Gestion d'objets persistants : du langage de programmation au système

Pascal Dechamboux

THESE

présentée par

Pascal DECHAMBOUX

pour obtenir le titre de

Docteur de l'Université
Joseph Fourier − Grenoble 1

(arrêté ministériel du 23 novembre 1988)

Spécialité : Informatique

Gestion d'objets persistants : du langage de programmation au système

Thèse soutenue devant la commission d'examen le :

3 Février 1993

Sacha Krakowiak	Président
Philippe Richard	Rapporteur
Patrick Valduriez	Rapporteur
Michel Adiba	Directeur
Didier Bert	Examinateur
Mauricio Lopez	Co−directeur

Thèse préparée au sein du Laboratoire de Génie Informatique de Grenoble

Gestion d'objets persistants : du langage de programmation au système

Résumé : Cette thèse décrit la définition et la mise en œuvre d'un langage de programmation pour bases de données : le langage à objets Peplom (PErsistent Programming Language for Object Management). L'approche adoptée a pour but d'offrir une sûreté importante du langage, une implémentation efficace et de la flexibilité pour le schéma des définitions (types, modules, etc...). Le langage proposé n'est pas totalement nouveau puisqu'il est syntaxiquement basé sur le langage C/C++ ce qui lui confère une conformité avec les standards.

La thése présente dans un premier temps un état de l'art en deux parties. La première partie s'attache à étudier les systèmes de types sous l'angle des modèles de mémoire nécessaires à leur implantation. La deuxième partie analyse les SGBD à objets existants et les compare suivant un certain nombre de critères. Le langage Peplom est ensuite décrit en insistant surtout sur l'effort d'intégration de concepts. Un typage fort et l'introduction orthogonale des concepts bases de données dans le langage de programmation lui confère une grande sûreté. Par ailleurs, Peplom apporte une solution intéressante au problème de structuration des programmes qui s'effectue à deux niveaux : les types abstraits structurent les définitions des entités modélisées alors que les modules structurent les données manipulées.

La production de programmes avec le langage Peplom s'effectue en deux phases. La première phase consiste à saisir les définitions composant l'application. Celles−ci sont mémorisées de manière incrémentale dans un dictionnaire géré par le compilateur. Le dictionnaire autorise les incohérences transitoires de la description des programmes, offrant ainsi une grande flexibilité pour le programmeur et une bonne base pour l'évolution de schémas. La deuxième phase réalise la génération d'applications exécutables. Le générateur produit du code supportant le modèle structurel et sémantique des données. Il s'appuie sur la couche système consistant simplement en un gestionnaire de mémoire (persistante et temporaire). Les performances du prototype implanté sont comparables à celles des systèmes C++ persistants offrant moins de fonctionnalités.

Mots clés : Bases de données, langages de programmtion à objets, langages de programmation pour bases de données, polymorphisme, compilation incrémentale, gérants d'objets.

Managing persistent objects: from the programming language to the system

Abstract: Peplom PErsistent Programming Language for Object Management is described in this thesis at two levels: model definition and implementation. Peplom covers three main issues: safety, efficiency and flexibility of the application schema (types, modules, etc...). It is not a completely new language as it is syntactically based on the C/C++ language thus keeping close to standards.

The state of the art, covering the combination of databases and programming languages, is composed of two parts. It first studies possible memory models to support various type systems. Second, it evaluates and compares existing object−oriented DBMS at different levels. The main design thrust of Peplom is the powerful integration of concepts. Thus, strong typing and the integration of database concepts within the programming language enhance safety. Moreover, the problem of programs structuration is tackled by introducing abstract types and modules: abstract types organize intentional data definitions while modules organize extensional data definitions.

There are two phases to produce Peplom programs. The first phase consists in defining application components. A dictionary, managed by the compiler, incrementally records all these components that compose the schema. It authorizes transient inconsistencies of this schema thus offering flexibility to the programmer and allowing schema evolution. The second phase consists in producing ready−to−run applications. The produced object code supports the structural and semantic data models and runs on a low level memory manager. The performance observed on the prototype implemented is comparable to that of persistent C++ systems offering less functionality.

Keywords: Databases, object−oriented programming langages, database programming languages, polymorphism, incremental compilation, object managers.

Je tiens à remercier

Monsieur Sacha Krakowiak, Professeur à l'Université Joseph Fourier de Grenoble et responsable du projet Guide, qui m'a fait l'honneur de présider le jury de cette thèse,

Monsieur Philippe Richard, Directeur de Recherches à l'Institut National de Recherche en Informatique et en Automatique, et Monsieur Patrick Valduriez, Directeur de Recherches à l'Institut National de Recherche en Informatique et en Automatique, qui ont accepté d'être les rapporteurs pour mon travail,

Monsieur Michel Adiba, Professeur à l'Université Joseph Fourier de Grenoble et responsable du projet Aristote, qui m'a accepté au sein du Laboratoire de Génie Informatique, qui m'a encadré tout au long de ces trois années et dont les critiques furent toujours constructives,

Monsieur Mauricio Lopez, Ingénieur au Centre de Recherche Bull de Grenoble et responsable de l'équipe bases de données, pour la confiance qu'il m'a accordée en m'accueillant dans son équipe et pour avoir persévéré dans ce sens,

Monsieur Didier Bert, Chargé de Recherches au Centre National de la Recherche Scientifique, pour avoir accepté de participer au jury et de porter sur mon travail un regard extérieur toujours enrichissant.

Je tiens également à mentionner tous les gens qui m'ont soutenu et encouragé pendant ces trois années de labeur et plus particulièrement André, Bruno, Christian, Christine, Claudia, Daniel, Fabienne, François, Jacques, Pitch, Samer, Serge et Xavier pour les discussions fructueuses que nous avons eues durant tout ce temps sans oublier tous les membres de l'Unité Mixte Bull−Imag ainsi que tous ceux du projet Aristote.

Chapitre I

Introduction

Tout homme est voué à n'être que ce qu'il aura été. Mais quel homme qui ne pense, lorsqu'il va fermer les yeux, aux autres hommes qu'il eût pu être et qui vont mourir avec lui ?

Maurice Genevoix − "Trente mille jours"

Les années 80 ont vu l'expansion, sur le marché des Systèmes de Gestion de Bases de Données (SGBD), des systèmes relationnels. Ils ont ainsi succédé aux systèmes hiérarchiques et aux systèmes CODASYL[40][103] dans le domaine des applications de gestion classiques. Parmi les points forts des SGBD relationnels, il y a la simplicité du modèle de données basé sur des tables composées d'attributs atomiques (relations en première forme normale). Il leur est associé une algèbre (algèbre relationnelle) leur conférant un solide fondemment théorique [33]. L'indépendance entre le niveau physique et le niveau logique de la description des données est un autre atout de ces SGBD. C'est principalement grâce à ce fondemment théorique que les SGBD relationnels ont des interfaces d'accès très voisines les unes des autres qui ont convergé vers le standard qu'est le langage SQL[10]. L'aspect déclaratif du langage SQL est d'ailleurs un élément essentiel du succès des SGBD relationnels.

Cette simplicité du modèle relationnel représente aussi une grande faiblesse lorsqu'il s'agit de modéliser des données plus complexes utilisées dans des domaines d'applications tels que ceux de la CAO, de la bureautique ou des applications géographiques. Cette faiblesse se caractérise par deux aspects différents. Premièrement, les données sont structurées de façon "plate" dans le sens où les structures avec des niveaux d'imbrication ne peuvent être représentées qu'à l'aide de plusieurs relations. La recomposition de la structure imbriquée nécessite alors d'effectuer des jointures (opération la plus coûteuse de l'algèbre relationnelle) entre ces différentes relations. Deuxièmement, il s'agit d'un modèle à base de valeurs ce qui rend difficile la représentation de certaines structures qui nécessite la notion de référence telles que les structures arborescentes (références père/fils). La notion de référence peut néanmoins être simulée par la gestion de clés étrangères qui doivent alors être maintenues par l'utilisateur ou à l'aide de contraintes d'intégrité référentielles. Le déréférençage nécessite comme dans le cas des structures imbriquées une opération de jointure.

Bien que des travaux de recherche soient encore poursuivis dans le cadre des systèmes relationnels notamment sur les aspects de parallélisation [30] ou de distribution, nous pouvons distinguer cinq axes de recherche qui ont succédé aux travaux sur le relationnel. Le premier axe concerne les systèmes non première forme normale dits aussi "NF2" ou à base d'objets complexes [1][56][91] qui permettent de pallier le premier inconvénient en autorisant la représentation de structures imbriquées. Ces modèles manquent cependant, pour l'instant, de richesse car ils ne permettent pas de représenter des structures récursives qui nécessitent la notion de référence.

Le deuxième axe couvre l'approche des SGBD relationnels étendus et des SGBD extensibles [28][45][101]. Les extensions concernent la plupart du temps l'extensibilité du système des types utilisables pour définir les domaines des relations ou encore la possibilité de définir des attributs calculés. Le reproche que nous pouvons faire à cette approche est que si nous autorisons qu'un attribut puisse être un objet identifiable, il faut alors étendre le SGBD relationnel à toutes les fonctionnalités d'un SGBD orienté−objet, tout cela en conservant deux mondes différents.

Le troisième axe concerne les SGBD déductifs [51][86] dont le but est d'ajouter des capacités de déduction à un SGBD. Cette approche est une suite logique des systèmes relationnels puisque leur modèle peut aussi être formalisé par le calcul de prédicats du premier ordre sur lequel se base DATALOG[50].

Le quatrième axe, qui a connu une activité intense ces dernières années, est représenté par les systèmes orientés objet [60][81] (SGBDOO) dont la puissance de modélisation couvre les deux manques soulignés précédemment. Cette approche prend sa source dans les modèles de représentation de la connaissance ("frames") ou dans les langages à objets. Outre une modélisation puissante et assez naturelle des données, ils offrent la possibilité de définir la sémantique de ces données en leur associant des opérateurs de manipulation (définition de types abstraits).

Le dernier axe correspond à une approche symétrique à celle des SGBDOO : celle des langage de programmation persistant. Il s'agit, dans un langage de programmation général, d'introduire la persistance. L'évolution naturelle pour ces langages est ensuite de supporter d'autres fonctionnalités des bases de données. De nombreux projets[8][12][35][64][77][94] travaillent actuellement sur cette approche. Tous n'offrent pas nécessairement un langage orienté objet.

Les SGBD orientés−objet offrent la notion de classe ou de type abstrait qui permet de décrire à la fois une structure de données et des traitements associés. Ces traitements pouvant être a priori aussi complexes que possible, le langage permettant de les exprimer doit alors être complet au sens de Turing. Cette classe de SGBD va donc, à l'inverse des SGBD relationnels, dans le sens d'une intégration plus forte entre données et traitements. Ils gardent néanmoins la philosophie des SGBD dans le sens où leur utilisation s'effectue à l'aide d'un Langage de Définition de Données (LDD) et d'un Langage de Manipulation de Données (LMD). Le travail effectué dans cette thèse s'incrit dans le cadre des SGBD orientés objet et des langages de programmation pour bases de données. Il s'attache aux problèmes liés à l'intégration des langages de programmation et des bases de données en essayant de garder

une vision purement langage des concepts des bases de données utilisés ; c'est une approche de type langage persistant.

Pour développer une application utilisant un SGBD, il était jusqu'alors nécessaire d'utiliser un langage de programmation classique (Cobol, Pascal[57], C[58]) coopérant avec le SGBD. Cette coopération introduit de gros problèmes de dysfonctionnement entre les deux mondes notamment avec les systèmes relationnels. Il y a d'abord un dysfonctionnement au niveau des systèmes de types où, en général, celui du langage de programmation est plus riche que celui du SGBD. Le deuxième dysfonctionnement se situe dans la partie langage où les paradigmes sont très différents : un langage de programmation impératif opérant des accès navigationnels coopérant avec un langage déclaratif opérant des accès associatifs et ensemblistes aux données. Les SGBD orientés objet apportent déjà de nombreux éléments de réponse à ce problème de dysfonctionnement.

Dans cette thèse, nous nous intéressons à l'approche des Langages de Programmation pour Bases de Données (LPBD) dont un des buts est de supprimer totalement les dysfonctionnements signalés ci−dessus. Ce travail a été effectué dans le cadre du projet Aristote[11] dont le but est de fournir un environnement pour le développement d'applications à objets et multimédia. Les LPBD doivent donc avoir la puissance algorithmique de langages impératifs classiques comme Pascal[57] ou C[58], ou de Langages de Programmation Orientés Objet (LPOO) tels que C++[102] ou Smalltalk[54] tout en offrant les fonctionnalités qui caractérisent les SGBD à savoir la persistance, la concurrence, les transactions, ou encore les accès associatifs et ensemblistes.

Définir un LPBD demande de travailler à plusieurs niveaux. Il faut tout d'abord définir le langage et notamment le système de types qu'il supporte, celui−ci définissant le modèle des données manipulées. Il faut aussi intégrer les concepts bases de données nécessaires au langage ce qui n'est pas nécessairement évident.

Il faut ensuite définir le fonctionnement et la structure du compilateur. Etant donné que les données peuvent persister, le problème de l'évolution des définitions et de leur stockage devient crucial. Il est par exemple nécessaire que le niveau de sécurité des définitions soit au moins aussi bon que celui des données qu'elles représentent. Ce problème est réglé dans les SGBD relationnels en stockant les définitions dans la base (métabase).

Enfin, il faut définir la machine base de données (gestion de la mémoire, gestion du disque) sur laquelle s'appuie la machine abstraite du langage. Le niveau de fonctionnalité de cette machine abstraite détermine l'importance de la sémantique du langage gérée par le compilateur (génération de code). La principale question est alors de déterminer à quel niveau doit être supporté le modèle de données.

I.1 Langages et systèmes de types

Le but de l'approche LPBD est donc d'intégrer toutes les fonctions généralement offertes par un SGBD dans un langage de programmation général. La première question à se poser est alors de savoir si nous allons définir un nouveau langage ou essayer d'introduire ces fonctions dans un langage existant. Il est évident qu'imposer un nouveau langage dans l'environnement industriel aussi bien que celui de la recherche est une chose quasiment impossible. La tendance actuelle concernant les langages de programmation est d'évoluer du langage C, qui est largement répandu, vers son extension objet C++. A partir de ce constat se pose alors une autre question qui est de savoir s'il est possible de faire de C++ un LPBD et si cette approche est viable. Nous essayons de donner dans cette thèse, si ce n'est une réponse catégorique, tout au moins des éléments de réponse, permettant d'évaluer les difficultés et les problèmes posés.

Un élément important dans ce contexte d'intégration des langages de programmation et des SGBD est que dans les deux domaines un consensus se dégage sur l'intérêt de l'approche objet. D'autres domaines, tel que la représentation de la connaissance ou l'intelligence artificielle, abondent aussi dans ce sens. Or derrière cette notion d'objets, nous trouvons une multitude de concepts qui varient en nombre mais aussi en sémantique suivant le domaine d'utilisation. Parmi ces concepts, nous pouvons citer l'abstraction de donnée, l'encapsulation, l'identificateur d'objet, le polymorphisme ou encore la notion de type, de classe, de référence ou de valeur. C'est d'ailleurs certainement à cause de cela qu'aucun standard de modèle d'objet n'a encore émergé malgré les efforts produit dans ce sens par exemple par l'OMG[37]. Le problème se complique encore lorsqu'on introduit d'autres concepts qui interfèrent avec les premiers comme la persistance. Nous allons analyser toutes les dimensions de cette notion d'objet et isoler celles qui nous intéressent dans le contexte d'un LPBD.

L'expérimentation que nous poursuivons dans le cadre des LPBD s'est traduite par le développement du langage PEPLOM (PErsistent Programming Language for Object Management). L'approche retenue a été d'intégrer de manière orthogonale et homogène des concepts des bases de données dans un langage de programmation dont la souche est le langage C++. Cette intégration s'est opérée à deux niveaux : dans le système de types et dans le langage.

Une des exigences de l'approche LPBD est d'avoir un système de type orthogonal à la persistance ce qui signifie que des données de n'importe quel type peuvent persister. C'est le cas de PEPLOM dont le système de types reprend pour une part celui de C++ (suppression de la notion de pointeur) avec des extensions telles que les constructeurs ensemble et liste, et les références aux objets. Concernant le modèle d'objets, nous avons conservé uniquement l'héritage virtuel de C++.

Un apport important de PEPLOM est qu'il offre deux concepts de structuration et d'encapsulation des traitements. D'une part les types abstraits (ou classes) qui sont la base des modèles à objets et d'autre part les modules qui permettent une structuration de plus forte granularité. Cette dernière notion s'apparente à celle définie par les langages modulaires.

Enfin, un dernier point intéressant du langage est que les traitements sont divisés en deux catégories : les fonctions et les procédures. Les fonctions n'ont pas d'effets de bord sur le contexte global, ce qui permet notamment d'assurer que les accès associatifs qui les utilisent ne modifient pas les données de la base. Cette distinction permet également de connaître statiquement le type d'accès à mettre en œuvre sur les données de la base (accès en lecture ou lecture/écriture).

Nous avons donc choisi une approche différente de celle d'un C++ persistant pour plusieurs raisons. Tout d'abord, l'intégration de la persistance dans C++ n'est, dans les C++ persistants existants, quasiment jamais orthogonale [67][87][105] (la plupart attache la persistance à certaines classes) et de plus, certaines fonctions de C++ ne sont pas supportées par les classes ou les objets persistants ce qui représente une perturbation grave pour le programmeur. De plus, un LPBD doit offrir un niveau de sécurité au moins équivalent à celui des SGBD actuels. Or ce n'est absolument pas le cas pour un C++ persistant dont la sureté est très faible et dont certains problèmes importants sont aux risques et périls de l'utilisateur. C'est notamment le cas des définitions liées aux données de la base qui sont généralement gérées par l'utilisateur dans des fichiers.

Avec le langage PEPLOM et l'environnement intégré qui l'entoure, nous supportons d'une manière intéressante différents points forts des SGBD relationnels :

- il est possible d'exprimer des accès ensemblistes associatifs "à la SQL" dans le langage. Cette possibilité est offerte sous la forme d'expressions du langage faisant intervenir des fonctions génériques (select, group, etc...).
- la notion de vue est fournie à deux niveaux. Au niveau des données ou instances où nous pouvons définir des variables auxquelles nous associons une expression du langage définissant sa valeur. Au niveau des définitions où nous pouvons introduire (suivant l'utilisateur) différents points de vue sur une définition donnée (différentes interfaces d'utilisation).
- nous voulons offrir aussi des notions de droits et de protection avec l'introduction d'utilisateurs et de groupes d'utilisateurs. Cette protection définie au niveau du schéma (espace des définitions du langage) est essentiellement statique (utilisée par les outils de l'environnement de développement).

Par ailleurs, nous considérons que la coopération avec le monde C et C++ est nécessaire. Nous proposons pour cela d'offrir une coopération uni−directionnelle qui permet au programmeur PEPLOM d'utiliser du code C ou C++ à travers un mécanisme de coopération simple. La partie persistante d'une application est donc développée exclusivement en PEPLOM ; nous supposons que cela ne représente pas une contrainte insurmontable dans la mesure où PEPLOM est très proche de C++.

I.2 Fonctionnement du compilateur

La mise en œuvre d'un LPBD tel que PEPLOM ne se résume pas à l'implantation d'un compilateur du type de celui du langage C ou C++. Nous avons dit qu'un LPBD devait garantir une sécurité au moins aussi bonne pour le schéma que pour les données de la base.

Pour cela, nous définissons un environnement qui s'occupe de gérer toutes les définitions du langage (les sources) qui s'appuie sur un dictionnaire persistant et qui s'assimile au gérant du schéma d'un SGBD. Puisque c'est désormais l'environnement de programmation qui gère les sources ainsi que leur cohérence, il faut fournir à l'utilisateur tout un ensemble d'outils permettant de manipuler le schéma.

Par ailleurs, l'approche LPBD diffère de celle d'un SGBD dans la mesure où les définitions ne sont plus seulement des définitions de structures de données et de points d'accès vers des données, mais aussi des définitions de traitements pouvant s'exécuter sur les données. Cela signifie notamment qu'au cours du développement, le schéma va évoluer fréquemment, ce qui n'était pas le cas pour un SGBD. Il faut donc pouvoir offrir au programmeur une certaine souplesse lui permettant notamment de passer par certains états incohérents comme c'est généralement le cas lorsqu'on développe dans un langage quelconque. Or nous avons pour le langage PEPLOM une approche compilée. Nous allons donc distinguer deux grandes phases dans le cycle de production d'une application PEPLOM exécutable : la phase d'édition/validation et la phase de génération.

Phase d'édition/validation

La première phase consiste à éditer les différentes définitions composant l'application et à les valider. Nous pouvons extraire deux sous−phases de cette première phase. La première sous−phase, appelée édition, consiste à saisir le source de la nouvelle définition ou de celle à modifier et d'opérer des vérifications de conformité de cette définition suivant les règles syntaxiques et sémantiques de PEPLOM. La deuxième sous−phase, appelée validation, survient lorsque l'utilisateur décide d'intégrer la nouvelle définition ou celle modifiée dans le dictionnaire des définitions. Cette phase peut être coûteuse car le dictionnaire va maintenir (et donc calculer) toutes les dépendances entre les différentes définitions. Il faut néanmoins souligner que les définitions intégrées dans le dictionnaire ne doivent pas obligatoirement être dans un état cohérent ce qui est un élément de souplesse ; le dictionnaire est en effet capable de gérer ces incohérences.

Phase de génération

La deuxième phase dans la production d'une application exécutable consiste à générer d'un côté le code correspondant aux différentes définitions PEPLOM qui la composent et d'un autre côté la base de données sur laquelle l'application va travailler ou plus précisément sa structure d'accueil. Le code généré est dans le cas de PEPLOM du code C++. Pour cette phase, l'environnement offre un outil de génération intéressant puisque le programmeur n'a pas à gérer tous les éléments qui composent une application. En effet, le générateur détermine automatiquement toutes les définitions nécessaires à l'application et seulement celles−ci. Il produit alors le code C++ correspondant ainsi que tous les éléments nécessaires à la compilation de ce code de manière à obtenir au bout du compte le binaire de l'application. Un atout important de l'approche de PEPLOM que nous retrouvons dans les C++ persistants [80][85][105] est qu'il ne s'agit pas d'un système "fermé". Cela signifie que l'application générée peut être exécutée comme n'importe quelle

application exécutable par le système d'exploitation (dans le cas présent Unix). Elle ne nécessite pas d'être dans un environnement propre au SGBD pour être lancée.

Nous voyons donc que l'environnement de développement offert pour le langage PEPLOM est un environnement totalement intégré qui offre aussi une grande souplesse au développeur. Cette souplesse se caractérise par le fait que certaines définitions peuvent être dans un état incohérent à un instant donné. Si une définition incohérente fait alors partie des composants nécessaires à la production d'une application exécutable, cela entraine actuellement l'impossibilité de la générer. Nous pouvons envisager par la suite de faire de la génération dégradée dans certains cas lorsque nous voulons produire un exécutable destiné à tester en partie une application pendant la phase de développement.

I.3 Génération de code et machine cible

Pour décrire complètement un LPBD tel que PEPLOM, il faut définir l'environnement d'exécution sur lequel il s'appuie. Le problème ici est de déterminer à quel niveau se situe la frontière entre les fonctions mises en œuvre par le compilateur et la machine abstraite sur laquelle s'appuient ces fonctions. Cette frontière consiste en fait à définir la barrière entre compilation et interprétation. La position de cette barrière a une influence à plusieurs niveaux :

- les conséquences sur les performances peuvent être importantes, l'interprétation étant généralement plus coûteuse.
- la quantité et la complexité du code à écrire pour l'environnement d'exécution peuvent être très variables.
- l'ouverture ou la possibilité d'inter−opérer avec le langage sont dépendants de ce choix.

Nous avons choisi, pour PEPLOM, de repousser le niveau d'interprétation le plus bas possible. Nous voulons donc que le compilateur (générateur de code) supporte le plus grande part de la sémantique du langage. La conséquence est une plus grande complexité du générateur de code. Nous considérons que cette approche est préférable à celle qui consiste à mettre la complexité dans le noyau du système en lui faisant, par exemple, supporter le modèle de données complet. En effet, nous disposons d'outils formels permettant de valider un compilateur notamment par l'utilisation de sémantique opérationnelle. La couche sur laquelle va s'appuyer le générateur consiste essentiellement en un gestionnaire mémoire évolué prenant en compte les aspects de persistance, de concurrence et de reprise après panne.

Le modèle de données (leur structuration) est ainsi en grande partie supporté par le compilateur. Le système de types du langage PEPLOM étant très proche de celui de C++, nous pourrions envisager d'implanter les objets sous la forme d'objet C++ et d'avoir une approche de persistance de ces objets. Cette solution a notamment l'intérêt de récupérer l'implantation de la gestion du polymorphisme des méthodes C++. Or cette gestion nécessite de nombreuses informations de lien (adresses virtuelles) qui sont des pointeurs dans le code programme C++ qui s'exécute. Cela signifie que le format mémoire d'un objet C++ ne peut

pas être le même que le format persistant. Il faut alors mettre à jour ces information de lien à chaque chargement d'un objet persistant, générant ainsi une surcharge de travail.

Nous avons donc choisi d'implanter notre propre mécanisme pour l'appel de méthode qui correspond mieux aux besoins de PEPLOM. Le format d'objet est propre à PEPLOM et ne contient pas d'adresse. La notion d'identificateurs (typés) sert au mécanisme d'appel de méthode et permet des optimisations. Le but de cette approche compilée est d'être le plus efficace possible et de réduire au maximum la complexité des couches basses.

L'architecture du système supportant un LPBD diffère de celle d'un SGBD. La tendance actuelle d'architecture pour les SGBD à objets comme des SGBD relationnels est l'architecture client/serveur. Les systèmes relationnels se prètent bien à ce genre d'architecture car l'interaction entre l'application (client) et le SGBD (serveur) est essentiellement basée sur des requêtes ensemblistes. L'approche client/serveur à ce niveau n'est plus viable pour un SGBDOO ou un LPBD, où un gros travail sur les données de la base doit pouvoir s'opérer directement depuis l'application (accès navigationnels aux données de la base). Des architectures possibles sont alors soit une approche client/serveur au niveau de la gestion de la mémoire persistante (e.g. serveur de pages ou d'objets) ou une approche mémoire partagée (distribuée). Les SGBD à objets actuels utilisent généralement soit des serveurs de pages, soit des serveurs d'objets. Le prototype de PEPLOM utilise un gestionnaire de pages s'appuyant sur la mémoire partagée et ne fonctionne qu'en centralisé. Nous pouvons néanmoins envisager de garder cette approche dans un contexte distribué (mono−serveur voir même multi−serveurs) en utilisant les mémoires virtuelles distribuées qui peuvent être implantées à l'aide d'une technologie de type micro−noyau comme celles de Chorus ou de Mach.

I.4 Organisation de la thèse

Nous pouvons distinguer deux grandes parties dans cette thèse. La première est une analyse des nombreux concepts introduits par l'orientation objet dans les bases de données. Elle se base sur une étude bibliographique et propose un état de l'art, qui n'est pas exhaustif, de SGBD à objets existants. Cette partie est couverte par les chapitres 2 et 3.

Le deuxième partie, qui contient les chapitres 4, 5 et 6, décrit le travail que nous avons effectué dans le domaine des langages de programmation pour base de données. Trois aspects sont principalement traités : la définition du LPBD PEPLOM, les techniques de compilation incrémentale et l'environnement d'exécution du langage.

Chapitre 2

> Cette analyse des différentes notions présentes dans les différents SGBD supportant le paradigme "objet" commence par l'étude de l'aspect structurel des modèles de données proposés par ces systèmes. Nous essayons notamment d'évaluer les incidences de ces modèles de données sur les modèles de mémoire à mettre en œuvre pour les supporter. Dans cette optique, une étude de cas sur différents systèmes à objets est proposée. Le chapitre se termine par une étude des concepts d'un niveau sémantique supérieur tels que l'héritage, le sous−typage ou encore la généricité.

Chapitre 3

Cet état de l'art propose une étude de différents SGBD à objets existants. Cette étude s'attache à faire une évaluation et une comparaison à cinq niveaux : le modèle de données structurel, la puissance sémantique du système de types, la déclarativité présente dans le langage de manipulation de données, la sécurité à la fois du langage et du système et, l'efficacité induite par les techniques de mise en œuvre utilisées par ces systèmes.

Chapitre 4

Ce chapitre contient la définition du langage de programmation pour base de données PEPLOM. Il décrit les deux niveaux de structuration de programme introduits : les types abstraits (programmation à objets) et les modules (programmation modulaire). Chacun de ces niveaux a un rôle très spécifique. Nous décrivons aussi de quelle manière l'aspect base de données est introduit dans le langage par deux concepts simples et orthogonaux aux concepts initiaux de programmation impérative à objets : les collections et la persistance.

Chapitre 5

Dans un contexte où les données persistent, il est essentiel que leur définition persiste aussi car ces données n'ont aucune signification si nous n'avons pas connaissance du type qui les a engendrées. Nous avons pour cela un dictionnaire de ressources persistant qui contient les définitions de schémas de bases ainsi que des programmes qui les manipulent. Il sert de base au compilateur du langage qui utilise des techniques de compilation incrémentale.

Chapitre 6

Ce chapitre décrit l'environnement d'exécution du langage PEPLOM. Nous présentons l'architecture du prototype implanté ainsi que la structuration logicielle de cet environnement. Nous avons suivi une approche compilée dans laquelle le code généré gère directement la mémoire (pas d'interprète de structure de données) pour toutes les structures adressables statiquement et s'appuie sur un interprète pour les collections. La machine de stockage qui est décrite est donc d'un niveau fonctionnel très primitif ; elle connaît principalement la notion d'identificateur logique et de segments de mémoire contiguë. Nous proposons aussi un format d'objet spécifique, où la mise en œuvre de l'héritage multiple est proche de celle de C++ sans avoir les inconvénients que cette dernière présente dans un contexte persistant. Enfin, quelques mesures des fonctionnalités implantées sont proposées et sont comparées (ordre de grandeur) à celles obtenues pour des systèmes existants.

Chapitre 7

La conclusion résume les différents aspects abordés par la thèse et met l'accent sur les apports du travail effectué. Elle propose une vision prospective du domaine des SGBD ou LPBD à objets. Elle se termine par les perspectives qu'ouvre ce travail.

Chapitre II

Systèmes de types et bases de données

Nous ne voyons pas les choses mêmes ; nous nous bornons, le plus souvent, à lire des étiquettes collées sur elles. Cette tendance, issue du besoin, s'est encore accentuée sous l'influence du langage. Car les mots (à l'exception des noms propres) désignent des genres.

Henri Bergson − "Le Rire"

Nous pouvons définir une base de données comme un espace de définition et de stockage d'un ensemble de données représentant d'une manière plus ou moins approximative des faits du monde réel. Nous parlons alors de modèle de données (modèle conceptuel) suivi par une base qui détermine, par sa puissance d'expression, une représentation possible du monde réel. Suivant le modèle conceptuel, l'influence sur le modèle physique est variable : elle détermine l'indépendance physique. Nous retrouvons de la même manière le souci de modélisation des données dans les langages de programmation à travers la notion de type. Un type détermine un domaine de valeurs potentielles manipulables dans un programme. Son rôle principal est de définir la cohérence des manipulations des données par les instructions du langage. Le rôle de modélisation est aussi prépondérant puisque le système de types d'un langage définit la représentation des entités manipulées ainsi que les liens d'interdépendances qui peuvent exister. Il est donc intéressant de regarder en quoi les concepts introduits par l'orientation objet permettent, de nouveau, d'envisager une coopération plus profonde entre les langages de programmation et les SGBD. L'objectif d'un LPBD est d'aboutir à une intégration la plus forte possible.

Nous nous proposons dans ce chapitre d'isoler et d'analyser les différents concepts introduits par les systèmes de types en insistant sur les besoins qu'ils créent au niveau des représentations physiques mises en œuvre. Cette étude commence par les SGBD de première et deuxième génération en cherchant à extraire la notion de type présente dans leur modèle. La section suivante introduit un modèle d'espace d'instanciation (espace contenant les données de la base). Ce modèle est alors utilisé comme cadre de comparaison pour différents modèles de données définis dans des SGBD ou LPBD à objets existant. Une autre section s'attache ensuite à faire une synthèse des différents concepts introduits par les langages à objets (héritage, polymorphisme, généricité, etc...) et à étudier les interactions entre ceux−ci dans le contexte d'un SGBD. Le chapitre se termine par une conclusion sur ce qui peut guider les choix qui peuvent être faits pour le modèle de données par rapport à différents objectifs.

II.1 Les types dans les SGBD traditionnels

L'une des caractéristiques importantes d'un SGBD est la puissance de modélisation qu'il offre. Cette section tente d'extraire les différentes notions de typage qui sont présentes dans les modèles de données hiérarchiques, réseaux et relationnels. Nous pouvons trouver des descriptions de ces différents modèles dans [40][43][103].

Pour illustrer la capacité de modélisation pour chacun de ces trois cas, nous utilisons le schéma de données représenté par le schéma entité−association de la figure Fig. 2.1.

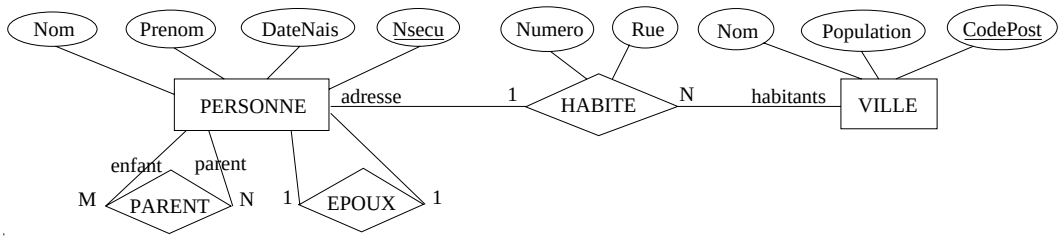


Fig. 2.1 : Schéma de la base d'exemple à modéliser

II.1.1 Les SGBD hiérarchiques

Une modélisation hiérarchique de données est une organisation que nous rencontrons fréquemment pour représenter des entités du monde réel. Cette technique est en effet couramment utilisée pour faciliter la compréhension d'une organisation. Elle permet notamment de décomposer les données en les classant par niveaux, ces niveaux offrant différentes abstractions plus ou moins proches du monde représenté. Nous pouvons citer de nombreux exemples d'organisations du monde réel se calquant sur ce modèle comme les taxinomies de plantes ou d'animaux, ou l'organisation fonctionnelle d'une entreprise ou d'une administration.

Dans le modèle hiérarchique, les données sont décrites sous la forme d'arborescences orientées et sans cycle où chaque noeud correspond à priori à une classe d'entités du monde réel et où les liens père/fils sont tous des liens multivalués (le lien fils/père étant monovalué). Chaque entité est définie par un enregistrement constitué d'attributs de valeurs atomiques (entités de base du modèle c'est à dire les chaînes de caractères et les entiers). C'est donc essentiellement un modèle à base de valeurs où chaque entité est représentée par l'enregistrement la définissant et toutes ses entités filles. L'exemple de la figure Fig. 2.1 se modélise de la manière décrite dans la figure Fig. 2.2.

L'entité VILLE est la racine de l'arborescence puisqu'elle contient naturellement un ensemble de personnes (entité PERSONNE) qui sont ses habitants. L'association HABITE est donc logiquement représentée par le lien hiérarchique entre ces deux entités. Par contre, les autres associations sont difficiles à représenter sous forme de liens hiérarchiques car il ne peut pas y avoir de relation contenant/contenu comme dans le cas précédent. En effet, une personne ne peut à la fois appartenir à une ville et à une autre personne. Il faut alors dupliquer

l’entité personne concernée ou au moins les attributs qui permettent de l’identifier (attributs clés). Il se pose alors le problème du maintien de la cohérence de ces informations dupliquées. Cette carence est liée au modèle à base de valeurs dans lequel les données ne peuvent pas être partagées par d’autres données.

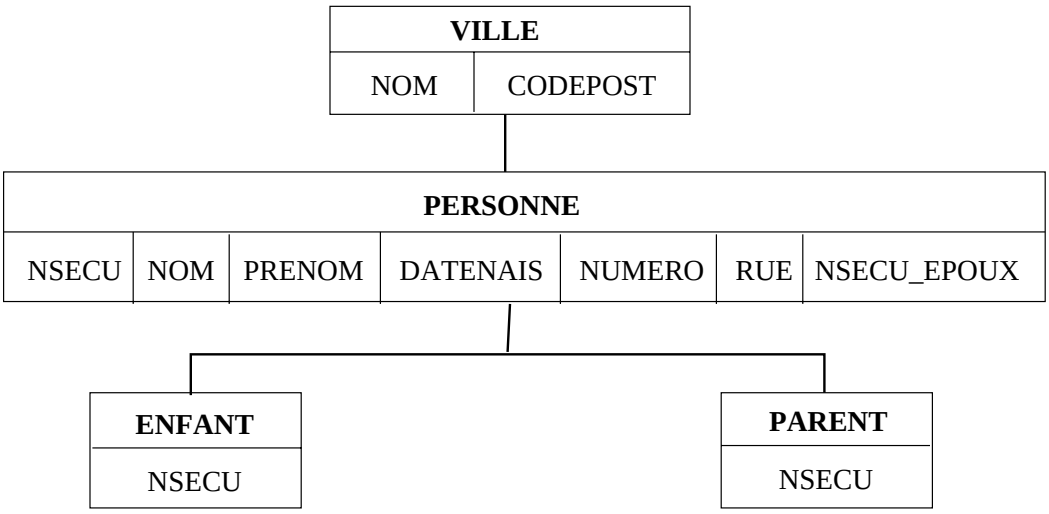


Fig. 2.2 : Représentation hiérarchique de la base d’exemple (sans référence)

Pour pallier cette faiblesse, le modèle hiérarchique est généralement étendu en introduisant la notion de pointeur ou d’attribut virtuel. Cette extension est nécessaire pour modéliser notamment les associations M:N, le partage d’une entité entre plusieurs autres ou encore les associations n−aires (pour n > 2). L’exemple peut alors être modélisé de la manière décrite dans la figure Fig. 2.3.

Avec l’introduction de pointeurs ou de références, le modèle hiérarchique permet de représenter n’importe quelle entité du monde réel. Par contre, une modélisation hiérarchique ne permet pas toujours une représentation très naturelle, et le fait que toute entité est obligatoirement contenue dans une autre (sauf la racine) et n’est accessible qu’à partir de celle−ci, est un inconvénient majeur. Si nous considérons qu’un type décrit une entité et ses liens vis−à−vis des autres, il décrit, dans le modèle hiérarchique, les attributs caractérisant une entité et il désigne le type de l’entité père. Le nom du type sert alors aussi à accéder à ses entités depuis l’entité "père". C’est la racine système qui permet d’accéder à l’ensemble des entités des types racines (types sans type père).

Pour résumer, le modèle hiérarchique comporte principalement deux concepts : l’enregistrement d’entité de base permettant de représenter les entités abstraites du monde modélisé et le lien hiérarchique (lien père/fils 1:N) permettant de représenter les associations entre ces entités. Ce modèle peut alors être représenté par un système de types où chaque type définit des entités et aussi l’entité à laquelle celles−ci appartiennent (lien hiérarchique vers le type "père"). Les entités de base du modèle sont les chaînes et les entiers et, avec l’introduction des pointeurs, la référence vers une entité. La représentation mémoire (essentiellement mémoire secondaire) nécessaire à l’implantation de ce modèle est donc relativement simple : des enregistrements de tailles fixes et des ensembles d’enregistrements

(qui sont les ensembles de fils pour les associations père/fils). Sans l'extension aux pointeurs, le modèle ne nécessiterait pas la notion d'adresse d'un enregistrement.

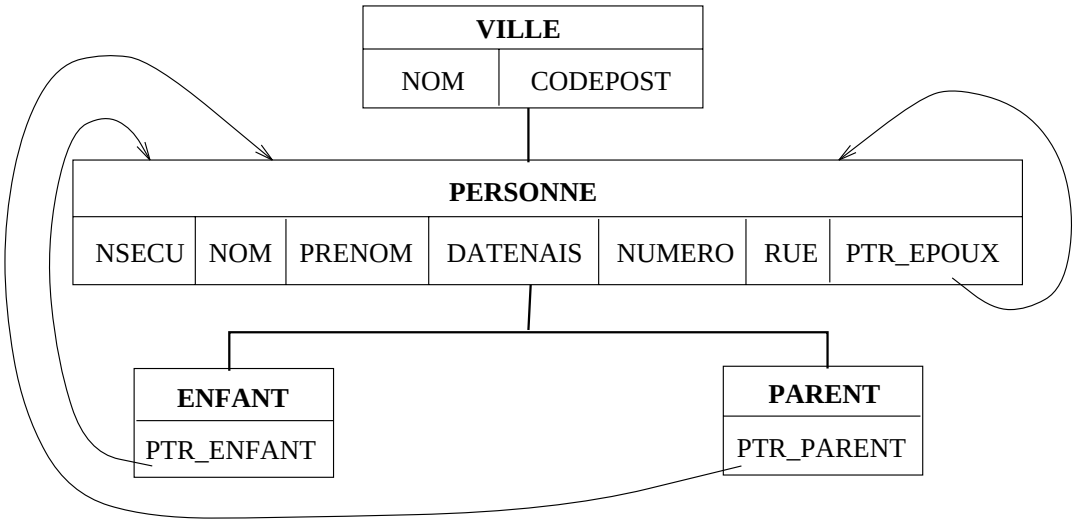


Fig. 2.3 : Représentation hiérarchique de la base d'exemple (avec références)

II.1.2 Les SGBD réseaux

A l'inverse du modèle hiérarchique, dans le modèle réseau ou CODASYL[40][43][103], toutes les entités représentées sont au même niveau et non décomposées hiérarchiquement. Pour décrire les entités, le modèle réseau offre aussi la notion d'enregistrement qui peut récursivement contenir d'autres enregistrements. En effet, un enregistrement peut avoir comme valeur d'un de ses attributs, un autre enregistrement. Un attribut peut non seulement être structuré en un nouvel enregistrement mais peut aussi être multivalué c'est à dire structuré sous forme d'un vecteur de valeurs atomiques ou structurées. Nous verrons que les modèles d'objets complexes [1][41][91] sont structurellement proches du modèle réseau. Une fois les entités définies, nous définissons les associations qui les relient. Le modèle permet exclusivement de traduire des associations 1:N grâce à la notion d'ensemble. Un tel ensemble appartient à une unique entité origine du lien 1:N et contient les entités associées à celle d'origine. L'exemple de la figure Fig. 2.1 peut se représenter par le schéma illustré dans la figure Fig. 2.4.

Nous utilisons donc essentiellement deux concepts dans une modélisation réseau : les enregistrements (à structure complexe) pour décrire les entités et les ensembles pour définir des liens 1:N entre les entités. Ce modèle peut alors être représenté par un système de types à deux niveaux où les types du premier ordre définissent les entités et les types du second ordre (construit à partir des types du premier ordre) définissent les associations entre entités. C'est un modèle à base de valeurs bien que plusieurs liens puissent aboutir à la même entité. Si la notion de référence n'existe pas au niveau du modèle conceptuel, elle est néanmoins nécessaire au niveau du modèle physique. La représentation de liens 1:1 et 1:N nécessite un ensemble alors que les liens M:N en nécessitent deux.

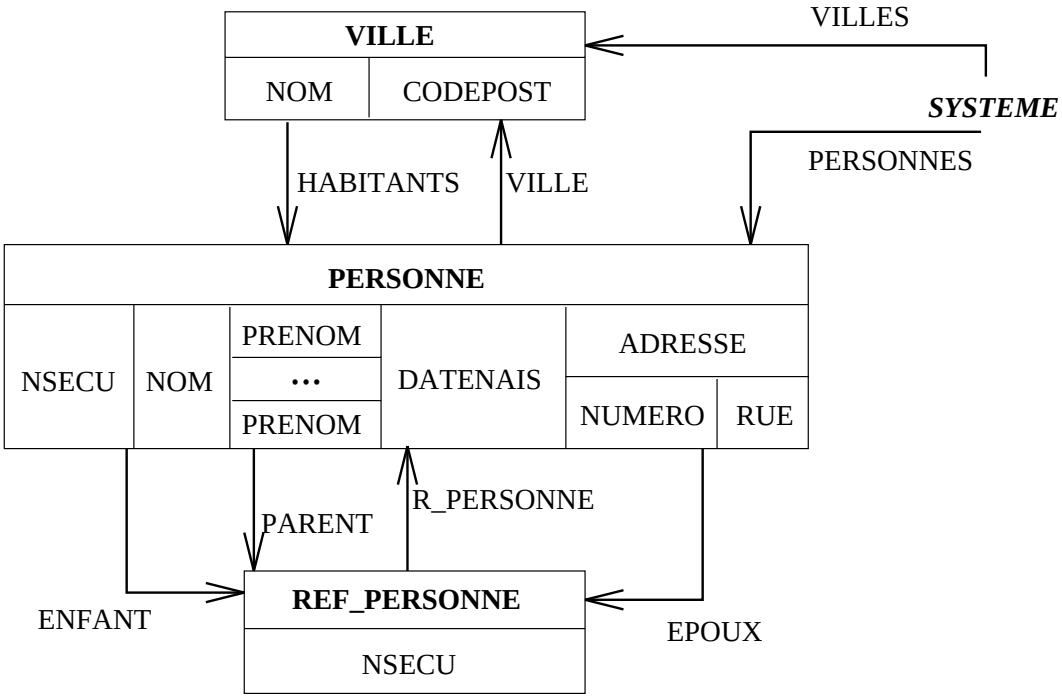


Fig. 2.4 : Schéma réseau de la base d'exemple

De plus, comme avec le modèle hiérarchique, le lien inverse doit être explicitement maintenu par l'utilisateur (les liens sont toujours unidirectionnels). C'est le cas dans l'exemple pour les liens VILLE/HABITANTS, PARENT/ENFANT ou encore EPOUX/EPOUX. Par ailleurs, le modèle réseau a une contrainte relativement génante : il ne permet pas de définir des liens d'un type d'entité vers lui−même. Pour cela, nous sommes obligés d'utiliser un artifice consistant à définir une entité qui simule une référence vers l'entité, notamment lorsque cette dernière veut s'auto−référencer. Cette technique est illustrée par les liens PARENT, ENFANT et EPOUX de l'exemple.

Les points d'entrée dans la base peuvent être n'importe quelle entité. Nous pouvons associer cette caractéristique à n'importe quel type d'entité ce qui fait que dans ce cas, le système maintient l'ensemble de toutes les entités de ce type (extension du type). Cette fonctionnalité est illustrée dans l'exemple par les liens "systèmes" VILLES et PERSONNES.

Le modèle physique de mémoire secondaire est, comme pour le modèle hiérarchique, divisé en deux parties qui sont le reflet du modèle conceptuel : le niveau des entités qui nécessitent des structures statiques (enregistrements et tableaux) et le niveau des liens qui nécessitent des structures dynamiques d'ensembles. De plus la notion d'adresse d'une entité est nécessaire car plusieurs ensembles représentant des liens peuvent contenir l'information de référence vers une même entité.

II.1.3 Les SGBD relationnels

Le modèle relationnel introduit par [33] est le plus simple des trois modèles décrits dans cette section. Il offre un seul concept permettant de modéliser à la fois les entités et les associations : la relation. Une relation est un ensemble de n−uplets dont les attributs ont tous des valeurs atomiques. C'est un modèle qui est uniquement à base de valeurs. Les liens entre les différentes entités modélisées s'appuient sur la notion de clé. L'exemple de la figure Fig. 2.1 se représente à l'aide des relations ci−dessous :

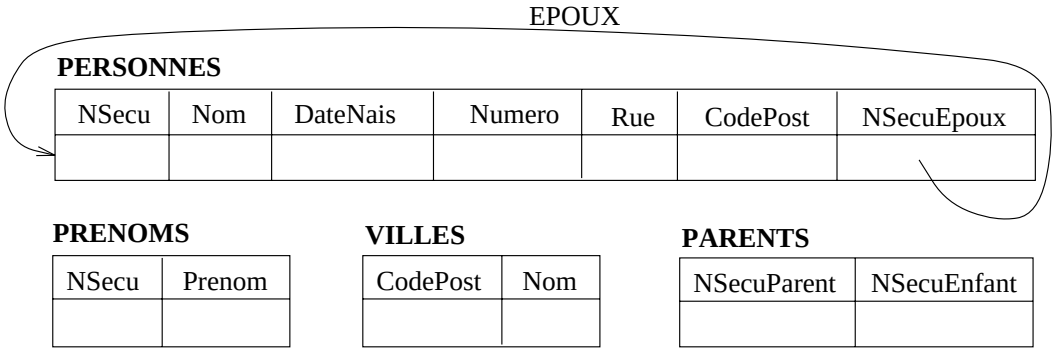


Fig. 2.5 : Schéma relationnel de la base d'exemple

Pour la représentation des entités dans des relations, nous pouvons remarquer que les structures imbriquées telles l'adresse d'une personne sont "applaties" dans une modélisation relationnelle. D'autre part, les attributs multivalués doivent être implantés dans une relation supplémentaire, à l'exemple des prénoms d'une personne. Nous perdons donc dans ces deux cas des informations sémantiques qui devraient apparaître dans la définition du type d'une personne (perte de la notion d'adresse et éclatement de la définition des prénoms dans une définition externe). Concernant les liens, si les liens 1:1 ou 1:N peuvent être rattachés directement à la définition d'une entité (ou de l'entité) mettant en œuvre le côté monovalué du lien, les liens N:M ne peuvent être définis que dans une relation supplémentaire aux deux relations associées. Par contre, cette modélisation a l'avantage de définir en même temps un lien et son symétrique dans une définition unique, ce qui évite d'avoir à le maintenir à l'utilisation.

Un des problèmes du modèle relationnel est la notion de clé pour la mise en œuvre des liens. Une clé identifie un n−uplet (correspondant à une entité) de manière discriminante à l'intérieur d'une relation. Une telle clé peut être composée de plusieurs attributs. Or la clé va apparaître dans toutes les associations auxquelles participe l'entité qu'elle identifie : nous avons alors une redondance des informations composant cette clé. Cela amène en général le concepteur à définir et à gérer une clé artificielle (numéro unique associé à chaque n−uplet d'une relation). Un autre problème, lié aussi aux liens, est celui qui fait qu'un lien peut subsister après qu'une des entités mises en œuvre dans ce lien n'existe plus. C'est donc normalement à l'utilisateur de gérer la cohérence des liens. La notion de contrainte référentielle présente dans la plupart des systèmes relationnels actuels permet d'alléger cette tâche. L'implantation de ce mécanisme est néanmoins coûteuse, à moins que le système ne gère des références physiques vers les n−uplets. De la même manière, traverser un lien est

très coûteux puisque cela nécessite une opération de jointure. Le principal reproche que nous pouvons faire au modèle relationnel est qu'il est souvent difficile de décrypter la sémantique d'un schéma. En effet, une grosse partie de cette sémantique est exprimée dans les requêtes manipulant les relations (c'est à dire dans le code !) : certaines jointures ne servent souvent qu'à recomposer les entités manipulées. Le système de types sous−jacent est donc relativement pauvre.

Le modèle physique de mémoire est complètement indépendant du modèle de données décrit par un schéma relationnel. Ce modèle physique reste aussi à base de valeurs dans le sens où nous n'avons pas en général de référence d'un n−uplet d'une relation vers un autre. L'intérêt du modèle relationnel réside dans sa simplicité et dans les langages et autres formalismes (algèbre, calcul du prédicat) qui sont offerts pour la manipulation des données et qui servent de fondement au modèle d'exécution.

II.1.4 Conclusions

La notion de modèle de données telle qu'elle est utilisée dans les modèles décrits ci−dessus sous−tend la notion de système de types qui est issue des langages de programmation. Une des caractéristiques importantes d'un système de types est sa capacité à modéliser plus ou moins facilement une situation donnée. La notion de type est présente dans tous les modèles ci−dessus mais elle n'est pas seulement intentionnelle. Elle est intégrée dans des concepts plus globaux permettant d'exprimer d'autres notions utiles dans les SGBD. Le concept décrivant le type d'une entité est notamment utilisé pour définir le contenant de ces entités.

Par ailleurs, comme les systèmes de types des langages de programmation décrivent le modèle de mémoire utilisé par l'environnement d'exécution du langage, les modèles de données déterminent le modèle mémoire implanté par les SGBD. Il y a deux niveaux de mémoire à mettre en œuvre (mémoire centrale / mémoire secondaire), ce qui explique que son système de types soit moins puissant que ceux des langages de programmation. Ces derniers travaillent en effet exclusivement en mémoire centrale. Nous pouvons donc dire que, pour ces différents types de SGBD, le modèle de données associé est grandement influencé par l'efficacité du modèle de mémoire qui peut être mis en œuvre (la réciproque étant vraie aussi), et surtout celle du modèle de stockage. Nous allons maintenant définir un modèle de mémoire, indépendant du niveau mémoire, nous permettant de comparer différents systèmes à objets.

II.2 Valeurs, objets et références

Comme nous avons pu l'observer dans la section II.1, les notions de valeur et de référence se retrouvent aussi bien dans le monde des langages de programmation, que dans celui des SGBD. Dans ce contexte, il est intéressant de savoir comment se place le concept d'objet introduit dans les langages ou systèmes orientés objet. Le but de cette partie est d'étudier les notions de valeur, d'objet et de référence dans les différents modèles de données

proposés par des SGBDOO ou des LPBD, et d'observer les similitudes avec les langages de programmation non persistants (notamment C++).

II.2.1 Définition d'un cadre d'analyse des modèles à objets

En général, le modèle de données des SGBD est destiné à fournir d'une part des structures de modélisation et d'autre part des méthodes de stockage et d'accès aux données correspondant à ces structures. Le prolongement de la définition d'un tel modèle est la définition de langages de requêtes de type ensembliste. Cet aspect n'est pas considéré dans cette étude. Cette partie s'intéresse à l'étude des modèles de données ou des systèmes de types du point de vue des modèles de mémoire à mettre en œuvre. C'est donc une analyse de l'aspect structurel induit par un modèle ou un système de types.

Pour cela, nous définissons un modèle de mémoire qui correspond à un espace d'instanciation de constantes. Ces constantes appartiennent à un espace qui est l'union de tous les domaines définis par les types du schéma de la base. Elles ont donc une existence réelle seulement lorsqu'elles sont instanciées. L'espace des noms défini par le schéma est aussi un élément important par rapport à l'espace d'instanciation ; il décore le graphe de la base dans cet espace. Ces différentes notions sont communes à tous les modèles de données et nous servent de cadre pour les comparer. Nous insistons sur la différence entre les modèles à base de valeurs seulement (modèle relationnel) et ceux qui supportent la notion de référence dont le rôle est réactualisé avec les modèles à objets.

La référence dans les modèles de données a essentiellement deux rôles : permettre le partage de données et permettre de modéliser des structures récursives. Ces deux motivations se retrouvent aussi dans les systèmes de types des langages de programmation mais le rôle de la référence est plus général. En effet, nous retrouvons implicitement la notion de référence derrière les mécanismes de désignation tels que les variables ou les paramètres. Par exemple, une variable globale permet à plusieurs procédures de partager des informations. Nous allons voir que nous pouvons en fait considérer les différentes notions de référence présentes dans différents modèles comme des adresses d'un espace d'adressage auxquelles est associée plus ou moins de sémantique.

II.2.1.1 L'espace des constantes (valeurs potentielles)

Dans les modèles formels présentés dans le contexte de bases de données à objets [1][2][17][74], nous avons la notion de schéma des données. Une base de données est définie alors comme une instance d'un schéma. Celui−ci décrit la structuration des données qui forment la base. Les données sont construites à partir de valeurs (constantes) qui appartiennent à des domaines de base (caractères, entiers, réels, etc...). L'union de ces domaines de base forme donc un espace de données potentielles V (cf. Fig. 2.6). Cet espace est infini et doit être, en partie, matérialisé par le système. Nous introduisons pour cela la notion d'espace d'instanciation.

II.2.1.2 L'espace d'instanciation

Les données n'ont une existance que lorsqu'elles sont instanciées dans la base. Nous assimilons ainsi la base de données à un espace d'adressage (espace d'instanciation) où toute

constante instanciée possède une adresse dans la base. Cette représentation se retrouve évidemment dans les langages de programmation avec la notion d'environnement d'exécution qui est un espace de désignation des données manipulées par un programme. Ces deux espaces se différencient par leurs caractéristiques de persistance ou de non persistance. C'est d'ailleurs cette différence qui est à l'origine de la plupart des problèmes liés à l'intégration de fonctionnalités des bases de données dans un langage général.

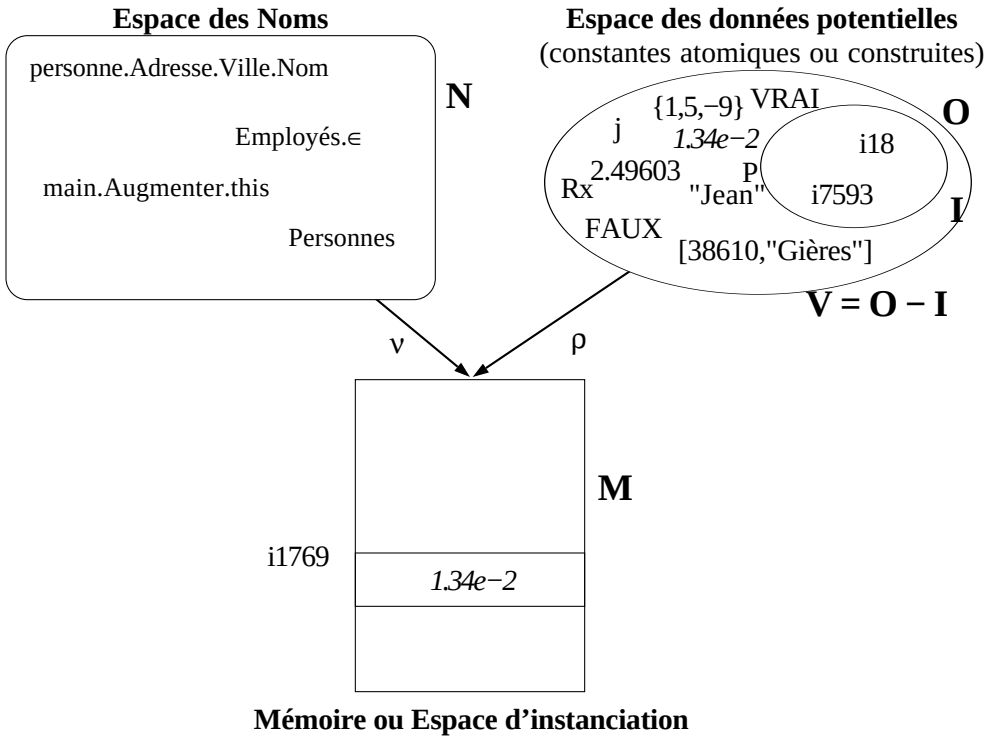
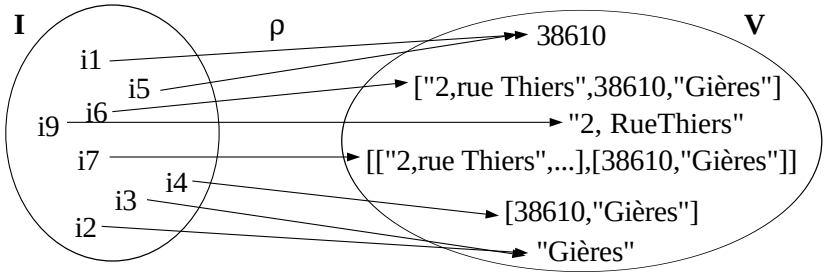


Fig. 2.6 : Vision structurelle d'une base de données

A partir de cet espace, nous définissons la notion de valeur instanciée qui est un couple (i, v) où v est la valeur associée à la référence i. La fonction $\rho : I \rightarrow V$ définit l'accès à une valeur référencée. Cette fonction est totale et monovaluée. Une même valeur de l'espace V peut donc être instanciée plusieurs fois dans l'espace M. Nous avons un exemple d'une telle instanciation dans la figure Fig. 2.7 où le graphe de la fonction ρ est le suivant :



Néanmoins, l'espace M ne peut à lui seul définir une base de données. C'est l'espace de noms N, défini par le schéma, qui détermine la sémantique que l'utilisateur veut attacher à cette base.

II.2.1.3 L'espace des noms

Dans l'espace de noms défini par un schéma, il est important d'en distinguer deux catégories : les noms permettant d'accéder à la structure d'une instance et ceux qui désignent les instances elles−mêmes. L'espace de noms n'est évidemment pas plat (n−uplet de valeurs atomiques). Les constructeurs permettent de structurer cet espace de telle manière qu'un nom peut devenir un chemin. Un chemin est une composition de noms comme "personne.Adresse.Ville.Nom", où "." est l'opérateur de composition. Nous pouvons en fait définir une base de données comme un graphe dont certains nœuds correspondent à des constructeurs et les autres à des valeurs instanciées (nœuds terminaux pour un modèle valué). Les arcs entre les nœuds du graphe sont caractérisés par un nom défini dans le schéma. Prenons par exemple deux données d'une base désignés par deux noms du schéma "ville" et "adresse" (ces deux noms correspondent à des variables désignant des instances) :

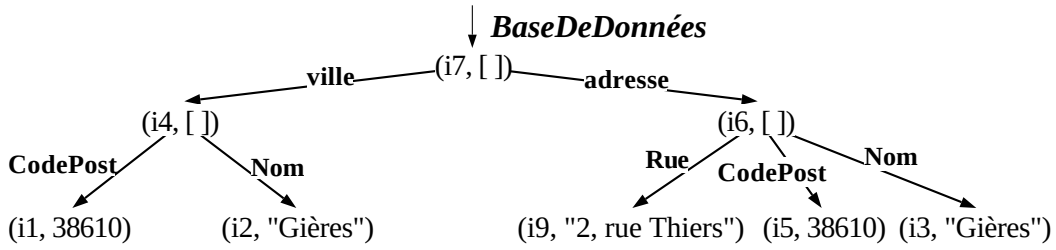


Fig. 2.7 : Exemple d'une base décorée par les noms du schéma

Ces deux données sont des valeurs qui ont une structure de n−uplet. Il est important de noter que chaque nom du graphe désigne une valeur unique instanciée. C'est en fait la liaison entre la référence d'une valeur instanciée et le nom qui est unique. Cette association est définie par la fonction $\nu : N \rightarrow I$. Nous avons par exemple ν(BaseDeDonnées.ville) = i4 ou ν(BaseDeDonnées.ville.Nom) = i2. La fonction ν est dans notre exemple mono−valuée et totale. Elle peut être multi−valuée notamment dans le cas du constructeur ensemble où tous les arcs vers les nœuds fils d'un nœud ensemble sont caractérisés par le même nom "$\in$". Par ailleurs, tout objet de la base possède un nom. Ainsi l'accès à une donnée de la base s'effectue obligatoirement par la fonction $\nu \circ \rho$.

L'espace de noms défini par un schéma est une arborescence. Cette arborescence est potentiellement infinie dans le cas des modèles à objets ou des systèmes de types autorisant les définitions récursives. La racine de l'espace de noms est dans tous les cas la base de données que nous nommons "BaseDeDonnées". La base de données est considérée comme un n−uplet dont les différents champs correspondant aux variables définissant les points d'entrée de la base (noms désignant des instances). A la limite, "BaseDeDonnées" peut être considéré comme le seul nom désignant une instance : l'instance "base de données".

II.2.1.4 Le concept d'objet

Dans la plupart des modèles à objets, nous distinguons la notion de valeur de celle d'objet abstrait. Cette distinction est généralement basée sur le fait qu'un objet abstrait est identifiable indépendamment de sa valeur grâce à la notion d'identificateur (oid). Ce dernier

est alors censé représenter l'objet en question de manière discriminante et constante durant toute son existence. Cette distinction est étudiée par C. Beeri dans [17]. En fait, l'identificateur d'objet n'est qu'un concept d'implantation de celui de référence auquel est attachée plus ou moins de sémantique.

D'une manière générale, le concept d'objet permet de désigner n'importe quelle entité. Dans le monde réel, nous désignons un objet en utilisant soit son nom comme par exemple "Paul" ou "Bull S.A.", soit en le montrant directement du doigt comme "ce chat" ou "cette voiture". De manière duale, le nom et la référence permettent de désigner n'importe quel objet d'une base de données. La notion d'objet utilisée ici est prise dans un sens général. Nous distinguons donc deux catégories d'objets dans une base : les valeurs et les objets abstraits.

Les valeurs

Dans les modèles distinguant objets abstraits et valeurs comme le modèle de O2[68], sont rangés dans la catégorie des valeurs, les objets des domaines de base tels que les entiers ou les réels. Cela se justifie parce que d'une part ce sont des abstractions connues et interprétées de manière universelle, et d'autre part parce qu'elles sont généralement directement supportées par le système. Ce n'est pas le cas pour un employé par exemple dont l'interprétation est dépendante de l'application. L'ensemble de ces valeurs de base est l'ensemble V défini dans la figure Fig. 2.6.

A partir de ces domaines de base, nous allons structurer les valeurs. Cette structuration s'appuie sur la notion de constructeur dont les plus répandus sont les tableaux, les n−uplets et, plus spécifiquement pour les bases de données, les ensembles. A partir de là, nous pouvons construire des valeurs dont la structure peut être une combinaison des différents constructeurs. En effet, les modèles d'objets complexes (constructeurs n−uplet et ensemble) ou les systèmes de types de langages impératifs usuels permettent de combiner ceux−ci de manière orthogonale (combinaison dans n'importe quel ordre avec une profondeur quelconque). Ce n'est pas le cas dans le modèle relationnel où le seul constructeur est la relation. Ce constructeur est d'ailleurs une composition du constructeur ensemble et du constructeur n−uplet. Les modèles "non première forme normale" (NF2)[91] sont une extension du modèle relationnel où le constructeur "relation" peut être utilisé récursivement : il est alors possible de définir des relations dont certains attributs sont eux−mêmes des relations et ainsi de suite.

L'exemple qui suit (cf. Fig. 2.8) montre le graphe d'une base relationnelle qui est une instance du schéma décrit dans la figure Fig. 2.5 (on a omis les références associées aux valeurs instanciées puisqu'elles sont déterminées par les noms). Pour des raisons de lisibilité de la figure, l'exemple ne montre qu'un seul n−uplet par relation c'est à dire un seul fils pour les nœuds "ensemble". Nous pouvons tout de même remarquer qu'une base relationnelle comme les bases supportant un modèle quelconque à base de valeurs est un arbre. Cet arbre a en plus une profondeur fixe dans le cas d'une base relationnelle.

Les valeurs, atomiques ou construites, permettent de structurer les objets à modéliser. Par contre, lorsqu'il est nécessaire de définir des associations entre ces objets, la solution

dans les modèles à base de valeurs est de fixer une clé dans les objets à associer. Cette clé est alors utilisée dans l'objet référençant (clé étrangère) pour désigner l'objet référencé. Le problème est qu'une telle clé n'identifie pas une valeur de la base de manière discriminante ; seule la valeur prise globalement s'auto−identifie. Si nous suivons ce raisonnement, c'est alors la valeur en entier qu'il faut mettre dans l'objet référençant ce qui mène à deux problèmes très importants :

- nous avons une redondance importante des données puisque nous instancions un objet (valeur) à chaque fois que nous avons besoin de le référencer et,
- chaque instance ainsi répétée représente le même objet ce qui nécessite de mettre à jour toutes ces instances lorsque l'une d'entre elles est modifiée.

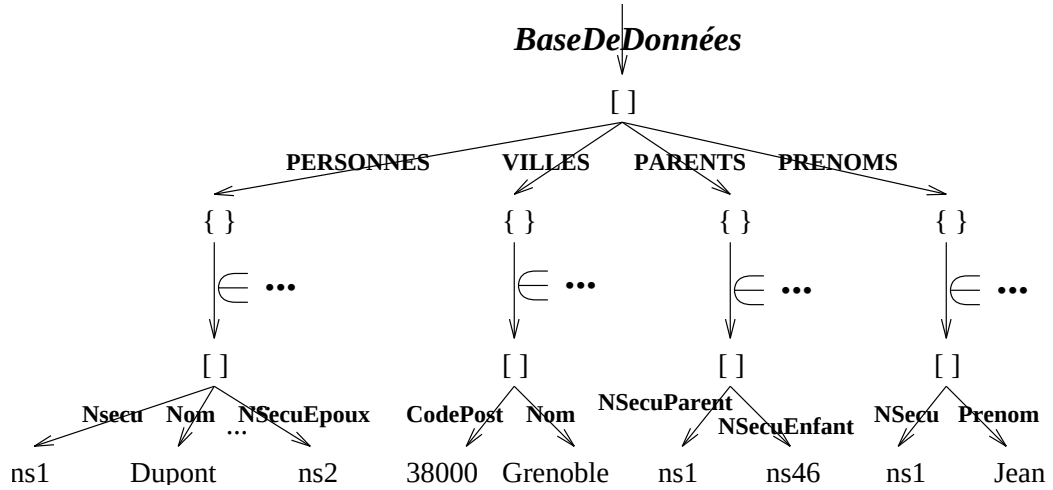


Fig. 2.8 : Graphe d'une base relationnelle

Cette situation n'est pas satisfaisante. Il est nécessaire de pouvoir référencer un objet ce qui conduit à n'avoir qu'une seule instance de celui−ci et à résoudre du même coup les deux problèmes précédents. La notion d'objet abstrait telle qu'elle est définie dans la suite comble le besoin de référençage même si ce n'est pas sa seule motivation.

Les objets abstraits

Jusqu'ici, nous avons parlé uniquement de valeurs dont une caractéristique est qu'elles s'identifient elles−mêmes dans l'espace V. Nous définissons à présent la notion d'objet abstrait comme une entité dont la désignation ne change pas au cours de sa vie et à laquelle nous associons un état qui lui peut évoluer. Cet état peut être considéré comme une valeur instanciée. Pour désigner un objet abstrait, il faut un couple (i, i_obj) où i, i_obj $\in$ I, ce qui permet d'accéder à son état défini comme (i_obj, état). Pour pouvoir définir des objets abstraits, il est nécessaire d'étendre la fonction d'instanciation en ayant $\rho : I \rightarrow I \cup V$ (= O). En effet, nous avons dorénavant des valeurs qui sont les identificateurs servant de relais d'indirection vers une autre valeur.

La notion d'objet telle qu'elle est définie ici ne nécessite pas la manipulation de références. En effet, le rôle de l'objet abstrait est de fournir un couple (i, o) où i doit permettre

de l'identifier de manière discriminante et ce de façon constante tout au long de sa vie[59]. o est l'état associé à l'objet abstrait que nous pouvons assimiler à une valeur instanciée. Dorénavant, l'accès à la valeur associée à un objet abstrait se fait par la fonction $\nu \circ \rho \circ \rho$. Nous avons donc introduit une indirection supplémentaire dans le chemin d'accès à la valeur : cette indirection est instanciée par une référence dans l'espace M.

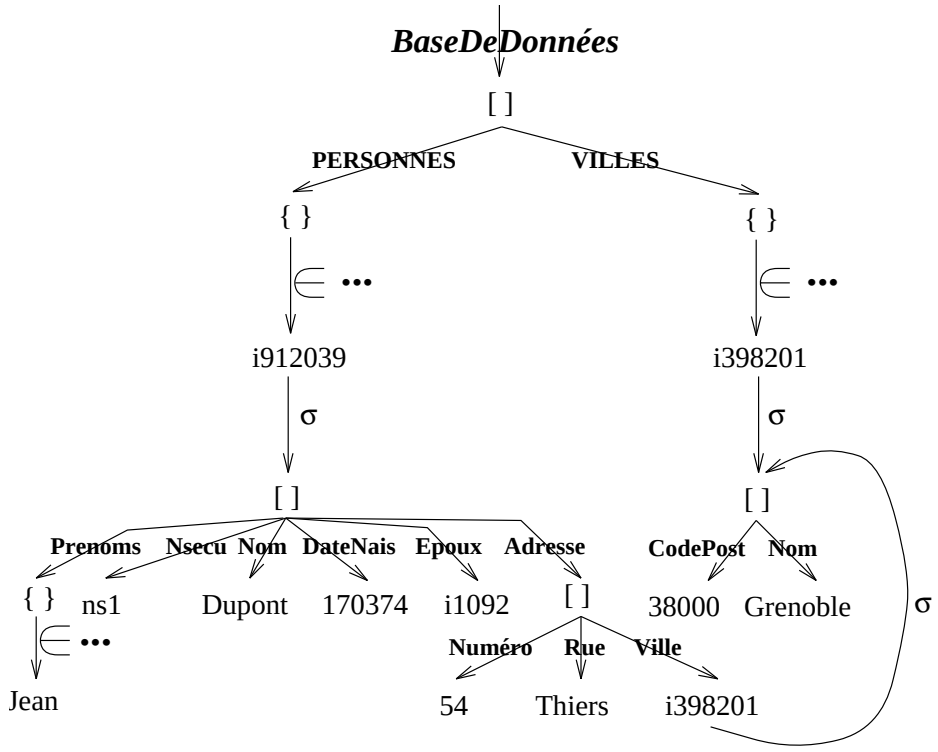


Fig. 2.9 : Graphe d'une base à objets

La méthode d'accès à un objet abstrait étant différente de la méthode d'accès à un objet valeur, la distinction doit être faite au niveau du schéma ou du système de types. De plus, comme l'identificateur de l'objet doit être unique dans l'espace M, c'est au système de le gérer ; l'utilisateur ne doit donc pas pouvoir manipuler l'identificateur comme n'importe quelle autre valeur. Il n'y a pas de constante "identificateur" manipulable par l'utilisateur. La figure Fig. 2.9 montre que les identificateurs n'ont pas à être manipulés explicitement puisque l'accès à la valeur d'un objet abstrait s'effectue à travers le nom générique σ désignant l'état. En fait, σ peut aussi être vue comme une fonction de déréférençage du relai d'indirection vers l'objet désigné.

Le modèle offre deux catégories d'objets sans modifier en aucune façon les opérations de manipulation (affectations, comparaisons, etc...) telles qu'elles existent dans un modèle à base de valeurs. Dans le cas d'un objet abstrait, ces opérations peuvent être faites soit au niveau de l'objet (représenté par son identificateur), soit au niveau des valeurs associées. Par exemple, comparer deux objets revient à comparer leurs identificateurs et, s'il s'agit de leurs valeurs associées, c'est la valeur qui est explicitement spécifiée.

Les objets abstraits offrent donc la possibilité de partager des données et de définir des cycles dans le graphe de la base de données. La notion de référence dans le modèle est donc utilisée pour mettre en œuvre le nom générique σ ce qui permet de manipuler tous les objets de manière symbolique (à travers les noms définis par le schéma).

Ce n'est pas le cas d'un modèle où nous aurions introduit la manipulation des références. En effet, pour résoudre les problèmes de partage et de cycles, nous pouvons introduire dans un modèle à base de valeurs la possibilité de récupérer la référence d'une valeur instanciée. C'est notamment le cas pour les langages C et C++ qui permettent la manipulation des références.

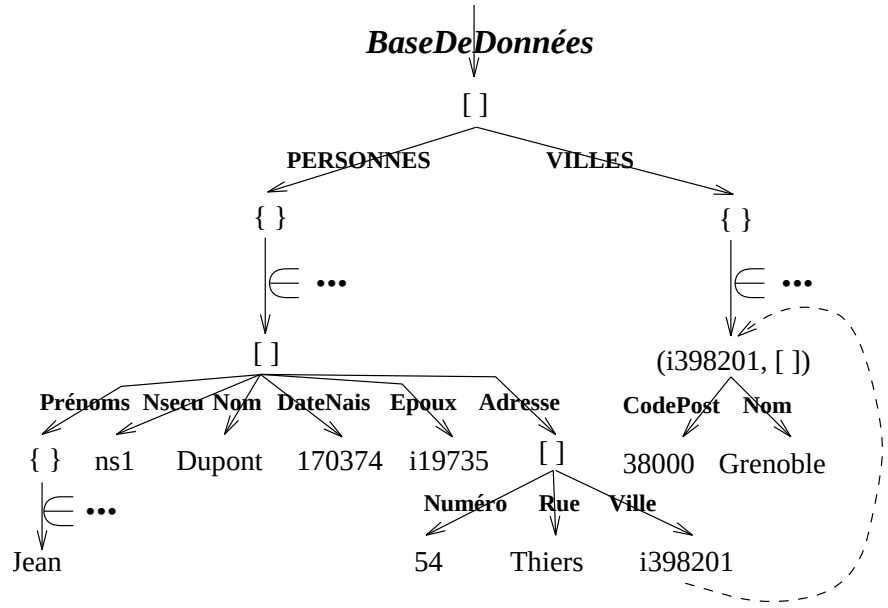


Fig. 2.10 : Graphe d'une base de valeurs avec manipulation de références

La figure Fig. 2.10 montre le graphe d'une base constituée de valeurs dans laquelle nous avons utilisé des références. Dans ce cas, les références sont des valeurs de base du même niveau que les entiers ou les réels et elles sont manipulées explicitement par l'utilisateur. Pour pouvoir définir de telles bases, il faut donc que l'utilisateur ait accès aux fonctions ν et ρ. Nous perdons ainsi l'intérêt d'avoir exclusivement une manipulation symbolique des objets de la base à travers leur nom. Par exemple, l'accès par l'utilisateur au nom d'une ville à partir d'une adresse se fait dans le cas de la figure Fig. 2.10 par "ρ(Adresse.Ville).Nom". Dans le cas de la figure Fig. 2.9, l'accès est complètement symbolique : "Adresse.Ville.σ.Nom". De la même manière, une affectation dans le premier cas s'exprime par "Adresse.Ville ← ρ(VILLES.∈)" (cf. Fig. 2.10) et dans l'autre cas par "Adresse.Ville ← Ville.∈ ".

Le cadre défini dans cette partie s'attache exclusivement à l'étude de l'aspect structurel des modèles de données à objet. La notion principale qui différencie les modèles à base de valeurs des modèles objets est la référence. Elle est utilisée pour dissocier l'identification

d'un objet de sa valeur définissant ainsi des objets abstraits. Nous parlons aussi dans ces modèles d'identificateur d'objet (oid) et d'état d'un objet. Dans la suite, nous proposons une étude de ces différentes notions dans différents modèles de systèmes existants.

II.2.2 Le modèle de données d'Orion

Orion[60] est un SGBD orienté objet développé au MCC à Austin (USA). Il s'agit d'une extension du langage Lisp, dans lequel sont introduits des concepts objets et base de données à travers la notion de classe. Dans le modèle d'Orion, tout est objet ; nous n'avons la notion de valeur qu'au niveau des attributs d'un objet. De plus, la classe apporte aussi des fonctionnalités base de données. En effet, comme la relation, la classe définit à la fois un type d'objet et désigne l'ensemble des instances de cette classe. Le schéma de l'exemple de la figure Fig. 2.1 s'exprime de la manière suivante (le graphe d'une base instanciant ce schéma se rapprocherait de celui de la figure Fig. 2.9) :

```
(make-class PERSONNE
    :attributes
        ( Nsecu :domain string )
        ( Nom :domain string )
        ( Prenoms :domain PRENOM )
        ( DateNais :domain date )
        ( Adresse :domain ADRESSE :composite True )
        ( Epoux :domain PERSONNE )
        ( Enfants :domain ENFANT ))
(make-class VILLE
    :attributes
        ( Nom :domain string )
        ( CodePost :domain integer ))
(make-class PRENOM
    :attributes
        ( Prenom :domain string )
        ( Suivant :domain PRENOM ))
(make-class ADRESSE
    :attributes
        ( Numero :domain integer )
        ( Rue :domain string )
        ( Ville :domain VILLE ))
(make-class ENFANT
    :attributes
        ( Enfant :domain PERSONNE )
        ( Suivant :domain ENFANT ))
```

D'un point de vue structurel, le modèle d'Orion est très proche du modèle relationnel puisque d'une part le constructeur ensemble est mis en œuvre seulement au niveau de la classe et d'autre part l'état d'un objet a une structure de n−uplet dont les attributs sont soit des valeurs de base, soit des objets abstraits. Le seul moyen de représenter des attributs multivalués est alors de gérer explicitement des structures de listes comme cela est fait pour les champs "Prenoms" et "Enfants" de la classe Personne.

De plus, comme nous ne pouvons pas non plus structurer un attribut en un autre n−uplet, pour pouvoir définir l'abstraction de l'adresse dans PERSONNE, nous devons définir un objet

de classe ADRESSE. Etant donné que nous considérons l'adresse comme propre à chaque personne, Orion offre la possibilité de définir des liens dit "composite" entre objet. Cela signifie que l'objet Adresse ne peut être partagé par aucun autre objet et qu'il existe une dépendance existentielle entre les deux objets (si un objet PERSONNE est détruit, l'objet "composite" ADRESSE l'est aussi). La sémantique de ces deux types de liens ne peut être mise en œuvre qu'au niveau de la référence ou de l'identificateur de l'objet car c'est une contrainte dynamique associée à l'objet et non à sa classe.

Par ailleurs, toutes les instances d'une classe Orion sont persistantes. Nous avons donc deux espaces d'instanciation bien séparés : l'espace temporaire des données Lisp (essentiellement composé de listes) et l'espace des objets persistants décrit par les classes. Nous constatons que le dysfonctionnement au niveau du système de type n'a pas été supprimé. Nous avons deux systèmes de types qui partagent néanmoins leurs domaines de base, ce qui réduit le dysfonctionnement. Les nouveaux concepts objet et base de données ne sont donc pas orthogonaux aux autres concepts du langage initial.

II.2.3 EXTRA : un modèle de données pour EXODUS

EXODUS[27] est un SGBD extensible développé à l'université de Wisconsin (USA). Il s'agit d'une boite à outils devant permettre d'implanter des SGBD et ce pour n'importe quel modèle de données. Pour le valider, un modèle de données, nommé EXTRA, et un langage de requêtes, nommé EXCESS, sont proposés dans [28]. C'est un modèle d'objets complexes assez proche de ceux proposés dans Galileo[8] ou dans Fad[15], dans lequel tous les objets sont des valeurs. L'introduction d'un constructeur **`ref`** permet de manipuler explicitement les références aux valeurs de la base. La notion de type permet de définir structurellement ces valeurs ; une base instanciant un schéma EXTRA se rapproche donc de celle décrite dans la figure Fig. 2.10. Le schéma de l'exemple de "personne" se résume alors aux définitions EXTRA suivantes :

```
define type personne:
    (
    Nsecu: char[16],
    Nom: char[20],
    Prenoms: {char[15]},
    DateNais: date,
    Adresse: (
        Numéro: int4,
        Rue: char[],
        Ville: ref ville
        ),
    Epoux: ref personne,
    Enfants: {ref personne}
    )
define type ville:
    (
    Nom: char[40],
    CodePost: int4
```

```
    )
create PERSONNES: {own ref personne}
create VILLES: {own ref ville}
```

Seules les valeurs générées à partir des types du schéma peuvent être référencées. Celles−ci sont introduites par le constructeur **own** dans la définition où leur type intervient. De plus, ces valeurs sont divisées en deux catégories : les valeurs référençables et celles qui ne le sont pas. Les valeurs référençables sont introduites par **own ref** alors que les autres le sont par **own** seulement. Le constructeur **own** introduit dans tous les cas une dépendance existentielle pour la valeur ainsi désignée.

La manipulation des valeurs de la base rend transparente la notion de référence. En effet, l'accès à une valeur par un nom qui la désigne directement ou par un nom qui désigne une référence vers cette valeur se fait de la même manière pour l'utilisateur. Le système infère automatiquement, par rapport au schéma, le fait qu'il accède à cette valeur par la fonction $\nu \circ \rho$ ou $\nu \circ \rho \circ \rho$ (pour les valeurs référencées). Par exemple, si nous supposons que "pers" est un nom qui désigne une valeur de type "personne", nous accédons de la même manière "pers.Nom" et "pers.Epoux.Nom" alors que "pers.Epoux" désigne une référence vers une valeur de type "personne".

Néanmoins, la différence étant explicite dans le schéma, elle est prise en compte lors de l'opération d'affectation. Nous en distinguons deux types : l'affectation de valeur (**copy to** ...) et l'affectation d'une référence à une valeur (**insert into** ...). Par contre, nous pouvons étendre les domaines de base avec les ADT (types de données abstraits). Les ADT sont des classes persistantes définies à partir du langage E[87] (extension de C++). Un exemple de leur utilisation est le champ "DateNais" dans le type "personne" dont le type Date est un ADT. Un ADT définit donc une valeur encapsulée par différentes méthodes qui lui sont attachées.

Toutes les valeurs créées sont persistantes. Nous avons donc un seul espace d'instanciation dont les données sont manipulées à travers un langage de requêtes (EXCESS) du même niveau qu'un langage comme SQL. Ce langage de manipulation n'est pas complet ; nous n'avons pas dans ce système une intégration entre un langage général et les aspects base de données.

II.2.4 Le modèle de données de O2

O2[81] est un SGBD orienté objet développé par le GIP−Altaïr à Paris (France). Son but est d'offrir un système multi−langages permettant à ces derniers de manipuler des objets, notamment persistants, à travers un modèle commun. Le modèle O2[68] distingue deux catégories de données : les valeurs et les objets (objets abstraits définis dans II.2.1.4). Le nommage des instances (**create name** ...) est séparé du nommage des types comme dans EXTRA. Il est d'ailleurs possible de nommer les types décrivant des valeurs (**create type** ...) aussi bien que ceux qui décrivent des objets abstraits (**create class** ...) comme le montre la représentation O2[79] de l'exemple des personnes de la figure Fig. 2.1 :

```
create class personne
type tuple
    (
    Nsecu : string,
    Nom : string,
    Prenoms : list (string),
    DateNais : integer,
    Adresse : adresse,
    Epoux : personne,
    Enfants : set (personne)
    )
end
create class ville
type tuple
    (
    Nom : string,
    CodePost : integer
    )
end
create type adresse :
tuple
    (
    Numéro : integer,
    Rue : string,
    Ville : ville
    )
create name PERSONNES: set (personne)
create name VILLES: set (ville)
```

O2 offre donc un modèle d'objets complexes faisant intervenir de manière orthogonale les constructeurs n−uplet (**`tuple`**), ensemble (**`set`**) et séquence (**`list`**). Nous n'avons pas explicitement la notion de référence et le partage de données se fait exclusivement à travers des objets abstraits.

Le système permet de créer des objets persistants ainsi que des objets temporaires. Il faut alors distinguer deux types de références suivant le statut de l'objet instancié qui engendre en fait deux espaces d'instanciation différents. Or, la notion d'objet abstrait impliquant qu'une référence peut être instanciée dans d'autres objets, il se pose le problème des références inter−espace. En effet, une référence dans l'espace persistant est valide d'une exécution d'un programme à une autre. Ce n'est pas le cas des références de l'espace temporaire. Le problème est résolu dans O2 en déplaçant tout objet abstrait temporaire dans l'espace persistant si la référence qui l'identifie est instanciée dans l'espace persistant.

Le modèle de O2 est intégré dans un langage de programmation usuel tel que C, Lisp ou Basic. Pour cela, le système de type défini par le modèle est ajouté au système de type du langage d'immersion. Cette intégration n'est pas orthogonale. Le dysfonctionnement au niveau du système de type n'est donc pas résolu dans les différents langages associés à O2 tels que O2C. Néanmoins, ce dysfonctionnement est moins sensible car le système de types pour la définition des données persistantes est au moins aussi puissant que celui du langage d'immersion même s'il ne partage pas les domaines de base. Des coopérations sont tout de

même possibles comme par exemple une affectation d'une entier C (int) dans un entier O2 (integer).

II.2.5 Le modèle de données de Guide

Guide[64] est un système orienté objet distribué développé par l'unité mixte Bull−Imag à Grenoble (France). Il sert de support à un langage orienté objet portant le même nom dédié à la construction d'applications réparties. Le système de type permet essentiellement de définir des objets abstraits qui sont des instances des classes Guide. La notion de type présente dans Guide correspond à une interface d'accès aux objets. Ces objets peuvent appartenir à différentes classes si celles−ci sont conformes à l'interface proposée par le type.

L'état associé à un objet Guide est une valeur qui peut être construite[78]. Le langage offre deux constructeurs : le n−uplet (**record**) et le tableau (**array**). Ils peuvent être combinés de manière orthogonale. Néanmoins, nous n'avons pas la puissance d'un modèle d'objets complexes, le tableau n'offrant pas la dynamicité du constructeur ensemble. Il faut alors comme pour Orion, gérer explicitement des listes chainées. Une modélisation possible de l'exemple des personnes en Guide est la suivante :

```
class personne implements personne is
    Nsecu : String[16];
    Nom : String[20];
    Prénoms : Array[3] of String[15];
    DateNais : Integer;
    Adresse : Record
                  Numéro : Integer;
                  Rue : String[80];
                  Ville : ref ville;
              end
    Epoux : ref personne;
    Enfants : ref enfant;
end personne
class enfant implements enfant is
    Enfant : ref personne;
    Suivant : ref enfant;
end enfant
class ville implements ville is
    Nom : String[40];
    CodePost : Integer;
end ville
```

D'un point de vue modèle, tous les objets Guide sont persistants. En fait, ils ne persistent réellement que lorsqu'ils sont accessibles à partir d'un nom. Or nous avons à faire ici à un langage de programmation ce qui permet de distinguer deux partitions dans l'espace de noms : les noms qui désignent des données de la base, et les noms qui désignent des entités dans l'environnement d'exécution. Nous pouvons par exemple répertorier dans cette deuxième partition, qui n'existe que dans le cas d'un langage de programmation, les noms désignant des variables locales d'une procédure.

Cette seconde catégorie de noms peut être déterminée statiquement par le schéma qui contient dans ce cas les définitions des traitements (procédures ou méthodes). Ce n'est plus

vrai lorsque le langage supporte la récursivité (empilage potentiellement infini d'environnements). L'environnement d'exécution comporte alors une pile impliquant la création de nouveaux noms à chaque nouvel appel de procédure. Nous avons donc une deuxième racine de nommage (la première étant la base de données nommée "BaseDeDonnées") qui est l'environnement d'exécution que nous nommons "Environ". Dans le cas d'un langage classique, nous lui attachons généralement deux fils : un nom désignant un n−uplet dont chaque attribut est une variable globale de l'application (nous l'appellons "Global"), et le nom désignant la pile d'exécution (nous l'appellons "Pile").

Dans Guide, nous pouvons ainsi créer des objets qui sont désignés par des noms de l'environnement d'exécution. Ils ont alors la durée de vie de ce nom, c'est à dire au maximum celle de l'exécution en cours, à moins qu'un autre nom de la base ne permette de les désigner avant la fin de cette exécution. Nous montrons un exemple de ce type de nom dans la figure Fig. 2.6 où "Environ.Pile.main.Augmenter.this" désigne le paramètre contenant l'objet sur lequel s'applique la méthode "Augmenter" qui a été appelée dans la procédure "main" correspondant au point d'entrée du programme.

En fin de compte, tous les objets Guide qui sont créés sont potentiellement persistants et deviennent réellement persistants (ils subsistent à l'exécution du programme qui les a créés) uniquement s'ils sont attachés (désignés) par un nom de la base. De la même manière que les systèmes décrits précédemment, Guide offre un service de nommage permettant de créer des points d'entrée vers les objets de la base. Par contre, ces noms ne font pas partie du schéma ce qui est une divergence remarquable. En effet, ces noms ne sont pas typés (alors que Guide offre un typage fort) ce qui implique le besoin de mettre en œuvre de la vérification de type à l'exécution lors de l'utilisation de ces noms.

II.2.6 L'approche C++ persistant

Le langage C++ tend à devenir le standard pour la programmation orientée objet même si d'autres langages tels que Smalltalk ou Eiffel[76] occupent une place importante pour certains types d'applications. A partir de ce constat, la recherche concernant les langages persistants a été très active durant la dernière décennie, de nombreux travaux ont été menés pour rendre C++ persistant et, au−delà, essayer d'en faire un véritable SGBD (ou LPBD dans ce contexte).

Le principal problème lorsque nous voulons ajouter la persistance dans un langage de programmation est de déterminer le comportement du système de types vis à vis de la nouvelle fonctionnalité. L'idéal est alors d'introduire la persistance de façon orthogonale au système de types, c'est à dire que des données de n'importe quel type peuvent devenir persistantes. Le premier langage persistant à avoir rempli cette condition est le langage PS−Algol[12]. Cette caractéristique est très importante et elle procure deux avantages à l'utilisateur :

- la mise en œuvre de l'extension à la persistance demande peu d'ajouts au langage et aucun des anciens concepts n'est perturbé par la nouvelle fonctionnalité et
- le code existant peut être facilement réutilisé et transformé pour gérer les données persistantes.

Parmi les C++ persistants existant, les deux approches d'intégration de la persistance, orthogonale ou non, sont utilisées. Nous allons étudier dans la suite les conséquences liées à ce choix d'intégration sur le modèle de données mis en œuvre par le système de types. Le modèle de données de C++ est à la fois proche d'un modèle à base de valeurs plus des références illustré par la figure Fig. 2.10 et d'un modèle à objets comme illustré dans la figure Fig. 2.9. L'introduction de la persistance dans ce contexte n'est donc pas triviale.

II.2.6.1 Les C++ persistants non orthogonaux

La plupart des C++ persistants existant introduisent la persistance de manière non orthogonale. C'est le cas de E[87], O++[6], Ontos[83] ou Versant[105]. Néanmoins, chacune de ces extensions utilise une méthode différente. Dans tous les cas, nous avons une séparation explicite, induite par le système de type, de l'espace des données persistantes (et potentiellement persistantes) de celui des données temporaires. Au niveau de l'instanciation, cela implique donc que nous avons deux catégories de références correspondant aux deux espaces d'instances.

Dans C++, la référence est implantée directement sous la forme d'une adresse virtuelle. Cela signifie que si les objets temporaires peuvent être manipulés comme ils l'étaient auparavant, ce n'est pas le cas pour les objets persistants. Cela pose principalement un problème par rapport aux références qui pouvaient être manipulées explicitement par l'utilisateur (avec notamment la possibilité d'incrémenter ou de décrémenter celles−ci). Le nouveau type de références va devoir supporter les mêmes opérations.

Par ailleurs, il se pose aussi le problème de la coopération entre les deux espaces. Il est généralement possible d'utiliser des références persistantes dans des objets temporaires mais pas l'inverse. De plus, O++ permet de définir des parties temporaires dans des objets persistants. C'est alors à la charge de l'utilisateur d'assurer la cohérence de ces parties. L'intérêt est que les parties non persistantes ne sont pas sauvegardées d'une session à l'autre.

Introduction de la persistance

1. Dans E, la persistance est liée à un système de types (les types persistants exprimés par **`db...`**) qui est un système totalement dual au système de types d'origine de C++ (**`int`** ↔ **`dbint`**, **`struct`** ↔ **`dbstruct`**, **`class`** ↔ **`dbclass`**, etc...). Toutes les données déclarées comme ayant un type persistant sont potentiellement persistantes. Elles ne deviennent réellement persistantes que lorsqu'elles sont désignées par un nom persistant ou qu'elles sont insérées dans une dbclass "collection" (cf. service de nommage ci−dessous). La collaboration entre les deux systèmes de types est unilatérale : seuls des types de données temporaires peuvent utiliser des types persistants, mais pas le contraire.
2. Le cas de O++ est relativement proche d'une persistance orthogonale. En effet, la persistance n'est pas aussi fortement liée au système de types qui, dans le cas présent, est étendu avec une seule nouvelle notion : la référence persistante. Cette référence permet alors de désigner les objets persistants et de les manipuler dans le langage comme n'importe quel objet manipulé par une référence temporaire. Le statut de persistance est ainsi associé à un objet à sa création, l'utilisateur indiquant

à ce moment là s'il veut créer le nouvel objet dans l'espace temporaire (allocation usuelle de C++ avec **`new`**) ou dans l'espace persistant (allocation avec **`pnew`**). Les deux catégories de références ont tout de même des rôles similaires. O++ offre ainsi la notion de référence duale permettant par exemple de définir une procédure dont les paramètres peuvent être des références temporaires ou persistantes.

3. L'introduction de la persistance dans Ontos se fait par l'intermédiaire du mécanisme d'héritage (sous−classe de la classe "Object") ce qui résout en partie le problème de coopération entre des objets de deux espaces différents (règlé par le mécanisme de typage). La persistance est donc définie exclusivement au niveau des classes. La manipulation des objets persistants dans le langage se fait alors à travers la notion de référence persistante. Le choix de Ontos est, de ce point de vue, dramatique dans le contexte de C++ car ces références ne sont jamais typées. Le type référence persistante est en fait une classe Ontos prédéfinie qui définit un certain nombre de méthodes permettant de les manipuler. L'une d'entre elles permet notamment de charger l'objet désigné en mémoire virtuelle et de rendre son adresse virtuelle. Cette référence C++ usuelle est alors typée explicitement par l'utilisateur par une méthode de coercition (technique de "cast") même si l'objet récupéré n'est pas conforme à ce type.
4. La technique utilisée par Versant est très proche de celle de Ontos. Les objets potentiellement persistants doivent être instances d'une classe C++ héritant de la classe "PObject" qui transmet la propriété de persistance. Ils deviennent persistants à leur création comme dans le cas de O++ (**`new`** / **`Persistent new`**). La manipulation des objets persistants dans le langage se fait comme pour les objets temporaires à travers les mêmes références (objet chargé en mémoire virtuelle). Par contre, les références persistantes au niveau des attributs d'un objet persistant sont différentes : elles sont définies par la notion de lien (type **`Link<`***ClasseName*`>`**). De la même manière que Ontos, lorsqu'on récupère un objet dans la base à partir d'un nom (notion de "container"), il a perdu son type ; tous les objets persistants utilisés dans l'application sont du type "PObject". Il faut alors appliquer une cœrcition du type sachant qu'il est possible de connaître le nom du type de création de l'objet en appliquant la méthode "className" de "PObject".

Service de nommage

1. La technique de nommage des instances utilisée dans E est une extension naturelle de C++. En effet, les noms donnant accès aux objets de la base sont des variables globales C++ qui sont déclarées persistantes. C'est la seule des quatre évolutions de C++ décrites ici qui permet de rendre persistantes des données de n'importe quel type C++ (pas seulement les objets de classes).
2. Dans O++, il y a un premier niveau de noms qui est implicite. Ces noms désignent des groupements ("cluster") d'objets qui sont en fait les extensions des classes (ensemble de tous les objets générés par la classe) pour lesquelles nous voulons instancier des objets persistants. Ces groupements portent le même nom que la classe de leurs objets et sont créés par l'utilisateur. Il est alors possible de

partitionner ces groupements ("subcluster") en créant des sous−groupements nommés à l'intérieur de ceux−ci. L'affectation à un nom se fait dans tous les cas à la création de l'objet persistant en l'affectant à un groupement ou à un sous−groupement.

3. Ontos permet de créer des noms d'instance qui désignent soit des objets persistants (de type "Object *") soit des groupements ("cluster") d'objets persistants (de type "List *"). Pour pouvoir activer ces objets en mémoire virtuelle, l'utilisateur doit appeler des procédures de chargement explicites.
4. Le service de nommage utilisé dans Versant définit la notion de contenants d'objets (classe "Container"). Ceux−ci peuvent contenir des objets de n'importe quelle classe C++ héritant de la classe "PObject". Lorsqu'un utilisateur crée un objet persistant, il est automatiquement inséré dans un "container" par défaut. Il peut ensuite migrer entre les différents "containers" existant. La récupération d'un objet d'un "container" se fait par un indice car un "container" est un tableau de taille variable.

La conclusion générale que nous pouvons tirer de l'approche non orthogonale est que, s'agissant des données persistantes, leur manipulation est souvent loin de ce qu'il est possible de faire normalement dans C++. Ceci est d'autant plus grave pour l'utilisateur, même si ces problèmes sont souvent signalés par les concepteurs des systèmes (par exemple Versant), que certaines manipulations (notamment sur les pointeurs) de ces objets qui sont syntaxiquement et sémantiquement acceptées par le compilateur peuvent conduire à des erreurs. Par ailleurs, certaines extensions ne sont pas homogènes avec la philosophie du langage C++ ; c'est le cas du typage des objets persistants récupérés dans la base qui est parfois laissé à la charge des utilisateurs.

II.2.6.2 Les C++ persistants orthogonaux

Il n'existe en fait qu'une seule extension du langage C++ dans laquelle la persistance est réellement orthogonale au système de types. Il s'agit du système ObjectStore[80] qui est un produit commercialisé. Pour parvenir à ce résultat, le seul moyen possible est d'avoir le même type de références vers les objets temporaires comme vers les objets persistants. Ces références étant matérialisées par des adresses virtuelles pour C++, une solution consiste alors à couper l'espace virtuel en deux parties. L'image d'une des deux parties persiste sur le disque entre deux exécutions consécutives d'un programme : elle s'assimile à un espace de pagination persistant.

La distinction entre les objets persistants et temporaires est faite lors de leur création. En fait, la persistance n'est pas totalement orthogonale au système de types car seuls les objets générés à partir de classes et alloués dynamiquement peuvent être persistants. Ils sont alors alloués dans l'espace d'adressage persistant étant pour partie géré comme un tas. C'est d'ailleurs une technique similaire qui a été utilisée dans PS−Algol[23].

Cette approche ne résout pas tous les problèmes liés à l'introduction de la persistance dans C++. En effet, la coopération entre les deux mondes est à l'origine d'une faiblesse accrue concernant la sûreté du langage (problème qui était déjà présent dans l'approche non orthogonale). Cette sûreté est essentiellement mise en cause par la manipulation de

pointeurs et le forçage de type ("cast") dans le C++ d'origine, défauts qui sont d'ailleurs un héritage de C. L'utilisateur doit donc se méfier de l'utilisation de pointeurs non persistants dans des objets persistants qui perdent toute signification d'une exécution à l'autre.

Concernant le nommage des instances de la base dans ObjectStore, la technique utilisée est la même que dans E. L'utilisateur déclare des variables globales persistantes et le système s'occupe alors d'allouer statiquement l'espace persistant pour l'instance désignée par cette variable. Cela permet donc à ObjectStore de stocker des objets de n'importe quel type C++.

Globalement, la seule fonctionnalité structurelle dont nous avons peu parlé jusqu'ici est l'introduction de constructeurs "dynamiques" du type ensemble. Aucun des C++ persistants présentés ci−dessus n'étend son système de types à ce genre de constructeurs ; ils n'ont donc pas la puissance de modélisation du modèle à objets complexes présenté dans la section II.2.1. Ces constructeurs sont la plupart de temps introduits par l'intermédiaire de classes paramétrables (classes génériques).

II.2.7 Conclusions

Nous avons présenté dans la section II.2 les aspects structuraux liés à des modèles de données offrant des capacités de définition d'objets abstraits (définis comme des objets identifiés). Tous n'ont pas la même puissance de modélisation, l'idéal étant pour un modèle de permettre une combinaison orthogonale de ces constructeurs. Par ailleurs, les modèles d'objets complexes paraissent être les plus aptes à une modélisation "naturelle" des données.

Dans tous les cas étudiés, nous avons pu observer que la notion de référence est nécessaire pour supporter les objets abstraits. Or le but de ces systèmes est en général de manipuler les données avec un langage complet. La conséquence est que les données peuvent être soit persistantes (données de la base) soit temporaires. Il faut alors qu'elles puissent être manipulées de la même manière. Les deux espaces d'instanciation induits font que cette différence, du point de vue structurel, est essentiellement supportée par la référence. Si nous voulons que la référence soit implantée sous la forme d'une adresse virtuelle (fonction $\rho \Leftrightarrow$ adressage virtuel (déréférençage)), il faut alors diviser l'espace virtuel en deux parties (c'est la solution la plus intéressante dans le cas de C++ persistant ; elle engendre peu de modifications du compilateur).

La plus mauvaise solution est d'avoir deux types de références comme c'est le cas pour une grande partie des systèmes étudiés. Cela rend difficile la coopération entre les deux espaces et notamment la migration des objets entre les deux espaces. Il faut donc un seul type de référence qui permette de désigner un objet, qu'il soit temporaire ou persistant. Comme nous l'avons vu, cette coopération pose aussi le problème de la cohérence des données de la base. En effet, ces dernières ne doivent pas contenir des références non persistantes. Une solution, notamment utilisée dans O2, consiste à propager la propriété de persistance à de telles références temporaires et à leurs données associées.

Le service de nommage d'instances est un autre élément primordial de tous ces modèles Celui−ci est généralement séparé de la définition des types et permet de créer des points d'accès aux données persistantes. C'est un système de nommage plat dans la mesure où il n'est pas possible de structurer ces noms en répertoires comme nous pouvons le faire avec un système de fichiers. Toutes les données de la base sont accessibles à travers un nom composé d'un préfixe désignant une instance suivi d'un chemin défini par le schéma des types. C'est le compilateur du langage de manipulation des données qui associe à chaque nom la référence de la donnée qu'il désigne. C'est le rôle de la fonction ν qui est mise en œuvre par le compilateur. Les propriétés de cette fonction sont importantes. En effet, si cette fonction est monovaluée (comme c'est le cas pour les constructeurs "statiques" tels que n−uplet ou tableau), le compilateur peut associer statiquement l'adresse au nom. Ce n'est plus le cas du constructeur ensemble pour lequel nous pouvons uniquement accéder dynamiquement (interprétation du nom à l'exécution) à chacune des références associées à ce nom (utilisation d'un itérateur). Il est donc clair qu'un modèle d'objets complexes (avec constructeurs dynamiques) est plus coûteux à mettre en œuvre puisqu'il nécessite des phases d'interprétation.

II.3 Synthèse des différents concepts des systèmes de types

L'aspect structurel des modèles de données étudiés précédemment est une dimension importante dans un SGBD car il détermine le modèle de mémoire à mettre en œuvre. Néanmoins, dans le contexte des LPBD (notamment orientés objet), nous avons un système de types très puissant offrant un grand nombre de concepts sémantiques. Cette section dresse un inventaire de tous ces concepts. Un des principaux besoins d'un système de types est la définition de règles d'équivalence entre les types. C'est essentiellement autour de ce point que s'articule l'étude de ces différents concepts.

II.3.1 Les types de base, les types construits et les types concrets

Tous les systèmes de types offrent la notion de types de base qui correspond aux domaines d'objets "atomiques" à partir desquels il est possible de construire des objets plus complexes. Les exemples de tels types sont les entiers (courts ou longs), les réels de précisions différentes, les caractères ou encore les booléens. Nous pouvons aussi ajouter à cette liste les types énumérés définissant des valeurs de base qui sont des symboles.

Pour définir des valeurs construites, les systèmes offrent des constructeurs plus ou moins intéressants. Nous divisons ces constructeurs en deux catégories : les constructeurs statiques et les constructeurs dynamiques. Cette distinction s'appuie sur le fait que la fonction ν (cf. II.2.1.3) est pour eux mono−valuée ou multi−valuée. Cela implique concrêtement que cette fonction peut alors être implantée statiquement par le compilateur ou de manière interprétée à l'exécution.

II.3.1.1 Les constructeurs statiques

Les constructeurs statiques sont couramment utilisés, notamment dans les langages compilés usuels comme C ou Pascal. Ils sont essentiellement au nombre de trois :

l'enregistrement (n−uplet), l'enregistrement à champs variants ("variant record") et le tableau (vecteur).

L'enregistrement

Le constructeur enregistrement ou n−uplet permet de construire un objet contenant un nombre fini de champs nommés. Il permet de définir des valeurs qui sont des agrégats de valeurs. Chaque nom de champ est unique dans l'enregistrement. Si "o" désigne un objet n−uplet, la désignation du champ "a" de celui−ci se fait par "o.a".

L'enregistrement à champs variant

L'enregistrement à champs variant permet de construire des objets qui peuvent être de plusieurs types (union de types). De tels objets sont d'un seul des types possibles quand ils sont instanciés. L'accès à un champ se fait par "o.a" comme pour le n−uplet. La sémantique n'est pas la même car dans le cas présent, il s'agit de choisir le type à travers lequel nous manipulons l'objet plutôt qu'une partie de l'objet. De plus, il n'est pas possible statiquement de savoir si le type de l'objet est compatible avec le type à travers lequel il est manipulé. Dans C ou C++, c'est l'utilisateur qui est garant du bon fonctionnement. Dans d'autres cas, comme Pascal, l'objet instancié contient un champ de typage qui permet une vérification dynamique de type soit par l'utilisateur soit par le langage.

Le tableau

Le tableau permet de définir des objets contenant un ensemble de taille fixe de sous−objets de même type. Ces sous−objets sont désignés par un nom qui est un indice dans le tableau. L'accès au $3^{ième}$ élément se fait par exemple par "o.3". Certains modèles offrent la possibilté de définir des tableaux de taille variable. Leur taille est alors définie à l'exécution, fixant ainsi dynamiquement les noms désignant les éléments du tableau.

II.3.1.2 Les constructeurs dynamiques

Les constructeurs dynamiques sont très utiles lorsque nous devons modéliser des ensembles dont nous ne pouvons pas prévoir la taille par avance. Il n'est effectivement pas viable de modéliser de telles structures avec des vecteurs d'une taille maximum que nous espérons supérieure aux prévisions les plus pessimistes. Ces constructeurs sont essentiellement utilisés dans le contexte des bases de données.

L'ensemble et le multi−ensemble

L'ensemble, dont l'intérêt est de définir une structure de taille variable, est généralement introduit avec deux comportements différents : avec ou sans duplication d'éléments. Autoriser les duplications (multi−ensembles) permet de réduire le coût des opérateurs de manipulation d'ensembles, notamment dans le cas des opérateurs de l'algèbre relationnelle[36]. Nous avons vu que le nom désignant chaque élément d'un ensemble "o" est "o.∈ ". Il devient alors nécessaire d'avoir un itérateur pour accéder successivement à chaque élément du type "**`pour tout`** e ∈ o".

La liste et la séquence

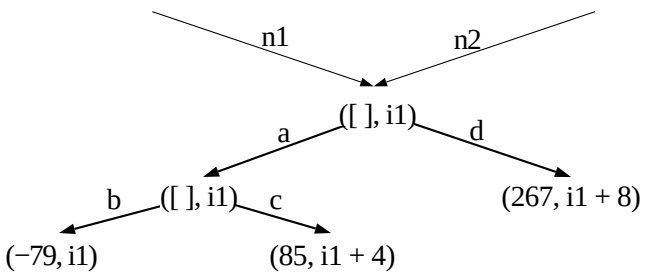
Le constructeur ensemble vu précédemment ne définit aucun ordre entre ses éléments. La liste peut être considérée comme un ensemble avec un ordre sur ses éléments. La séquence est une liste manipulable comme un tableau[79] de taille variable, permettant donc un accès direct aux éléments par rapport à leur position absolue dès l'instant où ils sont accédés. L'opérateur d'itération est toujours utilisable. D'autres opérateurs sont offerts pour profiter de la sémantique supplémentaire de la liste. Pour des listes à chaînage simple, des opérateurs permettent l'accès au premier élément et à l'élément suivant un élément donné. Il existe les opérateurs symétriques dans le cas de listes à chaînage double.

Des règles d'équivalence ou de conformité sont généralement définies entre les types construits avec ces constructeurs. Il s'agit alors d'équivalence structurelle[34] comme celle définie dans Galileo[8] ou O2[68]. Elle définit la politique d'affectation possible d'un objet d'un certain type à un nom d'un autre type (substitution de type). Il est souvent possible de nommer ces types ; nous parlons alors de types concrets. Il n'y a néanmoins aucune sémantique définie sur les noms associés à ces définitions de type dans les deux systèmes cités précédemment. Les règles d'équivalence entre types concrets restent structurelles.

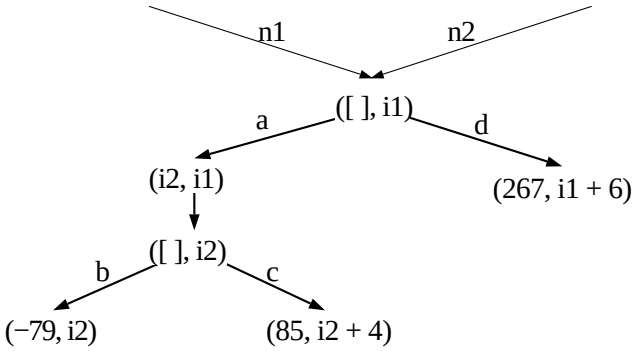
Toutes les règles d'équivalence se fondent la plupart du temps sur une relation d'ordre partiel, notée d'une manière générale ≤ ; il s'agit de la relation de sous−typage. Pour les types de base, nous avons généralement des règles d'équivalence par nom. S'il n'y a aucune dépendance sémantique introduite entre ces noms, comme des règles de sous−typage explicites (entier "sous−type" de réel ⇔ entier ≤ réel), l'équivalence entre les types de base se limite à une égalité (par exemple, nous ne pouvons affecter un entier qu'à une variable de type entier). Les règles d'équivalence structurelle au niveau des constructeurs peuvent aussi définir du sous−typage comme par exemple entre des n−uplets :

```
[a: t1, b: t2, c: t3] ≤ [a': t1', b':t2']
    où t1 ≤ t1' et t2 ≤ t2'
```

La définition de telles règles a des conséquence sur la mise en œuvre de la fonction ν. En effet, elle associe à chaque nom d'un champ de n−uplet une référence calculée à partir de la référence associée au n−uplet lui−même. Or la règle précédente implique qu'il n'est plus possible dans certains cas de calculer statiquement la référence d'un champ lors d'une substitution. Prenons un exemple où deux noms `n1` et `n2` désignent le même objet n−uplet de type `T = [a:[ b:entier, c:entier], d:entier]`. `n1` est de type `T` et `n2` de type `T' = [a:[b:entier], d:entier]` (nous avons `T ≤ T'`) :



Le calcul statique de la référence associée au champ "d" avec la fonction ν (cf. annexe A) se résume à $\nu(n2.d) = \nu(n2) + \delta(\tau(n2.a))$ où δ est la fonction qui calcule la taille des objets générés par un type et τ rend le type associé à un nom. Le calcul est le même dans le cas de n1 avec $\nu(n1.d) = \nu(n1) + \delta(\tau(n1.a))$. Par contre $\nu(n2.d) \neq \nu(n1.d)$ car $\delta(\tau(n1.a)) = 2*\delta(\text{entier})$ alors que $\delta(\tau(n2.a)) = \delta(\text{entier})$; l'objet donné ne peut donc pas être manipulé à travers le nom n1. Si nous supposons que δ(entier) = 4 et δ(référence) = 6., nous avons alors le graphe suivant :



La solution à ce problème est que chaque objet contienne sa description structurelle, l'accès aux champs étant alors interprété, ou de décomposer l'objet à chaque niveau de sous−structure comme le montre le schéma ci−dessus. Nous avons alors $\delta(\tau(n1.a)) = \delta(\tau(n2.a)) = \delta(\text{référence})$. C'est cette dernière solution qui est mise en œuvre dans le système O2. L'exemple décrit ci−dessus illustre bien le problème de la recherche d'un juste équilibre entre offrir des fonctions avancées (ici une conformité structurelle plus souple entre les types) et minimiser le coût de leur mise en œuvre.

Les constructeurs représentent d'ailleurs le plus gros effort d'implantation d'un modèle de données. Ce constat a engendré un axe de recherche intéressant[13] (notion de "bulk types") pour essayer de définir et d'implanter un "super" constructeur à partir duquel tous les autres peuvent être définis. Cette approche va dans le sens du langage Lisp dans lequel il n'y a qu'un seul constructeur (la liste) à partir duquel il est effectivement possible de simuler d'autres constructeurs tels que des n−uplets ou des tableaux. Cette généricité est intéressante et peut permettre l'extension des constructeurs d'un langage. Par contre l'efficacité des constructeurs ainsi implantés est moindre puisqu'ils sont interprétés en termes du "super" constructeur.

II.3.2 Les types abstraits, les classes et les types nommés

Les notions de types abstraits et de classes sont souvent très proches dans les systèmes orientés objet. La notion de type abstrait correspond à une approche algébrique des données définies, manipulables exclusivement à travers différents opérateurs spécifiques. C'est donc de cette notion que vient celle d'encapsulation. Les opérateurs peuvent être relativement complexes et notamment être paramétrés par des types différents (par rapport à un opérateur comme "+" dans l'ensemble des entiers). Nous avons alors des algèbres multi−sortes[24][55][74].

Un type abstrait définit donc une structure d'objet et le comportement de ceux−ci sous la forme d'un ensemble d'opérateurs généralement appelés des méthodes dans les langages à objets. Certains systèmes de types, comme celui de Guide, sépare le type des entités (c'est à dire l'ensemble des opérations définissant ces entités) de la manière dont elles vont être implantées. En Guide, c'est une classe qui implante un type. Plusieurs classes peuvent ainsi implanter un même type. Cet aspect du modèle ne parait pas utilisable de manière courante. Il peut être intéressant dans le cadre de types génériques comme l'ensemble qui peuvent être implantés de plusieurs manières (arbre B, liste, index haché, etc...).

La notion d'objet définie par un type abstrait est indépendante de celle d'objet abstrait définie dans la section II.2.1.4. En effet, les objets générés par les types peuvent être des objets abstraits ou des valeurs. Dans le système O2, il s'agit d'objets abstraits alors que dans Galileo, nous avons plutôt des valeurs qui par ailleurs sont référençables.

Dans certains modèles, comme ceux de C++ ou de O2, la notion de classe est équivalente à celle de type abstrait telle qu'elle a été définie ci−dessus. Dans d'autres modèles, la classe désigne l'ensemble de toutes les instances de la base qui ont un comportement commun (décrit par un type). C'est le cas de Orion ou de Galileo pour lequel les classes sont des ensembles d'instances de types potentiellement différents mais néanmoins conformes au type associé à la classe.

La conformité entre les types abstraits n'est donc plus seulement structurelle mais aussi comportementale. En effet, pour pouvoir définir une relation de conformité entre types abstraits, il faut définir une conformité entre les méthodes qui leur sont associées. Cet aspect est développé dans la section II.3.4 qui suit. La conformité ainsi définie s'appuie sur des règles fixes définies par le système de types.

Dans la majorité des systèmes, les types abstraits sont toujours nommés. De plus, de la même manière que pour les types de base, l'équivalence de types est définie par nom[34]. Le concept d'héritage permet alors d'étendre la relation de conformité des types abstraits (si T1 hérite de T2 alors $T1 \leq T2$). Cela permet notamment de factoriser des comportements communs. Lorsqu'un type hérite d'un autre, il hérite à la fois du comportement associé à ce dernier ainsi que de sa structure. Les types abstraits définis comme conformes, parce qu'ils sont en relation d'héritage par rapport à leur nom, le sont aussi du point de vue structurel et comportemental. L'héritage peut être vu comme un opérateur de composition de types[26] qui permet la réutilisation de définitions. Le nom n'est en fait qu'un raccourci d'écriture même s'il exprime généralement une grande partie de la sémantique de l'entité modélisée.

II.3.3 Les types génériques

Les types ou classes génériques sont paramétrés. Ils définissent des types à part entière lors de leur utilisation dans la définition du type associé à une variable. Ils représentent alors autant de types différents qu'ils ont eu de paramètres effectifs différents. Peu de systèmes de types offrent la possibilité de définir des types génériques. Les langages C++, Eiffel ou Guide sont parmi ceux offrant cette fonction.

Les constructeurs sont des exemples de types génériques. Un point important dans leur définition est l'ensemble des contraintes imposées aux paramètres que leur sont affectés.

Dans le cas des constructeurs couramment rencontrés, nous pouvons dénombrer essentiellement deux catégories de paramètres : des types (paramètre effectif = type) et des cardinalités qui sont des entiers naturels (paramètre effectif = instance). Des exemples de constructeurs montrent différentes combinaisons de ces catégories de paramètres :

- **`ensemble de`** *`T`* : constructeur ensemble avec un seul paramètre de la catégorie des types (type T).
- **`chaine (`***`20`***`)`** : constructeur chaîne de caractères avec un seul paramètre qui est une cardinalité (la taille maximum de la chaîne, soit 20).
- **`tableau (`***`1500`***`) de`** *`T`* : constructeur tableau de taille fixe avec comme paramètre le type T des éléments du tableau et sa taille (1500).

Les types génériques permettent donc d'étendre le nombre des constructeurs d'un système de types. Les exemples précédents nous ont montré des types génériques comportant au maximum deux paramètres. Certains systèmes de types, comme celui de E[88] (extension de C++), permettent de définir des types génériques paramétrables avec un nombre quelconque de paramètres. L'exemple suivant montre la définition d'un type générique représentant un arbre B avec le langage E :

```
class Arbre_B
    [
    class TypeClef {},
    class TypeElem {},
    int compare (TypeClef *, TypeClef *)
    ]
{
...
};
```

L'exemple définit un arbre B dont les éléments sont de type TypeElem et la clé d'ordonnancement de type TypeClef. Ces types peuvent être aussi bien des types abstraits (class C++), des types construits ou des types de base. Nous pouvons remarquer que le dernier paramètre est de type fonction de la même catégorie qu'une cardinalité. Quant au langage Guide[78], ses classes génériques peuvent être paramétrées avec au maximum un paramètre de dimensionnement et autant de paramètres types qu'il est nécessaire.

La définition de règles de conformité entre des types génériques instanciés est plus problématique que dans le cas de types abstraits ou de types construits. En effet, si nous prenons l'exemple des constructeurs, leur conformité est définie implicitement par le système de types et elle est différente pour chaque constructeur. Si les paramètres des types génériques n'étaient que des types, la conformité pourrait être définie (types d'instanciation deux à deux conformes). Par contre, avec des paramètres tels que des cardinalités, il est plus difficile de définir des règles générales. Les types paramétrés compliquent donc le système de types et cela d'autant plus si ce dernier permet l'héritage.

II.3.4 Le polymorphisme

Les langages de programmation usuels tels que Pascal ou C sont dits monomorphes car les fonctions, les variables ou les objets définis par le langage ont un type unique. L'équivalence de type est donc exclusivement une égalité de types. La plupart des langages récents et notamment les langages à objets sont polymorphes dans le sens où un objet ou une variable peut avoir plusieurs types. De la même manière, une fonction polymorphe est une fonction dont les paramètres effectifs peuvent être de différent types. Dans l'étude de Cardelli et Wegner[26], plusieurs catégories de polymorphisme sont distinguées : le polymorphisme d'inclusion, le polymorphisme paramétrique et le polymorphisme ad hoc.

II.3.4.1 Le polymorphisme d'inclusion

Le polymorphisme d'inclusion couvre les notions de sous−typage et d'héritage présentes notamment dans les langages à objets. Un type est un ensemble de valeurs ou domaine. Ainsi, la relation de sous−typage est une relation d'inclusion ensembliste de domaines. Si T1 *sous−type de* T2 (T1 ≤ T2) alors nous avons T1 ⊆ T2. Nous avons par exemple entier ≤ réel ou employé ≤ personne.

Tous ces ensembles sont des sous−ensembles de l'univers V des valeurs (cf. Fig. 2.6). L'ensemble des types dans V ordonné par l'inclusion ensembliste forme un treillis. Le plus grand élément est l'ensemble V qui correspond au type "top". Le plus petit élément est l'ensemble contenant le plus petit élément de V qui est généralement noté **`nil`**. C'est d'ailleurs la seule valeur polymorphe que nous retrouvons dans des langages monomorphes comme Pascal.

II.3.4.2 Le polymorphisme paramétrique

Le polymorphisme paramétrique se rapporte à la notion de type générique ou type paramétré vue précédemment (cf. II.3.3). Le polymorphisme paramétrique est obtenu lorsqu'une fonction s'exécute de manière uniforme pour un ensemble de types. Prenons par exemple la fonction de concaténation de listes :

```
concat (l1, l2) → l_res
```

Cette fonction s'applique aussi bien sur une liste d'entier que sur une liste de chaînes de caractères. Du point de vue de l'implantation, une fonction entre dans le cadre du polymorphisme paramétrique lorsque le code exécuté est le même quelque soit le type des paramètres effectifs (parmi les types possibles) qui lui sont passés. Ces paramètres ont alors en commun leur structure.

II.3.4.3 Le polymorphisme *ad hoc*

Dans le polymorphisme *ad hoc*, une fonction peut s'exécuter sur un ensemble de types différents qui ne partagent pas obligatoirement la même structure. De plus, le comportement de cette fonction peut être différent suivant le type de ses paramètres effectifs. Nous distinguons par ailleurs deux types de comportement dans le polymorphisme *ad hoc* : soit par surcharge, soit par cœrcition. Prenons comme exemple l'opérateur d'addition qui fonctionne pour les entiers et pour les réels :

```
3   + 4     (entier × entier → entier)
3.0 + 4     (réel × entier → réel)
3   + 4.0   (entier × réel → réel)
3.0 + 4.0   (réel × réel → réel)
```

Etant donné que les entiers et les réels ont des structures différentes (formats mémoire différents), l'opérateur "+" peut être surchargé sous la forme de quatre opérateurs différents suivant le types des paramètres. Une autre manière de résoudre ce polymorphisme est par exemple de n'avoir que deux opérateurs (un pour les entiers et un pour les réels) et de forcer (cœrcition) la valeur au bon type (réel) dans le cas où les paramètres ne sont pas homogènes.

II.4 Conclusion

Ce chapitre présente l'évolution opérée par les systèmes à base d'objets, des modèles de données utilisés par les SGBD usuels, vers des systèmes de types beaucoup plus puissants. Deux niveaux sont distingués pour caractériser la puissance d'un système de types : la puissance d'expression structurelle des données modélisées et les fonctionnalités introduisant la sémantique à associer à ces données.

Les modèles de données des SGBD définissent seulement l'aspect structurel des données. Cet aspect est prépondérant dans la mise en œuvre d'un modèle de mémoire permettant d'implanter le modèle de données. Il détermine notamment la mise en œuvre de l'association entre les noms définis par le schéma et les références des objets qu'ils désignent. Cette association peut être calculée statiquement (par le compilateur) ou interprétée à l'exécution entrainant ainsi une perte d'efficacité. Nous avons ainsi distingué deux catégories de constructeurs (statiques et dynamiques) correspondant aux deux types de résolution. Les constructeurs dynamiques sont essentiellement ceux qui définissent des collections dont la taille change au cours de l'exécution.

L'importance de la notion de référence, qui représente un manque important des SGBD relationnels, a été soulignée ; cette notion permet de pallier les problèmes de redondance, de partage et de modélisation de données cycliques. Cette notion était d'ailleurs déjà présente dans les SGBD réseaux dans lesquels elle était matérialisée par les curseurs. Les systèmes à objets lui associent généralement plus de sémantique qu'à de simples adresses mémoire à travers la notion d'identificateur d'objet.

Outre l'aspect structurel de la modélisation des données, les systèmes de types offrent de nombreux concepts permettant d'exprimer plus de sémantique mais aussi d'augmenter la sécurité à la manipulation des données. Parmi les concepts les plus intéressants, nous pouvons citer :

- L'encapsulation issue de la notion de type abstrait qui consiste à définir une interface de manipulation (composée de fonctions) pour des données définies par ailleurs.
- Les règles de sous−typages qui peuvent par exemple autoriser la manipulation de données de plusieurs types à travers une même variable. Ces règles ont aussi des conséquences sur la résolution de l'association nom/référence mémoire.

- L’héritage entre types qui permet de définir des règles de sous−typage explicites et de factoriser des définitions (ainsi que des comportements).
- Le polymorphisme permettant d’offrir soit la généricité soit le sous−typage. Pour la généricité, il s’agit de pouvoir définir des fonctions gardant la même sémantique lorsqu’elles s’appliquent sur des données de types différents. Dans le cas du sous−typage, cela permet, comme le deuxième point le montre, de manipuler de manière transparente des données de types différents (pas nécessairement avec la même sémantique).

Tous les concepts étudiés dans ce chapitre, qu’ils soient structurels ou sémantiques, sont à priori indépendants les uns des autres et peuvent donc être combinés de manière orthogonale dans un système de types. Cette richesse de concepts, notamment par rapport au modèle relationnel, est certainement à l’origine du fait qu’aucun standard de modèle à objets (si cette notion a un sens) n’a encore émergé [75]. C’est d’autant plus difficile que la définition d’un tel modèle est conflictuelle dans un contexte de concurrence entre des langages à objets qui n’ont pas obligatoirement les mêmes objectifs. De plus, certains concepts, comme les règles de sous−typage, ont des conséquences directes sur les possibilités d’implantation. Les choix et les combinaisons de ces concepts sont donc différents d’un système à l’autre. Ces choix sont principalement guidés par les objectifs à atteindre en termes de fonctionnalités et de performances.

Chapitre III

Langages et bases de données

Donner à une chose un nom, une identification, un titre ; la sauver de l'anonymat, l'arracher du lieu sans nom, en un mot l'identifier − eh bien, c'est une façon de faire naître la dite chose.

Salman Rushdie − "haroun et la mer des histoires"

Les principaux rôles des SGBD peuvent être résumés suivant trois grands axes :

- Le premier est d'offrir un stockage des données persistantes transparent à l'utilisateur (abstraction du fichier). Cette fonction se traduit par la définition d'un modèle structurel pour modéliser les données et d'un langage de manipulation de ces données.
- Le deuxième correspond au besoin qu'ont les utilisateurs de pouvoir accéder simultanément à des données tout en assurant la cohérence de ces accès (accès concurrents assurés par le système transactionnel).
- Enfin, le dernier et non le moindre, concerne la sécurité des données de la base. Cet aspect se décompose en deux parties : sécurité des données vis à vis des pannes (aspect transactionnel) et autorisation d'accès par rapport aux droits d'un utilisateur (notion de vue).

Du côté des langages de programmation, leur intérêt réside dans la puissance de traitement et aussi de modélisation qu'ils offrent. Les deux domaines sont complémentaires : le développement d'applications nécessitant une gestion intensive de données persistantes met généralement à contribution les deux domaines. Leur coopération plus ou moins heureuse a naturellement conduit à envisager leur fusion dans un même langage.

L'intégration des fonctionnalités généralement offertes par un SGBD dans un langage de programmation pose de nombreux problèmes. Le début du chapitre décrit succintement les différentes approches retenues pour concrétiser cette intégration. Ensuite, nous énumérons les différentes fonctions que nous retrouvons dans ces langages ou dans ces systèmes, en insistant sur leur importance. La dernière partie propose une analyse de SGBD orientés objet et de LPBD existants par rapport aux différentes fonctions énumérées auparavant. Un tableau comparatif est donné avant la conclusion.

III.1 Les différentes approches d'intégration entre langages de programmation et fonctions base de données

Les différentes approches d'intégration se divisent en deux grandes familles qui correspondent aux deux domaines impliqués dans la fusion. La première se compose des langages persistants qui consiste à étendre un langage de programmation général pour prendre en compte dans un premier temps la persistance puis d'autres fonctions base de données (par exemple la gestion de collections). L'autre approche, issue du domaine des bases de données, consiste soit à étendre un SGBD existant pour prendre en compte la notion d'objet encapsulé par des méthodes. Cela revient à définir un SGBD à base d'objets qui peut être intégré à un langage de manière plus profonde (réduction importante des problèmes de dysfonctionnement).

III.1.1 Les langages persistants

La première approche, issue du domaine des langages, veut faire tendre fonctionnellement ceux−ci vers des SGBD. Cela consiste généralement à étendre un langage de programmation existant. D'un point de vue stratégique, ce choix est très intéressant étant donné la difficulté pour promouvoir un nouveau langage. La tâche revient alors à introduire la persistance dans un langage de programmation classique. Elle peut être résolue de plusieurs manières :

- Une première manière est d'introduire dans le système de types un type ou un constructeur spécifique auquel est associée la propriété de persistance. C'est le cas de certains C++ persistants (cf. II.2.6.1) qui définissent une classe pour propager la propriété par héritage. Dans le même sens, l'approche d'un des pionniers dans le domaine, Pascal/R[92], consiste à additionner les systèmes de types du langage et du SGBD que nous voulons faire coopérer. C'est ce qui est fait en ajoutant au système de types de Pascal la notion de relation.
- Une autre façon d'ajouter la persistance dans un langage est de définir des concepts introduisant la propriété de manière orthogonale au système de types. C'est généralement réalisé à la création d'un objet ce qui implique que la propriété est définie soit par une variable (variables persistantes[87]), soit par un allocateur spécifique ("pnew"[80]). Le langage PS_algol[12], qui fut un des premiers à aller dans cette direction, illustre bien cette approche.

La persistance s'accompagne généralement de solutions pour les fonctions d'accès concurrents et de transactions. Ces fonctions sont pratiquement toujours transparentes au niveau du langage. Par contre, un autre aspect qui ne l'est pas concerne la gestion des collections.

Là encore, l'extension peut s'opérer soit par ajout de nouveaux constructeurs dans le système de types (orthogonaux ou non), soit à l'aide d'un type spécifique, généralement générique. Les manipulations de ces collections restent néanmoins très primitives et bien peu de ces langages offrent la possibilité d'effectuer des accès associatifs à celles−ci. Le principal

reproche que nous pouvons faire à cette approche est que l'introduction des nouveaux concepts base de données engendre de nombreuses incompatibilités, car le langage n'a pas été défini dans ce but. Sa sûreté s'en ressent souvent et la prévention de nombreux risques est laissée à la vigilance de l'utilisateur.

La deuxième approche consiste à définir un nouveau langage pour lequel nous sommes sûr qu'il n'y aura pas de problèmes d'incompatibilité entre les différents concepts retenus. C'est le cas de langages comme Fad[35], Napier[77], Galileo[8] ou Dbpl[93][94]. L'intérêt de cette approche est que nous définissons un langage propre du point de vue de l'intégration des concepts (notamment pour la gestion de données persistantes) et a priori d'un niveau de sureté supérieur. Son inconvénient, outre le problème de la diffusion d'un nouveau langage, est surtout la charge de travail pour l'implantation du langage (c'est à la fois un langage complet et un SGBD qu'il faut développer !!).

III.1.2 Les SGBD relationnels étendus et les SGBD orientés objet

Dans le domaine des bases de données, nous retrouvons aussi les deux tendances similaires à celles des langages. Une des approches consiste à étendre un SGBD existant. Il s'agit généralement de SGBD relationnels auxquels sont donc ajoutées des capacités objets. Il est alors possible d'avoir des objets comme valeurs d'un attribut d'une relation. En fait, ce type d'extension revient à vouloir introduire tous les concepts d'un système à objets dans un système relationnel (et donc toute leur complexité). Le problème est que dans ce contexte, avoir la relation comme objet de base ne se justifie plus puisque le modèle relationnel est en fait un cas particulier de modèle d'objets complexes. Cette approche a l'avantage d'être conservatrice vis−à−vis de toutes les applications qui utilisent des systèmes relationnels, en leur offrant de nouveau concepts plus puissants. Par contre, nous nous heurtons non seulement à la complexité d'implantation des concepts objet mais aussi à celle induit par leur intégration dans le système relationnel et notamment dans le langage SQL. Le système développé dans le projet EDS[45][51] en est un exemple, de même que le système POSTGRES[101].

L'autre approche, que nous pouvons qualifier de "native", consiste à définir un nouveau SGBD complet implantant un modèle à objets. Un tel modèle permet non seulement la définition de structures mais plus généralement d'opérations associées aux données décrites. Ces opérations doivent pouvoir être aussi complexes que possibles et nécessitent un langage complet. Ces systèmes offrent la plupart du temps un mécanisme de coopération évolué avec un langage de programmation existant (par exemple C, Lisp, Smalltalk) voire avec plusieurs. Il existe donc toujours deux sytèmes de types manipulés par ce langage. Le dysfonctionnement à ce niveau ne disparait donc pas mais il est grandement réduit par rapport à ce qui se passe avec un SGBD relationnel car le système de types du SGBD à objets est beaucoup plus puissant. A la limite, le programmeur peut se limiter à n'utiliser que celui du SGBD lorsqu'il programme. Cette approche a l'avantage d'être plus cohérente que la précédente concernant la gestion des données de la base. Elle ne résout pas totalement les problèmes de dysfonctionnement à cause du choix d'utilisation d'un ou plusieurs langages existants pour programmer. En ce sens, cette approche est très voisine de celle des langages persistants. Elle nécessite aussi, comme dans le cas des langages, une tâche d'implantation

plus importante. Les systèmes représentant cette approche sont les systèmes O2[81], GemStone[25] ou Orion[60].

III.2 Les fonctions langages et base de données

Dans cette section, nous présentons les différentes fonctions que nous retrouvons dans les langages offrant des concepts base de données ou bien dans les SGBD orientés objet. Le "manifesto"[14] des SGBDOO donne une liste relativement exhaustive de toutes ces fonctionnalités. Toutes ces fonctions ne sont pas nécessaires. Néanmoins, certaines d'entre elles sont essentielles pour que le LPBD remplisse réellement le rôle d'un SGBD. Nous divisons ces fonctions en cinq grandes catégories :

1. Le modèle de structuration des données qui détermine le modèle mémoire à mettre en œuvre et cela pour les différents niveaux mémoires (mémoire virtuelle, mémoire persistante). Il décrit aussi le modèle de persistance et notamment de coopération entre les différents espaces.
2. La puissance sémantique offerte par le système de types. Cela concerne en particulier le modèle de sous−typage, d'héritage ou encore la possibilité de protéger certaines parties des définitions (publiques/privées).
3. La déclarativité présente dans le langage. Nous la retrouvons pour les accès associatifs aux données de la base ou dans l'expression de vues ou de règles.
4. La sécurité qui est un élément essentiel dans le contexte des bases de données. Elle intervient à la fois pour le langage, le schéma (ensemble des définitions) et pour les données de la base.
5. L'efficacité qui est déterminée en premier lieu par la mise en œuvre de la vérification de type. Elle va en fait dépendre du niveau d'interprétation du langage car généralement, plus le compilateur fait de choses, plus le programme généré sera efficace. Elle dépend aussi des techniques d'implantation et d'accès aux collections.

Nous détaillons maintenant les différentes fonctions que nous retrouvons pour chaque catégorie en signalant si elles sont essentielles et pourquoi.

III.2.1 Modèle structurel

Cet aspect a déjà été traité dans le chapitre précédent. La puissance du modèle structurel dépend donc des constructeurs offert par le système de types. Le minimum de structures nécessaires pour entrer dans le cadre des bases de données est d'avoir la notion de n−uplet et une notion de collection (ensemble, liste ou séquence). L'idéal est un modèle d'objets complexes dans lequel ces constructeurs peuvent être combinés récursivement et dans n'importe quel ordre.

L'introduction de tels constructeurs (notamment les collections) sous la forme de classes génériques est une approche pénalisante car elle ne permet pas de séparer la description conceptuelle de la description physique des données.

Ces constructeurs déterminent le modèle de mémoire et de ce point de vue, l'idéal est d'avoir des structures similaires en mémoire centrale et en mémoire secondaire. Du point de vue de la coopération entre les mondes temporaire et persistant (s'il y en a), cela nécessite une notion de référence homogène (identificateurs d'objet) qui tient compte de cette différence.

III.2.2 Puissance sémantique

Un type définit un domaine de valeurs. Il exprime donc la sémantique de ces valeurs qui peut se limiter à un aspect structurel. Dans ce cas, le système de type peut exprimer des règles de conformité structurelle comme nous l'avons vu en II.3.1. Il s'agit alors de sous−typage implicite.

Il est aussi possible de définir explicitement des règles de sous−typage. C'est le rôle du mécanisme d'héritage. Il est principalement utilisé dans le cas des types abstraits dont les définitions sont conséquentes. Il permet alors de factoriser ces définitions et donc des comportements. Ce mécanisme permet la mise en œuvre des concepts de généralisation et de spécialisation des modèles sémantiques.

Du point de vue comportemental, les systèmes de types des langages à objets permettent de définir du polymorphisme de façon corrolaire aux règles de sous−typage. Nous retrouvons notamment la possibilité de surcharger une méthode entre deux types conformes. Cette notion importante nécessite l'implantation d'un mécanisme de résolution tardive qui peut être relativement complexe dans le cas de l'héritage multiple.

Des aspects intéressants concernant les types abstraits sont les notions de constructeurs et destructeurs présentes notamment dans C++[102]. Ce sont des méthodes particulières associées à un type qui sont automatiquement appelées lors de la création et de la destruction d'un objet. Dans le cas du constructeur, cela peut permettre d'imposer l'initialisation d'une partie de l'état de l'objet.

III.2.3 Déclarativité

La déclarativité introduite par les SGBD relationnels est un acquis important dans le domaine des bases de données. Les nouvelles générations de systèmes devront impérativement offrir des possibilités similaires. Le but principal est de pouvoir exprimer des accès associatifs aux données de la base et notamment aux collections. Ces accès nécessitent alors la mise en œuvre de techniques d'optimisation.

Cette déclarativité se retrouve aussi dans les systèmes déductifs comme EKS[104] (règles de déduction) ou encore dans les SGBD actifs[21] (règles de réaction à des événements se produisant dans le SGBD). Ces différents types de règles commencent à apparaître de le contexte de systèmes à objets[86] et représentent une extension prometteuse notamment pour l'expression des contraintes d'intégrité.

Enfin, la mise en œuvre de langages de requêtes déclaratifs peut se faire de deux manières : sous forme de langage *ad hoc* ou de manière intégrée au langage de programmation. D'ailleurs, il est probable que les deux aspects devront être offert dans des buts différents : une grande facilité d'utilisation et d'apprentissage pour le langage *ad hoc*

(concepts plus simples pour des utilisateurs peu avertis), et pour le langage intégré (s'adressant plus à des experts), une forte puissance d'expression.

III.2.4 Sécurité

La sécurité concerne deux niveaux dans une base de donnnées : la résistance aux pannes et la confidentialité. Dans le premier cas, le problème est résolu par le mécanisme transactionnel qui résout aussi la gestion des accès concurrents. Dans le contexte des LPBD, ce mécanisme doit être mis en œuvre différemment que dans un SGBD si nous voulons pouvoir par exemple faire des validation ou annulation de transactions au cours de l'exécution d'un programme.

Le problème des pannes logicielles est aussi plus aigu dans ce contexte car il dépend de la sûreté du langage. Cet aspect du problème représente d'ailleurs le talon d'achille des approches du type de celles des C++ persistants.

L'aspect confidentialité dans les SGBD OO ou les LPBD actuels est pratiquement inexistant. Les notions d'utilisateur et de droits devront néanmoins être pris en compte et couplés à la notion de vue comme dans les systèmes relationnels. Cette notion de vue peut dans ce contexte objet être quelque peu différente puisqu'à priori elle ne concerne pas seulement les données mais aussi les méthodes. La notion de vue sur les classes[3] est une approche intéressante notamment en la couplant aussi avec des droits.

III.2.5 Efficacité

Les performances représentent évidemment un facteur décisif pour la viabilité d'un LPBD. La plupart des SGBD relationnels utilisent une approche interprétée, entre autres, parce que les seuls traitements exécutés sont des requêtes. Or, celles−ci demandent peu de ressources du point de vue de la compilation ce qui justifie une telle approche.

Dans le cas d'un LPBD orienté objet, la quantité de traitements est beaucoup plus importante. L'approche compilée est donc plus intéressante ici, surtout lorsqu'elle est combinée avec une vérification de types entièrement statique.

Enfin, le LPBD gérant des objets non persistants aussi bien que persistants, il faut que l'efficacité sur ces premiers objets soit dans un ordre de grandeur voisin de celui des langages orientés objet non persistants. L'idéal pour cela est que la mise en œuvre de l'adressage des données (aussi direct que possible) soit effectuée par le compilateur (code généré) plutôt qu'à travers des fonctions d'un gestionnaire d'objets (interprétation).

III.3 Comparaison de différents LPBD ou SGBD OO existants

La recherche concernant les LPBD ou les SGBD OO a donné lieu au développement de nombreux prototypes. Certains sont même déjà devenus des produits commerciaux. Pour ces derniers, il semble évident que le domaine n'est pas encore arrivé à une totale maturité comme en témoignent de nombreuses évaluations menées à l'heure actuelle. Cette partie s'applique à décrire comment se placent différents prototypes ou produits par rapport aux

différents axes énumérés précédemment. Ces systèmes sont GemStone, Orion, O2, Exodus (E), ObjectStore, ONTOS et ODE. Les quatre derniers de la liste sont bâtis autour du langage C++. Nous aurions pu ajouter à ceux−ci le système Versant[105] qui est également un produit. Ceci n'a pu être fait à cause de notre manque de renseignements à son sujet.

III.3.1 GemStone

Le système GemStone[25] a été un des premiers SGBD orienté objet qui ait abouti à un produit. Son modèle de données est celui défini par le système de types de Smalltalk. Le langage permettant de manipuler la base de données est le langage OPAL qui est une extension de Smalltalk. Le statut de GemStone par rapport aux cinq axes définis précédemment est le suivant :

1. ***Modèle structurel*** : il s'appuie sur le système de types de Smalltalk (approche tout objet) et procède par héritage (par exemple classe objet persistant) pour l'ajout des aspects base de données. Le constructeur ensemble est une classe (classe "Set") dont il faut hériter ; il faut donc créer une classe pour chaque ensemble d'objets de classe différente. De la même manière, les classes d'objets de la base doivent hériter de la classe "Object" (objets potentiellement persistants). La persistance est assurée par des instances racines et par propagation.
2. ***Aspect sémantique*** : c'est la sémantique définie par Smalltalk[54] et son système de types. Nous avons ainsi de l'héritage simple qui définit les seules règles de sous−typage puisqu'il n'y a que la notion de classe. Par contre, la coopération entre des objets de classes persistantes et d'autres de classes Smalltalk normales n'est pas gérée par le système. La notion d'association (lien bi−directionnel) n'est pas offerte non plus.
3. ***Déclarativité*** : Nous n'avons pas de séparation entre les descriptions logiques et physiques notamment des ensembles. Par contre, il est possible de faire des accès associatifs aux ensembles avec la méthode "select" de la classe "Set". Ce type de requête permet exclusivement d'opérer des filtres sur des ensembles. L'aspect langage de requêtes *ad hoc* est relativement pauvre.
4. ***Sécurité*** : la sécurité des données de la base est assurée (transaction) ainsi que celle des définitions (classes et méthodes) qui sont stockées dans la base. Par contre, les notions de vues et d'autorisation ne sont pas du tout abordées.
5. ***Efficacité*** : l'approche interprétée, dont le système a hérité du langage Smalltalk, fait que les performances sont relativement médiocres[41].

GemStone a le mérite d'avoir ouvert la voie dans le domaine des SGBD orientés objet. Il est néanmoins pénalisé d'abord par ses performances et d'autre part par son approche extension d'un langage à objets non persistant pour lequel il conserve une approche tout objet. Cette optique est lourde du point de vue conceptuelle et ne permet pas la séparation des définitions logiques et physiques des données de la base.

III.3.2 Orion

L'approche du système Orion[60] n'est pas une extension d'un langage existant. Il définit son propre modèle de données. Les manipulations sont exprimées dans un langage hôte, en l'occurrence Lisp.

1. ***Modèle structurel*** : le modèle structurel de Orion est un modèle d'objets complexes. L'ensemble est introduit comme un constructeur ce qui permet la séparation de sa description physique (implantation).
2. ***Aspect sémantique*** : le système de types permet un héritage multiple qui comme pour GemStone définit les règles de sous−typage. Toutes les données de la base sont des objets abstraits. Différents types de liens entre objets sont offerts combinant des liens existentiels, avec ou sans possibilité de partage.
3. ***Déclarativité*** : Orion possède un langage de requêtes un peu plus puissant que GemStone construit autour d'un opérateur permettant de faire des sélections et des projections (ou calcul d'image). Ce dernier aspect est intéressant puisque la complétude de ce langage dans un contexte tout objet nécessite l'inférence de nouvelles classes.
4. ***Sécurité*** : le système assure la sécurité des données de la base et du schéma. Orion ne couvre aussi d'autres aspects comme les vues ou les droits des utilisateurs.
5. ***Efficacité*** : là encore, l'approche interprétée pénalise grandement l'efficacité du système.

Malgré les performances à priori médiocres liées à l'interprétation des manipulations de données, le système Orion est intéressant. En effet, il a su profiter de l'interprétation pour développer des mécanismes d'évolution de schéma puissants que nous ne retrouvons dans aucun autre système. Ce système se prête donc très bien à l'implantation d'applications base de données nécessitant une dynamique importante comme dans le domaine de la CAO.

III.3.3 O2

Le système O2[81] suit une approche similaire à celle de Orion dans la mesure où le système définit son modèle de données. Ces données sont ensuite manipulables à l'aide de plusieurs langages hôtes (C, Basic, Lisp). Cette approche multi−langages a l'intérêt de demander à un utilisateur connaissant l'un d'entre eux seulement l'apprentissage du modèle du SGBD.

1. ***Modèle structurel*** : le modèle structurel est un modèle d'objets complexes[68] qui offre deux constructeurs de collection : l'ensemble et la liste. Il distingue les valeurs des objets abstraits et permet de faire persister n'importe quelle donnée d'un type O2.
2. ***Aspect sémantique*** : le système de types permet l'héritage multiple entre les classes (types définissant des objets). Cet héritage induit des relations de sous−typage entre les classes. Par ailleurs, des règles de sous−typage implicites existent entre les types définissant des valeurs (cf. II.3.1.2). Le système de types est sûr mais, à cause de la co−variance des types des paramètres en

entrée des méthodes surchargées, ce qui introduit plus de souplesse, il peut se produire des erreurs de typage à l'exécution (vérification de types à l'exécution).

3. ***Déclarativité*** : le langage O2Query[31][32] est un langage de requête *ad hoc*, s'inspirant syntaxiquement du langage SQL, qui offre la puissance de manipulation des objets complexes. Par contre, il ne peut pas être utilisé de manière orthogonale dans le langage de programmation sur n'importe quels ensembles ou comme une expression.
4. ***Sécurité*** : la sécurité des données de la base et du schéma (méta−base) est assurée par le mécanisme transactionnel. Par contre, la sureté du langage pâtit de la souplesse de son système de types. Ces erreurs sont néanmoins récupérées par le système qui avorte dans ce cas la transaction courante. Enfin, la cohérence des référence inter−espaces (temporaire/persistant) est assurée par promotion des objets référencés par des objets persistants.
5. ***Efficacité*** : les performances du système sont relativement moyennes principalement pour deux raisons. La première est une conséquence du choix d'une plateforme multi−langages qui est mise en œuvre par un interprète (O2Engine) du modèle de données d'un niveau élevé. La deuxième est liée, comme nous l'avons vu précédemment, à la souplesse du système de types qui oblige à faire des vérifications de types à l'exécution.

Pour conclure, nous pouvons dire que O2 paye cher la souplesse de son système de types de même que son approche multi−langages. Cette dernière était intéressante mais force est de constater qu'un seul langage est disponible (O2C) pour le produit plus une "API" pour C++. Il est néanmoins indéniable que le système offre de nombreux outils très intéressants (notamment des outils graphiques) qui en font un outil de prototypage d'applications de base de données à objets très puissant.

III.3.4 Exodus et E

Le système Exodus[27] est un système dit extensible dont le but est d'offrir une plateforme de développement de SGBD spécialisé. Le langage E[87] est un langage persistant étendant C++ qui a été implanté au−dessus d'Exodus. C'est le premier langage à avoir voulu étendre le langage C++ a des fonctions base de données.

1. ***Modèle structurel*** : le modèle structurel est celui induit par le système de types de C++. Les collections sont introduites par l'intermédiaire des classes génériques que permet de définir E. Des données de n'importe quel type C++ peuvent devenir persistantes dès l'instant où leur type est persistant. Les deux systèmes de types sont duaux ; n'importe quel type C++ peut être redéfini comme persistant.
2. ***Aspect sémantique*** : les aspects sémantiques sont exactement ceux du langage C++ avec l'extension concernant les classes génériques. Les mouvements des données persistantes entre le disque et la mémoire sont transparents pour le programmeur.

3. ***Déclarativité*** : il n'y a pas de notion de langage de requête dans le langage E. Il propose seulement des itérateurs permettant par exemple de définir des parcours de collections.
4. ***Sécurité*** : les données de la base sont protégées par le mécanisme transactionnel. Par contre, le schéma, composé de définitions C++, est contenu dans des fichiers externes. Nous avons donc un niveau de sécurité moindre pour celui−ci et la cohérence entre le schéma et ses instances est à la charge de l'utilisateur. Comme pour O2, les objets peuvent être promus persistants lors d'un référençage.
5. ***Efficacité*** : une fois les objets chargés du disque vers la mémoire, l'efficacité de la manipulation des objets est alors celle de C++. Cette efficacité est bonne car C++ offre un typage fort et génère statiquement l'adressage des données manipulées. Par contre, le format des objets est différent entre le disque et la mémoire ce qui implique des conversions coûteuses aux chargements et déchargements.

Le langage E est, parmi les extensions du langage C++, celle qui offre une des meilleures intégrations de la persistance dans le système de types. La notion de variable persistante est très intéressante pour la définition de racines d'accès aux données de la base. Son plus gros point noir est la sécurité qui est principalement liée au manque de sûreté de C++ mais aussi au fait que le schéma n'est pas géré par le système.

III.3.5 ObjectStore

ObjectStore[67][80] est aussi une extension du langage C++ qui n'est pas issue d'un prototype de recherche mais qui a été directement développé dans le but d'une commercialisation. C'est le plus récent des SGBD à objets qui ont été commercialisés. Il semble cependant que ce soit le plus performant des systèmes de cette catégorie (à laquelle appartiennent aussi les produits ONTOS et Versant).

1. ***Modèle structurel*** : c'est celui défini par le système de types de C++. Les collections (listes, ensembles et multi−ensembles) sont introduites par l'intermédiaire de classes paramétrées. Leur structure physique peut évoluer dynamiquement suivant la cardinalité. La persistance est orthogonale au système de types bien que seules les instances de classes peuvent persister.
2. ***Aspect sémantique*** : les aspects sémantiques ajoutés à C++ concernent tout d'abord les associations. Nous pouvons en effet définir des contraintes de liens symétriques entre des attributs membres de différentes classes (liens inverses comme dans [8]). Ces associations qui peuvent faire intervenir des collections (liens 1:1, 1:N, N:M) sont maintenues automatiquement par le système. Un autre aspect intéressant qui va dans le sens d'une séparation entre les descriptions logique et physique des collections est la possibilité de définir leur politique de gestion ainsi que la manière dont elles peuvent évoluer au cours du temps.
3. ***Déclarativité*** : les accès associatifs sont intégrés dans le langage sous la forme d'une extension du langage des expressions. Cette intégration est intéressante puisqu'elle est totalement orthogonale aux autres expressions du langage. Par

contre, ce langage permet essentiellement de faire des sélections sur des collections. Le système n'offre pas de langage *ad hoc*.

4. ***Sécurité*** : les notions d'utilisateur et de groupe sont prises en compte dans ObjectStore ainsi que des mécanismes d'autorisation basés sur ceux du système d'exploitation (en l'occurence Unix). Le système transactionnel assure la protection des données de la base. Par contre, son schéma est géré par l'utilisateur dans des fichiers Unix. Enfin, la faible sûreté du langage C++ est ici dégradée par le fait que la cohérence des références inter−espaces est à la charge du programmeur.
5. ***Efficacité*** : c'est le point fort de ObjectStore qui utilise le mécanisme de pagination du système d'exploitation pour charger la base de données active (données demandées par la transaction courante) dans l'espace virtuel du processus. Néanmoins, étant donné que la base stocke les objets dans leur format mémoire, que les objets C++ peuvent contenir des adresses (adresse de "vtbl" par exemple) et que ceux−ci peuvent être chargés à des adresses différentes dans des transactions différentes, le système doit mettre à jour ces adresses au chargement.

Outre les performances qui placent ObjectStore parmi les systèmes les plus intéressants sur cet aspect crucial, l'extension de C++ à la persistance est un atout important. Elle permet en effet de porter une application précédemment écrite en C ou C++ dans un environnement persistant avec très peu de modification. Cet argument est d'ailleurs mis en exergue dans la présentation du produit. Le reproche principal que nous pouvons faire concerne encore la sécurité qui est hypothéquée par la faible sûreté du langage.

III.3.6 ONTOS

Le SGBD orienté objet ONTOS[41][85], développé par la société du même nom, est aussi construit comme une extension persistante du langage C++. C'est le successeur du système Vbase[9] qui fut un des premiers SGBD à objets. Il peut aussi être utilisé comme une extension persistante du langage Smalltalk.

1. ***Modèle structurel*** : c'est toujours celui de C++. La persistance de même que les collections sont introduites par héritage de classes prédéfinies. La persistance n'est donc pas orthogonale au système de types et, qui plus est, la manipulation des données persistantes n'est pas transparente dans le langage. C'est à l'utilisateur de charger et décharger les données de la base qu'il manipule.
2. ***Aspect sémantique*** : ONTOS permet de faire de la vérification de types dynamique (à l'exécution). Il utilise pour cela le schéma de la base qui est stocké comme une méta−base et qui est accessible à travers des méta−classes prédéfinies. Par ailleurs, comme pour ObjectStore, des liens bi−directionnels peuvent être définis.
3. ***Déclarativité*** : ONTOS possède un langage de requêtes *ad hoc* du même type que celui de O2. Il s'agit d'un SQL objet permettant notamment d'appliquer des méthodes dans des requêtes du type "`select/from/where`". Ce langage peut être appelé depuis C++ de manière *ad hoc* et n'est donc pas intégré dans le langage d'expressions.

4. ***Sécurité*** : la sécurité des données et du schéma est assurée par le mécanisme de transaction. Il n'y a aucune notion d'autorisation ni la possibilité de définir des vues. La sûreté du langage est faible notamment parce que les références vers des objets persistants sont non typées bien qu'il y ait moyen de faire des vérifications dynamiques.
5. ***Efficacité*** : comme pour les autres C++ persistants, ONTOS gère le même format pour les objets lorsqu'ils sont stockés en mémoire ou sur disque. Néanmoins, le relâchement opéré au niveau du typage peut conduire à faire beaucoup de vérifications de types à l'exécution, pénalisant ainsi les performances. Par contre, le chargement étant explicite, le système ne paye pas le coût du défaut d'objet.

Ce SGBD orienté objet est en droite ligne avec la philosophie de C ou C++ dans la mesure où il offre des fonctions d'assez bas niveau. Par contre, le non typage des références persistantes contrarie quelque peu cette approche. Le manque de transparence notamment pour les accès aux données de la base en font un système à part parmi ceux qui sont décrits dans ce chapitre car ce choix est très contraignant pour le programmeur. Ce système parait plus proche d'une approche SGBD extensible similaire à celle du système Exodus.

III.3.7 ODE

Le système ODE[6], développé dans les laboratoires d'AT&T, s'appuie également sur le langage C++ comme langage de définition des données de la base et comme langage de manipulation. L'approche est comme pour ObjectStore de permettre une manipulation homogène des données temporaires et persistantes.

1. ***Modèle structurel*** : l'extension du modèle structurel de C++ vis à vis des collections est matérialisé par l'ajout des ensembles sous la forme d'un constructeur du langage de même niveau que le constructeur tableau. La persistance est indépendante des types dans la mesure où elle est exprimée à la création d'un objet. Ce n'est pas totalement vrai car il faut déclarer des pointeurs "duaux" lorsqu'une référence peut servir à manipuler un objet qui peut être soit temporaire, soit persistant.
2. ***Aspect sémantique*** : ODE permet de définir de la sémantique supplémentaire dans la définition de classes C++. En effet, il est possible de définir des contraintes d'intégrité concernant les attributs membres de la classe. Ces contraintes suivent de plus les mêmes règles d'héritage que les attributs. Par ailleurs, des déclencheurs peuvent être exprimés dans une classe ce qui confère à ODE la puissance d'un SGBD actif.
3. ***Déclarativité*** : outre les aspects décrits dans le point précédent qui permettent d'exprimer notamment des règles de production, ODE offre des mécanismes permettant d'exprimer des requêtes. Il ne s'agit pas d'un langage *ad hoc* mais d'itérateurs qui sont des instructions particulières du langage. Ces requêtes ne sont donc pas des expressions du langage comme pour ObjectStore. Ces itérateurs de nature "**`pour tout`** <e1 ∈ E1, ..., en ∈ En> **`tel que`** <condition> **`faire`**

<instructions>" offrent néanmoins une puissance d'expression intéressante (expression de jointures) et peuvent donner lieu à des optimisations.

4. ***Sécurité*** : les données de la base sont protégées par le mécanisme transactionnel. Par contre, comme pour ObjectStore, le schéma n'a pas le même niveau de sécurité que la base. Les aspects d'autorisations ou de vues ne sont pas traités.
5. ***Efficacité*** : l'efficacité est celle de C++ lorsque les objets sont en mémoire. Une actualisation des adresses contenues dans les objects persistants est néanmoins nécessaire lors du chargement comme pour ObjectStore. Ces deux systèmes sont d'ailleurs très proches par rapport à leur choix d'extension de C++ et d'implantation.

Il serait intéressant de savoir à quel niveau se place ODE vis−à−vis des performances car, comme pour ObjectStore, les extension apportées à C++ sont minimes. De plus, il offre des fonctions avancées comme les contraintes d'intégrité ou les règles de production qui en font le système le plus riche de ce point de vue même si ONTOS offre aussi des déclencheurs.

III.3.8 Tableaux récapitulatifs

Les tableaux qui suivent donnent un récapitulatif des différentes implantations des fonctionnalités les plus marquantes des différents SGBDOO décrits au−dessus.

		GemStone	**Orion**	**O2**	**Exodus**
modèle structurel	modèle d'objet	Smalltalk (tout objet)	tout objet	objet complexe	C++
	persistance	par héritage (Object)	tout persistant	orthogonale aux types	orthogonale aux DB types
	collections	par héritage (Set, Bag)	classes = collections	constructeur (set, list)	classes génériques
	références inter−espaces	cohérence par l'utilisateur	un seul espace	cohérence par promotion	cohérence par promotion
aspects sémantiques	langages de manipulation	**OPAL**, C, C++	**Lisp**	**O2C**, C, C++	**E** (C++)
	héritage	simple (classe)	multiple (classe)	multiple (classe)	multiple (classe)
	sous−typage	classe	classe	classe/type	classe
déclarativité	langage de requêtes	intégré (méthodes)	intégré ("select")	*ad hoc* (SQLobjet)	itérateurs (bas niveau)
	règles (vues)	non	non	non	non
	contraintes d'intégrité	non	non	non	non
sécurité	sûreté du langage	moyenne	bonne	bonne	bonne
	catalogue des définitions	méta−base	méta−base	méta−base	fichiers Unix
	autorisations	non traitées	non traitées	(bases, schémas)	non traitées
efficacité	langage	interprété	interprété	compilé	compilé
	manipulation d'objets	interprété	interprété	interprété	compilé
	vérification de type	dynamique	dynamique	statique & dynamique	statique
	chargement d'objets	change de format	change de format	"swizling" des OID	mise à jour infos. C++

		ObjectStore	**ONTOS**	**ODE**
modèle structurel	modèle d'objet	C++	C++	C++
	persistance	orthogonale aux classes	par héritage (Object)	orthogonale aux types
	collections	classes paramétrées	par héritage (Set, List)	constructeur (ensemble "<>")
	références	cohérence par l'utilisateur	cohérence par l'utilisateur	cohérence par l'utilisateur
aspects sémantiques	langage de manipulation	**C++**	**C++**	**C++**
	héritage	multiple	multiple	multiple
	sous−typage	classe	classe	classe
déclarativité	langage de requêtes	intégré (expressions C++)	*ad hoc* (SQLobjet)	intégré (instructions C++)
	vues ou règles	non	déclencheurs	déclencheurs
	contraintes d'intégrité	non	non	oui (locales aux classes)
sécurité	sûreté du langage	moyenne	faible	moyenne
	catalogue des définitions	fichiers Unix	méta−base	fichiers Unix
	autorisations	bases (implanté sur celui d'Unix)	non traitées	non traitées
efficacité	langage	compilé	compilé	compilé
	manipulation d'objets	compilé	compilé	compilé
	vérification de type	statique	statique & dynamique	statique
	chargement d'objets	mise à jour infos. C++	mise à jour infos. C++	mise à jour infos. C++

III.4 Conclusion

Dans ce chapitre, nous avons étudié différents SGBD à objets suivant cinq grands axes. Nous allons dans cette conclusion essayer de synthétiser les points forts et les points faibles de ces systèmes. L'objectif est d'évaluer de quelle manière nous pensons que le domaine va évoluer.

Il est déjà possible de distinguer plusieurs générations comcernant ces systèmes à objets. La première génération, à laquelle appartient GemStone, Iris ou Orion, se caractérise par une volonté d'avoir des concepts objets très puissants et d'en tirer le maximum de fonctionnalités. Ces systèmes pêchent principalement par leur approche interprétée. Cela implique qu'il est peu réaliste de les utiliser dans un contexte industriel pour une utilisation intensive sur de grosses bases de données. Le système O2 ainsi que le langage E dans une moindre mesure sont des héritiers de ces systèmes. Ils améliorent de manière sensible les performances grâce à leur approche compilée. Ils restent néanmoins sur ce point en deça des systèmes du type C++ persistant. Ils forment d'excellents outils de prototypage notamment le système O2 qui offre un environnement graphique très puissant.

La deuxième génération se caractérise par l'approche des C++ persistants qui est beaucoup plus pragmatique. Elle s'appuie sur le fait que le langage C++ tend à devenir le langage standard pour la programmation à objets. L'idée d'ajouter la persistance et d'autres fonctionnalités base de données tout en gardant la compatibilité avec C++ parait évidemment alléchante. Les résultats obtenus sont relativement intéressants surtout du point de vue des performances car le système de types de C++ permet un accès direct (calculé statiquement) aux données manipulées. Malheureusement, il faut payer cette compatibilité avec C++, même du point de vue efficacité (le modèle d'exécution de C++ nécessite de nombreuses informations de liaison à l'intérieur des objets car certaines d'entre elles dépendent du contexte d'exécution). Le principal point noir réside dans la sûreté du langage comme nous l'avons vu précédemment. Ces systèmes sont intéressants du point de vue de leur puissance mais les fonctionnalités offertes sont souvent de bas niveau. Ils nécessitent ainsi des experts pour les manipuler.

Globalement, certains aspects sont très pauvres dans tous ces systèmes ou restent très primitifs. Nous pouvons citer dans ce cas l'aspect langage de requêtes ou d'accès associatifs, la séparation entre les descriptions conceptuelle et physique des données ou encore le manque de déclarativité (contraintes d'intégrité). Un long chemin a néanmoins été accompli par tous ces systèmes mais nous pouvons tout de même nous poser la question de savoir si un nouveau langage (pouvant s'inspirer de standard tels que C++ dans une large mesure) ne se justifie pas si celui−ci offrait à la fois plus de sûreté et de meilleures performances que les C++ persistants actuels qui semblent prendre le dessus. C'est donc le pari que nous faisons avec le langage PEPLOM que nous allons décrire dans le chapitre suivant.

Chapitre IV

PEPLOM : un langage de programmation pour bases de données

Les barrières étaient moins hautes, sa conscience était plus large, sa passion devenait plus forte. Aussi marcha−t−il dans sa carrière de gloire, de travail, d'espérance et de misère avec la fureur d'un homme plein de conviction.

Honoré de Balzac − "La recherche de l'absolu"

De tous les prototypes de SGBD orientés objet, et notamment ceux ayant débouché sur des produits, il n'y a en aucun qui remette profondémment en cause l'aspect langage de programmation. Tous s'appuient sur des langages existants qu'ils étendent ou font coopérer avec leur couche SGBD, la contrainte, pour tirer pleinement parti de cette approche, étant de rester compatible avec le langage d'origine.

Nous nous proposons de définir un langage dans le but d'avoir une meilleur intégration des aspects langages et base de données. Les objectifs principaux sont :

- Supprimer le dysfonctionnement occasionné par la manipulation des données de la base et des données temporaires. La solution est alors d'offrir le même système de types pour les deux catégories de données. Elles sont alors manipulées de la même manière (le fait qu'une donnée soit persistante ou non est transparent à l'utilisateur).
- Introduire dans le langage tous les concepts base de données de manière orthogonale de façon à ne pas perturber le programmeur par une multitude de cas particuliers dépendants du contexte d'utilisation. Cet aspect est primordial surtout dans le cas de langages orientés objet où certaines notions sont assez complexes à appréhender (sous−typage, héritage, polymorphisme, etc...). Les fonctionnalités concernées sont essentiellement la prise en compte de la persistance et l'introduction de la notion de collection dans le système de types (constructeurs "dynamiques" du type ensemble ou liste).
- Supprimer le dysfonctionnement entre le langage procédural et le langage de requêtes permettant d'opérer des accès associatifs aux données. Le problème est résolu ici en généralisant les expressions du langage aux expressions ensemblistes.

Elles peuvent être construites à l'aide de fonctions génériques (implantant des opérateurs algébriques) qui sont à la base de l'expressions d'accès associatifs.

- Avoir le langage le plus sûr possible pour garantir au maximum la sécurité des données de la base. Généralement, l'espace où sont gérées les données de la base est isolé de l'application qui n'accède à celles−ci qu'à travers des requêtes qui offrent une grande sûreté de manipulation notamment dans le cas des SGBD relationnels. Il faut donc que le langage contrôle le plus possible les notions qui permettraient la corruption de l'espace des données de la base (dépassement des bornes d'un tableau, manipulation explicite de pointeurs, etc...).

Ces différentes considérations nous ont donc conduit à la définition du LPBD orienté objet PEPLOM (PErsistent Programming Language for Object Management). L'approche que nous avons choisi est pragmatique car nous ne voulons pas imposer un langage complètement nouveau, ni même couvrant les fonctionnalités d'entrées/sorties (couvert par la coopération avec C ou C++). Nous sommes donc partis du langage C++[102] duquel nous n'avons retenu, dans un premier temps, que les notions qui nous paraissaient compatibles avec notre volonté d'intégration et de sécurité. La principale restriction apportée pour cela est la suppression des pointeurs du système de types. Dans une seconde phase nous avons ajouté les concepts base de données. Nous avons par exemple étendu le système de types aux collections (ensemble et liste). La persistance est totalement orthogonale au système de types ce qui était un des principaux objectifs.

L'ensemble des données accessibles par une application est différent suivant une approche système ou une approche langage. Dans le cas d'un SGBD, ce contexte est la base de données offrant un certain nombre de points d'entrée qui sont des noms externes définis dans le schéma (par exemple les relations pour les SGBD relationnels). Dans le cas d'un langage, ce contexte est défini par un ensemble de variables locales et globales. Ces dernières sont généralement définies dans des fichiers (maintenus par le programmeur) qui servent de support à la modularité comme par exemple dans C++. Dans PEPLOM, nous avons voulu garder une approche purement langage. Nous avons ainsi introduit le module comme un composant du schéma qui sert de support à la définition du contexte. Ce contexte se compose de variables globales qui peuvent être définies persistantes (points d'entrée dans la base de données) ou non (contexte temporaire). Nous verrons que le module est aussi un composant réutilisable à un autre niveau que les types.

Par ailleurs, le fait que le langage proposé soit persistant n'est pas neutre dans la manière dont le langage va être géré (organisation de la compilation). En effet, les types définissent le format des données de la base. Ces données n'ont donc un sens que si nous connaissons leur type. Il est donc essentiel que ces définitions soient contrôlées par le système comme c'est le cas pour les SGBD relationnels où celles−ci sont gérées sous la forme d'une métabase. Le problème est néanmoins plus compliqué dans le cas d'un LPBD car ces définitions ne sont pas seulement des définitions de structures de données mais aussi de fonctions ou procédures qui les manipulent. C'est pour cette raison que l'utilisation du langage ne se fait pas seulement à travers un compilateur mais à l'aide d'un environnement d'outils intégrés. La mise au point de cet environnement est le but du projet Aristote[11] dans lequel s'insère ce langage.

La première partie du chapitre fait une présentation générale des principaux concepts du langage PEPLOM et de son mode d'utilisation. Nous présentons ensuite son système de types en en donnant une définition formelle. Puis nous décrivons la notion de module qui introduit notamment la persistance dans le langage. Une section particulière est ensuite dédiée à la description des opérateurs algébriques permettant d'effectuer des accès associatifs sur les collections. La conclusion résume enfin les différents concepts présentés.

IV.1 Présentation générale

Le langage PEPLOM est un LPBD orienté objet principalement inspiré du langage C++. C'est donc un SGBD dont le schéma contient des définitions de données mais aussi de traitements. Le schéma est une partie importante d'un SGBD. Il n'est pas possible de se contenter, pour le matérialiser, d'un ensemble de fichiers gérés par l'utilisateur comme c'est le cas pour les langages usuels tels que C, Pascal ou bien C++. Le schéma contient dans le cas de PEPLOM à la fois des définitions sur les données de la base mais aussi sur les programmes qui les manipulent.

En ce qui concerne les données, un atout important du langage, que nous avons voulu récupérer des SGBD relationnels, est la séparation entre leurs définitions conceptuelles et physiques (structures d'implantation). C'est d'ailleurs généralement des utilisateurs de qualifications différentes qui gèrent les deux aspects. Le programmeur s'occupe exclusivement du modèle conceptuel des données qu'il manipule. Le modèle physique (schéma interne) concerne essentiellement la définition de structures adaptées aux contextes d'utilisation des collections. Il offre la possibilité d'ajuster ("tuning") les structures dans le but d'améliorer les performances. La définition de ce schéma interne est dévolue à l'administrateur de la base.

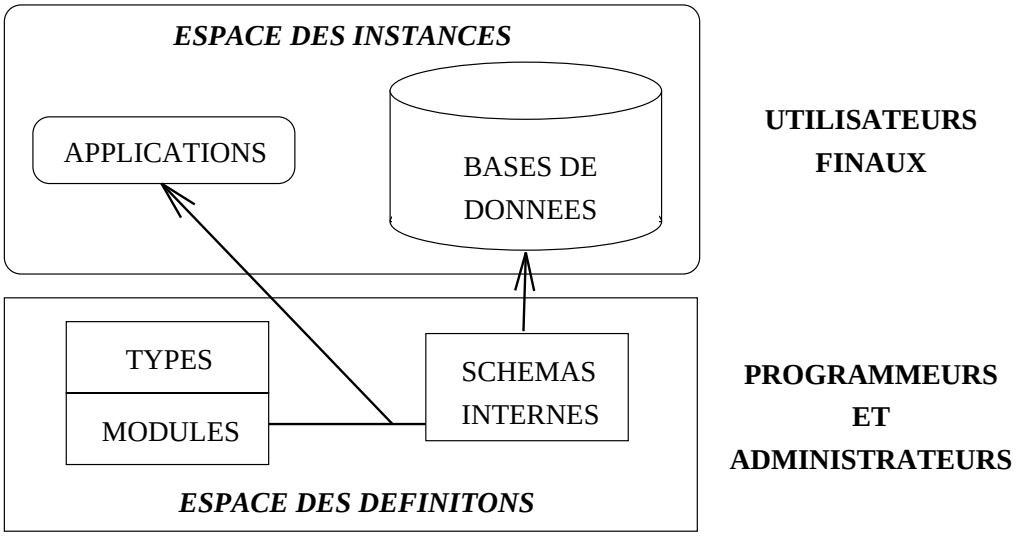


Fig. 4.1 : Modèle et organisation du langage PEPLOM

Les définitions conceptuelles des données sont réalisées à travers les types qui se divisent en deux catégories : les types concrets (générateurs de valeurs) et les types abstraits (générateurs d'objets abstraits). La description physique (bases physiques dans la figure Fig. 4.1) concerne évidemment les instances générées à partir de ces types mais aussi d'autres instances générées par des variables globales définies dans les modules. Les modules, comme nous l'avons dit précédemment, définissent le contexte global d'une application. C'est exclusivement à partir de ceux−ci que la persistance est introduite. Cela permet d'avoir des types abstraits beaucoup plus réutilisables de ce point de vue. Les types sont en effet indépendants de l'environnement persistant, le contraire aurait représenté une contrainte forte.

Les modules, qui s'appuient sur les définitions de types, offrent un deuxième niveau de réutilisation. Ils permettent en effet de mettre en commun pour plusieurs applications des instances ainsi que des opérations sur celles−ci (aspect extensionnel) alors que les types factorisent des structures de données et des comportements (aspect intensionnel).

A partir de toutes ces définitions, nous pouvons alors construire des applications PEPLOM exécutables. Avec des langages usuels, la cohérence de telles constructions est souvent complexe à gérer pour le programmeur (utilisation de fichier *Makefile*). Ce problème est ici réglé par l'environnement associé au langage et plus particulièrement grâce au schéma qui mémorise toutes les dépendances entre les définitions. Dans ce contexte, pour générer une application exécutable, l'utilisateur spécifie seulement un module contenant le point d'entrée de l'application et une définition de schéma interne spécifiant le format des données. L'environnement s'occupe alors de calculer la fermeture transitive de toutes les définitions nécessaires à la production du code de cette application.

La production d'un programme PEPLOM se déroule donc en deux phases. La première consiste à définir les différents composants du schéma de l'application et la deuxième à générer le code exécutable pour cette application. La première phase est matérialisée par un compilateur incrémental qui intègre les nouvelles définitions dans le schéma (sous la forme d'un code intermédiaire) et maintient à tout instant l'état de validité de chacune d'elles. Nous aurions pu, comme pour le système O2, définir un compilateur incrémental gérant du code exécutable pour chaque définition du schéma. Cette approche est beaucoup plus lourde à gérer et peu compatible avec notre choix de séparation des définitions conceptuelles et physiques des données. Cette séparation implique que le code d'un type abstrait (code composé par l'ensemble des méthodes qui lui sont attachées) peut être différent suivant sa description physique. La génération de code s'effectue donc dans une seconde phase qui rassemble les définitions à générer et génère le code exécutable correspondant. Le programme généré est alors exécutable comme n'importe quel programme Unix.

Après ce bref aperçu du langage PEPLOM et de la manière dont il est utilisé par le programmeur, nous allons décrire son système de types qui est un élément primordial du langage puisqu'il définit la puissance de modélisation des données manipulées.

IV.2 Le système de types de PEPLOM

La système de types du langage PEPLOM a été construit à partir de celui de C++. La principale divergence concerne la prise en compte de la notion de pointeur. Dans C++, les adresses, c'est à dire les points d'entrée dans l'espace d'adressage des objets, sont manipulables explicitement par l'utilisateur. Cet aspect combiné avec la possibilité de forcer le type des données désignées par ces pointeurs met en cause la sûreté des données d'une manière difficilement acceptable dans un contexte base de données. Ces deux notions ont donc été supprimées. Le partage de données, qui était auparavant assuré par les pointeurs, l'est dorénavant par les identificateurs d'objets qui sont gérés par le système (non utilisables dans le langage).

La notion de classe de C++ a été remplacée par celle de type abstrait ceci dans un but de clarification. Les types abstraits PEPLOM reprennent nombre d'aspects des classes C++ mais il y a suffisamment de divergences dans leur définition pour vouloir différencier les deux syntaxiquement. Nous pouvons citer parmi ces divergences la séparation des méthodes en deux catégories (fonctions et procédures) ou encore la définition de la structure de l'objet généré par ce type.

Enfin, les règles de conformité de types (définissant les substitutions de données possibles) de PEPLOM sont les mêmes que dans C++ pour la plupart des concepts communs (sous−typage, héritage, polymorphisme).

IV.2.1 Les types de base et les types construits

Comme dans le modèle du système O2[68], nous distinguons deux catégories d'objets qui sont les valeurs et les objets abstraits (cf. II.2.1.4). Nous utilisons dans la suite le terme d'objet dans le sens d'objet abstrait. Les valeurs sont des entités non partageables et sont générées à partir de types de base qui peuvent être combinées à l'aide de constructeurs.

Les types de base

Les types de base sont principalement ceux du langage C (ou C++) auxquels ont été ajoutés quelques autres. Nous avons les types qui définissent des entiers signés et non signés : char, short, int, long, unsigned char, unsigned short, unsigned int et unsigned long. Différents opérateurs arithmétiques permettent de construire des expressions du langage à partir de ces valeurs atomiques. Ces opérateurs sont des fonctions liées par des relations de polymorphisme paramétrique. Prenons par exemple l'opérateur d'addition entière "+" (fonction "add") :

```
add (char, char) → char
add (short, short) → short ...
```

De plus, les expressions arithmétiques peuvent faire intervenir des types entiers hétérogènes comme par exemple "a(short) + b(long) − c(unsigned char)". Ce genre de problème est résolu en C++ par le programmeur en appliquant des cœrcitions explicites ("cast" C). Dans PEPLOM, c'est le compilateur qui s'occupe d'appliquer des opérateurs de cœrcition. Ces opérateurs peuvent s'appliquer lorsque nous avons une relation d'inclusion

des domaines définis par les types de base concernés. Par exemple, nous avons "unsigned char ⊂ long". Nous pouvons définir un opérateur de cœrcition cast[TO,TV] (VAL) où [TO,TV] indique le type de cœrcition à effectuer (du type d'origine TO vers le type voulu TV) sur la valeur VAL. Il retourne une valeur de type TV. Dans l'exemple de l'expression ci−dessus, le compilateur doit effectuer la transformation suivante (le type de la variable se trouve entre parenthèse à droite de celle−ci):

```
a(short) + b(long) − c(unsigned char) devient
sub (add (cast[short,long](a),  b),
    cast[unsigned  char,long](c))
```

Parmi les autres types de base, il y a les réels de différentes précisions : float (simple précision) et double (double précision). Comme ils servent aussi à la construction d'expressions arithmétiques, des règles de cœrcition s'appliquent aussi permettant par exemple de passer d'un réel simple précision vers un double ou d'un entier vers un réel.

Nous avons introduit en plus le type booléen (bool) qui permet de construire des expressions booléennes. Les opérateurs booléens s'appliquent seulement sur des expressions booléennes non pas, comme dans C++, à n'importe quelle expression (par exemple des expressions arithmétiques). Le domaine correspondant au type booléen contient exclusivement deux valeurs : **`true`** et **`false`**.

Les types construits

A partir des valeurs atomiques générées par les types de base, des objets et références d'objets générés par les types abstraits (cf. IV.2.3), le langage permet de construire les valeurs complexes. Elles sont construites à l'aide des constructeurs disponibles dans le système de types. Ces constructeurs sont tout d'abord ceux du langage C : la structure (n−uplet), l'union et le tableau. La structure et le tableau ont la même sémantique qu'en C alors que la sémantique de l'union est étendue. En effet, prenons la définition suivante :

```
union {
      int cas1;
      char cas2[20];
      } var;
```

La variable `var` désigne une valeur qui est soit un entier, soit un tableau de caractères. PEPLOM gère automatiquement le champ auquel correspond la valeur désignée. L'utilisateur peut alors consulter à travers quel champ de l'union la valeur est manipulable avec le champ générique **`case`** :

```
var.cas2[7] = 'Q';
...
if (var.case == cas1)
    var.cas1 += 10;
else
    var.cas2[4] = 'Z';
...
```

Si l'utilisateur tente de manipuler la valeur désignée à travers le mauvais champ, le langage déclenche alors une exception[66].

A ces constructeurs, le langage PEPLOM ajoute l'ensemble et la liste. Ils sont pris en compte au niveau d'une définition de type comme les tableaux desquels ils sont

naturellement proches. Supposons que nous voulions définir un tableau `tab` de 20 tableaux de 50 entiers. L'expression en PEPLOM (la même qu'en C) serait alors la suivante :

```
int  tab[20][50];
```

Supposons maintenant que nous voulions que ce tableau soit un ensemble de tableaux de 50 entiers. Nous pouvons observer la similitude de l'utilisation du constructeur ensemble avec celle du constructeur tableau dans la définition suivante :

```
int  tab{}[50];
```

Le constructeur liste, qui s'exprime par "<>" (au lieu de "{}" pour l'ensemble), se comporte de la même manière que le constructeur ensemble. Tous les constructeurs peuvent être combinés dans n'importe quel ordre et de façon récursive comme dans le langage C.

Un certain nombre d'opérateurs de base sont attachés à chacun des constructeurs. Pour les structures et les unions, nous avons l'opérateur "." qui permet la sélection d'un champ, sachant qu'il permet de sélectionner le champ générique **`case`** dans le cas des unions. L'accès au $i^{ième}$ élément d'un tableaux se fait comme en C avec l'opérateur "[*i*]", sachant que l'indice i varie de 0 à n−1 où n est le nombre d'éléments. Pour l'ensemble, les opérations de base sont l'union (opérateur "+"), l'intersection (opérateur "*"), la différence (opérateur "−") et l'opération de calcul de la cardinalité (fonction générique "**`size`**(...)" qui retourne un entier de type long). Enfin, pour les listes, nous avons la concaténation (opérateur "+"), le calcul du nombre d'éléments (fonction générique "**`size`**(...)") et l'opérateur "[*i*]" permettant l'accès au $i^{ième}$ élément comme pour le tableau.

De plus, nous avons introduit des notions supplémentaires pour la manipulation de liste. Il s'agit essentiellement du curseur et des instructions qui lui sont associées. Le curseur agit comme un pointeur sur un élément d'une liste. Prenons par exemple le parcours du dernier au premier élément d'une liste d'entiers auxquels nous voulons ajouter 10 :

```
int liste<>, cursor c;
...
c = last (liste);
while (c.valid)
    {
    c.value += 10;
    c = prev (c);
    }
```

Un curseur est une variable spéciale (qui ne peut être déclarée que dans un bloc de code). Elle désigne une structure de deux champs génériques :

- **`valid`** est un booléen. Il signale si le curseur désigne un élément d'une liste ou non.
- **`value`** est du type déclaré pour le curseur (dans l'exemple précédent int). Il permet d'accéder à la valeur de l'élément désigné dans la liste et de modifier cet élément si nécessaire. Dans le cas où **`valid`** est faux, un accès au champ **`value`** provoque une exception.

Quatre fonctions génériques permettent la manipulation des curseurs : **`first`** et **`last`** qui rendent respectivement des curseurs sur le premier ou le dernier élément d'une

liste, et **next** et **prev** qui rendent respectivement des curseurs sur l'élément suivant ou précédent l'élément désigné par un curseur. Enfin, une instruction d'insertion positionnelle permet d'insérer un nouvel élément avant ou après l'élément désigné par un curseur (par exemple "**insert** (**next** c, 26);"). Nous verrons par la suite (cf. IV.4) qu'il y a d'autres instructions ou fonctions permettant notamment de manipuler les ensembles et les listes de façon associative.

Comme dans C, nous offrons aussi le constructeur **enum** qui permet de construire un domaine par extension. Il s'agit de domaine de symboles. Nous avons étendu la sémantique de ces types énumérés en autorisant la définition de tels domaines par union de domaines existants. Prenons l'exemple suivant :

```
typedef enum
    {Mark,Franc,Florin,Lire,Livre,...}   EuroDevise;
typedef enum  {DollarUS,DollarCan}
    +  EuroDevise  MonnaieOccidentale;
```

Le premier type (EuroDevise) définit le domaine des devises européennes et le deuxième (MonnaieOccidentale) le domaine des monnaies occidentales comme une union du premier domaine et du domaine énuméré définissant le Dollar US et le Dollar canadien. Les types énumérés définissent donc aussi des valeurs qui sont des symboles. La fonction **symb**(*symbole*) permet de récuper la chaine correspondant au symbole (par exemple **symb**(DollarUS) → "DollarUS").

Le dernier type générique du système de types de PEPLOM permet la définition de valeurs qui sont des chaînes de caractères. Par exemple, nous pouvons définir un nom de la manière suivante :

```
string(20) Nom;
...
Nom = "Durant";
Nom[5] = 'd';
```

La variable Nom est donc une chaîne contenant au maximum 20 caractères. Les opérateurs de base de manipulation des chaînes sont la fonction générique qui donne leur taille ("**size**(*chaîne*)"), la concaténation (opérateur "+") et l'accès au $i^{ième}$ caractère (opérateur "[*i*]"). Les deux dernières opérations peuvent provoquer des exceptions lorsque respectivement la capacité de la chaîne d'accueil est dépassée ou l'indice du caractère accédé est en dehors des bornes possibles.

Nous allons maintenant décrire dans la section suivante quelles sont les règles de conformité qui s'appliquent entre les types que nous appelons types concrets.

IV.2.2 Les types concrets

Nous avons vu dans la section précédente les différents éléments qui permettent de construire des types dans le langage PEPLOM. Ces types définissent des valeurs alors que nous verrons dans la section suivante les types abstraits qui engendrent eux des objets abstraits. Nous nommons les types générant des valeurs les types concrets par opposition aux types abstraits. Il est possible de leur associer un nom représentant ce type. Ces types peuvent être définis au moment où ils servent à la définition d'une variable désignant une

valeur. Ils peuvent aussi être nommés comme en C ou C++. Ce nom est alors un représentant du type auquel il est associé et sert essentiellement de raccourci lors des définitions de variables. Prenons un exemple de telles définitions :

```
typedef struct
    {
    int  Numero;
    enum {Rue, Boulevard, Place, Cours} Voie;
    string(30)  NomRue;
    string(20)  Ville;
    }
    Adresse;
```

Nous pouvons observer que le type `Adresse` définit un type structuré dont un des champs définit son propre type énuméré. Il est important, à partir de ces types, de définir les règles de conformité implicites ou explicite qui peuvent les lier. Elles définissent les différentes substitutions possibles d'un type par un autre.

IV.2.2.1 La conformité

Les règles de conformité ou de sous−typage s'appliquant aux types concrets sont exclusivement des règles de conformité structurelle. C'est le cas aussi pour les types concrets nommés pour lesquels aucune sémantique spécifique n'est associée au nom. Nous disons qu'un type T_1 est conforme à un type T_2 si une valeur générée à partir de T_1 peut se substituer à une valeur de type T_2. Nous notons cette relation de conformité "≤" et nous avons alors $T_1 \leq T_2$. Cette relation est définie de la manière suivante par rapport aux types concrets :

Définitions préalables :

- *les T dénotent des types concrets,*
- *struct{T_1 N_1, ..., T_n N_n} dénote une structure de n attributs N_i de type T_i,*
- *T[n] dénote un tableau de n éléments de type T,*
- *T{} et T<> dénote un ensemble et une liste d'éléments de type T et*
- *enum{S_1, ..., S_n} dénote une énumération de symboles S_i.*

- $T \leq T$.
- $\text{struct}\{T_1\ N_1;\ ...;\ T_n\ N_n;\} \leq \text{struct}\{T_1'\ N_1';\ ...;\ T_m'\ N_m';\} \Rightarrow m = n$ et $\forall\ i \in 1..n, T_i \leq T_i'$.
- $\text{union}\{T_1\ N_1;\ ...;\ T_n\ N_n;\} \leq \text{union}\{T_1'\ N_1';\ ...;\ T_m'\ N_m';\} \Rightarrow m = n$ et $\forall\ i \in 1..n, T_i \leq T_i'$.
- $T[n] \leq T'[m] \Rightarrow m = n$ et $T \leq T'$.
- $T\{\} \leq T'\{\} \Rightarrow T \leq T'$.
- $T<> \leq T'<> \Rightarrow T \leq T'$.
- $\text{enum}\{S_1, ..., S_n\} \leq \text{enum}\{S_1', ..., S_m'\} \Rightarrow \forall\ i \in 1..m, \exists\ j \in 1..n / S_j = S_i'$.

L'intérêt de ces règles est d'assurer une équivalence structurelle entre des types conformes, ce qui permet d'envisager un calcul statique de l'adressage pour les constructeurs "statiques" du langage (cf. II.3.1). Nous allons voir, dans la section IV.2.3 qui suit, une autre catégorie de types qui engendrent des objets abstraits et définir comment ceux−ci s'intègrent dans la définition de la relation de conformité.

IV.2.2.2 Les littéraux

Les littéraux du langage se composent des constantes des types de base et des littéraux construits à partir des constructeurs du langage. Nous nous intéressons ici essentiellement à la seconde catégorie. Pour plus de lisibilité, nous avons changé l'expression des littéraux de n−uplet par rapport à C. Au lieu de les exprimer par "{champ$_1$, ..., champ$_n$}", nous les notons "**struct**{champ$_1$, ..., champ$_n$}".

Il n'est pas possible d'exprimer les littéraux d'union en C. En PEPLOM, nous pouvons le faire de la manière suivante :

```
union {
    int Entier;
    string(20) Chaine;
    } var;
...
if (var == union{Chaine, "Hello world."}) ...
```

Nous voyons qu'un littéral d'union se compose toujours de deux champs. Le premier correspond au nom du champ sélectionné c'est à dire à la valeur de "`var.`**`case`**". Le second correspond à la valeur du champ sélectionné c'est à dire dans notre exemple à la valeur de "`var.Chaine`". L'exemple montre, par ailleurs, un littéral de chaîne de caractères (chaîne "Hello world.").

Enfin, les littéraux de tableau s'expriment de la même façon que dans le langage C, c'est à dire "{valeur$_1$, ..., valeur$_n$}". L'expression est aussi la même pour les littéraux d'ensembles et de listes.

IV.2.3 Les types abstraits

Un type abstrait permet d'instancier des objets et des références (que nous considérons comme des valeurs) vers ces objets. Il définit une structure de données qui est encapsulée par des méthodes. Nous avons vu dans le chapitre précédent qu'une encapsulation totale n'était pas intéressante dans un environnement base de données. Cela supprime effectivement toute possibilité de définir des méthodes d'accès sur la valeur pouvant être utilisées pour optimiser des accès associatifs aux ensembles d'objets.

En fait, l'aspect qui nous parait essentiel, par rapport à l'encapsulation, est que l'objet ne puisse évoluer qu'à travers les méthodes attachées à son type. Elles sont alors sensées le faire évoluer d'un état considéré comme cohérent par le programmeur de ce type abstrait vers un autre état cohérent. Cette approche est un prolongement naturel de la notion de base de données, qu'une transaction fait évoluer d'un état cohérent à un autre, vers un maintient de la cohérence des données (cohérence sémantique) à un niveau microscopique.

Nous pouvons décomposer la définition d'un type abstrait en quatre parties. Il faut définir les liens d'héritage qui existent entre le type défini et d'autres types abstraits. Un type peut alors hériter de plusieurs types (héritage multiple). Nous devons ensuite définir la structure de la valeur associée aux objets générés par le type. C'est fait en définissant le type de la variable générique **`value`** associée au type abstrait. Il ne reste plus alors qu'à définir les méthodes permettant de manipuler les objets générés. Ces méthodes sont divisées en deux catégories : les fonctions et les procédures. Les premières permettent de calculer un résultat lié à la valeur de l'objet. Elles n'ont aucun effet de bord sur le contexte externe à leur corps. Les secondes représentent le seul moyen pour modifier la valeur de l'objet auquel elles s'appliquent. Prenons un exemple de définition de types abstraits :

```
typedef abstract personne
    {
    struct {
        string(30) Nom;
        string(20) Prenoms<>;
        private:
        ulong DateNais;
        personne Parents{} inv value.Enfants;
        public:
        personne Enfants{} inv value.Parents;
        personne Epoux inv value.Epoux;
        struct {
            int Numero;
            string(80) Rue;
            ville Ville;
            } Adresse;
        } value;
    functions {
        uchar Age();
        personne(string(30), string(20), ulong);
        }
    };

typedef abstract imposable
    {
    struct {
        enum {Personne, Societe, Association}
            TypeEntite;
        } private value;
    functions {
        private:
        imposable(
            enum {Personne, Societe, Association});
        public:
        double Impot();
        }
    };

typedef enum {Ingenieur, Manager, Secretaire, ...}
    Position;
```

```
typedef abstract employe : personne, imposable
    {
    struct {
        Position  Position;
        double  Salaire;
        } value;
    functions {
        double Impot();
        employe(string(30), string(20), ulong);
        }
    procedures {
        Augmenter(float taux);
        ~employe();
        }
    };
```

Fig. 4.2 : Exemples de types abstraits

L'exemple défini ci−dessus montre trois types abstraits. Le premier décrit des objets représentant une personne, le second des objets représentant des entités imposables (qui peuvent être des personnes physiques comme des personnes morales) et le troisième des objets qui représentent des employés. Ces employés sont des personnes qui sont imposables puisque salariées. Le type `employe` hérite donc à la fois du type `personne` et du type `imposable`.

Nous allons à présent passer en revue les différentes notions utilisées dans cet exemple par rapport aux trois aspects de la définition d'un type abstrait : structurel, comportemental et sémantique (héritage).

IV.2.3.1 Définition de la valeur associée aux objets d'un type abstrait

C'est à travers la variable générique **`value`** qu'est définie la structure de la valeur associée aux objets d'un type abstrait. Par défaut, cette valeur est publique, c'est à dire qu'elle peut être accédée depuis l'extérieur mais exclusivement en lecture. Si nous voulons protéger tout ou partie de cette valeur des lectures externes, il faut la déclarer totalement privée comme pour le type `imposable` ou en partie comme les champs `DateNais` et `Parents` du type `personne`.

Cette distinction entre l'objet et sa valeur est très intéressante au niveau du langage car ça permet notamment de comparer des objets à deux niveaux : comparaison d'objet (même identificateur) ou comparaison de valeur (même contenu). La première est souvent définie comme l'identité d'objet et la seconde comme l'égalité superficielle ("shallow equality"). Dans notre approche, les deux s'expriment à travers le même opérateur définissant ainsi une unique sémantique pour l'égalité. Nous définissons donc l'égalité d'objets comme l'égalité de leurs identificateurs (au même niveau que l'égalité de valeur). Ces identificateurs sont en fait eux−mêmes des valeurs.

De plus, cette valeur définit les liens entre les différents objets des différents types abstraits. Dans PEPLOM, on distingue trois types de liens[28][60]. Le premier est la

référence non partageable ou lien composite. Il existe dans ce cas une dépendance existentielle entre l'objet référencé et l'objet référençant. L'objet référencé ne peut de plus jamais être référencé par un autre objet. Il n'y a pas d'exemple de ce type de lien ci−dessus ; il se différentie du lien partageable par l'ajout du mot clé **`own`** précédent le nom qui désigne l'objet non partageable référencé (sémantique différente liée à l'objet référencé).

Le deuxième type correspond aux références partageables. L'objet (objet partageable) référencé par ce type de lien peut l'être par d'autres objets. Nous avons un exemple de ce type de lien avec le champ **`value`**`.Adresse.Ville` dans le type `personne`.

Enfin, nous avons le lien symétrique qui permet de maintenir des associations binaires de type 1:1, 1:N ou M:N. Ce type de lien n'est pas compatible avec le lien composite, ce dernier ne pouvant être partagé. Il correspond à un couple de référence partageable que le langage maintient automatiquement ; le lien inverse est mis à jour lorsqu'une des deux références est affectée. Le lien symétrique Epoux/Epoux dans le type personne est un exemple d'association 1:1 alors que Parents/Enfants est une association M:N.

IV.2.3.2 Définition des méthodes (fonctions et procédures)

Les méthodes (leur signature au niveau du type abstrait) sont définies dans deux parties différentes. La première contient les définitions de fonctions qui ne modifient pas le contexte dans lequel elles s'exécutent (hormis son propre contexte local). Cette partie est donc introduite par le bloc de définition "**`functions`** {...}". Au niveau du code de ces méthodes, pour vérifier sémantiquement que c'est bien une fonction, les contraintes que nous avons imposé sont d'une part que les seules méthodes qui peuvent être appliquées sont d'autres fonctions et qu'une fonction ne peut pas créer de nouveaux objets. La seconde partie contient les définitions de procédures dont le rôle est de modifier l'objet sur lequel elles s'appliquent. Elle est introduite par le bloc de définition "**`procedures`** {...}". Dans chacun de ces blocs de définition, les méthodes déclarées peuvent être soit publiques soit privées (elles sont publiques par défaut). Nous observons un exemple d'expression de ces contraintes dans le bloc **`functions`** du type `imposable`.

Les paramètres des procédures aussi bien que ceux des fonctions sont passés exclusivement par valeur. D'un autre côté, le résultat d'une fonction ou d'une procédure peut être de n'importe quel type PEPLOM (cela peut par exemple être un ensemble). Enfin, il n'y a aucun moyen d'exprimer de la persistance ni dans les définitions de types abstraits ni dans le code de leurs méthodes. Cette contrainte, qui peut paraître forte, nous semble très intéressante car elle conduit les types abstraits à être beaucoup plus réutilisables (les objets générés pouvant être utilisés indépendamment dans un contexte persistant ou temporaire). Nous sommes conscients que toutes ces contraintes définissent le cadre d'une certaine méthode de programmation. C'est un aspect que nous voulons prospecter à travers ce nouveau langage.

Comme en C++, nous avons deux méthodes particulières qui sont le constructeur (il peut y en avoir plusieurs dans une définition de classe C++ mais un seul en PEPLOM) et le destructeur. Elles sont implicitement appelées respectivement lors de la création ou de la destruction d'un objet de ce type. Le destructeur doit obligatoirement être déclaré comme une procédure puisque le seul rôle qu'il peut remplir est d'effectuer des modification d'objets

référencés par l'objet détruit étant donné qu'il n'a aucun paramètre. De plus, il peut être déclaré privé ce qui signifie alors que les objets de ce type abstrait ne peuvent être détruits explicitement (instruction **`delete`**).

Le cas du constructeur est plus complexe dans PEPLOM. Tout d'abord, comme en C++, il est possible de déclarer des paramètres dans sa signature. Son rôle est d'initialiser l'objet créé par l'opérateur **`new`** appliqué au type abstrait. Le constructeur peut être déclaré comme une fonction ou comme une procédure. La sémantique d'une fonction "constructeur" est différente d'une fonction PEPLOM normale. En effet, elle modifie la valeur de l'objet créé puisqu'elle l'initialise. Nous considérons que c'est effectivement une fonction si les seuls effets qu'elle produit concernent l'objet créé (pas d'effet de bord sur d'autres objets du contexte de l'application). Dans le cas contraire, le constructeur est une procédure. C'est notamment le cas lorsqu'il initialise un lien symétrique. Le fait qu'un constructeur d'objet puisse être une fonction est intéressant car il pourra être utilisé le cas échéant pour qu'une expression ensembliste crée de nouveaux objets (dans le cadre des types abstraits du schéma). Enfin, si le constructeur est déclaré privé dans le type, ce dernier ne peut pas servir à la création d'objets.

IV.2.3.3 Définition de relations d'héritages

PEPLOM offre la possibilité de définir des liens d'héritage multiple entre les types abstraits. La relation d'héritage définit une règle de conformité par nom. Elle engendre un graphe orienté sans cycle ("DAG") qui forme un treillis de types. Pour compléter la relation de conformité entre types, nous devons ajouter la règle suivante aux règles définies dans IV.2.2 :

Définition préalable :

– *T_1 et T_2 sont des types abstraits.*

- $T_1 \leq T_2 \Leftrightarrow T_1$ hérite de T_2.

Par contre, lorsqu'un type abstrait hérite d'un autre, sa définition est soumise à différentes contraintes. Tout d'abord, le type de la valeur ne peut être surchargé (ajout d'attributs) que s'il s'agit d'un n−uplet ou d'une union. Pour ces deux derniers cas, nous n'avons pas de surcharge du type d'un attribut. Prenons l'exemple suivant :

```
typedef  abstract t1
{
struct {
    ta1 A1;
    } value;
}

typedef  abstract t2
{
struct {
    ta2 A1;
    } value;
}
```

```
typedef  abstract t3 : t1, t2
{
struct {
    ta3 A1;
    } value;
}
```

Dans les trois types abstraits définis dans l'exemple, l'attribut `A1` défini dans chacun d'eux représente un attribut différent. Cela implique que lorsqu'on utilise l'attribut A1 dans une méthode de `t3`, son nom est ambigü (trois attributs de même nom). Il n'est donc pas possible de déterminer lequel des trois doit être utilisé. La résolution de cette ambiguïté s'effectue comme en C++ en préfixant l'attribut réellement utilisé par le type dans lequel il a été défini. Par exemple, l'accès à l'attribut `A1` du type `t2` s'effectue par `t2::A1`.

Les méthodes définies dans les types peuvent être surchargées. Elles peuvent l'être au niveau de leur signature où le type du résultat peut être spécialisé et les types des paramètres généralisés (peu utile : nous pouvons nous limiter à l'égalité de type comme dans C++) correspondant à la règle définie par Cardelli dans [26]. Elle est d'ailleurs en cours d'introduction dans la définition du standard du langage C++[69]. Cette règle de surcharge, que nous notons aussi ≤, peut se définir de la manière suivante :

Définition préalable :

– *M est une méthode définie dans les types abstraits T et T' où T ≤ T'.*

- $M(T\ recept, T_1\ P_1, ..., T_n\ P_n) \rightarrow T_{res}$
  $\leq M(T'\ recept, T_1'\ P_1', ..., T_m'\ P_m') \rightarrow T_{res}' \Leftrightarrow m = n,\ T_{res} \leq T_{res}'$ et $\forall i \in 1..n, T_i' \leq T_i$

Par ailleurs, une méthode déclarée comme une fonction ne peut pas être redéfinie comme une procédure ni l'inverse. Les méthodes peuvent, comme dans le cas des attributs, introduire des conflits de noms. Ils sont alors résolus de la même manière que pour les attributs. Outre la signature, le code de la méthode redéfinie peut être surchargé.

Enfin, dans la définition de leur code, les constructeurs et les destructeurs doivent appeler respectivement tous les constructeurs ou destructeurs définis dans les super−types de leur type abstrait. Par rapport à l'exemple précédent, en supposant que chaque type définisse un constructeur, le constructeur du type `t3` aurait le format suivant :

```
t3 (t31 p31, ..., t3n p3n) :
    t1 (p11, ..., p1s),
    t2 (p21, ..., p2t)
{
...
}
où {p11, ..., p1s} ⊆ {p31, ..., p3n} et
   {p21, ..., p2t} ⊆ {p31, ..., p3n}
```

Ces constructeurs sont évalués des constructeurs des plus grands super−types vers ceux des plus petits. Pour ceux appartenant à des super−types de même niveau (héritage multiple),

ils sont appelés dans l'ordre donné par l'utilisateur (ici `t1` avant `t2`). Pour les destructeurs, ils sont évalués dans l'ordre inverse des constructeurs.

IV.3 Les modules dans PEPLOM

Dans cette section, nous définissons la notion de module introduite dans le langage PEPLOM. Nous allons voir de quelle manière ceux−ci définissent le contexte d'exécution (persistant et temporaire) d'une application. La plus grande partie des opérations, qui leur sont associées, sont destinées à la communication avec l'utilisateur. De plus, la persistance est introduite par les modules ou leurs opération. Nous décrivons, dans cette section, le modèle de persistance défini par le langage. Nous verrons enfin comment est prise en compte la notion de vue sur les données.

IV.3.1 Rôle des modules

Ils définissent tout d'abord le contexte global d'exécution d'une application. Chaque module peut donc définir des variables. Elles forment le contexte global d'exécution et peuvent être des points d'entrée dans la base de données comme nous le verrons dans la sous−section suivante. Elles peuvent être déclarées publiques ou privées. Dans le cas où elles sont privées, elles sont totalement encapsulées par les opération qui sont définies par le module. Nous sommes ici très proches de la notion de type abstrait mais nous avons voulu un concept supplémentaire à travers lequel nous définissons un contexte de données réelles. En effet, les types abstraits n'expriment que d'un point de vue intentionnel une structure de données encapsulée par des méthodes. Comme pour les types abstraits, les variables publiques sont accessibles en lecture seulement depuis les autres modules qui les importent. L'exemple suivant montre quelques unes des fonctionnalités offertes par les modules (nous utilisons dans cet exemple des définitions de l'exemple de la figure Fig. 4.2) :

```
moddef  GestionDuPersonnel
{
ref employe persist Pdg;
private:
struct {
    Position  pos;
    float  salaire;
    } SalairesMin{} as
            {
            struct  {Ingenieur,  12000},
            struct  {Manager,  16000},
            struct  {Secretaire,  6000},
            ...
            };
int  NbEmbauches  =  0;

functions {
    float   SalaireMoyen(Position);
    }
procedures {
```

```
        Embaucher();
        int  GestionDuPersonnel(string argv<>);
        }
    };
```

Fig. 4.3 : Exemple d'un module

Le module définit ci−dessus nous montre tout d'abord trois définitions de variables. La première (`Pdg`) est publique et désigne un objet qui est un employé. Elle contient donc une référence vers cet employé et son contenu (la référence) est persistant. Les deux variables suivantes sont privées. La première (`SalairesMin`) est en fait une constante désignant un ensemble de structure représentant les salaires minimum par position. La troisième (`NbEmbauches`) est une variable désignant un entier qui représente le nombre d'embauches effectuées au cours de la session. Cette variable, non persistante, est initialisée à 0 à chaque nouvelle exécution de l'application généré à partir de ce module.

Le module permet par ailleurs la définition d'opérations qui sont comme les méthodes des types abstraits divisées en fonctions et procédures. Le comportement des deux blocs de définition correspondant est le même que pour les types abstraits. Il y a de plus une opération spéciale caractérisée par le fait qu'elle porte le même nom que le module dans lequel elle est définie (comme le constructeur pour le type abstrait). Le rôle de cette opération est de servir de point d'entrée pour l'exécution d'une application PEPLOM. C'est le cas lorsque le module concerné sert de base pour la génération d'une application (cf. IV.1). La signature de cette opération, qui peut être une fonction ou une procédure, est toujours la même : elle retourne un entier qui correspond au code de sortie du processus exécutant l'application PEPLOM et elle possède un seul paramètre qui est une liste de chaînes de caractères correspondant aux paramètres de lancement de l'application. Cette opération est équivalente au "main" de C ou C++.

Le dernier aspect qui n'est pas couvert par l'exemple de la figure Fig. 4.3 est l'organisation inter−modules. En effet, il est possible d'exprimer des relations d'importation entre modules. Un module peut alors importer plusieurs autres modules. Dans ce cas, celui−ci peut utiliser toutes les définitions publiques des modules qu'il importe. Cette relation s'exprime syntaxiquement comme la relation d'héritage entre les types abstraits (par exemple "**`moddef`** `M` : M_1, M_2, ..., M_n {...}" définit le module M qui importe les modules M_1 à M_n). Le graphe engendré par la relation d'importation est un graphe orienté dans lequel nous pouvons avoir des cycles.

IV.3.2 Modèle de persistance

Nous avons vu que la persistance est introduite dans le langage PEPLOM au niveau des modules. La politique de gestion de la persistance est à la fois explicite et implicite. Elle est implicite dans la mesure où un objet devient automatiquement persistant lorsqu'il est référencé par un autre objet persistant. Ce choix permet de maintenir la cohérence des liens inter−objets d'une session à l'autre car une référence temporaire n'a plus le même sens d'une

exécution d'une application à l'autre. Pour que cette règle de propagation crée de la persistance, il faut que cette persistance ait été introduite par ailleurs.

C'est l'expression explicite de la propriété qui permet de créer ces entités persistantes de base. Nous les nommons les racines de persistance. Elles sont au nombre de deux : les variables persistantes et les objets racine de persistance. Les variables persistantes déclarées au niveau des modules (variable `Pdg` de l'exemple de la figure Fig. 4.3) forment donc la première catégorie des entités générant la persistance.

L'autre catégorie est formée par les objets qui sont explicitement déclarés comme des racines de persistance. Cela peut être fait soit à leur création avec l'instruction **`pnew`** soit dynamiquement au cours de la vie de l'objet par l'instruction **`persist`**. Ces objets, comme les autres objets persistants, sont toujours accessibles à partir de l'extension persistante de leur type. Cela nous amène à avoir deux sortes d'objets persistants : ceux qui sont des racines de persistance et ceux qui ont été rendus persistants par propagation. La différence entre les deux est que les premiers ne peuvent être détruits qu'explicitement par l'utilisateur à l'aide de l'instruction **`delete`**. Les seconds peuvent l'être par le ramasse miettes (ou glaneur de cellules) si ceux−ci ne sont plus référencés par aucun autre objet (seuls les objets persistants sont concernés par le ramasse miettes). Enfin, il est possible de tester si un objet est ou non persistant (fonction générique **`persistent`**) et si c'est ou non une racine de persistance (fonction générique **`proot`**) ; ces deux fonctions rendent un booléen.

IV.3.3 Les extensions de types abstraits

Le langage PEPLOM permet d'accéder à tout moment à l'extension d'un type abstrait. Elle est composée de tous les objets de ce type qui ont été créés soit par l'instruction **`new`** (création d'un objet temporaire) soit par l'instruction **`pnew`** (création d'une racine de persistance).

Cette extension, qui est gérée automatiquement par l'environnement d'exécution du langage, est en fait composée de deux parties accessibles indépendamment : l'extension des objets temporaires qui s'exprime par **`extension`**(*type abstrait*) et l'extension des objets persistants qui s'exprime par **`pextension`**(*type abstrait*). L'ensemble de tous les objets d'un type abstrait T accessibles durant une session est donc l'union des extensions temporaires et persistantes qui s'exprime par "**`extension`**(T) + **`pextension`**(T)".

De plus, il est possible d'exprimer l'extension contenant tous les objets d'un type et de tous ses sous−types. Cela s'exprime par exemple par "**`pextension`**(personne*)" qui correspond, par rapport à l'exemple de la figure Fig. 4.2, à "**`pextension`**(personne) + **`pextension`**(employe)".

IV.3.4 La notion de vue

La notion de vue détaillée dans cette partie concerne essentiellement les données. Une vue est définie ici comme une variable typée à laquelle est associée une expression du langage. La variable `SalairesMin` de la figure Fig. 4.3 est un exemple de vue définissant une expression constante. Cette expression peut néanmoins être quelconque et

utiliser par exemple d'autres variables dont le contenu est dynamique ou même des fonctions. Par exemple, nous pouvons exprimer une vue définissant la moyenne des salaires moyens des cadres de la manière suivante :

```
float MoySalaireMoyCadres as
    (SalaireMoyen (Ingenieur) +
    SalaireMoyen (Manager)) / 2;
```

La valeur d'une telle variable évoluant au cours du temps, son contenu est calculé (comme pour les vues relationnelles) lorsqu'elle est utilisée dans une autre expression en cours d'évaluation. L'expression peut être de n'importe quel type du langage et notamment des expressions ensemblistes ce qui nous ramène dans ce cas à des vues similaires aux vues relationnelles.

IV.4 Les expressions ensemblistes

Nous disposons dans PEPLOM d'un certain nombre d'opérateurs permettant d'effectuer des accès associatifs à des collections. Ces opérateurs sont très proches d'opérateurs algébriques tels que ceux définis dans [53][98][106]. Ils sont exprimés sous la forme de fonctions génériques ce qui permet de les utiliser dans la construction d'expressions ensemblistes. Nous avons choisi une approche fonctionnelle pour les accès associatifs car l'intégration au langage d'expression nous semble plus homogène même si l'expression de ces requêtes s'éloigne quelque peu d'une expression SQL. C'est un choix inverse qui est fait dans le langage O2SQL[31][32] qui privilégie une syntaxe proche de SQL mais qui n'intègre pas cet aspect comme une extension du langage d'expression (pas d'orthogonalité) comme nous l'avons fait dans PEPLOM.

IV.4.1 Les variables domaines

Nous allons voir dans les sections suivantes que la plupart des fonctions génériques permettant d'exprimer des manipulations associatives de collections (ensembles ou listes) utilisent la notion de variable domaine. Une définition de ce type de variable s'exprime de la manière suivante : "vd **in** D". Cela signifie que vd est une variable représentant un élément du domaine D qui peut être soit un ensemble, soit une liste.

Le domaine D pouvant être calculé, le langage se charge d'inférer son type. Par contre, dans le cas où le type inféré des éléments du domaine est soit n−uplet, soit union, la définition ci−dessus ne permet pas dans tous les cas la manipulation symbolique des éléments du domaine à travers la variable domaine. Il est alors possible de nommer les champs manipulés par la variable domaine en la définissant de la façon suivante : "vd(champ$_1$, ..., champ$_n$) **in** D". L'exemple suivant montre le rôle d'une telle définition :

```
struct{int c1; string(20) c2;} D1{};
struct{int a1; string(20) a2;} D2{};
...
... select (v(b1,b2) in D1+D2 ...
```

Le langage n'a pas de raison de choisir pour le noms des champs de l'élément représenté par v du domaine `D1+D2` ceux de la structure des éléments de `D1` ou ceux de la structure

des éléments de `D2`. C'est donc à l'utilisateur de définir ces noms dans la définition de la variable domaine `v` où ils sont nommés `b1` et `b2`.

IV.4.2 La fonction "select"

Le "select" est l'opérateur associatif de base du langage PEPLOM. Il permet de définir des expressions ensemblistes (retourne toujours un ensemble) très proches sémantiquement de requêtes de type "select/from/where" définies à l'aide du langage SQL ou de ses ersatz. Prenons l'exemple d'une expression qui calcule l'ensemble des personnes de la base qui ont moins de quarante ans et qui ont au moins un enfant de plus de vingt ans :

```
ref personne res{};
...
res = select (p in pextension(personne*);

                (p.Age() < 40) &&
                exist (e in p.Enfants;
                       e.Age() >= 20);

                p);
```

Nous voyons que l'opérateur "select" contient trois clauses. Il se compose de la manière suivante : **`select`**(*clause contexte*; *clause condition*; *clause résultat*). La clause contexte correspond à la clause "from" de SQL. Elle est composée d'une liste de définition de variables domaines du type "vd_1 **`in`** D_1, ..., vd_n **`in`** D_n". Elle détermine le contexte d'exécution de l'expression "select" que nous pouvons voir comme un sous−ensemble du produit cartésien des domaines exprimés dans la clause, c'est à dire pour l'exemple ci−dessus "$D_1 \times ... \times D_n$". Dans le cas où ce produit cartésien fait intervenir un domaine qui est une liste, le résultat est alors une liste où l'ordre des éléments du contexte est un ordre par défaut fixé par le langage.

La clause condition, correspondant à la clause "where" de SQL, est une expression conditionnelle dans laquelle peuvent intervenir des expressions construites à partir des variables domaines aussi bien qu'à partir de variables externes à l'expression "select". Cette expression permet d'effectuer un filtre du contexte d'évaluation défini ci−dessus par le produit cartésien des domaines.

Enfin, la dernière clause, appelée clause résultat, qui est l'équivalent de la clause "select" de SQL, permet de calculer les éléments de l'ensemble résultant de cette opération de sélection. Cette expression est évaluée pour chaque n−uplet du contexte filtré défini par les clause "from" et "where" précédentes. La cardinalité de cet ensemble résultat est donc au maximum celle du contexte filtré.

Dans l'exemple précédent, le contexte non filtré est donc "**`pextension`**(personne*)" qui correspond à l'ensemble de toutes les personnes de la base. Il est ensuite filtré par l'expression booléenne "(p.Age() < 40) && **`exist`** (e **`in`** p.Enfants; e.Age() >= 20)". La fonction générique **`exist`** qui est utilisée ici est décrite dans la section IV.4.4 qui suit. Cette expression vérifie si la personne désignée par `p` a moins de quarante ans et regarde s'il existe un enfant de plus de 20 ans dans l'ensemble `p.Enfants` de cette personne `p`.

L'expression résultat est enfin l'objet désigné par la variable `p` qui rempli la condition précédente. Il doit y avoir conformité de types entre le type de l'expression résultat et le type des éléments de l'ensemble auquel nous affectons le résultat du "select".

De plus, PEPLOM offre un opérateur similaire au "select" qui permet de récupérer un seul élément. Son expression est la suivante : **`get`**(*clause contexte*; *clause condition*; *clause résultat*). Le résultat retourné par la fonction **`get`** est une valeur structurée du type "**`struct`**{**`bool`** `trouve; T`$_{res}$ `elem`}". Si la fonction a trouvé au moins un résultat (quelconque parmi les résultats possibles) alors `trouve` est vrai et `elem` contient la valeur calculée par l'expression de la clause résultat.

IV.4.3 Les fonctions de restructration

Outre les constructeurs de littéraux qui permettent de structurer une expression, PEPLOM définit des opérateurs qui permettent de restructurer les éléments d'un ensemble. Nous en distinguons deux catégories. Les premiers sont issus des modèles NF2 ou de la manipulation de valeurs complexes[1][91][97]. Il s'agit des opérateurs **`nest`** et **`unnest`** qui s'appliquent à des ensembles de n−uplets. Ils ont tous les deux le même format d'utilisation à savoir : **`nest`**(*clause domaine*; *clause attribut*) et **`unnest`**(*clause domaine*; *clause attribut*). La clause domaine correspond à une définition de variable domaine du type "vd **`in`** D" où le domaine D est un ensemble de n−uplets. La clause attribut spécifie sur quel attribut des n−uplets s'effectue le regroupement ou l'éclatement des valeurs de cet attribut. L'exemple suivant nous montre comment nous pouvons recréer une relation plate définissant l'association parent/enfant (variable `ParentDe`) :

```
struct {
    ref personne parent;
    ref personne enfant;
    } ParentDe{};
...
ParentDe = unnest (t(parent,enfants) in select (
                          p in pextension(personne*);
                          /* condition vide */;
                          struct{p, p.Enfants});
                   t.enfants);
```

Cette requête pourrait être exprimée d'une manière plus simple mais le but est de montrer un exemple d'utilisation de l'opérateur **`unnest`**.

La deuxième catégorie d'opérateurs correspondent aux opérateurs de partitionnement (**`group`**) et d'applatissage (**`flatten`**). L'opérateur de regroupement existe d'ailleurs dans SQL où il est dissimulé derrière la clause "group by" utilisable dans les requêtes "select/from/where". L'opérateur **`flatten`** s'applique sur une structure d'ensembles ou de listes imbriquées. Il supprime ce niveau d'imbrication. C'est donc une fonction générique qui possède un seul paramètre dont le type est T{}{}, T{}<>, T<>{} ou T<><>. La structure de son résultat est la même que la structure la moins imbriquée de son expression d'entrée. Par exemple, **`flatten`**(T{}<>) rend un T{} et **`flatten`**(T<>{}) un T<>.

L'opérateur group s'applique sur un ensemble et partitionne cet ensemble suivant une caractéristique donnée. Son format est **`group`**(*clause domaine*; *clause caractéristique*).

La clause domaine correspond à la définition d'une variable domaine et la clause caractéristique est une expression dépendant de la variable domaine (sinon l'opération n'a aucun effet). Le résultat de cette fonction générique est un n−uplet dont le premier champ est la valeur de l'expression définie par la clause caractéristique et le second, l'ensemble des éléments du domaine qui engendre cette valeur. L'exemple qui suit regroupe les personnes par tranches d'age de 10 années :

```
struct {
    uchar Decenie;
    ref personne Tranche{};
    } Tranches{};
...
Tranches = group (p in pextension(personne*);
                  (p.Age() / 10) * 10);
```

IV.4.4 Les fonctions prédicatives

Deux fonctions prédicatives génériques sont offertes par le langage PEPLOM. Elles correspondent aux quantificateurs universel et existentiel. Elles ont toutes deux le même format d'utilisation, à savoir : **all**(*clause domaine*; *clause condition*) et **exist**(*clause domaine*; *clause condition*). La clause domaine est la même que précédemment et la clause condition est une expression conditionnelle dépendant de la variable domaine définie dans la première clause. Nous pouvons observer un exemple d'utilisation de la fonction **exist** dans une section précédente (cf. IV.4.2). Si le domaine utilisé dans la clause domaine est vide, les deux fonctions rendent comme résultat **false**.

IV.4.5 Les fonctions d'agrégat

Les dernières fonctions génériques fournies par le langage et travaillant sur les collections correspondent aux opérateurs d'agrégat. Ceux−ci sont très utiles de le contexte des bases de données notamment dans les applications de gestion (MIS). Nous n'offront que les opérateurs qu'on trouve dans les systèmes relationnels c'est à dire **sum**, **avg**, **min** et **max** qui calculent respectivement la somme, la moyenne, le minimum ou le maximum d'une expression dépendant des éléments d'une collection. Leur format d'utilisation est le même dans les quatre cas : par exemple **sum**(*clause domaine*; *clause expression*). Prenons par exemple une expression qui calcule la moyenne d'age des personnes de la base :

```
uchar AgeMoyen;
...
AgeMoyen = avg(p in pextension(personne*);  p.Age());
```

IV.5 Conclusion

Le langage PEPLOM, dont nous avons décrit les principaux concepts dans ce chapitre, met essentiellement l'accent sur l'intégration entre les aspects langage de programmation à objets et les fonctionnalités base de données et la sureté du langage. Nous avons notamment défini la persistance et les accès associatifs ensemblistes comme des concepts orthogonaux

aux autres concepts du langage. Cette approche garantit une définition claire du langage pour l'utilisateur.

Le système de types, défini à partir de celui de C++ (avec lequel il n'est pas compatible), se caractérise par un modèle structurel du type objet complexe. Par ailleurs, les types sont exclusivement des descriptions intentionnelles et ne sont liés à aucun concept d'instanciation (utilisation de variables globales, production de la persistance). La réutilisation en est ainsi grandement facilitée.

La notion de module introduite permet une introduction purement langage d'aspects base de données. Ils permettent en effet de considérer la base comme une extension du contexte global (variables globales) de l'application avec un contexte persistant caractérisé par des variables dites persistantes (points d'entrée dans la base).

Cette séparation entre les aspects intentionnels et extensionnels a un rôle méthodologique intéressant. Elle se caractérise aussi par la séparation entre les types abstraits et leurs extensions pour lesquelles la relation d'héritage entre type n'implique pas une inclusion des extensions ce qui offre une plus grande souplesse.

Un des aspects, qui est prépondérant dans les systèmes déclaratifs tels que les SGBD relationnels, est la séparation entre la description conceptuelle des données et leur description physique (format de stockage). Cela concerne principalement les collections sur lesquelles nous voulons pouvoir définir différentes méthodes d'accès (index haché, arbre B, etc...). Le problème est d'autant plus important que les collections interviennent non seulement au niveau des extensions de types mais aussi dans les types eux−mêmes.

Il reste que l'intérêt du langage ne sera réellement mis à jour que lorsque des applications auront été développées à l'aide de celui−ci. Ceci nous amène à préciser que le langage PEPLOM est ouvert au monde C/C++ : il est possible d'utiliser du code C ou C++ dans du code PEPLOM. La coopération dans l'autre sens n'est pas envisagée pour le moment.

Chapitre V

Gestion des définitions PEPLOM

> *C'est ce genre d'idée qui pourrait facilement déconditionner les esprits les moins solidement arrêtés parmi les castes supérieures, qui pourrait leur faire perdre la foi dans le bonheur comme Souverain Bien, et leur faire croire, à la place, que le but est quelque part au−delà, quelque part au−dehors de la sphère humaine présente ; que le but de la vie n'est pas le maintien du bien−être, mais quelque renforcement, quelque raffinement de la conscience, quelque accroissement de savoir...*
>
> *Aldous Huxley − "Le meilleur des mondes"*

Dans ce chapitre, nous décrivons les différents éléments qui composent l'environnement de développement offert pour le langage PEPLOM. L'approche suivie par PEPLOM est d'avoir un environnement totalement intégré, construit autour des différentes définitions du langage. Nous considérons notamment que le fait d'avoir un langage persistant n'est pas neutre vis à vis de la gestion des définitions (types, modules, méthodes, etc...). En effet, si nous prenons l'exemple d'un type abstrait, il définit une structure de stockage pour des instances persistantes. Il est donc important de contrôler cette définition pour conserver la cohérence entre la structure de telles instances et sa définition. L'exemple des SGBD relationnels est révélateur de ce point de vue puisque les définitions de relations sont stockées dans un catalogue persistant. Ce catalogue est la plupart du temps géré par le SGBD sous forme d'une méta−base [39][40] de manière à garantir le même niveau de sécurité que pour les données de la base.

Un LPBD doit suivre les mêmes règles de sécurité pour la cohérence entre les données et les définitions qui les décrivent. Le choix d'implantation sous la forme d'une méta−base pose deux problèmes. Premièrement, cette méthode engendre un problème s'assimilant à celui de savoir qui de la poule ou de l'œuf était le premier. Nous avons en effet besoin du LPBD pour implanter la méta−base et le LPBD ne peut fonctionner sans le catalogue des définitions. Ce problème est généralement résolu soit par une procédure manuelle d'initialisation, soit par un catalogue *ad hoc* permettant cette initialisation.

Deuxièmement, il n'est pas évident que le LPBD se prête bien à l'implantation du catalogue. Du point de vue des structures de données mises en œuvre par PEPLOM, cela ne

parait pas poser de problème. Par contre, si nous nous plaçons dans le contexte d'un environnement intégré utilisé simultanément par plusieurs programmeurs et que nous voulons offrir des moyens de faire du développement coopératif, cela nécessite des mécanismes de synchronisation appropriés différents de la synchronisation habituellement utilisée dans les SGBD[16].

Des travaux importants ont été faits pour définir des plateformes pour supporter ce type d'environnements intégrés qui ont notamment abouti à des systèmes à la norme PCTE[84]. Le but ici n'est pas d'étudier le support de l'environnement mais de définir les différents agents et entités y intervenant ainsi que les interactions qui les lient.

Pour bien comprendre l'organisation de l'environnement de développement de PEPLOM, il faut savoir que le choix qui a été fait d'une approche compilée a conduit à dissocier la phase de compilation en deux parties. La première partie correspond à une phase d'édition et de validation de définitions PEPLOM. Elle produit du code intermédiaire (code pivot) pour chaque définition. La deuxième partie permet, à partir des définitions, de générer des entités utilisables à l'exécution (telles que des applications exécutables).

Toutes les définitions liées au langage ainsi qu'à son environnement de programmation et d'utilisation sont conservées dans le catalogue. Elles sont décrites dans la section V.1. La section V.2 présente des mécanismes de vues et de protection sur les définitions définies précédemment. Le catalogue, qui contient les définitons, est l'élément central dans l'architecture de l'environnement de développement de PEPLOM. Son organisation ainsi que son utilisation par d'autres éléments de l'environnement sont décrites dans le section V.3.

V.1 Différentes entités d'un schéma PEPLOM

L'un des points importants du langage PEPLOM est qu'il définit des unités de structuration de l'environnement de programmation. Si nous prenons un langage comme C++, l'unité de structuration de l'environnement est le fichier. C'est alors le programmeur qui gère la cohérence entre les fichiers sources, objets et binaires exécutables (utilisation de fichiers "makefile"). Nous distinguons dans l'environnement de PEPLOM deux catégories d'unités. Les unités qui correspondent à des définitions et d'autres à des instances générées à partir de ces définitions. L'environnement gère la cohérence de ces unités où la notion de fichier source est remplacée par celle d'unité de définition. Il gère notamment toutes les dépendances entre ces unités et définit un état global du schéma de définitions à tout instant.

V.1.1 Les définitions PEPLOM

L'un des intérêts de l'approche LPBD est de fournir un langage permettant de manipuler des données persistantes de manière transparente (comme n'importe quelle donnée du langage) et aussi de les accéder de façon associative. C'est le cas dans le langage PEPLOM

où nous introduisons la persistance comme une notion orthogonale à celle de type et de variable.

Par contre, le fait que la persistance soit transparente pour le programmeur ne signifie pas pour autant que la notion de base de données devient obsolète. En effet, les accès associatifs impliquent que le plan d'accès aux données soit calculé par le compilateur comme dans le cas d'un SGBD relationnel. Pour améliorer l'optimisation de ces accès, il est alors intéressant de pouvoir définir des structures d'accès telles que des index. Le choix dans PEPLOM est de différencier les rôles le mieux possible et de dire ainsi que cette tâche d'administration de la base est présente dans l'environnement mais qu'elle doit être isolée de la phase de programmation.

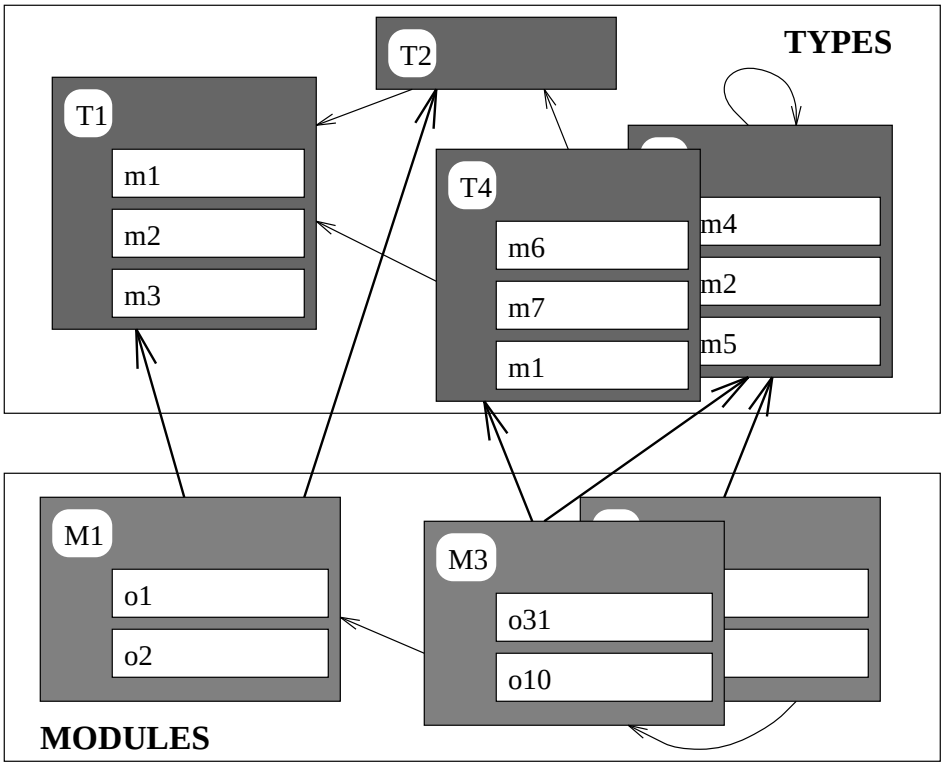


Fig. 5.1 : Organisation des définitions PEPLOM

V.1.1.1 Les définitions du langage

Le langage PEPLOM permet de définir des entités de base qui sont des unités de structuration pour l'environnement. A ce niveau, nous avons essentiellement deux unités primaires qui sont les types (types concrets et types abstraits) et les modules.

Les types abstraits et les modules peuvent définir des fonctions et des procédures. Ce sont donc deux autres unités secondaires de l'environnement qui dépendent toujours d'une unité primaire. Les deux unités primaires définissent deux espaces de structuration distincts qui ne coopèrent que dans un seul sens : les définitions de modules peuvent utiliser les définitions de types (flêches épaisses dans Fig. 5.1).

A l'intérieur de ces espaces, de nombreuses interdépendances existent entre les unités primaires. Toutes ces dépendances doivent être gérées par l'environnement de manière à pouvoir définir les effets d'une modification d'une unité de structuration sur les autres définitions PEPLOM. Les différentes relations pouvant exister entre les différentes unités sont données dans la figure Fig. 5.2. Les flêches d'épaisseur double montrent les associations entre les unités primaires et les unités secondaires. Nous observons qu'à la définition du code d'une fonction ou d'une procédure est associée une unique unité primaire qui est soit un type abstrait, soit un module. Cette association est totale car il ne peut pas y avoir de définitions d'opérations en dehors des types ou des modules. La relation inverse est multivaluée et partielle.

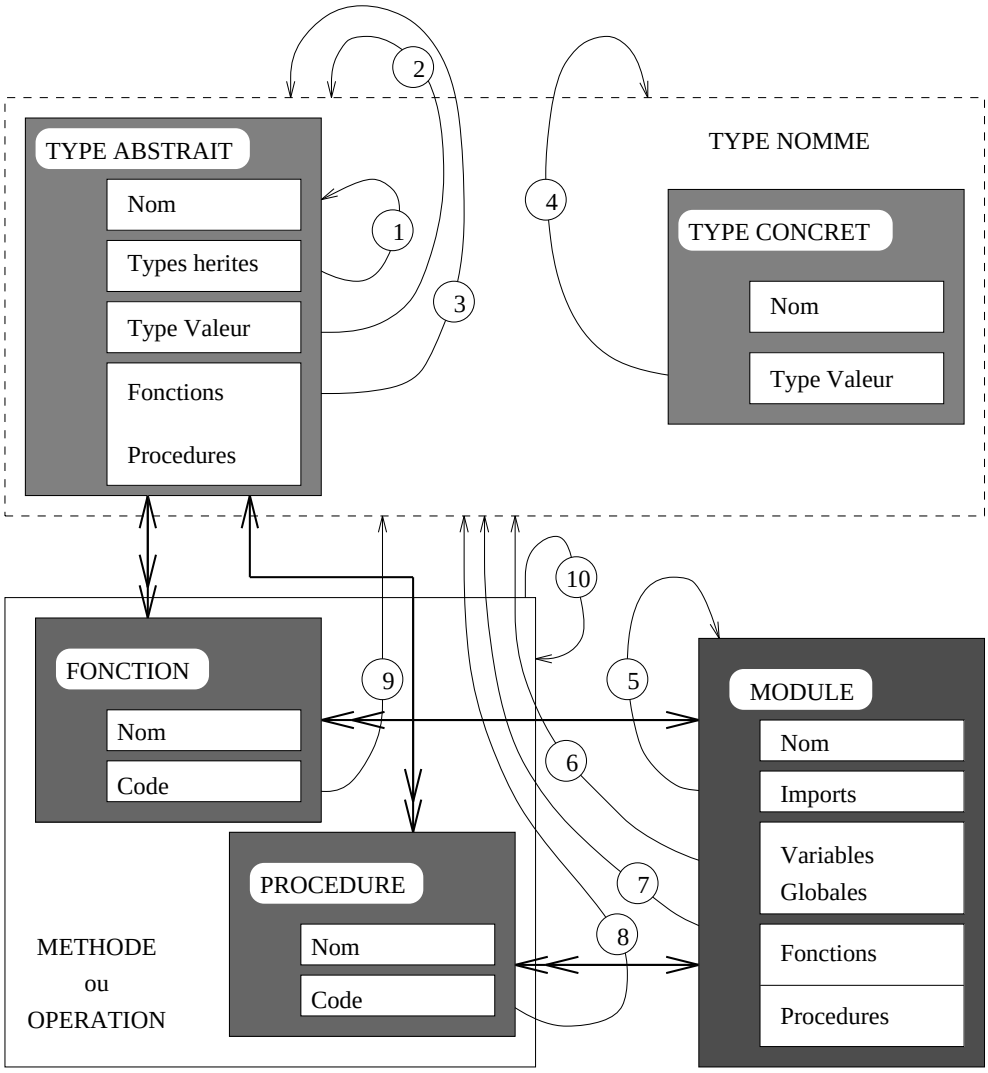


Fig. 5.2 : Associations et dépendances entre unités de structuration

Les flêches d'épaisseur simple indiquent les différentes dépendances qui peuvent exister entre les unités de structuration. Nous regroupons sous le terme de "type nommé" les unités de structuration qui sont des types. Elles sont divisées en deux catégories à savoir les types abstraits (toujours nommés) et les types concrets. Le cadre en pointillé permet donc de désigner soit un type abstrait, soit un type concret nommé. Nous dénombrons neuf dépendances possibles entre les unités de structuration. La signification de chacune de ces dépendances est la suivante :

(1) correspond aux dépendances d'un type abstrait vis à vis des autres types abstraits dont il hérite directement.

(2) détermine l'ensemble des types (abstraits ou concrets) qui sont utilisés dans la définition du type de la valeur des objets qui seront générés à partir de ce type abstrait.

(3) définit l'ensemble des types dont dépend un type abstrait qui utilise ceux−ci pour définir les types des paramètres de ses méthodes (fonctions ou procédures).

(4) est équivalent à (2) et contient les dépendances par rapport aux types utilisés dans la définition de ce type concret nommé.

(5) correspond aux dépendances inter−modules définies par les liens d'import/export.

(6) contient les dépendances des modules par rapport aux types utilisés dans les définitions de leurs variables globales.

(7) est équivalent à (3) et définit l'ensemble des types utilisés dans les définitions des paramètres des fonctions et procédures définies dans un module.

(8) et (9) correspondent aux dépendances des procédures ou des fonctions vis à vis des types utilisés pour les définitions de variables locales définies dans le code de celles−ci.

(10) définit les dépendances des procédures ou fonctions par rapport aux procédures ou fonctions utilisées dans leurs codes.

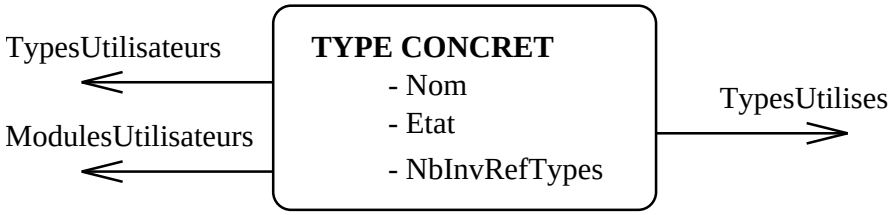
Toutes ces dépendances représentent le réseau d'interconnexions reliant les différentes unités de structuration gérées par l'environnement. Un des buts de l'environnement est de pouvoir dire à tout instant si une unité est correcte et, si elle ne l'est pas, quelles sont les incohérences qui posent problème. C'est pour cela que le catalogue conserve tous les liens de dépendance définis ci−dessus de même que leurs inverses. Ainsi, à chaque fois que le catalogue est modifié, une procédure permettant de calculer le nouvel état de chaque unité stockée est exécutée. Elle se sert alors des différents liens pour déterminer quelles sont les unités concernées par la modification.

Cet algorithme, exécuté par le gestionnaire du catalogue, est relativement coûteux. Il représente néanmoins la base de la technique utilisée pour faire de la compilation incrémentale. Il peut aussi permettre, dans le cadre de développements coopératifs, de rendre le gestionnaire de catalogue actif ; il serait possible, par exemple, de signaler au programmeur

responsable du développement d'un type, les conséquences sur ce dernier de modifications intervenues par ailleurs.

Avant de décrire cet algorithme, il faut d'abord définir les différents liens gérés par le catalogue entre les différentes unités. Nous représentons chaque unité comme une boîte englobant un certain nombre de caractéristiques. Les liens vers d'autres boîtes sont représentés par des flêches partant de cette boîte où les flêches qui sont du côté gauche représentent des liens du type "est utilisé par" et celles vers la droite des liens du type "utilise". Tous ces liens externes sont multivalués (ensemble de références) ; les liens monovalués sont représentés par des caractéristiques internes. Tous les liens définis à droite correspondent à des dépendances ou à des associations définies dans la figure Fig. 5.2. Ceux définis à gauche servent essentiellement à la propagation des conséquences d'une mise à jour d'une unité du schéma. Les boîtes représentant chaque catégorie d'unité sont définies comme suit :

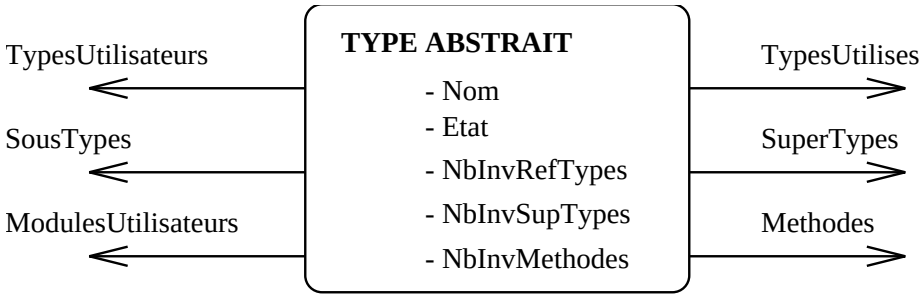
1. pour les types concrets nommés, nous définissons trois attributs qui sont le nom du type (Nom), le nombre de types utilisés dans sa définition qui sont invalides (NbInvRefTypes) et l'état du type lui−même (Etat). Cet état peut être soit syntaxiquement incorrect (SyntInc), soit sémantiquement incorrect (SemInc), soit valide (Valide).



   Un type sera sémantiquement incorrect s'il utilise par exemple un type non valide dans sa définition.

   Nous avons un seul lien de dépendance (dépendance (4)) matérialisé par l'ensemble des types utilisés dans la définition (TypesUtilises). De plus, le catalogue gère les références vers les types et les modules qui utilisent ce type concret dans leurs définitions (TypesUtilisateurs et ModulesUtilisateurs).

2. pour les types abstraits, nous définissons six attributs. Les trois premiers sont les mêmes que ceux définis pour les types concrets.

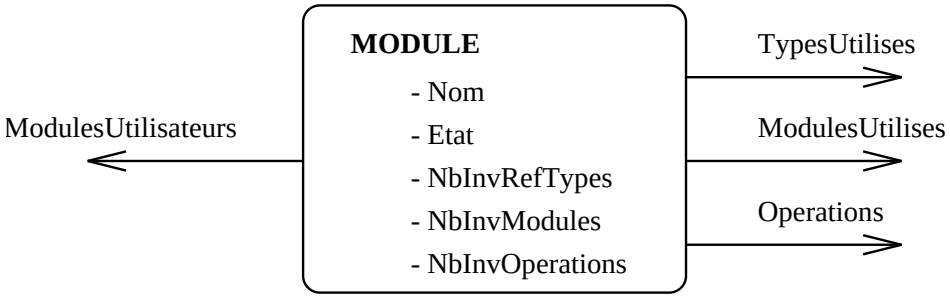


   Nous définissons en plus le nombre de supertypes associés au type abstrait qui sont non valides (NbInvSupTypes), le nombre de liens inverses définis dans la structure

du type qui sont incohérents (NbInvInverses) et enfin le nombre de méthodes associées au type qui sont non valides (NbInvMethodes). La sémantique de l'état est la même que pour les types concrets. Nous pouvons remarquer que l'état du type sera invalide dès que l'un des compteurs d'invalidité aura une valeur supérieure à 0.

Le premier ensemble de références (SuperTypes) correspondant à la dépendance (1), définit les liens d'héritage entre sous−types et super−types. Les dépendances (2) et (3) sont matérialisées par le même ensemble de référence qui est "TypesUtilises". "Methodes" correspond à l'association entre le type abstrait et les définitions de ses méthodes (liens unité primaire / unité secondaire). Nous maintenons les références vers les unités utilisant le type dans leur définition dans trois cas. Le premier cas concerne tous les types utilisant ce type abstrait dans leur définition (TypesUtilisateurs), le deuxième tous ses sous−types (SousTypes) et le troisième tous les modules qui l'utilisent dans leur définition (ModulesUtilisateurs).

3. pour les modules, nous définissons cinq attributs. Toujours les trois mêmes attributs pour le nom, l'état du module et les types utilisés dans sa définition.



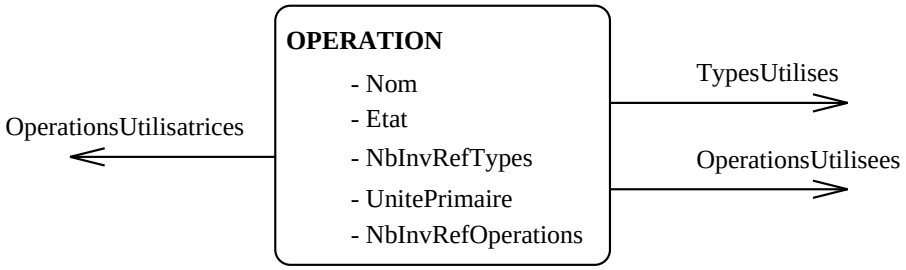
Les attributs Nom, Etat et NbInvRefTypes ont la même signification que pour les types. Le compteur "NbInvModules" indique le nombre de modules utilisés par ce module (modules importés) qui sont invalides. Enfin, comme pour les méthodes attachées aux types abstraits, un compteur (NbInvOperations) signale le nombre d'opérations du module dont le code n'est pas valide.

Concernant les liens vis à vis des autres unités, nous matérialisons la dépendance (dépendances (6) et (7)) par rapport aux types utilisés dans la définition du module ainsi que celle se rapportant aux modules importés (dépendance (5)). Les opérations associées au module sont référencées à travers l'ensemble "Operations". L'unique lien inverse maintenu est celui concernant les modules qui importent celui−ci (ModulesUtilisateurs).

4. pour les méthodes associées aux types abstraits ou les opérations associées aux modules, nous définissons une seule unité (unité secondaire) qui est l'OPERATION.

Nous définissons cinq attributs pour cette unité dont les trois premiers (Nom, Etat et NbInvRefTypes) sont similaires à ceux définies précédemment dans les unités primaires. L'attribut "UnitePrimaire" est la référence vers l'unité à laquelle

l'opération est associée. Le dernier attribut est un compteur (NbInvOperations) qui donne le nombre d'opérations utilisées dans le code de cette opération qui sont non valides.



Les liens de dépendance sont au nombre de deux : l'un (dépendances (8) et (9)) correspond aux types utilisés dans le code de l'opération (TypesUtilises) et l'autre (dépendance (10)) correspond aux références vers les opérations utilisées dans ce même code. Enfin, le lien inverse de ce dernier lien défini les opérations qui utilisent dans leur code celle qui est définie ici.

Tous les liens de dépendance (liens à droite dans les schémas) définis ci−dessus sont représentés sous la forme d'ensembles de couples (nu, ru) où **nu** est le nom de l'unité de définition référencée et **ru** la référence vers cette unité. Les liens "inverses" (définis à gauche dans les schémas) sont représentés par un ensemble de références vers les unités utilisant l'unité concernée. La référence **ru** peut être `nil` par exemple dans le cas où un type utilise dans sa définition un autre type de nom **nu** qui n'a jamais été défini. Pour exprimer les différents algorithmes de mise à jour des définitions, nous définissons le catalogue contenant les unités de définitions comme le quadruplet (TC, TA, MO, OP) où :

- TC est l'ensemble d'unités primaires correspondant aux définitions de types concrets nommés,
- TA est l'ensemble d'unités primaires correspondant aux définitions de types abstraits,
- MO est l'ensemble d'unités primaires correspondant aux définitions de modules et
- OP est l'ensemble des unités secondaires correspondant aux définitions de méthodes ou d'opérations.

Pour l'accès aux attributs d'une unité de définition aussi bien qu'à ses liens vers les autres unités, nous utilisons la fonction "." d'accès à un champ. Prenons l'exemple d'un type $tc \in TC$, "tc.Nom" est le nom du type et "tc.TypesUtilises" est l'ensemble des références vers les types utilisés dans la définition.

L'approche de compilation incrémentale utilisée pour PEPLOM implique que toutes les définitions sont stockées dans le catalogue. Dans la première phase d'édition et validation des définitions, nous considèrons que l'ensemble des définitions créées et modifiées simultanément par plusieurs programmeurs peuvent être dans un état incohérent à certains instants au cours du développement. Le but que nous nous fixons est d'autoriser ces incohérences tout en ayant la possibilité de renseigner le programmeur sur celles−ci à tout

moment. Or chaque modification d'une définition peut avoir des conséquences sur la cohérence de nombreuses autres définitions. C'est pour cela que le catalogue gère aussi tous les liens entre les définitions présentés ci−dessus. Nous allons maintenant voir quels sont les problèmes que pose la gestion de cet état global du catalogue de définitions.

Pour situer les problèmes, nous commençons par donner un exemple de définitions PEPLOM à gérer :

```
typedef  abstract personne
{...
};
typedef abstract proprietaire : personne
{struct {
    struct {
        voiture  Voiture;
        carte_grise  CarteGrise;
        }InfosVoiture;
    } value;
};
typedef  abstract voiture : vehicule
{struct {
    marque  Marque;
    proprietaire  Proprietaire;
    } value;
};
typedef  enum {Austin, Citroen, Peugeot, ...} marque;
```

Il y a deux incohérences dans cet ensemble de définitions à savoir qu'il manque les définitions de deux types abstraits : carte_grise et vehicule. La représentation de trois des définitions dans le catalogue serait donc les trois unités marque, propriétaire et voiture définies comme suit (l'emploi du nom d'une unité comme une valeur de champ équivaut à exprimer une référence vers celle−ci) :

- marque (TYPE CONCRET)
  - Nom : "marque"
  - Etat : Valide
  - NbInvRefTypes : 0
  - TypesUtilises : {}
  - TypesUtilisateurs : {voiture}
  - ModulesUtilisateurs : {}
- proprietaire (TYPE ABSTRAIT)
  - Nom : "proprietaire"
  - Etat : SemInc
  - NbInvRefTypes : 2
  - TypesUtilises : {("voiture", voiture), ("carte_grise", **nil**)}
  - NbInvSupTypes : 0
  - SuperTypes : {("personne", personne)}

  - NbInvMethodes : 0
  - Methodes : {}
  - TypesUtilisateurs : {voiture}
  - SousTypes : {}
  - ModulesUtilisateurs : {}
- voiture (TYPE ABSTRAIT)
  - Nom : "voiture"
  - Etat : SemInc
  - NbInvRefTypes : 1
  - TypesUtilises : {("proprietaire", proprietaire)}
  - NbInvSupTypes : 1
  - SuperTypes : {("vehicule", **`nil`**)}
  - NbInvMethodes : 0
  - Methodes : {}
  - TypesUtilisateurs : {proprietaire}
  - SousTypes : {}
  - ModulesUtilisateurs : {}

Nous nous intéressons dans cette section à la première phase de la compilation que nous avons appelé édition/validation. Le problème que nous nous fixons ici est de modifier le catalogue pour prendre en compte la nouvelle définition d'unité (qui existe peut−être déjà dans le catalogue). Nous récupérons cette définition à sa sortie de l'analyseur syntaxique et sémantique qui rend l'état de la définition (Nom, Etat, NbInvRefTypes, etc ...) par rapport aux informations concernant les autres définitions utilisées (fournies à l'analyseur sémantique par le gestionnaire du catalogue). Il faut donc déterminer si la nouvelle définition influence l'état des autres définitions répertoriées dans le catalogue. Si c'est le cas, il faut alors propager les conséquences de cette modification à toutes les définitions concernées.

Supposons par exemple que soit ajoutée la définition d'un nouveau type abstrait de nom "vehicule" et qu'elle soit valide. Le type voiture est alors concerné par cet ajout et deux attributs qui y sont attachés doivent être modifiés : NbInvSupTypes doit prendre la valeur 0 et SuperTypes la valeur {("vehicule", vehicule)}. La propagation des conséquences parait assez facile puisqu'en fait, il suffit de signaler à toutes les définitions qui utilisent la définition modifiée, ajoutée ou détruite de valider ou invalider leur référence à cette dernière. L'opération est répétée récursivement pour les unités qui changent d'état.

Cette solution ne fonctionne en fait que partiellement car elle ne traite pas le cas des cycles dans les définitions de types abstraits ou de modules. En effet, supposons que, dans l'exemple, nous définissions en plus le type abstrait de nom "carte_grise". Nous allons alors modifier le type personne en modifiant NbInvRefTypes qui va passer à 1 et TypesUtilises dont le couple ("carte_grise", **`nil`**) devient ("carte_grise", carte_grise). Nous n'avons plus à

priori d'incohérence dans le schéma mais proprietaire et voiture restent néanmoins incohérents.

La difficulté vient du fait que certains liens peuvent introduire des circuits dans le graphe d'interconnexions. Seul deux liens sont dans ce cas et ce sont TypesUtilises dans TYPE ABSTRAIT (et TYPE CONCRET) et OperationsUtilisees dans OPERATION. Nous nous limitons au cas des types pour illustrer le problème et sa solution.

Prenons l'exemple suivant qui n'a aucune incohérence :

```
typedef TA1 TC1;
typedef  abstract TA1
{struct {
    TA2 a11;
    TA3 a12;
    } value;
};
typedef  abstract TA2
{struct {
    TC1 a21;
    };
};
typedef  abstract TA3
{struct {
    TA1 a13;
    } value;
};
```

Quelque soit l'ordre dans lequel sont insérées ces définitions dans le catalogue, les circuits impliquent qu'un algorithme de propagation simple conduit à définir un état incohérent du schéma global. En effet, lors de l'analyse de la nouvelle définition, le gestionnaire de catalogue signale que les autres définitions qu'elle utilise sont invalides ; le gestionnaire ne peut pas savoir qu'elles seraient correctes si la nouvelle définition était prise en compte.

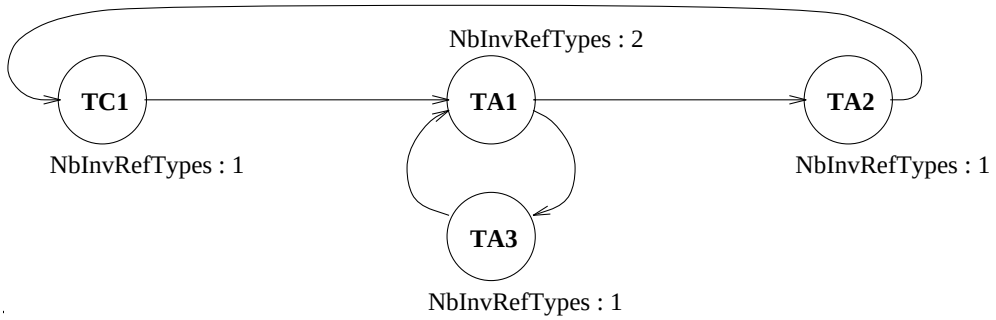


Fig. 5.3 : Illustration de circuits dans les définitions de types

Il faut donc une fois que les mises à jour possibles ont été faites, repérer les circuits pour savoir si d'autres mises à jour peuvent être faites. La précondition nécessaire pour qu'il soit intéressant de repérer les circuits pour des mises à jour approfondies est que tous les liens qui

n'ont pas le potentiel d'introduire des circuits soit correct car cette fonctionnalité n'est intéressante que dans le cas où nous cherchons à savoir si une définition non valide est en fait valide. Plus précisément, cela signifie que tous les compteurs (NbInv...) d'une définition de type en dehors du compteur NbInvRefTypes (concret ou abstrait) doivent être à la valeur 0. Cette précondition remplie, nous pouvons alors calculer les circuits et un type peut devenir valide si le nombre de circuit le traversant est égale à la valeur de NbInvRefTypes comme le montre la figure Fig. 5.3.

L'algorithme de calcul des circuits est ici relativement compliqué puisque si nous faisons du marquage, dans le cas où un chemin se divise en plusieurs autres, il faut créer autant de marques M_i que de chemins possibles et propager ces marques dans tous les nœuds déjà marqués comme le montre la figure Fig. 5.4. Outre la complexité de l'algorithme en $O(n^2)$ (où n est le nombre de nœuds), il faut des structures supplémentaires pour la représentation des circuits (une liste de couleurs par nœud). Ensuite, il faut parcourir les noeuds de chaque ensemble connexe de circuits pour vérifier s'il peuvent tous passer dans l'état valide et si c'est le cas, tous les valider.

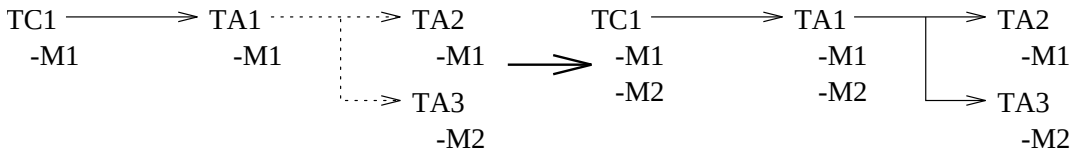


Fig. 5.4 : Calcul de circuits et propagation des nouvelles couleurs

Nous avons donc retenu une solution différente qui consiste, lorsqu'un type vérifie la précondition définie ci−dessus, à signaler aux types qui l'utilisent qu'il est en fait valide, tout en lui conservant son état d'invalidité dans le catalogue. Dans le cas d'un circuit, la validation revient donc au type duquel l'algorithme a démaré si tous les autres types du circuit ont été validés. Par contre, si à la fin de cet essai de validation, le type n'est pas dans l'état valide, il faut de nouveau l'invalider dans tous les types qui l'utilisent. Cette solution est intéressante puisque dans le cas favorable (passage à l'état valide) nous avons réduit la complexité à O(n) en gardant la même complexité $O(n^2)$ pour le cas défavorable.

La technique de compilation incrémentale utilisée est donc assez coûteuse lorsque nous faisons des mises à jour du catalogue des définitions. Par contre, l'intérêt d'avoir un état global et local des définitions à tout instant est grand. Il permet de déterminer instantanément si le code d'une application exécutable peut être généré. Cela ouvre aussi la porte à l'implantation d'outils interactifs d'édition des définitions capables de signaler les incohérences d'une définition et d'aider ainsi le programmeur à les corriger.

V.1.1.2 Les définitions de bases de données

Nous voulons qu'un programmeur utilisant PEPLOM ait le moins possible de nouveaux concepts, par rapport à un langage impératif tel que C++, dus à la dimension base de données du langage. La description physique des données est une fonctionnalité nécessaire dans les

SGBD. Dans les SGBD réseau et hiérarchique, la description logique et physique des données étaient très liées. C'est un apport important des SGBD relationnels d'avoir isolé les deux niveaux et nous voulons donc conserver cet atout dans PEPLOM et d'une manière générale dans les LPBD.

Dans un SGBD relationnel, la description physique est essentiellement dirigée par la notion d'index. En effet, la structure physique de stockage d'une relation va dépendre de son index primaire (si un tel index a été défini). La description concerne seulement la relation et pas le n−uplet. La raison est que le n−uplet est une structure statique alors que la relation ou plus généralement l'ensemble est une structure dynamique (taille variable). Les descriptions physiques dans un SGBD servent ainsi principalement à l'optimisation du stockage (espace occupé) et de l'accès à ces structures dynamiques.

Ces descriptions physiques sont d'autant plus importantes dans un LPBD supportant un modèle d'objets complexes que nous retrouvons des structures dynamiques (ensembles et listes dans PEPLOM) aussi bien pour les extensions de types abstraits (équivalent des relations) que dans la structure des objets eux−mêmes.

Pour PEPLOM, les descriptions ou contraintes physiques sur les données de la base sont définies dans l'unité de définition appelée "base de données". C'est une unité de structuration au même titre qu'un type ou qu'un module. Les descriptions physiques d'ensembles ou de listes qui y sont faites concernent la structure décrite par les types, la structure des valeurs des variables persistantes des modules et bien sûr les extension des types abstraits. Nous pouvons définir un exemple de base de données PEPLOM se rapportant aux types et aux modules que nous avons défini jusqu'ici :

```
dbdef BaseEmployes : PG8K
{
typeform {
    personne.value.Prenoms as internal stdlist(3,2);
    };
modform {
    GestionDuPersonnel.SalairesMin  as stdset;
    };
extform {
    personne as
        primary btree on value.Nom,
        hash(120)
            on value.Adresse.Ville.Departement
            with fhach_dep;
    };
};
```

Dans cet exemple, nous définissons le format d'une base composée de pages de 8 Ko (**PG8K**). La structure de la liste des prénoms d'une personne (clause **typeform**) est une liste standard, stockée sous la forme d'une séquence (idem pour les ensembles), pour laquelle nous allouons de la place pour au moins trois éléments à sa création. Si nous débordons ce nombre d'éléments, le système va allouer des extensions qui sont chainées à la structure de base et qui contiennent deux éléments. Il est défini par ailleurs que cette liste est stockée à

l'intérieur de son objet englobant et non sous la forme d'un objet indépendant représentant l'autre alternative de stockage d'une liste standard (idem pour les ensembles).

Dans la partie **`modform`**, nous définissons le format d'implantation d'une constante ensemble qui est au format standard.

Dans la partie **`extform`**, nous définissons deux index sur l'extension du type abstrait `personne`. Le premier est un index primaire − c'est à dire plaçant − qui est un B−arbre défini sur le champ `Nom` de la valeur des objets de ce type. Le deuxième définit un hachage, sur 120 entrées, des personnes sur leur département de résidence. La fonction de hachage associée est définie comme une fonction PEPLOM associée à la base (sa signature peut être inférée) :

```
ulong BaseEmployes::fhach_dep (Tdep departement)
{
...
}
```

Nous associons à l'unité de définition de base de données, comme pour les autres unités de l'environnement de développement, une boîte qui la représente et qui définit ses liens avec les autres unités.

Une définition de base de données, telle que proposée ici, définit un format de base de données. En effet, elle décrit le comportement et la structure au niveau physique d'une certaine partie des données décrites par le schéma de définitions PEPLOM. Cette définition peut alors servir pour plusieurs bases matérialisées. Le lien BasesRéellesUtilisatrices référence donc toutes les bases "réelles" (bases matérialisées) instanciées à partir de ce format de base.

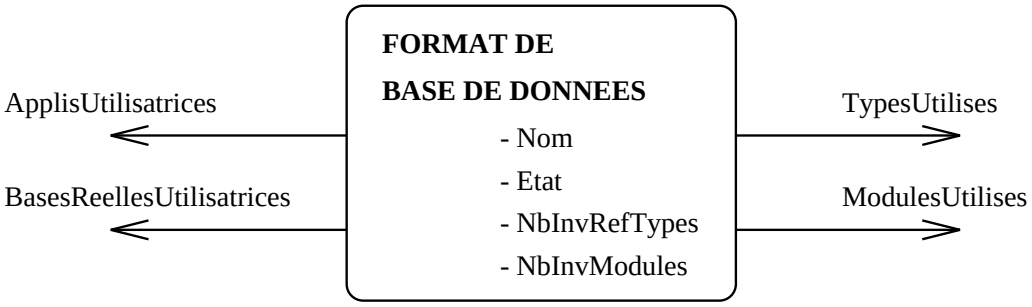


Fig. 5.5 : Boîte de l'environnement représentant un format de base de données

Les informations définies dans ces définitions de base de données vont aussi être utilisées pour la génération des applications PEPLOM exécutables car la structure des objets est définie dans leur code. Le lien "ApplisUtilisatrices" référence toutes les applications exécutables utilisant cette base. Toutes ces applications peuvent alors travailler sur n'importe quelle base "physique" qui suit ce modèle logique. La figure Fig. 5.5 montre par ailleurs les liens de dépendance qui lit une base "logique" aux types et aux modules qu'elle utilise. Cette unité est mise à jour comme les autres par le compilateur incrémental.

V.1.2 Les matérialisations de définitions PEPLOM

Nous avons vu que la production de programme avec PEPLOM se divise en deux phases. Une phase d'édition/validation des définitions dont nous avons décrit ci−dessus les différents éléments de définition et la manière dont le compilateur incrémental les gère. La deuxième phase permet de générer deux types d'entité de l'environnement qui vont être matérialisées : des exécutables d'applications PEPLOM et des instances de base de données sur lesquelles ces applications vont pouvoir travailler.

De la même manière que pour la première phase l'environnement offre des outils permettant d'éditer et de compiler (valider) des définitions, il va permettre dans cette phase "d'instanciation" d'éditer des définitions d'application exécutable ou de base matérialisée auxquelles sont ensuite appliquées des outils de génération. Les définitions servent à décrire de quelle manière nous voulons générer cette instance et à partir de quelles unités de définitions PEPLOM. Nous utilisons un langage spécifique pour ces deux nouveaux types de définition.

Une application exécutable PEPLOM a besoin essentiellement de deux informations : un module et une définition de base (format). Les autres informations qui peuvent intervenir sont des options de génération comme par exemple la vérification de dépassement des bornes d'un tableau ou encore la possibilité d'exécuter l'application en mode débogage. La définition d'une application exécutable pour la gestion d'une caisse d'assurance maladie basée sur les exemples définis précédemment s'exprime comme suit :

```
appligen GestEmp
{
module  GestionDuPersonnel;
base  BaseEmployes;
mode debug;
};
```

L'application exécutable de nom GestEmp est construite à partir du module `GestionDuPersonnel` et de la définition de base `BaseEmployes`. Le module associé à une application exécutable doit contenir une opération particulière de type "main" comme décrit dans la section IV.3. Cette opération est donc le point d'entrée de l'application générée. Ce module sert de racine au générateur qui va s'occuper de rechercher toutes les autres définitions nécessaires à la construction de l'application (fermeture transitive des liens définis à droite des boîtes − liens TypesUtilises, ModulesUtilises, SuperTypes, etc...).

La figure Fig. 5.6 montre l'unité de l'environnement qui décrit une application exécutable. Nous remarquons les deux références vers le module et la définition de base associés. Il existe aussi un champ d'état car l'application peut être incohérente notamment si le module ou la base référencée n'existe pas. Une fois cette définition prise en compte par l'environnement (stockée dans le catalogue), nous pouvons alors lui appliquer l'outil de génération de code qui peut être du O2C comme dans la première version [90] ou du code de plus bas niveau tel que du C (cf. VI).

Dans le premier cas, l'utilisateur de l'application doit l'exécuter sous le contrôle d'un système hôte (ici O2). Dans le deuxième cas, le binaire généré se comporte comme n'importe quel binaire géré par le système d'exploitation (en l'occurence Unix). L'intérêt est donc que l'utilisateur de l'application n'est confronté qu'à un seul environnement système qui est d'ailleurs celui qu'il utilise habituellement sur sa machine.

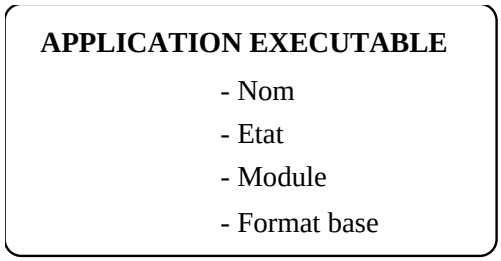


Fig. 5.6 : Boîte de l'environnement représentant une application exécutable

L'autre unité de l'environnement permet d'instancier une base de données. Pratiquement, cette base réelle définit un espace de données persistantes dans lequel une application exécutable va pouvoir stocker et récupérer des objets. Une base de données réelle est associée à un format de base et se définit de la manière suivante :

```
dbgen  EmployesBull
{
base  BaseEmployes;
};
```

Nous définissons dans cet exemple une instance de base de données dont le nom est `EmployesBull`. Elle va contenir les données nécessaires à la gestion des employés de Bull et utilise le format de base défini par `BaseEmployes`. Il décrit la structure physique (gestion mémoire) des données de cette base réelle.

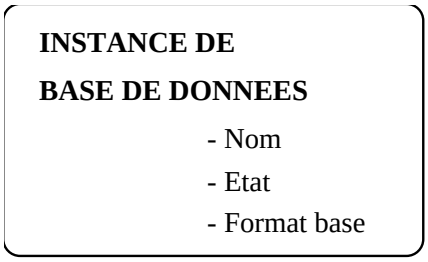


Fig. 5.7 : Boîte de l'environnement représentant une instance de base

La boîte décrite par la figure Fig. 5.7 est la dernière utilisée par le compilateur incrémental. De même que pour les applications exécutables, lorsqu'une définition d'instance de base est insérée dans le catalogue par le compilateur, cette définition peut être erronée. Nous avons donc un champ "Etat" qui donne le statut de cette définition comme pour toutes les autres unités de définition vues auparavant. Le dernier champ "Format base" correspond à la référence vers la définition de base associée.

V.2 Les espaces de définitions et d'utilisation

Un apport important des SGBD relationnels a été d'introduire des mécanismes de structuration et de protection : les schémas. Ils définissent un espace associé à un utilisateur (son propriétaire) dans lequel celui−ci peut notamment créer, détruire ou modifier des définitions de relations aussi bien que les relations elles−mêmes. Chaque espace défini par un schéma est alors isolé des autres. Des mécanismes d'autorisations et de droits sont mis en œuvre pour pallier la restriction qu'engendre l'isolation. Le propriétaire d'un schéma est ainsi responsable des relations qui y sont définies. Il peut alors donner des autorisations à d'autres utilisateurs sur ses relations suivant certains droits comme le droit d'accès en lecture et écriture d'une relation ou en lecture seulement ou encore le droit de modifier le schéma lui−même (i.e. ajout ou destruction de relations).

Dans le monde relationnel, la seule entité de définition est la relation. C'est donc à cette entité que sont attachés les droits d'accès pour un utilisateur donné. Si nous voulons attacher les droits à un niveau plus fin comme l'attribut d'une relation, le seul moyen est alors de passer par le mécanisme de vues. La fonctionnalité voulue ici est en fait de restreindre la vue qu'a un utilisateur des différentes colonnes d'une relation. Nous pouvons assimiler celle−ci à une restriction du "type" de la relation manipulée. Hors la sémantique des vues dans les systèmes relationnels est qu'elles calculent une nouvelle relation.

PERSONNE (table)

nom	age	num_secu	sexe
Dupont	23	NSS1	M
Durant	56	NSS2	M
Dupont	23	NSS3	M

PERSONNE_BIS (vue = **select** nom, age, sexe **from** PERSONNE)

nom	age	sexe
Dupont	23	M
Durant	56	M

Fig. 5.8 : Utilisation de vues pour la protection

De plus, c'est donc principalement parce qu'une relation désigne à la fois un type de données et les données elles−mêmes que nous aboutissons au problème de mise à jour à travers les vues. En effet, dans notre cas, nous voulons pouvoir modifier la relation par rapport au schéma que nous en connaissons. Pour illustrer cette difficulté rencontrée par les systèmes relationnels, nous définissons un exemple décrit par la figure Fig. 5.8 dans lequel nous avons une table qui contient des n−uplets représentant des personnes.

Cette table est définie dans un schéma S1 et nous voulons donner à un schéma S2 le droit de consulter et modifier la table PERSONNE mais seulement pour les colonnes nom, age et sexe. Nous allons pour cela utiliser le mécanisme de vue en définissant comme le montre la figure Fig. 5.8 une nouvelle relation PERSONNE_BIS qui est une projection de la relation PERSONNE sur les trois attributs par lesquels nous sommes intéressés. La sémantique de cette vue dans le contexte relationnel est qu'elle définit une nouvelle relation. Or faire une mise à jour à travers celle−ci correspond à vouloir modifier les n−uplets de PERSONNE et non ceux de PERSONNE_BIS. La sémantique d'une vue est alors différente suivant que nous opérons une consultation ou une mise à jour.

Cette sémantique pour la mise à jour pose d'autant plus de problème qu'une vue peut engendrer une perte d'information difficilement gérable. En effet, une projection peut engendrer une perte d'information comme pour PERSONNE_BIS qui n'a plus que deux n−uplets au lieu de trois. Si dans cet exemple, nous voulons incrémenter l'age de "Dupont", il y a deux n−uplets dans PERSONNE correspondant au n−uplet "Dupont ..." de PERSONNE_BIS. La modification d'un n−uplet de la vue engendre alors la modification de plusieurs dans la table PERSONNE. Par ailleurs, une telle mise à jour ne peut pas toujours être effectuée notamment dans le cas de vues faisant intervenir plusieurs tables.

Dans l'environnement de PEPLOM, nous voulons séparer clairement la notion de vue dans un contexte de protection en deux catégories : les vues sur les définitions et les vues sur les données. Par ailleurs, l'environnement d'un LPBD est plus complexe que celui d'un SGBD relationnel à cause du plus grand nombre de définitions à gérer. Les utilisateurs du SGBD ont peu de rôles à remplir. Nous pouvons différencier essentiellement deux rôles : les utilisateurs finaux qui manipulent les relations et les utilisateurs "gérant de base" (administrateurs) qui gèrent les schémas de base (définitions de relations). Dans un LPBD tel que PEPLOM, nous distinguons aussi ces deux niveaux. D'un côté, nous avons ceux qui gèrent les définitions qui sont les programmeurs, et les gérants de bases et d'applications. D'un autre côté, les utilisateurs finaux utilisent soit des instances de base à l'aide d'outils tels que des navigateurs ou à partir d'un langage de requêtes *ad hoc*, soit des applications exécutables.

V.2.1 Les programmeurs

Parmi les différentes catégories d'utilisateurs de l'environnement du LPBD PEPLOM introduites ci−dessus, la plus importante vis à vis du langage est celle des programmeurs.

Dans l'environnement PEPLOM, un programmeur définit un espace de définitions. Ces définitions peuvent être n'importe quelle définition vue auparavant. Une définition ne peut être modifiée que par le programmeur associé à l'espace auquel elle appartient. Les applications exécutables sont toujours privées à l'espace du programmeur et servent ici seulement pour générer des exécutables dans le but d'effectuer la mise au point. Le programmeur a donc la responsabilité des définitions de son espace de développement.

Toutes les définitions définies jusqu'ici appartiennent toujours à un espace de programmeur et à un seul. Un espace contient donc un ensemble de définitions lui appartenant. Comme le montre la figure Fig. 5.9, les espaces de définitions sont deux à deux disjoints. Or le but de l'environnement n'est pas d'isoler les programmeurs les uns des autres mais de les faire coopérer. C'est pour cela que nous introduisons un mécanisme de vue d'un espace de définitions pour définir les règles de coopération entre les différents espaces.

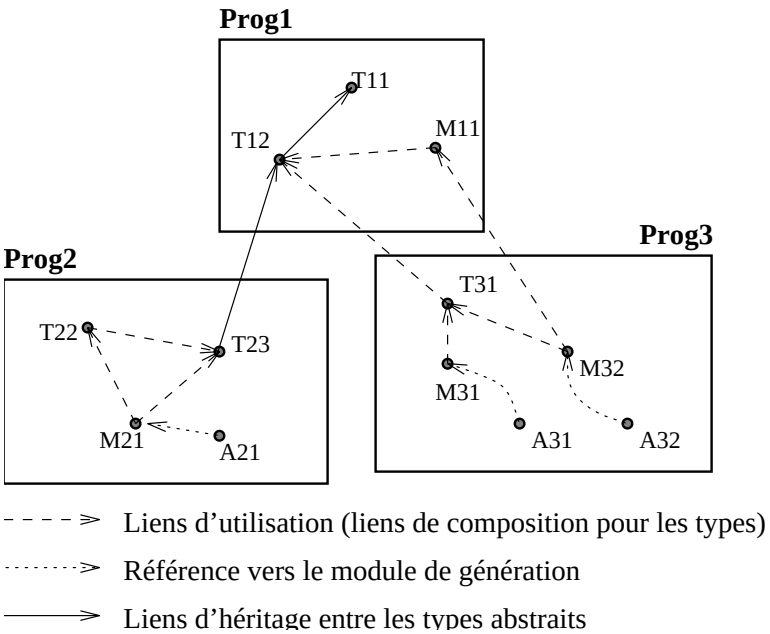


Fig. 5.9 : Espaces de définitions des programmeurs

Une vue d'un espace se définit comme un mécanisme de protection qui permet à un programmeur d'autoriser un autre programmeur à utiliser les définitions de son espace suivant certaines règles. Les premières règles qui s'appliquent sont celles imposées par le modèle de données de PEPLOM concernant notamment les règles d'héritage ou de composition.

Les vues permettent ensuite de définir des règles au−dessus de celles−ci qui sont du niveau de l'interface d'accès à une définition. Les unités de définitions dont nous pouvons offrir la visibilité à travers une vue sont essentiellement les définitions d'interfaces c'est à dire les types et les modules. La vue maximale qui peut être offerte d'un type ou d'un module correspond à l'ensemble des éléments publics de cette définition. Cette notion de vue appliquée au type abstrait dans PEPLOM s'apparente à la notion de classe virtuelle telle que définie dans [3] et est aussi très liée à la vision externe que nous pouvons vouloir donner d'une base de données [63].

Tous ces éléments publics ont un nom qui les identifie de manière discriminante dans la définition. Nous pouvons alors définir une vue restreinte de cette définition en donnant l'ensemble des noms publics visibles. Si nous supposons que les types définis précédemment

appartiennent au même espace et que la définition de la valeur des objets de chacun de ces types est publique, nous pouvons définir comme illustration la vue suivante :

```
viewdef  ConsultImpot
{
users P1, P2, ..., Pn;
types {
    personne(value.Nom, value.Adresse.Ville,  Age);
    imposable (all);
    };
modules {
    ...
    };
};
```

Dans cet exemple, le programmeur qui, soit possède les définitions de `personne` et `imposable`, soit en a la vision nécessaire, autorise les utilisateurs P1, P2, ..., Pn à utiliser ces deux types dans leurs définitions. Nous remarquons que dans le cas du type `personne`, ces programmeurs ne peuvent utiliser que les champs `Nom` et `Adresse.Ville` de la valeur associée au type, ainsi que la méthode `Age`. Dans le cas du type `imposable`, tous les éléments publics ou privés visibles (dans le cas d'un type visible à travers une vue) du type sont utilisables.

Nous définissons donc les espaces de programmeur ainsi que les vues que nous pouvons leur associer de la manière suivante :

1. e (u, D, V) est un espace de définitions pour programmeur où
   - **u** identifie le programmeur qui est associé à l'espace **e**.
   - D = {d} est l'ensemble des définitions qui appartiennent à l'espace **e**. **d** peut être, par exemple, soit un type, soit un module, soit une méthode ou une opération, soit une application exécutable.
   - V = {v} est l'ensemble des vues **v** qu'a le programmeur associé à **e** sur les autres espaces dont il peut utiliser les définitions.
2. v (**e**, DM) est une vue sur un espace de définitions où
   - e est l'espace sur lequel est définie la vue.
   - DM = {(d, M)} est l'ensemble des définitions **d** rendues visibles par la vue à travers le masque M = {n}. d est principalement soit une définition de type, soit une définition de module. M est l'ensemble des noms **n** des éléments publics ou privés visibles pour la définition **d**. Tous les éléments de la définition **d** sont visibles si M = vmax(d). vmax est une fonction qui, s'appliquant à une définition, rend le masque définissant la vue maximale de cette définition.

Les espaces associés aux programmeurs et les règles de coopération entre ces espaces sont pris en compte par le compilateur incrémental. Ils servent donc de filtre au compilateur qui, lorsqu'il compile une nouvelle définition dans un espace, possède un contexte de toutes les autres définitions qui vont pouvoir être utilisées par celle compilée. Ce filtre peut être

défini comme un ensemble F = {(d, M)} contenant les définitions utilisables associées au masque à travers lequel elles sont utilisables. A partir de la définition d'un espace et de vues, nous pouvons définir ce filtre comme suit :

1. on calcule d'abord l'ensemble DF(e) de toutes les définitions utilisables dans un espace **e** munies de leur masque :

$$DF(e) = \{(d, vmax(d)) / d \in e.D\} \cup \cup_{(\forall v \in e.V)} v.DM$$

2. dans DF(e), il est possible qu'il y ait plusieurs fois la même définition avec plusieurs masques différents d'où la définition de F(e) :

$$F(e) = \{(d, U_{(\forall i \in 1..n)} M_i) / \exists n, (d, M_i)_{\forall i \in 1..n} \in DF(e)\}$$

La définition ci−dessus spécifie notamment que si dans un espace il y a plusieurs vues de la même définition, c'est à dire plusieurs masques associés à celle−ci, la vue résultante sur cette définition est définie par un masque qui est l'union des masques des différentes vues. Nous avons décrit les espaces associés aux programmeurs et la manière dont ils coopèrent. C'est donc le compilateur incrémental qui fixe les règles de développement coopératif.

V.2.2 Les administrateurs de bases de données

Dans un SGBD relationnel, l'administration des bases de données est une tâche importante. Cette tâche se retrouve dans tous les SGBD mais un apport déterminant des systèmes relationnels est qu'ils ont isolé clairement la gestion des aspects physiques d'une base (définitions d'index, distribution des données sur un réseau ou dans les nœuds d'une machine parallèle) de sa définition logique ou conceptuelle. De la même manière, nous voulons isoler les aspects liés à l'administration d'une base de données de la partie programmation dont une des fonctions est de définir le niveau conceptuel de cette base.

Pour cela, nous définissons une autre catégorie d'utilisateurs de l'environnement PEPLOM qui sont les administrateurs de bases de données. A un administrateur est aussi associé un espace de définition dans lequel nous trouvons exclusivement trois types de définitions : des définitions de format de base, des définitions d'instance de base et des définitions d'applications exécutables. Un administrateur a la vision de toutes les définitions représentant la définition conceptuelle des données d'une base (types abstraits, modules, etc...). Le rôle de l'administrateur est donc essentiellement de définir les bases de données et les applications qui utilisent ces bases. Il peut alors donner les droits d'accès à ces différentes entités pour les utilisateurs finaux. Ces droits sont exprimés dans des vues sur les définitions d'applications exécutables ou d'instances de base :

```
viewdef Vue1
{
users U1, U2, U3;
applis GestEmp;
bases EmployesBull;
};
```

Les utilisateurs finaux vont interagir avec les bases de données soit à travers les applications PEPLOM exécutables, soit à travers des outils de navigation ou des langages de

requêtes *ad hoc*. Ainsi, l'administrateur va−t−il pouvoir définir des vues sur les définitions PEPLOM pour créer des espaces de protection des données de la base. Il utilise pour définir ces vues le même mécanisme que celui employé pour la coopération entre les espaces de programmeurs. Si nous voulons par exemple deux vues différentes de la base EmployesBull pour les utilisateurs U2 et U3, nous pouvons définir les vues suivantes :

```
viewdef Vue_U2
{
users U2;
types {
    personne (value.Nom, extension);
    };
};
viewdef Vue_U3
{
users U3;
types {
    personne (all, extension*);
    employe (value.Position);
    };
};
```

Ces deux vues servent donc de filtres pour les accès aux données de la base par les utilisateurs U2 et U3 lorsque ceux−ci travaillent par exemple avec un outil de navigation. L'utilisateur U2 a accès aux objets de type `personne` dont il ne voit que le nom. Il a de plus le droit d'accéder à l'extension de ce type. Les objets de type `personne` sont donc les seuls objets qu'il peut accéder. Dans le cas de l'utilisateur U3, il peut accéder les objets de type `personne` (par l'interface publique du type) et ceux de type `employe`. Il peut manipuler l'extension du type `personne` et de tous ses sous−types (`employe` compris) mais seulement avec une vision comme `personne`.

Le mécanisme de vue proposée dans cette section remplit bien le rôle de protection des données suivant les droits d'un utilisateur. Il est complémentaire du mécanisme de base à deux niveaux sur les types ou les modules (parties privée et publique). Si nous avions du remplir cette fonctionnalité en faisant de la projection sur certaines parties d'un objet (en générant un nouvel objet), nous nous serions alors heurté au fait qu'un objet doit être une entité atomique pour que les méthodes lui soient toujours applicables. Les vues définies ici conservent le type des objets tout en en restreignant dans certains cas l'interface publique.

V.2.3 Les utilisateurs finaux

Les utilisateurs finaux ne peuvent définir aucune entité dans l'environnement. Ils sont essentiellement repertoriés par l'environnement pour leur permettre d'utiliser soit des bases de données physiques, soit des applications PEPLOM exécutables.

Dans le cas des bases de données, ils peuvent les utiliser à partir d'outils de l'environnement tels que des outils de navigation ou des moniteurs leur permettant d'exécuter des requêtes dans un langage *ad hoc* et de visualiser leurs résultats. Ces outils peuvent être paramétrés à partir des vues sur les bases qui auront été affectées par un

administrateur à l'utilisateur final comme défini dans la section V.2.2. Par ailleurs, comme les applications sont exécutables comme une commande externe du système d'exploitation de la machine, l'identification des utilisateurs ainsi que les mécanismes de droits gérés par l'environnement peuvent être implantés soit sur ceux offerts par le système − en l'occurence Unix−, soit par le programme généré avec un protocole de connexion faisant intervenir le catalogue.

V.2.4 Administration de l'environnement

Dans l'environnement décrit dans les sections ci−dessus, il est nécessaire d'avoir un utilisateur spécial qui puisse accéder à toutes les ressources de celui−ci. Nous définissons pour cela l'administrateur de l'environnement dont la tâche principale consistera à organiser l'environnement et notamment à définir ses utilisateurs ainsi que leur rôle (programmeurs, administrateurs ou utilisateurs finaux). Un utilisateur pourra aussi avoir plusieurs rôles et choisir le rôle qu'il se donne lorsqu'il se connecte à l'environnement.

L'organisation est un aspect important de l'environnement notamment concernant l'organisation du travail de développement. Dans ce contexte, il paraît important d'avoir une notion telle que le projet. Il serait donc intéressant pour cela d'introduire la notion de groupe d'utilisateurs pour pallier ce besoin. Nous pourrions alors définir des conglomérats d'espaces de définitions auxquels nous pourrions aussi attacher des vues et qui représenteraient des espaces de définitions de granularité inférieure. Cela impliquerait aussi qu'un programmeur pourrait avoir plusieurs rôles puisqu'il pourrait appartenir à plusieurs groupes. Cette notion parait néanmoins intéressante à introduire dans l'environnement de manière aussi à factoriser des vues. Le groupe pourrait aussi servir au niveau des utilisateurs finaux pour factoriser aussi les droits d'accès aux bases ou d'exécution des applications disponibles.

V.3 Architecture du prototype de l'environnement PEPLOM

Cette section décrit l'architecture de l'environnement de développement et d'administration de PEPLOM. C'est en fait l'architecture du premier prototype pour lequel un seul utilisateur à la fois peut travailler sur l'environnement. Nous décrivons notamment comment est gérée la base de composants (unités de définition PEPLOM) et la manière dont les outils y accèdent.

Nous avons vu précédemment que le niveau de sûreté de la base de composants se devait d'être au moins aussi bon que celui des données qu'ils représentent. Cela signifie qu'il faudrait un SGBD complet pour supporter cette base de composants. Les systèmes du type PCTE[84] sont certainement dans ce domaine actuellement les systèmes les mieux appropriés au support de telles bases de composants notamment pour ce qui concerne la gestion de la synchronisation des accès.

Par ailleurs, nous pouvons citer d'autres mécanismes de base que le système de support de la base de composants devrait offrir. Il parait effectivement très utile voire nécessaire de

pouvoir gérer des versions de composants pour le support de l'évolution du schéma des définitions. Ensuite, des mécanismes d'événement/condition/action paraissent aussi intéressants dans un contexte de développement coopératif tel que celui que nous voulons mettre en place. En effet, il semble important que lorsqu'un programmeur entreprend une certaine action, les conséquences de cette action sur l'environnement de programmation d'un autre développeur soit signalées.

Pour des raisons de simplicité et de rapidité de prototypage, les contraintes sur la sûreté de la base de composants ont été ignorées. Dans ce premier prototype, les composants et des liens entre ceux−ci sont gérés sous la forme d'une arborescence valuée définie et manipulée à l'aide de l'outil EMIR[70]. Un composant est donc une représentation pivot d'une définition du langage. Les codes sources associés aux définitions PEPLOM représentées par les composants sont stockés dans des fichiers du Système de Gestion de Fichiers d'Unix.

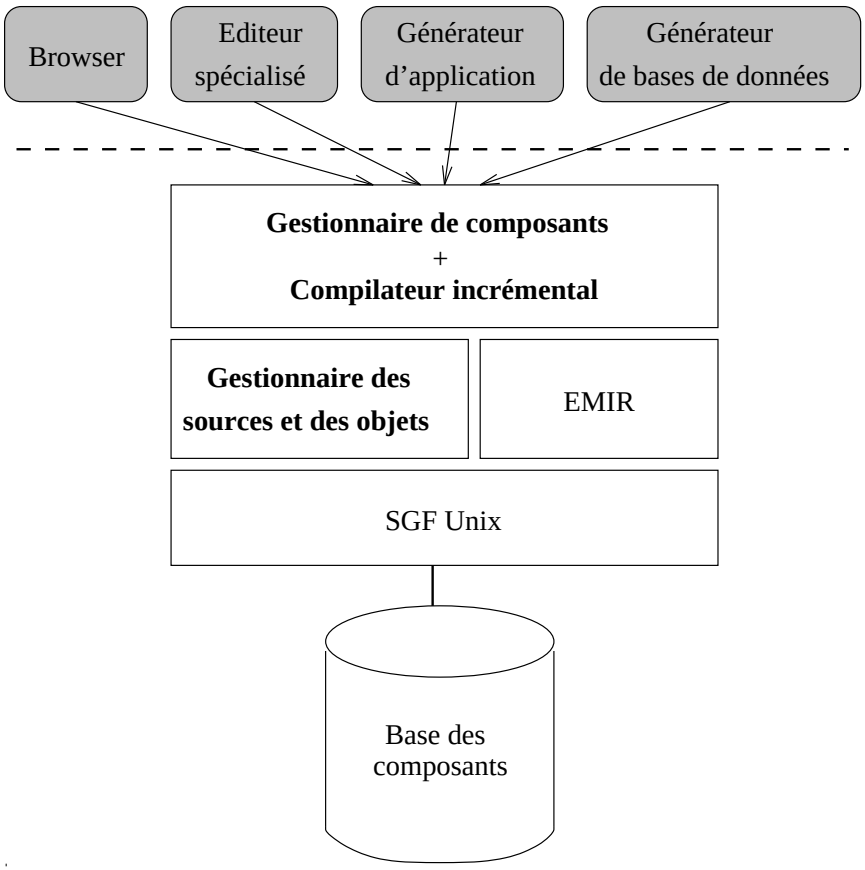


Fig. 5.10 : Architecture de la base de composants et des outils de l'environnement

Comme le montre la figure Fig. 5.10, les outils offerts aux utilisateurs de l'environnement s'appuient sur le gestionnaire de composants. La partie que nous nommons ici compilateur incrémental correspond en fait simplement à l'insertion d'un nouveau composant dans la base. Ce nouveau composant est soit un composant à créer, soit un composant à mettre à jour. Le composant inséré est alors la nouvelle valeur du composant modifié. Par contre, cette

fonction d'insertion va mettre en œuvre les procédures de propagation des conséquences de l'insertion du nouveau composant sur tous les composants qui en dépendent.

L'interface entre les outils et le gestionnaire de composants se compose d'un ensemble de fonctions permettant de manipuler chaque catégorie de composants. Par exemple, pour la manipulation des types, nous avons l'ensemble des fonctions suivantes :

- **`CatCreOrValType`** : signale la validité d'un type à tous les composants qui utilisent celui−ci dans leur définition.
- **`CatDelOrInvType`** : signale l'invalidité d'un type à tous les composants qui utilisent celui−ci dans leur définition.
- **`CatExistType`** : vérifie s'il existe dans le catalogue un composant représentant un type de nom donné.
- **`CatIsTypeErr`** : demande au catalogue si la définition d'un type est syntaxiquement erronnée.
- **`CatIsTypeInv`** : demande au catalogue si la définition d'un type est sémantiquement invalide.
- **`CatCreateType`** : insère un composant représentant un type dans le catalogue.
- **`CatDeleteType`** : détruit dans le catalogue un composant représentant un type.
- **`CatGetTypeFile`** : demande au catalogue le fichier contenant le code source de la définition d'un type.
- **`CatGetTypeTree`** : demande au catalogue l'arbre EMIR de définition du type donné.

Un peut remarquer dans un premier temps que cette interface contient un grand nombre de fonctions et que celles−ci sont liées à une catégorie de composants. Le niveau de cette interface est donc inadéquate si nous voulons faire évoluer la structure de la base de composants.

De plus, dans ce prototype, le compilateur incrémental est en partie mis en œuvre par les utilitaires puisque ceux−ci peuvent par exemple demander l'invalidation d'un composant. Il serait donc intéressant de cacher totalement cet aspect aux différents utilitaires en faisant du compilateur incrémental une fonctionnalité interne au gestionnaire de composants.

Par ailleurs, nous envisageons de faire du gestionnaire de composants un serveur. L'interface du gestionnaire décrite ici représente donc l'ensemble des fonctions d'accès à ce serveur. Dans une telle perspective, il paraît plus intéressant d'avoir une interface plus simple et plus générale du type lecture/écriture de composants.

La base de composants est donc un réseau dont les nœuds (les composants) sont des représentations pivot de définitions du langage. Le code source de ces définitions doit être conservé puisqu'un programmeur peut à tout instant modifier cette définition. Ces codes sources sont donc stockés dans une arborescence de fichiers Unix (cf. Fig. 5.11) dont la sémantique est déterminée par la base de composants.

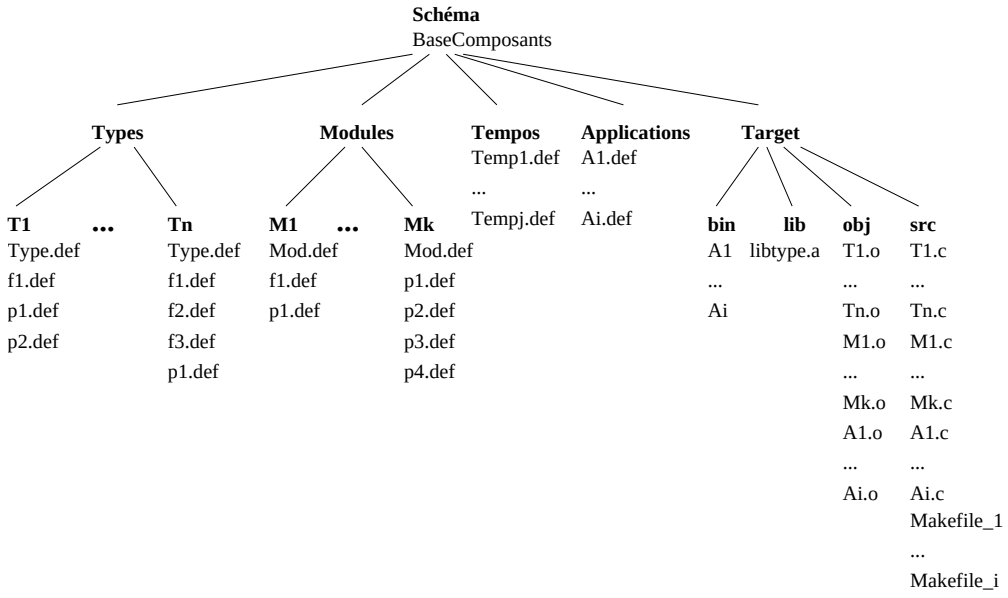


Fig. 5.11 : Arborescence des fichiers contenant les sources et codes des définitions PEPLOM

La figure Fig. 5.11 montre comment la base de composants et leurs fichiers associés (fichiers sources, objets ou binaires) sont organisés dans une arborescence Unix. Nous avons tout d'abord un fichier dans le répertoire principal qui s'appelle "BaseComposants" qui contient tous les composants dans leur format de code pivot ainsi que toutes les interconnexions qui les relient. C'est donc une arborescence EMIR qui est chargée en bloc par le gestionnaire de composants lorsque celui−ci est activé par un outil. Pendant toute l'activité de l'outil, l'arborescence est modifiée en mémoire et est sauvegadée seulement lorsque l'outil termine son action. Ce mode de fonctionnement entraine un grave problème en cas de panne puisque les fichiers contenant les sources des définitions représentées dans la base de composants sont alors incohérents par rapport aux composants de la base dont le nouvel état a été perdu. Ce problème devra être résolu dans la version suivante du gestionnaire de composants.

Le schéma est ensuite composé de plusieurs sous−répertoires contenant les sources de différentes catégories de composants ainsi que d'autres contenant des objets ou des binaires :

1. Le répertoire **`Types`** : il est composé d'un ensemble de sous−répertoires correspondant à autant de définitions de types (abstraits ou concrets). Le gestionnaire de composants associe un identificateur à chaque type. Cet identificateur est alors utilisé pour définir le nom du répertoire (construit par le gestionnaire de sources et de codes) associé à un type (par exemple T_1, ..., T_n). Le répertoire d'un type contient alors au moins un fichier contenant sa définition

(Type.def) et dans le cas d'un type abstrait, les fichiers contenant la définition des fonctions (f_i.def) ou procédures (p_i.def)qui lui sont associées. Les noms des fichiers associés aux méthodes sont construits comme pour les types à partir de leur identificateur au sein de ce type.

2. Le répertoire **`Modules`** : il est aussi composé d'un ensemble de sous−répertoires contenant des définitions de modules. Les noms des répertoires sont construits par le gestionnaires de sources et de codes comme pour les types. Ces sous−répertoires contiennent au moins un fichier du code source du module et des fichiers f_i.def ou p_i.def pour les définitions d'opérations associées au module.
3. Le répertoire **`Tempos`** : lorsqu'un programmeur tente d'insérer une nouvelle définition syntaxiquement erronée dans le catalogue, le compilateur incrémental n'est pas capable de la traiter comme une définition valide ou sémantiquement incorrecte. Elle est alors temporairement stockée dans un fichier de ce répertoire car nous voulons que le programmeur puisse la retrouver. Ce fichier est détruit lorsque la définition peut être traitée normalement par le compilateur.
4. Le répertoire **`Applications`** : il contient les fichiers correspondant aux codes sources des définitions d'applications exécutables. Ils sont essentiellement utilisés par les outils de génération de code.
5. Le répertoire **`Target`** : ce répertoire contient tous les fichiers générés par les outils de génération de code à savoir les fichiers objets associés aux définitions PEPLOM qui sont des fichiers contenant du code source C, les fichiers objets Unix générés par la compilation des fichiers C précédents, et les fichiers binaires exécutables générés pour chaque application exécutable. Il est composé de quatre sous−répertoires qui sont :
    - **`src`** qui contient les fichiers C générés pour les définitions PEPLOM. Ils sont de quatre sortes : les T_ic qui sont les fichiers associés aux types, les M_i.c qui sont ceux associés aux modules, les A_i.c qui sont ceux associés aux applications exécutables et les Makefile_i qui sont les fichiers de construction associés à chacune des applications exécutables A_i. Il y a de plus le fichier Makefile_t qui construit séparément la librairie contenant le code de tous les types abstraits.
    - **`obj`** qui contient tous les fichiers objets Unix générés par la compilation des fichiers C du répertoire **`src`** (T_io, M_i.o, A_i.o).
    - **`lib`** qui contient la librairie des types abstraits.
    - **`bin`** qui contient tous les programmes exécutables Unix associés à chaque application exécutables A_i. Ce binaire est directement exécutable comme n'importe quel programme Unix.

Nous aurions pu choisir de stocker tous ces différents fichiers d'une manière plate dans un seul répertoire. Outre le fait d'une plus grande lisibilité notamment pour la mise au point,

cette solution à base de sous−répertoires offre une possibilité intéressante. Cela permet en effet de pouvoir faire des versions de définition complète d'un type ou d'un module par exemple simplement par copie du répertoire associé. Nous avons vu auparavant qu'un mécanisme de version est très important. Le gestionnaire de composants peut alors l'utiliser soit pour historiser un composant détruit, soit pour faire une version d'un composant modifié.

V.4 Conclusion

Le prototype décrit dans cette section est opérationnel. Il a de nombreuses limitations telles que l'utilisation du gestionnaire de composants par un seul outil à un instant donné, le manque de souplesse pour l'évolution du schéma de la base de composants ou encore sa vulnérabilité face aux pannes. Il a néanmoins permis de mieux comprendre les fonctionnalités qu'il doit offrir aux outils et notamment le niveau d'interface nécessaire.

Aucune des fonctionnalités décrites dans la section V.2 n'est pour l'instant supportée dans le prototype. Cet espace de l'environnement est néanmoins très intéressant à étudier dans le cadre de développements coopératifs. De tels mécanismes permettent effectivement d'envisager une méthodologie de développement d'applications de base de données à objets.

Le prototype a été écrit en C et a une taille d'environ 30000 lignes de code pour l'ensemble des modules de gestion des fichiers sources et objets, gestion de la base de composants, compilateur incrémental et outils d'édition/validation des définitions PEPLOM.

Chapitre VI

Génération d'applications PEPLOM exécutables

L'air retentit d'un patois bizarre où s'entrecroisaient les fissures, les rimayes, le "Grépon−à−l'envers", la "Mummery" et les pitonnages. Les mérites comparés du Mont Blanc par la Brenva et de la Bionnassay par la face nord furent à l'ordre du jour, et tout se consomma dans un immense, dans un colossal mépris pour le reste du monde en général, et plus particulièrement pour les pâles tribus à faux cols qui continuaient à supputer dans les coins les chances d'une exploration à la Mer de Glace.

Samivel − "L'amateur d'abîmes"

Ce chapitre décrit l'environnement d'exécution mis en œuvre pour le langage PEPLOM. Nous donnons tout d'abord les différents niveaux d'abstraction dans lesquels se décompose cet environnement. Nous essayons brièvement de placer cet environnement dans le contexte d'une architecture client/serveur. L'approche que nous avons suivie tend à remonter le plus possible la gestion des objets, aussi bien structurelle que sémantique, dans le compilateur (générateur de code). Nous étudions dans la seconde section la machine abstraite sur laquelle va s'appuier le compilateur pour gérer les objets. L'intérêt de cette couche est quelle offre la possibilité de faire grossir dynamiquement des objets. La section suivante décrit les différentes fonctionnalités actuellement supportées par le compilateur. La gestion des expressions ensemblistes fait partie des fonctionnalités non implantées ; elle représente un travail important qui sera traité ultérieurement. Enfin, nous donnons avant la conclusion quelques mesures qui ont été faites à partir du "Cattell Benchmark"[2 9] qui valide de manière intéressante les choix qui ont été faits. Il est important de savoir que toutes les descriptions faites dans ce chapitre sont généralement très succintes. Ce choix est essentiellement lié à un problème de place.

VI.1 Architecture de l'environnement d'exécution de PEPLOM

La première contrainte qui guide nos choix de conception de l'environnement d'exécution est l'efficacité. Pour cela, nous avons choisi de pousser l'approche compilée le plus loin possible. Pour citer un exemple, nous voulons que l'accès à un champ d'un objet soit défini statiquement par le générateur de code comme le fait un compilateur C++. Il n'y aurait donc

pas interprétation par une machine abstraite comme c'est le cas dans O2. Ce souci d'efficacité nous a conduit à limiter le nombre de couches logicielles et à simplifier leur complexité.

La deuxième contrainte qui est très importante dans un contexte base de données est la sécurité. Elle peut être assurée totalement ou partiellement à deux niveaux :

- plus le langage est sûr et plus la sécurité des données qu'il manipule est grande. Il y a deux aspects qui mettent en cause la sûreté de PEPLOM hormis les problèmes d'opérateurs arithmétiques (division par zéro, dépassement de capacité, etc...) : des indices de tableaux hors bornes et des références invalides. Seul le problème des indices de tableaux peut introduire une corruption de données étrangères aux éléments de ces tableaux. Il est néanmoins possible de générer des vérifications dynamiques de bornes (même si cela coûte cher), un mécanisme d'exception prenant ensuite la relève pour gérer tous les cas d'erreur.
- la couche système peut résoudre pas mal de problèmes de sécurité. Le mécanisme de transaction est déjà une fonction qui augmente grandement la sécurité. Les LPBD offrant une sureté moindre que celle de langages comme SQL, il peut être intéressant d'isoler les espaces d'adressage de chaque transaction. L'approche client/serveur peut permettre une telle isolation.

Nous nous intéressons exclusivement, dans cette partie, parmi les différentes architectures client/serveur, à l'étude de celles qui conviennent le mieux à notre système. Nous allons voir que l'implémentation actuelle ne suit aucune de ces architectures, ceci étant du au fait que la couche de plus bas niveau a été récupérée dans le prototype DBS3[19][20] du SGBD relationnel développé dans le projet EDS pour lequel les besoins étaient différents.

VI.1.1 Organisation logique des différentes couches logicielles

Nous avons vu précédemment que, pour des raisons d'efficacité, nous voulons que le code généré pour une application PEPLOM soit d'un niveau le plus bas possible. Un autre aspect, garant d'efficacité pour un SGBD, est de limiter au maximum le nombre de couches entre le langage de manipulation des données et la gestion mémoire (centrale et secondaire) de ces données.

Or du point de vue de la manipulation des données, le plus efficace que nous puissions faire est d'accéder directement à leur structure en mémoire centrale. Le code généré va donc par exemple effectuer l'accès à un champ comme dans un langage de programmation classique tel que C : par simple accès mémoire. En revanche, nous avons vu (cf. II.3.1) que toutes les structures utilisées (constructeurs dynamiques ensemble et liste) n'offrent pas la possibilité de définir statiquement leur manipulation, comme dans le cas précédent, par génération d'un accès mémoire. Pour ceux−ci, il est nécessaire d'avoir une couche qui ne peut être qu'interprétée.

Concernant le transfert des données de la base entre la mémoire centrale et l'espace disque, l'idéal est que le format des objets PEPLOM sur disque soit le même que celui de la mémoire centrale. C'est cette technique que nous utilisons en nous appuyant sur une couche de plus bas niveau : le paginateur.

Par contre, cette couche offre une unité d'allocation qui a une granularité fixe et trop importante pour le plus grand nombre des objets. Pour pallier ce problème, nous introduisons une couche de gestion mémoire supplémentaire entre le paginateur et le code généré.

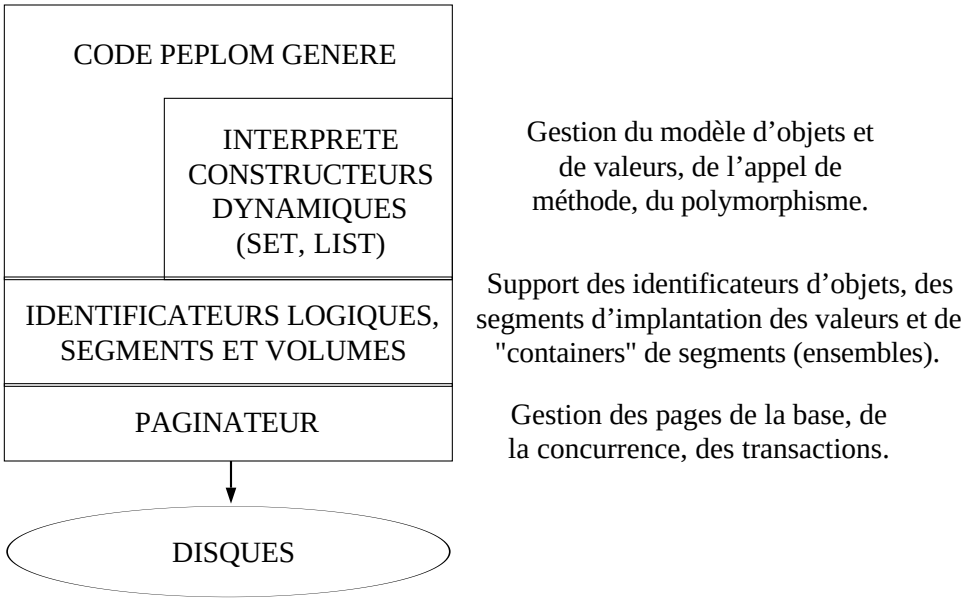


Fig. 6.1 : Architecture logicielle de l'environnement d'exécution de PEPLOM

Comme le montre la figure Fig. 6.1, l'architecture de l'environnement d'exécution du langage PEPLOM se décompose donc en trois niveaux (du plus bas au plus haut) :

1. La couche de pagination : elle effectue un couplage ("mapping") des pages de la base de données dans l'espace virtuel d'une application. C'est elle qui s'occupe de gérer les accès concurrents à ces pages et les transactions. Elle utilise une technique de mémoire à deux niveaux ("two level store")[45].
2. La couche de support d'objets : cette couche est construite au−dessus du gestionnaire de pages. Elle offre la notion d'identificateur d'objet qui permet de désigner un segment de mémoire. Ces segments sont aussi gérés à ce niveau et sont des unités de taille inférieure à la taille d'une page. Ils sont regroupés dans des volumes.
3. Le code généré et le support des constructeurs dynamiques : le code PEPLOM généré s'appuie directement sur la couche de support d'objets. Pour les constructeurs dynamiques, un interpréte a été construit au−dessus de la couche support d'objets. Il est utilisé par le code généré.

Dans la suite, nous allons décrire plus précisément certaines parties des deux couches que nous avons implantées partiellement (le paginateur étant un logiciel que nous avons récupéré dans le projet EDS) : la couche de support d'objets et le générateur de code.

VI.1.2 Architecture client/serveur

La décomposition client/serveur est principalement utilisée dans un contexte distribué. Elle permet d'assurer une meilleure localité des données utilisées dans le mesure où le service s'exécute là où ces données sont disponibles. Pour l'instant, nous nous cantonons à une vision centralisée de l'environnement d'exécution.

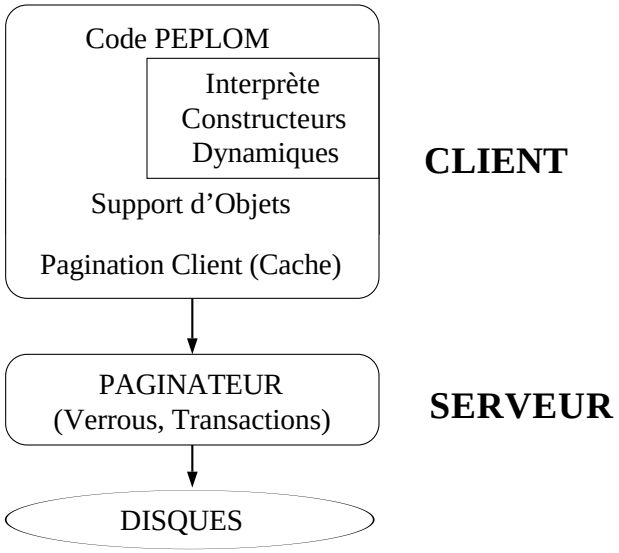
Cette décomposition est néanmoins utile dans un contexte centralisé. En effet, les données de la base sont partageables entre différentes applications PEPLOM. Ce partage peut être réalisé soit à l'aide d'une mémoire partagée couplée dans chaque espace virtuel des applications ou par un serveur qui délivre par message les données nécessaires.

VI.1.2.1 Les différentes solutions

Avant d'étudier l'architecture logicielle implémentant la décomposition client/serveur, il faut déterminer à quel niveau de l'environnement d'exécution nous voulons faire ce découpage. L'architecture de l'environnement d'exécution nous offre seulement deux possibilités : faire un serveur du niveau machine support d'objets ou du niveau serveur de pagination. Nous retrouvons d'ailleurs ces deux choix dans d'autres systèmes :

- O2 et ObjectStore gèrent un serveur de pages.
- Versant gère un serveur d'objet que nous pouvons assimiler à notre gestionnaire de segments.
- Ontos offre la possibilité de gérer les deux niveaux de serveur suivant les besoins de l'application.

Les deux approches ont leurs avantages et leurs inconvénients. Le serveur d'objets a l'avantage de permettre une concurrence d'accès plus fine (plus de parallélisme). Par contre, il nécessite une gestion du défaut d'objet qui est plus lourde en structures de contrôle qu'un paginateur pour une même quantité de données. De plus, il a tendance à accroitre le taux de communications entre le serveur et le client.



Dans le cas du paginateur, les arguments sont symétriques. Le déréférençage d'un objet ne provoque pas nécessairement un envoi de message au serveur car il peut être contenu

dans une page qui a été chargée pour un défaut d'objet précédent. Les informations de contrôle sont évidemment moins coûteuses. Par contre, la concurrence est plus faible qu'avec l'autre approche.

L'approche serveur de pages semble plus performante même si elle diminue la concurrence lorsqu'on a à faire à de petits objets. C'est donc plutôt vers cette solution que nous penchons, d'autant plus qu'elle paraît plus simple à implanter et que nous avons déjà un support de ce type. L'architecture logique de découpage client/serveur de l'environnement d'exécution défini dans la figure ci−dessus peut être implantée de différentes manières.

Le cache par exemple peut être une mémoire partagée entre les différents clients et le serveur, comme dans O2, lorsque clients et serveur sont sur la même machine. Le serveur traite les requêtes une à une ce qui assure la synchronisation pour l'allocation de verrous sur les pages de la base. L'avantage est que les pages partagées en lecture ne sont chargées qu'une seule fois (un cache unique) ce qui n'est pas le cas avec la solution suivante. Le défaut de cette approche, surtout dans le contexte d'un LPBD qui n'est pas généralement un langage sûr, est que les contextes base de données associés à chaque transaction des clients ne sont pas isolés les uns des autres. Une page liée à une transaction peut alors être abimée par le code incorrect exécuté dans une autre transaction.

Une autre approche, offrant plus de sécurité, consiste à ce que chaque client ait son propre cache (espace virtuel non partagé) qui ne contient que les pages qui sont liées à la transaction qu'il exécute. Cette solution est utilisée par les C++ persistants qui ont un serveur de pagination[67]. C'est d'ailleurs la seule qu'ils peuvent utiliser s'ils veulent garder la compatibilité avec la gestion de la résolution tardive de code utilisée par C++. En effet, comme nous le verrons dans la suite, C++ introduit dans les objets des adresses liées au code de l'application qui les utilise. Ces adresses doivent donc être rafraichies à chaque nouveau couplage de page dans le cache du client.

Cette dernière approche est intéressante du point de vue de la protection inter−transactions offerte. La première approche pourrait néanmoins être utilisée aussi dans le contexte de PEPLOM car son format d'objet est compatible avec celle−ci. Cet aspect de l'environnement d'exécution devra de toute manière être revu dans les évolutions futures du prototype car l'implantation actuelle, que nous décrivons ci−dessous n'est pas satisfaisante.

VI.1.2.2 La solution utilisée par le prototype

Le prototype actuel n'utilise pas une approche client/serveur. L'approche mémoire partagée que nous utilisons est issue du prototype du paginateur que nous avons récupéré dans le système DBS3 (cf. Fig. 6.2). Dans ce dernier, le découpage client/serveur est effectué, comme pour la plupart des systèmes relationnels, au niveau du modèle relationnel.

Le client envoie ses requêtes au serveur qui est chargé de les exécuter. Une fois le résultat calculé, il est retourné au client. L'interraction entre le client et le serveur est d'un niveau sémantique élevé (l'interface client/serveur connaît le modèle des données échangées) et d'une granularité forte (échange de relations). Cette approche se prête bien à

des systèmes relationnels. Nous pouvons observer que le paginateur est partie intégrante du serveur qui exécute la machine relationnelle dans le même processus.

Nous utilisons donc le paginateur de la même manière. Nous avons ainsi construit la machine de support d'objets comme une bibliothèque utilisant les services de la bibliothèque du paginateur. Le code généré, ainsi que la bibliothèque de l'interprète des collections, sont ensuite construits en utilisant les services de la bibliothèque du support d'objets. Le processus, exécutant une application PEPLOM, exécute donc aussi le code du paginateur. Le partage des données de la base en mémoire s'effectue par le fait que le paginateur de chaque application PEPLOM, qui s'exécute sur la même base, couple ses pages dans un segment de mémoire partagé Unix. Il n'y a donc pas de décomposition client/serveur.

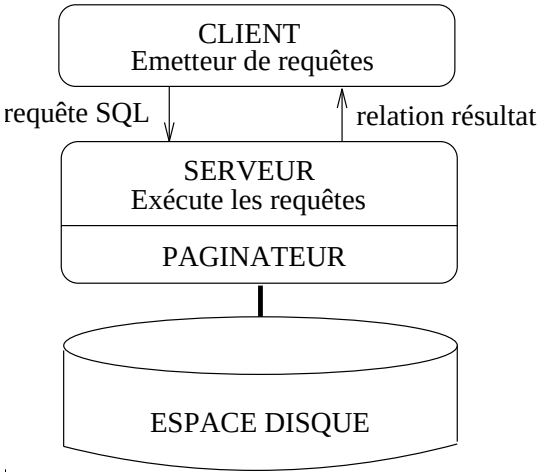


Fig. 6.2 : Architecture client/serveur du prototype DBS3

VI.2 Machine abstraite pour le support d'objets

L'objectif de la machine de support d'objets est d'offrir un niveau de gestion de la mémoire permettant l'implantation la plus efficace possible des objets PEPLOM. En fait, cette couche n'est absolument pas liée à un langage quelconque ni même à un modèle de données. Elle pourrait, par exemple, très bien être utilisée pour supporter un SGBD relationnel.

Le paginateur est un niveau de gestion mémoire trop primitif pour supporter des objets. Il n'est pas pensable par exemple d'allouer une page par objet dans le cas de petits objets. Or nous avons observé que, dans beaucoup d'applications, les petits objets (une centaine d'octets) sont largement majoritaires. De plus, si nous considérons que la taille des pages gérées par le paginateur est par exemple de 8 kilo−octets (Ko), il est alors possible d'y regrouper 80 objets de 100 octets. Nous avons donc décidé, dans un premier temps, que le niveau de gestion mémoire que nous offrons permette d'allouer des zones de mémoire contiguë de granularité inférieure à la page : les segments.

Un autre point important pour ce gestionnaire est qu'il doit être capable de supporter un modèle de données de type objet complexe. Vu globalement, de tels objets peuvent être

amenés à grossir car ils utilisent des constructeurs "dynamiques" (ensemble, liste, etc...). Une solution couramment utilisée est alors de décomposer les objets à la frontière des constructeurs (chaque partie de l'objet est dans une zone mémoire indépendante désignée par sa partie englobante). Cette technique est pénalisante à plusieurs points de vue pour de petits objets de taille variable :

1. dans l'hypothèse de petits objets, toutes les parties de l'objet sont censées tenir dans une page. Nous devons donc regrouper toutes ces parties dans la même page pour éviter de multiplier les entrées/sorties. Il devient alors très compliqué d'avoir d'une part, toutes les parties de l'objet, et d'autre part, plusieurs objets semblables dans une même page (regroupement dans deux dimensions).
2. toutes les informations de liens entre les différentes parties peuvent être très coûteuses en espace mémoire.
3. cette couche de gestion mémoire doit pratiquement supporter le modèle structurel utilisé.
4. l'allocation dans les pages permettant d'allouer des morceaux de n'importe quelle taille, cela amène à la longue (après de multiple allocations/désallocations) à un effritement de l'espace mémoire et donc à une perte d'espace qui peut devenir importante. Ce défaut peut être comblé par un mécanisme de recomposition de la mémoire qui est néanmoins complexe et coûteux à mettre en œuvre.

Le choix que nous avons fait est d'offrir une allocation par segments qui peuvent grandir (réallocation). Le problème dans ce cas est qu'un objet géré dans un segment peut bouger dans l'espace d'adressage changeant ainsi de désignation. C'est pour cela que nous introduisons les identificateurs logiques qui sont des relais d'indirection pour désigner un objet.

Enfin, un aspect essentiel dans les bases de données est de pouvoir accéder à des collections d'objets. Nous offrons pour cela les volumes qui sont des groupements de pages de segments. Un segment est donc toujours alloué dans un volume. Il est alors possible de parcourir tous les segments d'un tel volume à l'aide des curseurs qui leur sont associés.

Nous allons donc détailler maintenant les trois fonctionnalités de la machine de support d'objets : les identificateurs logiques, les segments et les volumes.

VI.2.1 Gestion d'identificateurs logiques

Nous avons vu précédemment que le paginateur est une mémoire de stockage à deux niveaux. Cela signifie que nous n'avons pas une correspondance directe entre une adresse disque et une adresse virtuelle. Nous allons distinguer deux niveaux d'adressage pour les identificateurs logiques persistants : le niveau disque et le niveau mémoire virtuelle.

Gestion sur disque

Ces identificateurs sont représentés sur le disque par une zone mémoire contenant des informations qu'ils sont censés maintenir. Dans notre cas, ces informations consistent principalement à définir des relais d'indirection vers des segments persistants de données.

Pour que ce relai soit efficace, il faut que cet identificateur, qui est donc utilisé pour référencer un segment, permette de retrouver le plus directement possible l'information qu'il représente.

Le déréférençage des identificateurs persistants peut s'opérer efficacement car ils sont stockés sous la forme d'une table implantée sur le disque. Cette table peut grandir au cours de l'évolution de la base. Un identificateur logique persistant a donc le format suivant :

INDICE TABLE	N° DISQUE INFO. PARAM.
32 bits	32 bits

Il est composé de deux mots de 32 bits. Le premier mot correspond à l'indice de l'identificateur logique dans la table de déréférençage disque. Le deuxième, divisé en deux sous−mots de 16 bits, contient dans le premier sous−mot, le disque contenant la table de déréférençage de l'identificateur. Le second sous−mot permet à l'utilisateur d'y stocker une information quelconque qui lui est nécessaire (l'environnement d'exécution l'utilise notamment pour différencier les identificateurs persistants de ceux qui sont temporaires).

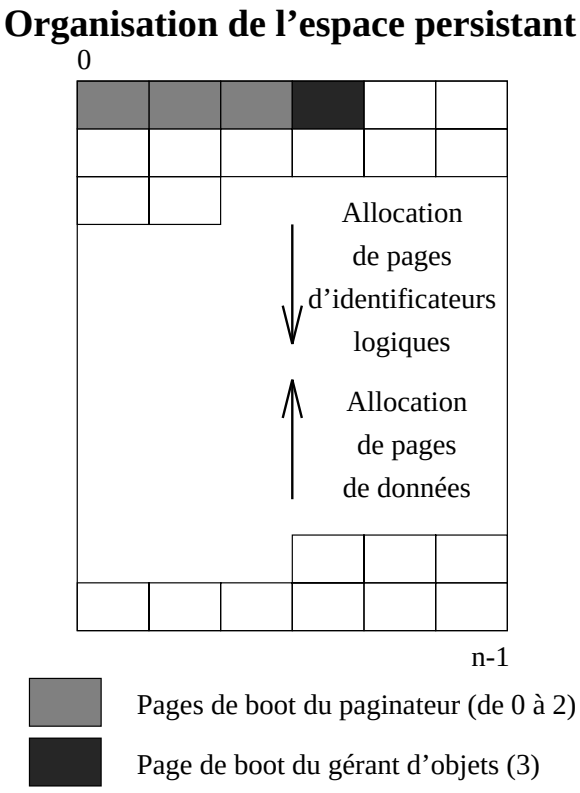


Fig. 6.3 : Organisation d'un disque géré par le paginateur

Le paginateur gère des disques qui sont des espaces persistants de N pages de taille fixe numérotées de 0 à N−1. Par exemple, pour une taille de page de 8 Ko, un disque contient N=65505 pages, la première étant utilisée pour stocker l'état d'allocation des pages de ce disque. C'est donc un espace d'adressage d'environ 510 méga−octets (Mo). De plus, le paginateur peut gérer 65536 disques (numéro de disque sur un mot de 16 bits) ce qui

représente alors un espace de stockage adressable d'environ 32 téra−octets (To), une adresse disque étant un triplet du type (n° disque, n° page, déplacement) codé sur 64 bits.

Pour représenter une table de déréférençage persistante, il nous faut gérer pour cela un espace de pages contiguës. Il y a alors deux solutions pour permettre d'allouer simultanément des pages d'identificateurs et des pages de segments. La première est d'allouer statiquement la place pour la table, sa taille étant alors fixe. Cette solution fixe le ratio pages d'identificateur/pages de segments, ce qui représente une grosse contrainte.

L'autre solution, que nous avons utilisée, est d'autoriser l'allocation suivant deux directions (cf. Fig. 6.3) : des adresses hautes vers les adresses basses et inversement. Le ratio pages d'identificateurs/pages de segments est alors variable suivant le type d'objets utilisé (gros objets ou petits objets). Cette représentation sur disque de la table de déréférençage permet ensuite, par un calcul simple, de déterminer l'adresse dans un disque ([n° page, déplacement]), des informations représentées par un identificateur, à partir de son indice.

Gestion en mémoire virtuelle

Nous avons étudié la manière dont sont gérés les identificateurs logiques persistants en espace secondaire. Nous devons maintenant offrir un mécanisme permettant de manipuler les objets qu'ils désignent en mémoire virtuelle. Il faut pour cela une table permettant de savoir si tel objet a été chargé en mémoire virtuelle ou non. Cette table est une image en mémoire virtuelle de la table sur disque. Elle contient non plus des relais d'adressage sur disque mais des relais en mémoire virtuelle. Nous avons choisi pour cela le mécanisme décrit dans la figure Fig. 6.4.

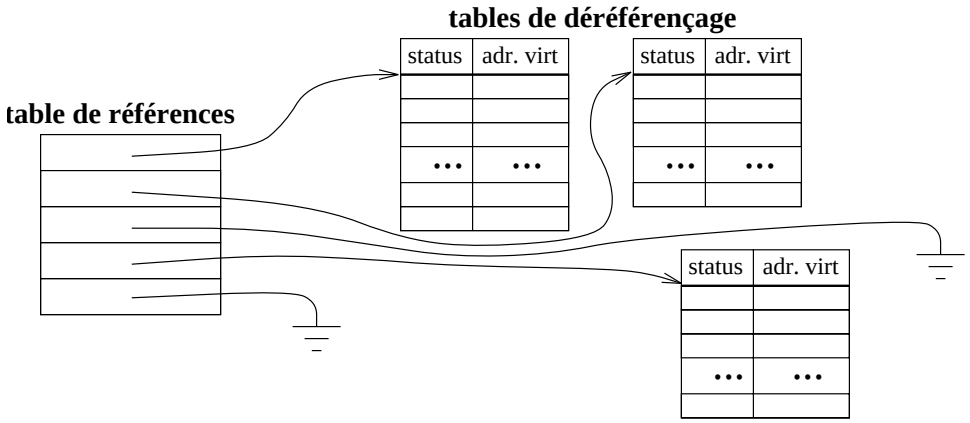


Fig. 6.4 : table de déréférençage en mémoire virtuelle

Nous observons que pour éviter de saturer l'espace virtuel, nous n'avons pas voulu allouer en mémoire, une table correspondant à l'image de la table persistante dans son entier. Cela aurait été très pénalisant, surtout lorsque seul un petit nombre d'objets de la base sont manipulés dans une session. Nous avons donc introduit un niveau d'indirection, qui nous permet de partitionner l'image en mémoire virtuelle de la table persistante, en un ensemble de

sous−tables. Une telle sous−table n'est créée que lorsqu'un identificateur de l'intervalle qu'elle représente est déréférencé.

Nous utilisons une technique similaire pour les objets non persistants ce qui permet d'avoir un mécanisme de désignation homogène des objets persistants et temporaires en mémoire virtuelle. La gestion des identificateurs temporaires et la gestion des identificateurs persistants sont néanmoins différentes dans la mesure où les identificateurs persistants peuvent être réalloués (gérés par un ramasse miettes) alors que les autres sont utilisés pour la désignation d'un unique objet temporaire au cours d'une session.

VI.2.2 Gestion de segments de données

Les identificateurs logiques décrits précédemment permettent d'implanter le concept d'identificateur d'objet du langage PEPLOM. Il faut ensuite pouvoir stocker la valeur associée à un objet. C'est le rôle des segments de données décrits dans cette partie.

Nous avons dit qu'il est nécessaire de pouvoir gérer des unités d'allocation de mémoire persistante inférieure à la taille d'une page. Une solution est alors de gérer une page comme un tas dans lequel nous allouons des morceaux de mémoire de taille quelconque au fur et à mesure des besoins. Cette technique a l'avantage de pouvoir optimiser le taux d'occupation d'une page lors des premières allocations. En revanche, le processus de désallocation/réallocation est complexe (trouver la page contenant un segment libre où il est possible d'allouer le segment demandé) et conduit a un effritement de la mémoire. De plus, ce mécanisme est lourd à mettre en œuvre du point de vue des informations nécessaires par exemple pour retrouver les segments libres (chainage).

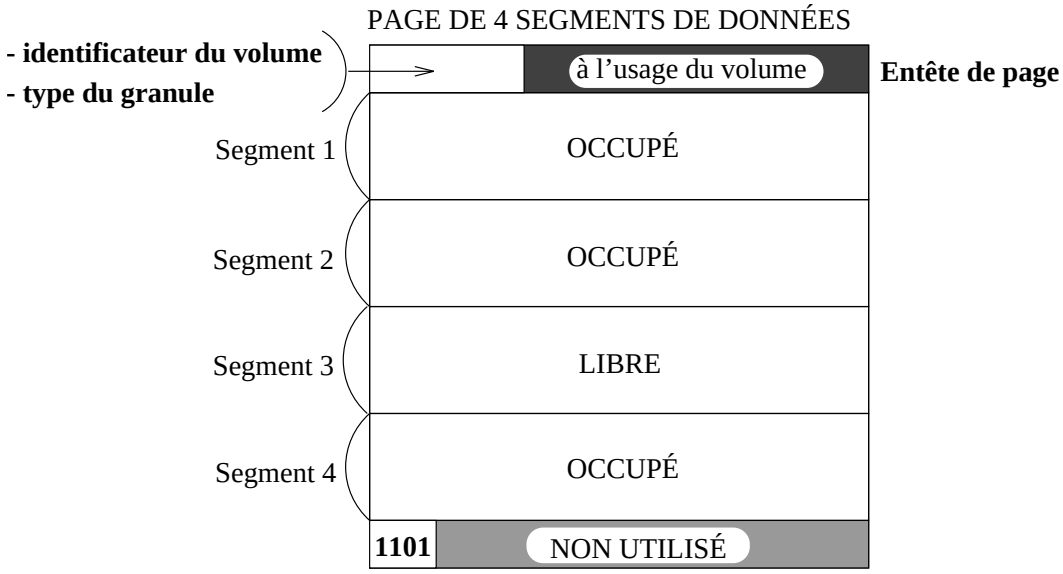


Fig. 6.5 : Format d'une page de granules

Notre choix s'est porté sur une autre solution qui consiste à fixer la taille des segments alloués dans une page. L'avantage est que le mécanisme d'allocation/désallocation est

simple et léger à mettre en œuvre et que le problème d'effritement est supprimé. Par contre, le taux d'occupation peut être plus faible qu'avec la solution pécédente.

Pour améliorer ce taux, nous définissons de manière statique quelles sont les tailles de segments qu'il est possible d'allouer dans une page de façon à minimiser la perte d'espace. Nous appelons ces segments fixes des granules. Une page de données va donc contenir un nombre fixe NG de granules de taille TG (cf. Fig. 6.5).

Nous voyons qu'une page de granules est composée de trois parties :

1. une entête qui contient l'identificateur du volume auquel elle appartient (cf. VI.2.3), le type de granule qui y est implanté, et d'autres informations utilisées pour la gestion de cette page dans son volume (chainage).
2. la zone des segments de données qui sont stockés de façon contiguë dans la page.
3. la table d'occupation de la page. C'est une chaine de bits où chaque bit signale si le granule de rang correspondant dans la page est occupée au non par un segment.

Il peut ensuite y avoir un certain nombre d'octets non utilisés qui sont alors perdus. Nous définissons statiquement les tailles de granules qui vont justement minimiser cet espace perdu. Les tailles choisies pour les granules sont de plus multiple de quatre octets. La figure Fig. 6.6 montre le nombre d'octets perdus pour les tailles de granules multiples de quatre et qui tiennent dans une page de 512 octets. La taille de l'entête est de 24 octets dans cet exemple.

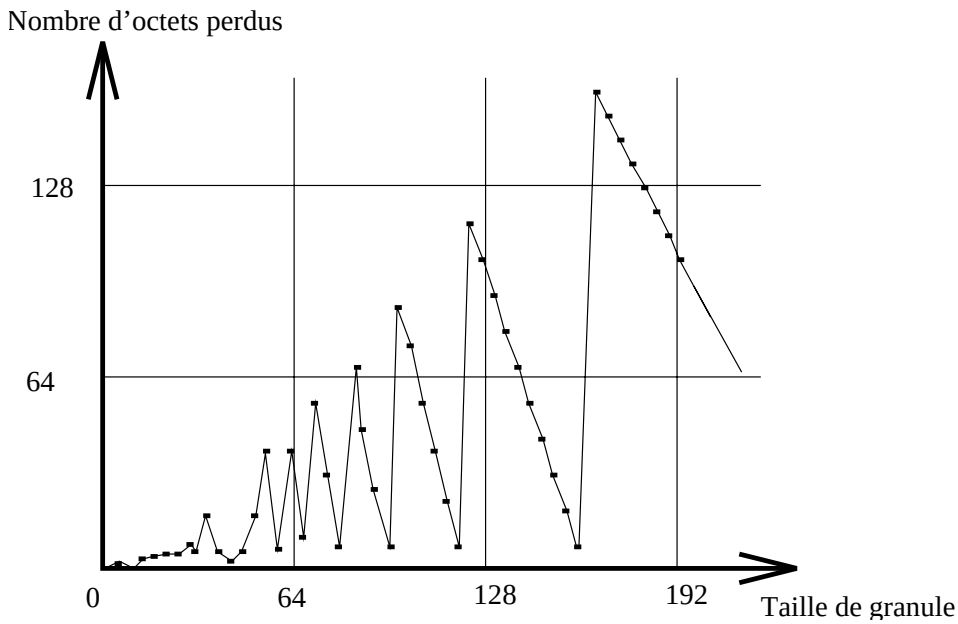


Fig. 6.6 : courbe du taux d'occupation d'une page suivant la taille du granule

Les tailles de granules retenues sont celles représentant une perte minimum aux alentours d'une taille donnée (minima observés sur la courbe ci−dessus).

Dans ce contexte, l'allocation d'un segment de données correspond à la recherche d'une page contenant des granules libres de la taille directement supérieure ou égale à la taille

requise. Il peut donc y avoir à nouveau une perte d'espace puisque le segment peut être plus petit que son granule. Cela peut devenir dans certains cas un avantage car si le segment doit grossir et qu'il peut le faire dans l'espace qu'il lui reste dans le granule, il n'aura pas besoin de changer de page. C'est en effet ce qu'un segment peut être amené à faire s'il ne tient plus dans son granule lorsqu'il grossit.

VI.2.3 Gestion de volumes

Les volumes, dans le contexte de notre support d'objets, permettent d'effectuer des regroupements de segments. C'est un moyen efficace pour implanter des collections d'objets. Ils sont d'ailleurs utilisés dans ce sens pour matérialiser les extensions de types abstraits pour PEPLOM. Un segment est donc toujours alloué dans un volume. C'est à l'utilisateur du volume de dire s'il ne met que des segments d'une même catégorie (représentant par exemple des objets d'un même type) dans le volume ou des segments hétérogènes.

Ce choix peut être important puisque le gestionnaire de volumes permet de parcourir tous les granules occupés qui ont été alloués dans celui−ci. Cette fonctionnalité est implantée à l'aide de descripteurs de parcours, d'une manière un peu similaire à la gestion des fichiers Unix.

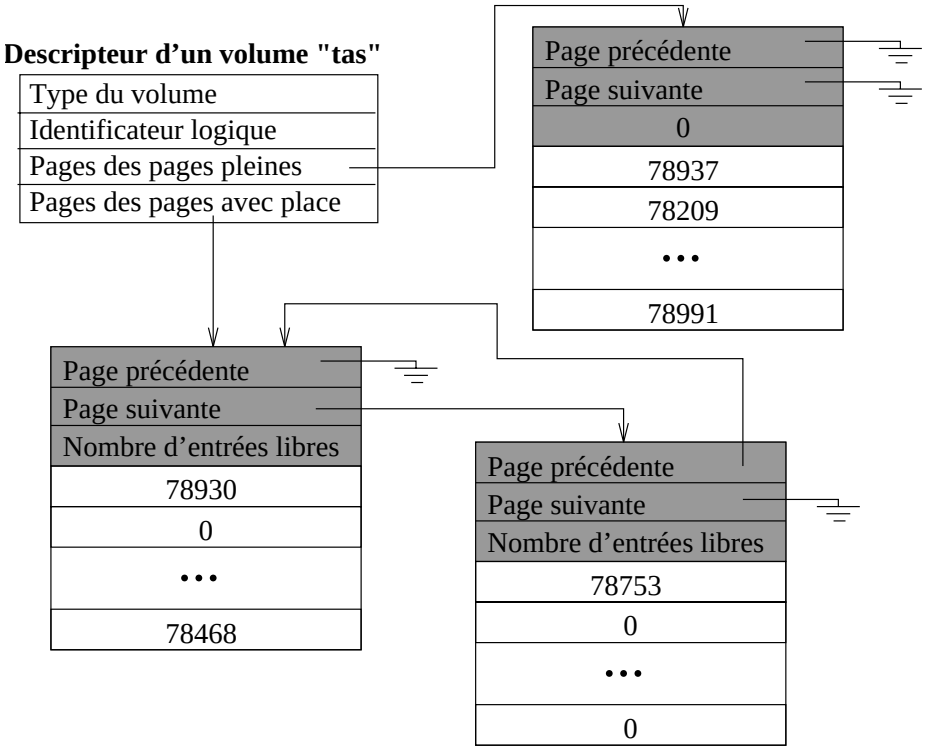


Fig. 6.7 : Organisation d'un volume "tas"

Nous avons pour l'instant deux types d'organisation des volumes. Pour chaque type, il y a des méthodes de parcours qui peuvent être différentes pour pouvoir justement utiliser au mieux cette structure d'accès. Les deux organisations sont les volumes "tas" et les volumes

hachés. Par ailleurs, nous utilisons le mécanisme d'identificateur logique pour désigner ces volumes.

VI.2.3.1 Les volumes "tas"

L'organisation des volumes sous la forme de tas est la plus simple qui est offerte. Un tas est ici un ensemble de pages contenant des segments de tailles différentes. Les segments contenus dans ces pages n'ont aucune organisation entre eux.

La figure Fig. 6.7 montre comment est organisé un volume "tas" sur le disque. Il est composé tout d'abord d'un descripteur, alloué dans le volume racine (cf. VI.2.3.3), qui contient quatre informations : le type du volume ("tas" ou haché), son identificateur logique, deux pointeurs (adresses de pages disques) sur des listes de pages.

Les pages de ces deux listes contiennent des adresses de pages disques. Les pages ainsi désignées sont celles qui renferment les segments de données alloués dans le volume. Nous aurions pu choisir simplement de chainer les pages de données pour représenter le volume. Cette approche limite les possibilités d'accès concurrents lorsqu'il y a des modifications.

Les pages désignées par les pages de la première liste (pages pleines) sont des pages où tous les granules ont été alloués. Celles désignées par les pages de l'autre liste ont au moins encore un granule libre. En revanche, aucune de ces pages n'a aucun granule occupé. Cette décomposition en deux listes permet d'accélérer la recherche de granules libres. Le volume "tas" permet donc un regroupement physique aléatoire de segments dans un certain nombre de pages. Nous allons voir que le second type de volume permet un regroupement plus précis.

VI.2.3.2 Les volumes hachés

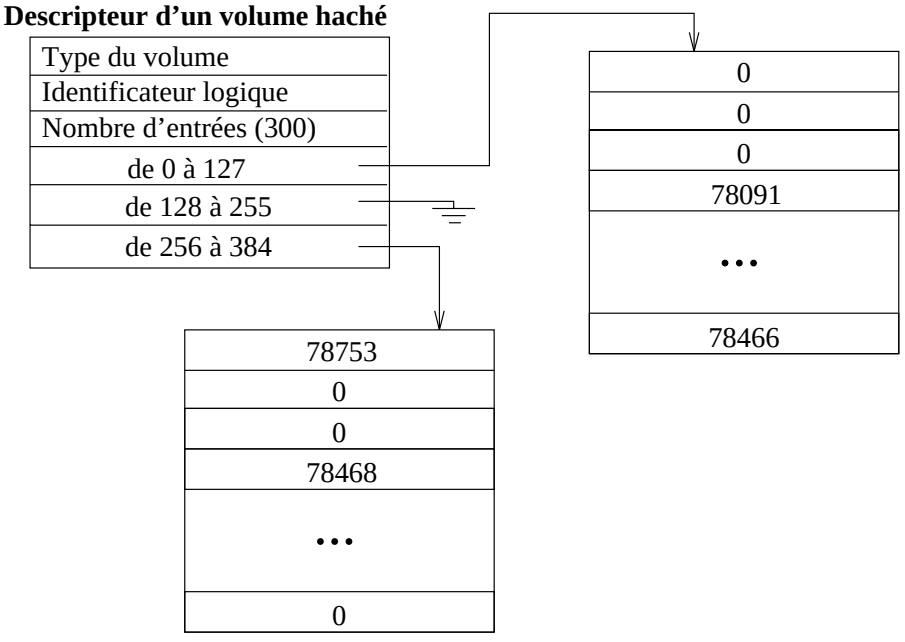


Fig. 6.8 : Organisation d'un volume haché

Le but des volumes hachés n'est pas seulement de regrouper un ensemble de pages de segments ; il est surtout de regrouper en plus certains segments partageant une caractéristique commune. Cette caractéristique commune est déterminée par le résultat d'une fonction de hachage qui s'applique aux données des différents segments alloués dans le volume. Un volume haché peut logiquement se représenter à l'aide d'une table de hachage où chaque entrée désigne une liste de pages de segments pour lesquels la fonction de hachage détermine la même entrée. A la création d'un volume haché, il est donc nécessaire de donner le nombre d'entrées de la table de hachage.

C'est ce que nous observons dans la figure Fig. 6.8 qui décrit l'organisation sur le disque d'un volume haché. Un descripteur de volume est dans ce cas composé de quatre parties : le type du volume (ici haché), son identificateur, le nombre d'entrées de la table de hachage et les adresses disques des pages représentant la table de hachage. Ces dernières ne sont allouées que lorsqu'une entrée de l'intervalle qu'elles représentent contient l'adresse disque de la tête d'une liste de pages contenant les segments de données. Dans notre exemple, nous considérons que la taille d'une page est de 512 octets, ce qui fait que pour une table de 300 entrées, nous avons besoin de trois pages (pouvant contenir chacune 128 adresses disques) pour représenter la table.

VI.2.3.3 Le volume racine

Une base de données PEPLOM peut être considérée comme un ensemble de volumes. Le volume est ainsi l'unité de description de données de plus haut niveau. Nous avons un volume spécial, le volume racine, qui contient tous les descripteurs des volumes composant la base. De cette manière, il est possible de parcourir tous les volumes.

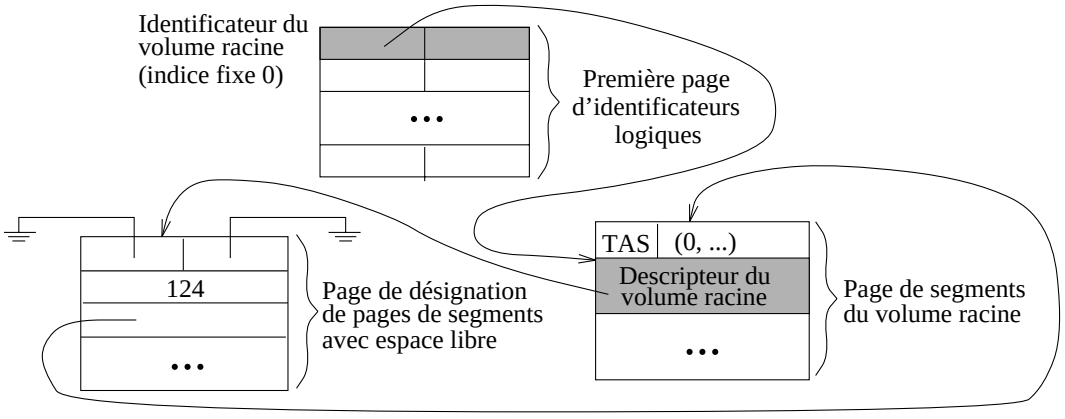


Fig. 6.9 : Organisation du volume racine

Ce volume racine, qui est de type "tas", peut donc être vu comme la racine de la base, le nom racine étant en fait l'identificateur logique de ce volume. Cet identificateur est fixé par convention comme le premier identificateur alloué dans l'espace disque, c'est à dire l'identificateur d'indice 0. Le volume racine est un méta−volume qui décrit tous les volumes de la base et qui se décrit donc lui−même. Ce mécanisme est intéressant, car il offre par

exemple la possibilité par la suite de faire des statistiques sur l'état d'allocation de l'espace disque pour la base.

La création de ce volume racine est particulière par rapport aux autres volumes puisque nous devons résoudre un problème de "boot−strap". En effet, lorsque nous le créons, nous devons le répertorier comme un nouveau volume dans le volume racine alors qu'il n'existe pas encore. Le problème est résolu par une procédure spéciale qui met en œuvre le schéma de liens décrit dans la figure Fig. 6.9. Cette procédure crée une page de segments qui va contenir le descripteur du volume racine. Elle alloue ensuite une page de désignation de pages de segments ayant de l'espace libre (une seule entrée est consommée) qui va référencer la page de segments précédemment allouée. Dans le même temps, l'entrée de la table disque des identificateurs logiques persistants correspondant au premier identificateur (indice 0) va contenir l'adresse disque du segment du descripteur du volume. Le mécanisme de gestion des volumes est alors prêt à être utilisé.

VI.2.3.4 Le parcours de volume

Une fonctionnalité essentielle que nous voulons offrir avec les volumes est la possibilité de parcourir tous les segments qui ont été alloués dans ceux−ci. Le mécanisme proposé pour cela est semblable à celui offert pour la lecture de fichiers Unix. Il y a une initialisation du parcours qui retourne un descripteur. Ensuite, nous effectuons une demande d'un segment du volume, chaque segment n'étant parcouru qu'une seule fois. Si tous les segments ont été parcourus, une fin de parcours est signalée et le descripteur peut alors être fermé. Cette technique à l'intérêt de permettre des parcours imbriqués.

Le problème est donc résolu de manière triviale tant que le volume n'est pas modifié pendant un parcours. Dans ce dernier cas, la sémantique du parcours devient beaucoup plus problématique. En effet, la technique de parcours décrite précédemment impose un ordre de parcours des segments.

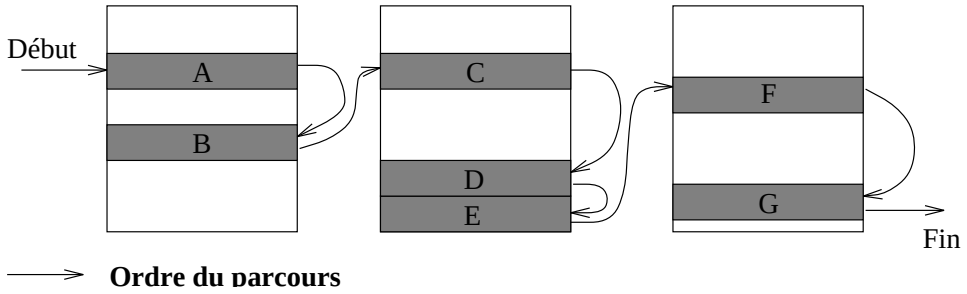


Fig. 6.10 : Parcours des segments d'un volume page par page

Si nous considérons qu'un volume peut être modifié pendant un parcours, cela signifie que nous pouvons soit désallouer un segment soit en créer un nouveau. Supposons que nous en créons un nouveau, nommé X, et qu'il soit créé entre les segments C et D de la figure Fig. 6.10. Cette figure montre que l'ordre de parcours initial est "ABCDEFG". Après l'ajout de X, l'ordre devient "ABCXDEFG". Ainsi, lors d'un parcours, suivant le moment où X est créé, il va être soit parcouru (descripteur de parcours sur A, B ou C) soit ignoré (descripteur de

parcours sur D, E, F ou G). Le même phénomème peut se produire dans le cas d'une désallocation de segments.

Il n'est pas acceptable pour les utilisateurs des volumes d'avoir un comportement aléatoire de la création ou de la destruction de segments lors d'un parcours. Le choix que nous avons fait est de différer la prise en compte de tous les ordres de création ou de destruction de segments jusqu'à la fin du parcours du volume concerné (fermeture du descripteur de parcours).

Le même type de problème se pose aussi lors d'une réallocation d'un segment dans un volume. Par contre, il ne s'agit pas ici de savoir si nous allons ou non parcourir ce segment (il faut le parcourir), mais de parcourir la bonne version du segment. Cette fois, nous enregistrons les déplacements de segments lorsqu'ils sont réalloués. Ainsi, lorsque nous accédons à un segment qui a été relogé dans un autre granule du volume, nous retrouvons sa nouvelle adresse et accédons à ce nouveau segment. Signalons que ce nouveau segment n'est pas parcouru par ailleurs, sa création n'étant validée qu'à la fin du parcours.

La sémantique définie pour le parcours des volumes est d'ailleurs équivalente à celle de fonctionnalités PEPLOM. C'est le cas pour la boucle "**`foreach`** (..." du langage qui peut être implantée par un parcours de volume. Elle a alors la même sémantique que pour la gestion des volumes c'est à dire que si l'ensemble parcouru est modifié à l'intérieur de la boucle, les modifications ne sont validées que lorsqu'on en sort.

VI.3 Code PEPLOM généré

Dans la section précédente, nous avons décrit la couche de support d'objets sur laquelle s'appuie le code généré par le compilateur PEPLOM. C'est donc à la partie génération que nous allons nous intéresser dans cette partie. Nous aurons ainsi passé en revue la presque totalité des couches logicielles mise en œuvre pour le langage PEPLOM. Seul l'interprète des constructeurs dynamiques n'est pas traité dans ce chapitre ni même dans la thèse.

Une grosse partie du code PEPLOM est très proche du code C. Il n'est donc pas intéressant de décrire cette partie de la génération qui se résume pratiquement à une simple réécriture de code C. En fait, l'aspect le plus important ici est le code généré pour le support d'objets.

Nous n'allons pas détailler la mise en œuvre de l'aspect structurel du modèle de données de PEPLOM. Tous les types PEPLOM sont en fait traduits sous la forme de types C, même pour les types abstraits. Nous aurions pu, pour ces derniers, les générer sous la forme de types C++ comme le font tous les C++ persistants, récupérant ainsi la mise en œuvre de l'héritage. Nous n'avons pas fait ce choix car il est coûteux à la fois en temps d'exécution (mise à jour des objets à chaque chargement) et en place mémoire (mise en œuvre par C++ de l'héritage). L'adressage de la structure des objets PEPLOM est essentiellement statique, chaque objet étant implanté dans un segment ou plusieurs dans le cas des constructeurs dynamiques.

Nous nous attachons dans la suite à décrire trois points fondamentaux implantés dans le prototype de l'environnement d'exécution de PEPLOM. Le premier point concerne la désignation des objets dans le langage, le déréférençage des objets persistants et l'appel de

méthode sur un objet. Le second point est lié au choix de ne pas utiliser le support d'objets de C++ pour les objets PEPLOM. La gestion de la résolution tardive de code (implantation du polymorphisme *ad hoc*) est mise en œuvre par un mécanisme spécifique à PEPLOM qui tient compte du fait que les objets peuvent être persistants. Enfin, le troisième point couvre l'implantation des modules PEPLOM et notamment des variables persistantes.

VI.3.1 Poignée d'objet et appel de méthode

Nous avons vu précédemment (cf. Fig. 6.4) que le déréférençage d'un objet se fait à travers une table représentée en fait par deux niveaux de table. Ce déréférençage est effectué à partir de l'indice de l'identificateur logique de l'objet. Le premier niveau de table donne l'adresse d'une autre table qui contient les références des objets chargés dont l'indice appartient à un intervalle donné.

Un déréférençage d'objet se fait donc en deux étapes. La première consiste à retrouver l'entrée qui doit contenir la référence de l'objet en mémoire. La seconde s'occupe de charger l'objet en mémoire s'il ne l'a pas encore été et à associer sa référence en mémoire virtuelle à l'entrée déterminée dans la première étape.

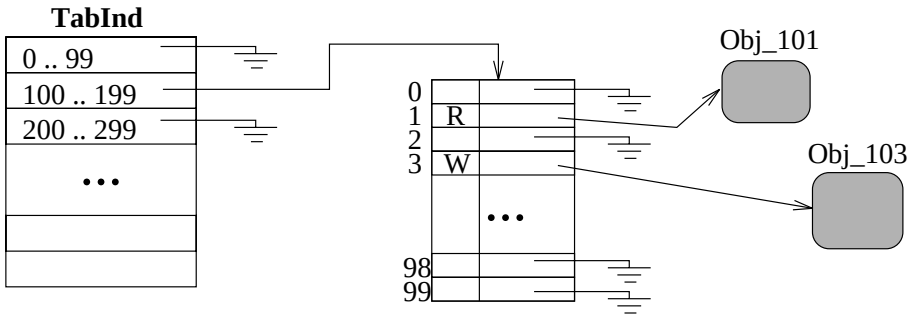


Fig. 6.11 : Mécanisme de déréférençage d'objets persistants

L'algorithme de la première phase est simple. Il s'agit de calculer dans quel intervalle se trouve la référence et de trouver la bonne entrée. L'algorithme de la fonction `Entry(i)→E` qui calcule l'adresse `E` de cette entrée d'indice `i` est le suivant (**`adr`** et **`val`** sont deux fonctions qui rendent respectivement l'adresse de l'élément désigné par une variable et la valeur désignée par l'adresse contenue dans une variable, équivalent aux opérateur & et * de C) :

```
si (TabInd[i div 100] == nil)
    TabInd[i div 100] = <allouer table deref.>
finsi
E = adr (val (TabInd[i div 100]) [i mod 100])
```

Nous utilisons dans notre exemple une taille d'intervalle de 100, cette taille étant la même pour toutes les tables de déréférençage (tables de deuxième niveau). Malgré la simplicité de l'algorithme du calcul de l'entrée de déréférençage, cela coûte cher si cette fonction est exécutée lors de chaque manipulation d'un objet. Supposons que nous ayons le code PEPLOM suivant :

```
{
personne  p;
...
if (p.Nom == "toto")
    x += p.Age();
...
}
```

Dans cet exemple, si le test est vrai, la variable p est utilisée deux fois de suite pour manipuler le même objet. Dans ce cas, il est inutile de dérouler deux fois le code pour déterminer l'entrée de l'objet manipulé. La solution que nous proposons est d'associer à chaque variable désignant un objet, une poignée contenant l'adresse de l'entrée de déréférençage. Malheureusement, cette technique ne peut être mise en œuvre que sur les variables locales à un bloc de code ou pour les variables globales définies dans les modules PEPLOM. Elle ne peut pas l'être sur les champs d'un objet par exemple. Voyons comment cette solution est mise en œuvre dans le code C généré (pour le code PEPLOM ci−dessus) :

```
{
T_LogId  lid_p,  **hdl_p  =  0;
...
if (!strcmp (
        ((hdl_p                                  ⎫
           ?  hdl_p                              ⎬ 1
           : (hdl_p=Entry(lid_p.Indice))         ⎭
         ),
         ((*hdl_p)                               ⎫
           ?  (*hdl_p)                           ⎬ 2
           : ((*hdl_p)=GetObject(lid_p,READ))    ⎭
         ),
         <résolution  accès  au  champ  "Nom">
        ),
        "toto")
   )
    {
    int  res_int;

    (x  +=  (<résolution  code  de  "Age">
                (lid_p,
                 (hdl_p                          ⎫
                    ?  hdl_p                     ⎬ 1'
                    : (hdl_p=Entry(lid_p.Indice))⎭
                 ),
                 &res_int
                ),
            res_int)
    )
    }
...
}
```

La variable `p` est donc traduite sous la forme de deux variables `lid_p` et `hdl_p`. La première est du type `T_LogId` qui correspond à la définition d'un identificateur logique (structure de trois champs : (indice de l'objet, numéro de disque, status)). Ce type est défini par la machine de support d'objets. Cet identificateur est aussi stocké au début de la structure d'un objet PEPLOM. C'est pour cette raison que nous définissons la poignée comme un pointeur vers une entrée contenant un pointeur vers l'objet en mémoire virtuelle (donc vers son identificateur).

A chaque fois que la variable p est utilisée pour manipuler l'objet qu'elle désigne, nous mettons à jours la poignée, si cela n'a pas déjà été fait. Nous observons dans l'exemple les deux phases **`1`** et **`2`** du déréférençage d'un objet. La première met à jour la poignée et la deuxième la référence de l'objet en mémoire. Lorsque ces deux phases ont déjà été exécutées une fois, les déréférençages suivants ne coutent que deux tests ce qui paraît acceptable. Par exemple lors de l'appel de la méthode `Age`, le code **`1`** ayant été exécuté auparavant, le code équivalent **`1'`** n'effectue qu'un seul test car la poignée est déjà déterminée.

Dans l'exemple ci−dessus, nous montrons la traduction de deux expressions différentes utilisant un objet. La première est un accès à un champ de la valeur d'un objet et la seconde, l'évaluation d'une méthode (fonction) sur cet objet.

Pour l'accès au champ, il s'agit d'une seule expression qui contient les actions de déréférençage. Nous utilisons pour cela l'opération d'évaluation séquentielle de C où l'expression E=(e_1, e_2, ..., e_n) correspond à l'évaluation de e_1 puis e_2 jusqu'à e_n. La valeur de l'expression E est alors celle de la dernière expression évaluée de la séquence, à savoir e_n. Dans le cas de l'accès au champ `Nom`, la première expression détermine la poignée, la deuxième calcule l'adresse de l'objet en mémoire et la troisième, qui n'est pas détaillée ici, résout l'adresse du champ.

Pour l'appel de méthode, il s'agit d'un simple appel de procédure C, où les paramètres qui correspondent à des variables de manipulation d'objets engendrent aussi deux paramètres comme pour les variables locales à un bloc. C'est notamment le cas pour le premier paramètre d'une méthode qui désigne l'objet auquel elle s'applique. Le premier paramètre contient l'identificateur logique de l'objet invoqué et le deuxième sa poignée. Nous pouvons remarquer que dans le cas du passage de paramètre, seule la poignée est déterminée, le déréférençage de l'objet étant réalisé dans la méthode.

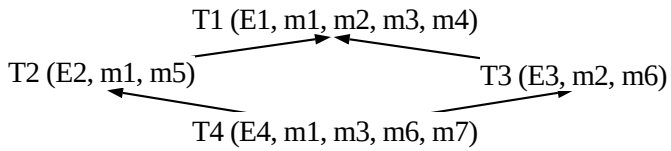
La technique de poignée est mise en œuvre exclusivement pour les noms auxquels le compilateur peut associer statiquement une adresse (ce n'est pas possible dans le cas d'un champ d'un objet chargé dynamiquement). Elle pourrait être étendue à tous les cas de noms désignant des objets ce qui reviendrait alors à définir des segments de liaison associés à chaque objet chargé dans l'espace de travail. Cette technique est d'ailleurs mise en œuvre par le noyau Eliott[49][65] du système Guide. Cette technique est néanmoins difficile à mettre en œuvre dans notre contexte. En effet, un objet PEPLOM pouvant contenir un ensemble (dynamique) de références à d'autres objets, le segment de liaison associé à un tel objet aurait alors une taille qui peut varier en cours d'exécution.

Nous allons voir dans la partie qui suit comment sont calculées à la fois l'adresse d'un champ d'un objet et la fonction C qui doit être appelée lors d'une invocation de méthode, ces deux actions ne pouvant être effectuées qu'à l'exécution puisque le type réel de l'objet manipulé n'est connu qu'à cet instant.

VI.3.2 Résolution tardive d'adressage et de code

La relation d'héritage multiple qui peut être définie entre des types abstraits PEPLOM et la possibilité de surcharger le code d'une méthode dans un sous−type engendrent deux phénomènes. L'adressage de l'état d'un objet et la sélection du code d'une méthode ne peuvent être effectués que dynamiquement. Il est néanmoins possible, comme le fait C++, de définir statiquement cette résolution dynamique.

Nous allons, dans un premier temps, étudier le mécanisme mis en œuvre dans C++[42] dans le cas d'un héritage "virtuel" (équivalent à l'héritage introduit dans PEPLOM), puis montrer le mécanisme mis en œuvre pour PEPLOM. Nous allons voir les avantages qu'il procure par rapport à l'approche de C++. Pour ce faire, nous allons utiliser la hiérarchie de types définie ci−dessous :



L'exemple définit quatre types T1, T2, T3 et T4, les flêches indiquant les relations d'héritage entre ceux−ci. Chaque type T est défini par un n−uplet (E, m_1, ..., m_n) où E est l'état défini par T et les m_i les méthodes qu'il définit. Ainsi, la méthode m_1 est définie dans le type T1 et redéfinie dans les types T2 et T4 (nous avons donc trois implémentations différentes de m_1).

VI.3.2.1 Résolution dans C++

La figure Fig. 6.12 présente deux objets C++ générés par la classe T2 pour le premier et par la classe T4 pour le second. Ces deux objets peuvent être désignés dans un programme C++ par la même variable V de type T2.

Dans l'hypothèse où nous accédons l'état E1 à travers V, sa position ne peut être déterminée statiquement comme le montre la figure Fig. 6.12, la partie ***P1*** (partie définie par la classe T1) de l'objet se trouvant une fois après la partie ***P2*** et l'autre fois après la partie ***P4*** qui n'ont pas nécessairement la même taille.

En fait, un pointeur sur un objet C++ ne pointe pas obligatoirement à son début. La valeur du pointeur sur un objet dépend en fait du type de ce pointeur. Dans le cas de la variable V, le pointeur est de type T2. Il pointe donc au début de l'objet dans le cas de l'objet **Obj_T2** alors que pour **Obj_T4**, il pointe au début de la partie ***P2***. La solution mise en œuvre par C++ pour retrouver l'état E1 est d'avoir un pointeur dans la partie ***P2*** vers la partie ***P1***. En fait chaque partie va contenir des pointeurs vers les parties de l'objet appartenant à toutes les super−classes directes de la classe à laquelle appartient la partie concernée. L'ordre de ces

pointeurs est déterminé statiquement et reste le même quelque soit la classe de l'objet dans lequel se trouve cette partie. Ils servent principalement pour le mécanisme de cœrcition du type des pointeurs et pour l'accès aux variables d'état d'un objet.

Une partie P, correspondant à une des classes auxquelles appartient l'objet englobant, est donc composée d'une entête de taille constante déterminée statiquement et de l'état E composé des variables définies dans la classe T qui définit P. L'entête contient non seulement les pointeurs (grisé clair) désignant toutes les parties définies par les super−classes directes de T mais aussi un pointeur (grisé foncé) vers une table appelée "table virtuelle".

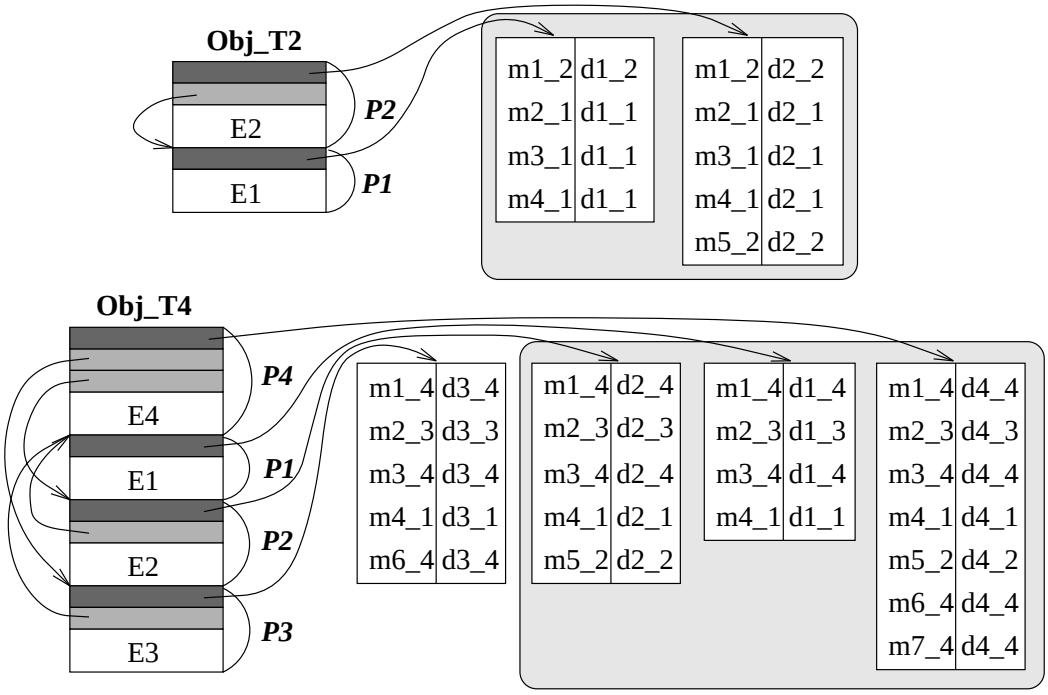


Fig. 6.12 : Gestion de l'héritage multiple dans C++

Cette table contient des pointeurs vers toutes les méthodes applicables à partir de la classe T (méthodes définies dans T et celles définies dans ses super−classes) et des déplacements permettant de calculer les pointeurs vers la partie sur laquelle elles s'appliquent. Ces méthodes sont différentes suivant la classe de l'objet auquel appartient la partie P engendrée par T. Nous pouvons observer cela dans la figure Fig. 6.12 où l'invocation "V.m3" invoque, lorsque V désigne **Obj_T2**, la méthode "m3" implémentée par T1 (nommée "m3_1") et, lorsque V désigne **Obj_T4**, la méthode "m3" implémentée par T4 ("m3_4").

Enfin, le compilateur C++ met en œuvre une optimisation au niveau de la définition des tables virtuelles. Nous pouvons remarquer que dans le cas d'un héritage simple, toutes les tables peuvent être factorisées dans la même. Celle−ci contient alors toutes les entrées possibles pour la classe à laquelle est associée la table. La figure Fig. 6.12 indique qu'il est possible de factoriser pour le type T2 (voir **Obj_T2**, cadre grisé très clair) les deux tables correspondant aux méthodes appelables à partir d'une variable de type T1 et T2. Cette

optimisation peut aussi être mise en œuvre dans le cas d'un héritage multiple sur une branche de l'arborescence (retour au cas d'héritage simple). Dans notre exemple, cela peut être fait pour la branche T1←T3←T4 (voir **Obj_T4**, cadre grisé très clair) où les tables correspondantes peuvent être factorisées.

Nous venons de décrire toutes les informations mises en œuvre dans les objets C++ et à l'extérieur pour gérer la sémantique de l'héritage virtuel. Observons maintenant comment elles sont utilisées pour calculer l'accès aux différentes parties de l'état d'un objet ou pour l'appel d'une méthode polymorphe (nous supposons que les objets **Obj_T2** et **Obj_T4** sont désignés à travers un pointeur de type T2 contenu dans une variable V) :

1. **Accès à l'état**

   Le coût de l'accès à l'état d'un objet varie suivant le type du pointeur sur l'objet et la partie de l'état à accéder. S'il s'agit de la partie correspondant au type du pointeur, l'accès ne coûte alors qu'une indirection. C'est le cas pour l'accès à E2 qui se traduit par "V−>E2" (pour **Obj_T2** comme pour **Obj_T4**). Par contre, si nous devons accéder à E1, cela se traduit par "V−>ptr_P1−>E1" (où ptr_P1 est le pointeur dans ***P2*** sur la partie ***P1***), c'est à dire une double indirection. En fait, il y a autant d'indirection (plus une) que le nombre de niveaux dans la hiérarchie séparant la partie pointée de celle de l'état à accéder (par exemple "P4−>ptr_P2−>ptr_P1−>E1" pour l'accès depuis P4 à l'état E1). L'accès à une variable d'état coûte donc n unités (pour n−1 niveaux de hiérarchie à traverser pour atteindre l'état voulu, d'où n ≥ 1).

2. **Appel de méthode**

   Le coût de l'appel de méthode est constant quelque soit la classe dans laquelle est implémentée la méthode appelée. Prenons comme exemple l'appel en C++ "V−>m2(...)". Le code est évidemment le même que l'appel se fasse sur l'objet **Obj_T2** ou sur l'objet **Obj_T4** ; il est traduit par :

   (*(V->vtbl[ind_m2].m)) (V + V->vtbl[ind_m2].d, ...)

   - m2_1 pour Obj_T2
   - m2_3 pour Obj_T4
   - d2_1 pour Obj_T2
   - d2_3 pour Obj_T4

   La première expression récupère l'implémentation pertinente de la méthode "m2" dans la table virtuelle associée à l'objet. L'indice dans la table est calculé statiquement et il est le même pour une méthode de nom donné dans toutes les classes où elle est visible dans un arbre d'héritage. La deuxième expression calcule le pointeur sur la partie sur laquelle travaille toujours la méthode appelée. C'est fait en récupérant le déplacement par rapport au pointeur d'appel, ce déplacement étant stocké dans la table virtuelle à l'indice correspondant à la méthode appelée. Nous évaluons le coût de l'évaluation de cette expression à 11 unités, comprenant des indirections, des sommes et des multiplications pour calculer l'adresse d'une entrée dans la table, et l'empilage du pointeur de l'objet appelé.

Le but principal de cette technique est la plus grande efficacité possible pour l'appel de méthode et l'accès à l'état d'un objet. De ce point de vue, il parait difficile de trouver une

meilleure solution par rapport aux contraintes de C++ telles que celle liée à la possibilité d'opérer des cœrcitions de pointeurs. Par contre, le coût de cette méthode est une perte d'espace important pour chaque objet créé, occasionnée par les différents pointeurs nécessaires. Cette perte est d'autant plus grande que le graphe d'héritage est large et profond. En rapport, la factorisation des tables virtuelles est une "économie de bouts de chandelles". De plus, tous ces pointeurs doivent être initialisés à la création de l'objet, cette perte de temps étant négligeable puisque l'objet n'est créé qu'une seule fois.

En fait, le plus grand défaut de cette méthode est lié à notre contexte persistant. En effet, d'une exécution à l'autre, un objet persistant de ce type doit être chargé à la même adresse virtuelle de même que le code de sa classe (contenant les tables virtuelles). La seule manière d'arriver à un tel résultat est d'avoir une machine de stockage à un niveau ("one level store") avec les problèmes que cela pose. Même cette solution ne résoud pas le problème des pointeurs vers les tables virtuelles. Il est en effet impossible (à moins de contraindre le code d'une classe à ne pas changer) d'assurer que ces tables seront toujours implantées à la même adresse virtuelle (même si elles sont implantées dans une librairie partagée). Dans tous les cas, il est donc nécessaire de rafraichir certaines informations à chaque chargement d'un objet persistant dans l'espace virtuel, ce qui peut être coûteux et complexe à mettre en œuvre.

VI.3.2.2 Résolution dans PEPLOM

Nous voulons définir une autre technique de gestion de l'héritage pour PEPLOM dans le but d'avoir des objets dont la structure est indépendante de l'implantation en mémoire virtuelle, ce qui, comme nous l'avons vu précédemment, n'est pas le cas pour les objets C++. Cette solution nous a paru au moins aussi simple à mettre en œuvre que les techniques de rafraichissement d'objets C++ à leur chargement que nous avons d'ailleurs expérimentées.

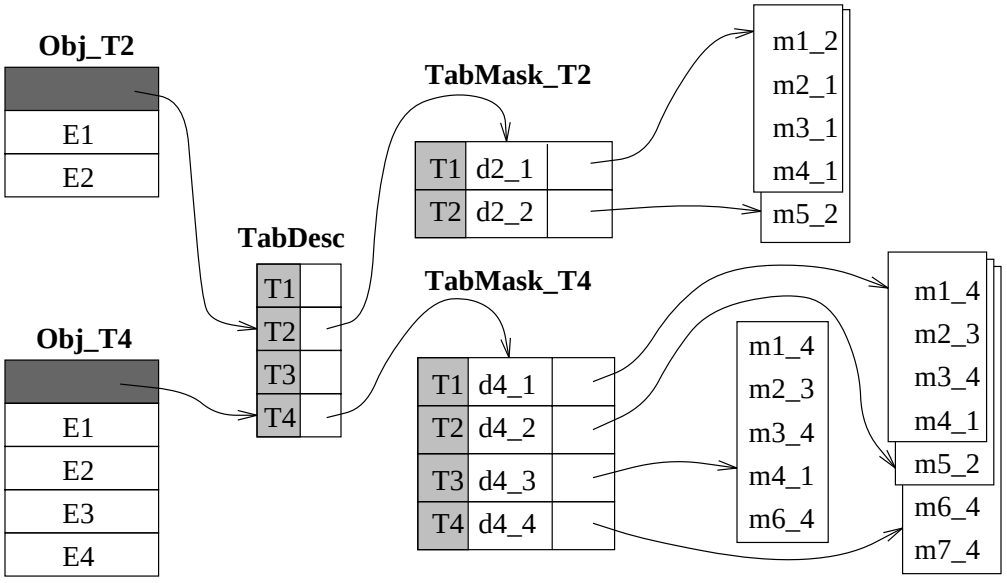


Fig. 6.13 : Gestion de l'héritage multiple dans PEPLOM

Nous reprenons dans la figure Fig. 6.13 le même exemple que celui de la figure Fig. 6.12 en appliquant cette fois la technique mise en œuvre par le compilateur PEPLOM. De la même manière, **Obj_T2** et **Obj_T4** sont des objets générés par les types abstraits PEPLOM T2 et T4. Nous les désignons à travers une variable V de type T2.

Le mécanisme proposé ici nécessite qu'un identificateur unique soit associé à chaque type abstrait présent dans une application (il est en fait unique à l'intérieur d'un schéma PEPLOM). Ces identificateurs servant d'indice dans des tables sont alloués à partir de zéro de façon incrémentale, et réalloués lorsque le type auquel ils sont associés est détruit. L'intérêt d'avoir cet intervalle d'identificateurs le plus dense possible est une économie d'espace perdue par les tables du deuxième niveau qui sont très creuses comme nous allons le voir.

Dans chaque entête d'objet PEPLOM, nous stockons l'identificateur de son type. C'est la seule information qui est utilisée pour la gestion de l'héritage (à la place de tous les pointeurs dans les objets C++). Cet identificateur correspond à une entrée dans une table que nous appelons table de description des types. Chaque type possède donc une entrée dans cette table.

Une entrée de cette table, correspondant par exemple à un type abstrait T, contient alors une référence vers une table d'un deuxième niveau. Cette nouvelle table correspond à un masque de manipulation du type concerné. Cela signifie qu'il va y avoir une entrée pour chaque type à travers lequel T peut être manipulé, c'est à dire une entrée pour chacun de ses super−types plus une pour lui−même. C'est donc le type T' de la variable à travers laquelle un objet est manipulé, ou bien le type T'' dans lequel est définie la partie accédée de l'état, qui détermine l'entrée dans cette table.

Une entrée de cette table contient deux informations. La première donne l'adresse relative (déplacement), par rapport à l'adresse de début de l'objet, de la partie de l'état définie par le type T'' déterminant l'entrée. La seconde est une référence vers une table, que nous pouvons comparer à une table virtuelle de C++, qui contient les méthodes applicables pour le type T (type réel de l'objet O manipulé) dans le cadre d'une utilisation à travers le type T' (type de la variable à travers laquelle est manipulé O). Nous pouvons remarquer que comme dans le mécanisme de C++, nous pouvons factoriser certaines tables.

Toutes ces informations sont donc utilisées pour calculer l'adressage de l'état d'un objet ou pour déterminer le bon code à exécuter lors de l'appel d'une méthode. Nous allons analyser le code généré par le compilateur PEPLOM pour ces deux types de manipulation (nous supposons que P est la poignée de l'objet déréférencé et que I est l'identificateur de l'objet manipulé) :

1. **Accès à l'état**

   Pour les objets PEPLOM, le coût d'accès à l'état est constant. Le code correspondant à l'accès à la partie E1 de l'état de l'objet **Obj_T2** comme de l'objet **Obj_T4** est le suivant :

   ```
   ((*P)  +  TabDesc[I.TypeId][T2].d)−>E1
   ```

   "(*P)" nous donne l'adresse de l'objet en mémoire virtuelle auquel nous ajoutons le déplacement pour avoir l'adresse de E1 extrait de la bonne table de masque. Cette

dernière est obtenue à partir "I.TypeId" (tout identificateur d'objet contient le type de celui−ci) qui donne l'identificateur réel de l'objet manipulé et de "T2" (défini statiquement) qui est l'identificateur du type de la variable de manipulation. Nous évaluons le coût d'un tel accès à 7 unités ce qui est, dans le cas général, plus que pour C++.

2. **Appel de méthode**

   Pour l'appel de méthode, le coût est aussi constant. Le code généré par le compilateur PEPLOM pour l'appel de la méthode "m2" est le suivant :

```
(*(TabDesc[I.TypeId][T2].vtbl[ind_m2]))
    (I, P, ...
```

   La première ligne de code détermine le code de la méthode à appeler et la deuxième montre le passage en paramètre de l'objet auquel elle s'applique. Nous passons pour cela l'identificateur de l'objet et la poignée. Le coût évalué pour un tel appel est le même que pour C++ à savoir 11 unités ce qui est intéressant. De plus, nous pouvons remarquer que nous n'avons pas besoin que l'objet soit chargé en mémoire pour pouvoir déterminer le code à exécuter, le défaut d'objet étant effectué par la méthode si besoin est.

L'évaluation du coût qui est faite ici pour C++ comme pour PEPLOM est évidemment approximative. Nous avons donc opéré quelques mesures qui nous ont notamment montré que dans l'état actuel du prototype, l'appel de méthode, à fonctionnalité équivalente (héritage virtuel et méthode virtuelle pour le code C++ évalué), était en PEPLOM environ 35% plus cher qu'en C++ (autour de 1,5µs sur un Bull−DPX2 à base de 68040). Le coût de l'indépendance des objets vis à vis de leur adresse d'implantation en mémoire qu'apporte le mécanisme mis en œuvre par PEPLOM n'est donc pas prohibitif.

VI.3.3 Modules et variables persistantes

Nous avons vu dans les sections précédentes comment les objets engendrés par les types abstraits PEPLOM étaient mis en œuvre, de même que les méthodes qui peuvent leur être appliquées. L'autre dimension du langage PEPLOM que nous n'avons pas étudiée dans ce chapitre d'implémentation est la notion de module.

Les modules PEPLOM, comme nous l'avons vu, contiennent d'une part des définitions de variables et d'autre part des définitions d'opérations. Les opérations PEPLOM sont implantées sous la forme de simples fonctions C. Les variables non persistantes sont, de la même manière, représentées par des variables globales C. Par contre, le problème est différent pour les variables persistantes.

En effet, lors du lancement d'une application PEPLOM, celle−ci doit faire le lien entre un désignateur, représentant une variable persistante, dans le code de l'application et le contenu de cette variable sur le disque. Le choix que nous faisons, pour résoudre le problème, est tout d'abord d'affecter un identificateur unique pour chaque variable persistante. Cet identificateur est alloué statiquement par le gestionnaire du catalogue de définitions. C'est à partir de cet identificateur que nous recherchons le contenu de la variable dans l'espace de stockage, comme nous allons le voir.

Nous représentons une variable persistante par un objet persistant. L'association entre l'identificateur logique de l'objet et l'identificateur de la variable est stockée dans un volume spécifique : le volume des définitions de variables persistantes. De même que le volume racine a un identificateur logique prédéfini, le volume des variables en a un aussi qui est celui d'indice un sur le disque primaire. L'intérêt de cette solution est qu'elle permet de définir dynamiquement de nouvelles variables persistantes d'une façon très simple.

VI.4 Performances

Par rapport à l'approche que nous avons suivie, les systèmes dont nous sommes les plus proches sont donc ceux de type C++ persistant. Bien que fonctionnellement le langage PEPLOM soit plus propre que ces systèmes du point de vue de l'intégration des concepts bases de données (orthogonalité totale dans PEPLOM), il est intéressant aussi d'évaluer notre implémentation par rapport à ces systèmes.

Pour pouvoir réellement se placer par rapport à ces autres systèmes, il aurait fallu que nous les ayons à disposition sur nos machines. Comme ce n'était pas le cas, les comparaisons que nous pouvons faire, par rapport à des résultats que nous avons récupérés d'évaluations externes de ces systèmes, sont subjectives. Le but n'est donc pas de faire une comparaison précise mais d'observer par rapport à des données absolues mesurées sur des machines différentes si les ordres de grandeur sont comparables.

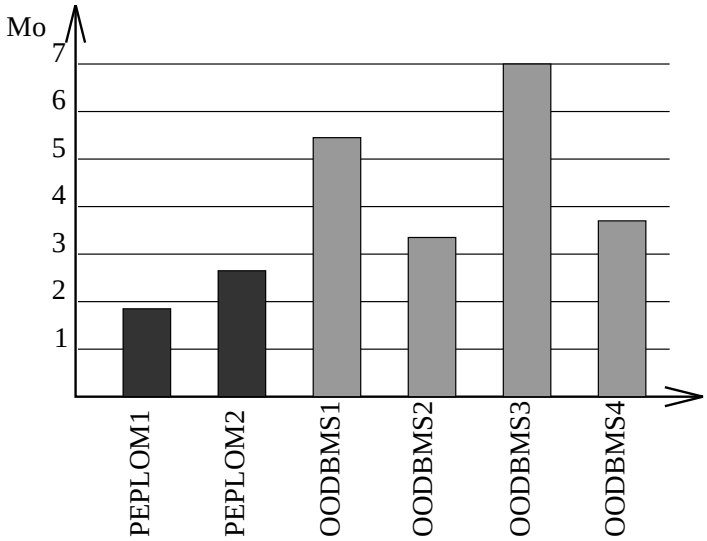


Fig. 6.14 : Comparaison des tailles des bases

Les évaluations de performances que nous avons faites concernent le test des SGBD à objets, dit "Cattell Benchmark", définit dans [2 9] . Ce test n'est pas non plus un outil de mesure absolu car il peut donner lieu à différentes interprétations donc à différentes implantations (les spécifications ne sont pas assez strictes). Les objets, mis en œuvre dans le test, correspondent à des parties d'un système dont le nombre est soit de dix mille, soit de vingt mille. Chaque partie est reliée à trois sous−parties déterminées aléatoirement parmi les

autres. Le lien inverse doit être maintenu pour les objets ainsi référencés. Deux réalisations sont proposées dans la définition du test : soit avec une seule classe définissant la partie contenant alors le triple lien et son inverse, soit avec deux classes, l'une définissant une partie et l'autre définissant les liens entre les parties. Nous avons choisi la première puisque le langage PEPLOM supporte un modèle d'objets complexes.

Nous avons implanté les objets de manière à ce que toute la valeur qui leur est associée soit stockée dans un seul segment. Ils peuvent donc être amenés à grossir puisqu'ils contiennent un ensemble. Enfin, nous avons écarté les mesures dites "à froid" car leur définition est ambiguë ; dans les mesures que nous avons pu récupérer, il n'est pas certain que les entrées/sorties disques soient réellement effectuées. En effet, même si le cache du SGBD est effectivement vide avant la mesure, les pages qui sont censées être lues depuis le disque peuvent très bien être "cachées" par le système d'exploitation. Les mesures que nous présentons pour les autres systèmes sont issues d'une plaquette explicative sur le test éditée par le fournisseur d'un SGBD à objets actuel. Les quatre SGBD que nous référençons sont ObjectStore, Objectivity, Ontos et Versant.

La première mesure concerne la taille occupée par la base sur le disque (cf. Fig. 6.14) et le temps de création (cf. Fig. 6.15) de celle−ci (commit compris). Nous avons deux résultats dans ce cas pour PEPLOM qui correspondent à deux nuances d'implantation des objets PEPLOM (réservation statique pour quatre éléments dans l'ensemble des références inverses pour PEPLOM1 et pour douze pour PEPLOM2).

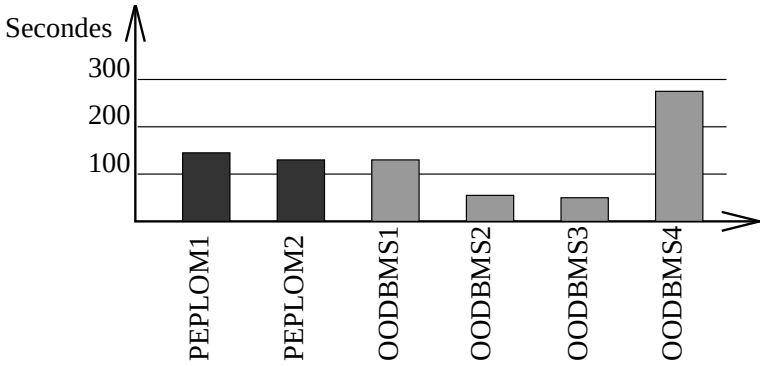


Fig. 6.15 : Comparaison des temps de création des bases

Nous pouvons observer que PEPLOM est très économe pour ce qui est de l'espace disque occupé et qu'au niveau des temps de création de la base, il se situe correctement par rapport aux autres systèmes.

Nous allons maintenant comparer les temps "à chaud" (les données sont dans le cache du SGBD) pour trois tests différents :

1. Le premier (cf. Fig. 6.16) est un test de parcours de l'arborescence des parties, à travers le triple lien, récursivement sur sept niveaux, avec application d'une méthode vide sur chaque objet traversé. Cela correspond à 3280 objets traversés.

2. Le deuxième test (cf. Fig. 6.17) recherche 1000 parties déterminées aléatoirement dans l'ensemble de toutes les parties et applique une méthode vide à chaque objet correspondant à une des parties recherchées.
3. Le troisième test (cf. Fig. 6.18) correspond à la création de 100 nouvelles parties et à les connecter comme les autres à trois parties déterminées aléatoirement. Le temps de commit fait partie du temps mesuré.

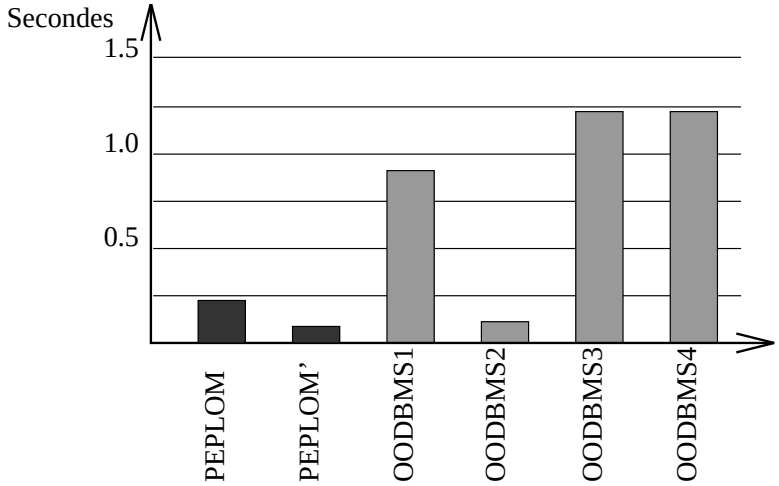


Fig. 6.16 : Comparaison des temps de traversée "à chaud"

Le test de traversée montre de très bons résultats pour PEPLOM qui se situe dans le même ordre de grandeur que le plus performant des SGBDOO de ce point de vue. Pour PEPLOM, ce temps n'est pas constant (la figure Fig. 6.16 donne un temps moyen) : il varie en fait entre 0.08 et 0.32 seconde.

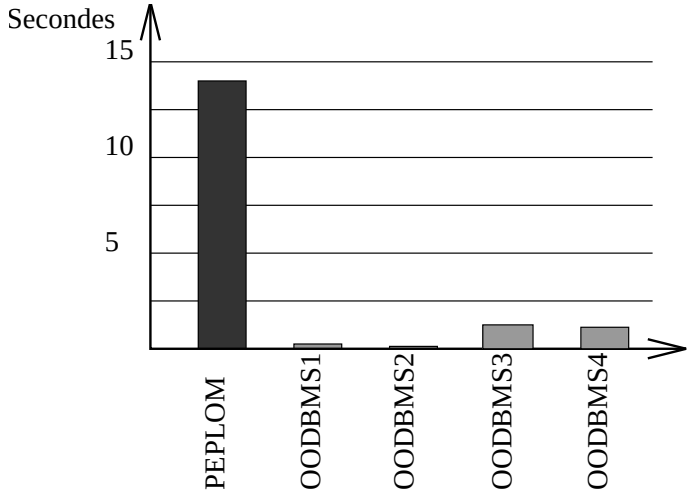


Fig. 6.17 : Comparaison des temps de recherche de 1000 éléments "à chaud"

Ceci est dû au fait que l'ensemble des parties est implanté de façon naïve et que la recherche d'une partie se fait séquentiellement, ce qui n'est pas le cas pour les autres

systèmes (recherche dans un index). Nous donnons, pour indication, le temps PEPLOM' qui correspond à la traversée des 3280 objets sans la recherche de l'objet de départ (ce temps est constant).

Le problème de l'implantation naïve de l'ensemble de toutes les parties pénalise encore plus la recherche des 1000 objets comme le montre la figure Fig. 6.17. Nous obtenons des temps de 10 à 30 fois supérieurs à ceux des SGBDOO ce qui est tout à fait logique.

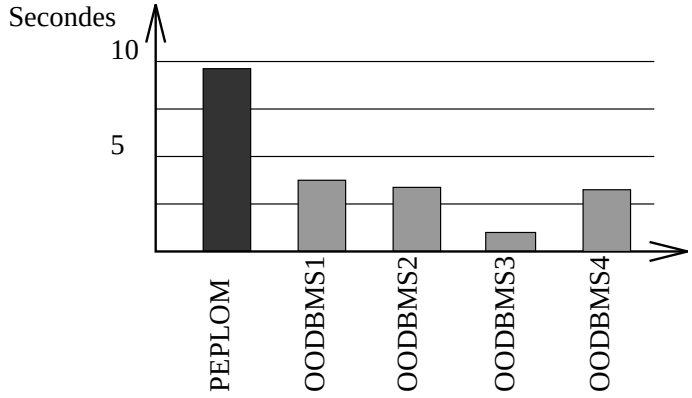


Fig. 6.18 : Comparaison des temps de création de 100 nouvelles parties "à chaud"

Nous observons aussi un temps de création de 100 nouvelles parties nettement supérieur aux autres systèmes ce qui peut paraître étonnant par rapport aux résultats obtenus pour la création de la base. En fait, c'est à nouveau le problème lié à la recherche séquentielle d'une partie. Cette recherche se fait une fois pour chacune des 300 nouvelles connexions effectuées. L'illustration de ce problème est effectivement démontrée par le fait que la création des 100 nouvelles parties sans connnexion (commit compris) ne coûte que 2.2 secondes.

VI.5 Conclusion

Dans ce chapitre, nous avons décrit l'environnement d'exécution du langage PEPLOM. Nous avons pu observer que l'approche utilisée consiste à ce que le maximum de fonctionnalités soient prises en compte par le générateur de code du compilateur. Les couches basses sur lesquelles s'appuie le code généré ont principalement pour rôle d'offrir une gestion de la mémoire virtuelle et de la mémoire secondaire simple et efficace. Nous avons néanmoins un premier niveau de support de la notion de collection offert par ce gestionnaire de mémoire.

Nous avons montré que le premier prototype que nous avons produit donne déjà des performances très intéressantes même si beaucoup de points restent encore à travailler. C'est notamment le cas de toutes les méthodes d'accès (B−arbre, index haché, etc...) qui sont nécessaires si nous voulons obtenir des performances satisfaisantes pour les accès ensemblistes.

Le code développé pour la machine de support d'objets, qui n'est pas totalement terminée, a un volume d'environ 20000 lignes de C, le paginateur sur lequel elle s'appuie actuellement représentant environ 9000 lignes supplémentaires. Le générateur de code PEPLOM quant à lui ne couvre à l'heure actuelle qu'une petite partie des fonctionnalités du langage (en fait, celles que nous voulions valider et que nous avons décrit ci−dessus). Il représente pour l'instant environ 12000 lignes de C. Enfin, il faudrait ajouter à ce code, pour couvrir tout ce qui a été développé concernant l'environnement d'exécution, une partie de l'implémentation de l'interprète des constructeurs "dynamiques" (environ 3000 lignes de C). Tout ce code a donc été validé avec différents tests, dont le plus important est le "Cattell Benchmark". Il n'empêche que sa fiabilité de même que son efficacité peuvent être améliorées.

Chapitre VII

Conclusion

Et pourtant, malgré tous ces fantastiques progrès et tous les soins et les fumures qu'elle recevait, la terre n'en pouvait plus. Elle qui avait fait vivre des générations de Vialhe et, avant eux, pendant des dizaines de siècles, d'autres générations d'inconnus, devenait d'année en année incapable de nourrir ceux qui l'entretenaient pourtant de mieux en mieux. (...) Même aux pires époques de crise, celles d'avant−guerre, jamais Pierre−Edouard n'avait vu un tel gâchis, un avenir si sombre pour ceux qui s'accrochaient à leur terre. S'y accrochaient, car elle était leur unique moyen de survivre, leur dernier combat. Ils ne savaient rien faire d'autre que se pencher vers elle pour la solliciter.

Claude Michelet − "L'appel des engoulevents"

Le travail présenté dans cette thèse relève du domaine des langages de programmation pour base de données. Ces langages sont issus de la volonté d'aller vers une plus forte intégration entre langage de programmation et système de gestion de base de données. Les concepts offerts par les langages à objets permettent effectivement d'envisager sérieusement cette intégration.

L'approche objet pour les SGBD conduit à avoir un langage complet pour exprimer le code des méthodes. Comme il est clair qu'il est très difficile d'imposer un nouveau langage général, la plupart des SGBD ou LPBD étendent un langage existant (C++, C, Smalltalk) aux concepts nécessaires à un SGBD : collection, persistance ou encore transaction.

Du point de vue du langage, nous constatons que cette approche est rarement fructueuse dans la mesure où ces nouveaux concepts sont rarement orthogonaux aux concepts d'origine de tels langages. Or, cette orthogonalité est à la base d'une utilisation sûre et cohérente d'un langage par un programmeur.

Du point de vue des SGBD, la sécurité des données manipulées est une fonctionnalité primordiale. Or, les langages de programmation usuels sont rarement sûrs, les langages à objets n'apportant pas non plus de réponse définitive à ce problème. La perte d'orthogonalité citée précédemment hypothèque d'autant plus cette sécurité.

Ces différentes remarques, parmi d'autres, nous ont conduit à envisager une approche plus risquée consistant à définir un nouveau langage. Ce langage nous sert alors à évaluer plus systématiquement et plus librement les choix de combinaisons de concepts des langages à objets et ceux des bases de données. Cette thèse s'attache donc à décrire cette étude qui couvre à la fois les choix de définition d'un modèle de données et d'un langage et, les choix

d'implantation de toutes les fonctionnalités prises en compte. Nous avons d'ailleurs pu voir que les différents choix sont très inter−dépendants.

VII.1 Les résultats

Le rôle de cette section est double. Nous voulons dans un premier temps faire un résumé du travail qui a été effectué au cours de cette thèse. Ce résumé couvre aussi bien les aspects d'étude bibliographique que les aspects d'implantation qui ont été mis en œuvre. Par rapport à l'expérience acquise à ces deux niveaux, nous essayons, dans un second temps, de donner notre vision sur la manière dont le domaine pourrait évoluer.

VII.1.1 Un résumé du travail effectué

La thèse débute par deux chapitres d'étude bibliographique. Le premier chapitre analyse la notion de modèle de données et plus généralement de système de types. Ces notions sont présentes dans les différentes générations de SGBD, de même que dans les langages de programmation. Nous nous attachons à comprendre les relations entre ces différents modèles de données et le modèle de mémoire nécessaire à leur support. Il est clair de ce point de vue que la nécessité d'avoir un modèle de stockage pour un SGBD impose des contraintes très différentes par rapport à un langage manipulant ses données uniquement en mémoire virtuelle.

Le principal problème à ce niveau est de limiter au maximum un trop grand fractionnement des données dans le cas d'un SGBD. En effet, un fractionnement important nécessite des techniques de regroupement complexes pour limiter le taux d'entrées/sorties disques lors d'une recomposition de l'objet fractionné. Nous avons vu que les choix au niveau du système de types et notamment des règles de sous−typage supportées ne sont pas neutres vis−à−vis de ce problème. D'une manière générale, nous essayons d'analyser le comportement de tous les concepts introduits par des langages orientés objet dans le contexte d'un SGBD. Nous étudions pour cela les systèmes de types d'un certain nombre de systèmes : Orion, Exodus (Extra), O2, Guide, E, ObjectStore, ONTOS, ODE.

Le deuxième chapitre est un état de l'art sur les différents SGBD à objets qui sont soit des prototypes, soit des produits commercialisés. Les systèmes étudiés sont GemStone, Orion, O2, Exodus (E), ObjectStore, ONTOS, ODE. Un SGBD à objets nécessite un langage complet pour exprimer le code des méthodes associées à ses objets. Nous remarquons que tous les systèmes décrits ici utilisent pour cela un langage existant, la plupart du temps C++. Nous distinguons aussi différentes approches d'implantation qui vont du système totalement interprété au système compilé, pour lequel la cible de compilation peut être de plus ou moins haut niveau. Cette cible peut être une machine abstraite supportant le modèle d'objet et l'appel de méthode ou un gestionnaire de mémoire relativement primitif et le simple appel de procédure.

Nous comparons donc tous ces systèmes du point de vue des fonctionnalités offertes ainsi que du point de vue des performances et de la sécurité qu'ils offrent. Une constatation que nous pouvons faire dans pratiquement tous les cas, notamment pour tous les systèmes

construits autour de C++, est que la volonté d'être au maximum compatible avec le langage de programmation initial est une contrainte importante. Elle diminue les libertés de choix aussi bien sur le plan fonctionnel qu'architectural et conduit bien souvent à une perte conséquente de sécurité pour le système.

C'est pour cette raison que nous avons choisi de ne pas nous lier à un langage existant. Nous avons défini un langage PEPLOM dans le but de prospecter, sans contrainte pénalisante, les meilleures combinaisons possibles des nombreux concepts présents dans les langages de programmation à objets avec ceux issus des SGBD. Ce n'est pas un langage totalement nouveau dans la mesure où sa base syntaxique est le langage C++. L'intérêt du LPBD PEPLOM est que le programmeur possède dans son langage la puissance des fonctionnalités d'un SGBD à travers un minimum de nouveaux concepts. En fait, les deux seuls nouveaux concepts qu'il peut utiliser sont les suivants :

- Les collections sont introduites sous la forme de nouveaux constructeurs du langage au même niveau que les tableaux des langages de programmation usuels. Ces nouvelles données sont manipulables à travers différents opérateurs. La possibilité de construire des expressions avec les collections est un des points forts du langage. Elle permet notamment de définir sous la forme d'expression, des manipulations associatives de collections à l'aide de fonctions génériques du type "select" ou "group" (équivalentes à des opérateurs du type SQL).
- La persistance est introduite soit lors de la création d'un nouvel objet, soit lors de la définitions d'une variable globale d'une application PEPLOM (variable définie dans un module PEPLOM). La validité des liens inter−objets est conservée grâce à la propagation de la propriété de persistance, lors du rattachement d'un objet non persistant à un objet persistant. Cette propagation est assurée par le système de support du langage.

Ces nouveaux concepts sont introduits de manière orthogonale aux concepts initiaux que nous avons emprunté à C++. Par exemple, les constructeurs de collection sont soumis aux mêmes règles que les autres constructeurs et la persistance est une caractéristique attachée aux données et non aux types. Cette orthogonalité est essentielle puisqu'elle garantit au programmeur une sémantique propre et une exécution correcte pour toutes les constructions qu'il peut définir à partir du langage.

PEPLOM offre, par ailleurs, deux niveaux de modularité : les types abstraits et les modules. Les types abstraits définissent des objets auxquels est associé un comportement commun. Ils sont organisés sous la forme d'un graphe orienté sans cycle (DAG) par la relation d'héritage multiple. Leur définition est purement "intentionnelle", n'est liée à aucune instance et ne comporte pas la notion de persistance. Elle offre ainsi un fort pouvoir de réutilisation.

Les modules ont pour rôle de définir un contexte d'instances formé de deux partitions : le contexte persistant (base de données) et le contexte temporaire (durée de vie égale à celle de l'exécution d'une application). Ce sont donc eux qui introduisent la notion de persistance. Le contexte persistant est partagé par les différentes applications travaillant sur la même base de données. Les modules définissent aussi des opérations dont une correspond au point d'entrée de l'application. Ils sont organisés sous la forme d'un graphe par la relation d'importation.

Les données créées par des applications PEPLOM pouvant être persistantes, cela engendre des problèmes concernant les définitions qui composent une telle application. Les données stockées dans la base n'ont en effet de sens que si nous connaissons aussi leur type. Nous arrivons alors au problème du maintien de la cohérence entre les données de la base et la définition de leur type.

C'est pour pouvoir envisager la résolution de ces problèmes que nous avons décidé que les définitions PEPLOM doivent être gérées par l'environnement de développement et non par le programmeur. Le langage s'utilise à travers un environnement basé sur des techniques de compilation incrémentale. Toutes les définitions sont stockées dans un dictionnaire de ressources qui fournit aux outils de l'environnement les définitions demandées par le programmeur. Nous parlons de compilation incrémentale dans la mesure où le gestionnaire du dictionnaire maintient un état global du schéma de définitions et propage les mises à jour lorsque c'est nécessaire (création, destruction ou modification d'une définition).

Cette approche est très intéressante car le programmeur n'a pas à se soucier des différentes inter−dépendances entre les définitions composant son application. Pour lui, la production d'une telle application consiste dans une première phase à créer et à valider toutes ses définitions et, lorsqu'il veut exécuter son application, à demander à l'environnement de générer un programme exécutable. C'est alors l'environnement du compilateur qui s'occupe de récupérer toutes les définitions qui lui sont nécessaires, de générer le code correspondant et de produire l'exécutable. La production d'un programme exécutable nécessite d'avoir une machine cible pour le générateur de code PEPLOM.

Le dernier chapitre s'attache donc à décrire comment s'organise la machine d'exécution d'un programme PEPLOM. La couche de plus bas niveau sur laquelle s'appuie cette machine est un paginateur, c'est à un dire un module qui simule une mémoire paginée persistante. L'unité de travail de cette couche est la page qui possède une taille fixe. Nous n'avons pratiquement pas étudié les problèmes liés à l'architecture (client/serveur) du SGBD que nous avons construit. Nous nous sommes principalement concentrés sur la décomposition logicielle de l'environnement d'exécution.

L'objectif, de ce point de vue, était d'offrir un environnement d'exécution le plus efficace possible. Il y a là essentiellement deux points qui influent sur les performances. Le premier est l'efficacité de la machine d'exécution du paradigme objet et le second le taux d'entrées/sorties disques pour charger un objet et, le coût de son déréférençage et de sa recomposition en mémoire d'exécution.

Pour la machine d'exécution des objets, nous avons suivi la même démarche que C++, c'est à dire compiler statiquement tous les adressages connus (accès aux champs d'un n−uplet, aux éléments d'un tableau, etc...). Pour cela, nous faisons l'hypothèse qu'un objet est fractionné le moins possible. Pour la gestion de la résolution tardive de code nécessaire au traitement du polymorphisme *ad hoc*, nous utilisons aussi une technique proche de celle implantée par C++ pour l'héritage virtuel et les méthodes virtuelles.

Nous avons néanmoins une technique qui a l'avantage de rendre les objets PEPLOM indépendants de leur implantation en mémoire d'exécution et de l'implantation en mémoire du code qui les manipule ce qui n'est pas le cas pour C++. Cette technique offre par ailleurs des performances très proches de celles de C++. L'intérêt du format des objets PEPLOM est

qu'il est le même en mémoire d'exécution et en mémoire secondaire. Le seul coût de chargement d'un objet est donc celui de son transfert de la mémoire secondaire dans la mémoire d'exécution et inversement. Ce coût est d'ailleurs incompressible.

La gestion de la mémoire secondaire permet de stocker des objets PEPLOM qui sont des zones de mémoire contiguë. Nous avons construit une couche de gestion mémoire au−dessus du paginateur qui gère de telles zones appelées des segments. Les objets PEPLOM pouvant grossir au cours de leur utilisation, le segment dans lequel ils sont stockés doit alors pouvoir grossir. Cette couche de gestion mémoire supporte cette fonctionnalité. L'allocation de segments se fait par granule, une page contenant toujours des granules de taille identique. Cette technique permet d'éviter l'effritement de la mémoire secondaire au cours des cycles d'allocation/désallocation.

Les segments sont par ailleurs alloués dans des volumes qui correspondent à des ensembles de segments. Le gestionnaire de volume permet de parcourir tous les segments occupés d'un volume. Ces volumes sont la base d'implantation de la notion de collection persistante et sont utilisés pour matérialiser les extensions persistantes des types abstraits PEPLOM.

Enfin les objets PEPLOM sont désignés par un mécanisme d'identificateur logique. Un identificateur logique est un relais d'indirection dans la désignation d'un objet. Ce relais permet notamment de résoudre le problème de réimplantation d'objets lorsque ces derniers grossissent. Nous avons néanmoins pu constater que notre implantation des identificateurs logiques pénalise peu les performances.

Les résultats que nous avons obtenus avec le prototypage des différentes fonctionnalités que nous voulions évaluer sont encourageants. Ils valident en grande en partie les choix d'implantation que nous avons faits. Ils nous ont notamment montré que l'utilisation d'identificateurs logiques comme relais d'indirection pour l'implantation d'un objet est viable du point de vue des performances. Cette technique ouvre des perspectives intéressantes notamment pour des mécanismes de ramasse−miettes en mémoire secondaire car les identificateurs logiques peuvent être gérés sous la forme d'une table stockée de façon contiguë sur le disque.

Pour résumer les principaux apports de cette thèse, nous pouvons commencer par l'analyse basée sur l'étude bibliographique. Elle nous permet de mieux comprendre quels sont les points de convergence et de divergence issus du foisonnement de SGBD ou LPBD à objets. Nous insistons notamment sur le fait que certains choix faits au niveau du modèle conceptuel imposent des contraintes importantes sur les implantations possibles. Nous poursuivons cette analyse par une comparaison des systèmes existants en essayant d'extraire de ceux−ci les points faibles et les points forts autant du point de vue du langage que du système. A partir de là a été défini le LPBD PEPLOM dont l'intérêt est d'introduire les fonctionnalités d'un SGBD dans un langage à objets proche de C++. Les concepts introduits sont simples et peu nombreux. Leur intégration est orthogonale par rapport aux autres concepts initiaux présents dans le langage ce qui nous paraît être un apport important. Nous montrons aussi quelles sont les étapes dans la production d'un programme PEPLOM et en quoi les techniques de compilation incrémentale sont essentielles dans un contexte de

programmation persistante. Le dernier aspect important que nous présentons est l'environnement d'exécution développé pour le langage. L'intérêt de notre approche consiste à avoir les couches basses du systèmes les plus simples possibles, la plus grande partie de la sémantique du langage étant supportée par le code généré. Nous avons pu constater qu'une telle approche est garante d'efficacité et de facilité de mise au point. Nous avons aussi validé les couches basses (gestion mémoire virtuelle et persistante) qui offrent la notion d'identificateur logique et qui permettent de faire grossir les objets, ce qui nous permet de fractionner le moins possible nos objets. Nous avons aussi défini un format d'objet, propre à PEPLOM, qui à l'avantage de rendre les objets indépendants de leur implantation en mémoire. Notre technique représente un plus important par rapport à des systèmes de type C++ persistant et nous garantit de meilleures performances au chargement et plus de libertés quant aux architectures client/serveur que nous pourrions utiliser. Les résultats obtenus avec le prototype actuel sont très encourageants et nous ont montré que la voie sur laquelle nous nous sommes engagés est très prometteuse.

VII.1.2 Un point de vue sur les évolutions du domaine

Si nous considérons le nombre croissant de systèmes qui apparaissent sur le marché des SGBD, la technologie à objets, dans le domaine des bases de données, semble être arrivée à une maturité suffisante. L'engouement pour cette technologie ne touche pas seulement les bases de données. Il a d'abord touché le domaine des langages de programmation, et concerne aussi l'intelligence artificielle ou les systèmes répartis.

Pour les bases de données, la question principale qui se pose est de savoir dans quelle mesure les SGBD à objets seront amenés à remplacer les systèmes relationnels, dont l'émergence a marqué la dernière décénie et dont la croissance est toujours forte, et si c'est le cas, dans quels délais[71].

La première constatation que nous pouvons faire est que le foisonnement actuel de nouveaux SGBD à objets montre que cette technologie est très prometteuse. Il reste que tous ces systèmes ne sont pas aussi consensuels que les SGBD relationnels qui partagent un même modèle de données et qui offrent un réel progrès en termes de simplicité et de souplesse d'utilisation. Ce n'est pas le cas des SGBD à objets qui imposent un apprentissage beaucoup plus fastidieux des nombreux concepts qui y sont combinés.

Dans ces systèmes, le modèle de données est induit par le système de types. Or, il nous paraît beaucoup plus difficile, voire impossible qu'il se dégage un jour un consensus sur un système de types standard pour la technologie à objets : il suffit de constater les nombreux efforts[37] engagés dans ce sens qui n'ont pas, jusqu'à présent, pu aboutir. En effet, nous avons vu que nombre de concepts sémantiques introduits par les systèmes de types (règles d'héritage, de sous−typages, etc...) engendrent des contraintes importantes vis−à−vis de leur implantation. Il nous semble plus raisonnable de penser que si un consensus se fait jour, ce sera au niveau du modèle structurel des données, sur la base d'un modèle de valeurs complexes. Un tel modèle pourrait servir de format d'échange pour les interactions entre SGBD.

Même si nous avons dit que les SGBD à objets ont atteint une certaine maturité, ils ont encore de nombreuses lacunes. La principale concerne la faiblesse des langages qu'ils offrent pour les accès associatifs aux données de la base. La technologie développée par les systèmes relationnels à ce niveau n'est pas incompatible avec les systèmes à objets même si le contexte est un peu plus complexe. Nous pensons que les accès ensemblistes associatifs peuvent être aussi performants, et même plus performants dans certains cas, dans les SGBD à objets que dans les SGBD relationnels. En effet, les modèles à objets ont tendance a moins fractionner les objets et donc à limiter les jointures nécessaires à leur manipulation dans les SGBD relationnels. Le seul avantage du fractionnement est qu'il nécessite pour certaines requêtes simples une plus faible quantité de données à manipuler que dans le cas des SGBD à objets, où les objets sont manipulables comme un tout. Nous pensons néanmoins que le non fractionnement des données est globalement moins coûteux que le fractionnement.

Par rapport aux investissements que représente un changement de technologie en matière de base de données, le passage à la technologie "objet" sera certainement long. Ce fut le cas pour le passage à la technologie relationnelle qui a souffert à ses débuts de ses carences en termes de performances. Pour la technologie "objet", le problème nous semble plus lié à la nature de l'offre qui est très hétéroclite. Il nous semble par exemple peu réaliste de penser que les SGBD à objets ressembleront aux actuels C++ persistants, qui semblent prendre une grosse part du marché des SGBD à objets, car ils sont de trop bas niveau et offrent une sûreté beaucoup trop faible. Leur utilisation requiert de réels spécialistes ; une utilisation comme système extensible pour produire des SGBD sur mesure nous paraît plus réaliste. L'évaluation et la comparaison de tous les SGBD à objets disponibles actuellement est donc très ardue [99].

C'est probablement pour cette raison, et pour le moindre coût de l'évolution vers la technologie à objets pour les utilisateurs, que les systèmes relationnels étendus à des concepts "objets" profiteront plus de la demande concernant les bases de données à objets. C'est ce que semblent indiquer des études marketing récentes [71], en tout cas à court et moyen terme. Il n'est pas sûr que cela reste vrai à long terme car de telles extensions peuvent remettre en cause beaucoup de choix faits par les concepteurs de ces systèmes et notamment des choix architecturaux (architecture client/serveur). Nous pouvons même penser que les SGBD relationnels étendus font, à long terme, le jeu des SGBD à objets car, lorsque les utilisateurs auront intégré la technologie à objets par ce biais, il est probable qu'ils utiliseront plus alors la puissance de modélisation de l'objet, l'aspect relationnel devenant ainsi obsolète.

Pour conclure, nous estimons que la technologie à objets à de bonnes chances de supplanter la technologie relationnelle à long terme, sachant qu'une grande partie de la technologie relationnelle sera adaptée au monde "objet". Cette technologie est néanmoins beaucoup plus complexe à appréhender et à gérer. Nous pensons donc que l'offre pour les bases de données à objets devra être plus globale et concerner le langage et le système aussi bien que le génie logiciel. Elle se composera probablement d'environnements d'outils intégrés simplifiant l'utilisation de cette technologie aussi bien du point de vue du langage que du système. C'est donc dans ce sens que nous avons commencé à travailler et c'est ce but d'une offre globale que nous voulons poursuivre.

VII.2 Les perspectives

Le travail dans lequel nous nous sommes engagés est un travail de grande ampleur car il couvre tous les niveaux de définition d'un SGBD et d'un langage. Les premières perspectives concernent la prise en compte dans le prototype, de toutes les fonctions qui sont présentes dans le langage. C'est notamment le cas pour les expressions ensemblistes qui nous amènent à envisager leur traitement avec par exemple une approche algébrique. Cela nous conduit alors aussi vers l'étude de techniques d'optimisations liées à la machine physique et son langage de description que nous avons proposé.

Les autres perspectives concernent d'une part l'extension du langage vers de nouvelles fonctionnalités et, d'autre part, à l'étude plus approfondie de l'architecture du système supportant le langage.

VII.2.1 L'évolution du langage PEPLOM

Les principales évolutions du langage que nous pouvons envisager recouvrent plusieurs axes. Le premier axe est d'envisager l'introduction dans le langage de contraintes d'intégrité[73]. Cet aspect a jusqu'à maintenant été très peu traité dans le contexte des bases de données à objets. Il peut être envisagé dans un contexte plus général de traitement de règles et de déclencheurs. Cet axe est en train de prendre beaucoup d'importance dans la communauté des bases de données : on parle alors de systèmes DOOD (Deductive and Object−Oriented Database systems). Ce travail est d'ailleurs déjà en cours dans le cadre du projet européen IDEA[44] (Intelligent Database Environment for Advanced Applications). Cet objectif n'est pas trivial à atteindre, même du point de vue du langage, car ce sont deux modes de raisonnement différents qu'il faut combiner : la programmation logique (déclarative) et la programmation à objets (impérative). Cet antagonisme opère aussi sur le modèle d'exécution mis en œuvre (chaînage avant ou chaînage arrière).

Un deuxième axe qui nous semble intéressant concerne la possibilité de traiter l'évolution des objets d'un type à un autre au cours de leur existance. Une telle fonctionnalité correspond d'ailleurs à des phénomènes naturels. Nous pouvons par exemple prendre une personne qui, au cours de sa vie, va changer de situation : elle va être un nouveau−né, puis un élève, puis un étudiant, puis un employé en même temps qu'un contribuable, etc... Il est probable qu'une telle fonction aurait avantage en plus à être couplée avec un mécanisme de gestion de versions[46]. La dynamicité des objets introduite par cette fonction semble très intéressante dans bien des applications. Néanmoins, l'implantation n'en est pas aisée car cela demande que les objets puissent évoluer structurellement. Un tel mécanisme est donc relativement lourd à mettre en œuvre.

VII.2.2 Le support du langage

Le système supportant l'environnement d'exécution du langage demande encore beaucoup de travail. L'état actuel des résultats acquis notamment concernant l'architecture n'est pas satisfaisant. La coopération, dans une architecture client/serveur, entre le client et le serveur n'est pas évidente. Nous voudrions en effet, à priori, que les traitements ensemblistes sur de grosses quantités de données s'exécutent sur le serveur qui possède plus

de puissance pour cela, et que les traitements plus poussés sur de petits nombres d'objets soient exécutés par le client. De tels objectifs ne sont pas simples à atteindre.

La parallélisation ou la distribution du serveur, suivant le but à atteindre (performances ou disponibilité), représente une voie importante dans l'évolution future des SGBD. Il semble en effet peu réaliste de pouvoir traiter efficacement des bases de données, pour lesquelles l'unité de taille envisagée est le téra−octet pour l'an 2000, avec des serveurs mono−processeur. Des systèmes visant des serveurs parallèles matérialisés soit par des machines multi−processeurs, soit par des environnements de type "cluster" ouvrent des champs d'investigation encore peu étudiés dans le contexte des bases de données à objets. C'est aussi le cas pour la distribution[48], dont le but n'est pas le même, mais qui pose des problèmes très proches de ceux issus du parallélisme.

Il semble que la recherche sur les systèmes d'information pris dans un sens très général se dirige vers une intégration de nombreuses technologies (langage, système, génie logiciel, intelligence artificielle, interface homme/machine, parallélisme). Cette tendance est relativement nette dans le domaine des bases de données, qui s'est déjà ouvert à la plupart de ces technologies. Les SGBD du futur seront probablement intégrés dans des offres globales couvrant tous ces aspects. Un des défis pour la recherche dans les années à venir pourrait bien être la gestion de la complexité de tels environnements intégrés.

Bibliographie

[1] S. ABITEBOUL et C. BEERI, *On the Power of Languages for the Manipulation of Complex Objects*, (INRIA Research Report #846), INRIA, Mai 1988.

[2] S. ABITEBOUL et P.C. KANELLAKIS, ''Object Identity as a Query Language Primitive'', *Proc. ACM SIGMOD 89 Conference*, Portland, Juin 1989.

[3] S. ABITEBOUL et A. BONNER, ''Objects and Views'', *Proc. ACM SIGMOD 91 Conference*, Denver, Mai 1991.

[4] M. ADIBA, C. LECLUSE et P. RICHARD, ''Rationale and Design of Serendip, a Database Programming Language'', *Proc. DEXA 91 Conference*, Berlin, Aôut 1991.

[5] M. ADIBA, C. COLLET, P. DECHAMBOUX et B. DEFUDE, ''Integrated Tools for Object−Oriented Persistent Application Development'', *Proc. DEXA 92 Conference*, Valencia, Septembre 1992.

[6] R. AGRAWAL et N.H. GEHANI, ''ODE (Object Database and Environment): The Language and the Data Model'', *Proc. ACM SIGMOD 89 Conference*, Portland, Juin 1989.

[7] A.M. ALASHQUR, S.Y.W. SU et H. LAM, ''OQL: a Query Language for Manipulating Object−Oriented Databases'', *Proc. VLDB 89 Conference*, Amsterdam, 1989.

[8] A. ALBANO, L. CARDELLI et R. ORSINI, ''Galileo: A Strongly−Typed, Interactive Conceptual Language'', *Readings in Object−Oriented Database Systems*, édité par S. Zdonik, D. Maier, Morgan−Kaufman, 1985.

[9] T. ANDREWS et C. HARRIS, ''Combining Language and Database Advances in an Object−Oriented development Environment'', *Proc. OOPSLA 87 Conference*, Orlando, Octobre 1987.

[10] ANSI, *(ISO/ANSI working draft) Database Language SQL2*, (X3H2−90−398), Octobre 1990.

[11] ARISTOTE (rapport intermédiaire), *Générateur d'applications orientées objet, multimédia*, (RAP014), Laboratoire de Génie Informatique et Centre de Recherche Bull, Grenoble, Juin 1991.

[12] M. ATKINSON et P. BUNEMAN, ''Types and Persistence in Database Programming Languages'', *ACM Computing Surveys*, 19(2), Juin 1987.

[13] M. ATKINSON, C. LECLUSE et P. RICHARD, *Bulk Types for Data Programming Languages − A Proposal*, (Altaïr 67−91), GIP ALTAIR, Février 1991.

[14] M. ATKINSON et al., ''The Object−Oriented Database Manifesto'', *Proc. DOOD 89 Conference*, Kyoto, Décembre 1989.

[15] F. BANCILHON, T. BRIGGS, S. KHOSHAFIAN et P. VALDURIEZ, ''FAD, a Powerful and Simple Database Language'', *Proc. VLDB 87 Conference*, Brighton, 1987.

[16] N.S. BARGHOUTI et G.E. KAISER, ''Concurrency Control in Advanced Database Applications'', *ACM Computing Surveys*, 23(3), Septembre 1991.

[17] C. BEERI, ''Formal Models for Object−Oriented Databases'', *Proc. DOOD 89 Conference*, Kyoto, Décembre 1989.

[18] C. BEERI et Y. KORNATZKY, ''Algebraic Optimization of Object−Oriented Query Languages'', *Proc. ICDT 90 Conference*, Paris, Décembre 1990.

[19] B. BERGSTEN, M. COUPRIE et P. VALDURIEZ, ''Prototyping DBS3, a shared store memory Parallel Database System'', *Proc. PDIS 91 Conference*, Décembre 1991.

[20] B. BERGSTEN, M. COUPRIE et P. VALDURIEZ, ''DBS3, an Implementation of the EDS Database System on a shared−memory Multiprocessor'', *ESPRIT CONFERENCE'91*, Bruxelles, Novembre 1991.

[21] C. BERRUT, M.F. BRUANDET, J.L. CHEVAL, P. DECHAMBOUX, S. JARWA et H. MARTIN, *La gestion de la dynamique : systèmes de gestion de bases de données vs systèmes d'intelligence artificielle*, (SUR003), Laboratoire de Génie Informatique et Centre de Recherche Bull, Grenoble, Octobre 1990.

[22] O. BOUCELMA et J. LE MAITRE, ''An Extensible Functional Query Language for an Object−Oriented Database System'', *Proc. DOOD 91 Conference*, Munich, Décembre 1991.

[23] A.L. BROWN, *Persistent Object Stores*, PhD Thesis, University of St Andrew & University of Glasgow, Mars 1989.

[24] K.B. BRUCE et P. WEGNER, ''An Algebraic Model of Subtype and Inheritance'', *Advances in Database Programming Languages*, édité par F. Bancilhon, P. Buneman, Addison−Wesley (ACM Press), 1990.

[25] P. BUTTERWORTH, A. OTIS et J. STEIN, ''The Gemstone Object Database Management System'', *Communication of the ACM*, 34(10), Octobre 1991.

[26] L. CARDELLI et P. WEGNER, ''On Understanding Types, Data Abstraction, and Polymorphism'', *ACM Computing Surveys*, 17(4), Décembre 1985.

[27] M. CAREY et al., *The Architecture of the EXODUS Extensible DBMS: A Preliminary Report*, (644), University of Wisconsin − Madison, Mai 1986.

[28] M. CAREY, D. DE WITT et S. VANDENBERG, ‘‘A Data Model and a Query Language for EXODUS’’, *Proc. ACM SIGMOD 88 Conference*, Chicago, Juin 1988.

[29] R.G.G CATTELL et J. SKEEN, ‘‘Object Operations Benchmark’’, *ACM Transactions on Database Systems*, 17(1), Mars 1992.

[30] C. CHACHATY−DOUMMAR, *Parallélisation de requêtes dans un système de gestion de bases de données relationnelles étendues*, Thèse de doctorat de 3ième cycle, Ecole Nationale Supérieure des Télécommunications, Octobre 1991.

[31] S. CLUET, C. DELOBEL, C. LECLUSE et P. RICHARD, ‘‘Reloop, an Algebra Based Query Language for an Object−Oriented Database System’’, *Proc. DOOD 89 Conference*, Kyoto, Décembre 1989.

[32] S. CLUET, *Langages et Optimisation de Requêtes pour Systèmes de Gestion de Base de Données Orientés−Objet*, Thèse de doctorat de 3ième cycle, Université de Paris−Sud − Centre d’Orsay, Juin 1991.

[33] E. CODD, ‘‘A Relational Model of Data for Large Shared Data Banks’’, *Communication of the ACM*, 13(6), Juin 1970.

[34] R.C.H. CONNOR, A.L. BROWN, Q.I. CUTS, A. DEARLE, R. MORRISON et J. ROSENBERG, ‘‘Type Equivalence Checking in Persistent Object Systems’’, *(FIDE/91/29)*, 1991.

[35] S. DANFORTH et P. VALDURIEZ, ‘‘A Fad for Data Intensive Applications’’, *IEEE Transaction on Knowledge and Data Engineering*, 4(1), Février 1992.

[36] U. DAYAL, N. GOODMAN et R.H. KATZ, ‘‘An Extended Relational Algebra with Control over Duplicate Elimination’’, *Proc. ACM PODS 82 Conference*, 1982.

[37] Digital Equipment Corporation, Hewlett Packard Company, HyperDesk Corporation, NCR Corporaration, Object Design Inc. et SunSoft, Inc., *The Common Object Request Broker: Architecture and Specification*, (90.10.5), OMG, Décembre 1991.

[38] P. DECHAMBOUX, *Construction d’un SGBD orienté objet au−dessus d’un SGBD relationnel − théorie et pratique*, DEA Informatique, Université Joseph Fourier − Grenoble I, Juin 1989.

[39] P. DECHAMBOUX, *The Catalogue Manager*, (EDS.DD.11B.1201), Esprit Project EDS, Janvier 1992.

[40] C. DELOBEL et M. ADIBA, *Bases de données et systèmes relationnels*, Dunod, 1982.

[41] C. DELOBEL, C. LECLUSE et P. RICHARD, *Bases de données : des systèmes relationnels aux systèmes à objets*, InterEditions, 1991.

[42] M.A. ELLIS et B. STROUSTRUP, *The Annotated C++ Reference Manual*, Addison−Wesley, 1990.

[43] R. ELMASRI et S.B. NAVATHE, *Funcdamentals of Database Systems*, Benjamin−Cummings, 1989.

[44] IDEA, *Technical Annex*, Esprit3: EP6333, Mars 1992.

[45] F. EXERTIER, *Extension orientée objet d'un SGBD relationnel*, Thèse de doctorat de 3ième cycle, Université Joseph Fourier − Grenoble I, Décembre 1991.

[46] M.C. FAUVET, *Modelling and Managing Versions and Histories in an Object−Oriented Environment*, (RAP015), Laboratoire de Génie Informatique et Centre de Recherche Bull, Grenoble, Juillet 1991.

[47] D. FISHMAN et al., ''IRIS: an Object−Oriented Database Management System'', *ACM TOIS*, 5(1), Janvier 1987.

[48] A. FREYSSINET, *Architecture et réalisation d'un système réparti à objets*, Thèse de doctorat de 3ième cycle, Université Joseph Fourier − Grenoble I, Juillet 1991.

[49] A. FREYSSINET, S. KRAKOWIAK et S. LACOURTE, ''A Generic Object−Oriented Virtual Machine'', *Proc. International Workshop on Object−Orientation in Operating Systems*, Palo Alto, Octobre 1991.

[50] G. GARDARIN et P. VALDURIEZ, *Relational Databases and Knowledge Bases*, Addison−Wesley, 1989.

[51] G. GARDARIN et P. VALDURIEZ, ''ESQL: An Extended SQL with Objects and Deductive Capabilities'', *Proc. DEXA 90 Conference*, 1990.

[52] A. GEPPERT, K.R. DITTRICH et V. GOEBEL, *Quod: A Query Language for NO2*, (Research Report ITHACA.Unizh.90.X.4.#2), Esprit Project ITHACA, Novembre 1990.

[53] A. GEPPERT, K.R. DITTRICH et V. GOEBEL, *An Algebra for the NO2 Data Model*, (Research Report ITHACA.Unizh.90.X.4.#3), Esprit Project ITHACA, Novembre 1990.

[54] A. GOLDBERG et D. ROBSON, *Smalltalk−80: The Language and its Implementation*, Addison−Wesley, 1983.

[55] R.H. GUTING, *Modeling Non−Standard Database Systems by Many−Sorted Algebras*, (255), Der Universität Dortmund, Mars 1988.

[56] R. HULL et J. SU, ''On Accessing Object−Oriented Databases: Expressive Power, Complexity, and Restrictions'', *Proc. ACM SIGMOD 89 Conference*, Portland, Juin 1989.

[57] K. JENSEN et N. WIRTH, *Pascal User Manual and Report*, Springer Verlag, 1977.

[58] W. KERNIGHAN et D. RITCHIE, *The C Programming Language*, Prentice Hall, 1978.

[59] S.N. KHOSHAFIAN et G.P. COPELAND, ''Object Identity'', *Proc. OOPSLA 86 Conference*, Portland, Septembre 1986.

[60] W. KIM et al., ''Integrating an Object−Oriented Programming System with a Database System'', *Proc. OOPSLA 88 Conference*, Septembre 1988.

[61] W. KIM et F.H. LOCHOVSHY, *Object−Oriented Concepts, Databases and Applications*, Addison−Wesley (ACM Press), 1989.

[62] W. KIM, ''A Model of Queries for Object−Oriented Databases'', *Proc. VLDB 89 Conference*, Amsterdam, 1989.

[63] R. KING et M. NOVAK, ''Building Reusable Data Representations with Face Kit'', *SIGMOD RECORD*, 21(1), March 1992.

[64] S. KRAKOWIAK, M. MEYSEMBOURG, H. NGUYEN VAN, M. RIVEILL et C. ROISIN, ''Design and implementation of an object−oriented, strongly typed language for disributed applications'', *Journal of Object−Oriented Programming*, 3(3), Septembre 1990.

[65] S. KRAKOWIAK et X. ROUSSET DE PINA, *Architecture du système Guide−2, résumé des choix de conception*, (13−92), Bull−IMAG, Grenoble, 1992.

[66] S. LACOURTE, *Exceptions dans les langages à objets*, Thèse de doctorat de 3ième cycle, Université Joseph Fourier − Grenoble I, Juillet 1991.

[67] C. LAMB, G. LANDIS, J. ORENSTEIN et D. WEINREB, ''The Objectstore Database System'', *Communication of the ACM*, 34(10), Octobre 1991.

[68] C. LECLUSE, P. RICHARD et F. VELEZ, ''O2, an Object−Oriented Data Model'', *Proc. ACM SIGMOD 88 Conference*, Chicago, Juin 1988.

[69] D. LENKOV, ''C++ standardization: top issues'', *Journal of Object−Oriented Programming*, 5(3), Juin 1992.

[70] C. LENNE, *EMIR : un outil de gestion de structures arborescentes attribuées*, BULL − Organisation de Recherche et Développements Avancés, Grenoble, Septembre 1990.

[71] P.C. LOCKEMANN, ''Object−Oriented Databases and Deductive Databases: Systems without Market? Market without Systems?'', *Proc. DEXA 92 Conference*, Valencia, Septembre 1992.

[72] G.M. LOHMAN, B. LINDSAY, H. PIRAHESH et K.B. SCHIEFER, ''Extensions to Starburst: Objects, Types, Functions, and Rules'', *Communication of the ACM*, 34(10), Octobre 1991.

[73] H. MARTIN, *Contrôle de la cohérence dans les bases objets : une approche par le comportement*, Thèse de doctorat de 3ième cycle, Université Joseph Fourier − Grenoble I, Janvier 1991.

[74] T. MATSUSHIMA et G. WIEDERHOLD, *A Model of Object−Identities and Values*, (Report #STAN−CS−90−1304), Stanford University, Février 1990.

[75] D. MAIER, ‘‘Why Isn’t There an Object−Oriented Data Model’’, *Information Processing 89*, édité par G.X. Ritter, North−Holland, 1989.

[76] B. MEYER, *Object−oriented Software Construction*, Prentice Hall, 1988.

[77] R. MORRISON, A.L. BROWN, R. CONNOR et A. DEARLE, *The Napier88 Reference Manual*, (PPRR−77), Universities of Glasgow and St. Andrews, 1989.

[78] H. Nguyen Van, M. Riveill et C. Roisin, *Manuel du langage Guide (V1.5)*, (3−90), Unité mixte Bull−Imag (Grenoble), Décembre 1990.

[79] O2 Technology, *The O2 User’s Manual*, 7, rue du Parc de Clagny 78000 Versailles (France), Décembre 1991.

[80] OBJECT DESIGN, *ObjectStore Documentation*, One New England Executive Park, Burlington, MA01803 (USA), Octobre 1990.

[81] O DEUX et al., ‘‘The Story of O2’’, *IEEE Transactions on Knowledge and Data Engineering*, 12(1), Mars 1990.

[82] A. OHORI, ‘‘Representing Object Identity in a Pure Functional Language’’, *Proc. ICDT 90 Conference*, 1990.

[83] Ontologic Inc., *ONTOS Developer’s Guide*, Février 1991.

[84] F. OQUENDO, G. BOUDIER, F. GALLO, R. MINOT et I. THOMAS, ‘‘The PCTE+’s OMS, a Software Engineering Distributed Database System for supporting Large−Scale Software Development Environments’’, *Proc. 2th International Symposium on Database Systems for Advanced Applications*, Avril 1991.

[85] ONTOS, *ONTOS Developer’s Guide*, 1990.

[86] G. PHIPPS et M.A. DERR, ‘‘Glue−Nail!: a Deductif Database System’’, *Proc. ACM SIGMOD 91 Conference*, Denver, Mai 1991.

[87] J.E. RICHARDSON et M.J. CAREY, ‘‘Implementing Persistence in the E Language’’, *International Workshop on Persistent Object System*, 1989.

[88] J.E. RICHARDSON, M.J. CAREY et D.T. SCHUH, *The Design of the E Programming Language*, (824), University of Wisconsin − Madison, Février 1989.

[89] J.E. RICHARDSON et P. SCHWARZ, *MDM: An Object−Oriented Data Model*, (RJ 8228 (75422)), IBM Almaden Research Center, Juillet 1991.

[90] C. RONCANCIO, *Génération d'applications de bases de données selon l'approche objet*, DEA Informatique, Université Joseph Fourier − Grenoble I, Juin 1991.

[91] M.A. ROTH, H.F. KORTH et A. SILBERSCHATZ, ''Extended Algebra and Calculus for Nested Relational Databases'', *ACM Transactions on Database Systems*, 13(4), Décembre 1988.

[92] J.W. SCHMIDT, ''Some High Level Language Constructs for Data of Type Relation'', *Proc. ACM SIGMOD 77*, Toronto, Août 1977.

[93] J.W. SCHMIDT et F. MATTHES, ''Naming Schemes and Name Space Management in the DBPL Persistent Storage System'', *Proc. Fourth Int. Workshop on Persistent Object Systems*, Morgan Kaufmann, Janvier 1991.

[94] J.W. SCHMIDT et F. MATTHES, *The Database Programming Language DBPL − Rational and Report*, (FIDE/92/46), Universität Hamburg, Juin 1992.

[95] M.H. SCHOLL et H.J. SCHEK, ''A Relational Object Model'', *Proc. ICDT 90 Conference*, 1990.

[96] A. SILBERSCHATZ, M. STONEBRAKER et J. ULLMAN, ''Database Systems: Achievements and Opportunities'', *Communication of the ACM*, 34(10), Octobre 1991.

[97] G.M. SHAW et S.B. ZDONIK, ''Object−Oriented Queries, Equivalence and Optimisation'', *Proc. DOOD 89 Conference*, Kyoto, Décembre 1989.

[98] G.M. SHAW et S.B. ZDONIK, ''A Query Algebra for Object−Oriented Databases'', *Proc. Data Engineering 90 Conference*, 1990.

[99] J. STEIN, ''Evaluating object database management systems'', *Journal of Object−Oriented Programming*, 5(6), Octobre 1992.

[100] M. STONEBRAKER et L. ROWE, *The POSTGRES Papers*, (Memorandum No. UCB/ERL M86/85), University of California−Berkeley, Novembre 1986.

[101] M. STONEBRAKER et G. KEMNITZ, ''The POSTGRES Next−Generation Database Management System'', *Communication of the ACM*, 34(10), Octobre 1991.

[102] B. STROUSTRUP, *The C++ Programming Language*, Addison−Wesley, 1986.

[103] J.D. ULLMAN, *Principles of Database Systems*, Computer Science Press, 1982.

[104] L. VIEILLE, P. BAYER, V. KUCHENHOFF et A. LEFEBVRE, ''EKS−V1, A Short Overview'', *SIGMOG'90 Technical Exhibition*, Atlantic City, Mai 1990.

[105] *VERSANT System Reference Manual, VERSANT Languages Interface Manual*, Versant Object Technology Corp., 4500 Bohannon Drive, Menlo Park, California 94025 USA, Septembre 1991.

[106] D. DE WITT et S. VANDENBERG, *Algebraic Support for Complex Objects with Arrays, Identity, and Inheritance*, (Computer Sciences Technical Report #987), University of Wisconsin−Madison, Décembre 1990.

Annexe A

Définition de ν et δ

La fonction ν permet d'associer à chaque nom une (ou plusieurs) référence dans l'espace d'instantiation. Quant à δ, elle calcule la taille des objets générés par un type. On suppose δ prédéfinie pour tous les types de base et la référence. De même, on suppose ν prédéfinie pour la racine des noms. On donne ici un exemple de définition de ces deux fonctions dans le cas de constructeurs "statiques".

Les n−uplets

Soit T = [a1: t1, a2: t2, ..., an: tn] et N un nom désignant un objet de type T :

$\delta(T) = \sum_{i=1}^{n} \delta(ti)$

Pour l'accès au champ ai, on a $\nu(N.ai) = \nu(N) + \sum_{j=1}^{i-1} \delta(ti)$

Les tableaux

Soit T = ta[S] et N un nom désignant un objet de type T :

$\delta(T) = \delta(ta) * S$

Pour l'accès au i élément, on a $\nu(N.i) = \nu(N) + (i\delta(ta) * (i - 1))$ si $i \neq 0$ et $\nu(N.0) = \nu(N)$

Les unions

Soit T = union [a1: t1, a2: t2, ..., an: tn] et N un nom désignant un objet de type T. Cet objet est un couple (s, v) où s est le sélecteur définissant le champ instancié de l'union et v la valeur de ce champ :

$\delta(T) = MAX(i\delta(ti))\ 1 \leq i \leq n$

Pour l'accès au sélecteur, on a $\nu(N.s) = \nu(N)$ et pour l'accès à la valeur, on a $\nu(N.v) = \nu(N) + \delta(i\tau(s))$ où $\tau(s)$ donne le type du sélecteur.

Chapitre I

Introduction

Chapitre II

Systèmes de types et bases de données

Chapitre III

Langages et bases de données

Chapitre IV

PEPLOM : un langage de programmation pour bases de données

Chapitre V

Gestion des définitions PEPLOM

Chapitre VI

Génération d'applications PEPLOM exécutables

Chapitre VII

Conclusion