Ingénierie dirigée par les modèles en EXPRESS :

un exemple d'application

Hondjack Dehainsala, Stéphane Jean, Nguyen Xuan Dung, Guy Pierra

LISI-ENSMA 86961 Futuroscope Cedex, France

Résumé

Nous présentons dans cet article les outils d'Ingénierie Dirigée par les Modèles (IDM) existant autour du langage EXPRESS. Moins généraux que les environnements construits autour d'UML car, ils se limitent à la modélisation des données et non des applications, les environnements EXPRESS sont tout aussi anciens et matures. Ils sont aussi, d'après notre expérience, plus facile à maîtriser car ils proposent une approche homogène et complète pour la modélisation des données et la génération de code. Nous illustrons la puissance de ces environnements sur nos travaux de modélisation, de gestion, et de publications de données à base ontologique réalisés autour du modèle d'ontologie PLIB. Cet exemple, où modèles et données sont représentés au même niveau, permet également d'illustrer l'intêret pour un utilisateur final d'accéder de façon uniforme aux données et aux modèles.

1. Introduction

L'ingénierie des systèmes informatiques se base, de plus en plus sur des modèles. Ces modèles permettent de décrire les systèmes en cours de développement et leur environnement à différents niveaux d'abstractions. Ces abstractions permettent de concevoir des applications indépendantes des plate-formes cibles. Pendant longtemps, les modèles ont seulement constitué une aide pour les utilisateurs humains, permettant de développer manuellement le code final des applications informatiques. L'approche de l'Ingénierie Dirigée par les Modèles (IDM) consiste à programmer au niveau des modèles, représentés comme une instance d'un méta-modèle, et à les utiliser pour générer le code final des applications. Le MDA [4] (Modèle Driven Architecture) de l'OMG est une approche typique d'ingénierie dirigée par les modèles pour la conception d'applications. Le MDA est basé sur le standard UML pour la définition des modèles et sur l'environnement de méta-modélisation (MOF)[1] pour la programmation au niveau modèle et la génération de code.

Nous présentons dans cet article un autre environnement d'IDM. Construit autour du langage EXPRESS [2][9], normalisé à l'ISO en 1994, cet environnement est au moins aussi ancien et mature que l'environnement basé sur les standards de l'OMG (UML, OCL, MOF, MDA et XMI). Il a un domaine d'application plus restreint, puisqu'il ne traite que les modèles de données et non les modèles de systèmes. Il est, par contre, beaucoup plus simple à mettre en œuvre car le même langage, EXPRESS, peut être utilisé pour les différentes tâches impliquées dans une activité d'IDM : modélisation, meta-modélisation et instanciation, transformation de modèles. Nous illustrons la mise en œuvre de cet environnement sur le développement, par génération de code, d'un système de base de données à base ontologique réalisé au laboratoire. Gérant de façon uniforme les données et les modèles, cet exemple permet également de montrer l'intérêt des concepts clés de l'IDM, meta-modélisation et action au niveau modèle, lorsqu'ils sont transposés au niveau de l'utilisateur final qu'il soit utilisateur intéractif ou programmeur d'application.

L'article s'articule comme suit. Dans la section suivante, nous décrivons les différents aspects du langage EXPRESS : son pouvoir d'expression, sa syntaxe textuelle et graphique, la représentation d'instances et ses opérateurs déclaratifs de transformation de modèles (EXPRESS-X). Dans la section 3, nous présentons brièvemet les environnements de modélisation EXPRESS et l'environnement de programmation qu'ils fournissent. Dans la section 4, nous présentons à travers la mise en œuvre de notre architecture de base de donnée à base ontologique, l'utilisation du langage EXPRESS dans une approche d'IDM.

2. Modélisation de l'information et l'échange de données en EXPRESS

Le langage EXPRESS a été défini dans le cadre d'un projet de normalisation de l'ISO intilulé STEP (STandard for Exchange Product model data) initié à la fin des années 1980. STEP avait pour objectif la normalisation de modèles de données pour les différents produits fabriqués en entreprise [5]. Aucun langage existant à l'époque pour représenter de façon formelle des modèles de données structurés, le projet STEP a donc commencé par developper à la fois un langage de modélisation et toute une technologie associé. Ceci a abouti à la production d'une série de normes définissant un environnement appelé EXPRESS :

- un langage de modélisation de l'information : EXPRESS (ISO 10303-11 :1994) ;
- un format d'instances qui permet l'échange de données entre systèmes (ISO 10303-21 :1994) ;
- une infrastructure de méta-modélisation associée à une interface d'accès normalisée, appelée SDAI, pour accéder et manipuler simultanément les données et le modèle de n'importe quel modèle EXPRESS. Cette interface associé à un méta-modèle d'EXPRESS en EXPRESS a d'abord été definie indépendement de tout langage de programmation (ISO 10303-22 :1998) puis des implémentations spécifiques ont été spécifiées pour le langage C++ (2000), Java (2000), C (2001) ;
- un langage déclaratif de transformation de modèles (ISO 10303-14 :2002).

Enfin, le langage EXPRESS possède un langage procédural complet (analogue à PASCAL) pour l'expression de contraintes. Au prix de quelques extensions mineures, ce langage peut également être utilisé comme langage impératif de transformation de modèles.

Depuis une dizaine d'années, de nombreux environnements de modélisation EXPRESS sont commercialisés et le langage EXPRESS est utilisé dans de nombreux domaines, que se soit pour modéliser et échanger des descriptions de produits industriels, ou pour spécifier des bases de données dans des domaines divers, ou même pour fabriquer des générateurs de codes dans des ateliers de génie logiciel [8].

Nous présentons succintement dans la section qui suit les concepts généraux du langage EXPRESS et son environnement de modélisation.

2.1. Le langage

EXPRESS est un langage de modélisation formel de l'information. Il permet de modéliser les concepts d'un domaine et de spécifier les structures de données permettant de représenter ces concepts sous forme de données traitables par machine. C'est un langage orienté objet : il supporte l'héritage et le polymophisme. Il est également ensembliste : il permet les manipulations de collections. Nous présentons succintement ses composantes structurelles, descriptives et procédurales.

2.1.1 Connaissance structurelle

Le langage EXPRESS utilise la notion d'*entité* (ENTITY), pour réaliser l'abstraction et la catégorisation des objets du domaine de discours. Une entité est similaire à une classe des langages à objets à la différence qu'elle ne définit pas de méthodes. Les entités sont hiérarchisées par des relations d'héritage qui peuvent relever de l'héritage simple, multiple ou répété.

```
SCHEMA Universitaire ;
TYPE CLASSE = ENUMERATION OF (A1, A2, A3);
END_TYPE;
ENTITY Personne ;
SUPERTYPE OF ONEOF (Etudiant, Salairié);
  noSS : NUMBER;
  nom : STRING ;
  prénom : STRING;
  age : INTEGER;
  conjoint : OPTIONAL Personne;
DERIVE
 nom_prenom : STRING := Nom_complet (SELF);
UNIQUE ur1 : NoSS;
END_ENTITY ;
ENTITY Notes ;
  module_ref : STRING;
  note_40 : REAL ;
INVERSE
 appartient_à : SET[0 :?] OF Etudiant FOR ses_notes ;
WHERE
  wr1: {0 <=  note_40 <= 40};
```

```
DERIVE
  note_20 : REAL :=  note_40/2;
END_ENTITY ;

ENTITY Etudiant ;
SUBTYPE OF (Personne);
 sa_classe : CLASSE;
 ses_notes : LIST[0 : ?] OF NOTES;
END_ENTITY ;
ENTITY Salarié;
SUBTYPE OF (Personne)
  salaire : REAL;
END_ENTITY ;
ENTITY Etudiant_Salarié;
SUBTYPE OF( Salarié, Etudiant);
END_ENTITY ;

FUNCTION Nom_complet(per : Personne): STRING;
RETURN (per.nom + ' ' + per.prenom);
END_FUNCTION;

END_SCHEMA ;
```

FIG. 1 – Exemple d'un schema EXPRESS.

2.1.2 Connaissance descriptive

Une entité est décrite par des attributs. Les attributs permettent de caractériser les instances d'une entité par des valeurs. Ces valeurs peuvent être définies indépendamment de tout autre attribut (attributs libres) ou calculées à partir de valeurs d'autres attributs (attributs dérivés). Elles appartiennent au type de données associé à chaque attribut. Le langage EXPRESS définit quatre familles de types :

- Les types simples : ce sont essentiellement les types chaînes de caractères (STRING), numériques (REAL, BINARY, INTEGER) ou logiques (LOGICAL, BOOLEAN) ;
- les types nommés : ce sont des types construits à partir de types existant auxquels un nom est associé. Un type nommé peut être défini par restriction du domaine d'un type existant. Cette restriction peut être faite par la définition d'un prédicat qui doit être respecter par les valeurs du sous domaine créé. Il peut également être défini par enumération (ENUMERATION) ou par l'union de types (SELECT) qui, dans un contexte particulier, sont alternatifs.
- les types agrégats : ce sont des types qui permettent de modéliser les domaines dont les valeurs sont des collections. Les types de collections disponibles sont les ensembles (SET), les ensembles multivalués (BAG), les listes (LIST) et les tableaux (ARRAY). Un type collection n'a pas a être nommé : il apparait directement dans la définition de l'attribut qu'il type.
- Les types entités : un attribut d'un tel type représente une association.

2.1.3 Connaissance procedurale

Le langage EXPRESS est très expressif au niveau des contraintes. Les contraintes peuvent être classifiées selon deux grandes familles : les contraintes locales, qui s'appliquent individuellement sur chacune des instances de l'entité ou du type sur lequel elles sont définies, et les contraintes globales, qui nécessitent une vérification globale sur l'ensemble des instances d'une entité donnée.

Les contraintes locales (WHERE) sont définies au travers de prédicats auxquels chaque instance de l'entité (ou chaque valeur du type), sur lequel elles sont déclarées, doit obéir. Ces prédicats permettent par exemple de limiter la valeur d'un attribut en définissant un domaine de valeurs, ou encore de rendre obligatoire la valuation d'un attribut optionel selon certains critères.

Concernant les contraintes globales :

- La contrainte d'unicité (UNIQUE) contrôle l'ensemble de la population d'instances d'une même entité pour s'assurer que les attributs auquel s'applique la contrainte possèdent une valeur unique sur toute cette population.
- La contrainte de cardinalité inverse (INVERSE), permet de spécifier la cardinalité de la collection d'entités d'un certain type qui référencent une entité donnée dans un certain rôle. L'attribut inverse

exprime l'association entre une entité sujette et des entités référençant l'entité sujette par un attribut particulier.

– Les règles globales (RULE) ne sont pas déclarées au sein des entités mais sont définies séparément. Elles permettent d'itérer sur une ou plusieurs population d'entités pour vérifier des fonctions logiques qui doivent s'appliquer à l'ensemble des instances de ces populations.

Des fonctions (FUNCTION) et des procédures (PROCEDURE) peuvent être écrites. De plus, le langage EXPRESS propose un ensemble de fonctions prédéfinies : QUERY (requête sur les instances), SIZEOF (taille d'une collection), TYPEOF (introspection : donne le type d'un objet), USED_IN (calcul dynamique des associations inverses).

Pour illustrer cette présentation d'EXPRESS considérons la figure 1. Dans cet exemple, nous définissons un schéma de nom *universitaire*. Ce schéma est constitué de cinq entités. Les entités *étudiant* et *salarié* qui héritent de l'entité *Personne*. L'entité *étudiant_salarié* qui hérite des entités (héritage multiple) *étudiant* et *salarié*. Enfin l'entité *notes* qui est co-domaine de l'attribut *ses_notes* de *étudiant*. L'entité *Personne* définit un attribut dérivé(*nom_prenom*) qui est associé à la fonction EXPRESS (*nom_complet*) retournant la concaténation de l'attribut *nom* et *prénom* d'une instance de l'entité *Personne*. On peut remarquer la contrainte locale dans l'entité *notes* qui permet de vérifier que les valeurs de l'attribut *note_40* sont comprises entre 0 et 40.

2.1.4 Représentation Graphique d'EXPRESS

EXPRESS possède également une représentation graphique (sauf pour les contraintes) appelée EXPRESS-G. Cette représentation a pour objectif de donner une vue synthétique d'un schema EXPRESS et est adapté aux phases préliminaires d'une conception. La représentation graphique du schéma EXPRESS de la figure 1 est présentée dans la figure 2

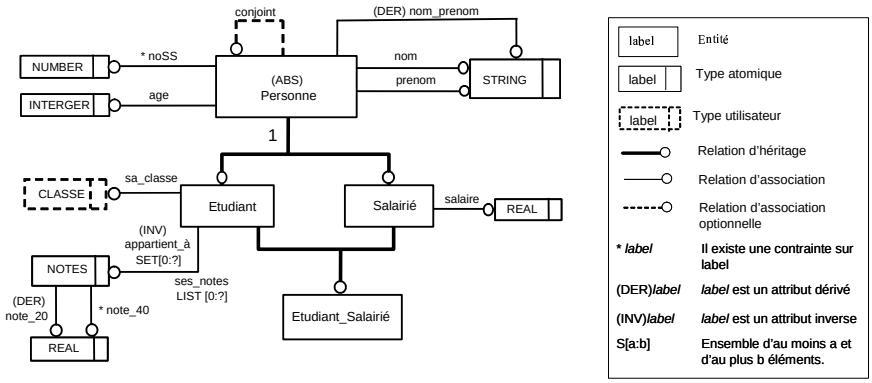

FIG. 2 – Exemple d'un Schema EXPRESS en EXPRESS-G.

2.1.5 Modularité

Destiné a concevoir des modèles de tailles importantes, par exemple l'ensemble des entités constituants un avion, EXPRESS intègre des mécanismes de modularité permettant la décomposition d'un modèle complexe en plusieurs sous-modèles. Ceci permet de faciliter la conception, la maintenance et la réutilisation d'un modèle. Un modèle EXPRESS appelé schéma peut faire référence à un ou plusieurs autres schémas soit pour intégrer tout ou partie des entités définies dans ceux-ci (USE), soit uniquement pour typer des attributs des schémas référencés (REFERENCE). Le découpage de la modélisation d'un domaine donné peut se faire selon deux approches qui peuvent être combinées :

– horizontal : chaque schéma modélise un sous domaine du domaine considéré ;

– vertical : chaque schéma représente une modélisation du domaine à un différent niveau d'abstraction.

2.2. EXPRESS comparé à UML pour la modélisation objet

Comme UML, EXPRESS est un langage de modélisation objet mais il possède quelques différences du point de vue de la représentation de certains concepts objets.

2.2.1 Représentation de relation d'association entre classes (entités)

Une association est une relation structurelle qui permet de lier n (n > 1) classes entre elles. Lorsque n est égal à 2 on dit que c'est une relation binaire. Contrairement à UML, Il n'existe pas de "constructeur" pour représenter des associations. Les associations s'expriment au moyen d'attributs dont le co-domaine est une autre entité et d'attributs inverses. Par exemple, pour une association binaire entre les classes *A* et *B*, on déclare un attribut *a2b* dans l'entité *A* et un attribut inverse *b2a* dans l'entité *B*. Les associations n-aires se représentent par une classe avec n attributs dont les co-domaines sont les entités.

2.2.2 Représentation des agrégations et compositions

Une agrégation correspond à une relation entre deux classes dans laquelle une extrémité de la relation (composite) joue un rôle prépondérant par rapport à l'autre (composant). En UML, il correspond à une association avec un losange du coté de l'agrégat. En EXPRESS, les agrégats s'expriment sous forme d'attributs de type collection (SET, LIST, BAG, ARRAY) d'entités. La composition est un type particulier d'agrégation (losange noir en UML) où les composants dépendent directement du composite (ou sont contenus physiquement dans le composite). En EXPRESS, elle s'exprime au moyen d'attributs de type entité par l'expression des cardinalités directes et inverses des attributs. Par exemple, une composition entre *A* et *B* s'exprimera par un attribut *a2b* de type *B*, son attribut inverse *b2a* ayant une cardinalité 0 :1 ou 1 :1. et pas d'autres associations.

2.2.3 Différents types d'héritage en EXPRESS

Comme UML, EXPRESS est un langage objet qui modélise l'univers en terme de catégories (classes). Dans le langage EXPRESS, une entité déclare ses super-types et peut déclarer aussi ses sous-types en utilisant trois opérateurs (ONEOF, AND, ANDOR) qui imposent ou permettent à une entité d'être instance d'une ou de plusieurs de ses sous-types. Illustrons ces opérateurs sur des exemples :

- Person SUPERTYPE OF ONEOF(male, female) : une instance de l'entité *person* peut être soit une instance de l'entité *male* soit une instance de l'entité *female*
- Person SUPERTYPE OF (employee ANDOR student) : une instance de *person* peut être une instance de l'instance *employee* mais aussi simultanément une instance de l'entité *student*.
- person SUPERTYPE OF (ONEOF(female,male) AND ONEOF(citizen, alien)) : une instance de l'entité *person* peut être une instance de l'entité *male* ou de *female* et une instance de l'entité *citizen* ou de l'entité *alien*.

2.2.4 Méthodes

EXPRESS est un formalisme de modélisation des données. Il permet de représenter contraintes d'intégrités, fonctions de dérivations, mais à la différence d'UML, il ne permet de représenter aucune autre méthode.

2.3. Représentation des instances

A tout modèle EXPRESS, est automatiquement associé un format de représentation textuel d'instances permettant de réaliser l'échange entre systèmes. Ce format est appelé fichier physique [ISO10303-21 :1994]. Chaque instance est identifiée pour un oid (Object IDentifier : #i). Elle est caractérisée par le nom de la classe instanciée. Enfin, elle est décrite par la liste des valeurs de ses attributs représentée entre

parenthèses. Les attributs optionnels sont désignés par '$', et les types énumérés par une notation encadrant l'identificateur par un point à chacune de ses extrémités. Les collections d'éléments sont définies entre parenthèses. Les attributs dérivés et inverses ne sont pas représentés. Les instances de la figure 3 sont celles des entités du schéma de la figure 2.

```
#2 = SALAIRE(13457, 'Jean', 'Hondjack' , 27, #3, 1000);
#3 = SALAIRE(23215, 'Jean', 'Nathalie', 25, #2, 2500);
...
#100 = ETUDIANT(02489, 'Nguyen', 'Sylviane' , 18, $, 'A3', (#101, #102));
#101 = NOTE('A3_120', 31);
#102 = NOTE('A3_121', 28);
```

FIG. 3 – Un exemple de fichier physique conforme aux entités de la figure 2.

2.4. Transposition de modèle : EXPRESS-X

EXPRESS-X est un complément déclaratif du langage EXPRESS (ISO 10303-14) [3] dont l'objectif est de permettre une spécification explicite des relations de correspondances (*mapping*) existant entre des entités de différents schémas EXPRESS. Ce langage supporte deux types de constructions spécifiques :

1. SCHEMA_VIEW qui permet de déclarer des *Vues* spécifiques des données d'un schéma EXPRESS ;
2. SCHEMA_MAP qui permet de déclarer des correspondances (*"mapping"*) de transformation entre des entités ou des vues d'un (ou plusieurs) schéma(s) EXPRESS source(s) vers un (ou plusieurs) schéma(s) EXPRESS cible(s).

Dans l'exemple de la figure 4, nous déclarons un mapping pour faire migrer les instances du schéma de la figure 2 en des instances du schema *person* (première colonne du tableau). La deuxième colonne du tableau montre la déclaration du mapping. Il transforme les instances de l'entité *salarié* en des instances d'entités *employee* de la manière suivante :

- le *noSS* de *salarié* correspond à l'attribut *id* de l'entité *employee* ;
- l'attribut *name* de l'entité *employee* est la concaténation des attributs *nom* et *prénom* de l'entité *salarié*.
- l'attribut *good_salary* de type BOOLEAN est dépendant de l'attribut *salaire* de l'entité *salarié*. La valeur de l'attribut est VRAI si la valeur de l'attribut *salaire* est supérieur à 1999 et FAUX dans le cas contraire.

La dernière colonne du tableau montre l'application du mapping sur deux instances de l'entité *salarié*. Si ce langage est essentiellement déclaratif, on peut néanmoins utiliser toute la puissance du langage procédural pour calculer des attributs (dérivés) du schema source destinés a être transposés dans le schéma cible.

Schéma cible	**Schéma de mapping**	**Instances de données**

SCHEMA Person ;
ENTITY Employee;
id : **NUMBER**;
name : **STRING** ;
good_salary : **BOOLEAN**;
UNIQUE url : id;
END_ENTITY ;
END_SCHEMA
	SCHEMA_MAP mapping_exemple;
	REFERENCE FROM Universitaire **AS SOURCE**;
	REFERENCE FROM Person **AS TARGET**;
	MAP U_Salarié_map **AS** emp : Employee;
	FROM sal : Salarié;
	SELECT
	emp.id := sal.noSS;
	emp.name := sal.nom_prenom;
	emp.good_salary := is_good(sal.salaire);
	END_MAP;
	FUNCTION is_good (salaire : **REAL**) : **BOOLEAN**;
	IF salaire > 1999 **THEN RETURN** ***TRUE***
	ELSE RETURN ***FALSE***;
	END_IF;
	END_FUNCTION;
	END_SCHEMA;
		(*Instances du schéma de source*)
		#2 = SALAIRE(13457,'Jean', 'Hondjack' , 27, #3, 1000);
		#3 = SALAIRE(23215,'Jean', 'Nathalie', 25, #2, 2500);
		
		(*Résultats de mapping *)
		#502 = EMPLOYEE(13457, 'Jean Hondjack', 'FALSE');
		#503 = EMPLOYEE(23215, 'Jean Nathalie', 'TRUE');

FIG. 4 – Exemple de transformation de modèle avec EXPRESS-X.

3. Environnement EXPRESS d'IDM

3.1. Les environnements de modélisation EXPRESS

EXPRESS étant un langage textuel, tout ensemble de schémas peut être compilé. le résultat de cette compilation est de générer des programmes qui permettent :

- de lire des instances figurant dans un fichier physique et de les représenter en mémoire ;
- d'accéder à ces instances par une interface graphique ;
- d'accéder à ces instances par programme : directement en EXPRESS, ou via une API dans les autres langages (C, C++, Java) ;
- de vérifier les contraintes définies au niveau schéma et de calculer les attributs implicites (dérivés et inverses) pour une population d'instances ;
- de sauvegarder les instances sous forme d'un fichier physique.
- d'accéder simultanément au schéma lui-même sous forme d'instance d'un méta-schema ;

Différents environnements existent. Certains permettent de faire persister les instances créées et manipulées.

Par exemple l'environnement ECCO [10] génére à partir de la compilation de modèle EXPRESS une application C++ qui représente automatiquement toutes les instances EXPRESS sous forme d'instances C++, et toutes les contraintes sous forme de méthodes C++. Le schéma lui-même est également reprsenté sous forme d'instances C++ d'un méta-schéma. Cette application, qui comporte en particulier les fonctions de lecture/ecriture de fichiers physiques, peut être exploitée par une interface graphique, par une API utilisable en C, C++, et Java et directement en EXPRESS par une application compilée avec le modèle. Une fois les traitements effectués, les instances doivent être sauvegardées sous forme d'un fichier physique pour être conservées.

D'autres environnements, tels que EXPRESS Data Manager (EPM Technology) sont liés à une base de données. Toute compilation d'un schéma EXPRESS génère automatiquement le schéma de base de données correspondant, les contraintes assurant l'intégrité de la base et l'interface SDAI permettant d'accéder à la fois aux schémas et aux instances.

3.2. Programmation EXPRES et transformation de modèles

Les environnements de modélisation EXPRESS permettent soit de coder un programme dans un langage classique (C, C++, JAVA) en utilisant des interfaces normalisées pour accéder aux données (niveau modèle et niveau instance) associées à un modèle EXPRESS, soit de coder un programme directement dans le langage procédural associé à EXPRESS. Si la première approche est adaptée pour certains types de programmes, elle présente l'inconvénient de nécessiter la connaissance à la fois d'EXPRESS et d'un langage de programmation. De plus, l'intégration de EXPRESS dans un langage de programmation pose le problème des dysfonctionnements (impedance mismatch) qu'impliquent les conversions de types. La seconde permet, par contre, une prise en main rapide qui permet d'exploiter un modèle EXPRESS dans un environnement homogène de développement. Le langage procédural associé à EXPRESS n'est pas complet. Pour en faire un langage complet de programmation, il ne lui manque que deux fonctions :

- les primitives d'entrée/sortie (read/write) ;
- la primitive ("run") permettant de demander l'execution d'une procédure.

Tous les environnements existants l'enrichissent de ces deux primitives pour en faire un langage de programmation à part entière permettant de réaliser un environnement simple à mettre en œuvre. Ce langage permet alors, en effet, de réaliser à la fois :

- le developpement d'un modèle conceptuel ;
- la spécification de l'ensemble des contraintes d'intégrités ;
- la manipulation et les transformations de modèles et d'instances.

Cette facilité de prise en main a été tester plusieurs fois au laboratoire : un stagiaire de niveau bac+5, ignorant EXPRESS, à réaliser une première version d'un programme de transformation de modèles lors

de sa première semaine de stage.

On présente dans ce qui suit quelques applications d'IDM développées dans le laboratoire au cours des dernières années dans des environnements de modélisation EXPRESS.

4. Un exemple de mise en œuvre de l'IDM en EXPRESS : la modélisation à base ontologique dans le projet PLIB

Le projet PLIB est un projet normatif de l'ISO visant à permettre le partage et l'échange sous une forme normalisée de catalogues de composants industriels. Un catalogue commençant toujours par définir ses propres classes de composants et ses propres propriétés de composants pour décrire ensuite chacun des composants constituant le catalogue, le projet PLIB a donc défini un modèle (la norme PLIB : série ISO 13584) permettant la description et l'échange simultané de deux niveaux de représentation :

- le niveau ontologique, où les classes et propriétés sont définies à la fois par des attributs (nom, définition, unité de mesure, ...), par des relations (propriétés applicables de chaque classe, propriété définissant le contexte d'évaluation d'autres propriétés, ...) et, éventuellement, par référence à des ontologies normalisées. C'est le modèle d'ontologie [6] ;
- le niveau extension où chaque composant est défini par son appartenance à une classe, dite classe de base, et des valeurs de propriétés. C'est le modèle de contenu.

4.1. Architecture PLIB

Ontologie et contenu étant échangé simultanément, tous deux sous forme d'instances de schémas EXPRESS, l'architecture des données PLIB correspond au trois premiers niveaux (M0-M1-M2) de l'architecture MOF [1].

M2	Entity(**Name**="CLASS", SuperEntité="",**Liste_Attributs=**(name, SuperClasse, Liste_Propriétés));
M1	Class(**Name**="Vis", **SuperClasse**= "Élément mécanique", **Liste_Propriétés=**(Longueur, Diamètre));
M0	Vis (**Longueur**=1, **Diamètre**=5);

FIG. 5 – Architecture PLIB.

L'exploitation des données PLIB amène alors naturellement à représenter explicitement les ontologies dans les bases de données de composants. Cette représentation permet en effet, (1) une intégration facile (automatique) de différents catalogues ; (2) une génération automatique d'interface d'accès aux données au niveau connaissance, c'est à dire au niveau ontologique. La représentation des ontologies dans les bases de données amène à la notion de Base de Données à Base Ontologique (BDBO) [6].

4.2. Base de Données à Base Ontologique

Une base de données à base ontologique possède deux caractéristiques : (1) ontologie et données sont stockées dans la BD et peuvent faire l'objet des mêmes traitements (insertion, mise à jour, requêtes, etc.) ; (2) toute donnée est associée à un élément ontologique qui en définit à la fois le sens et le type.

Une BDBO devant permettre les traitements similaires sur les parties ontologies et données, l'architecture de BDBO que nous avons définie, est constituée de quatre parties. Les deux premières parties sont constituées des deux parties traditionnelles des BDs : *données* et *méta-base* (qui contient les schémas logiques). Les deux autres parties sont similaires pour gérer les ontologies : la partie *ontologie* permet de gérer les ontologies, et la partie *méta-schéma* contient un méta-modèle réflexif du modèle d'ontologie

utilisée, et qui permet tous traitements sur les ontologies. Notons que cette architecture est tout à fait semblable aux différentes couches de l'architecture [IDM05]. Ontologies et composants correspondent aux niveaux M1 et M0. La couche méta-modèle M2 correspond au modèle d'ontologie, la couche méta-méta-modèle M3 (MOF model) correspond au méta-modèle du langage de définition du modèle d'ontologie lui-même réflexif afin de permettre de s'adapter facilement à des changements du modèle d'ontologie.

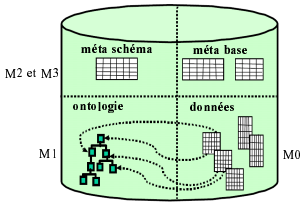

FIG. 6 – Architecture de BDBO : le modèle OntoDB.

4.3. Ingénierie dirigée par les modèles en EXPRESS

Nous montrons brièvement dans cette partie comment nous avons utilisé l'environnement EXPRESS d'IDM pour la mise en œuvre de notre architecture de BDBO, sachant que, pour pouvoir lire des catalogues conformes à la norme PLIB, nous devons pouvoir intégrer automatiquement et gérer de façon homogène, des populations d'instances dont les données, le schéma et l'ontologie sont chargés dynamiquement.

Notre implémentation de BDBO devant être effectuée ,pour des raisons de performances, dans un univers relationnel (ou relationnel objet), la mise en œuvre de notre architecture nécessite l'implémentation de chacun des modules suivants pour chaque partie de l'architecture (sauf la méta-base générée par le SGBD) :

1. générateur de la structure des tables ;
2. instancieur des tables avec des instances fournies sous forme de fichiers physiques EXPRESS ;
3. générateur d'APIs pour l'accès aux données des tables.

4.3.1 Représentation de l'*ontologie* et du *méta-schéma*

La représentation des parties *ontologie* et *méta-schéma* correspond à la représentation de deux modèles EXPRESS : le modèle d'ontologie PLIB et un méta-modèle d'EXPRESS en EXPRESS. L'implémentation des trois modules est donc identique pour la mise en œuvre de ces deux parties. L'implémentation des différents modules est faite de façon générique i.e. indépendamment de tout modèle EXPRESS particulier. Les schémas EXPRESS du modèle d'ontologie et d'EXPRESS sont donc traités comme un paramètre pour générer la représentation, respectivement, de la partie *ontologie* et *méta-schéma*. Le schéma EXPRESS paramètre est d'abords compilé et convertis sous formes d'instances d'un meta-schéma EXPRESS (SDAI-dictionary-Schema par exemple) par l'environnement d'IDM EXPRESS utilisé, puis nous développons un programme EXPRESS générique pour réaliser la génération du schéma logique. La figure 7a montre un exemple d'un méta-schéma (simplifié) d'EXPRESS. La figure 7b montre un exemple de modèle d'ontologie également simplifié. Dans la figure 7c, nous avons une instanciation du modèle d'ontologie sous forme d'instances EXPRESS du méta-schéma EXPRESS.

Les programmes de peuplement de l'*ontologie* et du *méta-schéma*, d'une part, et de générations d'APIs, d'autre part, sont implémentés sous forme de deux programmes génériques exploitant le schéma traité en tant qu'instances du méta-schema. Un aperçu du resultat est donné dans la figure 9. La figure 9a présente quelques tables de l'ontologie peuplées avec le schéma de la figure 2 (que nous l'avons considéré comme une ontologie). La figure 9b montre une classe (*data_property*) du package Java généré à partir du modèle d'ontologie pour l'accès aux données des tables de l'ontologie.

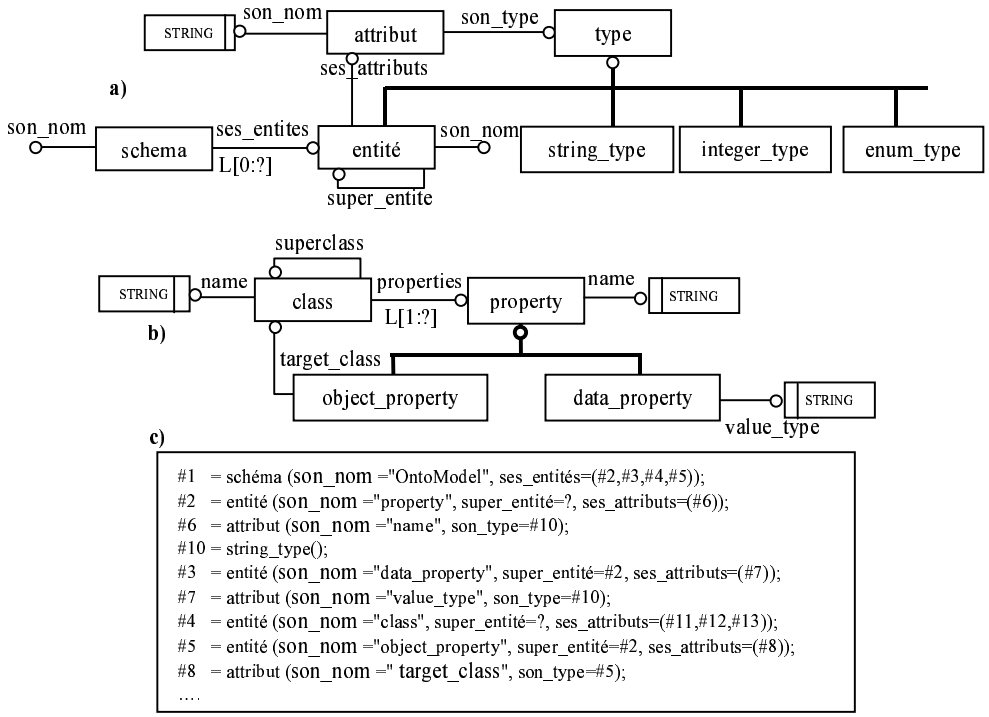

FIG. 7 – a) un exemple de Méta-Schéma EXPRESS simplifié ; b) un exemple de modèle d'ontologie en EXPRESS ; c) instanciation du modèle d'ontologie dans le méta-schema EXPRESS.

```
Soit s ∈ Schema
∀ e ∈ s.ses_entités;
        Affiche ("CREATE TABLE "+e.son_nom+"(")
        Affiche("  OID  INT32,")
        ∀ a ∈ e.ses_attributs ;
          SWITCH(TYPEOF(a.son_type)) {
                case "String_Type"   : Affiche(a.son_nom+"   VARCHAR,");
                case "Integer_Type" : Affiche(a.son_nom+"  INTEGER,");
                case "Entity_Type"   : Affiche(a.son_nom+"   INT32 FOREIGN KEY (OID,"+
                                                                  a.son_type.son_nom+),");
                case ...
        }
        …
```

Exemple de résultat :

```
CREATE TABLE object_property (
        OID INT32,
        target_class INT32
FOREIGN KEY (OID,class),
…
```

FIG. 8 – Exemple algorithme exploitant le méta-schema pour la génération de requêtes SQL (TYPEOF permet l'intropection).

a)

class			
OID	**name**	**superclass**	**…**
id_personne	personne		
id_etudiant	etudiant	id_personne	
id_salarie	salarie	id_personne	
…	…	…	

object_property				
OID	**name**	**invproperties**	**target_class**	**…**
id_conjoint	conjoint	id_personne	id_personne	
…	…	…	…	

data_property				
OID	**name**	**invproperties**	**value_type**	**…**
id_noSS	noSS	id_personne	String	
id_nom	nom	id_personne	String	
id_prenom	prenom	id_personne	String	
id_age	age	id_personne	Int	
id_salaire	salaire	id_salarie	Float	
id_sa_classe	sa_classe	id_etudiant	String	
…	…	…	…	

b)

```
Package Package_Ontologie;
class data_property extends property{
data_property(){…}
String getValue_Type(){return value_type}
void setValue_Type(String val){..}
….
```

FIG. 9 – a) Tables de la partie ontologie ; b) exemple de classe Java pour l'accès aux ontologies.

4.3.2 Représentation de la partie *données*

A la différence d'un modèle conceptuel qui *prescrit* les propriétés devant être utilisées pour décrire une instance de classe, une ontologie ne fait que *décrire* les propriétés qui *peuvent* être utilisées pour décrire des instances (propriétés dites *applicables*). Dans le modèle PLIB, l'extension d'une classe est définie sous forme (1) d'une liste instances chacune est donc définies sous forme d'une liste de couples (code propriété applicable, valeur correspondante), et (2) de contraintes d'unicité permettant d'identifier les propriétés susceptibles d'être utilisées comme clé. Toutes les propriétés utilisées doivent être applicables pour la classe. Si les différentes instances ne sont pas décrites par le même ensemble de propriété, alors l'union des propriétés est utilisée en complétant par des valeurs nulles les valeurs manquantes.

4.3.2.1 Définition du modèle logique (DDL) de la partie données :

Dans le modèle PLIB, les classes sont décrites par de propriétés qui peuvent être de type simple (chaîne de caractères, entier) ou complexe (agrégat, composition, association). Une classe peut aussi hériter des propriétés de ses super-classes. L'univers du SGBD cible, très souvent, ne possède pas le même pouvoir d'expression (par exemple le modèle relationnel). Il est donc indispensable d'établir une transposition des mécanismes objet de PLIB vers les mécanismes du SGBD cible. Le générateur que nous avons implémenté, reçoit en paramètre un fichier physique contenant un ensemble d'instances de classes définies dans une ontologie dèja intégrée (cf. section 4.3). Il produit, d'abord un modèle logique en fonction des correspondances que nous avons définies, puis, peuple ce modèle avec les instances lues. La figure 10a montre les tables d'instances de la classe *étudiant* et *salarié*.

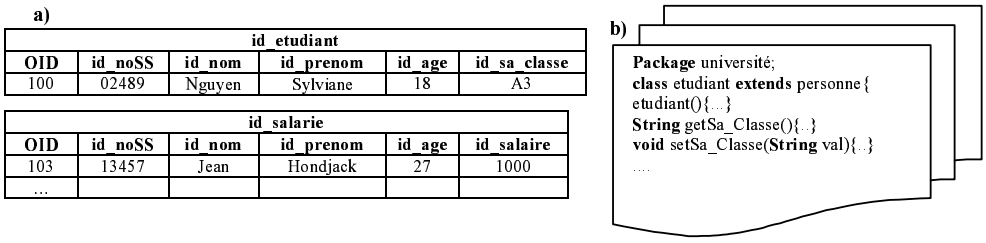

FIG. 10 – a) Tables partie données ; b) exemple de classe Java pour l'accès aux instances des classes.

A partir de l'exemple de la figure 10a, on peut faire les remarques suivantes : (1) les noms des tables et des colonnes sont respectivement l'identifiant des classes et des propriétés de la classe dans l'ontologie. Les correspondances entre les données et l'ontologie sont ainsi automatiquement représentées dans la partie *méta-base* ; (2) la base de données ne contient pas de tables d'instances pour la classe *personne* (qui est abstraite) et seules les propriétés valuées sont traduites sous forme de colonnes de tables. Ceci résulte du choix effectué (représenté dans le programme de génération) pour la transformation du modèle objet en modèle relationnel.

L'ontologie peut être utilisée pour générer une interface de programmation (API) permettant d'accéder à la partie *données*. Dans cette API, le code permettant de vérifier les contraintes définies au niveau de l'ontologie est généré pour assurer l'intégrité des données de la base de données. La figure 10b montre la classe Java *étudiant* qui hérite de la classe *personne* respectivement images des classes *étudiant* et *personne* stockées l'ontologie. Cette classe possède un constructeur pour créer des instances dans la BD et des méthodes get/set pour chaque propriété dans la classe d'ontologie.

Les ontologies peuvent être exploitées pour générer des interfaces utilisateurs pour créer, supprimer, modifier, des instances des classes. Ces interfaces utilisateurs sont également générées de façon générique et se basent sur les APIs générées précédemment pour réaliser les tâches qui leur incombent. La figure 11a montre sous forme d'une table, toutes les instances de la classe *étudiant*. La figure 11b montre une page Web dynamique permettant la mise à jour d'une instance de la classe *étudiant*.

La figure 12 montre l'ensemble de l'architecture dévéloppée en utilisant les techniques de l'IDM à partir de trois modèles EXPRESS :

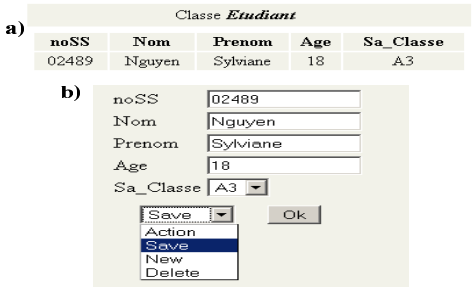

FIG. 11 – Interface graphique générée pour la manipulation des instances de la classe *étudiant*.

– le modèle PLIB de contenu de classe ;
– le modèle d'ontologie PLIB ;
– le méta-modèle d'EXPRESS en EXPRESS.

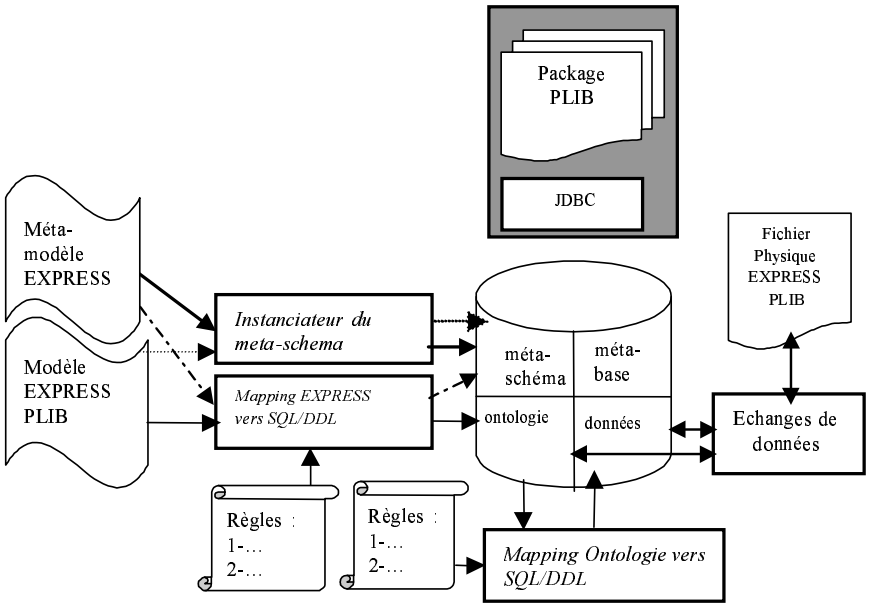

FIG. 12 – Architecture des modules développées.

Il est important de souligner que tous ces développements ont été faits dans un environnement de développement unique et homogène ECCO (EXPRESS Compiler COmpiler) qui ne nécessite que la maîtrise du langage EXPRESS pour effectuer :

– la modélisation de l'information ;
– la spécification des structures de données ;
– la spécification des contraintes ;
– la programmation des transformations.

Cette homogénéité rend les environnements d'IDM basés sur EXPRESS particulièrement simple.

5. Conclusion

Les techniques d'ingénierie dirigée par les modèles sont des techniques très prometteuses qui permettent, en manipulant des modèles sous forme d'instances de méta-modèles, d'une part de générer l'essentiel du code de nombreuses applications, et, d'autre part, de rendre les générateurs à la fois génériques par rapport à des familles de modèles, et paramétrables, par exemple, par la méta-description de la plate-forme cible.

Un environnement fréquemment utilisé est l'environnement développé à partir des standards de l'OMG : UML, OCL, MDA et MOF, XMI.

Dans ce papier, nous avons présenté un autre environnement basé sur le langage EXPRESS. Cet environnement est plus spécialisé, car il ne traite que de modèles de données et non de modèles de systèmes. Mais il est plus ancien, car normalisé à l'ISO à partir de 1994, au moins aussi mature, car de nombreux systèmes commercialisés existent implémentant l'ensemble des normes définies autour d'EXPRESS. Un tel environnement ne nécessite la maîtrise que d'un seul langage et il permet :

- de définir un modèle conceptuel ;
- de spécifier les données représentant des instances de ce modèle ;
- de contraindre précisément le modèle ;
- de programmer de façon déclarative et impérative des transformations de modèles.

Pour donner un ordre de grandeur de la complexité des environnements d'ingénierie dirigée par les modèles en EXPRESS, un étudiant de troisième cycle ne connaissant pas EXPRESS peut écrire ses premiers programmes basés sur une génération de code en une semaine environ.

Les techniques d'ingénierie dirigées par les modèles sont également intéressantes pour l'utilisateur final, qu'il soit utilisateur interactif ou programmeur d'application, en lui permettant d'accéder directement non seulement aux données, mais également et sous la même forme, au modèle de ces données. Nous avons illustré cette dimension de l'IDM en présentant un exemple de mise en œuvre d'IDM en EXPRESS portant sur la génération d'un environnement de modélisation à base ontologique développé dans notre laboratoire. Développé par génération de code à partir de trois modèles EXPRESS : le méta-modèle d'EXPRESS, le modèle d'ontologie PLIB et le modèle d'instances PLIB, l'environnement de base de données à base ontologique OntoDB gère, et rend disponible sous forme d'instances, à la fois les données, leur modèle physique et, la conceptualisation dont elles résultent. Ceci permet de fournir automatiquement à l'utilisateur interactif non seulement un accès direct aux données, mais aussi un accès direct aux ontologies et aux données via l'ontologie, indépendamment donc de la représentation physique des données. Ceci permet également de fournir au programmeur d'application, l'accès par programme aux quatre niveaux désignés par M0, M1 M2 et M3 dans l'architecture MOF.

Du point de vue perspective, nous travaillons actuellement, d'une part, sur des règles génériques de transformations de modèles [7] (exemple : absorption de classes, suppression des classes abstraites, etc.) permettant d'exploiter les contraintes d'intégrité existant au niveau instances dans un modèle EXPRESS pour rendre plus efficace les modèles de données objets lorsque l'on change d'environnement d'exploitation (Environnement EXPRESS, RDB, RODB, OODB, XML). D'autre part, nous travaillons sur la conception d'un langage de définition et de manipulation de données permettant de manipuler simultanément les instances de niveaux M0 et M1.

Références

[1] Object management group : Meta object facility specification version 1.4. *OMG document formal*, 2003.

[2] IS010303.02. Product data representation and exchange - part 2 : Express reference manual. *ISO*, (055), 1994.

[3] IS010303.14. Product data representation and exchange - part 14 : Express-x language reference manual. *ISO*, (088), 1999.

[4] OMG. Model driven architecture. *http ://www.omg.org/mda/*.

[5] J. Owen. Step : an introduction. *Informations Geometers*, 1993.

[6] G. Pierra, H. Dehainsala, Y. A. Ameur, and L.Bellatreche. Base de données à base ontologique : principe et mise en oeuvre. In *To Appear in Ingénierie des Systèmes d'Information (ISI)*, 2005.

[7] G. Pierra, H. Dehainsala, N. Ngabiapsi, and M. Bachir. Transposition relationnelle d'un modèle objet par prise en compte des contraintes d'intégrité de niveau instance. *congrès INFORSID'05*, pages 455–470, 2005.

[8] A. Plantec. *Utilisation de la norme STEP pour la spécification et la mise en œuvre de générateurs de code*. PhD thesis, Université de Bretagne Occidentale, fevrier 1999.

[9] D. Schenk and P. Wilson. *Information Modeling The EXPRESS Way*. Oxford University Press, 1994.

[10] G. Staub and M. Maier. Ecco tool kit - an environnement for the evaluation of express models and the development of step based it applications. *User Manual*, 1997.