MASTER EN SCIENCES INFORMATIQUES

Méthodes et outils pour la conception de bases de données XML natives

Estievenart, Fabrice

Award date:
2002

Awarding institution:
Universite de Namur

FUNDP
Institut d'Informatique

Rue Grandgagnage, 21
B - 5000 NAMUR (Belgique)

Méthodes et Outils pour la Conception de Bases de Données XML Natives

Fabrice Estiévenart

Mémoire supervisé par le Professeur Jean-Luc Hainaut

Institut d'Informatique
Facultés Universitaires Notre-Dame de la Paix
Namur

Septembre 2002

«Le savoir n'est pas une vulgaire matière première. Il ne vient jamais à épuisement. Au contraire, il s'accroît toujours grâce à la diffusion des connaissances.»
(Daniel Boorstin)

Résumé

En peu de temps, le langage XML a réussi à s'imposer comme format standard d'échange d'informations sur internet. Son succès est probablement dû à sa simplicité, sa portabilité et à son modèle de données semi-structuré, bien approprié pour représenter les informations circulant sur la toile. Sans être révolutionnaire, ce nouveau modèle vient perturber le monde des bases de données. Le stockage de données à structure hiérarchique dans les tables en première forme normale d'un SGBD relationnel classique requiert une transformation de modèle, pas toujours triviale et peu performante. Pour pallier ces désagréments, des bases de données dites «natives» proposent un moyen de stocker des documents XML sous leur forme brute, c'est à dire en conservant leur structure originale.

Dans le cadre de ce mémoire, nous proposons une méthode et des outils permettant la conception de telles bases de données. La méthode de conception est constituée de quatre processus : l'analyse conceptuelle modélise les besoins de l'utilisateur, la conception logique transforme le schéma conceptuel en un schéma logique conforme à un modèle XML, la conception physique paramètre le schéma de données afin d'améliorer les performances du futur système. Finalement, le codage génère le code d'une DTD (Document Type Definition) exploitable par un serveur de données XML natif.

Leader actuel sur le marché des systèmes XML natifs, Tamino (Transaction Architecture for Management of INternet Objects) de Software AG est le SGBD (Système de Gestion de Base de Données) que nous avons choisi pour exploiter les structures de données précédemment définies. Nous en présentons les fonctionnalités principales comme la création d'un base de données et de collections, l'importation et l'interrogation de données XML.

Le dernier chapitre de ce travail illustre, sur base d'une étude de cas concrète, les différentes étapes de la méthode de conception et la manipulation des outils sous-jacents.

Abstract

In a short period of time, the XML language has become the standard format for the exchange of information on the internet. The success of XML is probably due to its simplicity, portability and its semi-structured data model which is fit to represent the data on the web. Far from being revolutionary, this recent model disrupts the database community. Storing hierarchical data in flat relational databases requires a complex and expensive mapping. Specially designed to store XML documents without prior transformation, native XML databases currently offer an alternative.

In this report, we suggest a method and some tools for building such databases. The proposed method is made up of four processes : the conceptual analysis where the user's needs are expressed in a conceptual scheme, the logical design transforms this conceptual scheme in a logical one which is in accordance to a logical XML model. Following this step, the physical design sets up properties such as indexes in order to improve performance. Finally, coding generates a source text compliant with the W3C DTD specifications and which can be processed by a native XML data management system.

Leader on this market, Tamino (Transaction Architecture for Management of INternet Objects), developped by Software AG, was choosen to exploit the data structures previously defined. We introduce the main features and services offered by this XML tool : we explain how to create a database, to define collections, to import and query XML documents.

In the last chapter, we illustrate the method and tools by presenting a concrete example covering the entire process proposed for building native XML databases.

Remerciements

Je tiens à remercier toutes les personnes qui ont contribué, de près ou de loin, à la réalisation de ce mémoire :

Jean-Luc Hainaut, le promoteur de ce mémoire, pour la supervision de mon travail et ses nombreuses remarques pertinentes.
Jean Henrard et toute l'équipe du LIBD, Aurore François, Christine Delcroix, Anne-France Brogneaux pour leur disponibilité et leurs conseils avertis.
Yves Pigneur, mon maître de stage à l'Université de Lausanne, pour m'avoir intégré dans un projet exclusif et enrichissant.
Alexander Osterwalder, Sarra Ben Lagha, Olivier Shaw pour le partage de leurs connaissances et leur étroite collaboration au sein du projet «eBusiness Model Handbook»
Ethel Bonvin, Yelena Jussupova, Mathias Rossi, Antoine Christen pour le chaleureux accueil lors de mon arrivée à l'Inforge.
La communauté des utilisateurs Tamino pour le support rapide et efficace sur le forum du site.
Frédéric Burlet, Vincent Englebert, Quentin Pouplard, Frédéric Wautelet, François Vermaut pour leurs conseils de mise en page avec LaTeX.
Stéphane Cornet, Chrystelle Estiévenart pour la relecture attentive de l'introduction et de la conclusion.
Mathieu Formule, Jean-Yves Sohet pour leur complicité et la bonne ambiance à Lausanne.

Table des matières

Chapitre 1

Introduction

1.1 Genèse d'XML

Depuis plusieurs années, le langage HTML (HyperText Markup Language) est devenu le format de documents le plus courant sur internet. On a longtemps considéré que ce langage de présentation pouvait répondre de manière très satisfaisante aux besoins de la plupart des applications informatiques.

Mais au fil du temps, l'HTML a commencé à montrer ses limites et son incapacité à supporter des applications de plus en plus sophistiquées. On déplore d'ailleurs souvent deux inconvénients majeurs. D'une part, le mélange de contenu et de mise en page pose problème lorsqu'il s'agit de présenter la même information sur différents types de terminaux. D'autre part, l'absence de toute indication sur la sémantique des informations présentées entraîne certaines difficultés d'interprétation.

Certes, le SGML (Standard Generalized Markup Language) s'affranchissait déjà de ces limites, grâce notamment à l'introduction de balises personnalisées. Sa complexité le rendait cependant trop difficile d'utilisation.

Ce constat amer a conduit le W3C (World Wide Web Consortium) à mettre au point une alternative ; XML (eXtensible Markup Language) devait ainsi voir le jour pour pallier les faiblesses de l'HTML tout en offrant une mise en oeuvre plus aisée que le SGML. Cette nouvelle norme connut une ascension fulgurante, s'imposant comme format standard d'échange d'informations. Ses applications sont par ailleurs multiples, de l'échange de données d'un site web vers un navigateur à l'échange de données automatisé entre applications informatiques.

1.2 XML et les bases de données

Investies des rôles de gestionnaire et fournisseur de contenu, les bases de données se trouvent évidemment au coeur de la problématique XML. Le succès fracassant d'XML bouleverse la communauté des bases de données. De nombreux chercheurs proposent en effet un nouveau modèle de données mieux adapté à XML : le modèle semi-structuré.

1.2.1 XML est-il une base de données ?

Dans un document récent intitulé *XML and Databases* [Bou02], Ronald Bourret pose sans détour cette question fondamentale : un document XML peut-il être considéré comme une base de données ? Un tel document n'est autre qu'une collection de données structurées, autodescriptives, encodées dans un format standard assurant une bonne portabilité. XML (agrémenté des divers standards définis dans la norme) possède dès lors certaines caractéristiques propres aux SGBD :

1. un mode de stockage persistant via l'enregistrement dans un fichier XML.
2. un mécanisme de description de données grâce à la norme du World Wide Web Consortium (W3C) sur les Document Type Definitions (DTD) ou les XML Schemas.
3. un langage d'interrogation tel XQuery ou XQL (XML Query Language).
4. une interface avec des langages de programmation. Les deux plus connues sont SAX (Simple API for XML) et DOM (Document Object Model).

Toutefois, des concepts propres aux SGBD tels les index, la sécurité, la concurrence ou les transactions sont absents des mécanismes définis par la norme XML.

En pratique, on peut imaginer qu'un document XML soit utilisé comme une base de données dans des cas d'utilisation très limités, c'est-à-dire lorsque la quantité de données à stocker, les exigences en matière de performance ou le nombre d'utilisateurs concernés est assez faible.

1.2.2 Les modes de stockage d'un document XML

Actuellement, quatre stratégies plus avancées sont donc proposées pour assurer la persistance et l'exploitation de données XML. Ces quatre modes de stockage sont le stockage sous forme de BLOB (Binary Large OBject), le stockage relationnel avec «mapping» XML, le stockage orienté-objet ou le stockage natif.

Le stockage sous forme de BLOB

L'idée de cette stratégie est d'aplatir la structure hiérarchique d'un document XML afin de pouvoir stocker celui-ci sous forme d'objet binaire dans une colonne d'une base de données relationnelle. Grâce à l'ajout d'index sur les lignes des tables, il est possible d'implémenter son propre système d'indexation de documents. Ce mode de stockage est principalement utilisé dans les systèmes documentaires pour lesquels il est nécessaire de disposer d'un moteur de recherche et d'un mécanisme performant d'accès aux documents. En confiant ses documents à un SGBD compétent, on bénéficie des mécanismes complexes de transaction et de sécurité offerts d'origine par le système. La recherche au sein du document XML est rendue possible par l'utilisation des primitives de recherche de texte disponibles dans la plupart des SGBD.
Exemple de produit : *BladeRunner (http ://www.broadvision.com)*

Le stockage relationnel avec «mapping» XML

Dans cette approche, les données contenues dans le document XML sont stockées de manière traditionnelle dans les diverses tables d'une base de données relationnelle. Fondées sur un modèle théorique solide et utilisant des technologies bien éprouvées, les bases de données relationnelles sont actuellement les plus répandues. Le principal avantage de cette solution réside dans sa robustesse et sa facilité de réutilisation. En effet, il est intéressant pour une entreprise de pouvoir exploiter son système de gestion de données existant pour la publication de son catalogue sur son site internet. Mais à l'heure du web, quelques voix s'élèvent pour dénoncer les limites des bases de données relationnelles. Lors d'un récent séminaire donné à Lausanne [Gar01], Georges Gardarin osera parler d'un «rendez-vous manqué» entre le web et les bases de données. Les bases de données relationnelles, constituées de tables «plates» (en première forme normale) ne seraient plus appropriées pour stocker les documents typiquement arborescents qui circulent sur internet. Bref, entre le monde hiérarchisé des documents et le monde plat des tables relationnelles, il existe une incompatibilité de modèle qui semble bien lourde à gérer. Selon Georges Gardarin, les deux principaux arguments en défaveur du relationnel sont la complexité du «mapping» entre les deux modèles ainsi que la lourdeur du processus de

reconstitution d'un document éclaté dans diverses tables. Dans un article sur les Data Web-Houses [Gar], Georges Gardarin évoque également un autre type de problème lié à l'utilisation de bases de données relationnelles pour la publication, sur le web, de données dynamiques : les bases de données relationnelles nécessitent la définition de schémas rigides dont la difficulté d'évolution est bien connue.
Exemple de produit : *Oracle SDK (http ://www.oracle.com)*

Le stockage orienté-objet

Il s'agit, dans ce cas, de stocker les éléments XML sous la forme d'objets dans une base de données orientée-objet ; la hiérarchie entre les classes de la base de données reflétant l'arborescence des types d'éléments contenu dans le document XML. Les objets persistants ainsi créés peuvent être réutilisés et exploités par un serveur d'applications ou un simple programme Java.
Exemple de produit : *eXcelon Object Store (http ://www.exceloncorp.com/objectstore)*

Le stockage natif

Principal atout commercial du très populaire SGBD Tamino [1] de Software AG [AGb], le stockage natif consiste à stocker et exploiter des documents XML sous leur forme brute i.e. sans transformation préalable. Outre le gain de performance lié à l'absence de transformation de modèle («mapping»), cette solution, principalement conçue pour la publication sur internet, offre les fonctionnalités traditionnelles d'un système de gestion de bases de données : langage d'interrogation, sécurité, gestion des transactions, backup... Dans ce genre de base de données, la description des informations XML stockées se fait grâce à une DTD ou un XML Schema. Ce mode de stockage assez récent sera décrit plus amplement dans la suite de ce mémoire.
Exemple de produit : *Tamino (http ://www.xmlstarterkit.com)*

1.2.3 Le choix d'un mode de stockage

Le choix d'un mode de stockage dépend principalement de la nature du document XML à stocker. On distingue les documents XML «orientés-données» et les documents XML «orientés-documents» [Bou02].

Les documents XML «orientés-données»

Ces documents utilisent XML pour transporter des données brutes, destinées à être exploitées par des applications et pas par un utilisateur. Un document XML contenant les différentes commandes des clients de l'entreprise est un exemple type de document «orienté-données». Il se caractérise par des structures répétitives, une granularité des données assez fine et un ordre insignifiant entre les éléments (excepté pour des raisons de validation).

Les documents XML «orientés-documents»

Ces documents XML décrivent un livre, un article ou un email destiné à être directement exploité par un utilisateur final. Ils se caractérisent par une structure irrégulière, des unités de données sous la forme de blocs plus épais et un ordre souvent important entre les éléments.

[1] Transaction Architecture for Management of INternet Objects

Comparaisons

En pratique, la distinction entre les deux types de documents n'est pas toujours évidente : un document «orienté-données», tel une commande, peut contenir de larges blocs de données non-structurées.

En règle générale, un document «orienté-données» sera stocké dans les structures répétitives (tables, colonnes) d'un SGBD relationnel alors que pour les documents «orientés-documents» on préférera la souplesse du stockage natif. Mais cette règle n'est pas absolue et nous verrons comment un SGBD natif, Tamino en l'occurence, peut traiter des documents «orientés-données».

1.3 L'ingénierie des bases de données

1.3.1 Méthode de conception d'une base de données

Dans son cours intitulé *Ingénierie des base de données* [Hai00], Jean-Luc Hainaut définit la conception d'une base de données comme « un processus qui transforme l'expression des besoins des utilisateurs en une collection de produits (schémas et programmes) qui respectent des critères de qualité ». La base de données créée devrait être :

1. **correcte**, c'est-à-dire une représentation fidèle, complète et lisible du domaine d'application.
2. **opérationnelle**, dont la structure est conforme à l'environnement d'exploitation.
3. **efficace**, dont le temps de réponse et l'espace physique occupé sont réduits.
4. **conforme à certains critères techniques**, répondant à certaines exigences du point de vue de la fiabilité, de l'ergonomie ou de l'extensibilité.

Le schéma présenté Fig.1.1 illustre les processus (rectangles) constituant une méthode idéale de conception de base de données. Chaque processus élabore un produit (ovale) intermédiaire à partir d'autres produits fournis en entrée.

A partir de l'expression précise des besoins des utilisateurs, un **schéma conceptuel** correct et indépendant de toute technologie peut être esquissé. Dans ce schéma, les principaux concepts de la situation à modéliser sont mis en évidence et forment des types d'entités. Un type d'entités possède des propriétés qui lui sont propres (attributs) et peut être mis en relation avec d'autres types d'entités du schéma via des types d'associations. Ce processus de conceptualisation se nomme l'**analyse conceptuelle**.

Lors de l'étape de **conception logique**, le schéma de données doit être rendu opérationnel, c'est à dire conforme au modèle du SGBD ciblé. Un modèle se définit par une série de contraintes que doivent respecter les différents éléments du schéma. Un schéma respectant ces contraintes est dit **logique**. On l'obtient par des transformations successives appliquées sur le schéma conceptuel. Idéalement, ces transformations sont symétriquement réversibles, c'est-à-dire qu'elles ne dégradent pas la sémantique originelle du schéma conceptuel.

Lorsque le schéma est conforme à un modèle particulier, le processus de **conception physique** se charge de l'optimiser en y ajoutant des informations – purement physiques – sur les espaces de stockage ou sur les index à mettre en place afin d'améliorer les performances de la base de données.

A partir du schéma physique ainsi construit, le processus de codage génère le code de «description de données» (en anglais, Data Description Language) qui sera directement exploitable par le SGBD ciblé.

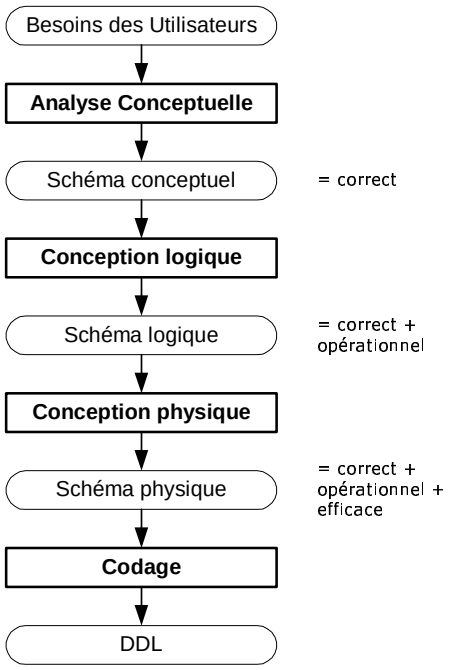

FIG. 1.1 – Les processus et produits d'une méthode idéale de conception de base de données

1.3.2 L'ingénierie des bases de données relationnelles

Sujet de nombreux travaux de recherche, l'**ingénierie des bases de données relationnelles** est une discipline assez bien maîtrisée actuellement. Le modèle relationnel possède des bases théoriques solides et des recherches rigoureuses ont mis au point des techniques et méthodes de conception proposant des jeux de transformations réversibles. Ces méthodes sont actuellement supportées par des outils CASE à la fois conviviaux et puissants. L'outil DB-Main fait partie de cette classe de logiciel d'aide à la conception de bases de données.

1.3.3 L'ingénierie des bases de données XML

Pour diverses raisons dont sa nature plus récente, l'**ingénierie des bases de données XML** ne procure pas encore les mêmes satisfactions.

L'émergence d'un nouveau modèle de données s'accompagne de nouvelles méthodes et d'outils de conception. Les méthodes servent à piloter le processus de création de structures de données conformes au modèle ; alors que les outils sont sensés supporter la méthode définie.

Quel que soit le mode de stockage des documents XML, les méthodes et outils pour la conception de bases de données XML sont actuellement encore trop peu nombreux. Néanmoins, certaines avancées ont déjà été réalisées en la matière.

Etat de l'art

Dans un document rédigé en avril 2001 ([Del01b]), Christine Delcroix propose un modèle logique XML auquel doit se conformer un schéma pour qu'il soit la représentation d'une DTD. Il s'agit d'un ensemble de contraintes définies pour chaque type d'éléments et pouvant être validées par l'outil d'analyse de schéma inclus dans DB-Main. Dans le cadre de son travail de recherche sur l'ingénierie XML, Christine Delcroix décrit également un plan de transformation d'un schéma conceptuel en un autre conforme au modèle XML ([Del01c]) et présente quelques primitives de transformation qui facilitent la réalisation de l'objectif.

La transformation d'un schéma conceptuel en une DTD est également le sujet d'un article de C. Kleiner et U.W. Lipeck[KW01]. Sur base d'un exemple, ces auteurs y abordent les questions relatives au choix de la racine, à la transformation des attributs...

Enfin, le livre *Designing XML Databases* [Gol01] est une référence pour tous ceux désirant utiliser un SGBD relationnel pour stocker des documents XML. Ce livre évoque notamment les difficultés d'un «mapping» entre les modèles XML et relationnel. On y présente également quelques solutions concrètes à mettre en oeuvre.

C'est à partir de ces différents matériaux que mes recherches ont commencé...

Ma contribution dans le cadre du présent mémoire consiste à réviser les traitements et les enchaînements des étapes proposées par Christine Delcroix dans son plan de transformation, et à apporter quelques améliorations fonctionnelles et ergonomiques à l'outil afin de le rendre utilisable par un public motivé certes mais pas forcément spécialiste de la question.

La démarche proposée dans le cadre de ce travail ne se limite pas à l'étape de conception logique. Nous montrons en quoi consiste la conception physique XML et comment implanter, exploiter une base de données Tamino sur base du schéma de données créé lors des étapes précédentes.

1.4 Contenu de ce mémoire

Le **chapitre 2**, intitulé *XML : Etat de l'art* dresse un bref état de l'art d'XML et des standards qui y sont associés (DTD, XML Schema, XPath, XSL...). Cette synthèse des recommandations du W3C se contente de fournir au lecteur les bases nécessaires à la compréhension du processus de conception de bases de données XML.

Dans le **chapitre 3** intitulé *Conception logique* nous proposons un modèle logique de données XML. Sur base des recherches menées par Christine Delcroix, nous présentons, dans ce même chapitre, une méthode théorique de transformation d'un schéma conceptuel en un schéma logique conforme à ce modèle. Nous terminons ce chapitre par une présentation des outils implémentant la méthode de transformation et développés pour l'environnement DB-Main. Deux outils complémentaires offrent un support à la conception logique de bases de données XML : le transformateur et l'analyseur de schémas.

Dans le **chapitre 4** consacré à la *Conception physique* et au *Codage*, nous présentons les propriétés physiques (index ...) que l'on peut définir sur une structure de données XML afin d'améliorer les performances du système. Nous montrons aussi comment générer le code d'une DTD à partir d'un schéma conforme au modèle XML.

La base de données Tamino de Software AG est certainement le SGBD XML natif le plus connu à l'heure actuelle. Cette entreprise est en outre l'une des plus actives dans ce domaine. Le **chapitre 5**, intitulé *Implantation dans un SGBD XML Natif*, présente cette société et son SGBD à succès. Nous montrons ensuite comment utiliser les différents composants logiciels de Tamino pour créer une base de données, définir la structure des documents et leurs propriétés physiques, importer des informations XML dans la base de données et les interroger.

Le **chapitre 6**, intitulé *Etude de cas* illustre, avec un exemple concret, le processus complet de conception d'une base de données XML natives. A partir de l'énoncé d'une situation, un schéma conceptuel est esquissé. A l'aide des outils présentés dans le chapitre 3, nous transformons ce schéma conceptuel en schéma logique XML. Nous discutons ensuite des valeurs à assigner aux propriétés physiques dans ce cas particulier et nous générons la DTD correspondant au schéma de données. Finalement, nous utilisons cette DTD pour définir la structure des données XML à stocker dans Tamino et nous montrons comment importer et interroger des données XML contenues dans ce SGBD.

Chapitre 2

XML : Etat de l'art

Il semble beaucoup trop ambitieux de vouloir présenter, dans ce chapitre introductif, la totalité des recommandations faites sur XML par le World Wide Web Consortium (W3C). C'est pourquoi nous proposons, dans les pages qui suivent, une introduction au langage XML (structure d'un document et syntaxe) et aux autres recommendations inclues dans la norme XML (DTD, XML Schema, XPath, XSL...). L'objectif est donc de fournir au lecteur les bases XML nécessaires pour la compréhension du processus de conception de bases de données natives. Le livre *XML : langage et applications* [Mic00] constitue une excellente référence pour ceux désirant approfondir le sujet. Tous les documents liés à la norme XML peuvent être consultés sur le site web du W3C ([Con]).

2.1 Présentation générale

Contrairement à HTML, qui est un langage défini et figé (avec un nombre de balises limité), XML peut être considéré comme un métalangage permettant de définir d'autres langages, c'est-à-dire de définir de nouvelles balises permettant de décrire le contenu du document plutôt que sa présentation. XML ne comprend donc aucune indication de formatage, mais décrit par contre de manière précise la signification des données transmises, ce qui les transforme en véritables informations qui peuvent dynamiquement être traitées par le programme récepteur d'un tel message. On peut résumer la philosophie XML en une phrase très simple : chaque donnée est accompagnée de sa propre description. XML a été mis au point par le XML Working Group sous l'égide du W3C dès 1996. Depuis le 10 fevrier 1998, les spécifications XML 1.0 ont été reconnues comme recommandations par le W3C, ce qui en fait un langage reconnu. XML est un sous ensemble de SGML (Standard Generalized Markup Language), défini par le standard ISO8879 en 1986. XML reprend la majeure partie des fonctionnalités de SGML, il s'agit donc d'une simplification de SGML afin de le rendre utilisable sur le web !

2.2 Atouts d'XML

Voici les principaux atouts de XML :

- Lisibilité : aucune connaissance ne doit théoriquement être nécessaire pour comprendre la structure et le contenu d'un document XML.
- Structure arborescente : permettant de modéliser la majorité des problèmes informatiques.
- Universalité et portabilité : les différents jeux de caractères sont pris en compte.
- Déploiement : il peut être facilement distribué par n'importe quel protocole à même de transporter du texte, comme HTTP.

- Intégrabilité : un document XML est utilisable par toute application pourvue d'un parser [1].
- Exensibilité : un document XML doit pouvoir être utilisable dans tous les domaines d'applications.

2.3 Structure d'un document XML

En réalité un document XML est structuré en 3 parties : le prologue, les instructions générales et le contenu du document.

2.3.1 Le prologue

La première partie, appelée «prologue» permet d'indiquer la version de la norme XML utilisée pour créer le document (cette indication est obligatoire) ainsi que le jeu de caractères utilisé dans le document. Ainsi le prologue est une ligne du type :

```
<?xml version="1.0" encoding="ISO8859-1"?>
```

2.3.2 Les instructions générales

Le prologue est directement suivi d'informations facultatives sur des instructions de traitement à destination d'applications particulières. Leur syntaxe est la suivante :

```
<?instruction de traitement?>
```

2.3.3 Le contenu d'un document XML

Enfin la dernière composante d'un fichier XML est une hiérarchie d'éléments vides ou non-vides. Un élément vide est uniquement composé d'une balise dont le nom se termine par un slash (/)
Exemple :

```
<Employee/>
```

Un élément non-vide est composé d'une balise ouvrante, du contenu de l'élément proprement dit et d'un balise fermante, i.e. une balise de même nom que la balise ouvrante correspondante mais commençant par un slash
Exemple :

```
<Employee>Dupont</Employee>
```

Récursivement, le contenu d'un élément peut également être une liste (ordonnée) d'éléments, c'est ce qui donne à un document XML sa structure hiérarchique. Un élément qui n'est fils d'aucun élément est appelé une racine. Dans un document XML, il n'existe qu'une et une seule racine. L'exemple ci-dessous nous montre une hiérarchie d'éléments correcte ; on dit que l'élément `Address` est l'élément «père» de deux éléments «fils», `Street` et `City`. Dans cet exemple, la racine est l'élément `Employee`.
Exemple :

[1] c'est-à-dire un logiciel permettant d'analyser un code XML

```
<Employee>
        <Name>Dupont</Name>
        <Address>
                <Street>Quality</Street>
                <City>London</City>
        </Address>
</Employee>
```

Il est interdit en XML de faire «chevaucher» des balises, c'est-à-dire d'avoir une succesion de balise du type :

```
<Employee>
        <Name>
</Employee>
        </Name>
```

Un élément XML (vide ou non-vide) peut comporter un ou plusieurs attributs. Un attribut est une paire (nom, valeur) écrit sous la forme `nom="valeur"`.
Exemple :

```
<Employee department="dept007" gender="male"/>
```

Le langage XML permet de créer ses propres balises. Toutefois, le nom choisi pour les balises doit respecter certaines règles syntaxiques. Ces règles sont également valables pour les noms d'attributs.

- le premier caractère est alphabétique ou est un underscore.
- les caractères suivants sont alphanumériques ou un caractère spécial parmis ceux-ci : tiret, underscore, point ou deux-points.
- le nom ne commence pas par le mot `xml`, qui est un préfixe réservé pour la norme.

Il est possible d'introduire des commentaires à n'importe quel endroit du document XML, grâce à la balise particulière

```
<!-- commentaire -->
```

Exemple :

```
<!-- ceci est un commentaire -->
```

Un document XML est dit «bien formé» si il est correct syntaxiquement, i.e. si il respecte toutes les règles pré-citées.

2.4 Les espaces de noms

XML est un méta-langage car il permet de définir le nom des balises d'un document ainsi que l'ordre dans lequel elles doivent apparaître. Cette abstraction de langage pose cependant un problème : il se peut que deux organismes indépendants mettent au point des langages dont certaines balises portent le même nom. Comment alors différencier ces balises si on désire utiliser des éléments des deux langages au sein d'un même document ?

La définition d'un espace de nom permet ainsi d'associer toutes les balises d'un langage à un groupe afin d'être capable de mêler différents langages à balise dans un même document XML. Afin d'assurer l'unicité d'un nom d'élément au sein d'un document, il est possible de préfixer ce nom par le nom du groupe auquel la balise est associée.

La déclaration d'un espace de noms se fait sous la forme d'un attribut dont le nom commence par `xmlns :` et dont la valeur est traditionnellement une URI servant à identifier l'espace de noms. La portée de l'espace de noms correspond à la portée de l'élément dans lequel il est déclaré. Plusieurs espace de noms peuvent être déclarés au sein d'un même élément.

Dans l'exemple ci-dessous, deux espaces de noms – `Fnac` et `CDStore` – sont déclarés dans l'élément `TheMusicShop`. Les balises `CD`, appartennant à des langages différents, sont préfixées par le nom du groupe auquel elles appartiennent.

```
<TheMusicShop xmlns:Fnac='http://www.fnac.com'
              xmlns:CDStore='http://www.cdstore.com'>
        <CDStore:CD>
                <Title>Celui qui chante</Title>
                <Band>Michel Berger</Band>
        </CDStore:CD>
        <Fnac:CD Title="Celui qui chante">
                <Price>21</Price>
                <Year>1978</Year>
        </Fnac:CD>
</TheMusicShop>
```

2.5 Définition de la structure de données XML

2.5.1 Document TypeDefinition (DTD)

Un document XML est dit «valide» s' il respecte la structure de données et la grammaire décrite dans la DTD qui lui est associée. Une DTD est donc une description d'une hiérarchie d'éléments et de leurs attributs, qui permet de valider un document. Bien que la norme n'impose pas l'utilisation d'une DTD pour un document XML, les DTD sont bien pratiques pour définir les règles d'un langage et permettre ainsi la compréhension entre deux parties, qui ne se connaîtraient pas mais qui se seraient mises d'accord sur une DTD à utiliser. Une DTD est définie selon une certaine syntaxe :

- sous une forme interne : au sein même du document.
- sous une forme externe : dans un fichier externe (.dtd) dès lors associé au document XML grâce à l'instruction `<!DOCTYPE nom_racine uri_DTD>` où `nom_racine` est le nom de l'élément racine du document et `uri_DTD` est l'adresse et le nom du fichier DTD.

Déclaration d'un élément

Pour pouvoir créer un document XML il est utile dans un premier temps de définir les éléments pouvant être utilisés. Ainsi pour définir un élément on utilisera la syntaxe suivante :

```
<! ELEMENT Nom_Element Modèle >
```

Le paramètre `Nom_Element` est le nom que l'on veut donner à l'élément
Le paramètre `Modèle` représente soit un type de donnée prédéfini, soit une règle d'utilisation de l'élément. Les types prédéfinis utilisables sont les suivants :

- ANY : L'élément peut contenir tout type de données, i.e. des chaines de caractères ou d'autres éléments non-définis.
- EMPTY : L'élément ne contient aucune donnée spécifique, il s'agit d'un élément vide.
- #PCDATA : L'élément doit contenir une chaîne de caractères quelconque.

Ainsi un élément nommé `Employee` contenant un type #PCDATA sera déclaré de la façon suivante dans la DTD :

```
<! ELEMENT Employee #PCDATA >
```

Cet élément pourra être écrit de la façon suivante dans le document XML :

```
<Employee>Dupont</Employee>
```

D'autre part il est possible de définir des règles d'utilisation, c'est-à-dire le(s) élément(s) XML (fils) qu'un élément (père) peut ou doit contenir. Dans ces règles, on spécifie le mode d'organisation des fils par rapport au père ainsi que la cardinalité des fils.

Les **modes d'organisation** sont les suivants :

Mode d'organisation	*Signification*	*Connecteur*
La séquence	tous les fils définis apparaissent dans un certain ordre	,
L'alternative	un seul des fils définis apparaît	|

Des parenthèses peuvent être utilisées afin de définir la priorité des connecteurs.

Les **cardinalités** possibles pour un fils sont résumées dans le tableau ci-dessous :

Cardinalité	*Signification*	*Opérateur*
1 à N	l'élément doit être présent au minimum une fois	+
0 à N	l'élément peut être présent plusieurs fois (ou aucune)	*
0 à 1	l'élément peut être présent une fois	?

Ainsi on peut créer la déclaration suivante dans la DTD :

```
<! ELEMENT personne (nom,prenom+,(telephone | email)>
<! ELEMENT nom #PCDATA >
<! ELEMENT prenom #PCDATA >
<! ELEMENT telephone #PCDATA >
<! ELEMENT email #PCDATA >
```

Les deux documents XML ci-dessous sont conformes à cette DTD :

```
<personne>
        <nom>Dupont</nom>
        <prenom>Jean-Francois</prenom>
        <prenom>Robert</prenom>
        <prenom>Luc</prenom>
        <email>webmaster@xml.org</email>
</personne>
```

ou

```
<personne>
        <nom>Martin</nom>
        <prenom>Jacques</prenom>
        <telephone>081/12.34.56</telephone>
</personne>
```

Déclaration d'attributs

Il est possible d'ajouter des propriétés à un élément particulier en lui affectant un attribut, c'est-à-dire une paire (nom,valeur). Ainsi dans une DTD, la syntaxe pour définir un attribut est la suivante :

```
<! ATTLIST Elément Nom_Attribut Type_Attribut Cardinalité >
```

Le paramètre `Elément` est le nom de l'élément auquel appartient l'attribut
Le paramètre `Nom_Attribut` est le nom donné à l'attribut
Le paramètre `Type_Attribut` représente le type de la valeur que peut prendre l'attribut. Nous présentons les quatre types principaux :

- CDATA : permet de définir une chaîne de caractères quelconque.
- ID : permet de définir un identifiant global au document. Cela signifie que la valeur prise par un attribut de type ID doit être unique pour tous les attributs du document ayant le type ID.
- IDREF : permet de définir un attribut de référence vers un attribut de type ID appartenant au document. La valeur prise par un attribut de ce type doit donc obligatoirement être une valeur d'un attribut de type ID défini dans le même document.
- *(énumération)* : une énumération prend la forme d'une liste (entre parenthèses) de valeurs séparées par le symbôle |. La valeur prise par un attribut de type énuméré doit être une des valeurs de la liste déclarée. `(célibataire | marié | divorcé | veuf)` est un exemple courant d'énumération pour un attribut `état civil`.

Le paramètre `Cardinalité` est un mot réservé permettant de spécifier le niveau de nécessité de l'attribut :

- #REQUIRED : signifie que l'attribut est obligatoire.
- #IMPLIED : signifie que l'attribut est facultatif.
- #FIXED *(valeur)* : signifie que l'attribut est une constante qui sera affectée de la valeur précisée dans la déclaration.
- *(valeur)* : précise la valeur par défaut de l'attribut.

Exemple d'une déclaration d'attribut :

```
<! ATTLIST disque IDdisk ID #REQUIRED >
<! ATTLIST disque type (K7 | MiniDisc | Vinyl | CD) "CD" >
```

Ce qui signifie que l'on affecte à l'élément `disque` deux attributs `IDdisk` et `type`. Le premier attribut est de type atomique, il s'agit d'un identifiant unique obligatoire. L'élément `type` peut valoir `K7`, `MiniDisc`, `Vinyl` ou `CD` mais la valeur par défaut est `CD`.

2.5.2 XML Schema

XML Schema est une recommandation du W3C publiée en 2001 et initiée afin de combler certaines lacunes des DTDs. Les XML Schemas répondent aux mêmes objectifs que les DTDs, à savoir la description de la structure des documents XML, essentiellement dans un but de validation. Certaines différences entre ces deux modèles sont cependant remarquables :

- Alors qu'une DTD respectait une syntaxe qui lui était propre, les XML Schemas ont adopté la syntaxe XML. Un XML Schema est donc désormais également un document XML.
- Les XML Schemas offrent un large pannel de types de données et autorisent diverses manipulations de ces types (restriction, extension, etc.), adoptant ainsi une perspective orientée objet.
- Les XML Schemas gèrent les espaces de noms.

- Les XML Schemas permettent une gestion beaucoup plus fine des cardinalités : il est en effet possible, selon cette norme, de préciser la cardinalité minimale et maximale d'un élément.
- Les XML Schemas, couplés à la recommandation XPath, offrent des mécanismes de gestion des contraintes référentielles et d'unicité beaucoup plus fins que les seuls attributs ID - IDREF(S) utilisés dans les DTDs.

Ces multiples apports se sont cependant traduits par un accroissement sensible de la complexité, plaçant XML Schema au centre d'une polémique portant sur sa pertinence et son utilisabilité...

2.6 XPath

XPath est une recommendation du W3C permettant de repérer un(des) noeud(s) dans un document XML. XPath n'a pas été construit pour être utilisé seul mais en association avec d'autres outils comme XSL (voir 2.7, page 22) ou XQuery (voir 5.1.3, page 77). La syntaxe définie pour XPath s'inspire fortement de la syntaxe utilisée pour dénoter les chemins («path») dans les systèmes de fichiers hiérarchiques UNIX. La localisation d'un noeud se fait soit à partir de la racine du document (chemin absolu) soit à partir d'un noeud de référence (chemin relatif).

Une expression XPath se décompose en trois parties : l'axe, le noeud et le prédicat (optionnel). La syntaxe est la suivante :

```
<axe><noeud>[<prédicat>]
```

L'expression XPath «*» est une valeur «wild-card» qui repère tous les noeuds du document XML.

2.6.1 L'axe

La relation entre la racine (ou le noeud de référence) et le(s) noeud(s) sélectionné. Les axes principaux sont :

Axe	Signification
`(rien)`	tous les fils du noeud de référence
`/`	tous les fils du noeud de référence
`.`	le noeud de référence
`..`	le noeud père du noeud de référence
`//`	tous les descendants du noeud de référence
`@`	les attributs du noeud de référence

2.6.2 Le noeud

Le nom du(des) noeud(s) sélectionné(s)

Exemples d'expressions XPath *sans prédicat*

Expression XPath	Sémantique
`chapter`	les éléments `chapter` qui sont fils du noeud de référence
`chapter/para`	les éléments `para` qui sont fils des éléments `chapter`, eux mêmes fils de l'élément de référence
`./chapter`	les éléments `chapter` qui sont fils du noeud de référence
`../chapter`	les élément `chapter` qui sont fils de l'élément parent du noeud de référence
`//line`	les éléments `line` qui sont descendants du noeud de référence
`@number`	l'attribut `number` du noeud de référence

2.6.3 Le prédicat

Une condition (facultative) que doit (doivent) respecter le(s) noeud(s) sélectionné(s).

Le prédicat peut se baser sur l'**existence** d'un élément ou d'un attribut. On indique alors l'axe et le nom de cet élément ou attribut. Exemple :

Expression XPath	Sémantique
`book/chapter[conclusion]`	tous les éléments `chapter` qui sont fils d'un élément `book` et qui possèdent un élément fils nommé `conclusion`

Le prédicat peut se baser sur la **valeur** d'un élément ou d'un attribut. On indique alors l'axe, le nom de cet élément ou attribut, ainsi que la valeur désirée pour celui-ci. Exemple :

Expression XPath	Sémantique
`chapter[@title="What's XML ?"]`	l'élément `chapter` dont l'attribut `title` vaut «What's XML ?»

Finalement, un prédicat peut inclure les opérateurs logiques `and`, `or`, `not()` et les opérateurs de comparaison `=`, `!=`, `<`, `<=`, `>` et `>=`.

2.7 XSL

Comme il a été dit, la philosophie d'XML consiste à bien séparer les données/documents (le fichier XML proprement dit) des traitements/présentations. Un document donné sera, lors de sa création, balisé uniquement en fonction de son contenu (sa sémantique) intrinsèque et indépendamment de sa restitution future (papier, écran, terminal Braille, synthèse vocale ou autre). Cette indépendance par rapport aux applications qui vont le traiter en général va lui conférer une grande interopérabilité (le même document XML va pouvoir être utilisé par une multitude d'applications de natures différentes) et une grande durabilité (le document ne deviendra pas obsolète avec l'évolution des techniques informatiques).

Ainsi XSL (eXtensible StyleSheet Language) est un langage recommandé par le W3C pour effectuer la représentation des données de documents XML (sauts de pages, liens de navigation, polices de caractères...). XSL permet donc de définir des feuilles de style pour les documents XML au même titre que les CSS (Cascading StyleSheets) pour le langage HTML. Toutefois, contrairement aux CSS, XSL permet aussi de retraiter un document XML afin d'en modifier

totalement sa structure, ce qui permet à partir d'un document XML d'être capable de générer d'autres types de documents (PostScript, HTML, Tex, RTF, ...) ou bien un fichier XML de structure différente.

XSL est lui-même défini avec le formalisme XML, cela signifie qu'une feuille de style XSL est un document XML bien formé.

XSL possède deux composantes :

- le langage de transformation des données XSLT (eXtensible Stylesheet Transformation) permettant de transformer la structure des éléments XML. Le processeur XSLT applique des transformations à un document XML source, selon des règles (appelées «template rules») contenues dans une feuille XSL. Le résultat est un arbre représentant, par exemple, la structure d'un document HTML. Chaque template rule définit des traitements à effectuer sur un élément (noeud ou feuille) de l'arbre source. On appelle «patterns» les éléments de l'arbre source.
- le langage de formatage des données (XSL/FO), c'est-à-dire un langage permettant de définir la mise en page (affichage de texte ou de graphiques) de ce qui a été créé par XSLT.

2.7.1 Structure d'un document XSL

Un document XSL étant un document XML, il commence obligatoirement par la balise suivante :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

D'autre part, toute feuille de style XSL est un document XML dont la racine est l'élément `<xsl :stylesheet ... >`

Les template rules

La balise `<xsl :stylesheet>` encapsule des balises `<xsl :template>` dans lesquelles sont contenues les templates rules, i.e. les transformations à faire subir à certains éléments du document XML utilisant la page XSL.

L'attribut `match` de la balise `<xsl :template>` permet de définir (grâce à la notation XPath) le ou les éléments du document XML sur lesquels s'applique la transformation.

Voici un exemple de feuille XSL permettant d'effectuer la transformation d'un document XML en document HTML.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
        <xsl:template match="/">
                <HTML>
                <HEAD>
                        <TITLE>Titre de la page</TITLE>
                </HEAD>
                <BODY BGCOLOR="#FFFFFF">
                        <xsl:apply-templates/>
                </BODY>
                </HTML>
        </xsl:template >
        <xsl:template match="personne" >
                <ul>
                        <li>
```

```
                                <xsl:value-of select="nom"/>
                                -
                                <xsl:value-of select="prenom"/>
                        </li>
                </ul>
        </xsl:template >
</xsl:stylesheet>
```

Cette feuille XSL possède deux template rules :
La première règle (`<xsl :template match="/">`) permet d'appliquer une transformation à l'ensemble du document (la valeur «/» de l'attribut match indique l'élément racine du document XML). Cette transformation introduit des balises HTML qui seront transcrites dans l'arbre résultat.
La seconde règle (`<xsl :template match="personne">`) précise les informations à générer dans l'arbre résultat lorsqu'une balise `personne` est rencontrée dans le document XML source.

En appliquant cette feuille XSL au document XML suivant,

```
<personne>
        <nom>Pillou</nom>
        <prenom>Jean-François</prenom>
</personne>
<personne>
        <nom>VanHaute</nom>
        <prenom>Nico</prenom>
</personne>
```

on obtient le document HTML :

```
<HTML>
        <HEAD>
          <TITLE>Titre de la page</TITLE>
        </HEAD>
        <BODY BGCOLOR="#FFFFFF">
                <ul>
                        <li>Pillou - Jean-François</li>
                        <li>VanHaute - Nico</li>
                </ul>
        </BODY>
</HTML>
```

Les éléments de transformation

Les éléments de transformations permettent de sélectionner et effectuer des opérations sur les éléments du document XML. Voici une petite liste des éléments de transformation :

<xsl :apply-templates> indique au processeur XSL de traiter tous les descendants de l'élément en cours de traitement en leur appliquant les template rules définies dans la feuille XSL. Soit le document XML suivant :

```
<voiture>
        <modèle>
```

```
                <marque>Volkswagen</marque>
                <couleur>bleu</couleur>
        </modèle>
        <immatriculation>9999 ZZ 99</immatriculation>
</voiture>
```

et la feuille XSL associée :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
        <xsl:template match="voiture">
        <Auto>
                <xsl:apply-templates/>
        </Auto>
        </xsl:template >
</xsl:stylesheet>
```

Cette feuille de style donnera le résultat suivant car aucune règle n'est définie pour les descendants de l'élément `voiture` :

```
<Auto></Auto>
```

<xsl :value-of> permet d'insérer dans l'arbre résultat le contenu de l'élément en cours de traitement (ou d'un de ses fils). L'élément dont on souhaite le contenu est précisé dans l'attribut `select` grâce à une expression XPath. En reprenant le même document XML que celui présent ci-dessus, il est possible de définir la feuille de style XSL suivante :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
        <xsl:template match="voiture">
                <table border="1">
                        <tr><th>Voiture</th></tr>
                        <tr><xsl:apply-templates/></tr>
                </table>
        </xsl:template >
        <xsl:template match="modèle">
                <td><xsl:value-of select="marque"/></td>
                <td><xsl:value-of select="couleur"/></td>
        </xsl:template >
        <xsl:template match="immatriculation">
                <td><xsl:value-of select="."/></td>
        </xsl:template >
</xsl:stylesheet>
```

Cette feuille de style donnera le résultat suivant :

```
<table border="1">
        <tr><th>Voiture</th></tr>
        <tr>
        <td>Volkswagen</td>
        <td>bleu</td>
        <td>9999 ZZ 99</td>
        </tr>
</table>
```

<**xsl :for-each**> permet d'appliquer un traitement identique à tous les fils de l'élément en cours de traitement. Le nom des éléments fils concernés est précisé dans l'attribut `select`. Ainsi, si on désire associer un même traitement à tous les chapitres du livre contenu dans ce document XML,

```
<book>
        <chapter>
                <title>A Mystery Unfolds</title>
                <paragraph>It was dark and stormy night...</paragraph>
        </chapter>
        <chapter>
                <title>A Sudden visit</title>
                <paragraph>Marcus found himself sleeping...</paragraph>
        </chapter>
</book>
```

on peut utiliser la balise `<xsl :for-each>` de cette manière :

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
        <xsl:template match="book">
                <ol>
                        <xsl:for-each select="chapter">
                                <li><xsl:value-of select="title"/></li>
                        </xsl:for-each>
                </ol>
        </xsl:template>
</xsl:stylesheet>
```

Ce qui donnera comme résultat le document HTML suivant :

```
<ol>
        <li>A Mystery Unfolds</li>
        <li>A Sudden visit</li>
</ol>
```

2.7.2 Association d'une feuille XSL à un document XML

Une feuille de style XSL (enregistré dans un fichier dont l'extension est .xsl) peut être liée à un document XML (de telle manière à ce que le document XML utilise la feuille XSL) en insérant la balise suivante au début du document XML :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet href="fichier.xsl" type="text/xsl"?>
```

Chapitre 3

Conception Logique

Dans ce chapitre, nous montrons comment il est possible de représenter la structure de données XML en utilisant les constructions propres au modèle générique G.E.R. La DTD est actuellement le formalisme le plus utilisé pour définir des données XML. Nous examinons donc comment chaque concept d'une DTD peut être transcrit dans un outil CASE implémentant le modèle G.E.R. : l'outil DB-Main. Le schéma logique ainsi défini doit respecter certaines contraintes ; l'ensemble de ces contraintes constitue le modèle logique XML.

Selon notre méthode de conception, le schéma logique doit être dérivé d'un schéma conceptuel lors de la phase de conception logique. Dans la suite du chapitre, nous nous attardons donc sur une méthode abstraite de transformation d'un schéma conceptuel en un schéma logique. Cette méthode comporte huit étapes successives.

Finalement, nous présentons les outils développés pour supporter la méthode théorique proposée : le transformateur et l'analyseur de schémas.

3.1 Le schéma conforme XML

3.1.1 Le modèle G.E.R.

Comme son nom l'indique, le modèle G.E.R. (Generic Entity Relationship) est un modèle de données suffisamment générique pour permettre la représentation de structures de données issues de multiples paradigmes (relationnel, système de fichiers, orienté objet, hiérarchique, ...) à différents niveaux d'abstraction (physique, logique ou conceptuel). Les principaux concepts admis dans ce modèle sont illustrées Fig.3.1. Cet exemple est fortement inspiré de [HEH+94]

Les **types d'entités** `DEPARTMENT` et `PERSON` sont mis en relation grâce à un **type d'associations** nommé `Works in`.

Le **rôle** joué par `DEPARTMENT` dans ce type d'associations est de **cardinalité minimale** égale à 0 et de **cardinalité maximale** égale à N. Toute occurence du type d'entités `DEPARTMENT` peut donc jouer ce rôle de 0 à N fois.

Un type d'associations jouant deux rôles est un **type d'associations binaire**.

Un type d'associations binaire dont les cardinalités maximales des deux rôles valent N est un **type d'associations N à N**.

Le type d'entités `EMPLOYEE` est caractérisés par des **attributs**. L'attribut `Name` est **atomique** alors que l'attribut `Address` est **composé**. L'attribut `CustId` est **obligatoire** alors que l'attribut `MaidenName`, de cardinalités 0-1, est **facultatif**. L'attribut `Address` est **monovalué** alors que l'attribut `Phone`, de cardinalités 1-5, est **multi-valué**. Même si cette propriété n'apparaît pas sur le schéma, signalons qu'un type et une longueur peuvent être définis pour chaque attribut. Les types les plus courants sont «entier», «chaîne de caractères» ou «booléen».

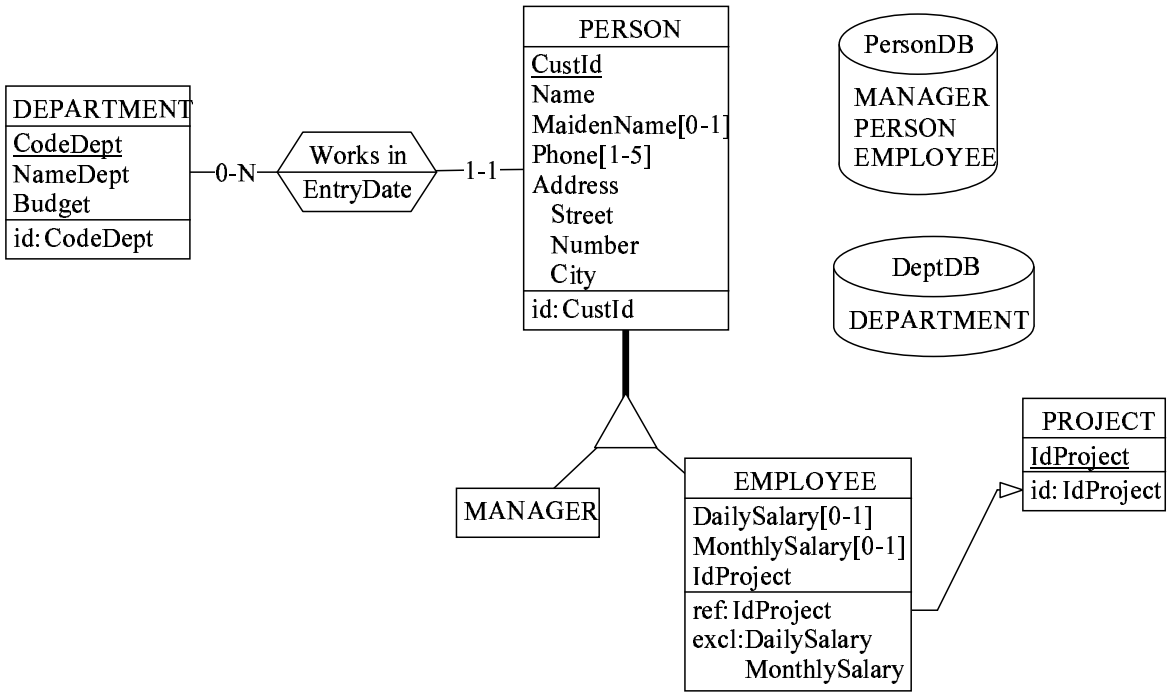

FIG. 3.1 – Représentation graphique des principaux concepts propres au modèle G.E.R.

L'attribut `CustId` du type d'entités `EMPLOYEE` appartient à un groupe identifiant, appelé **groupe «id»**. Deux occurences du type d'entités `EMPLOYEE` ne peuvent avoir la même valeur pour cet attribut.

Le type d'associations `Works in` est caractérisé par un attribut : `EntryDate`.

Le type d'entités `PERSON` est le **sur-type** d'une relation de généralisation/spécialisation appelée **relation IS-A**. Les types d'entités `MANAGER` et `EMPLOYEE` en sont les **sous-types**. Ils héritent de toutes les propriétés (attributs, rôles ...) du sur-type. Une contrainte peut être associée aux sous-types d'une relation IS-A ; elle est représentée par la lettre **D** pour Disjonction, **T** pour Totalité ou **P** pour Partition.

Les attributs facultatifs `DailySalary` et `MonthlySalary` du type d'entités `EMPLOYEE` appartiennent à un groupe associé à une contrainte d'exclusivité, appelé **groupe «excl»**. Si un des attributs du groupe contient une valeur, alors les autres ne doivent pas en posséder. D'autres types de contraintes peuvent être associées à un groupe ; citons les **contraintes «exact-1»** (un et un seul composant du groupe est défini) et **«at-least-1»** (au moins un composant du groupe est défini).

L'attribut `IdProject` du type d'entités `EMPLOYEE` appartient à un **groupe «ref»**. Ce groupe est l'origine d'une **clé étrangère** dont la destination est le groupe «id» défini dans le type d'entités `PROJECT`. La technique des clés étrangères nous vient du modèle relationnel et peut substituer un type d'associations 1 à N. La **contrainte référentielle** associée à cette clé étrangère est simple : l'ensemble des valeurs prises par l'attribut `EMPLOYEE.IdProject` **doit** être inclu dans l'ensemble des valeurs prises par l'attribut `PROJECT.IdProject`.

`PersonDB` et `DeptDB` sont des **collections**. Il s'agit d'un groupement de types d'entités ayant un point commun (au niveau conceptuel, logique ou physique).

Générique et indépendant de toute technologie, ce modèle G.E.R. peut donc être utilisé pour la représentation de la structure de données XML. Cette structure peut être décrite selon les deux mécanismes établis par le consortium W3C, responsable du développement de XML :

la DTD et – plus récemment – le XML Schema. Pour une raison de compatibilité avec le SGBD cible choisi [1], c'est uniquement sur la norme des DTD que porte ce travail [2].

3.1.2 L'outil de création d'un schéma conforme au modèle G.E.R.

Développé par le Laboratoire d'Ingénierie de Bases de Données des Facultés Universitaires de Namur, DB-Main est un outil CASE qui offre un support à l'ingénierie et rétro-ingénierie des bases de données. La version actuelle (version 6.5 - 28 mars 2002) supporte notamment les fonctionnalités suivantes :

- création, mise à jour et visualisation de schémas de données conformes à différents modèles (G.E.R., UML, COBOL ...) et à différents niveaux d'abstraction (conceptuel, logique, physique).
- représentation de l'historique d'un projet (processus exécutés et produits créés).
- différents modes de visualisation d'un même schéma.
- analyse de schémas et validation par rapport à un modèle défini.
- application d'opérateurs de transformation de schémas.
- génération automatique de code (SQL, COBOL, CODASYL ...) et de rapports.
- analyse de code source et détection de patterns.
- une série d'assistants destinés à aider l'utilisateur dans certaines tâches plus complexes telles la transformation, la validation ou l'intégration de schémas.
- définition et exécution de méthodologies.
- possibilité d'étendre les fonctionnalités en utilisant le langage de développement interne Voyager2.

DB-Main permet donc de créer, transformer, valider ... des schémas conformes à un pseudo-modèle G.E.R. Pour créer ces schémas, l'utilisateur a à sa disposition une boîte à outils qu'il utilise pour manipuler types d'entités, types d'associations, attributs... et tout autre concept propre à ce modèle. La Fig.3.2 montre l'environnement de travail de DB-Main. Pour de plus amples détails sur l'utilisation de cet outil CASE, veuillez vous référer à [Hai99]. La souplesse de l'outil permet même de personnaliser certains éléments du schéma. Ainsi, il est possible de définir de nouveaux types d'attributs, voire de créer des groupes autres que ceux cités ci-dessus («id», «ref», «excl», «exact-1» et «at-least-1»). La création de types d'attributs ou de groupes «user-defined» sera exploitée pour représenter des concepts propres au modèle XML.

3.1.3 Conventions de représentation d'un schéma conforme XML dans DB-Main

Afin de permettre à l'utilisateur de schématiser la structure de données de son futur document XML, il a donc fallu définir quelques conventions de représentation d'une DTD dans l'outil DB-Main. Cette représentation d'une DTD dans DB-Main est ce que l'on appelle un schéma conforme au modèle de données XML [3].

Dans cette section, nous examinons donc les concepts-clés des documents XML et nous expliquons comment ils peuvent être représentés en utilisant les objets - type d'entités, type d'associations, cardinalités, attributs... - à disposition dans l'outil CASE DB-Main. Nous nous limitons à la présentation des conventions principales ; l'objectif étant de comprendre plus aisément le plan de transformation détaillé dans la suite du chapitre. Les conventions complètes de représentation peuvent être trouvées dans [Del01a].

[1] Pour plus de précision sur la justification de ce choix, voir la présentation de l'éditeur de schémas Tamino, page 80

[2] Des recherches actuellement menées par la cellule REVERSE du CETIC (Gosselies) tentent d'établir un modèle similaire pour la norme plus récente des XML Schemas

[3] A ne pas confondre avec la norme des XML Schemas, citées ci-dessus

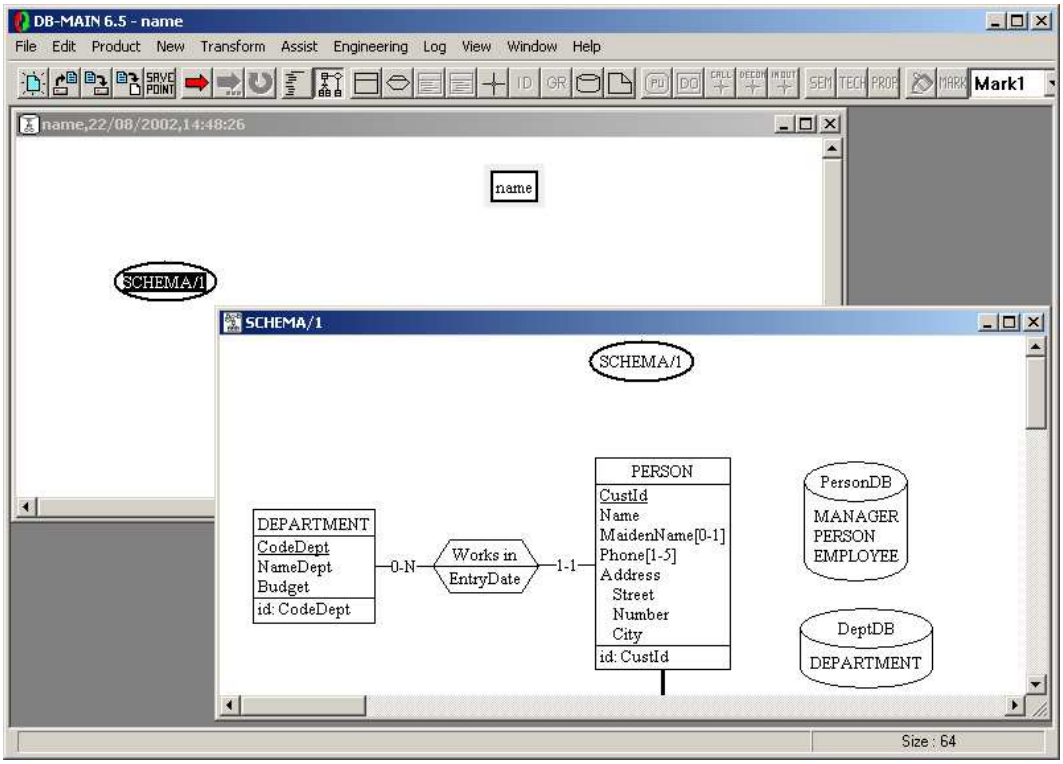

FIG. 3.2 – L'outil CASE DB-Main : création d'un schéma conceptuel conforme au modèle G.E.R.

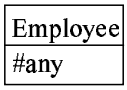

FIG. 3.3 – Représentation dans le modèle de données XML d'un type d'éléments ANY

Représentation d'un type d'éléments XML

L'élément de base d'un document XML est le type d'éléments. Dans un schéma conforme XML, un type d'éléments XML est représenté par un type d'entités dont le nom est le nom du type d'éléments XML.

Il est à signaler que les noms d'éléments, en XML, sont case-sensitives alors que, dans DB-Main, les noms des types d'entités ne le sont pas. Il est par conséquent impossible, dans le schéma conforme XML, de représenter deux types d'éléments uniquement différents de par leur casse.

Type de contenu ANY Un type d'éléments XML pouvant contenir d'autres types d'éléments ou des chaînes de caractères quelconques est un type d'éléments de type ANY. Ce type d'éléments est représenté par un type d'entités qui ne possède qu'un seul attribut nommé #any, de type quelconque. La Fig.3.3 illustre cette règle pour la déclaration suivante :

```
<!ELEMENT Employee ANY>
```

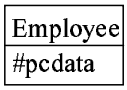

FIG. 3.4 – Représentation dans le modèle de données XML d'un type d'éléments PCDATA

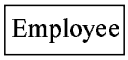

FIG. 3.5 – Représentation dans le modèle de données XML d'un type d'éléments EMPTY

Type de contenu PCDATA Un type d'éléments XML pouvant uniquement contenir une chaînes de caractères quelconques (parsed character data) est un type d'éléments de type PCDATA. Ce type d'éléments est représenté par un type d'entités qui ne possède qu'un seul attribut nommé `#pcdata`, de type quelconque. La Fig.3.4 illustre cette règle pour la déclaration suivante :

```
<!ELEMENT Employee (PCDATA)>
```

Type de contenu EMPTY Un type d'éléments XML peut ne contenir aucune information à l'intérieur de ses bornes ; il s'agit alors d'un type d'éléments EMPTY. Ce type d'éléments est représenté par un type d'entités qui ne possède aucun attribut. La Fig.3.5 illustre cette règle pour la déclaration suivante :

```
<!ELEMENT Employee EMPTY>
```

Les types d'éléments ANY, PCDATA et EMPTY sont des types d'éléments appelés «feuilles» - car ils ne contiennent pas d'autres types d'éléments définis dans la DTD - qui sont représentés dans le schéma conforme XML par des types d'entités dits «techniques».

Type de contenu ELEMENT

Notion de «groupe» Considérons, par exemple, la déclaration suivante :

```
<!ELEMENT Employer (Name, (Mail|Phone|Fax)+ , Activity*, ApplyProc)*>
```

Si on analyse le contenu d'un type d'éléments de contenu ELEMENT, on constate qu'il est formé d'un certain nombre de constituants. Chaque constituant est :

- soit un type d'éléments, éventuellement soumis à un opérateur de cardinalité (?, *, +). Dans ce cas, on parle de **constituant élémentaire**.
- soit un groupe de plusieurs constituants organisés sous forme de séquence ou de choix et regroupés au sein de parenthèses. Dans ce cas, on parle de **constituant composite** ou – plus simplement – de **groupe**. Un groupe peut également être soumis à un opérateur de cardinalité.

Dans la suite de ce document, nous désignerons par groupe, tous les regroupements délimités par des parenthèses, **y compris les parenthèses extérieures quand elles sont soumises à un opérateur de cardinalité**.
Ainsi, dans l'exemple ci-dessus, le contenu du type d'éléments `Employer` est formé d'un groupe délimité par les parenthèses les plus extérieures car celles-ci sont soumises a l'action de l'opérateur de cardinalité *. Ce groupe est formé successivement :

- d'un constituant élémentaire `Name`.
- d'un groupe soumis à l'opérateur + et composé de constituants élémentaires `Mail`, `Phone` et `Fax`.
- de deux constituants élémentaires `Activity` et `ApplyProc`, soumis à l'opérateur *.

Bien que cette terminologie puisse paraître étrange, elle trouve sa justification dans le fait que les parenthèses utilisées dans le contenu d'un type d'éléments ont un double rôle :

- délimiter la portée de l'opérateur de cardinalité. Les constituants concernés forment alors un **groupe**.
- séparer le nom du type d'éléments de la définition de son contenu. C'est le cas des parenthèses les plus extérieures.

Notion de «relation hiérarchique» Nous désignons par **relation hiérarchique**, le lien qui existe entre un type d'éléments (appelé père) et son contenu (appelé fils) ou entre un groupe (appelé père) et ses différents constituants (appelés fils).

Règles de représentation d'un type d'éléments de contenu ELEMENT La suite de cette section donne la succession des règles de représentation d'un type d'éléments de contenu ELEMENT. Ces règles seront illustrées sur l'exemple du type d'éléments `Employer`, déclaré ci-dessus.

Règle 1 : Chaque constituant (élémentaire ou groupe) du contenu du type d'éléments est représenté par un type d'entités. Si le constituant est :

- élémentaire, le type d'entités qui le représente porte le même nom.
- composite, le type d'entités qui le représente porte un nom quelconque et est soumis au stéréotype «tech».

Fig.3.6 illustre cette première règle : le type d'éléments `Employer` est représenté par un type d'entités de même nom. Son contenu est un groupe représenté par un type d'entités technique (appelé `GR`). Ce groupe a quatre constituants :

- trois constituants élémentaires (`Name`, `Activity` et `ApplyProc`), représentés, tous trois, par un type d'entités de même nom.
- un groupe de trois constituants élémentaires, représenté par un type d'entités technique dont le nom peut être librement choisi (ici, `GR1`). Les trois constituants de ce groupe donnent lieu à trois types d'entités nommés `Mail`, `Phone` et `Fax`.

Règle 2 : Chaque relation hiérarchique est représentée par un type d'associations binaire, sans attribut et dont le nom est quelconque. Dans un type d'associations, la cardinalité du rôle joué par le fils vaut toujours [1-1] sauf si le fils a plusieurs pères, auquel cas la cardinalité vaut [0-1]. Nous renvoyons au paragraphe suivant pour la justification de ce choix. Quant à la cardinalité du rôle joué par le père, elle dépend de l'opérateur de cardinalité auquel est soumis le constituant correspondant au fils dans la DTD :

- si l'opérateur de cardinalité est ?, la cardinalité du rôle est [0-1].
- si l'opérateur de cardinalité est *, la cardinalité du rôle est [0-N].
- si l'opérateur de cardinalité est +, la cardinalité du rôle est [1-N].
- si il n'y a pas d'opérateur de cardinalité, la cardinalité du rôle est [1-1].

Dans un type d'associations représentant une relation hiérarchique, le rôle joué par le père doit être nommé «f» (abrégé de **f**ather). Cette information permettra d'identifier le type d'entités père dans certains cas limites. Cette seconde règle, complétant la première, est illustrée Fig.3.7 : la relation hiérarchique existant entre le type d'entités `Employer` et le groupe correspondant à son contenu est modélisée par un type d'associations joignant le type d'entités `Employer` au type d'entités technique `GR` représentant ce groupe. Le rôle joué par `Employer` a la cardinalité [0-N] puisque le groupe correspondant aux parenthèses extérieures est soumis à l'opérateur de

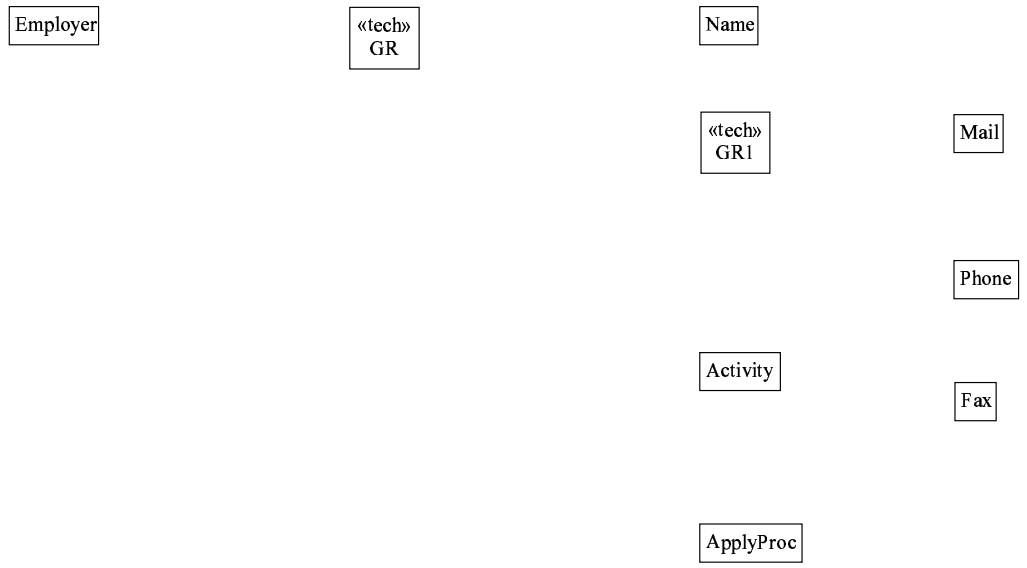

FIG. 3.6 – Règle 1 : Représentation des types d'éléments et des groupes

cardinalité *. Le rôle joué par `GR` vaut [1-1] car `GR` n'a qu'un seul père. `GR` a quatre fils auxquels il est lié par le biais de types d'associations et correspondant aux quatre constituants qui le forment. Seuls ses deuxième et troisième constituants sont soumis à un opérateur de cardinalité (resp. + et *). Les rôles joués par GR auront donc respectivement les cardinalités suivantes : [1-1], [1-N], [0-N] et [1-1].

Règle 3 : Il reste finalement à ajouter au schéma le mode d'organisation des constituants d'un groupe.

Les constituants au sein d'un groupe peuvent d'une part être ordonnés, donc organisés sous forme de **séquence**. Dans ce cas, ils sont séparés par une virgule dans la DTD. Dans le schéma, on exprime ce mode d'organisation par un groupe – au sens DB-Main – soumis à une contrainte «seq». Ce groupe «seq» appartient au type d'entités représentant le constituant composite et est composé des rôles joués par les constituants. Signalons que l'ordre des rôles dans un groupe «seq» est important car il détermine l'ordre d'apparition des fils dans le document XML. L'existence d'un groupe «seq» n'est pertinente que pour les types d'entités qui possèdent plusieurs fils.

Les constituants d'un groupe peuvent également présenter les diverses **alternatives** d'un choix. Dans ce cas, ils sont séparés par une barre verticale dans la DTD. Dans le schéma, on exprime ce mode d'organisation par un groupe – au sens DB-Main – soumis à une contrainte «choice». Ce groupe «choice» appartient au type d'entités représentant le constituant composite et est composé des rôles joués par les constituants.

La Fig.3.8 illustre cette troisième règle : un groupe «seq» a été ajouté au type d'entités `GR`. Il contient les rôles joués par les types d'entités fils : `Name`, `GR1`, `Activity` et `ApplyProc`. Un groupe «choice» a également été ajouté au type d'entités `GR1` pour représenter l'alternative existant entre les trois types d'entités fils : `Mail`, `Phone` et `Fax`.

La cardinalité des rôles fils En règle générale, la cardinalité d'un rôle fils est 1-1. Il est cependant possible qu'un type d'éléments fils ait plusieurs pères, à condition qu'il n'en ait qu'un seul à la fois au niveau des instances du document XML. Dans ce cas, la cardinalité du rôle fils vaut 0-1 et un contrainte «exact-1» est associée aux rôles pères correspondants. Pour

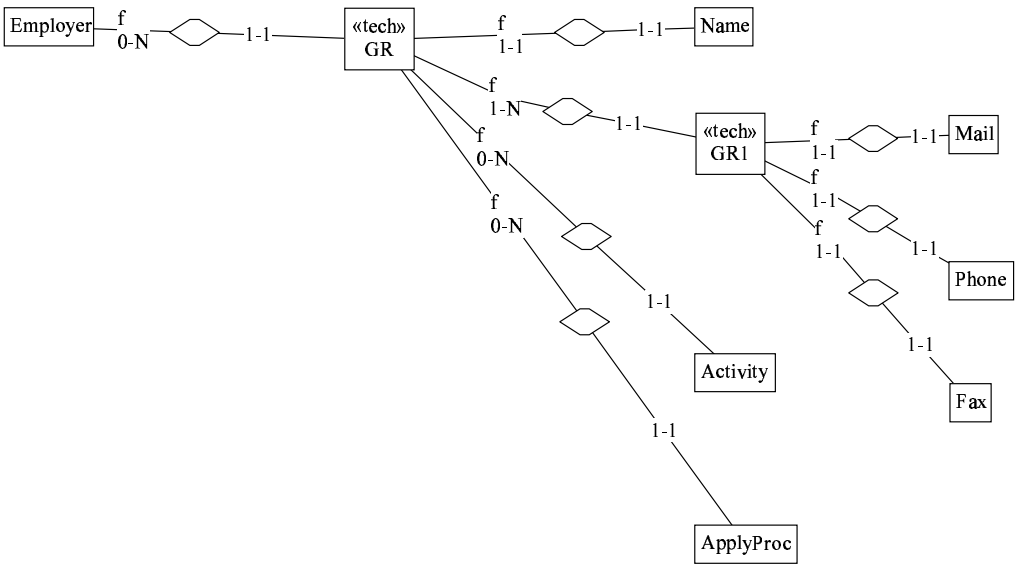

FIG. 3.7 – Règle 2 : Représentation des relations hiérarchiques

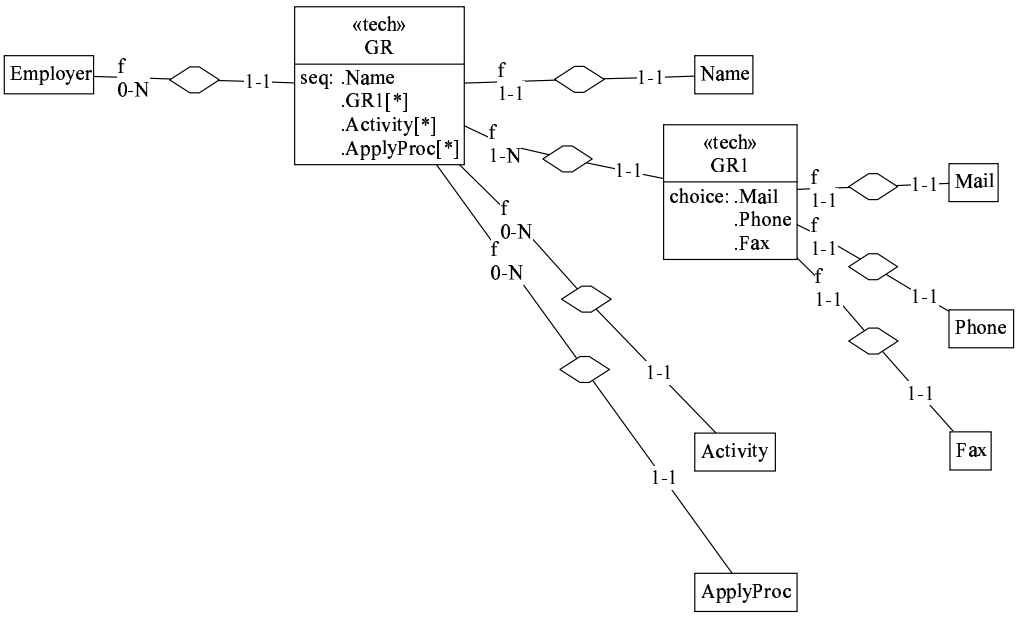

FIG. 3.8 – Règle 3 : Représentation des modes d'organisation des constituants d'un groupe

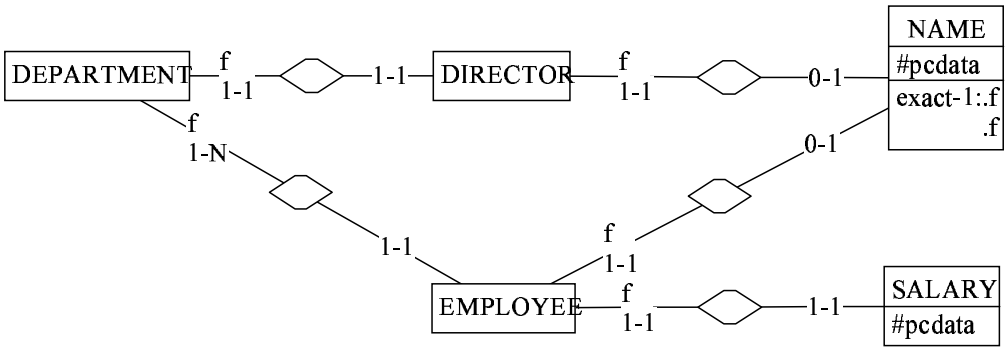

FIG. 3.9 – Le type d'entités Name possède plusieurs pères mais possède un groupe «exact-1» contenant les rôles joués par ces pères

illustrer ce cas particulier, considérons la DTD suivante :

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT Department (Director, Employee+)>
<!ELEMENT Director (Name)>
<!ELEMENT Employee (Name, Salary)>
<!ELEMENT Name (#PCDATA)>
<!ELEMENT Salary (#PCDATA)>
```

et un document XML conforme à cette DTD :

```
<?xml version="1.0" encoding="UTF-8"?>
<Department>
        <Director>
                <Name>Dupont</Name>
        </Director>
        <Employee>
                <Name>Martin</Name>
                <Salary>20</Salary>
        </Employee>
        <Employee>
                <Name>Robert</Name>
                <Salary>15</Salary>
        </Employee>
</Department>
```

On remarque que le type d'éléments `Name` – défini une seule fois dans la DTD – peut être aussi bien le fils du type d'éléments `Director` ou du type d'éléments `Employee` mais qu'au niveau du document XML, ce type d'éléments `Name` ne peut en aucun cas être fils des deux pères, à la fois.

Dans le modèle XML que nous définissons, cet exemple particulier se représente par un type d'entités `Name`, dont les pères sont `Director` et `Employee`. Les rôles fils joués par `Name` sont de cardinalités [0-1]. Un groupe «exact-1» contient les rôles des pères. Cette situation, illustrée Fig.3.9, est l'unique cas autorisé d'un fils acceptant plusieurs pères.

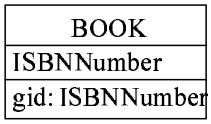

FIG. 3.10 – La représentation des attributs XML de type ID

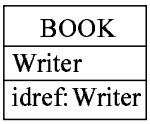

FIG. 3.11 – La représentation des attributs XML de type IDREF

Représentation d'un attribut XML

Dans le schéma logique XML, un attribut XML est représenté par un attribut du type d'entités représentant le type d'éléments XML auquel cet attribut appartient. Cet attribut peut être facultatif mais est obligatoirement simple (atomique) et mono-valué.

L'attribut de type ID Un attribut XML de type ID est représenté par un attribut de type quelconque, appartenant à un groupe nommé «gid» (en référence à «global id»). Ce groupe contient un et un seul attribut. La Fig.3.10 schématise la déclaration suivante :

```
<!ELEMENT BOOK EMPTY>
<!ATTLIST BOOK ISBNNumber ID #REQUIRED>
```

L'attribut de type IDREF Un attribut XML de type IDREF est représenté par un attribut de type quelconque appartenant à un groupe nommé «idref». Comme cet attribut référence un attribut simple de type ID, il ne peut y avoir qu'un seul attribut par groupe «idref». La Fig.3.11 montre comment la déclaration suivante est schématisée :

```
<!ELEMENT BOOK EMPTY>
<!ATTLIST BOOK Writer IDREF #REQUIRED>
```

Finalement, les types d'attributs CDATA sont représentés par des attributs du type user-defined «cdata».

Grâce à ces quelques conventions, une DTD peut donc être représentée par un schéma conforme XML dans l'environnement DB-Main.

3.2 Validation d'un schéma conforme XML

Dans la section précédente, nous avons montré comment représenter graphiquement une DTD dans l'environnement DB-Main. Il est possible de décrire le modèle XML de manière plus formelle en énonçant les propriétés que chaque structure (schéma, type d'entités, type d'associations...) se doit de posséder pour être conforme. Ces propriétés seront alors ensuite implémentées sous forme de prédicats dans un outil de validation de schéma (voir 6.19, page 103)

Voici donc les propriétés que tout schéma conforme XML doit absolument vérifier.

3.2.1 Contraintes sur le schéma

- Un schéma conforme XML ne contient pas de collection.
- Un schéma conforme XML ne contient pas de relation IS-A.
- Un schéma conforme XML contient une et une seule racine.

3.2.2 Contraintes sur les types d'entités

- Dans un schéma conforme XML, un type d'entités possède au plus un groupe soumis à une user-defined contrainte «gid» : cette contrainte traduit le fait que dans une DTD, un type d'éléments a au plus un attribut de type ID.
- Dans un schéma conforme XML, un type d'entités ayant plusieurs pères possède un groupe «exact-1» constitué des rôles joués par tous ses pères.
- Dans un schéma conforme XML, le nom d'un type d'entités non-technique se conforme à la règle des noms des types d'éléments du langage XML : un type d'entités non-technique représente un type d'éléments de la DTD. Son nom correspond donc au nom d'un type d'éléments de la DTD et suit les règles de nomination des éléments XML (voir 2.3.3, page 16).
- Dans un schéma conforme XML, un type d'entités non-technique ne possède aucun attribut nommé #pcdata ou #any.
- Dans un schéma conforme XML, un type d'entités technique possède au plus un attribut nommé #pcdata ou #any.
- Dans un schéma conforme XML, un type d'entités qui a moins de deux fils ne peut contenir de groupes soumis à une contrainte user-defined nommée «choice» ou «seq».
- Dans un schéma conforme XML, un type d'entités qui a au moins deux fils possède un groupe soumis à une user-defined contrainte nommée «choice» ou «seq». Ce groupe contient l'ensemble des rôles joués par les fils du type d'entités.

3.2.3 Contraintes sur les types d'associations

- Dans un schéma conforme XML, un type d'associations est binaire.
- Dans un schéma conforme XML, un type d'associations ne possède pas de groupe.
- Dans un schéma conforme XML, un type d'associations ne possède pas d'attributs.
- Dans un schéma conforme XML, un type d'associations a un et un seul rôle père, i.e. a un et un seul rôle nommé «f».

3.2.4 Contraintes sur les rôles

- Dans un schéma conforme XML, la cardinalité des rôles pères est 0-1, 1-1, 0-N ou 1-N ; tandis que celle des rôles fils est 1-1 si le type d'entités n'au qu'un seul père et 0-1 sinon : dans un schéma conforme XML, un type d'associations est utilisé pour représenter la relation hiérarchique entre un type d'éléments père et un type d'éléments fils. La cardinalité du rôle père représente l'opérateur de cardinalité auquel est soumis le type d'éléments fils. Ces opérateurs sont ?, *, ou +, correspondant respectivement aux cardinalités 0-1, 0-N et 1-N. L'absence de cardinalité se traduit par une cardinalité 1-1.
- Dans un schéma conforme XML, il n'existe pas de rôles multi-entités : cela revient à vérifier que le nombre de type(s) d'entités impliqué(s) dans chaque chaque rôle est égal à 1.

3.2.5 Contraintes sur les attributs

- Dans un schéma conforme XML, il n'existe pas d'attributs composés.

- Dans un schéma conforme XML, il n'existe pas d'attributs multi-valués.
- Dans un schéma conforme XML, un attribut vérifie une et une seule des conditions suivantes :
  - il est nommé #any, est de type quelconque, est le seul attribut du type d'entités auquel il appartient et ne fait partie d'aucun groupe.
  - il est nommé #pcdata, est de type quelconque, est le seul attribut du type d'entités auquel il appartient et ne fait partie d'aucun groupe.
  - il est de type quelconque et est l'unique composant d'un groupe soumis à une contrainte user-defined nommé «gid».
  - il est de type quelconque et est l'unique composant d'un groupe soumis à une contrainte user-defined nommé «idref».
  - il est de type user-defined «cdata».
- Dans un schéma conforme XML, un attribut dont le nom n'est pas #pcdata ou #any a un nom qui se conforme à la règle des noms des types d'éléments du langage XML : la règle des noms pour les attributs XML est identique la règle des noms des types d'éléments XML (voir 2.3.3 page 16).

3.2.6 Contraintes sur les groupes

- Dans un schéma conforme XML, tout groupe est soumis à l'une des contraintes suivantes : «gid», «idref», «seq», «choice», «exact-1».
- Dans un schéma conforme XML, un groupe «gid» ou «idref» ne contient qu'un et un seul attribut.
- Dans un schéma conforme XML, un groupe «seq» ou «choice» ne contient que des rôles.
- Dans un schéma conforme XML, un groupe «exact-1» dans un type d'entités contient les rôles joués par tous ses pères.
- Dans un schéma conforme XML, tout groupe n'est composé d'aucun autre groupe.

3.3 Transformation de schémas

3.3.1 Principes généraux

De manière générale, une **transformation de schéma** consiste en une modification (globale ou locale) d'un schéma source S afin d'obtenir un schéma cible S' ([HEH+94]). L'ajout d'un attribut à un type d'entités ou la substitution d'un type d'associations par un type d'entités équivalent sont deux exemples de transformation de schéma.

Une transformation Σ se note $< T, t >$ ou $< P, Q, t >$ où

- T est un **mapping structurel** qui remplace un type de constructions C appartenant au schéma S en un type de constructions C'. Mathématiquement, C' est donc le résultat de l'application de l'opérateur T sur un type de constructions C : $C' = T(C)$. C doit répondre à certaines préconditions P et le résultat C' répond aux postconditions Q. T représente la *syntaxe* de la transformation.
- t est un **mapping d'instance** qui définit comment obtenir une instance de C' à partir d'une instance de C. Mathématiquement, si c est une instance de C, alors $c' = t(c)$ est l'instance correspondant à $T(C)$. t représente la *sémantique* de la transformation.

3.3.2 Réversibilité

1. Une transformation $\Sigma_1 = < T_1, t_1 > = < P_1, Q_1, t_1 >$ est dite **réversible** ssi :
   $\exists$ une transformation $\Sigma_2 = < T_2, t_2 > = < P_2, Q_2, t_2 >$ tel que $\forall$ C (construction), $\forall$ c (instance de C) :

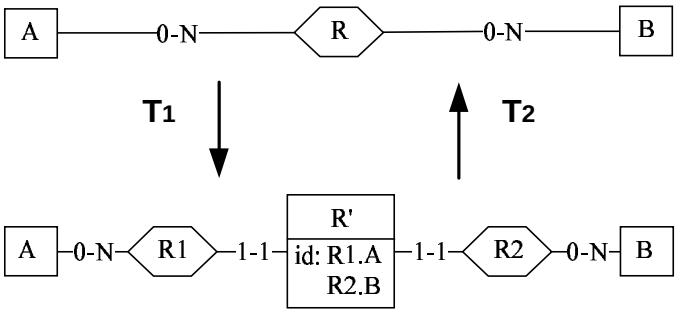

FIG. 3.12 – Transformation d'un type d'associations N à N en un type d'entités : représentation des mappings structurels T_1 et T_2

$$P_1(C) \Longrightarrow [\ T_2(T_1(C)) = C\ ] \text{ et } [\ t_2(t_1(c)) = c\ ]$$

Σ_2 est donc l'inverse de Σ_1 mais le contraire n'est pas spécialement vrai : une instance c' de $T(C)$ peut ne pas satisfaire la propriété suivante : $c' = t1(t2(c'))$.

2. Si Σ_2 est également réversible alors Σ_1 et Σ_2 sont **symétriquement réversibles**. Dans ce cas, $\Sigma_2 = < Q_1, P_1, t_2 >$ et l'unique notation $\Sigma = < P, Q, t_1, t_2 >$ est utilisée pour représenter les deux transformations.

   Les transformations symétriquement réversibles sont particulièrement souhaitées en ingénierie des bases de données car elles possèdent la propriété de conservation de la sémantique : lorsqu'une transformation symétriquement réversibles est appliquée à un schéma S, la sémantique du schéma S' ainsi obtenu est absolument identique à la sémantique du schéma S original. Lors de la conception logique, les transformations appliquées à un schéma conceptuel pour dériver un schéma conforme à un modèle logique doivent idéalement bénéficier de cette qualité. Malheureusement, certains modèles logiques étant si pauvres sémantiquement que cela n'est pas toujours possible. Le schéma logique est alors moins riche que le schéma conceptuel. On peut décider de laisser tomber les contraintes originales non représentées dans le schéma logique ou alors de les implémenter dans du code opératoire annexe.

3.3.3 Exemple

Les jeux de transformation implémentés dans DB-Main sont symétriquement réversibles. Un exemple typique est la transformation d'un type d'associations N à N en un type d'entités. La Fig.3.12 montre comment un type d'associations N à N (i.e. C) peut être remplacé par deux types d'associations 1 à N et un identifiant (i.e. $T(C)$). Cet exemple générique est une manière de décrire T, le *mapping structurel*.

3.4 Transformation d'un schéma conceptuel en schéma logique XML

Cette méthode de transformation, constituée de 8 étapes successives, s'applique sur un schéma entités-associations conforme au modèle défini par le G.E.R. L'objectif est de transformer ce schéma conceptuel en un schéma logique conforme au modèle XML. Ce schéma logique XML devra donc respecter toutes les contraintes du modèle XML (décrites dans la section précédente) tout en conservant la sémantique du schéma conceptuel original. Le modèle logique XML étant par définition plus contraint que le modèle du G.E.R., les transformations

proposées ne sont pas toutes symétriquement réversibles et nous verrons que dans certains cas, une perte de sémantique est inévitable lors du passage vers le schéma logique XML.

Il est à préciser que la méthode actuelle ne supporte la transformation ni de schéma récursifs, ni de schémas possédant une hiérarchie multiple. Un schéma est récursif si un type d'entités est lui-même présent dans la liste de ses propres descendants. Une hiérarchie est dite «multiple» si un sous-type possède plusieurs pères. Ces hypothèses faites sur le schéma de départ sont tout à fait acceptable et n'empêchent pas la modélisation d'une majeure partie des structures de données.

Cette méthode est loin d'être déterministe et à partir d'un même schéma de départ, la méthode peut produire comme résultats plusieurs schémas conformes XML différents. Certaines étapes requièrent en effet l'intervention de l'utilisateur qui devra effectuer certains choix sémantiques, ce qui influencera le résultat final. La plupart des étapes de la méthode restent toutefois entièrement déterministes et peuvent donc être automatisées dans un outil de transformation.

La première étape consiste à appliquer au schéma de départ des transformations symétriquement réversibles bien connues et bien maîtrisées pas les designers de base de données relationnelles. Ces transformations cherchent à éliminer des structures non-conformes au modèle XML, telles que les relations IS-A, les types d'associations complexes et les attributs composés et multivalués.

Lors de la deuxième étape, l'utilisateur crée la structure hiérarchique du schéma : il choisit les racines du document (i.e. les types d'entités qui représentent des concepts suffisamment importants pour être représentés en haut de la hiérarchie) et intervient dans l'élaboration des relations hiérarchiques entre ces racines et les autres types d'entités du schéma. Dans une structure arborescente, un type d'entités ne peut posséder qu'un seul père. Par conséquent, certains types d'associations ne pourront être transformés en relations hiérarchiques et deviendront des relations de référence (mécanisme semblable aux clés étrangères existant dans le modèle relationnel bien que se situant à un niveau global, voir section 2.5.1, page 20). Si le schéma possède plusieurs racines, un type d'entités (appelé *racine technique*) est ajouté comme père des racines choisies initialement par l'utilisateur (appelées *racines naturelles*).

La troisième étape modifie les cardinalités de certains rôles qui ne peuvent être représentées dans le modèle XML. Pour rappel, les cardinalités acceptées dans le modèle-cible sont : 1-1, 0-1,0-N et 1-N.

La quatrième étape transforme les identifiants simples et non-composés en groupe «gid» et les relations de référence en groupes «idref». Les groupes identifiants composés ou secondaires seront supprimés dans une étape ultérieure.

La cinquième étape se charge de transformer les groupes issus des relations IS-A afin de conserver la sémantique originale de ces relations.

La sixième étape concerne les attributs non-impliqués dans un groupe «gid» ou «idref». Selon la préférence de l'utilisateur, ces attributs peuvent être transformés en attributs ou en éléments XML.

La septième étape crée un groupe «seq» dans un type d'entités lorsque celui-ci possède plusieurs fils.

A l'issue de ces sept premières étapes, certains groupes non-acceptés dans le modèle-cible peuvent subsister. La huitième étape élimine ces groupes non-conformes.

Dans cette section, nous spécifions de manière formelle chacune des 8 étapes qui constituent cette méthode en détaillant le type de traitement (automatique/interactif), le traitement réalisé sur chaque élément du schéma et les différents choix offerts à l'utilisateur. Pour chaque étape, nous récapitulons les contraintes que doit respecter le schéma **après** exécution de la transformation.

Avant de détailler la première des étapes, rappelons que cette méthode s'applique sur un schéma entités-associations non-récursif, conforme au modèle du G.E.R. et ne possédant pas

de hiérarchies multiples.

3.4.1 Étape 1 : Transformation des relations IS-A, des types d'associations non-conformes, des attributs multivalués et composés

Type :

automatique

Traitement :

Située en amont de l'étape principale de construction de la hiérarchie, cette première étape, entièrement automatique, prépare le terrain en appliquant successivement des transformations simples et réversibles au schéma de départ.

Les relations que l'on rencontre dans une structure arborescente XML sont de 2 types : la relation hiérarchique père-fils entre 2 éléments XML ou la relation de référence entre 2 éléments quelconques, matérialisée par l'utilisation d'attributs de références ID et IDREF. Ces 2 types de relations jouissent cependant des mêmes propriétés : elles sont *binaires*, de *cardinalités maximales 1-1 ou 1-N* et *dépourvues d'attributs*. En transformant les types d'associations ne répondant pas à ces trois exigences en types d'entités, on obtient aisément des types d'associations compatibles avec le modèle XML, et cela sans aucune perte de sémantique.

Pour qu'un type d'associations du schéma conceptuel puisse être représenté par une relation XML hiérarchique, il est nécessaire que :

- l'un des deux rôles soit de cardinalité 1-1 (ou 0-1 assortis d'une contrainte «exact-1»). Il s'agit du rôle fils.
- l'autre rôle soit de cardinalité 1-1, 0-i,1-i ou 0-1 (avec $1 < i \leq N$). Il s'agit du rôle père.

Pour qu'un type d'associations du schéma conceptuel puisse être représenté par une relation XML de référence, il est nécessaire que :

- l'un des deux rôles soit de cardinalité 1-1, ce rôle est joué par le type d'entités qui possédera le groupe référentiel (l'origine de la référence).
- l'autre rôle soit de cardinalité 1-1, 0-i, 1-i ou 0-1(avec $1 < i \leq N$), ce rôle est joué par le type d'entités qui sera référencé par le groupe référentiel (la destination de la référence).

ou bien

- l'un des deux rôles soit de cardinalité 0-1, ce rôle est joué par le type d'entités qui possédera le groupe référentiel (l'origine de la référence).
- l'autre rôle soit de cardinalité 1-1 ou 0-i (avec $1 < i \leq N$), ce rôle est joué par le type d'entités qui sera référencé par le groupe référentiel (la destination de la référence).

Fig.3.13 illustre les 4 types d'associations pouvant être transformés indifféremment en relation hiérarchique XML ou en relation de référence XML. Fig.3.14 illustre le type d'associations devant obligatoirement être transformés en relation de référence XML.

En l'absence de groupes «exact-1», deux cas n'ont pas encore été évoqués, il s'agit des types d'associations représentés Fig.3.15. Ne possédant pas de cardinalité 1-1, ces types d'associations ne peuvent devenir hiérarchiques. Le type d'associations `R6` ne peut pas non plus devenir référentiel car si une des deux cardinalité 0-1 devient une clé étrangère, comment alors garantir que la cardinalité maximale opposée soit 1 ? Pour une raison similaire, le type d'associations `R7` ne peut devenir référentiel. En effet, si la cardinalité 0-1 devient une clé étrangère, la cardinalité minimale opposée (valant 1) ne peut être garantie.

On suggère donc de transformer ces types d'associations en types d'entités afin de faire apparaître des types d'associations de cardinalités [0-1,1-1] ou [1-N,1-1], compatibles avec le modèle cible.

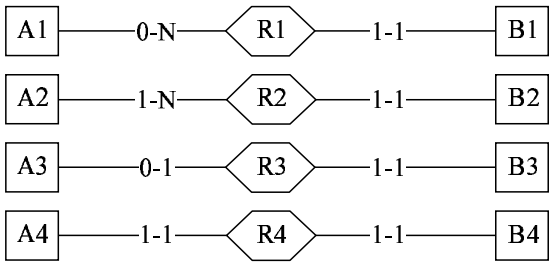

FIG. 3.13 – Les types d'associations hiérarchiques ou référentiels

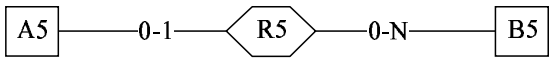

FIG. 3.14 – Un type d'associations référentiel

Une relation IS-A est transformée en types d'associations [0-1,1-1] entre le sur-type et le sous-type. Ces types d'associations binaires entre sur-types et sous-types possèdent cependant une caractéristique particulière : ils devront obligatoirement devenir une relation hiérarchique XML lors de l'étape suivante. Il est donc nécessaire de mémoriser, d'une manière quelconque, les types d'associations provenant de la transformation d'une relation IS-A. Lorsque la relation IS-A est accompagnée d'une contrainte de disjonction, de totalité ou de partition, la transformation en types d'associations entraîne respectivement la création d'un groupe «exclusive», «at-least-1» ou «exact-1», contenant les rôles joués par les sous-types. Nous verrons dans une étape ultérieure comment ces groupes seront traduits pour être conformes au modèle XML, sans perte de sémantique.

Le modèle XML n'accepte pas l'existence d'attributs multivalués. Si on désire représenter un certain nombre de fois une information d'un même type, cette information doit être contenue dans un élément XML auquel on associe un opérateur de cardinalité. C'est pour cette raison qu'une transformation en type d'entités par représentation des instances est appliquée à tout attribut multivalué.

Les attributs décomposés sont également exclus du modèle XML. Heureusement, par sa nature arborescente, le modèle XML possède un mécanisme efficace de modélisation d'attributs décomposés. Il est en effet aisé de reproduire la même information par une hiérarchie d'éléments XML. Tout attribut décomposé est donc également transformé en type(s) d'entités par représentation des instances. L'utilisateur aura le choix dans une étape ultérieure de représenter les attributs du modèle par des attributs ou par des éléments XML.

En résumé, le tableau ci-dessous montre comment les structures non-conformes vont être

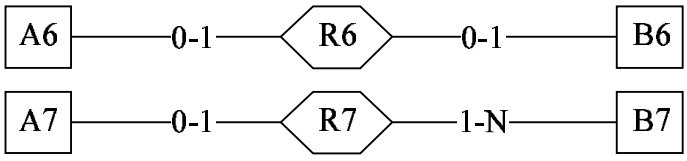

FIG. 3.15 – Les types d'associations à transformer en types d'entités

traitées et quelle transformation on propose de leur appliquer. Plus de détails sur ces transformations peuvent être trouvés dans [Hai00]

Structure non-conforme	*Transformée en*
relation IS-A simple	type d'associations 1-1
relation IS-A (disjointe)	type d'associations 1-1 + groupe «exclusive»
relation IS-A (totale)	type d'associations 1-1 + groupe «at-least-1»
relation IS-A (de partition)	type d'associations 1-1 + groupe «exact-1»
type d'associations complexe [4]	type d'entités
type d'associations [0-1,0-1]	type d'entités
type d'associations [0-1,1-N]	type d'entités
attribut multivalué	type d'entités (représentation des instances)
attribut composé	type d'entités (représentation des instances)

Résultat :

nous obtenons alors un schéma :
- sans relations IS-A.
- sans attributs multivalués.
- sans attributs composés.
- dont les types d'associations sont binaires.
- dont les types d'associations ne possèdent pas d'attributs.
- dont les cardinalités maximales des rôles d'un type d'associations sont 1-1 ou 1-N (exceptions aux cardinalités [0-1,0-1] et [0-1,1-N] qui sont interdites).

3.4.2 Étape 2 : Construction de la structure hiérarchique

Type :

interactif

Traitement :

En XML, la structure des données est typiquement hiérarchique : cela signifie que tout type d'éléments possède un et un seul type d'éléments père (mis à part la racine qui ne possède pas de père) et éventuellement un ou plusieurs types d'éléments fils.

Lors de la première étape du plan de transformation, les types d'associations du schéma ont été préparés pour figurer dans une arborescence XML : ils sont binaires, de cardinalités maximales 1-1 ou 1-N et ne possèdent pas d'attributs. Malheureusement, cela ne suffit pas pour former une structure hiérarchique comme celle imposée par le modèle XML : dans le schéma actuel, certains types d'entités peuvent être fils de plusieurs pères et rien ne laisse supposer qu'un seul type d'entités ne possède les propriétés pour devenir une racine.

Le principe sous-jacent de l'étape est trivial : il s'agit de décider de la nature des types d'associations du schéma : soit un type d'associations est hiérarchique, soit il est référentiel. Pour

[4] nous entendons par type d'associations complexe, un type d'associations ayant plus de 2 rôles, un type d'associations binaire N à N ou un type d'associations possédant un ou plusieurs attributs

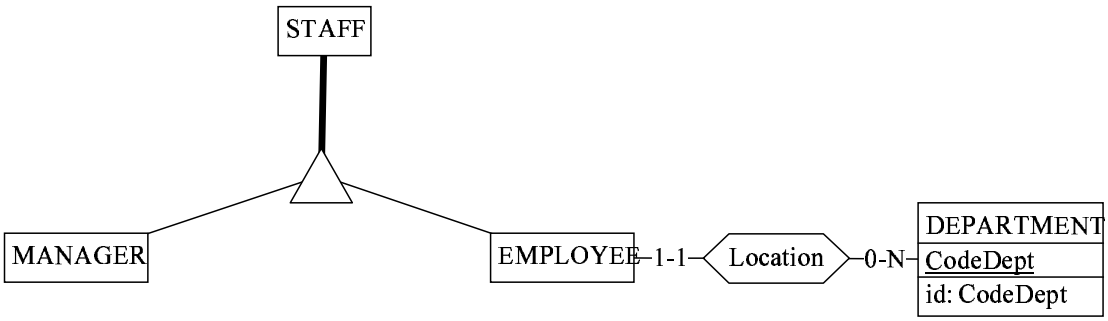

FIG. 3.16 – Le schéma avant la transformation de la relation IS-A

un type d'associations de cardinalités [0-1,0-N], le choix n'est pas possible : la transformation en relation de référence (facultative) est imposée.

L'objectif est de créer une structure hiérarchique où chaque type d'entités ne possède qu'un seul père et où un seul type d'entités ne soit la racine.

Plusieurs stratégies peuvent être envisagées pour résoudre ce problème délicat de design. Des solutions entièrement automatiques pourraient se révéler très pratiques sur des schémas assez conséquents ou lorsque le design du schéma final n'a que très peu d'importance [5].

Destinée à résoudre des problèmes tenant avant tout de la sémantique, la méthode proposée se veut semi-automatique : des informations évidentes peuvent être déduites automatiquement, alors que certains conflits sémantiques plus délicats requièrent l'intervention de l'utilisateur.

Le traitement des types d'associations issus des relations IS-A Le modèle XML, de par sa nature arborescente, se prête assez bien à la modélisation des relations hiérarchiques existantes entre un sur-type d'entités et ses différents sous-types. Il paraît en effet assez naturel que les types d'associations provenant des relations IS-A [6] deviennent des relations hiérarchiques : le sur-type d'entités jouant le rôle de père et les sous-types devenant les fils. Le père des sous-types étant connu, on peut alors également facilement déduire que les autres pères potentiels de ces sous-types ne pourront jouer le rôle de père dans la relation. Les types d'associations concernés deviennent alors des types d'associations référentiels. Cette transformation s'applique sur toutes les relations IS-A, quelque soit la contrainte (Partition, Totalité ou Disjonction) associée aux sous-types. Dans l'exemple présenté Fig.3.17, les types d'associations `IS-A Manager` et `IS-A Employee` sont issus de la transformation d'une relation IS-A (Fig.3.16). Selon la règle susmentionnée, ces types d'associations sont donc obligatoirement de nature hiérarchique : le type d'entités `STAFF` étant le père des types d'entités `MANAGER` et `EMPLOYEE`. Le conflit existant pour le choix du père du type d'entités `EMPLOYEE` est donc automatiquement résolu ; le type d'associations `Location` devient alors un type d'associations référentiel. Fig.3.17 illustre la structure hiérarchique finale déduite de la relation IS-A initiale.

Le traitement des types d'associations [1-1,1-1] Un rôle de cardinalité 1-1 possède une caractéristique particulière et intéressante : ce rôle peut en effet indifféremment être de type père ou de type fils. Un type d'associations binaire dont les deux rôles sont de cette cardinalité peut alors devenir une relation hiérarchique (ou référentielle) dans un sens (ou dans l'autre).

[5] Cela peut être le cas pour les documents XML destinés à être exploités automatiquement par des applications

[6] et qui ont été mémorisés lors de la phase précédente

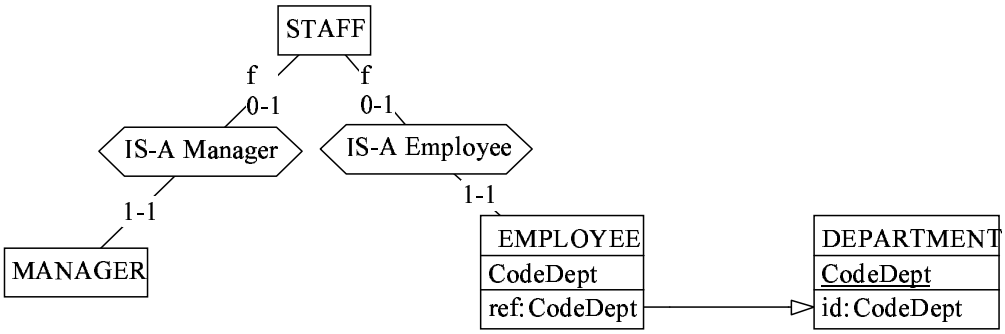

FIG. 3.17 – Transformation des types d'associations issus de la relation IS-A : `IS-A Manager` et `IS-A Employee` deviennent obligatoirement des relations hiérarchiques. Le type d'associations `Location` devient automatiquement une relation de référence

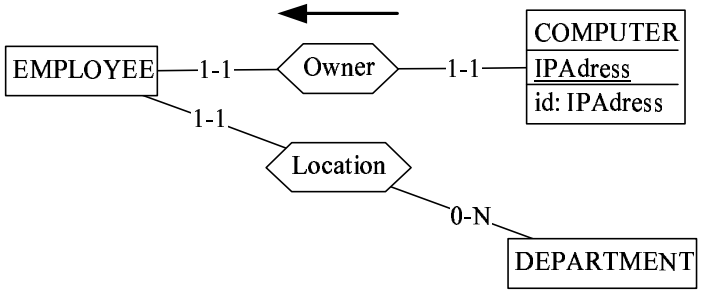

FIG. 3.18 – Un sens a été décidé pour le type d'associations `Owner` de cardinalités [1-1,1-1] – un conflit se crée car le type d'entités `EMPLOYEE` possède deux pères potentiels

De tels types d'associations peuvent notamment apparaître dans le schéma suite à la transformation par représentation des instances – dans la première étape – des attributs composés. Ce problème est considéré comme insoluble automatiquement et la première tâche de l'utilisateur dans ce plan de transformation sera de décider du sens des types d'associations de cardinalité [1-1,1-1]. Signalons que lors de cette phase de décision, il est demandé à l'utilisateur de décider le **sens** de la relation (de A vers B ou de B vers A) et non pas sa **nature** (hiérarchique ou référentielle). Les types d'associations concernés dans ce traitement – et pour lesquels un sens est attribué – ne sont pas forcément destinés à devenir des relations hiérarchiques. L'exemple présenté Fig.3.18 illustre le cas où un type d'associations de cardinalités [1-1,1-1] devient une relation de référence après qu'un sens lui ait été attribué : le type d'associations `Owner` est de cardinalités [1-1,1-1], un sens doit donc lui être attribué : du type d'entités `EMPLOYEE` vers le type d'entités `COMPUTER` ou du type d'entités `COMPUTER` vers le type d'entités `EMPLOYEE`. En choisissant cette deuxième possibilité, un conflit se crée : le type d'entités `EMPLOYEE` possède deux pères potentiels : le type d'entités `COMPUTER` ou le type d'entités `DEPARTMENT`. C'est à l'utilisateur de résoudre ce conflit (voir le paragraphe «Choix des pères : résolution des conflits») : si celui-ci décide que c'est le type d'entités `DEPARTMENT` qui doit devenir le père du type d'entités `EMPLOYEE`, alors le type d'associations `Location` devient hiérarchique et le type d'associations `Owner` sera de type référentiel. Cette situation est illustré Fig.3.19.

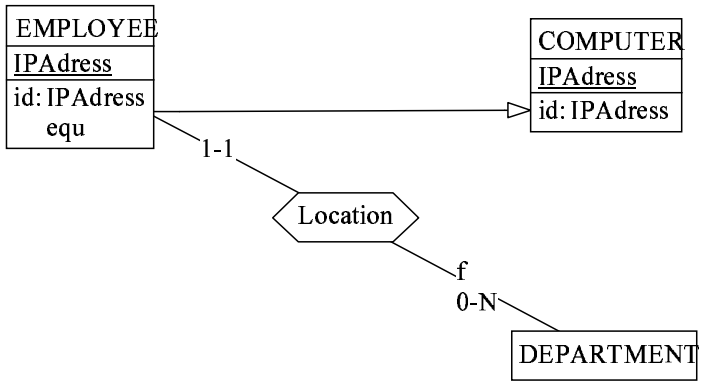

FIG. 3.19 – Le type d'entités `DEPARTMENT` a été élu comme père du type d'entités `EMPLOYEE`, le type d'associations `Owner` devient une relation référentielle

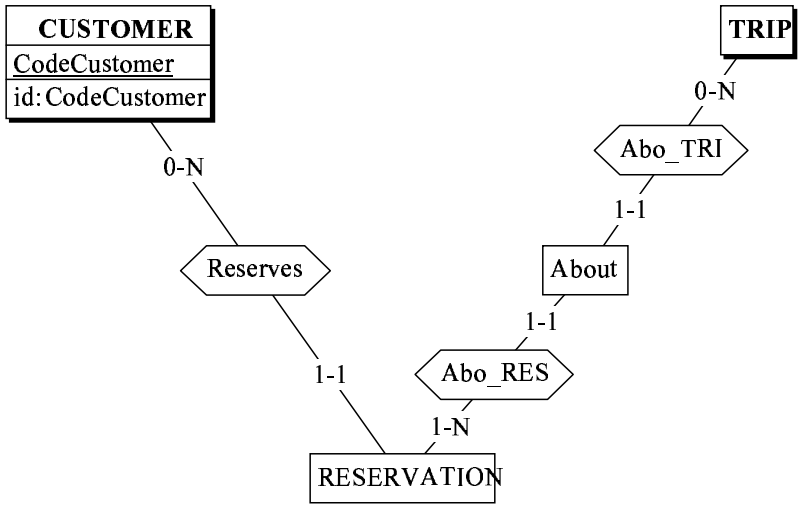

FIG. 3.20 – Les types d'entités `CUSTOMER` et `TRIP` sont des racines obligatoires

La mise en évidence des racines obligatoires Un type d'entités qui ne joue aucun rôle de cardinalités 1-1 et dont les rôles 0-1 ne font pas partie d'un groupe «exact-1» ne peut devenir fils d'aucun autre type d'entités. Ce type d'entités est, par conséquent, une racine obligatoire du schéma. Ces types d'entités représentent bien souvent des concepts majeurs de la situation modélisée et méritent donc bien d'être situés en haut de la hiérarchie. Dans l'exemple présenté Fig.3.20, les types d'entités `CUSTOMER` et `TRIP` ne jouent aucun rôle de cardinalité 1-1 : ils obtiennent donc le statut de racine.

La mise en évidence de racines facultatives Parmis les types d'entités qui n'ont pas été désignés comme racine, il peut subsister des concepts importants auxquels l'utilisateur souhaiterait donner le statut de racine. Dans ce plan de transformation, l'utilisateur a donc la possibilité de choisir ces types d'entités, appelés «racines facultatives». Pour des raisons de facilité, tous les rôles de cardinalité 1-1 joués par ces types d'entités sont automatiquement transformés en attributs référentiels. Dans l'exemple précédent présenté Fig.3.20, les types d'entités `CUSTOMER` et `TRIP` avaient été désignés racines. L'utilisateur peut souhaiter donner

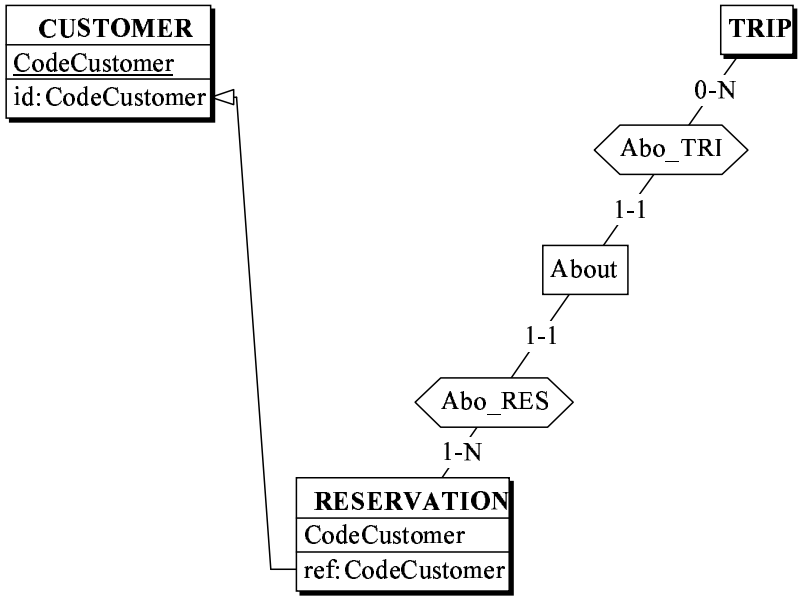

FIG. 3.21 – Le type d'entités `RESERVATION` obtient le statut de racine facultative – le type d'associations `Reserves` devient une relation de référence.

le même statut au concept de `RESERVATION`. La situation finale est illustrée Fig.3.21 : nous remarquons que le type d'associations `Reserves` devient dans ce cas une relation de référence. Ajoutons finalement qu'il est tout à fait envisageable de vouloir donner le statut de racine à tous les types d'entités du schéma : on se retrouve alors dans une situation de XML «plat», dans laquelle la structuration hiérarchique inhérente aux documents XML n'est pas exploitée. Le modèle ainsi défini est un modèle assez semblable au modèle relationnel dans lequel des tables «plates» (en première forme normale) sont liées par des clés étrangères.

Ajout d'une racine unique Suite à ces deux précédentes étapes concernant le choix des racines, il est évidemment possible que plusieurs types d'entités obtiennent le statut de racine (obligatoire ou facultative). Dans ce cas, assez fréquent, une racine (dite «technique») représentant l'ensemble du schéma est ajoutée sous la forme d'un type d'entités technique. Ce type d'entités supplémentaire – dont le nom est, par défaut, le nom du schéma de travail – devient le père des racines (dites «naturelles») précédemment trouvées. Dans l'exemple précédent présenté Fig.3.21, trois types d'entités ont été désignés comme racines naturelles. Afin d'assurer la conformité au modèle XML qui impose l'unicité de la racine, une racine technique, nommée `TRAVEL_AGENCY`, est automatiquement ajoutée au schéma. Ce type d'entités, situé au premier niveau de la hiérarchie, joue le rôle de père par rapport aux racines naturelles, les types d'entités `CUSTOMER`, `RESERVATION` et `TRIP`, situées au second niveau hiérarchique. Cette situation est illustrée Fig.3.22.

Choix des pères : résolution des conflits Un conflit apparaît lorsqu'un type d'entités possède plusieurs pères potentiels. Il s'agit alors de résoudre ce conflit en choisissant un et un seul père par type d'entités. Ce genre de conflit est principalement d'ordre sémantique et la manière dont ces conflits vont être résolus va directement influencer le design du schéma final. Envisager une solution automatique à ce type de problème n'a d'intérêt que pour des schémas conséquents. Lorsque le schéma est de taille respectable, nous préférons laisser l'utilisateur résoudre ces conflits un par un afin qu'il puisse décider et construire la hiérarchie qui lui

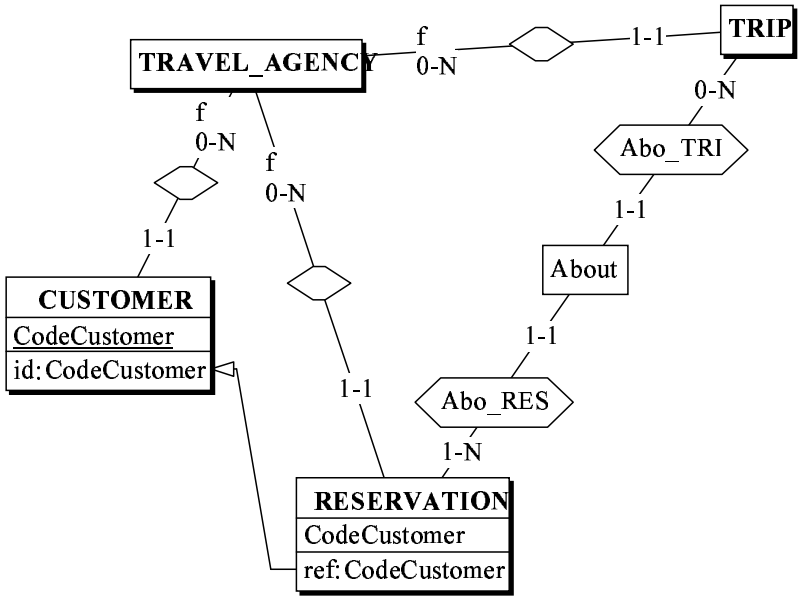

FIG. 3.22 – Une racine technique nommée `TRAVEL_AGENCY` est ajoutée au schéma afin de garantir l'unicité de la racine

semble la plus naturelle. Lorsqu'un père a été sélectionné parmi les pères potentiels d'un type d'entités fils, le type d'associations liant le type d'entités fils au type d'entités père devient par défaut hiérarchique. Les autres types d'associations liant ce même fils aux autres pères non-sélectionnés sont alors de type référentiels. Fig.3.23 et Fig.3.24 illustre la résolution d'un conflit. Le type d'entités `DETAIL` possède trois pères potentiels : les types d'entités `ORDER`, `PRODUCT` et `PROVIDER` (Fig.3.23). L'utilisateur juge que le concept `DETAIL` appartient au concept `ORDER` et nomme donc le type d'entités `ORDER` comme père du type d'entités `DETAIL`. Le type d'associations `Composition` fait partie de la hiérarchie et les deux types d'associations `About` et `Supply` deviennent des relations de référence (Fig.3.24) : le conflit est résolu.

Notons que certains conflits peuvent toutefois déjà avoir été résolus automatiquement dans une étape préalable. En effet, le traitement des types d'associations issus des relations IS-A et l'élection des racines (obligatoires et facultatives) peuvent de manière indirecte déterminer la nature de certains types d'associations et ainsi résoudre quelques conflits. Revenons à l'exemple présenté Fig.3.16 : le type d'entités `EMPLOYEE` possède deux pères potentiels (`STAFF` et `DEPARTMENT`) mais le type d'associations `IS-A Employee` entre les types d'entités `EMPLOYEE` et `STAFF` provenant d'une relation IS-A, c'est obligatoirement `STAFF` qui est choisi comme père d'`EMPLOYEE`. La Fig.3.17 suivante nous montre que le conflit est résolu automatiquement.

Ajout d'identifiants techniques Une relation de référence fonctionne de la même manière que le mécanisme des clés étrangères dans le modèle relationnel [7] : le type d'entités qui joue le rôle de cardinalité 1-1 (ou 0-1) possède un attribut de référence dont la valeur identifie une occurence du type d'entités opposé [8]. Avant de pouvoir matérialiser physiquement les relations de référence, il est par conséquent indispensable que ces types d'entités référencés possèdent un attribut identifiant non-composé. Le traitement proposé est donc d'ajouter un

[7] pour plus d'informations sur le modèle relationnel et le mécanisme des clés étrangères, veuillez vous référer à [Hai00]

[8] Dans une DTD, ce mécanisme de référence est rendu possible grâce à l'utilisation d'attributs de référence (de type IDREF) et identifiants (de type ID)

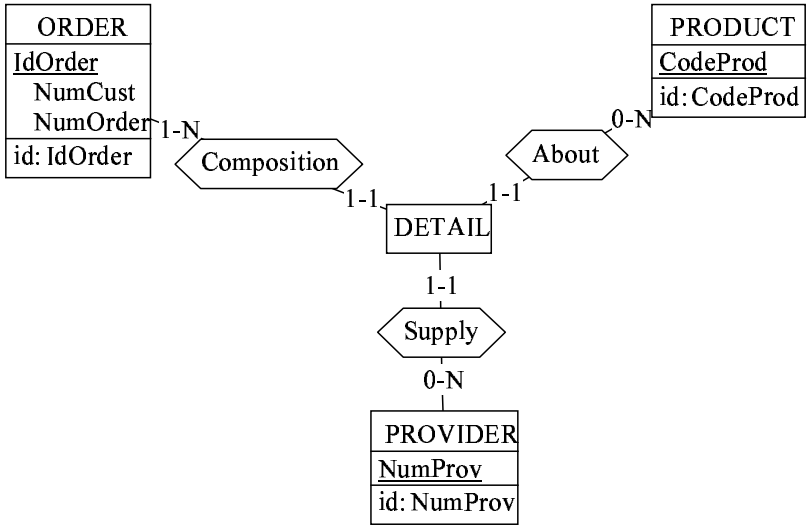

FIG. 3.23 – L'utilisateur doit choisir qui sera le père du type d'entités `DETAIL`

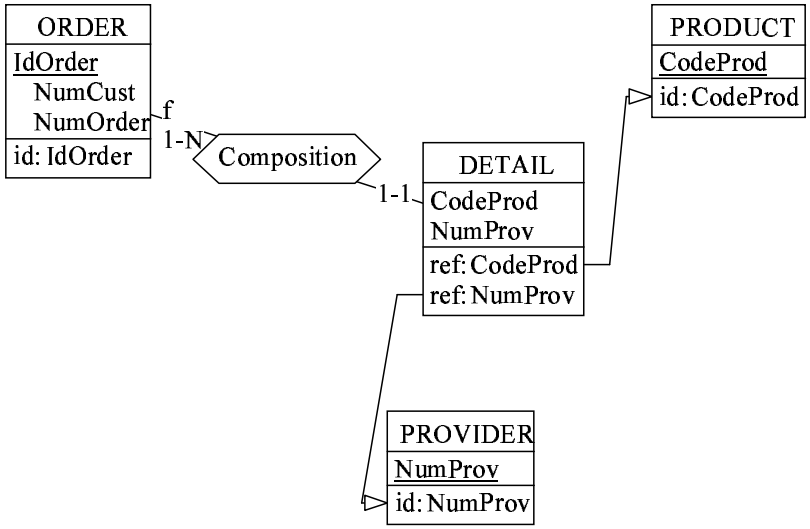

FIG. 3.24 – Le type d'entités `ORDER` est choisi, les types d'associations `About` et `Supply` deviennent des relations de référence

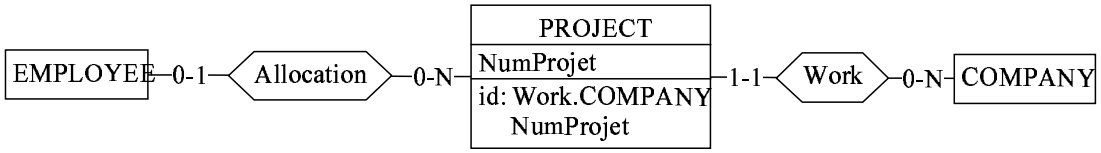

FIG. 3.25 – Le type d'entités `PROJECT` ne possède pas d'identifiant primaire simple – il n'est pas possible de matérialiser la relation de référence `Allocation`

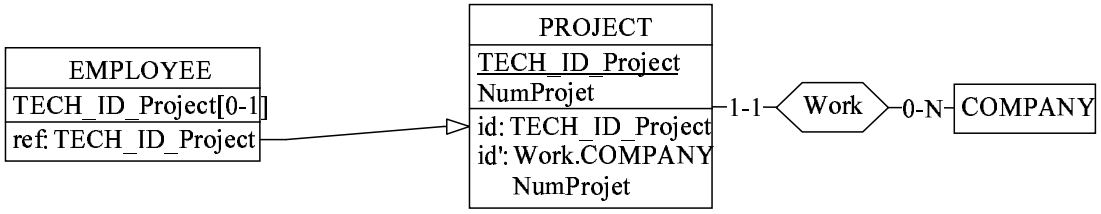

FIG. 3.26 – L'ajout d'un identifiant technique primaire et simple permet la matérialisation de la relation de référence

identifiant technique (primaire et non composé) aux types d'entités référencés, ne possédant pas cette propriété nécessaire à la relation. L'exemple Fig.3.25 montre un type d'associations `Allocation` dont les cardinalités sont [0-1,0-N]. Selon la règle vue précédemment, ce type d'associations doit être transformé en relation de référence. Dans l'état actuel du schéma, cette relation de référence est impossible à matérialiser car le type d'entités référencé `PROJECT` ne possède pas d'identifiant primaire non-composé. Un identifiant technique `TECH_ID_Project` est alors ajouté au type d'entités `PROJECT`. Grâce à cet attribut, il est possible de faire référence à une occurence de `PROJECT` en utilisant un attribut référentiel dans le type d'entités `EMPLOYEE` : la relation de référence peut donc être correctement matérialisée (Fig.3.26).

Résultat :

nous obtenons alors un schéma :

- sans relations IS-A.
- sans attributs multivalués.
- sans attributs composés.
- dont les types d'associations sont binaires.
- dont les types d'associations ne possèdent pas d'attributs.
- dont les cardinalités maximales des rôles d'un type d'associations sont 1-1 ou 1-N (exceptions aux cardinalités [0-1,0-1] et [0-1,1-N] qui sont interdites).
- dont chaque type d'entités joue un et un seul rôle fils (exception au type d'entité racine qui ne joue aucun rôle fils).
- dont chaque type d'entités joue éventuellement un ou plusieurs rôle(s) père(s) (ces rôles étant nommés «f»).
- dont un type d'entités est la racine.
- dont certaines relations peuvent être des références vers un type d'entités possédant un identifiant primaire simple.

3.4.3 Étape 3 : Modification des cardinalités

Type :

automatique

Traitement :

Les attributs multivalués ayant été transformés en types d'entités lors de la première étape, les cardinalités, dans l'état actuel du schéma, apparaissent uniquement au niveau des rôles joués par les types d'entités dans les types d'associations (binaires). Le modèle XML défini précédemment n'autorise toutefois pas la représentation de n'importe quelle cardinalité. Les cardinalités associées aux rôles fils sont de 1-1 (ou 0-1, lorsqu'un groupe exact-1 contient l'ensemble des rôles pères) alors que les cardinalités associées aux rôles pères (nommés «f», par convention) sont limitées aux cardinalités des opérateurs définis dans la norme des DTD : 1-1, 0-N, 0-1 et 1-N. Le modèle XML ne permet donc pas de définir des cardinalités aussi précises que 1-5, 3-7 ou 5-N [9]. La solution proposée pour faire face à cette limitation de la norme est d'élargir le champs des cardinalités non-conformes en appliquant les deux principes suivants :

- si la cardinalité minimale est strictement supérieure à 1, elle est ramenée à 1.
- si la cardinalité maximale est strictement supérieure à 1, on lui affecte la valeur N.

Avec cette démarche, les cardinalités 1-5, 3-7 et 5-N sont toutes trois transformées en 1-N.

Résultat :

nous obtenons alors un schéma :

- sans relations IS-A.
- sans attributs multivalués.
- sans attributs composés.
- dont les types d'associations sont binaires.
- dont les types d'associations ne possèdent pas d'attributs.
- dont les cardinalités maximales des rôles d'un type d'associations sont 1-1 ou 1-N (exceptions aux cardinalités [0-1,0-1] et [0-1,1-N] qui sont interdites).
- dont chaque type d'entités joue un et un seul rôle fils (exception au type d'entités racine qui ne joue aucun rôle fils).
- dont chaque type d'entités joue éventuellement un ou plusieurs rôle(s) père(s) (ces rôles étant nommés «f»).
- dont un type d'entités est la racine.
- dont certaines relations peuvent être des références vers un type d'entités possédant un identifiant primaire simple.
- dont les cardinalités des rôles pères sont 1-1, 0-N, 0-1 ou 1-N.

3.4.4 Étape 4 : Création des groupes «gid» et «idref»

Type :

automatique

Traitement :

Création des groupes «gid» Dans le modèle XML, il est possible de déclarer des groupes de type user-defined «gid». Ces groupes permettent d'identifier un élément **au sein d'un**

[9] [KW01] suggère toutefois un moyen de représentation de telles cardinalités, au prix d'une DTD plus lourde et moins lisible

document. La portée de cet identifiant est donc globale au document et non réduite à un type d'éléments [10]. Un groupe soumis à une contrainte «gid» a donc une portée plus large qu'un groupe soumis à une contrainte «id». C'est l'une des raisons pour lesquelles nous avons décidé de distinguer ces groupes, à priori assez semblables. Une autre raison de faire la distinction est qu'un groupe «gid» est beaucoup plus contraint qu'un groupe «id» : en effet, alors qu'un groupe «id» peut contenir plusieurs éléments de types différents (attributs ou rôles...) ; dans le modèle XML, un groupe «gid» ne peut être composé que d'un seul attribut. Malgré cet écart au niveau de la sémantique, nous suggérons de transformer les groupes «id» composés d'un seul attribut en groupes «gid». Les autres groupes «id» ne pouvant pas être représentés dans le modèle-cible devront disparaître dans une étape ultérieure.

La transformation vers un modèle moins riche entraîne inévitablement une perte de sémantique. Dans le cas de la transformation des groupes «id», la perte est plutôt sévère : les groupes identifiants simples, en devenant des groupes «gid», voient leur sémantique s'élargir en acquérant une portée globale alors que les autres groupes identifiants ne peuvent carrément pas être traduits dans le modèle-cible.

Ajoutons finalement qu'un type d'éléments XML ne peut être identifié que par un seul attribut. Si un type d'éléments possède plusieurs identifiants simples, c'est l'identifiant primaire qui sera choisi pour identifier le type d'éléments et donc appartenir au groupe «gid». Les autres identifiants (secondaires) disparaîtront purement et simplement.

Création des groupes «idref» Le modèle XML permet l'utilisation d'un mécanisme similaire aux clés étrangères par le biais de groupes «idref» composés d'un seul attribut et dont la valeur identifie un type d'éléments. Comme c'est le cas pour les attributs «gid», la portée d'un groupe «idref» est globale. La contrainte référentielle associée à ce groupe est donc beaucoup plus légère que cette même contrainte pour les groupes «ref» du modèle conceptuel : toute valeur prise par un attribut appartenant à un groupe «idref», doit être une valeur prise par un attribut appartenant à un groupe «gid», quelque soit le type d'entités auquel appartient le groupe. Malgré cette divergence, nous suggérons la transformation des groupes «ref» en groupes «idref».

Dans l'exemple présenté Fig.3.24, l'attribut `DETAIL.CodeProd` appartient à un groupe «ref» référençant l'attribut `PRODUCT.CodeProd` qui appartient à un groupe «id». La contrainte référentielle stipule que toute valeur de `DETAIL.CodeProd` doit être une valeur de `PRODUCT.CodeProd`. La transformation des groupes «id» en groupes «gid» et des groupes «ref» en groupes «idref» sous-entend une contrainte bien différente et beaucoup plus souple : toute valeur de l'attribut `DETAIL.CodeProd` doit être une valeur d'un attribut appartenant à un groupe «gid» du schéma. Fig.3.27 illustre une situation autorisée par cette contrainte : la valeur «P007» est bien une valeur d'un attribut appartenant à un groupe «gid» mais ce groupe n'appartient pas au type d'entités référencé par l'attribut `DETAIL.CodeProd`.

Résultat :

nous obtenons alors un schéma :
- sans relations IS-A.
- sans attributs multivalués.
- sans attributs composés.
- dont les types d'associations sont binaires.
- dont les types d'associations ne possèdent pas d'attributs.

[10] Grâce à l'utilisation de techniques simples de codage, il est toutefois possible d'implémenter un identifiant local à un type d'éléments : il suffit de préfixer chaque valeur d'identifiant par le nom du type d'éléments avant de l'encoder dans la base de données.

DETAIL.CodeProd
P005
P007
P008

PRODUCT.CodeProd
P005
P008

PROVIDER.NumProv
P007

FIG. 3.27 – Une situation autorisée par les contraintes référentielles XML, matérialisées par les attributs «gid» et «idref»

- dont les cardinalités maximales des rôles d'un type d'associations sont 1-1 ou 1-N (exceptions aux cardinalités [0-1,0-1] et [0-1,1-N] qui sont interdites).
- dont chaque type d'entités joue un et un seul rôle fils (exception au type d'entités racine qui ne joue aucun rôle fils).
- dont chaque type d'entités joue éventuellement un ou plusieurs rôle(s) père(s) (ces rôles étant nommés «f»).
- dont un type d'entités est la racine.
- dont certaines relations peuvent être des références vers un type d'entités possédant un identifiant primaire simple.
- dont les cardinalités des rôles pères sont 1-1, 0-N, 0-1 ou 1-N.
- pouvant contenir des groupes «gid», composés d'un et un seul attribut.
- pouvant contenir des groupes «idref», composés d'un et un seul attribut.
- dont les groupes «id» subsistants contiennent un rôle ou plusieurs attributs.
- dont les groupes «idref» subsistants contiennent un rôle ou plusieurs attributs.

3.4.5 Étape 5 : Transformation des groupes «exclusive», «at-least-1» et «exact-1» issus des relations IS-A

Type :

automatique

Traitement :

Lors de la première étape, les relations IS-A de disjonction, de totalité et de partition ont été transformées en types d'associations accompagnés respectivement d'un groupe «exclusive», «at-least-1» ou «exact-1» contenant les rôles joués par les types d'entités fils. La deuxième étape a imposé que les types d'associations issus des relations IS-A deviennent des relations hiérarchiques : le sur-type jouant le rôle de père et le(s) sous-type(s) étant le(s) fils. Il peut cependant encore subsister, dans l'état actuel du schéma, des groupes «exclusive», «at-least-1» ou «exact-1» au niveau du père. Grâce à l'utilisation des groupes «seq» et «choice» définis dans le modèle XML et à la création de types d'entités techniques, il est possible de représenter les contraintes de disjonction et de totalité associées à ces groupes. Pour être précis, le traitement proposé lors de cette étape s'applique à tous les groupes «exclusive», «at-least-1» et «exact-1» qui possèdent uniquement des rôles. Ce qui est évidemment le cas des groupes issus des relations IS-A.

Transformation des groupes «exact-1» La sémantique exprimée par un groupe «exact-1» et par un groupe «choice» est absolument identique : il s'agit du choix (obligatoire) d'un et un seul composant du groupe. On suggère donc simplement de renommer les groupes «exact-1»

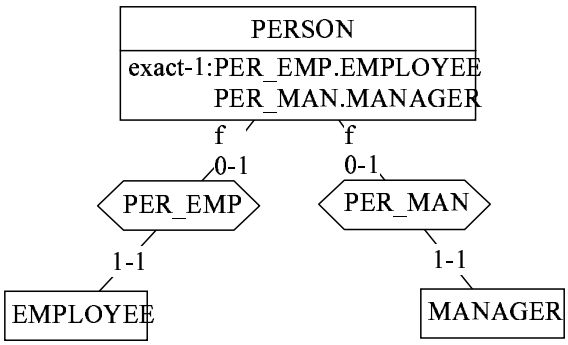

FIG. 3.28 – Un groupe «exact-1» contenant uniquement des rôles : avant transformation

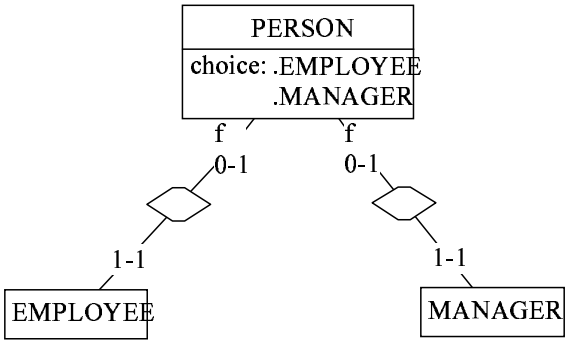

FIG. 3.29 – La transformation d'un groupe «exact-1» consiste simplement à renommer celui-ci en groupe «choice»

par le mot réservé «choice». Afin de rendre le choix obligatoire, la cardinalité des rôles joués par le sur-type passe de 0-1 à 1-1.

Fig.3.28 et Fig.3.29 illustrent la transformation d'un groupe «exact-1».

Transformation des groupes «exclusive» Un groupe «exclusive» exprime une alternative (groupe «choice») mais, contrairement au groupe «exact-1», celle-ci est facultative. Un type d'entités technique doit donc être créé pour représenter le groupe «choice» facultatif. Ce type d'entités intermédiaire devient le fils du sur-type d'entités et le père des sous-types. Les types d'associations originaux existant entre le sur-type et les sous-types étant évidemment supprimés. Le type d'entités technique s'intercalant ainsi entre le sur-type et les sous-type contient un groupe «choice» constitué des rôles fils joués par les sous-types. Pour représenter le caractère facultatif du choix, la cardinalité du rôle père joué par le sur-type sur le type d'entités technique vaut 0-1. La cardinalité des rôles pères joué par le type d'entités technique sur les sous-types est de 1-1. La cardinalité des rôles fils est évidemment de 1-1.

Fig.3.30 et Fig.3.31 illustrent la transformation d'un groupe «exclusive».

Transformation des groupes «at-least-1» La transformation des groupes «at-least-1» est moins triviale. Il s'agit de composer un groupe «choice» avec les multiples combinaisons des sous-types. Ce problème combinatoire devient rapidement complexe lorsque le nombre de sous-types est élevé. C'est le prix à payer pour conserver la semantique du groupe «at-least-1». Nous proposons, ici, une solution correcte pour deux sous-types.

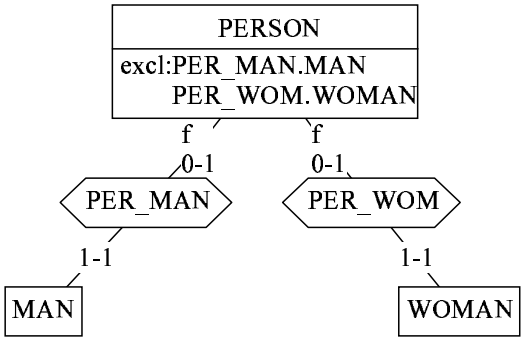

FIG. 3.30 – Un groupe «exclusive» contenant uniquement des rôles : avant transformation

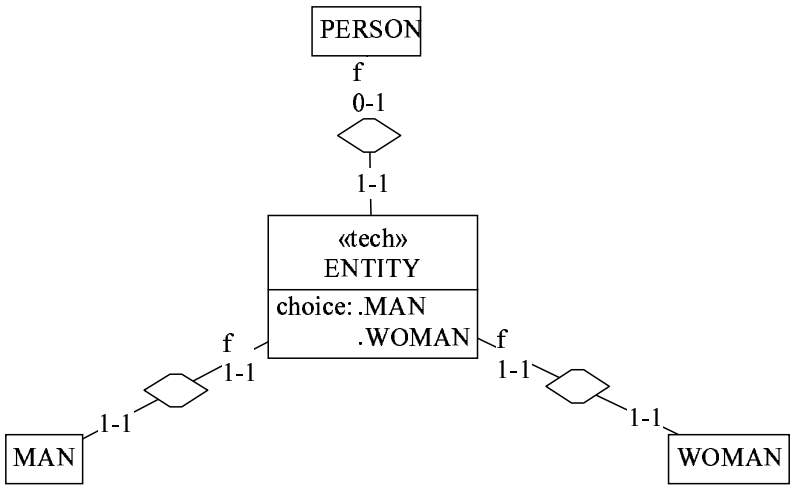

FIG. 3.31 – La transformation d'un groupe «exclusive» grâce à un type d'entités technique et un groupe «choice» – cardinalité du rôle père joué par le sur-type : 0-1

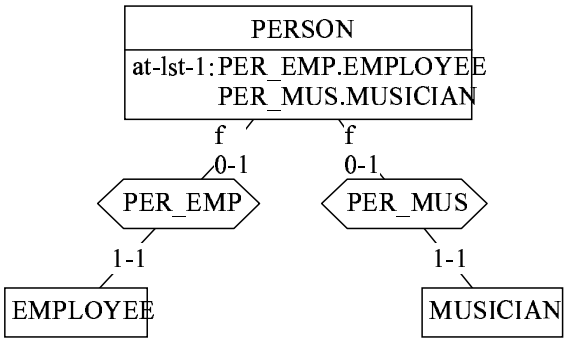

FIG. 3.32 – Un groupe «at-least-1» contenant uniquement des rôles : avant transformation

La sémantique associée au schéma présenté Fig.3.32 est la suivante : une personne est obligatoirement soit un employé, soit un musicien, soit les deux. Cette alternative se traduit, dans le schéma, par un groupe «choice» au niveau du sur-type. Ce groupe est composé des rôles joués par les fils du sur-type, qui sont au nombre de trois :

- le type d'entités `EMPLOYEE`
- le type d'entités `MUSICIAN`
- un type d'entités technique, nommé `ENTITY`, dont les fils, `EMPLOYEE` et `MUSICIAN`, forment une séquence.

Fig.3.33 illustre cette transformation.

Résultat :

nous obtenons alors un schéma :

- sans relations IS-A.
- sans attributs multivalués.
- sans attributs composés.
- dont les types d'associations sont binaires.
- dont les types d'associations ne possèdent pas d'attributs.
- dont les cardinalités maximales des rôles d'un types d'association sont 1-1 ou 1-N (exceptions aux cardinalités [0-1,0-1] et [0-1,1-N] qui sont interdites).
- dont chaque types d'entité joue un et un seul rôle fils (exception au types d'entité racine qui ne joue aucun rôle fils).
- dont chaque types d'entité joue éventuellement un ou plusieurs rôle(s) père(s) (ces rôles étant nommés «f»).
- dont un types d'entité est la racine.
- dont certaines relations peuvent être des références vers un types d'entité possédant un identifiant primaire simple.
- dont les cardinalités des rôles pères sont 1-1, 0-N, 0-1 ou 1-N.
- pouvant contenir des groupes «gid», composés d'un et un seul attribut.
- pouvant contenir des groupes «idref», composés d'un et un seul attribut.
- dont les groupes «id» subsistants contiennent un rôle ou plusieurs attributs.
- dont les groupes «idref» subsistants contiennent un rôle ou plusieurs attributs.
- sans groupes «exclusive» et «at-least-1».

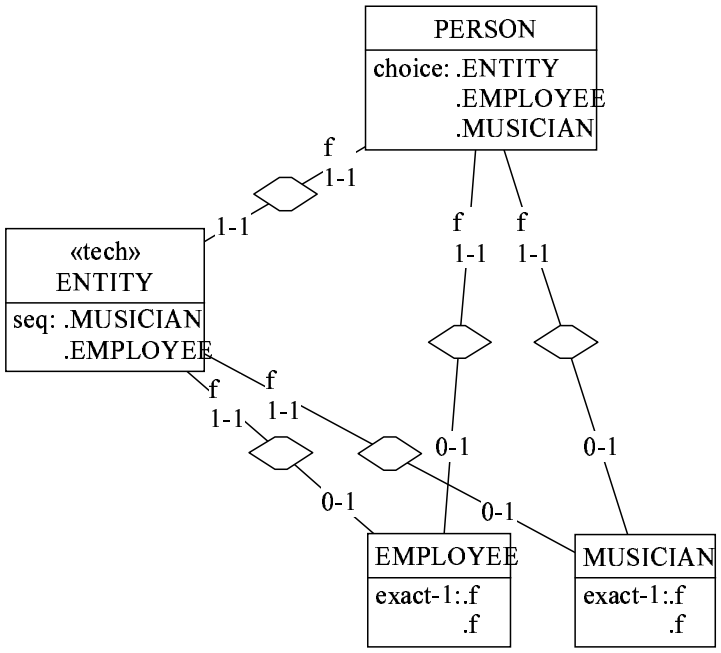

FIG. 3.33 – La transformation d'un groupe «at-least-1» avec deux sous-types

3.4.6 Étape 6 : Traitement des attributs non-impliqués dans un groupe «gid» ou «idref»

Type :

interactif

Traitement :

Lors de la première étape, les attributs composés et multi-valués ont été transformés en types d'entités. Il ne reste donc, dans le schéma actuel, que des attributs simples et mono-valués. Les attributs appartenant à un groupe «id» ou un groupe «ref» ont été traités dans la quatrième étape : ils gardent le statut d'attribut mais appartiennent respectivement à un groupe «gid» ou «idref». Dans le modèle XML, deux mécanismes sont envisageables pour représenter les attributs non-impliqués dans de tels groupes :

- soit ils deviennent des attributs XML de type CDATA, associés au type d'éléments auquel ils appartiennent.
- soit ils prennent le statut d'un type d'entités dont le contenu est de type textuel pur #PCDATA.

La première alternative nécessite simplement de changer le type de l'attribut afin qu'il soit du type user-defined nommé «cdata». La seconde alternative implique de transformer l'attribut en type d'entités (par représentation des instances), de nommer «f» le rôle joué par le type d'entités opposé (afin d'exprimer la nature hiérarchique de la relation) et enfin, de renommer l'attribut par le mot réservé «#pcdata». Fig.3.34 montre la transformation en types d'entités des attributs `Name`, `City` et `Age`, non-impliqués dans le groupe «gid».

L'utilisateur peut choisir l'une des deux transformations mais la transformation choisie s'applique à **tous les attributs** du schéma. Une discussion sur le choix des modes de représentation des attributs peut être trouvée dans [Gol01] et [KW01].

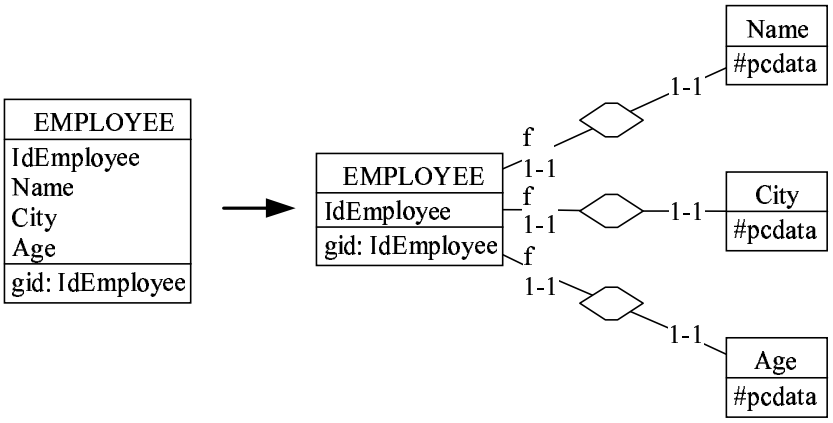

FIG. 3.34 – La transformation des attributs en types d'entités, par représentation des instances

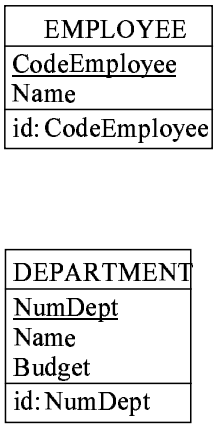

FIG. 3.35 – Un schéma **avant** transformation des attributs en éléments XML : l'attribut `Name` est présent dans les types d'entités `EMPLOYEE` et `DEPARTMENT`

Lorsque plusieurs attributs appartenant à des types d'entités différents possèdent le même nom, l'utilisateur doit également choisir parmis deux modes de représentation possibles. Les règles sont les suivantes :

- **Fusion des attributs de même nom :** si la sémantique est absolument identique pour tous les attributs, ceux-ci sont représentés par un seul type d'entités portant le nom des attributs. Ce type d'entités possède un groupe «exact-1> composé des rôles joués par ses pères.
- **Distinction des attributs de même nom :** si la sémantique associée à ces attributs est différente, chaque attribut est représenté par un type d'entités dont le nom est le nom de l'attribut, préfixé par les trois premières lettres du type d'entités auquel appartenait l'attribut.

A partir du même schéma de départ (Fig.3.35), les Fig.3.36 et 3.37 illustrent respectivement les règles de **Fusion** et de **Distinction**.

Pour des raisons de simplicité, la règle choisie s'applique à tous les attributs de même nom dans le schéma.

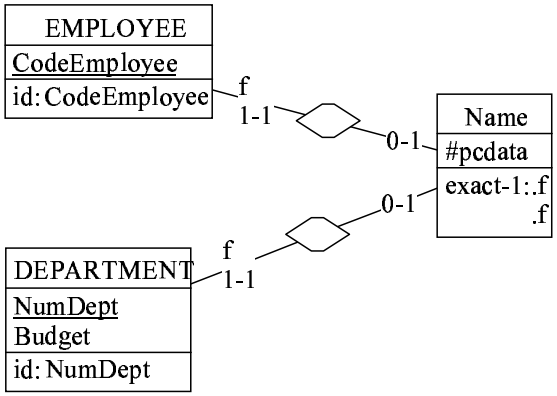

FIG. 3.36 – Le schéma **après** transformation des attributs en éléments XML : Fusion des attributs de même nom en un type d'entités

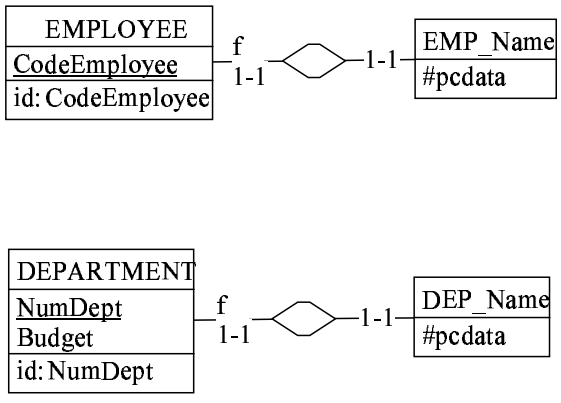

FIG. 3.37 – Le schéma **après** transformation des attributs en éléments XML : Distinction des attributs de même nom

Résultat :

nous obtenons alors un schéma :
- sans relations IS-A.
- sans attributs multivalués.
- sans attributs composés.
- dont les types d'associations sont binaires.
- dont les types d'associations ne possèdent pas d'attributs.
- dont les cardinalités maximales des rôles d'un type d'associations sont 1-1 ou 1-N (exceptions aux cardinalités [0-1,0-1] et [0-1,1-N] qui sont interdites).
- dont chaque type d'entités joue un et un seul rôle fils (exception au type d'entités racine qui ne joue aucun rôle fils).
- dont chaque type d'entités joue éventuellement un ou plusieurs rôle(s) père(s) (ces rôles étant nommés «f»).
- dont un type d'entités est la racine.
- dont certaines relations peuvent être des références vers un type d'entités possédant un identifiant primaire simple.
- dont les cardinalités des rôles pères sont 1-1, 0-N, 0-1 ou 1-N.
- pouvant contenir des groupes «gid», composés d'un et un seul attribut.
- pouvant contenir des groupes «idref», composés d'un et un seul attribut.
- dont les groupes «id» subsistants contiennent un rôle ou plusieurs attributs.
- dont les groupes «idref» subsistants contiennent un rôle ou plusieurs attributs.
- sans groupes «exclusive» et «at-least-1».
- dont tous les attributs sont soit impliqués dans un groupe «gid»/«idref», soit de type «cdata», soit nommés «#pcdata».

3.4.7 Étape 7 : Création des groupes «seq»

Type :

automatique

Traitement :

Lorsqu'un type d'entités père possède plusieurs types d'entités fils dont les rôles n'appartiennent pas à un groupe «choice», le modèle exige qu'un ordre (séquence) soit établi entre ces fils. Dans le modèle XML, cet ordre est matérialisé par un groupe «seq», constitué des rôles joués par les fils. Dans cette étape, ces groupes sont créés dans chaque type d'entités possédant plusieurs fils. L'ordre des rôles fils au sein d'un groupe est déterminé de manière aléatoire. Suite à l'étape 5, certains types d'entités, issus des relations IS-A, peuvent déjà posséder un groupe «seq» ou «choice» ; ces groupes ne sont évidemment pas concernés par ce traitement. Dans l'exemple présenté Fig.3.38, le type d'entités `DEPARTMENT` possède deux types d'entités fils : `EMPLOYEE` et `Budget`. Un groupe «seq», constitué des rôles des fils est alors ajouté au type d'entités `DEPARTMENT`.

Résultat :

nous obtenons alors un schéma :
- sans relations IS-A.
- sans attributs multivalués.
- sans attributs composés.
- dont les types d'associations sont binaires.

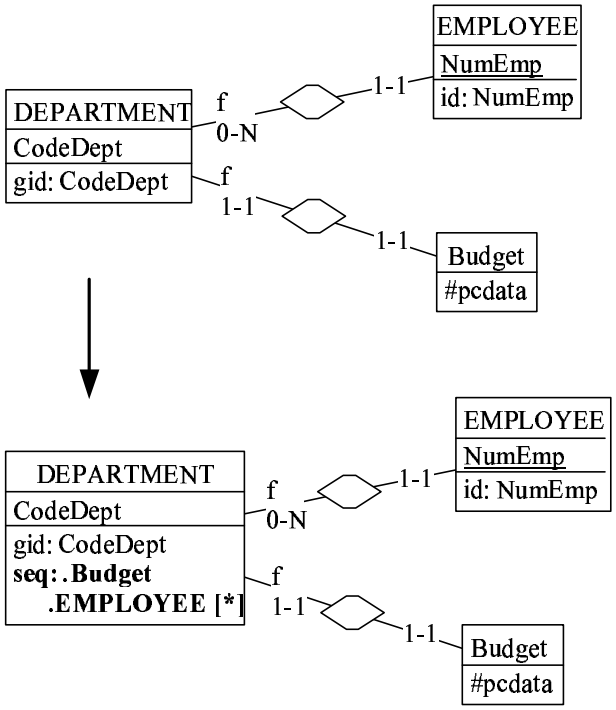

FIG. 3.38 – Un groupe «seq» est ajouté aux types d'entités possédant plusieurs fils

- dont les types d'associations ne possèdent pas d'attributs.
- dont les cardinalités maximales des rôles d'un type d'associations sont 1-1 ou 1-N (exceptions aux cardinalités [0-1,0-1] et [0-1,1-N] qui sont interdites).
- dont chaque type d'entités joue un et un seul rôle fils (exception au type d'entités racine qui ne joue aucun rôle fils).
- dont chaque type d'entités joue éventuellement un ou plusieurs rôle(s) père(s) (ces rôles étant nommés «f»).
- dont un type d'entités est la racine.
- dont certaines relations peuvent être des références vers un type d'entités possédant un identifiant primaire simple.
- dont les cardinalités des rôles pères sont 1-1, 0-N, 0-1 ou 1-N.
- pouvant contenir des groupes «gid», composés d'un et un seul attribut.
- pouvant contenir des groupes «idref», composés d'un et un seul attribut.
- dont les groupes «id» subsistants contiennent un rôle ou plusieurs attributs.
- dont les groupes «idref» subsistants contiennent un rôle ou plusieurs attributs.
- sans groupes «exclusive» et «at-least-1».
- dont tous les attributs sont soit impliqués dans un groupe «gid»/«idref», soit de type «cdata», soit nommés «#pcdata».
- dont tous les types d'entités possédant plusieurs fils possèdent un groupe «seq» ou «choice», constitué des rôles joués par les fils.

3.4.8 Étape 8 : Suppression des groupes non-conformes

Type :

automatique

Traitement :

Dans le modèle XML, seuls les groupes de type «gid», «idref», «seq», «choice» ou «exact-1» [11] sont autorisés. Ces groupes ont été créés lors des étapes 4, 5, 6 et 7. Il peut cependant encore subsister dans le schéma des groupes différents, par conséquent non-conformes au modèle XML : des groupes «id» constitués de plusieurs composants, des groupes «id'», des groupes «coex», voire des groupes soumis à des contraintes définies par l'utilisateur. Le traitement proposé à ces groupes non-conformes est assez radical : en effet, on propose simplement de les ignorer et de ne pas les représenter dans le modèle XML. La pauvreté sémantique du modèle XML nous laisse très peu de choix ; cette solution a le mérite d'être simple mais peut, dans certains cas, dégrader fortement la sémantique originale du schéma.

Résultat :

nous obtenons alors un schéma :

- sans relations IS-A.
- sans attributs multivalués.
- sans attributs composés.
- dont les types d'associations sont binaires.
- dont les types d'associations ne possèdent pas d'attributs.
- dont les cardinalités maximales des rôles d'un type d'associations sont 1-1 ou 1-N (exceptions aux cardinalités [0-1,0-1] et [0-1,1-N] qui sont interdites).
- dont chaque type d'entités joue un et un seul rôle fils (exception au type d'entités racine qui ne joue aucun rôle fils).
- dont chaque type d'entités joue éventuellement un ou plusieurs rôle(s) père(s) (ces rôles étant nommés «f»).
- dont un type d'entités est la racine.
- dont certaines relations peuvent être des références vers un type d'entités possédant un identifiant primaire simple.
- dont les cardinalités des rôles pères sont 1-1, 0-N, 0-1 ou 1-N.
- pouvant contenir des groupes «gid», composés d'un et un seul attribut.
- pouvant contenir des groupes «idref», composés d'un et un seul attribut.
- dont les groupes «id» subsistants contiennent un rôle ou plusieurs attributs.
- dont les groupes «idref» subsistants contiennent un rôle ou plusieurs attributs.
- sans groupes «exclusive» et «at-least-1».
- dont tous les attributs sont soit impliqués dans un groupe «gid»/«idref», soit de type «cdata», soit nommés «#pcdata».
- dont tous les types d'entités possédant plusieurs fils possèdent un groupe «seq» ou «choice», constitué des rôles joués par les fils.
- dont tous les groupes sont de type «gid», «idref», «seq», «choice» ou «exact-1».

A l'issue de ces huit étapes, le schéma obtenu répond aux contraintes imposées aux différentes structures par le modèle XML. Ce travail de transformation se veut être le plus automatique possible. L'objectif est atteint : sur les huit étapes qui composent ce plan, six sont entièrement automatiques. Malgré cette automatisation maximale, ce plan est particulièrement recommandé pour la transformation de schémas de taille raisonnable. En effet, lors de la délicate étape de construction de la hiérarchie, l'utilisateur est amené à faire des choix très précis et à résoudre les conflits un par un, tâche qui s'annonce plutôt laborieuse lorsque le nombre d'objets dans le schéma dépasse les 1000 unités.

[11] uniquement dans le cas d'un fils possédant plusieurs pères

3.5 Outil de support à la conception logique

3.5.1 Cadre de travail

Dans cette section, nous décrivons l'outil de transformation de schéma développé pour l'outil CASE DB-Main. Cet outil implémente la méthode de transformation décrite dans la section précédente : un schéma conceptuel créé dans DB-Main peut ainsi être transformé en un schéma logique conforme au modèle XML.

Pour réaliser cet outil, nous nous sommes basés sur le code source des modules développés par Christine Delcroix dans le cadre de son travail de recherche sur l'ingénierie XML. Elle y décrit un plan de transformation d'un schéma conceptuel en un schéma conforme XML et offre quelques primitives de transformation et de validation de schémas qui facilitent la réalisation de l'objectif. Des détails sur le plan et ces outils peuvent être trouvés dans [Del01c].

Lors des premières manipulations de l'outil réalisé par Christine, nous avons constaté que l'application de la méthode était plutôt laborieuse : les modules développés étant totalement indépendants, l'utilisateur n'était pas suffisamment guidé dans sa tâche. Bref, il n'était pas évident de déterminer et de retenir quelle procédure devait être exécutée, et à quel moment ? Dans la solution actuellement proposée pour ce mémoire, l'utilisateur n'interagit pas directement sur le schéma mais effectue ses choix par l'intermédiaire de listes de choix apparaissant de manière séquentielle. Ces choix déclenchant des actions sur le schéma de travail. Plus besoin de «fouiller» désespérément dans les nombreux menus de DB-Main pour trouver la prochaine action à effectuer ; il suffit de suivre les indications affichées à l'écran. Un feedback est également fourni à l'utilisateur, par l'utilisation de boîtes à messages : elles apparaissent afin d'informer l'utilisateur des conséquences importantes liées à ses choix.

Une majeure partie de nos réflexions se sont également attardées sur le problème de la construction de la hiérarchie : est-il préférable d'envisager une solution entièrement automatique ou, à l'inverse, doit-on obliger l'utilisateur à construire lui-même sa structure arborescente ? Ne pourrait-on pas trouver une manière de faire intermédiaire ? Quels choix proposer à l'utilisateur ? Comment l'utilisateur peut-il résoudre les conflits ? Quelles informations fournir à l'utilisateur ? La solution actuelle est une solution parmis d'autres, elle a été élaborée en imaginant les besoins et les exigences d'un utilisateur type, conscient des objectifs à atteindre mais souhaitant un processus convivial et aisé.

En automatisant certaines tâches, nous avons également fortement réduit le nombre d'actions demandées à l'utilisateur. Par exemple, lors de la transformation des attributs en attributs ou en éléments XML, l'utilisateur choisit – dans une boîte de choix qui s'ouvre à lui – une transformation globale (parmis les deux possibles) qui sera appliquée à tous les attributs du schéma ; il n'est alors plus nécessaire de traiter chaque attribut au cas par cas.

Notre tâche ne s'est pas limitée à assembler séquentiellement dans un programme des modules déjà tout faits ; nous avons également cherché des améliorations à apporter d'un point de vue fonctionnel : la transformation des relations IS-A par la création automatique de groupes «choice» et «seq» et de types d'entités techniques est une nouvelle fonctionnalité intéressante car elle permet la traduction systématique de contraintes exprimées dans le schéma initial.

3.5.2 Présentation de l'outil de transformation de schéma

L'outil de transformation de schéma est une implémentation de la méthode de transformation décrite dans la section précédente. L'outil se présente sous la forme d'une méthode de travail composée de processus (automatiques et interactifs) qui manipulent un schéma conceptuel afin de le rendre conforme au modèle XML. C'est l'utilisateur qui déclenche séquentiellement les processus. Lors de l'exécution d'un processus interactif, l'interaction se fait par le biais de boîtes à messages et de boîtes de choix.

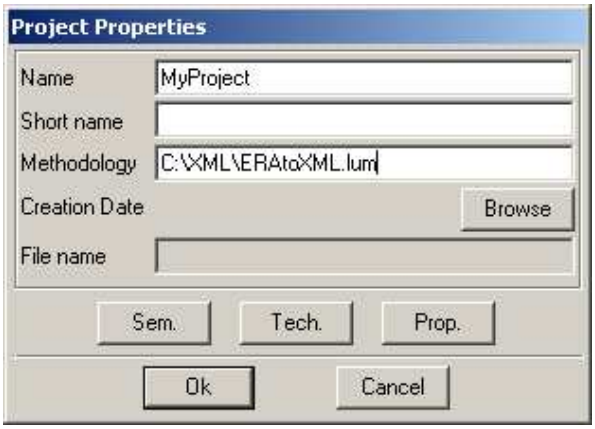

FIG. 3.39 – La fenêtre de création de projet DB-Main : le bouton *Browse* permet de choisir le fichier `ERAtoXML.lum` contenant la méthode

Techniquement, la méthode à suivre a été modélisée via l'environnement de Process Modeling [12] accompagnant l'outil DB-Main. Certains processus déclenchent l'exécution d'un programme Voyager 2 [13] qui analyse et agit sur le schéma en cours.

Limitations de l'outil

Vu la difficulté croissante du problème combinatoire sous-jacent, la transformation de relations IS-A soumises à une contrainte de totalité et possédant plus de deux sous-types, n'est pas supportée dans la version actuelle de l'outil.

Description du fonctionnement de l'outil

Création d'un projet dans DB-Main Lors de la création d'un nouveau projet dans DB-Main, il est possible d'associer au projet une méthode (fichier .lum) sensée guider la tâche de l'utilisateur au sein du projet. Fig.3.39 montre la fenêtre de création de projet DB-Main ; le bouton *Browse* permet de sélectionner le fichier .lum décrivant la méthodologie. Dans notre cas, le fichier se nomme `ERAtoXML.lum`.

La méthodologie apparaît alors graphiquement dans une fenêtre DB-Main : les rectangles représentent des processus et les ovales représentent des produits, i.e. des schémas intervenant dans les processus. Les flèches entre processus et produits indiquent le sens des entrées-sorties. Une flèche à double sens signifie que le processus met à jour (update) le produit.

Le processus en cours est représenté par un rectangle de couleur verte. Un clic droit suivi d'un clic sur l'option *Execute* permet d'exécuter le processus en cours. Fig.3.40 montre la méthodologie proposée pour transformer un schéma conceptuel en schéma conforme XML. La méthodologie comporte six processus alors que le plan de transformation décrit précédemment comportait huit étapes. En effet, pour des raisons pratiques, certaines étapes automatiques du plan ont été regroupées en un seul processus dans l'outil implémenté. Deux processus d'importation et de copie de schéma ont été ajouté par rapport à la méthode.

A tout moment, la fenêtre principale du projet trace l'historique des actions et des processus exécutés.

[12] pour plus d'informations sur le Process Modeling et son langage de description, voir [LIB02]

[13] Voyager 2 est le langage de programmation interne à DB-Main. Sa principale caractéristique est qu'il permet d'accéder aux objets du repository de DB-Main. Pour plus d'informations, veuillez consulter le manuel de référence [Eng00]

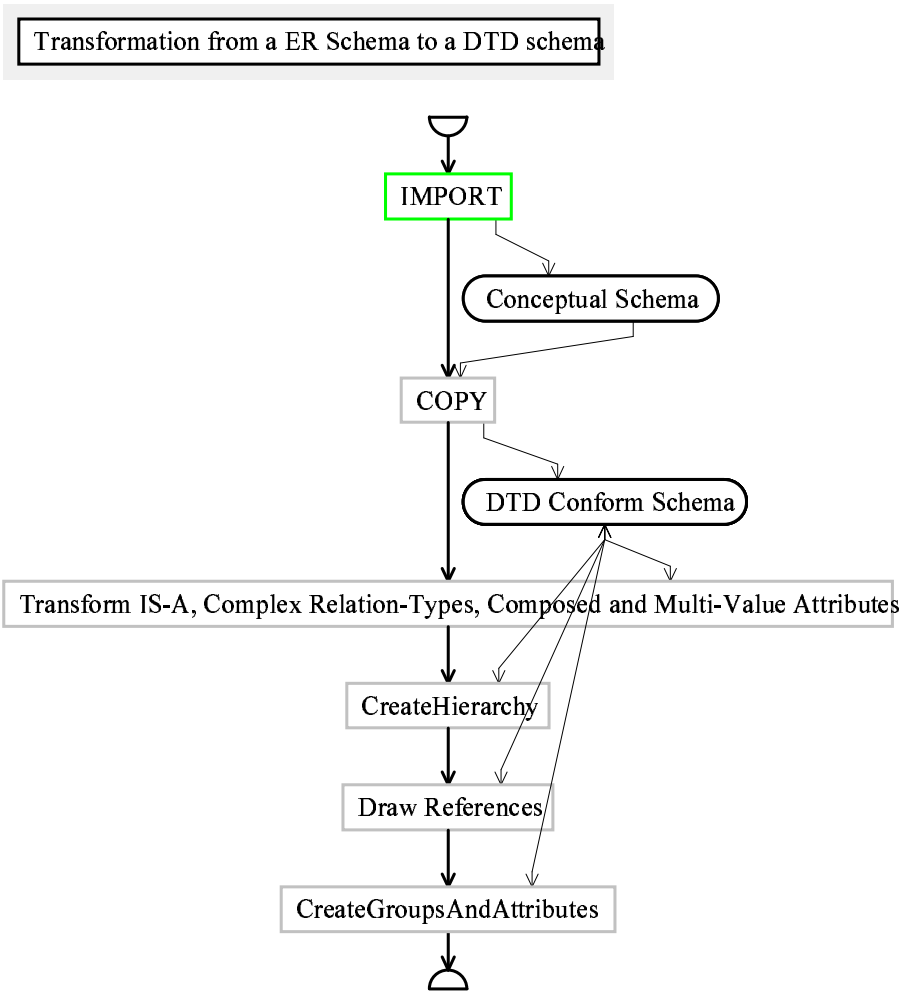

FIG. 3.40 – Les six processus séquentiels composant la méthode de transformation

L'application de la méthode débute après qu'un schéma de travail ait été importé et copié dans DB-Main. C'est le rôle des deux premiers processus : «IMPORT» et «COPY».

Le processus «IMPORT» Le premier processus de la méthode consiste à importer, dans le projet, le schéma conceptuel à transformer. L'exécution de ce processus ouvre une fenêtre dans laquelle on choisit un schéma de travail (exporté au format .isl [14]). Ce schéma – appelé dans la méthode `Conceptual schema` – apparaît alors dans la fenêtre principale retraçant l'historique du projet.

Le processus «COPY» Au cours de l'application de la méthode, le schéma conceptuel de départ va subir des transformations. Il est donc préférable, avant toute modification, de copier ce schéma dans un schéma «de travail» que nous appellerons, par simplification, le schéma DTD. Cette manière de faire permet une meilleure traçabilité car il est alors possible de se référer au schéma conceptuel initial à n'importe quel moment de la méthode, afin de suivre l'évolution du schéma. L'exécution du processus «COPY» s'ouvre sur une fenêtre dans laquelle on peut préciser les propriétés (Nom, Nom court, Version...) du nouveau schéma DTD.

Le processus «Transform IS-A, Complex Relation-Types, Composed and Multi-Value Attributes» Ce processus automatique applique au schéma DTD les transformations globales décrites dans la première étape de la méthode.

Le processus «Create Hierarchy» Ce processus fusionne les étapes 2 et 3 du plan de transformation afin de construire une structure arborescente dont les cardinalités des rôles pères sont conformes au modèle XML. A l'issue de l'exécution de ce processus, les relations de références, qui ont été décidées, restent toutefois matérialisées par des types d'associations [0-N, 1-1]. Elles deviendront des clés étrangères dans le processus suivant, intitulé «Draw References».

Ce processus constitue le coeur de la méthode et se veut également le plus interactif.

Le traitement des types d'associations issus des relations IS-A étant entièrement automatique, la première tâche de l'utilisateur sera de décider le sens des types d'associations 1-1 du schéma [15]. L'interface proposée pour réaliser ce choix est assez intuitive : pour chaque type d'associations de cardinalités 1-1, une boîte contenant deux propositions – correspondant aux deux orientations possibles – apparaît. Le choix est obligatoire, il se structure de la manière suivante :

«De `<type d'entités A>` vers `<type d'entités B>`»
ce qui signifie :
le `<type d'entités A>` est un père potentiel du `<type d'entités B>`

Fig.3.41 montre la boîte de choix implémentée dans l'environnement DB-Main : en sélectionnant une des deux possibilités, l'utilisateur précise l'orientation des types d'associations 1-1.

Le programme trouve ensuite automatiquement la (les) racine(s) obligatoire(s) et affiche le(s) nom(s) de celle(s)-ci dans une boîte à message, comme celle présentée Fig.3.42.

Le choix des racines facultatives se fait dans la fenêtre suivante : il s'agit de sélectionner une à une les racines parmis la liste des racines non-obligatoires. Lorsque l'on clique sur le bouton *OK*, le nom de la racine sélectionnée disparaît de la liste et il est alors possible de sélectionner une autre racine facultative. Une fois le choix des racines terminé, il suffit de cliquer sur le bouton *Annuler* pour passer à la fenêtre suivante.

[14] pour plus de détails sur ce format, consultez [Hai99]

[15] se référer au paragraphe «Le traitement des types d'associations [1-1,1-1]», page 44

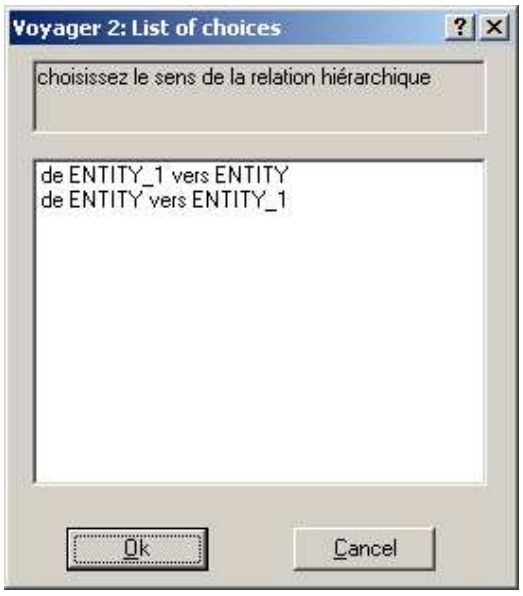

FIG. 3.41 – La fenêtre de choix de l'orientation des types d'associations [1-1,1-1]

FIG. 3.42 – Une boîte à message indique à l'utilisateur le nom des racines obligatoires trouvées

FIG. 3.43 – Les racines facultatives se choisissent une à une par l'intermédiaire de cette boîte de choix

FIG. 3.44 – L'interface de résolution de conflits

FIG. 3.45 – Les types d'entités pour lesquels un identifiants techniques a été ajouté sont repris dans une boîte de dialogue

En toute logique, il s'agit ensuite pour l'utilisateur de résoudre les conflits se produisant lorsqu'un type d'entités possède plusieurs pères potentiels. L'interface permettant de choisir le père d'un type d'entités se présente toujours sous la forme d'une liste de choix : pour chaque conflit, la liste des pères potentiels apparaît et l'utilisateur doit en sélectionner un seul. Le choix est obligatoire. L'interface de résolution de conflit est présentée Fig.3.44

Suite à ces différentes décisions, certains types d'associations prennent le statut de relation hiérarchique, d'autres deviennent des relations de référence. Pour ces dernières, le type d'entités référencé doit posséder un identifiant composé d'un attribut simple. Si ce n'est pas le cas, un identifiant simple technique est automatiquement ajouté au type d'entités. Les noms des types d'entités pour lesquels un tel identifiant a été ajouté, sont répertoriés dans une boîte de dialogue (Fig.3.45)

Finalement, lorsqu'une racine unique doit être ajoutée au schéma, l'utilisateur en est avertit par un message similaire au message présenté Fig.3.46.

Le processus «Draw References» Ce processus transforme automatiquement les types d'associations référentiels en clés étrangères dont l'origine est un groupe «ref» contenant un attribut de référence et la destination l'attribut identifiant le type d'entités référencé. Il s'agit d'une transformation de schéma bien connue dans le modèle relationnel : un type d'associations 1-N étant substitué par une clé étrangère. Le plan de transformation décrit ci-dessus illustre

FIG. 3.46 – L'ajout d'une racine unique provoque l'affichage de ce type de message

FIG. 3.47 – L'interface permettant de choisir comment représenter les attributs du schéma dans le modèle XML.

à maintes reprises cette transformation symétriquement réversible. Dans la Fig.3.24 page 49, les types d'associations référentiels `About` et `Supply` de la Fig.3.23 p.49 ont été transformés en références et des groupes «ref» ont été créés dans le type d'entités `DETAIL`.

Le processus «CreateGroupsAndAttributes» L'exécution de ce processus déclenche tout d'abord une série de traitements automatiques de création des groupes «gid», «idref» et «choice». Ces actions ont été détaillées dans les étapes 4 et 5 du plan de transformation.

Ensuite, l'utilisateur doit choisir le mode de représentation des attributs non-impliqués dans un groupe «gid» ou «idref» : trois choix sont possibles dans la boîte de choix présentée à l'écran (Fig.3.47) : la représentation par attributs XML, par éléments XML (avec application de la règle de **Fusion**) ou par éléments XML (avec application de la règle de textbfDistinction). Le choix est obligatoire et s'applique à l'entièreté du schéma. Aucune règle formelle ne permet de déterminer dans quelles circonstances une solution est préférable à l'autre.

La dernière intéraction du programme avec l'utilisateur est une boîte à message listant les groupes non-conformes qui ont été supprimé du schéma (Fig.3.48).

Enfin, un dernier traitement automatique va soigner l'apparence graphique du schéma conforme au modèle XML obtenu. Il s'agit nullement d'apporter des modifications supplémentaires au schéma mais de le présenter sous une forme arborescente. La racine est positionnée en haut à gauche, les fils sont représentés de haut en bas et les différentes générations, de gauche

FIG. 3.48 – Les groupes qui ont été supprimés du schéma sont répertoriés dans une boîte à messages

à droite.

Dans le modèle XML, le nom des types d'associations n'a aucune importance car aucune sémantique n'est associée à une relation hiérarchique. On suggère donc également dans cette dernière étape d'effacer les noms des types d'associations hiérarchiques.

3.5.3 Outil de validation d'un schéma XML

Dans le menu *Assist* de l'outil-case DB-Main, on trouve un puissant analyseur de schéma. Cet outil permet de valider un schéma par rapport à un modèle. Malgré toute l'attention portée au développement de la méthode de transformation, il est toujours préférable d'utiliser cet outil mis à notre disposition afin de s'assurer que le schéma obtenu réponde bien aux nombreux critères de conformité. Une description complète de l'analyseur de schéma se trouve dans l'aide de DB-Main [Hai99].

Pour vérifier la conformité d'un schéma au modèle XML, il a fallu implémenter certains prédicats. La première choses à faire est de charger la librairie comprenant ces prédicats. La fenêtre de chargement de librairie peut être atteinte grâce aux menus *Assist/Schema analysis.../Edit library/Load library*. Le fichier-librairie à sélectionner est `XML.anl`.

Une fois la librairie chargée, il faut charger un script de validation à partir de l'écran d'analyse de schéma (*Assist/Schema analysis...*). Le fichier-script se nomme `XML.ana`.

En cliquant sur OK, le schéma est analysé et les prédicats non-vérifiés par le schéma apparaissent dans une boîte de dialogue.

Chapitre 4

Conception Physique et Codage

Dans ce chapitre consacré à la conception physique et au codage, nous examinons les propriétés physiques à définir sur une structure de données XML afin d'améliorer les performances de la future base de données. Ensuite, nous expliquons comment générer le code d'une DTD à partir du schéma logique XML construit dans DB-Main lors de l'étape précédente.

4.1 Conception Physique

4.1.1 Généralités

Les performances d'une base de données XML native vont principalement dépendre de trois facteurs :

1. la **configuration matérielle** du serveur sur lequel tourne la base de données : il n'est pas à démontrer qu'une base de données implantée sur une configuration musclée fournira de meilleures résultats que sur une machine désuète.
2. la **formulation des requêtes** : une requête précise formulée sur un élément particulier sera bien plus performante qu'une requête d'ordre général, utilisant des «wild-cards» et concernant plusieurs types d'éléments.
3. les **propriétés physiques** associés aux données (index, organisation physique ...) : l'utilisation d'index sur les types d'éléments les plus souvent demandés améliore considérablement le temps de réponse des requêtes.

Avant de définir les critères de performance d'une structure de données XML, il est donc indispensable de pouvoir anticiper le type des requêtes qui seront le plus souvent exécutées : quel(s) type(s) d'élément(s) seront concernés par ces requêtes ? Les requêtes feront-elles usage de wild-cards ? Quels opérateurs de comparaison vont-elles utiliser ?

Les réponses à ces questions vont nous aider à sélectionner, de manière optimale, les propriétés physiques de la base de données. Certaines propriétés sont définies de manière générale pour toute la DTD alors que d'autres sont spécifiques à un élément ou un attribut XML. Nous examinons ces propriétés dans les deux prochaines sections.

4.1.2 Propriétés physiques générales d'une DTD

Le mode d'accès à la DTD

Le **mode d'accès** spécifie le(s) opération(s) autorisée(s) sur les données définies par la DTD. Les valeurs possibles sont :

- **LECTURE :** le contenu des données peut être consulté grâce à l'utilisation de requêtes.
- **INSERTION :** de nouveaux éléments peuvent être insérés.

– **MISE A JOUR :** des éléments peuvent être mis à jour.
– **SUPPRESSION :** des éléments peuvent être supprimés.

Plusieurs valeurs peuvent être précisées : «LECTURE INSERTION MISE A JOUR SUPPRESSION» autorise un accès complet aux données.

La structure d'index

Un élément de type ANY (voir 2.5.1, page 18) peut posséder des éléments qui ne sont pas explicitement déclarés dans la DTD. La **structure d'index** indique si ces éléments doivent être enregistrés dans un «repository» afin qu'ils puissent bénéficier d'un traitement plus rapide lors des requêtes les concernant. Les valeurs possibles sont :

– **CONDENSE :** les éléments non-déclarés dans la DTD sont enregistrés dans le «repository» comme éléments facultatifs.
– **COMPLET :** les éléments non-déclarés dans la DTD sont enregistrés dans le «repository».
– **AUCUNE :** les éléments non-déclarés dans la DTD ne sont pas enregistrés dans le «repository». Cette valeur n'est recommandée que si tous les éléments apparaissant dans les instances XML ont été déclarés au niveau de la DTD.

Exemple : Considérons la DTD suivante :

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT A ANY>
```

Nous savons, en plus, que dans certains cas l'élément `A` admet un élément fils nommé `B` mais celui-ci n'est pas déclaré dans la DTD.

Des requêtes telles `/A/B` ou `/A/*` seront résolues bien plus rapidement si la valeur «CONDENSE» ou «COMPLET» est définie pour la structure d'index.

4.1.3 Propriétés physiques propres à un élément / attribut XML

Le mode d'indexation

Le **mode d'indexation** indique comment est indexé chaque noeud (élément ou attribut) de la structure de données XML. Les valeurs possibles sont :

– **TEXTE :** ce mode d'indexation supporte l'opérateur « `~=` » du langage XQuery (voir 5.1.3, page 77) et est donc recommandé si on désire comparer et extraire des valeurs textuelles en utilisant des «wild-cards». Par exemple, la requête :
`/patient[name/surname~='atkin*']`
qui se charge de retrouver tous les patients dont le nom de famille commence par «atkin», sera beaucoup plus performante si un index de type TEXTE est ajouté à l'élément `surname`.
L'indexation de type TEXTE est une propriété qui se transmet de père en fils : lorsqu'un élément possède un index de ce type, tous les fils de cet élément en possèdent alors également un.
Lors du chargement de documents dans une base de données, la création de tels index est une opération assez lourde et coûteuse. Il s'agit donc d'exploiter cette technique d'indexation avec modération.
– **STANDARD :** ce mode d'indexation est recommandé pour la comparaison de données numériques dont le contenu est défini, c'est-à-dire sans «wild-cards». Par exemple, l'ajout d'un index de type STANDARD sur l'élément `born` permet à cette requête de s'exécuter dans de meilleures conditions :
`/patient[born > 1950]`

Ce mode d'indexation accélère également les opérations de tris (descendants ou ascendants). Il n'est autorisé que pour les éléments situés dans les «feuilles» de l'arbre XML.
- **TEXTE et STANDARD :** ce mode d'indexation est plutôt déconseillé car la création des index implique un temps de chargement dans la base de données assez important. Ce choix peut toutefois s'avérer judicieux si on désire classer par ordre alphabétique les valeurs d'un élément commençant par un certain préfixe.
- **AUCUN :** aucun index n'est associé au noeud.

Le type de «mapping»

Le **type de «mapping»** définit la manière dont les informations sont physiquement stockées. Les principales valeurs sont :
- **NATIF :** l'élément *et toute sa descendance* sont stockés tels quels dans un agrégat d'emplacements physiques que l'on appelle un «cluster». Il peut être intéressant de choisir ce type de mapping pour un élément qui sera souvent extrait avec tous ses fils. Dans l'exemple suivant, l'élément `ORDER` est déclaré de type NATIF afin d'être stocké dans la base de données en compagnie de tous ses éléments `DETAIL`.
```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT ORDER (DETAIL+)>
<!ATTLIST ORDER NumOrder CDATA #REQUIRED>
<!ELEMENT DETAIL EMPTY>
<!ATTLIST DETAIL IdProd CDATA #REQUIRED>
<!ATTLIST DETAIL Quantity CDATA #REQUIRED>
```
- **INFORMATION :** l'élément (ou attribut) et sa valeur sont stockés à un endroit qui ne dépend pas de leur père ou de leur(s) fil(s) potentiel(s).
- **COLONNE SQL :** la valeur de l'élément est stockée dans une colonne d'une base de données relationnelle. L'association entre un élément (ou attribut) XML et une colonne d'une base de données relationnelle est appelé un «mapping».
- **AUCUN :** l'élément (ou attribut) n'est pas stocké dans la base de données.

4.2 Le codage

La phase de codage consiste simplement à générer le code de la DTD qui correspond au schéma XML construit lors de la conception logique. Cette tâche est automatique et l'outil utilisé pour la réaliser est le générateur de DTD développé par Christine Delcroix et intégré dans l'outil CASE DB-Main.

4.2.1 Outil de génération de DTD

A partir d'un schéma conforme au modèle XML, le code de la DTD correspondante peut être automatiquement généré. L'opération se déclenche via le menu *File/Generate/XML...* et le fichier résultat est créé dans le répertoire courant. Le générateur de DTD ne fait aucune hypothèse sur le schéma de travail. Afin de ne pas produire des résultats absurdes ou incorrects, il est donc important de s'assurer que le schéma pour lequel on désire la DTD est bien conforme au modèle XML. Pour ce faire, l'utilisation de l'outil de validation de schéma (voir 6.19, page 103) est plus que conseillée.

Chapitre 5

Implantation dans un SGBD XML Natif

Dans ce chapitre, nous allons voir comment utiliser la DTD générée par DB-Main pour décrire la structure des documents stockés dans une base de données XML natif. Le SGBD choisi est Tamino (version 2.3.1), produit phare de la société allemande Software AG. Après avoir importé dans Tamino des données conformes à la DTD, nous présentons également comment exploiter ces données à partir d'un browser internet ou d'un client Java.

5.1 Présentation d'un SGBD XML Natif : Tamino

5.1.1 Software AG

Tamino est un produit développé par l'éditeur allemand de base de données : Software AG.

Fondée en 1969 par Peter Schnell, cette société a connu son heure de gloire dans les années 80 avec des produits innovateurs tels Adabas ou L4G Natural.

Depuis 1998, l'entreprise a concentré son développement sur les produits XML pour l'Internet et, grâce à ce choix, Software AG effectue actuellement un «come-back» remarquable : l'entreprise est l'un des principaux fournisseurs de logiciels systèmes destinés à la gestion des données et au développement de solutions de commerce électronique.

Tamino s'impose sur le marché des bases de données spécifiques mais ne compte pas en rester là ; en témoignent ses récents partenariats avec BEA, IBM et HP pour accompagner WebLogic, Websphere et Bluestone, leurs serveurs d'applications respectifs.

5.1.2 L'offre logicielle de software AG

En plus de Tamino, Software AG propose toute une gamme de produits complémentaires, permettant de mettre en place une solution totalement intégrée :

- **Entire X** est une solution middleware qui couvre l'ensemble des besoins dans le domaine de l'intégration d'applications (EAI).
- **Bolero** est un environnement de développement d'applications orientées e-business, permettant de réaliser des applications réparties pour les plates-formes Java. Grâce à Bolero, les applications Java peuvent traiter directement des documents XML. Ces documents sont ensuite archivés dans Tamino.
- **Adabas** reste un SGBD renommé pour ses bonnes performances et son excellente fiabilité.
- **Natural** est un des plus puissants L4G au monde. Il a été spécialement défini pour construire des applications flexibles, robustes et performantes.

5.1.3 Les caractéristiques de Tamino

Serveur de données complexes XML

Le langage XML s'impose de plus en plus comme le standard international permettant de structurer les informations complexes, de différentes catégories (textes, tableaux, vidéo,son...). Tamino est un serveur de données capable de stocker des informations en langage XML sans les convertir dans d'autres structures de données. C'est ce qu'on appelle le **stockage XML natif** : Tamino stocke en effet les documents XML dans leur structure native, sous forme d'objets XML hiérarchiques complexes. On évite ainsi la dissémination du document hiérarchique dans les tables et les colonnes propres à l'approche relationnelle. Lorsque ces documents sont stockés ou consultés, leur structure ne subit aucune conversion, ce qui assure une rapidité de stockage et de consultation, même pour de grosses quantités d'information. Les performances obtenues, en matière d'accès aux informations stockées dans les bases de données conviennent en particulier aux applications de **commerce électronique**.

Passerelle avec les bases de données existantes

Grâce au module X-Node inclus dans Tamino, il est possible d'intégrer des données provenant de bases de données existantes dans des structures XML. Une interface SQL fournit l'accès aux données relationnelles enfouies dans des documents XML. Avec ce dispositif, Tamino peut remplacer complètement un SGBD relationnel.

Les fonctionnalités classiques d'un bon SGBD

Outre les possibilités de stocker et d'interroger des documents XML, Tamino offre quelques fonctionnalités classiques des SGBD : possibilité de backup, gestion des transactions, définition de groupes d'utilisateurs...

Compatible avec de multiples plates-formes

Tamino supporte tous les types de plates-formes, des serveurs NT aux gros ordinateurs sous Linux ou OS/390 en passant par les systèmes UNIX. Les utilisateurs de ce système d'exploitation IBM peuvent exploiter XML pour intégrer leurs applications existantes dans des environnements e-business de toutes tailles.

Construit sur des standards de fait

Tamino s'appuie sur des standards tels que XML, TCP/IP et HTTP, ce qui réduit le risque d'obsolescence du système et augmente la compétitivité entre les acteurs qui proposent des solutions.

Un format de réponse standard

Lorsque l'on interroge Tamino, la réponse fournie par ce serveur de données se présente sous la forme d'un document XML. Le client qui reçoit cette réponse, formulée dans un langage standard, peut alors directement l'exploiter sans traduction intermédiaire. Si le client , par exemple, est un browser internet, celui-ci peut présenter au client de manière conviviale le résultat de la requête, par application de transformations XSL sur le document XML reçu. La technologie X-Machine (voir section 5.2.4, page 84) développée par Software AG permet d'agir directement sur une base de données Tamino à partir d'un client HTTP (browser internet) en insérant des commandes X-Machine spéciales dans la requête HTTP du client.

Un serveur extensible

Des extensions, sous forme de plug-in, peuvent être ajoutées au moteur de stockage de Tamino, comme par exemple un processeur XSL qui appliqueraient des transformations aux documents XML, au niveau du serveur.

Le dialogue avec Java

Tamino offre une librairie de composants Java permettant d'implémenter un client Java pour communiquer avec les bases de données Tamino. Les réponses de Tamino au client Java se présentent sous la forme d'un objet DOM que l'on peut facilement parser en utilisant les librairies appropriées.

L'administration des bases de données via le navigateur

L'outil d'administration des bases de données Tamino – appelé le System Management Hub – est une application s'exécutant dans n'importe quel browser internet. L'administration est donc indépendante de la plate-forme et peut se réaliser à distance. Le scénario de création d'une base de données via l'interface du System Management Hub est décrit dans la section 5.2.1 suivante.

Le langage XQuery pour interroger les données

Le langage d'interrogation des base de données Tamino est XQuery. XQuery est une variante de XPath : il en reprend la même syntaxe tout en y ajoutant quelques fonctionnalités supplémentaires qui relèvent du détail. Par exemple, XQuery supporte l'opérateur de comparaison « `~=` » qui autorise l'emploi de «wild-cards» dans la valeur de l'élément ou attribut.

5.2 Les composants logiciels de Tamino

5.2.1 Le System Management Hub

Le System Management Hub est l'outil d'administration des bases de données Tamino. Il s'agit exactement d'une application réalisée en Javascript/HTML, s'exécutant donc dans n'importe quel browser internet à une URL particulière. L'administration (création d'utilisateur, de bases de données...) est donc indépendante de la plate-forme et peut se réaliser à distance via des requêtes HTTP, à condition de connaître l'adresse du serveur Tamino et le mot de passe Administrateur.

Interface principale

Après l'étape d'identification [1], le programme souvre sur l'interface web présentée Fig.5.1 : l'URL du browser (`http ://localhost :9991/smh/login.htm`) nous indique que l'application tourne en local sur le port 9991. L'écran est divisé en trois parties :

- la partie supérieure gauche est un menu hiérarchique de navigation à travers les diverses fonctionnalités du programme. Un clic sur l'icône «+» permet de déployer le contenu d'un menu. Le menu *Administrators*, par exemple, permet d'ajouter ou de supprimer des administrateurs au système alors que le menu *Operating System* nous renseigne sur les propriétés du système d'exploitation et les processus en cours. Le menu qui nous intéresse particulièrement est évidemment *Databases*, il sera détaillée dans la section suivante.

[1] Le nom d'utilisateur et le mot de passe sont ceux utilisés lors d'une ouverture de session sur le système d'exploitation

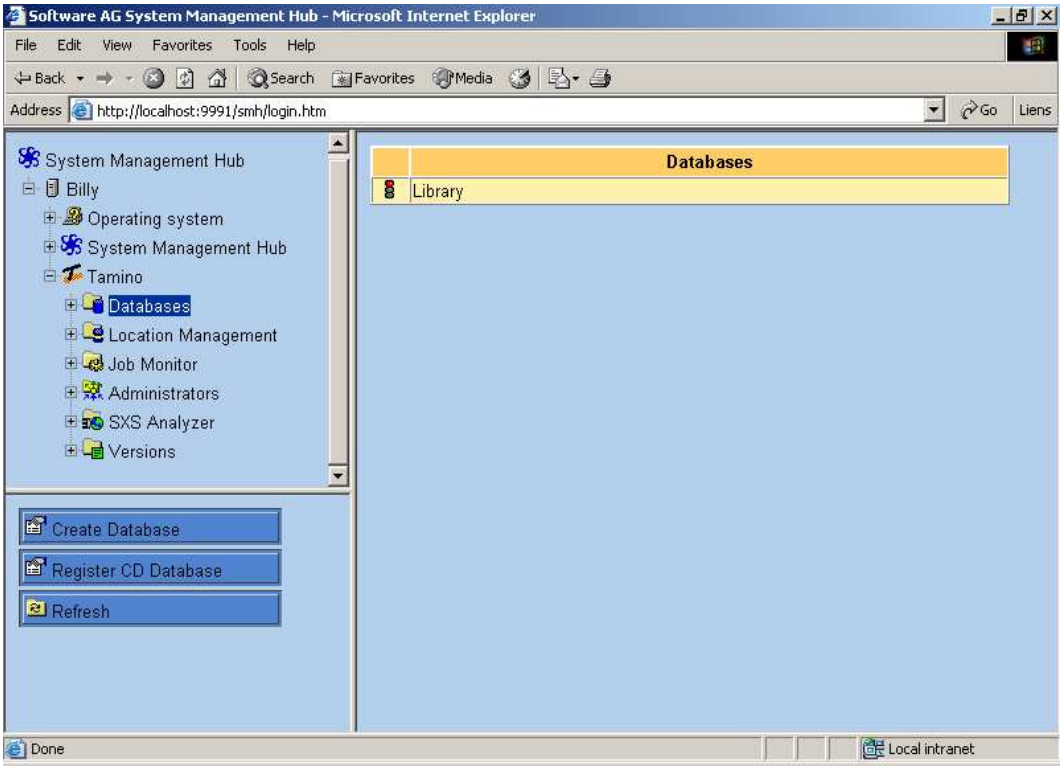

FIG. 5.1 – L'interface web du System Management Hub

- la partie inférieure gauche présente une liste de boutons correspondant aux actions réalisables dans le menu sélectionné. Par exemple, dans le menu *Administrators* cité précédemment, les actions possibles sont *Add Administrator* et *Delete Administrator*, correspondant respectivement à la création et la suppression d'un administrateur.
- la partie droite affiche le résultat – bien souvent sous la forme d'un tableau – des actions déclenchées par l'utilisateur.

Le menu *Databases*

Le menu *Databases* présente les bases de données Tamino créées et leur état (active ou inactive). La création d'une base de données se fait à partir de ce menu, via le bouton d'action *Create Database* : la seule information obligatoire à fournir lors de la création d'une base de données, c'est son nom. Par défaut, une base de données est créée à l'adresse suivante : `http ://localhost/tamino/<nom de la base de données>`. La Fig.5.2 présente le menu *Databases* du System Management Hub : les sous-menus `Library` et `Order` correspondent chacun à une base de données. L'état d'une base de données est indiqué par un feu de signalisation situé à côté du nom de la base de données (couleur «rouge» = inactive et couleur «verte» = active).

Le sous-menu `Library` est lui-même composé de plusieurs sous-menu correspondant à des catégories d'actions spécifiques à cette base de données :

- **Information :** fournit des informations de base sur l'activité de la base de données et les sessions utilisateurs.
- **Backups :** permet de réaliser un backup de la base de données. Un backup, avec Tamino, se fait toujours de manière complète et online, i.e. qu'il n'est pas nécessaire d'arrêter la base de données lors de l'opération. Il est demandé de choisir l'endroit où le backup doit

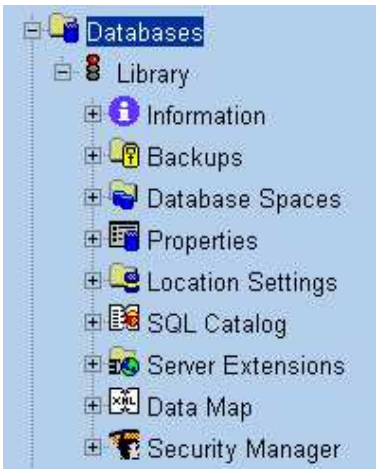

FIG. 5.2 – Le menu *Databases* du System Management Hub

être réalisé : il s'agit bien souvent d'un disque de capacité ou d'une bande magnétique. La restauration de données, suite à un crash, peut également être réalisée à partir de ce menu.

- **Database Spaces :** permet de définir la localisation (définie dans le menu *Location Settings*) et la taille des espaces physiques réservés aux données, aux index, aux backups...
- **Properties :** permet de modifier certaines propriétés assez techniques du serveur de base de données : le numéro des ports à utiliser pour s'y connecter, le nombre maximum de sessions concurrentes...
- **Location Settings :** permet d'associer une localisation à un emplacement physique bien particulier. Par exemple, un administrateur peut décider que la localisation nommée `Backup` correspondra à l'emplacement physique `c :Backup`. Dans le menu *Database Spaces* citée ci-dessus, il associera la localisation nommée `Backup` à la fonction de backup, nommée `Backup Spaces`.
- **SQL Catalog :** en se connectant à un catalogue SQL, on peut avoir accès aux méta-informations d'une base de données relationnelle (nom des tables et colonnes, privilèges des utilisateurs...). Il est possible d'utiliser ces informations pour charger dans une table des données présentes dans un fichier XML.
- **Server Extensions :** les fonctionnalités du serveur Tamino peuvent être élargies grâce à des extensions programmées par l'utilisateur ou fournies par Software AG, sous forme de plug-in.
- **Data Map :** dans ce menu, on peut consulter la liste des **Collections** définies pour la base de données, ainsi que leur DOCTYPE associé. Nous reviendrons sur ce terme de **Collections** dans la section suivante consacrée à l'éditeur de schémas Tamino.
- **Security Manager :** permet de créer des utilisateurs, de les affecter à des groupes et de définir des autorisations d'accès (en lecture et en écriture) à certains noeuds.

Les actions associées à ce sous-menu `Library` sont illustrées dans la Fig.5.3. Il s'agit de :

- **Start Database :** active la base de données. L'icône en forme de feu de signalisation passe au vert et la base de données est alors prête à recevoir les requêtes des clients.
- **Stop Database :** désactive la base de données. L'icône en forme de feu de signalisation passe au rouge et la base de données ne répond plus aux requêtes des clients.
- **Delete Database :** efface la base de données.
- **Rename Database :** renomme la base de données.
- **Set Version :** permet de changer la version de la base de données afin de la rendre

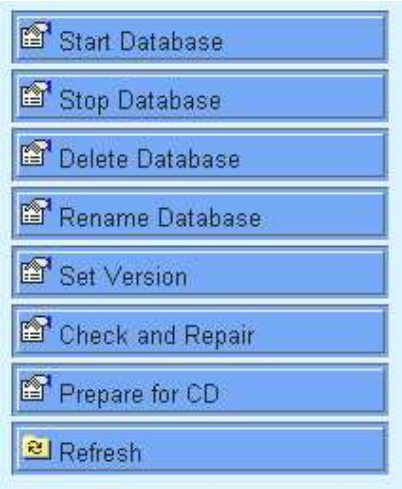

FIG. 5.3 – Les actions associées à une base de donnée particulière

conforme à la version actuelle de Tamino.
- **Check and Repair :** vérifie la consistence de la base de données et répare les éventuelles erreurs.
- **Prepare for CD :** produit une image de la base de données complète afin de permettre sa distribution via un support quelconque.

5.2.2 L'éditeur de schémas Tamino

La notion de «Schéma Tamino»

L'éditeur de schéma est l'outil de conception physique fournit avec Tamino. Il s'agit d'un environnement graphique utilisé pour définir la *structure* et les *propriétés physiques* (voir chapitre précédent, page 71) des données XML stockées dans les bases de données Tamino. Toutes ces informations forment un **schéma Tamino** et sont enregistrées dans un fichier XML, selon une syntaxe définie par Software AG. Ce fichier XML, généré automatiquement à partir des manipulations graphiques de l'utilisateur, peut être considéré comme l'équivalent du code DDL (Data Definition Language) utilisé dans les SGBD relationnels.

La DTD et le **schéma Tamino** sont deux syntaxes différentes utilisées pour décrire la structure de données XML. Le schéma Tamino est cependant plus riche car il possède des informations physiques supplémentaires.

La notion de «Collection»

Dans une base de données XML, les données XML stockées peuvent être structurées conformément à différents schémas Tamino. L'ensemble des données conformes au même schéma forment une **collection**. Par exemple, dans une base de données nommées `eCommerce`, on pourrait imaginer :
- une collection, nommée `CustomerCollection`, conforme à un schéma Tamino dont la racine (DOCTYPE) s'appelle `Customer`.
- une collection, nommée `ProductCollection`, conforme à un schéma Tamino dont la racine (DOCTYPE) s'appelle `Product`.

La jointure entre deux collections n'étant pas une opération évidente à réaliser, on préfèrera, pour des bases de données de taille raisonnable, rassembler toutes les données dans une seule

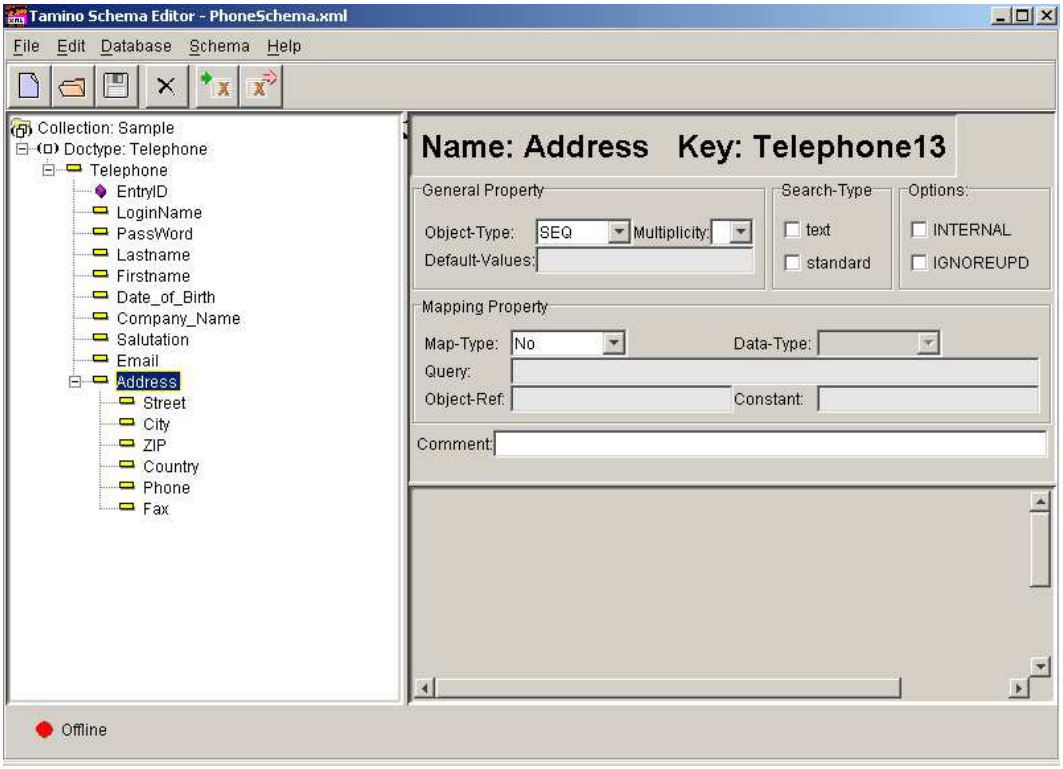

FIG. 5.4 – L'interface principale de l'éditeur de schémas Tamino

collection conforme à un schéma dont la racine représente tout le schéma de base de données : dans notre exemple, la racine serait nommée `eCommerce` ; `Customer` et `Product` seraient ses deux descendants directs.

Définition de la structure des données contenues dans une collection

Avec l'éditeur de schémas Tamino, il existe deux manières de définir la structure des données contenues dans une collection : soit l'utilisateur crée la structure graphiquement à partir de rien, soit il utilise la structure d'une DTD qu'il importe dans l'éditeur de schémas.

Définition de la structure des données à partir de rien L'utilisateur utilise la partie gauche de l'interface de l'éditeur de schémas pour construire l'arborescence des éléments XML : il choisit le nom de la collection décrite par le schéma Tamino et ajoute ensuite un à un les éléments descendants. La Fig.5.4 montre une structure hiérarchique créée avec l'éditeur de schémas Tamino : le nom de la collection est `Sample` et le nom de la racine (DOCTYPE) est `Telephone`.

Définition de la structure des données à partir d'une DTD Nous avons vu précédemment que les concepts de DTD et de schéma Tamino étaient assez similaires ; dans l'éditeur de schéma, il est possible de créer un schéma Tamino à partir d'une DTD sélectionnée dans le menu *File* > *Open DTD/XML...* (Fig.5.5). Nous justifions ainsi le choix fait lors de la conception logique d'aborder le problème de la génération d'une DTD plutôt que d'un XML Schema.

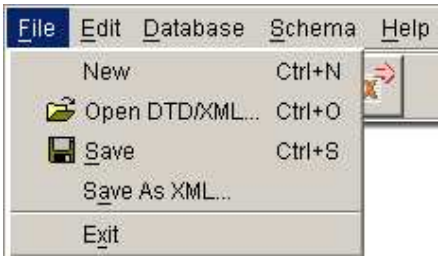

FIG. 5.5 – Le menu *File* de l'éditeur de schéma Tamino

FIG. 5.6 – La définition des propriétés physiques générales à la DTD

Définition des paramètres physiques

Quel que soit la manière dont la hiérarchie a été créée, la partie droite de l'éditeur affiche les propriétés de l'objet sélectionné dans la hiérarchie de gauche.

1. Si cet objet est la racine de l'arborescence (appelée DOCTYPE), les champs suivants sont à compléter (Fig.5.6) :
   - *Doctype Name :* le nom de la racine de ce schéma.
   - *Options :* le mode d'accès à la DTD. Les valeurs possibles sont READ (LECTURE), INSERT(INSERTION), UPDATE (MISE A JOUR) et/ou DELETE (SUPPRESSION).
   - *Structure Index :* la structure d'index. Les valeurs sont CONDENSED (CONDENSE) ou NO (AUCUNE).
2. Si cet objet est un élément ou un attribut, les informations à fournir sont les suivantes (Fig.5.7) :
   - *Object-Type :* le mode d'organisation de l'élément sélectionné. Les choix possibles sont PCDATA, EMPTY, CHOICE, SEQ, ANY ou CDATA (si on désire que l'élément prenne le statut d'attribut).
   - *Multiplicity :* l'opérateur de cardinalité appliqué à l'élément. Les choix possibles sont +, * ou ?.
   - *Search-Type :* le mode d'indexation. Les possibilités sont TEXT, STANDARD, les deux ou aucun des deux.
   - *Map-Type :* le type de «mapping». Parmis les choix possibles, on retrouve les valeurs décrites dans le chapitre précédent : NATIVE (NATIF), INFO FIELD (INFORMATION), SQL COLUMN (COLONNE SQL) ou NO (AUCUN).

FIG. 5.7 – La définition des propriétés physiques de l'élément `Address`

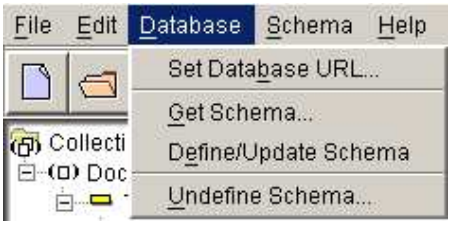

FIG. 5.8 – Le menu *Database* de l'éditeur de schéma Tamino

Définition du schéma dans une base de données

L'étape suivante consiste à exporter le schéma construit afin de définir une collection dans une base de données Tamino existante. Cela se fait très facilement en deux étapes, via le menu *Database* (Fig.5.8) de l'éditeur de schéma.

Préciser l'URL de la base de données (menu *Database* > *Set Database URL...*) L'URL d'une base de données se structure bien souvent de cette manière :
`« adresse du serveur » /tamino/ « nom de la base de données »`
Exemple : `http ://localhost/tamino/Order`

Définir la collection dans la base de données (menu *Database* > *Define/Update Schema*) Si la collection n'existe pas dans la base de données, elle est créée avec les informations du schéma en cours, sinon, elle est mise à jour.

5.2.3 L'interface interactive Tamino

L'interface interactive Tamino (Fig.5.9) est une application Javascript permettant de réaliser des opérations simples sur les données et les collections d'une base de données Tamino. Les résultats de ces opérations sont affichés dans la partie inférieure de l'interface sous la forme brute i.e. sous la forme du document XML généré par la base de données. Par exemple, une requête sur la base de données provoquera l'affichage d'un document XML contenant le ré-

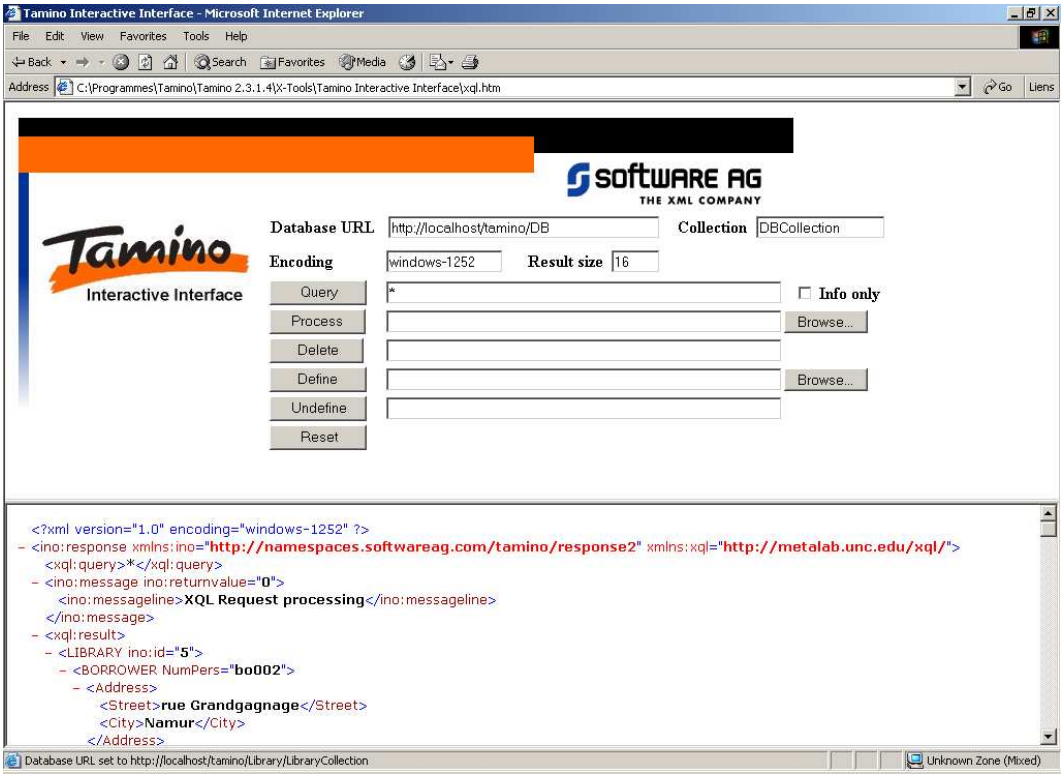

FIG. 5.9 – L'interface interactive Tamino

sultat de la requête ainsi que différentes méta-informations sur l'opération réalisée (messages d'erreurs, expression XQuery...). Pour cette raison, l'interface interactive sera rarement utilisée par l'utilisateur final de l'application ; elle peut par contre se révéler très utile pour le développeur d'applications qui souhaite tester le comportement de la base de données ou agir de manière simple et directe sur les documents XML offerts par le serveur Tamino.

Après avoir précisé l'URL de la base de données et la collection sur laquelle on désire travailler, il est possible de réaliser les opérations suivantes :

- **Query :** interroger la base de données en utilisant une expression XQuery.
- **Process :** importer un document XML dans la base de données.
- **Delete :** supprimer l'(les) élément(s) sélectionné(s) par une expression XQuery.
- **Define :** définir une nouvelle collection dans la base de données, à partir d'un fichier XML contenant le schéma Tamino.
- **Undefine :** supprimer une collection existante dans la base de données.

5.2.4 La technologie X-Machine

Développé à partir d'une nouvelle architecture spécialement conçue pour le langage XML, Tamino s'appuie sur la technologie X-Machine, qui est la première technologie de stockage de données capable de stocker des informations XML sans aucune conversion. Il est possible de dialoguer avec la X-Machine en utilisant des commandes X-Machine transportées par des requêtes HTTP de bas niveau. Grâce à ce principe, un client HTTP (browser internet) peut agir directement sur une base de données Tamino en envoyant des requêtes HTTP particulières. La X-Machine reçoit ces requêtes, les interprète, exécute la commande sur la base de données et renvoie le résultat au client sous la forme d'un document XML. Le navigateur internet peut afficher ce document XML de manière brute ou appliquer une transformation XSL au

document afin de présenter le résultat sous la forme d'un document HTML. Le chapitre suivant présente une application développée en Javascript qui réalise ces manipulations de manière très performante.

Syntaxe d'une commande X-Machine

La syntaxe générale d'une commande X-Machine est la suivante :

<URL prefix>/<collection>?<command>=<data>

où
<URL prefix> est l'URL d'une base de données Tamino
<collection> est une collection définie dans la base de données Tamino
<command> est le nom de la commande X-Machine à exécuter sur la base de données. Les valeurs possibles sont :

- **_admin :** exécute certaines fonctions d'administration de la base de données.
- **_commit :** effectue un commit d'une transaction.
- **_connect :** initialise une session avec une collection de la base de données.
- **_define :** crée une collection dans la base de données.
- **_delete :** supprime un (des) élément(s) dans la base de données.
- **_diagnose :** effectue des opérations de diagnostic de la couche HTTP.
- **_disconnect :** cloture une session avec la base de données.
- **_encoding :** spécifie l'encodage de caractères utilisé pour la requête.
- **_process :** importe un document XML dans la base de données.
- **_rollback :** effectue un rollback d'une transaction.
- **_undefine :** supprime une collection dans la base de données.
- **_xql :** interroge la base de données avec une expression XQuery.

Pour de plus amples informations sur ces différentes commandes, veuillez vous référer à la documentation fournie avec le serveur Tamino.
<data> est une expression XQuery qui précise l'(les) élément(s) concernés par la commande

Exemples de commandes X-Machine

Voici quelques exemples simples de commandes X-Machine. Ces commandes peuvent être directement encodées à la place d'une URL dans n'importe quel navigateur internet.

```
http://localhost/tamino/MyDB/CollectUser/?_connect=*
```

ouvre une session avec la collection `CollectUser` définie dans la base de données `MyDB`.

```
http://localhost/tamino/Library/LibraryCollection/?_xql=*
```

affiche le contenu complet de la collection `LibraryCollection` définie dans la base de données `Library`.

```
http://localhost/tamino/HospitalDB/Hospital?_delete=patient[@IdPat="4711"]
```

supprime – dans la collection `Hospital` définie pour la base de données `HospitalDB` – l'élément `patient` dont l'attribut `IdPat` vaut «4711».

5.2.5 Les API Java

Une base de données Tamino peut être exploitée par des clients HTTP de natures différentes : outre la possibilité d'utiliser un navigateur internet, il est possible de développer des clients Java, Javascript ou ActiveX en utilisant les librairies de fonctions et d'objets (API) fournies avec Tamino. Dans ce chapitre, nous présentons brièvement les principaux objets Java qui nous permettront dans le chapitre suivant d'implémenter un client Tamino simple et performant.

L'objet *TaminoClient*

L'objet *TaminoClient* permet la connexion à une base de données Tamino dont on spécifie l'URL dans le constructeur. Grâce aux méthodes query, process, insert, update et delete implémentées par cet objet, il est ainsi possible d'exploiter une base de données sans aucune connaissance du protocole HTTP sous-jacent.

L'objet *TaminoResult*

Les méthodes query, process, insert, update et delete de l'interface *TaminoClient* renvoient chacune un objet *TaminoResult*. Cet objet est un objet DOM qui représente le document XML contenant le résultat de l'opération demandée. En utilisant un parser DOM [2], un objet *TaminoResult* peut facilement être parsé afin d'obtenir les informations désirées et les stocker dans des variables Java. Une autre solution est d'utiliser un processeur XSL, comme Xalan[Pro], pour générer des pages HTML à partir de l'objet *TaminoResult*.

L'objet *TaminoError*

Toute interaction avec une base de données Tamino est susceptible de générer des erreurs. Ces erreurs sont représentées par un objet *TaminoError*.

Exemple de client Java pour le serveur Tamino

L'exemple de code Java ci-dessous nous montre comment utiliser ces objets Java pour interroger une base de données Tamino :

```
        import java.util.*;
        import org.w3c.dom.*;
        import com.softwareag.tamino.api.dom.*;

        public class DemoQuery
        {
                public static final void main(String arg[]) throws Exception
                {
(01):           TaminoClient tamino = new TaminoClient("localhost/tamino/
      MyTaminoDB/PhoneBook");
(02):           tamino.setPageSize(5);
(03):           tamino.startSession();
                        try
                        {
```

[2] Le parser DOM recommandé par Software AG est le parser de docuverse, disponible sur le site [Doc]

```
(04):                   TaminoResult tr = tamino.query("Telephone[LoginName="Dupont"
      );
(05):                   Enumeration e = tr;
                                int i = 0;
(06):                   while( e.hasMoreElements())
                                {
                                        Element el=(Element)e.nextElement();
                                        TaminoClient.printTree(el);
                                }
(07):                   tamino.endSession();
                        }
(08):           catch(TaminoError te)
                        {
                                System.out.println(te.getMessage());
                        }
                }
        }
```

Annotations :
(01) : création d'un objet *TaminoClient*. Le constructeur accepte un seul argument : l'URL de la base de données + la collection
(02) : le nombre maximum d'éléments que le serveur peut renvoyer à chaque requête vaut 5
(03) : ouverture d'une session avec la base de données
(04) : envoi d'une requête à la collection `PhoneBook` définie dans la base de données se trouvant à l'adresse `localhost/tamino/MyTaminoDB`. La requête demande tous les éléments `Telephone` dont un élément `LoginName` contient la valeur `Dupont`. Le résultat est un document XML stocké sous la forme d'un objet *TaminoResult*.
(05) : l'objet *TaminoResult* est transformé en objet *Enumeration* afin de pouvoir parcourir ses fils un à un
(06) : boucle *while* qui parcourt les résultats un à un et imprime l'arborescence des éléments `Telephone` recherchés dans la requête
(07) : fin de session avec la base de données
(08) : en cas de problème de connexion avec la base de données, une erreur *TaminoError* est générée et le message associé est imprimé à l'écran

Chapitre 6

Etude de Cas

Dans ce chapitre, nous illustrons, avec un exemple concret, le processus complet de conception d'une base de données XML. Nous commençons par décrire, dans un énoncé en français, la problématique. Ensuite, nous utilisons l'outil case DB-Main pour modéliser la situation dans un schéma conceptuel conforme au modèle GER. Dans l'étape suivante de conception logique, nous montrons comment utiliser les outils présentés au chapitre 2 pour obtenir un schéma conforme au modèle XML défini. Finalement, à partir de la DTD générée lors de la conception logique, nous définissons une collection dans une base de données Tamino et montrons comment importer et interroger des données XML grâce au serveur Tamino.

6.1 Enoncé de la situation

Une bibliothèque désire informatiser la gestion de ses emprunts. Elle décide alors de créer une base de données XML qui lui fournira, en temps réel, l'information sur les documents disponibles, les auteurs de ces documents, les emprunteurs et les projets justifiant les emprunts. Un document dans la bibliothèque est caractérisé par un numéro identifiant, un titre, une date de publication et, éventuellement, plusieurs mots-clés (maximum 10). Un document peut être écrit par plusieurs auteurs pour lesquels on ne retient que le nom et le prénom. Un document est soit un rapport, soit un livre. Un rapport est identifié par un code. Seuls les livres peuvent être empruntés car ils se déclinent en plusieurs exemplaires. Un livre possède un numéro ISBN identifiant et un éditeur. Un emprunteur est identifié par un numéro et on décide également de mémoriser son nom, son adresse (rue et ville, plus précisément), ainsi que son (ses) numéro(s) de téléphone. Avant d'emprunter l'exemplaire d'un livre, un emprunteur peut le réserver pour une certaine date. L'emprunt se fait alors à une date postérieure et une date de retour peut être précisée directement par l'emprunteur. Un emprunt se fait toujours dans le cadre d'un projet rassemblant un ou plusieurs emprunteurs et caractérisé par un code identifiant, un nom et un budget. Un exemplaire est acquis à une certaine date et peut être facilement retrouvé grâce à sa localisation dans la bibliothèque (travée, rayon et étage).

6.2 Schéma conceptuel

Présenté Fig.6.1, le schéma conceptuel modélisant la situation est de taille raisonnable mais possède quelques caractéristiques intéressantes à étudier lors du processus de transformation :

- une relation IS-A de disjonction entre un sur-type (`DOCUMENT`) et deux sous-types (`REPORT` et `BOOK`).
- trois types d'associations N à N (`writes`, `reserves` et `borrows`).
- deux types d'associations possédant des attributs (`reserves` et `borrows`).

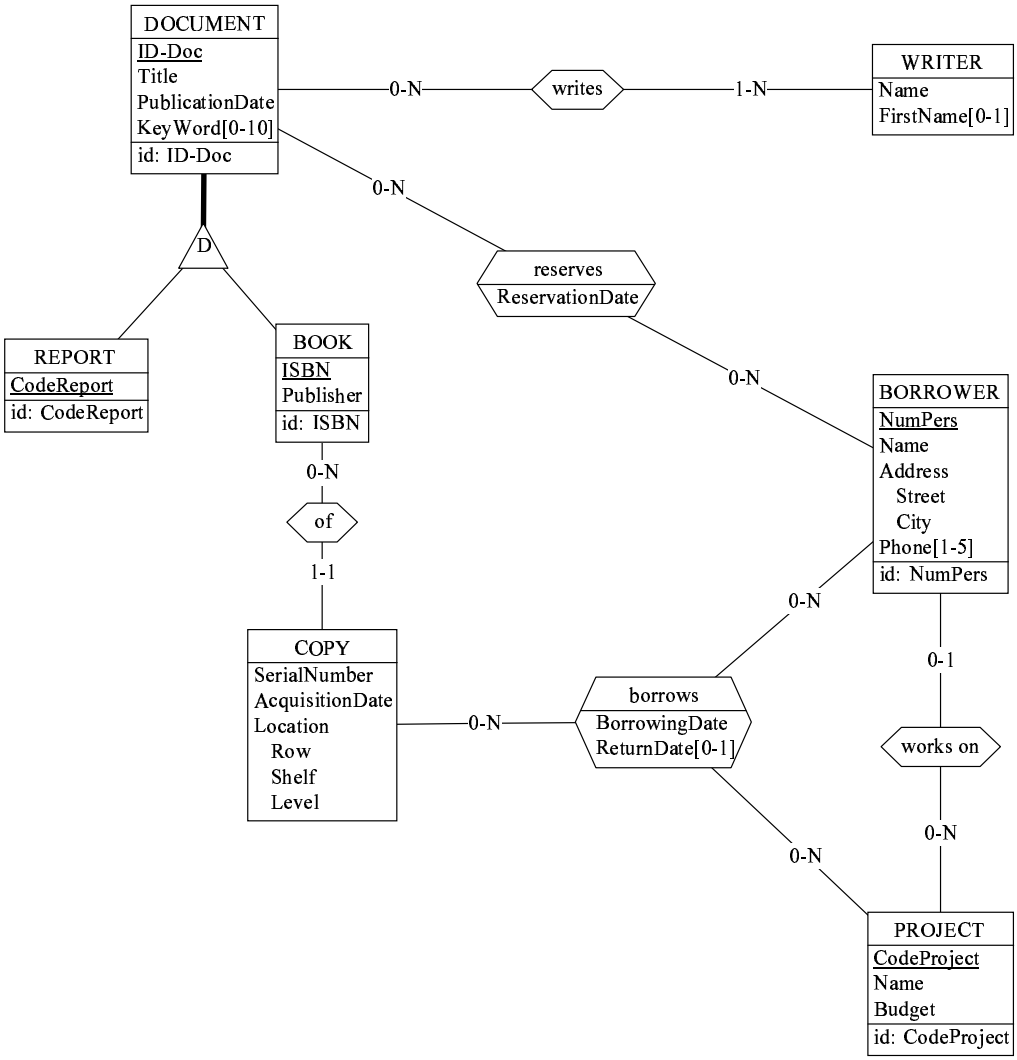

FIG. 6.1 – Le schéma conceptuel modélisant la problématique de la Bibliothèque

- un type d'associations ternaire (`borrows`).
- un type d'associations de cardinalités [0-N,0-1] (`works on`).
- deux attributs composés (`BORROWER.Address` et `COPY.Location`).
- deux attributs multivalués (`DOCUMENT.KeyWord` et `BORROWER.Phone`).
- deux types d'entités dépourvus d'identifiants primaires (`COPY` et `WRITER`).
- un attribut nommé `Name` dans trois types d'entités différents : `WRITER`, `PROJECT` et `BORROWER`.

6.3 Conception logique

Dans cette étape, nous utilisons les outils présentés dans la section 3.5, page 63 pour transformer le schéma conceptuel en un schéma logique conforme au modèle XML, à partir duquel la DTD sera automatiquement générée.

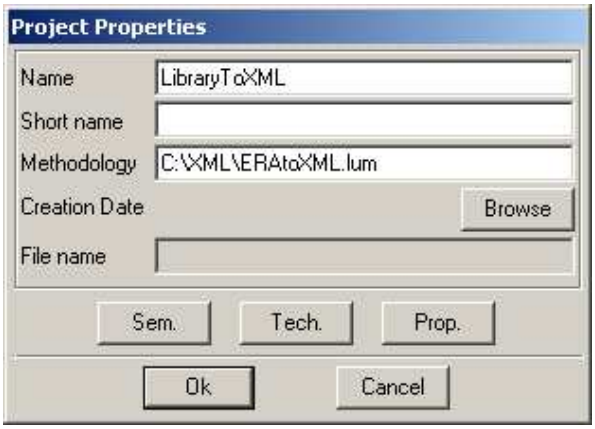

FIG. 6.2 – La fenêtre de création de projet DB-Main

6.3.1 La transformation de schéma

Création d'un nouveau projet dans DB-Main

1. Créer un nouveau projet DB-Main en cliquant sur l'option *New project...* du menu *File*.
2. La fenêtre présentée Fig.6.2 apparaît. Choisir un nom pour le projet et cliquer sur le bouton *Browse* pour sélectionner le fichier `ERAtoXML.lum` contenant la méthodologie.

Résultat : La méthodologie (processus et produits présentés Fig.3.40 page 65) apparaît dans la fenêtre principale. Une autre fenêtre affiche l'historique des processus exécutés et des produits mis à jour.

Le processus «IMPORT»

1. Exécuter le processus «IMPORT» [1].
2. Sélectionner le fichier .isl généré à partir du schéma conceptuel précédemment construit.

Résultat : Le schéma conceptuel `LIBRARY` est importé dans le projet en cours, son icône apparaît dans la fenêtre d'historique du projet.

Le processus «COPY»

1. Exécuter le processus «COPY».
2. Dans la fenêtre présentée Fig.6.3, choisir un nom et la version du nouveau schéma de travail.

Résultat : Une icône représentant le nouveau schéma de travail apparaît dans la fenêtre d'historique du projet.

Le processus Transform IS-A, Complex Relation-Types, Composed and Multi-Value Attributes

1. Exécuter le processus Transform IS-A, Complex Relation-Types, Composed and Multi-Value Attributes.

Résultat : Les structures non-conformes ont été transformées dans le schéma de travail (Fig.6.4). Quelques constatations intéressantes sont à faire :

[1] cliquer avec le bouton droit sur le processus et sélectionner l'option *Execute*

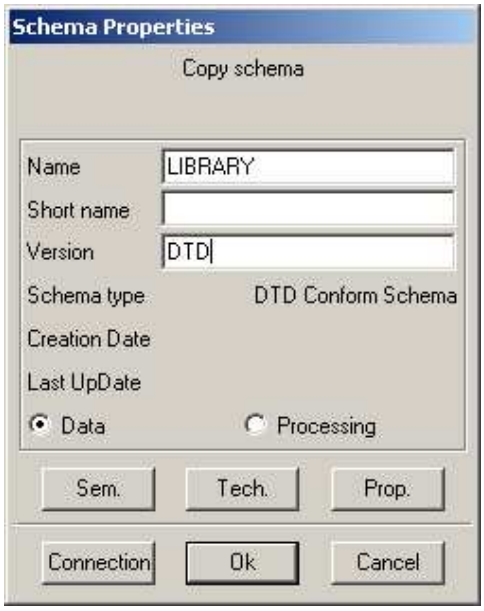

FIG. 6.3 – La fenêtre de copie de schéma

- la transformation des types d'associations N à N en types d'entités fait apparaître des groupes identifiants composés des rôles joués par les types d'entités concernés par le type d'associations initial. Ces identifiants ne peuvent être traduits dans le modèle XML et devront donc disparaître dans le schéma final.
- la transformation des attributs composés (`BORROWER.Address` et `COPY.Location`) en types d'entités fait apparaître des types d'associations de cardinalités [1-1,1-1] (`BOR_Add` et `COP_Loc`). Une orientation père-fils devra être décidée pour ces types d'associations dans l'étape suivante.

Le processus «Create Hierarchy»

1. Exécuter le processus «Create Hierarchy».
2. Choisir le sens des types d'associations [1-1,1-1] apparus lors du processus précédent :
   - pour le type d'associations `BOR_Add`, l'option la plus logique est `de BORROWER vers Adress` (Fig.6.5).
   - pour le type d'associations `COP_Loc`, l'option la plus logique est `de COPY vers Location` (Fig.6.6).
3. Les racines obligatoires sont automatiquement déduites du schéma et leurs noms apparaissent dans une boîte de dialogue (Fig.6.7). Il s'agit des types d'entités `BORROWER`, `DOCUMENT`, `PROJECT` et `WRITER`.
4. Si d'autres concepts importants du schéma méritent le statut de racine, ils peuvent être sélectionnés dans la boîte de choix suivante. On décide que le type d'entités `COPY` peut également être une racine (facultative) (Fig.6.8).
5. Lorsque le choix des racines facultatives est terminé, cliquer sur *Annuler*.

Résultat intermédiaire : La Fig.6.9 illustre l'état du schéma après l'élection des racines (obligatoires et facultatives). On y observe les propriétés suivantes :
- Les racines (obligatoires et facultatives) du schéma sont mises en évidence par un marquage en **gras**.

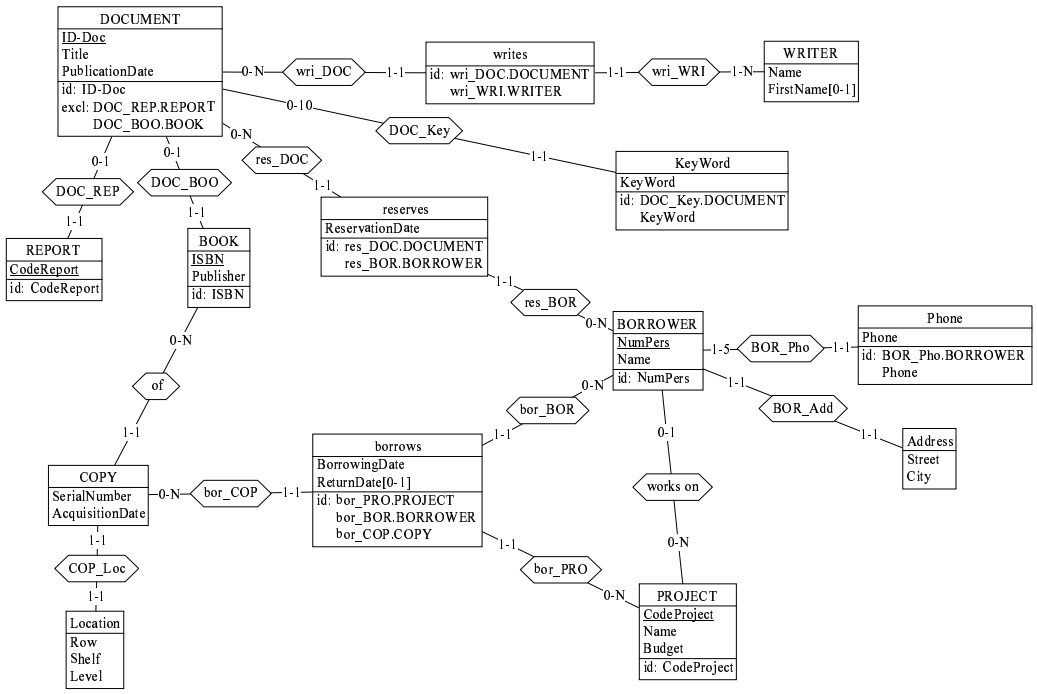

FIG. 6.4 – Le schéma de travail après l'étape de transformation des relations IS-A, des types d'associations complexes, des attributs composés et multivalués

FIG. 6.5 – Le choix de l'orientation du type d'associations `BOR_Add`

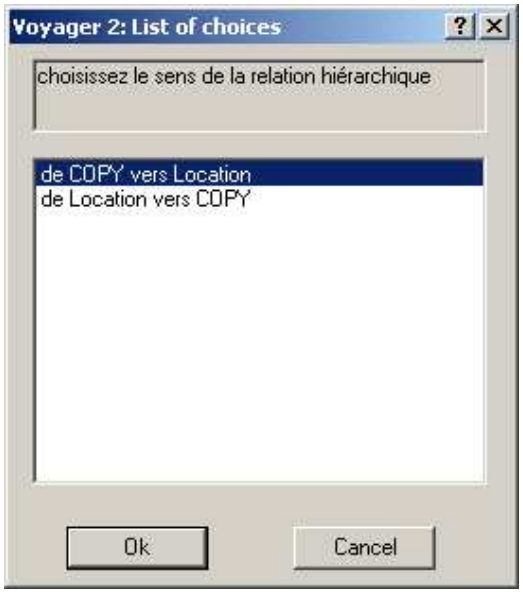

FIG. 6.6 – Le choix de l'orientation du type d'associations `COP_Loc`

FIG. 6.7 – Les racines obligatoires sont automatiquement déduites

FIG. 6.8 – Le type d'entités `COPY` mérite aussi le statut de racine (facultative)

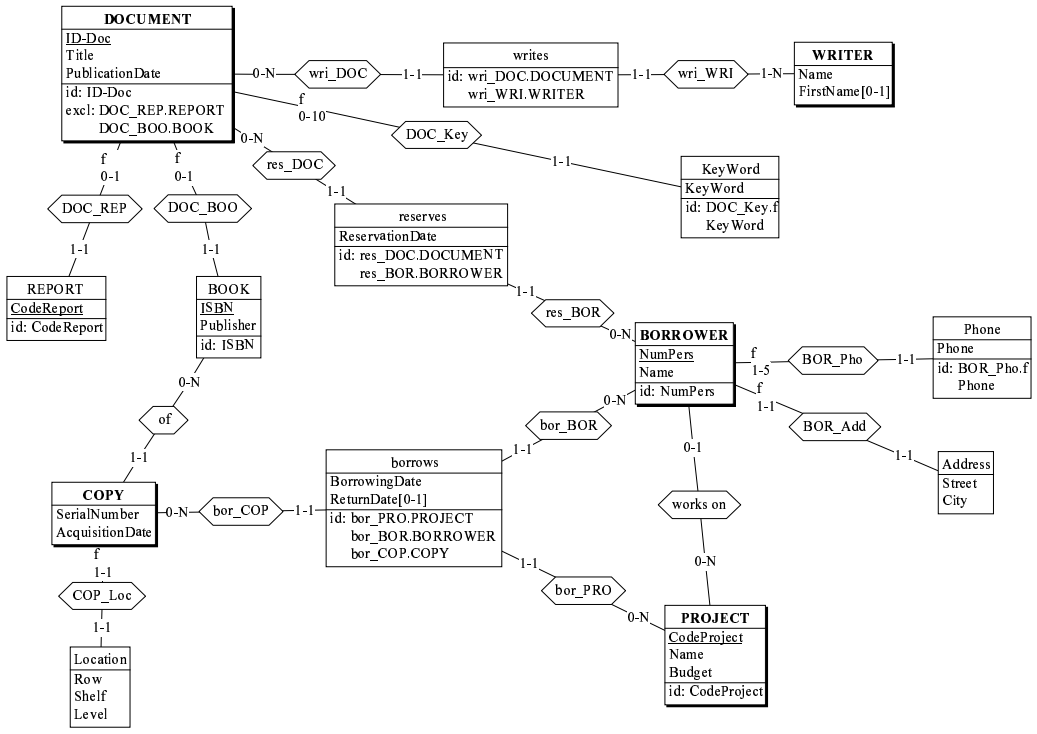

FIG. 6.9 – Le schéma après l'élection des racines (obligatoires et facultatives)

- La nature (hiérarchique ou référentielle) de certains types d'associations a été décidée. Les types d'associations dont un rôle est nommé «f» sont de nature hiérarchique : il s'agit des types d'associations issus de la relation IS-A (`DOC_REP` et `DOC_BOO`) et de ceux résultant de la transformation en types d'entités des attributs multivalués et composés (`DOC_Key`, `COP_Loc`, `BOR_Pho` et `BOR_Add`). Les types d'associations `works on` et `of` sont de nature référentielle mais cela n'apparaît pas sur le schéma. Dans le cas du premier, cela est dû à ses cardinalités [0-N,0-1] (voir section 3.4.1, page 41). La nature du type d'associations `of`, par contre, a été décidée suite à l'élection de `COPY` comme racine facultative (voir «La mise en évidence de racines facultatives», page 46).
- La nature des types d'associations impliqués dans un conflit n'a pas encore été décidée.

6. Les types d'entités `borrows`, `reserves` et `writes` possèdent plusieurs pères potentiels. Il faut, par conséquent, résoudre ces conflits un à un :
   - le type d'entités `borrows` possède trois pères potentiels : `BORROWER`, `COPY` et `PROJECT`. Il semble plus naturel d'associer l'action d'emprunter (`borrows`) au type d'entités qui réalise l'action i.e. le type d'entités représentant l'emprunteur (`BORROWER`). Ce choix est illustré Fig.6.10.
   - le type d'entités `reserves` possède deux pères potentiels : `BORROWER` et `DOCUMENT`. Pour la même raison que le conflit précédent, il semble plus naturel de choisir le type d'entités `BORROWER` comme père du type d'entités `reserves`. Ce choix est illustré Fig.6.11.
   - le type d'entités `writes` possède deux pères potentiels : `WRITER` et `DOCUMENT`. Nous décidons d'associer l'action d'écrire (`writes`) au type d'entités `WRITER`. Ce choix est illustré Fig.6.12.

7. Un message apparaît pour signaler à l'utilisateur qu'un identifiant technique a été ajouté au type d'entités `COPY` (Fig.6.13). L'explication est simple : le type d'entités `COPY`, n'ayant pas été choisi comme père pour le type d'entités `borrows`, va devenir la cible d'une re-

FIG. 6.10 – Choix d'un père pour le type d'entités `borrows` : le type d'entités `BORROWER`

FIG. 6.11 – Choix d'un père pour le type d'entités `reserves` : le type d'entités `BORROWER`

FIG. 6.12 – Choix d'un père pour le type d'entités `writes` : le type d'entités `WRITER`

FIG. 6.13 – Ajout d'un identifiant technique au type d'entités `COPY`

lation de référence. Or ce type d'entités ne possède pas d'identifiant primaire simple ; ce qui est une condition pour tout type d'entités référencé par une relation de référence.

Résultat intermédiaire : La Fig.6.14 illustre l'état du schéma après l'étape de résolution des conflits. On y observe les propriétés suivantes :
- La nature de tous les types d'associations a été décidée. Les types d'associations dont un des rôles est nommé «f» sont de nature hiérarchique alors que tous les autres sont de nature référentielle.
- Un identifiant technique a été ajouté au type d'entités `COPY`.

8. Un message apparaît pour signaler qu'une racine technique a été ajoutée au schéma (Fig.6.15). Ce type d'entités prend le nom du schéma en cours (`LIBRARY`) et devient le père des racines naturelles : `DOCUMENT`, `WRITER`, `PROJECT`, `BORROWER` et `COPY`.

Résultat : Une racine technique est ajoutée au schéma et les types d'entités sont réorganisés afin de mettre en évidence la structure hiérarchique ainsi créée : la racine technique `LIBRARY` est située en haut à gauche alors que les racines naturelles (`DOCUMENT`, `WRITER`, `PROJECT`, `BORROWER` et `COPY`) apparaissent l'une en dessous de l'autre, directement à droite de `LIBRARY`.

Le processus «Draw References»

1. Exécuter le processus «Draw References».

Résultat : Les types d'associations référentiels sont transformés en clés étrangères dont l'origine est un groupe «ref» contenant un attribut de référence et la destination l'attribut identi-

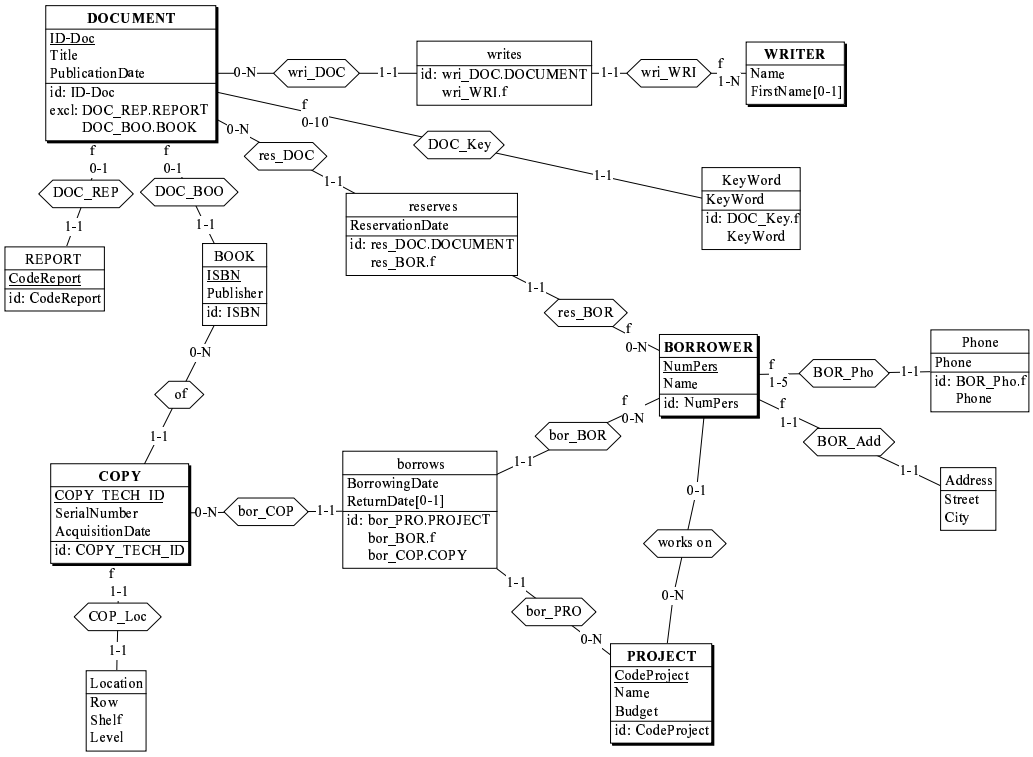

FIG. 6.14 – Le schéma après l'étape de résolution des conflits

FIG. 6.15 – Un type d'entités (racine technique) nommé `LIBRARY` est ajouté au schéma

fiant le type d'entités référencé (Fig.6.16).

Le processus «CreateGroupsAndAttributes»

1. Exécuter le processus «CreateGroupsAndAttributes».
2. Choisir le mode de représentation des attributs du schéma qui ne sont pas impliqués dans un groupe. Nous choisissons de les transformer en types d'éléments XML selon la règle de **Fusion** des attributs de même nom en un seul type d'entités. (Fig.6.17).
3. Les groupes qui ne peuvent être représentés dans le modèle XML sont affichés dans une boîte de dialogue.

Résultat :

- Les groupes «id» contenant un seul attribut sont transformés en groupes «gid». Exemple : le groupe «id» constitué de l'attribut `BORROWER.NumPers`.
- Les groupes «ref» sont transformés en groupes «idref».
  Exemple : le groupe «ref» constitué de l'attribut `borrows.CodeProject`.
- Les attributs qui ne sont pas impliqués dans un groupe sont transformés en types d'entités dont le nom est le nom de l'attribut. Ce type d'entités possède un seul attribut, nommé #pcdata.
  Exemple : l'attribut `DOCUMENT.Title` est transformé en type d'entités `Title` (par représentation des instances). Ce type d'entités devient le fils du type d'entités `DOCUMENT` et possède un seul attribut nommé #pcdata.
- La transformation des attributs `WRITER.Name`, `PROJECT.Name` et `BORROWER.Name` en type d'entités est particulière. En effet, dans ce cas, plusieurs attributs possèdent le même nom : un seul type d'entités, nommé `Name`, est alors créé mais un groupe «exact-1» rassemble les rôles pères joués par `WRITER`, `PROJECT` et `BORROWER`.
- Le groupe «excl» issu de la transformation de la relation IS-A est transformé en type d'entités technique `GR_0`. Ce type d'entités contient un groupe «choice» constitué des rôles joués par les sous-types `REPORT` et `BOOK`.
- Un groupe «seq» est créé dans les types d'entités possédant plusieurs fils. C'est notamment le cas de la racine technique `LIBRARY` dont le groupe «seq» rassemble les rôles fils joués par les racines naturelles.

Le schéma final conforme au modèle XML est présenté Fig.6.18.

6.3.2 La validation du schéma logique XML

1. Exécuter le module d'analyse de schéma, accessible dans DB-Main via le menu *Assist* > *Schema Analysis...* (Fig.6.19).
2. Charger la librairie de prédicats XML : cliquer sur *Edit library*, puis sur *Load library* et sélectionner le fichier `XML.anl`.
3. Cliquer sur *Close* pour fermer la fenêtre en cours.
4. Charger le script de validation XML : cliquer sur *Load* et sélectionner le fichier `XML.ana`.
5. Cliquer sur *OK* pour exécuter le script de validation de schéma.

Résultat : Un message indique si le schéma vérifie tous les prédicats du script de validation (Fig.6.20).

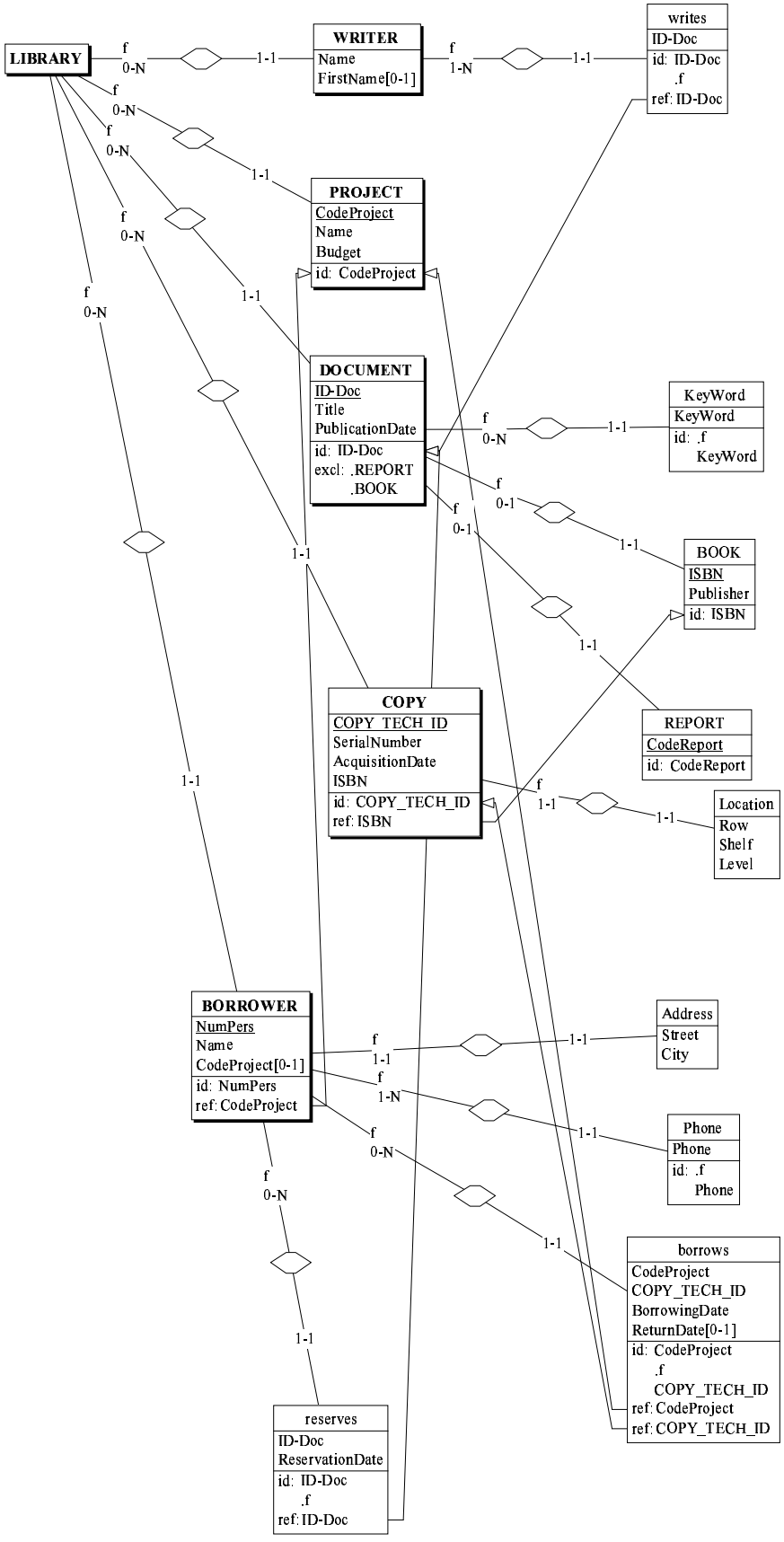

FIG. 6.16 – Les types d'associations référentiels sont transformés en clés étrangères. Des groupes «ref» apparaissent alors dans le schéma

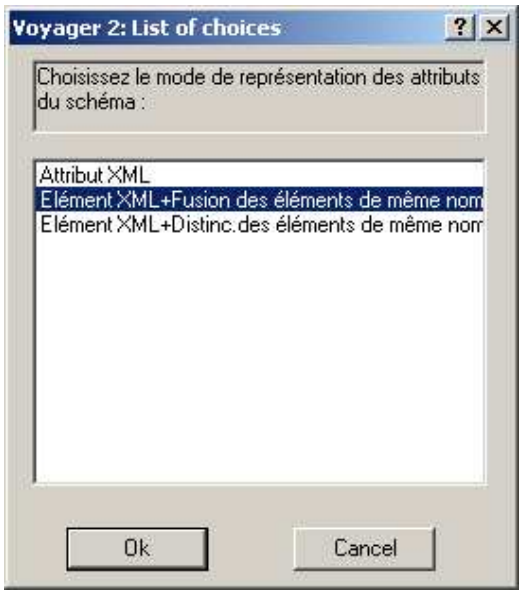

FIG. 6.17 – Les attributs qui ne sont pas impliqués dans un groupe deviennent des types d'éléments XML. Les attributs de même nom sont fusionnés dans un seul type d'entités

6.4 Conception physique

6.4.1 Propriétés physiques générales d'une DTD

Le mode d'accès à la DTD

On décide de donner un accès complet aux données XML du système. Le mode d'accès est donc «LECTURE INSERTION MISE A JOUR SUPPRESSION».

La structure d'index

Aucun élément du schéma ne possède un type de contenu mixte (ANY). Cela signifie que le contenu de tous les éléments est défini de manière stricte. On peut donc raisonnablement assigner la valeur AUCUNE à ce paramètre.

6.4.2 Propriétés physiques propres à un élément / attribut XML

Le mode d'indexation

Beaucoup de requêtes recherchent une occurence particulière d'un type d'éléments sur base de sa valeur (connue) identifiante. On décide donc de placer un index de type STANDARD (comparaison de valeurs sans «wild-cards») sur tous les attributs appartenant à un groupe «gid». Il s'agit des attributs `BORROWER.NumPers`, `COPY.COPY_TECH_ID`, `DOCUMENT.ID-Doc`, `BOOK.ISBN`, `REPORT.CodeReport` et `PROJECT.CodeProject`.

Le type de «mapping»

En règle générale, les types d'éléments dont le nom représente une action et pas un concept concret ont peu de signification lorsqu'ils sont considérés seuls. C'est par exemple le cas du type d'éléments `borrows` qui se doit d'être accompagné de ses éléments fils `BorrowingDate` et `ReturnDate` pour acquérir une certaine sémantique. Conformément à cette hypothèse, nous

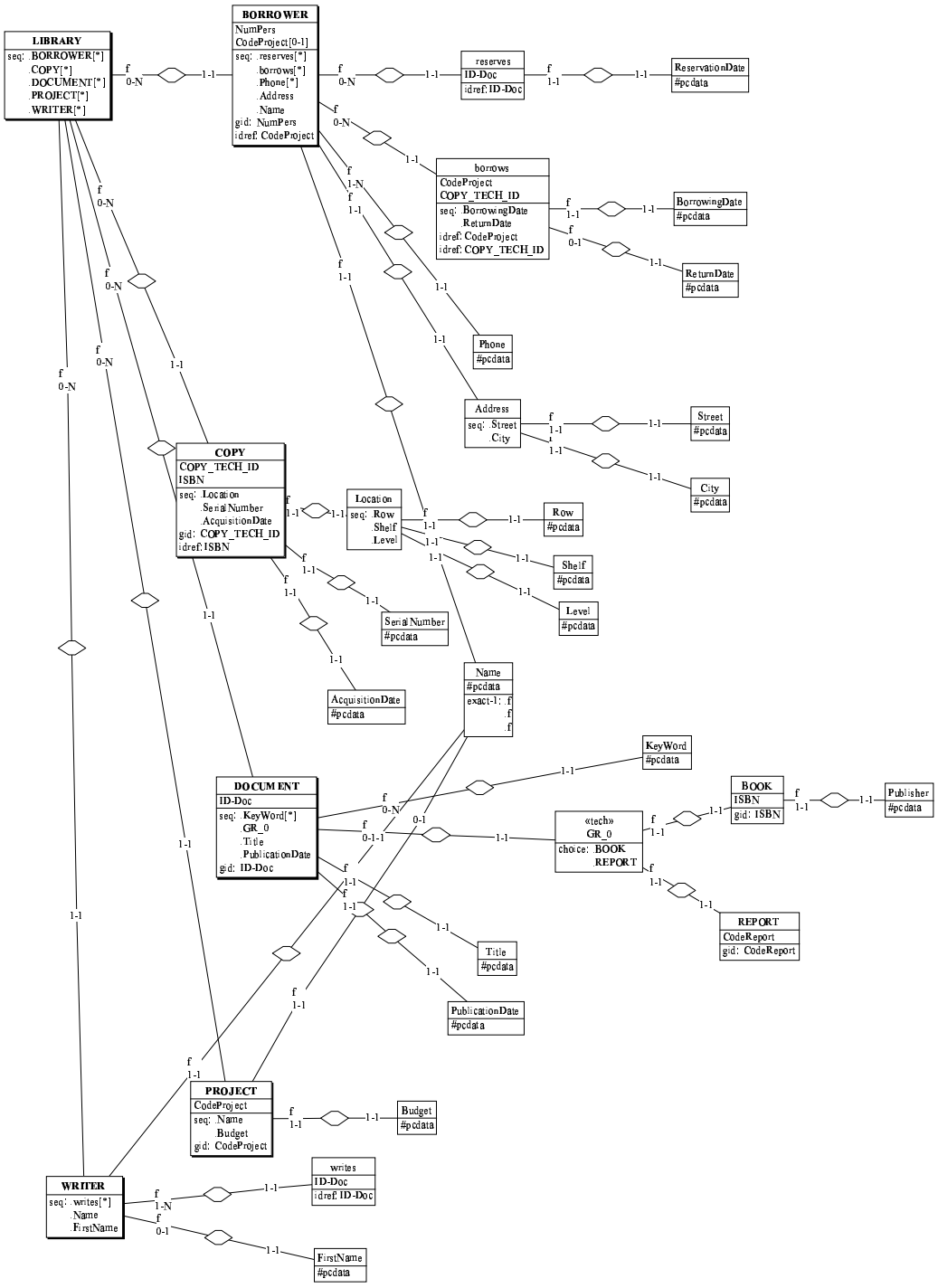

FIG. 6.18 – Le schéma conforme au modèle XML

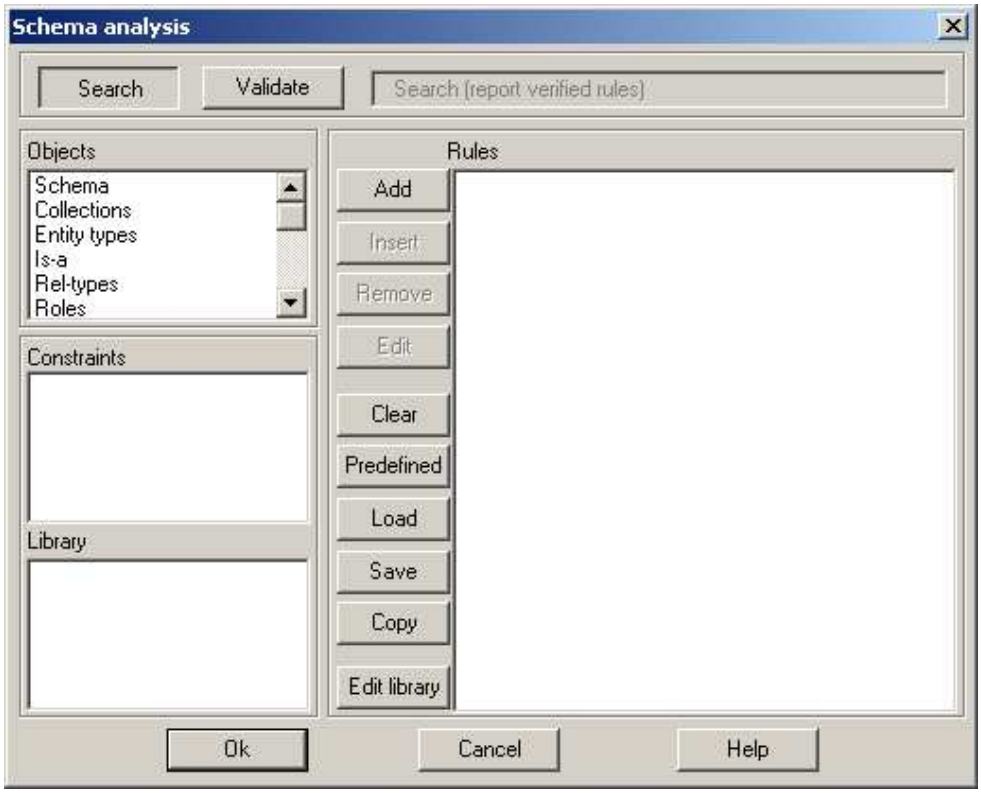

FIG. 6.19 – L'interface de l'analyseur de schéma dans DB-Main (*Assist > Schema Analysis...*)

FIG. 6.20 – Le message indiquant que le schéma est conforme au modèle XML

FIG. 6.21 – Un message informe l'utilisateur que la génération de la DTD est terminée

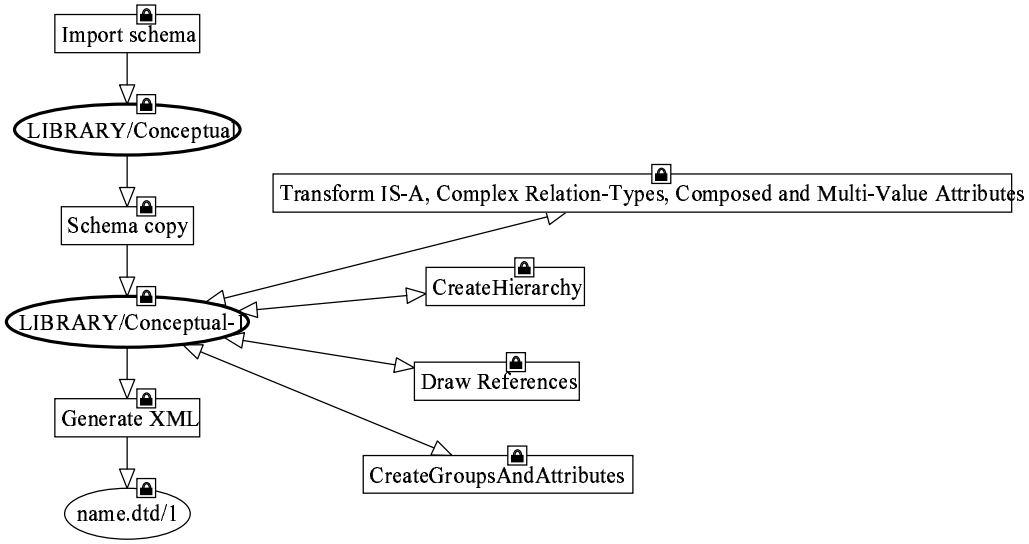

FIG. 6.22 – La fenêtre de l'historique du projet, à la fin de l'étape de conception logique

décidons d'associer un type de «mapping» NATIF aux types d'éléments `reserves`, `borrows` et `writes`.

6.5 Codage : génération du code de la DTD

1. Ouvrir le schéma logique XML obtenu lors de l'étape de conception logique.
2. Exécuter le générateur de DTD, accessible dans DB-Main via le menu *File* > *Generate* > *XML...*
3. Dans la boîte de dialogue *Save DTD Generation As...*, choisir le nom et l'emplacement du fichier DTD à générer.
4. Un message apparaît pour signaler la fin de l'opération (Fig.6.21).

Résultat : Un fichier DTD, correspondant au schéma XML en cours, est créé à l'emplacement choisi par l'utilisateur. Ce même fichier apparaît sous la forme d'un ovale (produit) dans la fenêtre détaillant l'historique du projet (Fig.6.22).

Le code de la DTD correspondant au schéma logique XML présenté Fig.6.18 se trouve en Annexe A.

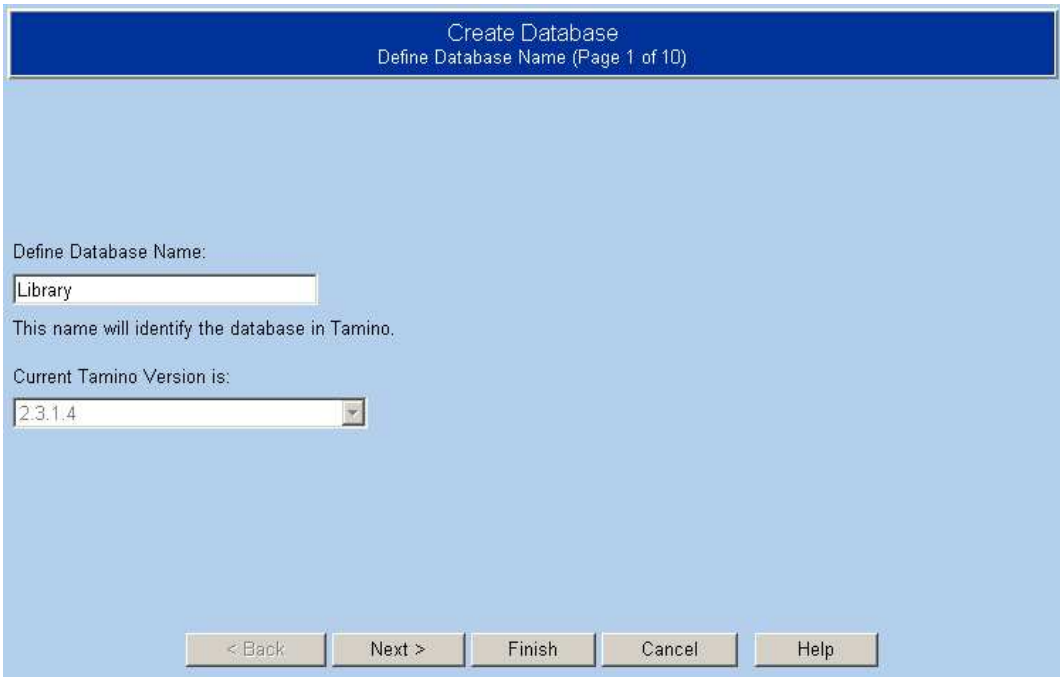

FIG. 6.23 – La création de la base de données `Library` avec la System Hub Manager

6.6 Implantation dans Tamino

6.6.1 Création et activation de la base de données `Library` avec le System Hub Manager

1. Lancer le System Hub Manager et s'identifier.
2. Dans le menu de navigation, sélectionner le menu *Databases* (voir la section 5.2.1, page 78) et cliquer sur le bouton *Create Database*.
3. Dans la partie droite de l'écran, préciser le nom de la base de données. Dans notre exemple, nous l'appelerons `Library` (Fig.6.23).
4. Cliquer sur le bouton *Finish* ; les autres paramètres de création de la base de données (localisations, profiles, taille des espaces physiques) seront alors fixés avec des valeurs par défaut. Le détail du processus de création de la base de données s'affiche dans la partie droite de l'écran (Fig.6.24) et un menu nommé `Library` apparaît dans le menu de navigation.
5. Cliquer sur le menu `Library` et activer la base de données en cliquant sur le bouton *Start Database*. Le feu de signalisation associé à la base de données `Library` passe au vert.

Résultat : La base de données est accessible à l'URL `http ://localhost/tamino/Library/` mais une collection conforme à la DTD `Library` n'a pas encore été définie dans la base de données `Library`.

6.6.2 Création du schéma Tamino et définition d'une collection dans la base de données `Library`

1. Lancer l'éditeur de schémas Tamino.
2. Importer la DTD générée lors de l'étape de conception logique en sélectionnant le fichier DTD dans le menu *File > Open DTD/XML...*

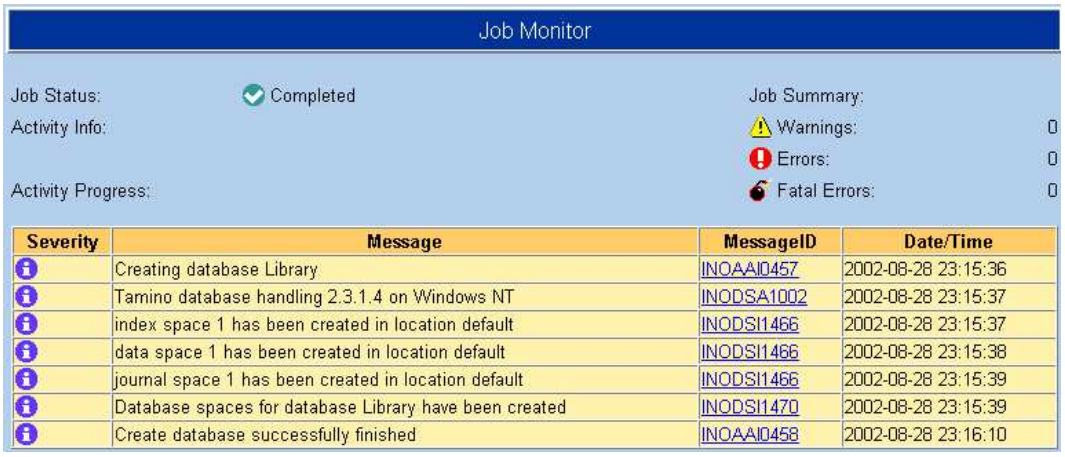

FIG. 6.24 – Le détail du processus de création d'une base de données Tamino

3. Choisir l'élément `LIBRARY` comme élément racine. La structure hiérarchique de la DTD apparaît dans la partie gauche de l'interface (Fig.6.25).
4. Dans le premier noeud de la structure hiérarchique, spécifier un nom pour la collection définie par la DTD. Dans notre exemple, il s'agit de `LibraryCollection`.
5. Cliquer sur l'élément *Doctype : LIBRARY* de la hiérarchie et spécifier les valeurs «READ INSERT UPDATE DELETE» et NO pour les champs *Options* et *Structure Index* (Fig.6.26).
6. Pour chaque attribut appartenant à un groupe «gid» dans le schéma logique : cocher l'option *standard* dans le cadre *Search-Type*.
7. Pour les types d'éléments `borrows`, `reserves` et `writes` : sélectionner le choix *Native* dans la liste déroulante *Map-Type*.
8. Sauver la structure hiérarchique sous la forme d'un schéma Tamino (fichier .xml) en sélectionnant le menu *File > Save As XML...* Le schéma Tamino généré à partir de la DTD `Library` se trouve en Annexe C.
9. Vérifier que l'URL de la base de données `Library` (`http ://localhost/tamino/Library`) est correcte dans le menu *Database > Set Database URL...*
10. Sélectionner le menu *Database > Define/Update Schema* pour définir la collection nommée `LibraryCollection` dans la base de données `Library`.

Résultat : Le schéma Tamino définissant la structure des données et les propriétés physiques pour la collection `LibraryCollection` est créé sous forme d'un document XML. La collection `LibraryCollection` est créée dans la base de données `Library`.

6.6.3 Importation de données XML dans la base de données

Les données à importer dans la base de données `Library` se trouvent dans un fichier XML conforme à la DTD Library. Le contenu de ce fichier se trouve en Annexe B

1. Lancer l'interface interactive Tamino.
2. Spécifier l'URL de la base de données (`http ://localhost/tamino/Library`) et le nom de la collection (`LibraryCollection`) dans les champs *Database URL* et *Collection*.
3. Cliquer sur le bouton *Parcourir* situé sur la même ligne que le bouton *Process* et sélectionner le fichier XML contenant des données à importer. Fig.6.27) présente l'interface interactive Tamino juste avant l'importation des données.

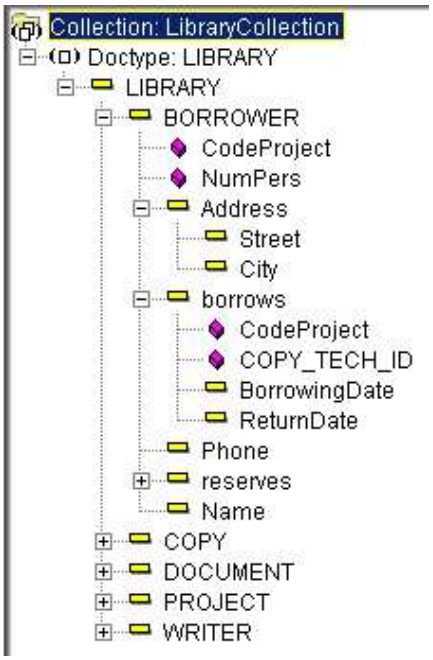

FIG. 6.25 – Lors de l'importation d'une DTD dans l'éditeur de schémas Tamino, la structure hiérarchique de la DTD apparaît graphiquement dans l'interface

FIG. 6.26 – Les propriétés physiques du schéma Tamino

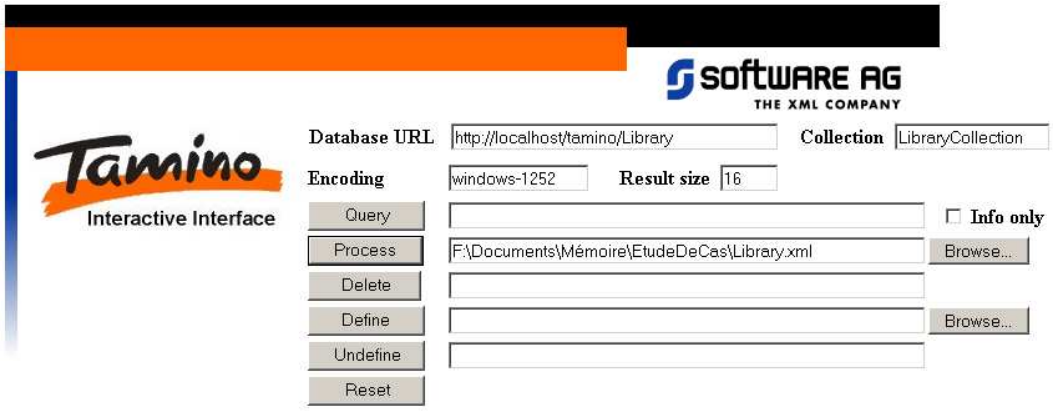

FIG. 6.27 – L'importation des données dans la base de données `Library` via l'interface interactive Tamino

4. Cliquer sur le bouton *Process* pour importer les données.

Résultat : Des données fictives sont importées dans la base de données `Library`.

6.7 Interrogation des données XML

Dans le chapitre précédent, nous avons présenté trois manières d'utiliser un client HTTP pour interroger une collection définie dans une base de données Tamino :

1. via le bouton *Query* de l'interface interactive Tamino.
2. via la commande X-Machine «xql» insérée dans l'URL d'un navigateur internet.
3. via la méthode «query» de l'objet Java TaminoClient.

Dans cette section, nous interrogeons la collection `LibraryCollection` en utilisant chacune de ces trois manières de faire. Les données XML interrogées correspondent aux données importées lors de l'étape précédente.

6.7.1 L'interface interactive Tamino

Pour interroger une collection avec l'interface interactive, il suffit de :

1. Lancer l'interface interactive Tamino.
2. Spécifier l'URL de la base de données (`http ://localhost/tamino/Library`) et le nom de la collection (`LibraryCollection`) dans les champs *Database URL* et *Collection*.
3. Introduire l'expression XQuery dans le champ situé à côté du bouton *Query*. Par exemple, pour obtenir la date d'acquisition de l'exemplaire situé travée 6, rayon 7 et étage 65, il suffit d'introduire l'expression suivante :

```
//COPY/AcquisitionDate[../Location/Row=6 and ../Location/Shelf=7
     and ../Location/Level=65]
```

4. Cliquer sur le bouton *Query*.

Résultat : Tamino répond à la requête en générant un document XML qui apparaît dans la partie inférieure de l'interface (voir Annexe D). La Fig.6.28 montre l'état de l'interface après la requête.

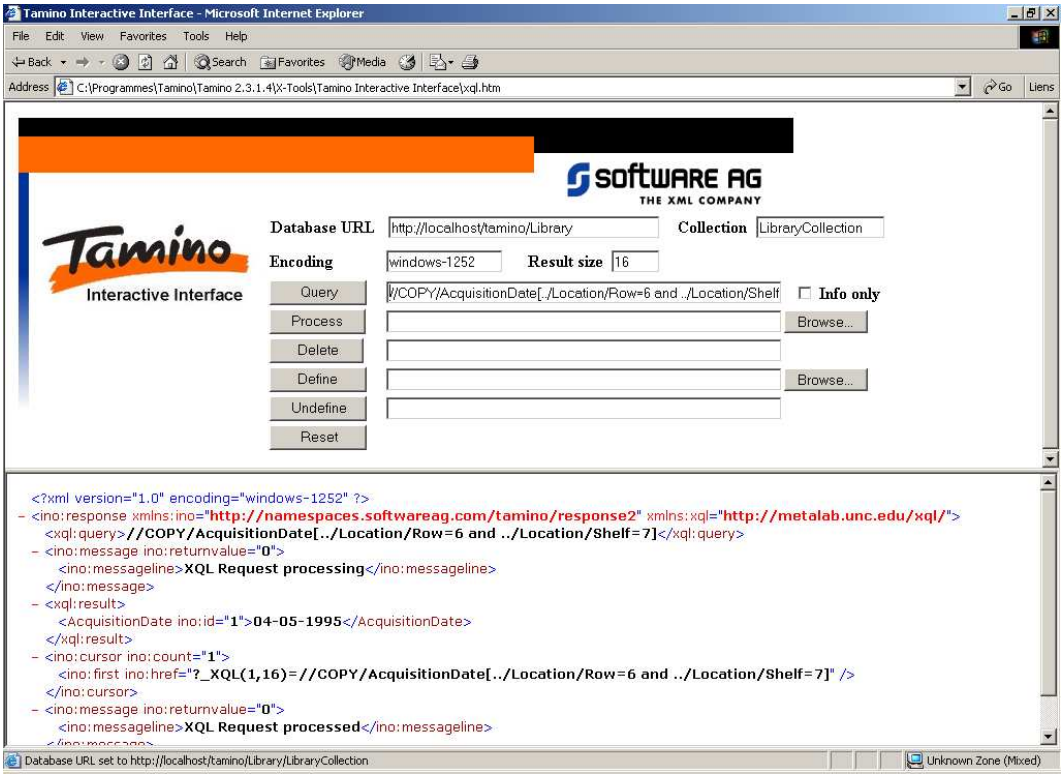

FIG. 6.28 – Après une requête avec l'interface interactive Tamino, le document XML (résultat) s'affiche dans la frame inférieure

6.7.2 Le browser internet

Pour illustrer l'interrogation d'une collection Tamino à partir d'un navigateur internet, une application nommée *QueryLibrary* a été réalisée en HTML et Javascript. L'interface de cette application (Fig.6.29) a été réalisée en HTML (voir Annexe E) et est assez semblable à l'interface interactive Tamino.

La frame supérieure contient des exemples de requêtes (exprimées de manière littérale), chacune accompagnée d'un bouton d'exécution *Query*. Certaines requêtes sont paramétrées et requièrent donc l'introduction d'une valeur dans le champ texte qui leur est associé. Un clic sur un bouton *Query* déclenche l'exécution d'une fonction Javascript conforme au pattern suivant :

```
function <FunctionName>()
{
        //Load XML
        var xml = new ActiveXObject("Microsoft.XMLDOM")
        xml.async = false
        var xquery = <X-MachineCommand>
        xml.load(xquery)

        //Write X-MachineCommand
        top.frames[1].document.write(xquery)
        top.frames[1].document.location.reload()

        // Load the XSL
```

```
        var xsl = new ActiveXObject("Microsoft.XMLDOM")
        xsl.async = false
        xsl.load(<XSLDocument>)

        // Transform
        top.frames[2].document.write(xml.transformNode(xsl))
        top.frames[2].document.location.reload()
}
```

où
`<FunctionName>` est le nom donné à la fonction Javascript
`<X-MachineCommand>` est la commande X-Machine conforme à la syntaxe vue dans la section 5.2.4, page 85
`<XSLDocument>` est le chemin et le nom du fichier contenant les transformations XSL à appliquer au résultat de Tamino

La fonction Javascript se décompose en quatre étapes successives :

1. **Load XML :** la commande X-Machine est exécutée et le résultat – un objet DOM – est stocké dans la variable nommée `xml`. Le code des fichiers XML renvoyés par Tamino pour chaque requête se trouve en Annexe F.
2. **Write X-MachineCommand :** la commande X-Machine exécutée est recopiée dans la frame intermédiaire.
3. **Load the XSL :** les transformations XSL à appliquer au résultat sont chargées dans la variable nommée `xsl` à partir d'un fichier .xsl. Le code des fichiers contenant les

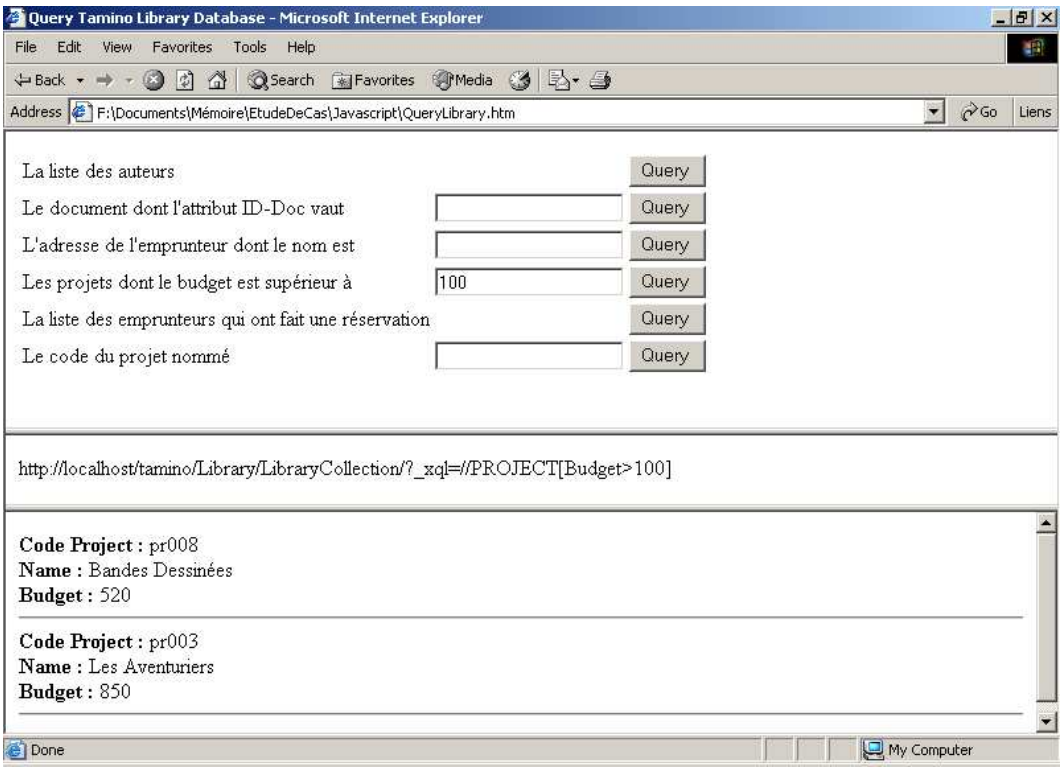

FIG. 6.29 – L'interface HTML de l'application *QueryLibrary*

transformations XSL à appliquer à chaque résultat se trouve en Annexe G.

4. **Transform :** les transformations XSL sont appliquées au résultat de la requête afin de générer un document HTML qui est affiché dans la frame inférieure.

Requête	Expression XQuery
La liste des auteurs	`//WRITER`
Le document dont l'attribut `ID-Doc` vaut «`X`»	`//DOCUMENT[@ID-Doc="X"]`
L'adresse de l'emprunteur dont le nom est «`X`»	`//BORROWER/Address[../Name="X"]`
Les projets dont le budget est supérieur à `X`	`//PROJECT[Budget>X]`
La liste des emprunteurs qui ont fait une réservation	`//BORROWER[./reserves]`
Le code du projet (attribut) nommé «`X`»	`//@CodeProject[../Name = "X"]`

Le tableau ci-dessus reprend les requêtes proposées dans l'application ainsi que l'expression XQuery permettant d'obtenir le résultat. L'utilisation de paramètres est représentée par la variable `X`. Ces différents exemples illustrent de manière significative les possibilités offertes par la syntaxe XQuery. On remarque que la notion de jointure n'est pas possible dans la norme XQuery implémentée par Tamino. Dans la section suivante, nous utilisons le code et les variables Java pour simuler les jointures entre élément XML.

6.7.3 Le client java

Avec son orientation XML, la possibilité d'interaction avec XSL et son langage de commande reposant sur le protocole HTTP, Tamino est un serveur de document idéal pour la publication de données sur internet. Dans cette section, nous montrons qu'il est possible d'exploiter Tamino comme un SGBD tout à fait conventionnel : nous utilisons les objets *TaminoClient*, *TaminoResult* et *TaminoError* pour construire un client pour Tamino en Java. Ce client implémente des méthodes qui interrogent Tamino à l'aide d'expresions XQuery, les documents XML renvoyés par Tamino sont parsés afin d'en extraire les informations pertinentes que nous stockons dans des variables Java, ces variables sont ensuite affichées dans une interface réalisée à l'aide d'un package graphique, comme *javax.swing*. Même si une telle application n'exploite pas toute la richesse des mécanismes vus précédemment, elle bénéficie toutefois des performances intéressantes du serveur Tamino et de sa X-Machine. Pour simuler les jointures, le client Tamino exécute des requêtes XQuery successives : le résultat d'une requête est stocké dans une variable intermédiaire qui sert de paramètre à la requête suivante.

Description du fonctionnement de l'application

L'application Java développée interroge la collection `LibraryCollection` pour afficher les emprunts en cours d'un emprunteur : l'utilisateur choisit dans une boîte déroulante le numéro identifiant d'un emprunteur (attribut `NumPers` de l'élément `BORROWER`), lorsque le choix est fait, le nom de l'emprunteur s'affiche dans un champ texte. Pour chaque emprunt, le titre du livre emprunté ainsi que le nom du projet concerné s'affiche dans le tableau des emprunts en cours. L'interface de l'application (Fig.6.30) est assez simpliste mais on imagine facilement la réalisation d'une interface plus riche, offrant plus de fonctionnalités.

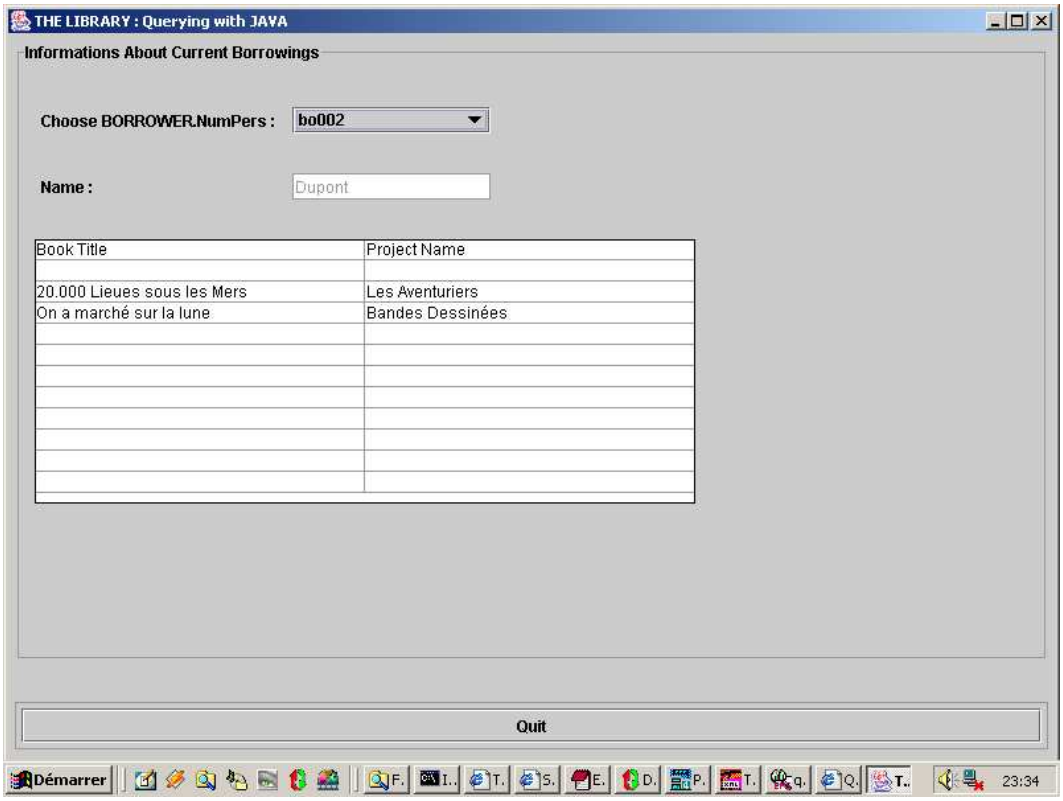

FIG. 6.30 – L'interface de l'application Java réalisée à l'aide du package *javax.swing*

QuelquesCommentaires sur le code Java

Le code Java complet du client Tamino se trouve en Annexe H : outre le constructeur qui se charge de créer l'objet *TaminoClient*, le client implémente trois méthodes : `GetNumPers()`, `GetBorrowerName(String NumPers)` et `GetBorrows(String NumPers)`. Il est intéressant de préciser le rôle de ces méthodes et de commenter les expressions XQuery qu'elles utilisent pour atteindre leur objectif.

La méthode `GetNumPers()` définit les valeurs de la liste déroulante en renvoyant la liste des valeurs des attributs `NumPers` appartenant aux éléments `BORROWER`. Pour obtenir ces informations, l'expression XQuery :

```
//BORROWER
```

rassemble dans un objet DOM tous les éléments `BORROWER` de la base de données. Grâce a diverses méthodes de parsing, la valeur de l'attribut `NumPers` est extraite de chaque élément `BORROWER`.

La méthode `GetBorrowerName(String NumPers)` obtient le nom d'un emprunteur à partir de son numéro identifiant. Ces informations se trouvent toutes deux dans la portée de l'élément `BORROWER`. L'expression XQuery qui réalise l'opération est triviale car aucune jointure n'est nécessaire :

```
//BORROWER/Name[../@NumPers="NumPers"]
```

La méthode `GetBorrows(String NumPers)` se charge de remplir le tableau des emprunts en cours.

Pour obtenir le titre des livres empruntés par un emprunteur dont le numéro identifiant est `NumPers`, trois requêtes XQuery sont nécessaires :

```
//borrows[../@NumPers="NumPers" and not(./ReturnDate)]
```

permet d'obtenir les emprunts en cours (sans élément `ReturnDate`) de l'emprunteur `NumPers`. Les élément `borrows` ainsi obtenu possèdent un attribut `COPY_TECH_ID` référençant l'exemplaire emprunté (élément `COPY`).

```
//@ISBN[../@COPY_TECH_ID="IdCopy"]
```

permet d'obtenir le numéro ISBN (attribut `ISBN` de l'élément `COPY`) de l'exemplaire emprunté. Ce numéro référence le livre emprunté (élément `BOOK`).

```
//DOCUMENT/Title[..//@ISBN="NumISBN"]
```

permet d'obtenir le titre du livre identifié par le numéro ISBN précédemment trouvé.

Finalement, le nom des projets pour lesquels se font les emprunts se déduisent après deux requêtes à la base de données. La première requête est identique à celle utilisée pour trouver le titre des livres :

```
//borrows[../@NumPers="NumPers" and not(./ReturnDate)]
```

Les élément `borrows` ainsi obtenu possèdent un attribut `CodeProject` référençant le projet concerné par l'emprunt (élément `PROJECT`).

```
//PROJECT/Name[../@CodeProject="NumProjet"]
```

permet d'obtenir le nom du projet identifié par le code précédemment trouvé.

Chapitre 7

Conclusion

Format standard d'échange d'informations sur internet, le langage XML et son modèle de données semi-structuré viennent perturber le monde paisible des bases de données. A l'heure de la publication massive de documents hiérarchisés sur la toile, le semi-structuré constitue sans doute une voie d'avenir pour les bases de données.

Impuissants face à la déferlante XML, certains théoriciens s'effraient en constatant la popularité de ce nouveau modèle. D'autres esprits plus pragmatiques voient en XML un moyen simple de réconcilier le web et les bases de données.

Dans les années septante, de nombreuses recherches furent menées pour mettre au point des méthodes et des outils de conception pour les bases de données relationnelles. De la même manière, face à l'émergence du modèle semi-structuré, il est actuellement nécessaire de disposer de méthodes fiables et d'outils performants pour la mise en oeuvre de bases de données XML.

7.1 Synthèses et critiques du travail réalisé

Dans le cadre de ce mémoire, nous nous sommes intéressés à la conception d'une base de données XML dont le mode de stockage est de type natif. Notre démarche se conforme au processus générique de conception de bases de données proposé dans [Hai00] et illustré Fig.1.1, page 13.

L'analyse conceptuelle n'a pas été évoquée dans ce rapport car il s'agit d'un processus de modélisation particulièrement bien maîtrisé et indépendant de toute technologie.

Le processus de conception logique, détaillé dans le chapitre 3, consiste à transformer le schéma conceptuel en un schéma logique conforme au modèle XML. Certaines étapes de la méthode proposée sont facilement automatisables. D'autres, comme la construction de la structure hiérarchique ou la transformation des attributs, requièrent l'intervention de l'utilisateur. Afin que l'utilisateur se sente guidé dans son travail de conception, l'outil supportant la méthode consiste en un enchaînement de processus et de boîtes de dialogue fournissant un «feedback» sur les actions réalisées. Dans la plupart des cas, les contraintes parfois sévères du modèle XML dégradent la sémantique originelle du schéma conceptuel. Certaines cardinalités de rôles doivent être élargies afin de respecter celles limitées par le modèle. Les identifiants et contraintes référentielles n'ont pas été épargnés par la restriction sémantique ; leur portée devenant globale à tout le document. Ajoutons la notion de type de données qui a complétement disparu dans le schéma logique. D'autres structures de données peuvent être représentées mais au prix d'un effort de modélisation lourd et difficile à mettre en oeuvre. C'est le cas des relations IS-A accompagnées de la contrainte de totalité.

Dans le chapitre 4 consacré à la conception physique, nous avons présenté certaines propriétés sensées améliorer les performances du futur système. Il n'existe pas de règles formelles

dictant les valeurs optimales à attribuer à ces paramètres. Il paraît cependant raisonnable d'attribuer un index de type standard aux attributs appartenant à un groupe «gid». Une fois le schéma physique construit, le codage consiste en un processus entièrement déterministe qui se charge de générer la DTD correspondante.

Lors du chapitre 5, nous avons présenté le fonctionnement d'un SGBD XML natif : Tamino. Nous avons montré comment générer un schéma physique Tamino à partir d'une DTD. Ce schéma, comportant des informations sur la structure des données et leurs propriétés physiques, est utilisé pour définir une collection dans une base de données. Ensuite, nous nous sommes attardés sur les mécanismes d'importation et d'interrogation des données XML. Bien que très performant, le langage d'interrogation XQuery nous a montré ses limites lorsqu'il s'agissait de formuler des requêtes plus complexes, utilisant des jointures. Il fallait alors s'accommoder de variables Java transitoires pour simuler le fonctionnement des clés étrangères. La mise à jour des données XML est une problématique qui n'a pas été abordée dans le cadre de ce mémoire. A ce sujet, le groupe de travail XUpdate [Gro] cherche à mettre au point un standard pour modifier de manière simple les données d'un document XML quelconque. Parallèlement, une proposition pour une syntaxe de mise à jour XQuery a été récemment proposée [Ogb02]. Reste à savoir quelle approche sera retenue par Software AG pour Tamino.

7.2 Perspectives

La méthode proposée pour transformer un schéma conceptuel en un schéma logique XML se veut semi-automatique. Elle est pertinente pour la transformation de schémas de taille petite ou moyenne. Pour la transformation de schémas plus conséquents ou pour répondre aux besoins d'utilisateurs peu soucieux de la structure finale des données, on pourrait imaginer une solution entièrement automatique. Lors de la construction de la structure hiérarchique, les conflits seraient alors résolus sans intervention de l'utilisateur : soit par le choix aléatoire d'un père, soit conformément à une stratégie générale. La stratégie de l'arbre le plus profond, par exemple, trouverait la plus longue descendance dans le schéma et en ferait une branche dans la hiérarchie finale. A l'inverse, la stratégie de l'arbre plat donnerait le statut de racine (naturelle) à tous les types d'entités du schéma. Entre ces deux stratégies extrêmes, il peut exister d'autres approches aussi judicieuses à mettre en oeuvre...

Comme nous l'avons signalé précédemment, la solution actuelle ne supporte pas les schémas récursifs. La difficulté supplémentaire à prendre en compte est la gestion des circuits dans le schéma initial. Un schéma récursif possède en effet un (ou plusieurs) circuit(s) qu'il s'agit alors d'éliminer. Un circuit peut être brisé en forçant l'utilisateur à élire, comme racine facultative, un des types d'entités composant le circuit.

L'un des apports intéressants des recherches réalisées pour ce travail est la représentation des relations IS-A. Il s'agit, par la même occasion, d'une des principales difficultés rencontrées lors du développement de l'outil. Nous avons contourné cet obstacle en optant pour une gestion des relations IS-A de totalité, limitées à deux sous-types. Afin de pouvoir gérer tous les types de relations, une solution plus générique devrait pouvoir être trouvée à ce problème combinatoire.

Pour des raisons d'interopérabilité avec Tamino, nous nous sommes particulièrement intéressés au modèle logique et à la génération automatique des DTD. Nouvelle norme adoptée dans la dernière version de Tamino, les XML Schemas et toutes leurs spécificités méritent également d'être pris en considération dans un futur travail.

Enfin, d'autres fonctionnalités peuvent être facilement ajoutées au transformateur de schémas. On pense, par exemple, à la génération d'un fichier journal (log) synthétisant les actions réalisées sur le schéma lors de l'étape de transformation ou encore la transformation des objets «post-it» en éléments de commentaire qui apparaîtraient dans le schéma logique.

Concernant le SGBD Tamino, il serait intéressant d'examiner en détail les nouvelles fonctionnalités offertes par la version 3.1.1. On peut citer, parmis celles-ci, la compatibilité avec un sous-ensemble des spécifications des XML Schemas ou l'existence d'un outil de navigation (Tamino X-Plorer) à travers le contenu et les schémas de la base de données.

Phénomène de mode, atout commercial, espéranto du web ou véritable modèle de bases de données ? La juste vocation d'XML dépendra de l'usage réel qu'il en sera fait lorsque les normes sortiront de leur phase de gestation. Les possibilités d'XML sont multiples mais l'abondance des standards émergents ne serait-elle pas une menace pour l'utilisation de cette nouvelle norme ?

Bibliographie

[AGa] Software AG. Documentation tamino. *http ://www.xmlstarterkit.com*.

[AGb] Software AG. Software ag. *http ://www.softwareag.com*.

[Bou02] Ronald Bourret. Xml and databases. *http ://www.rpbourret.com/xml/XMLAndDatabases.htm*, February 2002.

[Con] World Wide Web Consortium. Recommendations sur la norme xml. *http ://www.w3c.org/XML/*.

[Del01a] Christine Delcroix. Conventions de représentation d'un dtd dans le modèle ger. 2001.

[Del01b] Christine Delcroix. Modèle xml. 2001.

[Del01c] Christine Delcroix. Plan de transformation d'un schéma conceptuel en un schéma xml. 2001.

[Doc] Docuverse. *http ://www.docuverse.com/domsdk/index.html*.

[Eng00] Vincent Englebert. *Voyager 2 : Reference Manual (Version 6 - Release 0)*. FUNDP - Institut d'Informatique, Namur, November 2000.

[Gar] Georges Gardarin. Les datawebhouses arrivent. *http ://www.e-xmlmedia.fr/site_francais/documentation.htm*.

[Gar01] Georges Gardarin. Données hétérogènes sur le web : Interopérabilité, fédération de sources hétérogènes et bases de données. EPFL - Lausanne, December 2001.

[Gol01] C.F. Goldfarb. *Designing XML Databases*. Prentice Hall PTR, October 2001.

[Gro] The XUpdate Working Group. The xupdate recommendation. *http ://www.xmldb.org/xupdate*.

[Hai99] Jean-Luc Hainaut. *Computer-Aided Database Engineering - Volume 1 : Database Models*. FUNDP - Institut d'Informatique, Namur, db-main manual series edition, 1999.

[Hai00] Jean-Luc Hainaut. Cours fundp 1ère maîtrise : Ingénierie des base de données. 1999-2000.

[HEH+94] Jean-Luc Hainaut, Vincent Englebert, Jean Henrard, Jean-Marc Hick, and Didier Roland. Database evolution : the db-main approach. September 1994.

[KW01] Carsten Kleiner and Udo W.Lipeck. Automatic generation of xml dtds from conceptual database schemas. *Universität Hannover, Institut für Informatik, Lange Laube 22, 30159 Hannover, Germany*, 2001.

[LIB02] LIBD. *MDL Programmer's Guide, version 6.5*. FUNDP - Institut d'Informatique, Namur, db-main manual series edition, March 2002.

[Mic00] Alain Michard. *XML : langage et applications*. December 2000.

[Ogb02] Uche Ogbuji. Forte activité xquery. *http ://xmlfr.org/actualites/tech/020530-0002*, May 2002.

[Pro] The Apache Project. Présentation de xalan pour java. *http ://xml.apache.org/xalan-j*.

Annexe A

Code de la DTD `Library`

```
<!--
    This DTD file is automatically generated by DB-MAIN, the 24 /7 /2002
    from the schema "LIBRARY" of the project "LibraryToXML".
-->
<!ELEMENT LIBRARY (BORROWER*, COPY*, DOCUMENT*, PROJECT*, WRITER
      *)>
<!ELEMENT BORROWER (Address, borrows*, Phone+, reserves*, Name)>
<!ATTLIST BORROWER
        NumPers ID #REQUIRED
        CodeProject IDREF #IMPLIED
>
<!ELEMENT COPY (Location, AcquisitionDate)>
<!ATTLIST COPY
        COPY_TECH_ID ID #REQUIRED
        ISBN IDREF #REQUIRED
>
<!ELEMENT DOCUMENT (KeyWord*, (BOOK | REPORT)?, Title, PublicationDate)>
<!ATTLIST DOCUMENT
        ID-Doc ID #REQUIRED
>
<!ELEMENT PROJECT (Name, Budget)>
<!ATTLIST PROJECT
        CodeProject ID #REQUIRED
>
<!ELEMENT WRITER (writes+, Name, FirstName?)>
<!ELEMENT Address (Street, City)>
<!ELEMENT borrows (BorrowingDate, ReturnDate?)>
<!ATTLIST borrows
        CodeProject IDREF #REQUIRED
        COPY_TECH_ID IDREF #REQUIRED
>
<!ELEMENT Phone (#PCDATA)>
<!ELEMENT reserves (ReservationDate)>
<!ATTLIST reserves
        ID-Doc IDREF #REQUIRED
>
```

```
<!ELEMENT Name (#PCDATA)>
<!ELEMENT Location (Row, Shelf, Level)>
<!ELEMENT AcquisitionDate (#PCDATA)>
<!ELEMENT KeyWord (#PCDATA)>
<!ELEMENT BOOK (Publisher)>
<!ATTLIST BOOK
        ISBN ID #REQUIRED
>
<!ELEMENT REPORT EMPTY>
<!ATTLIST REPORT
        CodeReport ID #REQUIRED
>
<!ELEMENT Title (#PCDATA)>
<!ELEMENT PublicationDate (#PCDATA)>
<!ELEMENT Budget (#PCDATA)>
<!ELEMENT writes EMPTY>
<!ATTLIST writes
        ID-Doc IDREF #REQUIRED
>
<!ELEMENT FirstName (#PCDATA)>
<!ELEMENT Street (#PCDATA)>
<!ELEMENT City (#PCDATA)>
<!ELEMENT BorrowingDate (#PCDATA)>
<!ELEMENT ReturnDate (#PCDATA)>
<!ELEMENT ReservationDate (#PCDATA)>
<!ELEMENT Row (#PCDATA)>
<!ELEMENT Shelf (#PCDATA)>
<!ELEMENT Level (#PCDATA)>
<!ELEMENT Publisher (#PCDATA)>
```

Annexe B

Fichier XML contenant les données à importer dans la base de données `Library`

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE LIBRARY SYSTEM "F:\Documents\éMmoire\EtudeDeCas\Library.dtd">
<LIBRARY>
        <BORROWER NumPers="bo002">
                <Address>
                        <Street>rue Grandgagnage</Street>
                        <City>Namur</City>
                </Address>
                <borrows CodeProject="pr003" COPY_TECH_ID="co013">
                        <BorrowingDate>15-08-2002</BorrowingDate>
                </borrows>
                <borrows CodeProject="pr008" COPY_TECH_ID="co007">
                        <BorrowingDate>30-07-2002</BorrowingDate>
                        <ReturnDate>10-08-2002</ReturnDate>
                </borrows>
                <Phone>081/23.45.21</Phone>
                <Phone>0478/52.95.62</Phone>
                <reserves ID-Doc="doc412">
                        <ReservationDate>22-08-2002</ReservationDate>
                </reserves>
                <Name>Dupont</Name>
        </BORROWER>
        <BORROWER NumPers="bo006">
                <Address>
                        <Street>rue de la Bourse</Street>
                        <City>Bruxelles</City>
                </Address>
                <borrows CodeProject="pr003" COPY_TECH_ID="co246">
                        <BorrowingDate>20-08-2002</BorrowingDate>
                </borrows>
                <Phone>02/562.58.41</Phone>
                <Name>Martin</Name>
```

```
</BORROWER>
<BORROWER NumPers="bo003">
        <Address>
                <Street>bld Tirou</Street>
                <City>Charleroi</City>
        </Address>
        <Phone>071/85.47.62</Phone>
        <Name>Jones</Name>
</BORROWER>
<COPY COPY_TECH_ID="co007" ISBN="ISBN-552-45623-524">
        <Location>
                <Row>2</Row>
                <Shelf>5</Shelf>
                <Level>36</Level>
        </Location>
        <AcquisitionDate>26-07-1985</AcquisitionDate>
</COPY>
<COPY COPY_TECH_ID="co004" ISBN="ISBN-548-6321-852">
        <Location>
                <Row>6</Row>
                <Shelf>7</Shelf>
                <Level>65</Level>
        </Location>
        <AcquisitionDate>04-05-1995</AcquisitionDate>
</COPY>
<COPY COPY_TECH_ID="co013" ISBN="ISBN-548-6321-852">
        <Location>
                <Row>7</Row>
                <Shelf>6</Shelf>
                <Level>21</Level>
        </Location>
        <AcquisitionDate>27-04-1978</AcquisitionDate>
</COPY>
<COPY COPY_TECH_ID="co089" ISBN="ISBN-548-6321-852">
        <Location>
                <Row>23</Row>
                <Shelf>5</Shelf>
                <Level>9</Level>
        </Location>
        <AcquisitionDate>30-01-1992</AcquisitionDate>
</COPY>
<COPY COPY_TECH_ID="co246" ISBN="ISBN-123-7895-584">
        <Location>
                <Row>5</Row>
                <Shelf>9</Shelf>
                <Level>1</Level>
        </Location>
        <AcquisitionDate>12-11-1996</AcquisitionDate>
</COPY>
<DOCUMENT ID-Doc="doc058">
```

```
        <KeyWord>Mer</KeyWord>
        <KeyWord>Aventure</KeyWord>
        <KeyWord>Sous-Marin</KeyWord>
        <BOOK ISBN="ISBN-548-6321-852">
                <Publisher>Dufour</Publisher>
        </BOOK>
        <Title>20.000 Lieues sous les Mers</Title>
        <PublicationDate>23-05-1952</PublicationDate>
</DOCUMENT>
<DOCUMENT ID-Doc="doc052">
        <KeyWord>Reporter</KeyWord>
        <KeyWord>Train</KeyWord>
        <BOOK ISBN="ISBN-552-45623-524">
                <Publisher>Casterman</Publisher>
        </BOOK>
        <Title>Tintin au Congo</Title>
        <PublicationDate>15-06-1965</PublicationDate>
</DOCUMENT>
<DOCUMENT ID-Doc="doc412">
        <KeyWord>Phileas Phog</KeyWord>
        <KeyWord>Passe-partout</KeyWord>
        <KeyWord>èMontgolfire</KeyWord>
        <KeyWord>Voyage</KeyWord>
        <KeyWord>Monde</KeyWord>
        <BOOK ISBN="ISBN-123-7895-584">
                <Publisher>Dufour</Publisher>
        </BOOK>
        <Title>Le tour du monde en 80 jours</Title>
        <PublicationDate>02-10-1935</PublicationDate>
</DOCUMENT>
<DOCUMENT ID-Doc="doc007">
        <KeyWord>XML</KeyWord>
        <KeyWord>Base de édonnes</KeyWord>
        <KeyWord>DB-Main</KeyWord>
        <KeyWord>Tamino</KeyWord>
        <KeyWord>éSchma</KeyWord>
        <KeyWord>èModle</KeyWord>
        <REPORT CodeReport="re032"/>
        <Title>éMthode et outils pour la conception de bases de édonnes XML</
            Title>
        <PublicationDate>01-09-2002</PublicationDate>
</DOCUMENT>
<PROJECT CodeProject="pr008">
        <Name>Bandes éDessines</Name>
        <Budget>520</Budget>
</PROJECT>
<PROJECT CodeProject="pr003">
        <Name>Les Aventuriers</Name>
        <Budget>850</Budget>
</PROJECT>
```

```
        <WRITER>
                <writes ID-Doc="doc052"/>
                <Name>éHerg</Name>
        </WRITER>
        <WRITER>
                <writes ID-Doc="doc058"/>
                <writes ID-Doc="doc412"/>
                <Name>Verne</Name>
                <FirstName>Jules</FirstName>
        </WRITER>
</LIBRARY>
```

Annexe C

Le schéma Tamino généré à partir de la DTD `Library`

```
<?xml version="1.0"?>
<ino:collection  ino:name="LibraryCollection" ino:key="ID001">
        <ino:doctype ino:name="LIBRARY" ino:key="LIBRARY2" ino:options="READ␣
            INSERT␣UPDATE␣DELETE" ino:structure-index="NO">
          <ino:node ino:name="LIBRARY" ino:key="LIBRARY3" ino:obj-type="
              SEQ" ino:parent="LIBRARY2" ino:search-type="no" ino:map-type
              ="Native"/>
          <ino:node ino:name="BORROWER" ino:key="LIBRARY13" ino:obj-type
              ="SEQ" ino:parent="LIBRARY3" ino:multiplicity="*" ino:search-
              type="no" ino:map-type="No"/>
          <ino:node ino:name="CodeProject" ino:key="LIBRARY14" ino:obj-type=
              "CDATA" ino:parent="LIBRARY13" ino:search-type="no" ino:map
              -type="No"/>
          <ino:node ino:name="NumPers" ino:key="LIBRARY4" ino:obj-type="
              CDATA" ino:parent="LIBRARY13" ino:search-type="standard" ino
              :map-type="No"/>
          <ino:node ino:name="Address" ino:key="LIBRARY15" ino:obj-type="
              SEQ" ino:parent="LIBRARY13" ino:search-type="no" ino:map-
              type="No"/>
          <ino:node ino:name="Street" ino:key="LIBRARY47" ino:obj-type="
              PCDATA" ino:parent="LIBRARY15" ino:search-type="no" ino:map
              -type="No"/>
          <ino:node ino:name="City" ino:key="LIBRARY48" ino:obj-type="
              PCDATA" ino:parent="LIBRARY15" ino:search-type="no" ino:map
              -type="No"/>
          <ino:node ino:name="borrows" ino:key="LIBRARY5" ino:obj-type="SEQ
              " ino:parent="LIBRARY13" ino:multiplicity="*" ino:search-type="
              no" ino:map-type="Native"/>
          <ino:node ino:name="CodeProject" ino:key="LIBRARY49" ino:obj-type=
              "CDATA" ino:parent="LIBRARY5" ino:search-type="no" ino:map-
              type="No"/>
          <ino:node ino:name="COPY_TECH_ID" ino:key="LIBRARY50" ino:obj
              -type="CDATA" ino:parent="LIBRARY5" ino:search-type="no"
              ino:map-type="No"/>
```

<ino:node ino:name="BorrowingDate" ino:key="LIBRARY51" ino:obj-type="PCDATA" ino:parent="LIBRARY5" ino:search-type="no" ino:map-type="No"/>
<ino:node ino:name="ReturnDate" ino:key="LIBRARY52" ino:obj-type="PCDATA" ino:parent="LIBRARY5" ino:multiplicity="?" ino:search-type="no" ino:map-type="No"/>
<ino:node ino:name="Phone" ino:key="LIBRARY22" ino:obj-type="PCDATA" ino:parent="LIBRARY13" ino:multiplicity="+" ino:search-type="no" ino:map-type="No"/>
<ino:node ino:name="reserves" ino:key="LIBRARY6" ino:obj-type="SEQ" ino:parent="LIBRARY13" ino:multiplicity="*" ino:search-type="no" ino:map-type="Native"/>
<ino:node ino:name="ID-Doc" ino:key="LIBRARY53" ino:obj-type="CDATA" ino:parent="LIBRARY6" ino:search-type="no" ino:map-type="No"/>
<ino:node ino:name="ReservationDate" ino:key="LIBRARY54" ino:obj-type="PCDATA" ino:parent="LIBRARY6" ino:search-type="no" ino:map-type="No"/>
<ino:node ino:name="Name" ino:key="LIBRARY25" ino:obj-type="PCDATA" ino:parent="LIBRARY13" ino:search-type="no" ino:map-type="No"/>
<ino:node ino:name="COPY" ino:key="LIBRARY26" ino:obj-type="SEQ" ino:parent="LIBRARY3" ino:multiplicity="*" ino:search-type="no" ino:map-type="No"/>
<ino:node ino:name="ISBN" ino:key="LIBRARY27" ino:obj-type="CDATA" ino:parent="LIBRARY26" ino:search-type="no" ino:map-type="No"/>
<ino:node ino:name="COPY_TECH_ID" ino:key="LIBRARY7" ino:obj-type="CDATA" ino:parent="LIBRARY26" ino:search-type="standard" ino:map-type="No"/>
<ino:node ino:name="Location" ino:key="LIBRARY28" ino:obj-type="SEQ" ino:parent="LIBRARY26" ino:search-type="no" ino:map-type="No"/>
<ino:node ino:name="Row" ino:key="LIBRARY29" ino:obj-type="PCDATA" ino:parent="LIBRARY28" ino:search-type="no" ino:map-type="No"/>
<ino:node ino:name="Shelf" ino:key="LIBRARY30" ino:obj-type="PCDATA" ino:parent="LIBRARY28" ino:search-type="no" ino:map-type="No"/>
<ino:node ino:name="Level" ino:key="LIBRARY31" ino:obj-type="PCDATA" ino:parent="LIBRARY28" ino:search-type="no" ino:map-type="No"/>
<ino:node ino:name="AcquisitionDate" ino:key="LIBRARY32" ino:obj-type="PCDATA" ino:parent="LIBRARY26" ino:search-type="no" ino:map-type="No"/>
<ino:node ino:name="DOCUMENT" ino:key="LIBRARY33" ino:obj-type="SEQ" ino:parent="LIBRARY3" ino:multiplicity="*" ino:search-type="no" ino:map-type="No"/>
<ino:node ino:name="ID-Doc" ino:key="LIBRARY8" ino:obj-type="CDATA" ino:parent="LIBRARY33" ino:search-type="standard" ino

:map−type="No"/>
<ino:node ino:name="KeyWord" ino:key="LIBRARY34" ino:obj−type="
PCDATA" ino:parent="LIBRARY33" ino:multiplicity="∗" ino:search
−type="no" ino:map−type="No"/>
<ino:node ino:name="BOOK" ino:key="LIBRARY35" ino:obj−type="SEQ
" ino:parent="LIBRARY33" ino:multiplicity="?" ino:search−type="
no" ino:map−type="No"/>
<ino:node ino:name="ISBN" ino:key="LIBRARY9" ino:obj−type="
CDATA" ino:parent="LIBRARY35" ino:search−type="standard" ino
:map−type="No"/>
<ino:node ino:name="Publisher" ino:key="LIBRARY36" ino:obj−type="
PCDATA" ino:parent="LIBRARY35" ino:search−type="no" ino:map
−type="No"/>
<ino:node ino:name="REPORT" ino:key="LIBRARY37" ino:obj−type="
EMPTY" ino:parent="LIBRARY33" ino:multiplicity="?" ino:search
−type="no" ino:map−type="No"/>
<ino:node ino:name="CodeReport" ino:key="LIBRARY10" ino:obj−type=
"CDATA" ino:parent="LIBRARY37" ino:search−type="standard"
ino:map−type="No"/>
<ino:node ino:name="Title" ino:key="LIBRARY38" ino:obj−type="
PCDATA" ino:parent="LIBRARY33" ino:search−type="no" ino:map
−type="No"/>
<ino:node ino:name="PublicationDate" ino:key="LIBRARY39" ino:obj−
type="PCDATA" ino:parent="LIBRARY33" ino:search−type="no"
ino:map−type="No"/>
<ino:node ino:name="PROJECT" ino:key="LIBRARY40" ino:obj−type="
SEQ" ino:parent="LIBRARY3" ino:multiplicity="∗" ino:search−type
="no" ino:map−type="No"/>
<ino:node ino:name="CodeProject" ino:key="LIBRARY11" ino:obj−type=
"CDATA" ino:parent="LIBRARY40" ino:search−type="standard"
ino:map−type="No"/>
<ino:node ino:name="Name" ino:key="LIBRARY41" ino:obj−type="
PCDATA" ino:parent="LIBRARY40" ino:search−type="no" ino:map
−type="No"/>
<ino:node ino:name="Budget" ino:key="LIBRARY42" ino:obj−type="
PCDATA" ino:parent="LIBRARY40" ino:search−type="no" ino:map
−type="No"/>
<ino:node ino:name="WRITER" ino:key="LIBRARY43" ino:obj−type="
SEQ" ino:parent="LIBRARY3" ino:multiplicity="∗" ino:search−type
="no" ino:map−type="No"/>
<ino:node ino:name="writes" ino:key="LIBRARY12" ino:obj−type="
EMPTY" ino:parent="LIBRARY43" ino:multiplicity="+" ino:search
−type="no" ino:map−type="Native"/>
<ino:node ino:name="ID−Doc" ino:key="LIBRARY55" ino:obj−type="
CDATA" ino:parent="LIBRARY12" ino:search−type="no" ino:map−
type="No"/>
<ino:node ino:name="Name" ino:key="LIBRARY45" ino:obj−type="
PCDATA" ino:parent="LIBRARY43" ino:search−type="no" ino:map
−type="No"/>
<ino:node ino:name="FirstName" ino:key="LIBRARY46" ino:obj−type="

PCDATA" ino:parent="LIBRARY43" ino:multiplicity="?" ino:search
−type="no" ino:map−type="No"/>
</ino:doctype>
</ino:collection>

Annexe D

Exemple de document XML généré par Tamino suite à une requête

```
<?xml version="1.0" encoding="windows-1252" ?>
<ino:response xmlns:ino="http://namespaces.softwareag.com/tamino/response2" xmlns:xql
    ="http://metalab.unc.edu/xql/">
      <xql:query>
            //COPY/AcquisitionDate[../Location/Row=6 and ../Location/Shelf=7 and
                ../Location/Level=65]
      </xql:query>
      <ino:message ino:returnvalue="0">
            <ino:messageline>XQL Request processing</ino:messageline>
      </ino:message>
      <xql:result>
            <AcquisitionDate ino:id="5">04-05-1995</AcquisitionDate>
      </xql:result>
      <ino:cursor ino:count="1">
            <ino:first ino:href="?_XQL(1,16)=//COPY/AcquisitionDate[../Location/
                Row=6 and ../Location/Shelf=7 and ../Location/Level=65]" />
      </ino:cursor>
      <ino:message ino:returnvalue="0">
            <ino:messageline>XQL Request processed</ino:messageline>
      </ino:message>
</ino:response>
```

Annexe E

Code HTML de l'application *QueryLibrary*

E.1 QueryLibrary.htm

```
<html>
        <head>
                <title>Query Tamino Library Database</title>
        </head>
        <frameset rows="50%,15%,35%">
                <frame src="QueryFrame.htm" name="QueryFrame">
                <frame src="XQueryFrame.htm" name="XQueryFrame">
                <frame src="ResultFrame.htm" name="ResultFrame">
        </frameset>
</html>
```

E.2 QueryFrame.htm

```
<html>
        <head>
                <title>Query Tamino Library Database</title>
                <script Language="JavaScript" src="QueryLibrary.js"></script>
        </head>
        <body>
                <table>
                        <tr>
                                <form name="formListWriter">
                                        <td>La liste des auteurs</td>
                                        <td></td>
                                        <td align="left" width="105">
                                                <input name="butListWriter" type="
                                                    button" value="Query " onclick="
                                                    ListWriter()">
                                        </td>
                                </form>
                        </tr>
```

```
<tr>
        <form name="formListDocument">
                <td>Le document dont l'attribut ID-Doc vaut </
                    td>
                <td>
                        <input type=text size=20 name="
                            txtIdDoc" value="">
                </td>
                <td align="left" width="105">
                        <input name="butListDocument" type=
                            "button" value="Query " onclick="
                            ListDocument()">
                </td>
        </form>
</tr>

<tr>
        <form name="formListAddress">
                <td>L'adresse de l'emprunteur dont le nom est </
                    td>
                <td>
                        <input type=text size=20 name="
                            txtNameBorrower" value="">
                </td>
                <td align="left" width="105">
                        <input name="butListAddress" type="
                            button" value="Query " onclick="
                            ListAddress()">
                </td>
        </form>
</tr>

<tr>
        <form name="formListProject">
                <td>Les projets dont le budget est ésuprieurà  </
                    td>
                <td>
                        <input type=text size=20 name="
                            txtBudgetProject" value="">
                </td>
                <td align="left" width="105">
                        <input name="butListProject" type="
                            button" value="Query " onclick="
                            ListProject()">
                </td>
        </form>
</tr>

<tr>
        <form name="formListBorrow">
```

```
                                        <td>La liste des emprunteurs qui ont fait une é
                                            rservation</td>
                                        <td></td>
                                        <td align="left" width="105">
                                            <input name="butListBorrow" type="
                                                button" value="Query " onclick="
                                                ListBorrow()">
                                        </td>
                                    </form>
                                </tr>

                                <tr>
                                    <form name="formListCodeProject">
                                        <td>Le code du projet énomm </td>
                                        <td>
                                            <input type=text size=20 name="
                                                txtNameProject" value="">
                                        </td>
                                        <td align="left" width="105">
                                            <input name="butListCodeProject"
                                                type="button" value="Query "
                                                onclick="ListCodeProject()">
                                        </td>
                                    </form>
                                </tr>

                        </table>
                </body>
        </html>
```

E.3 XQueryFrame.htm

```
<html>
        <head>
                <title>Query Tamino Library Database</title>
        </head>
        <body>
        </body>
</html>
```

E.4 ResultFrame.htm

```
<html>
        <head>
                <title>Query Tamino Library Database</title>
        </head>
        <body>
        </body>
</html>
```

Annexe F

Code des fichiers XML renvoyés par Tamino en réponse aux requêtes de l'application *QueryLibrary*

F.1 Requête : La liste des auteurs

```
<?xml version="1.0" encoding="UTF-8"?>
<ino:response xmlns:ino="http://namespaces.softwareag.com/tamino/response2" xmlns:xql
      ="http://metalab.unc.edu/xql/">
        <xql:query>//WRITER</xql:query>
        <ino:message ino:returnvalue="0">
                <ino:messageline>XQL Request processing</ino:messageline>
        </ino:message>
        <xql:result>
                <WRITER ino:id="5">
                        <writes ID-Doc="doc052"/>
                        <Name>éHerg</Name>
                </WRITER>
                <WRITER ino:id="5">
                        <writes ID-Doc="doc058"/>
                        <writes ID-Doc="doc412"/>
                        <Name>Verne</Name>
                        <FirstName>Jules</FirstName>
                </WRITER>
        </xql:result>
        <ino:cursor ino:count="1">
                <ino:first  ino:href="?_XQL(1,16)=//WRITER"/>
        </ino:cursor>
        <ino:message ino:returnvalue="0">
                <ino:messageline>XQL Request processed</ino:messageline>
        </ino:message>
</ino:response>
```

F.2 Requête : Le document dont l'attribut `ID-Doc` vaut «doc007»

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<ino:response xmlns:ino="http://namespaces.softwareag.com/tamino/response2" xmlns:xql
      ="http://metalab.unc.edu/xql/">
        <xql:query>//DOCUMENT[@ID-Doc="doc007"]</xql:query>
        <ino:message ino:returnvalue="0">
                <ino:messageline>XQL Request processing</ino:messageline>
        </ino:message>
        <xql:result>
                <DOCUMENT ino:id="1" ID-Doc="doc007">
                        <KeyWord>XML</KeyWord>
                        <KeyWord>Base de édonnes</KeyWord>
                        <KeyWord>DB-Main</KeyWord>
                        <KeyWord>Tamino</KeyWord>
                        <KeyWord>éSchma</KeyWord>
                        <KeyWord>èModle</KeyWord>
                        <REPORT CodeReport="re032"/>
                        <Title>éMthode et outils pour la conception de bases de édonnes
                              XML</Title>
                        <PublicationDate>01-09-2002</PublicationDate>
                </DOCUMENT>
        </xql:result>
        <ino:cursor ino:count="1">
                <ino:first  ino:href="?_XQL(1,16)=//DOCUMENT[@ID-Doc=doc007
                     ]"/>
        </ino:cursor>
        <ino:message ino:returnvalue="0">
                <ino:messageline>XQL Request processed</ino:messageline>
        </ino:message>
</ino:response>
```

F.3 Requête : L'adresse de l'emprunteur dont le nom est «Jones»

```
<?xml version="1.0" encoding="UTF-8"?>
<ino:response xmlns:ino="http://namespaces.softwareag.com/tamino/response2" xmlns:xql
      ="http://metalab.unc.edu/xql/">
        <xql:query>//BORROWER/Address[../Name="Jones"]</xql:query>
        <ino:message ino:returnvalue="0">
                <ino:messageline>XQL Request processing</ino:messageline>
        </ino:message>
        <xql:result>
                <Address ino:id="1">
                        <Street>bld Tirou</Street>
                        <City>Charleroi</City>
                </Address>
        </xql:result>
        <ino:cursor ino:count="1">
                <ino:first  ino:href="?_XQL(1,16)=//BORROWER/Address[../Name=
                     Jones]"/>
        </ino:cursor>
        <ino:message ino:returnvalue="0">
                <ino:messageline>XQL Request processed</ino:messageline>
```

```
        </ino:message>
</ino:response>
```

F.4 Requête : Les projets dont le budget est supérieur à 800

```
<?xml version="1.0" encoding="UTF-8"?>
<ino:response xmlns:ino="http://namespaces.softwareag.com/tamino/response2" xmlns:xql
      ="http://metalab.unc.edu/xql/">
        <xql:query>//PROJECT[Budget>800]</xql:query>
        <ino:message ino:returnvalue="0">
                <ino:messageline>XQL Request processing</ino:messageline>
        </ino:message>
        <xql:result>
                <PROJECT ino:id="1" CodeProject="pr003">
                        <Name>Les Aventuriers</Name>
                        <Budget>850</Budget>
                </PROJECT>
        </xql:result>
        <ino:cursor ino:count="1">
                <ino:first  ino:href="?_XQL(1,16)=//PROJECT[Budget>800]"/>
        </ino:cursor>
        <ino:message ino:returnvalue="0">
                <ino:messageline>XQL Request processed</ino:messageline>
        </ino:message>
</ino:response>
```

F.5 Requête : La liste des emprunteurs qui ont fait une réservation

```
<?xml version="1.0" encoding="UTF-8"?>
<ino:response xmlns:ino="http://namespaces.softwareag.com/tamino/response2" xmlns:xql
      ="http://metalab.unc.edu/xql/">
        <xql:query>//BORROWER[./reserves]</xql:query>
        <ino:message ino:returnvalue="0">
                <ino:messageline>XQL Request processing</ino:messageline>
        </ino:message>
        <xql:result>
                <BORROWER ino:id="1" NumPers="bo002">
                        <Address>
                                <Street>rue Grandgagnage</Street>
                                <City>Namur</City>
                        </Address>
                        <borrows CodeProject="pr003" COPY_TECH_ID="co013">
                                <BorrowingDate>15-08-2002</BorrowingDate>
                        </borrows>
                        <borrows CodeProject="pr008" COPY_TECH_ID="co246">
                                <BorrowingDate>17-08-2002</BorrowingDate>
                        </borrows>
                        <borrows CodeProject="pr008" COPY_TECH_ID="co007">
                                <BorrowingDate>30-07-2002</BorrowingDate>
```

```
                        <ReturnDate>10-08-2002</ReturnDate>
                    </borrows>
                    <Phone>081/23.45.21</Phone>
                    <Phone>0478/52.95.62</Phone>
                    <reserves ID-Doc="doc412">
                        <ReservationDate>22-08-2002</ReservationDate>
                    </reserves>
                    <Name>Dupont</Name>
                </BORROWER>
        </xql:result>
        <ino:cursor ino:count="1">
                <ino:first  ino:href="?_XQL(1,16)=//BORROWER[./reserves]"/>
        </ino:cursor>
        <ino:message ino:returnvalue="0">
                <ino:messageline>XQL Request processed</ino:messageline>
        </ino:message>
</ino:response>
```

F.6 Requête : Le code du projet (attribut) nommé «Bandes Dessinées»

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- edited with XML Spy v3.5 NT (http://www.xmlspy.com) by Fab (Utilisation
      Personnelle) -->
<ino:response xmlns:ino="http://namespaces.softwareag.com/tamino/response2" xmlns:xql
      ="http://metalab.unc.edu/xql/">
        <xql:query>//@CodeProject[../Name = "Bandes éDessines"]</xql:query>
        <ino:message ino:returnvalue="0">
                <ino:messageline>XQL Request processing</ino:messageline>
        </ino:message>
        <xql:result>
                <PROJECT ino:id="1" CodeProject="pr008"/>
        </xql:result>
        <ino:cursor ino:count="1">
                <ino:first  ino:href="?_XQL(1,16)=//@CodeProject[../Name = Bandes
                    éDessines]"/>
        </ino:cursor>
        <ino:message ino:returnvalue="0">
                <ino:messageline>XQL Request processed</ino:messageline>
        </ino:message>
</ino:response>
```

Annexe G

Code des fichiers XSL utilisés dans l'application *QueryLibrary*

G.1 Requête : La liste des auteurs

```
<?xml version='1.0'?>
<xsl:stylesheet  version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
      xmlns:ino="http://namespaces.softwareag.com/tamino/response2" xmlns:xql="http
      ://metalab.unc.edu/xql/">
        <xsl:template match="/">
                <html>
                <body>
                        <table border="2">
                                <tr>
                                        <th>Name</th>
                                        <th>First Name</th>
                                </tr>
                                <xsl:for-each select="//WRITER">
                                <tr>
                                        <td><xsl:value-of select="Name"/></td>
                                        <td><xsl:value-of select="FirstName"/></td>
                                </tr>
                                 </xsl:for-each>
                        </table>
                </body>
                </html>
        </xsl:template>
</xsl:stylesheet>
```

G.2 Requête : Le document dont l'attribut `ID-Doc` vaut «X»

```
<?xml version='1.0'?>
<xsl:stylesheet  version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
      xmlns:ino="http://namespaces.softwareag.com/tamino/response2" xmlns:xql="http
      ://metalab.unc.edu/xql/">

        <xsl:template match="/">
```

```
                <html>
                <body>
                        <xsl:apply-templates/>
                </body>
                </html>
        </xsl:template>

        <xsl:template match="DOCUMENT">
                        <b>Title : </b>
                        <xsl:value-of select="Title"/>
                        <br></br>
                        <b>Publication Date : </b>
                        <xsl:value-of select="PublicationDate"/>
                        <br></br>
                        <b>Key Word(s) : </b>
                        <xsl:for-each select="KeyWord">
                                <xsl:value-of select="KeyWord"/>
                                <br></br>
                                <xsl:apply-templates/>
                        </xsl:for-each>
        </xsl:template>

        <xsl:template match="xql:query"/>

        <xsl:template match="ino:message"/>

        <xsl:template match="ino:cursor"/>

</xsl:stylesheet>
```

G.3 Requête : L'adresse de l'emprunteur dont le nom est «X»

```
<?xml version='1.0'?>
<xsl:stylesheet  version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
      xmlns:ino="http://namespaces.softwareag.com/tamino/response2" xmlns:xql="http
      ://metalab.unc.edu/xql/">

        <xsl:template match="/">
                <html>
                <body>
                        <xsl:for-each select="//Address">
                                <big><b>Address :</b></big>
                                <br></br>
                                <b>Street : </b>
                                <xsl:value-of select="Street"/>
                                <br></br>
                                <b>City : </b>
                                <xsl:value-of select="City"/>
                                <br></br>
                                <hr></hr>
```

```
                </xsl:for-each>
            </body>
            </html>
        </xsl:template>

        <xsl:template match="xql:query"/>

        <xsl:template match="ino:message"/>

        <xsl:template match="ino:cursor"/>

</xsl:stylesheet>
```

G.4 Requête : Les projets dont le budget est supérieur à `X`

```
<?xml version='1.0'?>
<xsl:stylesheet  version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
      xmlns:ino="http://namespaces.softwareag.com/tamino/response2" xmlns:xql="http
      ://metalab.unc.edu/xql/">

        <xsl:template match="/">
            <html>
            <body>
                <xsl:for-each select="//PROJECT">
                    <b>Code Project : </b>
                    <xsl:value-of select="@CodeProject"/>
                    <br></br>
                    <b>Name : </b>
                    <xsl:value-of select="Name"/>
                    <br></br>
                    <b>Budget : </b>
                    <xsl:value-of select="Budget"/>
                    <br></br>
                    <hr></hr>
                </xsl:for-each>
            </body>
            </html>
        </xsl:template>

        <xsl:template match="xql:query"/>

        <xsl:template match="ino:message"/>

        <xsl:template match="ino:cursor"/>

</xsl:stylesheet>
```

G.5 Requête : La liste des emprunteurs qui ont fait une réservation

```
<?xml version='1.0'?>
<xsl:stylesheet  version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
      xmlns:ino="http://namespaces.softwareag.com/tamino/response2" xmlns:xql="http
      ://metalab.unc.edu/xql/">

        <xsl:template match="/">
                <html>
                <body>
                        <xsl:apply-templates/>
                </body>
                </html>
        </xsl:template>

        <xsl:template match="//BORROWER">
                <b>Name : </b>
                <xsl:value-of select="Name"/>
                <br></br>
                <b>Street : </b>
                <xsl:value-of select="Address/Street"/>
                <br></br>
                <b>City : </b>
                <xsl:value-of select="Address/City"/>
                <br></br>
                <b>Phone(s) : </b>
                <xsl:for-each select="Phone">
                        <xsl:value-of select="Phone"/>
                        <br></br>
                        <xsl:apply-templates/>
                </xsl:for-each>
                <hr></hr>
        </xsl:template>

        <xsl:template match="xql:query"/>

        <xsl:template match="ino:message"/>

        <xsl:template match="ino:cursor"/>

</xsl:stylesheet>
```

G.6 Requête : Le code du projet (attribut) nommé «X»

```
<?xml version='1.0'?>
<xsl:stylesheet  version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
      xmlns:ino="http://namespaces.softwareag.com/tamino/response2" xmlns:xql="http
      ://metalab.unc.edu/xql/">

        <xsl:template match="/">
                <html>
                <body>
```

```
                    <xsl:apply-templates/>
                </body>
                </html>
        </xsl:template>

        <xsl:template match="//PROJECT">
                <b>Code Project : </b>
                <xsl:value-of select="@CodeProject"/>
                <br></br>
        </xsl:template>

        <xsl:template match="xql:query"/>

        <xsl:template match="ino:message"/>

        <xsl:template match="ino:cursor"/>

</xsl:stylesheet>
```

Annexe H

Code Java du client Tamino pour la base de données `Library`

```
import org.w3c.dom.*;
import com.softwareag.tamino.api.dom.*;
import java.util.*;

class LibraryClient {

        private final static java.lang.String LibraryDBLocation = "http://localhost/
                tamino/Library/";
        private final static java.lang.String LibraryCollection = "LibraryCollection";

        com.softwareag.tamino.api.dom.TaminoClient LibraryClient;

        public LibraryClient()
        {
                try
                {
                        LibraryClient = new com.softwareag.tamino.api.dom.TaminoClient(
                                LibraryDBLocation+LibraryCollection);
                }
                catch(com.softwareag.tamino.api.dom.TaminoError TError)
                {
                        System.out.println(TError.getMessage());
                }
        }



        public Vector GetNumPers()
        {
                Vector Result = new Vector();
                String NumPers;

                try
                {
```

```
                com.softwareag.tamino.api.dom.TaminoResult Borrower =
                        LibraryClient.query("//BORROWER");

                if(Borrower.getResult() != null)
                {
                        org.w3c.dom.Node NodeBorrower = Borrower.getResult().
                                getFirstChild();

                        while(NodeBorrower != null)
                        {
                                NumPers = NodeBorrower.getAttributes().
                                        getNamedItem("NumPers").getNodeValue();
                                Result.add(NumPers);

                                NodeBorrower = NodeBorrower.getNextSibling();
                        }

                }
        }
        catch(com.softwareag.tamino.api.dom.TaminoError TError)
        {
                System.out.println(TError.getMessage());
        }

        return Result;

}


public String GetBorrowerName(String NumPers)
{
        try
        {
                com.softwareag.tamino.api.dom.TaminoResult NameBorrower =
                        LibraryClient.query("//BORROWER/Name[../@NumPers
                        =\""+NumPers+"\"]");

                return(NameBorrower.getResult().getFirstChild().getFirstChild().
                        toString());
        }
        catch(com.softwareag.tamino.api.dom.TaminoError TError)
        {
                System.out.println(TError.getMessage());
        }
        return("inconnu");
}



public Vector GetBorrows(String NumPers)
```

```
{
        String NumProjet;
        String IdCopy;
        String NumISBN;

        Vector Result = new Vector();

        try
        {
                com.softwareag.tamino.api.dom.TaminoResult Borrows =
                        LibraryClient.query("//borrows[../@NumPers=\""+NumPers
                        +"\"␣and␣not(./ReturnDate)]");

                if(Borrows.getResult() != null)
                {
                        org.w3c.dom.Node NodeBorrows = Borrows.getResult().
                                getFirstChild();

                        while(NodeBorrows != null)
                        {

                                IdCopy = NodeBorrows.getAttributes().
                                        getNamedItem("COPY_TECH_ID").
                                        getNodeValue();

                                com.softwareag.tamino.api.dom.TaminoResult ISBN
                                        = LibraryClient.query("//@ISBN[../
                                        @COPY_TECH_ID=\""+IdCopy+"\"]");

                                NumISBN = ISBN.getResult().getFirstChild().
                                        getAttributes().getNamedItem("ISBN").
                                        getNodeValue();

                                com.softwareag.tamino.api.dom.TaminoResult Title
                                        = LibraryClient.query("//DOCUMENT/
                                        Title[..//@ISBN=\""+NumISBN+"\"]");


                                NumProjet = NodeBorrows.getAttributes().
                                        getNamedItem("CodeProject").getNodeValue
                                        ();

                                com.softwareag.tamino.api.dom.TaminoResult
                                        Project = LibraryClient.query("//PROJECT
                                        /Name[../@CodeProject=\""+NumProjet+"
                                        \"]");



                                Borrow borrow = new Borrow(Title.getResult().
```

```
                                        getFirstChild().getFirstChild().toString(),
                                        Project.getResult().getFirstChild().
                                        getFirstChild().toString());

                                Result.add(borrow);

                                NodeBorrows = NodeBorrows.getNextSibling();
                        }
                }
        }
        catch(com.softwareag.tamino.api.dom.TaminoError TError)
        {
                System.out.println(TError.getMessage());
        }
        return Result;
    }
}
```