Etude de la persistance dans les SGBDOO

Eric Amiel, Marie-Jo Bellosta, Patrick Valduriez, Fabienne Viallet

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

1 9 9 2

25ème anniversaire

N° 1592

Programme 1
Architectures parallèles, Bases de données,
Réseaux et Systèmes distribués

ETUDE DE LA PERSISTANCE DANS LES SGBDOO

Eric AMIEL
Marie-Jo BELLOSTA-TOURTIER
Patrick VALDURIEZ
Fabienne VIALLET

Février 1992

RR_1592

Etude de la persistance dans les SGBDOO*

Eric Amiel, Marie-Jo Bellosta-Tourtier[1], Patrick Valduriez, Fabienne Viallet[2]

INRIA Rocquencourt, BP.105, 78153 Le Chesnay-Cédex

Résumé

Les SGBDOO ont pour principale motivation d'apporter une solution nouvelle aux problèmes d'applications ayant à manipuler de grands volumes de données complexes. En combinant les avantages des SGBD et des systèmes OO, ils apportent un meilleur support des objets complexes et une plus grande productivité des programmeurs d'applications par rapport aux systèmes relationnels. Dans ce contexte, un problème important est la mise en oeuvre de la persistance dans un langage de programmation. Cet article fait le point sur les différentes solutions apportées à ce problème en examinant un échantillon représentatif des SGBDOO existants : Exodus, Ontos, Ode, O2, Orion et GemStone.

Mots-clés : *persistance, système de gestion de bases de données orientées-objet (SGBDOO), systèmes orientés-objets, langage de programmation, compilation.*

Persistence in OODBMS

Abstract

OODBMS' main motivation is to bring a new solution to the problems of applications dealing with large volumes of complex data. By combining the advantages of DBMS and OO systems, they achieve a better support for complex objects increasing productivity of application programmers compared to relational systems. In this context, a major problem is how to implement persistence in the programming language. This article offers a critical survey of the various solutions to this problem by examining a representative panel of existing OODBMS : Exodus, Ontos, Ode, O2, Orion et GemStone.

Mots-clés : *persistence, object-oriented database systems (OODBMS), object-oriented systems, programming languages, compiling.*

---

* Ce travail a été partiellement financé par une convention du CNET Lannion.

1 Collaboratrice extérieure : INT, 9 rue Charles Fourier, 91011 Evry-Cédex

2 Collaboratrice extérieure : France Telecom, 31000 Toulouse

SOMMAIRE

1. INTRODUCTION

Les SGBD traditionnels, bâtis sur le modèle réseau ou relationnel, ont été conçu pour les applications de gestion classique. En exploitant la régularité des données à structure simple, e.g., des fichiers d'articles, ils peuvent bien supporter les applications transactionnelles ou décisionnelles. Cependant, d'importants domaines d'applications qui ne relèvent pas de la gestion classique, e.g., la CAO ou la bureautique, expriment des besoins similaires en matière de bases de données (BD) avec des exigences nouvelles en puissance de calcul et en support d'objets complexes. Pour répondre à ces applications, il est nécessaire d'intégrer les techniques BD et langages de programmation (LP). Une première approche suivie par des langages tels que Pascal, CLOS ou Eiffel est d'utiliser un gestionnaire de fichier pour stocker les données sur disque. Mais un problème de concordance de typage se pose : une fois mis dans un fichier, l'objet est considéré comme une chaîne de bytes et perd son type; des procédures de traduction pour la lecture et l'écriture sont donc nécessaires. Une deuxième approche suivie par les SGBD classiques introduit des requêtes dans un langage de programmation (SQL dans C). Elle souffre, cependant, de la coexistence de deux systèmes de typage non concordants[3]. Une solution à ces problèmes de concordance de typage est d'avoir un modèle de données unique pour le langage et le système de stockage. Pour réaliser l'intégration du monde LP et SGBD, les SGBDOO choisissent l'approche orientée-objet (OO) parce qu'elle permet la description d'objets complexes et favorise la modularité, la réutilisation et l'extensibilité.

Dans cet article, nous nous intéressons aux aspects programmatifs des SGBDOO afin de dégager les mécanismes mis en oeuvre pour rendre persistant un langage de programmation. L'étude de la persistance d'un système débute par une présentation globale de ce système au travers de son *architecture fonctionnelle*. L'absence d'un modèle unique OO nous conduit à présenter les spécificités du *modèle de données* de chaque système. La mise en oeuvre de la persistance est alors abordée en présentant ses aspects modèle et langage : l'aspect modèle comprend ses principes et ses mécanismes de réalisation; l'aspect langage est sa mise en oeuvre dans des *langages de programmation*. L'architecture fonctionnelle, le modèle de données, le modèle de persistance et les langages de programmation persistants constituent donc les principaux points abordés pour l'étude de chaque SGBDOO. Pour une introduction aux BDOO, nous renvoyons le lecteur aux ouvrages spécialisés, e.g., [Del91, Gar90].

Un échantillon représentatif des SGBDOO existants, prototypes de recherche ou produits commerciaux, est présenté. Ces systèmes ont été choisis en regard de leur originalité et de la richesse de leurs fonctionnalités. Les trois premiers systèmes ont choisi le langage C++ mais présentent trois approches différentes au traitement de la persistance : Exodus [Car88,

[3] Impedance mismatch

Ric89, Ric90, Sch90] est un gérant d'objets extensible[4] introduisant syntaxiquement la persistance dans le langage C++ avec son langage E en modifiant le compilateur C++; Ontos [And87, Ont89] est une bibliothèque de classes C++ offrant la persistance par héritage d'une classe spécifique; Ode [Agr89a, Agr89b, Agr91, Geh91] introduit la persistance syntaxiquement avec le langage O++ grâce à un précompilateur en amont du compilateur C++. Les deux systèmes suivants ont leur propre modèle de données et leur propre langage : Orion [Kim87, Kim89a, Kim89b], basé sur le langage LISP, met l'accent sur la gestion de versions et l'évolution de schéma de BDOO; O2 [O291, O2B92] propose un langage de commandes pour la définition de son schéma, un langage O2C dérivé de C pour la manipulation de ses données, des interfaces multi-langages et un environnement de programmation très évolué. Enfin, le sytème [But91, Cop84, Mai86, Pen87, Pur87] est le premier système ayant doté un langage OO, Smalltalk, de la qualité de persistance. Il est à noter que les systèmes Ontos, O2 et GemStone sont des produits commerciaux.

Pour illustrer les différents systèmes, nous utilisons l'exemple défini dans [Bel91] qui modélise une maison d'édition. Une publication est soit un journal soit une conférence soit un livre; elle a un titre, un ensemble d'auteurs, un ensemble d'illustrations et un ensemble de pages. Une publication est imprimée, revue, distribuée et vendue; à chacune de ces actions correspond une méthode particulière. Un journal possède un nom, une date et un numéro. Un livre possède un éditeur, une page de préface et une page de titre. Une page de titre et une page de préface sont des spécialisations de pages.

Cette étude est organisée comme suit : la section 2 présente les caractéristiques des différents SGBDOO, pertinentes pour la résolution de la persistance; les sections 3 à 8 décrivent respectivement les systèmes suivants : Exodus, Ontos, Ode, O2, Orion et GemStone. La section 9 présente une synthèse de ces systèmes et conclut en insistant sur les nouvelles perspectives.

2. NOTIONS GENERALES

2.1. L'architecture fonctionnelle

L'architecture fonctionnelle des systèmes étudiés se décompose canoniquement en une *machine langage*, un *dictionnaire d'objets* et un *gérant d'objets* (cf. figure 1). La machine langage identifie les définitions, les instructions et les expressions liées à la persistance dans les programmes d'application et les traduit en appels de plus bas niveau au gérant d'objets. Le dictionnaire d'objets fournit toute l'information concernant les objets persistants et la définition des classes, appelée *schéma*. Le gérant d'objets fournit les fonctionnalités noyau

[4] Un gérant d'objets extensible offre à un concepteur de SGBD des outils permettant d'adapter son SGBD aux besoins spécifiques d'une application.

d'un système de bases de données : gestion d'objets complexes, index, reprise sur pannes et contrôle de concurrence.

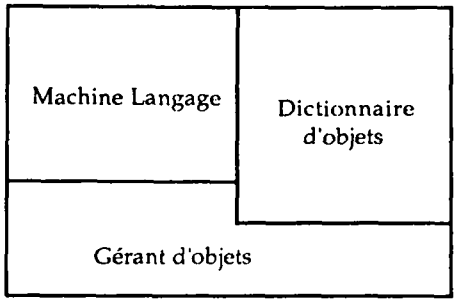

Figure 1 : Architecture Fonctionnelle Canonique des SGBDOO

2.2. Les Concepts Orientés-objet

L'approche OO souhaite répondre aux besoins de modularité, d'extensibilité et de réutilisabilité des logiciels actuels. Elle introduit une nouvelle méthode de conception dont la principale caractéristique est de décomposer une application en fonction des données contrairement aux approches antérieures (structurées par exemple) qui donnait la prérogative aux traitements. Ses concepts sont les suivants :

Objet

Les objets sont les entités de base du modèle. Ils ont une description statique, constituée d'attributs valués appelés *variables d'instances*, et une description dynamique constituée par un ensemble d'opérations appelées *méthodes*. Ce concept peut être vu comme une concrétisation de la notion formelle de *type abstrait de donnée* (état + ensemble d'opérations). La caractéristique importante est *l'encapsulation* des données : les objets communiquent uniquement via leurs *interfaces*; l'interface d'un objet est composée de l'ensemble des *signatures* des méthodes susceptibles d'être invoquées par un autre objet (publiques); seules les méthodes sont habilitées à connaître et à modifier directement son état (valeurs des variables d'instances).

La confrontation avec le monde des bases de données [Khos86] ajoute à l'objet la propriété d'*identifiant d'objet* (OID) : tout objet possède un identifiant, unique et invariable, permettant de le référencer, contrairement au modèle valeur (relationnel par exemple) où le partage est réalisé par copie de l'objet. L'OID permet la mise à jour efficace d'objets partagés : toute modification s'effectue directement sur l'objet via son OID et non sur des copies.

Classe

La classe factorise la structure et le comportement commun d'une collection d'objets. Elle sert de modèle ou de moule lors de la création d'un objet : elle contient les déclarations des variables d'instance, valuées lors de la création d'un objet, et les méthodes définissant le

comportement des objets. Dans certains modèles, elle est considérée comme une *extension*, ensemble de tous ses objets. Les caractérisques d'une classe rentrent dans la spécificité des modèles étudiés.

L'héritage

Les concepts de classe et d'objet favorisent l'objectif de modularité. Les objectifs d'extensibilité et de réutilisabilité du logiciel nécessitent le partage d'informations entre les classes. L'approche OO introduit donc la *relation d'héritage* entre les classes. Cette relation permet la factorisation des aspects structurel et comportemental d'une classe : les connaissances les plus générales, définies dans une classe (*super-classe*), sont partagées et spécialisées par les classes (*sous-classes*) héritant de cette dernière. La spécialisation d'une classe est réalisée par enrichissement ou par substitution : dans le premier cas, la sous-classe possède de nouvelles variables d'instances ou des méthodes permettant de décrire un nouveau sous-ensemble d'objets; dans le deuxième cas, seules les corps des méthodes sont redéfinies, l'interface et la structure de la sous-classe restant semblables à celles de sa super-classe. L'héritage permet le *polymorphisme d'objets* : une variable déclarée comme instance d'une classe peut référencer un objet de sa classe ou de toute sous-classe de sa classe. L'héritage peut être multiple si une classe hérite de plusieurs classes ou simple dans le cas contraire. La nature de l'héritage est partie intégrante du modèle.

L'envoi de message

L'envoi de *message* est le protocole de communication entre les objets. Un message envoyé à un objet (ou instance de classe), appelé receveur, permet d'activer une méthode de son interface : il comprend le receveur, le sélecteur (nom) de la méthode à activer et ses arguments. L'interface d'un objet définissant sa visibilité externe, l'envoi de message est l'unique moyen de communication entre objets. Le choix de la méthode à exécuter s'effectue suivant le type du receveur. En raison du polymorphisme d'objets, ce type peut être le type déclaré (statique) ou le type effectif à l'exécution (dynamique) : la méthode activée est celle de la classe réelle de l'objet. La plupart des langages OO opte pour le typage dynamique. Pour des raisons d'efficacité, certains langages préfèrent le typage statique : seule la méthode de la classe déclarée est activée; les redéfinitions de la méthode dans sa sous-hiérarchie sont ignorées.

2.3. La persistance

La *persistance* est la capacité d'un objet à exister indépendamment de l'exécution du programme l'ayant créé ou manipulé, contrairement aux données *temporaires* dépendantes de la durée de vie de leur programme créateur. Les langages persistants offerts par les SGBDOO gèrent les données temporaires et persistantes en s'appuyant sur un modèle de persistance.

Le modèle

Le modèle de persistance définit un ensemble de règles d'attribution de la persistance et de manipulation des objets persistants. La comparaison des différents modèles est basée sur le respect des principes *d'orthogonalité,* de *propagation* et de *transparence.*

Deux niveaux d'orthogonalité de la persistance sont isolés : par rapport aux types[5] ou par rapport à la création d'instance. La persistance est dite orthogonale au système de typage si toute donnée peut persister indépendamment de son type. Par exemple, un Livre peut persister sans la contrainte d'appartenance à une classe spécifique "LivrePersistant". La persistance est dite orthogonale à la création d'instance si la définition et la méthode d'allocation sont les mêmes pour les objets temporaires ou persistants. Par exemple, la déclaration du Livre persistant "L'Amant" n'est pas précédé du qualificatif "persistant".

La propagation de la persistance peut se faire par héritage ou par référence. Lorsque la persistance est propagée par héritage, toute sous-classe d'une classe spécifique gérant la persistante est automatiquement persistante. La persistance par référence permet de rendre persistant un objet ainsi que tous les objets qu'il référence : cet objet est alors appelé *racine de persistance.*

Enfin, le modèle de persistance respecte la transparence comportementale si la manipulation des données temporaires et persistantes est uniforme : aucune distinction n'est faite lors de l'accès ou de la mise à jour des instances ainsi que de leurs variables d'instance. Ceci permet d'écrire une méthode sur une classe dont certaines données sont persistantes et d'autres temporaires.

Les langages persistants

Les langages persistants se distinguent par leur mode d'utilisation et leur stratégie du traitement de la persistance. Le mode d'utilisation est compilé ou interprété. Il est à remarquer que ce critère dépend essentiellement de l'environnement d'accueil: par exemple, Orion, tout comme son environnement d'accueil Lisp, est interprété tandis qu'Exodus, s'appuyant sur C++, est compilé. De plus, les langages persistants des SGBDOO se subdivisent en deux grandes familles selon leur traitement de la persistance. La première offre une librairie de classes comprenant une classe de base; toute classe susceptible de posséder des instances persistantes doit hériter de la classe de base; cette classe possède des méthodes pour la création, la destruction et la référence aux objets persistants; l'implantation de telles classes nécéssite de fournir les méthodes étendant celles de la classe de base. La deuxième propose des adjonctions minimales de mots clefs dans la syntaxe du langage de programmation cible. Leurs modèles de persistance ne subissent aucune contrainte pour respecter les trois principes d'orthogonalité,

[5] type est pris au sens générique du terme, englobant la notion de classe

de propagation et de transparence, contrairement à ceux de la première famille où les principes d'orthogonalité au type et de transparence comportementale sont intrinsèquement violés.

3. EXODUS

Exodus est un système OO développé à l'université de Wisconsin (Madison). Il se démarque des autres systèmes par une approche boîte à outils qui permet de construire un système OO ou d'étendre un SGBD. Il s'adresse plus à des développeurs de systèmes qu'à des programmeurs d'applications. Les aspects novateurs d'Exodus sont : un support de stockage original pour les objets longs basé sur une variation des arbres B+; une des premières extensions du langage C++, le langage E offrant la transparence des transferts des objets entre la mémoire et le gérant d'objets; le modèle de données EXTRA et le langage de requêtes EXCESS issu d'une synthèse des modèles antérieurs de données. Dans la suite de cette section, nous présentons l'architecture fonctionnelle, le modèle C++, le modèle de persistance, le langage E et le compilateur associé.

3.1. L'architecture fonctionnelle

La machine langage (cf. figure 2) du système Exodus comporte un analyseur qui traduit en langage E les requêtes adressées à la base de données. En sus de la persistance, le langage E ajoute au langage C++ les notion de classes génériques et d'itérateurs : les classes génériques sont utiles dans la définition de collections telles que les ensembles ou les structures indexées; les itérateurs permettent d'énumérer uniformément les éléments d'une collection et d'exécuter un bloc de code. Le compilateur du langage E est une extension du préprocesseur C++ d'ATT, (version 2.0). La gestion de la persistance lui incombant, il insère des appels de lecture et de mise à jour d'objets dans la génération du code C; ces appels sont destinés au gérant d'objets. La précompilation des fichiers de définition des classes et types (fichiers .h) produit en mémoire le dictionnaire propre au schéma.

Le gérant d'objets offre une interface uniforme de stockage d'objets, indépendante de leur taille : les objets sont considérés comme une séquence d'octets non interprétés; tout objet persistant est identifié par son OID, implémenté par son adresse physique. La gestion de la persistance à travers le langage E repose essentiellement sur trois opérations de base, *readObject* , *writeObject* et *releaseObject*. La première permet d'établir une référence mémoire sur une partie d'un objet connaissant son OID : le gérant d'objets alloue une zone mémoire, copie la partie de l'objet dans cette zone et retourne l'adresse d'un descripteur contenant l'adresse de cette zone. Le descripteur est donc une poignée sur la donnée laissant le sytème libre de la déplacer en mémoire. Cependant, la donnée est épinglée jusqu'à l'envoi de l'opération *releaseObject* : la donnée peut alors être déplacée ou mise à jour sur le disque et

le descripteur est libéré. L'opération d'écriture *writeObject* est intégrée dans le système transactionnel du gérant d'objets.

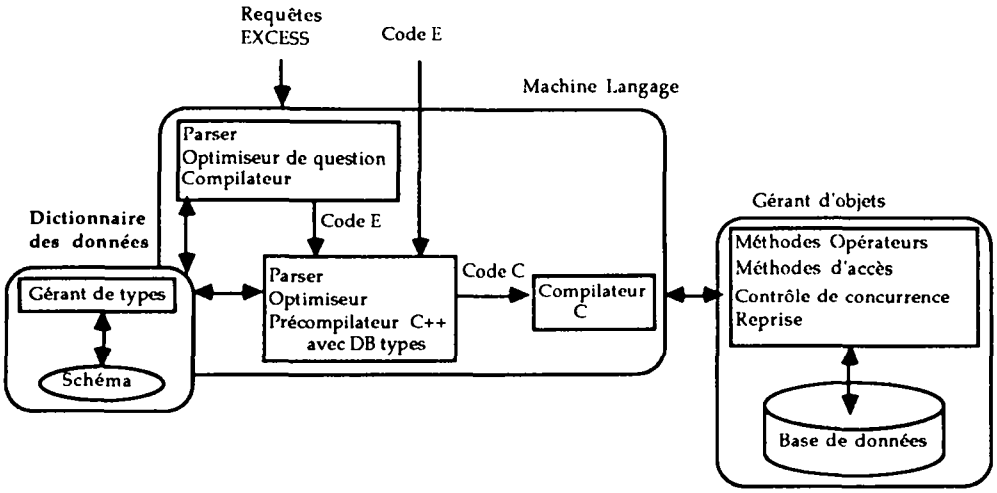

Figure 2 : Architecture Fonctionnelle d'Exodus

3.2. Le modèle de données

Le modèle de données du langage E est le modèle C++. Le concept essentiel défini dans C++ est la classe qui est une généralisation de la structure C. Sa définition inclut la représentation physique des instances et les signatures des opérations. Il n'existe pas de super-classe unique détenant les fonctions de création et d'initialisation; des méthodes spécifiques, appelées *constructeurs* et définies par l'utilisateur permettent de créer et d'initialiser les instances. L'héritage proposé est multiple. La nature statique du typage amène une résolution des message à la compilation; néanmoins, la liaison dynamique peut s'effectuer sur des méthodes dites *virtuelles*. En outre, C++ introduit des règles spécifiques de portée sur les classes et les instances. Les instances de classes différentes communiquent via leurs interfaces tandis que les instances d'une même classe ne peuvent rien se cacher : le domaine protégé est donc la classe et non l'instance. Une classe ou ses propriétés[6] peuvent être *privées* ou *publiques*. Une variable d'instance d'une classe publique peut être rendue privée : elle est alors accessible si une méthode d'accès explicite est définie. L'héritage de la qualité publique ou privée d'une variable d'instance peut être redéfinie au niveau des sous-classes. Une classe déclarée *amie* d'une autre classe A donne à ses instances l'accès aux propriétés privées de A. Ce qualificatif peut être plus sélectif puisqu'il peut s'appliquer au niveau des méthodes ayant besoin d'accéder aux variables d'instance privées de A. Cette caractéristique est alors définie dans la déclaration de la classe A. C++ propose également les concepts de surcharge des opérateurs et de variable de classe par le biais de variables d'instance *statiques*.

[6] Les variables d'instance et les méthodes d'une classe constituent ses propriétés.

3.3. La persistance

Dans le langage E, la définition de la persistance n'est pas purement orthogonale à celle du type. En effet, tout constructeur de type C++ susceptible d'avoir une qualité de persistance est différencié syntaxiquement de son homologue en étant préfixé par *db*. Les types prédéfinis sont donc : les types de base dbint, dbshort, dblong, dbdouble, dbchar et dbvoid; les constructeurs dbclass, dbstruct et dbunion; les pointeurs et les vecteurs sur des objets de db types. Les variables d'instance d'une dbclass ainsi que les attributs des dbstruct et dbunion doivent être de type persistant (db type). Par contre, il n'y a aucune restriction sur les arguments d'une méthode d'une dbclass.

L'appartenance à un db type est une condition de persistance nécessaire mais non suffisante. En effet, un objet doit, de plus, être alloué physiquement sur disque. Le langage E propose une allocation statique de la persistance en précédant la déclaration d'un objet du mot clef *persistent*. La persistance n'est donc pas orthogonale à la création de l'objet. Le langage E propose également une allocation dynamique de la persistance en offrant la dbclass *collection* : tout objet de db type inseré dans une collection persistante est de facto persistant. Sa durée de vie dépend alors d'une destruction implicite lors de la suppression de sa collection ou explicite en appelant une opération de destruction sur l'objet lui-même. Enfin, la manipulation des objets persistants est transparente à l'utilisateur : une fois définie l'identité de persistance sur un objet, c'est le compilateur qui crée les appels au gérant d'objets lors de la transformation du code E en C.

3.4. Le langage E

Le modèle de persistance étant défini, nous montrons par un exemple les extensions apportées au langage C++. La figure 3 présente la définition de la dbclass Livre, de la variable persistante Gallimard et de la variable externe LAmant. Les variables d'instance de la dbclass Livre sont toutes des référence à des dbclass ou des db types. Le programme principal définit Gallimard comme éditeur du livre LAmant. La méthode setEditeur sur Livre prend donc en argument une référence sur un éditeur. Les seuls ajouts syntaxiques sont le préfixage de class par db et l'indication de persistance lors de la déclaration de la variable persistante Gallimard, l'allocation s'effectuant par son constructeur Editeur.

```
/* fichier Livre.h */

        dbclass Livre : public Publication {

        /*   Définition des variables d'instances de Livre   */
         private :
                  PagePreface     *pagePreface;
                  PageTitre       *pageTitre;
                  dbchar  *prixLiteraire;
                  Editeur *editeur;
        public :
                  dbshort         prix;

        /*    Constructeur Livre  */
                  Livre (Auteur *unAuteur, Editeur *unEditeur);

        /*      Méthode  d'affectation de la variable d'instance editeur        */
                  void      setEditeur (Editeur *unEditeur);

        protected : ...
}


/* fichier source */
        #include livre.h

        extern Livre LAmant;
        persistent Editeur Gallimard;

        /* Implantation de la méthode setEditeur de Livre          */
        void Livre::setEditeur( Editeur *unEditeur)
        {
        editeur = unEditeur;
        }

        /* programme principal  */
        main () { ...
        LAmant.setEditeur( Gallimard);
        ...}
```

Figure 3 : Définition de la dbclass Livre et Manipulation d'Instances Persistantes

3.5. Le compilateur du langage E

Le compilateur du langage E est une extension du compilateur C++ d'AT&T (version 2.0). Sa fonction principale est de traduire un fichier source E en produisant un fichier C en sortie (cf figure 4). L'analyseur produit un arbre syntaxique C++ en différenciant, par des étiquettes, les noeuds de l'arbre associés à la persistance. La phase de transformation dérive un arbre syntaxique C de l'arbre syntaxique C++. Elle comprend une phase relative aux constructions typiquement C++ suivie d'une autre relative aux constructions liées aux db types et à la persistance.

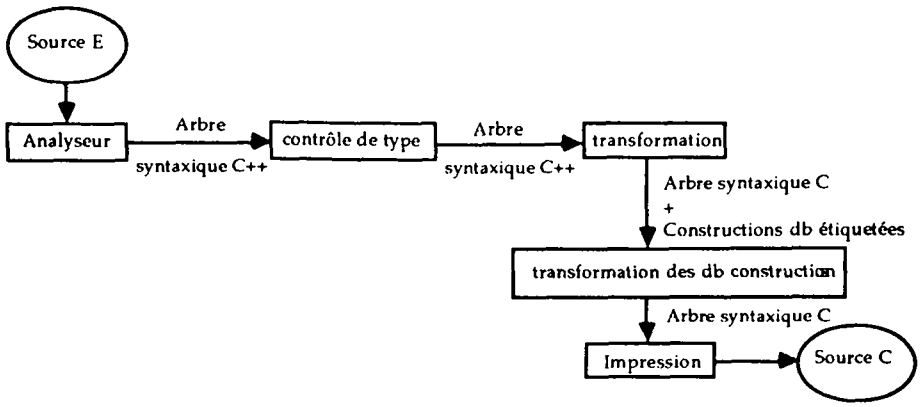

Figure 4 : Architecture du Compilateur E

Notion de DBREF

La distinction fondamentale entre les db types et les types C++ dans E est liée à la faculté des premiers de représenter ou référencer des objets persistants. Un objet persistant est représenté par un pointeur spécifique appelé DBREF. Une DBREF est une structure C++, composées de l'OID de l'objet et d'un déplacement permettant d'accéder à une partie de l'objet. Cependant, il est souvent impossible de déterminer à la compilation si une expression référence un objet temporaire ou persistant : il en est ainsi pour les arguments d'une fonction, les variables externes et les pointeurs sur les db types. Pour pouvoir traiter uniformément ces expressions, les DBREF permettent également de référencer une expression temporaire : la DBREF contient alors un OID spécifique indiquant un traitement en mémoire et le déplacement, l'adresse mémoire.

Définition de classes

En s'appuyant sur l'exemple précédent nous allons suivre les transformations des constructions liées à la persistance. Dans le fichier de définition de la classe Livre (cf figure 5), seules les données structurelles sont pertinentes. La première phase de transformation traduit la classe en structure C, elle copie les variables d'instance héritées des super-classes. La deuxième phase insère la définition de la structure DBREF et remplace tout pointeur sur un db type ou dbclass par une DBREF : ainsi, la variable d'instance auteur, de type pointeur sur la dbclass Auteur, est transformée en attribut de type DBREF. En revanche, un db type prédéfini est directement retraduit en C par son homologue : ainsi la variable d'instance prix de type dbshort devient un attribut de type short.

```
struct DBREF { /* Insertion de la structure DBREF   */
                    OID     oid;
                    int     offset;
             }
struct Livre {      /* Copie des définitions des variables d'instance héritée    */
                    struct DBREF E_titre;
                    struct DBREF E_Auteur;
                    etc...
                    /* Transformation des variables d'instance propres */
                    struct DBREF E_pageTitre;
                    struct DBREF E_pagePreface;
                    struct DBREF E_prixLitteraire;
                    struct DBREF E_editeur;
                    short        prix;
                    }
```

Figure 5 : Transformation de la Définition de la dbclass Livre

Déclaration de variables

Les déclarations de variables de db type sont transformées en fonction de la persistance de l'objet ou de sa portée (cf figure 6). Ainsi, la déclaration d'une variable persistante provoque un appel au gérant d'objets afin de créer physiquement la donnée. L'OID du nouvel objet est introduit sous la forme d'une structure DBREF, appelée *compagnon (companion)* dont le déplacement est initialisé à 0 : E_Gallimard est le compagnon de l'objet persistant Gallimard. Par ailleurs, une variable de portée externe de db type pouvant être temporaire ou persistante, sa déclaration est transformée en une référence externe d'un compagnon: la référence externe sur le Livre LAmant devient une référence externe sur le compagnon E_LAmant de type DBREF.

```
#include _livre.h

/* transformation d'une variable externe                 */
extern                                                    struct DBREF E_LAmant;

/* transformation d'une variable persistente             */
struct DBREF E_Gallimard={ /* Initialisation de la création de l'oid    */ };
```

Figure 6 : Transformation des Aspects Persistants d'un Source C++

Méthodes et envoi de messages

Lors de la première phase de transformation (cf figure 7), les méthodes et constructeurs d'une dbclass deviennent des fonctions C dont le premier argument est un pointeur sur le receveur du message, identifié par la variable _this : dans l'exemple, _this est un pointeur sur la dbclass _Livre. Dans le corps de la méthode, tout accès à une variable d'instance se traduit par un accès à un attribut de _this (par exemple _this->editeur). L'envoi de message sur un objet se traduit quant à lui par un appel à la fonction dont le sélecteur est le nom interne de la méthode avec, comme premier argument, le récepteur du message.

```
void    _Livre_setEditeur(_this, _unEditeur)
        Livre *_this;
        Editeur *_unEditeur;
        {
        _this->editeur = _unEditeur;
        }

main() { ...
        _livre_setEditeur (LAmant, Gallimard);
        ..}
```

Figure 7 : Transformation C++ --> C de la Méthode Livre::setEditeur et de l'Envoi de Message Associé

La deuxième phase de transformation traite les arguments de la signature en remplaçant tout pointeur sur un dbtype en un compagnon de type DBREF : E_this est ainsi lié au compagnon de la variable externe LAmant tandis que E_unEditeur est lié au compagnon de la variable persistante Gallimard.

Les règles de transformation des expressions et instructions suivent une approche récursivement descendante de l'arbre abstrait C en s'attachant aux noeuds étiquetés. Les appels récursifs de réduction considèrent la nature demandée (adresse ou valeur) du noeud fils : l'adresse du noeud retourne une DBREF tandis que la valeur du noeud insère un appel de lecture vers le gérant d'objets pour obtenir la valeur désirée à l'exécution. Par exemple, l'instruction d'affectation(cf figure 8) demande la valeur de son fils gauche et de son fils droit : $(\_this->editeur)_V$ = $(\_unEditeur)_V$. La descente sur le fils gauche demande la valeur du pointeur $(\_this\ )_V$ qui est son compagnon, E_this.

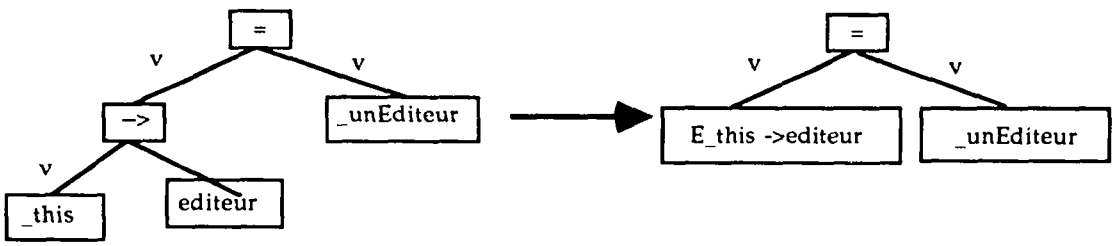

Figure 8 : Arbre Syntaxique de l'Instruction d'Affectation: _this->editeur = _UnEditeur

L'expression gauche (cf Figure 9) est transformée en un appel en lecture au gérant d'objets de la variable d'instance Editeur du compagnon E_this. L'expression droite est transformée en un appel en lecture du compagnon E_unEditeur. Ces deux évaluation résultent en l'obtention des deux références sur les descripteurs (_tmp00, _tmpEditeur) des objets concernés. L'assignation insère alors une demande en écriture en se servant de ces descripteurs. En fin de traitement, les descripteurs ne sont plus nécessaires et donc l'instruction ReleaseObjetc les libère.

```
void E_Livre_setEditeur(E_this,  E unEditeur)

  DBREF E_this;
  DBREF E_unEditeur;

  {
/* Variables temporaires représentant les descripteurs */
  Editeur *_tmpEditeur,  *_tmp00;

/* Transformation de _Livre->editeur :        lecture partielle de l'objet attaché à E_Livre        */
  ReadObject (E_this , GetOffset(Livre, E_editeur), sizeof(Editeur),  &_tmp00);

/* Transformation de _unEditeur :    lecture de l'objet entier*/
  ReadObject (E_unEditeur, 0, sizeof(Editeur), &_tmpEditeur);

/* Modification de  la variable d'instance editeur de _Livre     */
  WriteObject (_tmp00, GetOffset(Livre, E_editeur), sizeof(Editeur), _tmpEditeur);

/* Libération dans le cache des descripteurs            */
  ReleaseObject (_tmp00);
  ReleaseObject (_tmpEditeur);
}

main() { ...
          E_livre_setEditeur (E_Lamant, E_ Gallimard);
          ...}
```

Figure 9 : Transformation de la Fonction _Livre_setEditeur

Constructeurs et fonctions virtuelles

L'intégration de la persistance dans le langage C++ pose quelques problèmes tels que l'utilisation des constructeurs pour les objets persistants et le maintient du lien dynamique pour les fonctions virtuelles. Le traitement de l'initialisation des variables globales consiste à créer, pour chaque fichier, une fonction rassemblant tous les appels aux constructeurs. Au début du lancement d'un programme, toutes les fonctions d'initialisation associées aux fichiers liés sont appelées. La première version du langage E étend cette politique aux objets persistants. Cependant, la création ne pouvant avoir lieu qu'une seule fois, une variable booléenne persistante BOOL est maintenue pour chaque fichier : son rôle est d'indiquer si l'initialisation des variables persistantes globales a déjà eu lieu. Chaque exécution teste BOOL, entraînant donc le coût d'un accès disque par fichier. Par exemple, lors de la première exécution du programme principal, l'initialisation de l'objet Gallimard appelle le constructeur Editeur car BOOL est positionnée à FAUX. Une autre exécution du programme ne lance plus l'initialisation de l'objet Gallimard car BOOL est alors positionnée à VRAI.

Les méthodes virtuelles sont des méthodes permettant de choisir à l'exécution la méthode appropriée à un message suivant le type réel de l'objet. Le compilateur C++ utilise une table d'indirection pour chaque classe définissant des méthodes virtuelles et tout objet de cette classe contient une référence à cette table. Ce pointeur étant alloué à chaque exécution du programme, il n'est pas possible de le mémoriser. Le compilateur E substitue un indicateur de type à ce pointeur mémoire : à toute dbclass possédant des fonctions virtuelles est associée un

indicateur unique de type; tout objet possède cet indicateur. Le compilateur introduit une table de hachage globale temporaire pour effectuer la correspondance entre l'indicateur de type et l'adresse mémoire de la table. A l'exécution, l'indicateur de type mémorisé dans l'objet permet d'atteindre la référence de la table virtuelle associée à la classe.

3.6. Conclusion

Par maints aspects, Exodus se distingue des autres systèmes dans son introduction de la persistance en C++. Ainsi, la définition de la persistance n'est ni orthogonale au type (notion de dbtype), ni orthogonale à la création d'objets (qualification par *persistent*). Par contre , le langage E offre la transparence comportementale : les autres caractéristiques de C++ sont respectées telles que la manipulation des pointeurs, des constructeurs et des fonctions virtuelles. L'approche de compilation suivie est tout à fait originale. En effet, le compilateur est une extension du compilateur C++, il produit donc directement du code C en créant des appels au gérant d'objets pour les accès aux objets peristants. C'est lui qui traite les références aux objets persistants ou temporaires de db type en introduisant les DBREF dans le code. Son approche des accès disques est également très spécifique : le gérant d'objets est considéré comme une machine "load/store"; la compilation contrôle les ordres de lecture, d'écriture et de gestion de parties d'objets dans la zone mémoire du gérant d'objets (allocation, épinglage et libération). Cet accès partiel à l'objet améliore ainsi la gestion des objets longs et de la concurrence.

Cependant, cette approche présente l'inconvénient de créer trop d'appels au gérant d'objets : lors de l'itération sur une liste, chaque passage à un de ses éléments provoque au moins un appel d'accès en lecture suivie d'un appel de libération du descripteur. La première solution est d'offrir un optimiseur capable de détecter les parties d'un objet à fusionner afin de réduire les appels au gérant d'objets. Une autre solution est d'offrir une machine virtuelle au dessus du gérant d'objets capable de maintenir un contexte global durant toute l'exécution d'un programme : les informations maintenues sur les objets persistants permettent de filtrer les appels du compilateur au gérant d'objets.

La principale faiblesse du système Exodus est d'offrir la persistance des objets sans la persistance du schéma de type. En effet, la connaissance des db types et dbclass provient de l'inclusion des fichiers déclaratifs "....h", conduisant à des incohérences potentielles entre les définitions et les implantations : un objet peut ne plus correspondre avec la définition de son type. La persistance du schéma est indispensable pour permettre l'évolution de schéma de types et à la cohérence des instances.

4. ONTOS

Successeur de Vbase, Ontos[7] est un SGBDOO destiné à des applications de CAO, de systèmes experts, de bureautique et de génie logiciel. Ontologic le commercialise depuis 1989. Une librairie de classes prédéfinies étend les fonctionnalités du langage C++. Il se distingue des approches précédentes par : la gestion du schéma sur disque dans un environnement C++ permettant l'interrogation et la mise à jour, à l'exécution, de ses informations structurelles et dynamiques; la coexistence de deux sortes de pointeurs pour le traitement de la persistance; le regroupement physique des instances selon une relation de composition; différentes granularités de persitance; une classe prédéfinie permettant la gestion des exceptions.

4.1. L'architecture fonctionnelle

L'architecture d'Ontos (cf figure 10) suit partiellement le modèle SGBDOO réparti : un même processus (machine langage) peut accéder un ou plusieurs serveurs (gérant d'objets) sur différentes machines. La machine langage est composée de deux utilitaires *Classify* et *Cplus* destinés à la gestion dynamique du schéma : Classify s'intéresse à la définition des classes et Cplus à leur comportement. Le dictionnaire d'objets comprend le schéma de toutes les bases de données ainsi que les informations de distribution sur les serveurs, appelé repository.

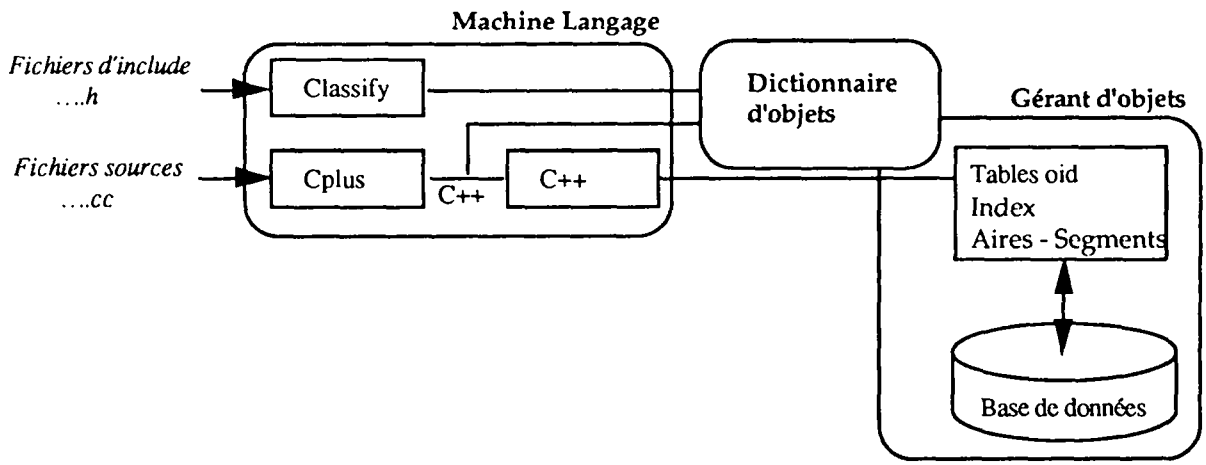

Figure 10 : Architecture d'Ontos

4.2. Le modèle

Le modèle C++ est enrichi de la notion d'identité d'objet pour toute instance persistante ainsi que d'une librairie de classes (cf figure 11). Les variables d'instance sont stockées directement dans l'objet ou représentées par des OID. La librairie de classes fournit la gestion du traitement d'exception, de la persistance, du regroupement d'instances et du

[7] L'étude porte sur la version 1.5

schéma. Les principales classes permettant ses fonctionnalités sont : CleanupObj, Entity, Object, Aggregate et Type. La classe *CleanupObj*, racine de la librairie, initialise les mécanismes de traitement d'exception et propose une méthode pour la désallocation mémoire qui remplaçe le destructeur C++. La classe *Entity* référence uniformément les variables des types primitifs C++[8] et les instances de classe : toute instance de classe dérivée d'Entity a un pointeur sur sa classe. Ces deux classes s'adressent aussi bien aux classes persistantes que non persistantes. La classe Object gère la persistance (cf II.4.2). La classe *Type* est une représentation de la définition des classes C++ : ses instances sont crées soit par programme, soit par l'utilitaire *Classify* . Pour chaque classe, elle spécifie son nom, sa super-classe, ses variables d'instance, ses méthodes et leurs arguments. La méthode Instantiate crée une instance de type. L'utilitaire Classify, activé à la compilation des classes persistantes, construit automatiquement les instances de Type associées. Ses sous-classes telles que *PropertyType* et *Procedure*, sont utilisées à l'exécution pour donner respectivement les définitions des variables d'instances, des méthodes et des arguments d'une méthode. La classe *Aggregate* fournit le comportement de base de regroupement d'instances. Ses classes dérivées sont différenciées suivant le mode d'accès : les classes *Array* et *Dictionnary* ont un accès aux éléments par clef de type respectif int et char; la classe *List* représente les séquences ordonnées; la classe *Set* est une collection sans doubles. De plus, la classe *Iterator* initialise le fonctionnement des itérateurs tels que Aggregatelterator, les itérateurs du schéma et d'autres itérateurs : ils sont construits pour une instance particulière et retournent séquentiellement ses valeurs telles que les instances d'un regroupement ou les informations associées au schéma. Ces classes itératives sont non persistantes : leurs instances sont utilisées lorsque l'agrégat est en mémoire. La méthode Reset permet de réinitialiser l'itérateur pour opérer sur la même instance ou sur une autre.

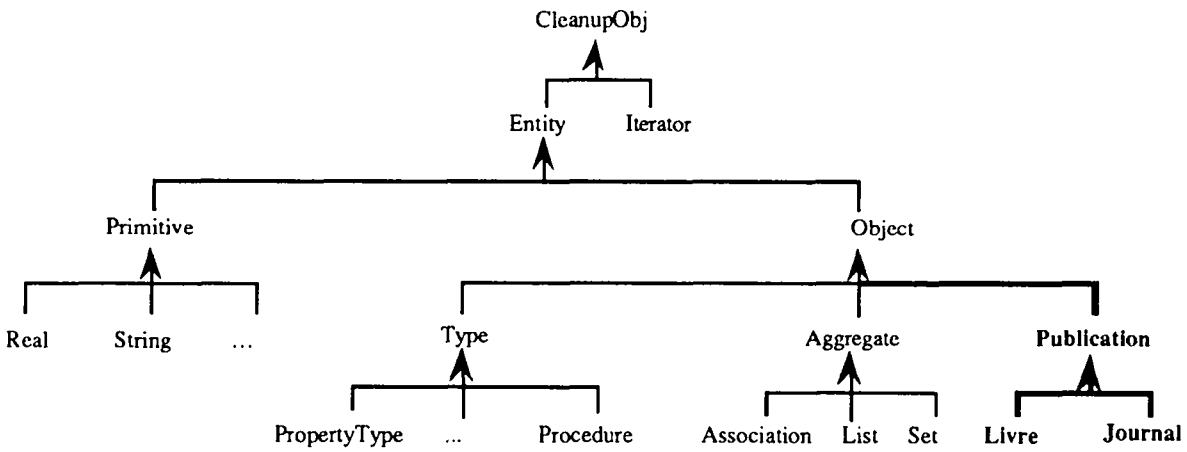

Figure 11 : Hiérarchie des Classes Ontos

[8] Les classes associées sont Integer, Pointer, Real et String

4.3. La persistance

Les classes persistantes forment un arbre dont la racine est la classe Object : elle définit les mécanismes de base de gestion des instances persistantes en offrant les méthodes de création, de lecture, d'allocation mémoire, de stockage et de destruction. La persistance n'est pas orthogonale au type mais à la création des instances : la qualité de persistance est octroyée exclusivement aux instances des classes persistantes, par l'envoi explicite d'un message. Les agrégats List, Set, Dictionnary ou Array constituent un granule de regroupement et de persistance : leurs éléments sont exclusivement des instances de classes persistantes; des méthodes liées à la persistance permettent de les manipuler globalement. De plus, Ontos propose le placement explicite des instances, à leur création, suivant deux critères : le groupement et l'unité de groupement.

Les OID ne sont jamais montés en mémoire; aussi toute manipulation d'instance persistante est d'une part précédée d'une opération de lecture et d'autre part suivie d'une sauvegarde sur disque. Ces opérations sont appelées respectivement *activation* et *désactivation*. L'activation d'une instance implique l'allocation d'une zone mémoire, le chargement de sa valeur et la transformation de son OID en adresse mémoire appelée *référence* . La désactivation, opération inverse, retraduit les adresses mémoires des instances en OID, désalloue optionnellement la zone mémoire et recopie l'instance dans la base en fin de transaction (commit).

Deux formes de références, spécifiées à la définition des classes, sont considérées en fonction du mode de traitement de la persistance des objets qu'ils référencent : *directe* ou *transparente*. Les références directes sont des pointeurs mémoire C++ pour lesquels les opérations d'activation/désactivation doivent être explicitement effectuées. La manipulation de références directes actives est transparente. Une référence directe sur une instance non active a la valeur prédéfinie *INACTIVE*. Toute demande d'accès sur des références inactives provoque une erreur système. La destruction d'une instance nécessite la gestion explicite de ses références directes. Les références directes sont associées aux OID dans une table système. Lors de l'activation d'une instance, les valeurs de toutes ses références directes sont remplacées automatiquement par sa nouvelle adresse mémoire. Lors de sa désactivation, les références directes reprennent la valeur INACTIVE. Pour pallier à cette gestion fastidieuse, dangereuse et coûteuse, les références transparentes, instances de la classe prédéfinie *TRef*, sont introduites. Les opérations d'activation/désactivation des références transparentes sont gérées automatiquement lors d'une demande d'accès. Leur manipulation s'effectue par des méthodes prédéfinies. Une seule réference transparente est associée à chaque instance : une table de hachage, indexée sur les OID, permet l'accès à ces références.

4.4. Le langage

La définition d'une classe persistante (cf. figure 12) doit satisfaire les conditions suivantes : dériver de la classe Object; comporter un constructeur d'activation et au moins un constructeur d'instance; écarter le mécanisme des destructeurs C++ en utilisant la méthode Destroy; comporter des méthodes de stockage et de destruction dans la base appropriée. Ainsi dans notre exemple, la classe Livre possède quatres variables d'instance privées et une publique : pagePréface et éditeur sont des références transparentes; pageTitre est une référence directe; prixLittéraire est une chaîne de caractères. Tout constructeur d'activation a un argument unique, pointeur sur une structure APL (liste de paramètres d'activation). Le constructeur d'activation de Livre appelle directement le constructeur de la classe Object sans code supplémentaire.

```
#include "Object.h"
#include "Publication.h"

class Livre : public Publication {
        private :
                TRef            *pagePreface;
                PageTitre       *pageTitre;
                char            *prixLitteraire;
                TRef            *editeur;

        public :
                short           prix;
                // constructeur d'activation
                Livre (APL *uneAPL) : (unAPL) { }

                // constructeur d'instance
                Livre (PagePreface *pagePreface, char * prixLitteraire, Editeur *editeur,
                        Object *where = 0, Clustering howNear = defaultClustering);

                // méthode d'accès aux variables d'instances
                char * prixLitteraire( ) {return prixLitteraire};
                Editeur *editeur( ) {return (Editeur *) editeur->Binding()};
                ...
                // méthode de désallocation mémoire
                Destroy (Boolean abandon);
                // méthode de désactivation
                putObject ( Boolean désallocation = FALSE );
                // méthode de destruction dans la base
                deleteObject ( Boolean désallocation = FALSE );
                ...};
```

Figure 12 : Création de Classe Persistante

La création des instances est réalisée par un constructeur similaire à ceux de C++. Outre son rôle d'allocateur de mémoire et d'initialisateur de variables d'instance, il effectue trois autres fonctions : établir le lien entre l'instance et la définition de sa classe[9] ; appeler le constructeur d'Object en précisant le nom de l'instance et les paramètres de placement; initialiser les références TRef ou directes. Le placement est déterminé par les mots clefs *where* et *howNear* : where désigne le regroupement sur une instance ou une instance d'agrégat;

[9] Instance de la classe Type

howNear identifie l'unité de regroupement qui peut être une aire (plusieurs segments), un segment (unité de transfert) ou une unité par défaut. Dans la figure 13, le constructeur de Livre appelle directement le constructeur d'allocation de la classe Object en passant la valeur nulle pour le nom de l'instance et les arguments where et howNear pour les options placement. La variable globale livreType est définie afin que l'appel à l'instance contenant la définition de la classe Livre puisse être effectué à chaque construction d'instance. Il est nécessaire de l'initialiser au premier appel du constructeur.

```
// pointeur global sur le type Livre
Type * livreType;

Livre::Livre (PagePreface *pagePreface, char * prixLitteraire, Editeur *editeur,
                Object *where , Clustering howNear ) : ((char*) 0, where, howNear);
        {
        // Initialisation de LivreType
        if (!livreType)
                {
                livreType = (Type*)OC_lookup("Livre");
        // établit le lien avec l'instance de Type contenant la définition de la classe Livre
                directType(livreType);
                }

        //initialise les variables d'instance
        pagePreface = pagePreface;
        pageTitre = NULL;
        ...
        prix = 0;}
...
// création d'une instance de livre sans précision de placement
Livre *unLivre(unePage, "Goncourt", unEditeur, 0, 0);
```

Figure 13 : Constructeur de Création d'Instances

Une instance peut être activée par quatre appels différents : OC_get... pour les instances et les agrégats nommés, OC_directActivate pour les références directes et la méthode Binding pour les références transparentes. Dans la figure 14, la référence sur LivreDeLaJungle étant nominale, la fonction *OC_getObject* est appelée. La liste des livres ayant eu le prix Goncourt appelée "LivreGoncourt" est une unité de placement; aussi la fonction *OC_getCluster* active tous les livres de cette liste. La page de titre du LivreDeLaJungle étant une référence directe non active et non nulle, la fonction *OC_directActivateObject* est utilisée. L'éditeur étant une référence transparente, son activation s'effectue par la méthode *Binding.*, encapsulée dans la méthode d'accès Livre::editeur(). De plus, l'assignation entre références directes non actives n'est pas transparente : la fonction *OC_directAssignObject* est nécessaire.

```
// Déclaration de variables utilisées dans cet exemple
Livre * unLivre, autreLivre;
List *LesLivresGoncourt;
pageTitre * une PageTitre;
Editeur *unEditeur;

// Activation par nom
unLivre = (Livre *) OC_getObject("LivreDeLaJungle", 0, 0 );
LesLivresGoncourt = (List *) OC_getCluster("LivreGoncourt", 0, 0 );
```

```
// Activation d'une référence directe
if (unLivre->pageTitre != NULL )
        {
        if ( unLivre->pageTitre == INACTIVE )
                OC_directActivateObject(&unLivre->pageTitre);
        unePageTitre = unLivre->pageTitre ;
        }

// Activation d'une référence transparente
// Rappel de la fonction d'accès sur la variable d'instance editeur spécifiée dans la déclaration de la classe
// Editeur *editeur( ) {return (Editeur *) editeur->Binding()};

unEditeur = unLivre->editeur();

// Assignation entre références directes non actives
OC_directAssignObject ( &(unLivre->pageTitre), &(autreLivre->pageTitre));   ...
```

Figure 14 : Activation d'Instances

La désactivation et la persistance sont définies par les appels suivants : *putObject* et *OC_putObject* pour les instances; *putCluster* et *OC_putCluster* pour les aggrégats. Dans la figure 15, la méthode putObject prend en compte la désactivation de la référence directe sur pageTitre et envoi le message putObject sur sa super-classe.

```
// redéfinition de la méthode putObject sur Livre avec le traitement de la référence directe pageTitre
Livre::putObject(Boolean desallocation = FALSE) {
        if (pageTitre != NULL) pageTitre->putObject();
        Publication::putObject();
        }
        ...
// désactivation d'une instance avec désallocation mémoire
unLivre->putObject(TRUE);
// désactivation par nom sans désallocation
OC_putObject ( "livredelajungle", FALSE );

// desactivation sur un aggrégat sans désallocation
LesLivresGoncourt->putCluster(FALSE);
```

Figure 15 : Méthode de Désallocation

Les mécanismes d'exception d'Ontos ne permettent pas l'utilisation des destructeurs C++ qui sont remplacés par la méthode *Destroy*, de la classe CleanupObj. Lorsqu'une classe doit nettoyer un environnement spécifique, elle peut redéfinir la méthode *Destroy*. Dans la figure 16, la variable d'instance prixLittéraire de Livre étant une chaîne de caractères, elle doit être désallouée explicitement. L'argument *abandon* détermine la cause de la destruction : exception ou autres. La destruction d'instances dans la base de données est effectuée explicitement avec la méthode *deleteObject* de la classe Object : cette méthode peut éventuellement être redéfinie.

```
void Livre::Destroy (Boolean abandon)
        { delete prixLittéraire;
        ObjectDestroy(abandon);
        }

//déstruction dans la base sans désallocation mémoire
unLivre->deleteObject (FALSE) ;
```

Figure 16 : Désallocation en Mémoire et Destruction dans la BD

4.5. Conclusion

La librairie de classes enrichit le modèle C++ des fonctionnalités telles que la gestion de la persistance, de collections d'instances et du traitement d'exception. Le schéma mémorisé dans la base, est interrogé à l'éxécution : la définition de chaque classe persistante est représentée sous forme d'instances de Type et de ses sous-classes. Le traitement de la persistance étend la granularité aux collections d'instances et offre le libre choix du regroupement physique d'instances. La gestion de la mémoire associée à chaque transaction, contrairement à Exodus, nécessite une traduction des OID physiques à l'aide des mécanismes d'activation et de désactivation.

5. ODE

Ode est un système expérimental développé dans les laboratoires d'AT&T depuis 1989. Il s'adresse à toute application désirant un modèle de données intégré à un langage de programmation. Les aspects originaux du système Ode sont : le langage O++, extension de C++, introduisant les concepts de persistance, de versions d'objets, de contraintes d'intégrité et de déclencheurs (ou trigger); l'approche de compilation du code O++; l'interface de son gérant d'objets constituée par une librairie de classes C++.

5.1. L'architecture fonctionnelle

La machine langage du système Ode (cf figure 17) est définie par le compilateur du langage O++. Contrairement à Exodus, ce compilateur se situe en amont du compilateur C++ (version 2.0). La gestion de la persistance lui incombant, il insére des déclarations, des méthodes ainsi que des appels de lecture et de mises à jour d'objets dans la génération de code C++; ces adjonctions sont dictées par l'interface avec le gérant d'objets. Le dictionnaire des définitions de classes est uniquement en mémoire, les définitions de classes étant incluses par les fichiers déclaratifs "....h".

Le gérant d'objets est une librairie de classes C++ génériques[10] considérant la *base d'objets* comme unité de stockage : seule entité nommée par l'utilisateur, la base est instance

[10] classes paramétrées par le type des objets ; elles sont offertes sous le nom de *template* dans la version 3.0 de C++ d'AT&T.

de la classe *ClusterGroup*; elle contient des *Cluster*, extensions des classes de la base. Une classe *ClusterIter* permet de les parcourir séquentiellement. Une application peut avoir accès à plusieurs bases : la création d'un cluster peut référencer sa base. En l'absence de référence, le cluster est stocké dans une base par défaut.

De plus, les objets persistants présentant généralement une copie en mémoire et une sur disque, les actions associées au ClusterGroup sont *l'activation* et son inverse la *synchronisation* : l'activation (méthode *activation*) permet la copie en mémoire d'un objet persistant; la synchronisation (méthode *sync*) consiste à écrire sur disque l'objet persistant modifié. Le gérant d'objets doit donc disposer pour toute classe, des méthodes d'écriture (*writeObj*) et de lecture (*readObj*). L'objet étant transféré dans sa globalité, une méthode définissant sa taille doit être également fournie (*diskSize*). Le système Ode s'appuie actuellement sur un gestionnaire de fichiers UNIX, fournissant juste les services de stockage.

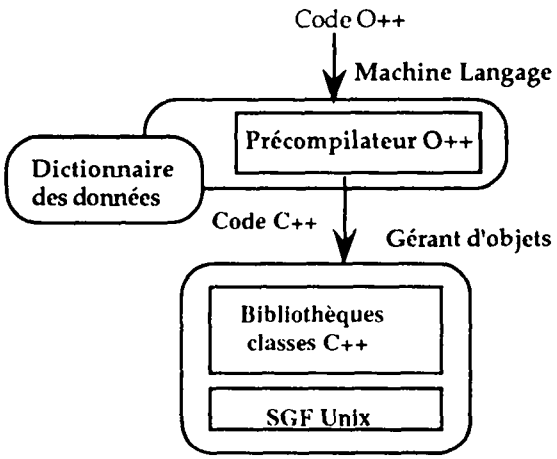

Figure 17 : Architecture Fonctionnelle d'Ode

5.2. Le modèle de données

Le modèle de donnée du langage O++ est le modèle C++. En plus de la persistance, ce modèle introduit les *contraintes d'intégrité* et les *déclencheurs* (triggers), rendant les bases *actives*. Les contraintes d'intégrités, définies lors de la déclaration de la classe, sont des expressions booléennes vérifiées lors de toute mise à jour des instance d'une classe; elles sont héritées et éventuellement redéfinies dans les sous-classes.

Un déclencheur permet l'activation conditionnelle d'un groupe d'instructions : il est donc composé d'un ensemble de conditions évaluées en fin de transaction et d'instructions exécutées comme une transaction séparée; un déclencheur est défini dans la déclaration de la classe et activé sur les instances. O++ propose des déclencheurs instantanés et des déclencheurs perpétuels : les premiers sont désactivés aprés chaque utilisation et nécessitent donc une activation explicite tandis que les seconds sont toujours actifs.

5.3. Persistance

Dans le langage O++, la persistance est orthogonale au type. Les objets temporaires sont alloués en mémoire comme des objets C++ ordinaires, tandis que les objets persistants sont alloués explicitement sur disque. L'identité d'objet est représentée par un pointeur logique sur un objet persistant, appelé par simplification *pointeur persistant*. La qualité de persistance est donc introduite à la déclaration de pointeur par le mot clef *persistent*. Les objets persistants peuvent être des copies d'objets temporaires et vice-versa en utilisant une simple assignation. O++ offre également la transparence comportementale. Comme dans le système Exodus, la manipulation des objets persistants est transparente à l'utilisateur : c'est le précompilateur qui crée les appels au gérant d'objets lors de la transformation du code O++ en C++.

5.4. Le langage O++

Le modèle de persistance étant défini, nous présentons les caractéristiques du langage O++ liées à la persistance. Pour une meilleure comparaison, la figure 18 reprend la définition de la classe Livre, les déclarations de la variable persistante Gallimard et du pointeur LAmant sur le livre persistant. L'instruction *pnew* alloue l'instance de Livre sur disque et assigne son OID au pointeur LAmant. O++ oblige à qualifier de *persistent* tout pointeur sur un objet potentiellement persistant : les variables d'instances pagePreface, pageTitre et editeur, les arguments du constructueur Livre et de la méthode setEditeur. Nous montrons également la copie de l'objet persistant LAmant sur un objet temporaire unLivre ainsi que la transparence de manipulation des variables d'instances au travers de l'assignation du prix du livre LAmant.

La création d'un objet persistant nécessite l'ouverture préalable du cluster dans lequel l'objet doit être créé : ainsi, l'instruction O++ *copen* ouvre les clusters de Livre et d'Editeur. Nous supposons que le cluster d'Auteur est ouvert avant la définition de la variable Duras.

```
/* fichier Livre.h */

    class Livre : public Publication {

    /*  Définition des variables d'instances de Livre  */
    private :
            persistent PagePreface  *pagePreface;
            persistent PageTitre    *pageTitre;
            char                    *prixLiteraire;
            persistent Editeur      *editeur;
    public :
            short   prix;

    /*  Constructeur Livre  */
            Livre (persistent Auteur *unAuteur, persistent  Editeur *unEditeur);

    /*   Méthode d'affectation de la variable d'instance editeur      */
            void    setEditeur (persistent  Editeur *unEditeur);
    }
```

```
/* fichier source */
        #include "livre.h"

        /* programme principal */
        extern persistent Auteur * Duras;

        main () {
        /* ouverture ou création des cluster associés aux classes Editeur et Livre */
        copen(Editeur);
        copen(Livre);

        /* Déclaration des pointeurs sur instances persistentes Gallimard et LAmant */
        persistent Editeur * Gallimard;
        persistent Livre *LAmant;
        Livre * unLivre;
        short prixCourant;
        ...
        /* allocation sur disque */
        LAmant = pnew (Duras, Gallimard);
        ...
        /* copie de l'objet persistant pointé par LAmant vers l'objet temporaire référencé par unLivre */
        *unLivre = *LAmant;
        LAmant->prix = prixCourant;
        ...
        LAmant->setEditeur( Gallimard);
        ...}
```

Figure 18 : Définition de la Classe Livre et Manipulation d'Instances Persistantes

5.5. Le compilateur du langage O++ : Ofront

Le compilateur du langage O++ est constitué d'un précompilateur *Ofront* en amont du compilateur C++. La liaison du code objet avec la bibliothèque du gérant d'objets produit ensuite l'exécutable. Nous nous intéressons à la génération du code effectué par O++ en présentant la gestion des pointeurs sur les objets persistants, l'interface et les appels au gérant d'objets ainsi que les méthodes engendrées automatiquement. En outre, nous évoquons la solution d'O++ au problème des pointeurs sur les tables de méthodes virtuelles.

Les Pointeurs Persistants

Rappelons que tout objet potentiellement persistant doit être référencé par un pointeur persistant contenant les informations liées à la localisation de l'objet référencé : son adresse disque ainsi que sa présence et éventuellement son adresse en mémoire. Dans la librairie du gérant d'objets, les pointeurs persistants sont modélisés par une hiérarchie de classes dont la racine est la classe générique *PersPtr<Classe>*. Ces classes contiennent un pointeur sur le ClusterGroup de l'objet ainsi que son numéro, utilisé pour retrouver à la fois son adresse disque et son adresse mémoire. Elles fournissent la sémantique usuelle des pointeurs C++ : les opérateurs tels que *, [], ->, ++ et -- sont définis. De plus, un pointeur persistant peut être affecté à un pointeur C++ et un pointeur persistant peut être utilisé à la place d'un pointeur C++.

Les pointeurs persistants sont des instances de la classe générique *SafePersPtr<Classe>*. Cette classe garantit la présence en mémoire de l'objet référencé avant tout accès de manière

analogue aux TREFs d'Ontos : un contrôle est effectué lors de la transformation d'un SafePersPtr en pointeur C++ ou d'un déréférençage.

Définition de Classes

Le compilateur O++ modifie les classes utilisateurs en ajoutant une super-classe PersBase, en remplaçant les pointeurs persistants par des instances de SafePersPtr(Classe), en définissant les méthodes requises par le gérant d'objets pour implanter la persistance ainsi que des macros définissant les classes Cluster(Livre) et PersPtr(Livre). L'héritage de PersBase permet de fournir le destructeur des objets persistants. La méthode diskSize calcule la taille de l'objet sur disque utilisée par les méthodes de lecture (readObj) et d'écriture (writeObj).

```
/* déclaration de la classe PersPtr <Livre>                    */
PersPtrDeclare(Livre);

class Livre : public Publication , virtual private PersBase {

/*   Définition des variables d'instance de Livre    */
private :
        SafePersPtr( PagePreface )    pagePreface;
        SafePersPtr( PageTitre )      pageTitre;
        char                          *prixLiteraire;
        SafePersPtr( Editeur ) editeur;
public :
        short    prix;

/*    Constructeur Livre  */
        Livre ( SafePersPtr(Auteur ) unAuteur, SafePersPtr( Editeur ) unEditeur );

/*      Méthode  protégée d'affectation de la variable d'instance editeur       */
        void     setEditeur ( SafePersPtr(Editeur)  unEditeur );

/* méthodes rajoutées */
virtual uint diskSize(void);
static void readObj( ClusterGroup *cgp, uint onbr, uint &doff, Livre *&objp );
virtual void writeObj ( ClusterGroup *cgp, uint onbr, uint &doff ) ;

/* déclaration de la classse Cluster<Livre>          */
ClusterDeclare( Livre ) ;
}
```

Figure 19 : Traitement de la Déclaration de la Classe Livre

Implémentation des méthodes associées à la persistance

La méthode diskSize effectue la somme de la taille de ses super-classes et de ses variables d'instance. Lorsque ces dernières sont de type simple, la fonction sizeof est utilisée. Les pointeurs non persistants, tels que prixLittéraire, ne sont pas stockés et l'espace qu'ils occupent n'est pas pris en compte par diskSize : leur initialisation est donc à la charge du programmeur.

Les méthodes readObj et writeObj utilisent notamment le ClusterGroup (cgp) et un déplacement disque (doff) pour pointer une zone de l'objet sur le disque. Elles sont appelées récursivement sur les variables d'instance. Lorsque ces dernières sont de type simple, des

méthodes de lecture et d'écriture de ClusterGroup sont utilisées avec une interface tampon. Par effet de bord, la méthode readObj met-à-jour la copie mémoire de l'objet. Si celle-ci n'existe pas, readObj commence par la créer en appelant le constructeur de la classe. Notons que cette implémentation résout le problème des pointeurs sur les tables des méthodes virtuelles : le constructeur crée un squelette C++ contenant les pointeurs de l'exécution courante.

```
/* SafePersPtr(PagePreface)::diskSize() donne la taille du pointeur persistant */
uint Livre::diskSize(void)
        {
        return   Publication::diskSize()
                  + SafePersPtr(PagePreface)::diskSize() + SafePersPtr(PageTitre)::diskSize()
                  + SafePersPtr(Editeur)::diskSize() + sizeof(prix) ;
        }

void Livre::readObj (ClusterGroup *cgp, uint onbr, uint&doff, Livre*&objp)
        {
        if (objp ==NULL)
                objp = new Livre;
        SafePersPtr(PagePreface)::readObj ( cgp, onbr,  doff, &objp->pagePreface );
        SafePersPtr(PageTitre)::readObj ( cgp, onbr,  doff, &objp->pageTitre );
        SafePersPtr(Editeur)::readObj ( cgp, onbr, doff, &objp->editeur );
        cgp->readObj ( onbr, doff, &objp->prix, sizeof(obj->prix) );   }

void Livre::writeObj (ClusterGroup *cgp, uint onbr, uint&doff )
        {
        pagePreface.writeObj ( cgp, onbr,  doff );
        pageTitre.writeObj ( cgp, onbr,  doff );
        editeur.writeObj ( cgp, onbr, doff );
        cgp->writeObj ( onbr, doff, &prix, sizeof( prix) );
        }

/* Implémentation des classes PersPtr(Livre) et Cluster(Livre) */
PersPtrImplement(Livre) ;
ClusterImplement(Livre) ;
```

Figure 20 : Implantation des Méthodes diskSize, readObj et writeObj

Traitement du programme principal

L'ouverture des clusters par *copen* est traduite par la création d'instances de clusters pour chaque classe : _clus_Editeur et _clus_Livre. L'instruction d'allocation persistante pnew est transformée en insertion dans le cluster de la classe d'une instance mémoire. La surcharge des opérateurs de déréférençage et de conversion pour les pointeurs persistants permet de laisser inchangé le reste du code.

```
/* programme principal */
extern SafePersPtr ( Auteur ) Duras;

main () {

/* macros Cluster instanciant les classes génériques */
Cluster( Editeur ) _clus_Editeur ("Editeur" ) ;
Cluster( Livre) _clus_Livre ("Livre" ) ;

/* Déclaration des pointeurs sur instances persistantes Gallimard et LAmant */
SafePersPtr(Editeur )  Gallimard;
SafePersPtr(Livre)  LAmant;
Livre * unLivre;
short prixCourant;
...
/* allocation sur disque */
LAmant = _clus_Livre.insObj ( new Livre(Duras, Gallimard) ) ;

...
/* copie de l'objet persistant pointé par LAmant  vers l'objet temporaire référencé par unLivre */
*unLivre = *LAmant;
LAmant->prix = prixCourant;
...
LAmant->setEditeur( Gallimard);
...}
```

Figure 21 : Traitement du Programme Principal

5.6. Conclusion

Dans son traitement de la persistance, Ode se situe entre Exodus et Ontos. En effet, il choisit une introduction syntaxique de la persistance, mais modifie les classes de l'utilisateur suivant l'approche d'Ontos : héritage d'une superclasse commune, pointeurs persistants équivalents aux TREFs, méthodes d'écriture et de lecture pour l'activation et la désactivation. Les ajouts syntaxiques ne portent que sur la création des instances, offrant l'orthogonalité au type et la transparence comportementale. L'obligation de spécifier l'éventualité de la persistance dans la déclaration des variables d'instance d'une classe nuit cependant à la simplicité d'expression de la persistance.

6. O2

O2 est un produit commercialisé depuis 1991 par O2 Technologie. Il fut d'abord un prototype de SGBDOO développé à l'INRIA dès 1986, au sein du GIP Altaïr. O2 est destiné principalement au développement d'applications dans les domaines de la gestion traditionnelle, de la bureautique et de la manipulation de données multimédia. Un point fort d'O2 est d'être un système complet, avec son propre environnement de programmation (O2Tools) et un générateur d'interfaces graphiques (O2Look). Le modèle de données présente la particularité de supporter les objets et les valeurs complexes. Le langage de programmation du système se singularise par la dichotomie existante entre les aspects de définition et de

manipulation des données : les données sont définies à partir d'un ensemble de commandes spécifiques O2 et elles sont manipulées avec le langage O2C, extension de C.

6.1. L'architecture fonctionnelle

Le système O2 opère sur une architecture répartie de type client/serveur. La station cliente contient O2Tools, O2Look et la machine langage. Le gestionnaire de schéma et le gérant d'objets sont répartis entre la station cliente et le serveur. O2Tools (cf figure 22) est un environnement de programmation graphique comportant un navigateur (browser) pour consulter et mettre à jour le schéma, un débogueur (debugger) symbolique pour mettre au point les programmes O2C, et une interface graphique pour utiliser le langage de requêtes. O2Look est un générateur d'interfaces graphiques qui assure une construction rapide et simple d'interfaces conviviales pour les utilisateurs finaux; il est composé d'un ensemble d'éditeurs génériques prédéfinis, opérationnels sur tous les types prédéfinis du système.

La machine langage d'O2 comporte un analyseur de commandes pour la définition du schéma, un compilateur pour le langage O2C et un interpréteur de requêtes. Le compilateur O2C produit du code C, avec des appels au gérant d'objets O2Engine. O2Engine est un serveur de pages composé de trois couches. La couche haute est le gestionnaire de schéma. Il est responsable de la création, recherche, mise-à-jour et destruction des classes, des méthodes et des racines de persistance tout en maintenant la cohérence du schéma. La couche intermédiaire est le gérant d'objets. Il implémente l'envoi de messages, les opérations sur les types structurés, le modèle de persistance par atteignabilité, le glâneur de cellules[11] et le placement des données. La couche basse est constituée par le système de stockage WiSS, développé à l'université du Wisconsin; il fournit les structures de base persistantes, telles que B-tree et les séquences d'octets non-interprétés. Il prend en charge le contrôle de concurrence, les reprises et la gestion du disque.

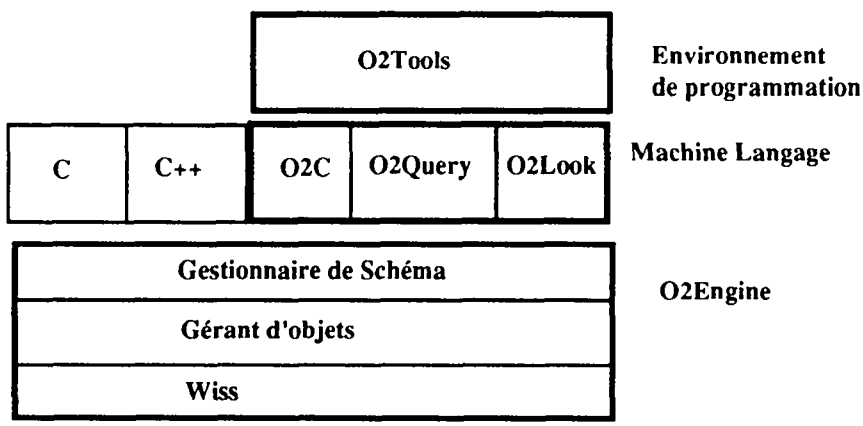

Figure 22 : Architecture Fonctionnelle d'O_2

[11] garbage collector en anglais

6.2. Le modèle de données

Le modèle du système O2 distingue les notions de valeurs et d'objets. Une valeur est une instance de type. Elle est manipulable à l'aide d'un ensemble figé d'opérateurs associé à son type; contrairement aux méthodes, ces opérateurs ne sont pas redéfinissable. Les types sont atomiques ou structurés. Les principaux types atomiques du système sont les entiers, les réels, les chaînes de caractères et les booléens. Un type structuré est construit récursivement à l'aide des trois constructeurs prédéfinis n-uplet, ensemble et liste. Les types structurés sont anonymes et donc l'équivalence entre types est purement structurelle.

Un objet est défini par un couple <identité, valeur> et est instance d'une classe. Une classe est définie par son type et ses méthodes. La structure et le comportement d'un objet sont définis avec une partie publique et une partie privée, contrairement aux valeurs dont la structure est publique. Les règles de visibilité sont analogues à celles définies dans le modèle C++. Une classe est créée à l'aide d'une commande spécifique du langage de définition. La seule classe prédéfinie est la classe Object, racine de la hiérarchie de classes; elle contient les méthodes standard d'égalité, de copie, et d'édition d'un objet. La création d'une instance particulière est réalisée par une instruction spécifique du langage O2C et non par un envoi de message à sa classe.

L'héritage proposé sur les classes est multiple, avec une résolution par renommage ou explicite des conflits. Il comporte un héritage structurel et un héritage comportemental, mais seul l'héritage structurel régit la relation d'héritage entre classes. L'héritage structurel repose sur la relation de sous-typage définie dans [Cardelli/Wegner 85] : une classe C est une sous-classe d'une classe C' si le type T de C est un sous-type du type T' de C'. La sémantique du sous-typage est fondée sur l'inclusion d'ensembles : une instance d'une classe est automatiquement une instance de ses super-classes. L'héritage comportemental impose la relation de sous-typage pour les arguments et l'objet retourné des méthodes redéfinies.

6.3. La persistance

La persistance d'un objet ou d'une valeur est octroyée par attachement à une racine persistante. Celle-ci, objet ou valeur, doit posséder un nom. Ce nom a alors une portée globale dans le système. Les racines persistantes sont déclarées dans le langage de commande tandis que les instances persistantes sont manipulées de manière transparente dans le langage de manipulation O2C. Les racines persistantes sont détruites explicitement dans le langage de commande alors qu'un mécanisme de glâneur de cellules, en O2C, assure la récupération automatique des instances non atteignables : la suppression d'une instance ne s'effectue pas explicitement. La persistance est donc orthogonale au type et à la création; elle se propage structurellement; la transparence comportementale est complète; les extensions de classes de type ensemble ou liste, contrairement à Orion, sont gérées par l'utilisateur. Enfin, la

représentation des objets en mémoire et sur disque est identique. L'identité d'objet est un OID logique qui est interprété à chaque accès à l'objet.

6.4. Le langage 02C

La figure 23 illustre la création d'une classe Personne et d'une classe Auteur. La classe Personne est implicitement une sous-classe de la classe Object. Via la clause *inherits*, la classe Auteur est définie comme une sous-classe de la classe Personne. Par défaut, les structures des classes Personne et Auteur sont privées. Dans la classe Auteur, l'attribut "edit" est un objet de la classe Editeur, et l'attribut "livres" est une valeur ensembliste, dont les éléments sont des objets de la classe Livre. De plus, toute racine persistante est introduite par le mot-clé *name*. Dans la figure 23, Serre est donc un objet persistant et Auteurs une valeur persistante de type ensemble. Ils sont accessibles par leur nom dans toute application O2.

```
class Personne
        type tuple (    nom: string,
                        adr : tuple (no : integer, rue : string, ville : string))
        method          Init () : Personne;

class  Auteur inherits Personne
        type tuple (    edit : Editeur,
                        livres : set (Livre) )
        method          Init () : Auteur ,
                        AjoutLivre (l : Livre) ,
                        SuppLivre (l : Livre) ,
                        NbLivres () : integer ;

        name Duras : Auteur;
        name Auteurs : set (Auteur);
```

Figure 23 : Définition de Classes et Création d'Instances

Le langage O2C permet une manipulation uniforme des instances persistantes et temporaires. Ainsi, la figure 24 présente un exemple de programme principal introduit par *run body* avec la création à l'aide de l'instruction *new* d'un objet temporaire "unAuteur", l'assignation uniforme du nom et l'attachement de "unAuteur" à la racine persistante Auteurs, rendant "unAuteur" persistant.

```
run body {      o2 Auteur unAuteur;
        unAuteur= new (Auteur);
        unAuteur->name = "Hugo";
        Duras->name = "Duras";
        Auteurs += set(unAuteur);
        }
```

Figure 24 : Transparence Comportementale

L'implémentation d'une méthode est séparée de sa déclaration et introduite par *methodbody* . Le mot clé *self.* désigne le receveur. Dans la figure 25, la méthode Init initialise

la valeur d'une instance d'Auteur déjà créée et l'ajoute à Auteurs, extension persistante de sa classe.

```
method body Init () : Auteur in class Auteur      {
        self->nom = input (o2 string);            /* lecture d'une chaîne */
        self->adr = input (o2 tuple);             /* lecture d'un tuple */
        self->editeur = nil;                      /* référence à un objet non créé */
self->livres = (o2 set(Livre) ) set ();          /* ensemble vide */
        Auteurs+= set (self);                     /* ajout d'un élément à un ensemble */
        return self;   }
```

Figure 25 : Exemple de Corps de Méthode

Le langage O2C offre une structure d'itération *for* sur les ensembles et les listes, avec possibilité de filtrage par l'emploi de la clause *when*. Le filtrage est utilisé à des fins d'optimisation. La figure 26 illustre l'emploi de cette structure pour éditer les objets de l'ensemble persistant Auteurs. L'exemple de la figure 26a réalise l'édition de tous les éléments de Auteurs, par envoi du message Edit à chacun des objets de l'ensemble. Dans l'exemple de la figure 27.b, seuls les auteurs parisiens sont édités.

```
O2C {  o2 Auteur a;
       for (a in Auteurs)
       [a Edit() ];   }
```

a : Sans filtrage

```
O2C {  o2 Auteur a;
       for ( a in Auteurs when (! strcmp (a->adr.ville, "Paris") )
       [a Edit() ];   }
```

b : Avec filtrage

Figure 26 : Utilisation de la Structure d'Itération for

6.5. Deux modes de fonctionnement

O2 propose deux modes de fonctionnement : un mode développement et un mode exécution. Le mode développement est activé pour concevoir le schéma et mettre au point les méthodes et les programmes O2C. Ce mode utilise les services d'O2Tools. Dans ce mode, le système est interprété donc hautement interactif aux dépens des performances et au risque d'erreurs à l'exécution dues à des incohérences possibles dans le schéma. Le mode exécution est mis en oeuvre pour exécuter une application compilée de manière fiable et efficace. Dans ce mode, le schéma est figé et les appels de méthode sont résolus statiquement dans la mesure du possible, afin de diminuer le surcoût imposé par la liaison dynamique.

6.6. Conclusion

Le système O2 est un SGBDOO complet, avec un environnement de programmation et un générateur d'interfaces graphiques. Il a son propre modèle de données distinguant les notions d'objet et de valeur. Le modèle de persistance (par atteignabilité) est simple, et assure la transparence comportementale. Le langage de programmation du système suit une approche fortement typée. L'existence de deux modes de travail favorise, d'une part le développement souple et flexible des applications, d'autre part l'exécution efficace des applications.

Aujourd'hui, les principales fonctionnalités absentes du système concernent l'évolution de schéma en mode exécution et la gestion des versions. Des interfaces de programmation sont proposées vers C et C++. Ces interfaces permettent à la fois de faire persister des données du langage cible dans O2 et d'utiliser des classes et instances d'O2 dans le langage cible.

7. ORION

Orion est un prototype de SGBDOO développé de 1986 à 1990 au MCC (Austin, Texas). Il a été à la base du produit Itaxe, commercialisé sur Itaxe Systems. Orion ajoute au langage LISP des fonctionnalités BD et OO. Il est destiné à des applications de bureautique, CAO et IA. L'accent est mis sur la gestion de versions, la définitions de contraintes sur les variables d'instance et l'évolution de schéma.

7.1. L'architecture Fonctionnelle

Orion offre une architecture client-serveur. La machine langage d'Orion (cf. figure 27), interfacée avec le langage Lisp, effectue le contrôle d'exécution : elle traite les messages répondant aux méthodes utilisateurs stockées dans Orion, aux méthodes systèmes et aux méthodes d'accès aux objets. Les méthodes systèmes incluent toutes les méthodes Orion pour la définition du schéma, la création et la destruction d'instances, la gestion de transactions etc.... Les méthodes d'accès permettent la recherche et la mise-à-jour des valeurs d'une variable d'instance d'une classe. Le dictionnaire d'objets (persistant) est composé du schéma des classes et des informations sur la localisation des objets persistants.

Le gérant d'objets est composé de trois sous systèmes : objet, transaction et stockage. Le *sous-système objet* fournit des fonctions de haut niveau telles que l'optimisation de requêtes, la gestion des objets complexes et le controle de versions. Le *sous-système transactionnel* s'occupe principalement du contrôle de concurrence et du mécanisme de reprise sur pannes. Le *sous-système de stockage* fournit les accès sur disque en serveur de pages : il gère l'allocation et la désallocation des segments de pages, le placement d'objets sur les pages et le transfert de page entre la mémoire et le disque. Il s'occupe également des index sur les variables d'instance.

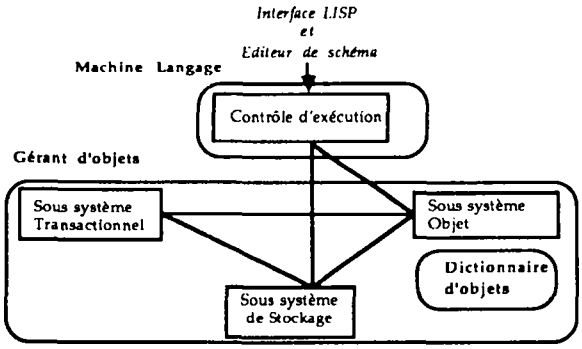

Figure 27 : Architecture Fonctionnelle d'Orion

7.2. Le modèle de données

Le modèle de données d'Orion suit une approche <tout objet>. Un objet possède un OID et une valeur qui peut être primitive ou complexe. Une valeur primitive, ou atomique, est un entier, un flottant, une chaîne de caractères ou un booléen. Une valeur complexe est composée de variables d'instances, qui sont elles-même des objets. Le comportement d'un objet est encapsulé dans des méthodes qui sont des fonctions LISP. Les classes, considérées comme des objets, sont créées par envoi de message à la classe prédéfine *Class*, détentrice des méthodes de création et d'initialisation.

Les classes forment un treillis dont la racine est la classe *Object*. L'héritage est multiple. Orion propose trois modes de résolution des conflits (RC): *automatique, explicite* ou par *renommage*. La RC automatique est effectuée via une liste de priorité. Cette liste est construite à partir d'un ordre de priorité sur les super-classes qui peut être défini automatiquement ou manuellement. Par défaut, la super-classe la plus prioritaire est la première dans la liste des super-classes définies par l'utilisateur. Lors d'un ajout d'une super-classe, celle-ci est considérée comme la moins prioritaire. Toutefois, l'utilisateur peut à tout moment redéfinir l'ordre de priorité. La RC explicite impose à l'utilisateur de spécifier la classe possédant la propriété demandée (par exemple Livre::auteur). La RC par renommage retourne la définition présentant un conflit avec les classes déjà définies dans le schéma et demande de renommer les propriétés concernées.

Orion possède un ensemble de classes prédéfinies (cf. figure 28) dont la racine est la classe *Object* : *Ptype, Class, Collection* et *Set*. La classe Ptype est la super-classe des classes primitives associées aux valeurs simples. La classe Class permet la création des classes utilisateurs. La classe Collection assure la gestion des collections d'objets, en offrant des méthodes (itératives) de parcours et de recherche d'objets. La classe Set, sous-classe de Collection assure la gestion persistante des extensions des classes utilisateurs..

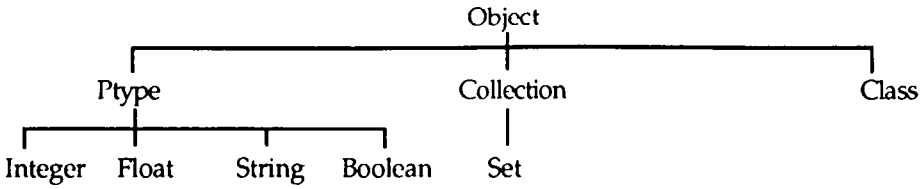

Figure 28 : Classes Prédéfinies d'Orion

Orion permet la définition de *contraintes de composition* sur les variables d'instance des classes. Ces contraintes sont définies au niveau de la classe et régissent les relations entre instances. Elles spécifient des liens de *dépendance* ou d'*exclusivité*. Un lien de dépendance entre deux classes A et B entraîne une dépendance entre la durée de vie d'une instance de A et celle de ses composantes instances de B : par exemple, une contrainte de dépendance entre la classe Livre et la classe PageTitre entraîne la destruction d'une page de titre lors de la destruction de son livre.

Un lien d'exclusivité entre une classe A et une classe B définit une contrainte de partage : Toute instance de B référencée par une instance de A ne peut plus être référencée par d'autres instances. Par exemple, un lien d'exlusivité entre la classe Livre et la classe Illustration et un lien de non exclusivité entre la classe Article et la classe Illustration entraîne que toute illustration d'un livre ne peut être partagée par contre une illustration d'un article peut se retrouver dans un autre article ou dans une autre publication hormis un livre. En l'absence de contrainte, la liaison structurelle entre deux classes est dite *libre*.

De plus, dans Orion, il est possible d'associer à toutes les instances d'une classe une valeur commune pour une de leurs variables d'instance. La variable d'instance est peut être alors à*valeur partagée ou à valeur par défaut* : la notion de valeur partagée est équivalente à celle de variable de classe; la valeur par défaut peut être éventuellement modifiée par l'utilisateur au niveau de chaque instance.

Enfin, Orion propose la *propagation* des valeurs à travers les liens de dépendance. Ces valeurs sont prises comme valeurs par défaut pour les composants dépendants de l'objet composite. Cette propagation ne peut s'effectuer que si les deux objets possèdent respectivement des variables d'instance de même nom. Par exemple, la propagation de la police de caractères à travers le lien de dépendance entre la classe Livre et la Classe PageTitre entraîne qu'un Livre de police "Palatino" a cette police dans sa page de titre. La propagation des valeurs est automatique et prioritaire sur l'héritage : la valeur par défaut définit sur la classe Page, super-classe de PageTitre n'est pas considérée.

7.3. La persistance

Dans Orion, le monde Lisp est temporaire tandis que le monde des classes est persistant. En effet, toutes les classes sont persistantes et toutes leurs instances sont de facto

persistantes. Les notions d'orthogonalité par rapport à la classe et par rapport à la création d'instances sont donc inadéquates. Lors de la création d'une classe C (cf figure 29), sa définition est mémorisée dans le dictionnaire d'objets et une classe miroir C* est définie comme une sous-classe de la classe Set. C* contient deux instances particulières : l'ensemble E des instances de C et l'ensemble E' qui est l'union de E avec les ensembles des sous-classes de C. E' est la fermeture transitive de C dans le graphe d'héritage. Ceci permet l'héritage par inclusion et optimisation de la mise à jour des instances des sous-classes d'une classe lors de la propagation d'une opération d'évolution de schéma.

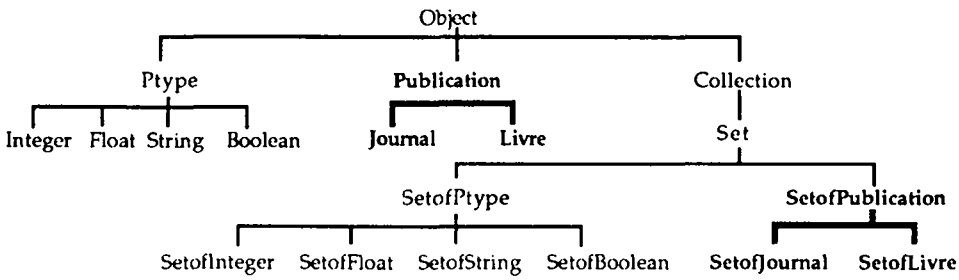

Figure 29 : Représentation des Extensions des Classes Persistantes

7.4. Le langage

Le langage de définition et de manipulation des données persistantes est une extension de Lisp. Orion propose des fonctions Lisp prédéfinies pour la définition des classes, la création des instances et l'expression des requêtes. Cependant, la syntaxe de définition des méthodes utilisateurs n'est pas précisée. La figure 30 illustre la création de la classe Page par envoi d'un message à la classe Class. La longueur par défaut d'une page est de 60 lignes. La largeur d'une page est représentée par une variable d'instance à valeur partagée : toutes les pages ont 80 caractères de large. Les paragraphes sont des variables d'instances dépendantes et non partagées.

```
(make-class   'Page
              :superclasses nil
              :attribute    '(
                    (police    :domain DescriptionPolice)
                    (longueur  :domain integer
                               :init 60)
                    (largeur   :domain integer
                               :share 80)
                    (paragraphes  :domain (set-of Paragraphe)
                               :composite true
                               :exclusive nil
                               :dependent true)) )
```

Figure 30 : Définition de la Classe Page

La création d'un objet est réalisée par l'envoi de message *make* à la classe utilisateur. Si cet objet est composant d'un ou plusieurs objets composites, alors le mot-clef *:parent* crée la référence. Dans ce cas, les contraintes de partage sur le lien de composition sont vérifiées. La

figure 31 crée l'auteur Boris Vian et le rattache à ses deux ouvrages "l'Ecume des jours" et "l'Arrache Cœur" par envoi d'un message make à la classe Auteur.

```
(make        'Auteur
                :parent      (( lEcumeDesJours      ouvrage)
                              (lArracheCœur   ouvrage))
                :nom         " Vian"
                :prénom      "Boris"
                ...)
```

Figure 31 : Exemple de Création d'Instance

7.5. L'évolution de schémas

L'évolution de schéma permet de mettre à jour les éléments de définition des classes : le nom, les variables d'instances, la hiérarchie d'héritage et les méthodes. Le problème est de s'assurer que chacune de ces mises à jour conserve la cohérence globale de la base de données. Celle-ci est définie par des axiomes du modèle de données, appelés invariants. Un ensemble de règles détermine les contrôles à effectuer pour chaque opération de mise à jour. Ces règles optimisent la vérification des invariants par sélection des éléments pertinents de la base de données. De plus, les mises à jour de schéma doivent être répercutées sur les instances de la base de données. Elles peuvent être immédiates ou différées. Dans ce dernier cas, la mise à jour est mémorisée et effectuée uniquement lors de l'accès aux instances.

Orion est un des premiers système à avoir étudié le problème de l'évolution de schéma. Cinq invariants sont définis dans Orion : (1) le treillis de classe est un graphe acyclique connexe qui ne possède qu'une seule racine, la classe Object; (2) toutes les variables d'instances et les méthodes d'une classe ont des noms différents; (3) toutes les variables d'instances et les méthodes d'une classe ont des origines distinctes; (4) l'héritage est total et non sélectif; (5) le domaine d'une variable d'instance ne peut être que spécialisé; (6) l'unicité de l'origine des attributs et des méthodes.

Quatre ensembles de règles sont définies : les règles de résolution de conflits, les règles de propagation des propriétés, les règles de manipulation du graphe d'héritage et les règles topologiques. Les règles de résolution de conflits fixent les priorités d'héritage par défaut des méthodes et des variables d'instances en cas d'héritage multiple. Les règles de propagation des propriétés assurent la propagation correcte et non ambiguë des mises à jour du contenu d'une classe sur toutes ses sous-classes. Les règles de manipulation du graphe d'héritage définissent les modalités de mises à jour des classes dans le treillis. Enfin, les règles topologiques modélisent les contraintes de modification des liens de composition.

La propagation des mises à jour sur les instances n'est adressée que pour les liens de composition. Les deux solutions immédiates et différées sont proposées. La solution immédiate consiste à balayer récursivement toutes les instances d'un objet composite et à mettre à jour ses caractéristiques. La solution en différé est réalisée à l'aide de compteurs de

références (CR) dans les classes et les instances : chaque nouvelle opération de mise à jour incrémente le CR de la classe. Lors de l'accès à une instance, la valeur de son CR est comparé à celui de sa classe, s'il est inférieur, des mises à jours doivent être effectuées; sinon, l'instance est à jour.

7.6. Conclusion

Le système Orion présente un modèle original en introduisant les contraintes de composition. Ces contraintes sont définies au niveau du langage, pour des besoins propres au monde de l'ingénierie (CAO, génie logiciel etc...). Elles ont une implication directe sur le reste du système. En effet, le système Orion les considère pour le placement des objets (les objets liés par des contraintes de composition sont regroupés ensemble sur disque) ou pour la gestion de concurrence. Ces extensions au modèle objet montre la nécessité d'offrir un système extensible où de nouveaux types de contraintes peuvent être définis suivant les besoins des applications.

Le système Orion se distingue également en offrant la persistance du schéma de classes et en gérant les opérations d'évolution sur ce schéma. Enfin, la gestion de la persistance est simplifiée car la cohabitation entre les objets temporaires et les objets persistants n'est pas abordée.

8. GEMSTONE

Pionnier des SGBDOO, GemStone est destiné à des applications de CAO, bureautique, systèmes experts et génie logiciel. Sa commercialisation, par Servio Logic, est effective depuis 1986 sur stations de travail (VAX/VMS, SUN/UNIX etc...). L'originalité de GemStone réside dans son extension purement persistante de Smalltalk appelée OPAL ainsi que dans sa proposition de manipulation des données persistantes au travers de Smalltalk ou de langages procéduraux.

8.1. L'architecture fonctionnelle

L'architecture de GemStone suit le modèle client-serveur. Chaque application cliente possède son processus GemStone et sa liste de dictionnaires permettant d'accéder par leur nom aux classes et aux objets de l'application. toute session GemStone possède un noyau de gestion de données, un interpréteur pour Opal ainsi qu'un langage de définition et de manipulation de données. Toutes les sessions GemStone communiquent avec le gérant d'objet Stone : toute demande d'accès à une donnée partagée provoque un appel de l'interpréteur d'Opal vers Stone. Ce dernier effectue les demandes d'accès sur disque, contrôle les règles de sécurité et coordonne les accès aux données dans un environnement multi-utilisateur. Cette architecture est illustrée sur la Figure 32.

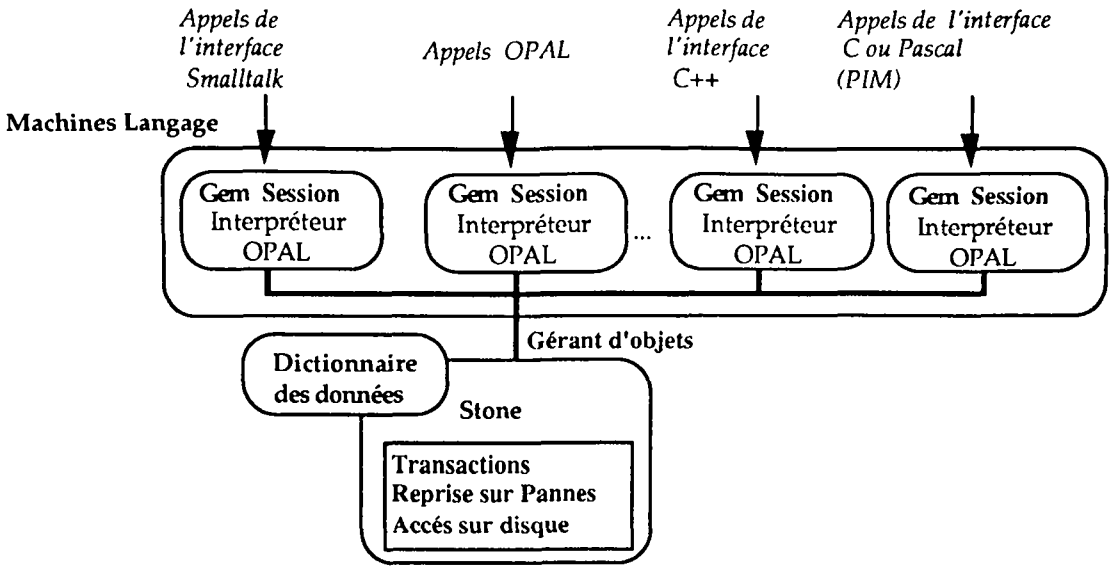

Figure 32 : Architecture Fonctionnelle de GemStone

8.2. Le modèle de données

OPAL prend le modèle de Smalltalk où l'approche <tout objet> est généralisée au niveau des classes. En effet, l'opération de création (cf figure 33)d'une classe entraîne en premier lieu la création automatique d'une méta-classe[12] qui factorise les propriétés spécifiques de la classe : les variables de classes, les informations nécessaires à la création des instances ainsi que les méthodes d'instanciation et d'initialisation. Dans un deuxième temps, la méta-classe crée sa classe qui est son instance unique. La création d'une instance d'une classe C s'effectue par l'envoi d'un message d'instanciation à sa méta-classe. L'approche <tout objet> de Smalltalk implique que les méta-classes soient également des instances : la classe *Metaclass* est introduite à cet effet. Cette dernière, comme toute classe, est instance unique d'une méta-classe appelée *Metaclass Class*. Une boucle d'instanciation est alors créée pour éviter une récursion infinie : *Metaclass Class* est elle-même instance de *Metaclass*.

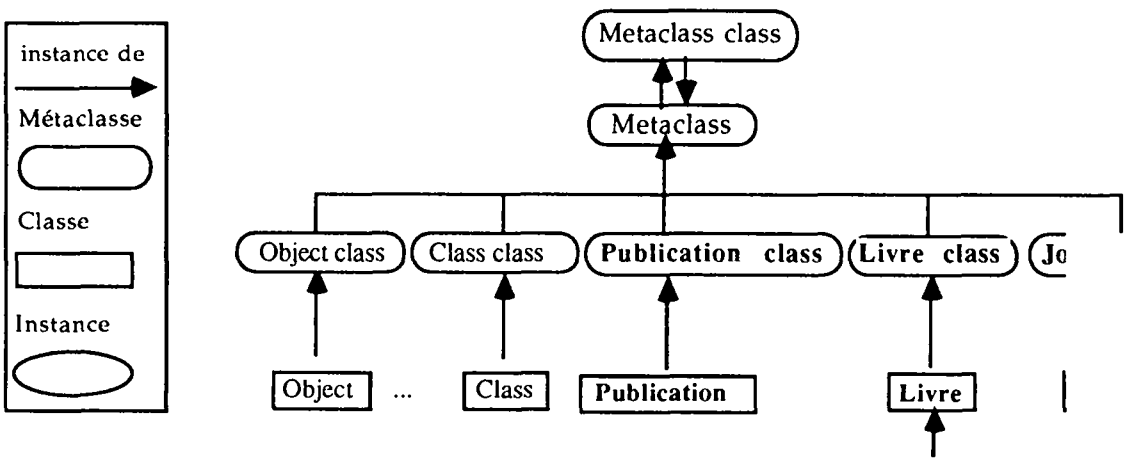

Figure 33 : Arbre d'Instanciation de Smalltalk

[12] Le nom de cette métaclasse est <nom_class > class

L'héritage étant simple, les classes forment un arbre (cf figure 34) dont la racine est la classe *Object*. Elle contient les méthodes définissant un comportement commun telles que l'égalité, la copie ou l'impression ainsi que des méthodes prédicatives telles que l'appartenance à une classe, l'héritage d'une classe ou l'existence d'une méthode. Les classes prédéfinies gérant les informations du schéma des classes sont *Behavior*, *ClassDescription*, *Class* et *MetaClass*. La classe *Behavior* gére le dictionnaire des méthodes, la création par défaut d'instances et l'accès aux instances ainsi qu'à leur structure; elle résoud également les messages. La classe *ClassDescription* gére la suppression ou l'ajout de variables d'instances et l'accès au nom de la classe. La classe *Class* gére la suppression ou l'ajout de variables de classes et permet de déclencher la création d'une sous-classe. Le protocole d'accès aux variables d'instance et aux variables de classe est l'envoi de message, permettant ainsi de définir universellement la portée et les droits sur la structure d'une classe. Les méta-classes reflétant les classes, la relation d'héritage entre deux classes est conservée pour leur méta-classe. Elles forment donc un arbre d'héritage identique à l'arbre d'héritage de leur classe. La méta-classe *Object Class* hérite des mécanismes d'instanciation de la classe *Class* : racine de ce graphe.

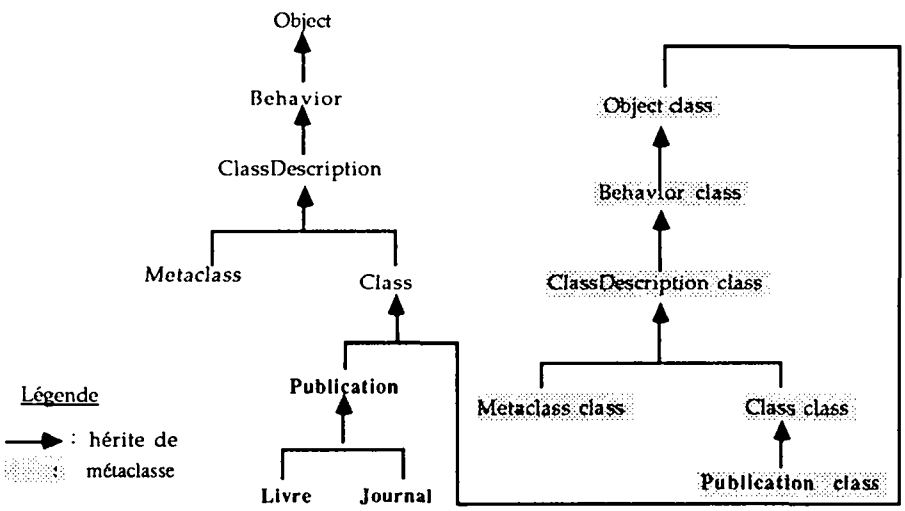

Figure 34 : Arbre d'Héritage des Classes Smalltalk

La considération des aspects base de données ont nécessité des extensions au modèle de Smalltalk telles que les contraintes de composition, la persistance et leur placement indexé. Les contraintes de composition sont spécifiées à la définition d'une classe. Elles apportent des conditions d'invariance et d'unicité de type d'une variable d'instance[13] Une méthode de placement est ajoutée à la classe Object : elle alloue une page disque à l'instance receveuse du message de placement. Des méthodes de regroupement plus fin, permettant le regroupement des instances référencées dans les variables d'instances, peuvent être définies ultérieurement pour une classe donnée.

[13] En Smalltalk, les variables d'instances peuvent avoir des valeurs de types différents

8.3. La persistance

OPAL est un langage purement persistant : il manipule uniquement les données stockées dans Stone. Un environnement de programmation lui est associé; il est composé d'un feuilleteur[14] de classes permettant d'examiner, d'ajouter et de modifier des classes de GemStone ainsi qu'un éditeur permettant la visualisation et l'exécution d'expressions OPAL. Cependant, les applications ont besoin de données temporaires pour des raisons d'efficacité, ainsi que de modules d'entrée/sortie qui sont inexistants en OPAL. GemStone propose donc des protocoles de communication avec C, Pascal, Smalltalk et C++.

Les langages procéduraux et GemStone

L'association du monde procédural au monde objet est similaire à l'intégration d'un système de gestion de fichiers : la gestion des objets persistants s'effectue explicitement à travers un module d'interface appelé PIM[15] (cf Figure 35). Smalltalk et OPAL s'appuient sur le même modèle, mais leurs données évoluent dans des mondes duaux et parallèles : un noyau dupliqué de classes prédéfinies leur permettent de communiquer.

Fetch (Oid, début_instance, fin_instance, Buffer);
Transfère la partie de l'instance référencée par Oid à l'adresse Buffer

Store (Oid, début_objet, fin_objet, Buffer);
Copie le contenu du Buffer dans la partie de l'instance référencée par Oid

FetchInfo (Oid, Taille, ClasseId, Format);
Retourne la taille en octets, le pointeur sur la classe et le format de l'instance référencée par Oid

Instantiate (ClasseId, NouvelObjetId);
Crée une nouvelle instance d'une classe référencée par ClassId et retourne son Oop

SendMessage (ResultatId, ReceveurId, MessageId);
Envoi le message OPAL dont le sélecteur est référencé par MessageId à l'instance référencée par ReceveurId. Le résultat de l'exécution de ce message est retourné dans ResultatId. Par simplification, le mode de passage des arguments n'est pas explicité

ExecuteStatements (ResultatId, Instructions);
Envoie une séquence d'instructions OPAL sous forme de texte, à GemStone, qui la compile, l'exécute et retourne le résultat dans ResultatId

Figure 35 : Procédures PIM de Transferts et d'Envoi de Messages

PIM constitue une interface de bas niveau permettant le contrôle de session, le transfert bilatéral d'instances et l'envoi de messages OPAL depuis le langage procédural. Seules les instances de classes OPAL peuvent être créés par GemStone : la persistance n'est donc pas orthogonale au type. Toute instance persistante est identifiée par un OID de type prédéfini

[14] Feuilleteur = browser
[15] PIM = Procedural Interface Module

Oop[16] . Les définitions des classes et des méthodes sont considérés comme des instances et donc référencées par des *Oop*. Les instances des classes primitives telles que SmallInteger, Character, Boolean et UndefinedObject ont leur valeur directement encodée dans leur *Oop*. L'accès aux variables d'instance peut s'effectuer par envoi de message et exécution dans GemStone, ou par transfert explicite en mémoire. De plus, les instances peuvent être partiellement transférées.

Smalltalk et GemStone

L'interface Smalltalk GemStone fournit un ensemble dupliqué de classes Smalltalk (une quarantaine environ) qui permettent aux applications de communiquer avec la base GemStone telles que la classe *GemStone* représentant la base de données GemStone elle-même; les classes prédéfinies telles que *Integer, String, Symbol, UndefinedObject, Boolean* ou *Float;* les classes similaires aux types constructeurs telles que *Array, Bag, Set* ou *dictionnary* . L'orthogonalité de la persistance par rapport au type n'est pas assurée : la qualité de persistance est donnée exclusivement aux instances des classes OPAL. La persistance des classes Smalltalk est explicite : l'utilisateur doit les définir en OPAL; une table lui permet d'établir la correspondance entre leur nom Smalltalk et leur nom OPAL[17] ; il doit également fournir des méthodes de transformation si le format des variables d'instance diffère. Une instance persistante réside toujours dans GemStone : une instance de la classe *GemStoneObject* , jouant le rôle de représentant (ou proxy), est créée dans l'espace mémoire Smalltalk; elle peut directement répondre au message d'identité et au message d'information sur sa classe d'appartenance; elle empile les autres messages Smalltalk, dont les méthodes associées sont écrites en OPAL, et les retourne vers l'instance persistante à l'exécution. L'envoi de ces messages est explicite : l'utilisation d'une syntaxe ou d'une méthode de la classe *GemStoneObject* spécifique permettent de les distinguer. La classe *GemStoneObject* possède également des méthodes permettant la copie d'une instance temporaire sur GemStone qui provoque le renvoi d'un représentant en mémoire. Cette copie brise les cycles de compositions mais ne préserve pas les références partagées : si deux instances partagent le même composant, celui-ci est copié deux fois.

C++ et GemStone

L'interface C++/GemStone permet la création et la gestion des objets persistants C++ dans une base de données GemStone. Elle comprend une librairie de classes et un*registre*.. La librairie de classes C++ de GemStone correspond au noyau de classes GemStone telles que Object, Array, Bag, Set, Dictionary. Ses classes peuvent directement être utilisées pour la création d'objets C++ persistants ou pour la définition de classes C++ persistantes : la propagation de la persistance se fait par héritage. Toutes ses classes possèdent des méthodes de

---

16 Oop = Object-Oriented Pointer

17 Par défaut, l'identité de nom est choisi par GemStone

transfert d'objets entre GemStone et C++ : *getFromGS()* et *putInGS()* permettent respectivement la lecture et l'écriture d'un objet; *createSymbolAssociation(...)* et *getSymbolAssociation*(...) permettent la définition et la recherche d'un objet par assignation d'un nom dans un dictionnaire.

Le registre crée une interface entre la base de données GemStone et l'application C++. Ses principales fonctions sont : la définition en OPAL des classes C++ persistantes; la génération pour chaque classe de deux fichiers C++ : un fichier de macros établissant la correspondance de format entre instances C++ et Gemstone; un fichier source redéfinissant les méthodes de transfert d'objets décrites plus haut. Ces fichiers sont compilés et liés avec tout programme utilisant les instances des classes persistantes.

8.4. Le langage de définition et de manipulation

OPAL reprend la syntaxe de Smalltalk en apportant des extensions nécessaires au traitement des aspects bases de données telles que la persistance, la concurrence ou le traitement des transactions. Le noyau de classes de Smalltalk a donc été réécrit : Object, Behavior, ClassDescription, Metaclass et Class. Outre ce noyau, Opal offre plus de 40 classes prédéfinies dont les plus usuelles sont : la classe Collection et ses sous-classes (Bag, Set, Dictionary, Array...) permettent la gestion d'une collection d'objets; la classe Magnitude et ses sous-classes (Character, DateTime Number etc...) permettent l'intégration des types simples. Les outils de création et de gestion d'interfaces utilisateurs de Smalltalk ont été abandonnés : leur traitement peut toutefois s'effectuer dans les programmes d'application en langage hôte.

La figure 36 illustre la création des classes Page et PageSet ainsi que des instances pages, goncourt et livreDeLaJungle. Le mot-clef *constraints* signifie que la variable d'instance police n'admet que des instances de DescriptionPolice et que les variables d'instance longueur et largeur n'admettent que des instances de la classe Integer. La méthode **new** sur la classe Livre est redéfinie afin de permettre l'initialisation de la variable d'instance Titre : il retourne le nouvel objet dénoté par *self*. Le message *new* est envoyé aux méta-classes PageSet class, Array class et Livre class.

```
Object  subclass:          'Page'
        instVars:          #('police' 'longueur' 'largeur' 'paragraphes')
        classVars:         #()
        inDictionnary:     UserGlobals
        constraints:       #[#[#police, DescriptionPolice],
                           #[#longueur, Integer],
                           #[#largeur, Integer]]
        isInvariant:       false

Set     subclass:          'PageSet'
        instvarNames:              #()
        classVars:         #()
        inDictionnary:             UserGlobals
        constraints:       Page
        isInvariant:       false
```

```
method: Livre
        new unTitre
                super new
                titre := unTitre
                ^self

pages             <- PageSet     class   new
goncourt          <- Array       class   new
livreDeLaJungle <- Livre         class   new    'Livre de la Jungle'
```

Figure 36 : Définition de Classes en OPAL

Dans Smalltalk, un ensemble de méthodes permettent de gérer l'interface avec OPAL : adjonction de la persistance à une instance, création d'un représentant, envoi de message sur ce représentant et exécution d'instructions OPAL. Le message *asGemStoneObject* entraîne la copie d'une instance temporaire dans GemStone et retourne un représentant. Cette méthode est connue par toutes les classes dupliquées, leurs classes dérivées et les classes utilisateurs par héritage. Deux actions successives sont nécessaires pour créer un représentant : recherche par le nom de l'instance dans le dictionnaire d'objets et création du représentant. La figure 37 illustre ces mécanismes : le message *objectName:* envoyé à la classe GemStone effectue l'accès à l'instance livreDeLaJungle; le message *asLocalObject* sur l'instance livreDeLaJungle crée un représentant de cette instance dans l'espace de travail Smalltalk.

```
robinsonCrusoe          asGemStoneObject
livre DeLaJungle <- (GemStone objetcNamed: 'livreDeLaJungle') asLocalObject
```

Figure 37 : Persistance dans Smalltalk

Les messages, exécutés par une instance de GemStone, sont envoyés via son représentant (cf figure 38) suivant deux protocoles : la méthode *remotePerform:* de la classe *GemStoneObject* est activée par un envoi de message sur son représentant, le nom du message est le premier argument; le nom du message est préfixé par *gs*. Les instances de Smalltalk, passées en arguments, sont copiées dans la base de données, avant l'envoi du message. Ainsi, ce message peut être complètement exécuté dans GemStone. Le résultat du message est un représentant, instance de GemStoneObject. L'exécution d'instructions OPAL dans Smalltalk utilisent la classe String : une de ses instances contient ces instructions.

```
livreDeLaJungle remotePerform: #augmenterPrix #taux
livreDeLaJungle gsaugmenterPrix #taux
```

Figure 38 : Envoi de Messages

8.5. Conclusion

Premier représentant des SGBDOO, GemStone propose plusieurs modes d'accès aux données : interactif via OPAL, interprété via le langage Smalltalk et compilé via les langages procéduraux C et Pascal et le langage OO C++. Cependant, il souffre de la complexité du modèle de données choisi (modèle de Smalltalk). Dans GemStone, la persistance est orthogonale à la création des instances mais elle n'est pas orthogonale au type. Son traitement

suit la même philosophie que celle des langages procéduraux : les messages *asLocalObject:*, *ObjectName:* et *asGemStoneObject:* ont des fonctionnalités similaires aux procédures *Fetch*, *Store* et *Instantiate*; le préfixe *gs* et la méthode *remotePerform:* permettent l'envoi de message à une instance GemStone au même titre que la procédure *SendMessage*; la classe *String* permet de demander une exécution OPAL comme *ExecuteStatement*.

9. CONCLUSION

9.1. Synthèse

Cette synthèse compare les systèmes précédents afin de dégager les points forts et les faiblesses de chacun. Cette comparaison est résumée sous forme de quatre tableaux décrivant respectivement les aspects généraux, l'évolution de schémas, le modèle de données et la résolution de la persistance .

Le tableau 1 décrit les caractéristiques générales, à savoir la période de référence sur laquelle porte l'étude, les langages s'interfaçant au système, l'état d'avancement (prototype ou produit) du système, et les fonctionnalités supplémentaires.

Généralités	Orion	Exodus	Gemstone	Ontos	ODE	O2
Origine	MCC	Wisconsin U	Serviologic	Ontologic	ATT-Bell Labs	O2Technologie
Référence	87 - 89	87-91	84 - 91	89 - 91	89 - 91	88 - 91
Langages Interfacés	Lisp	C++ E	Smalltalk Opal C Pascal	C++	C++	O2C C C++
Etat	Prototype	Prototype	Produit	Produit	Prototype	Produit
Fonctionnalités supplémentaires	Versions		Environnement Graphique		Déclencheur	Environnement Graphique

Tableau 1 : Caractéristiques Générales

Le tableau 2 décrit le statut du dictionnaire de données, le mode d'utilisation et les possibilités d'évolution. Nous remarquons l'influence de la persistance du schéma et du mode d'utilisation sur le traitement de l'évolution de schéma. En effet, l'évolution de schéma n'est possible que si le dictionnaire de données est persistant. C'est pourquoi tout système ne construisant qu'un dictionnaire temporaire lors de la compilation ne peut effectuer d'évolution de schéma que par une modification du source et une recompilation. De plus, le mode compilé entaîne une certaine rigidité du dictionnaire de données : Ontos ne permet aucun traitement d'évolution sur les classes définies en C++. Au contraire, Orion et GemStone évoluant en mode interprété, proposent des opérations d'évolution de schéma. Cependant, leur approche néglige l'étude détaillée de la propagation sur les instances et les évolutions liées aux comportements. Le problème de l'évolution de schéma est une faiblesse majeure de tous ces systèmes.

Evolution de schéma	Orion	Exodus	Gemstone	Ontos	ODE	O2
Dict. de données	Persistant	Temporaire	Persistant	Persistant	Temporaire	Persistant
Mode utilisation	Interprété	Compilé	Interprété	Compilé	Compilé	Compilé
Evolution Schéma	Propagation automatique	Non	Propagation automatique	Non	Non	mode développement

Tableau 2 : Evolution de schéma

Le tableau 3 présente les caractéristiques des modèles de données étudiés. Outre les différences dues à leur langage de référence, les systèmes se distinguent par le choix des contraintes qu'ils définissent. Les modèles s'appuyant sur un langage OO existant proposent peu d'extensions : seul le modèle ODE ajoute la notion de contrainte d'intégrité sur les variables d'instance et de déclenchement conditionnel de méthodes. Les modèles O2 et Orion ajoutent des contraintes entre composants et composés. De plus, dans le modèle Orion, la sémantique des liens entre un composé et un composant est enrichie avec les contraintes d'exclusivité et de dépendance. Le modèle O2 les reprend en limitant le nombre de combinaisons. Les contraintes de visibilité sont considérées uniquement lors de la compilation et n'ont aucune influence sur les instances persistantes.

Modèle de données	Orion	Exodus	Gemstone	Ontos	O++	O2
Modèle	Tout objet	C++	Smalltalk	C++	C++	Objet/valeur
Principe héritage	Classe	Classe	Classe	Classe	Classe	Classe/Type
Type héritage	Multiple	Multiple	Simple	Multiple	Multiple	Multiple
Résolution de conflits	3 Modes	Explicite	-	Explicite	Explicite	Renommage ou Explicite
Contraintes Structurelles	Composition	Composition Visibilité	Domaine	Visibilité	Visibilité Intégrité	Composition Visibilité
Contraintes Comportementales	-	Visibilité	-	Visibilité	Visibilité	Visibilité
Fonctionnalités supplémentaires	Valeurs propagées	-	-	-	Déclencheur	-

Tableau 3 : Modèle de Données

Le tableau 4 résume le modèle de persistance de chaque système. Trois approches de la persistance sont à distinguer. La première, adoptée par Orion et GemStone, propose un langage de classes purement persistant; elle nécessite, pour offrir des données temporaires, des interfaces vers des langages de programmation impliquant des conversions de données et des duplications de types ("impedance mismatch"). La deuxième, adoptée par Ontos, ajoute par l'héritage la persistance à un langage OO au moyen d'une librairie de classes; cette

approche proscrit l'orthogonalité de type, nuit à la transparence comportementale et contraint à redéfinir les méthodes de gestion de la persistance, notamment pour la propagation de la persistanceaux objets composants. La troisième, choisie par Exodus, Ode et O2, consiste à dénoter syntaxiquement la persistance à la création de toute racine persistante, puis de propager la persistance de manière implicite (Exodus et O2) ou explicite (Ode) aux objets composants.

Persistance	Orion	Exodus	Gemstone	Ontos	O++	O2
Langage	Lisp + classes	E	Opal	C++	C++	O2C
Principe	Classes toutes persistantes	Syntaxique	Classes toutes persistantes	Héritage	Syntaxique	Nommage
Orthogonalité / aux types	Non	DB Type	Non	Non	Oui	Oui
Orthogonalité / création d' instance	-	"persistent" "in"	Oui	Oui	"persistent" "pnew"	Oui
Racine persistante	Classe	Instance Collection	Instance	Instance Agrégats	Base d'objets	Instance
Propagation	Héritage et Structurelle	Structurelle	Héritage et Structurelle	Héritage	Non	Structurelle
Transparence comportementale	Non	Oui	Non	Hybride	Oui	Oui

Tableau 4 : Persistance

9.2. Conclusion

Outre les systèmes GemStone, O2 et Ontos, le monde industriel a vu naître de très nombreux SGBDOO, ces dernières années tels que Gbase, ObjectStore, Objectivity/DB et Versant. Le manque de documentation technique nuit à leur étude exhaustive. Cependant, nous constatons un consensus autour du langage C++ comme langage de programmation OO. Ces systèmes sont divisés sur l'ajout de la persistance par héritage de classes comme Ontos et Versant ou par dénotation syntaxique de la persistance comme ObjectStore. Pourtant, le succès immédiat et conséquent de ce dernier système[18] prouve l'attraction de cette approche auprès des utilisateurs. De plus, le modèle de propagation fait toujours l'objet d'un débat : la transparence du modèle d'atteignabilité est contrebalancée par sa complexité et son coût. Il nécessite notamment un glâneur de cellules, traditionnellement décrié par le monde industriel. Cependant, de récents travaux ont démontré non seulement l'efficacité [Sha90] mais aussi la nécessité d'un glâneur de cellules dans un environnement où le système est seul à pouvoir déterminer l'atteignabilité d'un objet [Gru92][19] .

[18] En 1 an de commercialisation, ce système a égalé Ontos dans le nombre et l'importance de ses clients.

[19] Le glaneur de cellules pouvant aussi servir au placement des objets.

Un autre problème est celui des performances des SGBDOOs. Initialement critiqués pour leur lenteur, les SGBDOO ont démontré leur compétitivité avec les SGBD classiques dans de récents bancs d'essai [Cat90]. Le rôle déterminant des couches basses a alors été mis en évidence, notamment dans la politique de migration des objets entre mémoire et disque, les niveaux d'adressages (OID logique ou physique) et niveaux de stockage (correspondance directe[20] ou non de l'image disque en mémoire). La combinaison d'OID physiqus et d'une correspondance directe entre disque et mémoire offre les meilleures performances. En utilisant les adresses mémoires virtuelles comme OID, cette approche introduit néanmoins le problème de la sûreté du langage, remettant en cause le choix de C++.

Enfin, un certains nombre de problèmes connus en bases de données reviennent au premier plan. Le premier est certainement celui de l'optimisation de requêtes , plus difficile qu'en bases de données relationnelles en raison notamment de la présence d'objets complexes. Le second est l'exploitation des systèmes répartis qui nécéssite une meilleure intégration des techniques de BD réparties et BDOO [Ozs91] afin d'assurer la transparence à la distribution des objets. Un dernier problème important est l'interopérabilité des différents SGBDOO entre eux ou avec des SGBD classiques. Là encore, les techniques de Multi-BD doivent être revues !

BIBLIOGRAPHIE

[Agr89a] R. Agrawal, N.H. Gehani,"ODE (Object Database Environment): The Language and the Data Model", on Int. Conf. *SIGMOD*, Portland, Oregon, Vol. 18(2) , June 89.

[Agr89b] R. Agrawal, N.H. Gehani, "Rationale for the Design of Persistence and Query Processing Facilities in the Database Programming Language O++", on Int. *Workshop on Database Programming Languages*, Portland, Oregon, June 1989.

[Agr91] R. Agrawal, S. Dar, N.H. Gehani, "The O++ Database Programming Language : Implementation and Experience", AT&T Internal Report, 1991.

[And87] T. Andrews, C. Harris, "Combining Language and Database Advances in an Object-Oriented Development Environment", on Int. Conf. *OOPSLA.*, Orlando, Florida, October 1987.

[Bel91] M.J. Bellosta, P. Valduriez, F. Viallet, "Design Considerations for OMNIS, an Object Management Interface System", on Int. Conf. *TOOLS*, Paris, France, March 1991.

[But91] P. Butterworth, A. Otis, J. Stein, "The GemStone Object Database Management System", Communications of the ACM, Vol. 34(10), October 1991.

[20] single-level store

[Car85] L. Cardelli, P. Wegner, "On Understanding Types, Data Abstraction, and Polymorphism", ACM *Computing Surveys*, Vol. 17(4), December 1985.

[Cat90] R.G. Cattell, J. Skeen, "Engineering Database Benchmark", Sun Microsystem Internal Report, april 1990, ACM *TODS*, à paraître.

[Cop84] G. Copeland, D. Maier, "Making Smalltalk a Database System", on Int. Conf. *SIGMOD*, Boston, Massachusetts, June 84.

[Del91] C. Delobel , C. Lécluse, P. Richard, "Bases de Données : des Systèmes Relationnels aux Systèmes à Objets", InterEditions (eds), Paris, 1991.

[Gar 90] G. Gardarin, P. Valduriez, "SGBD Avancés", Eyrolles (eds), Paris, 1990.

[Geh91] N. Gehani, H.V. Jagadish, "Ode as an Active Database: Constraints and Triggers", on Int. Conf. *VLDB*, Barcelona, Spain, September 1991.

[Gru92] O. Gruber, L. Amsaleg, L. Daynes, P. Valduriez, "EOS, an Environment for Object-based Systems", on Int. Conf. *HICSS* , Hawai, January 1992.

[Kho86] S.N. Khoshfian, G.P. Copeland, "Object Identity ", on Int. Conf. *OOPSLA*, Portland, Oregon, September 1986.

[Kim87] W. Kim, J. Banerjee, H. Chou, J.F. Garza, D. Woelk, "Composite Object Support in an Object-Oriented Database System", on Int. Conf. *OOPSLA*, Orlando, Florida, October 1987.

[Kim89a] W. Kim, F.H. Lochovsky (eds.), "Object-Oriented Concepts, Databases, and Applications", ACM Press/Addison-Wesley, 1989.

[Kim89b] W. Kim, E. Bertino, J.F. Garza, "Composite Objects Revisited", on Int. Conf. *SIGMOD*, Portland, Oregon, Vol. 18(2) , June 89.

[Mai86] D. Maier, J. Stein, A. Otis, A. Purdy, "Development of an Object-Oriented DBMS", on Int. Conf. *OOPSLA*, Portland, Oregon, September 1986.

[O291] O2 and al., "The O2 System", Communications of the ACM, Vol. 34(10), October 1991.

[O2B92] F. Bancilhon, C. Delobel, P. Kannelakis, "The O2 Book", Morgan Kaufman (eds), à paraître, 1992.

[Ont89] "The Ontos Reference Manual", Ontologic Inc., Burlington, MA, December 1989.

[Ozs91] T. Ozsu, P. Valduriez, "Distributed Database Systems : Where are we now ?", IEEE *Computer*, Vol. 24(1), Aout 1991.

[Pen87] D.J. Penney, J. Stein, "Class Modification in the GemStone Object-Oriented DBMS", on Int. conf. *OOPSLA*, Orlando, Florida, October 1987.

[Pur87] A. Purdy, B. Schuchardt, D. Maier, "Integrating an Object Server with Other Worlds", ACM Transactions on Office Information Systems, Vol. 5(1), January 1987.

[Ric89] J.E. Richardson, M. Carey, "Persistence in the E Language: Issues and Implementation", Software-Practice and Experience, Vol. 19(12), December 1989.

[Ric90] J.E. Richardson, "Compiled Item Faulting", On Int. *Whorkshop on Persistence Object Systems*, Martha's Vineyard, MA, September 1990.

[Sha90] M. Shapiro, O. Gruber, D. Plainfossé, "Garbage Detection Protocol for a Realistic Distributed Object Support System", Technical Report INRIA-1320, INRIA, Rocquencourt, France, November 1990.

[Sch90] D. Schuh, M. Carey, D. Dewitt, "Persistence in E Revisited - Implementation Experiences", On Int. *Whorkshop on Persistence Object Systems*, Martha's Vineyard, MA, September 1990.

Imprimé en France
par
.l'Institut National de Recherche en Informatique et en Automatique.

ISSN 0249 - 6399