LABORATOIRE D'INFORMATIQUE SCIENTIFIQUE ET INDUSTRIELLE

Rapport de recherche

N° 01 004

Confrontation et intégration des pouvoirs d'expression des approches modélisation conceptuelle et structuration documentaire

Jérôme CHOCHON, Eric SARDET

ÉCOLE NATIONALE SUPÉRIEURE DE MÉCANIQUE ET D'AÉROTECHNIQUE
Site du Futuroscope - B.P. 109 - 86960 FUTUROSCOPE Cedex - FRANCE
Tél. : +33 (0) 5-49-49-80-63 - FAX : +33 (0) 5-49-49-80-64
Web : http://www.lisi.ensma.fr

*L*ABORATOIRE D'*I*NFORMATIQUE *S*CIENTIFIQUE ET *I*NDUSTRIELLE

Rapport de recherche

N° 01 004

Confrontation et intégration des pouvoirs d'expression des approches modélisation conceptuelle et structuration documentaire

Jérôme CHOCHON, Eric SARDET

ÉCOLE NATIONALE SUPÉRIEURE DE MÉCANIQUE ET D'AÉROTECHNIQUE
Site du Futuroscope - B.P. 109 - 86960 FUTUROSCOPE Cedex - FRANCE
Tél. : +33 (0) 5-49-49-80-63 - FAX : +33 (0) 5-49-49-80-64
Web : http://www.lisi.ensma.fr

Résumé

Résumé

Au cours des quinze dernières années, deux approches assez différentes se sont développées pour échanger des données techniques. La première correspond aux échanges orientés vers les bases de données où l'accent est tout particulièrement mis sur la cohérence et l'intégrité des données à l'aide de formalismes de modélisation des données tel que le schéma *Entité/Association*, ou bien des approches orientées objets comme *OMT* et *UML* et *EXPRESS*. La seconde approche est plus orientée vers des échanges entre humains où elle utilise une présentation des données de type documentaire afin qu'elles soient compréhensibles par l'homme (*HTML* et *XML*). Cependant cette approche ne permet pas d'exprimer quelque contrainte d'intégrité pour le contenu documentaire d'un document.

L'utilisation d'Internet pour réaliser des échanges entre machines et pour permettre un accès humain à l'information, nécessite d'appréhender correctement ces deux approches. L'apparition récente des schémas XML sensés être utilisables pour ces deux types d'objectifs nécessite une étude détaillée et comparative de leur pouvoir d'expression par rapport aux formalismes de modélisation conceptuelle tel que le langage *EXPRESS*.

Ce rapport présente une introduction à la modélisation conceptuelle, illustrée par la présentation du langage *EXPRESS*, et une présentation des schémas *XML Schema* et de leur pouvoir d'expression. Ensuite des solutions de mapping pour représenter de façon systématique des modèles de données *EXPRESS* en termes de *XML Schema* et la transformation inverse, à savoir le passage d'un modèle exprimé en termes d'un schéma *XML Schema* dans un univers *EXPRESS* seront proposées. Cette dernière transformation établie, elle sera ensuite validée sous la forme du développement d'une application destinée à analyser et valider la structure d'un document XML conformément à un modèle de type *XML Schema*, le tout dans un univers *EXPRESS*.

Mots clés

Formalismes de modélisation, *EXPRESS*, *XML*, *schémas XML*, *XML Schema*, *mapping EXPRESS vers XML Schema*, *méta modélisation*, *validation de documents XML*

Sommaire

Introduction

Au cours des quinze dernières années, deux approches assez différentes se sont développées pour échanger des données techniques. La première correspond aux échanges orientés vers les bases de données. L'accent y est tout particulièrement mis sur la cohérence et l'intégrité des données, représentées au travers des notions de typage fort, cardinalité, unicité, contraintes ensemblistes. Cette sémantique est exprimée à l'aide de formalismes typiques de représentation des données tel le schéma *Entité/Association*, ou bien des approches orientées objets comme *OMT* et *UML*, et, dans le domaine technique, *EXPRESS*. L'avantage de cette approche est la possibilité de réaliser des échanges très sûrs entre machines, mais au détriment de toute représentation compréhensible par un humain (nécessité de médiatiser l'information). La seconde approche est plus orientée vers des échanges entre humains. Elle utilise une présentation des données de type documentaire afin qu'elles soient compréhensibles par l'homme. Les méthodes relevant de cette approche s'articulent autour de deux concepts clés: la description explicite de l'organisation structurelle des documents d'une part, et la séparation de la structure et de la présentation d'autre part. Les outils de description communément utilisés sont *HTML* et *XML* dont l'utilisation aujourd'hui ne cesse de progresser. La faiblesse de cette approche reste l'impossibilité d'exprimer quelque contrainte d'intégrité que ce soit concernant le contenu documentaire d'une instance de document dont la structure est définie en *XML*.

L'explosion du phénomène Internet, utilisé tant pour réaliser des échanges entre machines que pour permettre un accès humain à l'information, nécessite d'appréhender correctement ces deux approches. De plus, l'apparition récente des schémas XML sensés être utilisables pour ces deux types d'objectifs nécessite une étude détaillée et comparative de leur pouvoir d'expression par rapport aux formalismes de modélisation conceptuelle communément utilisés, et en particulier le langage *EXPRESS*.

Ce rapport présente les résultats des travaux de recherche effectués dans le cadre d'un DEA dont les objectifs étaient les suivants :

- Comparer les schémas *XML Schema* au langage *EXPRESS*.
- Définir, si possible, un processus de transformation systématique d'un modèle *EXPRESS* en termes de modèles de type *XML Schema*.
- Définir la transformation inverse en vue de pouvoir utiliser les outils *EXPRESS* pour valider des documents XML basés sur des descriptions *XML Schema*.

Le contenu de ce rapport est le suivant. Une première partie sera consacrée à une introduction à la modélisation conceptuelle, introduction illustrée par la présentation du langage *EXPRESS*. La seconde partie sera quant à elle consacrée à la présentation et l'étude des schémas *XML Schema* et à leur pouvoir d'expression comparé à celui des formalismes de modélisation conceptuelle. Une troisième partie évaluera la possibilité de représenter de façon systématique des modèles de données *EXPRESS* en termes de *XML Schema*. L'intérêt de cette étude est de mesurer la possibilité de fournir un nouveau format d'échange au langage *EXPRESS*. Une quatrième partie sera consacrée à l'étude de la transformation inverse, à savoir le passage d'un modèle exprimé en termes d'un schéma *XML Schema* dans un univers *EXPRESS*. Cette transformation établie, elle sera ensuite validée sous la forme du développement d'une application destinée à analyser et valider la structure d'un document XML conformément à un modèle de type *XML Schema*, le tout dans un univers *EXPRESS*.

Les langages de modélisation de données

I.Introduction

Avant de permettre à un ordinateur de manipuler tout un ensemble d'informations, il convient de leur donner une forme, une représentation compréhensible par la machine. De nombreux formalismes se sont ainsi développés. De l'approche relationnelle à l'approche objet, leur pouvoir d'expression a progressé pour approcher le raisonnement humain dans la manière d'identifier et de distinguer les éléments traités et les associations qui les lient. Ces points seront décrits dans ce chapitre.

La première partie sera consacrée à une petite introduction sur la modélisation de la connaissance. Les travaux de KANT dans ce domaine seront exposés et serviront de base pour évaluer le pouvoir d'expression des formalismes de modélisation. Par la suite, dans une seconde partie, nous présenterons en détail un langage de spécification de données: le langage *EXPRESS*.

II.Modélisation de la connaissance

Selon les travaux de KANT, la connaissance peut être représentée selon trois points de vue ou dimensions: la *connaissance structurelle*, *la connaissance descriptive* et *la connaissance procédurale*.

1°) La connaissance structurelle

Chaque élément de la connaissance est représenté par un concept, encore appelé catégorie. Ainsi une distinction est faite dans la façon de voir le monde et de le comprendre. Par exemple, dans le cadre de la technologie mécanique, une différence peut-être faite entre une *vis* et un *écrou*. Deux catégories bien distinctes viennent donc d'être crées: *vis* et *écrou*.

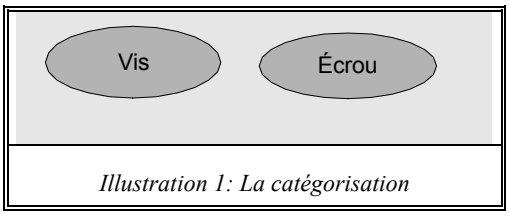


Illustration 1: La catégorisation

Ces catégories peuvent elles-mêmes être divisées et organisées hiérarchiquement. A partir des catégories, une identification plus précise est effectuée. En reprenant l'exemple précédent (Illustration 1), la catégorie *vis*, peut être spécialisée selon certains critères pour obtenir deux nouvelles catégories (Illustration 2):

- *vis à tête*, pour désigner une vis possédant une tête avec un mode d'entraînement par clé plate;
- *vis à tête cylindrique*, pour les vis devant être manipulées avec une clé à six pans creux.

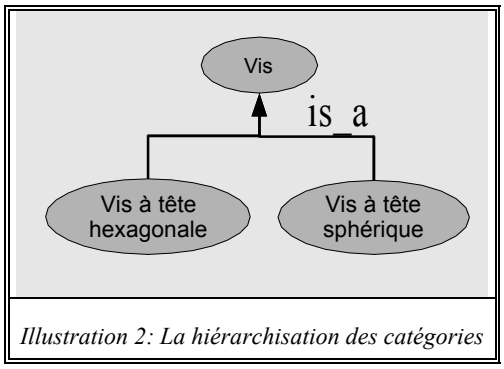


Illustration 2: La hiérarchisation des catégories

La relation ainsi établie entre ces catégories est appelée *est_un* (ou *is_a* en anglais).

Des relations peuvent être établies entre les catégories et il est donc possible d'associer la catégorie '*écrou*' avec '*vis*', par la relation dite d'association *est_monté_avec*.

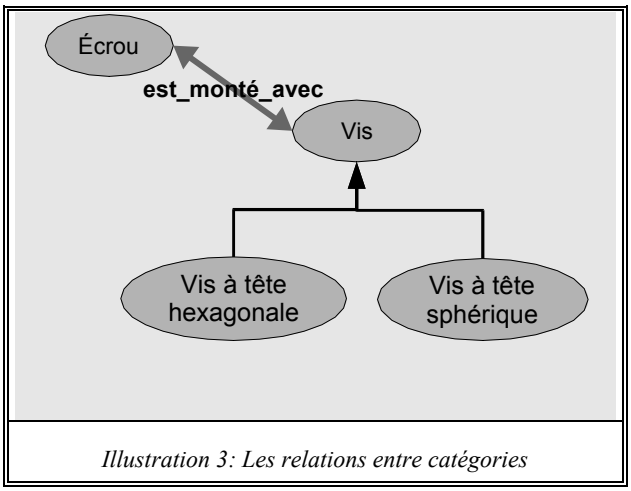


Illustration 3: Les relations entre catégories

2°) La connaissance descriptive

Chaque catégorie, outre son nom, peut-être décrite par des propriétés (ou attributs). Ainsi, la catégorie qui porte déjà le nom de vis est complétée d'un attribut venant donner sa *désignation* et d'un autre pour son *diamètre*. Ils viennent décrire le concept (Illustration 4).

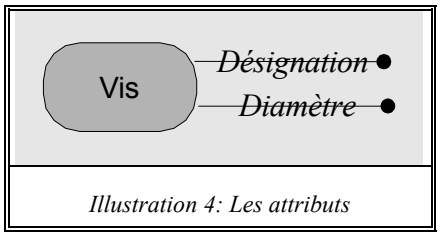


Illustration 4: Les attributs

3°) La connaissance procédurale

Pour compléter la schématisation de la connaissance, il reste à appliquer des règles de cohérence et de raisonnement. A partir de conditions, on affecte ainsi un moyen de vérification sur la catégorie (ainsi que sur ses instances) pour contrôler la conformité de l'information. Les conditions peuvent être de deux types: de type règle (par exemple un prédicat) ou de type fonctionnel.

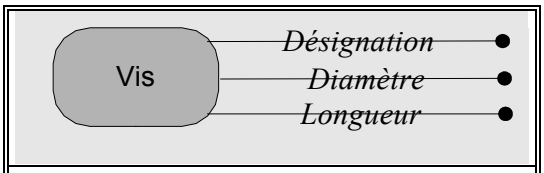


Illustration 5: Les attributs calculés

Dans le cas d'une condition de type *règle*, on peut spécifier que si l'attribut *Designation* n'est pas valué alors la valeur *'0'* doit être affectée à l'attribut *Diametre*.

```
Si non existe(Designation) alors Diametre=0
```

Illustration 6: Raisonnement de type règle

Dans le cas de condition de type 'fonctionnel', la valeur de l'attribut Longueur peut être calculée par la multiplication du *Diamètre* par une valeur (ici *1.5*).

```
Longueur=Diamètre*1.5
```

Illustration 7: Raisonnement de type fonctionnel

4°) Conclusion

Cette démarche de modélisation donnée par KANT apporte une certaine vision du raisonnement humain dans la dans la façon de voir et de représenter un domaine, un univers. Ces fondements vont servir de base pour analyser les formalismes de modélisation objet et principalement le langage *EXPRESS* dans la suite de ce chapitre.

III.La modélisation conceptuelle: le langage EXPRESS

1°) Introduction

Une étude bibliographique nous fournit déjà un large éventail de formalismes: *E/R* [Chen 76], *OMT* [Rumbaugh et al. 91], *NIAM* [Nijssen 81]. La référence en terme de modélisation dans le domaine technique est le langage *EXPRESS* [Schenk et al. 94] [ISO 10303-11 94].

Il fut développé dans les années 80 dans le cadre du projet *STEP* (*Standard for the Exchange of Product model and data*). Son objectif initial est la normalisation de modèles de données pour pouvoir les utiliser sur des plates-formes hétérogènes et surtout pouvoir les échanger et les réinterpréter sans difficultés. Le langage *EXPRESS* est utilisé pour exprimer formellement ces données.

Il s'agit avant tout d'un formalisme textuel développé par Douglas SHENCK. Peter WILSON a par la suite extrapolé une version graphique du formalisme, nommée *EXPRESS-G*, et un langage permettant de décrire des instances conformes à un modèle de données, *EXPRESS-I*.

Le format textuel d'*EXPRESS* en fait toute sa puissance d'expression par rapport aux autres formalismes et facilite les différents *mapping*. Par exemple, la génération des modèles de données dans des langages de programmation (*C++* ou *Java*) se fait de manière automatique. Dans les sections suivantes, nous présentons en détail le langage *EXPRESS*.

2°) Les concepts

Tout comme les autres formalismes vus précédemment, le langage *EXPRESS* dispose d'élément de base servant à la modélisation d'un univers du discours appelé **SCHEMA**: les entités, les attributs et les types.

2.1°) LES ENTITES

Une entité désigne un élément identifiable dans le domaine étudié et il doit avoir une existence autonome. En reprenant les principes vus dans la section Modélisation de la connaissance, la catégorisation est donc possible: chaque catégorie est représentable par une entité. Des relations vont pouvoir être exprimées: l'héritage entre entités et au travers de l'apport d'attributs vont permettre non seulement d'apporter à *EXPRESS* la dimension relative à la connaissance descriptive, mais aussi d'associer les entités entre elles.

En plus de ces concepts, on peut remarquer la notion d'abstraction d'une entité. Dans ce cas, l'entité ne peut-être instanciée qu'au travers de ses sous entités.

2.2°) LES ATTRIBUTS

Chaque entité est décrite par un ensemble d'attributs. Leurs domaines de valeurs vont être des domaines simples (les entiers, les réels,...), des collections (listes, ensembles,...), des entités (on rejoint alors le terme d'association) ou des domaines construits à partir d'un type utilisateur. L'ensemble de ces domaines est décrit plus en détails dans la partie suivante consacrée aux types. Les attributs quant à eux possèdent plusieurs statuts pour une entité.

- Libre: il est indépendant vis à vis des autres attributs et sa valeur (obligatoire) dépend d'une saisie utilisateur (saisie de sa valeur ou de l'entité à laquelle);
- Optionnel (mot clé **OPTIONAL**): l'utilisateur décide de donner ou non une valeur à cet attribut lors de l'instanciation de l'entité qu'il décrit. Si un attribut est déclaré optionnel dans une superclasse, il sera optionnel dans toutes les sous-classes mais peut devenir obligatoire au travers d'une opération de redéfinition;
- Dérivé (mot clé **DERIVE**): la valeur de l'attribut dépend d'autres attributs. Comme tous les autres attributs, il est hérité par toutes les sous-classes de la classe où il est déclaré.

2.3°) LES TYPES

Chaque attribut est associé à un type spécifiant son domaine de valeurs. Les types peuvent être de trois formes: les types simples, les collections et les types utilisateurs.

TYPES SIMPLES

Ils regroupent l'ensemble des types couramment utilisés dans les langages de programmation. On retrouve ainsi les booléens, les réels, les entiers, les chaînes de caractères (Illustration 8).

Types	Domaine de valeurs
BINARY	**0, 1**
BOOLEAN	**TRUE** ou **FALSE**
LOGICAL	**TRUE** , **FALSE** ou **UNKNOWN**
NUMBER	Union des types **INTEGER** et **REAL**
REAL	Nombre réel
INTEGER	Nombre entier
STRING	Chaîne de caractères

Illustration 8: Types simples en EXPRESS

TYPE COLLECTION

Les types peuvent être des collections (ou agrégats) d'entités, de types simples ou même d'autres agrégats. Les collections peuvent être de nature différentes (Illustration 9).

Agrégats	Types
LIST	Collection ordonnée d'éléments avec répétition
SET	Collection non ordonnée d'éléments sans répétition
BAG	Collection non ordonnée d'éléments avec répétition
ARRAY	Tableau de dimension finie

Illustration 9: Types collections en EXPRESS

TYPE UTILISATEUR

La dernière notion apportée par le langage *EXPRESS* est celle des types utilisateur. Il est en effet possible de créer des types à partir de types existants ou d'autres types utilisateurs et, par exemple, créer un type entier en restreignant l'ensemble des valeurs à l'intervalle *[0;20]*, ou encore de définir une chaîne de caractères dont la taille serait limitée à dix caractères.

Deux nouvelles notions sont ajoutées pour la création de type:

- Le type sélectif (mot clé **SELECT**): il permet de définir un domaine de valeurs pour un type comme l'union de domaines de valeurs d'autres types;
- Le type énuméré (mot clé **ENUMERATION**): il permet de définir une énumération de valeurs (encore appelées identificateurs).

2.4°) UN EXEMPLE

Afin de préciser les notions vues précédemment, nous présentons un exemple de modèle de données, un exemple considérant une société vendant des produits à d'autres sociétés.

Une entité *societe* est créée. Elle possède pour attribut un nom de type chaîne de caractères. Elle est déclarée abstraite (**ABSTRACT**) et ne peut donc être instanciée directement. Deux sous-classes *client* et *fournisseur* sont issues de *societe*. Elles héritent donc de l'attribut *nom*. De plus, une entité *article* est déclarée. Elle possède elle aussi des attributs de type simple (*nom* et *prix*) mais elle utilise une association avec une autre entité, *fournisseur*, par l'attribut *fabriquant*. Les deux autres attributs, *disponibilite* et *frais_de_port*, sont associées à des types utilisateurs définis par la suite, respectivement *t_stock* et *t_frais*. Le premier est une énumération de valeurs: un attribut utilisant ce type ne peut avoir que les valeurs *oui* ou *en_commande*. Le deuxième unis deux type totalement différents: les *booléens* et les *réels*. Un attribut utilisant ce type peut donc avoir les valeurs comprises dans l'un ou l'autre de ces deux domaines. L'entité *commande* possède un attribut de type collection, *contenu*, défini comme une liste d'entités *article*. Les articles vont donc être référencés par un agrégat ordonné, sans répétition (**UNIQUE**) avec au minimum un élément. Ainsi lors de la lecture du fichier d'instances, les entités seront restituées dans l'ordre où elles ont été saisies.

```
ENTITY societe
ABSTRACT SUPERTYPE OF (client,fournisseur);
  nom : STRING;
END_ENTITY;

ENTITY client
  SUBTYPE OF (societe);
END_ENTITY;

ENTITY fournisseur
  SUBTYPE OF (societe);
END_ENTITY;

ENTITY article;
  nom : STRING;
  prix : REAL;
  disponibilité : t_stock;
  frais_de_port : t_frais;
  fabriquant : fournisseur;
END_ENTITY;

ENTITY commande;
  contenu : LIST [1:?] OF UNIQUE article;
  expediteur : client;
END_ENTITY;

TYPE t_stock = ENUMERATION(oui,en_commande);
END_TYPE;

TYPE t_frais = SELECT(BOOLEAN,REAL);
END_TYPE;
```

Illustration 10: Un exemple de schéma EXPRESS

3°) Les contraintes

Les contraintes peuvent être appréhendés selon deux grandes familles:

- Les contraintes locales: elles s'appliquent uniquement sur chaque instance de l'entité où elles sont définies (**`WHERE`**);
- Les contraintes globales: elles apportent une vérification globale sur l'ensemble des instances d'un type donné (**`UNIQUE`**, **`INVERSE`** et **`RULE`**).

3.1°) CONTRAINTES LOCALES (`where`)

Ce type de contrainte vient s'appliquer sur les attributs d'une entité mais peut aussi s'appliquer sur un type, lors de la définition d'un type utilisateur. De telles contraintes définissent des prédicats auxquels chaque instance de l'entité où elles sont déclarées doit obéir. Ces prédicats permettent par exemple de limiter la valeur d'un attribut en définissant une échelle de valeurs ou d'éviter une redondance d'information ou encore de refuser la valuation d'un attribut selon certain critère. Dans l'exemple suivant, trois règles locales (*W1*, *W2* et *W3*) sont définies, exprimant chacune les cas envisagés.

```
ENTITY personne;
  nom : STRING;
  age : INTEGER;
  relation_de_travail : BOOLEAN;
  surnom : OPTIONAL STRING;
WHERE
  W1 : SELF\age >0;
  W2 : surnom <>nom
  W3 : NOT EXISTS(surnom) OR NOT(relation_de_travail);
END_ENTITY;
```

*Illustration 11: **WHERE**: contraintes sur les attributs*

- La première règle (*W1*) impose que l'attribut *age* soit supérieur à zéro. Elle vient donc contraindre le domaine de valeur de l'attribut;
- La deuxième (*W2*) impose une différence de valeurs entre les attributs *nom* et *surnom*; une personne ne peut avoir pour surnom son nom;
- La troisième (*W3*) empêche la création d'un attribut surnom si l'attribut *relation_de_travail* est vrai.

3.2°) CONTRAINTES GLOBALES

CONTRAINTES D'UNICITE (Unique)

Cette règle contrôle l'ensemble d'une population pour vérifier l'unicité de la valeur d'un attribut. L'exemple le plus probant est celui du numéro de sécurité sociale d'une personne. Ce numéro est unique et propre à chaque personne. Une déclaration *EXPRESS* donnerait la description suivante:

```
ENTITY personne;
  nom : STRING;
  numero_de_SS : INTEGER;
UNIQUE
  UR1 : numero_de_SS;
END_ENTITY;
```

*Illustration 12: **UNIQUE**: contrainte globale sur l'unicité*

CONTRAINTES DE CARDINALITE (Inverse)

Cette contrainte porte aussi le nom de contrainte d'existence. Elle est introduite par le mot clé **INVERSE** et permet de regrouper au sein d'un attribut une ou un ensemble d'entités faisant référence à l'entité possédant l'attribut. En reprenant l'exemple de la commande (Illustration 10) et en y ajoutant la description d'un article, une commande référence un ou plusieurs articles. Il peut être utile de compléter la définition de l'entité *article* par un attribut référençant toutes les commandes concernant cet article.

```
ENTITY article;
  nom : STRING;
  prix : REAL;
  disponibilité : t_stock;
  frais_de_port : t_frais;
  fabriquant : fournisseur;
INVERSE
  est_commande_par : SET[0:?] OF commande FOR contenu;
END_ENTITY;

ENTITY commande;
  contenu : LIST [1:?] OF article;
  expediteur : client;
END_ENTITY;
```

*Illustration 13: **INVERSE**, contraintes de cardinalité*

Cet attribut **INVERSE** assure une relation de cardinalité entre les entités *article* et *commande*: une commande possède plusieurs articles et un article peut lui même être référencée par plusieurs commandes.

REGLES GLOBALES (Rule)

Les règles globales ne sont pas déclarées au sein des entités comme nous avons pu le voir précédemment pour les règles locales. Elles sont définies séparément et permettent de vérifier des règles qui s'appliquent à un ensemble d'instances d'un type donné. L'exemple suivant (Illustration 14) décrit une règle globale limitant à 50 le nombre d'instance de l'entité *commande*.

```
RULE nombre_max_commande FOR (commande);
WHERE
  max_de_50 : SIZEOF(commande) <=50;
END_RULE;
```

*Illustration 14: **RULE**, contraintes globales*

4°) Les fonctions et les procédures

Le langage *EXPRESS* permet, grâce aux entités et à leurs attributs, de modéliser la connaissance structurelle et la connaissance descriptive. Pour prendre en charge la connaissance procédurale, des fonctions et des procédures prennent en charge la manipulation des attributs et de leurs valeurs possibles.

Les fonctions et les procédures sont introduites respectivement par les mots-clés **FUNCTION** et **PROCEDURE**. Le langage de description s'apparente au langage *PASCAL*. Des variables peuvent être déclarées localement et les structures de contrôle communes (**REPEAT**, **IF**,...) sont présentes. Dans l'exemple suivant, une entité vis possède trois attributs *son_nom*, *sa_longueur* et *son_diametre*. La longueur est calculée (**DERIVE**) à partir du *diametre* par l'appel d'une fonction *donneLaLongueur*. Elle prend l'instance dite courante de l'entité en entrée (**SELF**), calcule la valeur de la longueur à partir du *diamètre* par la multiplication d'un coefficient (*2,5*) et renvoie la valeur résultat.

```
ENTITY Vis;
  son_nom : STRING;
  son_diametre : INTEGER;
DERIVE
  sa_longueur : REAL:=donneLeDiametre(SELF);
END_ENTITY;

FUNCTION donneLeDiametre(la_vis : Vis):REAL;
LOCAL
  le_diametre : INTEGER :=la_vis.son_diametre;
  la_longueur : REAL ;
END_LOCAL;
  la_longueur:=le_diametre*2.5;
  RETURN (la_longueur);
END_FUNCTION;
```

Illustration 15: Utilisation d'une fonction

Le langage *EXPRESS* propose tout un ensemble prédéfinis: **QUERY** (requête sur les instances), **SIZEOF** (taille d'une collection).

5°) Le fichier physique

Une population de données conforme à un modèle de données *EXPRESS* est représentée au sein d'un fichier physique. Celui-ci est un simple fichier caractère répondant à une syntaxe stricte [ISO10303-21] précisant en particulier les règles de traduction de définitions *EXPRESS* pour décrire des instances.

Reprenons le modèle décrit précédemment (Illustration 10), l'entité *vis* étant complétée d'un attribut optionnel *avec_tete* et d'une association avec une autre entité, *fournisseur*, elle-même décrite par une association avec une nouvelle entité, *adresse*.

Ce schéma peut être instancié avec un fichier physique (Illustration 17) avec deux instances de l'entité *vis*, plusieurs instances de l'entité *fournisseur* et leurs entités *adresse* associées.

Une instance fait référence à l'entité dont elle est issue par son nom et un identificateur (**#**), suivi d'un nombre unique et contient la suite de valeurs de ses attributs entre parenthèses. Chaque attribut respecte son type défini dans le modèle. Les attributs dérivés, tout comme les attributs inverse sont calculés par le système receveur et ne sont pas présents dans le fichier. Un agrégat se présente sous la forme d'une liste d'attributs (dans l'exemple il s'agit de référence à d'autre entités pour l'entité *vis* référencée *#1*) comprise entre parenthèses. Un attribut déclaré optionnel ne possédant aucune valeur est représenté par *$*.

Un tel fichier permet aussi à deux systèmes informatiques d'échanger sans aucune ambiguïté des populations d'instances conforme à un modèle un modèle *EXPRESS*.

```
ENTITY vis;
  son_nom : STRING;
  son_diametre : INTEGER;
  avec_tete : OPTIONAL BOOLEAN;
  ses_fournisseurs : LIST OF fournisseur;
DERIVE
  sa_longueur : REAL:=donneLaLongueur(SELF);
END_ENTITY;

FUNCTION donneLaLongueur(la_vis : vis):REAL;
LOCAL
  le_diametre : INTEGER :=la_vis.son_diametre;
  la_longueur : REAL;
END_LOCAL;
  la_longueur:=le_diametre*2.5;
  RETURN (la_longueur);
END_FUNCTION;

ENTITY fournisseur;
  son_nom : STRING;
  son_adresse : adresse;
END_ENTITY;

ENTITY adresse;
  son_numero : OPTIONAL STRING;
  sa_rue : STRING;
  son_code_postal : INTEGER;
  sa_ville : STRING;
END_ENTITY;
```

Illustration 16: Exemple d'un schéma EXPRESS complet

```
#1=VIS('CHC',10,.T.,(#10,#11,#12));
#2=VIS('CS',8,.T.,(#12));
#10=FOURNISSEUR('ACTON',#21);
#11=FOURNISSEUR('TREVIS',#22);
#12=FOURNISSEUR('FTV',#23);
#21=ADRESSE('3BIS','Rue du faubourg de Couzon',42152,'L Horme');
#22=ADRESSE($,'Zone Artisanale d Illange,57972,'Yutz Cedex');
#23=ADRESSE($,'Z.I. Les Forges',08320,'Vireux-Molhain');
```

Illustration 17: Exemple d'un fichier physique EXPRESS

IV.Conclusion

Avec le pouvoir d'expression et les nombreuses fonctions que propose le langage *EXPRESS*, la modélisation de la connaissance selon les trois points de vue (structurel, descriptif et procédural) est totalement possible. Ajouter des fonctions au moyen d'un langage procédural proche du *PASCAL* vient offrir aux possibilités déjà présentes un pouvoir d'expression important. C'est avant tout pour ce pouvoir d'expression que ce langage a été créé pour le projet *STEP* et la modélisation des

produits mais est aussi utilisé travers d'autres projets similaires tels que *PLIB* pour la modélisation des composants.

Il faut néanmoins remarquer que le fichier d'échange de données, le fichier physique, ne dispose pas d'un format compréhensible par l'homme. L'information véhiculée par ces fichiers étant assez éclatée, il faut un certain temps pour la rendre comprendre et assimiler les différentes références entre entités. Il paraît donc utile d'étudier les langages de structuration documentaire et principalement leurs avantages pour faire circuler l'information traitable par les machines tout en la gardant intelligible pour l'être humain.

Les langages de structuration de l'Information

I.Introduction

Le chapitre précédent a montré la puissance d'expression des langages de modélisation de données et en particulier le langage *EXPRESS*. Bien que les différentes avancées aient permis d'approcher le raisonnement humain pour représenter l'univers d'un discours donné, il faut remarquer que le fichier d'échange d'informations n'est exploitable que par des machines. Ce chapitre est consacré à la présentation des langages dits de structuration documentaire et principalement à *XML* et *XML Schema*. L'un des avantages de ce type de langage est de pouvoir véhiculer de l'information entre machines tout en gardant une facilité de lecture pour l'homme, d'une part parce que l'information est structurée et explicitée, et, d'autre part, au travers de procédures de stylage.

Le contenu de ce chapitre sera donc le suivant. La première partie présentera le concept sous-jacent à la structuration de l'information: le concept de balisage sera ainsi introduit et son utilisation dans le cadre du langage *XML* sera présentée. La deuxième partie sera quant à elle consacrée à la présentation d'*XML Schema*, un langage de description de documents *XML* lui même défini en *XML*.

II.Les langages de balisage

1°) Introduction au balisage

Les balises servent à marquer l'information contenue dans une chaîne de caractères. En effet, par exemple, lors de l'énoncé d'une phrase ou d'une description d'un objet, l'être humain analyse la phrase et repère les différentes parties de l'information en les comparant à des références provenant de sa mémoire. Par exemple, la phrase suivante:

Une voiture Peugeot 206 grise avec un moteur 2 litres, une boite 5 vitesses

Illustration 18: Description d'une voiture

En analysant la phrase, l'être humain va comprendre chaque partie en se référant à des connaissances acquises et voir qu'avant tout elle désigne un objet appelé voiture et que l'ensemble des autres données vient compléter la description. La représentation sous forme d'ensembles (Illustration 19) donne un autre point de vue.

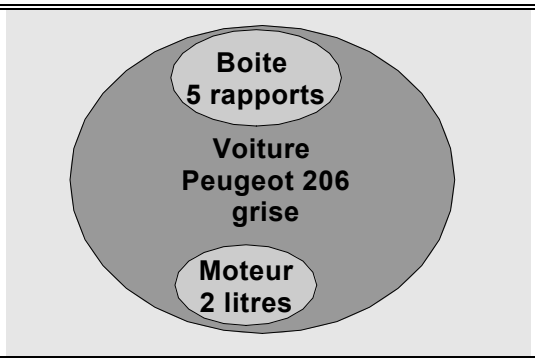


Illustration 19: La structuration sous forme d'ensembles

En revanche, pour un ordinateur, la phrase prise telle quelle ne pourra être interprétée que comme une simple chaîne de caractères. Les catégories et les différents liens d'association mis en jeu ne seront pas pris en compte. Il faut donc trouver une méthode pour *marquer* l'information, sans pour autant la rendre incompréhensible pour l'homme -c'est à dire l'éclater. La méthode utilisée par le langage *XML* consiste à mettre l'information entre des *balises*, l'une dite *entrante* et l'autre dite *fermante*. A partir de l'exemple précédant (Illustration 18) on souhaite donner une sémantique à la phrase. Ainsi, des balises **<voiture>** et **</voiture>** vont être insérer à chaque extrémité de la phrase pour la qualifier.

```
Une
<voiture>
  Peugeot 206 grise
  avec un moteur 2 litres,
   une boite 5 vitesses et,
</voiture>
```

Illustration 20: Exemple de la voiture sous forme XML

La description de la voiture est affinée par un ensemble de caractéristiques: sa couleur, sa marque et son modèle. De plus, certaines parties du véhicule peuvent être considérées comme des éléments ayant une existence propre. Ces informations peuvent être elles aussi balisées.

```
<VOITURE>
   <sa_marque>Peugeot</sa_marque>
   <son_modèle>206</son_modele>
   <sa_couleur>grise</sa_couleur>
   <son_moteur>
   <MOTEUR>
   <sa_cylindree>2 litres</sa_cylindree>,
   </MOTEUR>
   </son_moteur>
   <sa_boite>
   <BOITE>
   <ses_rapports>5 vitesses<ses_rapports>
   </BOITE>
   </sa_boite>
</VOITURE>
```

Illustration 21: Exemple complet de la voiture sous forme XML

En appliquant des balises, l'information peut maintenant être identifié par un programme et il peut être possible de reconstituer les objets et les relations qui les unissent. Notons enfin que ce document peut aussi être représenté sous la forme d'un arbre.

2°) Historique du domaine documentaire

Initialement, les documents utilisaient des formats propriétaires. Le but était de produire beaucoup de texte à faible coût. L'utilisation de la forme électronique permettait un échange rapide par les voies de télécommunication avec un gain de place pour le stockage (plus besoin de s'embarrasser avec des armoires remplies de documentation !). Cependant, l'échange de documents sous la forme électronique imposait l'utilisation des mêmes outils entre les personnes concernées par la consultation ou la modification. Dans le cas contraire, un retour au format papier était nécessaire. Il faut remarquer que dans la majorité des cas, ces documents étaient de simples recopies sans grande utilisation des pouvoirs du format électronique.

Normalisé standard *ISO8879* en 1986, *SGML* pour *Standard Generalized Markup Language* peut être considéré comme la première tentative de création de documents purement électroniques et non plus de simples copies de papier. La séparation entre les données et le style est l'idée principale. Dans un soucis de production de documents à faible coût, *SGML* propose de factoriser l'information en laissant les données dans un document et en produisant plusieurs documents de style (autant que de représentations souhaitées).

Pour une meilleure compréhension, prenons l'exemple d'une documentation de maintenance aéronautique. Les avions sont conçus et fabriqués par un très faible nombre de constructeurs. Cependant, ces avions sont vendus à plusieurs sociétés de transport souvent en charge d'effectuer la maintenance de leurs appareils. Pour la documentation, l'ensemble des données des opérations de maintenance (méthodes, pièces et outils nécessaires) sont représentées dans un document balisé au format *SGML* (identique à Illustration 20 et Illustration 21). Ensuite des feuilles de style permettent de présenter l'information en fonction du format papier ou des normes de présentation utilisées dans les sociétés de transport. Il y a donc bien une séparation entre les données disponibles dans un document et les feuilles de style destinées à l'affichage. Le nombre de documents est ainsi largement réduit: la taille de l'ensemble feuilles de styles-documents de données est inférieure à la taille de l'ensemble des documents réalisés pour chaque sociétés de transport.

L'utilisation d'un système de balisage assure la pérennité des documents. En effet, les versions successives des logiciels propriétaires rendent parfois les documents incompatibles entre versions, voir obsolètes. Or les documents *SGML* (données et feuilles de style) sont intégralement rédigés dans des fichiers textes. Le marquage utilisé dans un document *SGML* est normalisé en *ISO* et les programmes permettant le traitement des documents *SGML* doivent se plier aux règles définies sous peine de voir le document comme non valide. Cependant, la complexité du langage l'a restreint dans sa diffusion et seuls des organismes ayant des moyens financiers important (grosses sociétés ou secteur public) pouvaient s'investir dans son utilisation.

Au début des années 1990, le *Web* fait sa percée. Pour permettre l'échange d'information, le choix se porte sur un langage créé en 1990: *HTML*. Il s'agit en fait d'une application spécifique de *SGML*. Une *DTD SGML* (*DTD* pour *Définition de Type de Document*) fournit un ensemble de règles définissant les noms des balises autorisées ainsi que leur enchaînement. Toutefois, les documents représentés en *HTML* ne possèdent qu'une orientation style. Les balises disponibles permettent de définir comment afficher l'information mais ne viennent en aucun cas qualifier la sémantique de cette information. Trouvant le jeu de balises disponibles pour le formatage insuffisant, les concepteurs de navigateurs ont créés eux-mêmes des balises supplémentaires pour l'enrichir. Cependant, ces balises étaient incompatibles d'un navigateur à l'autre.

De plus, la démocratisation de l'Internet en fait non plus seulement un média pour l'échange de documents mais aussi pour l'échange de données. Il existe un besoin d'appliquer un sens, une sémantique, à l'information. Devant la limitation du langage *HTML* (liée à l'ensemble fini de balises orientées style qui le définissent), il était nécessaire d'ouvrir Internet à la définition de nouveaux

jeux de balises plus orientés données. *SGML* étant trop complexe, le *W3C* s'est concentré vers une restriction de ce langage portant le nom d'*XML*, permettant ainsi de définir de nouveaux ensembles de balises (ou applications *XML*) pour l'échange de données sur le *web*.

3°) Le langage XML

XML (*eXtensible Markup Language*) [Bray et al. 00], devrait permettre un retour vers la notion de données pour les documents du *web*. Afin de présenter l'intérêt d'utiliser *XML* pour l'échange de données, effectuons une comparaison avec le langage *HTML*.

3.1°) LES LIMITES D'HTML

Pour démontrer les limitations des documents *HTML* du point de vue sémantique, un exemple de la commande de composants mécaniques va être étudié. Il s'agit d'une commande destinée à afficher un ensemble de composants avec certaines propriétés (Illustration 22).

```
<HTML>
<HEAD>
   <TITLE>Commande de pieces</TITLE>
</HEAD>
<BODY>
<TABLE>
   <THEAD>
   <TR>
   <TD>Ref.</TD>
   <TD>Designation</TD>
   <TD>Prix</TD>
   </TR>
   </THEAD>
   <TBODY>
   <TR>
   <TD>10FR85</TD>
   <TD>Roulement à Billes</TD>
   <TD>15,20</TD>
   </TR>
   <TR>
   <TD>10FX85</TD>
   <TD>Coussinet</TD>
   <TD>12,40</TD>
   </TR>
   </TBODY>
</TABLE>
</BODY>
</HTML>
```

Ref.	Designation	Prix
10FR85	Roulement à Billes	15,20
10FX85	Coussinet	12,40

Illustration 22: Commande de composants sous forme HTML

Après avoir déclarer l'en-tête du document, (les balises **<HEAD>**), le contenu (**<BODY>**) du document décrit un tableau. On peut remarquer que ce langage est orienté style. Le navigateur ne permet pas de comprendre l'information et donc que *15,20* indique un prix et *10FR85* une référence catalogue. L'idéal est d'avoir un document où la commande est décrite explicitement (Illustration 23). Chaque composant est identifiable par les balises **<COMPOSANT>** ainsi que ses propriétés: sa référence (**<REFERENCE>**), sa désignation (**<DESIGNATION>**) et son prix (**<PRIX>**). Il ne reste plus qu'à appliquer une feuille de style pour afficher cet ensemble de données sous un format plus agréable.

```
<?xml version='1.0'>
<COMMANDE>
   <COMPOSANT>
   <REFERENCE>10FR85</REFERENCE>
   <DESIGNATION>Roulement à Billes</DESIGNATION>
   <PRIX>15,20</PRIX>
   </COMPOSANT>
   <COMPOSANT>
   <REFERENCE>10FX85</REFERENCE>
   <DESIGNATION>Coussinet</DESIGNATION>
   <PRIX>12,40</PRIX>
   </COMPOSANT>
</COMMANDE>
```

Illustration 23: Commande de composants sous forme XML

3.2°) LES REGLES S'APPLIQUANT A UN DOCUMENT XML

L'information contenue dans un document XML possède plusieurs formes:

- Encadré par des balises ouvrantes (**<COMPOSANT>**) et fermantes (**</COMPOSANT**>): elle porte le nom **element**. Ces balises peuvent contenir d'autres balises (appelées alors sous éléments). Aucun chevauchement n'est possible: une balise fermante ne peut intervenir qu'une fois que toutes les balises ouvertes, depuis la déclaration de la balise ouvrante du même nom, soient fermées. Des balises peuvent être vides (**<BALISE_VIDE/>**);
- Inclue dans une balise (**<PRIX DEVISE='FRANC'>**): elle porte le nom **attribute**;
- Remplacé par une abréviation ou un raccourci: elle porte le nom **entity**. Il est alors possible de l'utiliser de façon répétitive sans avoir à ressaisir la chaîne de caractères.

Si c'est principales règles syntaxiques sont respectées dans un document XML, ce document est dit *bien formé*. Il peut alors être traité tel quel et éventuellement, si une feuille de style lui est associé, visualisé dans un navigateur.

De plus il est aussi possible de référencer une *DTD* (*Définition de Type de Document*) dans le document XML. Celle-ci vient définir la structure arborescente à laquelle le document XML doit se plier en donnant le nom des balises autorisées, leur imbrication en indiquant l'ordre d'apparition, leur contenu. Si un document XML respecte sa *DTD*, il est dit *valide*. Le concept de *DTD* sera vu plus en détails par la suite.

3.3°) DTD: LA DEFINITION DE TYPE DE DOCUMENT

Pour introduire le concept de *DTD*, prenons l'exemple d'un document XML destiné à une gestion de contacts. Chaque contact ou personne est caractérisé par des informations telles que son nom et son prénom.

```
<CARNET>
   <CONTACT>
   <NOM>Ramirez</NOM>
   <PRENOM>Julio</PRENOM>
   </CONTACT>
   <CONTACT>
   <NOM>Chochon</NOM>
   </CONTACT>
   <CONTACT>
   <PRENOM>Ilario</PRENOM>
   <NOM>De La Vega</NOM>
   </CONTACT>
</CARNET>
```

Illustration 24 : Exemple de carnet de contacts

Dans l'exemple précédant (Illustration 24), trois contacts sont définis dans le carnet. Cependant la représentation des contacts n'est pas similaire pour les trois. Le premier inclue les balises **<nom>** et **<prenom>**, mais le deuxième possède uniquement **<nom>**. Quant au troisième, les balises ne sont pas dans le même ordre. Il peut être souhaitable par soucis d'affichage ou tout simplement par soucis de validité de l'information d'imposer la présence de certaines balises ainsi que leur ordre d'apparition. C'est le rôle d'une *DTD*. Ainsi voici la *DTD* pouvant être réalisée selon les exigences précitées et un document XML valide par rapport à cette *DTD*. (Illustration 25)

```
<!ELEMENT carnet (contact+)>
<!ELEMENT contact (nom,prenom)>
<!ELEMENT nom (#PCDATA)>
<!ELEMENT prenom (#PCDATA)>
```

```
<carnet>
   <contact>
   <nom>Ramirez</nom>
   <prenom>Julio</prenom>
   </contact>
   <contact>
   <nom>Chochon</nom>
   <prenom>Jerome</prenom>
   </contact>
   <contact>
   <nom>De La Vega</nom>
   <prenom>Ilario</prenom>
   </contact>
</carnet>
```

Illustration 25: Exemple de carnet de contacts avec DTD

Le terme **ELEMENT** désigne la déclaration d'une balise. Ainsi, quatre balises sont déclarées: *carnet*, *contact*, *nom* et *prenom*. La suite de la déclaration (ce qui est écrit entre parenthèses) indique le contenu listé de la balise. Ainsi la balise *carnet* contient la balise *contact*. Une indication sur le nombre d'occurrences vient compléter la déclaration: le signe **+** indique que la balise contact sera présente une ou plusieurs fois dans la balise *carnet*. La balise *contact* est suivie des balises *nom* et *prenom* séparé par, Cette ponctuation indique une *séquence* et donc ces balises apparaîtront dans cet ordre. Le terme **#PCDATA** indique que le type du contenu sera une chaîne de caractères.

Pourtant les *DTD* posent plusieurs problèmes aux utilisateurs *XML*:

- Le premier est qu'elles ne sont pas écrites en *XML* et nécessitent des outils supplémentaires pour vérifier leur cohérence générale;
- Le second se situe au niveau de leur pouvoir d'expression insuffisant pour modéliser des données destinées aux bases de données. En effet, les *DTD*s ne permettent que de vérifier la structure d'un document XML. Elles ne permettent pas de contrôler la nature des contenus informationnels (seules les chaîne de caractères sont utilisables). Elles ne sont donc pas assez riches pour décrire non plus des structures documentaires, mais plutôt des modèles de données structurés.

III.Les schémas XML

De nombreuses propositions ont été faites pour remplacer les *DTD*s. Elles sont regroupées sous le nom de *schéma* et imposent d'être écrites en *XML* pour en permettre leur exploitation avec les mêmes outils que les documents XML dont elles définissent le modèle.

1°) Comparatif

Le nombre de proposition des schémas est important et impose une étude préliminaire avant de savoir lequel correspond vraiment aux besoins. Si ils ont pour but de remplacer les *DTD XML*, ils possèdent une approche pas toujours similaire et une philosophie différente pour modéliser un document voire, pour les plus expressifs, un ensemble de données. Comparer leurs avantages et leurs inconvénients entre eux, semble être une première démarche pour sélectionner un schéma. Mais évaluer leur pouvoir d'expression par rapport aux de modélisation conceptuelle permet de connaître les réelles avancées par rapport à ces formalismes de modélisation établis depuis plusieurs années.

Dans la suite de cette partie, après avoir présenté les différentes propositions de *schémas XML*, deux résultats de travaux portant sur la comparaison des propositions seront évoquées.

1.1°) LES DIFFERENTES PROPOSITIONS

Un grand nombre de proposition de *schéma XML* sont aujourd'hui disponibles, toutes n'étant pas complètement arrêtées et implémentées. Elles sont présentées ci-après:

XML-DATA

XML-Data [Layman et al. 98] est une proposition provenant des sociétés *Microsoft, ArborTExt* et *DataChannel*. Cette spécification est arrivée avant la spécification définitive de *XML 1.0*. Un schéma *XML-Data* reprend les fonctions d'une *DTD* mais y ajoute la dimension typage des données. Utilisé largement dans les outils *Microsoft*, *XML-Data* a servi de base à l'élaboration des autres schémas et semble aujourd'hui abandonné.

DCD

DCD [Bray et al. 98] pour *Document Content Description* est son plus récent successeur. Proposé par *Textuality*, *Microsoft* et *IBM*, il propose d'utiliser exactement les mêmes fonctions que les *DTD* mais en y ajoutant la définition de types de données et une réutilisation de définitions d'éléments d'un schéma dans un autre schéma. Il permet aussi un contrôle sur les valeurs par défauts, minimum et maximum des éléments et attributs. L'héritage entre éléments n'est pas disponible et la conversion directe entre une *DTD* et un schéma n'est pas finalisée.

SOX

Schema for Object-oriented XML (*SOX*) [Davidson et al. 99] propose une approche programmation orientée objet pour la création de schémas. En plus de disposer des mêmes fonctions qu'une *DTD*, il permet l'héritage entre éléments, entre attributs et propose les mêmes fonctions que *DCD* tel que le typage de données et la réutilisation de schémas. Il permet aussi un contrôle sur les instances d'élément en contraignant le nombre d'occurrences. Cependant, la conversion directe *DTD-Schéma SOX* n'est pas encore finalisée.

DDML

Initialement nommé *XSchema* puis rebaptisé *DDML* [Bourret et al. 99] pour *Document Description Markup Language* est un schéma développé par une liste de diffusion *XML-Dev*. Défini comme simple d'accès, il reprend les fonctions des *DTD* mais ne prend pas en charge les types de données

XML SCHEMA

XML Schema [Fallside 01], [Thompson et al. 01], [Biron et al. 01] regroupe les multiples avantages des schémas précédents. Tout en offrant le même pouvoir d'expression qu'une *DTD*, il dispose en plus d'un grand nombre de types simples déjà définis. Il permet aussi la réutilisation d'éléments entre schémas et offre des mécanismes d'héritage par extension et restriction aussi bien pour des types simples que pour des types complexes Des contraintes sont disponibles pour contrôler les occurrences lors de l'instanciation, ou encore pour vérifier l'unicité d'un attribut.

XDR

XDR pour *XML-Data Reduced* [Frankston et al. 98] est une réactualisation de *XML-Data* auquel des fonctions de *DCD* ont été ajoutées.

SCHEMATRON

Schematron [Jeliffe 00] possède une place à part dans le monde des schémas. Il permet de créer des schémas en utilisant des règles ou des contraintes exprimés à partir d'*XPath* [Clark et al. 99].

DSD

DSD [Klarlund et al. 99] [Klarlund et al. 00] permet la description des éléments et attributs en y ajoutant des expressions de contraintes. Ce schéma est en cours d'écriture et donc pas encore finalisé.

1.2°) PREMIERE ETUDE: APPROCHE SPECIFICATION DE DONNEES POUR LES SCHEMAS XML

Une étude provenant de [Sardet 99] avait déjà mis en confrontation les schémas *XML-Data*, *RDF Schema*, *DCD* et *SOX*. Le langage *EXPRESS* servait de référence pour les différents concepts évalués.

Critères	XML Data	DCD	SOX	RDF Schema
Classification	X	X	X	X
Généralisation/Spécialisation	X	+ou-	X	X
Héritage	X	+ou-	X	X
Polymorphisme	X	+ou-	X	X
Typage Simple/Pointeur	X / -	X / -	X / -	X / -
Cardinalité	+ou-	+ou-	X	
Unicité	-	+ou-	-	
Ensembliste	-	-	-	
Assertionnelle	-	-	-	
Fonctionnelle	-	-	-	

X: prend en charge le critère
+ou-: répond partiellement au critère
-: ne prend par en charge le critère

Illustration 26: Comparatif des schémas XML -1

Le comparatif (Récapitulatif sommaire en Illustration 26) avait abouti sur une très grande faiblesse de ces nouveaux schémas pour exprimer des concepts proches du langage de modélisation *EXPRESS* ou plus généralement des langages de modélisation conceptuelle orientés objet. De plus très peu d'outils permettaient de valider les schémas. Il était difficilement possible de savoir si les schémas étaient conformes à la spécification et encore plus difficile de vérifier un document par rapport à son schéma.

1.3°) DEUXIEME ETUDE: POUVOIRS D'EXPRESSION DES SCHEMAS XML

Dans cette seconde étude [Lee et al. 00] menée deux années plus tard, six schémas sont comparés (Illustration 27). Il s'agit des *DTD XML*, pour servir de référence de comparaison, et des schémas *XML Schema*, *XDR*, *SOX*, *Schematron* et *DSD*. Durant ces années, de nouveaux schémas sont apparus pour remplacer les anciens sur les points où ils étaient en défaut.

Critères	DTD XML 1.0	XML Schema 1.0	XDR 1.0	SOX 2.0	Schematron 1.4	DSD 1.0
Schema						
Syntaxe en XML	-	+	+	+	+	+
Espace de noms	-	+	+	+	+	-
Inclusion (INCLUDE)	-	+	-	+	-	+
Importation (IMPORT)	-	+	-	+	-	-
Datatype						
Types atomiques	10	37	33	17	0	0
Types utilisateurs	-	+	-	+	-	+
Restriction du domaine	-	+	-	+ou-	+	+
Application du null	-	+	-	+	-	-
Attributs						
Valeur de défaut	+	+	+	+	-	+
Possibilité de choix	-	-	-	-	+	+
Optionnel/Obligatoire	+	+	+	+	+	+
Restriction du domaine	+ou-	+	+ou-	+ou-	+	+
Règles globales	-	-	-	-	+	+
Éléments						
Valeur de défaut	-	+ou-	-	-	-	+
Modèle de contenu	+	+	+	+ou-	+	+
Séquence ordonnée	+	+	+	+	+	+
Séquence non-ordonnée	-	+	+	-	+	+
Choix	+	+	+	+	+	+
Occurrence min./max	+ou-	+	+	+	+	+ou-
Modèle ouvert	-	-	+	-	+	-
Règles globales	-	-	-	-	+	+
Héritage pour type...						
simple extension	-	-	-	-	-	-
simple restriction	-	+	-	+	-	-
complexe extension	-	+	-	+	-	-
complexe restriction	-	+	-	-	-	-
Base de données						
Unicité sur attributs	+	+	+	+	+	+
Unicité sur autres	-	+	+ou-	-	+	-
Clé pour attributs	-	+	-	-	+	-
Clé pour autres	-	+	-	-	+	-
Ref. pour attributs	+ou-	+	+ou-	+ou-	+	+
Ref. pour autres	-	+	-	-	-	+
Divers						
Contrainte dynamique	-	-	-	-	+	-
Définition de version	-	-	-	-	-	+
Documentation	-	+	-	+	+	+
Utilisation d'HTML	-	+	-	+	+ou-	+
Meta Langage	-	+ou-	-	-	+ou-	+

X: prend en charge le critère
+ou-: répond partiellement au critère
-: ne prend par en charge le critère

Illustration 27: Comparatif des schémas XML -2

Les schémas se divisent en catégories différentes.

Certains tels que *SOX* ou *XML Schema* ont évolué dans le domaine de la modélisation pour approcher le pouvoir d'expression des langages de programmation. Les schémas ne définissent pas seulement des documents mais ils peuvent représenter un ensemble de données, une population, sous forme de concepts avec leurs liens (association, héritage) et des attributs devenant de plus en plus précis dans leur définition de domaine de valeurs.

D'autres, tel que *Schematron*, apporte une approche sous forme de règles pour la définition de modèle, validant les attributs par des conditions. Il ne permet cependant pas de définir un modèle de structure sous forme explicite et oblige donc à une modélisation avec un autre schéma pour parfaire ce point.

DSD se situe à mi-chemin entre ces deux principes, permettant de modéliser les connaissances structurelles et descriptives sans pour autant apporter la notion d'héritage. De plus, il apporte une meilleure prise en charge des contraintes pour approcher la connaissance procédurale. Cependant, ce schéma n'est aujourd'hui pas encore finalisé.

L'auteur classe les six schémas suivant deux axes orthogonaux (Illustration 28).

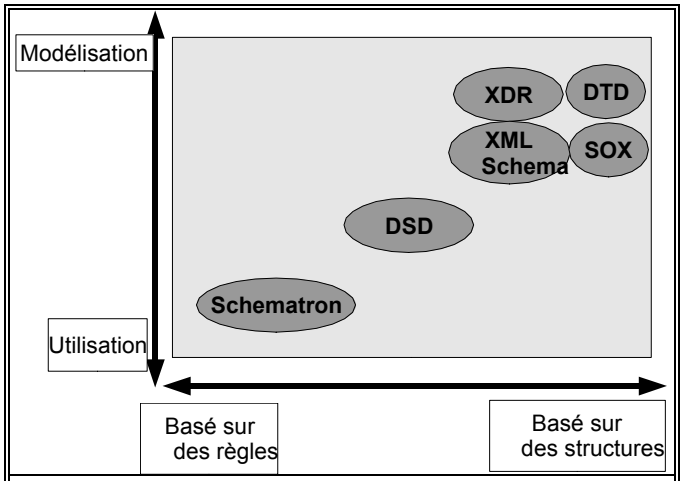


Illustration 28: Classification des schémas XML

Nous souhaitons modéliser des structures de données au moyen de schémas XML. Dans ce cas, seuls *SOX*, *XDR* et *XML Schema* le permettent. Le dernier possédant l'appui du *W3C* ainsi que de nombreux industriels, il s'impose donc dans le comparatif. Ce schéma va être développé dans la suite du chapitre.

2°) XML Schema

Un schéma désigne souvent un modèle et le document XML une instance de ce modèle. Un document XML comprend un jeu de balises qualifiées par un nom et éventuellement par des attributs. Les balises possèdent pour contenu d'autres balises ou éventuellement des chaînes de caractères. C'est la description de ces concepts au travers d'un schéma XML qui est développée dans les sections suivantes.

2.1°) LES BRIQUES DE BASE: ELEMENTS ET ATTRIBUTS

Une balise est définie au travers d'un élément (mot clé **ELEMENT**). Celui-ci est qualifié par un nom et associé à un type qui définit le contenu de la balise résultante. Un élément ne contenant pas d'autres éléments ou attributs est déclaré de type simple. Dès lors qu'il possède des attributs ou d'autres éléments, (voire les deux), il est considéré comme étant de type complexe.

Les attributs sont des caractéristiques qui précisent le sens d'une balise. En *XML Schema*,un attribut est définie par le mot clé **ATTRIBUTE** et est caractérisé par son nom, son type qui ne peut être que simple, et des contraintes d'occurrences exprimées par les attributs optionnels **use** et **value** (Illustration 29).

Valeur de USE	Valeur de VALUE	Commentaire
required		L'attribut doit apparaître dans le fichier d'instance.
required	15	L'attribut doit apparaître dans le fichier d'instance avec pour valeur 15
optional		L'attribut peut apparaître
fixed	15	L'attribut peut apparaître et dans ce cas sa valeur est obligatoirement 15
default	15	L'attribut peut apparaître. Si il apparaît sa valeur est celle spécifiée sinon c'est 15
prohibited		L'attribut ne doit pas apparaître

Illustration 29: Occurrences pour les attributs

Les attributs peuvent être factorisés au sein d'un groupe d'attributs pour être utilisés par plusieurs éléments. Un groupe d'attributs est identifié par un nom. Il sera par la suite référencé dans les éléments qui vont les utiliser par la suite au travers de ce nom.

L'Illustration 30 présente le schéma *XML Schema* correspondant à une gestion de contacts et un exemple de document XML respectant les règles structurelles définies dans ce schéma. Les éléments *contact* et *societe* sont chacun liés à un type, respectivement *t_contact* et *t_societe*. Ils possèdent les mêmes attributs date et auteur qui sont respectivement de type **date** et **string**. Dans la partie droite, un document instance, conforme aux règles définies dans le schéma *XML Schema*, est présenté. On remarque bien que les éléments sont instanciés en balise et que les attributs prennent place dans la balise pour la qualifier.

```
<SCHEMA>

<ELEMENT NAME='contact' TYPE='t_contact'/>
<COMPLEXTYPE NAME='t_contact'/>
   <ATTRIBUTE NAME='date' TYPE='DATE'/>
   <ATTRIBUTE NAME='auteur' TYPE='STRING'/>
</COMPLEXTYPE>

<ELEMENT NAME='societe' TYPE='t_societe'/>
<COMPLEXTYPE NAME='t_societe'/>
   <ATTRIBUTE NAME='date' TYPE='DATE'/>
   <ATTRIBUTE NAME='auteur' TYPE='STRING'/>
</COMPLEXTYPE>

</SCHEMA>
```

```
<carnet>

<contact date='28/05/00' auteur='chochon'>
</contact>

<contact date='28/05/00' auteur='chochon'>
</contact>

<societe date='28/05/00' auteur='chochon'>
</societe>

</carnet>
```

Illustration 30: Le carnet d'adresse en XML Schema: schéma et instances

Les attributs auraient pu être regroupés sous la forme d'un groupe d'attributs (Illustration 31).

```
<ELEMENT NAME='contact' TYPE='t_contact'/>
<COMPLEXTYPE NAME='t_contact'/>
   <ATTRIBUTGROUP REF='maj'/>
</COMPLEXTYPE>

<ELEMENT NAME='societe' TYPE='t_societe'/>
<COMPLEXTYPE NAME='t_societe'/>
   <ATTRIBUTGROUP REF='maj'/>
</COMPLEXTYPE>

<ATTRIBUTGROUP NAME='maj'>
   <ATTRIBUTE NAME='date' TYPE='DATE'/>
   <ATTRIBUTE NAME='auteur' TYPE='STRING'/>
</ATTRIBUTEGROUP>
```

Illustration 31: Utilisation de groupe d'attributs

2.2°) LES TYPES

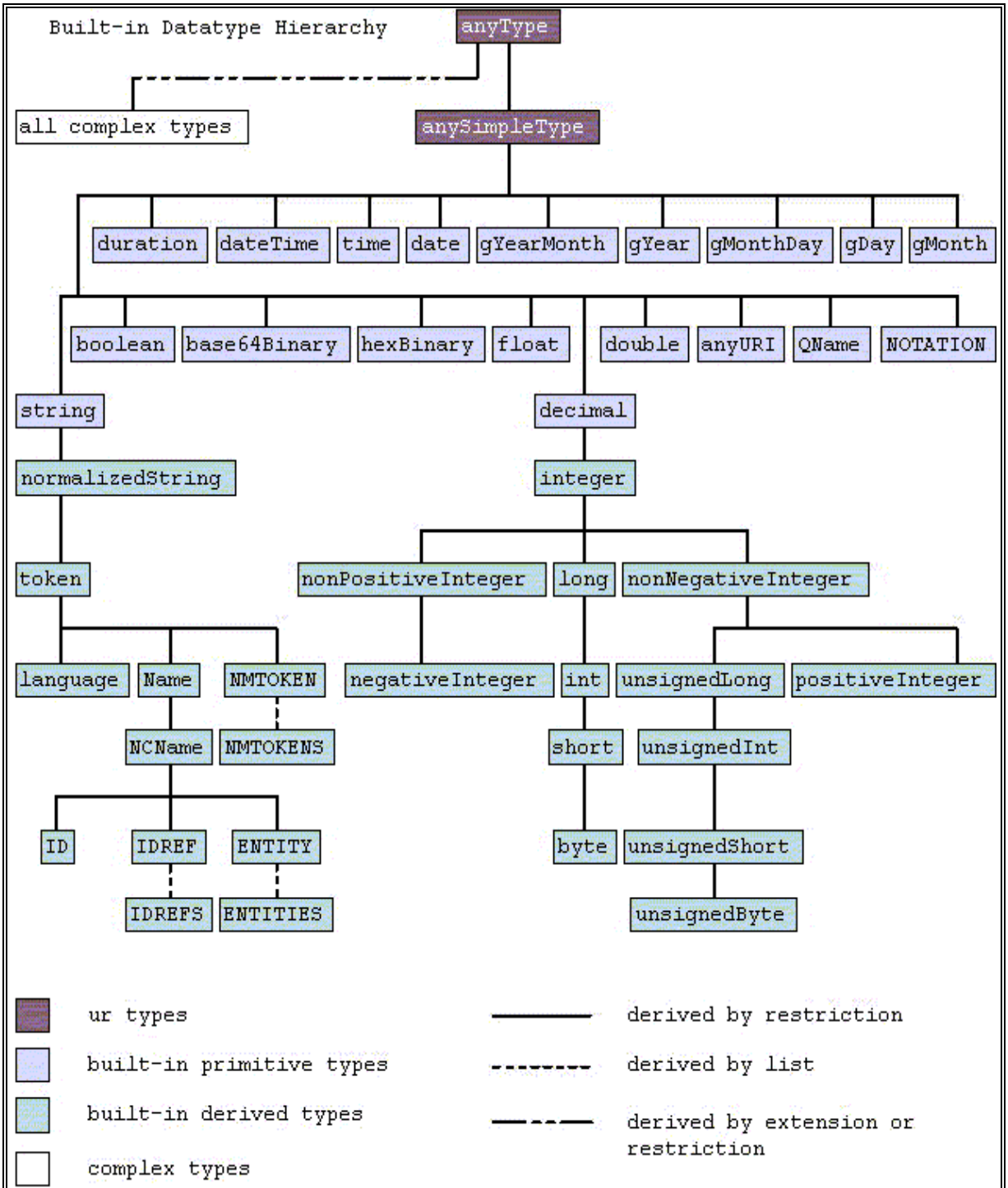
Deux types sont disponibles en *XML Schema*: les types simples et les types complexes.

TYPES SIMPLES

Il existe des types simples prédéfinis auxquels sont associés des règles de dérivation pour obtenir un total de 44 types de base. La dérivation se fait par restriction et certain type sont des listes d'autres types simples séparés par des espaces (c.f. Dérivation des types). L'Illustration 32 récapitule l'ensemble des types intégrés.

A partir de ceux-ci, l'utilisateur peut créer lui même des types par dérivation dont voici les principales catégories décrites:

- L'*union*: permet de définir un nouveau type en incluant toutes les valeurs possibles d'un certain nombre d'autres types;
- La *liste*: permet de définir un nouveau type à partir d'un type de base, où la valeur du nouveau type correspond à plusieurs valeurs de type de base séparées par des espaces;
- La *restriction*: limite les valeurs possibles d'un type donné en définissant des propriétés appelés facettes (longueur d'une chaîne de caractère, valeur maximale, valeur minimale,...);
- L'*extension*: consiste à définir un nouveau type à partir d'un type de base en lui ajoutant des sous éléments et/ou des attributs. Le résultat est donc un type de complexe;



ur désigne un supertype abstrait pour utiliser le polymorphisme. Ainsi, **anySimpleType** permet d'utiliser tout les types simples et **anyType** tout les types simples et complexes.

Illustration 32: Classification des types en XML Schema

TYPES COMPLEXES

Le type complexe peut se rapprocher du type *RECORD* du langage *ADA*, voire des classes des langages à objets. Il s'agit d'un type comprenant d'autres propriétés telles que des attributs ou des éléments (appelés sous éléments dans ce cas). La définition d'attributs a déjà été vue dans la partie précédente. Il reste donc à étudier la définition de modèle de contenu de chacun des éléments. Le modèle de contenu utilise des connecteurs pour indiquer un ordre d'apparition des éléments.

- Le connecteur **SEQUENCE** définit une suite d'éléments. Ces sous éléments doivent apparaître dans le même ordre que la déclaration;

```
<SCHEMA>

<ELEMENTNAME='uneSequence'
   TYPE='t_sequence'/>
<COMPLEXTYPE NAME='t_contact'/>
<SEQUENCE>
   <ELEMENT NAME='a' TYPE='INTEGER'/>
   <ELEMENT NAME='b' TYPE='STRING'/>
   <ELEMENT NAME='c' TYPE='STRING'/>
</SEQUENCE>
</COMPLEXTYPE>

</SCHEMA>
```

```
<exemple>

<uneSequence>
   <a> . . . </a>
   <b> . . . </b>
   <c> . . . </c>
</uneSequence>

</exemple>
```

Illustration 33: Un exemple de modèle de contenu de type Sequence

- Le connecteur **CHOICE** définit un choix d'apparitions. Si trois éléments sont déclarés sous le connecteur, seul l'un des trois doit apparaître;

```
<SCHEMA>

<ELEMENT NAME='unChoix'
TYPE='t_choix'/>
<COMPLEXTYPE NAME='t_choix'/>
<CHOICE>
   <ELEMENT NAME='a' TYPE='INTEGER'/>
   <ELEMENT NAME='b' TYPE='STRING'/>
   <ELEMENT NAME='c' TYPE='STRING'/>
</CHOICE>
</COMPLEXTYPE>

</SCHEMA>
```

```
<exemple>

<unChoix>
   <a> . . . </a>
</unChoix>

<unChoix>
   <b>...</b>
</unChoix>

<unChoix>
   <c>...</c>
</unChoix>

</exemple>
```

Illustration 34 Un exemple de modèle de contenu de type Choice

- Le connecteur **ALL** impose aux sous éléments d'apparaître une fois au maximum, mais n'impose aucun ordre.

```
<SCHEMA>

<ELEMENT NAME='unAll' TYPE='t_all'/>
<COMPLEXTYPE NAME='t_all'/>
<ALL>
   <ELEMENT NAME='a' TYPE='INTEGER'/>
   <ELEMENT NAME='b' TYPE='STRING'/>
   <ELEMENT NAME='c' TYPE='STRING'/>
</ALL>
</COMPLEXTYPE>

</SCHEMA>
```

```
<exemple>

<unAll>
   <a> . . . </a>
   <b>...</b>
   <c>...</c>
</unAll>

<unAll>
   <b>...</b>
   <c>...</c>
   <a>...</a>
</unAll>
```

Illustration 35: Un exemple de modèle de contenu de type ALL

Les sous éléments d'un type complexe peuvent être enrichis d'occurrences pour définir le nombre de fois où ils doivent apparaître dans le document instance. Les attributs **minOccurs** et **maxOccurs** viennent compléter ainsi la définition.

2.3°) DERIVATION DES TYPES

L'héritage ne concerne que les types et non les éléments. La dérivation est soit réalisée par restriction, soit par extension.

Dans le cas d'un type simple, la dérivation par restriction permet de créer un type utilisateur dont le domaine de valeurs est un sous-ensemble du domaine du type de base. Ces restrictions sont

appelées facettes. La restriction peut aller jusqu'à définir des valeurs. Ainsi un type peut être créé à partir d'une chaîne de caractères, puis le domaine est restreint à une énumération des chaînes *OUI* et *NON*. La dérivation par extension n'intervient que si l'on souhaite créer un type complexe à partir d'un type simple. En effet, un type simple ne possédant pas d'attributs, la création d'un type complexe basé sur le type simple permet de qualifier la balise avec des attributs.

La dérivation sur les types complexes se rapproche des concepts des langages de programmation orientés objet. En dérivant par restriction un type complexe, on peut redéfinir un ou plusieurs sous élément et donc redéfinir un type. La dérivation par extension permet de rajouter des sous éléments à un type et donc de lui rajouter des attributs.

La notion de type abstrait est elle aussi prise en charge. Dans ce cas, l'élément utilisant ce type ne peut être instancié que si on utilise ses sous-types. Ces notions seront développées dans le chapitre Express vers XML Schema (page 43)

La dérivation est contrôlée par l'ajout d'attributs à la définition d'un type complexe. Deux sont possibles: **FINAL** et **BLOCK**. Avec l'attribut **FINAL** on peut interdire la dérivation dans un schéma par extension, par restriction ou pour les deux. **BLOCK** autorise la création de type par dérivation mais peut empêcher leur utilisation par des éléments. Ces attributs prennent la valeur de **extension**, **restriction** ou **all** suivant que l'on souhaite appliquer la *restriction* sur les types dérivés par *extension*, par *restriction* ou *pour les deux*.

2.4°) CONTRAINTES DISPONIBLES

Deux types de contraintes peuvent être exprimées sont possibles en *XML Schema*:

- La contrainte *d'unicité* permet de déclarer une clé unique sur un attribut et donc d'imposer l'unicité au sein des instances d'un document XML;
- La contrainte *référentielle* autorise la création de clés référentielles et permet de spécifier que l'attribut d'un élément pointe non plus vers n'importe quelle instance issue d'un élément dans le document mais vers une instance spécifique de l'élément.

3°) Les logiciels à disposition

Très peu de logiciels permettent de modéliser les schémas XML de type *XML Schema*. Seuls deux ont été testés: *XML Spy* et *XML Authority*. Ils ne prennent pas en compte toutes les fonctions offertes par les schémas et ne sont disponible qu'à titre d'essai en attendant une complète implémentation.

- *XML Authority* permet de modéliser des types mais les instances de types complexes dérivés ne respectent pas leurs modèles de contenu;
- *XML Spy* modélise tous les types simples et complexes ainsi que les types dérivés. Mais les contraintes ne sont pas encore implémentées.

On peut donc conclure sur une insuffisance des logiciels de modélisation des schémas XML.

IV.Conclusion

Issu du domaine documentaire, l'approche par langage de structuration de l'information permet de véhiculer de l'information syntaxiquement compréhensible par l'homme et interprétable par des machines. Actuellement, plusieurs logiciels (tels que les SGBD) tendent à utiliser *XML* pour l'échange d'informations. Ils est donc nécessaire d'assurer l'intégrité de l'information (exemple: le

typage des données) et donc d'utiliser des schémas pour créer des modèles des fichiers XML d'échange. Nous avons donc un formalisme de modélisation avec le langage *EXPRESS* qui nous permet de représenter l'univers d'un discours en approchant le raisonnement humain et un langage de structuration documentaire, *XML Schema*, qui permet d'échanger de l'information compréhensible par l'homme. La création de modèles de passage entre ces deux langages ouvre de nouveaux horizons.

En réalisant le *mapping* d'*EXPRESS* vers *XML Schema*, les modèles réalisés dans un langage de modélisation de données à fort pouvoir d'expression peuvent s'échanger via un fichier lisible et compréhensible.

Le *mapping* inverse permet de modéliser les schémas XML et leurs instances et utiliser le pouvoir d'expression du langage *EXPRESS* pour valider des documents XML et donc vérifier si un document est valide par rapport à son schéma. De plus une telle approche pourrait aller jusqu'à enrichir le pouvoir d'expression de *XML Schema*.

Ces points seront développés dans les chapitres suivants, respectivement Express vers XML Schema et XML Schema vers EXPRESS.

Express vers XML Schema

I.Introduction

Le langage *EXPRESS* est utilisé pour la modélisation de données. Le pouvoir d'expression offert par ce langage permet au de créer des modèles riche sémantiquement. Cependant l'information véhiculée par ses fichiers d'instances, les fichiers physiques, est incompréhensible pour des lecteurs. L'approche des langages de structuration de documents a permis par contre de créer un formalisme de modélisation de données avec *XML Schema* et un fichier d'échanges d'instances, le document XML, parfaitement lisible aussi bien pour les machines que pour les humains. L'idéal serait d'allier le fort pouvoir sémantique de l'un avec les possibilités de structuration syntaxique de l'autre.

Prenons, l'exemple de *PLIB*. Cette norme est développée depuis une dizaine d'années au sein de l'organisme *ISO* à partir d'un modèle conceptuel proposé par le *LISI-ENSMA* en 1988. Elle permet de représenter formellement des bibliothèques de composants qui sont décrits par un méta modèle objet écrit en *EXPRESS*. Ce modèle fournit une structure neutre, indépendante de tout système informatique pour la saisie, l'échange, la gestion et l'exploitation de l'ensemble des informations liées à une bibliothèque. Mais les fichiers d'échanges de données, les composants, respectent la syntaxe des fichiers physique *EXPRESS*. Il serait utile de transformer les modèles *EXPRESS* en modèles *XML Schema* pour passer de la bibliothèque de composants au catalogue dont l'information est structurée et compréhensible pour l'homme.

La suite du chapitre sera décomposée de la manière suivante. La première partie exposera les méthodes d'*XML Schema* pour exprimer des concepts simples des langages à objets et la deuxième partie sera consacré à de concepts plus avancés liés à des contraintes sur les instances.

II.Représentation des concepts objets d'EXPRESS en XML Schema

1°) La catégorisation: représentation des entités

L'entité en langage *EXPRESS* désigne une classe instanciable. L'équivalent en *XML Schema* se nomme l'*élément*. Cependant, l'entité porte en plus une notion d'identificateur, attribut parfaitement visible lors de l'instanciation. Une entité sera donc représentée par un *élément* possédant comme attribut un identificateur de type *ID*, un type de base d'*XML Schema* assurant l'unicité lors de l'instanciation (Illustration 36, document XML correspondant en Illustration 37).

ENTITY E; ... **END_ENTITY;**	**<ELEMENTNAME='**E**'** **TYPE='**t_E**'/>** **<COMPLEXTYPE NAME='**t_E**'>** **...** **<ATTRIBUTENAME='**id**' TYPE='ID' USE='required'/>** **</COMPLEXTYPE>**

Illustration 36: Représentation d'une entité EXPRESS en XML Schema

`#1=VIS(...);`
`#2=VIS(...);`
	`<vis id='_1'>`
	`...`
	`</vis>`
	`<vis id='_2'>`
	`...`
	`</vis>`

Illustration 37: Fichier d'instances

2°) Les associations

Une association entre entités est définie en *EXPRESS* au travers d'un attribut. Ainsi on décrit le nom de l'association et l'entité à associer (Illustration 38).

`ENTITY vis;`
`...`
`END_ENTITY;`
`ENTITY ecrou;`
`...`
`END_ENTITY;`
`ENTITY assemblage;`
`sa_vis : vis;`
`son_ecrou : ecrou;`
`END_ENTITY;`
	`#1=VIS(...);`
	`#2=ECROU(...);`
	`#3=ASSEMBLAGE(#1,#2);`

Illustration 38: Un assemblage vis-écrou en EXPRESS

En *XML Schema*, deux approches peuvent être envisagés pour définir des associations.

- La première consiste à créer des types complexes tout en considérant les entités associées comme des *sous éléments* (Illustration 39). On remarque une meilleure lisibilité de l'instance de document résultante: à la première lecture les entités en relation sont clairement identifiables. Cependant, au niveau instance, la *vis* et l'*ecrou* n'ont pas d'existence propre mais dépendent de l'assemblage. Il est donc impossible de les référencées dans d'autres assemblages. Ainsi, si l'assemblage est supprimé, ces deux entités le sont aussi.

`<COMPLEXTYPE NAME='t_vis'>`
`<ELEMENTNAME='son_nom'`
`TYPE='STRING'/>`
`...`
`</COMPLEXTYPE>`
`<COMPLEXTYPE NAME='t_ecrou'>`
`<ELEMENTNAME='son_diametre'`
`TYPE='INTEGER'/>`
`...`
`</COMPLEXTYPE>`
`<ELEMENTNAME='assemblage'`
`TYPE='t_assemblage'/>`
`<COMPLEXTYPE NAME='t_assemblage'>`
`<ELEMENTNAME='sa_vis' TYPE='t_vis'/>`
`<ELEMENTNAME='son_ecrou' TYPE='t_ecrou'/>`
`</COMPLEXTYPE>`
	`<assemblage id='_1'>`
	`<sa_vis id='_2'>`
	`<son_nom>vis CHC</son_nom>`
	`</sa_vis>`
	`<son_ecrou id='_3'>`
	`<son_diametre>10</son_diametre>`
	`</son_ecrou>`
	`</assemblage>`

Illustration 39: Un assemblage vis-écrou en XML Schema

- La deuxième solution consiste à utiliser des contraintes dites référentielles. Avec cette méthode, les *vis* et les *ecrous* peuvent être déclarés indépendamment de l'assemblage et

référencés à posteriori. Ce point sera développé quand les contraintes référentielles seront abordées (Chapitre III Concept avancés).

3°) L'héritage

Au niveau *EXPRESS*, l'héritage s'applique aux entités. En *XML Schema*, l'héritage s'applique aux types, simples et complexes. Or jusqu'à maintenant, une **ENTITY** *EXPRESS* est représentée en **ELEMENT** *XML Schema*. Il faut donc considérer maintenant qu'une entité **EXPRESS** n'est plus seulement un *élément XML Schema* mais un *élément* associé à son *type complexe*, pour lequel une relation unique d'héritage peut être définie. Donc, la représentation de l'héritage multiple est impossible.

DERIVATION DES ENTITES

L'Illustration 40 montre la création d'une entité *vis_a_tete* à partir de l'entité *vis* traitée précédemment. Un attribut supplémentaire, *sa_hauteur_de_tete*, est inséré dans le cadre de cette dérivation.

```
ENTITY vis
SUPERTYPE OF(vis_a_tete);
  son_nom : STRING;
  son_diametre : INTEGER;
END_ENTITY;

ENTITY vis_a_tete
SUBTYPE OF(vis);
  sa_hauteur_de_tete : INTEGER;
END_ENTITY;
```

```
#1=VIS('H',10);

#2=VIS_A_TETE('CHC',12,2);
```

Illustration 40: Une spécialisation de vis EXPRESS

En *XML Schema*, il faut dériver par extension le type complexe *t_vis* précédemment créer pour obtenir le même résultat. Un sous élément *sa_hauteur_de_tete* est inséré dans la séquence. Il apparaîtra en dernier dans la liste des sous-ensembles dans le document XML (Illustration 41).

```
<ELEMENTNAME='vis'
   TYPE='t_vis'/>

<COMPLEXTYPE NAME='t_vis'/>
<SEQUENCE>
   <ELEMENTNAME='son_nom' TYPE='STRING'/>
   <ELEMENTNAME='son_diametre' TYPE='INTEGER'/>
</SEQUENCE>
</COMPLEXTYPE>

<ELEMENTNAME='vis_a_tete'
   TYPE='t_vis_a_tete'/>

<COMPLEXTYPE NAME='t_vis_a_tete'/>
<COMPLEXCONTENT>
   <EXTENSION BASE='t_vis'>
   <SEQUENCE>
   <ELEMENTNAME='sa_hauteur_de_tete' TYPE='INTEGER'/>
   </SEQUENCE>
   </EXTENSION>
</COMPLEXCONTENT>
</COMPLEXTYPE>
```

```
<vis id='_1'>
   <son_nom>
   H
   </son_nom>
   <son_diametre>
   10
   </son_diametre>
</vis>

<vis_a_tete id='_2'>
   <son_nom>
   CHC
   </son_nom>
   <son_diametre>
   12
   </son_diametre>
   <sa_hauteur_de_tete>
   2
   </sa_hauteur_de_tete>
</vis_a_tete>
```

Illustration 41: Une spécialisation de vis XML Schema

REDEFINITION DE TYPE

Admettons que l'on souhaite créer une nouvelle entité vis, baptisé *vis_speciale* qui possède les mêmes propriétés que l'entité *vis* mais dont le domaine de valeur du *diametre* est restreint. En *EXPRESS*, l'Illustration 42 montre la notion de restriction de valeurs pour un attribut.

```
TYPE t_diametre :=INTEGER;
  WHERE
  WR1 : SELF>0;
END_TYPE

TYPE t_diametre_limite :=t_diametre;
  WHERE
  WR1 : SELF<10;
END_TYPE

ENTITY vis
  SUPERTYPE OF (vis_speciale);
  son_nom : STRING;
  son_diametre : t_diametre;
END_ENTITY;

ENTITY vis_speciale
  SUBTYPE OF (vis_speciale);
  son_diametre : t_diametre_limite;
END_ENTITY;
```

```
#1=VIS('CHC',15);

#2=VIS('CHC',10);

#3=VIS_SPECIALE('CHC',9);
```

Illustration 42: Une redéfinition de type en EXPRESS

La représentation *XML Schema* au moyen d'une dérivation par restriction permet de définir cette redéfinition de type. Dans l'Illustration 43, le type simple *t_diametre_limite* est créé en dérivant par restriction le type *t_diametre* et en appliquant une *facette* (**MAXEXCLUSIVE VALUE=**). Il permet de définir le domaine de valeur de l'attribut restreint. Ensuite, le type complexe *t_vis_speciale* est dérivé par restriction du type *t_vis* et seules les propriétés à redéfinir sont présentes (le *diametre* dans l'exemple).

Il faut remarquer qu'un *élément* ou un type (*simple* ou *complexe*), peut être déclaré *abstrait*. On peut s'interroger maintenant sur le polymorphisme.

En reprenant l'Illustration 38 de l'assemblage en *EXPRESS*, on remarque qu'un assemblage uni un *ecrou* et une *vis* mais aussi toutes les entités filles de l'entité *vis*. Donc un assemblage peut relier un *ecrou* et une *vis_a_tete* ou un *ecrou* et une *vis_speciale*. En *XML Schema*, le polymorphisme est lui aussi représentable mais nécessite une syntaxe particulière pour indiquer quel élément on veut choisir au moment de l'instanciation. Un attribut **type** (associé à un espace de nom **xsi**) vient qualifier la balise. Il ne reste alors plus qu'à spécifier le modèle de contenu que l'on souhaite utiliser.

```
<SIMPLETYPENAME='t_diametre'
   <RESTRICTION BASE='INTEGER'>
   <MINEXCLUSIVE VALUE='0'/>
   </RESTRICTION>
<SIMPLETYPE>

<SIMPLETYPENAME='t_diametre_limite'>
   <RESTRICTION BASE='t_diametre'>
   <MAXEXCLUSIVE VALUE='10'/>
   </RESTRICTION>
<SIMPLETYPE>

<ELEMENTNAME='vis' TYPE='t_vis'/>

<COMPLEXTYPE NAME='t_vis'/>
<SEQUENCE>
   <ELEMENTNAME='son_nom' TYPE='STRING'/>
   <ELEMENTNAME='son_diametre' TYPE='t_diametre'/>
</SEQUENCE>
</COMPLEXTYPE>

<ELEMENTNAME='vis_speciale'
   TYPE='t_vis_speciale'/>

<COMPLEXTYPE NAME='t_vis_speciale'/>
<COMPLEXCONTENT>
   <RESTRICTION BASE='t_vis'>
   <SEQUENCE>
   <ELEMENTNAME='son_diametre' TYPE='t_diametre_limite'/>
   </SEQUENCE>
   </RESTRICTION>
</COMPLEXCONTENT>
</COMPLEXTYPE>
```

```
<vis id='_1'>
   <son_nom>
   CHC
   </son_nom>
   <son_diametre>
   15
   </son_diametre>
</vis>

<vis id='_2'>
   <son_nom>
   CHC
   </son_nom>
   <son_diametre>
   10
   </son_diametre>
</vis>

<vis_speciale id='_3'>
   <son_nom>
   CHC
   </son_nom>
   <son_diametre>
   9
   </son_diametre>
</vis_speciale>
```

Illustration 43: Une redéfinition de type en XML Schema

4°) La définition d'attributs

XML Schema possède un très grand nombre de types de base et propose la possibilité de créer des types utilisateurs. La majorité des types simples *EXPRESS* sont aussi exprimables sans problème. Cependant, l'expression des agrégats se heurte à quelques difficultés. Seule la **LIST** d'*EXPRESS* est exprimable directement en *XML Schema*. En effet, l'ordre est l'un des atouts du langage *XML*. Il est donc impossible de reproduire des **BAG**. Deux possibilités sont offertes pour créer la notion de liste: utiliser une liste *XML Schema* ou utiliser les indicateurs d'occurrences. Pour les exposer, l'entité *ecrou* va être complété d'une propriété dont le type est un agrégat.

```
ENTITY ecrou;
  son_diametre : INTEGER;
  ses_matériaux : LIST [1:?] OF STRING;
END_ENTITY;
```

```
#1=ECROU(8,(inox,aluminium),(#2,#3,#4,#5));
```

Illustration 44: Définition d'un écrou en EXPRESS

XML Schema permet de déclarer un type comme une liste de types simples (atomiques ou utilisateurs). Dans le document XML résultant, les instances des types simples vont apparaître séparées par des espaces. Dans l'Illustration 45, un type liste sera créé pour représenter **ses_matériaux**.

```
<ELEMENTNAME='ecrou'
   TYPE='t_ecrou'/>;

<COMPLEXTYPE NAME='t_ecrou'>
<SEQUENCE>
   <ELEMENTNAME='son_diametre' TYPE='INTEGER'/>
   <ELEMENTNAME='ses_materiaux_possibles' TYPE='t_listeMateriau'/>
   <ELEMENTNAME='ses_vis_possibles' TYPE='t_listeIDREF'/>
</SEQUENCE>
</COMPLEXTYPE>

<SIMPLETYPE NAME='t_listeMateriau'>
   <LISTITEM TYPE='STRING'/>
</SIMPLETYPE>

<SIMPLETYPE NAME='t_listeIDREF'>
   <LISTITEM TYPE='IDREF'/>
</SIMPLETYPE>
```

```
<ecrou id='_1'>
   <son_diametre>
   8
   </son_diametre>
   <ses_materiaux_possibles>
   inox aluminium
   </ses_materiaux_possibles>
   <ses_vis_possibles>
   '_2' '_3' '_4' '_5'
   </ses_vis_possibles>
</ecrou>
```

Illustration 45: Expression des agrégats avec des listes XML Schema

La deuxième solution proposée pour représenter un agrégat est la suivante. Chaque objet de la liste est considéré comme un élément. Avec des occurrences, on vient fixer les bornes de l'intervalle comme le montre l'Illustration 46.

```
<ELEMENTNAME='ecrou' TYPE='t_ecrou'/>;

<COMPLEXTYPE NAME='t_ecrou'>
   <SEQUENCE>
   <ELEMENTNAME='son_diametre'
   TYPE='INTEGER'/>
   <ELEMENTNAME='ses_materiaux_possibles' TYPE='STRING'
   MINOCCURS='0'
   MAXOCCURS='UNBOUNDED'/>
   </SEQUENCE>
</COMPLEXTYPE>
```

```
<ecrou id='_1'>
   <son_diametre>8</son_diametre>
   <ses_materiaux_possibles>
   inox
   </ses_materiaux_possibles>
   <ses_materiaux_possibles>
   aluminium
   </ses_materiaux_possibles>
</ecrou>
```

Illustration 46: Expression des agrégats avec des occurrences modèle Schema

La première solution permet de réduire le fichier en taille et répertorie toutes les propriétés dans les mêmes balises. Cependant, dans l'exemple traité, il est impossible de différencier la chaîne de caractères et une liste de chaîne de caractères. Par la suite des restrictions (en dérivant par restriction car une liste est un type simple) peuvent s'appliquer pour restreindre la taille de la liste. Il est par contre impossible de pointer sur une valeur de la liste.

Avec la deuxième solution, le document XML voit sa taille augmenter. Avec les attributs **minOccurs** et **maxOccurs**, on contrôle exactement le nombre d'occurrences. Il est aussi possible de qualifier chaque balise *ses_materiaux_possibles* par un attribut **ID** ou autre et la recherche de valeurs est elle aussi facilitée (via un **IDREF** ou une contrainte de type référentielle).

La conclusion peut donc être la suivante: si les valeurs de la liste n'ont pas besoin d'être référencées, une **LISTITEM** *XML Schema* suffit. Si ce n'est pas le cas, le choix avec les occurrences s'impose.

III.Concept avancés

1°) Contrainte d'unicité

Jusqu'à maintenant, les *DTD XML* possédaient déjà un mécanisme de représentation de l'unicité avec un attribut de type **ID**. L'analyseur vérifiait que deux attributs de type **ID** ne possédaient pas la même valeur. Dans ce cas il retournait une erreur.

XML Schema introduit de nouvelles notions. L'unicité ne concerne plus seulement un **ID** mais est étendu aux autres types simples (**string**, **integer**, ...) aux sous éléments d'un élément. Prenons par exemple la déclaration d'une entité *personne* dont *son_nom* serait unique au sein du domaine étudié (Illustration 47).

```
ENTITY personne;
  son_nom : STRING;
  son_prenom : STRING;
  son_age : INTEGER;
UNIQUE
  ur1 : son_nom;
END_ENTITY;
```

Illustration 47: Unicité en EXPRESS

Avec *XPath* [Clark et al. 99], un langage d'expression de chemin à l'intérieur d'un document représenté sous forme d'arbre, il est possible de voyager dans le document et de contrôler un attribut en particulier. Il est ainsi possible de contrôler l'unicité d'un attribut mais aussi des valeurs d'un sous-élément. Par contre, l'unicité n'est applicable que sur des sous éléments de types simples.

```
<ELEMENT NAME='personne' TYPE='t_personne'/>

<COMPLEXTYPE NAME='t_personne'>
   <SEQUENCE>
   <ELEMENT NAME='nom' TYPE='STRING'/>
   <ELEMENT NAME='prenom' TYPE='STRING'/>
   <ELEMENT NAME='age' TYPE='INTEGER'/>
   </SEQUENCE>
</COMPLEXTYPE>

<UNIQUE NAME='ur1'>
   <SELECTOR XPATH='/personne'/>
   <FIELD XPATH='nom'/>
</UNIQUE>
```

Illustration 48: Unicité en XML Schema

2°) Restrictions de domaines

Les restrictions de domaines ont déjà été développées dans les parties précédentes. Il est possible de définir des intervalles de valeurs pour des types contenant des nombres mais aussi de contrôler les chaînes de caractères (par exemple, vérifier qu'une adresse mail contient bien le signe @). L'énumération permet même de créer des listes de choix (ex: un sous élément *sexe* donnant le choix entre *homme* ou *femme*).

Toutes les règles locales (**WHERE**) concernant des limitations de domaines sont exprimables.

3°) Connaissance procédurale: les limites XML Schema

Pour démontrer les limites d'*XML Schema*, il faut étendre les exemples à la connaissance procédurale. Prenons l'exemple d'un catalogue de vis écrit en *EXPRESS*. Le principe est le suivant, une entité va décrire une famille de *vis* avec sa désignation (*sa_designation*) et son domaine de valeurs modélisé par deux attributs: *valeur_mini* et *valeur_maxi*. La contrainte stipule que *valeur_mini* doit être inférieur à la *valeur_maxi*.

```
ENTITY famille_de_vis;
  designation : STRING;
  valeur_mini : INTEGER;
  valeur_maxi : INTEGER;
WHERE
  wr1 : valeur_mini<valeur_maxi;
END_ENTITY;
```

Illustration 49: Contraintes entre attributs en EXPRESS

En *XML Schema* cette contrainte n'est pas exprimable mais des solutions de substitution permettent de contourner ce problème.

3.1°) UTILISATION D'AUTRES SCHEMAS

La première solution est d'utiliser un ou plusieurs schémas pour exprimer des contraintes additionnelles. Dans le chapitre Les langages de structuration de l'Information, la conclusion de la partie Deuxième étude: pouvoirs d'expression des schémas XML, indique que les langages peuvent être séparés en deux catégories: ceux qui modélisent explicitement la structure des modèles et ceux qui utilisent des règles de validation.

Schematron [Jeliffe 00], un langage de modélisation de type règle, possède le pouvoir de validation qui manque dans le cas de notre exemple. Des conditions vont être insérées dans le modèle écrit en *XML Schema* pour faire référence au formalisme *Schematron*. L'utilisation d'espace de nom est importante dans ce cas pour éviter les confusions entre les noms de balises. **XSD** est l'espace de nom par défaut de *XML Schema* et **SCH** celui de *Schematron*. La structure des données du document XML résultante sera validée au travers du schéma associé et les conditions XML seront contrôlées par les règles, écrites en *Schematron* et insérées dans le schéma. Le schéma correspondant est exposé en Illustration 50.

En conclusion, différents schémas peuvent être utilisés parallèlement pour accroître le pouvoir d'expression d'un modèle. Certains schémas ont été crées pour étendre les schémas en dehors des limitation d'*XML Schema*, mais chaque schéma possèdent ses propres limitations. Il peut donc être possible d'utiliser plus de deux schémas pour valider un document et donc de rendre illisible le schéma malgré la présence d'espace de noms différents. De plus, une telle approche nécessite d'appréhender il faut apprendre chaque vocabulaire, concept, sémantique propre à chaque schéma. Notons enfin que beaucoup de schémas ne sont pas encore à leur version finale et donc pas encore implémentés dans les outils de validation. L'exemple précédent n'a pu être testé car *Schematron* n'est pas pris en charge par les outils de validation de schémas *XML Schema*.

```
<XSD:SCHEMA>

<XSD:ANNOTATION>
   <XSD:APPINFO>
   <SCH:TITLE>Validation du schéma<SCH:TITLE>
   <SCH:NS PREFIX='d' uri='http://www.demo.org'/>
   </XSD:APPINFO>
</XSD:ANNOTATION>

<XSD:ELEMENT NAME='famille_de_vis' TYPE='t_famille_de_vis'>
   <XSD:ANNOTATION>
   <XSD:APPINFO>
   <SCH:PATTERN NAME='WR1'>
   <SCH:RULE CONTEXT='d:famille_de_vis'>
   <SCH:ASSERTTEST='d:valeur_mini < d:valeur_maxi'
   DIAGNOSTICS='plusGrandQue'>
   Valeur maxi plus grande que valeur mini
   </SCH:ASSERT>
   </SCH:RULE>
   </SCH:PATTERN>
   <SCH:DIAGNOSTICS>
   <SCH:DIAGNOSTIC id='plusGrandQue'>
   Erreur! A est plus grand que B
   A=<SCH:VALUE-OF SELECT='d:valeur_mini'/>
   B=<SCH:VALUE-OF SELECT='d:valeur_maxi'/>
   </SCH:DIAGNOSTIC>
   </SCH:DIAGNOSTICS>
   </XSD:APPINFO>
   </XSD:ANNOTATION>
<XSD:ELEMENT>

<XSD:COMPLEXTYPE NAME='t_famille_de_vis'>
   <XSD:SEQUENCE>
   <XSD:ELEMENT NAME='designation' TYPE='XSD:STRING'/>
   <XSD:ELEMENT NAME='valeur_mini' TYPE='XSD:INTEGER'/>
   <XSD:ELEMENT NAME='valeur_maxi' TYPE='XSD:INTEGER'/>
   </XSD:SEQUENCE>
</XSD:COMPLEXTYPE>

</XSD:SCHEMA>
```

Illustration 50: Contraintes entre attributs en XML Schema-Schematron

3.2°) LES FEUILLES DE STYLE

La seconde solution consiste à écrire une feuille de style, dans notre cas, en *XSLT* [Clark et al. 98] pour nous informer si la contrainte est bien respectée. La feuille de style va imposer la contrainte *valeur_mini* est inférieure à *valeur_maxi*. Pour information, **XSL** désigne l'espace de noms propre à *XSL* et **D** celui du document XML résultant. La feuille de style va afficher une information indiquant si les instances sont valides ou non.

```
<XSL:STYLESHEET ...>

   <XSL:OUTPUT METHOD='text'/>

   <XSL:TEMPLATE MATCH='/'>
   <XSL:IF TEST='/D:famille_de_vis/D:valeur_mini <
   /D:famille_de_vis/D:valeur_maxi' >
   <XSL:TEXT>Les instances sont valides</XSL:TEXT>
   </XSL:IF>
   <XSL:IF TEST='/D:famille_de_vis/D:valeur_mini >=
   /D:famille_de_vis/D:valeur_maxi'>
   <XSL:TEXT>L instance</XSL:TEXT>
   <XSL:VALUE-OF SELECT='/D:famille_de_vis/D:designation'/>
   <XSL:TEXT>n est pas valide</XSL:TEXT>
   </XSL:IF>
   </XSL:TEMPLATE MATCH='/'>
</XSL:STYLESHEET>
```

Illustration 51: Contraintes validés par une feuille de style XSL

Les limites du pouvoir d'expression ne sont plus celles des schémas mais celles du langage de transformation *XSL* La feuille de style est propre à la modélisation et les contraintes sont séparées du schéma. Cependant, ce langage est supporté par la plupart des outils et est utilisé majoritairement par les spécialistes d'XML pour afficher leurs données. La solution *XSL* apparaît donc comme la meilleure à ce jour. N'ayant pas été plus en aval dans l'étude de ce langage, il nous est impossible de définir précisément les limites quant à son pouvoir d'expression.

IV.Conclusion

Les concepts *EXPRESS* ont pu être représentés en *XML Schema*, malgré les quelques difficultés rencontrées.

Pour exprimer les différents niveaux de la connaissance, *XML Schema* apporte beaucoup de solutions. Les différentes notions objets offertes permettent de modéliser les entités avec leurs attributs et les relations qui les unissent tels que l'association et l'héritage. Cependant, pour ce dernier cas, l'héritage ne peut être multiple: un élément ne peut hériter que d'un seul élément. De plus le domaine des valeurs des types simples peut être restreint. Nous retrouvons donc les possibilités qu'offre le langage *EXPRESS* pour la définition de types.

Les problèmes rencontrés se situent principalement dans l'expression des contraintes. Des solutions ont été proposées pour représenter des contraintes locales comme celle d'utiliser des schémas complémentaires (solution mise en attente car soit la spécification définitive des schémas n'est pas encore publiée, soit très peu d'outils la prennent en charge) ou utiliser une feuille de style pour valider les instances. Dans ce cas, la limite d'expression est celle de *XSL*, et le travail ne fut poursuivi dans ce sens dans le cadre du DEA. Pour les contraintes globales, (effectuer des recherches sur les instances), aucune solution n'est proposée. *XML Schema* ne permet pas de travailler sur l'ensemble des instances, mis à part pour l'unicité et les contraintes référentielles. Les schémas concurrents n'ont apporté aucune réponse. Il ne sera pas possible non plus de modéliser les attributs **`INVERSE`**.

Le modèle *PLIB*, qui est écrit en *EXPRESS* n'est pas transformable directement en *XML Schema*. Cependant, il est possible de définir un modèle *PLIB* simplifié, n'utilisant pas des principes tels que l'héritage multiple ou des contraintes globales, qui lui sera modélisable en *XML Schema*, via les propositions faites dans ce chapitre. Bien que ce modèle aura perdu de sa richesse sémantique (dû à

la simplification au niveau du modèle *EXPRESS*), l'information véhiculée par les documents XML restera syntaxiquement structuré et compréhensible pour les lecteurs.

XML Schema vers EXPRESS

I.Introduction

Les études menées dans les chapitres précédents ont permis de présenter d'une part des langages de modélisation conceptuelle, EXPRESS, et d'autre part des langages de structuration documentaire, *XML* et *XML Schema*.

Cependant, les outils pour la mise en œuvre de documents XML et leurs schémas ne permettent pas d'exprimer toutes les possibilités offertes par les schémas *XML Schema*. L'objet de ce chapitre est de proposer un environnement pour la validation de documents XML conformes à des schémas *XML Schema* donnés. Pour cela nous proposons d'utiliser le langage *EXPRESS* et les outils développés autour de ce langage pour respectivement représenter en *EXPRESS* un schéma *XML Schema* et proposer un outil de validation travaillant directement sur la représentation en *EXPRESS* d'un schéma *XML Schema*.

Ce chapitre va donc exposer, dans les deux premières parties, les deux solutions envisagées pour exprimer les concepts *XML Schema* en *EXPRESS*. Enfin dans une troisième partie, une implémentation de cette étude sera proposée au travers d'un analyseur de documents XML basés sur des description *XML Schema* et développée dans un environnement *EXPRESS*.

II.Première approche: mapping directe

Cette approche consiste à exprimer chacun des concepts *XML Schema* en *EXPRESS*.

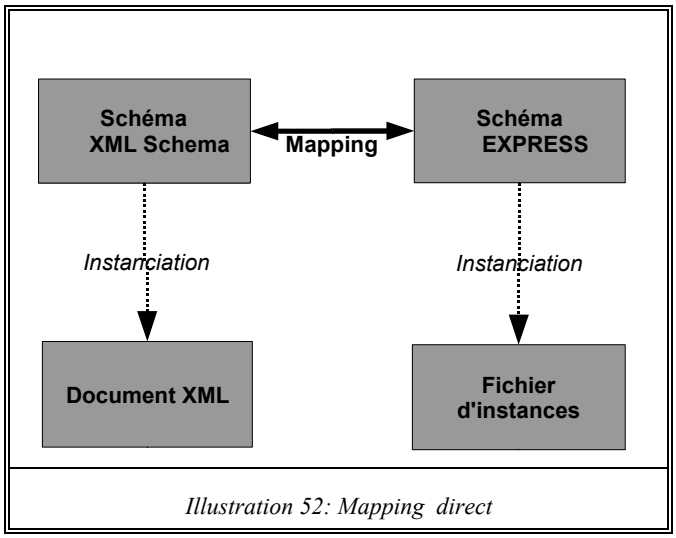


Illustration 52: Mapping direct

Ce problème de représentation avait déjà été posé pour *SGML* et ses *DTD* [Bicarregui et al. 95]. L'auteur souhaitait stocker et échanger des documents écrits en *SGML* et donc prouver qu'*EXPRESS* pouvait être utilisé dans un domaine autre que les données d'ingénierie. Plutôt que de modéliser directement chaque **ELEMENT** de *DTD* en **ENTITY** d'un schéma *EXPRESS*, et perdre certaine notion

comme, par exemple, l'ordre d'apparition des données, la démarche entreprise ressemble plus à une méta modélisation des *DTD* en *EXPRESS*, bien que ce terme ne soit nullement employé.

Plus récemment, des travaux ont eu pour objet d'évaluer la transformation de schémas *XML Schema* [Bourret 01a] et des *DTD* [Bourret 01b] vers des langages à objets (comme par exemple *C++*) ou encore vers des bases de données relationnelles.

Les résultats de cette étude ont établie que la modélisation des types *XML Schema* ne posaient aucun problème pour les langages de programmation, ainsi que les contraintes d'unicité ou référentielles et les restriction de domaines pour les types simples. Cependant, le modèle de contenu et la souplesse qu'il apporte tout en imposant une rigueur sur l'ordre parait plus difficile à représenter directement.

Pour le langage *EXPRESS* (qui est un langage de spécification orienté objets), les problèmes seront les mêmes. L'expression des types, de leurs domaines de valeurs et les différentes contraintes pourront être modélisés. Le chapitre précédent a montré que le pouvoir d'expression du langage de modélisation était supérieur dans ces domaines. Mais aucune solution ne peut être proposée pour l'expression de l'ordre d'un modèle de contenu. Il en est de même pour la création des types complexes *XML Schema*, impossible directement en *EXPRESS*, alors que certains langages (*ADA*, *C*,...) le permettent. En conclusion, une transformation directe et systématique des schémas *XML Schema* dans un modèle *EXPRESS* s'avère difficilement envisageable. Cependant une alternative existe et va consister à représenter non plus le schéma XML directement, mais au niveau méta.

III.Deuxième approche: mapping au niveau meta

1°) Présentation

Les schémas *EXPRESS* permettent de définir des modèles avec un très grand pouvoir d'expression. Ce modèle est défini selon un certain point de vue propre à l'utilisation courante. Comme ce modèle ne représente qu'un seul point de vue, il est considéré comme statique. Le langage *EXPRESS* va permettre de modéliser à un niveau d'abstraction supérieur en créant un modèle dont les instances représentent elles mêmes des modèles de données d'où le concept de méta modélisation. L'instanciation du méta modèle donne un modèle. Donc les données transférées via le fichier d'échange représentent un modèle de données. Le passage au niveau méta du modèle impose le passage similaire pour le fichier d'instances. Il faut donc définir un méta modèle des instances.

Le langage *EXPRESS* possède cette faculté à pouvoir se modéliser lui-même (on parle de réflexivité du langage). On peut remarquer que *XML Schema* possède la même faculté. Il existe d'ailleurs un schéma écrit en *XML Schema* des concepts véhiculés par *XML Schema*.

2°) Mise en oeuvre

Le but est de méta modéliser *XML Schema* en *EXPRESS* (c'est à dire la description des termes licites pour décrire un schéma XML, ainsi que leur enchaînement). Le travail consiste à modéliser les concepts de *XML Schema* en *EXPRESS*. Ainsi, en modélisation ce modèle *EXPRESS*, le fichier d'échanges contiendra une population de données *EXPRESS* correspondant à un schéma XML.

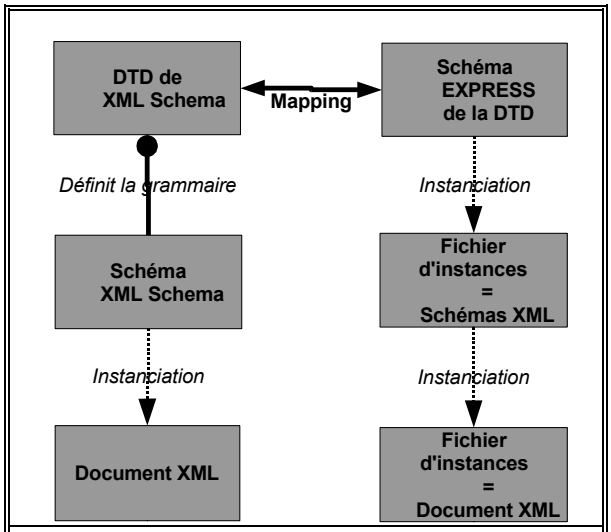


Illustration 53: Méta modélisation du langage XML Schema

Par ailleurs, il est aussi nécessaire de méta modéliser le document XML pour permettre d'échanger les instances, en plus du modèle et pouvoir par la suite valider des documents XML conformément à un schéma *XML Schema* donné. La structure d'un document XML possède une syntaxe bien définie. Celle-ci va être représentée par un modèle *EXPRESS*. Ainsi, une instanciation d'un tel schéma *EXPRESS* représentera un document XML.

Enfin, il faut aussi penser à la relation entre un document XML et son schéma afin de pouvoir en assurer la validité. Des fonctions de contrôle génériques entre les fichiers d'instances des deux modèles *EXPRESS*, celui des schémas et celui des documents, vont exprimer cette validation.

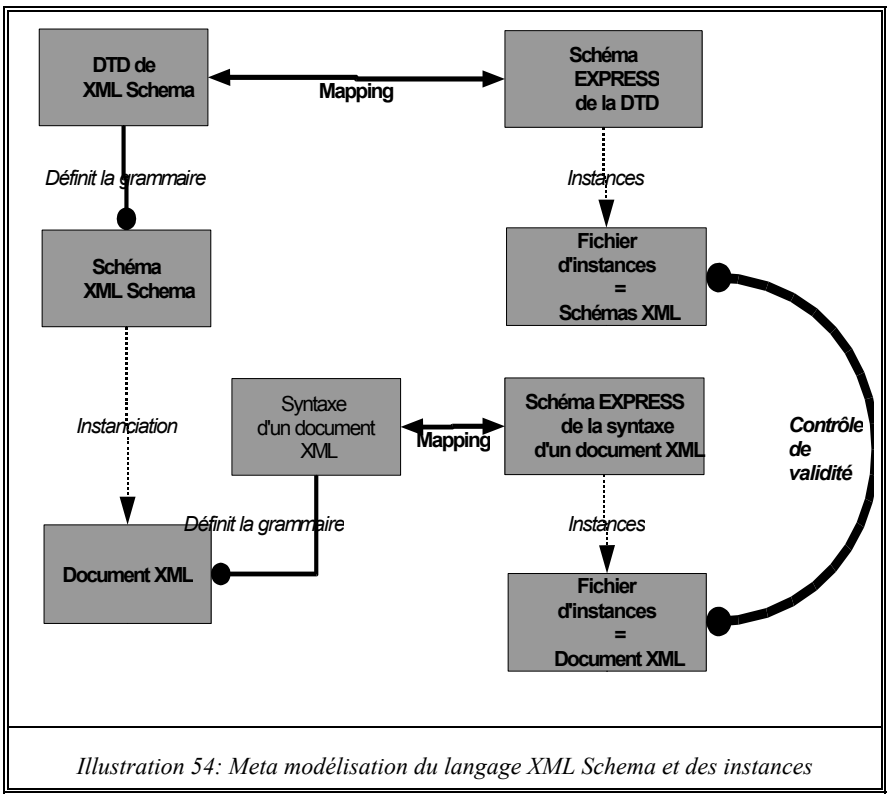


Illustration 54: Meta modélisation du langage XML Schema et des instances

3°) Meta-modélisation des concepts XML Schema

Dans cette section, nous présentons une première version de la représentation des concepts de *XML Schema* en *EXPRESS* et de la représentation d'un document XML en *EXPRESS*. Ces représentations ne sont réalisées que sur un sous-ensemble des schémas *XML Schema* sélectionné pour valider l'approche proposée.

3.1°) REPRESENTATION D'UN SCHEMA XML

Un schéma défini des éléments et des types, simples et complexes. Chaque **element** ou **type** est qualifié par *son_nom* comme propriété. L'*élément* possède en plus un type. L'Illustration 55 montre une représentation en *EXPRESS-G* de la méta modélisation.

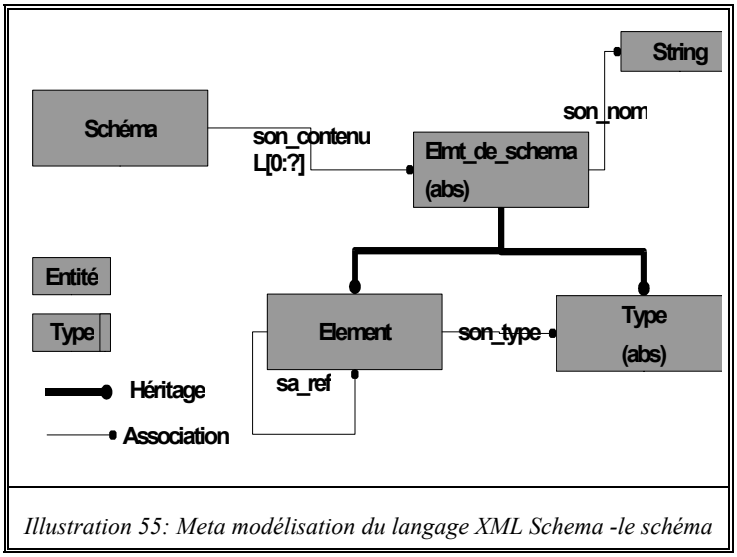


Illustration 55: Meta modélisation du langage XML Schema -le schéma

Les types simples sont décrits part un attribut *sa_base* qui les relient à une autre entité **simpletype**, type utilisateur ou à un type atomique. L'Illustration 56 présente cette relation.

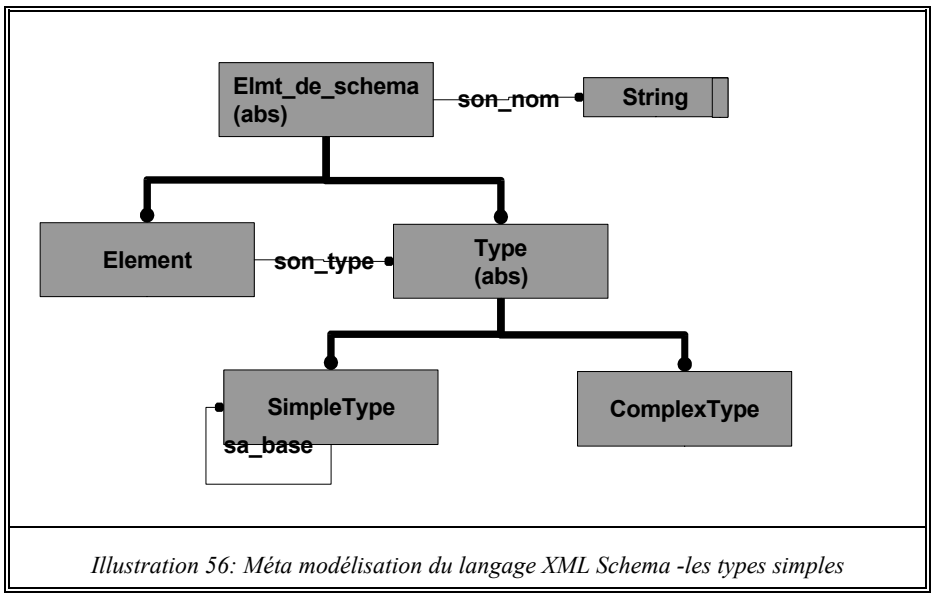
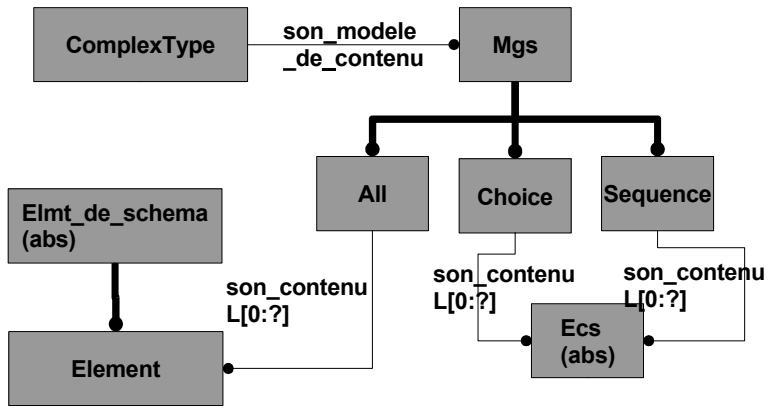


Illustration 56: Méta modélisation du langage XML Schema -les types simples

Les types complexe possèdent un attribut *son_modele_de_contenu*, qui, à l'aide des connecteurs **choix**, **sequence** ou **all** permettent de définir l'ordre d'apparition des sous éléments. Un connecteur contient donc soit des éléments, soit d'autres connecteurs. L'Illustration 57 montre la relation entre le modèle de contenu des types complexes et les différents connecteurs.



l'entité **ECS** désigne une entité abstraite, supertype des entités **element**, **choice** et **sequence** non représentée sur le dessin

Illustration 57: Méta modélisation du langage XML Schema -les modèles de contenu

3.2°) REPRESENTATION DU DOCUMENT XML

Le travail consiste à modéliser un document XML en termes d'un modèle *EXPRESS*, ceci à partir de la grammaire sous-jacente aux documents XML. Ainsi, un document se caractérise par une balise *racine*, associée par la propriété *sa_racine*. Cette balise peut contenir d'autres balises ou une chaîne de caractères (**string**) et est qualifiée par une propriété, *son_nom*. L'Illustration 58 décrit en *EXPRESS-G* cette représentation. La propriété *son_contenu* sera une liste de types utilisateurs **Bs**, un **TYPE SELECT** créé pour unir l'entité balise et le type chaîne de caractères.

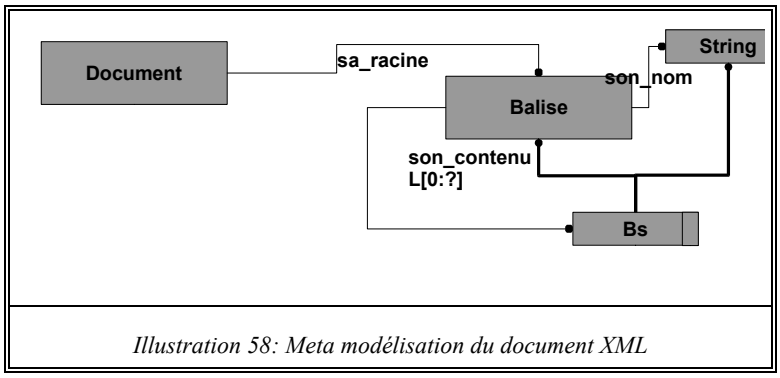


Illustration 58: Meta modélisation du document XML

4°) Conclusion

La méta modélisation permet de représenter les concepts *XML Schema* en langage *EXPRESS*. Pour valider cette approche, une implémentation était nécessaire. Un analyseur de document XML fut donc développé dans un environnement *EXPRESS* et il est présenté dans la partie suivante.

IV.Implémentation

1°) Description

L'application va permettre de prendre un document XML, accompagné de son schéma, puis vérifier si ce document est valide par rapport à son schéma. L'Illustration 59 résume le principe de fonctionnement.

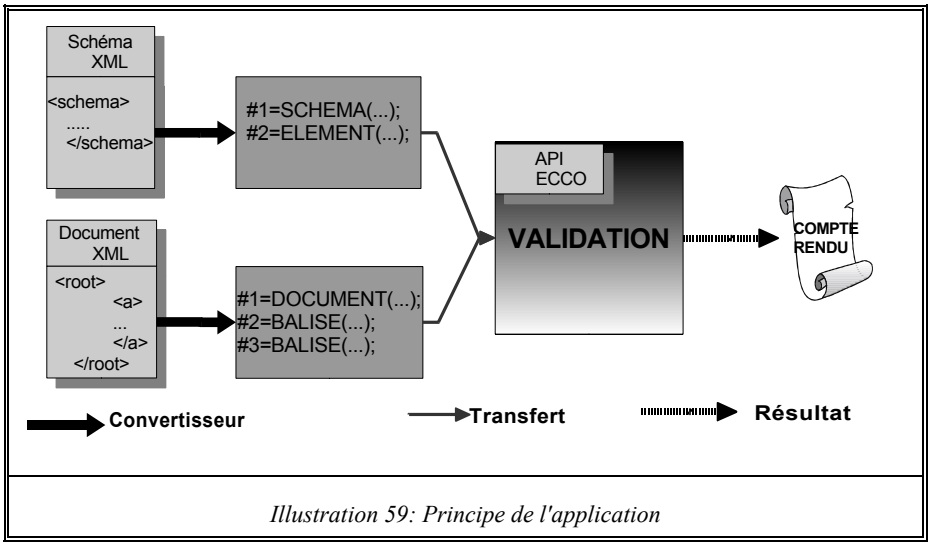


Illustration 59: Principe de l'application

Le schéma XML et le document XML sont convertis en fichier d'échanges *EXPRESS* conforme respectivement au modèle *EXPRESS* d'un schéma XML et au modèle *EXPRESS* d'un document XML. Par la suite, ces deux fichiers sont ouverts dans un outil de vérification de modèle *EXPRESS*, *ECCO*, puis validés en fonction de leurs modèles *EXPRESS*. En créant des fonctions de contrôle qui permettent de respecter les critères de validation d'un document XML par rapport à son schéma, l'environnement *ECCO* produira un compte rendu sur la validité du document.

La conversion de fichiers XML en fichiers d'échange *EXPRESS* n'a pas été traitée. L'accent a été mis sur la modélisation et l'expression des fonctions de contrôle.

2°) Outils de validation

2.1°) METHODES

L'outil va permettre dans un premier temps de valider l'ordre d'enchaînement des balises. Pour valider une balise, les trois critères suivants doivent être respectés.

- Premièrement, la balise doit posséder un modèle: il faut rechercher dans le schéma correspondant si un **ELEMENT** possédant le même nom existe. Si aucun modèle n'est présent alors la balise n'est pas valide.
- Deuxièmement, le contenu d'une balise doit être valide par rapport au modèle de contenu défini dans le schéma *XML Schema* associé. Si elle contient juste une chaîne de caractères, il doit être conforme au type définit dans le modèle (comme **simpleType**). Aucune solution n'est implémentée mais des ouvertures seront proposées par la suite. Si elle contient des balises alors l'ordre d'apparition de ces sous balises doit être vérifié par rapport au modèle de contenu défini dans le modèle (comme **complexType**). Les différents ordres d'apparition des balises seront modélisés sous une forme pour en permettre la validation. Ce point sera développé dans la partie section suivante.
- Troisièmement, les sous balises de la balise courante doivent être elles aussi valides.

La description *EXPRESS* d'une balise est donc complétée de trois attributs dérivés de type **BOOLEAN** pour valider ces trois conditions et d'un quatrième indiquant que la balise est valide.

2.2°) REPRESENTATION DU MODELE DE CONTENU D'UN TYPE COMPLEXE

Le premier souci est de donner une représentation au modèle de contenu d'un type complexe. L'exemple de l'Illustration 60 définit un élément *root* comme un *type complexe* possédant un *modèle de contenu*. Ce dernier est défini comme un *choix* entre deux *séquences* possibles: *a* puis *b*,

ou *c* puis *d*. Notons que l'élément *d* est seulement référencé: cela signifie que sa description est externe à son utilisation (voir partie droite de l'Illustration 60). Cet élément *d* est quant à lui décrit par un *modèle de contenu* définissant à nouveau un *choix* entre deux *séquences* de balises possibles: *z*, ou *y* puis *w*.

```
<element name='root'>
   <complexType>
   <sequence>
   <element name='x ' type=' . . '/>
   <choice>
   <sequence>
   <element name='a' type=' . . '/>
   <element name='b' type=' . . '/>
   </sequence>
   <sequence>
   <element name='c' type=' . . '/>
   <element ref='d'>
   </sequence>
   </choice>
   <element name='f' type=' . . '/>
   </sequence>
   </complexType>
</element>
```

```
<element name='d'>
   <complexType>
   <choice>
   <element name='z' type=' . . '/>
   <sequence>
   <element name='y' type=' . . '/>
   <element name='w' type=' . . '/>
   </sequence>
   </choice>
   </complexType>
</element>
```

Illustration 60: Exemple de schémas destinés à valider l'ordre

L'Illustration 61 démontre trois enchaînements de balises possibles dans un document XML conformément au *XML Schema* précédent.

```
<ROOT>
  <X></X>
  <A></A>
  <B></B>
  <F></F>
</ROOT>
```

```
<ROOT>
  <X></X>
  <C></C>
  <D>
  <Z></Z>
  </D>
  <F></F>
</ROOT>
```

```
<ROOT>
  <X></X>
  <C></C>
  <D>
  <Y></Y>
  <W></W>
  </D>
  <F></F>
</ROOT>
```

Illustration 61: Instances possibles

Par exemple, la validation de la deuxième instance se fera en plusieurs étapes exposées dans l'Illustration 62. La validation des types simples n'est pas encore implémentée mais une solution sera exposée dans les ouvertures en fin de section.

- Rechercher si l'**element** *root* existe dans le schéma XML;
- *root* est de type complexe donc vérifier que ces fils sont valides:
  - Rechercher si l'**element** *x* existe dans le schéma XML;
  - *x* est de type simple donc vérifier si sa valeur est conforme;
  - Mettre à jour l'attribut valide de la balise *x*;
  - Rechercher si l'**element** *c* existe dans le schéma XML;
  - *c* est de type simple donc vérifier si sa valeur est conforme;
  - Mettre à jour l'attribut valide de la balise *c*;
  - Rechercher si l'**element** *d* existe dans le schéma XML;
  - *d* est de type complexe donc vérifier que ces fils sont valides:
    - Rechercher si l'**element** *z* existe dans le schéma XML;
    - *z* est de type simple donc vérifier que sa valeur est conforme;
    - Mettre à jour l'attribut valide de la balise *z*;
  - Vérifier que le contenu de la balise *d* respecte l'ordre défini de son modèle de contenu;
  - Mettre à jour l'attribut valide de la balise *d*;
  - Rechercher si l'**element** *f* existe dans le schéma XML;
  - *f* est de type simple donc vérifier si sa valeur est conforme;
  - Mettre à jour l'attribut valide de la balise *f*;
- Vérifier que le contenu de la balise *root* respecte l'ordre défini dans son modèle de contenu;
- Mettre à jour l'attribut valide de la balise *root*.

Illustration 62: Etapes de validation

La mise en œuvre de cet algorithme général nécessite de pouvoir représenter à partir d'un schéma *XML Schema* l'ensemble des séquences de balises possibles conformément à une définition de modèle de contenu. Pour cela, nous proposons d'utiliser une structure de données particulières: un automate à états finis. L'idée est la suivante: à partir d'un modèle de contenu d'une balise, nous allons créer l'automate à états correspondants. Celui ci permettra de représenter l'ensemble des séquences de balises possibles à l'intérieur de la balise pour laquelle l'automate est construit. Puis, le document XML sera confronté progressivement pour chacune des balises à son automate afin de vérifier la validité de la séquence. L'Illustration 63 présente une application de cette approche au *XML Schema* de l'Illustration 60.

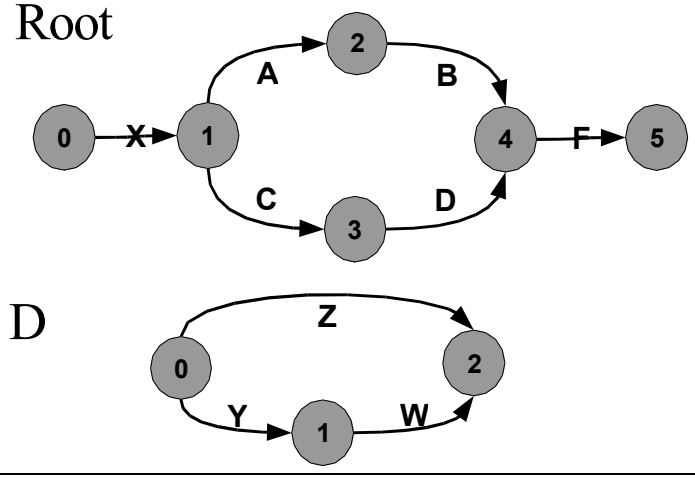


Illustration 63: Graphes de validation de l'Illustration 60

Ainsi si une balise contient la séquence de balise *x*, *a*, *b*, *f*, il y a conformité entre l'instance de document et la description *XML Schema*. Par contre si le contenu est *x*, *a*, *c*, *f*, l'instance n'est pas conforme au modèle de contenu défini.

Il reste maintenant à définir un processus de construction de cet automate. Celui-ci est exposé ci-après.

2.3°) REPRESENTATION PAR AUTOMATE

Un automate à états finis est défini de la façon suivante:

- un alphabet *A*;
- un ensemble fini d'états *E*;
- une relation de transition de *ExAxE*;
- d'un état initial I appartenant à *E*;
- d'un ensemble d'états finaux *F* incluse dans *E*.

Une transition (*e,a,e'*) dite de l'état e vers l'état e' et étiqueté par le symbole *a* est notée:

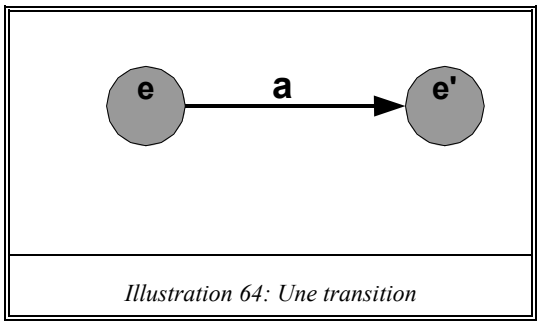


Illustration 64: Une transition

Un calcul de cet automate est une suite de transition.

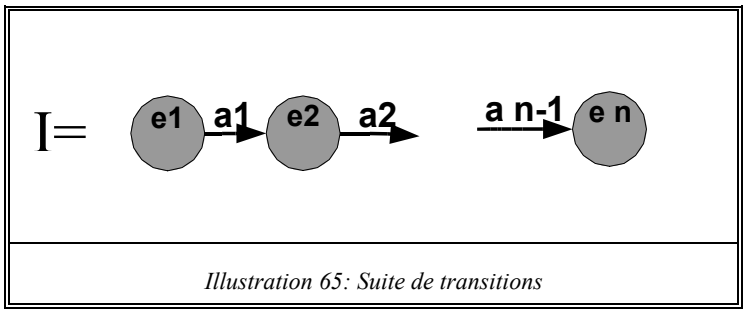


Illustration 65: Suite de transitions

Ce calcul est réussi si le dernier état e_n est un état final. On dit alors que le mot $a_1\ a_2...a_{n-1}$ est reconnu par l'automate.

Il suffit de poser le problème de la validation d'un **element**. Nous sommes dans un état d'attente et seule l'arrivée de la balise vraiment attendue (l'étiquette) permet de passer dans l'état suivant. Pour représenter cette condition, il suffit de modéliser deux états, le premier étant celui dans lequel on est et le deuxième celui que l'on souhaite obtenir. La transition porte comme étiquette le nom de l'**element** défini dans le modèle de contenu. Ainsi dans l'Illustration 64, l'**element** porte le nom *A* donc c'est la condition pour franchir la transition. Si la balise rencontrée possède le même nom alors l'état suivant est activé, sinon l'état renvoyé est un état indéterminé.

La **sequence** et e **choice** peuvent être représentés selon le même principe. La séquence est une suite de transition et le choix deux chemins différents pour atteindre le même état.

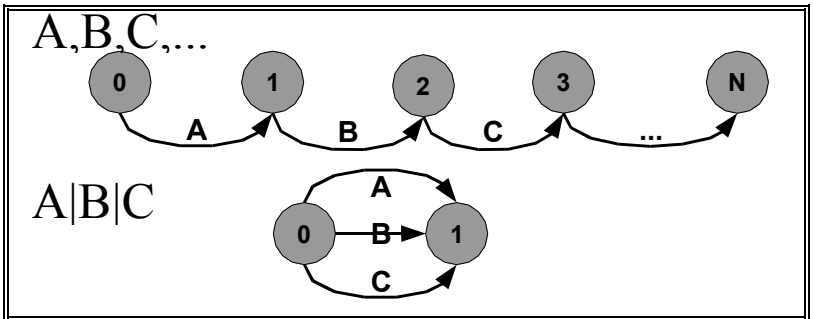


Illustration 66: Graphes de validation de la séquence et du choix

Il est maintenant possible de donner une représentation du modèle de contenu des exemples de l'Illustration 60, graphes donnés dans l'Illustration 63.

En arrivant sur une balise contenant des sous balises, la liste de leurs noms est comparée au différent chemins possibles. Dès lors qu'un nom de sous balise n'est pas présent dans les transitions attendues alors un état indéterminé est renvoyé et la balise n'est pas validée.

2.4°) OUVERTURES POSSIBLES

L'application ne prend pas en compte toutes les possibilités d'*XML Schema*, notamment le traitement des occurrences et la vérification des types simples. Pour ces deux problèmes, les automates peuvent encore être utilisés.

Lors de la déclaration d'un modèle de contenu, les **element**s peuvent être qualifiés des attributs **minOccurs** et **maxOccurs.** On spécifie le nombre de fois ou cet élément peut être instancié. Si ils ne sont pas saisis, la valeur est 1 pour les deux. L'Illustration 67 montre la représentation d'un **element** A dont **minOccurs=**3 et **maxOccurs=**5. Selon le chemin choisi, le nombre de A imposé sera soit de 3, soit 4, soit 5.

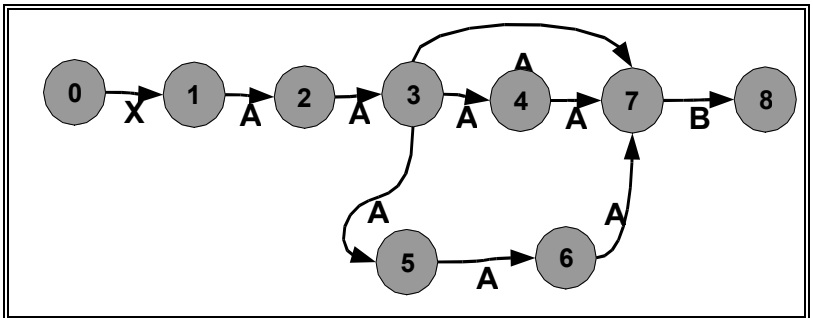


Illustration 67: Graphes des occurrences

Pour valider les types simples, la solution proposée utilise aussi des automates. Lorsque la balise est de type simple, la valeur à contrôler est dans une chaîne de caractères. Il suffit de balayer la chaîne (même méthode que celle employée pour vérifier la liste de balise) et de comparer chaque caractère à un domaine de valeur. Par exemple, si la chaîne de caractères doit être de type entier, le premier caractère peut être un signe + ou un signe – ou un nombre compris entre *0* et *9*. Les caractères suivant sont aussi compris dans l'intervalle *[0..9]*. Ce graphe ne possède pas d'état final.

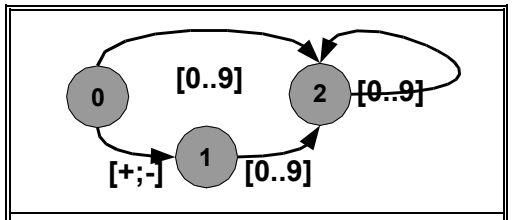


Illustration 68: Validation du type simple entier

Cette méthode de modélisation impose une étude au cas par cas pour chaque type de *XML Schema*.

V.Conclusion

La première partie du chapitre montrait les difficultés pour modéliser directement le langage *XML Schema* en *EXPRESS*. L'étude bibliographique a montré que le pouvoir d'expression d'*EXPRESS* était supérieur. Cependant, le *mapping* directe était assez difficile. La méta modélisation s'est donc imposée pour contourner ce problème et permettre d'exprimer tout ce que *XML Schema* peut transmettre avec notamment l'ordre d'apparition des balises. De plus, l'application développée a permis de valider cette approche. Cependant, tous les concepts tels que l'héritage ne sont pas encore implémentés.

Conclusion

Les travaux effectués dans le cadre de DEA avait pour but de confronter les pouvoirs d'expression des langages *EXPRESS* et *XML Schema*.

Le premier, langage de modélisation conceptuelle utilisé dans le domaine des données technique, apporte un pouvoir d'expression très puissant avec un ensemble de contraintes et fonctions venant compléter une modélisation et assurer sa cohérence. L'information véhiculée par les fichiers d'instances (les fichiers physiques) n'est pas compréhensible par l'homme. Le deuxième, langage de structuration de documents utilisé de plus en plus pour la modélisation de données et l'échange entre SGBD, permet de transmettre de l'information avec un fichier lisible et pouvant être interprété par les machines. Le pouvoir d'expression de la modélisation n'atteint cependant pas celui d'*EXPRESS*. Il était donc convenu de faire confronter ces deux approches pour modéliser des données riches sémantiquement et les représenter dans des fichiers qui possèdent une structuration syntaxique permettant une bonne lisibilité.

La partie mise en œuvre a permis de développer des principes de transformation entre ces deux langages. Ainsi, il est possible d'exprimer un modèle *EXPRESS* en *XML Schema* mais avec de nombreuses limitations. Les instances échangées dans un document XML gardent une structure compréhensible et les nombreux outils tels que les feuilles de style facilitent l'affichage de l'information. Il est par contre plus difficile d'exprimer un schéma XML en *EXPRESS*. Mais, la méta modélisation a contourné ce problème et les schémas *XML Schema* peuvent ainsi être exprimés et vérifiés en *EXPRESS*. L'approche fut validée en partie par le développement d'un analyseur de document XML

Les ouvertures sont nombreuses pour poursuivre le travail effectué.

Il faudrait dans un premier temps valider la transformation des concepts *EXPRESS* en *XML Schema*. Tous les concepts ne sont pas exprimables, et nous avons notamment vu des problèmes pour l'expression de contraintes globales. Cependant, il serait possible d'utiliser un modèle simplifié (par exemple un modèle simplifié de *PLIB*), ne contenant aucune contrainte ensembliste et de le transformer en schéma XML.

De plus, les limites des autres schémas, en cours d'écriture, et des feuilles de style *modèle* n'ont pas été abordées. Il faudrait approfondir l'étude sur ces langages.

L'application développée ne prend en charge qu'une faible partie du langage *XML Schema*. Il faudrait étendre la modélisation à la validation des types simples mais aussi à la dérivation et des contraintes d'unicité et référentielles pour en faire un analyseur complet de documents XML basés sur des description en *XML Schema*.

Bibliographie

[Bicarregui et al. 95] J. Bicarregui, B. Matthews
Integrating EXPRESS and SGML for Document Modelling in Control Systems Design; 6 Septembre 1995

[Biron et al. 01] P. V. Biron, A. Malhotra
XML Schema Part 2: Datatypes; W3C Recommendation, 2 Mai 2001
http://www.w3.org/TR/xmlschema-2/

[Bourret 01a] R. Bourret
Mapping W3C Schemas to Object Schemas, to Relational Schemas; Mars 2001
http://www.rpbourret.com/xml/SchemaMap.htm

[Bourret 01b R. Bourret
Mapping DTDs to Databases; published on XML.com, 9 Mai 2001
http://www.xml.com/pub/a/2001/05/09/dtdtodbs.html

[Bourret et al. 99] R. Bourret, J. Cowan, I. Macherius, S. St. Laurent
Document Definition Markup Language (DDML) Specification, Version 1.0, W3C Note, 19 Janvier 1999
http://www.w3.org/TR/NOTE-ddml

[Bray et al. 00] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler
Extensible Markup Language (XML) 1.0 (Second Edition); W3C Recommendation 6 octobre 2000
http://www.w3c.org/TR/2000/REC-xml-20001006

[Bray et al.98] T. Bray, C. Frankston, A. Malhotra
Document Content Description for XML; Submission to the World Wide Web Consortium, 31 Juillet 1998
http://www.w3.org/TR/NOTE-dcd

[Chen 76] P Chen
The Entity Relationship Model – Toward a Unified View of Data; ACM Transactions on Database Systems, Volume 1, N°1 (Mars 1976)

[Clark 99] J. Clark
XSL Transformations (XSLT) Version 1.0; W3C Recommendation, 16 Novembre 1999
http://www.w3.org/TR/xslt

[Clark et al. 99] J. Clark, S. De Rose
XML Path Language (XPath) Version 1.0; W3C Recommendation, 16 November 1999
http://www.w3.org/TR/xpath

[Davidson et al. 99] A. Davidson, M. Fuchs, M. Hedin, M. Jain, J. Koistinen, C. Lloyd, M. Maloney, K. Schwarzhof
Schema for Object-oriented XML 2.0, W3C Note, 30 Juillet 1999
http://www.w3.org/TR/NOTE-SOX/

[Fallside 01] D.C. Fallside

XML Schema Part 0: Primer; W3C Recommendation, 2 Mai 2001
http://www.w3.org/TR/xmlschema-0/

[Frankston et al. 98] C. Frankston, H.S. Thompson
XML-Data Reduced; 3 juillet 1998
http://www.ltg.ed.ac.uk/~ht/XMLData-Reduced.htm

[ISO 10303-11 94] ISO 10303-11
Industrial automation systems and integration -Product data representation and exchange - Part 11: Description Methods: The EXPRESS Language reference manual; 1994

[Jeliffe 00] R. Jeliffe
The Schematron: An XML Structure Validation Language using Patterns in Trees; Mai 2000
http://www.ascc.net/xml/resource/schematron/

[Klarlund et al. 00] N. Klarlund, A. Moller, M. I. Schwatzbach
DSD: a Schema Language for XML; Proc. 3rd ACML Workshop on formal Methods in Software Practice, 2000

[Klarlund et al. 99] N. Klarlund, A. Moller, M. I. Schwatzbach
Document Structure Description 1.0; 1999
http://www.brics.dk/DSD/dsddoc.html

[Layman et al. 98] A. Layman, E. Jung, E. Maler, H.S. Thompson, J. Paoli, J. Tigue, N.H. Mikula, S. De Rose
XML-Data; W3C Note, 5 Janvier 1998
http://www.w3c.org/TR/1998/NOTE-XML-data/

[Lee et al. 00] D. Lee, W.W. Chu
Comparative Analysis of Six XML Schema Languages; Août 2000
http://www.cobase.cs.ucla.edu/tech-docs/dongwon/ucla-200008.html

[Nijssen 81] G.M. Nijssen
An architecture for knowledge base systems; Proc. SPOT-2 conf., Stockholm, 1981

[Rumbaugh et al. 91] J. Rumbaugh, M. Blaha, W Premerlani, F. Eddy, W. Lorensen
Object Oriented Modelling and Design; Prentice-Hall International Editions, ISBN 0-13-630054-5, 1991

[Sardet 99] E. Sardet
Intégration des approches modélisation conceptuelle et structuration documentaire pour la saisie, la représentation, l'échange et l'exploitation d'informations. Application aux catalogues de composants industriels; Thèse de doctorat d'Informatique de l'Université de Poitiers, Septembre 1999

[Schenk et al. 94] D. Schenk, P. Wilson
Information Modelling The EXPRESS Way, Oxford University Press, 1994

[Thompson et al. 01] H.S. Thompson, D. Beech, M. Maloney, N. Mendelsohn
XML Schema Part 1: Structures; W3C Recommendation, 2 Mai 2001
http://www.w3.org/TR/xmlschema-1/