Transformation de types dans les systèmes d'édition de documents structurés

Extase Akpotsui

THESE

présentée par

Extase K. A. Akpotsui

pour obtenir le titre de

Docteur de l'Institut National Polytechnique de Grenoble

(arrêté ministériel du 30 mars 1992)

Spécialité : Informatique

Transformation de types dans les systèmes d'édition de documents structurés

Soutenue le 26 octobre 1993 devant la commission d'examen composée de :

Claude Delobel	Président et rapporteur
Paul Franchi−Zanettacci	Rapporteur
Sacha Krakowiak	Examinateur
Jacques Mossière	Examinateur
Vincent Quint	Directeur de thèse

Thèse préparée au sein de l'Unité Mixte Bull−IMAG/Systèmes (projet OPERA)

Je remercie

Chaleureusement Monsieur Vincent Quint, mon directeur de thèse, pour l'intérêt qu'il a porté à ce travail. La confiance qu'il m'a témoignée, ses conseils et ses encouragements ont été décisifs dans la réalisation de ce projet. Je lui exprime toute ma gratitude,

Monsieur Claude Delobel d'avoir accepté d'être rapporteur de cette thèse et de m'avoir fait des remarques de fond sur mon travail. Je le remercie également pour l'honneur qu'il m'a fait en acceptant la présidence du jury de cette thèse,

Monsieur Paul Franchi−Zanettacci d'avoir accepté d'être rapporteur de cette thèse,

Monsieur Sacha Krakowiak d'avoir accepté d'être membre du jury,

Monsieur Jacques Mossière d'avoir accepté d'être membre du jury,

Madame Cécile Roisin pour toutes les questions qu'elle m'a posées et pour les discussions qu'elle a eues avec moi. Ses critiques m'ont apporté une aide appréciable,

Madame Irène Vatton qui m'a fait partager sa grande connaissance de la documentation structurée et du système Grif.

J'ai une pensée amicale pour tous les membres de l'unité mixte Bull−IMAG/systèmes.

J'exprime ma reconnaissance à Monsieur Daniel Galand, Madame Rose−Marie Galand et Madame Jacqueline Galand, pour leurs encouragements et leur soutien inestimable.

à ma mère
à mon père

Introduction

Pendant longtemps, les documents étaient considérés comme un flux de caractères et de symboles sans réelles possibilités pour les éditeurs de distinguer leurs constituants sémantiques. Il n'y avait pas de représentation logique ni du document lui−même ni de ses composants. Ces constats ont conduit au concept de description logique des documents, en s'appuyant sur les grammaires hors contexte. Ces grammaires assez riches permettent la description de classes de documents, de leurs composants, des relations hiérarchiques et de voisinage que ces derniers entretiennent les uns avec les autres. Elles permettent aussi de décrire des informations d'ordre sémantique associées aux composants sous forme d'attributs.

L'introduction du concept de structuration dans l'édition électronique apporte des avantages certains dans la manipulation des documents, à l'éditeur lui−même et aux applications qui peuvent tirer partie de la structure. Cependant son utilisation encore limitée est due en grande partie aux problèmes nouveaux qu'elle induit et à la perte de fonctionnalités considérées comme acquises dans les éditeurs non structurés, notamment le couper−coller.

Ainsi, les systèmes d'édition de documents fondés sur ce concept, produisent des documents dont les éléments sont conformes à leur description logique dans la grammaire. Il en résulte une vérification rigoureuse, voire trop rigide, de la compatibilité de types, par exemple dans les situations suivantes :

- L'évolution de la description logique d'une classe de documents portant soit sur la réorganisation hiérarchique et de voisinage de ses composants, soit sur la redéfinition de ces derniers et de leurs attributs rend les instances de documents conformes à l'ancienne version de la classe de documents non éditables du fait du contrôle des types. Ce problème est connu sous le nom de transformation statique.
- Les transformations dynamiques elles, consistent à changer le type d'un élément de document en cours d'édition :
    - ♦ Une opération de couper−coller met en présence deux éléments de documents, souvent de types différents. Le contrôle de types exercé par l'éditeur résulte souvent en un rejet de la requête lorsque les types en présence ne sont pas compatibles.
    - ♦ Lorsque le type d'un élément change sans que ce dernier soit coupé, la structure des éléments qui l'entourent peut être aussi modifiée.

Le but de la présente étude est d'apporter une solution aux problèmes évoqués ci−dessus.

Chapitre I

Le chapitre 1 présente l'état de l'art de l'édition structurée en donnant une définition informelle de la structure logique générique, de la structure logique spécifique, de la structure physique générique et de la structure physique spécifique. Ces notions essentielles permettent de présenter les caractéristiques principales des systèmes d'édition de documents structurés identifiés ensuite par leurs fonctions et leur degré d'intégration de la structure ; on y distingue les éditeurs, les formateurs, les systèmes interactifs parmi lesquels les systèmes d'édition guidés par la structure. Les normes SGML, DSSSL, et ODA sont ensuite présentées étant donné le rôle qu'elles jouent dans l'échange de documents entre systèmes. Des domaines encore en développement, tels que les hypertextes, les hypermédias et l'édition coopérative, sont ensuite présentés. Tous les concepts qui guident l'édition structurée sont implantés dans le système Grif qui sert de base aux travaux de cette thèse.

Chapitre II

Le chapitre 2 présente le système d'édition de documents structurés Grif en insistant sur ses aspects les plus pertinents pour le problème de la transformation des types. Ce chapitre fait une présentation par l'exemple du langage S, qui possède toutes les caractéristiques de SGML. Les types définis dans un schéma de structure sont classés en types principaux et types associés. Ce chapitre insiste sur les constructeurs (connecteurs en SGML), les indicateurs d'occurrence, les références et les attributs. L'aspect modulaire des schémas de structure est mis en évidence à travers le rôle des schémas externes et des schémas d'extension dans les transformations de types.

Chapitre III

Les problèmes que pose la structuration aux éditeurs de documents structurés sont exposés dans le chapitre 3. On les classe en restructuration statique, restructuration dynamique et reconstitution de schémas de structure. Les problèmes posés par l'échange de documents entre systèmes hétérogènes et les travaux relatifs aux autres domaines sont ensuite présentés.

Chapitre IV

Le chapitre 4 s'appuie sur les caractéristiques des systèmes d'édition de documents structurés en général et le système Grif en particulier pour dresser une typologie de l'évolution des types et présenter plus en détail les restructurations dynamiques. L'étude de cette évolution porte sur toutes les définitions présentes dans un schéma de structure c'est−à−dire, les types principaux, les types associés, les attributs, les références, les schémas externes, les schémas d'extension et les relations autres que celles traduites par les références.

Chapitre V

Le chapitre 5 présente un modèle conceptuel pour les transformations statiques c'est−à−dire celles induites par l'évolution de la définition des types à travers les

schémas de structure. Ce modèle sert de base pour l'implantation des solutions proposées dans le chapitre suivant.

Chapitre VI

L'approche qui est faite du problème des transformations de structure est l'abstraction des caractéristiques des types. Cette abstraction est présentée dans le chapitre 6 en fonction de la nature des transformations. La représentation fonctionnelle des caractéristiques permet de résoudre les transformations statiques dites élémentaires. Les extensions d'ordre structural permettent de définir des relations d'équivalence, de facteur, de massif, de sous−type et de compatibilité entre types. De plus, elles établissent des relations d'ordre quotient entre classes, ainsi que l'élargissement des classes d'équivalence aux schémas externes. Ces relations et classes permettent de résoudre une grande partie des restructurations dynamiques et sont de nature à faciliter et à accélérer les transformations statiques. Les extensions basées sur le contenu considèrent un type comme une grammaire algébrique et une instance de ce type comme un mot de ce langage. L'appartenance d'un mot à un langage permet de conclure à la validité d'une transformation. L'enrichissement de ce langage par des symboles structuraux, la linéarisation des arbres sous forme de chaîne de caractères incluant la structure et le contenu, introduisent les empreintes génériques et effectives nécessaires aux restructurations dynamiques.

Chapitre VII

La conclusion de ce rapport résume les points essentiels de cette thèse et insiste sur les aspects de l'évolution des types qui méritent d'être approfondis.

Chapitre I

Systèmes d’édition de documents structurés : état de l’art

Résumé :

Ce chapitre donne un aperçu des systèmes d’édition de documents structurés. La section I.1 donne une définition informelle des notions de structures logique et physique. Les sections suivantes présentent les divers types d’éditeurs, les formateurs et les systèmes interactifs. Les sections I.5, I.6 et I.7 décrivent respectivement les normes SGML, DSSSL et ODA. Les hypertextes et les hypermédias sont présentés dans la section I.8. La section I.9 donne un aperçu de l’édition coopérative.

I.1 Introduction

Les éditeurs de documents structurés sont destinés à la production et à la consultation de documents dont ils contrôlent la structure. À travers l’expression "documents structurés", il faut entrevoir non seulement la nature des documents eux−mêmes (ce qui est certes l’objectif principal) mais aussi les caractéristiques des éditeurs de documents et les efforts de normalisation accomplis dans ce domaine. Les éditeurs de documents structurés servent avant tout à la production de documents techniques volumineux et complexes compte tenu de la diversité de la nature de leurs composants. Un document peut contenir alors du texte, des graphiques, des tableaux, des formules mathématiques, etc. La notion de structure ne s’applique pas qu’aux documents techniques et les éditeurs plus ou moins fondés sur la structure, présentent des caractéristiques différentes en fonction du type de documents qu’ils produisent. C’est ainsi que l’on peut considérer les sources des programmes, les partitions musicales, les formules mathématiques comme des documents structurés. Les systèmes et programmes qui aident à la production des documents dits structurés vont des traitements de texte aux vrais éditeurs de documents structurés en passant par les logiciels de PAO. La structuration et les problèmes qu’elle pose ne constituent pas les seuls axes de recherche du domaine de l’édition structurée qui est confrontée par ailleurs aux problèmes de formatage, d’édition coopérative, de conception de l’architecture idéale, d’échange de documents entre systèmes d’édition hétérogènes, etc. Le concept de documents structurés

lui−même évolue avec le multimédia dont le mariage avec les hypertextes permet de ne plus considérer un document comme un objet statique dont la composition et le contour sont figés. Une première étape en faveur de l'édition électronique en général passe par la normalisation de formats d'échange, d'architectures de production de documents, de langages de description de structures et de l'aspect graphique des documents. Les normes ODA, SGML, DSSSL contribuent beaucoup à l'émergence de ces systèmes. L'objectif de ce chapitre est de présenter leurs principales caractéristiques.

I.1.1 Le document

Il est difficile de donner une définition formelle du terme document. On peut considérer comme documents tous les produits d'impression (livres, revues, journaux, etc.) ou manuscrits (notes, lettres, adresses, etc.), qu'ils émanent des maisons d'édition ou des entreprises. On est en droit de considérer les codes d'exécution des programmes, les textes sources des programmes, les sons, les images, comme des documents. On se rend ainsi compte que les supports des documents sont variés : papier, disques, bandes, etc.

Un document peut être considéré comme le résultat de tâches distinctes confiées à des applications spécifiques. Les premiers éditeurs de documents faisaient de la production de documents un processus linéaire [Peels85] dans lequel intervenaient des outils spécialisés. Bien que l'architecture de la quasi−totalité des éditeurs actuels ne soit pas séquentielle, nous nous servons dans la figure Fig. 1.1 d'une représentation linéaire des tâches pour aider à la compréhension de l'exposé. À chaque étape correspond un ensemble d'outils spécialisés, mais surtout une tâche bien précise.

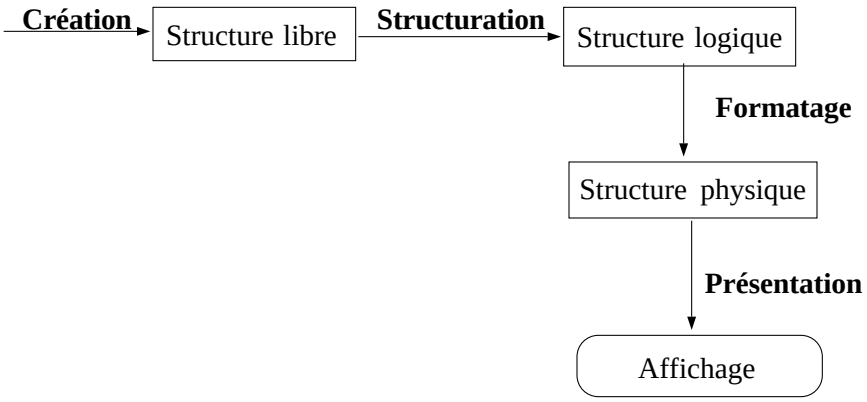


Fig. 1.1 : Modèle linéaire de production de documents

Création

La production du contenu d'un document se fait à l'aide d'au moins un éditeur. Le contenu d'un document est indifféremment constitué de données alphanumériques, de graphiques, de formules mathématiques, d'images, etc. Beaucoup d'éditeurs de documents structurés, soit produisent eux−mêmes de tels éléments soit permettent leur intégration une fois créés par des éditeurs spécialisés.

Structuration

On appelle structuration le découpage du contenu d'un document en parties dont chacune a une signification particulière pour l'utilisateur. L'organisation d'une lettre, par exemple, présente les parties suivantes : la date, l'adresse et l'identité de l'expéditeur, l'adresse et l'identité du destinataire, le titre de la lettre, son corps, la formule de politesse et la signature de l'expéditeur.

Formatage

On appelle formatage la mise en page d'un document et l'application d'attributs typographiques et de règles de découpage en lignes.

Sans entrer dans le détail des différentes phases de production de documents, nous donnons une définition informelle de la notion de structure de document, nécessaire à une bonne compréhension des problèmes induits par la structure.

I.1.2 Notion de structure de document

On peut classer les documents en trois catégories en fonction de leur degré de structuration. Certains documents sont uniquement une séquence d'objets, généralement des caractères, et sont produits par des éditeurs de texte tels que EMACS [Stallman81]. Ces documents ne sont pas concernés par la présente étude. D'autres ont une relative organisation spécifique, par exemple l'organisation d'un texte en paragraphes avec éventuellement des titres. La catégorie la plus intéressante pour notre étude est celle des documents structurés, c'est−à−dire des documents dont la structure est connue de l'éditeur. Le terme de ‘‘structure de document’’ peut être considéré de deux points de vue différents, celui de l'auteur et celui du typographe. Cette section a pour objectif de préciser la notion de structure selon ces points de vue et leurs traductions en termes de caractéristiques des systèmes d'édition de documents structurés (SEDS)(1).

I.1.2.1 Structure logique

Un auteur s'intéresse au contenu du document qu'il rédige, à son organisation, c'est−à−dire à son découpage en composants et aux relations entre ces composants. La structuration est perçue avant tout par l'auteur comme un des critères de bonne présentation visant à apporter au lecteur, un confort pendant la consultation du document. Aussi essaie−t−il d'associer à chaque composant une sémantique particulière. Un livre peut alors être découpé en titre, auteurs, résumé, chapitres, sections, annexes ; il peut aussi comporter des notes, des figures, etc. La notion de structure, du point de vue de l'auteur, prend également en compte l'ordre des composants. C'est ainsi que le titre d'un chapitre précède toujours son contenu. En effet, non seulement un titre renseigne sur le contenu de la suite mais il contribue aussi à une lecture plus aisée en offrant la possibilité de ne lire que les parties pour lesquelles on a de l'intérêt. Les composants d'un document et leurs relations déterminent l'organisation du document : c'est la structure logique du document et chaque composant est un élément de la structure logique.

---

(1) SEDS est l'acronyme de l'expression Système d'Edition de Documents Structurés

La notion de structure présentée jusqu'à maintenant n'a pas pris en compte la nature des composants. Le contenu d'un document, lorsqu'il n'est pas uniquement textuel, peut contenir des objets eux−mêmes structurés en fonction de leur nature. On peut citer les graphiques, les formules mathématiques, les programmes, les tableaux, etc. Les éléments de la structure logique d'un document contiennent souvent des éléments eux−mêmes structurés, faisant d'un document un objet très complexe.

En considérant la structure non pas d'un point de vue global mais en fonction de la nature du contenu, on se rend compte de la grande variété de structures logiques présentes dans un document. À titre d'exemple, la structure d'une formule mathématique est différente de celle d'un tableau, elle−même différente de celle d'un programme. La notion de structure logique est fondamentale et tous les SEDS proposent des modèles suffisamment puissants pour décrire la structure globale des documents en fonction de celle de leurs composants (voir I.4.2.2).

Il existe une grande diversité de documents, tels que les livres, les articles, les lettres, les contrats, les notes dont la composition et la structuration diffèrent. Pour une gestion efficace des documents, on considère chaque variété comme une classe de documents dont les éléments ont des structures logiques très voisines. À chaque classe, on associe une *structure logique générique* (voir I.4.2.2) qui donne une description exhaustive et précise des éléments qui entrent dans la composition de ses documents ainsi que les relations qui les lient. Aussi dispose−t−on des structures logiques génériques ‘‘Article’’, ‘‘Lettre’’, ‘‘Contrat’’, etc. Les SEDS se servent de ces structures génériques pour produire des documents appartenant à ces classes.

Deux documents de la même classe, par exemple ‘‘Article’’, ont la même structure logique générique mais leur composition peut ne pas être identique (voir Fig. 1.2). Ainsi le document Premier_Article n'a pas d'élément de type DateDeMiseAJour et est rédigé par un seul auteur alors que le document Deuxième_Article contient un élément de type DateDeMiseAJour et est rédigé par trois auteurs. Un document est toujours conforme à une structure logique générique mais possède une composition propre qui est sa *structure logique spécifique*.

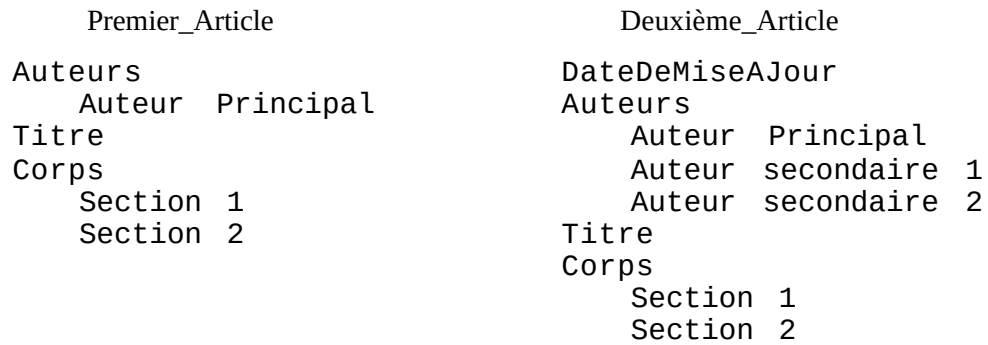


Fig. 1.2 : Deux articles de structures logiques spécifiques différentes

Les avantages de la structuration sont les suivants :

- La structuration facilite la lecture d'un document et sa mise à jour. De plus, une bonne structuration contribue à améliorer le travail des auteurs.
- On sait tirer avantage de la structure pour diriger l'édition en ce sens que seuls les éléments prévus par la structure générique pourront être créés ; ainsi, la création ou la destruction d'un élément n'est permise que si elle est compatible avec les conditions structurales locales. Cela aide à la création de documents normalisés.
- La structure logique permet de décharger l'utilisateur de la gestion des caractéristiques typographiques et de la mise en page, qui peuvent être dérivées de la structure.
- la numérotation des composants pourra être automatique et prise en charge par le formateur, en fonction de la structure.
- Des applications qui tirent parti de la structuration se voient ouvrir de nouvelles possibilités de traitement ; par exemple les systèmes d'hypertextes, les programmes de reconnaissance optique peuvent baser leur fonctionnement sur l'identification des constituants dont la structure est connue du programme.

I.1.2.2 Structure physique

Le point de vue du typographe est différent de celui de l'auteur. Il s'intéresse principalement à l'aspect graphique du document, c'est−à−dire à sa mise en page et aux attributs typographiques des caractères. La mise en page est le découpage d'un document en pages et la disposition spatiale des éléments de la structure logique que chaque page doit contenir. Elle précise la disposition d'un élément dans la page, les distances entre le bord supérieur de la page et le premier élément, la distance entre le bord inférieur et le dernier élément, la distance entre deux éléments de la structure logique, les dimensions des marges droite et gauche, la justification, l'espace entre les lignes, les sauts de page, etc.

Les attributs typographiques (police, corps, style) jouent un rôle important dans la structuration d'un document dans la mesure où ils sont destinés à attirer le regard du lecteur. Bien que le travail du typographe diffère de celui de l'auteur, il est essentiel que la typographie tienne compte de la structure logique puisque son objectif principal est la mise en évidence des éléments de la structure logique.

Les SEDS prennent en compte d'une part les besoins de structuration, d'autre part les problèmes liés au formatage et proposent pour la plupart un modèle dans lequel il y a une séparation nette entre la structure logique et l'aspect graphique du document. Le principe retenu est de construire l'image du document imprimé ou affiché en s'appuyant sur les éléments de la structure logique. Pour atteindre cet objectif, on associe à chaque élément de la structure logique un ensemble de règles qui spécifient sa mise en page et son aspect visuel : ce sont les *règles de présentation*. C'est ainsi que le titre d'un article est centré et affiché en gras avec des caractères de taille 14, que le corps de l'article est affiché après le titre et à 2 centimètres du bord gauche du support d'impression ou d'affichage, etc.

Ce principe qui ne souffre pas d'exception dans le domaine des SEDS, est présenté plus loin en détail (voir I.4.2.3). Cette conception présente de nombreux avantages :

- L'utilisateur, déchargé du travail de présentation, peut se concentrer sur la rédaction du document.
- La présentation du document est homogène, les éléments de même type étant affichés par application des mêmes règles de présentation.
- Les règles de présentation décrites en termes abstraits sont indépendantes des contraintes matérielles.
- Plusieurs vues peuvent être définies pour le même document si on associe plusieurs ensembles de règles de présentation à une même structure générique. Par exemple, on peut associer deux fenêtres au même document pour y afficher respectivement l'intégralité de son contenu et sa table des matières.

I.1.3 Caractéristiques des SEDS

D'une façon générale, les systèmes de production de documents électroniques existants sont différents quant à leurs architectures, aux fonctionnalités qu'ils offrent et à la représentation qu'ils font des documents. Certains servent essentiellement à la production de texte alors que d'autres, plus complets, tenant compte des exigences de domaines particuliers (bureautique, graphisme, édition scientifique), se sont spécialisés dans des fonctions graphiques et/ou de production de texte, fournissant des outils d'édition plus ou moins avancés. Il est intéressant de noter que les architectures continuent de susciter de l'intérêt et de poser des problèmes fondamentaux auxquels les chercheurs et les développeurs consacrent des efforts importants. Les solutions mises en œuvre à ce jour ne donnent pas entière satisfaction et les questions suivantes restent d'actualité :

- Quelles sont les caractéristiques du système idéal ?
- À quel moment de l'édition le formatage est−il réalisé ?
- Comment organiser un document, c'est−à−dire structurer son contenu ?
- Quels sont les problèmes soulevés par la structuration ?

Les systèmes de production de documents en général peuvent être classés selon plusieurs critères :

L'architecture

Plusieurs critères interviennent dans la qualification des architectures notamment le degré d'interactivité entre la création et le formatage. Certains systèmes évaluent systématiquement toute action de l'utilisateur et immédiatement mettent à jour l'affichage. D'autres, par contre, font une mise à jour de l'affichage seulement dans certaines conditions comme par exemple après un certain nombre de caractères saisis ou bien pendant le défilement de l'image, etc.

Les systèmes qui s'appuient entièrement sur la structure garantissent la production de documents corrects en guidant l'utilisateur pour la création de nouveaux éléments tout au long de l'édition. À l'opposé, avec les éditeurs−formateurs, l'utilisateur est libre de créer les éléments de son choix.

Les fonctionnalités

En dehors des fonctions d'édition il est appréciable qu'un SEDS offre des fonctions générales telles que la correction orthographique, la correction typographique, la gestion d'index, la gestion de versions, les annotations, le multilinguisme.

Généralement, chaque SEDS dispose de son propre format de sauvegarde de documents. En vue de s'ouvrir à l'extérieur, certains disposent de traducteurs qui génèrent des formats externes à partir des documents internes ou qui convertissent dans leur propre format les documents provenant d'autres systèmes. On peut citer les formats LaTeX, TeX, PostScript, Ascii, SGML, etc.

Aujourd'hui, il devient presque indispensable à un SEDS d'accepter et de produire des documents selon la norme SGML.

La représentation interne du document

La plupart des systèmes d'édition de documents structurés utilisent une structure arborescente pour représenter l'organisation des documents.

Dans le système Grif en particulier, un document est une forêt d'arbres car, comme nous le verrons plus loin (voir I.4.2.2), les éléments tels que les figures, les notes ont aussi une représentation arborescente et peuvent être placés n'importe où dans les documents.

Le degré d'indépendance entre la structure et la présentation

Presque tous les SEDS font une séparation entre la structure logique et la présentation.

La prise en compte de la structure

La première caractéristique des SEDS est la prise en compte de la structure logique des documents. L'intégration de la norme SGML est un avantage certain, étant donné son importance grandissante.

On distingue, compte tenu de ce dernier critère, deux catégories de systèmes de traitement de documents :

- Ceux qui considèrent le document comme une suite de caractères dans laquelle des instructions de présentation sont éventuellement introduites.
- Ceux pour lesquels un document est un objet structuré et qui font de la structuration la caractéristique principale du système et des objets édités. La structuration suppose donc l'existence d'éléments ayant des rôles suffisamment identifiés pour justifier leur isolement en tant qu'entités distinctes. Alors, les éléments constitutifs de la structure peuvent être considérés comme des unités syntaxiques qui permettent la composition d'un objet de plus haut niveau : le document.

I.2 Éditeurs

Les éditeurs sont des programmes informatiques dont le rôle est la production du contenu de documents avec un degré de structuration qui dépend de la nature de l'objet produit. Il est possible de les classer selon plusieurs critères :

- La nature du contenu qu'ils permettent de générer (texte, image, graphique, tableau, etc.).
- Le fait qu'ils soient interactifs ou non.
- Le modèle sur lequel ils sont bâtis.

Nous nous intéressons uniquement aux éditeurs qui produisent des documents avec une certaine structure.

I.2.1 Traitements de texte

Les systèmes de traitement de texte conservent toutes les fonctionnalités des éditeurs de texte et présentent en plus les caractéristiques suivantes : ils permettent la gestion par l'utilisateur de la présentation du texte (l'aspect de l'image affichée) en mettant à sa disposition un ensemble d'attributs typographiques tels que les styles de caractères (gras, italique, oblique, romain), les polices de caractères (Helvetica, Courrier, etc.), les corps (taille des caractères). Ils offrent des possibilités de mise en page assez sophistiquées (centrage, justification, etc .) et des possibilités d'impression. Les plus élaborés de ces outils, tels que Word [Microsoft92] autorisent une structuration du contenu du document, la constitution d'une table des matières. Sprint [Borland89] permet même de structurer un document en sections et sous−sections imbriquées. En général, ils produisent en plus du texte, des tableaux et des objets graphiques.

La caractéristique principale des traitements de texte est que la gestion des attributs typographiques est entièrement à la charge de l'utilisateur, soit directement par l'introduction de commandes appropriées à l'intention du formateur, soit par le regroupement de ces commandes dans une ressource externe appelée feuille de style [Johnson88]. Une feuille de style est un ensemble nommé de commandes de mise en page et d'attributs typographiques. L'invocation dans un document de l'identificateur d'une feuille de style est une requête faite au formateur pour qu'il applique les commandes de la feuille de style. La séparation de la description des feuilles de style du contenu du document permet de changer facilement de présentation en appliquant une autre feuille de style.

Même si les traitements de texte présentent quelques possibilités de structuration et peuvent séparer la présentation du document de son contenu, il ne sont pas considérés comme éditeurs de documents structurés : la présentation est très pauvre et incontrôlable (il n'y a pas de structure générique).

I.2.2 Éditeurs syntaxiques

Les éditeurs syntaxiques [Teitelbaum81] sont destinés à la production de programmes ; ils garantissent la production d'un texte sans erreur de syntaxe et, à cette fin, ils ont besoin de connaître le langage de programmation utilisé, en particulier sa syntaxe et sa sémantique

[Donzeau84a][Donzeau83]. Ils fournissent une assistance à l'utilisateur en lui procurant les éléments syntaxiques valides dans le contexte courant, pour qu'il se consacre uniquement à la saisie des instructions. Il est intéressant de remarquer que la plupart des langages de programmation disposent de leurs éditeurs syntaxiques spécifiques. C'est le cas des langages tels que C, Pascal (Cornell Program Synthesizer [Teitelbaum81]). Synthesizer est doté d'un générateur d'éditeurs qui produit un programme spécifique à chaque langage [Reps84]. Cependant, il existe des éditeurs syntaxiques généralisés capables de reconnaître plusieurs langages de programmation ; ils sont paramétrables par la description syntaxique des langages avec lesquels l'utilisateur désire travailler. PECAN [Reiss84] et Centaur [Borras87] font partie de cette dernière catégorie.

Dans le domaine des langages de programmation, des études ont été faites sur les considérations typographiques et de mise en page des documents que constituent les sources des programmes. À titre d'exemple on peut citer les travaux de Baecker et Marcus [Baecker90] fondés sur le langage C et ceux d'Oman et Cook [Oman90] qui considèrent qu'une bonne présentation contribue à la compréhension des programmes.

Les éditeurs syntaxiques sont plus proches des SEDS que les traitements de texte. La grammaire du langage de programmation joue le même rôle que les schémas de structure génériques des SEDS. Ils ne sont cependant guère utilisables pour produire des documents (voir en particulier [Quint90]).

I.2.3 Éditeurs spécialisés non textuels

Le contenu d'un document peut être de diverses natures. Un document peut contenir :

- du texte,
- des graphiques,
- des formules mathématiques,
- des photos, etc.

Le problème qui se pose ici est la production et l'intégration de ces éléments dans le document. Comme indiqué plus haut, certains traitements de texte gèrent directement la production d'objets non textuels en intégrant des éditeurs spécialisés. D'autres par contre ne permettent que l'inclusion d'objets non textuels produits avec des éditeurs spécialisés ou décrits avec un langage de formatage. On peut citer dans cette catégorie MacWrite [Williams84] qui permet l'inclusion d'éléments non textuels sans toutefois offrir la possibilité de les éditer.

Les éditeurs interactifs utilisent plusieurs outils spécifiques en vue de produire des éléments non textuels qui sont ensuite insérés dans les documents. Dans cette catégorie figurent MacDraw [Williams84], SuperPaint, xfig, etc, qui considèrent un objet graphique comme un ensemble de constituants de base (lignes, rectangles, ellipses, textes, etc.) positionnés relativement les uns par rapport aux autres. Il existe des éditeurs interactifs spécifiques à des domaines particuliers, tels que la musique, les mathématiques, etc.

I.3 Formateurs

Un formateur est un programme chargé de produire l'image (la présentation) d'un document à partir d'une description faite dans un langage qui lui est propre. Il s'agit alors de spécifier pour chaque partie du document un certain nombre de paramètres par l'intermédiaire du langage associé au formateur. La présentation peut être spécifiée pour la globalité du document ou pour toute partie de celui−ci à l'aide des caractéristiques suivantes :

- attributs typographiques des caractères (style, police, corps, graisse),
- espacement des lignes,
- détermination des marges,
- justification,
- césure des mots,
- placement de sauts de pages, etc.

Les premières générations d'outils de formatage obéissaient au concept suivant : le contenu du document est parsemé de balises qui sont en fait des instructions destinées à exprimer la représentation finale souhaitée. Ces instructions, comme celles offertes par T_EX, ne servent pas à exprimer la structure logique des documents mais seulement leur aspect graphique. Une telle approche est dite procédurale. Le document ainsi décrit est donné en entrée au formateur comme indiqué dans la figure Fig. 1.1. Il existe deux types de formateurs, les formateurs de bas niveau et les formateurs de haut niveau :

1. Les formateurs de bas niveau précisent la présentation des documents en terme des paramètres énumérés plus haut, rendant ainsi leur utilisation minutieuse. Leurs langages sont très riches et ils fournissent souvent une bonne qualité typographique. Cette description est faite séparément pour chaque instance de document rendant l'utilisation de ce type de formateurs assez fastidieuse. On peut citer en exemple T_EX [Knuth85a] et Troff [Saltzer65].

2. les formateurs de haut niveau soulagent l'utilisateur de la connaissance des langages de bas niveau, en fournissant des macro−instructions susceptibles de réaliser un ensemble d'actions de bas niveau. Il s'agit en fait d'un langage déclaratif, lequel indique au formateur les types d'éléments qui composent les documents. Les formateurs peuvent ainsi appliquer aux parties des documents dont la présentation utilise les mêmes macro−instructions, des traitements identiques garantissant une certaine homogénéité dans l'affichage et l'impression. Le lecteur trouvera des informations plus complètes sur la représentation des documents, l'évolution des systèmes de production et les concepts utilisés dans [Quint89]. On peut remarquer que les documents sont traités comme des entités structurées ou comme une composition d'objets typés grâce à un ensemble de descriptions typographiques disponibles dans le système et qui leur confèrent une homogénéité de présentation : il s'agit bien d'un début de structuration. Dans cette catégorie de formateurs, on peut citer Scribe [Reid80] et LAT_EX [Lamport85] une expression de haut niveau construite sur les commandes de bas niveau de T_EX.

Les formateurs disposent d'un langage qui permet d'inclure dans les documents la description de formules mathématiques et d'objets graphiques. Ils réalisent également les traitements d'ensemble suivants sur les documents :

- création de tables d'index,
- gestion des entrées bibliographiques,
- numérotation des parties,
- création de tables des matières,
- gestion des renvois.

Deux formateurs vont être présentés, Scribe pour le rôle historique qu'il a joué dans le concept de formatage de haut niveau, et Latex pour la référence qu'il constitue parmi les formateurs.

I.3.1 Scribe

Dans le domaine des systèmes pour les documents structurés, Scribe [Reid80] a joué un rôle important dans la mesure où il a introduit des concepts qui ont été certes améliorés par la suite mais qui constituent une contribution importante.

Scribe est un formateur qui se compose d'un langage et d'une base de données qui contient deux sortes d'informations :

- une collection de descriptions typographiques et de mise en page qui guide la présentation des documents. Une partie importante de Scribe est son langage déclaratif qui permet d'introduire dans le texte soit des commandes à l'intention de son compilateur, soit des marques de début et de fin qui délimitent des régions types appelées environnements dont voici quelques exemples :
  - énumération,
  - description,
  - italique,
  - citation,
  - exemple.
- une collection de descriptions abstraites de documents exprimées dans le langage Scribe :
  - article,
  - rapport,
  - thèse,
  - manuel, etc.

Les environnements peuvent être imbriqués et modifiés localement par l'utilisateur. Comme les autres formateurs évolués, Scribe gère les bibliographies, les renvois, la numérotation et autorise le découpage des documents volumineux en unités dont l'ensemble est traité comme un document unique du point de vue des fonctionnalités globales offertes. Il a inspiré le développement de nombreux systèmes, notamment LATEX.

I.3.2 LaTeX

LaTeX est un formateur de haut niveau qui fournit un ensemble de macro−instructions destinées à masquer la richesse de TeX jugé difficile à maîtriser. Son langage simple à utiliser permet la description des formules mathématiques, des tables et des objets graphiques. Il permet de créer des entrées bibliographiques, de construire des tables d'index et des matières, de numéroter les composants des documents, etc.

I.4 Systèmes interactifs

Un système de production de documents est dit interactif s'il a les caractéristiques suivantes :

1. Il intègre les outils de formatage et de saisie de contenu.
2. Les parties affichées du document édité sont formatées selon les règles typographiques et de mise en page qui leurs sont associées, de manière à fournir au rédacteur une image proche de celle obtenue une fois le document imprimé.
3. Il est doté d'une interface qui permet au rédacteur d'agir sur la présentation du document, notamment de modifier directement les attributs typographiques du document avec une répercussion immédiate sur l'image affichée.
4. Une mise à jour de l'affichage a lieu à chaque modification du contenu.

Les systèmes interactifs diffèrent en fonction du degré de structuration qu'ils autorisent dans la modélisation des documents. On y distingue deux familles : les éditeurs−formateurs et les éditeurs de documents structurés.

I.4.1 Éditeurs−formateurs

Les éditeurs−formateurs opérant en mode interactif comme Microsoft Word [Microsoft92], Interleaf [Morris85], FrameMaker et XPress n'ont pas une vision structurale de l'ensemble du document, en ce sens qu'ils n'offrent pas de description logique globale des documents. Les documents y sont construits selon un modèle de liste de composants tels que les paragraphes. Les éditeurs−formateurs offrent un ensemble de types qui décrivent les constituants plausibles d'un document. Le rédacteur se sert donc de ces types pour construire son document. Ces types, qui ressemblent aux feuilles de style, sont en fait des spécifications de règles de mise en page et d'attributs typographiques utilisés pour la présentation des composants. En réalité, il existe dans ces systèmes deux sortes de feuilles de style :

1. les feuilles de style pour les pages,
2. les feuilles de style pour les composants du document.

Ainsi, pour écrire un article, utilisera−t−on les feuilles de style *Date*, *Auteur*, *Titre*, *Résumé*, *Titre de Section*, *Titre de SousSection*, *Paragraphe*, *Annexe*, par exemple. Il est intéressant de remarquer que les feuilles de style ne servent pas tant à structurer le document dans sa globalité qu'à associer des présentations spécifiques à ses constituants. Cette catégorie d'éditeurs voit le document comme une séquence d'objets puisque ses

constituants peuvent être placés n'importe où dans le document sans aucune contrainte de structure (position, parenté). Il n'y a aucun traitement de cohérence entre ces constituants qui sont improprement appelés types. Ces éditeurs comportent pour la plupart des outils spécialisés pour l'édition de graphiques, de formules mathématiques, etc. Ces solutions partielles n'ont certes pas apporté de réponses réelles et globales aux problèmes de structuration, mais leur disponibilité a été appréciée par les utilisateurs et elles ont contribué de manière décisive à la conception du modèle.

I.4.2 Éditeurs de documents structurés

La tendance actuelle de la structuration des documents s'est largement inspirée des concepts sur lesquels sont basés les éditeurs syntaxiques et le formateur Scribe. Un environnement de production de documents structurés devrait intégrer un éditeur de documents et des possibilités de formatage et de structuration ; il devrait, en outre, être ouvert sur les systèmes externes en vue de permettre des échanges. De nombreux systèmes ont été proposés, dont certains sont commercialisés :

- Tioga [Teitelman85] (Xerox Parc) permet la création de documents sous forme arborescente en mettant à la disposition de l'utilisateur un ensemble de feuilles de style. Les documents sont sous forme d'arbres dont les composants ne sont pas contraints par des relations de parenté déterminées, ce qui laisse une certaine souplesse dans l'édition.
- Author/Editor [Maloney88] disponible sur Macintosh, PC et sous Unix, requiert des descriptions de la structure des documents au moyen du langage SGML.
- Grif [Quint87] sert de base aux travaux de la présente thèse et fera l'objet d'une présentation détaillée dans le chapitre suivant.
- Rita [Cowan91] est un système de production de documents structurés fondé sur le même modèle que Grif.
- Ensemble [Graham93] est un système de production de documents structurés doté d'un système original de présentation appelé Proteus [Graham92].

I.4.2.1 Modèle de document structuré

Les SEDS, à l'inverse des éditeurs−formateurs, ont une connaissance précise de la structure du document qu'il produisent, grâce à la structure logique générique qui a déjà fait l'objet d'une présentation informelle (voir I.1.2). Aussi, les SEDS ne laissent−ils pas l'utilisateur créer n'importe quels éléments n'importe où comme dans les éditeurs−formateurs, mais lui proposent−ils de ne créer que les éléments prévus par la structure logique générique.

Le modèle proposé par les éditeurs de documents structurés est fondé sur la représentation du document en trois niveaux distincts :

- la structure logique,
- la structure physique,
- le contenu.

Une définition plus formelle de la structure logique générique (voir I.4.2.2) et de la structure physique (voir I.4.2.3) permet de comprendre comment ces trois niveaux sont représentés dans les modèles.

À chacun des deux concepts de structure logique et de structure physique correspond une ressource indispensable aux SEDS pour aider l'utilisateur à produire un document correct du point de vue de la syntaxe, tout en le déchargeant des efforts de présentation.

I.4.2.2 Structure logique générique

On appelle structure logique générique l'organisation générale d'une classe de documents en termes de tous leurs composants possibles, sur la base d'une sémantique intelligible par l'homme ; par exemple Titre, Chapitre, Section, SousSection, Bibliographie, etc.

Pour exprimer cette organisation, les systèmes sont dotés d'un langage de description qui identifie les éléments de la structure logique générique et qui spécifie pour chacun d'eux les relations qu'il entretient avec d'autres éléments :

1. Un élément spécial est identifié comme l'élément principal de la structure générique. C'est lui qui représente l'ensemble des éléments qui composent un document. D'autres éléments seront identifiés comme contribuant à sa composition.
2. Les relations hiérarchiques pour un élément donné, précisent son ascendant immédiat (son père), l'élément placé immédiatement avant lui (son frère aîné), l'élément placé immédiatement après lui (son frère cadet) et tous ses descendants immédiats.

   Les éléments concernés par les relations hiérarchiques appartiennent au même document.
3. Les relations non hiérarchiques servent à exprimer des liens, comme les renvois à des figures, à des notes, etc.

   Les éléments concernés par les relations non hiérarchiques peuvent appartenir au même document ou à deux documents différents, lesquels peuvent provenir de deux structures logiques génériques différentes.

Définitions

L'ensemble des relations définies pour un élément donné est identifié par le nom de l'élément et doit être compris comme un ensemble de règles à appliquer pour produire un élément de document conforme aux dites règles. Pour la suite de l'exposé, le mot *type* désigne l'identificateur d'un ensemble de règles dans la structure logique générique. Nous dirons enfin que l'ensemble des règles correspondant à un type donné constitue la définition de ce type donné.

Un élément de document est produit conformément à un type et est identifié comme une *instance* de ce type.

La structure logique est dite générique dans la mesure où elle permet de produire des instances de documents conformes à un genre dont elle contient description. Sans décrire dans le détail les langages d'expression des structures logiques génériques (voir SGML I.5.1 et S), il est intéressant de remarquer qu'ils disposent tous de constructeurs dont les plus importants sont décrits ci−dessous :

- La liste exprime la possibilité de construire un certain nombre d'éléments contigus de même type dans un document ; ce nombre peut éventuellement être limité. C'est ainsi que le corps d'un document de type Article peut être défini comme une liste de sections.
- L'agrégat ordonné spécifie que les éléments qui le constituent doivent rester groupés et respecter un ordre donné. Par exemple, dans une instance de type Section, le titre doit précéder le corps.
- L'agrégat non ordonné spécifie que les éléments qui le constituent doivent rester groupés dans n'importe quel ordre.
- Le choix définit un ensemble composé d'un certain nombre d'autres types appelés options. Une seule option participe à la production des instances du type de constructeur choix auquel elle appartient. Par exemple, un paragraphe peut être défini comme un ensemble de types permettant de construire des objets divers tels que des formules mathématiques, des graphiques, du texte, des images, etc.

La description de la structure logique n'est pas toujours suffisante pour que les documents soient traités correctement. En effet, des informations telles que l'aspect typographique du contenu, la langue dans laquelle le document est écrit, ne sont pas définies comme des éléments de structure. Bien souvent, ces informations d'ordre sémantique sont nécessaires au bon fonctionnement d'applications telles que les formateurs, les correcteurs syntaxiques, les correcteurs typographiques. Elles sont définies dans les schémas de structure sous la forme d'attributs pour les types qui le nécessitent. Les instances de ces types pourront alors prendre des valeurs prévues par les attributs et permettre ainsi les traitements souhaités.

Un certain parallèle peut être fait avec les langages de programmation compte tenu d'une part de la vision structurale qu'ils ont des programmes, d'autre part du concept de type qui les caractérisent.

La plupart des systèmes de production de documents structurés donnent la possibilité à l'utilisateur de définir ses propres structures logiques génériques.

Notation

Tous les SEDS adoptent une représentation interne arborescente des documents qu'ils manipulent. Dans un arbre, un nœud est étiqueté par l'identificateur du type (éventuellement entouré de deux caractères spéciaux) dont il est la racine de l'arbre d'instance ou de type. Les caractères qui entourent l'identificateur de type dans un nœud, renseignent sur la nature de son constructeur comme résumé dans le tableau ci−après :

Agrégat ordonné	Agrégat sans ordre	Liste	Choix
{type**}**	↑type↓	**(**type**)**	**[**type**]**

L'exemple de la figure Fig. 1.3 est la représentation arborescente(2) partielle d'un document, structurée comme une *Section* ; la racine du sous−arbre est étiquetée avec l'identificateur du type correspondant (Section). Le premier fils de ce sous−arbre est le titre de la section repéré par le nom de son type (TitreSection) qui lui−même a pour fils unique une chaîne de caractères de type TEXTE. Le deuxième fils de l'arbre, le corps de la section, est de type CorpsSection et est composé de trois éléments de type Contenu. Un élément de type Contenu contient soit du texte, soit des formules mathématiques.

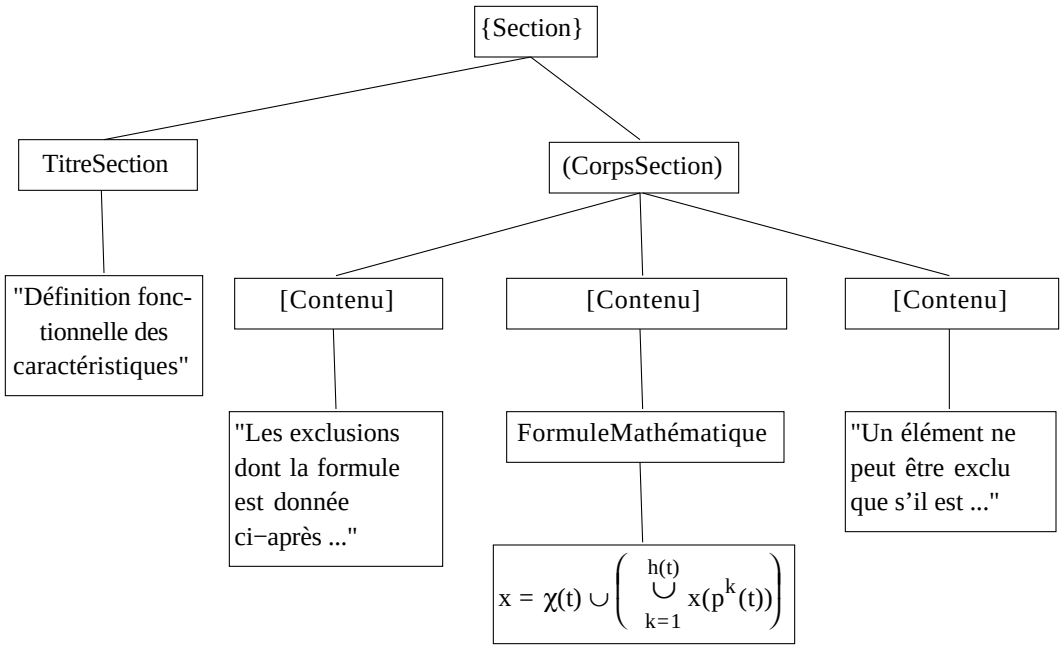


Fig. 1.3 : Représentation arborescente d'une instance de type section.

I.4.2.3 Modèle de présentation

On appelle *règles de présentation* l'ensemble des règles de mise en page et de spécification des attributs typographiques associés à un type défini dans un schéma de structure. Une règle de présentation comporte des règles de mise en page et une spécification d'attributs typographiques.

On appelle *modèle de présentation*, l'ensemble des règles de présentation de tous les types d'une même structure logique générique.

D'une façon générale, les règles de mise en page (composant des règles de présentation) s'appuient sur la notion de boîte telle que présentée dans TEX [Knuth85a]. À chaque élément logique on associe une boîte dont les règles de mise en page expriment la dimension et la position par rapport aux boîtes des éléments qui la précèdent et qui lui succèdent.

Ces règles de présentation sont exprimées au moyen d'un langage assez puissant pour

(2) Les nœuds dont l'étiquette ne comporte pas de caractères autres que ceux qui composent le nom de son type correspondent à un constructeur particulier appelé Identité qui fera l'objet d'une présentation détaillée au chapitre II.

permettre la description de l'aspect graphique et de la mise en page. À titre d'exemple on peut citer les langages P (Grif) et la norme DSSSL [I.S.O91a] destinés à exprimer la présentation des types définis dans les structures logiques génériques (voir I.4.2.2). Un autre exemple intéressant est Proteus [Graham92], le système de présentation du SEDS Ensemble [Graham93], d'ailleurs très proche de Grif.

La définition des modèles de présentation n'est pas détaillée ici parce qu'ils ne présentent pas d'intérêt pour le problème d'évolution de types ([GrifMint] [Quint86b] [Graham92] [Graham93]). Plusieurs modèles de présentation peuvent être définis pour une même structure logique générique.

I.4.2.4 Normes

Parallèlement aux travaux qui ont conduit à l'état actuel de l'édition structurée, il y a eu un effort de standardisation de langages de structuration et de formatage de documents afin de permettre une nette séparation de la structure et du contenu, des traitements applicables aux documents. Il en résulte un avantage considérable qui est la production de documents réutilisables puisqu'indépendants des applications. Ces documents pourront ainsi être échangés, avec comme garantie, le respect de la structure et de la présentation. Les modèles SGML (voir I.5), DSSSL (voir I.6) et ODA (voir I.7) sont présentés, ici d'une part parce qu'ils sont des standards, d'autre part parce qu'ils illustrent bien les concepts sur lesquels se fondent les documents structurés.

I.5 SGML

SGML (Standard Generalized Markup Language : ISO 8879) est un langage de représentation de documents destinés à être traités par des applications diverses (formateurs, systèmes de recherche d'information, etc.) et à faire l'objet d'échange entre systèmes. La norme exige d'un document qu'il comporte deux parties différentes, indépendamment de tout système de production de documents : son contenu et la description de sa structure générique. Le principe fondamental qui a conduit à la conception de SGML est le balisage des documents par des marques décrites dans les structures logiques génériques. Les documents sont, de ce fait, organisés en éléments typés, identifiés grâce aux marques (un élément de la structure logique générique est un type qui sert à marquer). Les applications qui manipulent de tels documents interprètent cette structure à leur manière et accomplissent des traitements spécifiques. Il n'est pas obligatoire d'inclure dans le document des commandes de traitement particulières à l'intention d'une quelconque application (même si SGML le permet). Ce langage offre les moyens suivants :

- une syntaxe abstraite,
- une syntaxe concrète de référence,
- une possibilité de structuration par la déclaration d'attributs et de types,
- une possibilité de référencer des éléments,
- la possibilité d'inclure dans les documents des commandes spécifiques à l'intention des applications.

Le langage est suffisamment riche pour permettre la description structurale des données de types texte, tableau, formules, etc. Les principaux apports [Barron89] de la norme sont ainsi résumés :

1. une séparation de la structure et de la présentation,
2. une indépendance vis à vis des contraintes matérielles,
3. une indépendance vis à vis des applications,
4. une prise en compte de données non textuelles,
5. la normalisation d'un format d'échange de documents SGML (SDIF).

SGML est un langage qui dispose de types de base et de constructeurs qui permettent, comme dans les langages de programmation, de construire des objets plus complexes. Ce sont ces types construits qui servent à marquer un document c'est−à−dire à le structurer. SGML va nous servir à illustrer les notions de structures logiques (voir I.4.2.2) et la déclaration des structures logiques génériques (voir I.5.3).

I.5.1 Marquage et structure

Un document conforme à la norme SGML comporte, en plus de son contenu, un ensemble de marques définies dans une structure générique appelée *DTD* (Document Type Definition : voir I.5.3.5).

On appelle *marquage* d'un document, l'ajout à son contenu, de données fournissant des informations utilisables par les applications qui auront à le manipuler.

Plusieurs sortes de marques sont nécessaires pour structurer un document. Les plus couramment utilisées sont :

- les marques descriptives,
- les instructions de traitement,
- les références à des objets.

Les marques descriptives

Les marques descriptives servent à donner sa structure au contenu d'un document ; ce sont les marques les plus utilisées. Le marquage d'un élément nécessite deux marques, une *marque de début* et une *marque de fin* qui servent à encadrer son contenu. Selon la syntaxe SGML, la marque de début d'un élément d'identificateur Type, s'écrit <*Type*> et sa marque de fin s'écrit </*Type*>. À titre d'exemple, la donnée représentée par la chaîne de caractères "Isabelle" est marquée comme suit : <Prénom> Isabelle </Prénom>.

Les références aux entités

Une entité est un identificateur auquel on a associé une donnée sous forme de chaîne de caractères. Elle doit être considérée comme une constante dont la référence est une requête faite à une application, d'inclure l'entité (le texte correspondant) en lieu et place de la référence. À cet effet, les facilités offertes par les gestionnaires de référence doivent inclure les fonctions communément appelées

substitution de symbole et *insertion de fichier*. Il existe deux types d'entités, les *entités générales* et les *entités paramètres*.

Les entités générales

Les entités générales sont des données incluses dans le contenu même du document et sont traitées par les applications qui manipulent le contenu du document. Elles ne peuvent être référencées que depuis le contenu d'un élément ou depuis une chaîne de caractères représentant la valeur d'un attribut. Une entité doit être déclarée avant son utilisation comme l'entité générale *OPERA* déclarée ci−après :

```
<!ENTITY OPERA "Outils Pour les documents
    Électroniques, Recherche et
    Applications">
```

Elle peut être utilisée ensuite dans un paragraphe. La référence à une entité générale commence par le caractère &, alors que la référence à une entité paramètre commence par le caractère %. Toutes deux finissent par un point−virgule.

```
<Paragraphe>
La partie de l'équipe &OPERA; résidant à
    Grenoble va organiser une ...
</Paragraphe>
```

La résolution de cette référence donne le résultat suivant :

```
<Paragraphe>
La partie de l'équipe Outils Pour les
    documents Électroniques, Recherche et
    Applications résidant à Grenoble va
    organiser une ...
</Paragraphe>
```

Les entités paramètres

Les entités paramètres, par contre, sont des données à l'intention du système SGML. Elles sont uniquement utilisables dans les déclarations de types et dans les sections marquées(3). Les sections marquées fournissent aux applications les moyens de faire des traitements sur les parties appropriées des documents comme dans une compilation conditionnelle.

La référence aux entités présente les avantages suivants :

- référence de longues chaînes de caractères dans le cas en particulier où la même chaîne est utilisée plusieurs fois ;
- importation de sous−documents en vue de composer un hyper−document.

---

(3) Les sections marquées sont des parties du document dont le traitement par une application est soumis à certaines conditions ; par exemple, le texte correspondant à plusieurs versions d'un même document est stocké dans le même fichier de telle sorte qu'il comporte les parties communes à toutes les versions et les parties qui leur sont spécifiques. Ces parties spécifiques sont marquées par les numéros de version adéquats, permettant ainsi le choix de la version convenable.

- utilisation individuelle des caractères, notamment spéciaux, facilitant en partie l'échange entre systèmes différents ;
- récupération dynamique de résultats de l'exécution d'instructions de traitement.

Les instructions de traitement

Ce sont des informations ajoutées au contenu d'un document à l'intention d'une application ou d'un système, en vue de lui faire accomplir des actions particulières, comme les commandes de formatage ou de récupération de la date courante, etc. Ces instructions, écrites dans un langage propre à un système, en sont dépendantes. Étant donné que les instructions de traitement réduisent la portabilité, SGML recommande qu'elles soient déclarées en tant qu'entités afin de faciliter leur localisation par le destinataire du document en cas de mise à jour. Il existe deux types d'instructions de traitement :

- celles dont l'exécution génère des données à l'intention de l'application ; elles doivent être déclarées comme entités ayant pour type l'un des types de base spéciaux prévu dans SGML ;
- celles dont l'exécution ne retourne pas de données (saut de page, changement de fonte, etc.).

Le lecteur intéressé est invité à consulter la clause 8 de [Goldfarb90] pour plus d'information.

Une fois marqué, le document est soumis à l'action d'un programme spécial appelé *analyseur syntaxique SGML* qui vérifie sa conformité aux DTD utilisées. À cet effet, l'analyseur parcourt la totalité du contenu du document et analyse individuellement ses caractères en vue de déterminer leur rôle.

I.5.2 Types de base

Les types de base SGML servent à préciser la nature du contenu des éléments et à déterminer le comportement de l'analyseur syntaxique SGML à leur égard. Ils précisent à l'analyseur si un élément doit être soumis à son action (données dites SGML) ou s'il est destiné à être traité par une autre application (données non SGML). Leur diversité permet de couvrir la quasi−totalité des sortes de données et permet la composition de documents de types multimédia. Les types de base sont brièvement décrits ci−après (pour plus de détail, consulter [Herwijnen91] [Goldfarb90]) :

PCDATA (parsed character data)

Un élément de type PCDATA est un élément terminal de type alphabétique. Il est interprété comme une liste de caractères qui sera parcourue par l'analyseur syntaxique SGML à la recherche de références à des entités, de références numériques à des caractères, de marques de début et de marques de fin de composants structuraux.

CDATA (character data)

Les éléments de type CDATA sont également des éléments terminaux de type alphabétique. Leur contenu est soumis à une analyse syntaxique particulière dans la mesure où seuls le couple de caractères </ et le caractère / sont reconnus en tant que tels. Ce type de base est réservé aux éléments qui font un usage non conventionnel des délimiteurs comme par exemple les formules mathématiques, les programmes, etc. L'exemple suivant correspond à la déclaration numéro 13 de la figure Fig. 1.4 :

```
<Programme  Langage=C++>
w = (a < b)? (c += 2): tab[a][b];
</Programme>
```

SDATA (specific character data)

Il s'agit d'un type de base pour des données qui ne sont pas soumises à l'analyseur syntaxique SGML. Ces données sont composées de caractères valides SGML et sont destinées à l'action d'un système particulier. Par exemple, une formule mathématique écrite en T_EX est incluse dans le document comme une notation de type T_EX pour être formatée.

Le type de base SDATA peut être présent dans la définition d'une entité d'instruction de traitement. Dans ce cas, il indique que l'instruction de traitement doit retourner un résultat à l'application qui traite le document.

SDATA sert également à déclarer en tant qu'entité des caractères inexistants dans le jeu de caractères utilisés pour coder un document. Les deux exemples suivants correspondent aux entités représentant respectivement les caractères 'À' et 'É' :

```
<!ENTITY Agrave SDATA "[Agrave]"  -- À -->
<!ENTITY Eaigu  SDATA "[Eacute]"  -- É -->
```

RCDATA (replaceable character data)

Les éléments de type RCDATA sont traités comme ceux de type CDATA par l'analyseur syntaxique à une exception près : les références aux entités et aux caractères sont recherchées en vue d'une résolution. Par exemple, l'écriture de l'expression $A[i][j] = A_{ij} - A_{ik} + A_{kk} * A_{kj}$ de type Expression nécessite les déclarations suivantes :

```
<!ENTITY   IndI "i"    --  indice i        -->
<!ENTITY   IndJ "j"    --  indice j        -->
<!ENTITY   IndK "k"    --  indice k        -->
<!ELEMENT  Expression  --  RCDATA         -->
```

Un tel élément sera déclaré comme suit :

```
<Expression>
A[i][j] = A&IndI;&IndJ; - A&IndI;&Indk
          + A&IndK;&IndK;*A&IndK;&IndJ;
</Expression>
```

NDATA (Non−SGML data)

NDATA permet de spécifier le codage des données avec des caractères autres que ceux de SGML. De telles données ne sont pas analysées par SGML et sont destinées exclusivement à des applications spécifiques. Il est indiqué pour les données de type binaire telles que les images numérisées dont les combinaisons de bits ne sont pas interprétées de la même façon que celles définissant les caractères ASCII par exemple.

ANY

Ce mot−clé indique que le contenu d'un élément est, soit de type PCDATA, soit de l'un des types définis dans la DTD.

EMPTY

Les éléments de contenu vide(9) sont de type *EMPTY* ; il s'agit généralement des éléments pour lesquels l'utilisateur ne rentre pas lui−même les données ou pour lesquels des applications sont chargées de générer le contenu. Ainsi, la date du jour, la table d'index et la table des matières peuvent−elles être dynamiquement engendrées par l'application. Par exemple, les références vers d'autres éléments du document sont des éléments toujours vides et l'élément référencé est indiqué par un attribut associé à l'élément vide.

I.5.3 Déclaration de DTD

Les types correspondant aux balises utilisées dans un document sont définis dans une structure générique appelée *DTD* (*Document Type Definition*). SGML donne à l'utilisateur la possibilité de nommer à sa guise les éléments qui conviennent le mieux au découpage logique qu'il aura imaginé.

Cette description est faite de manière à donner aux documents une structure arborescente dans laquelle un élément entretient des relations avec d'autres éléments du document selon le principe exposé en I.4.2.2.

(9) Un élément de contenu vide ne doit pas avoir de marque de fin.

```
01  <!ELEMENT  Article – –(Entête,Section+, Annexe*)>
02  <!ELEMENT  Entête       – – (Titre  &  Auteur+)  >
03  <!ENTITY  %Ident           "(Prénom, Nom)"     >
04  <!ELEMENT  Auteur        – – (Photo?, %Ident;)  >
05  <!ELEMENT  (Prénom | Nom) – – (#PCDATA)         >
06  <!ELEMENT  Titre         – – (#PCDATA)          >
07  <!ELEMENT  Section    – – (Titre, Paragraphe+)    >
08  <!ELEMENT  Paragraphe    – – (#PCDATA|ParaRef,
                                  Programme|Figure
                               )                    >
09  <!ELEMENT  Figure  – – (Illustration, Titre)      >
10  <!ELEMENT  Illustration – –   EMPTY               >
11  <!ELEMENT  Annexe   – – (Paragraphe+)–(Figure)  >
12  <!ELEMENT  RefPara       – o EMPTY              >
13  <!ELEMENT  Programme     – – CDATA              >
14  <!––   Définitions   d'attributs                ––>
15  <!ATTLIST Paragraphe  IdPara    ID   #REQUIRED >
16  <!ATTLIST RefPara    Para IDREF   #REQUIRED >
17  <!ATTLIST Article Edition (Read,ReadWrite)Read >
18  <!ATTLIST Programme Langage (C,  C++,  Pascal)C   >
```

Fig. 1.4 : Extrait de la DTD d'un article

Un exemple de DTD est donné dans la figure Fig. 1.4. Il s'agit de la définition logique générique de la structure d'un article composé d'un en−tête, d'une liste de sections et d'une liste d'annexes. Les numéros en tête de chaque ligne de déclaration ne font pas partie de la syntaxe SGML ; il s'agit d'une disposition de commodité en vue de référencer aisément les définitions. La définition d'un élément (type) commence par la marque de début de définition d'élément *<!ELEMENT* suivi de son identificateur, du corps de la définition et du caractère de fin de définition '>'. Le corps d'une définition donne une précision sur la composition du type, son nombre d'occurrences, l'ordre de ses composants.

I.5.3.1 Composition

La définition du contenu d'un élément est appelée *modèle de contenu*. Le modèle de contenu est l'énumération des composants du type. Il est constitué de deux parties : le *groupe de modèle*(4) et les *exceptions* optionnelles au groupe de modèle. La déclaration 11 de la figure Fig. 1.4 associe le modèle de contenu **(Paragraphe+) −(Figure)** au type Annexe alors que la déclaration 01 associe le modèle de contenu **(Entête, Section+, Annexe*)** au type Article.

Un groupe de modèle est un ensemble (placé entre parenthèses) d'identificateurs de types définis par ailleurs dans la DTD, de types de base et/ou d'entités. Ces identificateurs sont séparés par des *connecteurs* qui indiquent les relations qui les unissent. Ils peuvent aussi être suffixés par un caractère spécial qui précise les conditions de leurs occurrences.

Les exceptions modifient les groupes de modèle en y imposant l'exclusion de certains de leurs composants et/ou en y autorisant l'inclusion d'éléments qui ne font pas partie de leur définition immédiate. Les effets des exceptions s'appliquent aussi aux groupes de modèle

(4) Dans la terminologie SGML, un groupe est une unité syntaxique dans la déclaration d'un type.

des composants du groupe de modèle qui appartient au même modèle de contenu qu'elles. La déclaration 11 associe l'exception **−(Figure)** au groupe de modèle **(Paragraphe+)** du type Annexe ; il s'agit d'une exclusion. Un élément ne peut être exclu que s'il est optionnel et répétable I.5.3.3, ou s'il appartient à un groupe d'éléments au choix. Les exceptions sont des raccourcis d'écriture qui permettent l'utilisation dans une DTD de types déjà définis pour définir d'autres types, évitant ainsi de réécrire une définition largement présente. Elles expriment la définition d'un type en terme de certaines différences que ce dernier présente avec une définition déjà existante :

1. La définition 11 de la figure Fig. 1.4 définit le type Annexe comme identique au type Paragraphe+ diminué du type Figure :

   <!ELEMENT Annexe − − (Paragraphe+) −(Figure)>

2. Si on avait voulu autoriser les notes de bas de page dans tout l'article, l'inclusion suivante serait apportée à la définition 01 :

   <!ELEMENT Article − − (Entête, Sections+, Annexe*) +(Notes)>

I.5.3.2 Connecteurs

La définition d'un type indique l'ordre dans lequel doivent coexister les instances correspondant à ses composants. Cet ordre s'exprime par des caractères spéciaux appelés *connecteurs*, placés entre les éléments constitutifs d'un groupe de modèle. Les caractéristiques que confèrent les connecteurs au groupe de modèle sont les suivantes :

- **Structure de groupe ordonné**

  Le groupe ordonné a la même signification que l'agrégat ordonné (voir I.4.2.2). Il est indiqué par le caractère ',' (virgule) utilisé comme séparateur dans un groupe de modèle. Dans ce cas, les instances des types du groupe de modèle doivent rester groupées et dans le même ordre que leur type respectif dans le groupe de modèle. D'après la définition 01, une instance de type Article est constituée en première position d'un élément de type Entête suivi d'une liste d'au moins une Section, suivi éventuellement d'Annexes.

  D'un point de vue théorique, un groupe ordonné est un produit cartésien de types ; un type t composé des types t_1, t_2, t_3 et défini comme un groupe ordonné s'écrit : $t = t_1 \times t_2 \times t_3$.

- **structure de groupe non ordonné**

  Le groupe non ordonné a la même signification que l'agrégat non ordonné (voir I.4.2.2). En SGML, un groupe de modèle est non ordonné si le caractère & est utilisé pour séparer ses composants. Par exemple la déclaration 02 de la figure Fig. 1.4 spécifie qu'une instance de type Entête se compose d'un Titre et d'une liste d'Auteurs, dans n'importe quel ordre.

  D'un point de vue théorique, le groupe non ordonné est identique au n−uplet du modèle relationnel des bases de données ; un type t composé des

types t_1, t_2, t_3 et défini comme un groupe non ordonné s'écrit : $t = [t_1, t_2, t_3]$.

- **unicité de composant**

  Lorsque le caractère '|' (barre verticale) est utilisé pour séparer les composants d'un groupe de modèle, ce groupe de modèle est un choix et un seul de ses composants peut apparaître dans toute instance du type ainsi défini. D'après la définition 08, une instance de type Paragraphe est constituée d'un seul élément dont le type appartient au groupe de modèle (#PCDATA | ParaRef | Programme | Figure).

I.5.3.3 Occurrences

Le nombre d'occurrences des instances d'un type est explicitement exprimé dans la définition au moyen de caractères spéciaux appelés indicateurs d'occurrence. Un indicateur d'occurrence, lorsqu'il existe, est placé après l'identificateur de type et avant le connecteur s'il est présent.

- **élément obligatoire et répétable**

  Un type suffixé par l'indicateur d'occurrence '+' (plus) est obligatoire et peut avoir plus d'une occurrence. D'après la déclaration 01, le type Article doit avoir au moins une section et peut en avoir plusieurs.

- **élément optionnel et répétable**

  L'indicateur d'occurrence '*' (astérisque) rend un type optionnel et répétable. Un tel type peut ne pas avoir d'occurrence, il peut aussi avoir une ou plusieurs occurrences. La déclaration 01 spécifie qu'une instance de type Article peut ne pas contenir d'Annexes, en contenir une ou plusieurs.

- **élément facultatif et non répétable**

  Le point d'interrogation '?' indique le caractère facultatif d'un type. Il indique que le type qu'il suffixe peut avoir au plus une instance. D'après la définition 04, une instance de type Auteur peut ne pas contenir de photo ou contenir au plus une seule photo.

- **élément obligatoire et unique**

  En l'absence de l'un des indicateurs d'occurrence (+, *, ?), le composant concerné est obligatoire et doit avoir une instance. Selon la définition 01 de la figure Fig. 1.4, le type Entête est obligatoire et une instance de type Article doit avoir un élément de type Entête.

I.5.3.4 Attributs

Les attributs sont globalement présentés en I.4.2.2. SGML permet d'associer aux types définis dans une DTD des attributs destinés à accompagner leurs instances. La déclaration de ces attributs précise l'ensemble des valeurs possibles qu'ils peuvent prendre et leurs valeurs par défaut. D'après la définition 17, on associe aux instances de type Article un

attribut nommé Édition qui peut prendre les valeurs Read ou Write, la valeur par défaut étant Read.

Les attributs servent également en SGML pour établir des liens de type référence (voir I.5.3.5) entre deux types de la structure générique.

I.5.3.5 Référence

Il est possible d'exprimer en SGML des relations non hiérarchiques de type référence entre éléments. Les éléments référencés appartiennent au même document que le renvoi ou à d'autres documents. Les instructions 12, 15 et 16 Fig. 1.4 sont indispensables pour référencer un paragraphe. La définition de la ligne 15 indique qu'un Paragraphe doit avoir l'attribut IdPara qui est un identificateur dont la valeur doit être unique dans le document. Cette unicité est signalée par le mot−clé ID. Exemple :

```
<Paragraphe  IdPara=P1>
La partie de l'équipe &OPERA; résidant à Grenoble
    va organiser une ...
</Paragraphe>
```

Ce paragraphe porte désormais l'identificateur unique P1, lequel pourra être utilisé pour le désigner. La référence est un élément vide, en l'occurrence RefPara (voir la définition 12 de la figure Fig. 1.4), qui a un attribut nommé Para (voir la définition 16 de la figure Fig. 1.4) lequel prend pour valeur la valeur d'un attribut identifiant un élément de façon unique. Dans l'exemple qui suit, la référence <ParaRef RefPara=P1> est un renvoi vers le paragraphe P1.

```
<Paragraphe  IdPara=P2>
Les demandes de participation à la manifestation
    présentée dans le paragraphe <RefPara Para=P1>
    doivent parvenir à ...
</Paragraphe>
```

I.5.4 Échange de documents

SGML est indépendant des contraintes matérielles. Tout système SGML(5) comporte un certain nombre de ressources, dont la *déclaration SGML*. L'indépendance vis à vis des contraintes matérielles s'exprime dans la déclaration SGML en identifiant le jeu de caractères utilisé dans le contenu du document, le rôle des délimiteurs, les classes de caractères(6) et la syntaxe concrète. Il en résulte une garantie de portabilité entre systèmes hétérogènes (par exemple systèmes ayant des jeux de caractères différents), sans perte

(5) Un système SGML est un environnement logiciel comportant un outil de production de documents SGML, un analyseur syntaxique de documents SGML et, éventuellement, des outils de traitement de documents (par exemple, un traducteur qui, à partir d'un document SGML, produit un document temporaire à l'intention d'un formateur).

(6) La syntaxe SGML définit 18 classes de caractères dont 8 constituent le groupe des caractères de fonction, 7 celui des caractères d'identification, la classe des caractères de données, la classe des caractères non SGML et la classe des délimiteurs.

d'information et sans mauvaise interprétation des caractères. Il suffit, pour réaliser cette portabilité, de préciser la correspondance entre les caractères utilisés par le document (*DESCSET)* et ceux du jeu de caractères du système destinataire.

L'échange de document lui−même se fait selon le format *SDIF* (SGML Document Interchange Format : ISO 9069) qui définit une syntaxe abstraite permettant de combiner dans le même flux de transmission un document principal(7) et ses documents associés(8) (voir la clause 6 de [Goldfarb90]).

I.6 DSSSL

Le langage SGML ne permet pas de spécifier proprement la présentation des documents. Ce rôle est dévolu à la norme DSSSL (Document Style Semantics and Specification Language) [I.S.O91a] dont le modèle de traitement est illustré par la figure Fig. 1.5. Cette norme, destinée avant tout au formatage, permet de spécifier la mise en page et les attributs typographiques des documents SGML. Elle offre également les ressources nécessaires à l'expression d'autres sémantiques de traitement. Ces objectifs sont atteints grâce à la normalisation des éléments suivants :

1. *Location models* : un langage d'interrogation qui permet de rechercher des éléments en navigant dans la structure arborescente des documents. Les requêtes de localisation s'expriment en fonction, non seulement des types et des attributs des éléments, mais aussi des relations qu'ils entretiennent entre eux.
2. *General Language Transformation Process* (GLTP) : un modèle et un langage qui servent à construire un document SGML à partir d'un autre document SGML. Le GLTP fournit des fonctions de transformation (suppression, duplication, réarrangement, regroupement, etc.) indépendantes des sémantiques de traitement.
3. Semantic−Specific Process (SSP) : un modèle et un langage d'expression des sémantiques de formatage.

---

(7) On appelle document principal le document qui fait l'objet de l'échange.

(8) On appelle document associé un document utilisé conjointement avec le document principal mais qui n'est pas référencé comme sous−document du document principal ou comme partie du document principal.

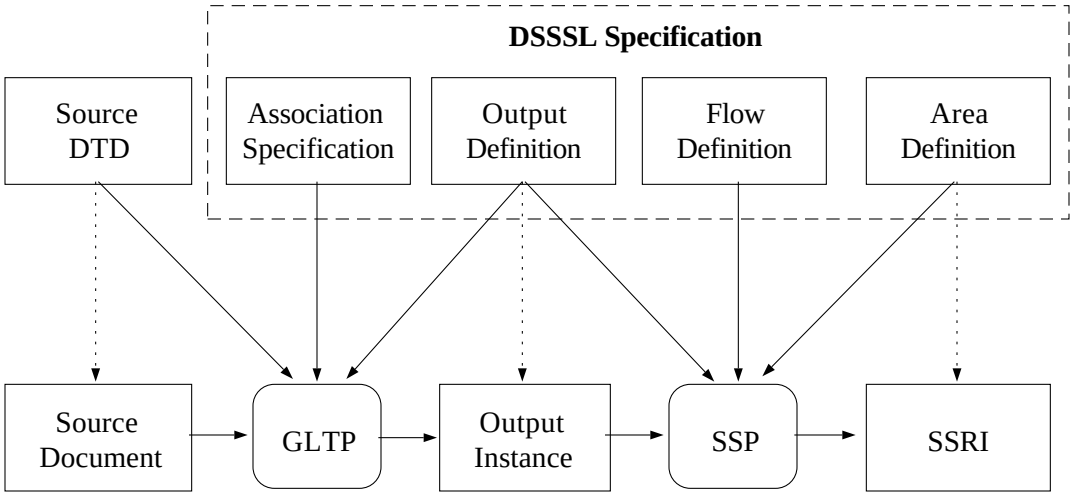


Fig. 1.5 : Modèle de traitement de DSSSL

Le résultat de la transformation du document source (Source Document) par le GLTP est un autre document appelé Output Instance. Le SSP applique les sémantiques de formatage normalisées au document issu du GLTP (Output Instance) et produit un document formaté, dénommé *Semantic−Specific Result Instance* (SSRI). L'action du GLTP et du SSP est guidée par les informations contenues dans l'ensemble de ressources (DSSSL Specification) que constituent *Association Specification*, *Output Definition*, *Flow Definition* et *Area Definition*, dont les fonctions sont brièvement exposées ci−dessous.

Output Definition Il s'agit d'une définition de structure hiérarchique des éléments virtuels résultant de l'action du GLTP. Par exemple dans le cas d'une transformation préalable du document source, le résultat de l'action du GLTP peut consister en la création d'éléments nouveaux qui n'ont pas de correspondance dans la DTD d'origine. De tels éléments doivent être définis dans la ressource Output Definition. Lorsque le résultat de l'action du GLTP (output Instance) est soumis au SSP (par exemple un formateur), la présence des éléments nouveaux doit être signalée. Dans ce cas, le contenu des éléments définis dans la ressource Output Definition est placé dans des boîtes instanciées par le formateur. L'intérêt de cette ressource est de permettre une éventuelle réorganisation du document source. Du point de vue de la syntaxe, les éléments de Output Definition sont définis comme dans une DTD selon ISO 8879 et leur sémantique est fonction du traitement spécifique (SSP).

Association Specification Cette ressource exprime une correspondance entre le contenu des éléments du document source et les éléments virtuels définis dans Output Definition.

Area Definition Il s'agit de la définition de zones rectangulaires que le formateur instancie sur le support d'impression ou d'affichage et qui servent à recevoir le contenu des éléments de Output Definition, par exemple page, colonne, haut de page, bas de page, etc. La définition d'une zone comporte son identificateur, ses dimensions, sa position, etc.

Flow Definition Les éléments de la ressource Output Definition sont mis en correspondance avec des éléments de Area Definition. L'ensemble de ces correspondances constitue la ressource Flow Definition.

I.7 ODA

Tout comme SGML, ODA (Open Document Architecture) a été définie pour faciliter l'échange de documents dont le contenu peut être de type texte, graphique, image. Les deux normes sont fondées sur la description de la structure logique des documents. ODA se distingue en prenant en compte le formatage et donne les moyens de l'exprimer. Le document échangé peut prendre l'une des trois formes suivantes :

- il est édité et non formaté ; le destinataire peut le rééditer et le formater.
- il est formaté et disponible uniquement pour l'impression et l'affichage.
- il est formaté mais il est disponible pour le formatage et l'édition.

Dans un document ODA, on distingue une structure logique spécifique et une structure physique spécifique. Cette norme a été conçue pour permettre l'expression d'une interprétation commune à l'expéditeur et au destinataire, des structures logique et physique des documents échangés. L'ensemble des règles de description de structure, de présentation, de mise en page et d'échange de documents porte le nom d'*architecture de document*.

I.7.1 Les structures de documents

Le concept de base de ce modèle est de décrire un document en terme de sa structure logique et de sa structure physique. On distingue les structures suivantes :

- Structure logique spécifique

  La structure logique spécifique identifie le contenu d'un document comme formé d'un certain nombre de composants logiques ; elle représente l'organisation effective du document.

- Structure physique spécifique

  La structure physique répartit le contenu d'un document en pages, ensemble de pages, colonnes, etc.

- Structure logique générique

  L'ensemble des objets de mêmes propriétés constitue une *classe d'objets* dont les caractéristiques sont contenues dans la *description de*

la classe d'objets correspondante. La description de l'ensemble des classes d'objets qui constituent un document, représente la *structure logique générique*. Le document étant lui même un objet, il appartient à une *classe de documents* à laquelle correspond une description de *classe de documents.*

- Structure physique générique

  Une structure physique générique est un ensemble de définitions de classes d'objets destinés à la présentation tels que les pages, les composants des pages, etc. Une définition précise les composants des objets et leurs relations.

Les structures physique et logique sont des ensembles hiérarchisés des éléments qu'elles représentent. Les propriétés de ces éléments et les relations qu'ils entretiennent les uns avec les autres sont exprimées sous forme de couples attribut−valeur. Il existe plusieurs types d'attributs tels que :

- les attributs d'identification d'objets qui permettent d'identifier de façon unique les objets auxquels ils s'appliquent. Si l'objet décrit est un objet physique, l'attribut précise sa nature. Dans ce cas les natures possibles sont :

  1. la racine du document,
  2. l'ensemble de pages,
  3. la page, composée de cadres et de blocs,
  4. le cadre qui peut être découpé en cadres et en blocs si besoin,
  5. le bloc qui est le composant de base de la structure physique spécifique.

  Si la description concerne un objet logique spécifique, l'attribut précise si l'objet est de l'une des natures suivantes :

  1. racine logique du document,
  2. objet logique composé (exemple Section),
  3. objet logique de base.
- les attributs d'expression des relations entre objets,
- les attributs de présentation,
- les attributs d'expression des propriétés (position, dimension),
- les attributs de formatage, etc.

La relation entre les objets des structures logique et physique est illustrée de façon générale par la figure Fig. 1.6. Cette figure est constituée de deux parties sous forme arborescente. L'arbre inférieur de racine ‘‘Ensemble de page’’ dont les feuilles sont des blocs, correspond à la structure physique spécifique d'un élément de document de type Section. À cet élément, correspond l'arbre supérieur qui représente sa structure logique spécifique dont les feuilles correspondent au contenu du document. Les deux arbres partagent les mêmes feuilles. L'objet logique Section est associé à l'objet physique Ensemble de pages. Il en résulte que si le contenu d'un paragraphe ne peut tenir dans une page, il est réparti sur

plusieurs pages ; ceci est illustré sur la figure Fig. 1.6 par la frontière de pages (trait en pointillé).

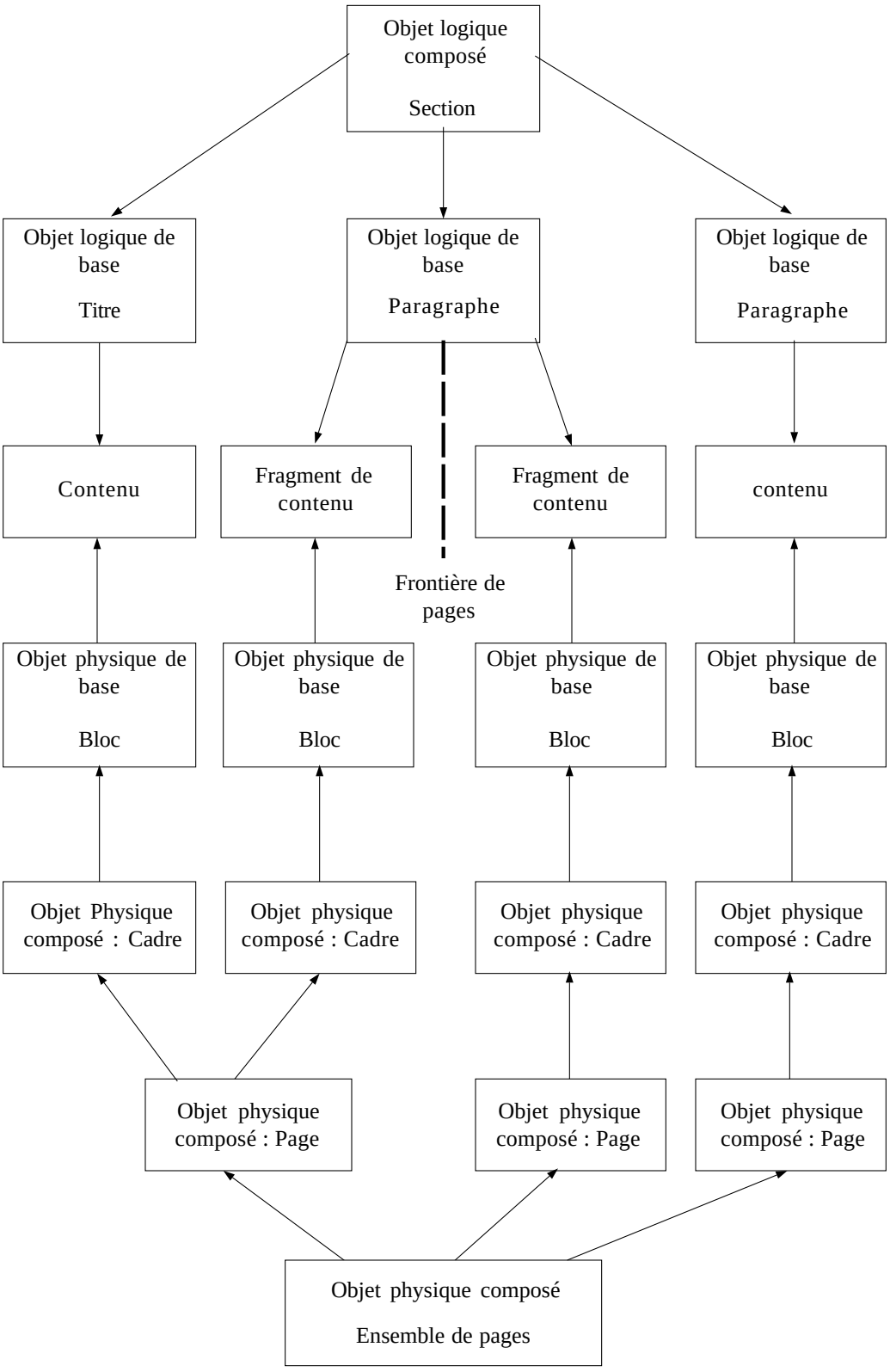


Fig. 1.6 : Exemple de relation entre objets spécifiques logiques et physiques

I.7.2 Architectures de contenu

Les éléments qui composent un document sont de natures diverses. Ils sont de type alphanumérique, graphique, image, etc. L'idée de base est d'associer un type aux divers contenus afin de leur appliquer si nécessaire des fonctions de contrôle et des traitements spécifiques. Les architectures de contenu sont des éléments de contenu auxquels correspondent des structures logiques et physiques particulières. Un contenu de document est exprimé au moyen des architectures de contenu. La spécification d'une architecture de contenu (voir [I.S.O89]) précise trois sortes d'information :

- information de structure : on précise ici le type de l'architecture de contenu, ainsi que les règles qui définissent sa structure interne, sa position et sa présentation.
- information de contenu : elle détermine l'information effective associée à l'architecture de contenu et les actions à lui appliquer.
- les valeurs possibles des attributs de présentation, leurs valeurs par défaut, et les paramètres des fonctions de contrôle.

I.7.3 SGML et ODA

Il n'y a pas de différence fondamentale entre ODA et SGML. La particularité de SGML est qu'il ne prend pas en compte la présentation des documents à la différence d'ODA. Cette séparation de la structure et de la présentation simplifie l'utilisation même du langage et facilite sa mise en œuvre, même si le problème de la présentation n'est pas encore entièrement résolu.

SGML se fonde sur une description abstraite de sa syntaxe, laissant la liberté à l'utilisateur de préciser une syntaxe concrète définissant ainsi du moins partiellement son propre vocabulaire. Cette possibilité n'est pas offerte pour ODA et rend à priori SGML d'un abord plus facile. À titre d'exemple, les attributs de ODA sont imposés par la norme et servent aussi bien à la structure logique qu'à la présentation introduisant un niveau supplémentaire de complexité dans l'écriture des règles de structure en plus de l'apprentissage d'un vocabulaire obligatoire.

En raison de la diversité de la nature des éléments qui composent un document, SGML prend en compte des données autres que textuelles (données spécifiques, tables, formules mathématiques, etc.). Il n'en est pas ainsi dans ODA qui prévoit l'utilisation des architectures de contenu pour résoudre les problèmes posés par l'intégration de données non textuelles. Cette différence met en évidence la difficulté d'implantation d'un système ODA. À ce jour, il n'existe pas de système complet dans lequel le modèle de traitement de document stipulé par ODA soit complètement implanté, malgré les projets Hérode et PODA [Robinson87] lancés dès la publication de la norme.

I.8 Hypertexte et hypermédia

Les hypertextes sont des documents électroniques, subdivisés en unités (documents complets ou fragments de documents) accessibles grâce à des liens de type référence [Brown88] identifiés comme suit :

- Des liens hiérarchiques organisés en fonction de leur sémantique et de la granularité des fragments considérés.
- Des liens plus généraux portant sur des fragments du même document ou de documents différents.

D'une façon générale, un hypertexte est un document organisé comme un graphe orienté dont chaque nœud est un fragment et dont les arêtes sont des liens électroniques qui permettent la navigation et la consultation des informations contenues dans les nœuds.

Le concept de document structuré et la norme SGML sont aujourd'hui pris en considération dans la plupart des systèmes d'hypertextes tel que Guide [Brown89]. C'est tout naturellement que les éditeurs de documents structurés [André89], notamment Grif [Quint92] [Quint86a] implantent des liens fondés en partie sur la structure dont ils tirent avantage [Scotts91].

L'hypermédia est l'union de deux technologies de traitement de l'information : le multimédia et l'hypertexte. Ce terme désigne les systèmes qui permettent de naviguer entre des documents quel que soit le type de leur contenu (texte, graphique, son, vidéo, etc.). De nombreux systèmes hypermédia sont présentés dans l'article de Conklin [Conklin87] et dans le livre de Nielsen [Nielsen90]. Parmi ces systèmes, Intermedia [Han92] se distingue par son aspect multi−utilisateur et par la disponibilité au niveau système de la gestion des liens.

D'un point de vue global, les systèmes d'hypermédias garantissent la synchronisation temporelle des données de type vidéo [Kahn91] et généralement du son et des images animées [Palaniappan90]. Une solution plus générale est proposée par HyTime [I.S.O91b], un langage destiné à faire une représentation structurée de fragments d'hypermédias. Basé sur la norme SGML, HyTime a pour objectif de permettre l'adressage et la synchronisation d'éléments de documents de type hypermédia, et ceci quelles que soient les applications qui ont servi à les produire.

La plupart de ces systèmes sont encore des prototypes et il reste encore beaucoup à faire dans ce domaine.

I.9 Édition coopérative

L'édition coopérative est la coopération de plusieurs auteurs à la rédaction du même document. Cet objectif est atteint de deux façons différentes. Certains formateurs permettent de faire une description statique d'un document comme étant un ensemble de modules dont l'édition est confiée à des auteurs différents. Une telle édition est statique dans la mesure où la description des modules est figée et que plusieurs auteurs ne peuvent pas éditer simultanément le même module. Les divers modules sont ensuite regroupés pour composer le document final [Lamport85]. La coopération dynamique, elle, consiste à autoriser l'édition

d'un même document simultanément par plusieurs auteurs. Dans l'édition coopérative dynamique, le problème se pose de déterminer les parties du document que chaque auteur pourra éditer librement. Pour certains auteurs, les rédacteurs peuvent éditer tout le document sans qu'il soit nécessaire de faire des attributions individuelles de parties. Une telle approche est basée sur l'exclusion mutuelle des rédacteurs. Un autre point de vue est le partage du document en parties attribuées avec des droits et actions bien spécifiés pour chaque groupe d'auteurs. L'un des problèmes à résoudre est la détermination de la taille des parties à attribuer aux auteurs : c'est le problème de la granularité. Les solutions proposées dans l'ensemble ne tiennent pas compte de la structure des documents et sont basées sur la délimitation de fragments de texte [Leland88] [Greif92] avec plus ou moins de flexibilité dans leur détermination [MACE91].

À l'opposé, l'édition coopérative structurée tire avantage de la structure logique pour régler le problème de la granularité. N'importe quel composant d'un document structuré peut être désigné comme fragment à attribuer à un groupe de rédacteurs. Sont concernés par cette approche les systèmes d'édition de documents structurés, notamment les hypermédias [Rein91] [Neuwirth90].

Les applications d'édition coopérative assurent l'édition des documents et offrent des services pour la distribution des parties et la coopération entre les différents acteurs, notamment :

- La gestion des rôles des utilisateurs : un lecteur est autorisé à consulter le document, un rédacteur à le modifier et un gestionnaire a la responsabilité de distribuer les rôles.
- La distribution des fragments de documents attribués aux utilisateurs.
- La reconstitution du document final par la collection des parties préalablement distribuées.

Griffon est une application de ce type en cours de développement par l'équipe du projet OPERA au laboratoire Bull−IMAG de Grenoble. Elle s'appuie d'une part sur Grif, d'autre part sur les fonctionnalités de distribution et de coopération offertes par le système Guide [Decouchant88].

I.10 Conclusion

Dans ce chapitre, nous avons fait en premier lieu une présentation informelle des notions de structure logique et de structure physique des documents. L'exposé qui est ensuite fait sur les caractéristiques des éditeurs de documents plus ou moins fondés sur la structure, permet de se rendre compte de l'intérêt que présente la structuration dont les vertus ne sont plus contestées. En effet l'intégration croissante de la structure dans les éditeurs de documents, le succès des éditeurs syntaxiques, la commercialisation d'éditeurs de documents structurés, l'émergence de normes fondées sur la structure et la prise en compte de la structure par des applications sont la preuve de l'importance grandissante de l'édition structurée. Un facteur supplémentaire est la convergence de la plupart des SEDS vers le langage SGML, adoptant ainsi une abstraction bénéfiques aux applications externes.

Cependant beaucoup de problèmes restent à régler, notamment ceux liés à l'évolution des structures et aux restructurations dynamiques, qui font l'objet de cette thèse.

Chapitre II

Grif : un système d'édition de documents structurés

Résumé :

Le présent chapitre a pour but de présenter les documents structurés tels qu'ils sont considérés dans le système Grif. Cette présentation porte sur les notions de classe de documents et de structure, la définition des types, les attributs et les références.

II.1 Présentation générale de Grif

Grif est un système d'édition de documents structurés développé dans le cadre du projet OPÉRA (Outils Pour les documents Électroniques, Recherche et Applications). Il est destiné à la production de documents structurés techniques, scientifiques ou généraux. Les documents produits par Grif peuvent eux−mêmes contenir des objets structurés tels que des tableaux, des formules mathématiques, des graphiques ou des programmes. L'approche retenue dans Grif est celle exposée dans le chapitre I, selon laquelle les éléments constitutifs d'un document sont décrits par des règles de grammaire qui, comme dans les éditeurs syntaxiques, guident l'édition. Comme tous les éditeurs de documents structurés, Grif s'appuie essentiellement sur la structure logique du document.

Comme exposé dans le chapitre précédent, les règles qui décrivent les constituants logiques d'un document et les relations qu'ils entretiennent les uns avec les autres, forment une structure logique générique appelée *schéma de structure* dans Grif (un exemple est donné en Fig. 2.1).

```
STRUCTURE Article;
DEFPRES ArticleP;

ATTR
    Importance = Définition, MotImportant;
    Programmation = Identificateur,MotClé;
STRUCT
Article (ATTR NuméroPremPage = Integer) =
      BEGIN
          ?DateDeMiseAJour =  TEXTE;
          Auteurs = LIST OF (Auteur);
          TitreArticle = TEXTE;
          CorpsArticle = LIST OF (Section);
      END;

Auteur(ATTR TypeAuteur = Principal, Secondaire)
       = TEXTE;

Section = BEGIN
              TitreSection = TEXTE;
              CorpsSection = LIST OF (Contenu);
          END;
Contenu =
        (CASE OF
             TEXTE;
             IMAGE;
             GRAPHIQUE;
            RefBiblio  =  REFERENCE(Biblio(RefBib));
            RefFigure  =  REFERENCE(Figure);
            RefNote  =  REFERENCE(Note);
            RefTout  =  REFERENCE(ANY);
          END;
         );

ASSOC
Figure  = BEGIN
             Illustration = CASE OF
                              Dessin  =  GRAPHIQUE;
                              Autre  =  NATURE;
                            END;
              TitreFigure = TEXTE;
          END;
Note = LIST OF (Contenu);
```

Fig. 2.1 : Extrait du schéma de structure Article

Les schémas de structure sont écrits dans un langage appelé S et Grif est un système paramétrable qui offre à l'utilisateur la possibilité de définir ses propres schémas en fonction de ses besoins.

Dans le système Grif, il y a une séparation nette entre la structure et la présentation. On peut associer à chaque type défini dans une structure logique générique un ensemble de

règles de présentation qui indiquent ses attributs typographiques et ses contraintes de mise en page. Un ensemble de règles de présentation correspondant à une structure logique générique est appelé *schéma de présentation*. Comme pour le schéma de structure, Grif permet à l'utilisateur d'écrire ses propres schémas de présentation au moyen d'un langage appelé P [Quint86b].

Les documents produits par Grif sont sauvegardés sous un format propre appelé *forme pivot* qui ne convient pas forcément à d'autres systèmes ou applications. Aussi, pour permettre les échanges avec d'autres systèmes, produit−il également des sorties conformes aux formalismes les plus répandus, tels que SGML, T_EX, LAT_EX, PostScript, Ascii. Grif dispose à cette fin du langage T qui, étant donné un élément de structure générique, spécifie comment transcrire ses instances dans un autre formalisme. Cette spécification porte le nom de *règle de traduction*. L'ensemble des règles de traduction d'une structure logique générique forme un *schéma de traduction.*

Le système d'édition de documents structurés Grif offre le mécanisme de référence qui permet d'établir des liens non hiérarchiques entre deux parties d'un même document (référence interne) ou de deux documents différents (référence externe). Non seulement les références servent à désigner des éléments de la structure logique tels que les sections, les sous−sections, etc., mais encore sont−elles indispensables pour établir des liens avec des éléments flottants tels que les tables, les notes de bas de page, les figures, etc.

Comme dans la plupart des SEDS, les documents Grif sont représentés sous forme d'arbres et tous les traitements d'édition sont faits en manipulant cette représentation. Il existe plusieurs sortes d'arbres dans Grif, un arbre principal et des arbres associés. L'arbre principal correspond à l'instance du corps du document. Le type de sa racine est la première définition après le mot−clé STRUCT du schéma de structure dont il porte le nom.

Les arbres associés correspondent à des instances dont les types sont définis comme tels dans le schéma de structure. Ils n'entretiennent pas de relation hiérarchique avec le type principal ou ses composants. Un arbre associé n'appartient donc pas à l'arbre principal bien que ses instances appartiennent au document. De tels arbres représentent des éléments tels que les figures, les notes, les formules mathématiques, etc. À chaque sorte d'élément associé correspond une arborescence distincte. De ce fait, un document est une forêt d'arbres reliés par des liens de type référence. Le compilateur S ajoute dans le schéma de structure une définition de type pour l'arborescence de chaque type associé. Le type correspondant à une telle arborescence est défini comme une liste d'un type associé et a pour identificateur celui du type associé suffixé, par le caractère 's' ; ainsi les définitions suivantes correspondent respectivement aux arborescences des types associés Note et Figure :

```
Notes = LIST OF(Note);
Figures = LIST OF(Figure);
```

La figure Fig. 2.2 représente l'arbre principal d'une instance de document du schéma Article dont la définition est donnée en Fig. 2.1. Pour des raisons de commodité, l'arbre correspondant au corps de l'article est donné en Fig. 2.3 et en Fig. 2.4. Chaque nœud ombré de la figure Fig. 2.3 contient une référence (représentée par (1) et (2)) aux éléments associés

de type Note de la figure Fig. 2.5. En revanche le nœud ombré de la figure Fig. 2.4 contient une référence (représentée par Fig. 2.6) à l'élément associé de type Figure de la figure Fig. 2.6. Il faut noter que les numéros de référence des éléments associés ne font pas partie des instances mais sont des éléments de présentation générés par le formateur à partir du schéma de présentation.

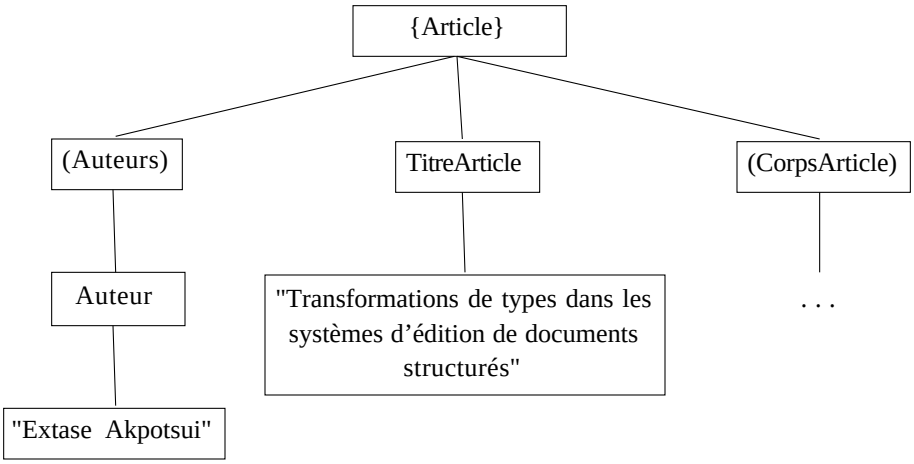


Fig. 2.2 : Arbre principal partiel d'une instance de type Article

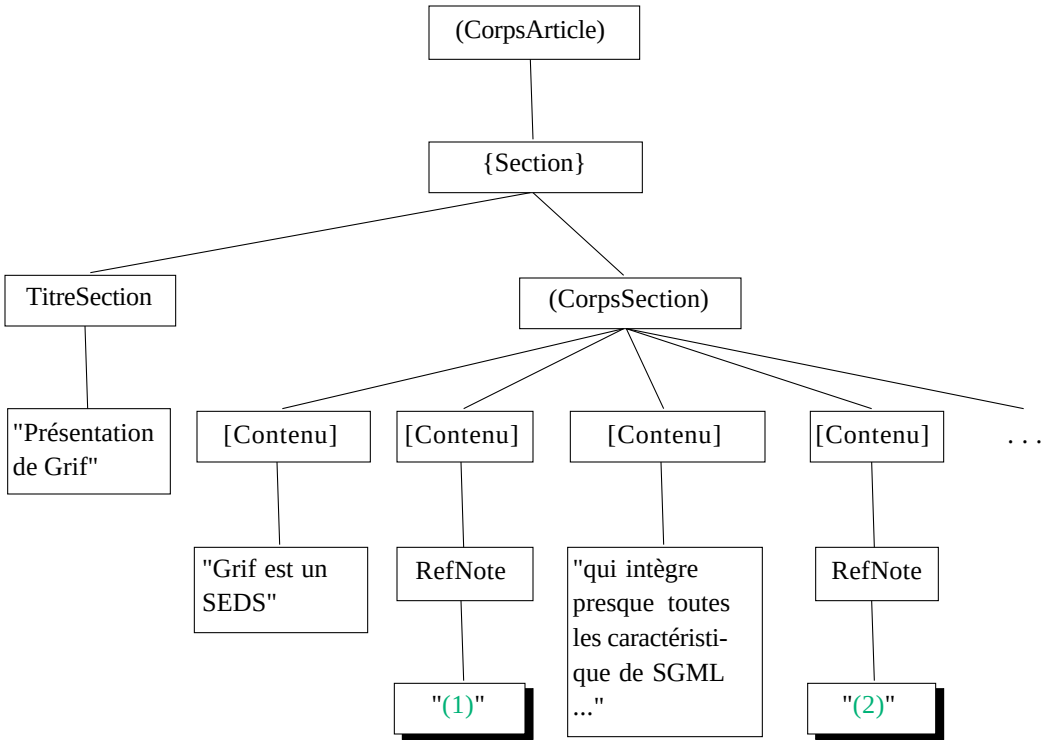


Fig. 2.3 : Arbre partiel d'une instance de type CorpsArticle

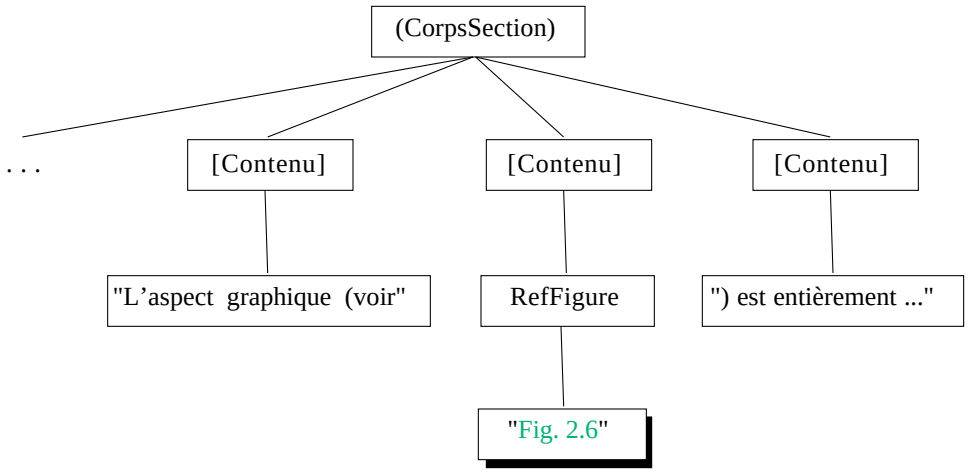


Fig. 2.4 : Arbre partiel d'une instance de type CorpsSection

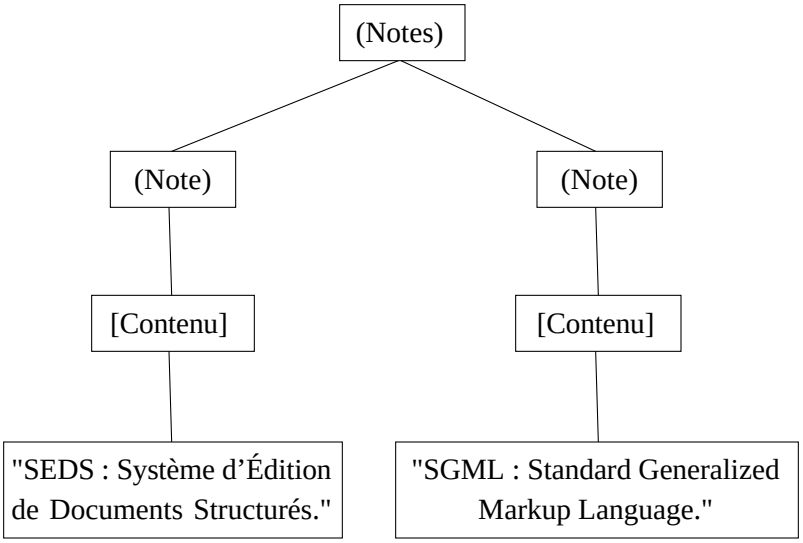


Fig. 2.5 : Arborescence des éléments associés de type Note

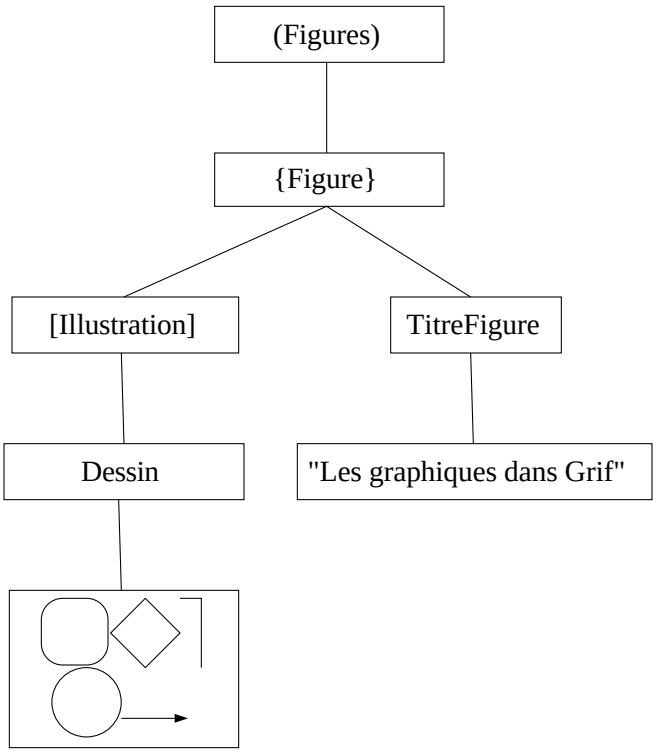


Fig. 2.6 : Arborescence des éléments associés de type Figure

En plus de l'édition et de la création, Grif dispose d'un certain nombre de services sous forme d'applications qui contribuent à la production de documents, notamment un correcteur orthographique [Richy92], un gestionnaire d'index et un outil d'annotation. Les travaux en cours portent sur les transformations de structure, la correction typographique, les aspects multilingues, l'édition coopérative et le formatage.

Ce chapitre se limite à la présentation des caractéristiques de Grif qui, directement ou indirectement, concernent les problèmes induits par l'évolution de structures et des types.

II.2 Éléments de structure

Un schéma de structure se compose de plusieurs parties appelées sections, chacune précédée par un mot−clé tel que ATTR, STRUCT, ASSOC, etc., (voir Fig. 2.1). On parle de section ASSOC, etc. Selon la section à laquelle ils appartiennent, les types présentent des caractéristiques particulières qui seront précisées à l'occasion. Le schéma de structure sert à définir la composition logique d'un document en termes du type de ses composants, des relations que ces composants entretiennent les uns avec les autres et des attributs qui peuvent accompagner les éléments du document. Les types définis dans un schéma de structure sont appelés *types construits* par opposition aux *types de base* qui, eux, ne sont pas définis dans les schémas de structure. Le langage S développé pour écrire des schémas de structure a les mêmes caractéristiques que SGML. Comme SGML et de nombreux langages de programmation, il comporte des types de base, des constructeurs nécessaires à la définition des types construits et des indicateurs d'occurrence. Cette section a pour objectif de montrer

comment les types sont définis en S. Le lecteur habitué à SGML peut facilement comprendre l'exposé, la différence entre les deux langages étant essentiellement d'ordre syntaxique.

Deux règles fondamentales gouvernent la définition des types et des schémas de structure dans le système Grif :

1. Dans chaque schéma de structure, tous les types doivent porter des identificateurs différents.
2. De même que pour les types, l'identificateur d'un schéma de structure est unique dans le système Grif.

II.2.1 Types de base

Comme dans les langages de programmation, les types de base sont fournis par le système et servent à définir des types construits. Grif offre les types de base suivants :

Texte — Le type Texte sert à construire des chaînes de caractères, vides ou non, et est représenté par le mot−clé TEXTE.

Graphique — Ce type désigne les éléments graphiques de base du système tels que les traits, les cercles, les carrés, les ellipses, dont la composition permet de construire des objets graphiques complexes. Il est représenté par le mot−clé GRAPHIQUE.

Symbole — L'ensemble des caractères spéciaux du système forme le type de base Symbole et est représenté par le mot−clé SYMBOLE. Par exemple, les caractères qui servent dans les notations mathématiques tels que le signe d'intégration, le Σ, etc., dont les dimensions dépendent du contexte, sont de type Symbole.

Image — Ce type de base représenté par le mot−clé IMAGE désigne les matrices de points (bit maps).

Référence — Le type Référence sert à construire les références. Il est représenté par le mot−clé REFERENCE.

II.2.2 Types construits

Les types qui participent à la structure logique du document, notamment le type qui définit la racine de l'arbre principal, sont déclarés après le mot−clé STRUCT. Un type construit est défini au moyen d'un constructeur et d'un certain nombre de types susceptibles de participer à la production de ses instances. La sémantique associée à un type construit est fonction de son constructeur, lequel précise le rôle des composants des instances et éventuellement les relations qu'ils entretiennent. Un constructeur est un ensemble d'unités syntaxiques qui précise pour un type donné, ses composants, le nombre de ces derniers et la relation d'ordre qui les unit. Les constructeurs du système, énumérés ci−dessous, sont présentés individuellement dans cette même section :

- Identité,
- Agrégat ordonné,

- Agrégat sans ordre,
- Liste,
- Choix,
- Nature,
- Unité.
- Référence,
- Paire.

Pour chaque constructeur, un exemple de définition de type est donné et la représentation arborescente d'une instance de ce type expliquée. Une telle arborescence est appelée *arbre d'instance*. D'une manière générale, la définition d'un type respecte la syntaxe suivante :

IdentificateurDeType = Construction;

Le caractère '=' est un méta−caractère qui confère à la définition la signification suivante : le symbole à gauche du méta−caractère, en l'occurrence IdentificateurDeType, est l'identificateur du type dont la définition, en l'occurrence Construction, est donnée à sa droite. Ce caractère joue le même rôle que le caractère BNF '::='. Le terme Construction est une protophrase qui contient un ensemble, éventuellement vide, de symboles permettant d'identifier le constructeur.

Identité

Lorsque la protophrase à droite du méta−caractère '=' est un type de base ou un type construit, le constructeur du type ainsi défini est Identité. Dans ce cas, il n'y a pas d'unités syntaxiques qui expriment le constructeur, puisque la partie droite de la définition se résume à un identificateur. Ce constructeur permet de spécifier qu'un type a la même définition qu'un type déjà existant, que ce dernier soit un type de base ou un type construit. Il est important de remarquer qu'il n'implique pas que les types situés de part et d'autre du caractère '=' sont identiques. L'exemple suivant indique que le type TitreArticle a la même définition que le type TEXTE.

```
TitreArticle = TEXTE;
```

La racine de l'arbre d'instance du type TitreArticle (voir Fig. 2.7) est identifiée par le nom TitreArticle et elle a pour fils unique un contenu de type Texte.

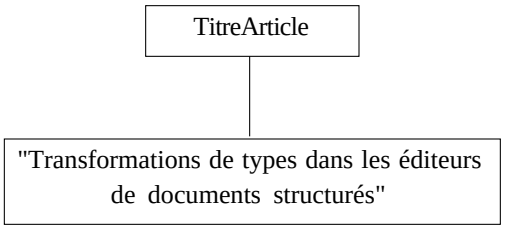


Fig. 2.7 : Arbre d'instance du type TitreArticle

En supposant que le type construit SuiteSections soit déjà défini, la définition suivante indique que le type CorpsArticle a la même définition que le type SuiteSections :

```
CorpsArticle = SuiteSections;
```

Il est important de noter qu'il ne s'agit que d'un raccourci d'écriture permettant de ne pas répéter une définition déjà écrite. Pour Grif, il s'agit de deux types différents, chacun ayant un identificateur unique. Il en découle que même si deux types sont définis comme *ayant la même définition*, leurs instances, elles, sont considérées comme de types différents. Aussi une instance de type SuiteSections ne peut−elle se substituer à une instance de type CorpsArticle. L'arbre d'une instance de type CorpsArticle est partout construit de la même façon que l'arbre d'une instance de type SuiteSections, exceptée sa racine qui a une étiquette différente, en l'occurrence, (CorpsArticle).

Grif fait une utilisation ambiguë du méta−caractère '=' interprété tantôt comme un méta−caractère tantôt comme un constructeur et un méta−caractère. Lorsque la partie droite d'une définition est un type construit, Grif attribue le constructeur Identité au type ainsi défini mais le traite comme s'il avait le même constructeur que le type qui a servi à le définir. En conséquence, l'étiquette de la racine de l'arbre est formée de l'identificateur de son type ainsi que des caractères spéciaux hérités du type qui a servi à le définir. Par exemple si SuiteSections avait pour constructeur Liste, la racine de l'arbre du type CorpsArticle aurait pour étiquette (CorpsArticle).

Agrégat ordonné

L'agrégat ordonné permet de définir un type construit comme constitué d'un ensemble ordonné de types appelés composants. Cet ensemble est délimité par les mots−clés **BEGIN** et **END** et ses éléments sont frères entre eux. Ils sont aussi les fils du type qu'ils contribuent à définir. L'ordre de cet ensemble est déterminé syntaxiquement par la position de ses éléments dans la définition. Dans une instance dont le type a pour constructeur Agrégat ordonné, les éléments doivent avoir le même ordre que le type de leurs composants dans la définition.

Selon la définition suivante, une instance de type Article est un arbre comme celui de la figure Fig. 2.8 dont la racine est identifiée par le nom Article et dont les fils correspondent dans l'ordre aux arbres d'instances des types DateDeMiseAJour, Auteurs, TitreArticle et CorpsArticle.

```
Article = BEGIN
              ?DateDeMiseAJour =   TEXTE;
              Auteurs = LIST OF (Auteur);
              TitreArticle = TEXTE;
              CorpsArticle = LIST OF (Section);
          END;
```

L'indicateur d'optionnalité est représenté par le caractère '?' (point d'interrogation) et ne peut être placé qu'en tête d'un composant d'agrégat, ordonné ou non. Il traduit le fait qu'une instance de ce composant n'est pas obligatoire et qu'elle ne peut

apparaître qu'une seule fois. Un composant préfixé par le caractère '?' est dit optionnel, comme par exemple le type DateDeMiseAJour.

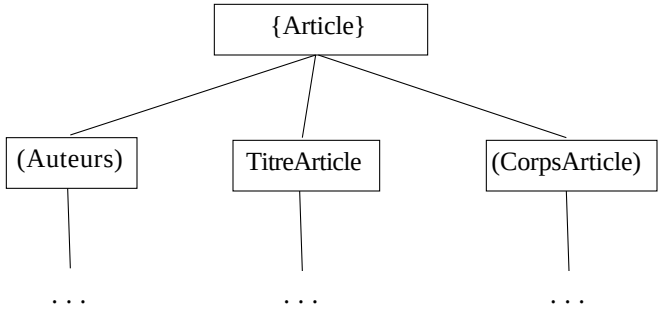


Fig. 2.8 : Représentation arborescente de l'agrégat Article

Agrégat sans ordre

Le constructeur Agrégat sans ordre permet de définir un type dont les composants sont compris entre les mots−clés **AGGREGATE** et **END**. Ces composants ne sont pas ordonnés et, dans une instance dont le type a pour constructeur Agrégat sans ordre, un élément correspondant à un composant n'a pas de position prédéterminée par rapport aux éléments correspondant aux autres composants. Les composants d'un type de constructeur Agrégat sans ordre peuvent aussi être optionnels. L'arbre d'instance a les mêmes caractéristiques que celui d'un agrégat ordonné à l'ordre des sous−arbres fils près. L'étiquette de la racine de l'arbre est formée de l'identificateur du type encadré par les caractères '↑' et '↓'.

Liste

Le constructeur Liste permet de définir un type dont les instances comportent un certain nombre d'éléments de même type. Un type de constructeur Liste est défini avec les mots−clés **LIST** et **OF** suivi d'un identificateur ou d'une définition de type entre parenthèses. L'élément placé entre parenthèses est l'*élément de liste*. Dans la définition d'une liste, le nombre d'éléments n'est pas obligatoirement limité. L'exemple suivant décrit CorpsSection comme une liste d'éléments de type Contenu.

```
CorpsSection = LIST OF (Contenu);
```

Les mots−clés LIST et OF peuvent être séparés par une protophrase indiquant le nombre minimum et maximum d'éléments, comme dans l'exemple suivant :

```
CorpsArticle = LIST [2 . . *] OF (Section);
```

Cette unité syntaxique représente l'indicateur d'occurrences du constructeur Liste. Dans cet exemple, une instance de type CorpsArticle doit contenir au moins deux sections. L'astérisque indique que le nombre maximum de sections n'est pas limité. Ce nombre doit être supérieur ou égal au nombre minimum. La racine de l'arbre d'instance du type CorpsArticle (voir Fig. 2.9) est étiquetée par l'identificateur CorpsArticle placé entre parenthèses et elle a au moins deux fils qui sont des arbres d'instances de type Section.

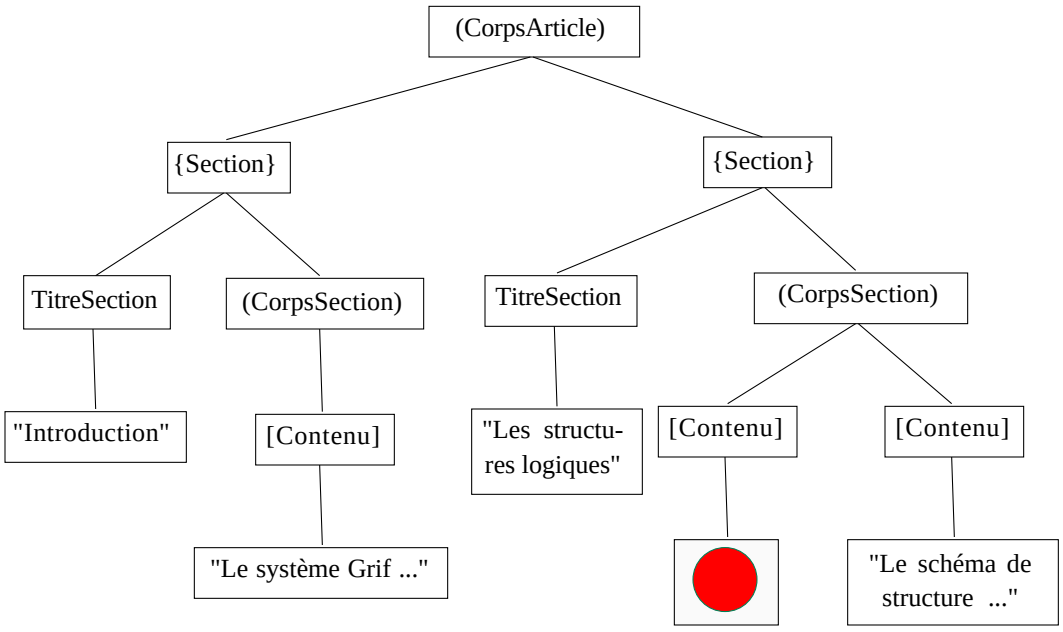


Fig. 2.9 : Arbre d'instance de type CorpsArticle

Choix

Un type de constructeur choix est défini comme un ensemble de types appelés options dont un seul peut être utilisé pour la construction de son instance. La définition des options commence après les mots−clés ***CASE OF*** et finit avec le mot−clé **END**. Par exemple, une seule des options TEXTE, IMAGE, GRAPHIQUE, RefBiblio, RefFigure, RefNote et RefTout participera à la création d'une instance du type Contenu défini ci−dessous :

```
Contenu =
        (CASE  OF
              TEXTE;
              IMAGE;
              GRAPHIQUE;
             RefBiblio   =   REFERENCE(Biblio(RefBib));
             RefFigure   =   REFERENCE(Figure);
             RefNote   =   REFERENCE(Note);
             RefTout   =   REFERENCE(ANY);
          END;
        );
```

L'arbre d'instance du type Contenu est identifié par sa racine de nom Contenu placé entre crochets et par son unique fils qui est un arbre d'instance de l'une de ses options.

Nature

Le constructeur Nature (mot−clé NATURE) désigne l'un quelconque des schémas de structure du système. Comme pour le choix, la Nature désigne un ensemble d'options dont une seule participe à l'instance du type qu'elles contribuent à définir. L'arbre d'instance correspondant a les mêmes caractéristiques que celui d'un

choix. Par exemple, l'illustration (voir la définition ci−dessous) d'une instance de type Figure est, soit un graphique, soit du type de n'importe quel schéma de structure.

```
Illustration = CASE OF
                    Dessin  =  GRAPHIQUE;
                    Autre  =  NATURE;
               END;
```

Référence

Un type de constructeur Référence permet de créer des éléments en vue d'établir des liens non hiérarchiques avec d'autres éléments dans le même document ou dans un autre document. Le type de l'élément référencé peut appartenir ou non au même schéma de structure que celui de la référence. Il est placé après la parenthèse ouvrante qui suit le mot−clé REFERENCE et est éventuellement suivi du nom entre parenthèses de son schéma de structure et se termine par une parenthèse fermante. Dans l'exemple suivant, le type RefBiblio permet de créer des renvois vers des éléments de type Biblio défini dans le schéma RefBib :

```
RefBiblio  =  REFERENCE(Biblio(RefBib));
```

Le type de l'élément référencé peut être remplacé par le mot−clé ANY, lequel désigne n'importe quel type, comme dans l'exemple qui suit :

```
RefTout  =  REFERENCE(ANY);
```

Les autres sortes de référence du système Grif, jugées non pertinentes pour le problème de transformation de structures, ne sont pas présentées ici. Le système Grif implémente les liens de type hypertexte [Quint92] qui permettent non seulement de référencer des éléments appartenant à d'autres documents mais aussi de composer un document par référence à d'autres documents.

Paire

Le constructeur paire, représenté par le mot−clé PAIR, sert à définir un type dont les instances sont toujours créées par paire et sont utilisées pour délimiter des fragments de contenu. Par exemple, un passage indexé est marqué par deux éléments de constructeur Paire dont le type est déclaré comme suit :

```
Marque_d_Index  =  PAIR;
```

II.3 Éléments associés

Dans un schéma de structure, les types associés sont définis après le mot−clé ASSOC. Leurs instances sont accessibles uniquement grâce à des références. L'exemple suivant extrait de la figure Fig. 2.1, définit les types Figure et Note comme types associés :

```
    RefFigure  =  REFERENCE(Figure);
    RefNote  =  REFERENCE(Note);
ASSOC
     Figure  = BEGIN
               Illustration  =  CASE  OF
```

```
                                    Dessin  =  GRAPHIQUE;
                                    Autre  =  NATURE;
                                   END;
                TitreFigure = TEXTE;
               END;
Note = LIST  OF  (Contenu);
```

La seule différence entre les éléments de structure et les éléments associés est que les premiers appartiennent à l'arbre principal du document et les seconds à leur propre arborescence. Il existe une arborescence par type associé tel que les instances engendrées par un même type associé appartiennent à une même arborescence. Dans l'exemple choisi, il y a une arborescence des instances de type Figure et une arborescence distincte des instances de type Note. Les références de type RefNote servent à créer des liens vers des instances de type Note et les références de type RefFigure servent à créer des liens vers des instances de type Figure.

L'ensemble de ces arborescences fait d'un document une forêt d'arbres que le processus de conversion d'instances, suite à une évolution de types, doit prendre en compte.

II.4 Exclusion et inclusion

L'exclusion et l'inclusion sont des mécanismes qui assouplissent la rigueur induite par la structuration. L'inclusion autorise dans certaines conditions, l'insertion d'éléments qui ne font pas partie de la structure logique et l'exclusion empêche la création d'instances qui normalement pourraient exister. Ce sont les mêmes notions que dans SGML (voir I.5.3.1).

Inclusion

Le mécanisme d'inclusion permet d'exprimer, dans la définition d'un type, que l'on peut insérer dans son arbre d'instance des éléments à position structurale quelconque et sans contrainte sur le nombre d'occurrences. Toutefois, le type inclus ne doit pas être un type qui apparaît au premier niveau de la définition. Par exemple on ne doit pas exprimer l'inclusion des types DateDeMiseAJour, Auteurs, TitreArticle et CorpsArticle dans la définition du type Article. Les inclusions sont signalées par le caractère '+' suivi, entre parenthèses, d'une liste d'identificateurs de type séparés par des virgules.

Dans l'exemple suivant, on autorise d'inclure des éléments de type SYMBOLE partout dans l'arborescence des instances de type Article :

```
Article = BEGIN
            ?DateDeMiseAJour =  TEXTE;
            Auteurs =  LIST  OF  (Auteur);
            TitreArticle  =  TEXTE;
            CorpsArticle  =  LIST  OF  (Section);
          END + (SYMBOLE);
```

Exclusion

L'exclusion est le mécanisme inverse de l'inclusion. Il interdit de produire une instance qui normalement pourrait apparaître dans l'arbre d'instance du type dont la

définition contient l'exclusion. Ne peuvent être exclus que des types non obligatoires, des éléments de liste, et des options de choix. Dans ce cas, les exclusions expriment plutôt une utilisation restrictive des types par l'éditeur, de telle sorte que les instances correspondant à ces types aient une structure spécifique valide même en leur absence (exclusions).

Les exclusions sont signalées par le caractère '−' suivi, entre parenthèses, d'une liste d'identificateurs de type séparés par des virgules.

D'après la définition suivante du type CorpsArticle, le corps d'un article ne peut pas contenir d'éléments de type IMAGE, comme l'illustre la figure Fig. 2.10 :

```
CorpsArticle = LIST OF (Section) − (IMAGE);
```

La définition du type Contenu de constructeur Choix autorise en principe la présence d'éléments de type IMAGE dans les instances de type CorpsArticle (la feuille de forme circulaire de la figure Fig. 2.9 représente une image). Les instances correspondant à cette nouvelle définition de CorpsArticle restent valides et auraient pu être obtenues même en l'absence de l'exclusion.

Lorsque la définition d'un type comporte simultanément une inclusion et une exclusion portant sur le même élément, la priorité est accordée à l'exclusion.

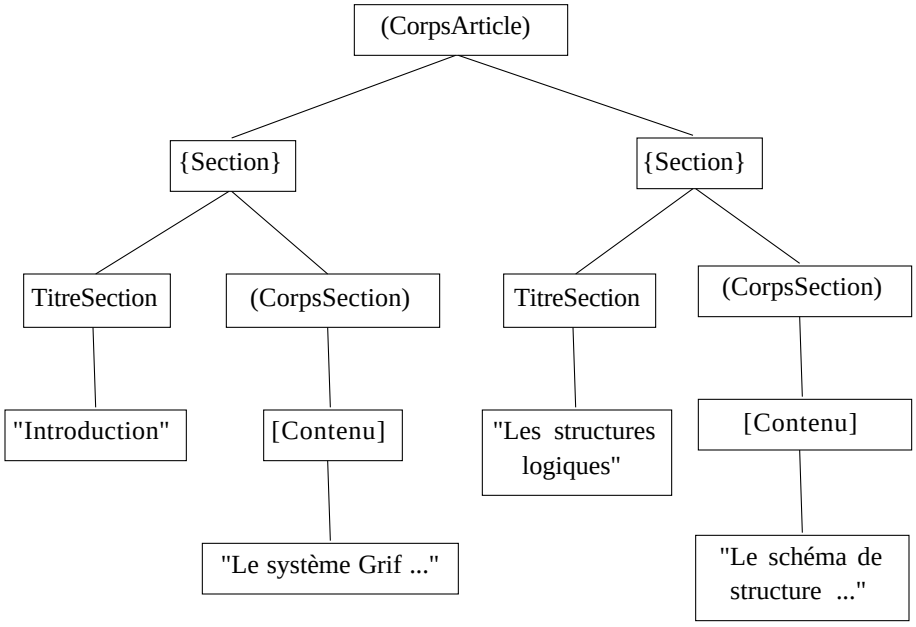


Fig. 2.10 : Illustration de l'exclusion du type IMAGE du type CorpsArticle

II.5 Schémas externes

Les éléments contenus dans un document produit par Grif peuvent provenir de plusieurs schémas de structure. Il s'agit de l'utilisation d'un schéma de structure dans la définition d'un autre schéma de structure. Ceci s'exprime des deux façons suivantes :

- par l'utilisation du constructeur Nature présenté en II.2.2 ;
- par référence explicite à un autre schéma de structure. Il est en effet possible de définir un type avec le constructeur Identité tel que la partie droite de la règle soit le nom d'un schéma de structure. Une instance de type Formule défini ci−après est composée d'un élément de la classe de structure Math, défini par le schéma de structure Math.

```
Formule = Math;
```

  L'arbre d'instance correspondant est un arbre de racine Formule dont l'unique fils est un arbre d'instance du schéma externe Math.

II.6 Schémas d'extension

Certaines applications nécessitent d'inclure des éléments spécifiques dans un document pour leur bon fonctionnement ; par exemple :

- Pour indexer un élément, un système d'indexation l'entoure de marques directement insérées dans le document.
- Pour rendre identifiables les parties spécifiques aux différentes versions d'un même document, les gestionnaires de versions les entourent de marques directement introduites dans le document.
- etc.

Les applications de ce genre ont besoin de connaître la définition des éléments qu'elles sont sensées inclure dans les documents et elles doivent pouvoir fonctionner avec tous les documents, quels que soient les schémas de structure qui ont servi à les produire. La solution à ce problème consiste à définir le type de ces éléments dans un schéma de structure spécial, dit schéma d'extension, propre à une application. Le système Grif permet alors l'inclusion de ces éléments dans les documents qui utilisent ces applications.

Il existe une différence notable entre les schémas externes et les schémas d'extension ; alors que l'utilisation des schémas externes est clairement exprimée dans un schéma de structure (par le mot−clé NATURE ou le nom du schéma voulu), la participation d'un schéma d'extension à l'élaboration d'un document n'existe qu'à travers les applications qui les manipulent. Cette remarque est importante pour l'évolution des types car la lecture d'un schéma de structure ne renseigne pas sur tous les schémas qui peuvent contribuer à la production des documents de sa classe. Il faut donc considérer que les schémas d'extension du système peuvent servir à l'élaboration de tous les documents. À cette différence près, les schémas d'extension et les schémas externes peuvent être assimilés.

II.7 Les attributs

Comme dans SGML (voir I.5.3.4) un attribut est une information associée à une partie ou à l'ensemble d'un document et qui est utile aux applications qui effectuent un traitement sur le document, y compris les SEDS eux−mêmes. En effet, la description de la structure logique ne suffit pas toujours pour qu'un document soit correctement traité.

Certaines informations d'ordre sémantique pourraient être traduites par un aspect typographique particulier. Ainsi un mot qui constitue une définition ou l'auteur principal d'un document sont−ils affichés ou imprimés avec des attributs typographiques spécifiques afin de les mettre en évidence. Dans ce cas, ce ne sont pas les attributs typographiques eux−mêmes qui sont associés aux éléments dans le contenu du document, mais des informations (des attributs) qui viennent qualifier les éléments et qui sont porteuses d'une sémantique à l'intention des applications qui manipulent les documents. Un formateur pourra les traduire par de l'italique, un système de recherche documentaire en fera un autre usage. Dans l'exemple de la figure Fig. 2.1,

- Un auteur peut être qualifié de ‘‘Principal’’ ou de ‘‘Secondaire’’.
- Un mot ou une expression peut être qualifié de ‘‘Définition’’ ou de ‘‘MotImportant’’.

En plus de la création et de l'édition, les SEDS offrent des facilités telles que la correction typographique, la correction orthographique, la coupure des mots, etc. Ces facilités correspondent soit à des ressources, soit à des algorithmes qui dépendent de la langue dans laquelle le document est rédigé. Un attribut sert alors à indiquer la langue dans laquelle sont rédigées les parties du document, lesquelles ne contiennent pas les règles typographiques ou orthographiques à leur appliquer mais uniquement l'identificateur de la langue que les applications sauront exploiter à bon escient. Les SEDS eux−mêmes ont besoin de connaître la langue utilisée pour choisir le jeu de caractères adéquat.

Ces informations d'ordre sémantique, les attributs, sont définis en même temps que les éléments de la structure logique générique et servent à renseigner les applications.

Les attributs de Grif sont classés selon plusieurs critères : les éléments sur lesquels ils portent, leurs types, et le fait qu'ils soient obligatoires ou non.

On distingue deux sortes d'attributs, les attributs locaux et les attributs globaux. Les attributs locaux sont définis exclusivement pour un type donné et ne peuvent accompagner que des instances de ce type. Par exemple, l'attribut NuméroPremPage, local au type Article (voir Fig. 2.1), informe le formateur sur le numéro de la première page du document. L'attribut TypeAuteur est également un attribut local, mais pour le type Auteur. Les attributs globaux par contre s'appliquent à tous les types définis ou utilisés dans un schéma de structure. Ainsi, l'attribut langue peut s'appliquer à toute partie du document et est indispensable dans un document multilingue. Dans la figure Fig. 2.1, les attributs Importance et Programmation sont globaux.

Les attributs, qu'ils soient locaux ou globaux sont de l'un des types suivants :

- Les attributs numériques prennent des valeurs négatives, nulles ou positives (par exemple NuméroPremPage).
- Les attributs énumération prennent leurs valeurs dans un ensemble explicitement indiqué dans les schémas de structure ; par exemple l'attribut Importance peut prendre soit la valeur Définition soit la valeur MotImportant et l'attribut TypeAuteur peut prendre l'une des deux valeurs suivantes : Principal, Secondaire.

- Les attributs référence désignent un élément de la structure logique.
- Les attributs textuels ont pour valeur une chaîne de caractères.

Les attributs peuvent avoir les caractéristiques suivantes quel que soit leur type. Lorsqu'un attribut local est obligatoire, il est créé automatiquement par l'éditeur pour toute nouvelle instance du type auquel il s'applique. S'il ne s'agit pas d'un attribut à valeur imposée (spécifiée dans le schéma de structure), l'éditeur impose à l'utilisateur de lui fournir une valeur.

II.8 Conclusion

Ce chapitre a donné un aperçu général du système Grif qui implémente les concepts de base des éditeurs de documents structurés. Grif se présente comme un système générique paramétrable dans lequel les utilisateurs ont la possibilité d'écrire les schémas de structure et de présentation qui conviennent le mieux à leurs besoins. L'abstraction de la composition des documents à travers les schémas de structure permet l'utilisation de Grif pour illustrer non seulement l'évolution des types et les problèmes qu'elle pose mais aussi les problèmes généraux posés par l'édition structurée. C'est le thème du chapitre suivant.

Chapitre III

Les problèmes induits par la structure

Résumé :

Ce chapitre présente quelques problèmes rencontrés dans les SEDS et qui sont induits par la structuration des documents. Après une introduction donnée en III.1, les sections III.2 et III.3 présentent respectivement les restructurations statiques et dynamiques. La section III.4 aborde le problème de la reconstitution de schémas de structure, la section III.5 traite de l'échange de documents entre systèmes hétérogènes et la section III.6 identifie les problèmes similaires dans d'autres domaines, les travaux en cours et les solutions proposées.

III.1 Introduction

Des avantages certains découlent de la structuration, dont les plus importants sont une assistance contextuelle appropriée pendant la création d'un document, une garantie d'homogénéité de l'action des formateurs, une description séparée de la présentation et de la structure. Il n'en demeure pas moins que quelques problèmes restent posés dans le domaine de l'édition structurée, qui sont le fait même de la structuration et qui expliquent le relatif succès que rencontrent les SEDS. Ces problèmes viennent pour la plupart de la disparition de certaines facilités d'édition acquises par les éditeurs de documents non structurés. Il convient de les résoudre afin de conférer aux éditeurs de documents structurés les facilités que proposent les éditeurs de documents non structurés.

L'opération du couper/coller est un exemple typique de fonctionnalité perdue. Les autres problèmes auxquels les éditeurs de documents structurés sont confrontés sont l'évolution des documents au sens large du terme et l'échange de documents entre systèmes hétérogènes.

III.2 Restructurations statiques

On s'intéresse ici aux transformations des documents rendues indispensables par l'évolution de leur structure logique générique. En effet, on peut souhaiter ajouter un nouvel élément à une classe de documents, ou supprimer d'une classe de documents un élément

déjà existant. On peut vouloir aussi réorganiser la composition d'une classe de documents. Cela mène à une nouvelle structure générique. Dans Grif, la nouvelle structure générique est matérialisée par un nouveau schéma de structure destiné à diriger la construction de nouveaux documents.

L'une des particularités des systèmes de traitement de documents fondés sur la structure logique est qu'un document ne peut être traité correctement que si sa structure logique spécifique correspond à une structure générique connue du système, c'est à dire si elle est conforme à un schéma de structure. L'évolution des schémas de structure rend les anciennes instances de documents incompatibles avec le nouveau schéma de structure et ces documents ne peuvent pas être traités comme souhaité avec la version à jour du schéma de structure.

Une évolution de schéma de structure ne devrait cependant pas se traduire par la perte (l'impossibilité de les éditer) des instances de documents conformes à l'ancien schéma. Le problème qui se pose alors est la conversion des anciennes instances de documents pour qu'elles deviennent conformes au nouveau schéma de structure. Étant donné leur nombre éventuellement élevé, il est souhaitable d'automatiser leur transformation. On peut remarquer que les documents non transformés sont toujours éditables avec leurs schémas d'origine. Le problème est qu'ils ne correspondent plus aux réalités qui justifient la nouvelle définition du schéma de structure.

L'expression *nouveau schéma de structure* désigne en fait beaucoup de choses qu'il est nécessaire d'expliquer :

- Il peut s'agir de l'évolution effective du schéma de structure d'origine dans lequel on a par exemple ajouté de nouveaux types. Dans ce cas précis, on se rend bien compte que si la conversion des documents n'est pas faite, l'édition des anciennes instances (avec le schéma d'origine) ne permettrait pas de créer des instances correspondant aux nouveaux types ajoutés.
- Le nouveau schéma de structure pourrait être un tout autre schéma de structure. C'est la généralisation du problème d'évolution de type par la conversion d'une instance de document compatible à un schéma de structure donné en une instance compatible à un autre schéma de structure connu du système.

  À titre d'exemple, les publications scientifiques imposent des modèles de structure et de présentation auxquels les auteurs doivent se conformer pour que leurs articles soient acceptés. Un article qui doit paraître dans des journaux différents est réécrit selon les recommandations propres à chaque journal. Ces recommandations, dont quelques exemples suivent, sont très variées :

  - existence ou non de mots−clés,
  - regroupement ou non de la bibliographie en fin d'article,
  - tri ou non de la bibliographie,
  - importance du résumé,
  - conformité à un style, à une fonte, à un corps pour des éléments tels que les titres, les légendes, les citations,

- présentation des références bibliographiques,
- valeurs des marges,
- etc.

Les auteurs d'articles scientifiques doivent donc, à l'intention de chaque journal, assumer la fastidieuse tâche de rendre leurs documents conformes aux modèles imposés, un même article pouvant être destiné à plusieurs journaux.

Une telle restructuration peut être automatisée en permettant la rédaction des articles selon un schéma assez général et leur conversion, s'il en est besoin, vers les schémas appropriés aux publications. Cependant, le problème tel qu'il se pose de façon générale est la transformation d'un document, quel que soit son schéma, en un document conforme à un autre schéma. Il suffit pour cela de supposer l'existence d'un programme capable de faire une telle conversion. L'évolution de la présentation ne pose aucun problème particulier, car si le nouveau schéma de structure ne dispose pas déjà d'un schéma de présentation avec les attributs typographiques requis, il suffit d'en créer un nouveau.

Les restructurations logiques des documents dues à un changement de schéma de structure sont dites *statiques*.

Il est intéressant de remarquer que les restructurations statiques consistent en deux catégories de transformations qui peuvent être concomitantes :

- la réorganisation structurale induite soit par un déplacement de composants dans la définition d'un type de constructeur Agrégat ordonné, soit par un changement d'un constructeur, etc.
- le changement de contenu par utilisation de nouveaux types de base à la place de ceux qui ont servi à définir le type qui évolue.

III.3 Restructurations dynamiques

Par opposition aux restructurations précédentes, celles qui ont lieu en cours d'édition, c'est à dire le couper/coller et les transformations locales, sont dites *dynamiques*.

Couper/coller

Le couper/coller est une opération qui consiste à copier ou à couper un bloc d'information (bloc source) d'un document (document source) et à tenter de l'intégrer à un élément (bloc cible) d'un document quelconque (document cible). Dans un SEDS, le fort typage des constituants des documents entraîne le rejet d'une telle opération si les types des blocs source et cible sont différents. Ce genre de problème est une perte de fonctionnalité en comparaison des éditeurs de documents non structurés. Des éditeurs comme QuarkXpress et PageMaker permettent de réaliser ces opérations avec une facilité appréciable, que le document cible soit ou non le même que le document source.

Lorsque les types source et cible sont identiques, l'opération se ramène à une superposition d'arbres identiques, étant donné que chaque élément est représenté

par un arbre. On tente alors de transférer le contenu de l'arbre source dans l'arbre cible. Mais l'identité de structure des arbres source et cible ne garantit pas le succès de l'opération. En effet, le contenu d'un élément se trouve dans les feuilles de son arbre et l'intégration d'un contenu à un autre contenu dépend des opérations permises sur les types de base puisque ce sont eux qui indiquent la nature des informations. Deux cas sont à prévoir :

- Si l'arbre cible est vide, il s'agit de créer un élément vide de même type que l'élément source et d'y transférer le contenu de l'élément source. La solution est triviale et consiste à insérer l'élément source à l'endroit indiqué.
- Si l'arbre cible n'est pas vide, l'opération revient à incorporer le contenu de chaque nœud source au contenu du nœud cible correspondant. Les difficultés majeures se situent au niveau des feuilles de l'arbre. Lorsqu'un nœud source et un nœud cible sont de même type, le problème se pose de savoir comment réaliser les intégrations. Par exemple comment intégrer du texte à un texte, comment intégrer un graphique à un graphique, comment intégrer un symbole à un symbole. Si l'on sait répondre à l'intégration d'un texte à un autre texte par l'opération de concaténation, en revanche, le problème reste ouvert pour les autres intégrations. On pourrait envisager des opérations proches de la concaténation pour intégrer le contenu d'un nœud de type GRAPHIQUE ou IMAGE au contenu d'un nœud de même type.

Lorsque les types des blocs source et cible sont différents, le problème reste entier et porte aussi bien sur les types des nœuds internes des arbres que sur leurs feuilles. Dans ce cas, la solution consiste à trouver pour chaque nœud de l'arbre source, le nœud correspondant dans l'arbre cible et à transférer les informations contenues dans les nœuds sources.

Transformations locales

Un autre problème de transformation dynamique est la transformation locale de structure qui ne passe pas par l'opération de couper/coller. Les problèmes posés par ce genre de transformation sont propres aux éditeurs de documents structurés. Un élément d'un type donné pourrait être localement transformé en un élément d'un autre type, juste en le sélectionnant et en indiquant le nouveau type qu'on souhaite lui attribuer. Il s'agit d'une transformation complexe qui dépend du contexte. Le problème qui se pose est de contrôler les conséquences d'une transformation locale sur l'environnement. En effet, le résultat de la transformation locale a−t−il une conséquence sur les ascendants, les frères et les descendants de l'élément transformé ? Pour répondre à cette question, il faut considérer les constructeurs de l'ascendant de l'élément à transformer.

On peut citer les exemples suivants :

1. La transformation d'un élément de type t en un élément de type t' telle que t et t' soient deux options d'un même choix, abstrait une sorte de

transformation locale. Par exemple la transformation d'un élément de type Exemple en un élément de type SimpleParagraphe.

2. La transformation d'un type t en un type t' telle que t et t' soient composants d'un même élément de type Agrégat ordonné ou Agrégat sans ordre est complexe et nécessite que les conditions de sa réalisation soient précisées. Sans apporter ici une réponse globale à cette question, on comprend qu'une requête de transformation d'un composant en un autre composant optionnel non encore créé soit acceptée et traitée. On comprend que si le constructeur du père du composant transformé était un Agrégat ordonné, la transformation pourrait provoquer une réorganisation des composants, réorganisation qui ne serait pas de rigueur si le constructeur avait été Agrégat sans ordre. La transformation d'un élément en un élément frère est une formulation qui abstrait ce genre de restructuration.
3. La transformation d'un élément en un élément de même type que son ascendant abstrait une autre sorte de restructuration locale.

Problèmes d'interface utilisateur

L'un des avantages de l'édition structurée est de tirer profit de la structure logique des documents pour ne permettre que la création des constituants logiques décrits. L'inconvénient est que l'utilisateur se voit contraint de subir toutes les étapes nécessaires à la définition du type de l'élément à créer. De plus, la structure des documents étant arborescente, un élément fils (une sous−section par exemple) ne peut être créé avant son père (la section correspondante) imposant ainsi une rédaction descendante intimement liée à la structure logique. L'utilisateur subit la structure. Sans perdre de vue l'objectif des SEDS (la production de documents structurés), l'idéal serait d'offrir à l'utilisateur l'illusion de faire de l'édition non structurée, ceci par le truchement d'une interface qui le soulagerait des contraintes indiquées ci−dessus et lui permettrait de typer son texte dès qu'il le jugera nécessaire.

Généralisation

Une façon générale de poser le problème est d'envisager l'intégration d'un bloc source à un élément, ou la création d'un nouvel élément à partir du bloc copié ou coupé ; dans les deux cas, il est souhaitable que les types mis en présence soient quelconques. Le bloc source peut directement provenir, soit du document en cours d'édition, soit d'un autre document, que ces documents soient ou non des instances du même schéma de structure.

III.4 Reconstitution de schémas de structure

Un autre problème propre aux SEDS est l'attribution d'un schéma de structure à un document qui n'en a pas. Deux cas intéressants peuvent se présenter :

1. Le document est un texte plat ou un texte avec un minimum d'organisation non décrite dans un schéma de structure, tel qu'un texte issu d'un éditeur de texte ou d'un éditeur de documents non structurés.
2. Le schéma de structure d'un document est détruit par mégarde ou n'est pas transmis avec le document. Ce problème est d'autant plus important que le nombre de documents issus du schéma de structure perdu est éventuellement élevé. Il est impératif de reproduire le schéma de structure.

Dans les deux cas, l'alternative pour éditer ces documents est soit de les rendre conformes à l'un des schémas de structure connus du système soit, de leur générer un schéma de structure sur mesure. Ces schémas de structure sont dits *schéma de structure d'accueil*.

Lorsque le document n'est pas issu du système, sa conversion pourrait se faire avec l'assistance de l'utilisateur qui choisit le schéma de structure d'accueil qu'il juge le plus convenable. L'utilisateur devrait alors sélectionner des blocs d'information et exprimer pour chacun d'eux le type (décrit dans le schéma de structure d'accueil) qu'il voudrait leur attribuer.

Lorsque le document est issu du système, le problème posé est la régénération du schéma de structure. Cette régénération devrait être le fait exclusif du système c'est−à−dire sans l'intervention de l'utilisateur. Le système, dans ce cas, sait que le document est compatible avec la structure générique perdue. En appliquant des règles internes de reconnaissance de blocs, il pourrait reconstruire une structure logique générique minimale qui permettrait l'édition du document. Cette reconstitution non assistée d'un schéma de structure, permettrait au moins dans un premier temps de pouvoir éditer le document. Dans l'hypothèse où plusieurs reconstitutions de schémas concerneraient des documents issus de la même structure logique générique, un schéma de structure commun devrait être proposé pour que les documents appartiennent à nouveau à une même classe. Ce schéma de structure commun serait l'union ensembliste des schémas de structure d'accueil reconstitués par chaque document. Les documents compatibles aux schémas de structure d'accueil seraient alors considérés comme des objets migratoires vers le schéma de structure commun en leur appliquant une restructuration statique comme défini en III.2.

III.5 Échange de documents entre systèmes hétérogènes

Les conversions dont il a été question jusqu'à maintenant sont internes à un même système dans la mesure où elles mettent en œuvre des schémas de structure qui lui sont propres. Lorsqu'un document doit passer d'un système à un autre, le problème qui se pose est double :

1. Le format de représentation interne de documents du système source est différent de celui du système destinataire.

2. Les langages de spécification de la structure logique générique ne sont pas les mêmes.

Étant donné le nombre élevé de systèmes existants, la solution la plus simple à ce problème est l'adoption par les systèmes de la norme SGML conçue pour permettre l'échange de documents et des DTD qui ont servi à les produire.

Lorsque le système destinataire n'est pas conforme à la norme SGML, le premier problème à résoudre est la conversion de l'instance de document conformément au format de représentation interne du système hôte. Le deuxième problème est la génération de la structure logique générique à partir de l'instance de document (voir III.4).

III.6 Domaines concernés et travaux relatifs

L'objectif premier de l'abstraction de la structure dans le domaine de l'édition structurée est de fournir à l'utilisateur un environnement d'édition agréable. Des applications autres que les éditeurs pourront tirer partie de la structure. Bien que ce domaine ne soit pas encore totalement exploré, la plupart des travaux de recherche portent sur la présentation [Brown90] [Hansen90].

Édition structurée

Dans le domaine de l'édition structurée, la plupart des solutions proposées concernent les restructurations statiques. Elles ne sont pas entièrement satisfaisantes. Une solution est proposée pour le SEDS Rita [Cowan91], qui permet d'effectuer des restructurations simples. Dans Rita, un élément de type t ne peut être converti en un élément de type t' que si les composants de t apparaissent dans le même ordre dans t'. En cas d'échec, l'utilisateur de Rita peut effectuer une conversion manuelle dans une zone prévue à cet effet, où le contrôle de validité de structure est désactivé. Le résultat de la transformation manuelle est ensuite inséré dans le document si le contexte le permet.

Une amélioration apportée par Cole et Brown à cette solution permet de réaliser les opérations de couper/coller [Cole92]. Cette nouvelle solution est basée sur l'ajout au schéma de structure d'une liste de couples de types, chaque couple étant une instruction de conversion. La sémantique d'un couple de types est qu'une instance conforme au premier type peut être transformée en une instance conforme au deuxième type. Par exemple le couple suivant spécifie qu'une instance de type Paragraphe peut être convertie en une instance de type SimpleParagraphe.

```
Paragraphe > SimpleParagraphe
```

Si l'application d'une instruction ne donne pas le résultat attendu, la conversion se poursuit avec d'autres instructions. Les conversions peuvent rendre invalide la structure logique englobante. Dans ce cas, l'éditeur essaie de corriger l'erreur induite en vérifiant la validité structurale des éléments frères et ascendants [Cole90].

Cette solution n'est pas très pratique dans la mesure où, pour un schéma de structure définissant n types, $(n^2 - n)$ instructions sont nécessaires pour exprimer les conversions possibles. Étant donné le rôle que peuvent jouer les schémas externes dans la définition des types, il semble difficile de réaliser correctement les conversions avec cette technique lorsque les éléments concernés proviennent de schémas externes. La raison est que les instructions de conversion sont limitées à un schéma de structure. Il est donc impossible d'effectuer des conversions entre documents de schémas différents. De plus, les instructions de conversion sont établies ''à la main'' et rien ne garantit qu'elles produisent des structures valides.

Une autre approche a été proposée par Furuta et Scotts [Furuta87]. Elle permet d'établir une correspondance entre les éléments source et destination en faisant usage de la théorie des graphes hiérarchiques.

DSSSL [I.S.O91a] dispose d'un langage appelé *General Language Transformation Process* (GLTP) destiné à exprimer la transformation d'un document conforme à une DTD donnée, en un document conforme à une autre DTD. Cette transformation nécessite la connaissance des DTD concernées, du document à transformer et d'une association entre les types de l'ancienne DTD et ceux de la nouvelle DTD. L'écriture manuelle de ces règles d'association ne garantit pas la conformité du résultat de la transformation au nouveau schéma de structure. Alors que Brown, Wakayama et Blair [Brown92] proposent un nouveau langage dérivé de DSSSL pour une expression plus simple des transformations de documents SGML, la solution préconisée par Güting [Güting89] est algébrique et suppose l'existence d'opérateurs que l'on peut appliquer aux éléments de documents en vue de les transformer.

Toutes les solutions proposées ci−dessus ne s'appliquent qu'aux transformations statiques, les restructurations dynamiques restant encore inexplorées, sauf dans Rita. L'auteur de la présente thèse a déjà fait une analyse globales des transformations statiques [Akpotsui92] et dynamiques [Akpotsui93].

Bases de données

La notion de schéma telle que définie dans les SEDS n'a pas la même sémantique que dans les SGBD. Dans les SEDS, un schéma est un type auquel correspondent des instances. En revanche dans les SGBD relationnels, le schéma d'une relation [Delobel82] est un couple $\langle X, \Sigma \rangle$ qui définit un ensemble de relations de type X dont les éléments sont des n−uplets qui satisfont aux contraintes d'intégrité Σ. Il n'y a pas de traitement global de type sur l'entité relation mais sur les n−uplets qui la composent ; cette remarque s'applique également au schéma relationnel (S, Σ) d'une base de données, défini comme suit :

$$S = \{\langle X_i, \Sigma_i \rangle \mid i \in [1 \,..\, n]\} \text{ et } \Sigma = \bigcup_{i=1}^{n} \Sigma_i.$$

De même que pour les bases de données relationnelles, la notion de schéma dans les SGBDOO correspond à un niveau d'abstraction absent des SEDS et représente

généralement un ensemble de classes. Par exemple, le modèle proposé par S. Abiteboul et C. Kanellakis [Abiteboul89] définit un schéma comme un triplet (R, P, T) où R est un ensemble fini de relations, P un ensemble fini de classes et T une fonction telle que $T : R \cup P \rightarrow types(P)$. L'évolution d'un schéma résulte alors d'au moins deux choses :

1. la re−définition de sa composition en schémas de relation ou en classes,
2. l'évolution des schémas de relation ou des classes qui entrent dans sa définition.

Du simple point de vue de l'évolution des types, les schémas des SEDS jouent le même rôle que les classes ou les schémas de relation dans les SGBD. L'évolution des schémas dans les SGBDOO est abordée sous des aspects divers dont quelques uns sont donnés en exemple :

1. L'utilisation de bases temporelles est une approche qui se fonde sur la sauvegarde d'informations constituées à partir de la datation des données et des méta−données de la base [Roddick91] [Roddick92] [Rose91].
2. L'approche fondée sur la gestion des versions des classes d'objets et des schémas (AVANCE [Björnerstedt88] [Björnerstedt89], CLOSQL [Monk93]) autorise la coexistence dans le système de plusieurs versions d'une même classe. Ainsi, moyennant une interface système adéquate, les données sont transformables de la version courante vers les versions antérieures ou vers les versions plus récentes. Dans CLOSQL, toute évolution de classe, donc l'ajout d'une nouvelle version de classe, nécessite l'écriture de deux méthodes qui garantissent la conversion des données entre les deux versions (voir figure Fig. 3.1). Tout se passe comme si les données étaient de type indéterminé et leur formatage, une tâche dévolue aux requêtes en fonction des spécifications de l'utilisateur où des programmes d'application.
3. Une dernière approche consiste à ne retenir que la version la plus récente des classes qui composent un schéma.

Le problème d'évolution de types tel qu'il se pose dans les bases de données orientées objet, présente des particularités induites par le concept d'objet. En effet, le concept d'héritage et l'abstraction des données introduisent un degré de complexité qui rend les approches différentes de celles proposées pour les SEDS. L'évolution des types dans le domaine des SGBDOO porte aussi bien sur les méthodes que sur les classes et conduit logiquement à considérer différemment les problèmes d'évolution de structure dans ce domaine. Les travaux effectués dans ORION [Banerjee87] ont permis d'identifier le problème comme la possibilité de modifier les schémas et les méthodes sans induire des erreurs à l'exécution ou des comportements anormaux des méthodes. Ces problèmes sont classés en deux catégories, la cohérence de la structure qui se rapporte à la composante statique de la base de données et la cohérence des méthodes relative à la composante dynamique de la base de données. Dans la plupart des SGBDOO tels que O_2

[Bancilhon92], ORION, GemStone [Penney87], ENCORE [Skara86] [Skara87], des transformations élémentaires de structure sont identifiées (cohérence de la structure) et des solutions proposées.

L'évolution des méthodes induit des problèmes [Zicari89] complexes auxquels O_2 et ENCORE apportent des solutions partielles.

Dans le modèle proposé par S. Abiteboul et C. Kanellakis une instance appartenant à un schéma S = (R, P, T) est un triplet I = (ρ, μ, ν). Le langage IQL permet de faire une transformation d'instances dans les conditions suivantes. Un programme IQL noté $\Gamma(S, S_{in}, S_{out})$ crée une instance I' appartenant à l'ensemble des instances de S_{out} à partir d'une instance I appartenant à l'ensemble des instances de S_{in}, S_{in} et S_{out} étant des projections de S. Un schéma S est une projection d'un schéma S' = (R', P', T') si $R' \subseteq R$, $P' \subseteq P$ et si T' est une restriction de T à $R' \cup P'$; dans ces conditions, I' la projection de I sur S_{out} est notée $I' = I[S_{out}]$. IQL étant un langage d'interrogation, on peut considérer qu'il permet de réaliser des transformations dynamiques.

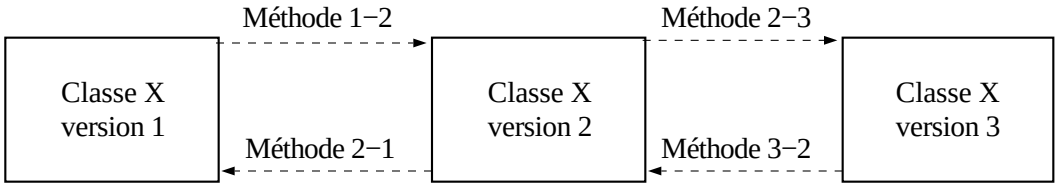


Fig. 3.1 : évolution de classe fondée sur la gestion des versions selon CLOSQL

Éditeurs Syntaxiques

Les documents (textes des programmes) obtenus avec les éditeurs syntaxiques sont représentés par des arbres syntaxiques. Les grammaires des langages de programmation et les arbres syntaxiques sont aux éditeurs syntaxiques ce que les schémas de structure et les documents sont aux SEDS. Tout changement de la grammaire rend les programmes, donc les arbres syntaxiques, non conformes et conduit souvent à des résultats erronés de l'exécution des programmes. Parmi les travaux relatifs au problème de l'évolution des types dans ce domaine, on peut citer les solutions [Garlan86] proposées pour le système Gandalf [Notkin86], qui nécessitent que le programmeur exprime les différences simples entre les deux grammaires. Lorsque cette méthode ne donne pas les résultats attendus, le programmeur peut, soit fournir des routines spécifiques pour la transformation de ses programmes, soit utiliser le langage ARL [Staudt86] pour exprimer de manière plus complète les différences à l'intention du programme chargé de la transformation.

Langages de programmation

Dans les langages de programmation, les solutions apportées aux problèmes de transformation des types sont, soit exprimées par le programmeur soit appliquées dynamiquement pendant l'exécution. Deux types sont équivalents s'ils sont définis

avec le même identificateur de type, c'est le principe retenu dans les langages tels que ADA, C, etc. A l'origine, la transformation des types a été abordée sous l'angle de l'équivalence structurale (ALGOL 68) mais la complexité des algorithmes est telle que les langages actuels retiennent la solution fondée sur l'équivalence des définitions [Cardelli85].

III.7 Sujet de la thèse

Dans la suite de cette thèse, nous nous intéresserons aux problèmes de transformation de types dans les éditeurs de documents structurés.

Les restructurations statiques et dynamiques feront l'objet d'une étude approfondie. Les solutions apportées sont suffisamment générales pour s'appliquer à l'ensemble des systèmes de traitement de documents et ont été implantées et validées dans le système Grif en particulier.

Chapitre IV

Évolution des types et restructurations dynamiques

Résumé :

Ce chapitre présente une typologie des transformations de types à travers leur définition dans un schéma de structure. La section IV.1 fixe les objectifs du chapitre et le cadre dans lequel s'expriment ces transformations. La section IV.2 présente l'évolution des types, la section IV.3 l'évolution des schémas de structure externes et la section IV.4 les particularités de l'évolution des types dans les schémas d'extension. Les sections IV.5 et IV.6 exposent respectivement les modifications possibles des attributs et les restructurations dynamiques. La section IV.7 présente les invariants de l'évolution des types.

IV.1 Introduction

La transformation de la structure d'un document est due soit à l'évolution des types, soit aux requêtes exprimées par l'utilisateur pendant une session d'édition. L'évolution d'un type doit être comprise comme une nouvelle définition de ses caractéristiques dans un schéma de structure. On pourra ainsi, en comparant deux versions de la définition d'un même type, identifier leurs différences et établir un répertoire des transformations possibles. Une telle étude qui se déroule en dehors de l'éditeur par la seule considération des types concernés, est une opération statique qui conduit à la mise à jour des instances, guidée par les différences constatées au préalable.

Les transformations statiques diffèrent des restructurations dynamiques par le fait que dans les restructurations dynamiques, les différences entre les types concernés ne sont pas connues d'avance. Pendant une session d'édition, les requêtes de couper/coller peuvent être émises par l'utilisateur quels que soient les types des instances source et cible. La transformation souhaitée par l'utilisateur est la restructuration dynamique de l'instance source, qui peut être perçue comme une évolution dynamique et virtuelle de type.

```
STRUCTURE Article;

ATTR
    Importance = Définition, MotImportant;
    Programmation = Identificateur, MotClé;
STRUCT
Article (ATTR Numero_prem_page = Integer) =
      BEGIN
          ?DateDeMiseAJour =  TEXTE;
          Auteurs = LIST OF (Auteur);
          Titre = TEXTE;
          Corps = SuiteSections;
          ?Esquisse = Section;
      END;

Auteur = BEGIN
           Photo = IMAGE;
           Nom = TEXTE;
           Affiliations = LIST OF (Affiliation);
           ?Grade = TEXTE;
         END;
Affiliation = CASE OF
                    Nom;
                    RefNote;
              END;

SuiteSections = LIST [2 . . *] OF (Section);

Section = BEGIN
              KeyWord = TEXTE;
              Titre = TEXTE;
               SuiteParagr;
          END;

SuiteParagr = LIST OF (Contenu);
Contenu~ = CASE OF
             TEXTE;
             IMAGE;
             RefNote = REFERENCE(Note);
             RefTout = REFERENCE(ANY);
             RefKeyWord = REFERENCE(KeyWord);
             RefBiblio = REFERENCE(Biblio(RefBib));
           END;

ASSOC
Figure = BEGIN
             Illustration = Contenu +(GRAPHIQUE);
             TitreFigure = TEXTE;
         END;
Note = SuiteParagr  −(IMAGE);
```

Fig. 4.1 : Schéma de structure Article

L'objectif de ce chapitre est de faire l'inventaire des différentes sortes de modification de types afin de proposer une méthode de transformation efficace des instances. On peut considérer que la définition d'un type se compose de plusieurs parties :

- La définition de son contenu.
- La définition de ses attributs.
- La définition de ses relations avec d'autres types.

L'évolution d'une définition peut porter sur n'importe laquelle de ces parties. Les conséquences de l'évolution des types ne se limitent pas à la mise en conformité des seules instances concernées dans la mesure où l'environnement peut être modifié :

1. La transformation d'une instance de document peut conduire à la production de plusieurs documents : un document principal et des documents secondaires. Par exemple, la transformation d'un document peut être telle que certaines de ses parties ne peuvent plus être contenues dans le résultat de la transformation ; si ces parties sont référencées et si leurs transformées le demeurent, alors un document dit secondaire est créé dans le but, non seulement de conserver les références, mais aussi les informations contenues dans les éléments référencés.
2. La cohérence des références dont les éléments référencés sont ou étaient dans des documents secondaires, doit être garantie.

L'analyse du problème de l'évolution des types doit tenir compte non seulement des schémas de structure mais aussi de l'environnement en vue de proposer des solutions consistantes. Ce chapitre se limitera à l'identification des évolutions sans établir de relation entre les types.

IV.2 Évolution des types

On s'intéresse ici aux transformations des documents rendues indispensables par l'évolution de leur structure logique générique. On se place dans le cadre du système Grif présenté dans le chapitre II. La description de la nouvelle structure générique est matérialisée par un nouveau schéma de structure destiné à diriger la construction de nouveaux documents.

Les exemples qui illustrent ce chapitre sont basés sur le schéma de structure Article de la figure Fig. 4.1, qui doit être considéré comme l'ancien schéma de structure. Par souci de commodité et pour différencier les définitions anciennes et nouvelles, l'identificateur d'un type dont la définition a changé est constitué de l'ancien identificateur suffixé par le caractère '~'.

Nous présentons maintenant les différentes transformations qui peuvent être effectuées sur les types et pour chacune, nous indiquons les conséquences pour les instances de documents conformes aux anciennes définitions.

IV.2.1 Extension

Un type ayant pour constructeur Agrégat ordonné, Agrégat sans ordre ou Choix peut évoluer par ajout de nouveaux composants ou options ; on dit qu'il fait l'objet d'extension ou qu'il est étendu. Dans l'exemple ci−dessous, le champ Résumé est ajouté dans la définition du type Article~ et les options Formule, Figure~ et GRAPHIQUE sont ajoutées au type Contenu~ :

```
Article~ = BEGIN
             ?DateDeMiseAJour = TEXTE;
             Auteurs = LIST OF (Auteur);
             Titre = TEXTE;
             Résumé = Contenu;
             Corps = SuiteSections;
             ?Esquisse = Section;
           END;
Contenu~ =
  CASE OF
      TEXTE;
      IMAGE;
      GRAPHIQUE;
      RefNote = REFERENCE(Note);
      RefTout~ = REFERENCE(Figure~);
      RefKeyWord~ = REFERENCE(KeyWord(SchKeyWord));
      RefBiblio~ = REFERENCE(CitationBiblio);
      Figure~;
      Formule = Math;
  END;
```

Fig. 4.2 : Extension des types Article et Contenu

S'il s'agit d'un agrégat et que le nouveau composant est obligatoire, une instance correspondante vide doit être insérée comme descendant immédiat de toute instance du type étendu.

Dans le cas d'un choix, aucune instance correspondant à la nouvelle option n'est produite puisque, par définition, les options ne sont pas obligatoires.

IV.2.2 Restriction

La restriction est la transformation duale de l'extension. Les types de constructeurs Agrégat ordonné, Agrégat sans ordre et Choix peuvent perdre certains de leurs composants ; on dit qu'ils font l'objet de restriction ou qu'ils sont restreints.

Dans l'exemple qui suit, le type Auteur est restreint par suppression du composant Photo :

```
Auteur~ = BEGIN
              Nom = TEXTE;
              Affiliations = LIST OF (Affiliation);
              ?Grade = TEXTE;
          END;
```

Fig. 4.3 : Restriction du type Auteur par suppression du composant Photo

Toute instance dont le type est supprimé doit être retirée des instances du type restreint.

IV.2.3 Diminution

La diminution est la réduction du nombre maximum d'éléments dans une liste. Seuls les types de constructeur Liste peuvent être diminués. Soit la définition suivante, tirée de la figure Fig. 4.1, selon laquelle une instance de type SuiteSections doit comporter au moins deux éléments de type Section et un nombre illimité de sections (indiqué par l'astérisque).

```
SuiteSections = LIST [2 . . *] OF (Section);
```

La redéfinition suivante du type SuiteSections~ limite à 10 le nombre de sections dans ses instances :

```
SuiteSections~ = LIST [1 . . 10] OF (Section);
```

Il se pose alors le problème de savoir ce que deviennent les éléments de liste supplémentaires. Ils peuvent être adoptés ou supprimés. La diminution du nombre minimum d'éléments de liste ne nécessite aucun traitement.

IV.2.4 Augmentation

L'augmentation est le relèvement du nombre minimum d'éléments dans une liste. Seuls les types de constructeur Liste peuvent être augmentés.

La redéfinition suivante du type SuiteSections~ porte de 2 à 4 le nombre minimum de sections.

```
SuiteSections~ = LIST [4 . . *] OF (Section);
```

Lorsque le nombre minimum d'éléments de liste est relevé, la transformation consiste à inclure dans la liste, des éléments vides à concurrence du nombre minimum requis. Le relèvement du nombre maximum d'éléments de liste ne nécessite aucun traitement.

IV.2.5 Réarrangement

Un type de constructeur Agrégat ordonné est tel que les instances engendrées par ses composants doivent être créées en respectant l'ordre dans lequel leurs types sont définis. Cette caractéristique suppose l'existence d'un ordre sur l'ensemble des fils d'une instance. Cet ordre doit être préservé par l'évolution des types uniquement lorsqu'il s'agit d'un type de constructeur Agrégat ordonné. On considère qu'à chaque composant correspond un rang qui est sa position par rapport aux autres composants du même agrégat. En revanche, les composants des types de constructeur Agrégat sans ordre apparaissent dans n'importe quel ordre

et peuvent par conséquent ne pas faire l'objet d'un réarrangement. Un nouvel arrangement peut être explicite ou induit.

Changement de rang explicite

Dans la définition d'un type ayant pour constructeur Agrégat ordonné, le déplacement d'un composant par rapport aux autres composants est un changement explicite de rang qui entraîne le changement du rang de tous les composants qui se retrouvent entre son ancienne et sa nouvelle position. Dans l'exemple suivant, le composant Titre est passé en première position, le composant Auteurs en deuxième position alors que les autres gardent leur position.

```
Article~ = BEGIN
             ?DateDeMiseAJour = TEXTE;
             Titre = TEXTE;
             Auteurs = LIST OF (Auteur);
             Résumé = Contenu;
             Corps = SuiteSections;
             ?Esquisse = Section;
           END;
```

Changement de rang induit

Dans les types de constructeur Agrégat ordonné, les restrictions et les extensions peuvent être la cause du changement de rang de certains composants. Les restrictions induisent le changement de rang de tous les composants qui avaient un rang supérieur à celui du composant retiré (voir Fig. 4.3). Les extensions provoquent le changement de rang de tous les composants qui se retrouvent après les nouveaux composants dans la définition de l'agrégat (voir Fig. 4.2).

Les changements de rang peuvent être la conséquence de la combinaison des causes évoquées ci−dessus. Dans tous les cas, chaque fils immédiat de toute instance du type réordonné doit être positionné de façon à correspondre au rang de son type.

IV.2.6 Obligation d'occurrence

Les types de constructeur Agrégat ordonné ou Agrégat sans ordre peuvent avoir des composants optionnels, par exemple le composant Esquisse du type Article et le composant Grade du type Auteur. La disparition de l'indicateur d'option '?' rend obligatoire la présence d'une instance engendrée par le composant redéfini. Par conséquent, lorsqu'un type devient obligatoire, on doit s'assurer qu'il lui correspond un élément dans les instances du type dont il est composant. Dans l'exemple suivant, le composant *Grade~* devient obligatoire :

```
Auteur~ = BEGIN
            Photo = IMAGE;
            Nom = TEXTE;
            Affiliations = LIST OF (Affiliation);
            Grade = TEXTE;
          END;
```

En revanche, l'ajout d'un indicateur d'option n'impose pas la transformation des instances concernées.

IV.2.7 Retrait d'inclusions

On ne peut inclure dans un type que des types qui n'apparaissent pas dans sa définition en tant que composants immédiats. La disparition d'une inclusion est identifiée comme une évolution de type appelée *retrait d'inclusion*. Dans la nouvelle définition suivante du type Figure~, le type de base GRAPHIQUE n'appartient plus à la liste des inclusions du type Illustration~ :

```
Figure~ = BEGIN
              Illustration~ = Contenu;
              TitreFigure = TEXTE;
          END;
```

En interprétant localement les conséquences d'une telle disparition, la sémantique retenue est qu'aucune instance de type *GRAPHIQUE* ne peut appartenir aux instances de type Illustration~. Il faut noter tout de même que la sémantique associée à une définition de type est rarement le fait de la seule définition du type concerné. Nous verrons par la suite que, par le truchement du mécanisme d'héritage, un type peut interagir sur plusieurs autres. C'est ainsi qu'un autre type peut rétablir, dans des conditions qui seront présentées plus tard, des inclusions localement supprimées. En l'occurrence ici, il ne sera pas nécessaire de retirer les instances de type GRAPHIQUE puisque le type GRAPHIQUE est une option du type Contenu qui sert à définir Illustration~.

IV.2.8 Exclusion

Ne peuvent être exclus que les éléments non obligatoires, les éléments qui peuvent avoir plus d'une occurrence et les options de choix. L'apparition de toute nouvelle exclusion implique la combinaison des deux modifications suivantes :

1. Retrait d'éventuelles inclusions : il peut arriver que par la combinaison de l'héritage et d'une définition locale, un élément localement inclus dans un type en soit en même temps exclus. Dans un tel cas, l'exclusion a la plus grande priorité et les anciennes instances incluses doivent disparaître.
2. Retrait de composants non obligatoires ou de composants susceptibles d'avoir plus d'une occurrence ou d'options de choix.

La définition ci−dessous n'autorise pas la présence d'instances de type IMAGE dans les instances de type Note.

```
Note~ = SuiteParagr  −(IMAGE);
```

IV.2.9 Restauration d'exclusions

La restauration d'exclusions se produit lorsque dans une nouvelle définition de type, des composants auparavant exclus ne le sont plus. Cela suppose que les instances correspondant aux anciennes exclusions soient restaurées. Soit les définitions suivantes telles que AuteurSecondaire~ soit la nouvelle définition du type AuteurSecondaire :

```
AuteurSecondaire = Auteur − (Grade);
AuteurSecondaire~ =
          BEGIN
```

```
        Nom = TEXTE;
        Affiliations = LIST OF(Affiliation);
        Grade = TEXTE;
END;
```

Une instance de type Grade doit être incorporée dans toutes les instances de type AuteurSecondaire~. Étant donné la nature des éléments susceptibles d'être exclus, il faut noter que la restauration des exclusions n'est pas obligatoire sauf si les types vers lequels elles évoluent l'exigent.

IV.2.10 Migration

Les schémas de structure comportent deux catégories de types : les types associés et les types principaux. À chacune de ces deux catégories correspond une famille distincte d'instances dans un document : les éléments associés et les éléments principaux. Le passage d'un type de la catégorie des types associés à celle des types principaux ou vice versa est désigné sous le nom de migration. Les migrations sont illustrées par les nouvelles définitions de la figure Fig. 4.4.

Migration de types associés

Le passage d'un type de la catégorie des types associés à celle des types principaux est la migration d'un type associé. Ceci suppose que l'ancien type associé n'appartient plus à son ancienne catégorie. Les instances correspondantes des types qui migrent passent de leur arborescence à l'arborescence des éléments principaux. Dans l'exemple de la figure Fig. 4.4, la présence du type Figure~ dans la définition du type Contenu~ est considérée comme une migration du fait de son absence de la catégorie des types associés. Après le transfert des éléments associés, leur arborescence, en principe réduite à un nœud (la racine), doit être supprimée du document.

Migration de types principaux

Le passage d'un type principal dans la catégorie des types associés est identifié comme la migration d'un type principal. L'ancien type principal n'appartient plus à la catégorie des types principaux. Les instances correspondantes des types qui migrent passent de l'arborescence principale à leur arborescence d'éléments associés. Dans l'exemple de la figure Fig. 4.4, la présence du type `Esquisse~` dans la catégorie des types associés est identifiée comme une migration du fait de sa disparition de l'ensemble des composants du type `Article~` et surtout parce qu'il est défini après le mot−clé ASSOC. L'ancienne instance de type Esquisse passe dans l'arborescence d'éléments associés dont la racine a pour étiquette (Esquisses).

```
Article~ = BEGIN
             ?DateDeMiseAJour = TEXTE;
             Titre = TEXTE;
             Auteurs = LIST OF (Auteur);
             Résumé = Contenu;
             Corps = SuiteSections;
           END;
Figure~ = BEGIN
              Illustration~ = Contenu;
              TitreFigure = TEXTE;
          END;
Contenu~ =
   CASE OF
      TEXTE;
      IMAGE;
      GRAPHIQUE;
      RefNote = REFERENCE(Note);
      RefTout~ = REFERENCE(Figure~));
      RefKeyWord~ = REFERENCE(KeyWord(SchKeyWord));
      RefBiblio~ = REFERENCE(CitationBiblio);
      Figure~;
      Formule = Math;
   END;
ASSOC
     Esquisse~ = Section;
     CitationBiblio = RefBib;
     Note = SuiteParagr;
```

Fig. 4.4 : Illustration des migrations

IV.2.11 Adoption

On appelle adoption la récupération par un type père d'un type quelconque issu d'un autre père.

L'adoption doit être vue comme une généralisation de la migration. Il s'agit d'une opération globale qui peut être appliquée à n'importe quel type défini dans un schéma de structure. Elle est détectée en faisant un rapprochement entre les opérations de restriction et d'extension. En effet, lorsqu'un composant qui disparaît à la suite d'une restriction, est localisé comme extension d'un autre type, il s'agit d'une adoption. Il existe deux types d'adoption :

1. Une adoption est dite *stable* lorsque le père et le fils adoptif sont issus de la même catégorie de types.
2. Une adoption est dite *instable* lorsque le fils adoptif provient d'une catégorie autre que celle du père.

Dans les deux cas, les instances engendrées par les composants retirés du fait des restrictions seraient perdues si le rapprochement entre la restriction et l'extension n'avait pas

été fait. La solution consiste à incorporer aux instances des types étendus, les instances retirées des types restreints.

IV.2.12 Changement de constructeur

Le changement de constructeur est une évolution complexe constatée lorsque, dans sa nouvelle définition, un type n'a pas gardé le même constructeur. Dans l'exemple suivant, le type Affiliation qui était un choix devient un agrégat :

```
Affiliation~ = BEGIN
                    Code = SYMBOLE;
                    Etablissement = CASE OF
                                        Nom;
                                        RefNote;
                                    END;
               END;
```

Un autre cas intéressant de changement de constructeur est le passage d'un agrégat non ordonné à un agrégat ordonné ou vice versa. Dans ce cas, le changement de constructeur s'identifie au changement d'ordre exposé en IV.2.5.

Le passage d'un agrégat ordonné à un agrégat non ordonné ne nécessite aucun traitement.

Pour les autres changements de constructeurs, la connaissance des schémas de structure par l'utilisateur est indispensable pour exprimer correctement les transformations. La récupération du contenu des éléments sources n'est pas garantie si l'évolution est mal formulée. On se rend compte qu'en réalité, ce n'est pas seulement la nature des constructeurs qui détermine totalement la transformation mais aussi la structure globale (la composition) des types.

IV.2.13 Changement d'identificateur

Le système Grif suppose que les types et les attributs sont identifiés de façon unique dans leur schéma de structure. Le changement de l'identificateur d'un type est une évolution sémantique qui ne peut être résolue que si l'utilisateur fournit explicitement une indication permettant de faire une association entre l'ancien identificateur et le nouveau.

IV.2.14 Changement de référence

Le changement de référence est constaté dans les deux cas suivants :

- Le type d'éléments référencés a changé.
- Le schéma de structure auquel appartient le type d'éléments référencés a changé.

Dans l'exemple ci−dessous, le type RefTout~ ne permet de référencer que des instances du type Figure~ défini dans le même schéma de structure que lui :

```
RefTout~ = REFERENCE(Figure~);
```

À l'inverse, le type RefKeyWord~ de l'exemple ci−après permet de créer des références vers des instances du type KeyWord défini dans le schéma de structure SchKeyWord. Dans la définition de RefKeyWord~, le type d'éléments référencés KeyWord est suivi entre parenthèses du nom du schéma de structure (SchKeyword) dans lequel il est défini.

```
RefKeyWord~ = REFERENCE(KeyWord(SchKeyWord));
```

Il convient de mettre à jour les liens vers les anciens éléments référencés en fonction du nouveau type d'éléments référencés. Dans la définition d'une référence, le paramètre du mot−clé REFERENCE représente un ensemble de types dont les instances sont susceptibles d'être référencées. Cet ensemble contient au moins un élément. À titre d'exemple, dans l'ancienne définition du type RefTout, le paramètre ANY désigne l'ensemble de tous les types du schéma de structure, y compris le type Figure. Il en découle que la transformation des instances de type RefTout doit préserver les références vers les instances de type Figure~.

IV.2.15 Intégration

On appelle intégration, la fusion d'une copie totale ou partielle d'un document dit secondaire au document principal à convertir.

Le mécanisme de référence est le seul moyen pour établir une relation entre deux documents. Certaines évolutions de références sont à l'origine de l'intégration de documents. Il existe deux types d'éléments référencés dans une instance de document :

- Des éléments dont les types sont définis dans le même schéma de structure que les références.
- Des éléments dont les types ne sont pas définis dans le même schéma de structure que les références.

On parle d'intégration lorsque la définition du type d'éléments référencés et celle de la référence sont faites dans le même schéma alors qu'elles ne l'étaient pas. Deux cas sont à considérer :

1. Le nouveau type est une extension à un type principal ou à un type associé. S'il s'agit en même temps d'une restriction, on est conduit à résoudre deux problèmes : l'adoption et l'intégration.
2. Le nouveau type d'éléments référencés est un type associé. Dans l'exemple de la figure Fig. 4.5, le type RefBiblio~ référence le type RefBib par l'intermédiaire de *CitationBiblio*. On remarque que la référence et l'élément référencé sont définis dans le même schéma alors qu'ils ne l'étaient pas dans l'ancien schéma (voir figure Fig. 4.1).

La solution consiste à intégrer dans le document principal, la transformée d'une copie des éléments du document secondaire qui sont référencés depuis le document principal. Dans l'exemple de la figure Fig. 4.5, RefBiblio ne référence plus d'éléments externes de type Biblio, censés appartenir à un document secondaire de type RefBib ; par contre, il référence des éléments de type CitationBiblio qui font maintenant partie du même document. Cela

impose l'intégration des éléments référencés de type Biblio (document secondaire) dans le document principal après qu'ils aient été transformés en instances de type CitationBiblio.

```
RefBiblio~  =  REFERENCE(CitationBiblio);
ASSOC
     Esquisse~ = Section~;
     CitationBiblio = RefBib;
     Note = SuiteParagr;
```

Fig. 4.5 : Illustration de l'intégration

IV.2.16 Division

La division est la création de plusieurs documents à partir d'un seul document. C'est l'opération inverse de l'intégration. Elle a lieu lorsque le type d'éléments référencés n'est plus défini dans le même schéma de structure que la référence. Dans ce cas, un nouveau document est constitué avec les anciens éléments référencés qui ne peuvent plus faire partie du document converti. La division est illustrée par les nouvelles définitions des types Section~ et RefKeyWord~ dans l'exemple suivant :

```
Section~ = BEGIN
              Titre = TEXTE;
               SuiteParagr;
           END;
RefKeyWord~  =  REFERENCE(KeyWord(SchKeyWord));
```

Le type KeyWord n'est pas un composant du type Section~ et ne se trouve nulle part ailleurs défini dans le nouveau schéma de structure Article~. En revanche, il est déclaré comme appartenant au schéma de structure SchKeyWord et est toujours référencé par RefKeyWord~.

Toutes les instances de type KeyWord doivent être retirées du document converti, rassemblées dans un document secondaire et référencées comme avant.

IV.3 Évolution des schémas externes

L'utilisation d'un schéma externe par un schéma principal est exprimée explicitement. Lorsqu'un schéma externe évolue, sa nouvelle version a un nouvel identificateur qui sert éventuellement pour la définition des types du schéma principal. À l'inverse des types définis localement, le schéma principal n'a pas connaissance de cette évolution et considère simplement qu'un autre schéma externe est utilisé à la place de l'ancien. Le programme chargé de réaliser la conversion des documents doit être informé de cette évolution afin de faire le rapprochement entre les deux versions de schémas externes et appliquer les transformations attendues. Dans une instance de document en cours de conversion, les instances dont les types sont définis dans un schéma externe sont convertis en fonction de l'évolution de la définition des éléments suivants :

- leurs propres types tels que définis dans leur schéma de structure externe ;
- le type de leurs ascendants dans le schéma principal à cause de l'héritage des inclusions et des exclusions ;
- la règle racine du schéma principal à cause de l'héritage des attributs globaux.

IV.4 Évolution des schémas d'extension

À l'inverse des schémas externes, il n'existe pas de relation directe possible entre un schéma de structure principal et un schéma d'extension. Il en découle que l'analyse statique ne permet pas de constater l'évolution d'un schéma d'extension, alors que ce dernier est susceptible de modifier la définition de certains types du schéma principal pour une instance de document. Tout se passe comme si la définition d'un type était répartie dans deux schémas de structure dont il serait coûteux de faire le rapprochement dans une analyse statique. La prise en compte de cette évolution ne peut se faire que dynamiquement, pendant la conversion des instances de document. Les éléments dont les types sont définis dans le schéma principal sont soumis à deux sortes de conversion :

1. La première est le fait de l'évolution de leur définition dans le schéma principal.
2. La deuxième est liée au retrait de types inclus définis dans les schémas d'extension. En effet, la disparition d'une inclusion définie dans un schéma d'extension entraîne le retrait de ses instances.

Les éléments du document en cours de conversion, dont les types sont définis dans le schéma d'extension, sont traités comme ceux des schémas externes.

IV.5 Évolution de la définition des attributs

Il existe plusieurs types d'attributs :

- Les attributs numériques
- Les attributs textuels
- Les attributs énumérés
- Les attributs référence.

Changement de type

D'un point de vue syntaxique, il est possible de faire évoluer un attribut en lui donnant n'importe quel autre type. Sur le plan sémantique, la récupération des informations associées aux attributs qui évoluent demeure un problème pour lequel il n'existe pas toujours de solution satisfaisante. Généralement, la conversion appliquée dans ce genre d'évolution est l'affectation d'une valeur par défaut. Cependant, certaines évolutions offrent la possibilité de traitement spécifique comme dans les exemples suivants :

1. La transformation d'un attribut numérique en attribut énuméré peut consister à considérer sa valeur numérique comme une valeur énumérée à

condition que la valeur numérique appartienne au domaine des valeurs énumérées ; il en résulte que les valeurs négatives ne sont pas récupérables.

2. Dans la transformation d'un attribut énuméré en attribut numérique, la valeur entière correspondant au rang de la valeur énumérée peut être retenue. Cette considération est toujours possible.
3. La conversion d'un entier en chaîne de caractères.

Evolution de la liste des valeurs énumérées

Une valeur énumérée est identifiée par un numéro, lequel est sa position dans la liste des valeurs énumérées à laquelle elle appartient. Les opérations suivantes sont de nature à changer ce numéro :

1. Ajout de nouvelles valeurs.
2. Suppression de valeurs.
3. Déplacement de valeurs.

La solution à ces problèmes consiste à mettre à jour, par leurs nouveaux numéros d'identification, les valeurs des attributs concernés. Dans l'exemple suivant, la nouvelle définition de l'attribut Importance~, par ajout de la valeur Exemple, fait passer le numéro de la valeur énumérée MotImportant de 2 à 3.

```
Importance~ = Définition, Exemple, MotImportant;
```

Obligation d'occurrence

Un attribut peut devenir obligatoire. Dans les instances de documents, un tel attribut doit être créé avec une valeur par défaut ou une valeur fournie par l'utilisateur, à moins qu'une valeur imposée ne soit fournie dans le schéma de structure.

Changement de valeurs imposées

La valeur imposée d'un attribut peut changer dans le schéma. Une mise à jour des instances s'impose alors.

Suppression

Lorsque dans un schéma un attribut local est supprimé de la liste des attributs d'un type, cet attribut doit être retiré des instances du type concerné.

Lorsque dans un schéma, un attribut global est supprimé, cet attribut doit être retiré de tous les éléments du document qui le portent.

Changement de type d'éléments référencés

L'évolution de la définition des attributs références est identique à celle des types. Les traitements effectués en cas d'évolution des attributs références sont les suivants :

1. Lorsque le type d'éléments référencés change, les liens vers les anciens éléments référencés sont supprimés et l'attribut ne référence plus aucun élément. En revanche, si l'ancien type d'éléments référencés est quelconque, les liens sont conservés.

2. Lorsque le schéma de structure du type d'éléments référencés n'est plus le même que celui de l'attribut, soit on crée un nouveau document par division soit on détruit les liens vers les anciens éléments référencés.
3. Lorsque le schéma de structure du type d'éléments référencés devient le même que celui de l'attribut, les éléments référencés du document secondaire sont intégrés dans le document principal après une éventuelle transformation.

Ajout

La liste des attributs locaux ou globaux peut accueillir de nouveaux attributs, ce qui peut avoir des conséquences sur certaines caractéristiques de certains des anciens attributs. Il s'agit en principe de détails d'implantation qui, sans être considérés comme une évolution des attributs, contribuent néanmoins à perturber la structure du document. Dans le système d'édition de documents structurés Grif, un ajout d'attributs à une position autre qu'en fin de liste des attributs entraîne le changement du numéro des attributs.

Renommage

Le changement d'identificateur d'un attribut est reconnu uniquement si l'utilisateur fournit explicitement une indication permettant de faire une association entre l'ancien identificateur et le nouveau. Dans ce cas, les changements de définition indiqués ci−dessus sont recherchés et traités.

Remarque

L'évolution de la définition des attributs globaux concerne toutes les instances de tous les types définis dans le schéma de structure. Celle des attributs locaux, par contre, ne concerne que les instances du type concerné. Les solutions doivent en tenir compte.

IV.6 Restructurations dynamiques

Un schéma de structure est un ensemble de n types et d'attributs. La disponibilité de l'opération couper/coller dans un SEDS laisse supposer la possibilité de faire évoluer un type vers n'importe lequel des $(n-1)$ autres types du même schéma ou vers les types définis dans les schémas externes utilisés par le schéma principal. Un rapprochement entre les restructurations dynamiques et les transformations statiques permet d'envisager, dans l'absolu, l'application aux restructurations dynamiques des solutions retenues pour les transformations statiques.

IV.7 Invariants de l'évolution des types

Les systèmes d'édition de documents structurés confèrent aux instances de documents des propriétés fondamentales qui reflètent les caractéristiques structurales des types. La stabilité de ces propriétés est garantie par les SEDS quelles que soient les opérations

d'édition effectuées. Celles concernées par les transformations statiques et les restructurations dynamiques sont présentées ci−après :

- Une instance ne peut appartenir qu'à une seule des deux familles d'arborescences suivantes : les éléments associés, les éléments principaux.
- À une instance d'élément correspond un type identifié de manière unique dans son schéma de structure.
- Une instance d'élément peut être accompagnée d'instances d'attributs identifiés de manière unique dans la liste de ses attributs locaux.
- Une instance d'élément peut être accompagnée d'attributs globaux définis dans son propre schéma de structure ou dans les schémas de structure de ses ascendants.
- Une instance d'élément hérite des inclusions et exclusions de l'ensemble des types de ses ascendants.
- Les fils immédiats d'une instance de constructeur Agrégat ordonné doivent respecter l'ordre de déclaration de leur type.

Ces propriétés sont désignées comme invariantes et, en tant que telles, doivent être préservées par les restructurations nécessitées par l'évolution des types et par les opérations de restructuration dynamique.

IV.8 Conclusion

Le système d'édition de documents structurés Grif a servi de cadre aux réflexions qui ont conduit à cette typologie. Certaines particularités de ce système sont prises en compte dans l'exposé, telles que :

- la possibilité de préciser le nombre minimum et le nombre maximum des éléments de liste,
- l'absence d'attributs à valeur multiple à l'inverse de SGML,
- l'existence du concept de schéma d'extension,
- le nombre restreint de types de base, qui ne change pas fondamentalement la nature des problèmes de l'évolution des types, puisque les types de base ne sont pas construits dans les schémas de structure.

Le chapitre suivant, fondé sur l'analyse de ces évolutions statiques, propose un modèle conceptuel de conversion de documents qui ne prend pas en compte les restructurations dynamiques (couper/coller) traitées plus loin (voir chapitre VI).

Chapitre V

Modèle conceptuel de transformation statique

Résumé :

L'objectif de ce chapitre est de proposer une méthode générale pour la transformation des documents lorsque les types qui ont servi à les produire ont évolué. La section V.2 donne une présentation du schéma conceptuel de transformation sur lequel se base cette étude. La section V.3 positionne, dans le schéma conceptuel, les schémas de structure concernés par l'évolution des types. La section V.4 présente les règles de transformation, expression des différences entre les types.

V.1 Introduction

L'évolution des types peut induire la conversion d'une quantité importante de documents qu'il serait fastidieux de réaliser manuellement. La solution proposée est fondée sur le souci de soulager l'utilisateur autant que possible, en automatisant d'une part, l'identification des différences entre deux définitions de types, d'autre part la conversion des instances de documents. Le modèle proposé s'adapte bien aux transformations statiques, c'est−à−dire celles induites par l'évolution des définitions de types.

V.2 Schéma conceptuel de transformation de structure

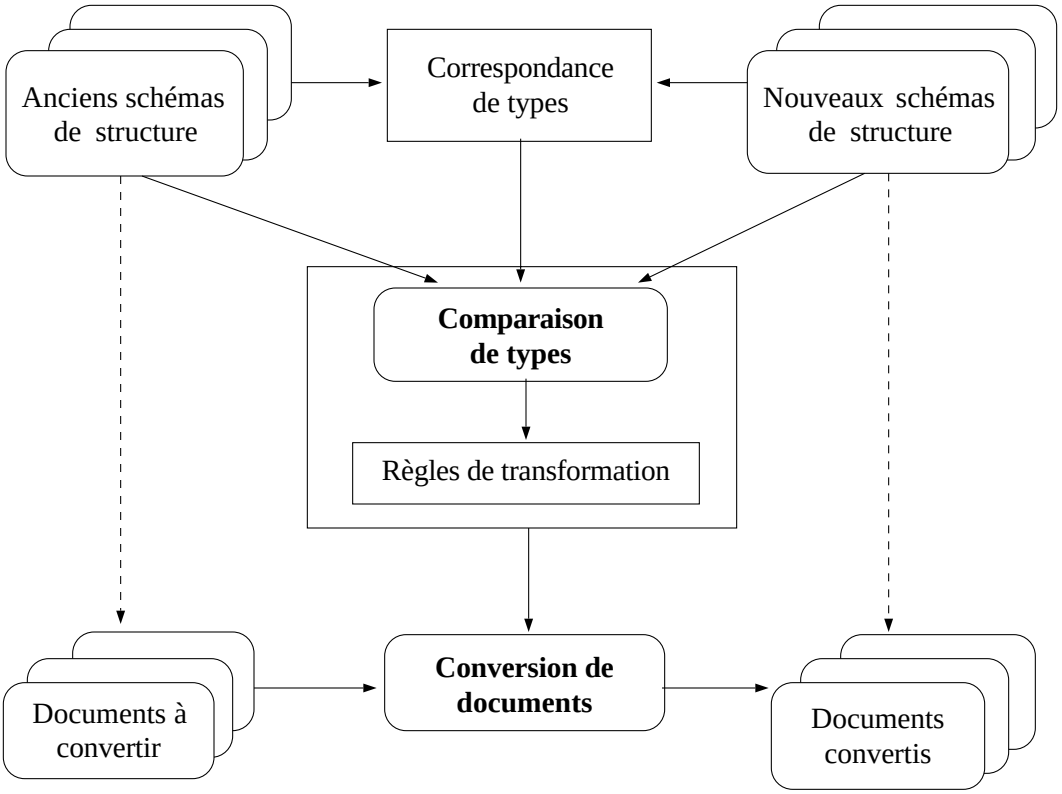


Fig. 5.1 : Schéma conceptuel de transformation de structure

Le schéma conceptuel de transformation proposé (voir Fig. 5.1) est construit autour de deux composants importants, le processus de comparaison de schémas et le processus de conversion de documents. L'idée générale retenue est de disposer des différences entre deux définitions de type données. Ces différences exprimées sous forme de règles appelées *règles de transformation*, doivent servir ensuite pour réaliser la conversion des documents. Les composants de ce schéma sont présentés ci−après.

V.2.1 Comparaison de types

La comparaison de types est un processus qui compare deux schémas de structure, l'ancien schéma de structure et le nouveau schéma de structure. Dans notre cas, il s'agit d'un programme appelé *comparateur*.

L'ancien schéma de structure est, soit un schéma de structure dont la définition a évolué comme exposé dans le chapitre précédent, soit un schéma de structure dont on veut rendre les documents conformes à un autre schéma de structure. Le nouveau schéma de structure est, soit une autre version de l'ancien schéma de structure, soit un schéma de structure quelconque vers lequel on désire faire évoluer les instances de documents de l'ancien schéma de structure. Les instances de document produites à partir de l'ancien schéma vont être transformées en fonction du nouveau schéma.

Ces deux schémas de structure sont donc donnés en entrée au comparateur dans le but de mettre en évidence leurs différences de manière automatique. Toutes les différences de types analysées dans le chapitre précédent sont recherchées par le comparateur. Le comparateur constate les différences et les exprime sous forme de règles de transformation. Les règles de transformation permettront ensuite de convertir les documents convenablement.

Il est important de remarquer que la comparaison de deux schémas peut induire, si besoin, celle des schémas externes utilisés pour leurs définitions. Dans ces conditions, les règles de transformation doivent être vues comme un ensemble de règles issues de la comparaison de plusieurs couples de schémas, c'est−à−dire du couple formé par les deux schémas principaux ainsi que des couples formés de schémas externes. Le comparateur n'est pas en mesure de savoir si un schéma d'extension a servi ou non à la production des documents qui vont être convertis. Il ne peut donc pas comparer automatiquement les schémas d'extension comme il le fait pour les schémas externes. La comparaison des schémas d'extension doit donc être demandée explicitement.

Pour fonctionner, le comparateur a besoin de connaître exactement les types à comparer. Il est clair qu'un type de l'ancien schéma est comparé à un type donné du nouveau schéma de structure vers lequel on veut faire évoluer ses instances. Compte tenu du fait que les types ne sont pas porteurs de sémantique, la constitution des couples de types (voir V.3.1) soumis au comparateur est une opération partiellement manuelle. L'ensemble de ces couples dits couples d'évolution, constitue une ressource appelée *correspondance de types*, destinée au comparateur.

Le comparateur produit plusieurs sortes de règles :

- Règles de transformation de structure : elles reflètent les différences de structure entre deux définitions de types.
- Règles de transformation d'attributs locaux : elles reflètent les différences entre les attributs locaux des types comparés.
- Règles induites de transfert de contenu : elles correspondent à des règles produites par le comparateur dans le but de ne pas perdre le contenu d'un élément.
- Règles de destruction d'éléments.
- Règles de transformation d'attributs globaux : elles reflètent les différences entre les attributs globaux des schémas comparés.

Ces règles sont associées aux types et attributs de l'ancien schéma de structure. Elles sont détaillées dans la section V.4.

V.2.2 Conversion de documents

La conversion de documents est assurée par un processus qui porte le nom de convertisseur. Le rôle du convertisseur est de transformer les documents conformes à l'ancien schéma de structure (*anciens documents*) en documents conformes au nouveau schéma de structure (*nouveaux documents*). En réalité, une transformation de documents peut nécessiter plusieurs schémas. Le processus de conversion réalise les transformations à partir des

documents à transformer, des anciens schémas de structure, des nouveaux schémas de structure et des règles de transformation générées par le comparateur.

Le document à transformer est chargé dans la mémoire de l'ordinateur et parcouru en vue d'identifier ses composants logiques. Chaque élément du document est transformé comme indiqué dans les règles de transformation associées à son type. Le résultat de la transformation est sauvegardé comme un nouveau document qui pourra être édité avec le nouveau schéma.

V.3 Correspondance de types

Le terme *types* dans l'expression *correspondance de types* désigne aussi bien les éléments de structure que les attributs. Un *couple d'évolution* contient l'identificateur d'un type de l'ancien schéma de structure (*ancien type*), celui d'un type du nouveau schéma de structure (*nouveau type*) et une liste de couples d'attributs obtenus à partir des attributs locaux des types du couple.

L'automatisation de la comparaison de types suppose la connaissance par le comparateur des couples de types à comparer et le problème se pose alors de déterminer les éléments des couples d'évolution. Cette automatisation tire avantage du fait que l'identificateur d'un type est unique dans son schéma de structure. Systématiquement, le comparateur mettra dans un même couple deux types tels que leurs identificateurs soient identiques. L'existence d'un même identificateur dans deux schémas de structure est le *principe d'homonymie*. Il faut noter qu'un type peut garder le même identificateur et être pourtant redéfini ; c'est d'ailleurs le cas le plus courant. La constitution des couples détermine les transformations et les couples formés automatiquement pourraient ne pas convenir. C'est pour cette raison que le comparateur admet que l'utilisateur fournisse manuellement les couples d'évolution en fonction des transformations souhaitées. Ces couples ont priorité sur ceux produits automatiquement.

La correspondance entre les types de l'ancien et du nouveau schéma s'exprime au moyen du *langage K* dont la grammaire est donné en Fig. 5.4.

V.3.1 Constitution des couples

Le choix des types appartenant à un couple d'évolution est régi par une sémantique qu'il est nécessaire de définir, l'homonymie n'étant pas le seul critère du choix du nouveau type pour un couple d'évolution. Les situations suivantes sont courantes :

- Deux types d'identificateurs différents et qui n'appartiennent pas au même schéma de structure peuvent, soit avoir la même sémantique, soit avoir la même définition.
- Comme présenté dans le chapitre précédent, l'évolution d'un type peut consister à changer son identificateur tout en lui gardant la même sémantique.

Dans ces deux cas, bien que leurs identificateurs soient différents, il est indispensable que les deux types appartiennent au même couple d'évolution, si la transformation des instances de l'ancien type en celles du nouveau type est souhaitée.

D’une façon générale, un ancien type peut être comparé à un nouveau type dans les conditions suivantes :

- L’ancien type n’a pas évolué et son identificateur demeure le même dans l’ancien et le nouveau schéma de structure.
- L’ancien type garde le même identificateur mais sa définition a évolué.
- La définition de l’ancien type a évolué mais son identificateur reste inchangé.
- La définition et l’identificateur de l’ancien type ont changé.
- On souhaite que l’ancien type évolue vers un autre type.

Dans ces cas, les deux types forment un couple d’évolution et leurs attributs locaux sont aussi regroupés en couples pour être comparés. Les raisons qui conduisent au choix des éléments d’un couple d’attributs sont les mêmes que pour les types.

L’utilisateur ne donne pas manuellement la liste complète des couples mais seulement les couples dont il fait évoluer l’ancien type autrement que par le principe d’homonymie. Le comparateur applique le principe d’homonymie et complète la liste des couples d’évolution sur la base des couples fournis par l’utilisateur. Les règles produites par le comparateur dépendent de la sémantique des couples :

1. Les instances de l’ancien type sont transformées en instances compatibles au nouveau type.
2. Les instances des types de l’ancien schéma qui n’auront pas d’équivalents dans le nouveau schéma sont détruites à moins qu’une règle d’adoption n’en décide autrement.

La constitution des couples est régie par les conditions suivantes qui doivent être connues de l’utilisateur :

- Pour détruire les éléments dont les types n’appartiennent pas au nouveau schéma, le comparateur engendre des couples d’évolution particuliers dans lesquels le nouveau type est **#NULL**(1).
- À un type de l’ancien schéma de structure correspond au plus un seul type du nouveau schéma de structure, c’est−à−dire qu’un même type de l’ancien schéma de structure ne doit pas apparaître dans plus d’un couple d’évolution. Ce choix s’explique par le fait que les types sont génériques et qu’un même type peut être utilisé librement dans la définition d’autres types ; par exemple, le type SuiteParagraphes est librement utilisé dans la définition des types Section et SousSection de la figure Fig. 5.2. L’invocation du seul identificateur d’un type dans un couple d’évolution suppose la conversion de toutes les instances de ce type conformément au nouveau type. En ajoutant de la sémantique aux éléments des couples d’évolution, il est possible de préciser le contexte des instances concernées par les types dans les couples d’évolution. Le langage K permet d’exprimer cette sémantique par la spécification, si nécessaire, du chemin d’accès aux types dont les instances sont à

(1) Le caractère ‘‘#’’ est un indicateur de mots−clés qui permet à l’utilisateur de donner les identificateurs STRUCTURE, ENDSTRUCTURE, GLOBAL, ENGLOBAL et NULL aux types et attributs qu’il définit dans les schémas de structure.

convertir et/ou aux types vers lesquels on les fait évoluer. Il s'agit de faire précéder ces types de la liste de leurs ascendants séparés par le caractère d'indirection '.' (point). L'élément immédiatement à gauche du caractère d'indirection est le père de l'élément qui se trouve immédiatement à sa droite ; en d'autres termes, l'élément immédiatement à gauche du caractère d'indirection est un type dans la définition duquel apparaît l'élément immédiatement à sa droite. Dans les exemples suivants, SuiteParagraphes est un composant de Section et TitreSousSection est un composant de SousSection.

```
Section.SuiteParagraphes
SousSection.TitreSousSection
```

La sémantique d'un élément de couple d'évolution est la limitation du choix des instances à celles dont le type des ascendants apparaît dans l'élément et dans le même ordre. La présence de ces couples particuliers n'empêche ni le comparateur ni l'utilisateur de générer les couples (SuiteParagraphes SuiteParagraphes) et (SousSection #NULL).

- À un même type du nouveau schéma de structure peut correspondre plus d'un type de l'ancien schéma de structure. Soit les types Section et SousSection (voir figure Fig. 5.2) supposés appartenir à l'ancien schéma de structure. Soit le type Section~ (voir figure Fig. 5.3) du nouveau schéma de structure vers lequel on souhaite faire évoluer les instances de type Section. Le couple d'évolution suivant est automatiquement généré par le comparateur :

  ```
  (Section  Section~)
  ```

  Dans l'hypothèse de la conversion des instances de type SousSection en instances de type Section~, l'utilisateur doit fournir les couples d'évolution suivants :

  ```
  (SousSection  Section~)
  (TitreSousSection  TitreSection)
  ```

- À un attribut de l'ancien type correspond au plus un seul attribut du nouveau.
- À un attribut du nouveau type correspond au plus un seul attribut de l'ancien type. La raison est qu'à un élément n'est associée au plus qu'une occurrence d'un attribut donné. Il est donc inutile de produire plus d'une fois le même attribut nouveau en l'associant à plus d'un attribut ancien dans plusieurs couples d'attributs.

```
Section = BEGIN
              TitreSection  =  TEXTE;
               SuiteParagraphes;
               SousSection;
            END;

SousSection = BEGIN
                  TitreSousSection  =  TEXTE;
                   SuiteParagraphes;
              END;
```

Fig. 5.2 : Exemples d'anciens types

```
Section~ = BEGIN
              TitreSection  =  TEXTE;
               SuitePAragraphes;
            END;
```

Fig. 5.3 : Exemples de nouveaux types

Remarques

- Il existe un fichier de correspondance par couple de schémas comparés.
- Un fichier de correspondance comporte les deux parties distinctes suivantes :
    1. correspondance des types et de leurs attributs locaux ;
    2. correspondance des attributs globaux des deux schémas.

V.3.1.1 Couples de types et couples d'attributs locaux

Cette partie présente la déclaration des couples d'évolution de types et de leurs attributs locaux. Une déclaration de couples commence par le mot−clé **#STRUCTURE** et s'étend jusqu'au mot−clé **#ENDSTRUCTURE**. Elle comporte au moins un couple d'évolution obligatoire qui indique les deux schémas de structure à comparer. Dans un couple d'évolution de types, le premier élément désigne l'ancien type, le deuxième élément désigne le nouveau type. Dans la liste des couples d'attributs, le premier élément désigne l'ancien attribut, le deuxième le nouvel attribut. La grammaire de déclaration des couples d'évolution sert à constituer les couples de types et/ou d'attributs dont les éléments n'obéissent pas au principe d'homonymie : la chaîne de caractères représentée par "nouveau type" n'est pas la même que celle représentée par "ancien type". Elle sert également à exprimer les conversions contextuelles en précisant les ascendants des types des couples d'évolution. Elle est expliquée ci−après à travers quelques exemples :

1. Dans certains cas, les types comparés ne sont pas dans les schémas que l'on compare, comme l'illustre l'exemple suivant. La première ligne de l'exemple appartient à l'ancien schéma et la deuxième ligne au nouveau schéma :

    ```
    RefKeyWord  =  REFERENCE(KeyWord);
    RefKeyWord~  =  REFERENCE(KeyWord(SchKeyWord));
    ```

    On cherche ici à transformer les éléments de type KeyWord défini dans l'ancien schéma de structure en éléments de type KeyWord défini dans le schéma de structure SchKeyWord. On a besoin dans ce cas de préciser le schéma de définition de l'élément KeyWord dans le but de récupérer sa définition en vue de la comparaison. Sans cette précision, le comparateur ne trouverait pas le type KeyWord dans le nouveau schéma et générerait le couple d'évolution (KeyWord #NULL) entraînant la destruction des éléments de type KeyWord. L'utilisateur fournit alors le couple d'évolution suivant :

    ```
    (KeyWord   SchKeyWord.KeyWord)
    ```

2. En présence du couple (Section, Section~) le comparateur constate la restriction du type Section et génère une règle de suppression du type SousSection. En l'absence du couple d'évolution pour la conversion des instances de type SousSection, le comparateur génère la règle (SousSection #NULL) entraînant la destruction des instances de type SousSection dont le contenu peut être récupéré en déclarant un couple d'évolution. Lorsque la récupération est totale, les couples d'évolution suivants doivent être déclarés :

   ```
   (SousSection  Section~)
   (TitreSousSection  TitreSection)
   ```

   Le résultat attendu dans ce cas est la conversion des instances de type SousSection en instance de type Section~.

   Lorsque la récupération est partielle, le premier élément du couple d'évolution doit préciser le type des éléments à récupérer, par exemple :

   ```
   SousSection.SuiteParagraphes
   ```

   Lorsque l'élément vers lequel on fait évoluer l'ancien type dépend du contexte, ce dernier doit être précisé, par exemple :

   ```
   Section~.SuiteParagraphes
   ```

   Le résultat attendu est alors l'incorporation de la suite de paragraphes des instances de type SousSection à celle des instances de type Section~.

3. S'il en est besoin, une liste de couple d'attributs est placée après les identificateurs de type dans le couple d'évolution comme dans les exemples suivants :

   ```
   (Liste  Liste~  [Marque_Item  MarqueItem])
   ```

   Dans cet exemple, l'attribut de type Marque_Item d'une liste est transformé en attribut de type MarqueItem.

Une déclaration de couples sert également à changer la valeur d'un attribut, que l'attribut ait été redéfini ou pas, que le type auquel il se rapporte ait été redéfini ou pas. Lorsque la nouvelle valeur porte sur le premier attribut, cela suppose que l'attribut n'a pas été redéfini, par exemple :

```
(Liste  Liste~  [Marque_Item  =  Diamant])
```

Par contre, si la valeur porte sur le deuxième attribut, il faut d'abord transformer l'attribut avant de lui affecter sa nouvelle valeur qui appartient forcément au domaine des valeurs du nouvel attribut, par exemple :

```
(Liste  Liste~  [Marque_Item  MarqueItem  =  Diamant])
```

```
CorrespondanceDeType
    ::=  TypesEtAttributsLocaux  [  AttributsGlobaux  ]

TypesEtAttributsLocaux
    ::= '#STRUCTURE' Règle {Règle} '#ENDSTRUCTURE'

Règle
    ::= '('CouplesTypes  [  Attributs ] ')'

Règle
    ::= '('IdentType  Attributs')'

CoupleTypes
    ::=  IdentType  IdentType

Attributs
    ::= '['  CoupleAttr  {CoupleAttr}  ']'

CoupleAttr
    ::=  IdentAttr  DeuxièmeAttr

DeuxièmeAttr
    ::=  IdentAttr  [ValeurAttr]
      |  ValeurAttr

ValeurAttr
    ::= '='  Valeur

IdentType
    ::= [  IdentSchéma  '.' ] [Accès] Ident

IdentSchéma
    ::=  Ident

Accès
    ::=  Ident'.' {Accès}

IdentAttr
    ::=  Ident

Ident
    ::= Lettre {Lettre  |  Chiffre}

Valeur
    ::= [  '+'  |  '-'  ] Chiffre {Chiffre} | Ident

AttributsGlobaux
      ::= '#GLOBAL' Attributs '#ENDGLOBAL'
```

Fig. 5.4 : Langage K : correspondance de types et d'attributs locaux

L'exemple de la figure Fig. 5.5 présente la partie correspondance de types et attributs locaux de l'évolution d'un schéma de structure Chapitre vers un schéma de structure

Article. L'attribut NumeroPremPage de la règle Chapitre a au moins changé d'identificateur et devient PremièrePageArticle. Toutes les instances de type TitreChapitre doivent être transformées en instances de type TitreArticle défini dans le schéma Article et leurs éventuels attributs de type Indexer doivent prendre la valeur Passive, ces attributs n'ayant pas été redéfinis. Les anciens éléments de type Paragraphe (schéma externe) sont convertis en éléments de type ParagrapheThèse (schéma externe).

```
#STRUCTURE
    (Chapitre    Article
        [NumeroPremPage   PremièrePageArticle]
    )

    (TitreChapitre    Article.TitreArticle
        [Indexer  =  Passive]
    )

    (Paragraphe  ParagrapheThèse)
#ENDSTRUCTURE
```

Fig. 5.5 : Exemple de couples d'évolution de types et d'attributs locaux

V.3.1.2 Couple d'attributs globaux

La composante *couple d'attributs globaux* de la *correspondance des types* comporte les couples d'évolution d'attributs globaux des deux schémas comparés. Les attributs globaux n'étant rattachés à aucun type en particulier, elle ne comporte pas de couples d'ancien et nouveau types mais uniquement des couples d'attributs globaux. Comme pour la correspondance de types, elle est décrite par une grammaire formulée comme suit :

```
AttributsGlobaux
      ::= '#GLOBAL' Attributs '#ENDGLOBAL'
```

La dérivation du non terminal Attributs est la même que celle de la figure Fig. 5.4. Un exemple de couple d'attributs, relatif à l'évolution du schéma de structure Chapitre vers le schéma de structure Article est donné ci−après :

```
#GLOBAL
    [A_indexer  Indexer
     Langue  =  Français
    ]
#ENDGLOBAL
```

Dans cet exemple, l'attribut global A_indexer du schéma Chapitre, change au moins d'identificateur et son nouvel identificateur est Indexer. L'attribut Langue prend la valeur Français et n'est pas redéfini.

V.3.2 Correspondance de types et SGML

Dans un environnement SGML, la syntaxe du LINKTYPE (Explicit Link) peut être partiellement utilisée pour exprimer la correspondance entre les types. Ce choix se justifie comme suit :

1. Les responsables de la gestion des documents sont sensés connaître déjà cette syntaxe.
2. Les règles contenues dans un LINKTYPE sont destinées à exprimer aussi la transformation d'éléments d'une DTD donnée en éléments d'une autre DTD.
3. SGML n'adjoint pas de sémantique à la syntaxe, laissant le soin aux applications de faire les interprétations qui leur conviennent.
4. Toute la richesse des tests réalisables sur les attributs permettront au convertisseur d'effectuer des traitements spécifiques sur les éléments. En effet, les valeurs des attributs peuvent être testées, changées, etc.

Nous rappelons qu'il s'agit uniquement de l'utilisation de la syntaxe pour exprimer la correspondance. L'interprétation suivante est faite par le comparateur des règles, exprimées dans la syntaxe du LINKTYPE. Dans l'exemple de la figure Fig. 5.6, Chapitre_Article est le nom de l'ensemble des couples de types dont les éléments sont issus des schémas Chapitre et Article. La règle <!LINK #INITIAL regroupe tous les couples de types. Chaque type est suivi entre crochets de la liste des attributs à comparer. Un attribut de l'ancien type est comparé à l'attribut de même indice du nouveau type. Les deux types #Chapitre et #Article désignent non pas les types Chapitre et Article mais les schémas de structure Chapitre et Article, permettant ainsi la comparaison des attributs globaux. Le lecteur intéressé est invité à consulter [Goldfarb90].

```
<!LINKTYPE Chapitre_Article Chapitre Article [
    <!LINK #INITIAL
        Chapitre [NumeroPremPage]
        Article [PremièrePageArticle]
        TitreChapitre
        Article.TitreArticle [Indexer = Passive]
        Paragraphe ParagrapheThèse
        #Chapitre [A_indexer]
        #Article [Indexer Langue = Français]
    >
]>
```

Fig. 5.6 : Correspondance de types exprimée avec le LINKTYPE

Remarque

Le modèle conceptuel présenté dans ce chapitre est similaire au modèle de transformation proposé par DSSSL, à une différence importante près. Le module ‘‘Association Specification’’ de DSSSL, l'équivalent de la correspondance de types proposée dans notre schéma, associe un élément d'instance de document à un élément du résultat de la transformation (Output Instance) ou à un type du nouveau schéma (Output Definition). Dans le modèle proposé, le processus de comparaison a besoin d'une association, non pas entre instances, mais entre éléments génériques, puisque notre intention est de transformer toutes les instances de documents quelles que soient leurs compositions.

V.4 Règles de transformation

Étant donné un couple d'évolution, une règle de transformation est un ensemble d'informations et d'actions utilisées par le convertisseur pour effectuer la transformation des instances de l'ancien type de manière à les rendre compatibles au nouveau type. Il existe deux types de règles de transformation, les règles de transformation d'attributs et les règles de transformation des éléments qui portent ces attributs. Une règle de transformation comporte deux parties, un préfixe dont la composition est commune à toutes les règles et un suffixe optionnel qui dépend de l'évolution constatée. Il est important de noter que les règles de transformation sont entièrement produites par le comparateur et ne sont utilisées que par le convertisseur. La forme sous laquelle ces règles sont gardées a peu d'importance pour le principe de la conversion elle−même. C'est pour cette raison qu'aucune description n'en est faite et que nous nous contentons d'expliquer sa constitution.

La nature d'une règle dépend de la différence constatée entre les éléments du couple d'évolution. À chaque sorte d'évolution correspond un mot−clé qui renseigne le convertisseur sur la nature du traitement à effectuer. L'ensemble des mots−clés disponibles est présenté ci−après :

AJOUTER	extension,
ENLEVER	restriction de types ou disparition d'attributs,
DIMINUER	diminution du nombre maximum d'éléments de liste,
AUGMENTER	augmentation du nombre minimum d'éléments de liste,
DÉPLACER	changement de rang,
CONSTRUIRE	changement de constructeur,
OBLIGER	obligation d'occurrence,
LIER	changement du type d'élément référencé,
INCLURE	retrait d'inclusion,
EXCLURE	exclusion d'éléments,
RESTAURER	restauration d'éléments exclus,
ADOPTERASSOC	adoption d'un type associé par un type principal,
ADOPTERPRINC	adoption d'un type principal par un type associé,
DIVISER	l'élément référencé demeure référencé mais il est désormais défini dans un schéma de structure autre que celui de sa référence,
INTÉGRER	l'élément référencé demeure référencé mais il est désormais défini dans le même schéma de structure que sa référence,
FIXER	l'attribut doit prendre une valeur fixe,
IDENTIFIER	changement d'identificateur de type ou d'attribut.

Le préfixe d'une règle de transformation comporte les éléments suivants : un mot−clé rapportant l'évolution constatée, l'identificateur de l'ancien type et l'identificateur du nouveau type. Si la règle concerne un couple d'attributs, elle comporte en plus, l'identificateur de l'ancien attribut et l'identificateur du nouvel attribut. Le suffixe d'une règle de transformation dépend de l'évolution constatée. En effet, si l'évolution est une disparition d'attribut, le suffixe indique le numéro de l'attribut à supprimer. Par contre s'il s'agit d'un changement d'élément référencé, le suffixe comporte l'ancien élément référencé, le schéma de structure de l'ancien élément référencé, le nouvel élément référencé et le schéma de structure du nouvel élément référencé.

V.5 Conclusion

Ce chapitre a présenté le schéma conceptuel de transformation de documents retenu et ses composants, le comparateur, le convertisseur et la correspondance de types. Le fait que les éléments des couples d'évolution soient des types, suppose que leurs caractéristiques soient représentées pour permettre l'action du comparateur. C'est ce formalisme des caractéristiques des types qui fait l'objet du chapitre VI.

Chapitre VI

Modèle de type

Résumé :

Ce chapitre présente un modèle restreint de types destiné à exprimer les transformations élémentaires et complexes, qu'elles soient statiques ou dynamiques. En plus de l'expression des transformations élémentaires, le modèle permet d'établir des relations d'ordre et d'équivalence entre des types qui composent un schéma de structure. Ces relations permettent ainsi d'identifier un certain nombre de transformations, uniquement par l'analyse statique des schémas de structure.

La section VI.1 fixe le cadre dans lequel la modélisation est faite. La section VI.2 présente des définitions et conditions préliminaires à la modélisation des types présentée dans la section VI.3. La section VI.4 met en œuvre le modèle précédemment défini pour exprimer les transformations élémentaires. L'objet de la section VI.5 est d'extraire les sémantiques possibles sous−jacentes aux requêtes de transformation exprimées par les utilisateurs. Les conclusions de cette analyse conduisent aux extensions présentées dans les sections VI.6 et VI.7. Ces sections élargissent le modèle aux requêtes de transformations dynamiques en proposant respectivement des extensions d'ordre structural et basées sur le contenu. La section 8 donne les algorithmes de reconnaissance des structures d'extension structurale.

VI.1 Cadre de la modélisation

Le chapitre précédent a conduit à l'identification des transformations susceptibles d'intervenir dans un document. Le chapitre courant présente un modèle mathématique de types pour les documents structurés, afin d'aider à la réalisation des transformations. Ce modèle est fondé sur les éléments suivants :

- L'ensemble β des types de base : β = {TEXTE, GRAPHIQUE, IMAGE, SYMBOLE, REFERENCE, ε}.
- L'ensemble Φ des types définis dans un schéma de structure : un type structuré est un type qui n'est pas un type de base ; il est défini au moyen d'autres types et d'un

constructeur. Soit C = {Agrégat ordonné, Agrégat sans ordre, Liste, Référence, Choix, Identité}, l'ensemble des constructeurs disponibles dans le système.

- L'ensemble des schémas de structure du système.

Définition

On appelle type structuré tout type défini dans un schéma de structure.

La transformation structurale et la récupération des contenus sont les deux objectifs pris en compte pour la conception du modèle. En réalité, ces deux objectifs ne sont pas disjoints, même si dans la pratique on attribue à l'un plutôt qu'à l'autre une priorité plus ou moins grande en fonction des résultats souhaités :

Priorité à la structure Lorsque la conservation de la structure est le critère de plus grande priorité, une analyse aussi poussée que possible de la structure des types est souhaitée. La récupération du contenu se fait seulement après, même s'il doit y avoir perte d'information. Cette situation propre aux transformations statiques et dynamiques, n'exclut pas la possibilité d'exprimer explicitement la récupération des éléments menacés de disparition.

Priorité au contenu En revanche, si la récupération du contenu est le critère de plus grande priorité, l'analyse de la structure se révèle secondaire (mais non absente) au bénéfice d'une prise en compte prioritaire de la nature des contenus. Les opérations de couper−coller sont particulièrement concernées par ce scénario.

La définition du modèle va se faire en deux étapes :

1. La représentation fonctionnelle des arbres dont l'objectif est d'exhiber la caractéristique arborescente des types.
2. La définition fonctionnelle des caractéristiques structurales des types.

VI.2 Représentation fonctionnelle des arbres

De manière intuitive, nous considérons que les types ont une structure arborescente. Un rapprochement informel avec les éléments arborescents de documents permet de voir les types comme la forme arborescente générique servant à la création des éléments de documents.

Dans cette section, nous introduisons une définition fonctionnelle des arbres. Elle servira à illustrer la nature arborescente des types et des instances de documents.

VI.2.1 Définitions

Cette partie présente une définition fonctionnelle des arbres, des massifs d'arbres et de l'isomorphisme d'arbres [Frilley89]. Ces notions essentielles servent par la suite comme matériel de base dans la définition des relations entre types, relations nécessaires à l'élaboration des solutions aux problèmes de restructurations.

VI.2.1.1 Arbres

Un arbre est un couple (E, p) où E est un ensemble muni d'un élément distingué ρ, et p une application de E dans E exprimant une précédence telle que ρ vérifie les conditions suivantes :

1. $p(\rho) = \rho$

   ρ est la racine(2) de l'arbre et le point fixe de la fonction p.

2. $\forall x \in E, \exists k \in N \mid p^k(x) = \rho$

Un arbre est un graphe connexe sans cycle, orienté par la fonction de précédence. La terminologie et les définitions suivantes sont utilisées dans la suite de l'exposé.

y est le père de x	$y = p(x)$ et $y \neq x$
x est le fils de y	par la définition précédente
les fils de y	$fils(y) = \{x \in E \mid y = p(x)\}$
les ascendants de x	$A(x) = \{y \in E \mid \exists k \in N , y = p^k(x)\} - \{x\}$
les descendants de y	$d(y) = \{x \in E \mid \exists k \in N , y = p^k(x)\} - \{y\}$
hauteur de x	$h(x) = \min \{k \in N \mid \rho = p^k(x)\}$
hauteur de l'arbre	$H(E) = \sup \{h(x), x \in E\}$
éléments de hauteur k	$E(k) = \{x \in E \mid h(x) = k\}$
arbre ordonné	On appelle arbre ordonné, tout arbre (E, p) de racine ρ, muni d'une relation d'ordre total $\leq_t$ sur E, relation qui vérifie les conditions suivantes : 1. $\forall x, y \in E, x \leq_t y \Rightarrow p(x) \leq_t p(y)$ 2. $\forall x \in E, \rho \leq_t x$

Soit (E, p) un exemple d'arbre (voir Fig. 6.1) dont la définition est donnée ci−après :

E = {A, B, . . , Z}.

p : A = p(B), A = p(L), B = p(C), B = p(F), C = p(D), C = p(E), F = p(G), F = p(H), H = p(I), I = p(J), I = p(K), L = p(M), L = p(T), M = p(N), M = p(P), N = p(O), P = p(Q), Q = p(R), Q = p(S), T = p(U), T = p(X), U = p(V), V = p(W), X = p(Y), Y = p(Z), A = p(A).

---

(2) Dans un souci de clarté, la racine d'un arbre (E, p) est identifiée par la lettre ρ. S'il est besoin de la distinguer d'une autre racine, alors ρ est indicée par la fonction de précédence, en l'occurrence p, par exemple ρ_p.

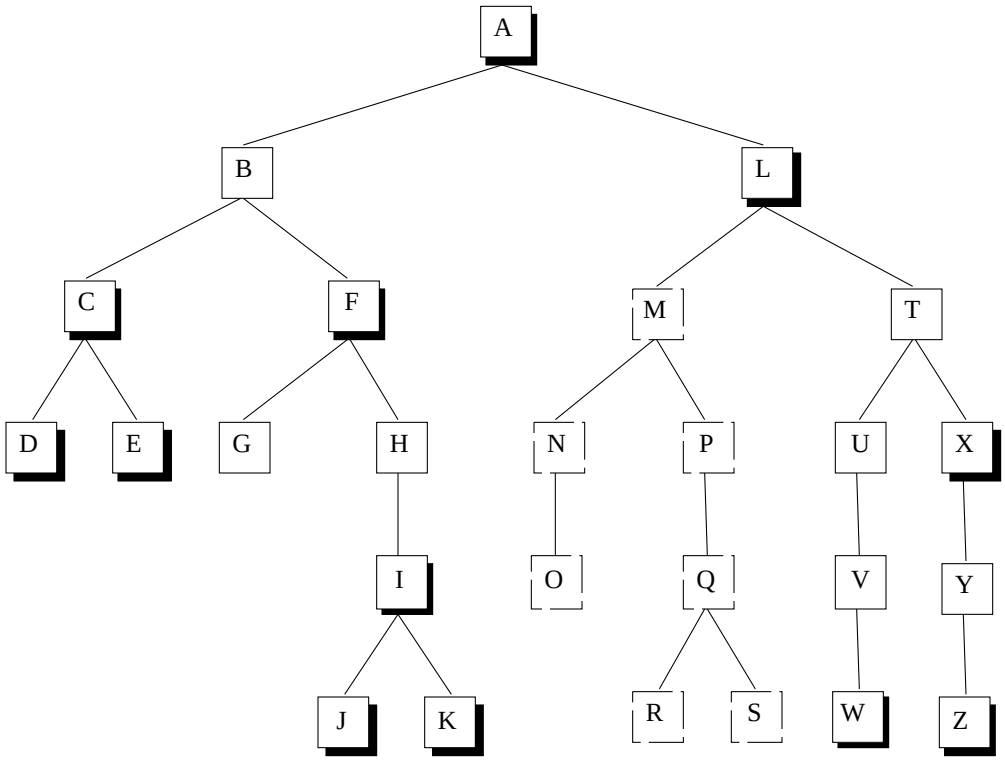


Fig. 6.1 : Arbre (A, p)

VI.2.1.2 Sous−arbres

Soit (E, p) un arbre appelé arbre de référence. On appelle sous−arbre de (E, p), tout couple (S, q) vérifiant les propriétés suivantes :

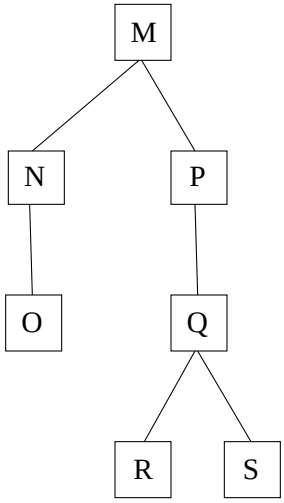
1. (S, q) est un arbre de racine ρ_q.
2. $S \subset E$
3. $\forall x \in S, x \neq \rho_q, p(x) = q(x)$

Soit (S, q) un exemple de sous−arbre tiré de l'arbre de la figure Fig. 6.1 et défini comme suit :

S = {M, N, O, P, Q, R, S}.

q : M = q(N), M = q(P), N = q(O), P = q(Q), Q = q(R), Q = q(S), M = q(M).

Le sous−arbre (S, q) correspond au nœud en pointillé de l'arbre (E, p).



VI.2.1.3 Massifs d'arbres

Étant donné un arbre de référence (E, p), on appelle massif, tout couple (M, s) vérifiant les propriétés suivantes :

1. (M, s) est un arbre de racine ρ_s
2. $M \subset E$
3. $\forall u \in M, u \neq \rho_s, \exists n \in N$, tel que :

   - $s(u) = p^n(u)$:

     le père de tout nœud u d'un massif est un ascendant de u dans l'arbre d'origine.

   - $\forall i \in N, i > 0, p^i(u) \in M \Rightarrow i \geq n$:

     tout ascendant de u dans l'arbre d'origine, qui appartient au massif doit être un ascendant de s(u).

Propriétés

- Tout sous−arbre d'un arbre (E, p) est un massif de cet arbre.
- L'arbre vide est un massif de n'importe quel arbre.

Intuitivement un massif peut être interprété comme un ensemble de sous−arbres d'un même arbre de référence. Il s'agit d'une interprétation approximative fondée sur la caractéristique principale des massifs : la chaîne des ascendants d'un nœud dans le massif peut être incomplète en comparaison de la chaîne des ascendants du même nœud dans l'arbre de référence. L'arbre (M, s) (voir Fig. 6.2) défini comme suit, est un massif de l'arbre (E, p) de la figure Fig. 6.1:

M = {A, C, D, E, F, I, J, K, L, W, X, Z}.

s : A = s(C), A = s(F), A = s(L), C = s(D), C = s(E), F = s(I), I = s(J), I = s(K), L = s(W), L = s(X), X = s(Z).

La chaîne des ascendants du nœud d'étiquette k dans le massif est CM = (I, F, A) alors que dans l'arbre de référence elle est CE = (I, H, F, B, A). L'absence de certains nœuds est

telle qu'un massif plongé dans l'arbre de référence n'est pas connexe mais éclaté en "sous−arbres". Les nœuds H et B de la chaîne CE sont absents de la chaîne CM. Le massif (M, s) correspond au nœud ombrés de l'arbre (E, p).

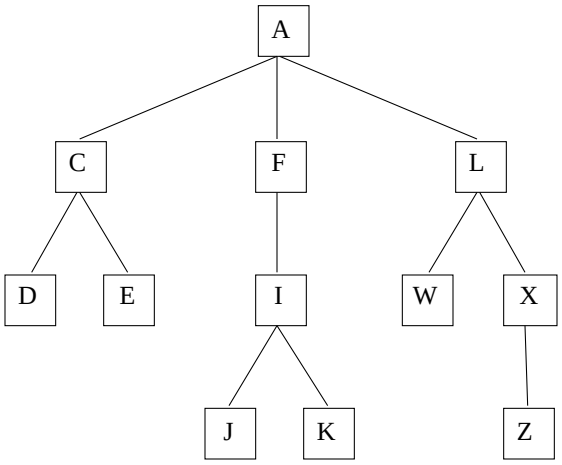


Fig. 6.2 : Arbre (M, s) massif de (E, p)

VI.2.1.4 Isomorphisme d'arbres

Étant donné deux arbres (E, p) et (E', p') et φ une application de E dans E', on dit que φ est un isomorphisme de E dans E' si :

1. φ est bijective
2. $\forall u \in E, \varphi(p(u)) = p'(\varphi(u))$.

VI.2.2 Application aux types

Le modèle de type présenté plus loin est basé sur la représentation arborescente des types, représentation qui suppose qu'un nœud est identifié par un identificateur unique. Afin de représenter la structure arborescente complète des types, les types qui interviennent plusieurs fois dans le schéma de structure sont renommés avec des identificateurs qui n'appartiennent pas déjà au schéma de structure. Le schéma de structure ainsi modifié est sous une forme dite *canonique*. La mise sous forme canonique est une opération qui se fait en deux étapes compte tenu des particularités du langage S. Il s'agit d'une opération d'étiquetage, cachée de l'utilisateur.

VI.2.2.1 Attribution d'identificateurs aux constructions

On rappelle que Φ est l'ensemble des types définis dans un schéma de structure. Le système Grif suppose que les définitions sont de l'une des formes suivantes :

1. IdentificateurDeType = Construction ;
2. IdentificateurDeType = IdentificateurDeType ;
3. IdentificateurDeType ;

La troisième forme de définition suppose que IdentificateurDeType représente, soit un type déjà construit, soit un type de base et que l'on déclare un type qui doit avoir la même définition que celle du type représenté par IdentificateurDeType. Le Type ainsi défini n'a pas d'identificateur et ceci n'est pas commode dans la mesure où pour être comparés, les types doivent être identifiés. Dans la définition suivante du type Petit, la première invocation du type de base *TEXTE* ne correspond pas à un type construit, alors que dans celle du type Grand il lui correspond le type construit Prénom :

```
Petit = BEGIN                    Grand = BEGIN
          TEXTE;                         Prénom = TEXTE;
         Nom = TEXTE;                    Nom = TEXTE;
        END;                               END;
```

Une première opération consiste à attribuer un identificateur aux définitions qui n'en ont pas. Cette opération transforme l'ensemble Φ des définitions de départ en un ensemble Γ dans lequel toutes les définitions sont de la forme IdentificateurDeType = PartieDroite, telle que chaque définition ait un identificateur. Le type Petit, une fois complété, aura une définition semblable à la suivante :

```
Petit = BEGIN
          Petit.TEXTE.0 = TEXTE;
          Nom = TEXTE;
        END;
```

Une deuxième opération consiste à construire l'arborescence correspondant à chaque construction c'est−à−dire à chaque type de Γ. La nouvelle forme ainsi obtenue est la forme canonique appelée T. Une relation est établie entre les éléments de Γ et ceux de T (voir VI.2.2.3).

VI.2.2.2 Identificateur

Un type est repéré par son identificateur unique dans le schéma de structure. Cet identificateur désigne en fait la partie droite de sa définition c'est−à−dire une construction en S qui précise sa composition et ses attributs.

VI.2.2.3 Origine d'un type

Une application surjective **org** associe à chaque type de T le type de Γ qui a servi de modèle à sa définition : **org : T** $\rightarrow \Gamma$.

VI.2.2.4 Construction de la forme canonique

Une autre facilité du langage S est l'utilisation des identificateurs de type comme paramètres dans la définition d'autres types. Dans ce cas, les définitions sont de la forme IdentificateurDeType = AutreIdentificateurDeType, c'est−à−dire la partie droite de la définition est un identificateur de type déjà défini ou un identificateur de type de base. Ce langage offre également une possibilité de définition récursive qui se traduit par l'utilisation de l'identificateur du type en cours de définition dans la définition de ses composants. La forme canonique de la définition d'un type est une forme qui ne fait pas usage d'identificateurs de types déjà définis ; en d'autres termes, c'est une forme dans laquelle les identificateurs de types qui apparaissent dans la partie droite des définitions sont soit inédits, soit des identificateurs de types de base. Le but de l'opération est d'associer à chaque type une arborescence propre

dont les nœuds pourront être désignés sans ambiguïté. Une telle opération consiste à ajouter à Γ un ensemble de types inédits tels que la transformée de Γ est T, la *forme canonique* de Φ. Pour atteindre cet objectif il faut associer à chaque invocation d'identificateurs de types déjà définis, une définition complète. Cette opération peut introduire des cycles dans la définition à cause des types récursifs. On note $t \rightarrow fils(t)$ la définition de t par les éléments de fils(t). Un type t est dit récursif si la condition suivante est vérifiée :

$$t \rightarrow fils(t) \text{ et } \exists t' \in fils(t) \text{ tel que } p^{i}(t') = t$$

Dans ces conditions t' est la première invocation récursive de t. Le problème du cycle est important parce que l'objectif final de la mise sous forme canonique est la représentation arborescente des types et un arbre est défini comme un graphe connexe sans cycle. Pour éviter les cycles, les définitions sont réécrites de façon à satisfaire à la contrainte suivante :

$$t \rightarrow fils(t) \text{ et il n'existe pas } t' \in fils(t) \text{ tel que } p^{i}(t') = t.$$

Lorsque dans un schéma de structure, un schéma externe apparaît dans une définition, la définition du schéma externe ne remplace pas son identificateur. Ce choix permet au modèle de respecter la granularité des définitions dans Grif. Par exemple, dans la définition *Formule = Math*, l'identificateur du schéma externe Math reste tel quel. Tout se passe comme si l'ensemble des types de base du système incluait également les schémas externes.

Pour rendre canonique un schéma de structure, il faut lui appliquer l'algorithme suivant :

```
FormeCanonique(def,  T)

SI (def == invocation récursive)
ALORS
    Donner l'identificateur inédit u à def
    T = T ∪ {u}
    org(u) = def
    Retour
FINSI
SI (def == Type de base)
ALORS
    SI (Constructeur(Pere(def)) = Identité)
    ALORS
        Retour
    SINON
        Mettre def sous la forme x = def
        org(x) = x
        T = T ∪ x
        Retour
    FINSI
FINSI
SI (def == Schéma externe)
ALORS
    SI (Constructeur(Pere(def)) = Identité)
    ALORS
        Retour
    SINON
```

```
        Mettre def sous la forme x = def
        org(x) = x
        T = T ∪ x
        Retour
    FINSI
FINSI
/* def est de la forme t = PartieDroite) */
SI (t est un identificateur existant)
ALORS
    Remplacer t par un identificateur nouveau t'
    org(t') = t
    T = T ∪ t'
FINSI
SI (PartieDroite estun type construit w)
ALORS
    Remplacer w par sa partie droite dans T
FINSI
pfils = PremierFils(t)
TANTQUE(pfils)
    SI (n'est pas de la forme u = PartieDroite)
    ALORS
        Mettre pfils sous la forme v = pfils
        org(v) = pfils
        FormeCanonique(v,  T)
    SINON
        FormeCanonique(pfils,  T)
    FINSI
    pfils = Suivant(pfils)
FINTANQUE
```

Conséquence

Cette opération a permis la construction de l'ensemble T dans lequel les éléments sont individuellement identifiables. La définition formelle des arborescences telle que présentée en VI.2.1.1 est applicable alors aux schémas de structure à travers leurs formes canoniques T. L'application $f : T \rightarrow \wp(T)$ de l'ensemble canonique T des types dans $\wp(T)$, associe à chaque type $t \in T$ l'ensemble des types qui servent à le définir au premier niveau : $fils(t) \in \wp(T)$ tel que $\forall u \in fils(t)$, u sert à définir t. Les avantages d'une telle modélisation sont les suivants :

- associer à chaque type l'ensemble des types qui servent à le définir ;
- caractériser les types de base : en effet, $t \in \beta \Leftrightarrow fils(t) = \varnothing$;
- définir des classes d'équivalence dans T, ainsi que des relations entre les éléments de T. En effet, une requête de restructuration dynamique peut être facilitée par la nature de la relation qui lie les éléments du couple d'évolution ; par exemple, une appartenance à la même classe d'équivalence garantit le succès de la transformation. La relation d'équivalence $t R_f u \Leftrightarrow fils(t) = fils(u)$ induite par la fonction fils exprime le fait que t et u sont définis avec les mêmes types au premier niveau.

- D'une façon plus générale, cette fonction va servir dans la définition de relations entre les types pour permettre des transformations plus élaborées.

Définition

On appelle fils d'un type t, tout type appartenant à fils(t), c'est−à−dire les types qui apparaissent dans la définition syntaxique de t.

VI.2.2.5 Exemple de schémas de structure classique et canonique

Bien qu'elle soit incomplète, la définition du schéma de structure Rapport de la figure Fig. 6.3 contient les éléments nécessaires pour rendre compte du passage d'une forme classique à une forme canonique (Fig. 6.4). Cet exemple servira pour illustrer les notions présentées dans la suite de ce chapitre.

```
STRUCT
   Rapport = BEGIN
               Titre = Texte;
               Corps = LIST [2 .. *] OF (Section);
               ?Esquisse = Section;
             END;
   Section = BEGIN
                TEXTE;
               CorpsSection = LIST OF (Paragr);
             END;
   Paragr = CASE OF
                Para = Paragraphe;
                 Photo = IMAGE;
                 Formule = Math;
            END;
END
```

Fig. 6.3 : Forme classique du schéma de structure Rapport

```
STRUCT
   Rapport =  BEGIN
               Titre  =  TEXTE;
               Corps  =  LIST  [2 . . *]  OF  (Section);
                ?Esquisse;
              END;
   Section  =  BEGIN
                  TitreSection  =  TEXTE;
                  CorpsSection  =  LIST  OF
                    (Paragr  =  CASE  OF
                                  Para   =   Paragraphe;
                                    Photo = IMAGE;
                                    Formule = Math;
                                END;);
               END;
   Esquisse  =  BEGIN
                 TitreEsquisse  =  TEXTE;
                 CorpsEsquisse  =  LIST  OF
                      (ParagrEsquisse   =
                       CASE   OF
                          ParaEsquisse   =   Paragraphe;
                            PhotoEsquisse = IMAGE;
                            FormuleEsquisse = Math;
                        END;);
                END;
END
```

Fig. 6.4 : Forme canonique du schéma de structure Rapport

VI.2.3 Représentation arborescente des types

La représentation arborescente des types de la forme canonique fait usage de la définition donnée en VI.2.2. Il existe une différence entre la représentation des types et celle des instances de documents comme on le verra dans les exemples ci−après. Un type est une structure générique qui représente un ensemble d'instances et qui est susceptible de comporter dans sa définition tous les éléments permettant d'exprimer cette généricité, c'est−à−dire les choix, les alternatives, les répétitions. Ces informations apparaîtront dans les étiquettes des nœuds des arbres d'instance et de type comme suit :

- **{ }** La paire d'accolades exprime le constructeur Agrégat ordonné.
- **[ ]** La paire de crochets exprime le constructeur Choix.
- **( )** La paire de parenthèses exprime le constructeur Liste.
- ↑ ↓ La paire de flèches exprime le constructeur Agrégat sans ordre.

Avertissement

- Dans la suite de l'exposé, les identificateurs des nœuds dans l'arbre des types sont issus de la forme canonique de leur schéma de structure.
- Les nœuds encadrés en pointillé correspondent aux schémas de structure externes dont les arborescences ne sont pas développées par souci de clarté. Normalement, ils ne font pas partie de l'arbre des types, leur présence ayant uniquement pour objectif d'aider à la compréhension des arbres.
- Les appels récursifs d'un type gardent l'identificateur du type suffixé par un numéro incrémental.

Types de base

Les types de base ne sont pas représentés dans l'arbre des types construits. Dans l'arbre des instances, par contre, les feuilles sont uniquement faites de types de base.

Nature des feuilles

Une feuille de l'arbre des types est un élément $t \in T$ qui vérifie l'une des propriétés suivantes :

- $fils(t) \in \beta$. Il s'agit des types définis comme identiques aux types de base. Étant donné les définitions suivantes, *Titre = TEXTE* et *Photo = IMAGE*, fils(Titre) = {TEXTE} $\subset \beta$ et fils(Photo) = {IMAGE} $\subset \beta$.
- t est un type construit sans descendance. Il s'agit de la première occurrence d'un type récursif dans sa propre définition (voir Fig. 6.9).
- fils(t) est un singleton dont l'élément est un schéma de structure ; par exemple, dans la définition *Formule = Math*, fils(Formule) = {Math}.

Choix

Soit n est le nombre d'options qui apparaissent dans la définition d'un choix. Le nombre réel d'instances qu'il est possible d'engendrer par ce choix est n + 1. L'option supplémentaire est implicite et correspond au nœud de contenu vide, c'est−à−dire au type de base ε. Il est considéré comme toujours présent et son absence doit être comprise comme une facilité d'écriture. Un sous−arbre de type choix devra comporter toutes ses options, y compris l'option implicite. La sémantique associée à un choix autorise une instance de type choix à n'avoir qu'un seul fils dont le type est choisi parmi les options proposées par le choix, y compris l'option relative au nœud de contenu vide. Cette différence est illustrée par les exemples suivants (voir Fig. 6.5 et Fig. 6.6) :

```
Paragr = CASE OF
                Para  =  Paragraphe;
                Photo  =  IMAGE;
                Formule  =  Math;
         END;
```

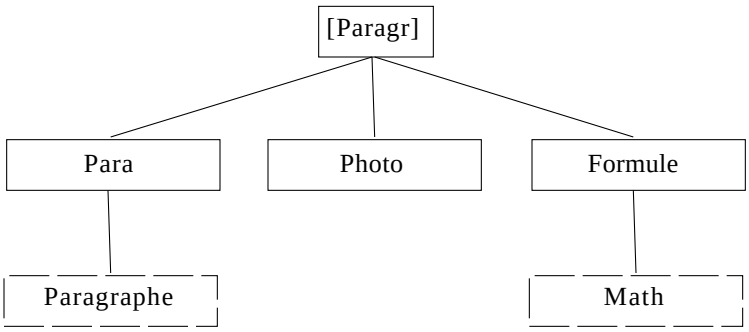


Fig. 6.5 : Représentation arborescente du **type** *Paragr*

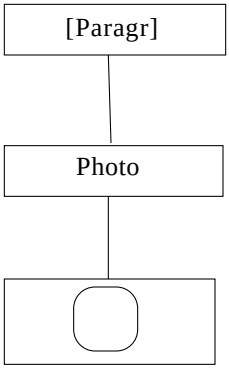


Fig. 6.6 : Représentation arborescente d'une **instance** *de type Paragr*

Agrégat

Un type dont le constructeur est Agrégat ordonné ou Agrégat sans ordre peut comporter des éléments optionnels qui apparaissent obligatoirement dans l'arbre de son type. En revanche, les éléments optionnels n'étant pas obligatoires par définition, ils peuvent ne pas apparaître dans les instances et par conséquent dans les arbres représentant ces instances.

Le type Rapport (voir Fig. 6.4) est un agrégat ordonné dont le composant Esquisse est optionnel. La figure Fig. 6.7 donne une représentation arborescente partielle de ce type. La figure Fig. 6.8 est une représentation arborescente d'une instance de type Rapport qui ne comporte pas d'élément optionnel de type Esquisse.

```
Rapport = BEGIN
            Titre = TEXTE;
            Corps = LIST [2 . . *] OF(Section);
            ?Esquisse = Section;
          END;
```

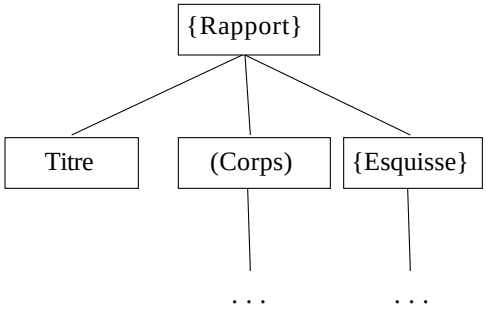


Fig. 6.7 : Représentation arborescente du **type** *Rapport (Agrégat ordonné)*

L'instance suivante de type Rapport ne comporte pas d'élément du type optionnel Esquisse :

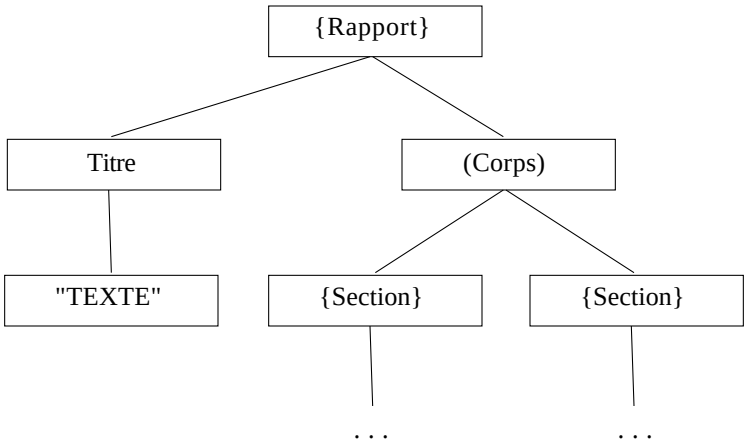


Fig. 6.8 : Représentation arborescente d'une **instance** *de type Rapport*

Types récursifs

Dans un arbre de types, la première invocation récursive d'un ascendant (identité de nom dans la forme classique) est considérée comme une feuille, arrêtant la dérivation de la branche à laquelle appartient l'appel récursif. À l'opposé, la profondeur des instances de type récursif n'étant pas limitée, leurs représentations arborescentes diffèrent de celles de leurs types génériques. Dans l'exemple ci−dessous (voir Fig. 6.9), les appels récursifs terminaux sont en gris.

```
Paragraphe = CASE OF
                SimpleParagraphe;
               ElemListe = LIST   OF(ItemListe);
               CoteACote  =
                  BEGIN
                   Gauche  =  LIST  OF(Paragraphe);
                   Droite  =  LIST  OF(Paragraphe);
                  END;
             END;
SimpleParagraphe = LIST OF(ElementDePara = TEXTE);
ItemListe = LIST OF(Paragraphe);
```

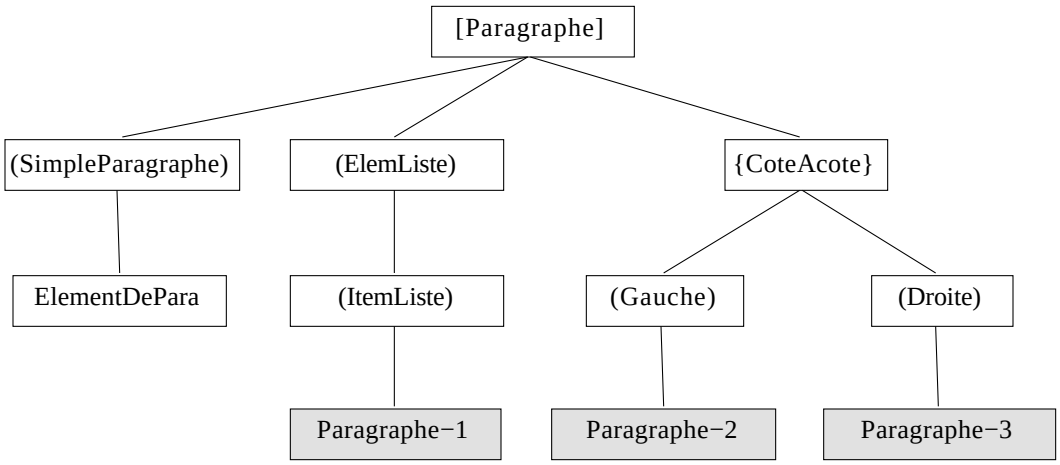


Fig. 6.9 : Représentation arborescente du **type récursif** *Paragraphe*

Liste

Dans un arbre des types, un sous−arbre de constructeur Liste n'a qu'un seul fils comme l'illustre l'arbre de la figure Fig. 6.10.

```
ItemListe = LIST OF(Paragraphe);
```

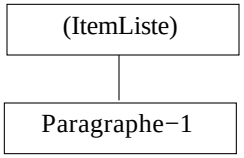


Fig. 6.10 : Représentation arborescente du **type** *ItemListe*

Dans l'arbre correspondant d'une instance, le nombre de descendants d'un nœud de constructeur Liste est égal au nombre de fils effectifs de l'instance. Dans l'exemple de la figure Fig. 6.11, l'instance de type ItemListe a trois fils.

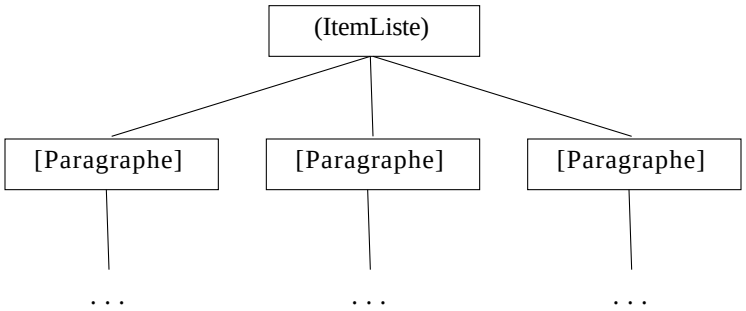


Fig. 6.11 : Représentation arborescente d'une **instance** *de type ItemListe*

VI.3 Définition fonctionnelle des caractéristiques

À ce niveau de la modélisation, il est loisible de considérer les types comme des arborescences à travers toutes les fonctions qui caractérisent les arbres. Cependant, la représentation arborescente des types n'exhibe que partiellement les relations qui peuvent unir deux types. En effet, un type défini dans un schéma de structure a des caractéristiques autres que les relations qu'il entretient avec certains autres types. C'est pour rendre compte de ces caractéristiques que la définition fonctionnelle est introduite. C'est une extension de la représentation arborescente, fondée sur la théorie des ensembles et les représentations fonctionnelles.

Pour un type donné, on définit une fonction pour représenter chacune de ses caractéristiques (ces fonctions sont détaillées dans les sections suivantes) :

1. son identificateur,
2. son origine,
3. ses fils,
4. son constructeur,
5. ses exclusions,
6. ses inclusions,
7. son cardinal,
8. le rang de ses fils,
9. ses fils optionnels,
10. sa famille.

VI.3.1 Origine d'un type

L'origine d'un type appartenant au schéma de structure canonique est le type construit qui correspond à sa partie droite dans la forme non canonique complétée Γ du même schéma. Par exemple selon la définition suivante tirée de la figure Fig. 6.3, l'origine du type Esquisse est Section.

```
?Esquisse = Section;
```

Lorsque dans une forme non canonique une définition se résume uniquement à l'invocation d'un type construit, la mise sous forme canonique attribue un identificateur inédit à cette définition et développe sa partie droite. La fonction org (voir VI.2.2.3) dans ce cas établit un lien avec le type construit qui a servi de modèle à la nouvelle définition. Son intérêt est de permettre d'établir aisément l'identité de structure des types qui ont la même origine, puisqu'ils ont la même définition donc la même structure d'arbre de type. La définition de la fonction org est rappelée ci−après : $\mathbf{org : T \rightarrow \Gamma}$.

VI.3.2 Fils d'un type

La fonction fils permet de représenter les fils d'un type. Elle garde la même signification que dans la définition fonctionnelle des arbres (voir VI.2.1.1) :

fils : T → þ(T).

VI.3.3 Constructeur

À chaque élément de l'ensemble T on fait correspondre son constructeur. Un constructeur est représenté par la fonction **constr : T → C** où C = {Agrégat ordonné, Agrégat sans ordre, Liste, Référence, Choix, Identité}.

Les exemples suivants sont tirés de la figure Fig. 6.4.

```
constr(Section} = Agrégat   ordonné.
constr(Paragr} = Choix.
constr(CorpsSection} = Liste.
```

VI.3.4 Exclusions

Les exclusions sont représentées par l'application **excl : T → þ(T)** telle que excl(t) = {u | u ∈ d(t)} est l'ensemble des types dont les éléments ne pourront pas appartenir à l'arborescence des instances de type t.

En réalité, des éléments autres que ceux appartenant à excl(t) sont susceptibles d'être exclus des instances de type t ; ce sont ceux hérités des règles d'exclusion des ascendants. L'ensemble $\chi(t)$ des exclusions locales de t et de toutes les exclusions héritées de ses ascendants s'écrit de la façon suivante (1) :

$$\chi(t) = \mathrm{excl}(t) \cup \left(\bigcup_{k=1}^{h(t)} \mathrm{excl}(p^k(t)) \right) \qquad (1)$$

Chaque type fait partie d'un environnement défini comme l'ensemble de ses ascendants, de ses descendants et de ses frères. Les ascendants ont une influence sur les instances d'un type, ceci par le truchement de leurs exclusions locales. Ainsi, les types principaux peuvent hériter des exclusions de leurs ascendants.

En revanche, l'environnement ascendant des types associés est un ensemble vide. Il en découle que les éléments associés n'ont pas d'exclusions héritées. Pour de pareils types, $\chi(t)$ = excl(t).

VI.3.5 Inclusions

Les inclusions sont représentées par l'application **incl : t → þ({T \ {t} ∪ fils(t)})** telle que incl(t) est l'ensemble des types dont les instances sont autorisées dans l'arborescence des instances de type t. Les fils directs de t c'est−à−dire fils(t) et t lui−même ne peuvent pas faire partie de la liste des inclusions ; ceci se traduit par T \ {t} ∪ fils(t).

$\Psi(t)$ est l'ensemble des inclusions locales de t, augmentées de celles de ses ascendants. Les instances engendrées par les éléments de l'ensemble $\Psi(t)$ (2) sont autorisées dans toute l'arborescence des instances de type t :

$$\psi(t) = \mathrm{incl}(t) \cup \left(\bigcup_{k=1}^{h(t)} \mathrm{incl}(p^k(t)) \right) \qquad (2)$$

Comme pour les exclusions, les inclusions réelles d'un type associé se résument à ses inclusions locales telles qu'exprimées dans sa définition : $\Psi(t) = incl(t)$.

Remarque

La caractéristique optionnelle des inclusions et le fait qu'elles n'ont pas de position syntaxique définie conduisent à les traiter séparément. Dans le cadre de l'évolution des types, deux sortes de traitement peuvent être fait sur un type :

- Des traitements qui sont une conséquence de l'évolution de la composante fixe d'un type ; par exemple la restriction, etc.). Ces traitements considèrent les instances des types inclus comme virtuellement absentes, ce qui justifie la définition des fils concrets (voir VI.3.6).
- Des traitements qui portent sur les types inclus ; par exemple, retrait de types inclus, évolution de la définition de type inclus).

VI.3.6 Fils concrets

Étant donné un type t, nous avons donné en VI.2.1 la définition suivante de l'ensemble fils(t) de ses fils directs : $fils(t) = \{u \in E \mid t = p(u)\}$. Une telle définition est dénuée de sémantique puisqu'elle ne prend en compte ni les éléments de $\psi(t)$, ni ceux de $\chi(t)$. En réalité, les éléments apparaissant dans une instance de type t, appartiennent à l'ensemble $fils(u) - \chi(u) \cup \psi(u)$. On peut considérer qu'il existe dans fils(t) des éléments utiles (ceux qui pourront générer des instances) et d'autres qui ne le pourront pas du fait des exclusions locales ou héritées.

Définition

On appelle fils concrets, l'ensemble **$Fils(t) = fils(t) - \chi(t)$** des fils syntaxiques de t, diminué de l'ensemble de ses exclusions locales et héritées. Il s'agit de l'ensemble des fils réels de t, autorisés à produire des instances.

VI.3.7 Descendants concrets

De même que pour fils(t), la définition qui a été faite de d(t) doit être restreinte aux descendants concrets D(t) (3) :

$$D(t) = \bigcup_{i=1}^{|\,d(t)\,|} F_i(t) \qquad (3)$$

VI.3.8 Cardinal d'un type

Le cardinal d'un type t représenté par la fonction **$card : T \rightarrow N$**, définit le nombre maximum d'éléments fils pouvant être engendrés par ce type ; cette quantité intervient particulièrement dans les transformations qui impliquent les types t tels que $constr(t) \in$ {Liste, Agrégat ordonné, Agrégat sans ordre}. Dans l'algorithme ci−dessous de calcul du cardinal, la notation #Fils(t) représente le nombre d'éléments de l'ensemble Fils(t) :

```
SI(Ψ(t) ≠ ∅)
ALORS
    card(t) = ∞ /* Le nombre d'éléments engendrés n'est pas limité */
SINON
    SI (constr(t) ∈ {Agrégat ordonné, Agrégat sans ordre}
    ALORS
        card(t) = #Fils(t)
    SINON
        SI (constr(t) == Liste)
        ALORS
            card(t) est une donnée syntaxique /* comme dans une liste */
        SINON
            SI (constr(t) == Choix
              ||
              constr(t) == Référence
              ||
              constr(t) == Identité
             )
            ALORS
                card(t) = 1
            FINSI
        FINSI
    FINSI
FINSI
```

Les exemples suivants sont tirés de la figure Fig. 6.3.

```
card(CorpsSection} =  ∞.
card(Rapport)  =  3.
card(Paragr)  =  1.
```

VI.3.9 Rang d'un type

La définition qui est faite du rang ne tient pas compte des inclusions qui, rappelons le, n'ont pas de position syntaxique fixe. On appelle rang d'un type fils u, la position de l'instance correspondante parmi les instances dont les types appartiennent à Fils(p(u)). En ce sens, le rang doit être vu comme l'image par une fonction g de l'ensemble des fils auquel appartient u dans l'ensemble N. Cet ensemble des fils n'est calculable que par le père de u et est égal à Fils(p(u)). L'ensemble d'arrivée N est restreint à l'intervalle [1 . . card(p(u))]. La fonction g est ainsi définie :

$g : Fils(p(u)) \rightarrow [1 . . card(p(u))]$

Deux types t_1 et t_2 peuvent être tels que $Fils(t_1)$ soit égal à $Fils(t_2)$ et que les éléments de $Fils(t_1)$ ne soient pas ordonnés de la même façon que ceux de $Fils(t_2)$. Il existe donc pour tous les types qui ont la même image dans Þ(T) un ensemble G de fonctions tel que $\forall g_i \in G$, g_i est l'application propre à t_i. Cette application est obtenue par la bijection suivante : $\forall t \in T$, $rang : T \rightarrow G = \{g = rang(t) \mid rang(t) : Fils(t) \rightarrow [1 . . card(t)]\}$. Le rang d'un type u est alors défini par la fonction de rang comme suit :

rang(p(u)) : Fils(p(u)) → [1 . . card(p(u))]

Cette fonction n'a d'intérêt que pour les types qui ont pour constructeur Agrégat ordonné. Soient u et u' deux composants de deux types p(u) et p(u') de constructeur Agrégat ordonné tels que les couples d'évolution suivants soient formés : (u, u') et (p(u), p(u')). La comparaison de u à u' produira si nécessaire une règle de placement si rang(p(u))(u) ≠ rang(p(u'))(u').

On généralise en faisant correspondre à chaque type t, une fonction de rang rang(t) qui permet de calculer le rang des instances de ses fils $u_i \in$ Fils(t) tel que i ∈ [1 . . #Fils(t)]. Compte tenu des constructeurs, la fonction rang(t) a la signification suivante :

- si constr(t) = Agrégat ordonné,

  alors rang(t) : Fils(t) → [1 . . card(t)].

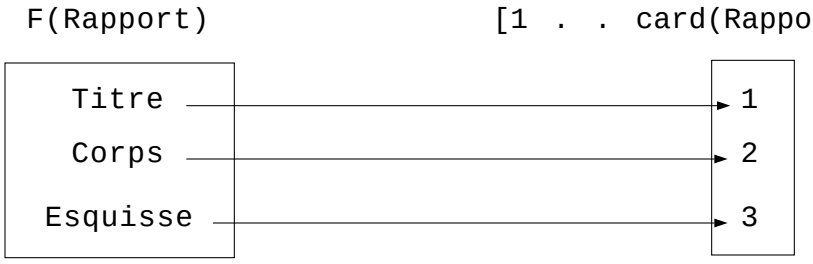
  Dans le cas présent, le rang d'un type fils est sa position syntaxique dans la définition de t, c'est−à−dire sa position dans Fils(t).
- si constr(t) = Agrégat sans ordre, alors les éléments de Fils(t) n'ont pas de rang fixe. Ce problème peut être résolu en attribuant aux éléments de Fils(t) non pas une valeur comme rang, mais un ensemble de valeurs. Dans ce cas, chaque élément de Fils(t) peut prendre pour rang n'importe laquelle des valeurs de l'ensemble, en l'occurrence {1, .., card(t)}. Si cette approche était retenue, l'ensemble des valeurs de rang d'un composant dont le père a pour constructeur Agrégat ordonné serait un singleton.
- si constr(t) = Liste,

  alors rang(t) : Fils(t)→[1. . card(t)]
- constr(t) = Choix,

  alors rang(t) : Fils(t)→[1 . . card(t)] c'est−à−dire [1 . . 1].

  Toutes les instances dont le type est une option sont de rang 1 puisqu'une instance dont le constructeur est un choix ne peut avoir qu'un fils.

A titre d'exemple, étudions le rang des composants du type Rapport défini en Fig. 6.4 :



Remarque

La relation d'ordre induite sur Fils(t) par rang(t) est utilisée plus loin dans le modèle VI.6.3.

VI.3.10 Indicateur d'occurrence

Les éléments de Fils(t) sont caractérisés par le fait qu'ils sont optionnels ou non. Remarquons que deux types t et t' peuvent être tels que Fils(t) = Fils(t') sans qu'ils aient les mêmes éléments optionnels ; pour cette raison, les éléments optionnels sont rattachés à t. La fonction **opt : T →þ(T)** associe à t le sous−ensemble de Fils(t) dont les éléments sont constitués des éléments optionnels t. À titre d'exemple, les ensembles d'éléments optionnels suivants sont tirés de la figure Fig. 6.4.

```
opt(Section) = ∅.
opt(Rapport) = {Esquisse}.
```

VI.3.11 Famille d'un type

Il existe deux familles de types dans un schéma de structure :

- La famille des types principaux regroupe les types déclarés après le mot−clé STRUCT et avant n'importe quel autre mot−clé ou avant la fin du schéma de structure.
- La famille des types associés regroupe les types déclarés après le mot−clé ASSOC et avant n'importe quel autre mot−clé ou avant la fin du schéma de structure. Appartiennent également à cette famille, les types qui sans être des types associés, ne sont pas non plus des types principaux ; ce sont les types directement définis à l'intérieur de la définition des types associés.

Du point de vue de la forme classique des schémas de structure, les types principaux qui apparaissent dans la définition de types associés gardent leur appartenance à la famille des types principaux. Par exemple, le type principal Paragraphe est utilisé dans la définition du type associé Note. Ils sont uniquement utilisés dans la définition des types associés. Du point de vue de la forme canonique, tous ces types sont redéfinis et ont un identificateur unique dans le schéma de structure. Par exemple, l'utilisation du type Paragraphe dans le type Note doit être de la forme *ParaNote = Paragraphe*. Dans ces conditions, l'appartenance d'un type à une seule famille est évidente. Par exemple, le type racine du document appartient uniquement à la famille principale et tous les types associés appartiennent à l'unique famille des types associés.

Parallèlement à cette classification des types, il existe dans une instance de document deux ensembles d'arborescences, une par famille de types :

- L'ensemble principal ne contient qu'une seule arborescence qui représente le corps du document et qui est engendrée par le premier type défini dans le schéma de structure (racine du document) : c'est l'*arbre principal*. Cette arborescence ne contient pas d'éléments de type associés.
- L'ensemble des arborescences d'éléments associés peut contenir autant d'arborescences que de types associés tel que toutes les instances d'un même type associé appartiennent à la même arborescence.

Définition

La famille principale regroupe les types déclarés après le mot−clé STRUCT.

La *famille associée* regroupe les types déclarés après le mot−clé ASSOC.

Le changement de famille, connu sous le nom de migration est une évolution de type qu'il faut prendre en compte. Pour cette raison, on associe à chaque type, la famille à laquelle il appartient : **fam : T → {ASSOC, STRUCT}**. Une famille est représentée par le mot−clé qui la caractérise.

VI.3.12 Définition complète d'un type

En conclusion, toute invocation de t dénotera le n−uplet suivant :

(**org**(t), **fils**(t), **constr**(t), **rang**(t), **card**(t), **excl**(t), **incl**(t), **opt**(t), **fam**(t)).

VI.4 Application du modèle aux transformations

La comparaison de types suppose l'existence de deux ensembles de types (schémas de structure) T et T' et de couples de types (t ∈ T, t' ∈ T') tels que t est le type source et t' le type destination. Une fonction d'association a : T →T' permet de spécifier pour un type t, le type a(t) vers lequel il évolue. Le principe de la constitution des couples est basé sur une caractéristique fondamentale du système : l'hypothèse d'unicité des identificateurs de types par laquelle on suppose que tous les types d'un même schéma de structure ont un identificateur unique dans la dite structure. La forme canonique des schémas de structure nous garantit une telle unicité.

Dans un processus de transformation statique, étant donné deux schémas de structures, le système prend en charge la construction des couples de types homonymes. Toutefois, l'utilisateur est en droit d'ajouter manuellement des couples pour forcer la comparaison de types de noms différents mais probablement de même sémantique. L'ensemble des couples manuellement ajoutés constitue les *règles de renommage*. Les instances générées par des types qui n'auront pas d'équivalents par renommage ou par homonymie seront retirées du document source, à moins qu'ils ne fassent l'objet d'une adoption explicitement exprimée par l'utilisateur. Le triplet (*T, a, T'*) est une ressource dont le comparateur a besoin. Le triplet (T, a, T') est l'abstraction de la correspondance de types du schéma conceptuel présenté dans le chapitre V. Le cadre est bien défini pour exprimer les transformations élémentaires au moyen du modèle présenté en VI.3.

Dans un processus de transformation dynamique, notamment le couper−coller, le couple de types mis en jeu est composé des types des instances source et cible impliquées.

Étant donné le coût élevé du processus de comparaison, il est souhaitable de ne réaliser la comparaison des schémas de structure qu'une seule fois et de produire un fichier de règles de transformation spécifique à un couple de schémas de structure. Ce fichier de règles est réutilisable et la transformation ne mettra en œuvre que le processus spécifique de conversion.

Toutes les transformations qui vont suivre supposent l'existence d'un couple (t, t' = a(t)) dit *couple d'évolution*. Dans cette partie, nous nous servons du modèle pour identifier l'évolution des caractéristiques de types. Soient les types Thésard et Chercheur dont les couples d'évolution (Thésard, Chercheur), (Nom, Nom_R), (Laboratoire, Affiliations), (Sujet, Projet) vont servir pour illustrer l'évolution de la plupart des caractéristiques présentées en VI.3.

```
Thésard = BEGIN
              Photo = IMAGE;
              Nom = TEXTE;
              Laboratoire = Paragraphe;
              ?Sujet = TEXTE;
          END − (Autre) + (DescPublication);
```

Fig. 6.12 : Type Thésard

```
Chercheur = BEGIN
              Prénom = TEXTE;
              Nom_R = TEXTE;
              Adresse  TEXTE;
              Affiliations = LIST OF(Affiliation);
              Projet = LIST OF (Paragraphe);
            END − (SYMBOLE);
Affiliation = Paragraphe;
```

Fig. 6.13 : Type Chercheur

VI.4.1 Restriction

On appelle restriction la disparition d'un composant pendant l'évolution d'un type. Étant donné le couple d'évolution (t, a(t)), on dit que t fait l'objet de restriction si la condition suivante est respectée :

∀u ∈ Fils(t),
  SI(!(∃w ∈ Fils(a(t)) tel que org(w) == org(a(u))))
  ALORS
    SI(!(∃z ∈ Ψ(a(t)) tel que org(a(u)) == org(z)))
    ALORS
      u ne peut pas être fils direct de a(t)
      u ne peut pas être une inclusion dans a(t)
    FINSI
  FINSI

Une comparaison entre Thésard et Chercheur identifie le type Photo comme absent de Fils(Chercheur). Aucune instance de type Photo ne peut être descendante immédiate d'instances de type Chercheur.

La sémantique d'une association étant la comparaison de structure et la récupération du contenu du type source, le comparateur dans un premier temps génère une règle de

restriction rattachée au type Thésard. Cette règle exprime le retrait des instances dont le type appartient à Fils(t) − Fils(a(t)). Dans un deuxième temps, il peut produire des règles d'adoption par a(t), des instances engendrées par les types appartenant aux ensembles Fils(t) − Fils(a(t)) et $\Psi(t) - \Psi(a(t))$. L'utilisateur peut éviter la production de ces règles d'adoption, en fournissant des couples d'évolution contextuels pour chaque élément appartenant aux ensembles Fils(t) − Fils(a(t)) et $\Psi(t) - \Psi(a(t))$.

VI.4.2 Extension

L'extension est l'apparition d'un nouveau fils dans la définition d'un type. Elle est identifiée si la condition suivante est respectée :

$\forall w \in$ Fils(a(t)),
SI(!($\exists u \in$ Fils(t) tel que a(u) $\in$ Fils(a(t)) et org(a(u)) == org(w)))
ALORS
w est un élément nouveau dans a(t)
FINSI

Une règle d'extension est alors générée pour exprimer le fait qu'un élément nouveau apparaît dans a(t).

Considérons les types Thésard et Chercheur tels que définis par les figures Fig. 6.12 et Fig. 6.13. Chaque élément de l'ensemble {Prénom, Adresse} sera identifié comme type nouveau dans a(t) et une instance correspondante devra être incorporée aux instances de type Thésard pour en faire des instances de type Chercheur. L'opération effective d'extension n'est réalisée que si le type nouveau n'est pas optionnel.

VI.4.3 Changement de rang

Étant donnée une instance de type t, le problème est de s'assurer de l'exactitude du rang de sa transformée.

SI(constr(p(a(t))) == Agrégat ordonné)
ALORS
SI (rang(p(t))(t) ≠ rang(p(a(t)))(a(t)))
ALORS
t n'a pas le même rang que sa transformée a(t).
FINSI
FINSI

Une règle de translation exprimera le fait que tout élément de type t devra désormais avoir pour rang rang(p(a(t)))(a(t)). Étant donné le couple d'évolution (Thésard, Chercheur), l'identification des changements de rang présentés ci−dessous est basée sur les fonctions rang(Thésard) et rang(Chercheur) associées aux types Thésard et Chercheur :

- Le rang de Sujet passe de 4 à 5. En effet, rang(Thésard)(Sujet) = 4 alors que rang(Chercheur)(Projet) = 5.
- Le rang de Laboratoire passe de 3 à 4. En effet, rang(Thésard)(Laboratoire) = 3 alors que rang(Chercheur)(Affiliations) = 4.

VI.4.4 Exclusion

On appelle exclusion, l'apparition d'un descendant concret d'un type dans la liste des exclusions de sa nouvelle définition :

```
∀u ∈ D(t)
    ∀u' ∈ χ(a(t))
        SI(org(a(u)) == org(u'))
        ALORS
            une règle d'exclusion de a(u) est produite.
        FINSI
```

Étant donné le couple d'évolution (Thésard, Chercheur), excl(Chercheur) = {SYMBOLE}. Supposons que ∀z ∈ excl(Chercheur) ∃y ∈ D(Thésard) tel que a(y) = z ; c'est le cas dans le couple d'évolution courant puisque {SYMBOLE} ⊂ D(Paragraphe) ⊂ D(Chercheur). Une règle d'exclusion est produite pour chacun des types appartenant à {SYMBOLE}. Dans le cas présent, une règle d'adoption induite n'est pas générée. Ce choix est justifié par le fait que, ∀z ∈ excl(a(t)), les instances de type z sont estimées indésirables dans les instances de type a(t), bien que z appartienne à d(a(t)).

Si malgré tout, une règle d'adoption est produite par l'utilisateur au bénéfice de a(t), la récupération du contenu de l'élément adopté ne devra pas se faire dans un sous−arbre de type z puisque ce dernier est supposé ne pas appartenir à D(a(t)). Il faut mettre en œuvre des techniques adéquates d'adoption consistant à appliquer l'une des méthodes suivantes :

- Rechercher des types équivalents dans D(a(t)) ;
- Éclater le sous−arbre à adopter dans a(t) ;
- Faire passer la récupération du contenu avant la conservation de la structure.

Tous ces points seront étudiés plus loin.

VI.4.5 Restauration d'exclusions

À chaque fois qu'une évolution de type résulte de la disparition d'une exclusion, une règle de restauration d'exclusions est produite. La condition de restauration d'exclusions est donnée ci−après :

```
∀u ∈ χ(t)
    ∀u' ∈ D(a(t)) /* u' ∉ χ(a(t)) */
        SI(org(u') == org(a(u))
        ALORS
            une règle de restauration de a(u) est produite.
        FINSI
```

Cette règle consiste à incorporer dans les instances de type u, les instances de type a(u) partout où elles devraient exister dans a(t). Dans le cas où a(u) ne serait ni explicitement défini comme étant optionnel, ni une option dans un choix, la restauration d'un élément vide est indispensable.

À titre d'exemple, les instances de type Thésard ne contiennent pas d'éléments dont les types appartiennent à excl(Thésard) = {Autre}. En revanche, supposons que ∀z ∈ excl(Thésard) a(z) ∈ D(Chercheur) ; c'est le cas dans le couple d'évolution courant

puisque {Autre} ⊂ D(Paragraphe) ⊂ D(Chercheur). Le comparateur génère dans ce cas une règle de restauration des instances de type a(z) si ces derniers ne sont pas optionnels dans la définition du type Chercheur donnée de la figure Fig. 6.13.

VI.4.6 Retrait d'inclusions

À chaque fois qu'une évolution de type résulte de la disparition d'une inclusion, une règle de retrait d'inclusions est produite. La condition de retrait d'inclusions est donnée ci−après :

∀u ∈ Ψ(t)
SI(a(u) ∉ Fils(a(t)) && !∃u' ∈ Ψ(a(t)) tel que org(a(u)) == org(u'))
ALORS
alors une règle de retrait de u est produite
FINSI

Considérons l'inclusion (DescPublication) de la définition du type Thésard et comparons ce dernier avec le type Chercheur. Il est possible d'inclure des éléments de type DescPublication dans les instances de type Thésard. De tels éléments ne sont pas autorisés dans les instances du type Chercheur puisque DescPublication ∉ Ψ(Chercheur) et que DescPublication ∉ D(Chercheur). Afin de rendre les instances de type Thésard compatibles avec le type Chercheur, le comparateur génère une règle d'exclusion de toutes les anciennes inclusions à présent indésirables, du type DescPublication.

VI.4.7 Changement de constructeur

Étant donné le couple d'évolution (t, a(t)), le changement de constructeur est identifié si la condition suivante est vérifiée :

$$constr(t) \neq constr(a(t))$$

À titre d'exemple, considérons le couple d'évolution (Sujet, Projet) dont les définitions sont rappelées ci−après :

```
Sujet = TEXTE;
Projet = LIST OF (Paragraphe);
```

On remarque la différence suivante des constructeurs : constr(Sujet) = Identité, constr(Projet) = Liste. Le comparateur génère une règle de changement de constructeur pour la conversion de Sujet en Projet.

Il faut noter que dans la plupart des cas, les transformations dues au changement de constructeur ne donnent pas les résultats attendus. La transformation d'un type de constructeur Agrégat ordonné ou Agrégat sans ordre en un type de constructeur Liste et vice versa en est un exemple. Pour illustrer la particularité du problème posé, les éléments suivants sont à prendre en considération lorsque les constructeurs concernés appartiennent à l'ensemble {Agrégat ordonné, Agrégat sans ordre, Liste} :

- ∀u ∈ Fils(t), est−ce que a(u) ∈ Fils(a(t)) ?
- card(a(t)) est−il égal à card(t) ?

- Si constr(a(t)) = Liste, il est souhaitable que constr(fils(a(t))) = Choix.
- Que deviennent les card(t) − card(a(t)) éléments non adoptés ?

VI.4.8 Obligation d'occurrence

L'obligation d'occurrence est identifiée lorsqu'un type optionnel devient obligatoire comme l'illustre la condition suivante :

```
∀u ∈ opt(t)
    SI(u ∈ Fils(t))
    ALORS
        SI(a(u) ∈ Fils(a(t)) && a(u) ∉ opt(a(t)))
        ALORS
            une règle d'obligation d'occurrence est produite.
        FINSI
    FINSI
```

On identifie ainsi les types optionnels qui deviennent obligatoires. Si des instances de ces types sont absentes, elles doivent être créées et insérées au bon endroit, même avec un contenu vide. Considérons l'ensemble des types optionnels de Thésard opt(Thésard) = {Sujet} et celui des types optionnels de Chercheur opt(Chercheur) = ∅. Les instances de type Thésard dépourvues d'éléments de type Sujet seraient complétées avec des éléments vides de type Projet, étant donné le couple d'évolution (Sujet, Projet).

VI.4.9 Diminution de cardinal

La diminution de cardinal est exprimée en tant que transformation élémentaire, généralement pour les couples d'évolution (t, a(t) tel que constr(t) = constr(a(t)). Elle est identifiée si la condition suivante est vérifiée :

$$card(a(t)) < card(t)$$

Elle traduit la diminution du nombre d'éléments autorisés par le type générique. Il conviendrait de retirer (card(t) − card(a(t))) éléments, fils immédiats de l'instance de type t. Dans le cas où les constructeurs appartiennent à l'ensemble {Agrégat ordonné, Agrégat sans ordre}, les (card(t) − card(a(t))) éléments retirés sont identifiés par le mécanisme de restriction et le problème du choix des instances à supprimer ne se pose pas. Il en est de même pour les listes de choix. La comparaison d'un type de constructeur Agrégat ordonné ou Agrégat sans ordre à un type de constructeur Liste ou vice versa, induit également le problème de la cardinalité.

Lorsque **constr**(t) = Choix, t ne peut pas faire l'objet de diminution de cardinal car card(t) = 1.

VI.4.10 Changement de famille (migration)

Étant donné le couple d'évolution (t, a(t)), une migration est détectée si la condition suivante est vérifiée :

$$fam(t) \neq fam(a(t))$$

Une règle de migration est générée pour faire passer les instances concernées de l'arbre principal vers les arbres associés ou vice versa.

VI.5 Sémantique des transformations statiques de types

Les transformations élémentaires peuvent être considérées comme résolues au moyen du modèle. Une procédure est implantée pour chacune des fonctions afin d'identifier les transformations élémentaires. Le plus souvent, l'utilisateur qui demande une transformation souhaite récupérer la totalité de l'information contenue dans les instances candidates à la transformation. Le modèle de représentation présenté ci−dessus décrit certes la structure des types mais n'en fait qu'un usage limité pour la comparaison des types. Cette situation peut conduire à une récupération partielle du contenu du sous−arbre transformé. Pour répondre à ces problèmes, il convient d'une part d'exploiter au mieux la structure des types, d'autre part de comprendre, autant que possible, la sémantique associée par l'utilisateur à une requête de transformation, dans le but de proposer des extensions au modèle de description de structure.

VI.5.1 Les intentions de l'utilisateur

L'application des transformations élémentaires peut conduire à la disparition partielle d'éléments appartenant aux instances à transformer ; ce sont les instances engendrées par les types de l'ensemble Fils(t) − Fils(a(t)). Ces instances ne sont pas forcément destinées à être détruites. Leur sort dépend de l'utilisateur :

1. Elles peuvent être supprimées.
2. Leur contenu peut être récupéré si l'utilisateur fournit des *règles d'adoption explicites* qui doivent être comprises comme un transfert de contenu.
3. L'utilisateur peut fournir des couples d'évolution contextuels pour leur conversion.

Lorsque les intentions de l'utilisateur ne sont pas précisées, se pose alors le problème de la détermination de l'attitude du comparateur. Cette situation résulte de l'automatisation du processus de comparaison. La stratégie la plus facile pour le comparateur est alors la suppression des instances. En revanche, le transfert de contenu (par production de *règles d'adoption induites*) et la conversion placent le comparateur dans une situation d'indétermination :

1. Comment choisir le type des instances sensées adopter les contenus transférés ?
2. Comment choisir le type des couples d'évolution contextuels ?

Soit $t' \in$ Fils(t) − Fils(a(t)). Les questions générales suivantes se posent :

- Existe−t−il un élément $u \in D(a(t)) \cup \{t\}$, susceptible de permettre la transformation induite ?
- Si cet élément existe, comment l'identifier ?
- Existe−t−il d'autres éléments présentant des caractéristiques voisines de celles de t' ?

On se rend ainsi compte que la transformation ne doit pas reposer sur des règles d'adoption induites étant donné le caractère aléatoire des résultats. Il est souhaitable de définir sur l'ensemble des types, un certain nombre de relations ayant pour objectif de faciliter le choix des types transformateurs non exprimés clairement.

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. Étant donné une requête de transformation (t, a(t)), les réponses aux questions suivantes faciliteront le choix des stratégies et des structures adéquates :

1. Tous les types membres de D(t) devront−ils appartenir à D(a(t)) ?
2. L'ordre de ces éléments doit−il être préservé ?
3. Quelle priorité accorde−t−on à la récupération des informations associées aux types de base ?
4. Quel sort l'utilisateur réserve−t−il aux éléments dont le type appartient à Fils(t) − Fils(a(t)) ?
5. Est−il souhaitable de combiner la conservation de la structure et la récupération du contenu ?

VI.5.2 Les limites de l'homonymie

Le fait de baser les restructurations dynamiques sur les couples d'homonymes est responsable de l'échec de certaines transformations. Il convient de comprendre pourquoi afin de proposer des solutions. Dans l'exemple de la figure Fig. 6.14, l'échec de la transformation dynamique d'une instance de type Section en une instance de type Esquisse par un processus basé sur l'homonymie, est due à la différence entre les deux identificateurs.

```
Section = BEGIN
              TitreSection = TEXTE;
              CorpsSection = LIST OF
                (Paragr = CASE OF
                             Para  =  Paragraphe;
                               Photo = IMAGE;
                               Formule = Math;
                            END;);
          END;
Esquisse = BEGIN
             TitreEsquisse = TEXTE;
             CorpsEsquisse = LIST OF
                  (ParagrEsquisse  =
                   CASE  OF
                      ParaEsquisse  =  Paragraphe;
                        PhotoEsquisse = IMAGE;
                        FormuleEsquisse = Math;
                     END;);
            END;
```

Fig. 6.14 : Types non transformables à cause du renommage

Dans une transformation statique, l'utilisateur a les moyens d'exprimer de manière exhaustive, les associations de types à l'intention du comparateur. Tel n'est pas le cas des transformations dynamiques. En effet, dans les transformations statiques, le langage *K* permet d'écrire des règles dites *règles de correspondance* qui spécifient pour un type donné le type vers lequel ses instances doivent être converties : c'est le couple d'évolution notée (t, a(t)), dont la sémantique est exprimée comme suit :

a(t) est la nouvelle définition de t.

Dans une transformation dynamique, le langage K n'est pas disponible même si dans l'absolu, il est possible d'envisager une correspondance graphique entre les types. Cette dernière peut se révéler fastidieuse à cause de la profondeur des types et surtout elle suppose une bonne connaissance de la sémantique associée aux types. Généralement, l'utilisateur n'a pas une connaissance approfondie des schémas de structure.

Nous donnons ci−après quelques exemples qui illustrent l'échec des transformations fondées uniquement sur l'homonymie. Ces exemples mettent en évidence la nécessité d'étendre le modèle avec des informations relatives à la structure et au contenu. Soient les types A (voir Fig. 6.15) et F (voir Fig. 6.16) définis comme suit :

```
A = BEGIN
        B  =  TEXTE;
        C  =  TEXTE
        D  =  IMAGE;
        E  =  TEXTE;
    END;
```

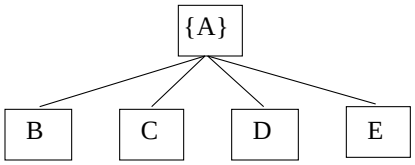


Fig. 6.15 : Représentation arborescente du **type** *A*

```
F = BEGIN
        B  =  TEXTE;
        H  =  BEGIN
                  C  =  TEXTE
                  J  =  BEGIN
                            D  =  IMAGE;
                            E  =  TEXTE;
                        END;
              END;
    END;
```

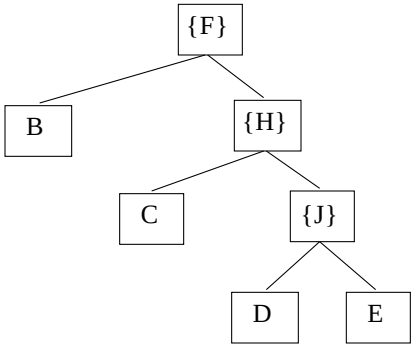


Fig. 6.16 : Représentation arborescente du **type** *F*

Dans la définition du type F, les feuilles gardent les mêmes identificateurs que dans la définition de A, afin de faciliter la compréhension du problème(3). Intuitivement, la restructuration dynamique des instances de type A en instances de type F semble possible, et pourtant elle échoue. Cet échec est dû à l'absence de couples d'évolution qui oblige le processus chargé de réaliser la transformation à fonder son fonctionnement sur le principe d'homonymie.

Nous donnons ci−dessous une illustration du fonctionnement du processus chargé de réaliser l'évolution du type Étudiant (voir Fig. 6.17, Fig. 6.18) vers le type Enseignant (voir Fig. 6.19, Fig. 6.20).

```
Étudiant = BEGIN
               Photo  =  IMAGE;
               Nom  =  TEXTE;
               Rue  =  TEXTE;
               Ville  =  TEXTE;
               NumSs  =  TEXTE;
               Note  =  TEXTE;
           END;
```

Fig. 6.17 : Type Étudiant

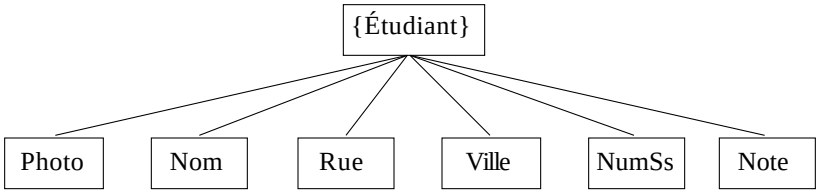


Fig. 6.18 : Représentation arborescente du **type** *Étudiant*

(3) Il suffit de considérer que les types A et F appartiennent à deux schémas de structure différents. Dans l'hypothèse où ils appartiendraient au même schéma de structure, les feuilles auraient des identificateurs différents et le problème semblerait plus complexe. L'usage de la fonction org permettrait alors de retrouver les identificateurs d'origine et donc le cadre actuel.

```
Enseignant  =  BEGIN
                  Photo_E  =  IMAGE;
                  Nom_E  =  TEXTE;
                  Confidentiel_E  =
                      BEGIN
                        Appréciation_E  =  SYMBOLE;
                        Adresse_E  =  BEGIN
                                        Rue_E  =  TEXTE;
                                        Ville_E  =  TEXTE;
                                      END;
                        Affiliations_E  ;
                        NumSs_E  =  SYMBOLE;
                        Divers_E  =  TEXTE;
                      END;
                  END;
Affiliations_E  =  LIST  OF(Affiliation_E);
Affiliation_E  =  CASE  OF
                    Ident_E  =  TEXTE;
                    RefIdent_E   = REFERENCE(Ident_E);
                  END;
```

Fig. 6.19 : Type Enseignant

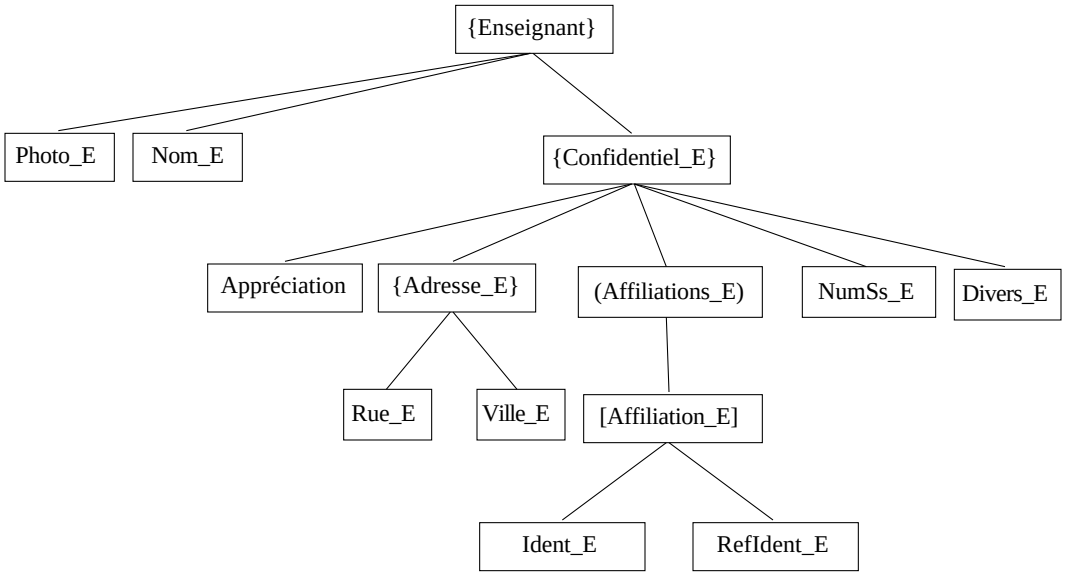


Fig. 6.20 : Représentation arborescente du **type** *Enseignant*

La tentative d'instanciation des éléments de Fils(Étudiant) aux éléments de Fils(Enseignant) ne donne rien. Le résultat de la restructuration est une instance de type Enseignant vide. Dans l'hypothèse où, org(Photo) = org(Photo_E) et org(Nom) = org(Nom_E), les éléments engendrés par les composants Photo et Nom seraient récupérés. Intuitivement, il n'existe aucun élément de Fils(Étudiant) qui ait la même origine que Confidentiel_E. Dans ces conditions, l'élément de type Confidentiel_E de la transformée d'une instance de type Étudiant, a un contenu vide.

L'analyse des intentions de l'utilisateur a conduit à la définition d'un certain nombre de besoins, en terme de sémantique des transformations. Ces besoins sont traduits sous forme de contraintes structurales entre types, comme par exemple :

- Le type destination doit avoir une structure identique à celle du type source.
- La structure du type source doit être incluse dans la structure du type destination.
- L'ensemble du contenu du type source doit se retrouver dans le même ordre, dans celui du type destination.
- L'ensemble du contenu du type source doit se retrouver dans celui du type destination quel que soit son ordre.

L'idée retenue est de comparer les structures que sont les types, à la recherche d'éventuelles relations qui les uniraient. Ces comparaisons de structures pourraient conduire également à la construction de classes d'équivalence autorisant une transformation entre deux types quelconques appartenant à la même classe. Une relation d'ordre en revanche ne permettrait la transformation que dans un sens donné.

La conclusion sur l'analyse des intentions de l'utilisateur, donc sur la sémantique des transformations, est que les requêtes à satisfaire peuvent ne contenir aucune contrainte de structure (simple récupération du contenu), ou au contraire exiger une identité de structure. L'étude de la structure des types VI.6 permet de couvrir un plus grand nombre de transformations.

VI.6 Extensions d'ordre structural (≡)

Étant donné un ensemble T de types, l'intérêt de l'extension structurale des mécanismes fondés sur l'homonymie est d'établir une relation d'ordre entre les éléments de T. On mettra ainsi en évidence des caractéristiques autres que celles mises en œuvre jusqu'à maintenant (principe d'homonymie) pour réaliser davantage de transformations. Une autre raison non moins importante de cette étude est la possibilité de conclure à la faisabilité ou non d'une transformation uniquement par l'analyse statique sur les types, ce qui est tout de même moins coûteux qu'une comparaison entre instances de documents.

Cette section conduit à la définition des notions d'équivalence, de facteur et de massif de types. Les caractéristiques présentées dans la section VI.3 vont servir à définir un certain nombre de relations et à établir les conditions nécessaires et suffisantes pour réaliser les transformations.

Cette étude ne se fonde plus uniquement sur les caractéristiques des types telles que définies en VI.3 mais aussi sur leur structure arborescente.

VI.6.1 Relations d'équivalence

Il s'agit d'identifier et de regrouper les types qui ont la même structure arborescente. La relation est notée ≡ et t ≡ t' se lit t est équivalent à t'.

VI.6.1.1 Isomorphisme de types

Intuitivement, on conclut à l'identité de structure des types Esquisse et Section, en considérant la définition suivante tirée de la figure Fig. 6.3 :

```
?Esquisse = Section;
```

On peut facilement en être convaincu puisque cette définition indique que Esquisse a la même définition que Section. De plus, dans la forme canonique, la partie droite de cette définition est remplacée par la partie droite de la définition du type Section dans laquelle les identificateurs sont tous inédits. Informellement, à ce changement d'identificateurs près, on admet le fait que la structure arborescente de Esquisse soit la même que celle de Section. La recherche de l'équivalence de structure ne se limite pas aux types définis explicitement comme identiques à d'autres types. En effet il peut exister des types définis autrement que par le constructeur Identité et qui ont pourtant la même structure que d'autres types du même schéma. Il peut exister également deux types appartenant à deux schémas de structure différents et qui ont la même structure. Il convient alors de les identifier. De tels types sont dits *isomorphes* et font partie de la même classe d'équivalence.

Définition

Soient deux types t et t' ; l'application $\varphi : D(t) \cup \{t\} \rightarrow D(t') \cup \{t'\}$ est un isomorphisme si $\varphi(t) = t'$ et $\forall u \in D(t) \cup \{t\}$ elle satisfait l'une des conditions suivantes :

1. **Élément de base**

   $u \in \beta \rightarrow org(u) = org(\varphi(u))$

   u est le même type de base que $\varphi(u)$.

2. **Homonymie**

   $org(u) = org(\varphi(u))$ c'est−à−dire u et $\varphi(u)$ ont le même identificateur dans la forme classique du schéma de structure.

3. **Élément construit**

Bijection	$\#D(u) = \#D(\varphi(u))$.
Identité de constructeur	$constr(u) = constr(\varphi(u))$.
Égalité de cardinal	$card(u) = card(\varphi(u)$.
Identité de niveau	$\#Fils(u) = \#Fils(\varphi(u))$ et $\forall z \in Fils(u)\ \exists z' \in Fils(\varphi(u))$ tels que $rang(u)(z) = rang(\varphi(u))(z')$ et $z \equiv z'$.

Il faut remarquer que l'égalité de cardinal est une condition forte qui empêche particulièrement deux types t et t' de constructeur Liste d'appartenir à la même classe d'équivalence, seulement parce que leurs cardinaux diffèrent, c'est−à−dire $card(t) \neq card(t')$.

La fonction φ ne se limite pas à la fonction d'association a. Il est également souhaitable de rechercher d'autres associations possibles en vue de réaliser des transformations autres que celles prévues par l'utilisateur. Les inclusions ne font

pas partie de cette définition car elles sont considérées comme optionnelles. Si le processus de transformation ne tient pas compte des inclusions, il pourrait y avoir perte des informations contenues dans les éléments éventuellement inclus.

Remarque

On note $t \equiv t'$ l'équivalence de structure entre t et t'. Si t et t' sont deux types, si φ est un isomorphisme de $D(t) \cup \{t\}$ dans $D(t') \cup \{t'\}$ alors la bijection réciproque φ^{-1} est un isomorphisme de $D(t') \cup \{t'\}$ dans $D(t) \cup \{t\}$.

En effet si $\varphi(Fils(t)) = Fils(\varphi(t))$ alors $\varphi(Fils(\varphi^{-1}(t'))) = Fils(t')$; donc $Fils(\varphi^{-1}(t')) = \varphi^{-1}(Fils(t'))$.

L'intérêt de la relation réciproque est que t' est transformable en t.

On peut ainsi établir une relation d'équivalence entre des types qui présentent une identité de structure. Ces types dits isomorphes suivant cette définition, appartiennent à la même classe d'équivalence notée $[t] : t \equiv t' \Leftrightarrow [t] = [t']$. On en conclut que t est transformable en t' et t' est transformable en t.

VI.6.1.2 Élargissement de la relation d'équivalence à un ensemble de schémas

En raison de la modularité des types (schémas de structure) qui caractérise les éditeurs de documents structurés, les classes d'équivalence sont définies pour un schéma de structure donné. Cependant, les requêtes de restructurations dynamiques peuvent concerner deux documents de schémas différents. Toutes les autres caractéristiques supposées égales par ailleurs, les éléments de deux classes d'équivalence de schémas différents peuvent ne différer que par leurs identificateurs. Il est souhaitable de profiter de cette identité de structure pour étendre la notion d'équivalence à plusieurs schémas. Tout se passe comme si on construisait une classe d'équivalence plus grande par une réunion virtuelle de plusieurs autres classes appartenant à des schémas de structure différents. À cette fin, il suffit de comparer deux représentants de deux classes d'équivalence (un pour chaque classe), pour conclure ou non à leur réunion virtuelle. On peut ainsi transformer entre elles des instances produites par des types appartenant à la même classe d'équivalence à la différence près que la dite classe d'équivalence est virtuelle. Dans la figure Fig. 6.21, les motifs en pointillé représentent des classes d'équivalence virtuelles, construites à partir d'extraits des schémas de structure Article (voir Fig. 6.23) et Paragraphe (voir Fig. 6.22).

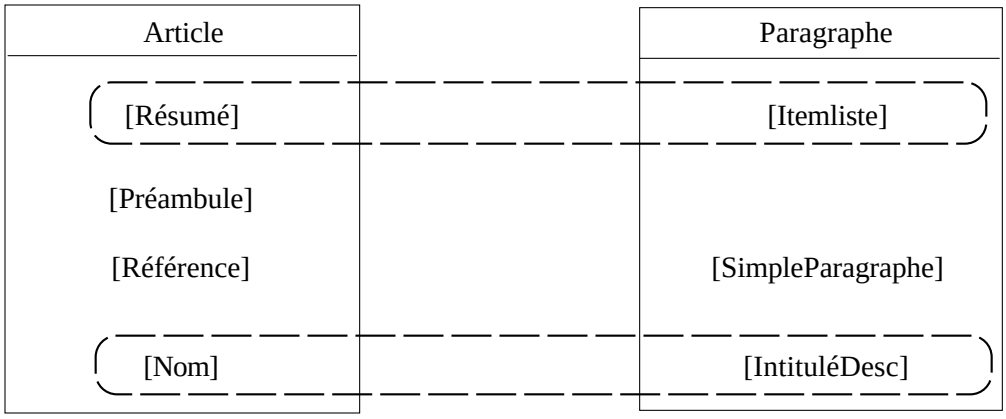


Fig. 6.21 : Illustration de l'élargissement de classes d'équivalence

```
Paragraphe =
  CASE OF
       SimpleParagraphe = LIST OF (ElementDePara);
       Citation = LIST OF (ElementDePara);
       Exemple = LIST OF (LigneExemple);
        ElemListe = LIST OF (ItemListe);
        ParagrapheTitré =
            BEGIN
                 TitreParagraphe = SimpleParagraphe;
                 ContenuPara = LIST OF (Paragraphe);
            END;
       CoteACote = BEGIN
                     Gauche = LIST OF (Paragraphe);
                     Droite = LIST OF (Paragraphe);
                   END;
       Description =
          BEGIN
              IntituléDesc = TEXTE;
               TexteDesc = LIST OF (Paragraphe);
          END;
       Programme = LIST OF (ElementDeProgramme);
        ParagraphesGroupés = LIST OF (Paragraphe);
       Énumération = LIST OF(ItemÉnumération);
  END;
LigneExemple = LIST OF (ElementDePara);
ItemListe = LIST OF (Paragraphe);
ItemÉnumération = LIST OF (Paragraphe);
ElementDeProgramme = CASE OF
                            Instruction = TEXTE;
                             BlocInstructions;
                       END;
BlocInstructions = LIST OF (ElementDeProgramme);
```

Fig. 6.22 : Extrait du schéma de structure Paragraphe

```
Article = BEGIN
              ?Référence =  Contenu;
              ?Statut = Contenu;
              ?DateMiseAjour = Contenu;
              Titre = Contenu;
              Auteurs = LIST OF (Auteur);
              Résumé = LIST OF (Paragraphe);
              Abstract = LIST OF (Paragraphe);
              ?MotsClés = Contenu;
              ?Préambule = SuiteParagraphes;
              Corps = SuiteSections;
          END;
Contenu = LIST OF (Unité = UNIT);
Auteur = BEGIN
           Photo = IMAGE;
           Nom = TEXTE;
           Affiliations = LIST OF (Affiliation);
           ?Grade = TEXTE;
         END;
Affiliation = Contenu;
SuiteParagraphes = LIST OF (Paragraphe);
SuiteSections = LIST [2 . . *] OF (Section);
Section = BEGIN
              TitreS = TEXTE;
              CorpsSection = SuiteParagraphes;
                ?SuiteSousSections;
          END;
SuiteSousSections = LIST OF(SousSection);
SousSection = BEGIN
              TitreSs = TEXTE;
              CorpsSousSection = SuiteParagraphes;
                ?SuiteSousSousSections;
             END;
SuiteSousSousSections = LIST OF(SousSousSection);
SousSousSection =
          BEGIN
           TitreSss = TEXTE;
           CorpsSousSousSection = SuiteParagraphes;
          END;
```

Fig. 6.23 : Extrait du schéma de structure Article

VI.6.2 Relations d'ordre

Il est possible d'établir des relations d'ordre entre deux types. Cette sous−section définit les conditions de facteur de type et de massif de type.

VI.6.2.1 Facteur de type (↵)

La notion d'isomorphisme de types est très forte et ne correspond pas toujours à toutes les transformations possibles. La notion de **facteur** est à rapprocher de celle de **sous−arbre**.

Définition

Étant donné deux types t et u, on dit que u est un facteur de t si la condition suivante est respectée :

$$\exists z \in D(t), \text{ tel que } u \equiv z \text{ ou } org(u) = org(z).$$

La relation qui lie deux types est alors la relation *être facteur de*. Cette notion est la généralisation de la notion de composant dans la mesure où si v est un facteur de u et si u est un facteur de t alors v est un facteur de t. Il s'agit d'une relation transitive.

Notation

On notera par u ↵ t le fait que u est un facteur de t.

Les éléments de type u sont ainsi assurés de trouver un sous−arbre d'accueil dans t. Le type Résidence défini ci−dessous et illustré par le sous−arbre de la figure Fig. 6.24, est un facteur du type Enseignant de la figure Fig. 6.20 :

```
Résidence = BEGIN
                Voie = TEXTE;
                Localité = TEXTE;
            END;
```

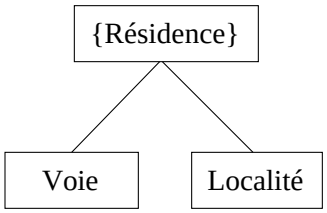


Fig. 6.24 : Arbre du **type** *Résidence*

Pour prouver que Résidence est un facteur de Enseignant, nous montrons qu'il est équivalence au type Adresse_E qui est un sous−arbre du type Enseignant :

Résidence ≡ Adresse_E
φ(Résidence) = Adresse_E
#D(Résidence) = #D(Adresse_E) = 2
rang(Résidence)(Voie) = rang(Adresse_E)(Rue) = 1
rang(Résidence)(Localité) = rang(Adresse_E)(Ville) = 2
Voie ≡ Rue_E
φ(Voie) = Rue_E
#D(Voie) = #D(Rue) = 1
Localité ≡ Rue
φ(Localité) = Ville_E
#D(Localité) = #D(Vile) = 1

VI.6.2.2 Massif de type ($\perp$)

La notion de massif de type est une généralisation de la notion de facteur de type, dans la mesure où le massif de type peut être considéré comme un ensemble de facteurs. La définition qui a été faite d'un massif en VI.2.1.3 est essentiellement fondée sur la relation de précédence entre les nœuds d'un arbre. Cette définition suppose en outre que l'ensemble des nœuds d'un massif soient un sous−ensemble de l'ensemble des nœuds de l'arbre destination, autrement dit qu'un nœud dans le massif garde le même identificateur que dans l'arbre destination. Cette hypothèse n'est plus du tout vérifiée puisque les transformations de structure sont généralisées à un couple de types quelconques, donc à deux ensembles qui à priori ne sont liés par aucune des relations d'équivalence ou de facteur de type. Les remarques suivantes s'imposent :

1. Les conditions de transformation sont telles que les ensembles d'identificateurs des nœuds des deux types d'un couple d'évolution peuvent être disjoints.
2. La seule relation de précédence est insuffisante pour l'identification d'un massif en fonction de caractéristiques structurales.

Afin de permettre l'identification d'un massif dans de telles conditions, il est fait une nouvelle définition du massif, qui est en fait un enrichissement de la définition précédente (voir VI.2.1.3) avec les caractéristiques structurales des types :

Définition

Soient deux types t et t' dont les arbres sont définis ci−après :

- L'arbre (M, s) de racine t est l'arbre du type t.
- L'arbre (E, p) de racine t' est l'arbre du type t'. (E, p) est l'arbre de référence dans lequel on recherche une instance de l'arbre (M, s). Du point de vue de la restructuration, (E, p) est l'arbre du type vers lequel on fait évoluer les instances du type t.

Soit une application injective $m : M \rightarrow E$. On dit que t est un massif de t' si les conditions suivantes sont satisfaites :

1. $\#M < \#E$;
2. $\forall u \in M\ u \neq t, \exists i \in N$ tel que :
   - $u \in \beta \rightarrow org(u) = org(m(u))$
   - $constr(u) = constr(m(u))$ et $card(u) \leq card(m(u))$
   - $\exists u' = p^i(m(u))$ tel que $constr(u') = constr(s(u))$ et $card(s(u)) \leq card(u')$

     Un ascendant de l'image m(u) a le même constructeur que le père de u dans le massif et a un nombre autorisé de fils au moins égal à celui du père de u.
   - $\forall j \in N, j > 0, p^j(m(u)) \in M \Rightarrow j \geq i$.

     Tout ascendant de l'image m(u) dans l'arbre de référence, reste ascendant de tous ses descendants qui appartiennent au massif.

Autrement dit, la relation d'ordre dans la chaîne à laquelle appartient m(u) doit être respectée dans la chaîne correspondante du massif.

Nous pouvons définir la relation d'ordre *être un massif de* notée $\perp$ telle que t $\perp$ t' signifie que t est un massif de t'. L'application m pourrait être la fonction d'association. Nous donnons ci−après deux exemples qui illustrent la notion de massif et son utilité :

Exemple 1

Le type Pion (voir Fig. 6.25, Fig. 6.26) est un massif du type Enseignant (voir Fig. 6.19, Fig. 6.20). Cette conclusion est illustrée par l'application m définie comme suit : $\{D(Pion) \cup Pion) \rightarrow \{D(Enseignant) \cup \{Enseignant\}$.

m(Pion) = Enseignant,

m(Nom_P) = Nom_E,

m(Confidentiel_P) = Confidentiel_E,

m(CodePion_P) = Appréciation_E,

m(Information_P) = Divers_E.

```
Pion = BEGIN
         Nom_P = TEXTE;
         Confidentiel_P  = BEGIN
                             CodePion_P  =  SYMBOLE;
                             Information_P  =  TEXTE;
                           END;
       END;
```

Fig. 6.25 : Définition du type Pion dans le langage S

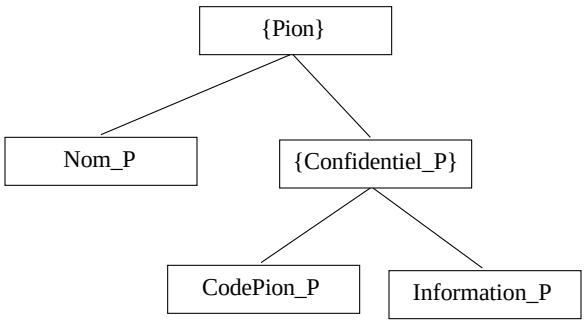


Fig. 6.26 : Représentation arborescente du **type** *Pion*

Exemple 2

Soient les types t et t' de constructeur Liste dont la seule différence porte sur leur cardinal : card(t) < card(t'). Il en résulte la non appartenance de t et t' à la même classe d'équivalence. En revanche, la relation t $\perp$ t' est vraie.

VI.6.2.3 Sous−type ([)

Soient deux types t et t’ et leurs arbres respectifs définis comme suit :

- L’arbre (F, q) de racine t est l’arbre du type t.
- L’arbre (E, p) de racine t’ est l’arbre du type t’.

Soit une application injective s : E → F. On dit que t est un sous−type de t’ si les conditions suivantes sont satisfaites :

1. #E ≤ #F
2. s(t’) = t.
3. constr(t’) = constr(t).
4. #Fils(t’) ≤ #Fils(t).
5. ∀u’ ∈ Fils(t’), s(u’) ∈ Fils(t) tels que u’ ≡ s(u’) ou u’ est un sous−type de s(u’).

La notation suivante est retenue pour exprimer la relation de sous−typage entre t et t’ : t [t’. On dira que t est un sous−type de t’ ou que t’ est un super−type de t. La notion de sous−type garde sa sémantique traditionnelle d’inclusion de valeurs. Ces valeurs sont représentées par les types de base et l’identité d’un contenu est garantie par la relation d’équivalence.

De manière Informelle, la notion de sous−type peut être expliquée comme suit : un type t, sous−type d’un type t’ est obtenu par extension de t’ et/ou des éléments de D(t’). À titre d’exemple, le type ChefDeProjet (voir Fig. 6.28) est un sous−type du type Développeur (voir Fig. 6.27).

```
Développeur = BEGIN
                Nom  =  TEXTE;
                 Projet = BEGIN
                           Intitulé  =  TEXTE;
                           Module  =  SYMBOLE;
                          END;
              END;
```

Fig. 6.27 : Super−type du type ChefDeProjet

```
ChefDeProjet = BEGIN
                  Nom  =  TEXTE;
                   Projet = BEGIN
                             Intitulé  =  TEXTE;
                             Module  =  SYMBOLE;
                             Contrainte  =  TEXTE;
                            END;
                  Équipe  =  LIST  OF(Membre);
                 END;
Membre  =  BEGIN
              Nom  =  TEXTE;
              Délai  =  TEXTE;
              Appréciation  =  SYMBOLE;
           END;
```

Fig. 6.28 : Sous−type du type Développeur

Remarque

t [t' ⇒ t' ⊥ t.

La réciproque n'est pas toujours vraie.

Étant donné le couple d'évolution (t, t') la conversion attendue est la suppression dans les instances de t, des éléments engendrés par les types de D(t) qui n'ont pas d'antécédents dans D(t') par la fonction s.

VI.6.2.4 Compatibilité de type (◊)

Les conditions d'équivalence ne permettent de regrouper dans la même classe d'équivalence que des types qui ont le même cardinal et le même constructeur. Ces conditions fortes ne permettent pas d'envisager des relations structurales entre des types qui pourtant ont des structures très voisines. Nous donnons ci−dessous un exemple qui illustre cette remarque.

```
Individu  =  BEGIN
                Photo  =  IMAGE;
                Nom  =  TEXTE;
                Situation  =  SYMBOLE;
             END;
```

Fig. 6.29 : Type Individu

```
Postulants  =  LIST  OF(Postulant);
Postulant  =  CASE  OF
                    Photo  =  IMAGE;
                    Nom  =  TEXTE;
                    Situation  =  SYMBOLE;
                    CurriculumVitae  =  TEXTE;
               END;
```

Fig. 6.30 : Type Postulants

Une requête de transformation dynamique d'une instance de type Individu Fig. 6.29 en une instance de type Postulants Fig. 6.30 n'est résolue par aucun des mécanismes exposés, ni même par l'existence d'une règle de changement de constructeur puisque la comparaison statique des types concernés n'a pas été faite. On a pourtant l'intuition d'une possibilité de transformation, mise en évidence par l'analyse ci−dessous. Étant donné le couple (t, t'), t est compatible avec t' (notée t ◊ t') si les conditions suivantes sont satisfaites :

1. constr(t) ∈ {Agrégat, Agrégat sans ordre}
2. constr(t') = Liste et ∃u' ∈ fils(t') tel que constr(u') = Choix. Il faut noter que #fils(t') = 1 et qu'il est impossible d'exclure fils(t'). Dans ces conditions, fils(t') = Fils(t').
3. ∀u ∈ Fils(t), ∃u' ∈ fils(t') tel que ∃v' ∈ Fils(u') tel que u ≡ v' ou u ◊ v'.
4. card(t) ≤ card(t')

Lorsque la relation t ◊ t' est vraie, la transformation correspondante des instances consiste à effectuer les opérations suivantes :

- Changer le constructeur de t en Liste.
- Intercaler un nœud de constructeur Choix entre les instances correspondant aux fils de t et le nœud racine de l'instance générée par t.
- Transformer éventuellement les instances correspondant aux fils de t.

VI.6.2.5 Élargissement des relations d'ordre à un ensemble de schémas

Dans un schéma de structure, on réunit dans une même classe d'équivalence tous les types qui sont massifs d'un type donné. Les instances engendrées par les types appartenant à cette classe d'équivalence ne sont pas forcément convertibles entre elles. Cependant, lorsqu'il s'agit de faire des transformations impliquant des types appartenant à des schémas de structure différents, il est souhaitable d'élargir les relations d'ordre aux schémas de structure concernés. Cette analyse se fonde sur la relation ⊥.

En effet, soient le schéma S_1 et les relations suivantes : $u_1 \perp t_1$, $v_1 \perp t_1$, $w_1 \perp t_1$. Soit la classe d'équivalence $[u_1] = \{u_1, v_1, w_1\}$ construite avec la relation *être massif d'un même arbre que*. Soient les relations suivantes, $u_2 \perp t_2$, $v_2 \perp t_2$, $w_2 \perp t_2$ dont les éléments appartiennent au schéma S_2. On construit la classe d'équivalence $[u_2] = \{u_2, v_2, w_2\}$ avec la même relation que pour $[u_1]$.

Supposons la relation $t_1 \perp t_2$. L'idée est de conclure à la possibilité d'évolution des éléments de $[u_1]$ en t_2 : c'est−à−dire $\forall u_i \in [u_1]$, $u_i \perp t2$. Cette conclusion nous permettrait d'envisager la conversion entre des éléments dont les schémas de structure sont différents. On entrevoit l'élargissement à plusieurs schémas, des relations localement définies (voir VI.6.4). Les classes d'équivalence dans ce cas servent uniquement à regrouper les types qui sont des massifs d'un même type. L'élargissement d'une relation d'ordre à plusieurs schémas de structure est analysée à travers l'étude de la compatibilité des relations d'équivalence avec l'orde (voir VI.6.4). En considérant l'ensemble des types définis dans tous les schémas de structure, le modèle permet d'établir directement des relations entre n'importe quels types, sans tenir compte des problèmes d'implémentation et/ou de concept, tels que la

classification, la modularité, etc. Dans ce cas, il ne s'agira pas d'élargissement de relations à d'autres schémas, mais de définition des mêmes relations sur sur un ensemble de types, plus important.

VI.6.3 Mesure de la dispersion d'un massif

Les solutions envisagées pour résoudre les transformations complexes sont basées sur les relations définies en VI.6.1 et en VI.6.2. Par définition, un massif est une dispersion d'un arbre dans un autre. Il est souhaitable de mesurer cette dispersion afin de fournir à l'utilisateur un critère de choix dans l'hypothèse de l'existence de plusieurs massifs. Le calcul de cette mesure est fondé sur la relation d'ordre total dans un arbre. L'ordre préfixe est retenu car il correspond à l'ordre dans lequel les documents sont édités et lus. Il permet d'affecter à chaque nœud un poids qui sera utilisé dans le calcul de la dispersion.

Nous avons montré en VI.3.9 qu'il existe un ordre total et linéaire sur Fils(t), donné par la fonction rang(t) : Fils(t) $\rightarrow$ [1 . . #Fils(t)]. Cette fonction définie en fonction des instances, s'appliquait jusqu'à maintenant aux types dans la mesure où le problème ne s'est pas posé de s'en servir pour ordonner les arbres de type. Elle ne permet pas d'ordonner un arbre de type. Pour cette raison, une autre fonction de rang est introduite spécifiquement pour les types :

ordre : Fils(t) $\rightarrow$ [1 . . #Fils(t)].

Cette fonction associe à n'importe quel composant de type, sa position syntaxique dans l'ensemble des composants qui ont le même père que lui, quel que soit le constructeur de ce dernier. La figure Fig. 6.31 résume les différences entre les fonctions ordre et rang, en fonction des constructeurs.

Constructeurs et rang des fils	Liste	Choix	Agrégat ordonné	Agrégat sans ordre
rang()	[1 . . card(t)]	[1. . 1]	[1 . . #Fils(t)]	Sans position fixe
ordre()	[1 . . #Fils(t)]	[1 . . #Fils(t)]	[1 . . #Fils(t)]	[1 . . #Fils(t)]

Fig. 6.31 : Domaines de valeurs des fonctions rang et ordre

Nous avons vu aussi qu'il existe un ordre partiel induit par la fonction p telle que : t = p(u) $\rightarrow$ t $\leq$ u. Étant donné que nous cherchons avant tout à comparer des types, l'analyse de la dispersion sera faite sur les arbres de type ; par conséquent, la fonction ordre va être utilisée à la place de la fonction rang.

Nous allons maintenant montrer l'existence d'un ordre total dans un arbre compatible avec les ordres évoqués ci−dessus.

Proposition

Étant donné un arbre (E, p), de racine ρ, il existe sur E un ordre total $<_t$ compatible avec les ordres induits par les fonctions p et ordre tel que :

1. $\forall t, u \in E, t = p(u) \Rightarrow t <_t u$.
2. $\forall t \in E, \forall u, z \in Fils(t), ordre(t)(u) < ordre(t)(z) \Rightarrow u <_t z$.

VI.6.3.1 Ordre total préfixe

Pour prouver la proposition ci−dessus, nous allons construire un ordre total sur les types. L'ordre total préfixe noté $<_t$ ordonne les nœuds d'un arbre et respecte l'ordre de parcours des documents. La construction de cet ordre est basée sur la notion de chaîne rappelée ci−dessous.

Définition

Une chaîne dans un ensemble ordonné (E, <) est une suite $u_0, u_1, \ldots, u_n$ croissante telle que $u_0 < u_1 < \ldots < u_n$.

On note $\mu(E)$ le plus petit élément de (E, <) tel que $\forall u \in E, \mu(E) \leq u$. Dans la chaîne $u_0 < u_1 < \ldots < u_n$, $\mu(E) = u_0$.

On note $\omega(E)$ le plus grand élément de (E, <) tel que $\forall u \in E, u < \omega(E)$. Dans la chaîne $u_0 < u_1 < \ldots < u_n$, $\omega(E) = u_n$.

On vérifie sans peine que tous les ascendants d'un nœud quelconque d'un arbre forment une chaîne :

Soit u un nœud de l'arbre (E, p) ; soit h(u) la hauteur de u ; alors on sait que $h(p(u)) = h(u) - 1$ et que $h(p^{h(u)}(u)) = h(\rho) = 0$ (hauteur de la racine) ; les ascendants de u forment donc la suite $p^{h(u)}(u), p^{h(u)-1}(u), \ldots, p(u), u$. On définit une chaîne de la manière suivante : étant donné un arbre (E, p), soit $u \in E$, et soit h(u) le niveau de u dans l'arbre ; l'ensemble $\{t \in E \mid \exists i \in N, t = p^i(u)\}$ ordonné par la fonction *p* est une chaîne dont u est le plus grand élément et la racine de l'arbre, le plus petit élément.

Construction de l'ordre préfixe

Dans un arbre, nous allons construire une chaîne unique selon un ordre compatible avec les ordres induits par les fonctions p et ordre. Cette construction est basée sur le parcours préordre de l'arbre et sur l'attribution d'un numéro aux nœuds rencontrés. Dans un arbre, on note t−chaîne un sous−arbre de racine t, qui dispose des caractéristiques suivantes :

$\forall u \in$ t−chaîne et $u \neq t$,
$\exists z \in$ t−chaîne tel que
z = p(u) et
ordre(z)(u) = 1
En d'autres termes, $u = \mu(Fils(z))$ ce qui signifie que u est le plus petit élément de la chaîne formée par les éléments de l'ensemble Fils(z) ordonné par la relation ordre(z).

Dans toute t−chaîne, la relation d'ordre induite par p est conservée par définition.

Notation

On note $PFX_i(t)$ l'ensemble ordonné des rangs des éléments rencontrés pendant le parcours préfixe de l'arbre de racine t, y compris la f_i−chaîne du $i^{ème}$ fils de t. Par exemple (voir Fig. 6.32), PFX_1(Enseignant) = 1, 2, PFX_2(Enseignant) = 1, 2, 3.

Initialisation de $PFX_1(\rho)$

Étant donné un arbre (E, p) de racine ρ, on commence par initialiser la chaîne $PFX_1(\rho)$ correspondant à son ordre préfixe tel que $PFX_1(\rho) = \rho$-chaîne. Dans ρ-chaîne, on affecte à chaque élément u de la chaîne un numéro égal à h(u) + 1. C'est cette affectation de numéros consécutifs qui crée l'ordre préfixe $<_t$. Soit RANG une fonction qui retourne le position d'un élément dans une chaîne : RANG(ρ) = 1 et RANG(u) = RANG(t) + 1 si t = p(u). Dans ρ−chaîne, la relation d'ordre induite par p est compatible avec l'ordre $<_t$ car $\forall u \in$ Fils(p(u)), p(u) apparaît dans la chaîne avant u et RANG(p(u)) $<_t$ RANG(u).

Extension de $PFX_i(\rho)$

Il s'agit d'étendre $PFX_i(\rho)$ aux éléments de (E, p) qui n'appartiennent pas encore à $PFX_i(\rho)$.

Cette extension va se faire par construction, de la façon suivante : $PFX_i(\rho)$ est parcourue depuis $\omega(PFX_i(\rho))$ vers $\mu(PFX_i(\rho))$, en visitant pour chaque élément u, son père p(u) qui appartient forcément à la chaîne. Soit u l'élément courant. La $PFX_i(\rho)$ est prolongée par une z−chaîne telle que $z \in$ Fils(p(u)) et que $z \notin PFX_i(\rho)$ et que ordre(p(u))(z) = ordre(p(u))(u) + 1. Si p(u) = ρ, z est le $(i + 1)^{ème}$ fils de ρ et $PFX_i(\rho)$ devient $PFX_{i+1}(\rho)$. On affecte à chaque élément z' de la z−chaîne un numéro tel que RANG(z') = RANG(z') + RANG($\omega(PFX_i(\rho))$).

L'ordre $<_t$ est compatible avec l'ordre induit par la fonction ordre. En effet, soit u = $\omega(PFX_i(\rho))$ d'avant la prolongation. RANG(u) $<_t$ RANG(z) et comme u et z sont des frères, on sait que ordre(p(u))(u) < ordre(p(u))(z). La $PFX_i(\rho)$ est prolongée, ρ reste le plus petit élément et ω(z−chaîne) devient $\omega(PFX_i(\rho))$.

Dans un arbre, l'ordre préfixe $<_t$ ordonne d'une unique façon les fils d'un même nœud, engendrant un unique ordre d'arbre.

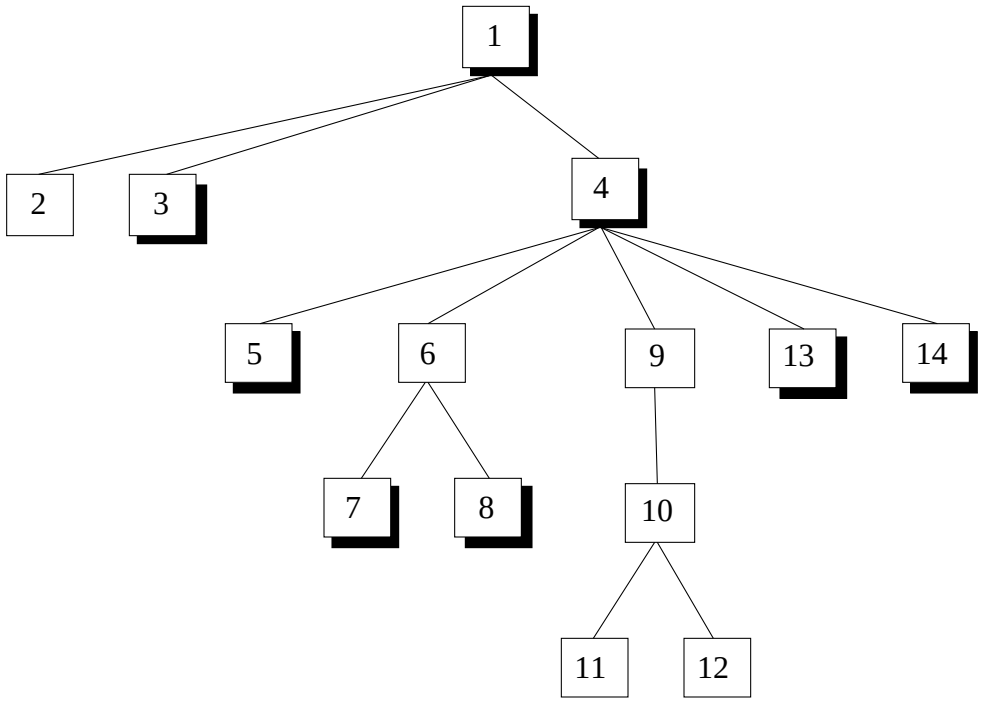


Fig. 6.32 : Ordre préfixe de l'arbre du **type** *Enseignant*

VI.6.3.2 Dispersion d'un arbre

Soient deux types t et t' et m une application injective de D(t) ∪ {t}dans D(t') ∪ {t'}. On associe à t une quantité d(t, m, t') qui donne une mesure de la dispersion de sa transformée par m (4) :

$$d(t, m, t') = \sum_{\forall x \in D(t)} RANG(m(x)) \qquad (4)$$

Soient m_i un ensemble de fonctions de D(t) ∪ {t} dans D(t') ∪ {t'} tel que l'image de t par m_i soit un massif dans t'. Il se pose alors le problème de choisir le massif le plus convenable. La dispersion est un critère de choix possible dans la mesure où il permet de retenir le massif le plus compact ; ce qui revient à choisir la plus petite dispersion soit min(d(t, m_i, t')). Nous allons illustrer cette notion à partir de l'exemple du type Pion (voir Fig. 6.33), massif du type Enseignant (voir Fig. 6.20).

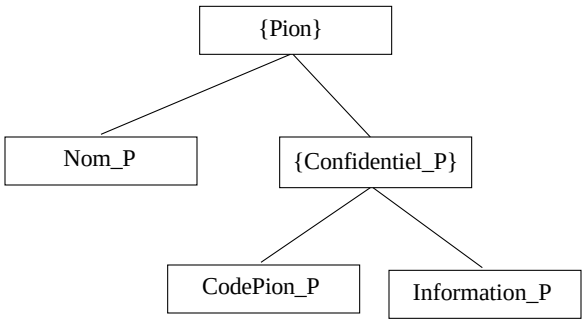


Fig. 6.33 : Représentation arborescente du **type** *Pion, massif du type Enseignant*

Soient quatre applications m_1, m_2, m_3 et m_4 de D(Pion) $\cup$ {Pion} dans D(Enseignant) $\cup$ {Enseignant} ainsi définies :

m_1

m_1(Pion) = Enseignant,

m_1(Nom_P) = Nom_E,

m_1(Confidentiel_P) = Confidentiel_E,

m_1(CodePion_P) = Appréciation_E,

m_1(Information_P) = Rue_E.

m_2

m_2(Pion) = Enseignant,

m_2(Nom_P) = Nom_E,

m_2(Confidentiel_P) = Confidentiel_E,

m_2(CodePion_P) = Appréciation_E,

m_2(Information_P) = Ville_E.

m_3

m_3(Pion) = Enseignant,

m_3(Nom_P) = Nom_E,

m_3(Confidentiel_P) = Confidentiel_E,

m_3(CodePion_P) = Appréciation_E,

m_3(Information_P) = Divers_E.

m_4

m_4(Pion) = Enseignant,

m_4(Nom_P) = Nom_E,

m_4(Confidentiel_P) = Confidentiel_E,

m_4(CodePion_P) = NumSs_E,

m_4(Information_P) = Divers_E.

On peut définir les classes d'équivalence suivantes sur les composants des types Pion et Enseignant :

- [Nom_P] = {Nom_P, Nom_E, Information_P, Rue_E, Ville_E et Divers_E}.
- [CodePion_P] = {CodePion_P, Appréciation_E, NumSs_E}.

Par définition, la notion de classe d'équivalence suggère de choisir n'importe lequel des éléments de [CodePion_P] − {CodePion_P} pour adopter les instances de type CodePion_P. Il en est de même des types Nom_P et Information_P en regard de l'ensemble [Nom_P] − {Nom_P, Nom_E, Information_P}. Dans cet exemple, on voit d'après les formules 5, 6, 7 et 8 que le massif de dispersion minimale est m_1 :

$$d(\text{Pion}, m_1, \text{Enseignant}) = 1 + 3 + 4 + 5 + 7 = 20 \quad (5)$$

$$d(\text{Pion}, m_2, \text{Enseignant}) = 1 + 3 + 4 + 5 + 8 = 21 \quad (6)$$

$$d(\text{Pion}, m_3, \text{Enseignant}) = 1 + 3 + 4 + 5 + 14 = 27 \quad (7)$$

$$d(\text{Pion}, m_4, \text{Enseignant}) = 1 + 3 + 4 + 13 + 14 = 35 \quad (8)$$

VI.6.4 Ordre quotient

Rappelons qu'un document est une forêt d'arbres construits sur l'ensemble T des types d'un schéma de structure. Rappelons aussi que la relation *être un massif de* notée $\perp$ est une relation d'ordre partiel sur T et que la relation d'isomorphisme notée $\equiv$ est une relation d'équivalence sur T. L'idée est d'étendre les relations aux structures déjà construites sur T, notamment les classes d'équivalence.

Soit le triplet (T, $\equiv$, $\perp$) un ensemble de types, partiellement ordonné par la relation *être un massif de*, et muni de la relation d'équivalence *avoir la même définition que*. L'ensemble T quotienté par la relation d'équivalence $\equiv$ est noté [T] = {[t] | si t $\in$ T, sa classe d'équivalence est [t]}.

L'ordre quotient, en considération de l'ensemble des relations construites sur T, permet de dire si étant donné deux types t et t' tels que [t] $\neq$ [t'], t peut être transformé en t' ou t' peut être transformé en t. Cette démarche est faite en considérant l'ensemble quotient [T] muni d'une relation d'ordre partiel appelée ordre quotient qui compare des classes d'équivalence entières. Il s'agit d'identifier les relations avec lesquelles la relation d'équivalence est compatible. Dans ce qui suit, il va être démontré que la relation d'équivalence est compatible avec la relation d'ordre *être un massif* ; c'est−à−dire si u' $\in$ [u] et t' $\in$ [t] et u' $\perp$ t' alors $\forall$u' $\in$ [u], $\forall$t' $\in$ [t], u' est transformable en t'.

Définition

La relation $\equiv$ est dite compatible avec l'ordre $\perp$ si la propriété suivante est respectée :

$$\forall [t], [u] \in [T], [u] \perp [t] \Leftrightarrow \forall u' \in [u], \exists t' \in [t] \text{ tels que } u' \perp t' \qquad (9)$$

La formule (9) est une relation d'ordre (appelée ordre quotient) sur l'ensemble quotient [T]. Il va être démontré que :

Lemme

La relation d'isomorphisme construite sur un schéma de structure est compatible avec l'ordre *être un massif de* ($\perp$) construite sur le même schéma de structure.

Réflexivité

elle est évidente : $[u] \perp [u]$.

Antisymétrie

Il s'agit de démontrer l'antisymétrie de la relation (9).

Soit $[u] \perp [t] \Leftrightarrow \forall u' \in [u], \exists t'' \in [t]$ tel que $u' \perp t''$. (A)

Soit l'hypothèse suivante :

$[t] \perp [u] \Leftrightarrow \forall t' \in [t], \exists u' \in [u]$ tel que $t' \perp u'$. (B)

De (A), on a $\exists t'' \in [t]$, tel que $u' \perp t''$.

Puisque $t'' \equiv t'$ on en déduit $u' \perp t'$. (C)

En rapprochant (B) et (C) on a $t' \perp u'$ et $u' \perp t'$. Puisque $\perp$ est une relation d'ordre, on en déduit que $u' \equiv t'$, donc $[u'] = [t']$ et comme $[u'] = [u]$ et $[t'] = [t]'$, on conclut que $[u] = [t]$.

Pour confirmer cette démonstration, soient L(t') la longueur de la chaîne de t', L(u') celle de u. La relation $u' \perp t'$ implique que $L(u') < L(t')$. $t' \perp u'$ supposerait $L(t') < L(u')$, ce qui est impossible sauf si $t' = u'$.

Transitivité

Soient [a], [b], [c] trois classes d'équivalence de [T] ; supposons $[a] \perp [b]$ et $[b] \perp [c]$. Il en résulte qu'il existe quatre types $a_1 \in [a]$, $b_1 \in [b]$, $b_2 \in [b]$, $c_1 \in [c]$ tels que $a_1 \perp b_1$ et $b_2 \perp c_1$ avec $b_1 \equiv b_2$. Soit φ l'isomorphisme de b_1 sur b_2. Soit $m : a_1 \rightarrow b_1$ telle que a_1 est un massif b_1. $m(a_1) \in b_1$. $\varphi(m(a_1))$ est un massif de b_2 (puisque $b_1 \equiv b_2$) et est isomorphe à a_1. Comme $b_2 \perp c$ alors $\varphi(m(a_1)) \perp c_1$ puisque $b_2 \perp c_1$. Par transitivité de $\perp$, $\varphi(m(a_1)) \perp c_1$ et puisque $[a_1] = [a]$ et $[c_1] = [c]$ alors $[a] \perp [c]$.

Conclusion

Étant donné le triplet $(T, \equiv, \perp)$, et la compatibilité exprimée plus haut, le domaine des valeurs de la fonction de conversion des éléments d'une classe d'équivalence [u] ne se limite plus à [u], mais est étendu à [t], pourvu que l'on ait la relation $[u] \perp [t]$.

Cette possibilité reste valable pour les triplets suivants :

- (T, ≡, ↵) : en effet, étant donné qu'un facteur est un cas particulier de massif, on peut conclure que la relation ≡ est compatible avec la relation ↵.
- (T, ≡, [).
- (T, ≡, ◊)

Remarque

Les classes d'équivalence telles que définies en VI.6.1.1 supposent une équivalence structurale. Il faut envisager l'existence d'autres relations d'équivalence qui permettent de tirer profit des analyses statiques lorsqu'on est confronté aux transformations dynamiques entre documents de schémas différents. Par exemple considérons les relations suivantes :

- *être facteur d'un même type que*,
- *être compatible avec le même type que.*

Il faut remarquer que les éléments de ces classes d'équivalence ne sont pas transformables entre eux. Leurs intérêts résident dans l'ordre quotient et les conclusions qu'on en tire pour les transformations dynamiques entre documents de schémas différents.

VI.7 Extensions basées sur le contenu

Deux types peuvent n'entretenir aucune des relations décrites jusqu'à maintenant. Lorsque les relations d'homonymie, d'équivalence, de facteur, de sous−type et de massif de type échouent, d'autres moyens doivent être mis en œuvre pour tenter de récupérer au moins partiellement le contenu du sous−arbre à transformer. Le but de cette section est de définir un langage associé aux types et de s'en servir à des fins de transformation.

```
Diplôme = CASE OF
                Universitaire = TEXTE;
                Professionnel = TEXTE;
          END;

Vacataire = BEGIN
              Photo = IMAGE;
              Nom = TEXTE;
              Affiliations = LIST OF(Affiliation);
               ?Diplôme;
            END  −(Professionnel);
Affiliation = CASE OF TEXTE RefAff END;
RefAff = REFERENCE (Affiliation);
```

Fig. 6.34 : Type Vacataire

Soit le type Vacataire (voir Fig. 6.34) tiré d'un contexte où le type Diplôme est utilisé dans d'autres définitions. Vacataire n'est ni un facteur, ni un sous−type, ni une équivalence

du type Enseignant de la figure Fig. 6.20. Il n'en est pas non plus un massif parce que son fils Diplôme ne satisfait pas aux conditions nécessaires. Les instances de type Vacataire ne sont donc pas transformables en celles de type Enseignant par les mécanismes présentés jusqu'ici. Un autre mécanisme basé sur la notion de langage, va être mis en œuvre pour satisfaire les requêtes de restructuration. Il s'agit de considérer le schéma de structure comme une grammaire générant un langage. Ce qui est valable pour un schéma de structure l'est aussi pour n'importe quel type défini dans ce schéma. La transformation de structure dans ce cas reviendra à faire une comparaison entre les langages des types source et destination. En réalité, il s'agira de comparer le mot effectif d'une instance source, aux mots du langage propre à un type destination. Le mot effectif d'une instance appartient au langage engendré par son type générique. L'existence d'une seule correspondance suffit à conclure à la faisabilité de la transformation, au pire à la récupération du contenu de l'instance source. Cette section ne vise pas la construction de relations entre langages. Elle se contente de démontrer l'existence d'une autre possibilité de transformation, lorsque celles fondées sur la structure ont échoué.

VI.7.1 Langage associé à un type

On considère qu'un type est défini au moyen d'un ensemble X de terminaux et d'un ensemble V de non−terminaux encore appelés variables. Les variables sont les types construits alors que les terminaux sont les types de base du système. Nous allons montrer que le langage généré par un type est un langage algébrique (voir [Autebert87]) afin de disposer d'un cadre connu. Pour ce faire, on associe à chaque type t, une grammaire algébrique G_t construite à partir de la définition en langage S de t. Cette étude est basée sur le type Vacataire de la figure Fig. 6.34 dont la grammaire associée est donnée en Fig. 6.36 ainsi que sur le type Enseignant de la figure Fig. 6.20.

Par souci de clarté, le passage du langage S à la notation algébrique va se faire en deux étapes. La première n'intègre pas les exceptions (inclusions, exclusions) mais comporte la réécriture des définitions en fonction des constructeurs et de l'indicateur d'option. Elle donne pour résultat la grammaire $G0_t$. La deuxième étape modifie $G0_t$ en fonction des exceptions et fournit la grammaire G_t.

Réécriture sans les execptions

Réécriture en fonction des constructeurs :

- Retirer les mots−clés.
- Les composants déclarés entre les mots−clés BEGIN et END sont mis les uns à la suite des autres dans l'ordre de leur déclaration et séparés par un espace.
- Les composants déclarés entre les mots−clés AGGREGATE et END sont mis les uns à la suite des autres dans l'ordre de leur déclaration et séparés par un espace.

- À un type de constructeur Choix correspondent autant de règles que d'options, telles que chaque règle est de la forme *type* → *Option* :

  $t \rightarrow f_i(t)$ avec $i \in [1 . . \#Fils(t)]$ tel que $f_i(t) \in Fils(t)$

- À un type t de constructeur Choix correspond une règle particulière qui génère le mot vide :

  $t \rightarrow \varepsilon$

- À un type t de constructeur Liste correspondent deux règles ainsi définies :

  1. $t \rightarrow Fils(t)$
  2. $t \rightarrow t\ Fils(t)$

  Il faut noter que constr(t) = Liste $\Rightarrow$ #Fils(t) = 1.

Réécriture des types optionnels :

La réécriture d'un type optionnel t nécessite l'application des règles suivantes :

- $t \rightarrow \varepsilon$
- Réécrire t en fonction de son constructeur

```
Vacataire → Photo Nom Affiliations Diplôme
Photo → 'IMAGE'
Nom → 'TEXTE'
Affiliations → Affiliation
Affiliations → Affiliations Affiliation
Affiliation → 'TEXTE'
Affiliation → RefAff
Affiliation → 'ε'
RefAff → 'REFERENCE'
Diplôme → Universitaire
Diplôme → Professionnel
Diplôme → 'ε'
Universitaire → 'TEXTE'
Professionnel → 'TEXTE'
```

Fig. 6.35 : Grammaire $G0_{Vacataire}$

Prise en compte des exceptions

À un type t défini avec le langage S, correspond au moins une règle de production dans la grammaire $G0_t$. Le symbole le plus à gauche dans une règle de production, est l'identificateur du type tel que défini avec le langage S. La prise en compte des

exceptions se fait en deux étapes telle que la première intègre les exclusions et la deuxième les inclusions.

Intégration des exclusions :

Soit t le type dont on réécrit la définition. À ce type correspond l'arbre (F, q) de racine t. $\forall U \in F$, $\forall Z \in$ excl(U) les règles suivantes sont appliquées :

1. Si $Z \in$ Fils(U) supprimer Z de Fils(U).
2. soit $W \rightarrow \alpha\ Z\ \beta$ la production de $G0_t$ telle que $U \overset{+}{\rightarrow} \pi_1\ Z\ \pi_2$.
    1. Dupliquer toutes les productions de la forme $W \rightarrow \Pi$ où Π est une protophrase quelconque.
    2. Donner un même identificateur inédit W~ aux productions dupliquées.
    3. Supprimer Z de la production $W\sim \rightarrow \alpha\ Z\ \beta$.
    4. Repérer la production $V \rightarrow \pi_3\ W\ \pi_4$
    5. Dupliquer toutes les productions de la forme $V \rightarrow \Pi$ où Π est une protophrase quelconque.
    6. Donner un même identificateur inédit V~ aux productions dupliquées.
    7. Répéter les étapes 3, 4, 5 en substituant V à W jusqu'à la production $U \rightarrow \pi_5\ V\ \pi_6$ qui ne doit pas être traitée.
    8. Remplacer V par V~.
3. Répéter les opérations 1 et 2 tant que $U \overset{+}{\rightarrow} \pi_7\ Z\ \pi_8$.

Soit M l'ensemble des non terminaux de la grammaire non réduite $G0_t$. Cette grammaire doit être réduite vis à vis de t en satisfaisant les conditions suivantes :

- $\forall U \in M$, $L_{G0t}(U) \neq \varnothing$: aucun non terminal n'engendre un langage vide.
- $\forall U \in M$, $\exists\ \alpha, \beta \in (X \cup V)^*$ tels que $\alpha U \beta$ appartienne au langage élargi $\hat{L}_{G0t}(t) = \{v \in (X \cup V)^* \mid t \overset{*}{\rightarrow} v\}$: toutes les variables peuvent apparaître par dérivation à partir de t.

L'application de l'intégration des exclusions à $G0_t$ donne $G1_t$.

Intégration des inclusions :

Les inclusions n'ont pas de position structurale fixe, ne sont pas limitées en nombre et ne sont pas obligatoires. De manière informelle, il s'agit d'insérer dans $G1_t$ un symbole représentant une liste optionnelle, en fonction des constructeurs. Leur intégration modifie profondément la grammaire et s'obtient en appliquant la procédure suivante $\forall U \in$ incl(t) :

1. Lorsque la partie droite d'une production comporte plus d'un symbole et n'appartient pas à une règle récursive, un non terminal V est inséré avant et après chacun des symboles qui la composent.
2. Pour un groupe de règles de la forme $A \rightarrow \Pi_i$ et qui représente une alternative, ajouter les règles suivantes :
   - $A \rightarrow V$
   - $A \rightarrow V\ \Pi_i\ V$.
3. Pour une règle récursive de la forme $A \rightarrow A\ B$ ajouter deux règles de la forme :
   - $A \rightarrow A\ V$.
   - $A \rightarrow V$.
4. Transformer une règle de la forme $A \rightarrow B$, telle qu'il n'existe pas d'autres règles avec A en partie gauche, en $A \rightarrow V\ B\ V$.
5. Dériver V comme une liste optionnelle en produisant les trois règles suivantes :
   - $V \rightarrow \varepsilon$.
   - $V \rightarrow U$.
   - $V \rightarrow VU$.

L'application de l'intégration des inclusions à $G1_t$ donne G_t.

```
Vacataire → Photo Nom Affiliations Diplôme~
Photo → 'IMAGE'
Nom → 'TEXTE'
Affiliations → Affiliation
Affiliations → Affiliations Affiliation
Affiliation → 'TEXTE'
Affiliation → RefAff
Affiliation → 'ε'
RefAff → 'REFERENCE'
Diplôme~ → Universitaire
Diplôme~ → 'ε'
Universitaire → 'TEXTE'
```

Fig. 6.36 : Grammaire $G_{Vacataire}$

Le résultat de la transcription d'une définition en S, par application des procédures sus−indiquées, est une grammaire algébrique G = <X, V, P, D> définie comme suit :

1. $D \in V$ est l'axiome de la grammaire ; par exemple D = Vacataire.
2. X, l'union de l'ensemble β des types de base et des schémas de structure du système, joue ici le rôle d'alphabet dit terminal.

3. V est l'ensemble des variables telles que $V \cap X = \varnothing$. Par exemple, V = {Vacataire, Photo, Nom, Affiliations, Affiliation, Diplôme~, Universitaire}.
4. P est l'ensemble des règles de production telles que P est un sous−ensemble du produit fini $V \times (X \cup V)^*$.

Le langage associé au type t est noté $L_G(t) = \{b \in X^* \mid t \overset{*}{\Rightarrow} b\}$, comme L_G(Vacataire) auquel appartient le mot IMAGE.TEXTE. Un mot est normalement une suite finie et ordonnée des éléments de X obtenus à partir des feuilles de l'arbre des types. Etant donnée l'importance de l'ordre des éléments dans la récupération du contenu des instances, il est nécessaire de prouver que les feuilles sont ordonnées par l'ordre préfixe. La nécessité d'une telle démonstration est due au fait que la méthode de récupération retenue est fondée sur l'ordre préfixe, lequel suppose que l'ensemble des feuilles est ordonné.

VI.7.1.1 Ordre linéaire des feuilles dans un arbre

Définition

Les feuilles d'un arbre sont linéairement ordonnées par l'ordre préfixe.

La première feuille rencontrée dans un parcours préfixe d'un arbre de racine t est ω(t−chaîne). Par définition, $PFX_i(t)$ conserve l'ordre des descendants de t. Si tous les descendants de t sont des feuilles, la démonstration est triviale. Nous supposons que les descendants ne sont pas terminaux. Supposons que $PFX_i(t)$ ordonne les feuilles jusqu'au $i^{ème}$ descendant ; $\omega(PFX_i(t))$ est alors la dernière feuille de l'ordre préfixe sur t incluant le $i^{ème}$ fils f_i de t. A ce niveau, si f_i est une feuille alors $f_i = \omega(PFX_i(t))$ et $\omega(PFX_{i-1}(t)) <_t f_i$ par construction. Si f_i n'est pas une feuille, on a $\omega(PFX_{i-1}(t)) <_t \omega(PFX_i(t))$ par construction. La démonstration de l'ordre incluant f_{i+1} est triviale, que f_{i+1} soit une feuille ou non. Si f_{i+1} est une feuille alors $PFX_1(f_{i+1}) = f_{i+1}$−chaîne = f_{i+1} et $RANG(f_{i+1}) = RANG(\omega(PFX_i(t))) + 1$. Si f_{i+1} n'est pas une feuille, $PFX(f_{i+1})$ est une chaîne et $\omega(PFX_1(f_{i+1}))$ est une feuille et par définition, la fonction RANG attribue à tous les éléments de $PFX_1(f_{i+1})$ notamment à $\omega(PFX_1(f_{i+1}))$ un numéro d'ordre supérieur à ceux des éléments de $PFX_i(t)$.

VI.7.1.2 Particularité du problème

L'objectif de la présente partie est de calculer le mot effectif d'une instance et de le faire reconnaître par la grammaire du type destination. La reconnaissance du mot effectif par un automate n'est pas retenue pour les raisons suivantes :

1. À une grammaire algébrique correspond un automate à pile qui reconnaît le langage qu'elle engendre. Il faut garder en mémoire que les non terminaux des grammaires algébriques issues des schémas de structure ont une signification sémantique pour l'utilisateur. Or les opérations de normalisation et de mise sous forme de Greibach d'un automate sont de nature à transformer la structure d'origine exprimée par l'utilisateur. Il serait souhaitable en cas de succès, de dire à quel sous−arbre

correspond le mot effectif reconnu. A titre d'exemple considérons le couple d'évolution (Vacataire, Enseignant) et l'instance de type Vacataire donnée en Fig. 6.37. Le mot "INRIA" de l'exemple, généré par le type Affiliation pourrait être adopté par une instance de n'importe lequel des types suivants : Rue_E, Ville_E, Divers_E. Le résultat attendu est que le mot en question soit adopté par une instance de type Affiliation_E. Il paraît nécessaire, non seulement de garder le type Affiliation_E, mais aussi de disposer de plus d'information, afin de proposer des solutions aussi proches que possible de celles attendues par l'utilisateur.

2. Étant donné un mot du langage destination, on cherche à savoir si le mot source lui est équivalent ou bien s'il en est un sous−mot ou un facteur. L'usage des automates restreindrait ces objectifs à la seule reconnaissance d'un mot en tant qu'entité.
3. Une instance pourrait ne pas comporter des sous−instances correspondant aux types optionnels. Elle pourrait aussi ne comporter aucune sous−instance correspondant aux options d'un choix. Elle pourrait enfin ne comporter aucun élément de liste. Dans ces cas là, le mot vide est explicitement généré et fait partie du document. La présence d'un mot vide peut aider à rapprocher un mot des variables susceptibles de les générer.

La solution retenue est de représenter le langage correspondant à un type par une chaîne de caractères comportant outre les éléments de l'alphabet, des *méta−caractères* qui expriment ses caractéristiques structurales : c'est l'*empreinte générique* d'un type. L'ensemble M des méta−caractères est constitué des caractères suivants : {, }, ↑, ↓, [,], (,), +, <, >. Étant donné un type t, G_t n'a pas inclus les méta−caractères, non seulement par souci de clarté, mais surtout pour mettre en évidence l'importance de la structure, même dans cette approche basée sur le contenu. Pour garder la trace de la structure, la première étape de la transcription d'une définition en S devient :

- Retirer les mots−clé.
- Remplacer le caractère '=' par '→'.
- Une accolade ouvrante est placée avant le premier composant d'un type de constructeur Agrégat ordonné et une accolade fermante est placée après le dernier composant. Le caractère '+' sépare deux composants.
- Une flèche montante est placée avant le premier composant d'un type de constructeur Agrégat sans ordre et une flèche descendante est placée après le dernier composant. Le caractère '+' sépare deux composants.
- Un crochet ouvrant est placé avant la première option d'un type de constructeur Choix et un crochet fermant est placé après la dernière option. Le caractère '+' sépare deux options.
- Les éléments de liste sont placés entre parenthèses.

Le résultat de l'application de ces nouvelles règles de transcription sur le type Vacataire donne le résultat suivant :

```
Vacataire → '{' Photo '+' Nom '+' Affiliations
            '+' Diplôme~ '}'
Photo → 'IMAGE'
```

```
Nom → TEXTE
Affiliations → '(' Affiliation ')'
Affiliation → '[' 'TEXTE' '+' RefAff '+' 'ε' ']'
RefAff → 'REFERENCE'
Diplôme~ → '[' Universitaire '+' ε ']'(4)
Universitaire → 'TEXTE'
```

Une empreinte générique E est interprétée comme l'expression rationnelle d'un langage élargi (aux méta−caractères) et c'est ce point de vue qui est adopté dans la suite de l'exposé :

$$E \in (\beta \cup \lambda \cup M)^*$$

où λ est l'ensemble des schémas de structure du système, M l'ensemble des méta−caractères exprimant la structure et β l'ensemble des types de base du système. L'interprétation d'une empreinte, d'une part comme une expression rationnelle et d'autre part comme une structure du langage S est fondée sur les considérations suivantes :

- La paire de parenthèses exprime l'itération de l'élément qu'elle encadre. Elle correspond à une règle récursive dans une grammaire algébrique et au constructeur Liste dans le langage S.
- La paire de crochets regroupe les options d'un choix. elle correspond à une règle alternative dans la grammaire algébrique et au constructeur Choix dans le langage S.
- La paire d'accolades regroupe les composants d'un agrégat.
- Les éléments de structure d'une empreinte sont ceux rencontrés lors du parcours préfixe de l'arbre du type correspondant.

L'empreinte générique caractérise un type et correspond à la famille des instances qu'il peut générer. À une instance correspond une *empreinte effective* dont la compatibilité avec son empreinte générique est garantie par l'éditeur. Dans une requête de restructuration dynamique, une empreinte effective est construite pour l'instance à transformer et une empreinte générique pour le type destination. La comparaison d'une empreinte effective à une empreinte générique, et celle de l'empreinte générique du type de l'élément source à celle du type destination, devra donner des solutions très satisfaisantes.

---

(4) La grammaire aurait pu être optimisée en interprétant cette règle, non comme un choix, mais comme un élément optionnel.

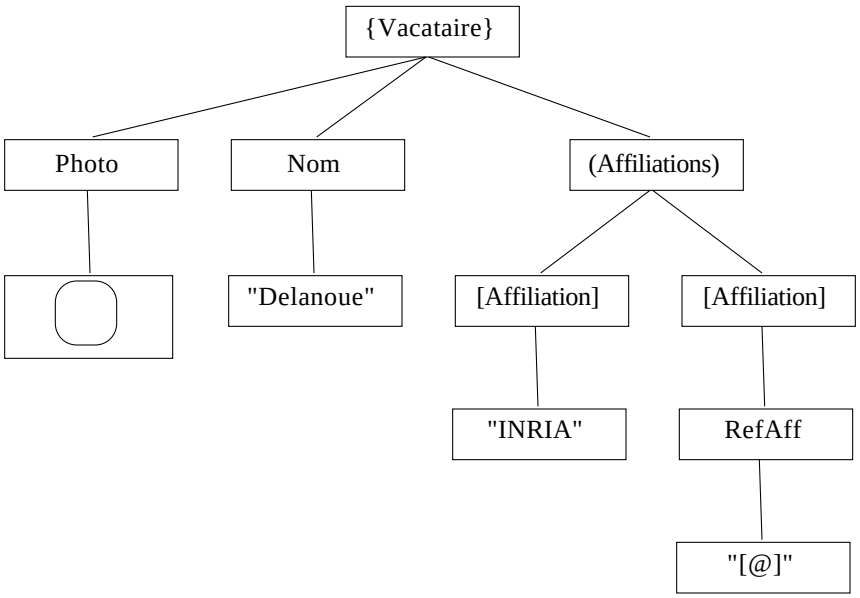


Fig. 6.37 : **Instance** *de type Vacataire*

VI.7.2 Construction des empreintes

Cette sous−section explique la construction des empreintes génériques et effectives.

Avertissement

Dans les empreintes, un type de base est représenté par la première lettre de son identificateur :

T	pour TEXTE
R	pour REFERENCE
I	pour IMAGE
S	pour SYMBOLE
G	pour GRAPHIQUE

VI.7.2.1 Empreinte générique

La construction d'une empreinte E_t est basée sur le parcours préfixe de l'arbre du type t et s'obtient par application de l'algorithme suivant.

```
EmpreinteGénérique(E, i, t, Z)
/*============================================================
E empreinte du type t, est un tableau de caractères. i indice
du prochain caractère à insérer, est passé par référence. Z est
le mot formé par les feuilles de l'arbre de t, récupérées dans
l'ordre préfixe.
============================================================*/

SI(Constructeur(t) == SchemaNature))
```

```
ALORS
    E[i++] = '<'
    Concaténer(E,  Nom(Schema2Structure))
    i  +=  longueur(Nom(Schema2Structure))
    E[i++] '>'
SINON
    SI(Constructeur(t) == Liste)
    ALORS
        E[i++] = '('
    SINON
        SI(Constructeur(t) == Choix)
        ALORS
            E[i++] = '['
        SINON
            SI(Constructeur(t) == Agrégat ordonné)
            ALORS
                E[i++] = '{'
            SINON
                SI(Constructeur(t) == Agrégat sans ordre)
                ALORS
                    E[i++] = '↑'
                SINON
                    SI(Constructeur(t) == Identité)
                    ALORS
                        Z[i] = t
                        E[i++] = fils(t)
                    FINSI
                FINSI
            FINSI
        FINSI
    FINSI
    pfils = PremierFils(t)
    TANTQUE(pfils)
        EmpreinteGénérique(E, pfils, i, Z)
        SI(pfils != Premier(Fils(t))
        ALORS
            E[i++] = '+'
        FINSI
        pfils = Suivant(pfils)
    FINTANTQUE
FINSI
SI(Constructeur(t) == Liste)
ALORS
    E[i++] = ')'
SINON
    SI(Constructeur(t) == Choix)
    ALORS
        E[i++] = ']'
    SINON
        SI(Constructeur(t) == Agrégat ordonné)
        ALORS
            E[i++] = '}'
        SINON
            SI(Constructeur(t) == Agrégat sans ordre)
            ALORS
                E[i++] = '↓'
            FINSI
```

```
        FINSI
    FINSI
FINSI
```

VI.7.2.2 Exemples d'empreintes génériques

Les figures Fig. 6.38, Fig. 6.39 et Fig. 6.40 sont les résultats de l'application de l'algorithme de calcul de l'empreinte générique sur les types Vacataire (voir Fig. 6.34), Enseignant (voir Fig. 6.19) et Section (voir Fig. 6.23).

{I+T+([T+R])+[T]}

Fig. 6.38 : Empreinte générique du type Vacataire

{I+T+{S+{T+T}+([T+R])+S+T}}

Fig. 6.39 : Empreinte générique du type Enseignant

{T+(<Paragraphe>)+({T+(<Paragraphe>)+({T+(<Paragraphe>)})})}

Fig. 6.40 : Empreinte générique du type Section

VI.7.2.3 Empreinte effective

L'empreinte effective est une particularisation de l'empreinte générique. Son calcul est basé sur le parcours simultané d'une instance et de l'empreinte générique du type de l'instance. La motivation du calcul de l'empreinte effective est le fait qu'une instance peut n'être générée que par une partie de son type générique. Cette possibilité est exprimée dans la définition des types par la présence des types optionnels (type déclaré explicitement optionnel ou option de choix). Alors, si t est le type de l'instance, l'empreinte effective peut être vue comme la concaténation de l'empreinte des parties de t qui ont contribué à la production de l'instance.

Dans une empreinte effective, les types qui n'ont pas participé à la production de l'instance dont on calcule l'empreinte, sont représentés par la chaîne vide ε entourée de la paire de méta−caractères correspondant à leur constructeur.

```
EmpreinteEffective(E, s, i)

/*=========================================================
E est un tableau de caractères, i indice du prochain caractère
            à insérer, est passé par référence.
=========================================================*/

t = TYPE(s)
SI(Constructeur(s) == SchemaNature))
ALORS
    E[i++] = '<'
    Concaténer(E,  Nom(Schema2Structure))
    i += longueur(Nom(Schema2Structure))
    E[i++] '>'
```

```
SINON
    SI(Constructeur(t) == Liste)
    ALORS
        E[i++] = '('
    SINON
        SI(Constructeur(t) == Choix)
        ALORS
            E[i++] = '['
        SINON
            SI(Constructeur(t) == Agrégat ordonné)
            ALORS
                E[i++] = '{'
            SINON
                SI(Constructeur(t) == Agrégat sans    ordre)
                ALORS
                    E[i++] = '↑'
                SINON
                    SI(t ∈ β)
                    ALORS
                        E[i++] = t
                    FINSI
                FINSI
            FINSI
        FINSI
    FINSI
    pfils = PremierFils(s)
    u = Premier(Fils(t))
    TANTQUE(pfils)
        TANTQUE(u != TYPE(pfils)
            SI(Constructeur(u) == Choix)
            ALORS
                E[i++] = '['
            SINON
                SI(Constructeur(u) == Liste)
                ALORS
                    E[i++] = '('
                SINON
                    SI(Constructeur(u) == Agrégat    ordonné)
                    ALORS
                        E[i++] = '{'
                    SINON
                        SI(Constructeur(u) == Agrégat    sans
                        ordre)
                        ALORS
                            E[i++] = '↑'
                        FINSI
                    FINSI
                FINSI
            FINSI
            E[i++] = 'ε'
            SI(Constructeur(u) == Choix)
            ALORS
                E[i++] = ']'
            SINON
                SI(Constructeur(u) == Liste)
                ALORS
                    E[i++] = ')'
```

```
                    SINON
                        SI(Constructeur(u) == Agrégat    ordonné)
                        ALORS
                            E[i++] = '}'
                        SINON
                            SI(Constructeur(u) == Agrégat    sans
                            ordre)
                            ALORS
                                E[i++] = '↓'
                            FINSI
                        FINSI
                    FINSI
                FINSI
                SI(u != Premier(Fils(t))
                ALORS
                    E[i++] = '+'
                FINSI
                u = Suivant(Fils(t), u)
            FINTANTQUE
            EmpreinteEffective(E, pfils, i)
            pfils = ElementSuivant(pfils)
            SI(pfils && pfils != Premier(Fils(t))
            ALORS
                E[i++] = '+'
            FINSI
            u = Suivant(Fils(t), u)
        FINTANTQUE
    FINSI
    SI(Constructeur(t) == Liste)
    ALORS
        Retirer les motifs redondants de la liste
        E[i++] = ')'
    SINON
        SI(Constructeur(t) == Choix)
        ALORS
            Retirer les motifs redondants du choix
            E[i++] = ']'
        SINON
            SI(Constructeur(t) == Agrégat ordonné)
            ALORS
                E[i++] = '}'
            SINON
                SI(Constructeur(t) == Agrégat sans ordre)
                ALORS
                    E[i++] = '↓'
                FINSI
            FINSI
        FINSI
    FINSI
```

VI.7.2.4 Exemples d'empreintes effectives

L'empreinte effective de la figure Fig. 6.41 est le résultat de l'application de l'algorithme de construction d'empreinte effective sur l'instance de type Vacataire donnée en Fig. 6.42.

{I+T+([ε])+[T]}

Fig. 6.41 : Empreinte effective d'une instance de type Vacataire

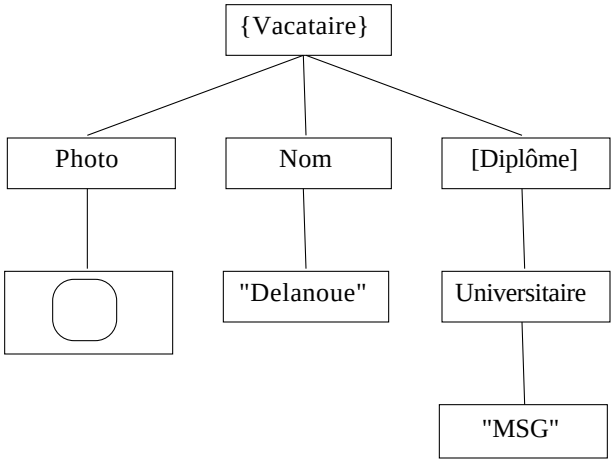


Fig. 6.42 : **Instance** *de type Vacataire sans l'élément de type Affiliations*

L'empreinte effective de la figure Fig. 6.43 illustre l'application de l'algorithme de construction d'empreinte effective sur l'instance de type SousSection donnée en Fig. 6.44.

{T+(<Paragraphe>)+({T+(<Paragraphe>)})}

Fig. 6.43 : Empreinte effective d'une instance de type SousSection

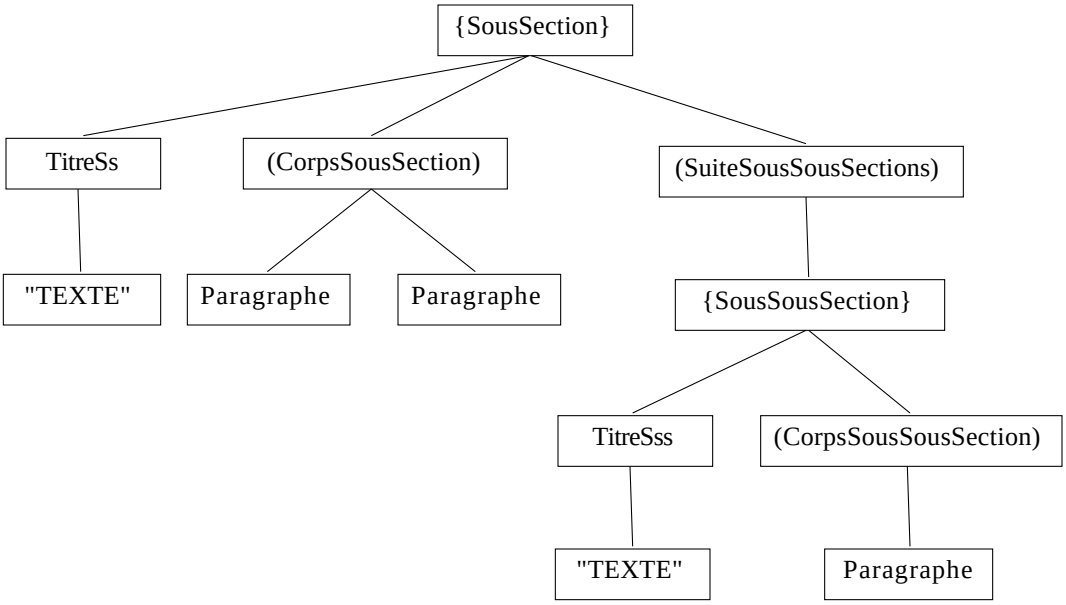


Fig. 6.44 : **Instance** *de type SousSection*

L’empreinte effective de la figure Fig. 6.45 est celle d’une instance incomplète de type SousSection (voir Fig. 6.46) dans laquelle ne figurent pas les éléments obligatoires de type CorpsSousSection et TitreSss.

{T+(ε)+({ε+(<Paragraphe>)})}

Fig. 6.45 : Empreinte effective d’une instance de type SousSection

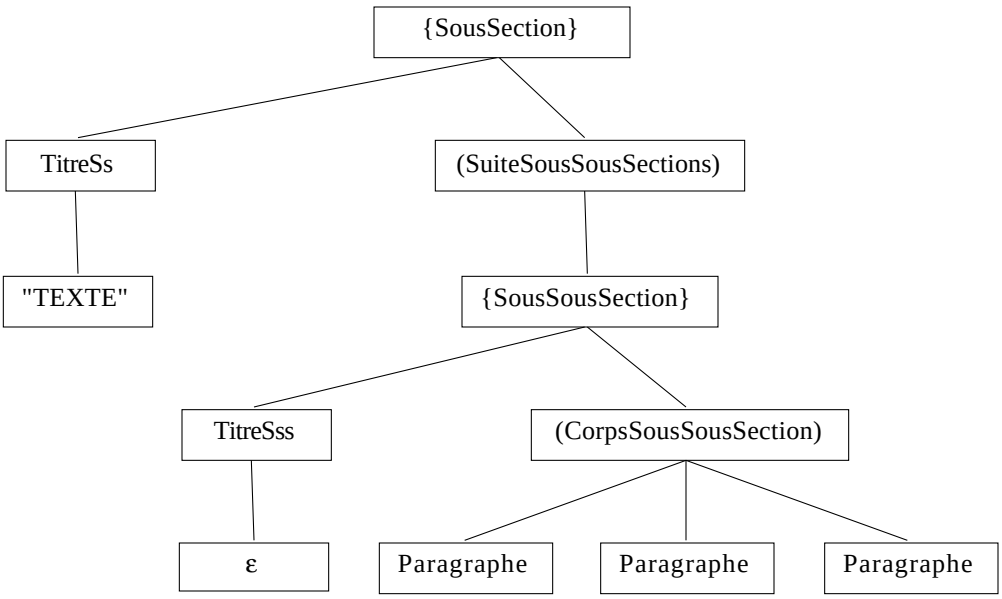


Fig. 6.46 : **Instance** *de type SousSection sans son élément de type CorpsSousSection*

VI.7.3 Récupération de l’arbre source

L’utilisation du langage associé aux types permet de conclure uniquement à la possibilité ou non de la restructuration. Pour rendre effective cette restructuration, il faut déterminer le sous−arbre d’accueil à l’intérieur de l’arbre destination. La connaissance de ce sous−arbre permet de faire un rapprochement avec les nœuds de l’arbre source dans le but de récupérer un maximum de structures. Par exemple, si un nœud t de l’arbre source et un nœud t’ de l’arbre destination sont tels que org(t) = org(t’), les sous−arbres dont ils sont racines sont identiques et la restructuration consiste alors à récupérer le sous−arbre source. Dans cette hypothèse, on récupère également les attributs sans aucune transformation.

Cependant toutes les restructurations dynamiques ne réunissent pas des conditions aussi favorables, l’identification du sous−arbre d’accueil et la récupération des attributs demeurant des problèmes auxquels ce sous−chapitre apporte une solution.

VI.7.3.1 Détermination du sous−arbre d’accueil

La détermination du sous−arbre d’accueil se fait grâce à la base Z calculée par l’algorithme de construction de l’empreinte générique donné en VI.7.2.1. La base contient les feuilles de l’arbre des types. L’objectif est de déterminer les ascendants des types de base qui

entrent dans la constitution de l'empreinte et de leur trouver un nœud commun, racine du plus petit sous−arbre auxquel ils appartiennent. La fonction p sert à récupérer les ascendants des types construits. Or les empreintes ne contiennent que des types de base en dehors des éléments de structure. Malheureusement, la fonction de précédence $p^{(5)}$ est définie sur un ensemble qui ne peut pas contenir β. Il n'est donc pas possible de reconnaître les ascendants à partir de l'empreinte générique. Heureusement, la plupart des feuilles, à l'exception des types récursifs et des schémas externes, sont définies comme identiques aux types de base. Le moyen retenu pour calculer les ascendants des types de base de l'empreinte générique passe par l'établissement d'une correspondance entre leurs occurrences dans E_t et les éléments qui leurs sont identiques dans Z puisque, contrairement aux types de base, ces derniers appartiennent au domaine de définition de p.

Correspondance entre base et empreinte générique

La correspondance est faite dans l'algorithme de calcul d'empreinte générique (voir VI.7.2.1) par la fonction g définie comme suit : à chaque occurrence b d'un type de base dans une empreinte générique, on fait correspondre le type z tel que $b \in$ Fils(z). Étant donné E_t, deux occurrences d'un même type de base s'y distinguent par leur position, permettant ainsi de construire $B \subset E_t$ tel que B peut contenir plus d'une occurrence d'un même type de base. Dans ces conditions, la fonction g est définie comme suit :

$$g : B \rightarrow Z \text{ tel que } \forall b \in B, b \in \beta \rightarrow b \in \text{Fils}(g(b)) \qquad (10)$$

Par exemple, si E_t et Z sont des tableaux, il suffit de placer z dans Z au même indice que $\text{Fils}(z)^{(6)}$ dans E_t. Il faut garder en mémoire que les éléments de Z sont distincts puisqu'ils appartiennent à la forme canonique d'un schéma de structure.

Identification du sous−arbre d'accueil

Étant donné une empreinte effective E^i, soit δ le sous−ensemble de Z dont les éléments sont obtenus par la fonction $g : B \rightarrow Z$. Il faut identifier le plus grand ancêtre (selon l'ordre total $<_t$) commun à tous les éléments de δ, en trois étapes :

1. Détermination de l'ensemble A(u) des ancêtres de chaque élément u de δ :

$$A(u) = \bigcup_{i=1}^{h(u)} p^i(u)$$

2. L'intersection de tous les $A(u_j)$ donne l'ensemble A_δ ordonné par $<_t$ des ancêtres communs aux éléments de δ :

$$A_\delta = \bigcap_{j=1}^{\#\delta} A(u_j)$$

---

(5) Il faut garder à l'esprit que la fonction de précédence p n'est applicable qu'aux types construits y compris les types récursifs et les types définis comme identiques aux schémas externes.

(6) #F(z) = 1

3. Les éléments de A_δ ordonné par $<_t$ forment une chaîne dont le plus grand élément est la racine recherchée du plus petit sous−arbre d'accueil :

$$\omega(A_\delta)$$

VI.7.3.2 Récupération des éléments de structure

Le sous−arbre d'accueil étant identifié, il faut faire un rapprochement entre ses nœuds et ceux de l'arbre source. Les cas suivants sont pris en considération :

1. Les éléments source et destination appartiennent au même schéma de structure. La fonction org permet alors de tester s'il y a homonymie de deux nœuds et de récupérer dans l'affirmative le sous−arbre source correspondant.
2. Lorsque les éléments source et destination n'appartiennent pas au même schéma de structure, la prise en compte des classes d'équivalence virtuelles et des ordres quotients s'impose.
3. Lorsque les types source et destination sont identiques à un même schéma externe, la récupération du sous−arbre source est immédiate.

VI.7.3.3 Récupération des attributs

Dans les cas de récupération d'éléments de structure, les attributs accompagnent les nœuds auxquels ils appartiennent, sans aucune transformation. Dans tous les autres cas, il faut tenter la récupération des attributs d'un nœud source par les nœuds de la chaîne des ancêtres correspondants dans l'arbre destination.

VI.7.4 Utilisation des empreintes

Les empreintes sont utilisées de deux façons différentes :

1. Comparaison d'une empreinte effective à une empreinte générique en vue d'une transformation dynamique ou d'un couper−coller.
2. Comparaison d'une empreinte générique à une autre empreinte générique en vue d'identifier la relation (voir VI.6) qui lie leurs types respectifs.

L'algorithme de comparaison accorde une plus grande priorité à la recherche des motifs représentant une structure, qu'aux types de base dans les empreintes.

Empreinte effective

L'algorithme utilisé pour la comparaison d'une empreinte effective et d'une empreinte générique est basé sur la comparaison de chaînes de caractères. Il recherche dans l'empreinte générique les motifs structuraux équivalents tels que la liste représentée par une paire de parenthèses, le choix représenté par une paire de crochets et l'agrégat représenté par la succession des symboles. En cas d'échec, la comparaison reprend avec les règles suivantes :

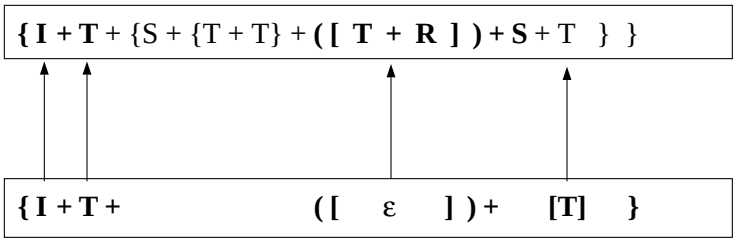
- La recherche de correspondance entre l'empreinte d'une option et celle d'un champ d'agrégat est autorisée.

- La recherche de correspondance entre l'empreinte d'un champ d'agrégat et celle d'une option est autorisée.
- La recherche de correspondance entre l'empreinte d'un champ d'agrégat et celle d'une option dont le choix est élément de liste, est autorisée.
- La recherche de correspondance entre l'empreinte d'un élément de liste et celle d'un champ d'agrégat est autorisée.

Exemples

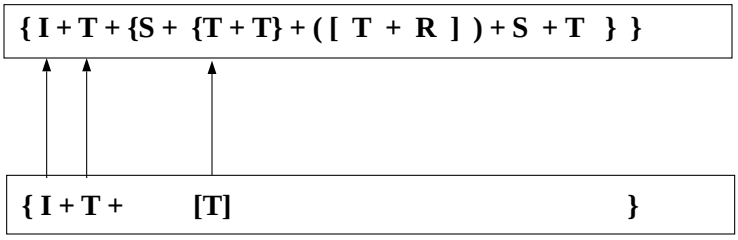
- La comparaison de l'empreinte effective {*I+T+([ε])+[T]}* à l'empreinte générique {I+T+{S+{T+T}+([T+R])+S+T}} du type Enseignant (voir Fig. 6.39), conclut à la transformation de l'instance de type Vacataire de la figure Fig. 6.42 en une instance de type Enseignant.

Empreinte générique du type Enseignant



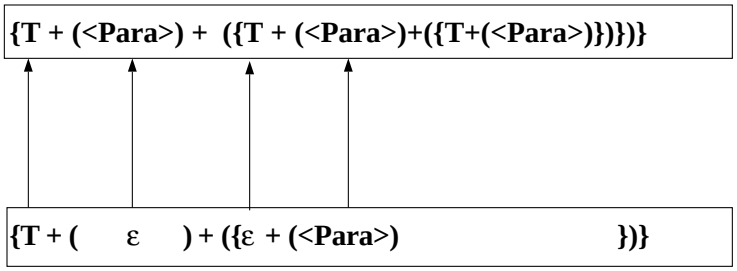
Empreinte effective d'une instance de type Vacataire

En l'absence de l'empreinte vide de l'élément absent, l'empreinte effective serait {*I+T+[T]}* et la comparaison donnerait le résultat suivant :



- La comparaison de l'empreinte effective *{T+(ε)+({ε+(<Paragraphe>)})}* à l'empreinte générique {T+(<Paragraphe>)+({T+(<Paragraphe>)+({T+(<Paragraphe>)})})} du type Section conclut à la transformation attendue de l'instance de type SousSection de la figure Fig. 6.46 en une instance de type Section. Le résultat de la comparaison illustrée ci−après, considère l'empreinte effective du type SousSection comme une sous−chaîne de la chaîne représentée par l'empreinte générique du type Section. Tout se passe comme si l'empreinte effective était celle d'une instance engendrée par un type, massif du type Section :

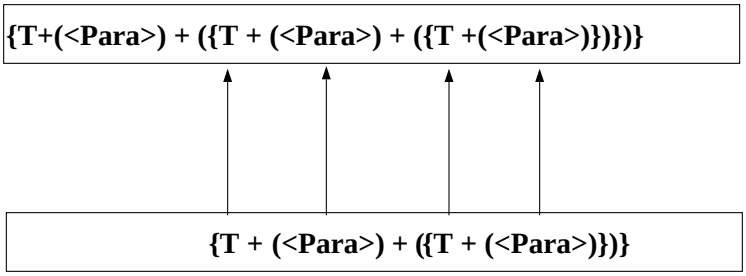
Empreinte générique du type Section



Empreinte effective d'une instance de type SousSection

- La comparaison suivante de l'empreinte effective *{T+(<Paragraphe>)+({T+(<Paragraphe>)})}* (voir figure Fig. 6.44) à l'empreinte générique du type Section illustre le fait que l'empreinte effective est un modèle (chaîne de caractères) appartenant à l'empreinte générique. Intuitivement, l'empreinte effective serait celle d'une instance engendrée par un type, facteur du type Section.

Empreinte générique du type Section



Empreinte effective d'une instance de type SousSection

Empreinte générique

La comparaison d'empreintes génériques permet d'établir la relation entre les types concernés (voir VI.8).

VI.8 Reconnaissance des structures

Les algorithmes de reconnaissance de structures sont fondés sur la comparaison d'empreintes génériques.

VI.8.1 Reconnaissance des équivalences

Les conditions de bijection, d'identité de constructeur, d'égalité de cardinal et d'identité de niveau ont permis de faire la démonstration de la transformation entre eux de deux types équivalents. On en déduit aisément la proposition et le théorème suivants qui serviront par la suite dans la reconnaissance des autres structures et relations :

Proposition

Deux types équivalents ont leurs empreintes génériques identiques.

Théorème

Lorsque deux types ont leurs empreintes génériques identiques, leurs instances sont identiques donc transformables entre elles.

Cette conclusion suppose que la forme canonique des définitions ne contienne pas de composants de constructeur Choix. Cela revient alors à produire dans une forme canonique autant d'arbres que d'options. Si tel n'est pas le cas, l'identité de structure entre deux types n'entraînerait pas celle de leurs instances et des contenus pourraient être irrécupérables. À titre d'exemple, on peut citer deux instances I_1, I_2 d'un type t comportant un composant u de constructeur Choix, définies comme suit :

- I_1 comporte un élément engendré par une option $v_1 \in$ Fils(u).
- I_2 comporte un élément engendré par une option $v_2 \in$ Fils(u) et $\exists b \in \beta$ tel que b appartient à l'empreinte effective de I_1 et b n'appartient pas à l'empreinte effective de I_2.

VI.8.2 Reconnaissance des facteurs

Rappel

Un mot E_u est un facteur d'un mot E si $E = e_1 E_u e_2$ tels que e_1 et e_2 ne sont pas tous deux simultanément égaux à ε. $e_1 = \varepsilon \rightarrow E_u$ est facteur gauche de E. $e_2 = \varepsilon \rightarrow E_u$ est facteur droit de E. $(e_1 \neq \varepsilon$ et $e_2 \neq \varepsilon) \rightarrow E_u$ est un facteur propre de E.

Proposition

Un type u est un facteur d'un type t si l'empreinte générique E_u de u est un facteur propre de l'empreinte générique E_t de t.

Preuve

Selon la définition d'un facteur de type, u est un facteur de $t \rightarrow \exists v \in$ D(t) tel que $u \equiv v$ et $E_t = e_1 E_u e_2$. La preuve est simple. La construction d'une empreinte est telle que les éléments de structure et de contenu des composants rencontrés avant et après v sont respectivement dans e_1 et e_2. Il existe au moins l'un des éléments de structure suivants : {, [, (, ↑

dans e_1, relatif au nœud racine t. Il en résulte que e_2 contient obligatoirement l'un des éléments de structure suivants : },],), ↓. Dans ces conditions, $E_t = (E_v)$ si constr(t) = Liste. Il apparaît donc que E_v est un facteur propre de la chaîne de caractères E_t.

Réciproquement, si v est un facteur de t alors E_v ne peut être ni facteur droit ni facteur gauche de E_t puisque $e_1 \neq \varepsilon$ et $e_2 \neq \varepsilon$.

On vient de voir que l'empreinte de t est un mot dont l'empreinte de son facteur v est un facteur propre, c'est−à−dire $E_t = e_1 E_v e_2$. Selon la proposition donnée en VI.8.1, $E_v = E_u$ et $E_t = e_1 E_v e_2 = e_1 E_u e_2$.

Théorème

Lorsque deux types u et t sont tels que l'empreinte générique de u est un facteur propre de l'empreinte générique de t, les instances de u peuvent être converties en instances de t.

Preuve

La transformation s'opère en construisant une instance minimale de type t. Cette instance contient un sous−arbre de type v équivalent à u et selon le théorème de la sous−section VI.8.1, le sous−arbre de type v est isomorphe à l'instance de type u. À la transformation des attributs près, le sous−arbre de type v peut être remplacé par l'instance de type u.

VI.8.3 Reconnaissance des massifs

Proposition

Un type u est un massif d'un type t si l'empreinte générique de u est une sous−chaîne de l'empreinte générique de t.

Preuve

Un massif est construit à partir d'un arbre destination en parcourant ce dernier dans l'ordre préfixe et en retenant seulement certains de ses nœuds. Si un nœud intermédiaire est retenu, son arborescence dans le massif doit contenir au moins une des feuilles de son arborescence dans l'arbre destination. Fondamentalement, un massif est caractérisé par l'absence de nœuds appartenant au type destination. Il est clair qu'il n'y a aucune information relative aux nœuds absents dans l'empreinte du massif. Dans une empreinte, il y a deux sortes de nœuds :

- Les nœuds intermédiaires matérialisés par leurs caractéristiques structurales, par exemple la paire de crochets pour le choix, la paire de parenthèses pour la liste, la paire d'accolades pour l'agrégat.
- Les nœuds terminaux matérialisés par l'identificateur du type de base, du schéma externe ou du type récursif qu'ils représentent.

Soient E_t l'empreinte générique du type destination et E_u l'empreinte générique du massif. Une condition suffisante pour qu'une chaîne $E_u = ze_u$ soit sous−chaîne d'une chaîne $E_t = w_1zw_2e_t$ est :

1. $w_2 \neq \varepsilon$
2. e_u est une sous−chaîne de e_t ou $e_u = e_t$.

Soient i et j les indices de deux caractères contigüs de E_u tels que i = j + 1. La démonstration s'appuie sur la valeur des indices i', j' de ces mêmes caractères dans E_t. En d'autres termes, i étant l'indice du dernier caractère de z dans E_u et j celui du premier caractère de e_u, il faut montrer que j' > i' + 1. w_2 contient au moins un caractère, l'élément de début de structure d'un nœud absent de E_u. Cet élément est à l'indice i' + 1. Qu'importe le nombre de caractères dans w_2, on sait que l'indice j' du premier caractère de e_u dans E_t est au moins j' = i' +2. On conclut que E_u est une sous−chaîne de E_t.

Théorème

Lorsque deux types u et t sont tels que l'empreinte générique de u est une sous−chaîne de l'empreinte générique de t, les instances de type u peuvent être converties en instances de type t.

Preuve

La transformation d'un massif consiste à compléter ce dernier par l'insertion des nœuds du type destination qui n'y ont pas de correspondants. La transformation des informations sémantiques des nœuds u d'un massif en celles des nœuds m(u) de l'arbre destination est requise.

VI.9 Conclusion

L'objectif de ce chapitre est de modéliser les types en représentant leurs caractéristiques principales. On a pu ainsi comprendre les problèmes de transformation de types et exhiber les différences entre les transformations statiques et dynamiques. En associant une procédure à chaque représentation fonctionnelle des caractéristiques, on est assuré d'appréhender l'évolution structurale des types et de garantir la conversion des instances de documents correspondantes.

Le modèle ne représente pas les attributs, car par définition, ces derniers ne sont pas des éléments de structure. Cependant, la transformation de structure doit être comprise comme une restructuration et une récupération d'un maximum d'éléments de structure. La récupération d'un élément de structure suppose alors que les attributs d'un élément de structure sont convenablement convertis.

Le modèle pourrait dans l'absolu être utilisé pour les restructurations dynamiques à condition que l'utilisateur fournisse les couples d'évolution induits par la requête de

transformation. La lourdeur de cette condition nécessaire et la méconnaissance par l'utilisateur de la sémantique des types définis dans les schémas de structure justifient les extensions d'ordre structural et l'association d'un langage algébrique aux types.

Les extensions d'ordre structural ne se contentent pas de constater l'existence de relations d'ordre, d'équivalence, de compatibilité etc. Elles fournissent les éléments nécessaires à leur identification, afin de permettre l'accélération des traitements : par exemple, $u' \in D(t)$ tel que $u \hookleftarrow t$ et $u \equiv u'$.

L'existence d'un langage algébrique permet de résoudre les restructurations dynamiques non couvertes par les extensions d'ordre structural. Le langage algébrique peut être utilisé comme une méthode générale de restructuration puisqu'il permet la reconnaissance de structures algébriques et de relations construites sur un ensemble de types.

Chapitre VII

Conclusion

VII.1 Bilan

L'évolution des types est d'une importance capitale pour les utilisateurs actuels et potentiels des SEDS et les solutions apportées à ce problème aideront au succès tant attendu de l'édition structurée. L'auteur a considéré deux sortes de transformations d'éléments de documents :

- Les transformations statiques des instances de documents induites par l'évolution de la définition des structures logiques génériques.
- Les restructurations dynamiques qui ont lieu uniquement pendant les sessions d'édition.

Cette catégorisation n'exprime pas une différence entre les problèmes mais se justifie par le contexte dans lequel ils surviennent, par la nature des solutions proposées et l'approche qui est faite des deux problèmes. Une typologie complète des évolutions de types est dressée en fonction de tous les éléments définis dans une structure logique générique : les éléments de la structure logique, les éléments associés, les références, les attributs, les schémas externes et les schémas d'extension.

Les solutions proposées aux transformations statiques constituent l'illustration d'un modèle conceptuel articulé autour de trois composants principaux : la correspondance des types, la comparaison des types et la conversion des documents. Le résultat de la comparaison et de la conversion dépendent de la correspondance des types exprimée partiellement par l'utilisateur en fonction des évolutions qu'il souhaite. Cette correspondance est complétée automatiquement par le comparateur sur la base du principe d'homonymie. D'une manière globale, l'évolution des types se traduit par des règles, dites règles de transformation, nécessaires pour réaliser la conversion des documents. La comparaison de types est partiellement fondée sur la représentation fonctionnelle de leurs caractéristiques locales. Les différences constatées entre les caractéristiques de même nature de deux types donnés portent le nom de transformations élémentaires et elles sont facilement mises en œuvre dans la conversion. L'identification des transformations dites complexes nécessite non seulement la connaissance des caractéristiques locales des types mais aussi de celle de leur structure arborescente. La comparaison de deux structures permet d'identifier des relations d'ordre et des relations d'équivalence qui lient deux types. Les solutions qu'apporte cette approche structurale s'appliquent aussi bien aux transformations statiques élémentaires et complexes qu'aux transformations dynamiques.

Cependant pour permettre une récupération plus complète du contenu des éléments lors des restructurations dynamiques, une approche grammaticale du système des types a été développée : Le contenu d'une instance est considéré comme un mot appartenant au langage associé à son type. L'appartenance du mot de cette instance au langage d'un autre type permet de conclure à la transformation de cette instance sans toutefois dire comment la réaliser. Une telle conclusion est rendue possible par la réécriture de la définition d'un type sous forme de grammaire algébrique. L'augmentation de cette grammaire par des symboles exprimant la structure, les constructeurs, permet une représentation de l'arbre des types sous forme de chaîne de caractères : les empreintes générique et effective. La présence des symboles exprimant la structure permet de faire des recherches en fonction des constructeurs, traduisant ainsi un contrôle strict de la structure. Les transformations de la structure logique des documents reviennent alors pour la plupart des évolutions de types, à effectuer la translation des sous−arbres correspondant aux facteurs de l'empreinte effective dans l'empreinte générique.

La transformation des attributs, des références et des liens hypertextes font partie des transformations élémentaires. La conversion des instances de type référence ou hypertexte peut mettre en œuvre les mécanismes exposés plus haut à travers la transformation des éléments référencés.

Certains points méritent pourtant d'être éclaircis.

La forme canonique des schémas de structure joue un rôle capital dans le modèle de type proposé, dans la mesure où elle détermine profondément le modèle lui−même. Le rôle joué par la forme canonique des schémas n'a cessé de grandir, à un tel point que sa définition actuelle ne nous paraît pas aussi "canonique" qu'elle pourrait l'être. À cet effet, davantage d'aspects des langages de définition de types devraient faire l'objet d'une mise sous forme canonique, notamment, les inclusions, les exclusions, les types de constructeur Choix.

Les exceptions

Il est possible de faire disparaître les exceptions de la forme canonique des schémas de structure. La nécessité de la prise en compte des exceptions s'est révélée lors du calcul des empreintes (voir VI.7.1). Cette transformation ne pose aucune difficulté technique.

Les types de constructeur Choix

Le modèle proposé s'est efforcé de faire un rapprochement entre la structure des types et celle des instances, en espérant que ces structures soient isomorphes. Or la structure d'un type de constructeur Choix (toutes les options sont ses fils) diffère de celle de ses instances (un seul fils engendré par une seule option). Pour obtenir un isomorphisme de structures entre un type t' de constructeur Choix et ses instances o_i ($i \in [1 . . \#Fils(t')]$), il suffit de remplacer tout type t, tel que $t' \in Fils(t)$, par autant de types différents t_j ($j \in [1 . . \#Fils(t')]$) , que d'options dans le choix. Dans tout nouveau type t_j, t' est remplacé l'option o_i tel que i = j.

Les types de constructeur Agrégat sans ordre méritent une attention particulière à cause de l'absence d'ordre fixe sur leurs fils directs. Étant donné un type t, de constructeur Agrégat

sans ordre, la sémantique associée à ce constructeur autorise de permuter librement les éléments de Fils(t). Il se pose alors le problème du choix d'un arbre de type dans une requête de transformation. L'arbre de type obtenu à partir de la définition de t n'est qu'un arbre parmi d'autres. Il n'est pas réaliste de produire factoriel(#Fils(t)) types nouveaux dans la forme canonique, par crainte de l'explosion du nombre de types dans le schéma. Il paraît raisonnable de confier à l'outil chargé de réaliser la conversion, l'interprétation de la sémantique du constructeur Agrégat sans ordre et l'adoption d'une stratégie appropriée.

VII.2 Implémentation

Ce travail de thèse a comporté une part importante de développement pour mettre en œuvre le modèle proposé. Cette réalisation a contribué au raffinement du modèle et d'une manière globale à toute la réflexion. L'implémentation des transformations statiques est réalisée entièrement et comporte les éléments suivants :

Le compilateur K — Un compilateur est écrit pour le langage K d'expression des couples d'évolution. L'ensemble des couples d'évolution est une ressource appelée *correspondance de types*.

Le comparateur — Il s'agit d'un programme qui étant donné un couple de schémas de structure principaux, et un ensemble de correspondances de types, compare les types des couples d'évolution pour le couple de schémas principaux et pour les couples de schémas externes. Il produit un ensemble de fichiers de règles dites *règles de transformation*.

Le convertisseur — Le convertisseur est un programme qui prend en entrée un document source et le fichier des règles correspondant au couple d'évolution de schémas dans lequel le schéma de structure du document est l'ancien type à transformer. Le résultat de la conversion, en fonction des règles de transformation, est un ensemble de documents : un document principal et des documents secondaires. Le convertisseur recherche automatiquement le fichier des règles relatives aux éléments engendrés par des schémas externes ou des schémas d'extension. Tous les éléments concernés par ces règles sont transformés.

Le programme est mis en service public à l'unité mixte et a déjà servi à une conversion massive de l'ensemble des documents de l'unité mixte, engendrés par le schéma de structure Article qui a subi une évolution.

Pour ce qui est des restructurations dynamiques, le programme en cours de développement, permet la construction des empreintes générique et effective. Ces éléments serviront à la construction des relations nécessaires à la réalisation des transformation dynamiques.

VII.3 Perspectives

Beaucoup de problèmes importants restent posés, notamment la reconstitution des schémas de structure ainsi que l'édition non structurée avec un éditeur structuré. La reconstitution de schémas de structure semble être un problème vaste. Cependant, si des informations de classe étaient ajoutées aux documents, il serait alors possible de classer les documents et de réduire le problème à la reconstitution de la structure logique générique de chaque classe de documents.

Pour ce qui est de l'évolution des transformations statiques, il serait souhaitable de normaliser un langage qui permette l'expression des transformations sur les types génériques.

Bibliographie

[Akpotsui92] E. Akpotsui et V. Quint, ‘‘Type Transformation in Structured Editing Systems’’, *EP92*, édité par C. Vanoirbeek et G. Coray, pp. 27–41, Cambridge University Press, Cambridge CB2 1RP, avril 1992.

[Akpotsui93] E. Akpotsui, V. Quint et C. Roisin, ‘‘Type Modelling for Document Transformation in Structured Editing Systems’’, *Mathematical and Computer Modelling*, À PARAITRE.

[Abiteboul89] S.Abiteboul et P. C. Kanellakis, ‘‘Object Identity as a Query Language Primitive’’, *Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data*, 18(2), pp. 159–173, juin 1989.

[Adobe91] Adobe Systems Incorporated, *PostScript Language*, Addisson–Wesley, Menlo Park California, novembre 1991.

[GrifMint] J. André, ‘‘GRIF + MINT’’, *Protext III, Proceedings of the Third International Conference on Text Processing Systems*, édité par J. J. H. Miller, Boole Press, Dublin, octobre 1986.

[André89] J. André, R. Furuta et V. Quint, *Structured Documents*, Cambridge University Press, 1989.

[Autebert87] J. M. Autebert, *Langages algébriques*, Masson, 120, Bd Saint–Germain Paris, Avril 1987.

[Williams84] G. Williams, ‘‘The Apple Macintosh computer’’, *Byte*, pp. 30–50, février 1984.

[Ayers84] R. M. Ayers, J. Horring, B. W. Lampson et J. G. Mitchell, *A proposal for a Standard for the Interchange of Editable Documents*, Xerox Corporation, Palo Alto California, janvier 1984.

[Baecker90] R. M. Baecker et A. Marcus, *Human Factors and Typography for More Readable Programs*, Addison–Wesley, Reading Massachusetts, 1990.

[Bancilhon92] F. Bancilhon, C. Delobel et P. Kanellakis, eds, *Building An Object–Oriented Database System – the Story of O_2*, Morgan Kaufman Publishers, San Mateo, 1992.

[Beach83] R. Beach et M.Stone, ‘‘Graphical style: Towards high quality illustrations’’, *Computer Graphics*, 17(3), pp. 127–135, juillet 1983.

[Banerjee86] J. Banerjee, W. Kim, H. J. Kim et H. F. Korth, ‘‘Schema evolution in object–oriented persistent databases’’, *Proceedings of the 6th Advanced Database Symposium*, édité par Information Processing Society of Japan’s Special Interest

Group on Database systems, pp. 23–31, Tokyo, août 1986.

[Banerjee87] J. Banerjee, H. Chou, J. F. Garza, W. Kim, D. Woelk et N. Ballou, ‘‘Data Model Issues for Object–Oriented Applications’’, *ACM Transaction on office Information systems*, 5(1), pp. 3–26, janvier 1987.

[Barron89] D. Barron, ‘‘Why use SGML?’’, *Electronic Publishing––Organisation, Dissemination, and Design*, 2(1), pp. 3–24, avril 1989.

[Bentley86] J. L. Bentley et B. W. Kernighan, ‘‘A language for typesetting graphs’’, *Communications of the ACM*, 29(8), pp. 782–792, août 1986.

[Becker84] J. D. Becker, ‘‘Multilingual word processing’’, *Scientific American*, 251(1), pp. 142–150, juillet 1984.

[Björnerstedt88] A. Björnerstedt et S. Britts, ‘‘AVANCE: An Object Management System’’, *Proceedings of OOPSLA’88*, pp. 206–221, 1988.

[Björnerstedt89] A. Björnerstedt et C. Hulten, ‘‘Version Control in an Object–Oriented Architecture’’, *Object–Oriented Concepts, Applications and Databases*, édité par W. Kim et F. Lochovsky, Addison–Wesley, 1989.

[Borland89] Borland international, *Sprint*, Borland international, California, 1989.

[Borras87] P. Borras, D. Clément, T. Despeyroux, J. Incerpi, B. Lang, G. Kahn et V. Pascual, *Centaur: the system. Rapport de recherche INRIA*, (777), INRIA, décembre 1987.

[Brown90] A. L. Brown et H. A. Blair, ‘‘A Logic Grammar Foundation for Document Representation and Document Layout’’, *EP90*, édité par R. Furuta, pp. 47–64, Cambridge University Press, 1990.

[Brown92] A. L. Brown, T. Wakayama et H. A. Blair, ‘‘A Reconstruction of Context–Document Processing in SGML’’, *EP92*, édité par C. Vanoirbeek et G. Coray, pp. 1–25, Cambridge University Press, Cambridge CB2 1RP, Avril 1992.

[Brown92] H. Brown, F. Cole et E. Oxborrow, ‘‘An Object–Oriented Toolkit for Structured Documents’’, *EP92*, édité par C. Vanoirbeek et G. Coray, pp. 95–111, Cambridge University Press, Cambridge CB2 1RP, Avril 1992.

[Brown88] J. P. Brown, ‘‘Linking and searching within hypertext’’, *Electronic Publishing––Organization, Dissemination and design*, 1(1), pp. 45–53, avril 1988.

[Brown89] J. P. Brown, ‘‘A hypertext system for UNIX’’, *Computing Systems*, 2(1), pp. 37–53, 1989.

[Cardelli85] L. Cardelli et P. Wegner, ‘‘On Understanding Types, Data Abstraction, and Polymorphism’’, *Computing Surveys*, 17(4), pp. 470–522, décembre 1985.

[Chamberlin81] D. C. Chamberlin, J. C. King, D. R. Slutz, S. J. P. Todd et B. W. Wade, ‘‘JANUS: An Interactive System for Document Composition’’, *Proceedings of the*

ACM SIGPLAN SIGOA, 16(6), pp. 82–91, juin 1981.

[Cole90] F. Cole et H. Brown, *Editing a Structured Document with Classes*, (report n. 73), Computing Laboratory, University of Kent, 1990.

[Cole92] F. Cole et H. Brown, ''Editing structured documents––problems and solutions'', *Electronic Publishing––Origination Dissemination and Design*, 5(4), pp. 209–216, décembre 1992.

[Conklin87] J. Conklin, ''Hypertext: An introduction and survey'', *IEEE Computer Society*, 20(9), pp. 17–41, septembre 1987.

[Cowan91] D. D. Cowan, E. W. Mackie, G. de V. Smit et G. M. Pianosi, ''Rita — An Editor and User Interface for Manipulating Structured Documents'', *Electronic Publishing––Origination Dissemination and Design*, 4(3), pp. 125–150, mars 1991.

[Decouchant88] D. Decouchant, A. Duda, A. Freyssinet, M. Riveill, X. Rousset de Pina, R. Scioville et G. Vandôme, ''Guide: An implementation of the Commandos object–oriented architecture on Unix'', *Proceedings of the EUUG Autumn Conference*, pp. 181–193, octobre 1988.

[Delobel82] C. Delobel et M. Adiba, *Bases de données et systèmes relationnels*, Dunod Informatique, Paris, 1982.

[Donzeau84a] V. Donzeau–Gouge, B. Lang et B. Melèse, ''Practical applications of a syntax directed program manipulation environment'', *Proceedings–7th International Conference on Software Engineering*, édité par IEEE Computer Society, pp. 346–354, 1984.

[Donzeau84b] V. Donzeau–Gouge, G. Kahn, B. Lang et B. Melèse, ''Document structure and modularity in Mentor'', *Software engineering notes*, 9(3), pp. 141–148, mai 1984.

[Donzeau83] V. Donzeau–Gouge, G. Kahn, B. Lang et B. Melèse, ''Outline of a tool for document manipulation'', *Proceedings of IFIP congress*, pp. 615–620, décembre 1983.

[Ellis91] C. A. Ellis, J. Gibbs, G. L. Rein et M. Cohen, ''Quilt: A Collaborative Tool for Cooperative Writing'', *Communications of the ACM*, 34(1), pp. 38–58, janvier 1991.

[Estublier83] J. Estublier, S. Krakowiak, J. Mossière et Y. Rouzeau, ''Design principles of the Adèle programming environment'', *International Computing Symposium on Application systems Development*, édité par ACM, mars 1983.

[Frilley89] F. Frilley, *Différenciation d'ensembles structurés*, Informatique, Université de Paris VII, Mars 1989.

[Furuta86] R. Furuta, *An integrated but not exact representation, editor/formatter*, (86–09–08), Departement of computer science Fr–35, University of Washingtown Seattle WA 98195, septembre 1986.

[Furuta87] R. Furuta et P.D. Scotts, *Specifying structured document transformations*, (CS–TR–1913), University of Maryland, College park MD 20742, septembre 1987.

[Furuta92] R. Furuta, ''Important papers in the history of document preparation systems: basic sources'', *Electronic Publishing*, 5(1), pp. 19–44, mars 1992.

[Garlan86] D. Garlan, C. W. Krueger et B. J. Staudt, ''A Structural Approach to the Maintenance of Structure–Oriented Environments'', *ACM SIGSOFT Symposium on Practical Software Development Environments*, pp. 160–170, 1986.

[Goldfarb81] C. F. Goldfarb, ''A generalized approach to document markup'', *Proceedings of the ACM SIGPLAN SIGOA symposium on text manipulation, SIGPLAN Notices*, 16(6), pp. 68–73, juin 1981.

[Goldfarb90] C. F. Goldfarb, *The SGML Handbook*, Oxford University Press, Oxford, 1990.

[Gonzalez78] R. C. Gonzalez et M. G. Thomason, ''Syntax–Directed Transduction'', *Syntactic Pattern Recognition*, pp. 57–60, 1978.

[Graham92] S. L. Graham, M. A. Harrison et E. V. Munson, ''The Proteus Presentation System'', *Proceedings of the ACM SIGSOFT Fifth Symposiumon Software Development Environments*, édité par ACM Press, pp. 130–138, ACM Press, Tyson's Corner VA, décembre 1992.

[Graham93] S. L. Graham, M. A. Harrison et E. V. Munson, ''Separating Presentation and Control in the User Interface of Ensemble'', *UIST*, avril 1993.

[Greif92] I. Greif, R. Seliger et W. Weihl, ''A Case Study of CES: A Distributed Collaborative Editing System Implemented in Argus'', *IEEE Transactions on Software Engineering*, 18(9), pp. 827–839, septembre 1992.

[Güting89] R. H. Güting, R. Zicari et D. M. Choy, ''An Algebra for Structured Office Documents'', *ACM Transactions on Office Information Systems*, 7(4), pp. 123–157, 1989.

[Han92] B. J. Han, P. Kahn, V. A. Riley, J. H. Coombs et N. K. Meyrowitz, ''IRIS Hypermedia'', *Communications of the ACM*, 35(1), pp. 36–51, janvier 1992.

[Hansen90] B. S. Hansen, ''A function–based formatting model'', *Electronic Publishing––Origination, Dissemination and Design*, 3(1), pp. 3–28, février 1990.

[I.S.O86] I.S.O., *Information processing – Text and office systems – Standard Generalized Markup Language (SGML), ISO 8879*, octobre 1986.

[I.S.O89] ISO, *Information Processing – Text and Office System – Office Document Architecture (ODA) and interchange format*, (ISO/DIS 8613/1), ISO, septembre 1989.

[I.S.O91a] I.S.O., *Information technology – Text and office systems – Document Style Semantics and Specification Language (DSSSL), ISO/IEC DIS 10179*, 1991.

[I.S.O91b] I.S.O., *Information technology −− Hypermedia/Time−based Structuring Language (HyTime), ISO/IEC DIS 10744*, octobre 1991.

[Joloboff86] V. Joloboff, ‘‘Text Processing and Document Manipulation’’, *Trends and standards in document representation*, édité par J.C. van Vliet, pp. 107−124, Cambridge University Press, avril 1986.

[Joloboff89] V. Joloboff, ‘‘Document representation: Concept and Standards’’, *Structured documents*, édité par J. André, R. Furuta & V. Quint, pp. 75−105, Cambridge university press, Cambridge G. B., 1989.

[Johnson88] J. Johnson et R. J. Beach, ‘‘Styles in document editing systems’’, *Computer*, 21(1), pp. 32−43, janvier 1988.

[Kahn91] P. Kahn et B. J. Haan, ‘‘Video in hypermedia: The design of InterVideo’’, *Visual Ressource*, VII pp. 353−360, 1991.

[Kernighan82] B. W. Kernighan, ‘‘A language for typesetting graphics’’, *Software Practice and experience*, 12(1), pp. 1−21, janvier 1982.

[Kernighan78] B. W. Kernighan, M. E. Lesk et J. F. Ossanna, ‘‘UNIX time−sharing system: Document Preparation’’, *The Bell System Technical Journal*, 57(6), pp. 2115−2135, juillet−août 1978.

[Knuth79] D. E. Knuth, *Mathematical typography*, (1), Digital Press and the American Mathematical Society, décembre 1979.

[Knuth85a] D. E. Knuth, *The T_EX book*, Addison Wesley, Reading, 1985.

[Knuth85b] D. E. Knuth, ‘‘Lessons learned from METAFONT’’, *Visible Language*, 19(1), pp. 35−53, 1985.

[Lamport85] L. Lamport, *LAT_EX: A document preparation system*, Addison Wesley, Reading Massachusetts, avril 1985.

[Leland88] M. D. P. Leland, R. S. Fish et R. E. Kraut, ‘‘Collaborative Document Production Using Quilt’’, *Proceedings of the Conference on Computer Supported Cooperative Work*, pp. 206−215, ACM Press, Portland (OR), novembre 1988.

[Louarn90] P. Louarn, ‘‘Traitement d’index avec L_AT_EX’’, *Cahiers GUTenberg*, (7), pp. 23−281, Novembre 1990.

[MACE91] MACE, ‘‘A fine Grained Concurrent Editor’’, *Proceedings of the Conference on Organizational Computing Systems*, pp. 240−254, ACM Press, novembre 1991.

[Maloney88] M. Maloney, Rubinski, P. Sharpe et R.Spencer, *Author/Editor Advance version User’s Guide*, SoftQuad Inc., Toronto, Canada, 1988.

[Meyrowitz82] N. Meyrowitz et A. Van Dam, *Interactive editing systems in document preparation system*, North Holland publishing company, Amsterdam, 1982.

[Microsoft92] Microsoft Incorporation, *Microsoft Word : Guide de l'utilisateur*, Microsoft Incorporation, 1992.

[Monk93] S. Monk et I. Sommerville, ''Schema Evolution in OODBs Using Class Versioning'', *SIGMOD RECORD*, 22(3), pp. 16–22, septembre 1993.

[Morris85] R. A. Morris, ''Is what you see enough to get? A description of the Interleaf publishing system'', *PROTEXT II: Proceedings of the second international conference on text processing systems*, édité par J. J. Miller, pp. 56–81, Bool Press, Octobre 1985.

[Neuwirth90] C. M. Neuwirth, D. S. Kaufer, R. Chandhok et J. H. Morris, ''Issues in the Design of Computer Support for Co–authoring and Commenting'', *Proceedings of the Conference on Computer Supported Cooperative Work*, pp. 183–195, ACM Press, octobre 1990.

[Nielsen90] J. Nielsen, *Hypertext and Hypermedia*, Academic Press, San Diego, California, 1990.

[Notkin86] D. S. Notkin et A. N. Habermann, ''Gandalf Software Development'', *IEEE Transactions on Software Engineering*, 1986.

[Oman90] P. Oman et C. R. Cook, ''Typographic style is more than cosmetic'', *Communication of the ACM*, 33(5), pp. 506–520, mai 1990.

[Palaniappan90] M. Palaniappan, N. Yankelovich et M. Sawtelle, ''Linking active anchors: A stage in the evolution of Hypermedia'', *Hypermedia*, 2(1), pp. 47–66, 1990.

[Peels85] A. J. H. M. Peels, N. J. M. Janssen et Wop Nawijn, ''Document architecture and text formatting'', *ACM Transsactions on Office Information Systems*, 3(4), pp. 347–369, octobre 1985.

[Penney87] D. J. Penney et J. Stein, ''Class Modification in the GemStone Object–Oriented DBMS'', *ACM OOPSLA*, octobre 1987.

[Quint87] V. Quint, *Une approche de l'édition structurée des documents*, Thèse d'état, Université scientifique technologique et médicale de Grenoble, Mai 1987.

[Quint89] V. Quint, *Structured Documents*, Cambridge University Press, Cambridge CB2 1RP, 1989.

[Quint93] V. Quint, *Les langages de Grif*, Projet OPERA, Bull–Imag/systèmes 2, avenue Vignate 38610 Gières, Mai 1993.

[Quint86a] V. Quint et I. Vatton, ''Grif: an Interactive System for Structured Document Manipulation'', *Text Processing and Document Manipulation, Proceedings of the International Conference*, édité par J. C. van Vliet, pp. 200–213, Cambridge University Press, 1986.

[Quint86b] V. Quint, I. Vatton et H. Bedor, ''Grif, an Interactive Environment for TEX'',

T_EX for Scientific Documentation, édité par J. Désarménien, pp. 145–158, Springer–Verlag, 1986.

[Woodman89] V. Quint et I. Vatton, ‘‘Modularity in structured documents’’, *Woodman’89*, édité par J. André & J. Bézivin, pp. 170–177, Bigre num. 63–64, IRISA, Rennes, mai 1989.

[Quint92] V. Quint et I. Vatton, *Hypertext aspects of the Grif structured editor: Design and application*, (1734), INRIA, Domaine de Voluceau B.P. 105 78153 Le chesnay Cédex, juillet 1992.

[Quint90] V. Quint, M. Nanard et J. André, ‘‘Towards Document Engineering’’, *EP90*, édité par R. Furuta, pp. 17–29, The press Syndicate of the University of Cambridge, Cambridge CB2 1RP, 1990.

[Reid80] B. K. Reid, *Scribe: A Document Specification Language and its Compiler*, PhD thesis, Carnegie–Mellon University Computer Science Department, Pittsburg, PA, octobre 1980.

[Rein91] G. L. Rein et C. A. Ellis, ‘‘sIBIS: a real–time group hypertext system’’, *International Journal of Man–Machine Studies*, (34), pp. 349–367, 1991.

[Reiss84] S. P. Reiss, ‘‘Graphical program development with PECAN’’, *Software Engineering Notes*, 9(3), pp. 30–41, mai 1984.

[Reps84] T. Reps et T. Teitelbaum, ‘‘The Synthesizer Generator’’, *Software Engineering Notes*, 9(3), mai 1984.

[Richy92] H. Richy, P. Frisson et E. Picheral, ‘‘Multilingual String–to–String Correction in Grif, a Structured Editor’’, *EP92*, édité par C. Vanoirbeek G. Coray, pp. 183–198, The press Syndicate of the University of Cambridge, Cambridge CB2 1RP, 1992.

[Robinson87] P. J. Robinson, ‘‘The Esprit PODA demonstration of ODA at cebit 87’’, *Achievements and impact, Proceedings of the 4th annual Esprit Conference*, édité par Esprit’87, pp. 1378–1388, North–Holland, 1987.

[Roddick91] J. F. Roddick, ‘‘Dynamically changing schemas within database models’’, *Australian Computer Journal*, 23(3), pp. 105–109, 1991.

[Roddick92] J. F. Roddick, ‘‘SQL/SE – A Query Language Extension for Databases Supporting Schema Evolution’’, *SQL/SE*, 21(3), pp. 10–16, 1992.

[Rose91] E. Rose et A. Segev, ‘‘TOODM – A temporal Object–Oriented Data Model With Temporal Constraints’’, *Proceedings of Entity Relational Approach – 10th International Conference*, 1991.

[Salton89] G. Salton, *Text processing*, Addison–Wesley, New York, 1989.

[Saltzer65] J. Saltzer, *TYPESET, RUNOFF in The Compatible Time–Sharing System: A programmer’s guide*, P. A. Crisman, The MIT Press, 1965.

[Scotts91] P. D. Scotts et R. Furuta, ‘‘Hypertext 2000: Databases or Documents?’’, *Electrocin Publishing*, 4(2), pp. 119−121, juin 1991.

[Skara86] A. H. Skara et S. B. Zdonik, ‘‘The Management of Changing Types in an Object−Oriented Database’’, *ACM OOPSLA*, septembre 1986.

[Skara87] A. H. Skara et S. B. Zdonik, ‘‘Type Evolution in an Object−Oriented Database’’, *Research and Directions in Object Oriented Systems*, édité par B. Shriver and P. Wagner, MIT Press, septembre 1987.

[Stallman81] R. M. Stallman, *EMACS Manuel for TWENEX Users*, MIT Artificial Intelligence Laboratory, 545 Technology Square Cambridge, MA 02139 USA, octobre 1981.

[Staudt86] B. J. Staudt et V. Ambriola, *The ALOE Action Routine Language*, technical manual, Computer Science Department, Carnegie−Mellon University, Pittsburgh, PA., 1986.

[Teitelman85] W. Teitelman, ‘‘A tour through Cedar’’, *IEEE Transactions on Software Engineering*, 11(3), pp. 285−302, mars 1985.

[Teitelbaum81] T. Teitelbaum et T. Reps, ‘‘The Cornell Program Synthesizer’’, *Communication of the ACM*, 24(9), pp. 563−573, septembre 1981.

[Herwijnen91] E. van Herwijnen, *Practical SGML*, Kluwer Academic Publishers, P.O. Box 17 3300 AA Dordrecht The Netherlands, 1991.

[Zicari89] R. Zicari, *A Framework for O_2 Schema Updates*, (38), Altaïr, octobre 1989.

Chapitre I

Systèmes d'édition de documents structurés : état de l'art

Chapitre II

Grif : un système d'édition de documents structurés

Chapitre III

Les problèmes induits par la structure

Chapitre IV

Évolution des types et restructurations dynamiques

Chapitre V

Modèle conceptuel de transformation statique

Chapitre VI

Modèle de type

Chapitre VII

Conclusion

Titre :

Transformation de types dans les systèmes d'édition de documents structurés

Résumé :

Les systèmes d'édition de documents fondés sur la description logique des composants des documents s'appuient sur les grammaires hors contexte. Ces grammaires assez riches permettent la description des classes de documents (schémas de structure), de leurs composants, des relations hiérarchiques et de voisinage que ces derniers entretiennent les uns avec les autres, et d'informations d'ordre sémantique associées aux composants sous forme d'attributs. La vérification rigoureuse de la compatibilité de types, bénéfique par ailleurs, induit aussi des inconvénients dont les principaux sont le rejet des couper−coller, l'impossibilité d'éditer les documents dont les schémas de structure ont évolué, l'impossibilité de réaliser des restructurations en cours d'édition.

Le but de cette thèse est d'étudier l'évolution des types, de proposer des solutions à ces problèmes et de les mettre en œuvre dans le système Grif.

La thèse présente, dans une première partie, un état de l'art et les problèmes de restructuration dans les systèmes d'édition de documents structurés (SEDS) en général, notamment l'éditeur Grif qui sert de cadre à cette étude.

La deuxième partie présente une typologie de l'évolution des structures et des attributs et un modèle conceptuel de conversion automatique des instances de documents concernées par l'évolution statique de structures.

La troisième partie de cette thèse présente en trois points un formalisme des types pour les SEDS :

1. Un modèle mathématique de types dans les SEDS, fondé sur la représentation fonctionnelle des caractéristiques structurales des types, qui permet d'exprimer avec rigueur les évolutions possibles de structure.
2. Un ensemble de définitions des relations structurales entre types (facteur, sous−typage, massif, compatibilité, équivalence), utiles dans les transformations statiques et dynamiques.
3. Une approche grammaticale pour les transformations dynamiques : un schéma de structure peut être transformé en une grammaire algébrique, un document pouvant être interprété comme un mot du langage issu de cette grammaire. Le langage retenu par la présente thèse est construit sur un alphabet terminal composé de l'ensemble des types de base du système, de l'ensemble des identificateurs des schémas de structure du système et de l'ensemble des symboles exprimant la structure des types.

Mots−clés

Document structuré, modèle de documents, SGML, DSSSL, transformation de types, arbre, type, langage algébrique, empreinte générique, empreinte effective.