Top +
~,2F cm"
                                                            (- (CDR POS)
                                                               (* (SLOT-VALUE BOX 'HEIGHT)
```

```
                                                                    SCALE))))))
            (FORMAT STREAM "~8tSize: ~A;~%" (SLOT-VALUE BOX 'SIZE))
            (FORMAT STREAM "~8tFont: ~A;~%" (SLOT-VALUE BOX 'FONT))
            (FORMAT STREAM "~8tStyle: ~A;~%" (SLOT-VALUE BOX 'STYLE))
            (FORMAT STREAM "~8tIndent: ~A;~%" (SLOT-VALUE BOX 'INDENT))
            (FORMAT STREAM "~8tJustify: ~A;~%" (SLOT-VALUE BOX 'JUSTIFY))
            (FORMAT STREAM "~8tLineSpacing: ~A;~%" (SLOT-VALUE BOX 'LINESPACING))
            (FORMAT STREAM "~8tAdjust: ~A;~%" (SLOT-VALUE BOX 'ADJUST))
            (FORMAT STREAM "~8tUnderline: ~A;~%" (SLOT-VALUE BOX 'UNDERLINE))
            (FORMAT STREAM "~8tThickness: ~A;~%" (SLOT-VALUE BOX 'THICKNESS))
            (FORMAT STREAM "~8tEND;~2%"))))

(DEFMETHOD EXPORT-P ((PAGE PAGE)
                     STREAM &OPTIONAL ARG)

;;; écrit la description de la page en langage P dans le STREAM

   (LET* ((SCALE (SECOND (MEMBER 'SCALE-FACTOR (SLOT-VALUE PAGE 'DISPLAY-PARAMETERS))))
          (WIDTH (* (SLOT-VALUE PAGE 'WIDTH)
                    SCALE))
          (HEIGHT (* (SLOT-VALUE PAGE 'HEIGHT)
                     SCALE)))
         (FORMAT STREAM "~6t~A:~%" (STRING-UPCASE (NAME PAGE)))
         (FORMAT STREAM "~8tBEGIN~%")
         (FORMAT STREAM "~8tWidth: ~,2F cm;~%" WIDTH)
         (FORMAT STREAM "~8tHeight: ~,2F cm;~%" HEIGHT)
         (FORMAT STREAM "~8tHorizPos: Left = Enclosing . Left;~%")
         (FORMAT STREAM "~8tVertPos: Top = Enclosing . Top;~%")
         (FORMAT STREAM "~8tEND;~2%")))

(DEFMETHOD EXPORT-P ((ELEMENT COMPOSED-LOGICAL-ELEMENT)
                     (GUIDE GUIDE)
                     &OPTIONAL STREAM)

;;; lance la méthode qui permet d'afficher sur le STREAM la description de la boîte qui
correspond à cet élément et effectue les appels récursifs aux sous-elements

   (LET ((CORRESPONDING-BOX (OR (LOOP FOR X IN (PCL::CLASS-SLOT-VALUE (FIND-CLASS 'BOX)
                                                                      'INSTANCES)
                                      THEREIS
                                      (WHEN (EQUAL (NAME X)
                                                   (NAME ELEMENT))
                                            X))
                                (LOOP FOR X IN (PCL::CLASS-SLOT-VALUE (FIND-CLASS
'COMPOSED-BOX)
                                                                      'INSTANCES)
                                      THEREIS
                                      (WHEN (EQUAL (NAME X)
                                                   (NAME ELEMENT))
                                            X))))
         (SUBORDINATES (GET-ORDERED-SUBORDINATES ELEMENT))

         ;; retourne la boîte qui correspond a l'objet logique

         )
        (WHEN CORRESPONDING-BOX
            (EXPORT-P CORRESPONDING-BOX STREAM 'VARIABLE))
        (LOOP FOR X IN SUBORDINATES DO (EXPORT-P (REFERENCE (CAR X)
                                                            GUIDE T)
                                                 GUIDE STREAM))))

(DEFMETHOD EXPORT-P ((ELEMENT BASIC-LOGICAL-ELEMENT)
                     (GUIDE GUIDE)
                     &OPTIONAL STREAM)

;;; lance la méthode qui permet d'afficher sur le STREAM la description de la boîte qui
correspond à cet élément

   (LET ((CORRESPONDING-BOX (OR (LOOP FOR X IN (PCL::CLASS-SLOT-VALUE (FIND-CLASS 'BOX)
                                                                      'INSTANCES)
                                      THEREIS
                                      (WHEN (EQUAL (NAME X)
                                                   (NAME ELEMENT))
                                            X))
                                (LOOP FOR X IN (PCL::CLASS-SLOT-VALUE (FIND-CLASS
'COMPOSED-BOX)
```

```
                                                    'INSTANCES)
                                        THEREIS
                                        (WHEN (EQUAL (NAME X)
                                                     (NAME ELEMENT))
                                              X))))

         ;; retourne la boite qui correspond a l'objet logique

         )
        (WHEN CORRESPONDING-BOX
           (EXPORT-P CORRESPONDING-BOX STREAM 'VARIABLE))))

(DEFMETHOD EXPORT-P ((OBJECT ELEMENT-TYPE)
                     (GUIDE GUIDE)
                     &OPTIONAL STREAM)

;;; si on arrive sur cette methode, c'est une erreur

   (LET ((OBJECT-NAME (NAME OBJECT)))
        (PRINT-ERRORS (LIST (FORMAT NIL "Description of ~A incomplete" (NAME GUIDE))
                            (FORMAT NIL "Export P Aborted")))))

(DEFMETHOD EXPORT-SGML ((OBJECT BASIC-LOGICAL-ELEMENT)
                        (MODEL GUIDE)
                        STREAM)

;;; Ecrit la description de l'objet en format SGML dans le stream

   (DECLARE (SPECIAL *PRINTED-OBJECTS*))
   (LET ((OBJECT-NAME (NAME OBJECT)))
        (COND
           ((ASSOC OBJECT *PRINTED-OBJECTS*)
            (INCF (THIRD (ASSOC OBJECT *PRINTED-OBJECTS*))))
           (T (PUSH (LIST OBJECT (FORMAT STREAM "<!ELEMENT ~A~20T- - " OBJECT-NAME)
                          1)
                    *PRINTED-OBJECTS*)
              (FORMAT STREAM "~A~%" (CASE (GET-CONTENT OBJECT)
                                          (TEXT "(#PCDATA)>")
                                          (GRAPHIC (FORMAT NIL
                                                           "EMPTY>~%<!ATTLIST ~A~%~9T grfile ENTITY
#IMPLIED>"
                                                           OBJECT-NAME)))))))))

(DEFMETHOD EXPORT-SGML ((OBJECT COMPOSED-LOGICAL-ELEMENT)
                        (GUIDE GUIDE)
                        STREAM)

;;; Ecrit la description de l'objet en format SGML dans le stream

   (DECLARE (SPECIAL *PRINTED-OBJECTS*))
   (LET ((OBJECT-NAME (NAME OBJECT)))
        (COND
           ((ASSOC OBJECT *PRINTED-OBJECTS*)
            (INCF (THIRD (ASSOC OBJECT *PRINTED-OBJECTS*))))
           (T (PUSH (LIST OBJECT (FORMAT STREAM "<!ELEMENT ~A~20T- - (" (IF (EQUAL
OBJECT-NAME "Root"
                                                                            )
                                                                            (NAME GUIDE)
                                                                            OBJECT-NAME))
                          1)
                    *PRINTED-OBJECTS*)
              (LET* ((SUBORDINATES (GET-ORDERED-SUBORDINATES OBJECT))
                     (INHERITED-SUB (GET-ORDERED-SUBORDINATES (KIND-OF OBJECT)))
                     (CONSTRUCTOR (GET-CONSTRUCTOR OBJECT))
                     (SEP (CASE CONSTRUCTOR
                                (SEQUENCE ", ")
                                (CHOICE " | ")
                                (AGGREGATE " & ")))
                     (CTR -1))
                    (COND
                       (SUBORDINATES (LOOP FOR X IN SUBORDINATES DO (FORMAT
                                                                     STREAM "~A~A~A"
                                                                     (IF (EQUAL X (CAR
SUBORDINATES))
                                                                         ""
                                                                         SEP)
```

```
                                                                    (NAME (CAR X))
                                                                    (CASE (SECOND X)
                                                                        (? "?")
                                                                        (* "*")
                                                                        (+ "+")
                                                                        (T "")))))
                                    (FORMAT STREAM ")>~%")
                                    (LOOP FOR X IN SUBORDINATES DO (EXPORT-SGML (REFERENCE (CAR X)
                                                                                           GUIDE T)
                                                                                GUIDE STREAM)))
                            (T (FORMAT STREAM "#PCDATA)>~%")))))))))

(DEFMETHOD EXPORT-SGML ((OBJECT ELEMENT-TYPE)
                        (GUIDE GUIDE)
                        STREAM)

;;; Crée un fichier contenant la description SGML du modèle

   (LET ((OBJECT-NAME (NAME OBJECT)))
        (PRINT-ERRORS (LIST (FORMAT NIL "Description of ~A incomplete" (NAME GUIDE))
                            (FORMAT NIL "Export SGML Aborted")))))

(DEFMETHOD FM-ABORT :AROUND
   ((FM-WINDOW COMPONENT-DESCRIPTION))
   (CALL-NEXT-METHOD)
   (REPAINT (GETWINDOWPROP FM-WINDOW 'EDIT-WINDOW)))

(DEFMETHOD FM-APPLY ((FM-WINDOW COMPONENT-DESCRIPTION))
   (LET* ((WINDOW (GETWINDOWPROP FM-WINDOW 'EDIT-WINDOW))
          (STATE (GET-STATE FM-WINDOW))
          (NAME (SECOND (MEMBER :NAME STATE)))
          (CONSTRUCTOR (SECOND (MEMBER :CONSTRUCTOR STATE)))
          (CONDITION (SECOND (MEMBER :CONDITION STATE)))
          (COMPONENT)
          (OBJECT (GETWINDOWPROP FM-WINDOW 'EDITED-OBJECT))
          (SUBORDINATE-REF))
         (SETQ CONSTRUCTOR (COND
                             ((STRING= CONSTRUCTOR "Optional (?)")
                              '?)
                             ((STRING= CONSTRUCTOR "Repeat (+)")
                              '+)
    SETQ OP-HEIGHT 'FIXED))
            ((AND (EQUAL OP-HEIGHT 'FIXED)
                  ((AND (EQUAL OP-HEIGHT                                ((STRING= CONSTRUCTOR
"Mandatory")
                        NIL)))
         (SETQ CONDITION (COND
                           ((STRING= CONDITION "")
                            NIL)
                           (T CONDITION)))
         (LOOP FOR X IN (PCL::CLASS-SLOT-VALUE (FIND-CLASS 'ELEMENT-TYPE)
                                               'INSTANCES)
               THEREIS
               (WHEN (EQUAL (NAME X)
                            NAME)
                     (SETQ SUBORDINATE-REF X)))
         (SETF (CONTEXT (EDITED-MODEL WINDOW))
               (CONS (LIST SUBORDINATE-REF (REFERENCE SUBORDINATE-REF (EDITED-MODEL WINDOW)
                                                      T))
                     (CONTEXT (EDITED-MODEL WINDOW))))

         ;; Si l'objet n'existe pas, on le crée

         (SETQ COMPONENT (LIST SUBORDINATE-REF CONSTRUCTOR CONDITION))
         (CASE (GETWINDOWPROP FM-WINDOW 'OPERATION)
             (ADD-SUBORDINATE (ADD-SUBORDINATE (KIND-OF* OBJECT)
                                               (EDITED-MODEL WINDOW)
                                               COMPONENT))
             (CHANGE-SUBORDINATE (CHANGE-SUBORDINATE (KIND-OF* OBJECT)
                                                     (EDITED-MODEL WINDOW)
                                                     (GETWINDOWPROP FM-WINDOW 'EDITED-SUBORDINATE)
                                                     COMPONENT)))
         (CLOSEW FM-WINDOW)
         (REPAINT* WINDOW)))

(DEFMETHOD FO-DNIK* ((OBJECT ELEMENT-TYPE))
```

```
   ;; Renvoie la liste des descendants de l'objet obtenue en appliquant récursivement la
fonction FO-DNIK

   (LOOP FOR X IN (FO-DNIK OBJECT)
         APPEND
         (APPEND (LIST X)
                 (FO-DNIK* X))))

(DEFMETHOD GET-COMPOSED-OF ((OBJECT COMPOSED-LOGICAL-ELEMENT))

;;; retourne la composition de l'objet

   (CONS (GET-CONSTRUCTOR OBJECT)
         (GET-SUBORDINATES OBJECT)))

(DEFMETHOD GET-CONSTRUCTOR ((OBJECT COMPOSED-LOGICAL-ELEMENT))

;;; retourne le constructeur de OBJECT

   (OR (CONSTRUCTOR OBJECT)
       (GET-CONSTRUCTOR (KIND-OF OBJECT))))

(DEFMETHOD GET-CONSTRUCTOR ((OBJECT ELEMENT-TYPE))
   'SEQUENCE)

(DEFMETHOD GET-CONTENT ((OBJECT BASIC-LOGICAL-ELEMENT))

;;; Retourne le contenu de OBJECT

   (OR (CONTENT OBJECT)
       (GET-CONTENT (KIND-OF OBJECT))))

(DEFMETHOD GET-CONTENT ((OBJECT ELEMENT-TYPE))
   'TEXT)

(DEFMETHOD GET-ORDERED-SUBORDINATES ((OBJECT COMPOSED-LOGICAL-ELEMENT))

;;; retourne la liste ordonnée des composants de l'objet

   (LET ((SUBORDINATES (GET-ORDERED-SUBORDINATES (KIND-OF OBJECT)))
         (REORDER-LIST (REORDER-LIST OBJECT)))
        (LOOP FOR X IN (SUBORDINATES OBJECT)
              DO
              (LET ((OP (CAR X))
                    (OBJ (SECOND X)))
                   (COND
                      ((EQUAL OP 'REMOVE)
                       (SETQ SUBORDINATES (REMOVE OBJ SUBORDINATES :KEY #'CAR :COUNT 1)))
                      ((EQUAL OP 'ADD)
                       (SETQ SUBORDINATES (APPEND SUBORDINATES (LIST (CDR X)))))
                      ((EQUAL OP 'REPLACE)
                       (SETQ SUBORDINATES (SUBST (CDDR X)
                                                 OBJ SUBORDINATES :KEY #'CAR))))))
        (IF REORDER-LIST
            (LET ((ORDERED-SUBORDINATES (LOOP FOR X IN REORDER-LIST COLLECT (NTH (- X 1)

SUBORDINATES))))
                 (APPEND ORDERED-SUBORDINATES (SET-DIFFERENCE SUBORDINATES
ORDERED-SUBORDINATES)))
            SUBORDINATES)))

(DEFMETHOD GET-ORDERED-SUBORDINATES ((OBJECT ELEMENT-TYPE)))

(DEFMETHOD GET-SUBORDINATES ((OBJECT COMPOSED-LOGICAL-ELEMENT))

;;; retourne les composants de l'objet

   (LET ((SUBORDINATES (GET-SUBORDINATES (KIND-OF OBJECT))))
        (LOOP FOR X IN (SUBORDINATES OBJECT)
              DO
              (LET ((OP (CAR X))
                    (OBJ (SECOND X)))
                   (COND
                      ((EQUAL OP 'REMOVE)
                       (SETQ SUBORDINATES (REMOVE OBJ SUBORDINATES :KEY #'CAR :COUNT 1)))
```

```
                    ((EQUAL OP 'ADD)
                     (SETQ SUBORDINATES (APPEND SUBORDINATES (LIST (CDR X)))))
                    ((EQUAL OP 'REPLACE)
                     (SETQ SUBORDINATES (SUBST (CDDR X)
                                               OBJ SUBORDINATES :KEY #'CAR))))))
        SUBORDINATES))

(DEFMETHOD GET-SUBORDINATES ((OBJECT ELEMENT-TYPE)))

(DEFMETHOD GET-SUBORDINATES ((OBJECT COMPOSED-ELEMENT))

;;; retourne les composants de l'objet

   (LET ((SUBORDINATES (WHEN (KIND-OF OBJECT)
                             (GET-SUBORDINATES (KIND-OF OBJECT)))))
        (LOOP FOR X IN (SUBORDINATES OBJECT)
              DO
              (LET ((OP (CAR X))
                    (OBJ (SECOND X)))
                   (COND
                      ((EQUAL OP 'REMOVE)
                       (SETQ SUBORDINATES (REMOVE OBJ SUBORDINATES :KEY #'CAR :COUNT 1)))
                      ((EQUAL OP 'ADD)
                       (SETQ SUBORDINATES (APPEND SUBORDINATES (LIST (CDR X)))))
                      ((EQUAL OP 'REPLACE)
                       (SETQ SUBORDINATES (SUBST (CDDR X)
                                                 OBJ SUBORDINATES :KEY #'CAR))))))
        SUBORDINATES))

(DEFMETHOD GET-SUBORDINATES ((OBJECT BOX))

;;; Pas de sous objets pour les objets physiques de base

   )

(DEFMETHOD GET-SUPERORDINATES ((NODE BOX)
                               OBJECT)

;;; On cherche si Object est un sous objet de NODE

;;; Faux car NODE n'a pas de sous objets

   )

(DEFMETHOD INITIALIZE-INSTANCE :AROUND
   ((MODEL GUIDE)
    &KEY)
   (CALL-NEXT-METHOD)
   (LET ((W CREATEW '(0 0 52 75)
            NIL 1)))
        DSPFONT FONTCREATE 'MODERN 10 'BRR)
                W)
        (FORMAT W "~A~%" (NAME MODEL))
        DSPFONT FONTCREATE 'HELVETICA 4 'MRR)
                W)
        (LOOP FOR X FROM 1 TO 10 DO (FORMAT W "hfjdskal fjsa kfhjkfh askjf hskjfsdha"))
        (SETF (ICON MODEL)
              BITMAPCREATE 52 75))
        BITBLT IL:|ScreenBitMap| 0 0 (ICON MODEL)
               0 0 52 75)
        CLOSEW W)))

(DEFMETHOD KIND-OF ((OBJECT COMPOSED-BOX)))

(DEFMETHOD KIND-OF* ((OBJECT ELEMENT-TYPE))
   (IF (KIND-OF OBJECT)
       (KIND-OF* (KIND-OF OBJECT))
       OBJECT))

(DEFMETHOD KIND-OF? ((MODEL1 GUIDE)
                     (MODEL2 GUIDE)
                     &OPTIONAL RECURSEFLG)

;;; Indique si MODEL1 hérite de MODEL2 par la relation KIND-OF.

;;; Si RECURSEFLG est différent de NIL, la relation KIND-OF est appliquée plusieurs fois
```

```
   (LET ((KIND-OF (KIND-OF MODEL1)))
        (OR (EQUAL KIND-OF MODEL2)
            (AND KIND-OF RECURSEFLG (KIND-OF? KIND-OF MODEL2 T)))))

(DEFMETHOD KIND-OF? ((OBJECT1 ELEMENT-TYPE)
                     (OBJECT2 ELEMENT-TYPE)
                     &OPTIONAL RECURSEFLG)

;;; Indique si OBJECT1 hérite de OBJECT2 par la relation KIND-OF.

;;; Si RECURSEFLG est différent de NIL, la relation KIND-OF est appliquée plusieurs fois

   (LET ((KIND-OF (KIND-OF OBJECT1)))
        (OR (EQUAL KIND-OF OBJECT2)
            (AND KIND-OF RECURSEFLG (KIND-OF? KIND-OF OBJECT2 T)))))

(DEFMETHOD LEFT-BUTTON-EVENT ((WINDOW GUIDE-EDITOR))

;;; Traite l'appui sur le bouton du milieu de la souris

   (LET ((SELECTED-OBJECT '(NOTHING (0 0 0 0)))
         (NEW-SELECTED-OBJECT)
         (OLD-POS '(0 . 0))
         (NEW-POS))
        (LOOP WHILE MOUSESTATE IL:LEFT)
              DO
              (SETQ NEW-POS (GET-CURSOR-POSITION WINDOW))
              (SETQ NEW-SELECTED-OBJECT (LOOP FOR X IN (EDITED-OBJECTS WINDOW)
                                              THEREIS
                                              (WHEN INSIDEP (SECOND X)
                                                            NEW-POS)
                                                    X)
                                              FINALLY
                                              (RETURN '(NOTHING (0 0 0 0)))))
              (UNLESS (EQUAL SELECTED-OBJECT NEW-SELECTED-OBJECT)
                  (DRAW-REGION WINDOW (SECOND SELECTED-OBJECT)
                         :OPERATION
                         'IL:ERASE)
                  (DRAW-REGION WINDOW (SECOND NEW-SELECTED-OBJECT)
                         :OPERATION
                         'IL:PAINT)
                  (SETQ SELECTED-OBJECT NEW-SELECTED-OBJECT)
                  (SETQ OLD-POS NEW-POS)))
        (UNLESS (EQUAL (CAR SELECTED-OBJECT)
                       'NOTHING)
            (MODIFY (REFERENCE (CAR SELECTED-OBJECT)
                               (EDITED-MODEL WINDOW)
                               T)
                    WINDOW))))

(DEFMETHOD MEMORIZE-ACTIVE-REGION ((OBJECT HEADER)
                                   (WINDOW LAYOUT-MODEL-EDITOR)
                                   WINDOW-REGION POSITION)

;;; Mémorise les régions actives associées à l'entête

   (PUTWINDOWPROP WINDOW 'REGIONS (CONS (LIST WINDOW-REGION OBJECT WINDOW-REGION POSITION
                                              'ADD-SUBOBJECT)
                                        (GETWINDOWPROP WINDOW 'REGIONS)))
                                                    ; Mémo de l'ensemble de l'objet
   (PUTWINDOWPROP WINDOW 'REGIONS (CONS (LIST (MAKE-REGION :LEFT (REGION-LEFT WINDOW-REGION)
                                                           :BOTTOM
                                                           (- (REGION-BOTTOM WINDOW-REGION)
                                                              2)
                                                           :WIDTH
                                                           (REGION-WIDTH WINDOW-REGION)
                                                           :HEIGHT 5)
                                              OBJECT WINDOW-REGION POSITION 'MOVE)
                                        (GETWINDOWPROP WINDOW 'REGIONS)))
                                                    ; Mémo du bas de l'entête,
région active permettant le déplacement
   )

(DEFMETHOD MEMORIZE-ACTIVE-REGION ((OBJECT FOOTER)
                                   (WINDOW LAYOUT-MODEL-EDITOR)
```

```
                                        WINDOW-REGION POSITION)

;;; Mémorise les régions actives associées au pied de page

   (PUTWINDOWPROP WINDOW 'REGIONS (CONS (LIST WINDOW-REGION OBJECT WINDOW-REGION POSITION
                                              'ADD-SUBOBJECT)
                                        (GETWINDOWPROP WINDOW 'REGIONS)))
                                                    ; Mémo de l'ensemble de l'objet
   (PUTWINDOWPROP WINDOW 'REGIONS (CONS (LIST (MAKE-REGION :LEFT (REGION-LEFT WINDOW-REGION)
                                                           :BOTTOM
                                                           (- (REGION-TOP WINDOW-REGION)
                                                              2)
                                                           :WIDTH
                                                           (REGION-WIDTH WINDOW-REGION)
                                                           :HEIGHT 5)
                                              OBJECT WINDOW-REGION POSITION 'MOVE)
                                        (GETWINDOWPROP WINDOW 'REGIONS)))
                                                    ; Mémorisation du haut du pied
de page pour permettre le déplacement
   )

(DEFMETHOD MEMORIZE-ACTIVE-REGION ((OBJECT PAGE)
                                   (WINDOW LAYOUT-MODEL-EDITOR)
                                   WINDOW-REGION POSITION)
   (PUTWINDOWPROP WINDOW 'REGIONS NIL)
   (PUTWINDOWPROP WINDOW 'PAGE OBJECT))

(DEFMETHOD MEMORIZE-ACTIVE-REGION ((OBJECT BOX)
                                   (WINDOW LAYOUT-MODEL-EDITOR)
                                   WINDOW-REGION POSITION)

;;; mémorise la région active associée à l'objet, son nom, la région entière occupée par
l'objet sur la fenêtre, sa position et la fonction appelée lorsque cette région est
sélectionnée

   (PUTWINDOWPROP WINDOW 'REGIONS (CONS (LIST WINDOW-REGION OBJECT WINDOW-REGION POSITION
                                              'MOVE)
                                        (GETWINDOWPROP WINDOW 'REGIONS))))

(DEFMETHOD MODIFY ((OBJECT BASIC-LOGICAL-ELEMENT)
                   (WINDOW GUIDE-EDITOR))

;;; Permet à l'utilisateur de modifier l'objet sélectionné

   (LET ((CHOICE (EXECUTE (MAKE-INSTANCE 'MENU :TITLE " Modify Option(s) " :ITEMS
                                '(("Change Content" 'CHANGE-CONTENT)
                                  ("Convert to Composed" 'CONVERT-TO-COMPOSED))
                                :CENTERFLG T))))
        (CASE CHOICE
            (CHANGE-CONTENT
               (SETQ CHOICE (EXECUTE (MAKE-INSTANCE 'MENU :TITLE " Possible Contents " :ITEMS
                                        '(("Type TEXT" 'TEXT)
                                          ("Type GRAPHIC" 'GRAPHIC)
                                          ("Fixed Content" 'FIXED)
                                          ("Variable Content" 'VARIABLE)))))
               (WHEN CHOICE
                   (WHEN (EQUAL CHOICE 'FIXED)
                       (SETQ CHOICE (PROMPT-FOR-CONTENT)))
                   (CHANGE-CONTENT (KIND-OF* OBJECT)
                          (EDITED-MODEL WINDOW)
                          CHOICE))
               (REPAINT* WINDOW))
            (CONVERT-TO-COMPOSED (LET ((NEW-OBJECT (MAKE-INSTANCE 'COMPOSED-LOGICAL-ELEMENT))
                                       (GENERIC-OBJECT (KIND-OF* OBJECT)))
                                   (SETF (CONSTRUCTOR NEW-OBJECT)
                                         'SEQUENCE)
                                   (SETF (KIND-OF NEW-OBJECT)
                                         GENERIC-OBJECT)
                                   (PUSH NEW-OBJECT (FO-DNIK GENERIC-OBJECT))
                                   (SETF (CONTEXT (EDITED-MODEL WINDOW))
                                         (SUBST (LIST GENERIC-OBJECT NEW-OBJECT)
                                                GENERIC-OBJECT
                                                (CONTEXT (EDITED-MODEL WINDOW))
                                                :KEY
                                                #'CAR))
                                   (REPAINT* WINDOW)))
```

```
            (T (REPAINT WINDOW)))))

(DEFMETHOD MODIFY ((OBJECT COMPOSED-LOGICAL-ELEMENT)
                   (WINDOW GUIDE-EDITOR))

;;; Permet à l'utilisateur de modifier l'objet sélectionné

   (DECLARE (SPECIAL IL:FM-GENERIC-COMPONENT-DESCRIPTION))
   (SETF (NTH 13 (NTH 3 IL:FM-GENERIC-COMPONENT-DESCRIPTION))
         (LOOP FOR X IN (PCL::CLASS-SLOT-VALUE (FIND-CLASS 'ELEMENT-TYPE)
                                               'INSTANCES)
               UNLESS
               (EQUAL (NAME X)
                      "<<New Object>>")
               COLLECT
               (NAME X)))

   ;; Modification du free-menu de manière à proposer un choix parmi tous les objets
existants

   (LET* ((LIST-SUB (GET-SUBORDINATES OBJECT))
          (MENU (MAKE-INSTANCE 'MENU :TITLE " Modify Option(s) " :ITEMS
                       (APPEND '(("Add a Component" 'ADD-SUBORDINATE))
                               (WHEN LIST-SUB
                                   '(("Remove a Component" 'REMOVE-SUBORDINATE)
                                     ("Remove all Components" 'REMOVE-SUBORDINATES)
                                     ("Change a Component" 'CHANGE-SUBORDINATE)
                                     ("Change the Connector" 'CHANGE-CONSTRUCTOR)))
                               (WHEN (AND (> (LENGTH LIST-SUB)
                                             1)
                                          (EQUAL (GET-CONSTRUCTOR OBJECT)
                                                 'SEQUENCE))
                                   '(("Reorder" 'REORDER)))
                               '(("Convert to Basic" 'CONVERT-TO-BASIC)))
                       :CENTERFLG T))
          (FM (MAKE-INSTANCE 'COMPONENT-DESCRIPTION :DESCRIPTION
IL:FM-GENERIC-COMPONENT-DESCRIPTION
                     :TITLE "Description"))
          (CHOICE))
         (MOVEW FM IL:LASTMOUSEX IL:LASTMOUSEY)
         (SETQ CHOICE (EXECUTE MENU))
         (PUTWINDOWPROP FM 'OPERATION CHOICE)
         (PUTWINDOWPROP FM 'EDIT-WINDOW WINDOW)
         (PUTWINDOWPROP FM 'EDITED-OBJECT OBJECT)
         (CASE CHOICE
             (ADD-SUBORDINATE (OPENW FM))
             (CHANGE-SUBORDINATE
                (SETQ CHOICE (EXECUTE (MAKE-INSTANCE 'MENU :TITLE "Which one ?" :ITEMS
                                            (LOOP FOR X IN LIST-SUB COLLECT
                                                  (LIST (NAME (IF (LISTP X)
                                                                  (CAR X)
                                                                  X))
                                                        '',X)))))
                (WHEN CHOICE
                    (UNLESS (LISTP CHOICE)
                        (SETQ CHOICE (LIST CHOICE)))
                    (CHANGE-STATE FM :NAME (NAME (CAR CHOICE)))
                    (CHANGE-STATE FM :CONSTRUCTOR (CASE (SECOND CHOICE)
                                                        (? "Optional (?)")
                                                        (+ "Repeat (+)")
                                                        (* "Optional Repeat (*)")
                                                        ((NIL) "Mandatory")))
                    (PUTWINDOWPROP FM 'EDITED-SUBORDINATE (CAR CHOICE))
                    (OPENW FM)))
             (REMOVE-SUBORDINATE
                (SETQ CHOICE (EXECUTE (MAKE-INSTANCE 'MENU :TITLE "Which one ?" :ITEMS
                                            (LOOP FOR X IN LIST-SUB COLLECT
                                                  (LET ((OBJ (IF (LISTP X)
                                                                 (CAR X)
                                                                 X)))
                                                       (LIST (NAME OBJ)
                                                             '',OBJ)))))))
                (REMOVE-SUBORDINATE (KIND-OF* OBJECT)
                       (EDITED-MODEL WINDOW)
                       CHOICE)
                (REPAINT* WINDOW))
```

```
            (REMOVE-SUBORDINATES
               (LOOP FOR X IN LIST-SUB DO (REMOVE-SUBORDINATE (KIND-OF* OBJECT)
                                                  (EDITED-MODEL WINDOW)
                                                  (IF (LISTP X)
                                                      (CAR X)
                                                      X)))
               (REPAINT* WINDOW))
            (CHANGE-CONSTRUCTOR
               (SETQ CHOICE (EXECUTE (MAKE-INSTANCE 'MENU :TITLE "Select new one" :ITEMS
                                                '(("Aggregate (AGR)" 'AGGREGATE)
                                                  ("Sequence (SEQ)" 'SEQUENCE)
                                                  ("Choice (CHO)" 'CHOICE)))))
               (WHEN CHOICE
                   (CHANGE-CONSTRUCTOR (KIND-OF* OBJECT)
                            (EDITED-MODEL WINDOW)
                            CHOICE))
               (REPAINT* WINDOW))
            (CONVERT-TO-BASIC (LET ((NEW-OBJECT (MAKE-INSTANCE 'BASIC-LOGICAL-ELEMENT))
                                    (GENERIC-OBJECT (KIND-OF* OBJECT)))
                                (SETF (CONTENT NEW-OBJECT)
                                      'TEXT)
                                (SETF (KIND-OF NEW-OBJECT)
                                      GENERIC-OBJECT)
                                (PUSH NEW-OBJECT (FO-DNIK GENERIC-OBJECT))
                                (SETF (CONTEXT (EDITED-MODEL WINDOW))
                                      (SUBST (LIST GENERIC-OBJECT NEW-OBJECT)
                                             GENERIC-OBJECT
                                             (CONTEXT (EDITED-MODEL WINDOW))
                                             :KEY
                                             #'CAR))
                                (REPAINT* WINDOW)))
            (REORDER
               (REORDER-SUBORDINATES (KIND-OF* OBJECT)
                        (EDITED-MODEL WINDOW))
               (REPAINT* WINDOW))
            (T (REPAINT WINDOW)))))

(DEFMETHOD MODIFY ((OBJECT ELEMENT-TYPE)
                   (WINDOW GUIDE-EDITOR))

;;; Permet à l'utilisateur de modifier l'objet sélectionné

   (LET ((CHOICE (EXECUTE (MAKE-INSTANCE 'MENU :TITLE " Modify Option(s) " :ITEMS
                                (IF (REFERENCE OBJECT (EDITED-MODEL WINDOW))
                                    '(("Make Composed Element" 'MAKE-COMPOSED)
                                      ("Make Basic Element" 'MAKE-BASIC))
                                    '(("Modification NOT Allowed" NIL)))
                                :CENTERFLG T)))
         (NEW-OBJECT))
        (COND
           (CHOICE (SETQ NEW-OBJECT (MAKE-INSTANCE (CASE CHOICE
                                                     (MAKE-COMPOSED
'COMPOSED-LOGICAL-ELEMENT)
                                                     (MAKE-BASIC 'BASIC-LOGICAL-ELEMENT))))
                   (CASE CHOICE
                       (MAKE-COMPOSED (SETF (CONSTRUCTOR NEW-OBJECT)
                                            'AGGREGATE))
                       (MAKE-BASIC (SETF (CONTENT NEW-OBJECT)
                                         'TEXT)))
                   (SETF (KIND-OF NEW-OBJECT)
                         OBJECT)
                   (PUSH NEW-OBJECT (FO-DNIK OBJECT))
                   (SETF (CONTEXT (EDITED-MODEL WINDOW))
                         (SUBST (LIST OBJECT NEW-OBJECT)
                                OBJECT
                                (CONTEXT (EDITED-MODEL WINDOW))
                                :KEY
                                #'CAR))
                   (REPAINT* WINDOW))
           (T (REPAINT WINDOW)))))

(DEFMETHOD MOVE ((OBJECT FOOTER)
                 (WINDOW LAYOUT-MODEL-EDITOR)
                 REGION)

;;; Déplacement de la limite supérieure du bas de page
```

```
   (DECLARE (SPECIAL *MOVE-BITMAP*))
   (LET* ((IL-WINDOW (SLOT-VALUE WINDOW 'IL-WINDOW))
          (INITIAL-YPOS (REGION-TOP REGION))
          (INITIAL-XPOS (CAR (GET-CURSOR-POSITION WINDOW)))
          (OLD-YPOS INITIAL-YPOS)
          (NEW-YPOS INITIAL-YPOS)
          (UPPER-LIMIT)
          (LOWER-LIMIT)
          (REGIONS)
          (LINE-XPOS)
          (LINE-LENGTH))
         (PUT-CURSOR-POSITION WINDOW (CONS INITIAL-XPOS INITIAL-YPOS))
                                                          ; Place le curseur sur la ligne
à bouger
         BITBLT *MOVE-BITMAP* 0 0 IL-WINDOW (- INITIAL-XPOS 8)
                (- INITIAL-YPOS 2)
                NIL NIL 'IL:INPUT 'IL:INVERT)             ; Affiche le symbole move
         (SETQ LINE-XPOS (1+ (REGION-LEFT REGION)))
         (SETQ LINE-LENGTH (- (REGION-WIDTH REGION)
                              3))
         (SETQ REGIONS (LOOP FOR REG-INFO IN (GETWINDOWPROP WINDOW 'REGIONS)
                             COLLECT
                             (THIRD REG-INFO)))           ; crée la liste de toutes les
régions de la fenêtre
         (SETQ UPPER-LIMIT (1- (APPLY #'MIN (LOOP FOR REG IN REGIONS WHEN (> (REGION-BOTTOM
REG)
                                                                             OLD-YPOS)
                                                  COLLECT
                                                  (REGION-BOTTOM REG)))))
         (SETQ LOWER-LIMIT (1+ (APPLY #'MAX (CONS (REGION-BOTTOM REGION)
                                                  (LOOP FOR REG IN REGIONS WHEN (<
(REGION-TOP REG)
                                                                                OLD-YPOS)
                                                        COLLECT
                                                        (REGION-TOP REG))))))
         (LOOP WHILE KEYDOWNP 'IL:LEFT)
               DO
               (SETQ NEW-YPOS (KEEP-IN-LIMITS (CDR (GET-CURSOR-POSITION WINDOW))
                                              LOWER-LIMIT UPPER-LIMIT))
               (UNLESS (= OLD-YPOS NEW-YPOS)              ; A chaque déplacement du
curseur
                   BITBLT *MOVE-BITMAP* 0 0 IL-WINDOW (- INITIAL-XPOS 8)
                          (- OLD-YPOS 2)
                          NIL NIL 'IL:INPUT 'IL:INVERT)   ; Efface le symbole move
                   (MOVE-TO WINDOW (CONS LINE-XPOS OLD-YPOS))
                   (RELDRAW-LINE WINDOW LINE-LENGTH 0 :WIDTH 1 :OPERATION 'IL:ERASE)
                                                          ; Efface la ligne précédente
                   (SETF (REGION-BOTTOM REGION)
                         NEW-YPOS)                        ; Change le bas de la région
                   (MOVE-TO WINDOW (CONS LINE-XPOS NEW-YPOS))
                   (RELDRAW-LINE WINDOW LINE-LENGTH 0 :WIDTH 1 :OPERATION 'IL:PAINT)
                                                          ; Redessine la ligne
                   BITBLT *MOVE-BITMAP* 0 0 IL-WINDOW (- INITIAL-XPOS 8)
                          (- NEW-YPOS 2)
                          NIL NIL 'IL:INPUT 'IL:INVERT)   ; Réaffiche le symbole move
                   (SETQ OLD-YPOS NEW-YPOS)))
         BITBLT *MOVE-BITMAP* 0 0 IL-WINDOW (- INITIAL-XPOS 8)
                (- NEW-YPOS 2)
                NIL NIL 'IL:INPUT 'IL:INVERT)             ; Efface le symbole move
         (LET ((DELTA-Y (* (- NEW-YPOS INITIAL-YPOS)
                           (GETWINDOWPROP WINDOW 'PAGE-SCALE))
                        ))                                ; Déplacement de la ligne
              (SETF (SLOT-VALUE OBJECT 'HEIGHT)
                    (+ (SLOT-VALUE OBJECT 'HEIGHT)
                       DELTA-Y)))                         ; MAJ de l'objet
         (REDISPLAY (GETWINDOWPROP WINDOW 'MODEL)
                    WINDOW)))

(DEFMETHOD MOVE ((OBJECT HEADER)
                 (WINDOW LAYOUT-MODEL-EDITOR)
                 REGION)

;;; Déplacement de la limite inférieure de l'entête
```

```
   (DECLARE (SPECIAL *MOVE-BITMAP*))
   (LET* ((IL-WINDOW (SLOT-VALUE WINDOW 'IL-WINDOW))
          (INITIAL-YPOS (REGION-BOTTOM REGION))
          (INITIAL-XPOS (CAR (GET-CURSOR-POSITION WINDOW)))
          (OLD-YPOS INITIAL-YPOS)
          (NEW-YPOS INITIAL-YPOS)
          (UPPER-LIMIT)
          (LOWER-LIMIT)
          (REGIONS)
          (LINE-XPOS)
          (LINE-LENGTH))
         (PUT-CURSOR-POSITION WINDOW (CONS INITIAL-XPOS INITIAL-YPOS))
                                                          ; Place le curseur sur la ligne
à bouger
         BITBLT *MOVE-BITMAP* 0 0 IL-WINDOW (- INITIAL-XPOS 8)
                (- INITIAL-YPOS 2)
                NIL NIL 'IL:INPUT 'IL:INVERT)             ; Affiche le symbole move
         (SETQ LINE-XPOS (1+ (REGION-LEFT REGION)))
         (SETQ LINE-LENGTH (- (REGION-WIDTH REGION)
                              3))
         (SETQ REGIONS (LOOP FOR REG-INFO IN (GETWINDOWPROP WINDOW 'REGIONS)
                             COLLECT
                             (THIRD REG-INFO)))           ; crée la liste de toutes les
régions de la fenêtre
         (SETQ UPPER-LIMIT (1- (APPLY #'MIN (CONS (REGION-TOP REGION)
                                                  (LOOP FOR REG IN REGIONS WHEN (>
(REGION-BOTTOM
                                                                                  REG)
                                                                                 OLD-YPOS)
                                                        COLLECT
                                                        (REGION-BOTTOM REG))))))
         (SETQ LOWER-LIMIT (1+ (APPLY #'MAX (LOOP FOR REG IN REGIONS WHEN (< (REGION-TOP REG)
                                                                             OLD-YPOS)
                                                  COLLECT
                                                  (REGION-TOP REG)))))
         (PUT-CURSOR-POSITION WINDOW (CONS INITIAL-XPOS INITIAL-YPOS))
                                                          ; Place le curseur sur la ligne
à bouger
         (LOOP WHILE KEYDOWNP 'IL:LEFT)
               DO
               (SETQ NEW-YPOS (KEEP-IN-LIMITS (CDR (GET-CURSOR-POSITION WINDOW))
                                              LOWER-LIMIT UPPER-LIMIT))
               (UNLESS (= OLD-YPOS NEW-YPOS)              ; A chaque déplacement du
curseur
                   BITBLT *MOVE-BITMAP* 0 0 IL-WINDOW (- INITIAL-XPOS 8)
                          (- OLD-YPOS 2)
                          NIL NIL 'IL:INPUT 'IL:INVERT)   ; Efface le symbole move
                   (MOVE-TO WINDOW (CONS LINE-XPOS OLD-YPOS))
                   (RELDRAW-LINE WINDOW LINE-LENGTH 0 :WIDTH 1 :OPERATION 'IL:ERASE)
                                                          ; Efface la ligne précédente
                   (SETF (REGION-BOTTOM REGION)
                         NEW-YPOS)                        ; Change le bas de la région
                   (MOVE-TO WINDOW (CONS LINE-XPOS NEW-YPOS))
                   (RELDRAW-LINE WINDOW LINE-LENGTH 0 :WIDTH 1 :OPERATION 'IL:PAINT)
                                                          ; Redessine la ligne
                   BITBLT *MOVE-BITMAP* 0 0 IL-WINDOW (- INITIAL-XPOS 8)
                          (- NEW-YPOS 2)
                          NIL NIL 'IL:INPUT 'IL:INVERT)   ; Réaffiche le symbole move
                   (SETQ OLD-YPOS NEW-YPOS)))
         BITBLT *MOVE-BITMAP* 0 0 IL-WINDOW (- INITIAL-XPOS 8)
                (- NEW-YPOS 2)
                NIL NIL 'IL:INPUT 'IL:INVERT)             ; Efface le symbole move
         (LET ((DELTA-Y (* (- NEW-YPOS INITIAL-YPOS)
                           (GETWINDOWPROP WINDOW 'PAGE-SCALE))
                                                          ; Déplacement de la ligne
               )
              (SUPERORDINATE (GETWINDOWPROP WINDOW 'PAGE)))
             (SETF (SLOT-VALUE OBJECT 'HEIGHT)
                   (- (SLOT-VALUE OBJECT 'HEIGHT)
                      DELTA-Y))
             (LOOP FOR X IN (SLOT-VALUE SUPERORDINATE 'SUBORDINATES)
                   DO
                   (IF (EQUAL OBJECT (SECOND X))
                       (SETF (CDR (THIRD X))
                             (+ (CDR (THIRD X))
                                DELTA-Y))))
```

```
            (LOOP FOR X IN (SLOT-VALUE OBJECT 'SUBORDINATES)
                  DO
                  (SETF (CDR (THIRD X))
                        (- (CDR (THIRD X))
                           DELTA-Y))))                          ; MAJ de l'objet
        (REDISPLAY (GETWINDOWPROP WINDOW 'MODEL)
                WINDOW)))

(DEFMETHOD MOVE :BEFORE
   ((OBJECT BOX)
    (WINDOW LAYOUT-MODEL-EDITOR)
    REGION)

;;; Efface le nom de l'objet avant un déplacement

   (PRINT-STRING WINDOW (SLOT-VALUE OBJECT 'NAME)
         :PLACEMENT REGION :H-JUSTIFICATION 'LEFT :V-JUSTIFICATION 'TOP :OPERATION
'IL:ERASE))

(DEFMETHOD MOVE ((OBJECT BOX)
                 (WINDOW LAYOUT-MODEL-EDITOR)
                 REGION)

;;; Déplacement de l'objet sur l'éditeur

   (DECLARE (SPECIAL *MOVE-BITMAP*))
   (WITH-SLOTS (IL-WINDOW)
       WINDOW
     (LET* ((INITIAL-POS (GET-CURSOR-POSITION WINDOW))
            (INITIAL-DELTA (POSITION-SUBTRACT INITIAL-POS (REGION-POSITION REGION)))
            (OLD-POS INITIAL-POS)
            (NEW-POS INITIAL-POS)
            (DELTA-POS)
            (REGIONS)
            (OLD-REGION (COPY-LIST REGION))
            (NEW-REGION (COPY-LIST REGION))
            (SUPER-REGION))
           BITBLT *MOVE-BITMAP* 0 0 IL-WINDOW (- (CAR INITIAL-POS)
                                                 8)
                 (- (CDR INITIAL-POS)
                    2)
                 NIL NIL 'IL:INPUT 'IL:INVERT)                  ; Affiche le symbole move sur
l'objet
           (SETQ REGIONS (LOOP FOR REG-INFO IN (GETWINDOWPROP WINDOW 'REGIONS)
                               COLLECT
                               (THIRD REG-INFO)))               ; Liste des régions de la
fenêtre
           (SETQ SUPER-REGION (LOOP FOR REG IN REGIONS THEREIS (AND SUBREGIONP REG REGION)
                                                                    (NOT (EQUAL REG REGION))
                                                                    REG)))
                                                                ; Région ou se trouve la région
de l'objet
           (SETQ SUPER-REGION (REGION-INTERIOR SUPER-REGION)); On enlève les bords de
SUPER-REGION
           (SETQ REGIONS (LOOP FOR REG IN REGIONS WHEN SUBREGIONP SUPER-REGION REG)
                               UNLESS
                               (OR (EQUAL REGION REG)
                                   (EQUAL SUPER-REGION REG))
                               COLLECT REG))                    ; On enlève les régions qui ne
sont pas incluses dans SUPER-REGION
                                                                ; La région de l'objet
                                                                ; et la SUPER-REGION

           ;; La région de l'objet REGION-WINDOW est  déplacée si elle ne chevauche pas une
region de REGIONS et si elle est incluse dans SUPER-REGION

           (LOOP WHILE KEYDOWNP 'IL:LEFT)
                 DO
                 (SETQ NEW-POS (GET-CURSOR-POSITION WINDOW))
                 (SETQ DELTA-POS (POSITION-SUBTRACT NEW-POS OLD-POS))
                                                                * IL:\; "Déplacement du
curseur")
                 (SETQ NEW-REGION OLD-REGION)
                 (LET ((TEST-REGION (COPY-LIST NEW-REGION))
                       (INCREMENT (IF (PLUSP (CAR DELTA-POS))
                                      1
```

```
                                    -1)))
                    (LOOP FOR I FROM 0 TO (ABS (CAR DELTA-POS))
                          DO
                          (SETQ NEW-REGION (COPY-LIST TEST-REGION))
                          (INCF (REGION-LEFT TEST-REGION)
                                INCREMENT)
                          ALWAYS
                          SUBREGIONP SUPER-REGION TEST-REGION)
                          NEVER
                          (LOOP FOR REG IN REGIONS THEREIS REGIONSINTERSECTP REG
TEST-REGION)))
                    (SETQ TEST-REGION (COPY-LIST NEW-REGION))
                    (SETQ INCREMENT (IF (PLUSP (CDR DELTA-POS))
                                        1
                                        -1))
                    (LOOP FOR I FROM 0 TO (ABS (CDR DELTA-POS))
                          DO
                          (SETQ NEW-REGION (COPY-LIST TEST-REGION))
                          (INCF (REGION-BOTTOM TEST-REGION)
                                INCREMENT)
                          ALWAYS
                          SUBREGIONP SUPER-REGION TEST-REGION)
                          NEVER
                          (LOOP FOR REG IN REGIONS THEREIS REGIONSINTERSECTP REG
TEST-REGION)))
                    )                                          ; Tests de positionnement de
l'objet
                (UNLESS (EQUAL NEW-REGION OLD-REGION)
                    (SETQ NEW-POS (POSITION-ADD OLD-POS (POSITION-SUBTRACT (REGION-POSITION
NEW-REGION)
                                                                           (REGION-POSITION
OLD-REGION))))
                    BITBLT *MOVE-BITMAP* 0 0 IL-WINDOW (- (CAR OLD-POS)
                                                          8)
                          (- (CDR OLD-POS)
                             2)
                          NIL NIL 'IL:INPUT 'IL:INVERT)    ; Efface le symbole move
                    (DRAW-REGION WINDOW OLD-REGION :OPERATION 'IL:ERASE)
                                                           ; Efface l'objet
                    (DRAW-REGION WINDOW NEW-REGION :OPERATION 'IL:PAINT)
                                                           ; Redessine l'objet
                    BITBLT *MOVE-BITMAP* 0 0 IL-WINDOW (- (CAR NEW-POS)
                                                          8)
                          (- (CDR NEW-POS)
                             2)
                          NIL NIL 'IL:INPUT 'IL:INVERT)    ; Réaffiche MOVE
                    (SETQ OLD-REGION (COPY-LIST NEW-REGION))
                    (SETQ OLD-POS NEW-POS)))
            BITBLT *MOVE-BITMAP* 0 0 IL-WINDOW (- (CAR OLD-POS)
                                                  8)
                  (- (CDR OLD-POS)
                     2)
                  NIL NIL 'IL:INPUT 'IL:INVERT)            ; Efface le symbole move sur
l'objet
            (LET* ((SCALE (GETWINDOWPROP WINDOW 'PAGE-SCALE))
                   (DELTA (POSITION-MULTIPLY (POSITION-SUBTRACT (REGION-POSITION NEW-REGION)
                                                                (REGION-POSITION REGION))
                                             SCALE))           ; Déplacement total de l'objet
                   (SUPERORDINATE (LOOP FOR REG IN (GETWINDOWPROP WINDOW 'REGIONS)
                                        THEREIS
                                        (WHEN (EQUAL SUPER-REGION (REGION-INTERIOR (THIRD
REG)))
                                              (SECOND REG)))))
                  (LOOP FOR X IN (SLOT-VALUE SUPERORDINATE 'SUBORDINATES)
                        DO
                        (WHEN (AND (EQUAL OBJECT (SECOND X))
                                   (EQUAL (THIRD X)
                                          (FOURTH (GETWINDOWPROP WINDOW 'SELECTED-REGION))))
                              (SETF (THIRD X)
                                    (POSITION-ADD (THIRD X)
                                                  DELTA)))))
            (REDISPLAY (GETWINDOWPROP WINDOW 'MODEL)
                  WINDOW))))

(DEFMETHOD NAME ((OBJECT ELEMENT-TYPE))
```

```
   (OR (SLOT-VALUE OBJECT 'NAME)
       (NAME (KIND-OF OBJECT))))

(DEFMETHOD OK ((WINDOW LAYOUT-MODEL-EDITOR))

;;; Mémorise l'objet et ferme la fenêtre

   (CLOSEW WINDOW)
   CURSOR IL:WAITINGCURSOR)                                    ; Affiche un sablier
   (COPY (GETWINDOWPROP WINDOW 'MODEL)
         (GETWINDOWPROP WINDOW 'MODEL-MEMO))
   CURSOR T))

(DEFMETHOD OK ((WINDOW BOX-EDITOR))

;;; Recopie l'objet édité dans l'objet mémorisé

   (DECLARE (SPECIAL *LAYOUT-OBJECT-BROWSER*))
   (LET* ((LAYOUT-OBJECT (GETWINDOWPROP WINDOW 'LAYOUT-OBJECT))
          (LAYOUT-OBJECT-MEMO (GETWINDOWPROP WINDOW 'LAYOUT-OBJECT-MEMO))
          (SCALE-FACTOR (GETWINDOWPROP WINDOW 'SCALE-FACTOR)))
         (SETF (SLOT-VALUE LAYOUT-OBJECT 'DISPLAY-PARAMETERS)
               (LIST 'UNIT (GETWINDOWPROP WINDOW 'MEASURE-UNIT)
                     'ROUND-UNIT
                     (GETWINDOWPROP WINDOW 'ROUND-UNIT)
                     'SCALE-FACTOR SCALE-FACTOR 'H-SCALE (* (GETWINDOWPROP WINDOW 'H-SCALE)
                                                            SCALE-FACTOR)
                     'V-SCALE
                     (* (GETWINDOWPROP WINDOW 'V-SCALE)
                        SCALE-FACTOR)))                       ; Modification des paramètres
d'affichage de l'objet
         (COND
            ((EQUAL (SLOT-VALUE LAYOUT-OBJECT-MEMO 'NAME)
                    "<<New Object>>")
             (COPY LAYOUT-OBJECT (MAKE-INSTANCE 'BOX :MEMORIZE T))
             (REPLACE-ITEMS *LAYOUT-OBJECT-BROWSER* (GET-BROWSER-ITEMS 'BOX)))
                                                              ; Si l'objet est nouveau, on
recopie l'objet édité dans une nouvelle instance et on réaffiche le browser
            (T (COPY LAYOUT-OBJECT LAYOUT-OBJECT-MEMO)))      ; Sinon, on recopie simplement
les slots dans 'LAYOUT-OBJECT-MEMO
         (CLOSEW WINDOW)))

(DEFMETHOD PRINT-DESCRIPTION ((OBJECT BASIC-LOGICAL-ELEMENT)
                              (WINDOW GUIDE-EDITOR))

;;; Affiche la description de l'objet dans le stream

   (LET ((OBJECT-NAME (NAME OBJECT)))
        (COND
           ((ASSOC OBJECT (EDITED-OBJECTS WINDOW))
            (INCF (THIRD (ASSOC OBJECT (EDITED-OBJECTS WINDOW)))))
           (T (CHANGE-FONT WINDOW :WEIGHT (IF (REFERENCE (KIND-OF* OBJECT)
                                                         (EDITED-MODEL WINDOW))
                                              'IL:BOLD
                                              'IL:REGULAR))
              (PUSH (LIST OBJECT (PRINT-STRING WINDOW (FORMAT NIL "~A~8T" OBJECT-NAME))
                          1)
                    (EDITED-OBJECTS WINDOW))
              (PRINT-STRING WINDOW (FORMAT NIL "<-- ~A;~%" (GET-CONTENT OBJECT)))))))

(DEFMETHOD PRINT-DESCRIPTION ((OBJECT COMPOSED-LOGICAL-ELEMENT)
                              (WINDOW GUIDE-EDITOR))

;;; Affiche la description de l'objet dans la fenêtre

   (LET ((OBJECT-NAME (NAME OBJECT)))
        (COND
           ((ASSOC OBJECT (EDITED-OBJECTS WINDOW))
            (INCF (THIRD (ASSOC OBJECT (EDITED-OBJECTS WINDOW)))))
           (T (CHANGE-FONT WINDOW :WEIGHT (IF (REFERENCE (KIND-OF* OBJECT)
                                                         (EDITED-MODEL WINDOW))
                                              'IL:BOLD
                                              'IL:REGULAR))
              (PUSH (LIST OBJECT (PRINT-STRING WINDOW (FORMAT NIL "~A~8T" OBJECT-NAME))
                          1)
                    (EDITED-OBJECTS WINDOW))
```

```
                (PRINT-STRING WINDOW (FORMAT NIL "<-- "))
                (LET ((SUBORDINATES (GET-ORDERED-SUBORDINATES OBJECT))
                      (INHERITED-SUB (GET-ORDERED-SUBORDINATES (KIND-OF OBJECT)))
                      (CONSTRUCTOR (GET-CONSTRUCTOR OBJECT))
                      (CTR -1))
                     (WHEN (> (LENGTH SUBORDINATES)
                              1)
                         (CHANGE-FONT WINDOW :WEIGHT (IF (CONSTRUCTOR OBJECT)
                                                         'IL:BOLD
                                                         'IL:REGULAR))
                         (PRINT-STRING WINDOW (FORMAT NIL "~A " (CASE CONSTRUCTOR
                                                                  (SEQUENCE "SEQ")
                                                                  (CHOICE "CHO")
                                                                  (AGGREGATE "AGR")))))
                     (LOOP FOR X IN SUBORDINATES DO (CHANGE-FONT
                                                     WINDOW :WEIGHT
                                                     (IF (AND (REFERENCE (KIND-OF* OBJECT)
                                                                         (EDITED-MODEL WINDOW))
                                                              (NOT (ASSOC (CAR X)
                                                                          INHERITED-SUB)))
                                                         'IL:BOLD
                                                         'IL:REGULAR))
                           (PRINT-STRING WINDOW (FORMAT NIL "~A~A~A " (NAME (CAR X))
                                                        (CASE (SECOND X)
                                                            (? "?")
                                                            (* "*")
                                                            (+ "+")
                                                            (T ""))
                                                        (IF (AND (THIRD X)
                                                                 (NOT (EQUAL (THIRD X)
                                                                             "")))
                                                            (FORMAT NIL "{~A}" (THIRD X))
                                                            ""))))
                     (CHANGE-FONT WINDOW :WEIGHT 'IL:BOLD)
                     (PRINT-STRING WINDOW (FORMAT NIL ";~%"))
                     (LOOP FOR X IN SUBORDINATES DO (PRINT-DESCRIPTION (REFERENCE (CAR X)
                                                                                  (EDITED-MODEL
WINDOW)
                                                                                  T)
                                                                       WINDOW)))))))

(DEFMETHOD PRINT-DESCRIPTION ((OBJECT ELEMENT-TYPE)
                              (WINDOW GUIDE-EDITOR))

;;; Affiche la description de l'objet dans le stream

   (LET ((OBJECT-NAME (NAME OBJECT)))
        (COND
           ((ASSOC OBJECT (EDITED-OBJECTS WINDOW))
            (INCF (THIRD (ASSOC OBJECT (EDITED-OBJECTS WINDOW)))))
           (T (CHANGE-FONT WINDOW :WEIGHT (IF (REFERENCE OBJECT (EDITED-MODEL WINDOW))
                                              'IL:BOLD
                                              'IL:REGULAR))
              (PUSH (LIST OBJECT (PRINT-STRING WINDOW (FORMAT NIL "~A~8T" OBJECT-NAME))
                          1)
                    (EDITED-OBJECTS WINDOW))
              (PRINT-STRING WINDOW (FORMAT NIL "<-- ~A;~%" "(Unbound)"))))))

(DEFMETHOD PRINT-DESCRIPTION ((MODEL LAYOUT-MODEL)
                              STREAM)

;;; Ecrit la description du modèle physique dans STREAM

   (FORMAT STREAM "~%~%~A = {" (SLOT-VALUE MODEL 'NAME))
   (LOOP FOR PAGE IN (GET-SUBORDINATES MODEL)
         DO
         (FORMAT STREAM "~%~2T~A = {" (SLOT-VALUE (CAR PAGE)
                                                  'NAME))
         (LOOP FOR PAGE-DIV IN (GET-SUBORDINATES (CAR PAGE))
               DO
               (FORMAT STREAM "~%~4TAT ~A ~A" (SECOND PAGE-DIV)
                       (SLOT-VALUE (CAR PAGE-DIV)
                                   'NAME))
               (WHEN (GET-SUBORDINATES (CAR PAGE-DIV))
                   (FORMAT STREAM " = {")
                   (LOOP FOR LAYOUT-OBJECT IN (GET-SUBORDINATES (CAR PAGE-DIV))
```

```
                        DO
                        (FORMAT STREAM "~%~6TAT ~A ~A" (SECOND LAYOUT-OBJECT)
                               (SLOT-VALUE (CAR LAYOUT-OBJECT)
                                      'NAME))
                        FINALLY
                        (FORMAT STREAM "}"))))
         (FORMAT STREAM "~%~4T}"))
   (FORMAT STREAM "~%~2T}"))

(DEFMETHOD PRINT-DESCRIPTION :BEFORE
   ((OBJECT COMPOSED-BOX)
    STREAM)

;;; Aprés avoir imprimé la description de l'objet, lance l'impression des sous objets

   (LOOP FOR SUB IN (GET-SUBORDINATES OBJECT)
         DO
         (PRINT-DESCRIPTION (CAR SUB)
                STREAM)))

(DEFMETHOD PRINT-DESCRIPTION ((OBJECT BOX)
                              STREAM)

;;; Ecrit la description de l'objet physique

   (WITH-SLOTS (WIDTH MAX-WIDTH OP-WIDTH HEIGHT MAX-HEIGHT OP-HEIGHT LEFT-MARGIN RIGHT-MARGIN
                      TOP-MARGIN BOTTOM-MARGIN LEADING-SEPARATION TRAILING-SEPARATION
LEADING-OFFSET
                      TRAILING-OFFSET)
       OBJECT
     (FORMAT STREAM "~%~A =" (SLOT-VALUE OBJECT 'NAME))
     (IF (EQUAL OP-WIDTH 'FIXED)
         (FORMAT STREAM "~%~2TWIDTH= FIXED ~A" WIDTH)
         (FORMAT STREAM "~%~2TWIDTH= VARIABLE (~A TO ~A)" WIDTH MAX-WIDTH))
     (IF (EQUAL OP-HEIGHT 'FIXED)
         (FORMAT STREAM "~%~2THEIGHT= FIXED ~A" HEIGHT)
         (FORMAT STREAM "~%~2THEIGHT= VARIABLE (~A TO ~A)" HEIGHT MAX-WIDTH))
     (FORMAT STREAM "~%~2TMARGIN: RIGHT=~A LEFT=~A TOP=~A BOTTOM=~A" LEFT-MARGIN RIGHT-MARGIN
           TOP-MARGIN BOTTOM-MARGIN)
     (FORMAT STREAM "~%~2TSEPARATION: LEADING=~A TRAILING=~A" LEADING-SEPARATION
TRAILING-SEPARATION)
     (FORMAT STREAM "~%~2TOFFSET: LEADING=~A TRAILING=~A" LEADING-OFFSET TRAILING-OFFSET)))

(DEFMETHOD REDISPLAY ((OBJECT LAYOUT-MODEL)
                      (WINDOW LAYOUT-MODEL-EDITOR))

;;; Affiche ou réaffiche le modéle sur la fenêtre

   (LET ((CURRENT-PAGE (NTH (GETWINDOWPROP WINDOW 'CURRENT-PAGE)
                            (GET-SUBORDINATES OBJECT))))
        (DISPLAY- (CAR CURRENT-PAGE)
               WINDOW :POSITION (SECOND CURRENT-PAGE)
               :FREE-REGION
               (MAKE-REGION :LEFT (SLOT-VALUE OBJECT 'LEFT-MARGIN)
                      :BOTTOM
                      (SLOT-VALUE OBJECT 'BOTTOM-MARGIN)
                      :WIDTH
                      (- (SLOT-VALUE OBJECT 'WIDTH)
                         (SLOT-VALUE OBJECT 'RIGHT-MARGIN)
                         (SLOT-VALUE OBJECT 'LEFT-MARGIN))
                      :HEIGHT
                      (- (SLOT-VALUE OBJECT 'HEIGHT)
                         (SLOT-VALUE OBJECT 'TOP-MARGIN)
                         (SLOT-VALUE OBJECT 'BOTTOM-MARGIN))))))

(DEFMETHOD REDISPLAY-WINDOW ((WINDOW BOX-EDITOR))

;;; Réaffiche les données initiales de l'objet lorsque le bouton REDISPLAY est sélectionné

   (LET* ((DESCR-WINDOW (CAR (ATTACHEDWINDOWS WINDOW)))      ; Fenêtre contenant la
description détaillée de l'objet
          (OBJECT (GETWINDOWPROP WINDOW 'LAYOUT-OBJECT))     ; Objet édité
          (OBJECT-MEMO (GETWINDOWPROP WINDOW 'LAYOUT-OBJECT-MEMO))
                                                             ; Objet initial
          )
         (LET ((NAME (SLOT-VALUE OBJECT 'NAME)))
```

```
            (COPY OBJECT-MEMO OBJECT)
            (SETF (SLOT-VALUE OBJECT 'NAME)
                  NAME)                                    ; Recopie tous les slots de
l'objet initial dans l'objet édité, sauf le nom qui ne peut pas être modifié
            )
        (DISPLAY-DESCRIPTION OBJECT DESCR-WINDOW)          ; Affiche la description de
l'objet
        (COMPUTE-REGIONS OBJECT WINDOW)                    ; Détermine les différentes
régions occupées par l'objet et initialise les propriétés correspondantes de la window
        (DRAW-REGIONS WINDOW)                              ; Trace le contour des régions
occupées par l'objet
        ))

(DEFMETHOD REFERENCE ((OBJECT ELEMENT-TYPE)
                      (MODEL GUIDE)
                      &OPTIONAL RECURSEFLG)

;;; Retourne la référence de OBJECT, dans le contexte de MODEL

;;; Si la référence n'est pas trouvée et que RECURSEFLG est différent de NIL, on va chercher
la référence de l'objet dans le supérieur du modèle

   (LET ((REF (SECOND (ASSOC OBJECT (CONTEXT MODEL)))))
        (COND
          (REF)
          ((NOT RECURSEFLG)
           NIL)
          ((NOT (KIND-OF MODEL))
           OBJECT)
          (T (REFERENCE OBJECT (KIND-OF MODEL)
                        T)))))

(DEFMETHOD REMOVE-SUBORDINATE ((OBJECT COMPOSED-LOGICAL-ELEMENT)
                               COMPONENT &OPTIONAL OTHER)

;;; Enlève dans la composition de OBJECT, le composant COMPONENT

   (SETF (SUBORDINATES OBJECT)
         (APPEND (SUBORDINATES OBJECT)
                 '((REMOVE ,COMPONENT)))))

(DEFMETHOD REMOVE-SUBORDINATE ((OBJECT ELEMENT-TYPE)
                               (MODEL GUIDE)
                               &OPTIONAL COMPONENT)

;;; Enlève dans la composition de OBJECT, utilisé dans le contexte de MODELE, le composant
COMPONENT

   (REMOVE-SUBORDINATE (CLONE-IF-NECESSARY OBJECT MODEL)
          COMPONENT))

(DEFMETHOD REORDER-SUBORDINATES ((OBJECT COMPOSED-LOGICAL-ELEMENT)
                                 MODEL)

;;; Réordonne les subordonnés de OBJECT

   (SETF (REORDER-LIST OBJECT)
         NIL)
   (LET* ((CHOICE)
          (REORDER-LIST)
          (LIST-SUB (GET-ORDERED-SUBORDINATES OBJECT))
          (INHERITED-SUB (GET-ORDERED-SUBORDINATES (KIND-OF OBJECT)))
          (LOCALLY-DEFINED-SUB)
          (CTR 1))
         (SETQ INHERITED-SUB (LOOP FOR X IN (INTERSECTION INHERITED-SUB LIST-SUB)
                                   COLLECT
                                   (LIST (CAR X)
                                         (1+ (POSITION (CAR X)
                                                       LIST-SUB :KEY #'CAR)))))
         (SETQ LOCALLY-DEFINED-SUB (LOOP FOR X IN (SET-DIFFERENCE LIST-SUB INHERITED-SUB :KEY
                                                                  #'CAR)
                                         COLLECT
                                         (LIST (CAR X)
                                               (1+ (POSITION (CAR X)
                                                             LIST-SUB :KEY #'CAR)))))
         (LOOP FOR ELT FROM 1 TO (+ (LENGTH LOCALLY-DEFINED-SUB)
```

```
                                    (LENGTH INHERITED-SUB))
                DO
                (SETQ CHOICE
                      (EXECUTE (MAKE-INSTANCE
                                'MENU :TITLE (FORMAT NIL " Element No ~d " ELT)
                                :ITEMS
                                (APPEND (WHEN INHERITED-SUB
                                              (LIST (LIST (NAME (CAAR INHERITED-SUB))
                                                          '',(CAR INHERITED-SUB))))
                                        (LOOP FOR X IN LOCALLY-DEFINED-SUB COLLECT
                                              (LIST (NAME (CAR X))
                                                    '',X)))))))
                (SETQ REORDER-LIST (APPEND REORDER-LIST (CDR CHOICE)))
                (SETQ INHERITED-SUB (REMOVE (CAR CHOICE)
                                            INHERITED-SUB :KEY #'CAR))
                (SETQ LOCALLY-DEFINED-SUB (REMOVE (CAR CHOICE)
                                                  LOCALLY-DEFINED-SUB :KEY #'CAR)))
          (SETF (REORDER-LIST OBJECT)
                REORDER-LIST)))

(DEFMETHOD REORDER-SUBORDINATES ((OBJECT ELEMENT-TYPE)
                                 (MODEL GUIDE))

;;; Réordonne les éléments de OBJECT utilisé dans le contexte de modèle

   (REORDER-SUBORDINATES (CLONE-IF-NECESSARY OBJECT MODEL)
          MODEL))

(DEFMETHOD REPAINT ((WINDOW GUIDE-EDITOR))
   (CLEARW WINDOW)
   (SETF (EDITED-OBJECTS WINDOW)
         NIL)
   (PRINT-DESCRIPTION (REFERENCE (ROOT (EDITED-MODEL WINDOW))
                                 (EDITED-MODEL WINDOW)
                                 T)
          WINDOW))

(DEFMETHOD REPAINT ((WINDOW BOX-EDITOR))

;;; Redessine le contenu de l'éditeur

   (DRAW-H-SCALE WINDOW)
   (DRAW-V-SCALE WINDOW)
   (DRAW-REGIONS WINDOW))

(DEFMETHOD REPAINT* ((WINDOW GUIDE-EDITOR))

;;; Cette méthode réaffiche la description du modèle édité ainsi que de tous les autres
modèles édités à l'écran dont la description est dépendante du modèle courant

   (DECLARE (SPECIAL *EDITED-GUIDES*))
   (LOOP FOR X IN *EDITED-GUIDES* DO (WHEN (KIND-OF? (CAR X)
                                                     (EDITED-MODEL WINDOW)
                                                     T)
                                           (REPAINT (SECOND X))))
   (REPAINT WINDOW))

(DEFMETHOD RESHAPE ((WINDOW BOX-EDITOR))

;;; Redessine le contenu de l'éditeur

   (DRAW-H-SCALE WINDOW)
   (DRAW-V-SCALE WINDOW)
   (DRAW-REGIONS WINDOW))

(DEFMETHOD SPECIALIZE ((MODEL GUIDE)
                       &OPTIONAL NAME)

;;; Spécialise le modèle en un nouveau modèle de nom NAME

   (LET ((NEW-MODEL (MAKE-INSTANCE 'GUIDE :NAME NAME :MEMORIZE T)))
        (SETF (KIND-OF NEW-MODEL)
              MODEL)
        (PUSH NEW-MODEL (FO-DNIK MODEL))
        (SETF (ROOT NEW-MODEL)
              (ROOT MODEL))
```

```
        NEW-MODEL))

(DEFMETHOD SPECIALIZE ((OBJECT ELEMENT-TYPE)
                       &OPTIONAL NAME)

;;; Spécialise l'objet

   (LET* ((NEW-OBJECT (MAKE-INSTANCE (CLASS-NAME (CLASS-OF OBJECT))
                          :NAME NAME)))
         (SETF (KIND-OF NEW-OBJECT)
               OBJECT)
         (PUSH NEW-OBJECT (FO-DNIK OBJECT))
         NEW-OBJECT))

(DEFMETHOD UPDATE-OBJECT ((OBJECT BOX)
                          (FREE-MENU BOX-EDITOR-DESCRIPTION))

;;; Modifie l'objet en fonction des données se trouvant dans le freeMenu et le redessine

   (END-EDIT FREE-MENU T)                                    ; Force la fin de l'édition
   (LET* ((MAIN-WINDOW (MAINWINDOW FREE-MENU))
          (SCALE-FACTOR (GETWINDOWPROP MAIN-WINDOW 'SCALE-FACTOR))
                                                             ; Diviseur  à appliquer aux
données du FreeMenu pour obtenir la dimension de l'objet en BMUs
          (FM-STATE (GET-STATE FREE-MENU))
          (WIDTH (GETF FM-STATE :EDITMINWIDTH))
          (OP-WIDTH (COND
                      ((GETF FM-STATE :WIDTHFIXED)
                       'FIXED)
                      ((GETF FM-STATE :WIDTHMINSIZE)
                       'MINSIZE)
                      ((GETF FM-STATE :WIDTHMAXSIZE)
                       'MAXSIZE)))
          (MAX-WIDTH (IF (EQUAL OP-WIDTH 'FIXED)
                         WIDTH
                         (GETF FM-STATE :EDITMAXWIDTH)))
          (HEIGHT (GETF FM-STATE :EDITMINHEIGHT))
          (OP-HEIGHT (COND
                       ((GETF FM-STATE :HEIGHTFIXED)
                        'FIXED)
                       ((GETF FM-STATE :HEIGHTMINSIZE)
                        'MINSIZE)
                       ((GETF FM-STATE :HEIGHTMAXSIZE)
                        'MAXSIZE)))
          (MAX-HEIGHT (IF (EQUAL OP-HEIGHT 'FIXED)
                          HEIGHT
                          (GETF FM-STATE :EDITMAXHEIGHT)))
          (LEFT-MARGIN (GETF FM-STATE :EDITLEFTMARGIN))
          (RIGHT-MARGIN (GETF FM-STATE :EDITRIGHTMARGIN))
          (BOTTOM-MARGIN (GETF FM-STATE :EDITBOTTOMMARGIN))
          (TOP-MARGIN (GETF FM-STATE :EDITTOPMARGIN))
          (LEADING-SEPARATION (GETF FM-STATE :EDIT-LEADING-SEPARATION))
          (TRAILING-SEPARATION (GETF FM-STATE :EDIT-TRAILING-SEPARATION))
          (LEADING-OFFSET (GETF FM-STATE :EDIT-LEADING-OFFSET))
          (TRAILING-OFFSET (GETF FM-STATE :EDIT-TRAILING-OFFSET))
          (ERRORS NIL))

         ;; Tests de validité des informations saisies

         (COND
            ((> WIDTH MAX-WIDTH)
             (PUSH "WIDTH is greater than MAXIMUM WIDTH" ERRORS)))
         (COND
            ((> HEIGHT MAX-HEIGHT)
             (PUSH "HEIGHT is greater than MAXIMUM HEIGHT" ERRORS)))
         (COND
            ((> (+ LEFT-MARGIN RIGHT-MARGIN)
                WIDTH)
             (PUSH "RIGHT and LEFT MARGINS are greater than the object WIDTH" ERRORS)))
         (COND
            ((> (+ TOP-MARGIN BOTTOM-MARGIN)
                HEIGHT)
             (PUSH "TOP and BOTTOM MARGINS are greater than the object HEIGHT" ERRORS)))
         (COND
            (ERRORS (PRINT-ERRORS ERRORS)
```

```
                    NIL)                                          ; Si tout est OK, on met a jour
l'objet en effectuant la conversion
              (T (SETF (SLOT-VALUE OBJECT 'WIDTH)
                       (ROUND WIDTH SCALE-FACTOR))
                 (SETF (SLOT-VALUE OBJECT 'MAX-WIDTH)
                       (ROUND MAX-WIDTH SCALE-FACTOR))
                 (SETF (SLOT-VALUE OBJECT 'OP-WIDTH)
                       OP-WIDTH)
                 (SETF (SLOT-VALUE OBJECT 'HEIGHT)
                       (ROUND HEIGHT SCALE-FACTOR))
                 (SETF (SLOT-VALUE OBJECT 'MAX-HEIGHT)
                       (ROUND MAX-HEIGHT SCALE-FACTOR))
                 (SETF (SLOT-VALUE OBJECT 'OP-HEIGHT)
                       OP-HEIGHT)
                 (SETF (SLOT-VALUE OBJECT 'LEFT-MARGIN)
                       (ROUND LEFT-MARGIN SCALE-FACTOR))
                 (SETF (SLOT-VALUE OBJECT 'RIGHT-MARGIN)
                       (ROUND RIGHT-MARGIN SCALE-FACTOR))
                 (SETF (SLOT-VALUE OBJECT 'BOTTOM-MARGIN)
                       (ROUND BOTTOM-MARGIN SCALE-FACTOR))
                 (SETF (SLOT-VALUE OBJECT 'TOP-MARGIN)
                       (ROUND TOP-MARGIN SCALE-FACTOR))
                 (SETF (SLOT-VALUE OBJECT 'LEADING-SEPARATION)
                       (ROUND LEADING-SEPARATION SCALE-FACTOR))
                 (SETF (SLOT-VALUE OBJECT 'TRAILING-SEPARATION)
                       (ROUND TRAILING-SEPARATION SCALE-FACTOR))
                 (SETF (SLOT-VALUE OBJECT 'LEADING-OFFSET)
                       (ROUND LEADING-OFFSET SCALE-FACTOR))
                 (SETF (SLOT-VALUE OBJECT 'TRAILING-OFFSET)
                       (ROUND TRAILING-OFFSET SCALE-FACTOR))
                 T)
              (COMPUTE-REGIONS OBJECT MAIN-WINDOW)
              (DRAW-REGIONS MAIN-WINDOW))))

(DEFMETHOD UPDATE-OBJECT ((OBJECT BOX)
                          (WINDOW BOX-EDITOR))

;;; Détermine les dimensions (en BMUs) de l'objet physique contenu dans la window en fonction
des régions graphiques figurant sur la fenêtre, modifie en conséquence les slots de l'objet
physique et réaffiche le FREE-MENU contenant la description

   (DECLARE (SPECIAL ROUND-UNIT))
   (WITH-SLOTS (WIDTH MAX-WIDTH OP-WIDTH HEIGHT MAX-HEIGHT OP-HEIGHT LEFT-MARGIN RIGHT-MARGIN
                      BOTTOM-MARGIN TOP-MARGIN)
       OBJECT
     (LET ((V-SCALE (GETWINDOWPROP WINDOW 'V-SCALE))
           (H-SCALE (GETWINDOWPROP WINDOW 'H-SCALE))
           (MAIN-REGION (GETWINDOWPROP WINDOW 'MAIN-REGION))
           (MAX-REGION (GETWINDOWPROP WINDOW 'MAX-REGION))
           (MARGIN-REGION (GETWINDOWPROP WINDOW 'MARGIN-REGION)))
          (SETQ WIDTH (ROUND (* (REGION-WIDTH MAIN-REGION)
                                H-SCALE)))
          (SETQ MAX-WIDTH (ROUND (* (REGION-WIDTH MAX-REGION)
                                    H-SCALE)))
          (COND
             ((= MAX-WIDTH WIDTH)
              (SETQ OP-WIDTH 'FIXED))
             ((AND (EQUAL OP-WIDTH 'FIXED)
                   (> MAX-WIDTH WIDTH))
              (SETQ OP-WIDTH 'MINSIZE)))
          (SETQ HEIGHT (ROUND (* (REGION-HEIGHT MAIN-REGION)
                                 V-SCALE)))
          (SETQ MAX-HEIGHT (ROUND (* (REGION-HEIGHT MAX-REGION)
                                     V-SCALE)))
          (COND
             ((= MAX-HEIGHT HEIGHT)
              (SETQ OP-HEIGHT 'FIXED))
             ((AND (EQUAL OP-HEIGHT 'FIXED)
                   (> MAX-HEIGHT HEIGHT))
              (SETQ OP-HEIGHT 'MINSIZE)))
          (SETQ LEFT-MARGIN (ROUND (* (- (REGION-LEFT MARGIN-REGION)
                                         20)
                                      H-SCALE)))
          (SETQ RIGHT-MARGIN (ROUND (* (- (REGION-RIGHT MAIN-REGION)
                                          (REGION-RIGHT MARGIN-REGION))
                                       H-SCALE)))
```

```
(SETQ BOTTOM-MARGIN (ROUND (* (- (REGION-BOTTOM MARGIN-REGION)
                                 20)
                              V-SCALE)))
(SETQ TOP-MARGIN (ROUND (* (- (REGION-TOP MAIN-REGION)
```

Annexe B

Les développements en langage Haskell

Annexe B.1
Les fonctions de parsing utilisées

```
module Parse(Parser(..), (+.+), (..+), (+..), (||||), (>>), (>>>),
(|||!),

            into, lit, litp, many, succeed, fail, sepBy, count,
sepBy1,
            parse, simpleLex) where
infixl 8 +.+ , ..+ , +..
infix  6 >> , >>>, 'into'
infixr 4 ||| , |||!

type Parser a b = [a] -> [(b, [a])]

-- Alternative
(||||) :: Parser a b -> Parser a b -> Parser a b
p |||| q = \as-> p as ++ q as
-- Alternative, but with committed choice
(|||!) :: Parser a b -> Parser a b -> Parser a b
p |||! q = \as-> case p as of { [] -> q as; xs -> xs }

-- Sequence
(+.+) :: Parser a b -> Parser a c -> Parser a (b,c)
p +.+ q = \as-> [((b,c), as'') | (b,as') <- p as, (c,as'') <- q as' ]
-- Sequence, throw away first part
(..+) :: Parser a b -> Parser a c -> Parser a c
p ..+ q = \as-> [(c, as'') | (b,as') <- p as, (c,as'') <- q as' ]
-- Sequence, throw away second part
(+..) :: Parser a b -> Parser a c -> Parser a b
p +.. q = \as-> [(b, as'') | (b,as') <- p as, (c,as'') <- q as' ]

-- Action
(>>) :: Parser a b -> (b->c) -> Parser a c
p >> f = \as-> [(f b, as') | (b, as') <- p as ]

-- Action on two items
(>>>) :: Parser a (b,c) -> (b->c->d) -> Parser a d
p >>> f = \as-> [(f b c, as') | ((b,c), as') <- p as ]

-- Use value
into :: Parser a b -> (b -> Parser a c) -> Parser a c
p 'into' fq = \as->[ca | (b, as')<-p as, ca<-fq b as']

-- Succeeds with a value
succeed :: b -> Parser a b
succeed v = \as->[(v, as)]

-- Always fails.
fail :: Parser a b
fail = \as->[]

-- Kleene star
many :: Parser a b -> Parser a [b]
many p = p +.+ many p >>> (:)
     ||| succeed []

-- Parse an exact number of items
count :: Parser a b -> Int -> Parser a [b]
count p 0 = succeed []
count p k = p +.+ count p (k-1) >>> (:)

-- Non-empty sequence of items separated by something
sepBy1 :: Parser a b -> Parser a c -> Parser a [b]
p 'sepBy1' q = p +.. q +.+ (p 'sepBy' q) >>> (:)
          ||| p >> (: [])
```

```
-- Sequence of items separated by something
sepBy :: Parser a b -> Parser a c -> Parser a [b]
p 'sepBy' q = p 'sepBy1' q
         ||| succeed []

-- Recognize a literal token
lit :: (Eq a) => a -> Parser a a
lit x = \as->
       case as of
       a:as' | a==x -> [(a, as')]
       _ -> []

-- Recognize a token with a preicate
litp :: (a->Bool) -> Parser a a
litp p = \as->
       case as of
       a:as' | p a -> [(a, as')]
       _ -> []

-- Parse, and check if it was ok.
parse :: Parser a b -> [a] -> b
parse p as =
       case [b | (b,[]) <- p as] of
       [] -> error "No parse"
       [b] -> b
       _ -> error "Ambiguous parse"


----------------------------------------
-- quelques fonctions utiles ajoutees --
-- a celles figurant dans Parse.hs    --
----------------------------------------
module Parse1 where
import Parse

---------- reconnait un ou plusieurs elements satisfaisant p
some :: Parser a b -> Parser a [b]
some p = p +.+ many p >>> (:)

---------- reconnait une suite de caracteres
token :: (Eq a) => [a] -> Parser a [a]
token [] = succeed []
token (x:xs) = (lit x) +.+ (token xs) >>> (:)

---------- reconnait un token en ignorant les maj/min
token1 :: [Char] -> Parser Char String
token1 [] = succeed []
token1 (x:xs) = ((lit (toUpper x)) ||| (lit (toLower x))) +.+ (token1 xs) >>> (:)

---------- on cree une fonction booleenne qui retourne vrai pour tous
---------- les caracteres imprimables a l'exceptions des @#!*? '<' et '>'
isAutorise :: Char -> Bool
isAutorise c = (c> ' ' && c<= ';') || c=='=' || (c>= '?' && c<='~')

---------- quelques raccourcis
----------
-- une suite de caracteres differents de space
chars :: Parser Char String
chars = some (litp isAutorise)

-- n'importe quels caracteres
anyChars :: Parser Char String
anyChars = many (litp isAutorise ||| lit ' ')

-- une suite de caracteres
someChars :: Parser Char String
someChars = some (litp isAutorise ||| lit ' ')

-- un identifiant
name :: Parser Char String
name = some (litp isAlphanum ||| lit '-' ||| lit '#' ||| lit '.')

-- au moins un espace
sep :: Parser Char String
sep = some (litp isSpace)
```

```
-- des espaces eventuels
spaces :: Parser Char String
spaces = many (litp isSpace)

-- des espaces ou des sauts de ligne
separ :: Parser Char String
separ = many (litp isSpace ||| lit '\n')

-- retourne la valeur associee a un identifiant
-- dans une liste de couples (ident, valeur)
find :: (Eq a) => a -> [(a, b)] -> b
--find name [] = []
find name ((id,val):xs) = if (name == id) then val else find name xs

---------- fonctions specifiques au parsing d'un fichier SGML
-- reconnait un tag de debut
pInitTag :: Parser Char String
pInitTag = lit '<' ..+ name +.. lit '>' +.. many (lit '\n')

-- reconnait un tag de fin
pEndTag :: Parser Char String
pEndTag = token "</" ..+ name +.. lit '>' +.. many (lit '\n')

-- reconnait un tag de nom donne
pTag :: String -> Parser Char String
pTag name = lit '<' ..+ token name +.. lit '>' +.. many (lit '\n')

-- separe un fichier sgml en un couple compose de tete et corps
-- en fonction du separateur "]>\n"
hsepar :: String -> (String,String)
hsepar str = hsepar1 ("",str)

hsepar1 :: (String,String) -> (String,String)
hsepar1 (h,']':'>':'\n':fic) = (h ++ "]>\n",fic)
hsepar1 (h,a:b:c:rst) = hsepar1 (h ++ [a],b:c:rst)
```

Annexe B.2
Sources du module de transformation DTD --> EDTD

```
-------------------------------------------------------
-- construction de l'arbre de parsing correspondant --
-- a l'analyse d'une DTD SGML                        --
-------------------------------------------------------
module ParseSGML (pSgml) where
import Parse

---------- reconnait un ou plusieurs elements satisfaisant p
some :: Parser a b -> Parser a [b]
some p = p +.+ many p >>> (:)

---------- reconnait une suite de caracteres
token :: (Eq a) => [a] -> Parser a [a]
token [] = succeed []
token (x:xs) = (lit x) +.+ (token xs) >>> (:)

---------- reconnait un token en ignorant les maj/min
token1 :: [Char] -> Parser Char [Char]
token1 [] = succeed []
token1 (x:xs) = ((lit (toUpper x)) ||| (lit (toLower x))) +.+ (token1 xs) >>> (:)

---------- on cree une fonction booleenne qui retourne vrai pour tous
---------- les caracteres imprimables a l'exceptions des @#!*? '<' et '>'
isAutorise :: Char -> Bool
isAutorise c = (c> ' ' && c<= ';') || c=='=' || (c>= '?' && c<='~')

---------- quelques raccourcis
--
-- une suite de caracteres differents de space
chars = some (litp isAutorise)

-- n'importe quels caracteres
anyChars = many (litp isAutorise ||| lit ' ')

-- un ou plusieurs caracteres alphanumeriques
name = some (litp isAlphanum ||| lit '-' ||| lit '#')

-- au moins un espace
sep = some (litp isSpace)

-- des espaces eventuels
spaces = many (litp isSpace)

-- des espaces ou des sauts de ligne
separ = many (litp isSpace ||| lit '\n')

---------- definition des types
data Commande = Element String Bool Bool ModelGroup |
                EmptyCde |
                OtherCdes String
                deriving Text
type Comment = String

---------- reconnaissance d'un Tag
pTag :: Parser Char Bool
pTag = (lit '-' >> \x->False) |!| (lit 'o' >> \x->True)

-------------------------------------------------------
---------- parser d'analyse d'un Model Group ----------
-------------------------------------------------------
data ModelGroup = Rep ModelGroup |
```

```
                OptRep ModelGroup |
                Opt ModelGroup |
                Name String |
                Parenth ModelGroup |
                Seq [ModelGroup] |
                And [ModelGroup] |
                Choice [ModelGroup]
                deriving Text

-- le nom d'un element
pName :: Parser Char ModelGroup
pName = name >> build where build s = Name s

-- la repetition
pRep :: Parser Char ModelGroup
pRep = (pName ||| pParenth) +.. lit '+' >> build where build s = Rep s

-- le repetition optionnelle
pOptRep :: Parser Char ModelGroup
pOptRep = (pName ||| pParenth) +.. lit '*' >> build where build s = OptRep s

-- l'option
pOpt :: Parser Char ModelGroup
pOpt = (pName ||| pParenth) +.. lit '?' >> build where build s = Opt s

-- la sequence
pSeq :: Parser Char ModelGroup
pSeq = pModelGroup +.+ some (spaces +.. lit ',' +.. spaces ..+ pModelGroup)
                >> build where build (m1,m2) = Seq (m1:m2)

-- l'aggregat
pAnd :: Parser Char ModelGroup
pAnd = pModelGroup +.+ some (spaces +.. lit '&' +.. spaces ..+ pModelGroup)
                >> build where build (m1,m2) = And (m1:m2)

-- le choix
pChoice :: Parser Char ModelGroup
pChoice = pModelGroup +.+ some (spaces +.. lit '|' +.. spaces ..+ pModelGroup)
                >> build where build (m1,m2) = Choice (m1:m2)

-- une structure parenthesee
pParenth :: Parser Char ModelGroup
pParenth = lit '(' ..+ spaces ..+ (pSeq ||| pAnd ||| pChoice ||| pModelGroup)
                +.. spaces +.. lit ')' >> build where build s = Parenth s

-- reconnaissance d'un Model Group
pModelGroup :: Parser Char ModelGroup
pModelGroup = pName ||| pRep ||| pOptRep ||| pOpt ||| pParenth

-----------------------------------------------------------------
---------- parser d'analyse d'une ligne SGML ----------
-----------------------------------------------------------------

-- reconnaissance de la commande ELEMENT
pElement :: Parser Char Commande
pElement = token1 "ELEMENT" +.. sep ..+ name +.. sep +.+ pTag +.. sep +.+ pTag +.. sep +.+
pModelGroup +.. spaces >> build
       where build (((s,t1),t2),expr) = Element s t1 t2 expr

-- reconnaissance de la commande vide
pEmptyCde :: Parser Char Commande
pEmptyCde = succeed (EmptyCde)

-- reconnaissance des autres commandes
pOtherCdes :: Parser Char Commande
pOtherCdes = (token1 "ATTLIST" ||| token1 "ENTITY" ||| token1 "DOCTYPE" ||| token1 "USEMAP" |||
token1 "SHORTREF" ||| token1 "[" ||| token1 "NOTATION") +.+ anyChars >> build where build (a,b) =
OtherCdes (a++b)

-- reconnaissance des commandes
pCommande :: Parser Char Commande
pCommande = pElement ||| pAttlist ||| pOtherCdes ||| pEmptyCde

-- reconnaissance des commentaires
pComment :: Parser Char Comment
pComment = token "--" ..+ anyChars +.. token "--"
```

```
-- reconnaissance d'une ligne
pPhrase :: Parser Char (Commande, String)
pPhrase = token "<!" ..+ pCommande +.+ (pComment ||| succeed []) +.. spaces +.. token ">" +..
separ

-----------------------------------------------------------------
---------- parser d'analyse d'un texte SGML ----------
-----------------------------------------------------------------
pSgml :: Parser Char [(Commande, String)]
pSgml = (some pPhrase)



-------------------------------------------------------------
-- transformation de l'arbre de parsing de la DTD analysee --
-- en EDTD et conversion en ASCII                          --
-------------------------------------------------------------
module ParseSGML2 (dtd2edtd) where
import Parse
import ParseSGML

--
-- transformation de la DTD en EDTD
--
convert :: ([(Commande, String)],String) -> [(Commande, String)]
convert ([],y) = []
convert (x:xs,y) = removeDuplicate (transcris x ++ convert (xs,y))

-----------------------------------
-- transformation d'une commande --
-----------------------------------

transcris :: (Commande, String) -> [(Commande, String)]

transcris(Element s b1 b2 mg@(Parenth (Choice [a])),cmt)=[(Element s b1 b2 (Parenth (Choice ((map
newmg [a]) ++ [(Name (concat["Choice-",s]))]))),""), (Element (concat["Choice-",s]) False False
(Parenth (Seq (addCond [a])))),concat ["choix conditionne de ",s])] ++ map creeElt (aCreer mg)

transcris(Element s b1 b2 mg,cmt)=[(Element s b1 b2 (newmg mg),cmt),(creeElt (concat
["case-",s]))] ++ map creeElt (aCreer mg)

transcris(EmptyCde,cmt) = [(EmptyCde,cmt)]

transcris(OtherCdes s,cmt)=[(OtherCdes s,cmt)]

-------------------------------------
-- transformation d'un Model Group --
-------------------------------------

-- premier element d'un couple
first (a,b) = a

-- second element d'un couple
second (a,b) = b

-- retourne le nouveau ModelGroup
--
newmg :: ModelGroup -> ModelGroup
newmg mg = first(transforme (mg,[]))

-- retourne la liste des nouveaux elements a creer apres changement
-- du ModelGroup
--
aCreer :: ModelGroup -> [String]
aCreer mg = second(transforme (mg,[]))

-- retourne une liste de tous les elements a creer a partir d'une
-- liste de ModelGroup
--
collectaCreer :: [ModelGroup] ->[String]
collectaCreer (mg:[]) = aCreer mg
collectaCreer (mg:autresMg) = aCreer mg ++ collectaCreer autresMg

--
```

```
-- transformation du ModelGroup
--
transforme :: (ModelGroup,[String]) -> (ModelGroup, [String])
transforme ((Name s),[]) = ((Name s),[])
transforme ((Rep (Name a)), []) = (Parenth (Choice [Rep (Name a), Name (concat [a,"-rep"])]),
[(concat ["rep-", a])])
transforme ((Rep a),[]) = ((Rep (newmg a)),aCreer a)
transforme ((OptRep (Name a)), []) = (Parenth (Choice [OptRep (Name a), Name (concat
["rep-",a])]), [(concat ["orep-",a])])
transforme ((OptRep a),[]) = ((OptRep (newmg a)),aCreer a)
transforme ((Opt a),[]) = ((Opt (newmg a)),aCreer a)
transforme ((Parenth a),[]) = ((Parenth (newmg a)), aCreer a)
transforme ((Seq a),[]) = ((Seq (map newmg a)),collectaCreer a)
transforme ((And a),[]) = ((And (map newmg a)),collectaCreer a)
transforme ((Choice a),[]) = ((Choice (map newmg a)),collectaCreer a)

--
-- Enleve les lignes en double dans la nouvelle EDTD
--
removeDuplicate :: [(Commande, String)] -> [(Commande, String)]
removeDuplicate [] = []
removeDuplicate (x@(Element elt _ _ _,_):xs) = x:removeDuplicate (filter (\(Element s _ _ _,_) ->
(s /= elt)) xs)
removeDuplicate (x:xs) = x:removeDuplicate xs

-- ajoute une condition pour tous les elements
--
addCond (s:[]) = [(Opt (Name "cond")),s]
addCond (s:xs) = [(Opt (Name "cond")),s] ++ addCond xs

-- creation des nouveaux elements representant
-- les structures conditionnelles
--
creeElt :: String -> (Commande, String)
creeElt ('c':'a':'s':'e':'-':xs) = (Element (concat [xs,"-case"]) False False (Parenth (Rep
(Parenth (Seq [Opt (Name "cond"), Name xs])))),concat ["pour conditionner l'element ",xs])
creeElt ('o':'r':'e':'p':'-':xs) = (Element (concat [xs,"-orep"]) False False (Parenth (Seq
[(Name "cond"), Name xs])), concat ["repetition optionnelle de ",xs])
creeElt ('r':'e':'p':'-':xs) = (Element (concat [xs,"-rep"]) False False (Parenth (Seq [(Name
"cond"), Name xs])), concat ["repetition de ",xs])


-- transformation DTD -> EDTD
--
dtd2edtd :: String -> String
dtd2edtd x = edtd2ascii(convert(head(pSgml x)))

-----------------------------------
-- conversion de l'EDTD en ASCII --
-----------------------------------

-- conversion en ascii des minimisations de Tags
--
mini2ascii :: Bool -> String
mini2ascii True = "o"
mini2ascii False = "-"

-- conversion en ascii d'un Model Group
--
mg2ascii :: ModelGroup -> String
mg2ascii (Name s) = s
mg2ascii (Rep mg) = (mg2ascii mg) ++ "+"
mg2ascii (OptRep mg) = (mg2ascii mg) ++ "*"
mg2ascii (Opt mg) = (mg2ascii mg) ++ "?"
mg2ascii (Parenth mg) = "(" ++ (mg2ascii mg) ++ ")"
mg2ascii (Seq (x:[])) = (mg2ascii x)
mg2ascii (Seq (x:xs)) = (mg2ascii x) ++ ", " ++ (mg2ascii (Seq xs))
mg2ascii (And (x:[])) = (mg2ascii x)
mg2ascii (And (x:xs)) = (mg2ascii x) ++ "&" ++ (mg2ascii (And xs))
mg2ascii (Choice (x:[])) = (mg2ascii x)
mg2ascii (Choice (x:xs)) = (mg2ascii x) ++ "|" ++ (mg2ascii (Choice xs))


-- conversion en ascii d'une commande
--
commande2ascii :: (Commande, String) -> String
```

```
commande2ascii (OtherCdes s1,"") = "<!" ++ s1 ++ ">\n"
commande2ascii (OtherCdes s1,cmt) = "<!" ++ s1 ++ "  -- " ++ cmt ++ " -->\n"
commande2ascii (Element s b1 b2 mg,"") = "<!ELEMENT " ++ s ++ " " ++ (mini2ascii b1) ++ " " ++
(mini2ascii b2) ++ " " ++ (mg2ascii mg) ++ ">\n"
commande2ascii (Element s b1 b2 mg,cmt) = "<!ELEMENT " ++ s ++ " " ++ (mini2ascii b1) ++ " " ++
(mini2ascii b2) ++ " " ++ (mg2ascii mg) ++ "  -- " ++ cmt ++ " -->\n"

-- conversion en ascii de l'EDTD
--
edtd2ascii :: [(Commande, String)] -> String
edtd2ascii (x:[]) = commande2ascii x
edtd2ascii (x:xs) = commande2ascii x ++ edtd2ascii xs


-------------------------------------------------
---------- module de dialogue principal ----------
-------------------------------------------------
module Main(main) where
import Parse
import ParseSGML
import ParseSGML2

--
-- le module main demande un nom de fichier a l'utilisateur
-- il lit ce fichier (contenant une DTD) sur le disque
-- il l'analyse, le transforme en EDTD et l'ecrit sur le disque
-- dans un fichier suffixe par .edtd
--
main :: Dialogue
main = appendChan stdout "Enter the name of the DTD:\n" exit (
       readChan stdin exit (\ userInput ->
       let (name:_) = lines userInput in (
       readFile (concat [name,".dtd"])      (\ ioerror -> appendChan stdout "can't open file\n"
exit done)
                        (\ contents -> writeFile (concat [name,".edtd"]) (dtd2edtd contents) exit
done))))
```

Annexe B.3
Sources du module d'assemblage

```
-----------------------------------------------
---------- module de dialogue principal ----------
-----------------------------------------------
module Main(main) where
import Parse
import Parse1
import Analyse
import ReadVars
import ReadCond

--
-- le module main demande le nom du fichier ou sont stockees les donnees
-- il l'analyse et memorise le contenu dans la variable 'ExternalVars'
-- il demande et analyse ensuite le fichier ou sont stockees les conditions
-- qu'il memorise dans la variable 'Conditions'
-- il demande ensuite le nom du fichier SGML ou se trouve le gabarit
-- (fichier suffixe par '.gab')
-- Ce module produit en sortie un fichier '.sgml' qui contient le document
-- SGML adapte aux donnees fournies
--
main :: Dialogue
main = readChan stdin exit (\ userInput ->
       appendChan stdout "Enter the name of the Data file: " exit (
       let names = lines userInput
           name1 = head names
           name2 = head (tail names)
           name3 = head (tail (tail names))
       in (
       readFile name1 (\ ioerror ->
              appendChan stdout "can't open file\n" exit done)
                     (\ variables ->
       appendChan stdout "Enter the name of the Conditions File: " exit (
       readFile name2 (\ ioerror ->
              appendChan stdout "can't open file\n" exit done)
                     (\ conditions ->
              appendChan stdout "Enter the name of the Gabarit: " exit (
       readFile (name3 ++ ".gab")
                     (\ ioerror -> appendChan stdout "can't open file\n" exit done)
                     (\ gabarit  -> writeFile (name3 ++ ".sgml") (assembleGab gabarit (readCond
conditions)(readVars variables)) exit done)))))))))


-----------------------------------
---------- module Analyse ----------
-----------------------------------
module Analyse where
import Parse
import Parse1
import ReadCond

data Element = ConditionElt String Element |
               VariableElt String |
               EmptyElt String |
               ComposedElt String [Element] |
               BasicElt String [Element] |
               TextElt String
               deriving Text

pVariableElt = pTag "VAR" ..+ name +.. pTag "/VAR" >> VariableElt
pConditionElt = pTag "COND" ..+ name +.. pTag "/COND" +.+ pElement >>> ConditionElt
pBasicElt = pInitTag +.+ (some (pVariableElt ||| pTextElt)) +.. pEndTag >>> BasicElt
pComposedElt = pInitTag +.+ (some (pComposedElt ||| pConditionElt  ||| pBasicElt ||| pEmptyElt))
+.. pEndTag >>> ComposedElt
pTextElt = someChars >> TextElt
```

```
pEmptyElt = pInitTag +.. pEndTag >> EmptyElt

---------- reconnait un element
pElement :: Parser Char Element
pElement = pVariableElt ||| pConditionElt ||| pComposedElt
                        ||| pBasicElt     ||| pEmptyElt

---------- retourne l'analyse d'un gabarit
gabarit :: String -> (String, Element)
gabarit x = let {a = hsepar x} in (fst a,fst(head(pElement (snd a))))

---------- evalue un element
evalElt :: Elt -> [(String, String)] -> String
evalElt (VarElt s) var = evalVar s var
evalElt (AlphElt elt) _ = elt
evalElt (NumElt elt) _ = elt

---------- evalue un predicat
evalPred :: Predicat -> [(String, String)] -> Bool
evalPred (PredUn (NumElt elt)) _      = (read elt)>0
evalPred (PredUn (AlphElt elt)) _     = (token1 "true" elt)/=[]
evalPred (PredUn (VarElt elt)) var    = (token1 "true" (evalVar elt var))/=[]
evalPred (PredBi e1 Less e2) v              = (evalElt e1 v) <  (evalElt e2 v)
evalPred (PredBi e1 LessOrEq e2) v    = (evalElt e1 v) <= (evalElt e2 v)
evalPred (PredBi e1 Eq e2) v          = (evalElt e1 v) == (evalElt e2 v)
evalPred (PredBi e1 Greater e2) v     = (evalElt e1 v) >  (evalElt e2 v)
evalPred (PredBi e1 GreaterOrEq e2) v = (evalElt e1 v) >= (evalElt e2 v)
evalPred (PredBi e1 Diff e2) v              = (evalElt e1 v) /= (evalElt e2 v)

---------- evalue une liste de predicat
---------- evalCond retourne TRUE si tous les predicats sont vrais
evalCond :: [Predicat] -> [(String, String)] -> Bool
evalCond cond var = all (\x -> evalPred x var) cond

---------- evalue une condition
eval :: String -> [(String,[Predicat])] -> [(String, String)] -> Bool
eval "True" _ _ = True
eval "False" _ _ = False
eval name cond var = evalCond (find name cond) var

---------- recherche la valeur de la variable dont on donne le nom
---------- les variable sont memorisees dans une liste de couples
---------- (nom-variable, valeur)
evalVar :: String -> [(String, String)] -> String
evalVar name var = find name var

---------- Choisit l'element de la liste associee a une condition vraie
chooseCase :: [Element] -> [(String, [Predicat])] -> [(String, String)] -> Element
chooseCase [] _ _ = EmptyElt "nothing"
chooseCase (ConditionElt name elt:xs) cond var = if (eval name cond var) then elt else
(chooseCase xs cond var)
chooseCase (x:_) _ _ = x -- cas ou l'element n'est pas conditionne

---------- Cree une liste composee de 'n' elements elt
repeatElt :: (Num n) => n -> Element -> [Element]
repeatElt 0 _ = []
repeatElt n elt = elt:(repeatElt (n-1) elt)

---------- retourne le contenu d'un element (sans les tags)
content :: Element -> [Element]
content (ComposedElt _ elts) = elts
content (BasicElt _ elts) = elts
content (EmptyElt _) = []

---------- assemble un element
assemble :: Element -> [(String, [Predicat])] -> [(String, String)] -> String

assemble (TextElt content) _ _ = content
assemble (EmptyElt name) _ _ = "\n<" ++ name ++ ">" ++ "</" ++ name ++ ">"
assemble (VariableElt name) _ var = evalVar name var
assemble (BasicElt name elts) cond var = "\n<" ++ name ++ ">"
       ++ concat (map (\elt -> assemble elt cond var) elts)
       ++ "</" ++ name ++ ">"
assemble (ComposedElt ('C':'A':'S':'E':'-':_) elts) cond var =
       concat (map (\elt -> assemble elt cond var) (content (chooseCase elts cond var)))
--assemble (ComposedElt ('R':'E':'P':'-':_) (cond:elt:[])) =
```

```
--      concat (map assemble (content (repeatElt (read(evalvar cond))::Int elt)))
--assemble (ComposedElt ('O':'R':'E':'P':'-':_) cond elt) =
--      concat (map assemble (content (repeatElt (read(evalvar cond))::Int elt)))
assemble (ComposedElt name elts) cond var = "\n<" ++ name ++ ">"
        ++ concat  (map (\elt -> assemble elt cond var) elts) ++ "</" ++ name ++ ">"
assemble (ConditionElt name elt) cond var = if (eval name cond var) then (assemble elt cond var)
else ""

assembleGab :: String -> [(String, [Predicat])] -> [(String, String)] -> String
assembleGab gab cond var = fst(gabarit gab) ++ (assemble (snd(gabarit gab)) cond var)

-------------------------------------
---------- module ReadVars ----------
-------------------------------------
module ReadVars where
import Parse
import Parse1

-- reconnais une ligne variable+valeur
pLigne :: Parser Char (String, String)
pLigne = spaces ..+ name +.. sep +.+ chars

-- parse le fichier de variables
pVars :: Parser Char [(String, String)]
pVars = some pLigne +.. separ

-- retourne une liste de couples correspondant au fichier de data
readVars :: [Char] -> [(String, String)]
readVars fic = fst(head (pVars fic))

-------------------------------------
---------- module ReadCond ----------
-------------------------------------
module ReadCond where
import Parse
import Parse1

data Elt =      VarElt String |
                AlphElt String |
                NumElt String
                deriving Text
data Op =       Less | LessOrEq | Eq | Greater | GreaterOrEq | Diff
                deriving Text

data Predicat = PredUn Elt |
                PredBi Elt Op Elt
                deriving Text


pAlphElt :: Parser Char Elt
pAlphElt = lit '"' ..+ anyChars +.. lit '"' >> AlphElt

pNumElt :: Parser Char Elt
pNumElt = some (litp isDigit) >> NumElt

pVarElt :: Parser Char Elt
pVarElt = name >> VarElt

pEl :: Parser Char Elt
pEl = pTag "ELEMENT" ..+ (pNumElt ||| pVarElt ||| pAlphElt) +.. pTag "/ELEMENT"

pOp :: Parser Char Op
pOp = (token ">" >> \x -> Greater)  ||| (token ">=" >> \x -> GreaterOrEq) |||
      (token "&lt;" >> \x -> Less)  ||| (token "&lt;=" >> \x -> LessOrEq) |||
      (token "!=" >> \x -> Diff)    ||| (token "=" >> \x -> Eq)

pComp :: Parser Char Op
pComp = pTag "COMP" ..+ pOp +.. pTag "/COMP"

pPredUn :: Parser Char Predicat
pPredUn = pEl >> PredUn

pPredBi :: Parser Char Predicat
pPredBi = pEl +.+ pComp +.+ pEl >> build
       where build ((el1,op),el2) = PredBi el1 op el2
```

```
pExp :: Parser Char Predicat
pExp = pTag "EXP" ..+ (pPredUn ||| pPredBi) +.. pTag "/EXP"

pPredicat :: Parser Char [Predicat]
pPredicat = pTag "PREDICAT" ..+ (some pExp) +.. pTag "/PREDICAT"

pNomCond :: Parser Char String
pNomCond = pTag "NOMCOND" ..+ name +.. pTag "/NOMCOND"

pCondition :: Parser Char (String, [Predicat])
pCondition = pTag "COND" ..+ pNomCond +.+ pPredicat +.. pTag "/COND"

pConditions :: Parser Char [(String, [Predicat])]
pConditions = pTag "CONDITION" ..+ (some pCondition) +.. pTag "/CONDITION"

readCond :: String -> [(String, [Predicat])]
readCond fic = fst(head(pConditions (snd(hsepar fic))))
```

Annexe C

Exemple d'un fichier 'P' généré par BIBLE

La description en langage 'P' suivante a été automatiquement produite en résultat d'une commande *Export* effectuée sur l'éditeur de modèle physiques présenté dans la figure 4.19. Le guide utilisé pour la génération est celui de la figure 4.12.

```
PRESENTATION LETTRE;
VIEWS
   firstView;

DEFAULT
   BEGIN
   HorizRef: Enclosed . HRef;
   VertRef: * . Left;
   Width: Enclosed . Width;
   Height: Enclosed . Height;
   VertPos: Top = Previous . Bottom;
   HorizPos: Left = Enclosing . Left;
   Justify: Enclosing =;
   LineSpacing: Enclosing =;
   Break: Yes;
   Visibility: Enclosing =;
   Font: Enclosing =;
   Style: Enclosing =;
   Underline : Enclosing =;
   thickness : Enclosing =;
   Size: Enclosing =;
   Adjust: Enclosing =;
   Indent: Enclosing =;
   Depth : 3;
   END;

BOXES

{----- Definition des pages -----}

   PAGE:
    BEGIN
    Width: 19.56 cm;
    Height: 27.94 cm;
    HorizPos: Left = Enclosing . Left;
    VertPos: Top = Enclosing . Top;
    END;

RULES

   LETTRE:
    BEGIN
    Page(PAGE);
    Size: 10;
    Font: Times;
    Style: Roman;
    Indent: 0;
```

```
 Justify: No;
 LineSpacing: 1;
 Adjust: Left;
 Underline: NoUnderline;
 Thickness: Thin;
 END;

ENTETE:
 BEGIN
 Width: 19.56 cm;
 Height: 9.31 cm;
 HorizPos: Left = Enclosing . Left + 0.0 cm;
 VertPos: Top = Enclosing . Top + 0.00 cm;
 Size: 10;
 Font: Times;
 Style: Roman;
 Indent: 0;
 Justify: No;
 LineSpacing: 1;
 Adjust: Left;
 Underline: NoUnderline;
 Thickness: Thin;
 END;

DEST:
 BEGIN
 Width: 6.99 cm;
 Height: 4.23 cm;
 HorizPos: Left = Enclosing . Left + 10.58 cm;
 VertPos: Top = Enclosing . Top + 2.54 cm;
 Size: 10;
 Font: Times;
 Style: Roman;
 Indent: 0;
 Justify: No;
 LineSpacing: 1;
 Adjust: Left;
 Underline: NoUnderline;
 Thickness: Thin;
 END;

VOSREF:
 BEGIN
 Width: 4.23 cm;
 Height: 1.06 cm;
 HorizPos: Left = Enclosing . Left + 1.06 cm;
 VertPos: Top = Enclosing . Top + 7.20 cm;
 Size: 10;
 Font: Times;
 Style: Roman;
 Indent: 0;
 Justify: No;
 LineSpacing: 1;
 Adjust: Left;
 Underline: NoUnderline;
 Thickness: Thin;
 END;

DEST1:
 BEGIN
 Width: Enclosing . Width;
```

```
  Height: Enclosing . Height;
  HorizPos: Left = Enclosing . Left + 0.0 cm;
  VertPos: Top = Previous . Bottom + 0.0 cm;
  Size: 10;
  Font: Times;
  Style: Roman;
  Indent: 0;
  Justify: No;
  LineSpacing: 1;
  Adjust: Left;
  Underline: NoUnderline;
  Thickness: Thin;
  END;

NOM:
  BEGIN
  Width: Enclosed . Width;
  Height: 1.06 cm;
  HorizPos: Left = Enclosing . Left + 0.50 cm;
  VertPos: Top = Previous . Bottom + 0.0 cm;
  Size: 10;
  Font: Times;
  Style: Roman;
  Indent: 0;
  Justify: No;
  LineSpacing: 1;
  Adjust: Left;
  Underline: NoUnderline;
  Thickness: Thin;
  END;

ADR:
  BEGIN
  Width: Enclosed . Width;
  Height: Enclosed . Height;
  HorizPos: Left = Enclosing . Left + 0.50 cm;
  VertPos: Top = Previous . Bottom + 0.0 cm;
  Size: 10;
  Font: Times;
  Style: Roman;
  Indent: 0;
  Justify: No;
  LineSpacing: 1;
  Adjust: Left;
  Underline: NoUnderline;
  Thickness: Thin;
  END;

DEST2:
  BEGIN
  Width: Enclosing . Width;
  Height: Enclosing . Height;
  HorizPos: Left = Enclosing . Left + 0.0 cm;
  VertPos: Top = Previous . Bottom + 0.0 cm;
  Size: 10;
  Font: Times;
  Style: Roman;
  Indent: 0;
  Justify: No;
  LineSpacing: 1;
  Adjust: Left;
```

```
    Underline: NoUnderline;
    Thickness: Thin;
    END;

  PARAG:
    BEGIN
    Width: Enclosed . Width;
    Height: Enclosed . Height;
    HorizPos: Left = Enclosing . Left + 1.00 cm;
    VertPos: Top = Previous . Bottom + 0.0 cm;
    Size: 10;
    Font: Times;
    Style: Roman;
    Indent: 0;
    Justify: No;
    LineSpacing: 1;
    Adjust: Left;
    Underline: NoUnderline;
    Thickness: Thin;
    END;

END
```

Annexe D

La table de personnalisation utilisée dans BIBLE

Annexe D.1
La DTD Condition

```
<!-------------------------------------------->
<!-- DTD pour la representation des condition -->
<!-------------------------------------------->
<!DOCTYPE CONDITION [
<!ELEMENT Condition - - (Cond+)>
<!ELEMENT Cond      - - (NomCond, Predicat)>
<!ELEMENT NomCond   - - (#PCDATA)>
<!ELEMENT Predicat  - - (Exp+)>
<!ELEMENT Exp       - - (Element, (Comp, Element)?)>
<!ELEMENT Element   - - (#PCDATA)>
<!ELEMENT Comp      - - (#PCDATA)>
]>
```

Annexe D.2
Le fichier de présentation

```
{-----------------------------------------------}
{-- Presentation associee a la DTD CONDITION --}
{-----------------------------------------------}
PRESENTATION CONDITION;
VIEWS
        firstView, SGML;
#define INDENT 1
#define TAG_FONTSIZE 10 pt
#define PARAG_FONTSIZE 12 pt

DEFAULT
        BEGIN
        HorizRef: Enclosed . HRef ;
        VertRef: * . Left;
        Width: Enclosed . Width;
        Height: Enclosed . Height;
        VertPos: Top = Previous . Bottom;
        HorizPos: Left = Enclosing . Left;
        Justify: Enclosing =;
        LineSpacing: Enclosing =;
        Break: Yes;
        Visibility: Enclosing =;
        Font: Enclosing =;
        Style: Enclosing =;
        Underline : Enclosing =;
        thickness : Enclosing =;
        Size: Enclosing =;
        Adjust: Enclosing =;
        Indent: Enclosing =;
        Depth : 3;
        IN SGML BEGIN
                Visibility: 1;
                HorizPos: Left = Enclosing . Left + INDENT;
                VertPos: Top = Previous . Bottom;
                Width: Enclosing . Width - INDENT;
                Size: PARAG_FONTSIZE;
                Font: helvetica;
                Style: Roman;
                Underline: NoUnderline;
                LineSpacing: 1;
                Justify: No;
                Adjust: Left;
                Indent: 0;
                END;
        END;

BOXES
 CONDITION_S:
        BEGIN
        Content: TEXT '<CONDITION>';
        visibility: 0;
        IN SGML BEGIN
                HorizPos: Left = Enclosing . Left;
                Width: Enclosed . Width;
                Size: TAG_FONTSIZE;
                Style: Bold;
                visibility: 1;
                END;
        END;
 CONDITION_E:
        BEGIN
        Content: TEXT '</CONDITION>';
        visibility: 0;
        IN SGML BEGIN
```

```
                HorizPos: Left = Enclosing . Left;
                Width: Enclosed . Width;
                Size: TAG_FONTSIZE;
                Style: Bold;
                visibility: 1;
                END;
        END;
COND_S:
        BEGIN
        Content: TEXT '<COND>';
        visibility: 0;
        IN SGML BEGIN
                HorizPos: Left = Enclosing . Left;
                Width: Enclosed . Width;
                Size: TAG_FONTSIZE;
                Style: Bold;
                visibility: 1;
                END;
        END;
COND_E:
        BEGIN
        Content: TEXT '</COND>';
        visibility: 0;
        IN SGML BEGIN
                HorizPos: Left = Enclosing . Left;
                Width: Enclosed . Width;
                Size: TAG_FONTSIZE;
                Style: Bold;
                visibility: 1;
                END;
        END;
NOMCOND_S:
        BEGIN
        Content: TEXT '<NOMCOND>';
        visibility: 0;
        IN SGML BEGIN
                HorizPos: Left = Enclosing . Left;
                Width: Enclosed . Width;
                Size: TAG_FONTSIZE;
                Style: Bold;
                visibility: 1;
                END;
        END;
NOMCOND_E:
        BEGIN
        Content: TEXT '</NOMCOND>';
        visibility: 0;
        IN SGML BEGIN
                HorizPos: Left = Enclosing . Left;
                Width: Enclosed . Width;
                Size: TAG_FONTSIZE;
                Style: Bold;
                visibility: 1;
                END;
        END;
PREDICAT_S:
        BEGIN
        Content: TEXT '<PREDICAT>';
        visibility: 0;
        IN SGML BEGIN
                HorizPos: Left = Enclosing . Left;
                Width: Enclosed . Width;
                Size: TAG_FONTSIZE;
                Style: Bold;
                visibility: 1;
                END;
        END;
PREDICAT_E:
        BEGIN
        Content: TEXT '</PREDICAT>';
        visibility: 0;
        IN SGML BEGIN
                HorizPos: Left = Enclosing . Left;
                Width: Enclosed . Width;
                Size: TAG_FONTSIZE;
                Style: Bold;
```

```
                visibility: 1;
                END;
        END;
EXP_S:
        BEGIN
        Content: TEXT '<EXP>';
        visibility: 0;
        IN SGML BEGIN
                HorizPos: Left = Enclosing . Left;
                Width: Enclosed . Width;
                Size: TAG_FONTSIZE;
                Style: Bold;
                visibility: 1;
                END;
        END;
EXP_E:
        BEGIN
        Content: TEXT '</EXP>';
        visibility: 0;
        IN SGML BEGIN
                HorizPos: Left = Enclosing . Left;
                Width: Enclosed . Width;
                Size: TAG_FONTSIZE;
                Style: Bold;
                visibility: 1;
                END;
        END;
ELEMENT_S:
        BEGIN
        Content: TEXT '<ELEMENT>';
        visibility: 0;
        IN SGML BEGIN
                HorizPos: Left = Enclosing . Left;
                Width: Enclosed . Width;
                Size: TAG_FONTSIZE;
                Style: Bold;
                visibility: 1;
                END;
        END;
ELEMENT_E:
        BEGIN
        Content: TEXT '</ELEMENT>';
        visibility: 0;
        IN SGML BEGIN
                HorizPos: Left = Enclosing . Left;
                Width: Enclosed . Width;
                Size: TAG_FONTSIZE;
                Style: Bold;
                visibility: 1;
                END;
        END;
COMP_S:
        BEGIN
        Content: TEXT '<COMP>';
        visibility: 0;
        IN SGML BEGIN
                HorizPos: Left = Enclosing . Left;
                Width: Enclosed . Width;
                Size: TAG_FONTSIZE;
                Style: Bold;
                visibility: 1;
                END;
        END;
COMP_E:
        BEGIN
        Content: TEXT '</COMP>';
        visibility: 0;
        IN SGML BEGIN
                HorizPos: Left = Enclosing . Left;
                Width: Enclosed . Width;
                Size: TAG_FONTSIZE;
                Style: Bold;
                visibility: 1;
                END;
        END;
CloseTag:
```

```
        BEGIN
        Content: TEXT '>';
        visibility: 0;
        IN SGML BEGIN
                HorizPos: Left = Previous . Right;
                VertPos: Top = Previous . Top;
                Width: Enclosed . Width;
                Size: TAG_FONTSIZE;
                Style: Bold;
                visibility: 1;
                END;
        END;
 DiamStart:
        BEGIN
        Content: TEXT G '\250\240';
        visibility: 0;
        IN SGML BEGIN
                HorizPos: Left = Enclosing . Left;
                Width: Enclosed . Width;
                Size: TAG_FONTSIZE;
                Style: Bold;
                visibility: 1;
                END;
        END;
 DiamEnd:
        BEGIN
        Content: TEXT G '\240\250';
        visibility: 0;
        IN SGML BEGIN
                HorizPos: Left = Previous . Right;
                VertPos: Top = Previous . Top;
                Width: Enclosed . Width;
                Size: TAG_FONTSIZE;
                Style: Bold;
                visibility: 1;
                END;
        END;

 AndBox:
        BEGIN
        Content: TEXT ' & ';
        visibility: 1;
        IN SGML BEGIN
                visibility: 0;
                END;
        END;


RULES
 CONDITION:
        BEGIN
        Width: Enclosing . Width;
        Create(CONDITION_S);
        CreateLast(CONDITION_E);
        END;
 COND:
        BEGIN
        Width: Enclosing . Width;
        Create(COND_S);
        CreateLast(COND_E);
        END;
 NOMCOND:
        BEGIN
        Line; Adjust: Left; Justify: No;
        Width: 3 cm;
        Create(NOMCOND_S);
        CreateLast(NOMCOND_E);
        IN SGML BEGIN
                Line; Adjust: Left; Justify: No;
                END;
        END;
 PREDICAT:
        BEGIN
        Width: Enclosing . Width - 3.5 cm;
        VertPos: Top = Previous . Top;
        HorizPos: Left = Enclosing . Left + 3.5 cm;
```

```
        Create(PREDICAT_S);
        CreateLast(PREDICAT_E);
        END;
 EXP:
        BEGIN
        Width: Enclosed . Width;
        VertPos: Top = Previous . Top;
        HorizPos: Left = Previous . Right;
        if Not First CreateBefore(AndBox);
        Create(EXP_S);
        CreateLast(EXP_E);
        END;
 ELEMENT:
        BEGIN
        Line; Adjust: Left; Justify: No;
        Width: Enclosed . Width;
        VertPos: Top = Enclosing . Top;
        HorizPos: Left = Previous . Right + 0.1 cm;
        Create(ELEMENT_S);
        CreateLast(ELEMENT_E);
        IN SGML BEGIN
                Line; Adjust: Left; Justify: No;
                END;
        END;
 COMP:
        BEGIN
        Line; Adjust: Left; Justify: No;
        Width: Enclosed . Width;
        VertPos: Top = Enclosing . Top;
        HorizPos: Left = Previous . Right + 0.1 cm;
        Create(COMP_S);
        CreateLast(COMP_E);
        IN SGML BEGIN
                Line; Adjust: Left; Justify: No;
                END;
        END;

 AGR_EXP:
        BEGIN
        Width: Enclosed . Width;
        VertPos: Top = Previous . Top;
        HorizPos: Left = Previous . Right;
        IN SGML BEGIN
                Width: Enclosing . Width;
                HorizPos : Left = Enclosing . Left;
                END;
        END;

 EmptyContents:
         BEGIN
         Visibility: 0;
         IN SGML BEGIN
                 Visibility: 0;
                 END;
         END;


 TEXT_UNIT:
         BEGIN
         IN SGML BEGIN
                 Width: Enclosed . Width;
                 END;
         END;

END
```

Gestion et conception de documents structurés par le contexte

Résumé : A l'instar des programmes informatiques, les documents possèdent une structure logique définie par des règles syntaxiques et sémantiques. Ce constat a permis d'utiliser les acquis du génie logiciel pour définir les base de la modélisation et de la manipulation des documents structurés. Documents et programmes diffèrent cependant par de nombreux aspects. Dans ce mémoire, nous nous sommes interessés à deux particularités des documents :

Alors que les programmes sont principalement destinés à être exécutés, les documents sont destinés à de multiples usages : stockage, lecture, échange, etc. A chaque usage correspond un type de document particulier défini par un modèle. Passer d'un modèle à un autre nécessite souvent d'effectuer des transformations structurelles difficiles et le nombre important de modèles de documents empêche l'écriture de convertisseurs spécifiques. Nous nous sommes interessés aux langages de prototypes qui permettent de décrire des objets par copie différentielle. Nous avons adapté ce principe à la représentation des modèles de documents. Ainsi, les modèles ne sont plus des entités isolées mais possèdent tous au moins un lien avec un autre modèle. Par chaînages successifs, on définit la suite des transformations permettant de passer d'un modèle à un autre et on en déduit les modifications à effectuer sur les documents.

La seconde particularité vient de l'opération de lecture. Comme l'exécution d'un programme, la lecture d'un document produit soit une action, soit un changement d'état chez le récepteur. La singularité de la lecture vient de ce que le résultat dépend non seulement du contenu du document, mais aussi du profil du lecteur. Une adaptation des documents aux lecteurs, ou plus généralement au contexte, est donc souhaitable. Nous réalisons la conception contextuelle de documents en décomposant la phase de production en deux étapes : une phase manuelle de création d'un gabarit qui regroupe les principes communs à l'élaboration d'un groupe de document, et une phase automatique de production de documents en fonction d'un profil client. Un gabarit est représenté par une structure abstraite de données dans laquelle certaines parties peuvent être choisies parmi un ensemble de structures candidates. Nous utilisons, pour être assuré de la validité des documents générés, un modèle de gabarit obtenu par une extension sémantique du modèle définissant les documents.

Mots clés : Document structuré, SGML, Génération de Documents, Héritage des Modèles, prototypes.

The design and management of context-controlled documents

Abstract: Documents, like computer programs have a logical structure defined by syntaxical and semantical rules. This fact allowed the use of software engeneering knowledge to define the foundations for representation and handling of structured documents. However, documents and programs are different in many aspects. In this thesis, we consider two characteristics of documents:

Programs are intended to be runned while documents are intended for multiple purposes : storage, reading, exchange, etc. Each use corresponds to a particular kind of document defined by a document model. Translating a document to another one often requires difficult structure transformations, and the number of document models prevent the development of specific converters. We aimed our study on prototype languages which allow the description of objects by differential copy. We have adapted this principle for the representation of document models. In this way, document models are not longer isolated entities. Each of them have at least one link with another model. By successive linking, we collect the set of transformations to move from a model to another one and we infer the modifications to be done on documents.

The second feature comes from the reading process. Like the execution of a program, the reading of a document causes either an action or a change in the receiver. The singularity of the reading process is that the result depends not only on the content of the document, but also on the reader's profile. Adapting documents to readers, or more generally to a certain context would be desirable. We realize the design of context controlled documents by breaking the generating process in two stages : the design of a gauge which brought together principles common to the design of a set of documents, and an automatic phase of document generation depending on a client profile. A gauge is represented by an abstract data structure in which some parts can be selected among a set of candidate structures. In order to ensure the correctness of generated documents, we use a gauge model obtained by a semantic extension of the document model.

Keywords : Structured document, SGML, Documents generation, Models inheritance, prototypes.

Gestion et conception de documents structurés par le contexte

Résumé : A l'instar des programmes informatiques, les documents possèdent une structure logique définie par des règles syntaxiques et sémantiques. Ce constat a permis d'utiliser les acquis du génie logiciel pour définir les base de la modélisation et de la manipulation des documents structurés. Documents et programmes diffèrent cependant par de nombreux aspects. Dans ce mémoire, nous nous sommes interessés à deux particularités des documents :

Alors que les programmes sont principalement destinés à être exécutés, les documents sont destinés à de multiples usages : stockage, lecture, échange, etc. A chaque usage correspond un type de document particulier défini par un modèle. Passer d'un modèle à un autre nécessite souvent d'effectuer des transformations structurelles difficiles et le nombre important de modèles de documents empêche l'écriture de convertisseurs spécifiques. Nous nous sommes interessés aux langages de prototypes qui permettent de décrire des objets par copie différentielle. Nous avons adapté ce principe à la représentation des modèles de documents. Ainsi, les modèles ne sont plus des entités isolées mais possèdent tous au moins un lien avec un autre modèle. Par chaînages successifs, on définit la suite des transformations permettant de passer d'un modèle à un autre et on en déduit les modifications à effectuer sur les documents.

La seconde particularité vient de l'opération de lecture. Comme l'exécution d'un programme, la lecture d'un document produit soit une action, soit un changement d'état chez le récepteur. La singularité de la lecture vient de ce que le résultat dépend non seulement du contenu du document, mais aussi du profil du lecteur. Une adaptation des documents aux lecteurs, ou plus généralement au contexte, est donc souhaitable. Nous réalisons la conception contextuelle de documents en décomposant la phase de production en deux étapes : une phase manuelle de création d'un gabarit qui regroupe les principes communs à l'élaboration d'un groupe de document, et une phase automatique de production de documents en fonction d'un profil client. Un gabarit est représenté par une structure abstraite de données dans laquelle certaines parties peuvent être choisies parmi un ensemble de structures candidates. Nous utilisons, pour être assuré de la validité des documents générés, un modèle de gabarit obtenu par une extension sémantique du modèle définissant les documents.

Mots clés : Document structuré, SGML, Génération de Documents, Héritage des Modèles, prototypes.

The design and management of context-controlled documents

Abstract: Documents, like computer programs have a logical structure defined by syntaxical and semantical rules. This fact allowed the use of software engeneering knowledge to define the foundations for representation and handling of structured documents. However, documents and programs are different in many aspects. In this thesis, we consider two characteristics of documents:

Programs are intended to be runned while documents are intended for multiple purposes : storage, reading, exchange, etc. Each use corresponds to a particular kind of document defined by a document model. Translating a document to another one often requires difficult structure transformations, and the number of document models prevent the development of specific converters. We aimed our study on prototype languages which allow the description of objects by differential copy. We have adapted this principle for the representation of document models. In this way, document models are not longer isolated entities. Each of them have at least one link with another model. By successive linking, we collect the set of transformations to move from a model to another one and we infer the modifications to be done on documents.

The second feature comes from the reading process. Like the execution of a program, the reading of a document causes either an action or a change in the receiver. The singularity of the reading process is that the result depends not only on the content of the document, but also on the reader's profile. Adapting documents to readers, or more generally to a certain context would be desirable. We realize the design of context controlled documents by breaking the generating process in two stages : the design of a gauge which brought together principles common to the design of a set of documents, and an automatic phase of document generation depending on a client profile. A gauge is represented by an abstract data structure in which some parts can be selected among a set of candidate structures. In order to ensure the correctness of generated documents, we use a gauge model obtained by a semantic extension of the document model.

Keywords : Structured document, SGML, Documents generation, Models inheritance, prototypes.