50376
1992
97

USTL
FLANDRES ARTOIS

LIFL
U.A 369 du C.N.R.S

LABORATOIRE D'INFORMATIQUE FONDAMENTALE DE LILLE

THÈSE

présentée
en vue de l'obtention

du titre de DOCTEUR DE L'UNIVERSITE DE LILLE I
Label Européen
Spécialité : Informatique

par

Stéphane BRESSAN

BIBLIOTHEQUE UNIVERSITAIRE
SECTION DE SCIENCES
LILLE

Sujet de la thèse :

Représentation et Modélisation des Connaissances : Möbius, un Modèle Intégrant les Aspects Conceptuels et Déductifs

soutenue le vendredi 20 mars 1992 devant le jury composé de :

MM.	Vincent CORDONNIER	Président
	Stefano CERI	Rapporteur
	François RECHENMANN	"
	Joachim SCHMIDT	"
	Gérard COMYN	Directeur

50376
1992
97

Remerciements

Je tiens tout d'abord à remercier les membres du jury :

- Je remercie le professeur Vincent Cordonnier pour avoir accepté de présider ce jury et de juger cette thèse.
- Je remercie les professeurs Stefano Ceri et François Rechenmann de m'avoir fait l'honneur de s'intéresser à ce travail et de le juger.
- Je remercie le professeur Joachim Schmidt de m'avoir fait l'honneur de juger ce travail et de m'avoir acceuilli dans son laboratoire pour me faire part de ses commentaires et suggestions.
- Je remercie Gérard Comyn pour m'avoir guidé ces dernières années dans la réalisation de ce travail de thèse ; je lui témoigne ici mon amitié et ma reconnaissance.

Je remercie tous ceux qui m'ont accompagné pendant ces trois ans, mes collègues du Laboratoire d'Informatique Fondamentale de Lille et l'équipe des enseignants et secrétaires de l'Ecole Universitaire D'Ingénieurs de Lille.

Enfin, je remercie tous mes collègues de L'ECRC pour leur aide, leurs conseils et leur soutien, et tout particulièrement Mike Freeston qui m'a accueuilli au sein du groupe *Knowledge Bases*, Rainer Manthey qui a piloté ce travail, et mes amis Petra Bayer, Benoit Baurens et Alexandre Lefebvre.

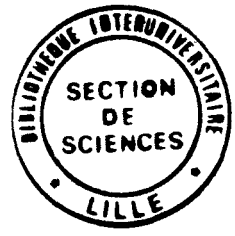

Table des matières

Table of Contents

Liste des figures

Chapitre 1

Introduction

"Sans la raison, la mémoire est incomplète et inefficace."
G. Bachelard, *la dialectique de la durée.*

1.1 Logique et Objets

Le nombre de propositions qui proclament avoir résolu l'intégration des approches logiques et orientées objet croît de jour en jour depuis une dizaine d'années [Glo89] [Alb85].

The number of proposals claiming they have solved the integration of logical and object-oriented approaches has steadily increased during the last ten years [Glo89] [Alb85].

Notre point de vue est un peu plus sceptique quant à la problématique et quant aux objectifs. Nous considérons en effet qu'il n'existe pas une solution unique car, de part et d'autre, les paradigmes objets et logiques sont des ensembles de concepts hétérogènes.

Our attitude is more skeptical to the problem and the objectives. Indeed, we consider that there does not exist a unique solution, as, on both sides, object-oriented and logical paradigms are heterogeneous sets of concepts.

Bien sûr, l'approche logique de l'informatique bénéficie d'un solide fondement mathématique. Cependant, elle ne constitue pas une démarche unique, ni en ce qui concerne les objectifs, ni en ce qui concerne la méthode.

Of course, the logical approach benefits by a solid mathematical foundation. However, it does not constitute a unique approach, neither as far as the objectives are concerned, nor as far as the method is concerned.

Par exemple, en partant de son représentant le plus fameux, le langage Prolog [GKP85], en allant vers les langages de la famille Datalog [CGT89], on voit apparaitre une grande variété de travaux sur le spectre qui s'étend de la programmation procédurale en logique à la représentation déclarative des connaissances. Par conséquent, toute tentative d'intégration doit préciser au préalable si elle se place dans un contexte de programmation ou de représentation des connaissances.

For instance, one can see a great variety of works on the spectrum lying between procedural programming languages like Prolog [CM81] and declarative knowledge representation languages like those of the Datalog family [CGT89]. Therefore, any attempt at integration must clarify beforehand whether it takes place in a programming or a knowledge representation context.

Quoi qu'il en soit, il convient de souligner que l'approche logique a aujourd'hui fait la démonstration de sa viabilité : des langages, des systèmes et des machines existent et sont comparables en efficacité (au sens le plus général du terme) aux langages, systèmes et machines plus traditionnels. Tout le monde se souvient de la spectaculaire adoption, par l'ICOT japonais, du langage Prolog pour son projet d'ordinateurs de la cinquième génération. Dans le même temps, aux Etats Unis, dans les laboratoires de recherche en intelligence artificielle, la programmation logique remplaçait la programmation fonctionnelle pour le prototypage. En Europe, la recherche en programmation logique et sur les bases de données déductives a régulièrement évolué.

Anyway we must underline that the logical approach has now proven its feasibility : languages, systems and machines are as efficient (in the general sense of the term) as more traditional languages, systems and machines. Everyone remembers the adoption of the Prolog language by the ICOT for the fifth generation computers project. Meanwhile, in the United States, in the artificial intelligence research laboratories, logic programming was replacing functional programming for prototyping. In Europe, the logic programming and deductive databases research steadily evolved.

Globalement, on peut aujourd'hui évaluer l'impact de cette recherche et de cette technologie par le nombre et l'importance des conférences internationales qui abordent ce thème : the International Logic Programming Symposium, the International Conference on Logic Programming, the conference on Principles of DataBase Systems, *etc. Pourtant, pour que le transfert technologique vers l'industrie soit possible, un effort ergonomique reste a faire. L'élégance et la puissance des approches logiques, leur formalisme, peuvent séduire ou rebuter l'utilisateur. Les notions manipulées semblent naturelles mais ne sont malheureusement pas toujours intuitives (le retour arrière en Prolog, la négation, les quantificateurs).*

Globally, one can evaluate the impact of this research and technology by the number of international conferences on the topic: *the International Logic Programing Symposium, the International Conference on Logic Programming, the conference on Principles of DataBase Systems,* etc. However, the technological transfer to industry is only possible if effort towards ergonomics is made. The élégance and power of logical approaches, their formalism, may attract or put off the user. The underlying notions seem to be natural, but they may not be intuitive (backtracking in Prolog, negation, quantifiers).

La situation du paradigme objet est quelque peu différente. Tout d'abord, les langages porte-drapeau de la programmation orientée objet, comme Smalltalk, ont rapidement proposé des environements de programmation complets et interactifs. Cela, non seulement démontrait les propriétés de ces outils (extensibilité, adaptabilité) mais encore permettait une diffusion immédiate de cette technologie. L'effet a été tel qu'être orienté objet était une condition nécessaire au succés tant pour les travaux de recherche que pour les systèmes commerciaux:

In the case of the object-oriented paradigm, things are a little different. First of all, the standard object-oriented languages, like Smalltalk, quickly offered complete and interactive programming environments. This has not only emphasized the properties of such tools (extensibility, adaptability) but it has also allowed a quick marketing of the technology. The effect was such that being object-oriented was a necessity for research works and commercial products:

> "If I were to sell [my cat], I would not stress that he is gentle to humans and is self-sufficient, living mostly on field mice. Rather, I would argue that he is object-oriented." Roger King, [KL89].

De cette situation il resulte une langue de bois et un amas de notions: classes, objets, messages, héritage, méthodes, etc.

From this situation remain buzz words and a mass of notions: classes, objects, messages, inheritance, methods, etc.

Sans entrer dans les querelles de paternité, il est aussi clair que la plupart des concepts mis en lumière par l'approche objet lui ont préexisté. Par exemple, en représentation des connaissances, dès les années 70, les travaux sur les réseaux sémantiques étudiaient les problèmes d'identité, d'agrégation, de généralisation, de classification et les inférences basées sur l'héritage.

Without entering the paternity quarrels, it is clear that most of the concepts highlighted by the object-oriented approach were already discovered before. For instance, in the early 70's, the work on semantic networks for knowledge representation studied the problems of identity, aggregation, generalization, classification and the inferences based on inheritance.

Nous retiendrons, pour notre part, de l'approche orientée objet la volonté d'étudier ces problèmes et leur intérêt en modélisation et représentation des connaissances.

We will retain from the object-oriented approach the aim to address these problems, and their interest for knowledge modeling and representation.

1.2 Systèmes de gestion des connaissances

La problèmatique que nous étudions dans cette thèse est l'intégration des concepts empruntés aux bases de données déductives et de concepts qualifiés d'orientés objets dans le cadre de la modélisation des connaissances.

The problem we address in this thesis is the integration of concepts borrowed from deductive databases together with concepts labeled as object-oriented in a knowledge modeling framework.

Nous avons, marginalement, réfléchi à d'autres aspects de cette intégration (langage de manipulation et de mise à jour des connaissances, architecture des systèmes de gestion des connaissances, etc) dans le cadre du développement de prototypes pour notre modèle. Nous voulons d'ores et déjà souligner la différence de nature qui existe, a priori, entre ces problèmes.

We have marginally investigated other aspects of this integration (manipulation and update language, knowledge management system architecture, etc) while developing prototypes for our model. We want to underline here the a priori difference between those problems.

Afin d'illustrer et de présenter notre problématique nous avons choisi de faire un catalogue raisonné des differents outils pour les bases de connaissances développés par le groupe Knowledge Bases *de l'ECRC, au sein duquel nous avons réalisé la partie concrète du travail exposé ici. Cette présentation nous permettra de montrer que le modèle que nous proposons se situe dans la continuité d'une recherche (cf. figure 1.1).*

As an illustration and a presentation of our problem, we have chosen to make a catalogue raisonné of the tools for knowledge bases developed by the ECRC knowledge bases group. The concrete part of the work presented here has been developed within this group. We want to show through this presentation that the model we propose is a natural continuation of previous research (cf. figure 1.1).

1.2.1 Prolog et les bases de données

Constatant le succés de la logique comme langage de programmation et devant la nécessité d'accroître la puissance des systèmes de gestion de bases de données relationnels, sur la base d'un fondement théorique commun, plusieurs systèmes ont été développés qui intégraient un SGBD relationnel dans un environement Prolog. A l'ECRC, trois systèmes furent successivement étudiés et implantés.

As a consequence of the success of logic as a programming language, faced with the necessity of increasing the power of relational database systems and taking advantage of the common theoretical background, several systems coupling Prolog with a database management system were developed. At ECRC, three systems were successively studied and implemented.

Le premier, EDUCE [BP88], realisait le couplage du système MU-Prolog avec le SGBD INGRES. Les accés à la base de données étaient réalisés par l'intermédiaire des canaux UNIX sur lesquels transitaient les requêtes en QUEL (le langage de requêtes de INGRES) et les réponses.

The EDUCE system [BP88] couples the MU-Prolog system with the INGRES DBMS. Data access are performed through the UNIX pipes on which queries in QUEL (the INGRES query language) and answers are transferred.

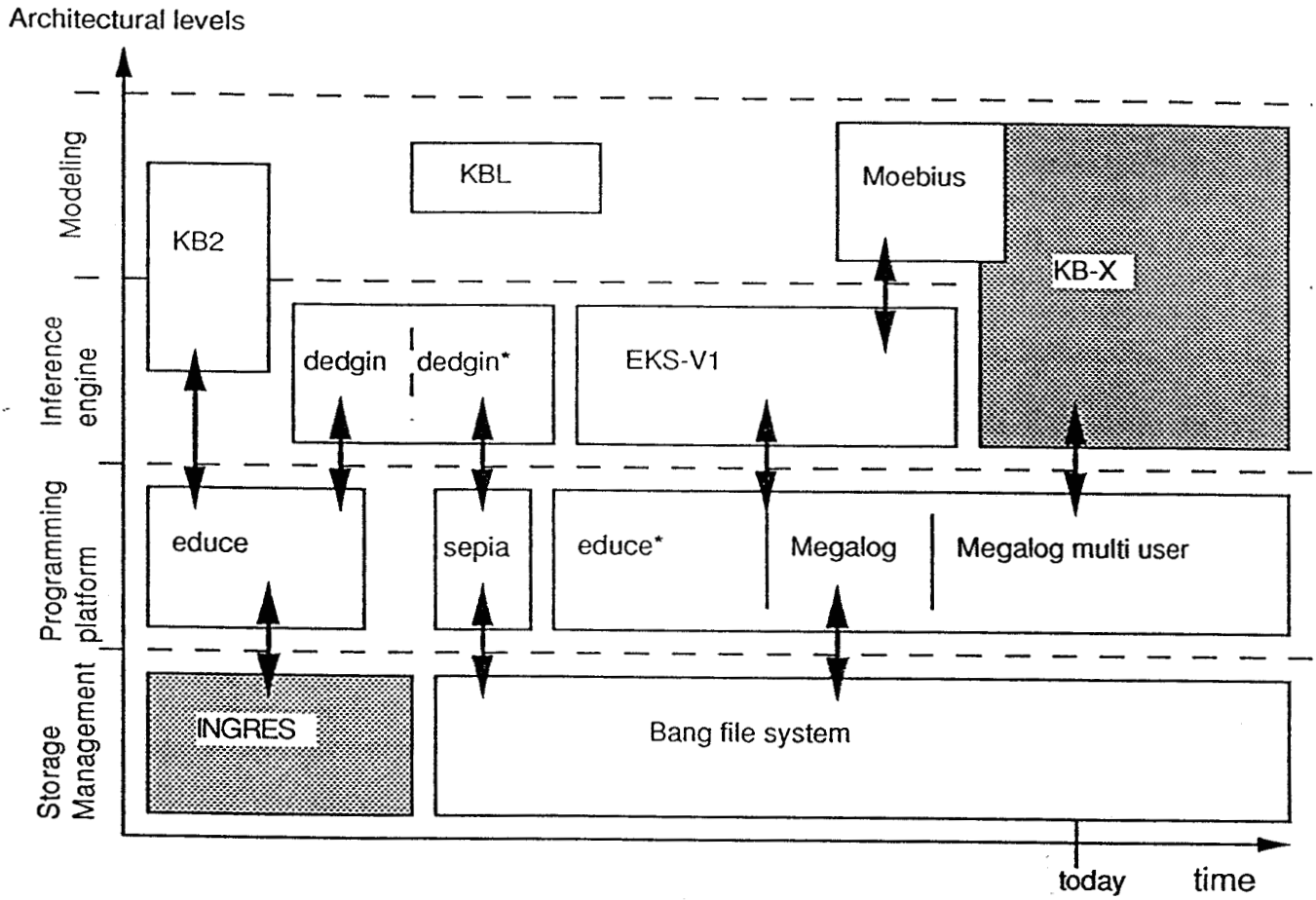

Figure 1.1: Knowledge Base Management Systems at ECRC

D'une part, la capacité du SGBD est augmentée par la puissance des mécanismes d'inférence de Prolog qui joue alors le rôle de langage de manipulation (et non de langage de requête). D'autre part, Prolog bénéficie d'un système de stockage et de récupération de données en mémoire secondaire. Il s'agit d'une première approximation d'une base de faits sur disque.

The DBMS capacity is augmented by the power of Prolog inference mechanisms. Prolog plays the role of a manipulation language (and not of a query language). Prolog benefits by a secondary storage and retrieval system. It is a first approximation to a Prolog database on disk.

Cependant, en toute généralité, les besoins de stockage de Prolog ne sont pas limité à une structure relationnelle en première forme normale. Il faudrait, aussi, pouvoir stocker des termes complexes et des clauses, comme c'est le cas dans la database *interne de la plupart des systèmes Prolog. C'est dans ce sens que vont les systèmes EDUCE* et Megalog [HBD89]. Tous les deux sont basés sur le système de gestion de fichiers BANG [Fre87] qui permet une gestion efficace de données éventuellement complexes (à noter qu'en présence de variables et de termes, l'algèbre relationnelle n'est plus disponible).*

However, the storage needs of Prolog are not limited to a relational structure in first normal form. It should be possible to store complex terms and clauses, as it is the case in the internal Prolog database in most Prolog systems. The EDUCE* and Megalog systems [HBD89] take this direction. They are both based on the BANG file management system [Fre87] allowing the storage and efficient retrieval of complex data (in the presence of terms and variables the relational algebra is no longer available).

Megalog, qui est l'aboutissemnet de ces travaux, constitue donc une plateforme de développement de systèmes de gestion de connaissances. En effet, il fournit:

Megalog, the most recent system, offers a platform for the development of knowledge management systems. Indeed, it provides:

- *un environement de programmation Prolog étendu avec des primitives de manipulations (accés et mises à jour) des données sur disque ;*
- *un système de gestion de base de données relationnel intégré ;*
- *un système de gestion de données complexes intégré.*

- A Prolog programming environment extended with primitives for manipulation of data on disk (access and updates);
- an integrated relational database management system;
- an integrated complex data management system.

1.2.2 Les bases de données déductives

Sur les supports de développement de systèmes de gestion de bases de connaissances précédement présentés, des systèmes de gestions de bases de données déductifs ont été prototypés. Comme nous le verrons ultérieurement, un système déductif est principalement composé d'un langage de description de vues complexes et d'un évaluateur complet pour ce langage. Ce type de langage est, en général, appelé Datalog (logique pour les données) et accompagné de quelques exposants précisant les extensions au langage de base (Datalog^{agg} pour les fonctions d'agrégation, Datalog$^{\neg}$ pour la négation, etc). A la base Datalog est un langage de clauses de Horn sans symbôles de fonctions.

Deductive database management systems prototypes have been developed on top of the platforms described above. As we will see later, a deductive system is mainly composed of a complex view description language and of an evaluator which is sound and complete for this language. This kind of language is called Datalog (Logic for Data) and may be accompanied by several exponents indicating the extensions to the basic definition (Datalog^{agg} for aggregation functions, Datalog$^{\neg}$ for negation, etc). Basically, Datalog is a language of Horn clauses without function symbols.

Ce type de langage est déclaratif, c'est à dire que sa syntaxe ne contient aucun élément de contrôle. Il ne sert pas à spécifier une évaluation mais à dénoter un ensemble de données. Par conséquents, un évaluateur du langage doit garantir la complétude du calcul : il doit produire toutes les données dénotées, pas plus, pas moins. La résolution SLD Prolog standard n'est pas complète même dans le cas de Datalog à cause des éventuels prédicats récursifs. Par conséquent, il convenait de developper d'autres mécanismes d'évaluation. Les méthodes d'évaluation de la famille QSQ [Vie86, Vie87] sont complètes pour Datalog (il n'existe pas d'évaluateur complet pour la logique du premier ordre).

This kind of language is declarative. Its syntax does not contain any control expression. It is not used to specify an evaluation but to denote a set of data. As a consequence, an evaluator for that language must guarantee the completeness of the result: it must answer all the denoted data, no more, no less. The standard Prolog SLD resolution is not complete even in the case of Datalog. This is due to the possible recursive predicates. It was therefore necessary to develop new evaluation mechanisms. The evaluation methods of the QSQ family [Vie86, Vie87] are complete for Datalog (there does not exist a complete evaluation method for full first order logic)

Ces méthodes d'évaluation constituent le coeur des systèmes Dedgin, Dedgin [LV90] et EKS-V1 [VBK+91b] qui sont les trois prototypes successifs de systèmes de gestion de base de données déductifs de l'ECRC. Le système le plus moderne, EKS-V1, intègre, en plus de l'évaluateur de requêtes, un gestionnaire de contraintes d'intégrité, un système de gestion de vues materialisées et un langage de manipulation et de mise à jour permettant des mises à jour conditionnelles et du raisonnement hypothétique.*

Such evaluation methods are the kernel of the Dedgin, Dedgin* [LV90] and EKS-V1 [VBK+91b] systems. These are the three successive prototypes of the ECRC deductive database management systems. The EKS-V1 system integrates, in addition to the query evaluator, an integrity constraints checker, a materialized view management system and a manipulation and update language allowing conditional updates and hypothetical reasoning

Une partie des expérimentations que nous avons réalisées pour le modèle que nous présentons dans cette thèse ont été implantées au dessus de EKS-V1.

We have implemented prototypes for the model we present in this thesis on top of the EKS-V1 system.

1.2.3 La modélisation des connaissances

Malgré leur puissance et leur expressivité, ces systèmes déductifs manquent d'outils permettant la modélisation de données et de connaissances. Toutes les formes de connaissance nécessaires à une application sont représentées dans le formalisme logique. Les notions naturelles d'agrégation, de généralisation et de spécialisation pour l'organisation générale des connaissances doivent être traduites, à la main, en terme de faits, de règles et de contraintes par le programmeur. La disparition de cette organisation explicite des connaissances, de cette méta connaissance, interdit son exploitation ultérieure (optimisation sémantique, réponses intentionnelles, etc).

Despite their power and expressivity, such deductive systems are lacking tools for data and knowledge modeling. All kinds of knowledge needed for an application are represented in the logical formalism. Natural notions, such as aggregation, generalization or specialization for the general organization of knowledge, must be translated by hand to facts, rules and constraints. The disappearance of an explicit form of this knowledge , this meta-knowledge, forbid its later exploitation (semantic optimization, intentional query answering, etc).

On voit donc apparaitre plusieurs directions de réflexion :

- *la définition d'un modèle de données et de connaissances intégrant les aspects d'organisation des connaissances et les aspects déductifs ;*
- *l'implantation d'un système de gestion de connaissances fondé sur ce modèle ;*
- *l'exploitation des informations sémantiques pour optimiser et enrichir le système de gestion.*

Several research directions appear:

- the definition of a data and knowledge model integrating knowledge organization and deductive aspects;
- the implementation of a knowledge management system based on this knowledge;
- the exploitation of the semantic information to optimize and enrich the management system.

Un premier système intégrant des outils de modélisation évolués dans un cadre déductif a été prototypé à l'ECRC sous le nom de KB2 [Wal86]. Il était directement implanté sur la plateforme de développement EDUCE. Le modèle de KB2 est une extension du modèle entité association à la classification. Il contient deux niveaux distincts d'information : les données et le schéma.

A first system integrating advanced modeling tools in a deductive framework has been prototyped in ECRC under the name KB2 [Wal86]. It was directly implemented on the platform EDUCE. The KB2 data model is an extension of the Entity-Relationship model to classification. It contains two separate levels of distinct information: the data and the schema.

Le principal problème mis en évidence par le développement d'un prototype pour KB2 est celui de la mise en correspondance entre les éléments du modèle et les structures de stockage (efficacité versus intégrité, cf. [Wal85]). Il nous semble que la définition et les résultats des évaluation de KB2 ont été influencés par la plateforme sur laquelle le prototype a été implanté.

The main problem emphasized by the development of a prototype for KB2 is the mapping of the model to a relational structure (efficiency versus integrity, cf. [Wal85]). It seems to us that the definition and the result of the evaluation of KB2 were influenced by the platform on which it has been implemented.

Par la suite, un groupe de travail a œuvré à la définition d'un langage de modélisation des connaissances. Le résultat de cette recherche est la proposition KBL [MKW89]. Les efforts ont porté sur les aspects linguistiques et ont permis de clarifier un grand nombre de concepts (en particulier la distinction entre les aspects et les constructions dénotationels et opérationnels). La nécessité de représenter des formes "méta" de la connaissance à aussi été soulignée, mais le modèle de KBL est basé sur un nombre fixe et fini de niveaux distincts. Seul un analyseur syntaxique du langage a été prototypé.

Later, a working group tried and defined a modeling language. The result of this research is the KBL proposal [MKW89]. This work has concentrated on the linguistic aspects and it has clarified a number of concepts (in particular the distinction between denotational and operational aspects and constructs). The necessity of explicitly representing meta-knowledge was also emphasized. However, the KBL model is based on a fixed number of distinct levels. Only a parser of the language has been implemented.

Il semble aujourd'hui nécessaire de reprendre ces réflexions dans le cadre global du développement d'un système de gestion des connaissances, à la lumière des possibilités ouvertes par l'évolution de la technologie déductive. Cela constitue une partie des objectifs immédiat du groupe Knowledge Bases *de l'ECRC (projet KB-X). Notre travail s'inscrit en préliminaire de cette recherche.*

It seems to be necessary to come back to these reflections in the global framework of the development of a knowledge management system, taking advantage of the new possibilities offered by the deductive technology. This is the current objective of the ECRC knowledge Bases group (KB-X project). Our work is a preliminary for this research.

1.3 Möbius

[1]

Möbius est tout d'abord un modèle sémantique de données. Il s'agit d'offrir, dans un contexte de base données, la possibilité d'organiser la connaissance en termes de classification et de déduction : un mariage des approches déductives et orientées objet pour la modélisation.

First of all, Möbius is a semantic data model. It offers, in a database context, the capability to organize knowledge in terms of classification and deduction: a marriage between deductive and object-oriented approaches for modeling.

[1] *Nous avons choisi le nom Möbius en référence aux rubans de Möbius qui sont des espaces topologiques n'ayant ni intérieur ni extérieur. Cette propriété est une allusion à l'architecture métacirculaire du modèle.*
We choose the name Möbius in reference to the Möbius strips. They are topological spaces without inside and outside. This property is an allusion to the metacircular architecture of the model.

Nos choix de base furent de nous inspirer des modèles de la famille entité-association et des modèles de réseaux sémantiques en les enrichissant de la dimension de classification. Nous avons donc considéré un modèle dans lequel les objets sont de simples noeuds dans un réseaux d'associations, de liens, qui leur donnent leur signification. Nous avons retenu trois notions de base : l'entité, l'attribut (lien entre deux entités) et la classe.

Our basic choices consisted in taking inspiration from the models of the Entity-Relationship family or from the semantic networks models, trying to add to them a classification dimension. We considered a model in which objects are simple nodes in a network of associations that give them their meaning. We retained three notions: the entity, the attribute (link between two entities) and the class.

Par ailleurs, nous souhaitions un modèle extensible et qui représente uniformément toutes les informations que le concepteur et l'utilisateur peuvent exprimer. C'est pour cette raison que nous sommes partis d'une définition métacirculaire du modèle. Tout est entité, y compris les attributs, et, par conséquent, la sémantique et les mécanismes applicables aux données sont aussi applicables aux éléments du schéma et de tous les méta niveaux nécessaires.

Moreover, we wished an extensible model representing uniformly all kinds of information the designer or the user may have expressed. For that reason, we started from a metacircular definition for the model. Everything is an entity, including attributes, and, therefore, the semantics and mechanisms for data are applicable for the schema and all the necessary meta-level components.

Le noyau métacirculaire est décrit par un ensemble de données interconnectées et définies en extension (faits), en intention (règles) et régies par un ensemble de contraintes (contraintes d'intégrité). Le noyau est principalement constitué de deux entités : `entity` *et* `class`*, respectivement la classe de toutes les entités du système et la classe de toutes les classes. L'extension du noyau est controlée par des contraintes qui garantissent sa sémantique.*

The metacircular kernel is described by a set of interconnected data defined in extension (facts), in intention (rules) and ruled by a set of constraints (integrity constraints). The kernel is mainly composed of two entities: `entity` and `class`, respectively the class of all entities in the system and the class of all classes. The kernel extension is controlled by the constraints guaranteeing its semantics.

En résumé, les principaux traits du modèle sont :

- *les entités ;*
- *les classes ;*
- *la hiérarchie de classe et d'instances ;*
- *les attributs et les associations ;*
- *les définitions en extension d'attributs ou de classes ;*
- *les définitions en intention d'attributs ou de classes ;*
- *l'héritage ;*
- *les contraintes.*

En fait, Möbius va offrir une dualité méthodologique puisqu'une application pourra être décrite soit du point de vue de la hiérarchie de classes (en associant les définitions d'attributs aux classes), soit du point de vue des définitions d'attributs (en associant des domaines, des classes, aux définitions d'attributs).

Le modèle est accompagné d'un langage déclaratif. Ce langage est utilisé pour définir des requètes, des vues ou bien des contraintes. La syntaxe est celle de Datalog. Elle comprend cependant des éléments supplémentaires qui permettent de controler l'héritage statiquement ou dynamiquement (nom complet ou vues et points de vue).

La sémantique du modèle est donnée par une traduction en logique du premier ordre. Les formules obtenues peuvent servir (nous le montrons dans nos expériences de prototypage) à l'implantation sur un système de gestion de base de données déductif lorsqu'elles sont transformées en règles et contraintes d'intégrité.

To summarize, the main features of the model are:

- entities;
- classes;
- the class and instances hierarchy;
- attributes and associations;
- attribute or classes extensional definitions;
- attribute or classes intentional definitions;
- inheritance;
- constraints.

In fact, Möbius offers a methodological duality. An application may be designed either from the point of view of the class hierarchy (in associating attribute definitions to classes) or from the point of view of attribute definitions (in associating domains (classes) with attribute definitions)

The declarative query language is used to define queries, views or constraints. Its syntax is basically the same as Datalog. It comprises supplementary elements which allow a static and dynamic control of inheritance (full name, or views and viewpoints)

The model semantics is given by a translation into first order logic. The formulae, transformed into rules and constraints, can be used (as we show in the prototyping experiments) for a direct implementation on a deductive database management system.

1.4 Présentation

Dans cette thèse nous nous sommes attachés à poser la problématique qui nous a amenés à réaliser ce travail, à présenter notre proposition pour un modèle conceptuel sémantique de connaissance, et à discuter nos expériences de réalisation de prototypes de systèmes de gestion de connaissances basés sur notre modèle.

In this thesis we analyse our problem, we present our proposition for a semantic conceptual knowlededge model and we discuss our experiments in the implementation of knowledge management systems prototypes based on our model.

Dans le chapitre 2, nous passons en revue et présentons les différentes approches de la représentation des connaissances qui ont influencé notre travail. Il s'agit des approches dites déductives des bases de données, de la modélisation et représentation des connaissances et des approches orientées objet des bases de données.

In chapter 2, we review and present the several approaches of knowledge representation that influenced our work. It is about deductive approaches to databases, knowledge representation and modeling and object-oriented approaches to databases.

Dans les chapitres 3, 4 et 5, nous présentons notre proposition pour un modèle de connaissances. Nous présentons successivement le noyau métacirculaire du modèle, les élements de base, attributs et classes d'attributs, les problèmes et solutions liés à l'héritage et ceux liés aux notions d'instance et d'identité d'objet.

In chapters 3, 4 and 5, we present our proposal for a knowledge model. We introduce successively the metacircular kernel of the model, the basic components, attributes and attribute classes, the problems and solutions related to inheritance and those related to the notions of instance and identity.

*Dans les chapitres 6 et 7, nous décrivons les expériences d'implantation et présentons nos réflexions sur la définition d'un système de gestion de connaissances fondé sur notre modèle. Nous discutons le problème de la mise en correspondance (*mapping*) du modèle avec les structures relationnelles de stockage. Nous étudions la définition et le prototypage d'un évaluateur pour notre langage déclaratif. Nous analysons les besoins pour un langage de manipulation et de mise à jour.*

In chapters 6 and 7, we describe the implementation experiments and present our ideas about the definition of a knowledge management system. We discuss the problem of mapping the model to relational storage structures. We study the definition and the prototyping of an evaluator for our declarative language. We analyse the prerequisites for a manipulation and update language.

Dans la conclusion, nous établissons un bilan du travail présenté et réalisé et nous envisageons les différentes perspectives ouvertes.

In the conclusion, we sum up the presented and realized work and we look at the different future prospects.

Chapitre 2

Knowledge-base Models and Systems

Modèles et systèmes pour les bases de connaissances

Dans ce chapitre, nous présentons les trois domaines à l'intersection desquels ce situe notre travail.

Il s'agit d'une part de présenter la technologie developpée pour les bases de données déductives. Nous en analysons les deux principales composantes, à savoir, la définition et l'évaluation de vues complexes éventuellement récursives et la définition et la gestion de contraintes d'intégrité.

D'autre part, nous passons en revue les principales contributions (dans le domaine de l'intelligence artifcielle et des bases de données) à la modélisation de données. Nous essayons de dégager les aspects essentiels d'un modèle sémantique de données. Notre analyse retient cinq notions:

- *la représentation et la gestion de l'intégrité référentielle ;*
- *l'organisation des données selon les trois axes:*
  - *agrégation ;*
  - *classification ;*
  - *généralisation ;*
- *la nécessité de représenter des données implicites.*

Enfin, nous étudions l'apport des approches dites "orientées objet" des bases de données à la définition d'un système de gestion des connaissances, ainsi que leur intégration dans un contexte déductif.

Ce chapitre réunit des travaux qui ont, d'une manière ou d'une autre, influencé ou motivé la définition du modèle et du système Möbius.

2.1 Deductive Databases

The first deductive database systems, relying on the common relational background shared by both relational databases and logic programming languages, aimed at combining the two paradigms [GMN84], [CGT90]. This had the advantage of adding persistent storage means to logic programming languages like Prolog and of providing an inference mechanism (deduction) for a database manipulation language. But today, because of the procedural nature of Prolog, such approaches should be classified in the specific paradigm of persistent programming languages. Such integrated or coupled systems are certainly offering adapted platforms for deductive databases systems development but are not themselves deductive databases (cf. e.g. [BP88], [CW86], [HBD89], etc).

Indeed, one should understand deductive databases as systems exploiting the declarative aspects of logic rather than its procedural aspects. In particular, such a definition comprises not only the capabilities of expressing complex views by means of rules and of computing complex queries expressed in first order logic, but also the capability to express any kind of knowledge in logic, for instance integrity constraints.

Today, deductive databases technology is effective. The prototypes are no more laboratory prototypes but first beta-versions of commercial systems [MNS+87], [PDR91], [TZ86]. The question is to investigate the field of specific applications and to use the feedback to improve and give a direction to the future developments [Tsu91].

An example of such a deductive database system prototype is the EKS-V1 system developed at ECRC [VBK+91b]. It is one of the only deductive database systems incorporating both a query evaluator for Datalog$^{\neg agg}$ and an efficient integrity constraints checker [VBK+91c]. The query evaluator implements a compilation based version of the QSQ algorithm [Vie87]. The constraints checker is based on an update propagation mechanism [VBK91a]. We used EKS-V1 as a basis for several experimental implementations of Möbius.

2.1.1 Complex Views

Deductive databases systems are a class of systems whose purpose is to extend the capacity of relational databases systems. Mainly, they allow the expression of complex views, possibly recursive, by means of logical rules. Traditional relational databases allow to store and retrieve data (tuples) thanks to a query language based on the relational calculus - for instance SQL-. The purpose of views is to express virtual relations, intentionally defined relations. Because of the strong relationship between relational calculus and first order logic, the query and the view language for a relational database can be a first order logic language. Let us consider, as an illustration, a database of parts used by a bicycle constructor (example 2.1.1).

Example 2.1.1 The bikes and their subparts are stored in a relation `part(part, subpart, quantity)` and the basic parts are stored in a relation `basicpart(part, cost)`. The data are illustrated by the tables below:

part	subpart	quantity
bike1	frontwheel-230	1
bike1	backwheel-239	1
bike2	backwheel-323	1
bike2	pedal-12	2
backwheel-239	tyre-24	1
...	...	...

basicpart	cost
pedal-10	10
pedal-11	12
tyre-24	5
...	...

Several relational queries can be asked, for instance in a SQL-like language. They are expressed using the relational operators: join, selection and projection. The following expression:

```
SELECT part FROM part WHERE part.subpart=pedal-12.
```

denotes the set of parts, bikes, having as a subpart the pedal number 12.

All such relational queries can be expressed by means of logical formulae and views can be expressed by means of rules. The above query would then have the form:

```
query(X) <- part(X,pedal-12,_).
```

The relations(or predicates) defined by such rules are called views. An important aspect of deductive databases is the easiness to define recursive views. In our example, we may want to compute the set of needed basic parts for building `bike1`. This would be denoted thanks to the recursive view[1] `subpart(part, subpart)`:

```
subpart(X,Y)<- part(X,Y,_).
subpart(X,Y) <- part(X,Z,_) and subpart(Z,Y).
```

and the query:

```
query(X) <- subpart(bike1, X) and basicpart(X, _).
```

Deductive databases also extend the view and query language of the database system to non monotonic features such as negation or aggregate functions. In the example, aggregate functions could be used to define views and queries to compute the total cost of a given bike (e.g. the canonical bill of materials example [Lef91b]).

As the logic programming languages, like Prolog, have demonstrated that logic could be used for both specification and computation, the idea came naturally to use logic as a language for data description. Such a first order logic based language for data is called Datalog. As opposed to Prolog, it has not a procedural semantics, it is not a procedural

[1]Recursive views where not expressible in the early version of SQL and relational database query languages. Now, most of them allow their expression and can handle their evaluation.

language. Datalog is a declarative language with a denotational semantics. A Datalog expression denotes a set of ground tuples with the only assumption that an eventual evaluation will be sound and complete (it will compute the exact set of answers, no more, no less). There is no evaluation algorithm for full first order logic that is sound and complete because first order logic is undecidable. However there exist useful subsets of first order logic that have this good property. Datalog, basically, is restricted to first order logic without function symbols. When limited to Horn clauses, a Datalog program has a unique minimal model which is computable (the set of Herbrand atoms is finite).

The challenge was also to extend Datalog to bigger classes of problems: for instances $\text{Datalog}^{\neg}$, $\text{Datalog}^{\neg strat}$, Datalog^{f}, Datalog^{agg} respectively extend Datalog to negation, stratified negation, function symbols and aggregate functions. Two semantic approaches can be taken for Datalog and its extensions: the model theoretic semantics and the proof theoretic semantics. In some cases the model theoretic approach is weaker as there does not exist a minimal model. The proof theoretic approach has also the advantage to give hints for the definition of evaluation algorithms. Such algorithms are usually based on a fixpoint computation.

Several methods have been developed to evaluate Datalog expressions, some of them have been extended to Datalog*. The main method are the Magic Sets methods (see also the Alexander method), the OLDT method and the QSQ method.

The Magic Sets method ([BMSU86]) is said to be bottom-up. Each rule or query is rewritten into a rule producing the answers by a forward chaining from the data in the base. The Magic rules are such that they allow to push the selections with the original rules, therefore avoiding redundant computation, as opposed to the simplest method, called the naïve method, which produces the answers by a forward chaining on the initial rules.

The OLDT method (Ordered Linear resolution for Definite clauses with Tabulation, [TS86]) is a top-down method. It uses the horn clauses in a backward chaining à la Prolog. The idea, to avoid infinite loops due to recursion in the proof tree construction, was to store the subqueries and answers in a table. The OLDT resolution can avoid to expand already stored subqueries while using the produced answers.

The QSQ method, Query/SubQuery ([Vie86], [Vie87]), is similar but does not assume any order among the literals in a rule body or a query. QSQ allows a local selection function.

The three methods need a fixpoint computation. In fact, although the top-down and bottom-up approaches look very different [Ull89], it has been shown that they are equivalent [Bry90] even for the each computation step [Sek89]. For instance, the bottom-up rules of the magic set method simulate the backward chaining on the original rules. In general terms, the two goals of a query evaluation technique are to guarantee termination (in the case of recursion) and to avoid redundant computation. This can be achieved either by rewriting or by memoization techniques.

Thus a deductive system is, at least, a query evaluator for a view and query language on top of a relational database. Deductive database architectures differ whether they

are based on a loose or a tight-coupling with the relational database system. A loose-coupled deductive system will generate queries for the relational database in the query language and send them to the relational database. A tight coupling architecture, on the contrary, achieves a stronger integration of the two systems. The deductive engine can retrieve information from the disk. It has the advantage that no time is wasted in communication and that no redundant optimization is done by the query evaluator of the relational system. Loose coupled architectures are easier to implement. They only require the development of an interface between two, a priori, existing systems. A tight coupled architecture requires the redevelopment of, at least, a part of the database system, the code of commercial database being generally not in the public domain.

2.1.2 Integrity Constraints

Integrity constraints on a knowledge base are affirmations, according to certain semantic conditions arising from the domain of the application, that define the valid states and evolutions of the knowledge base (cf. [BDM88]). Constraints on states are called static constraints. Constraints on the evolution, refering to several states and to the temporal order among them, are called dynamic constraints. For instance, statements like:

"The salary of every employee should not be lower than the legal minimum salary"

or

"The salary of every employee should not decrease"

are, respectively, static and dynamic constraints for the knowledge base of a company.

Dynamic constraints have been less studied than static ones, in particular in the context of deductive databases. Their complexity is higher since they deal with several states of the knowledge base and the temporal order among them. The Möbius system makes use of only static constraints and, therefore, in the following, we will develop only this notion.

The first kind of constraints studied by the relational database community was related to the problem of referential integrity posed by the value-oriented nature of the relational representations. Namely, maintaining the identity of data defined by means of values grouped into relations leads to the definition of functional dependencies. For instance in a database of persons, functional dependencies allow to state that a person is uniquely identified by its name and surname. If a relation `R` has the attributes `Name`, `Surname` and `Person`, this functional dependency is represented as follow:

$$Name, Surname \longrightarrow Person$$

In general, for a relation schema $R(A_1, A_2, ..., A_n)$ and X, Y subsets of the set of attributes $\{A_1, A_2, ..., A_n\}$, there is a functional dependency:

$$X \longrightarrow Y$$

if and only if, for x a value on X and y and y' values on Y:

$$R(x,y) \text{ and } R(x,y') \Rightarrow y = y'$$

In other words, there corresponds at most a unique value of Y to a value of X.

Other dependencies have been defined: multi-valued dependencies, inclusion dependencies. All those dependencies, together with the notion of key, were used to define the several normal forms of a relational schema.

However, first order logic (or Datalog and its extensions) is a more general language for expressing constraints. All the above cited constraints can be translated into Datalog formulae. For instance a functional dependency can be as well expressed by the formula:

$$\forall XYY'((R(X,Y) \wedge R(X,Y')) \Rightarrow Y = Y')$$

Thus integrity constraints are encoded as Datalog formulae. A supplementary and reasonable condition is that these formulae are range-restricted. I.e., the variables occurring in the formulae, free or quantified, must range over a domain defined by an expression. Intuitively, a range defines the possible ground substitutions for the variables. This ensures that each formula always denotes a set of ground tuples without intervention of the Herbrand universe. Let us give examples of integrity constraints expressed by range-restricted Datalog and Datalog* formulae:

- "each department must have a leader"

  $\forall X \exists Y(dep(X) \Rightarrow leader(X,Y))$

- "an employee of a research centre is either a researcher or an administrator"

  $\forall X(employee(X) \Rightarrow (researcher(X) \vee admin(X)))$

- "group and team leaders cannot be the same person"

  $\forall XY(\neg groupleader(X) \vee \neg teamleader(X))$

- "an employee cannot work for more than 3 projects"

  $\forall XYI(employee(X) \Rightarrow (count([Y], workfor(X,Y), I) \wedge I < 4))$[2]

Thus the query language of deductive databases constitutes a substantively rich language for expressing integrity constraints. What for?

First of all, when designing the schema of an application, the designer can express the integrity constraints among the different components. Such constraints can be used

[2] $count([Y], workfor(X,Y), I)$ stands for the aggregation function

automatically or manually to design the relational schema: they can be transformed into structural constraints. E.g., if a group is always lead by exactly one group leader and a leader leads exactly one group, the two data can be stored in the same relation. This point of view is the point of view of normalization. More generally, it indicates that there is a strong relationship between the several components of the knowledge base: facts, deduction rules and constraints. Choosing the schema of the base relations and defining the rules can guaranty the validity of some integrity constraints.

This is particularly clear between rules and constraints. They are both means to express knowledge. It is not always clear whether a formula should be considered as a rule or an integrity constraint. Let us compare the following rule and constraint in the EKS-V1 syntax:

- rule:

```
name(X, N) <- single(X) and father(X, Y) and name(Y, N)
```

- constraint:

```
forall [X Y N]:
        single(X) and father(X, Y) and name(Y ,N) -> name(X, N)
```

They are both the same formula expressing that singles have the same name as their father. The difference stands only if one thinks of a rule as a means to answer a query and of a constraint as a statement to be verified. However both rules and constraints are part of the knowledge. Recent researchs studied the possible interactions among these two kinds of knowledge. Indeed, constraints can be used to optimize the query answering or even to give generic answers to queries. If a constraint states that nobody is both a group and a team leader you can immediately answer negatively the query: `groupleader(X) and teamleader(X)`. If one knows that singles have the same name as their fathers, one can, at least, answer `single(X)` to the query: `name(X, N) and father(X, Y) and name(Y, N)`, instead of giving the exhaustive list of singles which is probably less expressive. The two above cited possibilities are respectively called semantic optimization[CGM87] [CGM90] and intentional query answering [PR89]. In general the situation is that constraints are pieces of knowledge that should be used for query answering.

An interesting point to underline is the eventuality of an inconsistent state of the knowledge base because of the constraints and rules among them. In particular two constraints together may be unsatisfiable. This is a problem that suffers from a theoretical limitation since satisfiability of first order formulae has been showed to be undecidable.

Last but not least, the main use of integrity constraints is to guaranty the validity of a knowledge base state. Any transaction, or single update, modifies the model of the database. The question is: is the new knowledge base consistent with regards to the integrity constraints? Here constraints can be treated as "yes/no" queries in the query language. If the query is answered by "no" (or its negation by "yes"), the constraint is violated. The knowledge base is inconsistent. When a new constraint is stated the entire database much be searched to verify the integrity. Over a database of 10000

employees, if one states that every employee must have a reasonable salary, this has to be checked for the 10000 in any way. However, when assumed that the state before a transaction is valid, one could think of propagating a minimum set of information susceptible to influence the constraints. For instance, when adding a new employee, the constraint about the salary could be checked for this single new employee. The general case is not always as simple, but the principle is there. A means of efficiently evaluating integrity constraints is to compile them into propagation rules fired by the updates. The technique of propagation rules and so called generated predicates consists in maintaining the extension of some intentionally defined predicates for which one expects or knows that the time of their computation is worth being exchanged against the space they consume. Then one creates propagation rules minimizing the computation of their extension after a transaction. For each integrity constraint, one can consider the negation of the constraint as being an intensionally defined predicate:

```
violate_ic <- not ic
```

As it can be very expensive to recompute such predicates after each update and as maintaining their extension is very cheap since one expects it to be empty (no tuple should violate the constraint), they are treated as generated (materialized) predicates. This is the actual way integrity constraint checking is implemented in EKS-V1 (cf. [VBK91a]).

2.2 Object-orientation

2.2.1 Data Models and Knowledge Representation Systems

Initially developed in parallel, artificial intelligence and database research met in the mid-1970's to fill the gap of modeling techniques. The weakness of early data models, hierarchical, network and even relational model, called an insight on modeling tools. Artificial intelligence is concerned with the knowledge representation, manipulation and reasoning. The artificial intelligence models are based on the cognitive and physiological psychological studies. The conceptual models, addressing the problem of the representation of knowledge, strongly influenced the database community.

The role of a knowledge base management system (also called information system) is to reflect, partially, the reality of a given application domain. The manipulation language links the system to the world, mainly allowing interactions modifying the system's knowledge. The data model is the frame of the representation of any kind of knowledge about the world. The representation will probably corrupt the reality:

- the model is never as general as wished and may not be able to represent all kinds and forms of knowledge;
- the representation may voluntarily omit irrelevant information;

- the representation may change the nature of knowledge.

For instance, in the simple data model of a deductive database, one can choose to represent knowledge through rules and facts that do not directly correspond to laws and observations of the reality. The choice of the distorsion of the reality is under the designer responsibility. Therefore, the data model, and in general the knowledge system, must provide a variety of representations as rich and close to the vision of the world in the designer mind as possible. A data model should allow the designer to express the constraints over the real world and to represent the structure of information and the way he views it. A data model should provide tools to express structures, views and constraints. It would then be called a semantic data model.

Relying on psychological studies, the conceptual approach to knowledge representation is based on the assumption that the world should be modeled in terms of objects, structured, interrelated and classified, since it seems to fit with our mental representations ([Sow84]). Connexionists would argue that knowledge, structures and reasoning, are in fact encoded at a lower level: the level of the network of neurons where knowledge is distributed and results of the network configuration. Whatever the mechanisms of our brains are, the conceptual and symbolic approaches to knowledge representation have proved their utility and validity. The richness of the data model with respects to such psychological criteria is however important. Indeed the data model will serve as a user interface: its role will be to make the user aware of the organization of the knowledge he deals with. For that reason, sophisticated graphical and interactive interfaces are often associated with data models. The model can often give, at least, as much information as the application data themselves. For instance, when looking for a reference in a library database, the organization of books and publications browsed during the search may tell more about the topic than the contents of the paper itself.

What are the features of a semantic conceptual data model?

First of all, the model must respect as much as possible the one to one correspondence between entities in the world and objects in the model. An entity should not be modeled by more than one object as it is done, for instance, in relational models when an entity, represented by the value of its attributes, is split into several tuples. A violation of such a principle may not only lead to a deplorable discordance between the model and the reality but also bring technical anomalies such as update anomalies in relational databases. Symmetrically, an object in the model should not represent more than one entity. For instance, classes in object-oriented models are either abstractions or sets of their instances. They should not be used as a representation of each instance[3]. It is often tempting to build the object identity (the one to one correspondence) from a subset of the associations describing the entity structure: a person is known from its name and surname. There may exist homonyms breaking the referential integrity. We will come back on this point in chapter 5.

Considering objects representing entities, the data model must support the organization of these objects along three dimensions:

[3]cf. the class variables in some object-oriented languages.

- aggregation, association;
- classification;
- generalization.

Entities are not identified by their interrelationship but they are understood through them. In the first dimension, named links relate objects, concrete or abstract, to other objects they are composed of (aggregation) or in relation with (association). The difference between aggregation and association is not always clear. It depends on the way the designer or the user views the world. We will not argue on this difference. If needed, it can be complementary to the basic solution of having only association. Indeed, attributes can be seen as constrained binary associations. A syntactic difference is provided in Möbius.

Several objects sharing the same general structure could be abstracted,classified, as well as their description, in a new object. Such a general mould for objects is usually referred to as a class. This mechanism seems to correspond to a natural way for organizing knowledge. An object in the system is associated to a class and can be structured or linked to the other objects according to the available description of its structure by the class. This second dimension is thus associated with the inheritance inference mechanism: an object inherit its structure from the class it is an instance of.

Classes being general descriptions and a support for conception of the knowledge schema, their organization in a generalization-specialization hierarchy constitutes the third dimension for knowledge structuring. Brachman in [Bra83], discussed several interpretations of this class-subclass classification. The main problem is the semantics of the inheritance mechanism exploiting the class hierarchy. Namely one must define how descriptions given in the more general classes are available for the instances of the less general classes. As long as the difficult problems of exceptions, redefinitions, overriding and multiple inheritance are not considered, inheritance of descriptions by instances is equivalent to inheritance in the class hierarchy. The principle is: if a class `C1` is a subclass of a class `C2`, then all the instances of `C1` benefit by the descriptions given in both classes.

The early data models emphasized in different manners the several aspects discussed above. The Entity-Relationship (ER) model of P. Chen [Che76] suggests a description of a database schema in terms of structured entities (entities with attributes), values (integers, strings, etc), grouped into sets and associations among them. Chen did not consider the organization of entity sets in a hierarchy. The ER model has been extended in several ways. The Extended Entity-Relationship (EER) model, for instance, adds complex attributes (built from values sets using Cartesian product or set constructors) and a hierarchical organization of entity sets. At the same time, all the work initiated by Quillian's semantic networks [Fin79] led to the proposal of Roussopoulos (in [MB88]) to use semantic networks as the support of data modeling and can be related to models of the semantic binary network [Abr74] family. Of course, Minsky's work on frames [Min87] had also an important impact on data modeling since it addresses the problems of aggregation and classification. The Semantic Hierarchical Model of Smith and Smith

(in [MB88]) already summed up the requirements we have listed above and proposed them as an extension of the relational model.

Until this point, the only kind of inference we have talked about is inheritance. The general representation of derived data has not been only studied by the deductive database community. The data definition and manipulation language Daplex (in [MB88]) is a very important and influencing contribution in that domain. Relying on the functional interpretation of attributes (object structure), the Daplex model offers, in a class hierarchy, the possibility to describe the structure of entities by means of extensionally or intentionally defined attributes (functions). I.e., an attribute is given either as a function storing the list of its values or as one whose evaluation computes the values from other attributes. Möbius is in the list of modern data models that have revisited the Daplex proposal replacing the functional point of view by the deductive one.

Finally, we want to quote a number of contributions, strongly influenced by the artificial intelligence approach, which insist on a reflective definition of the data model. When manipulating classes or meta-classes, one may want to take advantage of the same modeling and manipulation capabilities as those provided for the application data. A first solution consists in having a finite number of separate conceptual levels (data, schema, meta-schema,...) and in reproducing the modeling environment to achieve uniformity. This leads to a rigid system since the number of level is fixed and the communication among the different levels is limited. The alternative is a reflective definition of the model. In that case any object, class, meta-class or application object, is defined in the same, unique model. The model kernel itself is self-defined. Reflexivity was mainly studied in the artificial intelligence community [MN88]. Artificial intelligence knowledge systems (e.g. [Fer84], [Cas86]) tried this approach. It is also strongly related to the meta-programming techniques developed in the field of logic programing [Ven84]. A few data models have a reflective definition: for instance the Taxis model [NCL+87] or the Telos model [MBJK90]. We discuss this approach in the chapter 3.

As a summary, the table below lists the main models and systems that had an influence on the definition of Möbius:

year	name	reference
1974	Semantic Binary Model	[Abr74]
1976	Semantic Network Model	in [MB88]
1976	Entity-Relationship model	[Che76]
1977	Semantic Hierarchical Model	in [MB88]
1981	Daplex	in [MB88]
1984	Taxis	[Nix87]
1981	Mering (AI)	[Fer83]
1981	Lore (AI)	[Cas86]
1989	Telos	[MBJK90]

2.2.2 Object-oriented Databases

From the lack of structuring and orthogonality of the relational model on the one hand, and from the influence of the object-oriented programming languages on the other, a new kind of databases has emerged during the last years: object-oriented databases (e.g. [Ba88]). From both the programming and the modeling points of view, object orientation emphasizes features that are interesting for the database systems and models. The initial lack of theory for objects makes more difficult the definition of what an object-oriented database system is. Nevertheless, following the common point of view of both Object Oriented Database [Aa90] and Third-Generation Database system [fADF90] Manifestos, a definition can retain some required features. An object-oriented database system must implement:

- complex objects and an algebra for their manipulation;
- object identity;
- classes (or types) organized in a hierarchy together with an inheritance mechanism.

Indeed, research on object-oriented databases was motivated by the lack of structuring offered by the relational models and systems at both modeling and programming levels. Complex objects are a means to overcome the rigid structure of tuples and algebraic terms. Complex objects are built from elementary types: integers, strings, etc, using constructors such as the set constructor or Cartesian products. Complex objects constructors, as opposed to relational ones, are orthogonal: they can be applied recursively on any complex objects.

Where complex objects are the first step to keep the structure of the real world entities in the database, object identity is the means to reified the real world entities. In the relational models, entities existence depends on the entity values: they are said to be value-oriented. Object identity allows entities to be independent from their state and values. Entities of the application exist in time and space regardless to their relationships to other entities. Object identity solves many of the problems posed by referential integrity constraints, in particular, the update anomalies studied by the normalization techniques in relational databases.

In [Abi90], Abiteboul proposes a definition of an object as a triple: (identifier, type, value). As the value is no more characteristic of what an entity is, its attachment to a class or its type defines its general structure. All person have names, surnames and a date of birth, they may have an address. It seems that the conception of an application schema is eased by the support of classes. Organization of schema information in a generalization, specialization hierarchy seems to be natural. The classes or types are organized in such a hierarchy exploited by an inheritance mechanism: students are persons, they have teachers and inherits from the characteristics of the persons.

Another point emphasized by the object-oriented approach is the management of objects behaviours, namely, the possibility to associate manipulations to objects collections. Writing methods as functions associated to classes is the classical concretization of this

concept. Together with encapsulation and inheritance, such a technique has a lot of advantages from the methodological point of view. It ensures code re-usability and forces independence of the different pieces of code.

2.2.3 Deductive Databases and Object-orientation: the Engagement

[4]

Both approaches aim at extending the capabilities of relational databases: adding view mechanisms and exploiting the power of logic for update languages, integrity constraints checking etc, adding entity structuring and a support for conceptual modeling. Is the integration possible? The main problem remains the opposition between a value based approach and an identity based approach. The recent proposition for combining objects and logic for databases have focussed on this problem of identity. There exists a number of articles dealing with this question [Zan89], [KC86], [AK89]. However, the absence of a general theory for logic and objects made the task even more difficult. Today, there is a family of proposal for such a theory for complex objects. All these proposals refer to the previous work, presented in [AKN86], aiming at combining object-oriented features, inheritance and structuring in Prolog by extending the terms structure. Ψterms are a generalization of functional terms to labelled record structures. Ψterms are built from a lattice of basic types and a set of labels. Thus each Ψterm is itself a type and the set of all Ψterms has an induced lattice structure.

An example of Ψterm is:

```
person(name =>"jean":string,
       date_of_birth => date_of_birth(day => 10:integer,
                                      month => december:month,
                                      year => 1967:integer))
```

The central idea of Ψterms was to replace the unification on algebraic terms by the computation of a lower upper bound in the Ψterms lattice. Indeed, such an extended unification simulates some kind of inheritance on structured terms. However, Ψterms are abstractions, types, and could not be straightforwardly related to the database problems where the main concerns are not only types but also instances. So, on the one hand Ψterms have been extended to handle more complex classification problems à la KL-ONE [BS85]: feature-terms and structured types, and on the other hand they inspired the database community to develop languages for complex objects in a logical framework. The first proposal in this direction was the one by Maier, revisited by Kifer, called O-Logic [Mai86], [KW89]. There has been several extension and variation on that theme: F-Logic [KLW90] and C-Logic [CW89] in particular. We illustrate this family in discussing shortly C-Logic. C-Logic is a logical language based on structured terms à la Login. C-Logic constructors include types organized in a finite partially

[4]Inspired by the article title: "Object orientation and logic programming for databases: a season flirt or long term marriage?" [CCT90]

ordered hierarchy, labels (attributes) variables, constants and functional terms. C-Logic terms and formulae have a denotational semantics given by a direct mapping to first order logic. This has the advantage to reduce the semantics of C-Logic to a known one and to allow the computation of C-Logic programs (when they are limited to definite clauses) to the computation of their translation in first order logic (which is then also limited to definite clauses). The authors argue that the translation may introduce unnecessary redundancies in loosing some indications that could be useful for a query evaluator (but not compromising the completeness of the result). They suggest that query evaluation technique should be adapted to work directly on C-Logic programs and benefit by the possible optimizations. A key point of C-Logic is the translation of the type hierarchy into a set of definite clauses among unary predicates representing the types: for instance if two types `T1` and `T2` are such that `T1 < T2` then the semantics is given by the existence of two unary predicates `PT1` and `PT2` and a rule `PT2(X):- PT1(X)`. C-Logic has the advantage to emphasize the possibility to manipulate complex objects and a form of inheritance in a simple context. The problematic notions of sets and 'oids' find an interesting representation and semantics in C-Logic. Sets are implemented by multi valued functions (labels) that can have alternatively, a multi valued or a set interpretation. Problems linked to the invention of 'oids' are solved by the use of terms interpreted as Skolem functions.

In general the O-, C- and F-logic approaches have the advantage of a strong theoretical basis. Their first order semantics link them to the existing query evaluation techniques. However the data modeling capabilities are still a bit raw. Problems like overriding are not handled and there is no rich and uniform context for the schema manipulation (except for F-logic which has higher order constructs). The same judgment applies to other attempts to describe a logical language for complex objects [TZ86],[AV88], [Abi90], [AB91].

Chapitre 3

The Möbius Model

Le modèle Möbius

Möbius est un modèle de connaissances fondé sur un noyau métacirculaire. Afin de réaliser les trois principes empruntés au modèle OBJVLISP (cf. [Coi87]):

- *tout est entité ;*
- *Toute entité est instance d'une classe ;*
- *toute classe peut être sous classe d'une autre classe ;*

le noyau de Möbius contient deux entités de base: `entity` *et* `classe`*, respectivement, la classe de toutes les entités du modèle et la classe de toutes les classes. Les liens sémantiques* `sc`*, lien de spécialisation, et* `isa`*, lien d'instance, sont établis entre ces deux éléments. Leur sémantique est donnée par des règles logiques et garantie par des contraintes d'integrité.*

L'intérêt de l'approche métacirculaire est multiple. Pour nous, ce choix fut d'abord un guide méthodologique pour la définition du modèle. Le bénéfice pour l'utilisateur est que tous les niveaux d'informations sont accessibles : entités de l'application, classes et méta-classes. Nous montrons comment, à l'aide de contraintes d'intégrité, le modèle peut être, a posteriori et si nécessaire, organisé en niveaux.

Möbius peut être vu, présenté et utilisé de manières duales. D'une part, c'est un modèle de classes, hiérarchie de classes et d'instances qui doit servir de support de conception. D'autre part, c'est un modèle d'entité reliées entre elles par des liens élémentaires (attributs), et de définitions génériques de ces liens (classes d'attributs). De ce dernier point de vue, Möbius se rapproche des modèles du type Entité-Association de Chen [Che76] et des modèles de réseaux sémantiques, in *[MB88].*

La sémantique du modèle et, en particulier de l'héritage, est donnée par traduction des différents composants en formules de la logique du premier ordre.

3.1 Metacircular Definition

3.1.1 Reflectivity and Uniformity

The uniformity principle, emphasized by most of the object-oriented systems and languages (e.g. [Gol85], [Coi87]) has several advantages. Adopting this principle for the definition of a knowledge representation model is, at least, a methodological guide. For each new concept to be introduced in the model (when defining it) one must take care to realize uniformity. If the new concept does not fit in the model definition, it is an indication sign that either the concept is wrongly formulated or that the initial notions are not sufficient.

A natural way to realize uniformity is to try and find a reflective definition for the model [Coi87], [Fer84],[Mae87b]. A reflective model is a model that, at least partly, represents - understands - its own structure. The model definition is reflectively given in terms of the model itself. In terms of a computational system, the set of self representing data can be queried in the same way as the application data: the system knows about itself. The set of reflective data can be modified or extended: the system can evolve. It is said to be causally connected to itself [Mae87a].

A reflective definition consists of a minimal set of concepts that need each other to be understood and that found the whole model. From a theoretical point of view, it has the advantage of simplicity. From a practical point of view, reflectivity allows extensibility. A reflective system is an open system in the sense that it can be enriched and adapted by any user. This dynamic aspect makes it a particularly interesting tool for experimentation.

This approach has also several drawbacks. First of all the definition of a reflective kernel is a difficult task. It is hard to have a simple overview of the whole model without adopting a simplified point of view. This is summed up by Goldberg's quotation about the Smalltalk documentation :

> "So it is almost the reader must know everything before knowing anything".

Just like writing recursive functions or procedures in programming languages, defining a reflective model leads to short descriptions that may be inexplicit and hardly understandable. This is the price to pay to get the properties of reflective models. However, even though the definition process is more difficult, it forces a deeper insight into the fundamental notions.

Another point, often criticized, is the extreme consequence of uniformity: conceptual differences are no longer supported by explicit differences in the model. As a canonical example, in object-oriented systems, a simple operation like adding two numbers must be supported by the general framework of message passing. Adding `one` to `two` to compute `three` becomes sending a message to the object `one` to tell that it must cooperate with the object `two` to compute a third object `three`. However, the extensibility of these systems offers the capability to recreate such conceptual differences by customizing the

interface. The message passing form for numbers and operations on numbers can be used with the natural syntax: $1 + 2$. In our particular case, as we will see below, we will have to find a characterization of the different levels in a knowledge base (instances, schema and meta-information) as the metacircular definition has, apparently, flatten them.

A more crucial problem is the difficulty to ally uniformity and reflectivity with efficiency. On the one hand, treating every concept in the same manner is, a priori, in contradiction with special purpose optimizations. Of course, one can argue that, precisely thanks to reflectivity, the system, knowing about itself, can parameterize its treatments and intelligently behave. However, a reflective system is fragile: any modification can break the kernel integrity. Even extensions - possibly overriding the kernel definitions - can break its consistency. Optimizations can only be made through a kind of compilation stage where the user ensures that the compiled information is definitely safe. Unless this insurance is given, the system must permanently spend time verifying its own integrity. Clearly, several reflective systems make compromises with respects to that point. As an example, most of the reflective class and object systems assume that once objects are created their instantiation class will not be modified (by static inheritance of instance variables).

Traditionally, conceptual data models for knowledge bases have a fixed finite number of separate levels. This is the case, for instance, in the KB2 [WKT88] and KBL [MKW89] models. At most they have an instance level, a schema level, and possibly a meta-level. In terms of classes and objects, the instance level is the level of the application objects, the schema level contains the classes describing the general form and behaviour of the instances and the meta-level is a limited set of meta-classes used to generate the schema. Experiences with object-oriented languages and knowledge representation systems have shown the importance of being able to have supplementary levels. In particular, the modularity of some applications is more naturally expressed when one can create meta-classes and meta-meta-classes etc. The metacircular schema we choose, inspired from the OBJVLISP model [Coi87], does not make any restriction with respect to that point and allows an infinite creation of levels.

3.1.2 The Metacircular Kernel

The model is based on three principles:

- everything is an entity;
- every entity is representative of, at least, one class;
- a class may be a subclass of another class.

In order to fulfill these principles we start from a schema containing two entities: `entity` and `class`. They will, respectively, represent the class of all entities in the model and the class of all classes. The vertical relations [EJ90], the direct instance link `isa_d`, linking an entity with the class it directly represents, and the direct superclass link `sc_d`,

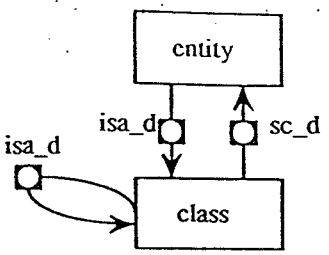

Figure 3.1: Metacircular Kernel

linking a class and its direct superclass, are established among these two components. From these direct links we infer the semantic vertical relations `sc` and `isa`. They are semantically defined by the following logical rules :

- `isa(E,C) <- isa_d(E, C),`
- `isa(E,C) <- isa(E, C1) and sc(C1, C),`
- `sc(C1,C2) <- sc_d(C1, C2),`
- `sc(C1,C2) <- sc_d(C1, C3) and sc(C3, C2).`

`isa(e,c)` is interpreted as `e` is a representative of `c`. `sc(c1,c2)` is interpreted as `c1` is a subclass of `c2`. According to this interpretation we can verify that the schema given in figure 3.1 respects the three principles. This schema contains the three basic links:

- `entity` is a `class`, `isa_d(entity, class)`,
- `class` is a `class`, `isa_d(class, class)`,
- `class` is subclass of `entity`, `sc_d(class, entity)`.

It is clear that the consequence of the above rules and facts is that:

- **`class` and `entity` are entities:**
  - **`isa(entity, entity)`,**
  - **`isa(class, entity)`,**
- **`entity` and `class` are instance of a class:**
  - **`isa(entity, class)`,**
  - **`isa(class, class)`,**

Here, we must question the validity of the `isa_d` and `sc_d` links we use and of the `isa` and `sc` links we infer. We have assumed implicitly that `sc_d` and `sc` link two classes, and that `isa` and `isa_d` link an entity to a class. In order to guaranty it, we have two possibilities:

- We can use tests within the rules, like, for instance,:

```
sc(X,Y) <- isa(X, class)
           and isa(Y, class)
           and sc_d(X,Y).
```

- We can use integrity constraints, like, for instance,:

```
forall [X, Y]: sc(X,Y)  -> isa(X, class)
                        and isa(Y, class).
```

Tests have the advantage that they do not force the stored data to be all correct, but restrict the interpretation to the correct data. They allow any extension to the model. In particular the definition by the user of a new link, semantically different, named sc.

Integrity constraints reflect a stronger point of view. The data must be correct and any extension must respect the constraints.

For the sc_d, sc, isa_d and isa, we adopt an intermediate solution. Indeed, direct links and semantic links are of a different nature:

- the isa_d and the sc_d link are implementation links; they represent how the class and instance hierarchy is actually stored;
- the isa and the sc link are semantic links; they represent how the class and instance hierarchy actually is.

One of the targets of the Möbius model is to be flexible as far as its connection with a relational schema is concerned. If the database is created by Möbius, we want to be able to choose among several form of mapping. If the database exists and contains data, the model acts as a filter on the database and interprets only the relevant data.

For that reason, the sc_d and isa_d links, the system structuring links, will filter and manipulate only the valid data.

On the other hand, isa and sc, as vertical relation, define the elementary semantics of the whole model. It is therefore natural to limit their extensibility within the scope of this semantics. Any, even user defined, isa and sc link must represent the subclass and the instance relation. Thus we state the following constraints:

- `forall C1 C2: sc(C1, C2) -> isa(C1, class) and isa(C2, class),`
- `forall E C: isa(E, C) -> isa(C, class) and sc(C, entity),`

The Möbius model kernel is defined by a semantic network whose semantic is given by a direct translation into Datalog facts, rules and constraints.

Now we shall examine the meaningful extensions to this schema. Each new item introduced in the model must verify the three basic principles. Each new component must

be an entity, it means that it must be created simultaneously with a `isa_d` link. The integrity constraints guarantee that it can only be linked to a class which is a subclass of `entity`. Without the integrity constraints, the user must verify, at creation time that he has fulfilled the principles. As this should also be done for any modification of the schema, it is clear that integrity constraints are the best way to ensure the safety of the model with regards to the three principles. The strong consequence of integrity constraints is that `isa` and `sc` cannot be defined for domains different than those specified by the constraints. This characterizes the particular role played by these vertical links. For instance, the name `isa` cannot be used for another purpose than representing the instance relation.

To summarize, we can say that with a very simple encoding of the three principles we chose for the metacircular definition by means of logical rules, facts and integrity constraints, we have the basis of an extensible (we did not say yet how to extend it) model without any limitation in the number of conceptual levels. The question is now: is it still possible to characterize the intuitive levels of an application?

3.1.3 Level Characterization

Intuitively, when designing a knowledge base, one can think of a finite number of conceptual levels.

The basic level is the terminal instance level. It contains the data of the application. Generally, it corresponds to all the stored facts and implicitly the data possibly available through intentional definitions. The schema level is the set of definitions - rules, classes, class hierarchy - describing the general mould in which the terminal instances are poured.

At the schema level, classes, attribute definitions (cf. section 3.2) and rules can be viewed as data. Thus it is again natural to think that there exists a schema (meta-schema) abstracting these data. It can be interesting to have access to the design of this meta-level. Everything being an entity, the separated levels do not come from the model definition. However, all the information and the tools to characterize and control the levels and their interactions are given in the model. The levels can be defined from the `isa` and `sc` links by means of rules. Their interactions can be controlled by means of constraints.

We define a predicate `level/2` associating an entity to its level. We use numbers to name the levels. The terminal instance level, level 0, is the level containing only entities that are not classes (rule 1). The `isa` relation identifies the frontier between two levels relating the instances to their classes. Two entities linked by an `isa` link belong to two consecutive levels (rules 2 and 3). Inside a given level (higher than 0), classes are organized in the inheritance hierarchy thanks to the `sc` link. Two entities linked by a `sc` link belong to the same level (rules 4 and 5).

```
rule 1: level(E, 0) <- not isa(E, class),

rule 2: level(E, I) <- isa(E1, E) and level(E1, J) and I is J + 1,
```

```
rule 3: level(E, I) <- isa(E, E1) and level(E1, J) and I is J - 1,

rule 4: level(E, I) <- sc(E, E1) and level(E1, I),

rule 5: level(E, I) <- sc(E1, E) and level(E1, I).
```

The problem with the second rule is that it only characterizes the level of a class if the class has instances with a computble level. However, it is obvious that a subclass of `entity` which is not a subclass of `class` is of level 1 (this level corresponds to the 'schema' in terms of the classical data modeling terminology). We must define the level of an entity by the following rules:

```
rule 1: level(E, 0) <- not isa(E, class),

rule 1bis: level(E, 1) <- sc(E, entity) and not sc(E, class),

rule 2: level(E, I) <- isa(E1,E) and level(E1,J) and I is J + 1,

rule 3: level(E, I) <- isa(E,E1) and level(E1,J) and I is J - 1,

rule 4: level(E, I) <- sc(E, E1) and level(E1, I),

rule 5: level(E, I) <- sc(E1, E) and level(E1, I).
```

The two components of the kernel `entity` and `class` have a very particular status. They belong to every possible level (an infinite number). This is not an interesting new information: it is the foundation of the model. For this reason we must exclude them from the above definition: `level(E,I)` is defined for E different from `entity` and `class`.

Figure 3.2 gives an example of the level characterization in the case of terminal instances (entities `E1`, ..., `E4`), a schema (classes `C1`, ..., `C4`) and a meta-schema (meta-classes `MC1` and `MC2`).

There remain situations where a level is not calculable. However, these situations correspond to a partially defined model where it is impossible, even intuitively, to associate a level to the entity.

Example 3.1.1 We assume we introduce the new class `'c'` with the following declarations (cf. figure 3.3):

```
sc(c, class),
isa(c, class),
```

It is clear that c is not of level 0 nor 1 ; it is a meta-class. But unless we provide more information (about its instances) we do not know yet to which level it belongs. ◇

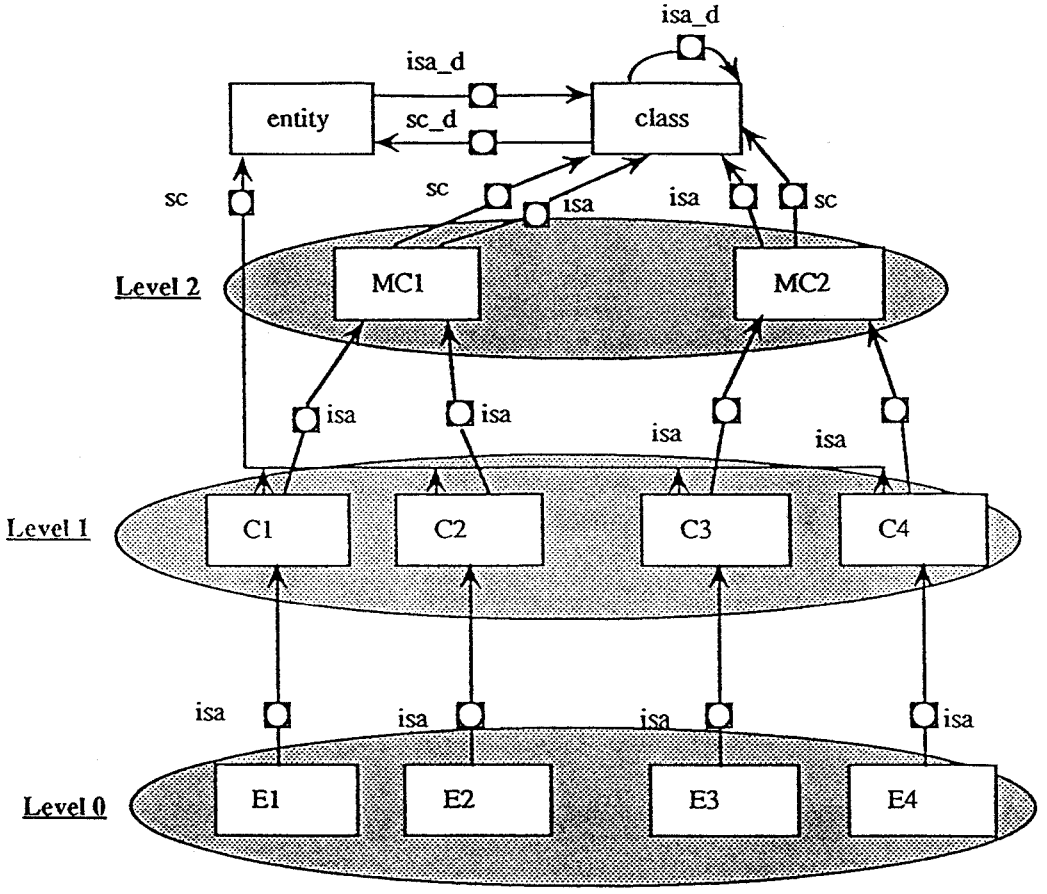

Figure 3.2: Horizontal Levels

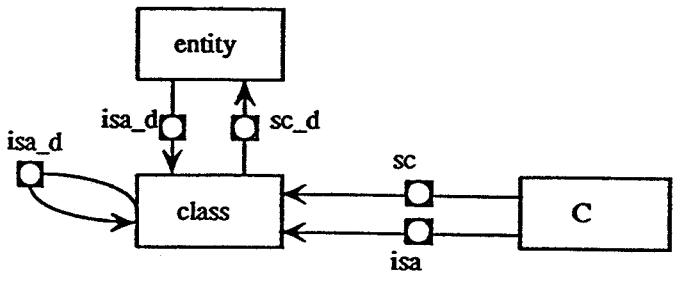

Figure 3.3: Partially Defined Schema

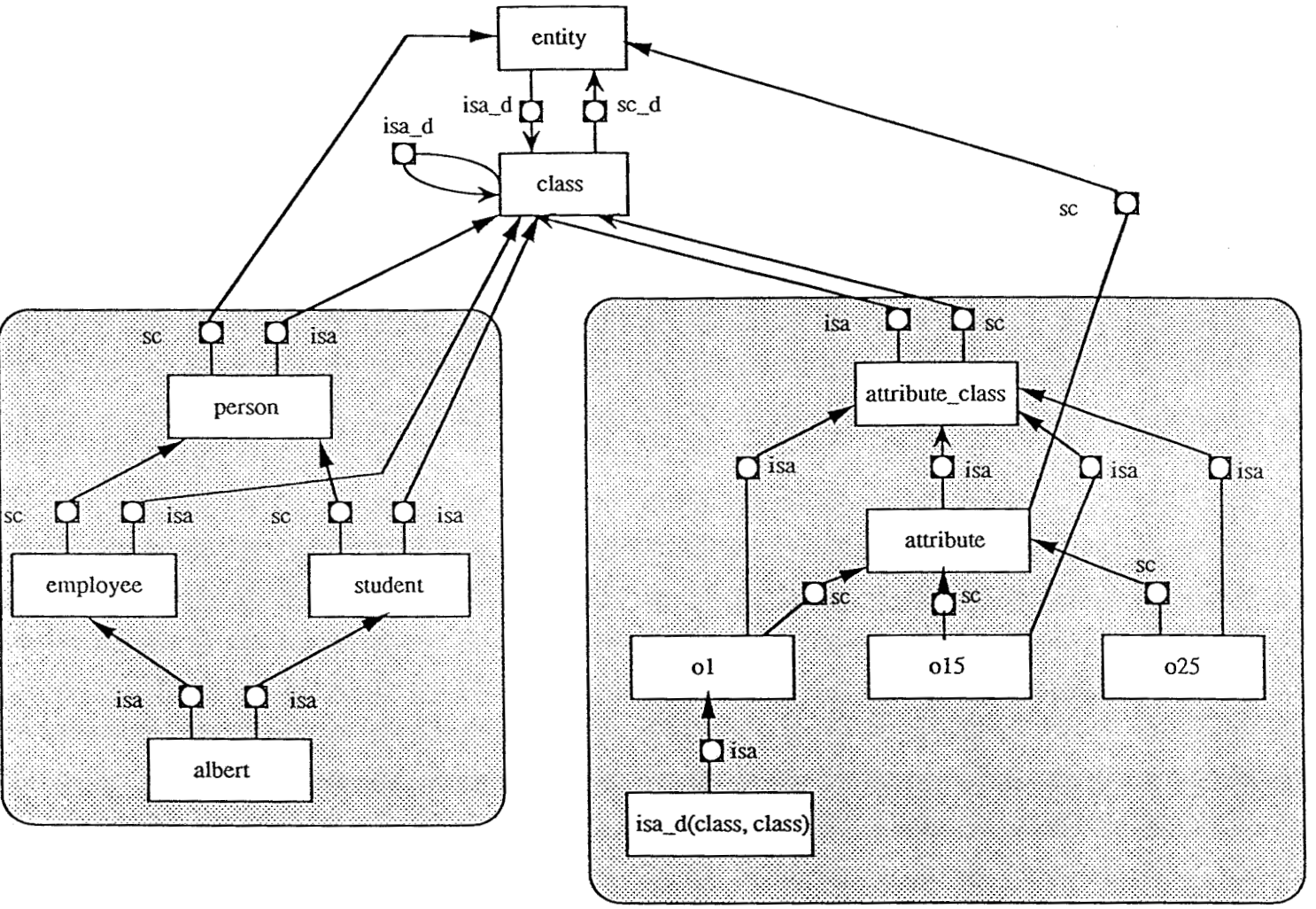

Figure 3.4: Vertical Levels

However the structuring of the general schema in levels does not exactly represent the intuition. Indeed, apart from this horizontal stratification there exist a vertical one. For instance, in the Möbius kernel, as we will see below, attributes are entities which belong to the level 0 according to the above definition. It is not reasonable to consider that entities of an application like a person or a number belong to the same conceptual level as the fact that a person's age is a given number. Again, by adding new constraints, one can identify vertical organizations of data. For instance, multiple representation can be limited to a given subtree of the global hierarchy. In the case of attributes (cf. figure 3.4) the subtree of root `attribute_class` corresponds to this sub-hierarchy. In that sense a vertical level is defined from the root of the selected subtree. The link `vertical_level` associate an entity to a vertical level it belongs to.

```
vertical_level(E, C) <- isa(E, C).

vertical_level(E, C) <- sc(E, C).
```

```
vertical_level(E, E).

vertical_level(E,C) <- isa(E, C1) and vertical_level(C1, C).
```

The characterization of both horizontal and vertical levels is useful in terms of organization of the data. It can be used, for instance, for security purposes: one can restricts the access to certain levels. It is also a methodological information one can use to enforce a clean organization of the model. In particular one can add constraints enforcing an entity to belong to a single horizontal level:

```
forall [E, I, J]: level(E, I) and level(E, J) -> I = J.
```

Vertical levels can be used to constraint the multiple representation of entities (cf. subsection 3.3.2). Let us assume we have a class `person` as the root of a subgraph of our hierarchy. We want the instances of `person` to stay in this subgraph. This can be done by stating the following constraint:

```
forall [E, C]: vertical_level(E, person)
          and vertical_level(E, C) -> vertical_level(C, person)
                                      or vertical_level(person, C).
```

The metacircular kernel extension can be organized in subsets of coherent information. The vertical links `isa` and `sc` can be used for such a characterization. For a complete characterization, one must take into account both vertical and horizontal stratifications. These aspect always lead to constraints, one must add to the model, restricting the intersection of levels to the empty set. The two components of the metacircular kernel cannot belong to any level because they implicitly belong to every level.

One can think of many other characterization of the model components. Providing integrity constraints is a means of structuring the model. Classes in exclusion (cf. figure 3.5) or necessarily sharing their instances can be encoded thanks to integrity constraints. Here we point out the interest of being able to manipulate data and meta-information within the same language and model. There are many situations where one wants to manipulate the data, and the meta-schema at the same time. We conclude that having a metacircular definition for the model is not confusing but, on the contrary, is a useful tool for structuring and customizing. Of course, this is only feasible by providing the sophisticated tools that are deduction rules and integrity constraints together with a data model defined with them. The approach we take for the level characterization is a general philosophy of Möbius: we drop the restriction normally included in a data model definition and provide tools (rules and constraints) to express them explicitly. Therefore, the user or the designer have the capability to customize their model.

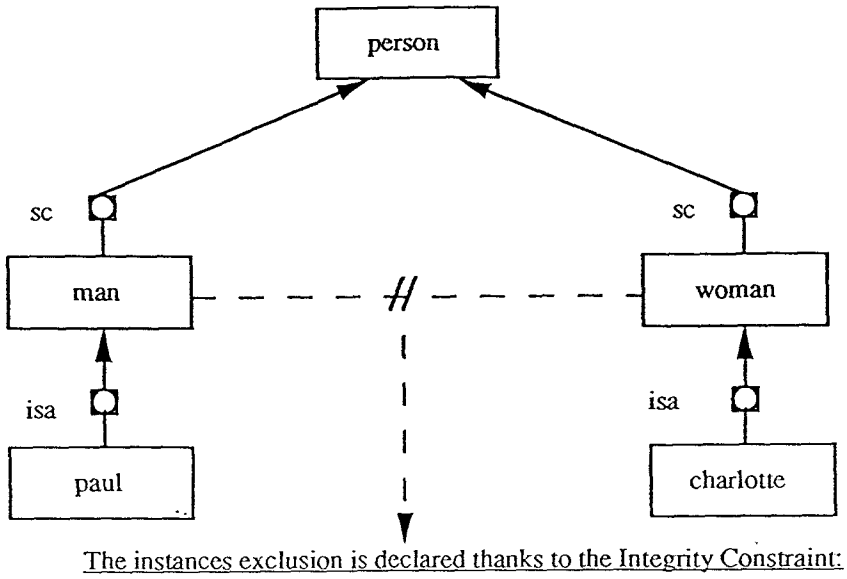

Figure 3.5: Classes in Exclusion

3.2 Attributes and Associations

3.2.1 Attributes as Entities

This section presents one of the main concepts of the Möbius model : the attribute. This notion is the basic constructor of the schemas. If we had to classify our model in one of the several knowledge representation paradigms, we would say it is of the family of the semantic network approaches [Da77], [Bra77], [Hen78], [Fin79], [Sow84]. What we call an entity is an empty object (cf. chapter 5 for a discussion about object identity). An entity is reduced to its identifier. In some limited cases (numbers, strings or other "printable identifier") it contains its meaning. Otherwise, the semantics of the entity is given by its position in the network of attributes. At least the `isa` link gives the minimal information that the entity exists. An attribute is an oriented labeled link between two entities, respectively the source and the target. Examples of attributes are `isa(class,entity)` and `age(jean,23)`.

This point of view extends the concepts introduced by the frame approaches, [Min87], where facets are associated to attributes (slots) in order to specify their properties and behaviours (for example to declare default values, daemons etc). Usually, facets are "terminal concepts" and are not allowed to be manipulated as entities. In the Möbius model, attributes are themselves entities which benefit by the same uniform way of representation and manipulation than the other kinds of entities: they are associated with attributes and are instances of the related attribute classes. From this point of view, our model shares (and is inspired by) a lot of features of the frame based systems,

[Rec88], and the so called hybrid systems [Fer83], [BS85], [Bob86], [Duc88], [Dug89], [BCP86]. They both consider attributes as normal entities (frames, objects ...). In frame languages, frames are used as prototypes, i.e. as instances used to create similar entities by a copy mechanism. In hybrid languages, frames are integrated in a class-superclass-instance schema similar to ours.

In [Fer84], Ferber points out the risk of an infinite regression when attributes can have attributes. This problem is solved when attributes are considered as entities which can be defined (computed) only when needed.

The example 3.2.1 [1] below shows that the Möbius evaluation can handle the infinite regression for the two attributes of attributes v1 and v2, giving respectively the first and the second argument.

Example 3.2.1 Let us assume that an attribute age links a person to its age (an integer).

```
?- eval([age(jean,X)]).
X = 25

yes
?- eval([v1(age(jean,X),Y)]).
Y = jean
X = 25

yes
?- eval([v2(age(jean,_),X)]).
X = 25

yes
?- eval([v2(v2(v2(age(jean,_),_),_),_),X)]). % etc
X = 25

yes
```

◇

3.2.2 Attribute Definitions as Classes

Attributes are representatives of more general entities: attribute classes. Attribute classes specify the attributes definitions:

- the name of the attribute;

[1] The host language is basically Prolog extented with several primitives. `eval/1` evaluates a list of literals and produces the answers tuple-at-a-time. The procedure is invoked from the Prolog top-level. In the following the syntax of examples is not formally presented. It corresponds to the syntax (host and query language) used in the experiments.

- the source domain: the class of which the first element of the attributes must be a representative;
- the target domain: the class of which the second element of the attributes must be a representative;
- the extension: how the attributes can be retrieved by the storage system used for the model implementation;
- the intention: a logical expression of the query language from which the attribute can be derived.

The above data are associated to attribute classes in the same way as entities are associated together. It is then natural to consider these associations themselves as attributes of the attribute classes.

Example 3.2.2 Let us consider the attribute class of name `isa` (cf. figure 3.6). As a class of entities ("everything is an entity"), its instances will inherit of a `isa` attribute. `o18` is the entity identifier of an attribute class whose name is `isa`.

```
?- eval([isa(isa(jean,person),X)]).
X = o18
        more? -- ;
X = attribute
        more? -- ;
X = entity

yes
?- eval([name(o18,X)]).
X = isa

yes
?- eval([sd(o18,X)]).
X = entity

yes
?- eval([td(o18,X)]).
X = class

yes
?- eval([int(o18,X)]).
X = int(_g12,_g13,[isa_d(_g12,_g13)])

yes
?- eval([int(o18,X)]).
X = int(_g12,_g13,[isa_d(_g12,_g14) and sc(_g14,_g13)])

yes
```

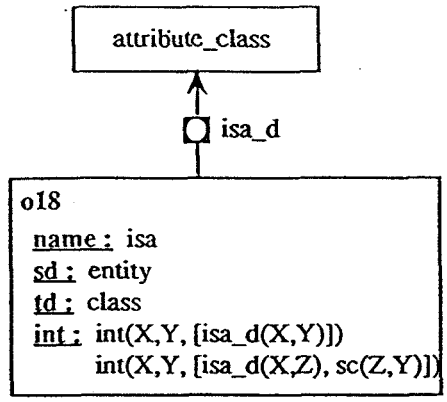

Figure 3.6: The o18 (ISA) Attribute Class

The o18 attribute class defines attributes of name isa. They are representatives of the classes o18, attribute and entity. They link entities to classes. They are not stored on disk but defined by the two rules:

```
isa(X,Y) <- isa_d(X,Y).
isa(X,Y) <- isa_d(X,Z) and sc(Z,Y).
```

◇

However, we would like the user to be able to deal with a schema interpretation in accordance with an object-oriented conceptual point of view, i.e. we want the designer to be able to extend it, relying on the classical notions of class, class hierarchy, attributes associated to classes. Our model is dual. One can see the class hierarchy on the one hand and the attribute classes on the other. But attribute classes are associated to classes thanks to their source and target domain. Attributes named att_d and att link a class to the attribute classes respectively directly associated with it or inherited. This double point of view is illustrated by the example 3.2.3 (cf. figure 3.7) below.

Example 3.2.3 **Attribute classes are associated to classes through their source domain. Classes are associated to attribute classes through the att_d and att attributes.**

```
?- eval([att_d(class,X),name(X,N)]).
N = o17
X = sc
        more? -- ;
N = o2
X = sc_d
        more? -- ;
N = o29
X = att
        more? --
```

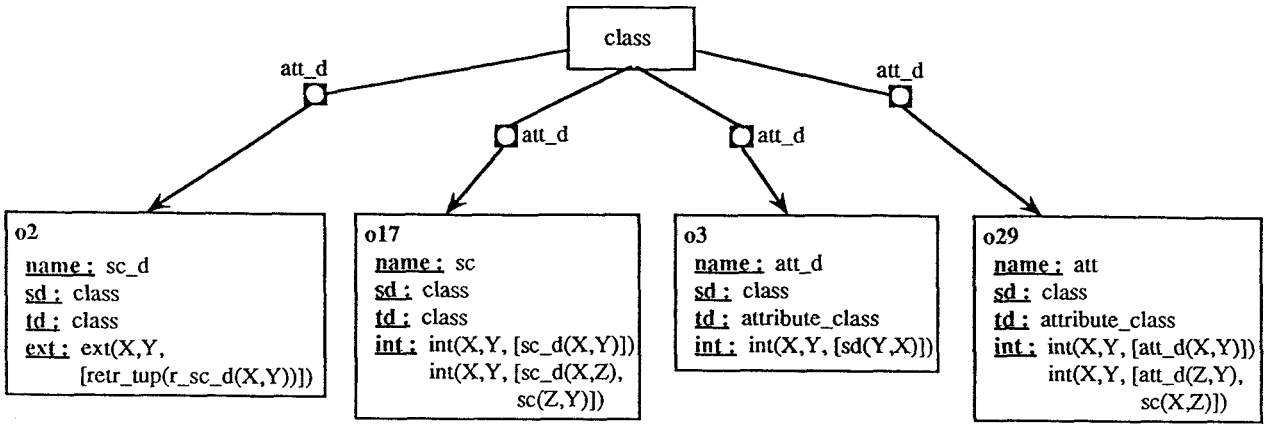

Figure 3.7: att-d and att

```
yes
?- eval([att(class,X),name(X,N)]).
N = o1
X = isa_d
        more? -- ;
N = o18
X = isa
        more? -- ;
N = o17
X = sc
        more? --

yes
?- eval([sd(o1,X),name(o1,N)]).
N = isa_d
X = entity

yes
```

o1 is associated to `entity` and is inherited by `class`, its subclass. ◇

3.2.3 Extension to the Metacircular Kernel

In order to be consistent with the metacircular definition presented in the previous section 3.1, the vertical relations `isa_d`, linking an entity with the class it directly represents, and the `sc_d` attribute, linking a class and its direct superclass, are associated to the meta-classes `entity` and `class` as the main elements of the meta-circular kernel. The `isa_d` attribute definition is associated to the meta-class `entity`, the `sc_d` attribute definition is associated to the meta-class `class`. So are their respective transi-

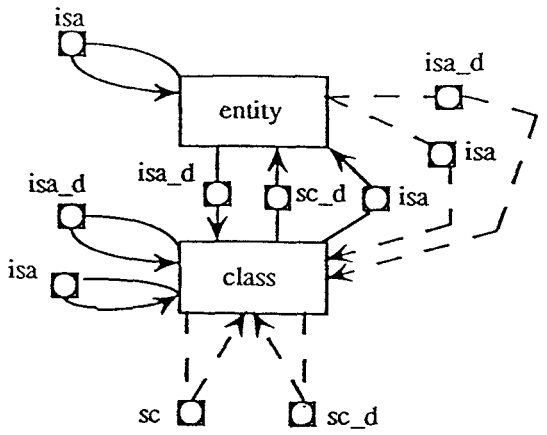

Figure 3.8: Attributes Classes Associated to the Kernel

tive closures, i.e. `isa` and `sc`. The metacircular kernel can be represented by the figure 3.8.

Nevertheless, to be complete with our definition, we have to position as well the attribute classes in the schema. Classes have a general definition specified by the class `class`. Attribute classes have some additional features since they play a constructive role in the definition of the model itself: they define semantic links between the model entities. Then, we introduce a particular class specifying the definition of the attribute classes. This class is called `attribute_class` and allows the creation and the representation of attribute classes. The properties mentioned above are modeled as attribute definitions associated to `attribute_class` :

- Name (`name`): name of the attribute class, used as a constructor for the expressions. Several attribute classes can have the same name (cf. section 3.3).
- Source domain (`sd`): class of the source entities;
- Target domain (`td`): class of target entities;
- Intentional definition (`int`);
- Extensional definition (`ext`): mapping.

Some other additional attributes are also defined to deal in particular with some inheritance problems (cf. chapter 4). A general attribute class called `attribute` is defined as a direct representative of the class `attribute_class`. This class, is associated, in particular, with the `v1` and `v2` attributes linking an attribute to its arguments. The model kernel is illustrated by the figure 3.9.

3.2.4 Associations

In Möbius, no strong distinction is made between the notion of attribute and the notion of association. For us, an association is just the generalization of an attribute to any ar-

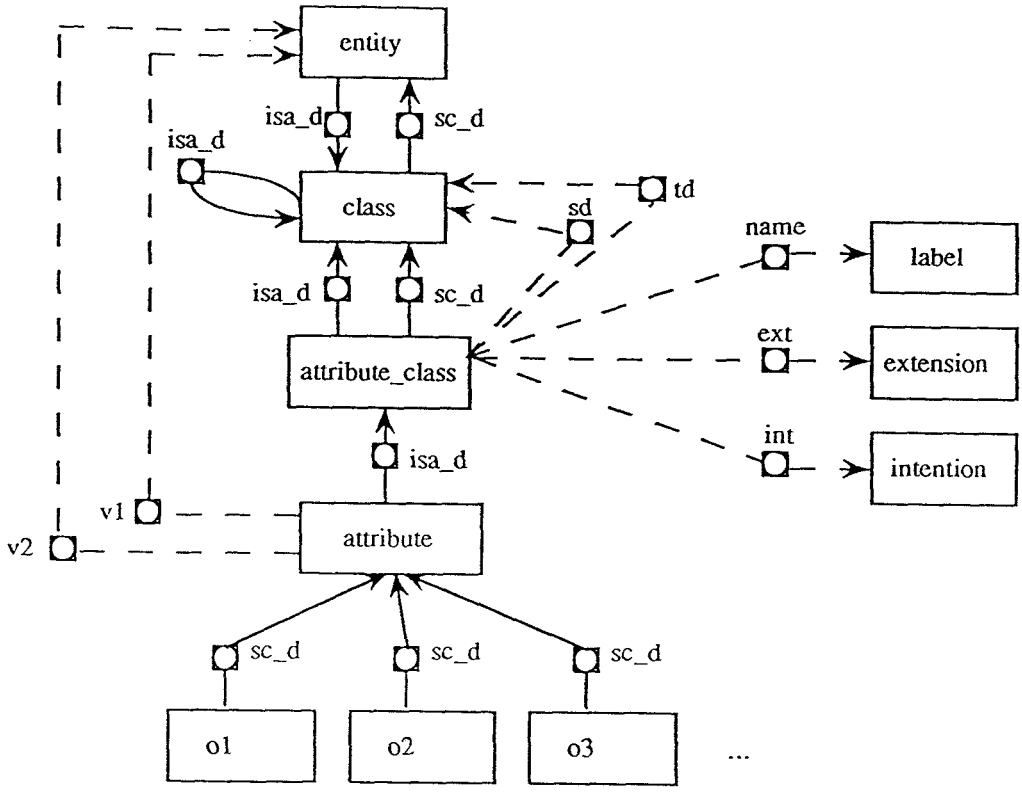

Figure 3.9: Complete Kernel

ity. The difference between attribute and association existing in the Entity-Relationship (ER) models no longer holds. In the ER models, entities have attributes which are records for values. Some extensions of the ER model accept complex and structured values like sets, Cartesian product, etc. Associations relate entity sets (equivalent to our classes) and are entities themselves possibly with attributes. The difference between value sets and classes does not exist in our model. Values are special entities inherited from the host language (Prolog) like strings or integer. But the main point is that value sets are classes.

The system kernel can be built using the notion of attribute. That is why we separate this notion from the more general notion of association. Association definitions are classes defined in the Möbius kernel. They have a name, an arity, they provide an attribute vn for each component ($0 < n < \mathit{arity} + 1$) for their instances. They denote sets of elementary associations. They can also define other attributes for their instances. Mainly, in the rest of this document, we will consider that associations are a straightforward generalization of attributes and we will not give special explanations for their case.

3.3 Model Semantics

3.3.1 Semantics

The Möbius objects are accessible to the user through a query language (Datalog). The basic components of this language are attributes and associations. Although we have experimented (cf. chapter 5) other syntax like functional ones, we adopt here a predicate notation where attributes and associations are n-ary literals of the form:

```
attribute_name(arg1, arg2)
```

or

```
association_name(arg1, ..., argn),
```

where arg_i are either variables or an entity identifiers. In some special cases, identifiers can be structured terms.

An expression is composed of the above literals, logical connectives and quantifiers. Therefore, the query language is, apart from the special constructs used for controlling inheritance (views, full-name, viewpoints) (cf. chapter 4), of the family of Datalog languages. The definition of Möbius includes stratified negation which is necessary for the definition of overriding (cf. section 4.2). In the experimental versions of the Möbius system, we have included aggregate functions and procedural literals, relying on the functionalities of the implementation environment EKS-V1. For practical applications, such features appeared to be necessary.

The query language is used in three circumstances:

- One can state integrity constraints over the model. Notice here that the constraints are not associated to classes but are global to the application. Such a choice can be criticized as it seems to be in contradiction with the philosophy of the Möbius model. It does not allow to structure the constraints using the class hierarchy. We actually want to delay a better integration of constraints in the class hierarchy until we have a better understanding of the practical possibilities of Möbius and feedback from the applications.

- In Möbius, attribute definitions can be given in terms of other attributes. The so called attribute intension is an expression of the query language. This expression denotes a set of pairs which are instances of the attribute class.

- Of course queries can evaluated from the host language through the `eval/1` procedure. This primitive, as included in a Prolog environment, has a 'success/failure' mode and produces the tuples one at a time with backtracking.

The other objects of Möbius cannot appear in queries as literals. In particular the relational expressions giving the attribute classes extensions cannot be used directly in the queries, intentions or constraints.

Similarly to other approaches aiming at the integration of object-oriented aspects in a logical context ([Dal89], [CW89]), the semantics of Möbius is given by a translation into a first order logic program. This solution is not only simple and clean but also allow a quite direct implementation of the Möbius system on top of existing deductive database systems.

As the whole model is founded by the notion of attribute and attribute class, the semantics is given by the translation of attribute classes into first order logic programs. For each attribute class, for each intension (`INT`) and extension (`EXT`), the program is a set of rules of the form:

```
attribute_name(X,Y) <- isa(X, sd) and isa(Y,td)
                       and INT.
attribute_name(X,Y) <- isa(X, sd) and isa(Y,td)
                       and EXT.
```

`INT` and `EXT` are the expressions stored in the `int` and `ext` attributes of the attribute class; `attribute_name`, `sd`, `td` are, respectively the name, source domain and target domain. We assume here that the extension is a relational expression syntactically compatible with Datalog.

A single attribute definition may have more than one name, source domain or target domain, as well as several extensions and intentions. Therefore, the number of generated rules depends on all the possible combinations.

An expression in the query language denotes the same thing as its translation in first order logic. The evaluation of an expression corresponds to the evaluation of the same expression against the translated program. We will see in the following that the actual translation needs to be a little more complicated. In particular, the specification of overriding introduces new components in the generated rules and uses negation.

Now, we can analyse the problem of inheritance and see how the above translation gives a first solution to it. In chapter 6 we discuss the direct use of the generated rules as an implementation for the Möbius language on top of a deductive database.

3.3.2 Inheritance

The inheritance relation is a relation defined on a graph and that determines, for certain nodes, the availability of certain information contained in other nodes. The most common forms of inheritance are instance variables and methods inheritance in a class and instance hierarchy, delegation in prototypes languages or inheritance of properties in an aggregation graph.

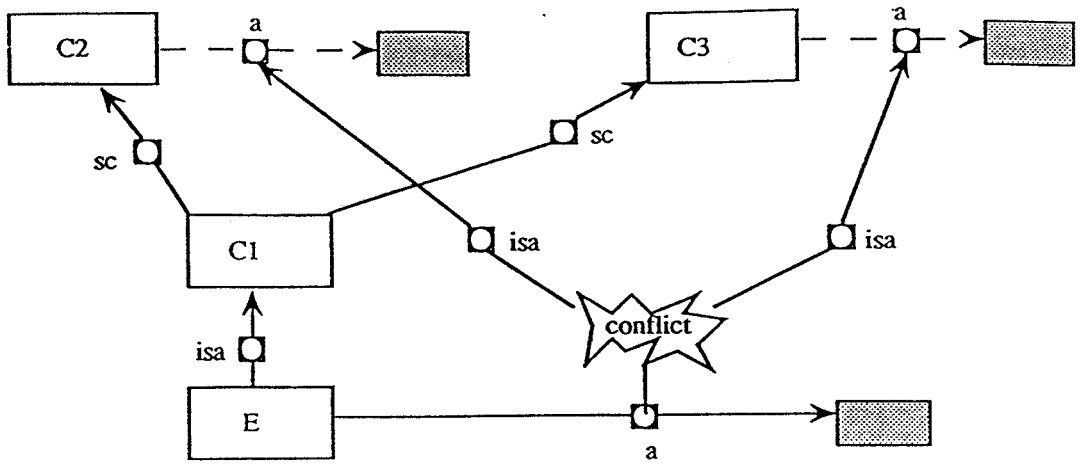

Figure 3.10: Multiple Inheritance

Inherited information can be of several natures: it can be structural information (instance variables, attribute definitions), properties in some semantic networks [Hen78], behaviours or methods or attributes values (class variables in Smalltalk). In this section we deal with the problem of inheritance for attribute (and association) definitions. As opposed to a shared point of view in the object and logic-oriented community, we consider that rules can be used to define both attributes and methods and that the separation should be kept only in the sense that methods are logic programming rules with control and side effects. In particular, methods may update the knowledge base while attribute rules only denote intentional information (cf. chapter 7).

We consider a class hierarchy, organized by a `sc` superclass link, which is an oriented and acyclic graph whose single root is the class `entity`. Terminal instances are entities attached to classes in this graph by the `isa` instance relation.

When not restricting ourselves to a tree structure we allow the typical situation of multiple inheritance (cf. figure 3.10): an instance `E` of a class `C1` may inherit attribute definitions with the same name from two unrelated parents of `C`.

This kind of clash in inheritance can also occur in the case of a tree structure when attribute definitions of the same name occur along a single branch. We call this situation multiple definition (cf. figure 3.11).

Moreover, allowing an entity to be a representative of several unrelated classes, we can face the situation where an attribute definition for the same name is inherited from several classes. This situation is called multiple representation (cf. figure 3.12).

These three situations are potentially generating conflicts. In this section we show that our approach allows to ignore these potential conflicts. It is however clear that ignoring them is not always an acceptable solution. In the next chapter (chapter 4), we analyse the possible conflicts and present tools to solve them.

Most of the object-oriented approaches, when considering attributes definitions (instance variables) or methods inheritance, take a deterministic point of view. They usually do

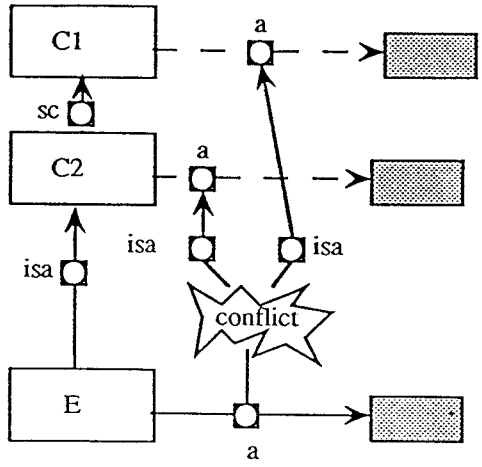

Figure 3.11: Multiple Definition

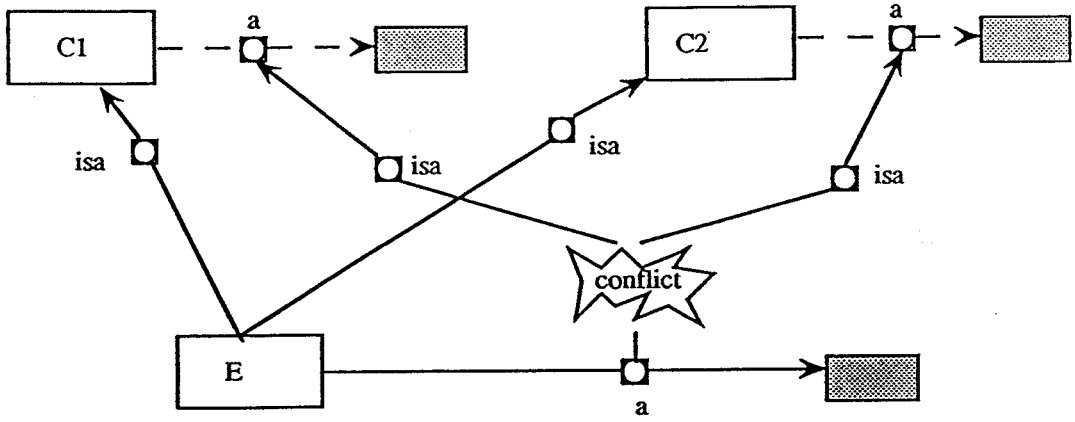

Figure 3.12: Multiple Representation

not support techniques able to take several definitions into account nor to produce one definition from several (this second feature is provided by some frame based systems and is called semantic unification [Dug89]). In the context of logic programming and deductive databases we choose, programs or definitions are sets of rules and facts and therefore can be easily stuck together. If the evaluation of such definitions (one or more) leads to several solutions, these solutions can be grouped into a set or produced individually by backtracking. We choose a tuple at a time evaluation with backtracking interface (cf. chapter 6). The rules and facts of our definitions can be used without taking care of the order in which they are computed, as we assume that they neither contain nor need control statements and that the evaluation algorithm is sound and complete.

3.3.3 Inheritance in Möbius

For the reasons exposed above, inheritance in the classical tools is always associated with strategies for solving the conflicts or with constraints on the language. Indeed, the simplest constraint avoiding all the conflicts consists in forbidding two attribute definitions with the same name globally in the system, or on the subgraph where conflicts may appear if multiple representation is not allowed or limited. The algorithmic solutions consist in a more or less intelligent graph traversing strategy used to select the single definition to be used. However, there remain situations where this "look up" procedure cannot guess information which is not encoded in the system but trivial for the designer of the knowledge base. For this reason, we investigate, in the next chapter, solutions where the information about inheritance and conflicts is explicitly represented in the system.

Thanks to the properties induced by the logical framework and its model theoretic interpretation (as opposed to an operational semantics), as we announced, the basic mechanism for inheritance ignores conflicts. In the cases of multiple inheritance, multiple definition and multiple representation all the inherited definitions are exploited. On the other hand, the algorithmic search of these definition by a graph traversal is replaced by an associative search and a compatibility control encoded in rules, normally evaluated by the evaluator. Namely, each intention or extension is augmented by tests checking if the produced data are compatible with the respective domains on which the attribute is defined. We can underline here that, in that case, inheritance exists not only on the source domain (the class associated with the attribute) but also on the target domain. For that reason, associations with several domains are a simple generalization of attributes to any arity.

The inheritance mechanism is illustrated by the following example.

Example 3.3.1 We assume that we have a class `worker`, subclass of the class `person` (cf. figure 3.13). `francois` is a `worker`, `jean` is a `person` (direct instantiation links). The attribute classes `name`, `forename`, `address` and `phone_number` are defined at the level of the class `person`. The workers are also associated with the `address` and the `phone_number` attributes (those of the company they are working for). New attribute classes with possible distinct mappings or intentional definitions are therefore created. We can obtain the following answers :

```
?- eval([isa(francois,X)]).
X = worker
        more? -- ;
X = person
        more? --

yes
?- eval([isa(jean,X)]).
X = person
        more? --
yes
?- eval([name(francois,X), forename(francois,Y)]).
Y = "Francois Xavier"
X = "Bastide"
```

The name and forename attributes have been inherited from the class person.

```
?- eval([address(jean,X), phone_number(jean,Y)]).
Y = 612600
X = "36 rue des alouettes 31400 Toulouse"

?- eval([address(francois,X)]).
X = "11 rue Marcel Pagnol 31100 Toulouse"
        more? -- ;
X = "Z.I. des touristes 31250 Blagnac"
```

Two answers are given : the private address of the person **francois** and the company address of the worker : the two attribute classes **address** have been used for the evaluation.

```
?- eval([phone_number(francois,X)]).
X = 61264012
        more? -- ;
X = 61050505
```

◇

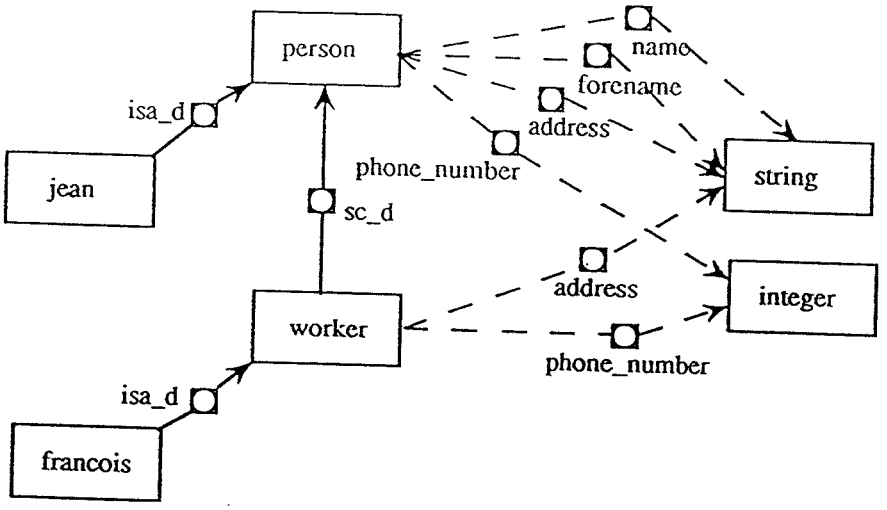

Figure 3.13: Example

Chapitre 4

Tools for Inheritance

Outils pour l'héritage

La sémantique de l'héritage dans Möbius permet, a priori, d'ignorer tous les conflits d'héritage classiques:

- *définition multiple : plusieurs attributs de même nom sont définis le long d'une branche du graphe d'héritage ;*
- *héritage multiple : plusieurs attributs de même nom définis sur plusieurs branches du graphe d'héritage sont hérités par un même noeud ;*
- *représentation multiple : une même entité hérite de plusieurs attributs de même nom par des branches différentes.*

Cependant, il nous semble que des outils plus sophistiqués sont nécessaires pour contrôler l'héritage et faire face à des situations de modélisation complexes.

Nous proposons un outil de masquage des définitions d'attributs pour les instances d'une classe donnée qui, normalement, hériteraient de cette définition. Cet outil statique de contrôle de l'héritage permet, en particulier, de gérer les redéfinitions d'attribut. Encore une fois, la sémantique du masquage est intégrée dans la traduction en formules logiques.

Lorsque le schéma de l'application est fixé, nous montrons comment un mécanisme dynamique de vues (inspiré des points de vues *décrits dans [Car89]), que nous avons étendu à des expressions complexes, permet de contôler dynamiquement l'héritage pendant les requêtes.*

Enfin, les définitions d'attributs étant identifiées par le triplet nom, domaine de départ (classe de départ), domaine d'arrivée (classe d'arrivée), nous proposons d'intégrer au modèle une forme de contrôle extrême par l'utilisation du nom complet des attributs dans les expressions.

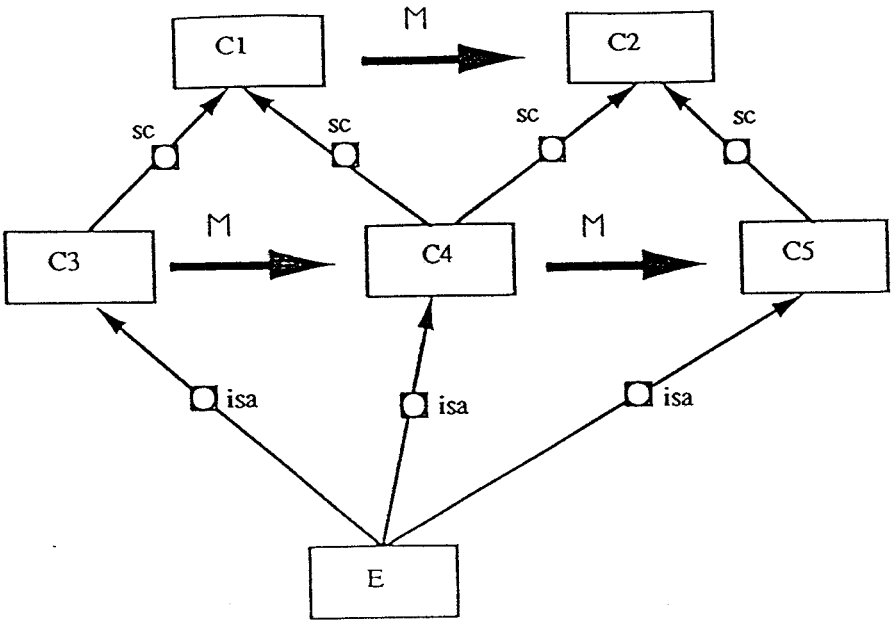

Figure 4.1: The Multiplicity Relation

4.1 Conflicts

As mentioned in the previous chapter, the three possible conflictual situations for inheritance are multiple inheritance, multiple definition and multiple representation (cf. figures 3.10, 3.11, 3.12). A conflict arises when one inherits from several pieces of information and wants only certain of them. Typically, one may not want to inherit of several attribute definitions with the same name. The general idea to solve such conflicts is to sort information according to the class and instance hierarchy. This sorting aims at selecting the information to deal with, either using some additional information (which can be explicitly or implicitly specified in the model), or exploiting special tools and concepts defined for this purpose.

The simplest tool is overriding. Where the raw inheritance principle would have selected two or more attribute definitions, overriding means that one of these definitions cancels the others. The main problem consists in providing an "intelligent" implementation of this feature, i.e., in a knowledge representation context, an implementation which is able to fit as well as possible with the various cases encountered in the reality. The common implementation of such a feature consists in choosing a unique definition according to a graph traversal algorithm implementing the inheritance strategy. The kernel of this strategy is based on a linearisation of the graph: one reads the inheritance graph from the concerned leaf to the root and takes the first available information. The inheritance graph is then flattened to a linear chain, without duplicates. One must define an order relation called multiplicity among the candidates for linearisation. E.g., one must order the direct super classes of a class or the several classes an entity is a direct instance (cf. figure 4.1 possible linearisations are: (C3 C4 C5 C1 C2) or (C3 C1 C4 C2 C5)).

An often used solution consists in taking the order in which the classes were created.

Now there is a vertical (super class and instance) order and an horizontal order (multiplicity), a strategy is a depth-first or a breadth-first traversal of the resulting graph from the leaves to the root. The most serious objection to this solution is that it uses an information (the order in which classes where created) that is not accessible to the user. At least it does not belong to the model. Whatever the chosen general algorithmic solution is, it never exploits the semantics of the application. For that reason we believe that solving conflicts must rely on semantic information. We dispose of an inheritance mechanism that, basically, can ignore the conflicts.

The first solution we describe to enrich the static description of inheritance is an explicit overriding capability integrated in the model semantics. But, here, explicit does not mean that it has to be set by hand for each single item in the model. On the contrary it can be programmed by a designer and parameterized by the model itself. This is described in the section 4.2.

Once the inheritance graph is augmented statically with overriding information, during a query or a session, a user may want to consider dynamically, for one or more entity, a coherent subgraph (cf example 4.1.1). Here, a coherent subgraph means a subgraph of the initial hierarchy that fulfills the three basic principles presented in 3.1.2. In particular it means that `entity` must be the root of this graph.

Example 4.1.1 Let us assume that we know a person `jean` who is both a student and an employee. We may want to refer to `jean` in the limited context of the coherent sub-hierarchy including the super and the subclasses of the class `employee`. The figure 4.2 shows this situation. In that case, the use of the attribute named `card_number` will refer to the card owned by every employee. ◇

The tool we provide to handle such a situation, called view, is comparable to views in relational databases. A view is a restriction applied on the whole class hierarchy. This is shortly described in the section 4.3.

The static or dynamic control of inheritance raises the problem of how far a user should be aware of the hierarchy structure when he queries it. As the tools above do not solve all the possible conflictual situations, we provide the extreme control on inheritance: the user is allowed to point out the single attribute definition he wants to deal with. We have assumed that an attribute definition is uniquely identified by a triple (name, source domain, target domain) (see section 3.2.2). All ambiguities are removed when the domain information is given in the query. We call this possibility the full-name operation as it consists in ticking off the literals in the query by their source or target domain and therefore, using a name uniquely identifying the attribute definition to be used. This is described in the section 4.4.

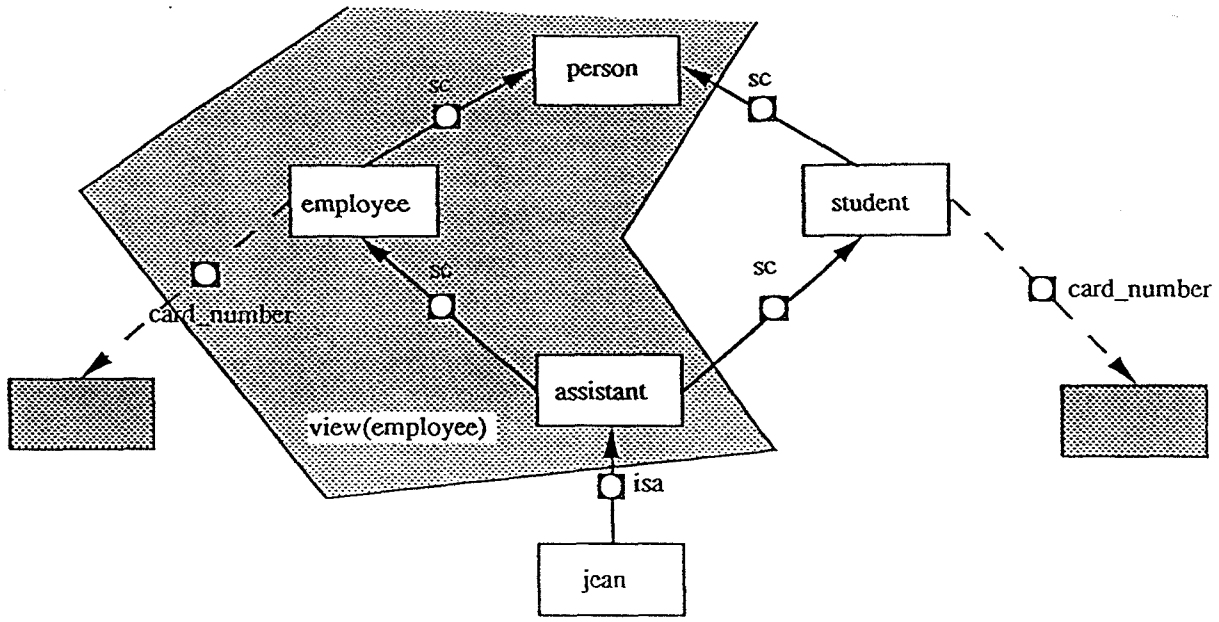

Figure 4.2: A Viewpoint on Jean

4.2 Overriding

4.2.1 Redefinition on the Source Domain

Overriding is a classical form of control in object-oriented programming languages. In general, it consists in a redefinition of a method along the class hierarchy. For a given method name, the lowest definition in the class hierarchy overrides the other definitions. This is the situation of multiple definition. In the other cases, multiple inheritance or multiple representation, an arbitrary order is used to decide which method definition is to be kept.

In a knowledge representation context, overriding is a different problem. Indeed, when considering a programming language together with inheritance, overriding is a form of control. It indicates how the different pieces of code will be exploited (cf. chapter 7). For a declarative language, in a knowledge representation model, overriding must be declarative. It is a way of expressing the validity (the scope) of the definitions (attribute definitions) in the class hierarchy. For instance, one can define an attribute `salary` for a class of employees, although this definition does not hold for the class of managing directors which is a subclass of `employee`.

In the Möbius model, no restriction is made a priori on the inherited attribute definition to be exploited. Therefore, there is no conflictual situation that cannot be treated: if several definitions are candidates, they are all used. Such a semantics is obtained by the translation into rules with the introduction of the `isa` constraining literals, checking the source and target domains.

At the model level, overriding will be explicitly represented as an attribute of attribute

classes. The attribute osd (overridden on the source domain) represents the information that an attribute definition is not available for instances of a given class. For example, the link osd(o41, managing_director) indicates that the attribute class o41 defining the salary of employees is not valid for the instances of the class managing_director.

If overriding consists in a redefinition, the attribute rsd (redefine on source domain) is used. It links two attribute definitions indicating that the first one overrides the second. If we want to define a salary attribute (entity identifier: o42) for the managing directors, the link rsd(o42, o41) between the two attribute definitions tells the system that there is a redefinition. The attribute definition o42 redefines the attribute definition o41. Therefore, o41 is overridden for the source domains of o42. This is translated into the following rule:

```
osd(X, Y) <- rsd(Z, X) and sd(Z, Y).
```

where sd is the source domain attribute.

In the same way inheritance is represented by supplementary literals in the rules defining the attribute semantics, overriding is encoded as a test in the translation:

```
attribute_name(X, Y) <- isa(X, sd) and isa(Y, td)
                        and INT
                        and not(isa(X, nsd)).

attribute_name(X, Y) <- isa(X, sd) and isa(Y, td)
                        and EXT
                        and not(isa(X, nsd)).
```

where rsd is the domain (class) for which the attribute definition is redefined: osd(att_id, nsd).

A practical use of redefinition declarations is illustrated by the following example, in the particular case of the multiple definition along a branch of the inheritance graph:

Example 4.2.1 As the example in 3.3.3, we assume we have a class **worker**, subclass of the class **person**. **francois** is a **worker** (direct instantiation link, cf. figure 4.3). The attribute classes **name**, **forename**, **address** and **phone_number** are defined at the level of the class **person**. The workers are also associated with the **address** and the **phone_number** attributes (those of the company they are working for). But in that case, we want these attribute definitions to override the **address** and the **phone_number** defined at the **person** level. Hence, the queries on **francois** will refer only to the company's address and phone number. The redefinition is specified by setting the rsd of the corresponding attribute classes.

```
?- eval([isa(francois, X)]).
X = worker
        more? -- ;
X = person
```

```
        more? --
yes
?- eval([address(francois, X)]).
X = "Z.I. des touristes 31250 Blagnac"
        more? -- ;
yes
?- eval([phone_number(francois, X)]).
X = 61050505
        more? -- ;
yes
```

A single answer is given: the private address and phone number of the person `francois` have been overridden. ◇

Classically, in the proposal for integrating object-oriented features to logic programming, overriding is translated into control: cuts or ordered rules in a Prolog program. This is possible because of the operational semantics of the target language. Here, the use of negation and of a relational representation of overriding has several advantages:

- the semantics is independent from the evaluation strategy of the target language;
- the translated rules can directly be used as an implementation;
- dynamic modification of the schema will not compel a complete modification of the implementation; the overriding information is locally encoded in the rules and does not depend on the order among them;
- for meta-manipulations, the relational information is available; it can be queried and used for other purposes; in particular it would be easy to implement a systematic redefinition in the case of multiple definition. For instance, the following rule define a `rsd` link between each pair of attribute definitions in conflict with the multiple definition:

```
rsd(X, Y) <- name(X, N) and name(Y, N)
             and sd(X, XSD) and sd(Y, YSD)
             and sc(XSD, YSD).
```

4.2.2 Redefinition on the Target Domain

The conception of a knowledge base with the support of a class hierarchy puts the stress on the association of attribute definitions to classes through their source domain. Therefore; inheritance is seen in most applications on the source domain. As we have seen in the preceding chapter, in Möbius, inheritance operates on both source and *target* domains. Therefore, in the same way we have overriding and redefinition on the source domain, we have overriding and redefinition on the target domain.

We define symmetrically the two attributes `otd` and `rtd`, indicating, respectively, the overriding of an attribute definition for the instances of a class for its target domain and

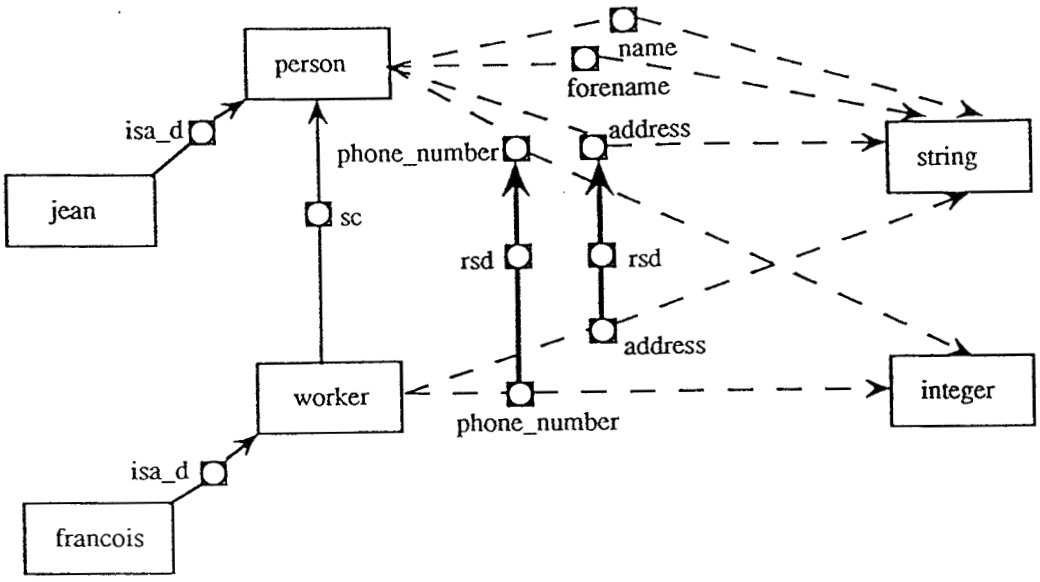

Figure 4.3: Redefinition

the redefinition of an attribute by another definition according to their target domain. The translated rules must be again augmented by a test that takes overriding on the target domain into account:

```
attribute_name(X, Y) <- isa(X, sd) and isa(Y, td)
                        and INT
                        and not(isa(X, nsd))
                        and not(isa(Y, ntd)).

attribute_name(X, Y) <- isa(X, sd) and isa(Y, td)
                        and EXT
                        and not(isa(X, nsd))
                        and not(isa(Y, ntd)).
```

where `ntd` is the domain for which the attribute definition is overridden (on the target domain): `otd(att_id, ntd)`.

The following example shows a practical situation where both the `rsd` and the `rtd` attributes are used.

Example 4.2.2 We assume that we want to deal with the connection between several flights. It is given that 'normal' flights impose to the passengers a waiting time of 15 minutes for boarding. An international flight imposes 25 minutes. We consider that a maximum of 15 minutes of delay for normal flights is allowed to take any connection into account. An international flight is allowed 30 minutes of delay.

Two classes are defined for the flights and four `connection` attribute definitions are attached to them. For each flight, the attribute says whether it exists a connecting flight taking into account their arrival and departure times (cf. figure 4.4). Distinct formulae must be used

depending on the kind of flight specified in the queries and on their role in the attribute (first flight or second flight of the journey). A `rsd` link is not sufficient: when one wants to know about the arriving flights which can be connected to a given flight `f`, the type of `f` must determine the formula to use (because the boarding times are different). This distinction is declared thanks to the overriding links.

The formulae defining the four attributes intentionally can be written as follow:

Formula 1 checks whether two normal flights can be connected (15 + 15 minutes of tolerance)

```
connection(X, Y) <- arrival(X, Z),
   departure(Y, T),
   D = T - Z,
   D >= 30.
```

Formula 2 checks whether a normal flight can be connected to an international one (15 + 25 minutes of tolerance)

```
connection(X, Y) <- arrival(X, Z),
   departure(Y, T),
   D = T - Z,
   D >= 40.
```

Formula 3 checks whether an international flight can be connected to a normal one (30 + 15 minutes of tolerance)

```
connection(X, Y) <- arrival(X, Z),
   departure(Y, T),
   D = T - Z,
   D >= 45.
```

Formula 4 checks whether two international flights can be connected (30 + 25 minutes of tolerance)

```
connection(X, Y) <- arrival(X, Z),
   departure(Y, T),
   D = T - Z,
   D >= 55.
```

We assume that the following facts are defined, giving the times of departure and of arrival for our favourite airport.

```
arrival(  f1, 14:00)
arrival(  f4, 14:00)
departure(f2, 14:30)
departure(f3, 15:00)
departure(f5, 14:30)
departure(f6, 15:00)
```

```
?- eval([connection(f1, Y)]).
Y = f6
        more? -- ;
Y = f2
        more? -- ;
Y = f3
        more? -- ;
yes
```

The formulae 2 and 1 are used (`f1` is a "normal" flight).

```
?- eval([connection(f4, Y)]).
Y = f6
        more? -- ;
Y = f3
        more? -- ;
yes
```

The formulae 4 and 3 are used. The formulae 2 and 1 are not taken into account because of the `rsd` links (`f4` is an international flight).

```
?- eval([connection(X, f2)]).
X = f1
        more? -- ;
yes
```

The formulae 3 and 1 are used (`f2` is a "normal" flight).

```
?- eval([connection(X, f6)]).
X = f1
        more? -- ;
X = f4
        more? -- ;
yes
```

The formulae 4 and 2 are used. The formulae 3 and 1 are not taken into account because of the `rtd` links (`f6` is an international flight). $\Diamond$

This last example points out that the redefinition is sometimes used as a way of escaping a badly designed hierarchy for a given point of view: in our case, it would have been easier to define two "sister"-classes (respectively national flights and international flights). Such a solution would have avoided the necessity of redefinition links between the attributes we dealt with. Nevertheless, the schema we have adopted for the example could also be justified for other applicative purposes.

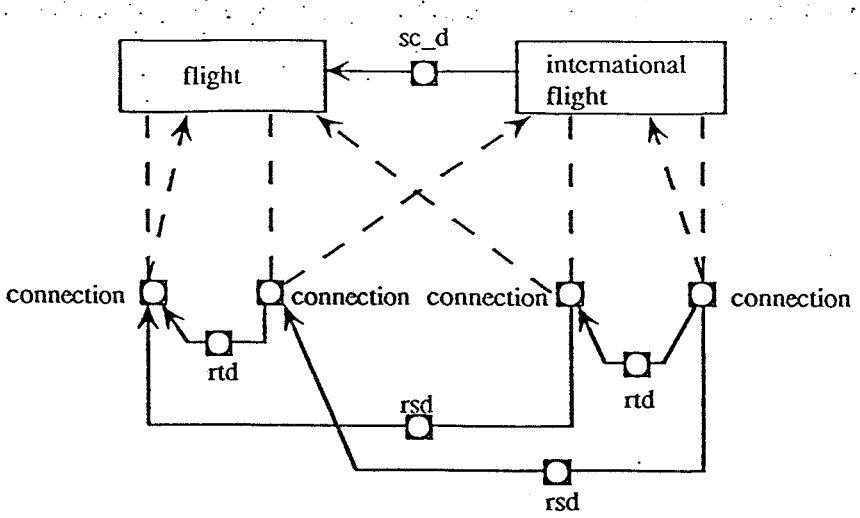

Figure 4.4: Redefinition on the Target Domain

4.2.3 More about Overriding

In the object-oriented context, the need to override attribute definitions with the same name is often associated with the problem of the multiple definition: along a branch of the hierarchy, one looks for a deterministic method selecting a single attribute definition. The relationship aims at specifying the redefinition of the attributes: the selected one redefines the others. The overriding feature is also applicable to the other conflictual situations, in particular in order to solve some clashes occurring with multiple inheritance and multiple representation. The overriding relationships declared between the attribute definitions provide an order relation, i.e. an explicit linearisation of the inheritance graph in accordance to a given attribute definition name. In such a way, the overriding links can be exploited to define the multiplicity (see 4.1) among several attribute definitions. The role of the overriding is then to mask other definitions for the sub-hierarchy. This feature is illustrated by the following example.

Example 4.2.3 **We consider the hierarchy of the figure 4.5. The class `student` is a subclass of both `person` and `insured` (the insured items of the application). `car` is a subclass of `insured`. We define also two age attributes, one related to the age of the persons (an integer from 0 to 100 for instance), the other related to the insured objects (intervals). We want the students to be considered as persons, at least concerning the age attribute definition. Hence, we must specify an order relation between the two definitions as we want the age attached to persons, to mask the one attached to the insured objects. This order is set by an overriding link. Every entity, instance of `student`, inherits the age definition of the persons:**

```
?- eval([isa(caroline, Y)]).
Y = person
        more? -- ;
Y = insured
        more? --
yes
?- eval([age(caroline, Y)]).
```

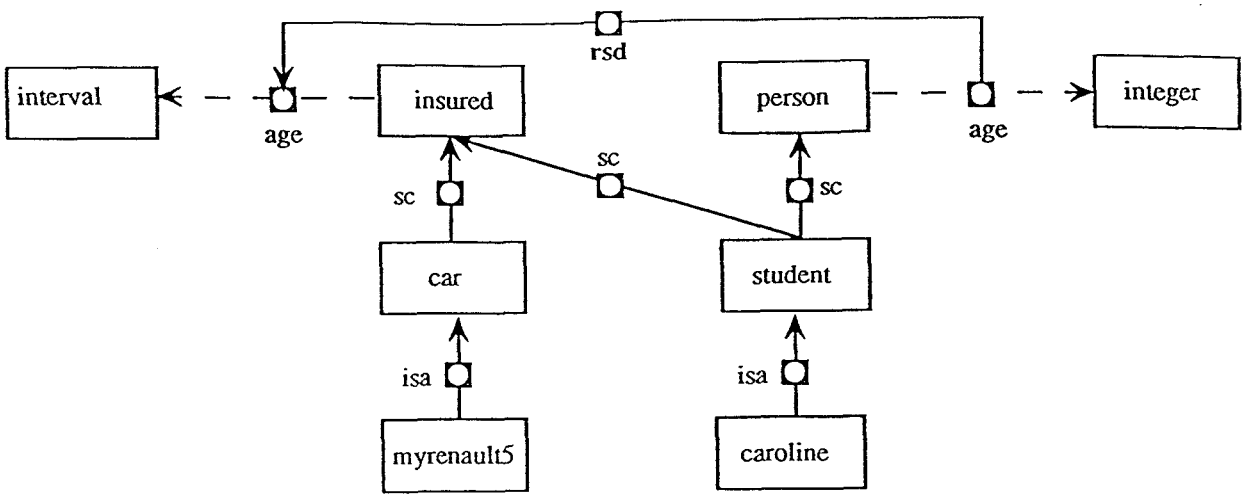

Figure 4.5: More about Overridding

```
Y = 19
        more? -- ;
yes
?- eval([age(myrenault5, Y)]).
Y = [5, 9]
        more? -- ;
yes
```

◇

Generally, the multiplicity relationship is defined at the class level: the order is set between the classes and therefore, the associated attribute definitions are ranked (if several are available) thanks to the same principle. This generality can produce unreasonable behaviour. In this context, the order between same-named attribute definitions is strictly defined by the order between the classes. As every definition reflects an atomic information, we consider that the order must be defined between the attribute definitions themselves. This approach allows also the specification of exceptions in the inheritance hierarchy in the same way as exception links modify the inheritance resolution mechanism in some knowledge representation systems (e.g. [Duc88]).

Example 4.2.4 Let us assume we build a simple classification of animals (cf. figure 4.6). We define two sub-classes of the class `animal` : `mammal` and `oviparous`. We give the two attribute definitions :

- the reproduction mode (`reproduction`): gestation for the mammals and laying of eggs for the ovipara. We define it intentionally as a kind of default value for the class representatives.
- the way the animal feeds its young (`feed`): milk for the mammals, predigested food for the ovipara, also defined intentionally.

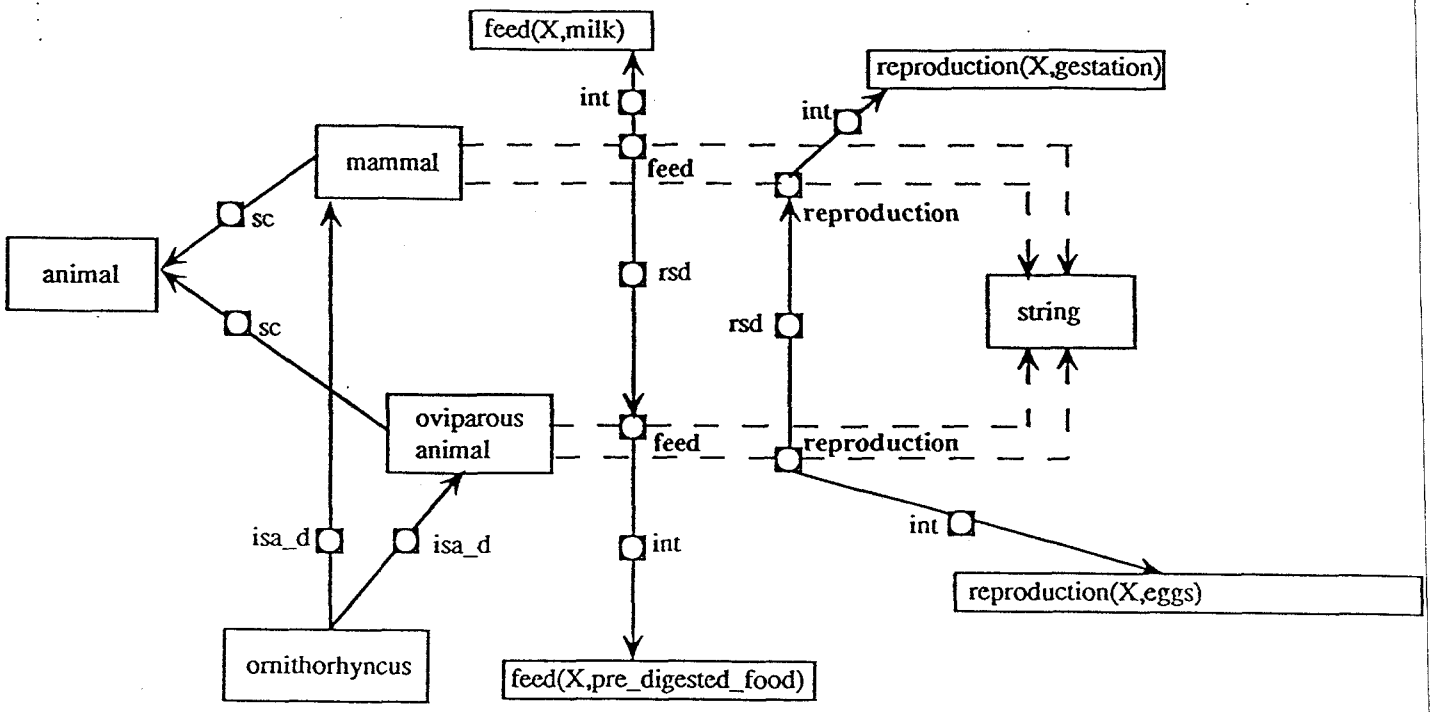

Figure 4.6: The Duck-billed Platypus Example

But, it happens that we want to introduce the ornithorhynchus (also called duck-billed platypus), the strangest animal of the world, which is both a mammal and an oviparous: it lays eggs and feeds its young with milk! The problem can be solved by setting redefinition links between the attribute definitions `reproduction` from the ovipara to the mammals, and on the other hand, between the attribute definitions `feed` from the mammals to the ovipara. We defined in this way two local multiplicity relations between the attribute definitions. These links are taken into account only for the ornithorhynchus. Other "normal" animals benefit by the classical definitions. ◇

As pointed out above, the overriding mechanism provides the designer with a powerful tool for a wide variety of inheritance clashes thanks to static declarations included in the model. However, these techniques are often not sufficient from a semantical point of view: some behaviours cannot be declared merely or, for other cases, a static declaration is too restricted. For instance, in the previous example (example 4.2.4), any other animal representative of the two classes, will be considered as laying eggs and feeding its young with milk: the static multiplicity set for a particular exception does not apply necessarily for all kind of exceptions. The limits of static declarations can be overcome by dynamical specifications of the context to be considered for the evaluation.

4.3 Views and Viewpoints

In the following, we call $\mathcal{E}$ the set of all entities manipulated in the system:

$$\mathcal{E} = \{E/isa(E, entity)\}$$

The set of classes is called $\mathcal{C}$:

$$\mathcal{C} = \{C/isa(C, class)\}$$

4.3.1 Views on the Class Hierarchy

In the situations of multiple inheritance and multiple representation, when conflicts are ignored, an entity may inherit from several classes attribute definitions with the same name. However, there obviously exits a subgraph of the class hierarchy in which the conflicts disappear (cf. figure 4.2).

Views on the class hierarchy are a means to select, dynamically during evaluation, a coherent subgraph for a given entity. All the attributes of this entity will only be considered if they are inherited in this subgraph. This does not avoid the conflicts automatically. It is, however, a safe means to control inheritance during evaluation. It is safe since the selected subgraph is always coherent: at least it always contains the class `entity` and, therefore, the entity is an entity (cf. chapter 3). Let us try and formalize this notion of views. First of all, we give some definitions to deal with the classes and the instances. We borrowed this notion from [Car89]. The author call them *viewpoints*. Our understanding of views and viewpoints being sightly different, we adopt a different terminology.

Definition 4.3.1 (representation set [Car89]) *The representation set of an entity E is the set of all classes of which E is an instance:*

$$Rep(E) = \{C \in \mathcal{C}/isa(E, C)\}$$

◇

Definition 4.3.2 (subclasses set of a class [Car89]) *The subclasses set of a class C is the set of all classes of which C is a super class:*

$$\forall C \in \mathcal{C}, Sub(C) = \{C1 \in \mathcal{C}/sc(C1, C)\}$$

◇

Definition 4.3.3 (super classes set of a class [Car89]) *The super classes set of a class C is the set of all classes of which C is a subclass:*

$$\forall C \in \mathcal{C}, Sup(C) = \{C1 \in \mathcal{C}/sc(C, C1)\}$$

◇

Definition 4.3.4 (dependence set of a class [Car89]) *The dependence set of a class C is the set of all classes of which C is a subclass or a super class; C belongs to its dependence set:*

$$\forall C \in \mathcal{C}, Dep(C) = \{C1 \in \mathcal{C} / sc(C, C1) \text{ or } sc(C1, C)\} \cup \{C\} = Sub(C) \cup Sup(C) \cup \{C\}$$

◇

A view for a given entity `E`, according to a class `C`, is the dynamical restriction of the class hierarchy, for the inheritance computation, to the dependence set of `C`. A query `att_name(E @ view(C), X)` indicates that the computation must be restricted to `Rep(C)` for inheritance on `E`. Concretely:

$$isa(E@view(C), C1) \text{ is true if and only if } isa(E, C1) \text{ and } C1 \in Dep(C).$$

In other words, the representation set of the entity under a class is restricted to its intersection with the dependence set of the class. This set `Rep(E @ view(C))` is never empty as it always contains `entity`.

Definition 4.3.5 (representation set of an entity under a class view) *The representation set of an entity E under a class C is:*

$$\forall C \in \mathcal{C}, \forall E \in \mathcal{E}, Rep(E@view(C)) = Dep(C) \cap Rep(E)$$

◇

The viewpoints described in [Car89] are similar. However, a stronger restriction is made on the validity of views (called viewpoints in that context). Because of the "frozen and unique representation constraint"[1], the direct instance link plays a stronger role. As a consequence, the instantiation class (unique in that case) must belong to the selected subgraph: the direct instance link is frozen and must be interpretable (not hidden by the view). In that case if there is a link `isa_d(E, C)`, a view (viewpoint) `C` is valid if and only if $C \in Rep(C1)$. For us, the `isa_d` link is an implementation link, the semantic link being `isa`.

In the case of extensionally defined attributes, the view only serves the purpose of selecting their definition in the class hierarchy. However, when a selected definition consists of an intentional part, the view is propagated in the body of the rule, wherever the concerned entity appears. It means that the subgraph is used not only for the selection of the attribute definition for the initial query, but also for every attribute

[1] In French: CRUF, contrainte de représentation unique et figée.

definition needed in the computation, when it involves the entity on which the view is stated.

Views can be used in queries. They can also be used within the rules in the intension of the attribute classes. The problem, then, is the composition of views. In the example 4.3.1, we show what we naturally expect from this composition.

Example 4.3.1 Considering the schema of the figure 4.7, John wants to fill the tax file. He has the choice to declare himself as a `teenager` or as a `middle aged` as he is 19 years old. He chooses `teenager` because he knows that he may get a discount in this case. However, both taxes are computed under the view `adult` when searching the income. Indeed, children incomes are pocket money. Consider the initial query:

```
?-eval(tax(john @ view(teenager), X).
```

The selected subgraph contains the classes `person`, `child`, `adult` and `teenager`. However, when using the rule:

```
tax(X, Y) <- income(X @ view(adult), I), ..., f(..., I, Y).
```

one generates the subquery:

```
income((john @ view(teenager)) @ view(adult), I).
```

The answer is computed in the view intersection of `teenager` and `adult`. The view under `adult` contains the classes `person`, `adult`, `teenager` and `middle aged`. The resulting view contains the classes `person`, `adult` and `teenager`.

When the two composed views are unrelated, they exclude each other and the intersection is like the one illustrated by figure 4.8. It represents the composition of the views `view(C2)` and `view(C3)`. ◇

Views represent subgraphs of the class hierarchy, sets of classes. Composing views naturally corresponds to valid operations on graphs and sets: union and intersection. We must notice that the complementary operation is not valid since the resulting graph cannot contain `entity`. We define two operators on views: . and +, respectively representing the intersection and the union. They are defined in terms of the representation set of an entity under the composed views.

Definition 4.3.6 $\forall E \in \mathcal{E}, \forall V1, V2$ *views :*

- $Rep(E@(V1.V2)) = Rep(E@V1) \cap Rep(E@V2)$
- $Rep(E@(V1 + V2)) = Rep(E@V1) \cup Rep(E@V2)$

◇

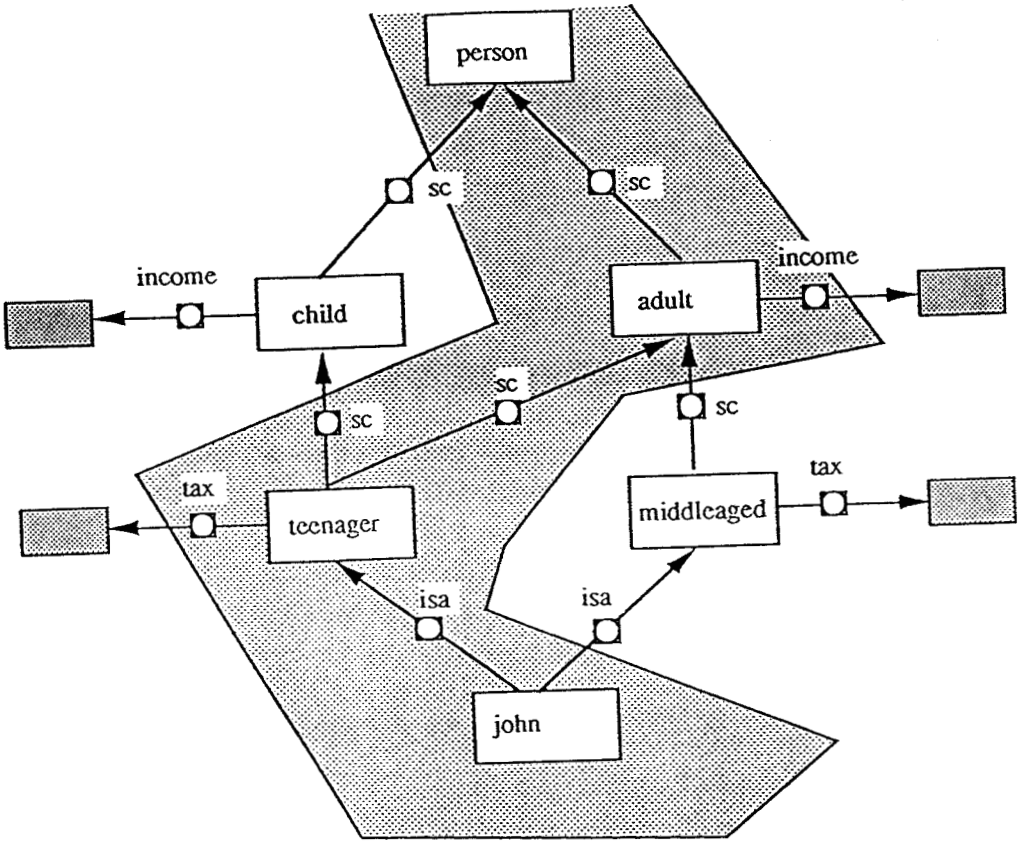

Figure 4.7: View Composition

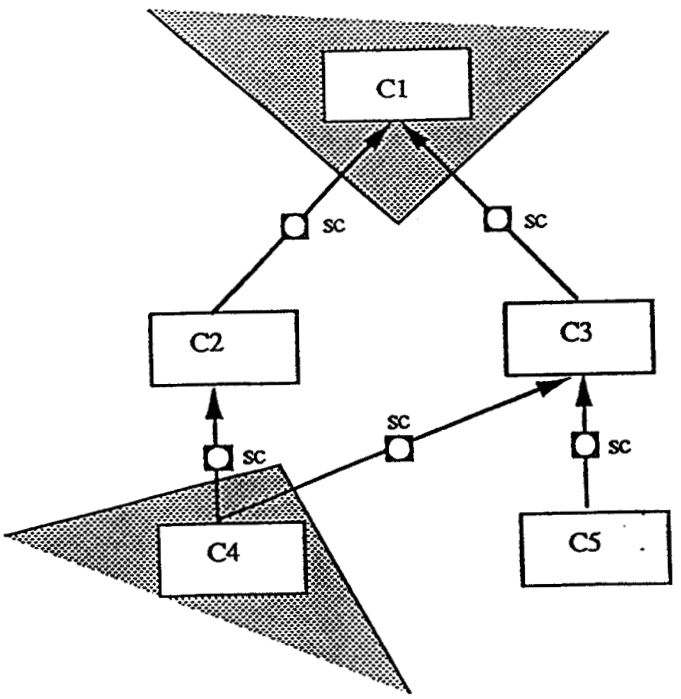

Figure 4.8: View Composition with Unrelated Classes

We have seen that the expected feature of view composition through the rules is the double selection of a subgraph by the two views. View composition is translated into an intersection:

$$Rep((E@V1)@V2) = Rep(E@(V1.V2))$$

A composed view does not, in general, correspond to a view under an existing class. Neither does it correspond to a view under a virtual class: a non existing class whose position in the hierarchy can be computed systematically. In the section 4.3.3, we define views with more complex expressions than simple classes: viewpoints. Indeed, viewpoints are virtual classes. The question whether the user should be allowed to manipulate view composition explicitly is not yet clear for us. We have seen that, if views are allowed in the rules body, then, at least, intersection must be computable by the system. The problem is that we provide a mechanism for building complex views, the viewpoints, that is founded by the semantics of the class hierarchy and the class-subclass order. view composition and views under viewpoints often resemble one another, but they are very different: view compositions are operations on sets of classes, viewpoints are operations on classes. We have chosen to describe the two mechanisms. We are aware that the possible resulting confusion is a drawback.

4.3.2 Viewpoints

The class hierarchy is designed to reflect the whole application. The view mechanism described above is a flexible means to take a particular point of view during a session or for a query. The views are attached to certain entities and project, for them, the class hierarchy. However, views are built from the class hierarchy. The mathematical structure of the class hierarchy with the class-subclass order does not contain all the classes needed to built all the possible views. For instance, one may not be able to built a view from two classes considering their union or intersection (in the intuitive sense of the union of intersection of descriptions and instances). Indeed, the class hierarchy is not a lattice, and union and intersection of classes may not be classes. We present in this section a general extension of the notion of class. The viewpoint set and its partial order is built from the class hierarchy and the class-subclass order. We explore in the next section the use of viewpoints for defining complex views on the viewpoint hierarchy. Although we do not investigate this aspect here, one can remark that viewpoints could generalize the notion of class everywhere this latter appears: source and target domains of attributes, instance relation, etc.

We need to construct a hierarchy which is a natural extension of the class hierarchy and which is a lattice. Indeed, the lattice structure guarantees that all the compositions will be internal operations. The principle of the construction is the following:

- We start from the set $\mathcal{C}$ and the partial order $\leq_{\mathcal{C}}$. This partial order is defined by:

$$\forall C_1, C_2 \in \mathcal{C}\ (C_1 \leq_{\mathcal{C}} C_2 \Leftrightarrow sc(C_1, C_2) \text{ or } C_1 = C_2)$$

$\leq_C$ is a partial order, antisymmetric, reflexive and transitive, because of the definition of the subclass relation on C.

- A set, $C\mathcal{V}$, is syntactically built from C and the functors $\vee$, $\wedge$ and $\uparrow$[2]. The new set is ordered by an extension of the $\leq_C$ order: $\leq_{C\mathcal{V}}$. $\leq_{C\mathcal{V}}$ is a pre-order, it is reflexive and transitive but not antisymmetric.
- The antisymmetry is due to the syntactical construction. There exist elements of $C\mathcal{V}$ intuitively equivalent that are different. Therefore, we built a third set, $\mathcal{V}$, quotient of $C\mathcal{V}$ by the equivalence relation. For this set the new relation $\leq_{\mathcal{V}}$ is an order that confers $\mathcal{V}$ a lattice structure.

Finally we verify the properties of $(\mathcal{V}, \leq_{\mathcal{V}})$ and show that it is the needed extension of $(C, \leq_C)$.

Definition 4.3.7 (viewpoints) *A viewpoint is an expression built from classes and connectives ($\vee$, $\wedge$ and $\uparrow$). The set of well formed viewpoints $C\mathcal{V}$ is defined by:*

- $\forall C \in C, C \in C\mathcal{V}$;
- $\forall V1, V2 \in C\mathcal{V}, (V1 \vee V2) \in C\mathcal{V}$;
- $\forall V1, V2 \in C\mathcal{V}, (V1 \wedge V2) \in C\mathcal{V}$;
- $\forall V \in C\mathcal{V}, (\uparrow V) \in C\mathcal{V}$.

$\diamond$

We can now define the extensions of $\leq_C$ (sc) on $C\mathcal{V}$:

Definition 4.3.8 ($\leq_{C\mathcal{V}}$) *The $\leq_{C\mathcal{V}}$ link on $C\mathcal{V}$ is defined by:*

$\forall V1, V2 \in C\mathcal{V}, V1 \leq_{C\mathcal{V}} V2 \Leftrightarrow$

- *$V1, V2 \in C$ and $V1 \leq_C V2$;*
- *or, $V1$ is $V11 \wedge V12$ and $V11 \leq_{C\mathcal{V}} V2$ or $V12 \leq_{C\mathcal{V}} V2$;*
- *or, $V1$ is $V11 \vee V12$ and $V11 \leq_{C\mathcal{V}} V2$ and $V12 \leq_{C\mathcal{V}} V2$;*
- *or, $V1$ is $\uparrow V11$ and $V11 \leq_{C\mathcal{V}} V2$;*
- *or, $V2$ is $V21 \wedge V22$, and $V1 \leq_{C\mathcal{V}} V21$ and $V1 \leq_{C\mathcal{V}} V22$;*
- *or, $V2$ is $V21 \vee V22$ and $V1 \leq_{C\mathcal{V}} V21$ or $V1 \leq_{C\mathcal{V}} V22$.*

$\diamond$

[2] $\uparrow$ is not needed for the lattice structure but is very useful for the view selection as we will see later.

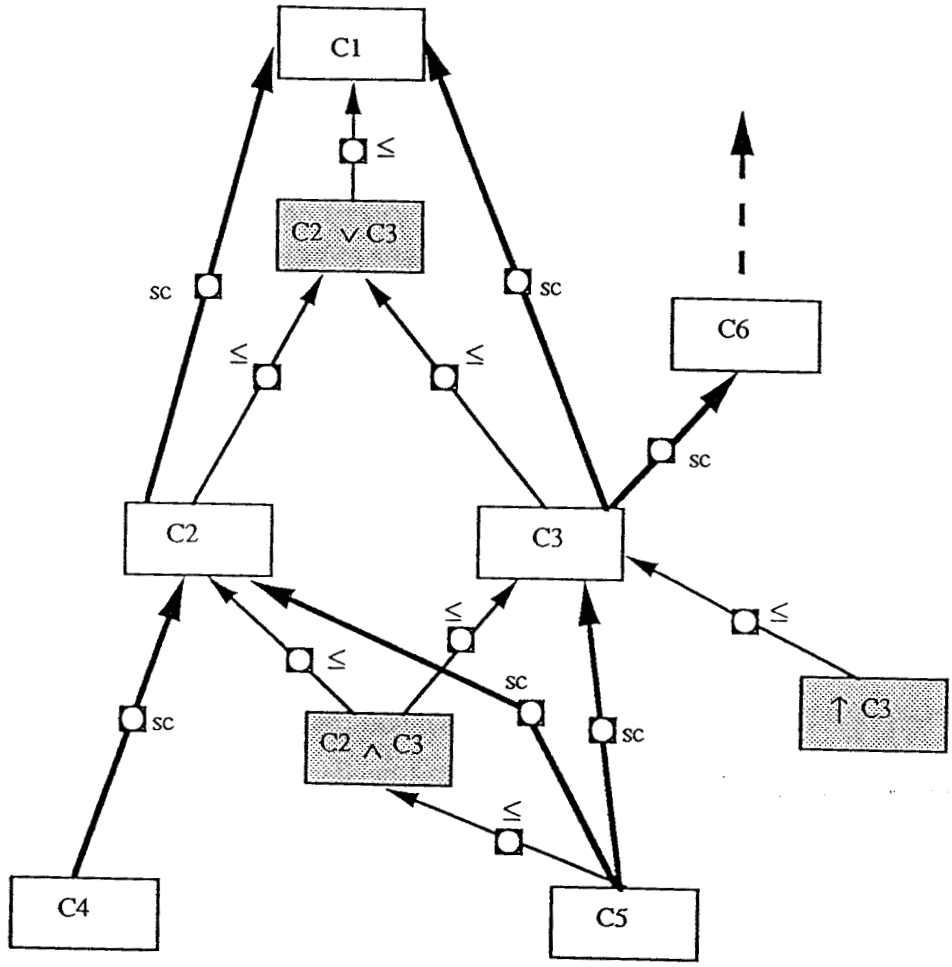

Figure 4.9: The $\leq_{\mathcal{CV}}$ Hierarchy

Considering classes and viewpoints as sets of instances, $\wedge$, $\vee$ and $\uparrow$ could be seen respectively as the intersection, union and copy of their arguments. We do not give the straightforward definition of the extension of the `isa` link to the viewpoint hierarchy.

The viewpoint hierarchy is an extension of the class hierarchy. The connectives $\wedge$, $\vee$ syntactically intend to build, respectively, the greatest lower and the lowest upper bounds of their arguments. A simple equivalence relation on viewpoints will make this point clear. $\uparrow$ creates a virtual copy of its argument that has no subclasses. This feature will be useful when one wants to consider classes without their specializations in a view. The figure 4.9 illustrates the viewpoint hierarchy.

In fact, $\mathcal{CV}$ with the $\leq_{\mathcal{CV}}$ relation has not got a satisfying structure. For instance, for two classes $C1$ and $C2$ the two viewpoints $C1 \wedge C2$ and $C2 \wedge C1$ are such that $C2 \wedge C1 \leq_{\mathcal{CV}} C1 \wedge C2$ and $C1 \wedge C2 \leq_{\mathcal{CV}} C2 \wedge C1$. They clearly represent the same thing. In order to solve this problem, we build a set $\mathcal{V}$ from $\mathcal{VC}$ and the equivalence relation $\equiv_{\mathcal{CV}}$.

Definition 4.3.9 (equivalence relation) $\forall V_1 V_2 \in \mathcal{CV}$:

$(C_1 \equiv_{\mathcal{CV}} C_2 \Leftrightarrow C_1 \leq_{\mathcal{CV}} C_2$ *and* $C_2 \leq_{\mathcal{CV}} C_1)$ $\diamond$

Property 4.3.10 (equivalence relation) *The following properties stands on the viewpoints:*

- $\forall V \in \mathcal{CV}$:
  - $V \vee V \equiv_{\mathcal{CV}} V$
  - $V \wedge V \equiv_{\mathcal{CV}} V$
- *commutativity:* $\forall V1, V2 \in \mathcal{CV}$:
  - $V1 \vee V2 \equiv_{\mathcal{CV}} V2 \vee V1$
  - $V1 \wedge V2 \equiv_{\mathcal{CV}} V2 \wedge V1$
- *associativity:* $\forall V1, V2, V3 \in \mathcal{CV}$:
  - $V1 \vee (V2 \vee V3) \equiv_{\mathcal{CV}} (V1 \vee V2) \vee V3$
  - $V1 \wedge (V2 \wedge V3) \equiv_{\mathcal{CV}} (V1 \wedge V2) \wedge V3$
- *distributivity:* $\forall V1, V2, V3 \in \mathcal{CV}$:
  - $V1 \vee (V2 \wedge V3) \equiv_{\mathcal{CV}} (V1 \vee V2) \wedge (V1 \vee V3)$
  - $V1 \wedge (V2 \vee V3) \equiv_{\mathcal{CV}} (V1 \wedge V2) \vee (V1 \wedge V3)$
- *absorption :* $\forall V1, V2 \in \mathcal{CV}$:
  - $V1 \vee (V1 \wedge V2) \equiv_{\mathcal{CV}} V1$
  - $V1 \wedge (V1 \vee V2) \equiv_{\mathcal{CV}} V1$

∇

Definition 4.3.11 ($\mathcal{V}$) ***We can now define the set $\mathcal{V}$, we will call abusively the set of viewpoints:***

$$\mathcal{V} = \mathcal{CV} / \equiv_{\mathcal{CV}}$$

$\diamond$

The set $\mathcal{V}$ contains an element per class of $\mathcal{C}$ and new elements representing the union and intersection of classes. The constructors $\vee$ and $\wedge$ can be seen as operators in the lattice structure. Indeed, for the induced relation $\leq_{\mathcal{V}}$, $\mathcal{V}$ has a lattice structure:

Definition 4.3.12 ($\leq_\nu$) $\forall V_1 V_2 \in \mathcal{V}, (V_1 \leq_\nu V_2 \Leftrightarrow \forall v1 \in V_1$ *and* $v2 \in V_2, v1 \leq_{C\nu} v2)$

$\diamond$

Property 4.3.13 *$(\mathcal{V}, \leq_\nu)$ has a lattice structure.* ∇

Proof 4.3.13 *The proof consists in showing that, for each element of the set $\mathcal{V}$ containing $V1$ and $V2$ of $C\mathcal{V}$, the elements containing $V1 \wedge V2$ and $V1 \vee V2$ are respectively the infimum and supremum (therefore unique) for $\leq_\nu$.* Δ

The case of the operator $\uparrow$ remains unclear. However, as we intend to use it for the selection of views, we can abusively assume the following properties:

Property 4.3.14 (distributivity) *The following properties can be used:*

- *distributivity:* $\forall V1, V2 \in C\mathcal{V}$*:*
  - $\uparrow (V1 \vee V2) \equiv_{C\nu} (\uparrow V1 \vee \uparrow V2)$
  - $\uparrow (V1 \wedge V2) \equiv_{C\nu} (\uparrow V1 \wedge \uparrow V2)$
- *idempotence:* $\forall V \in C\mathcal{V}$*:* $\uparrow\uparrow V \equiv_{C\nu} \uparrow V$

∇

Then, the syntactical construction and the lattice structure allows us to exploit a normal form for viewpoints. This is very interesting for the implementation of a view mechanism using viewpoints:

Property 4.3.15 *$\forall V \in C\mathcal{V}$, there exists a canonical transformation into a canonical form CV such that $CV \equiv_{C\nu} V$.* ∇

The definition of the equivalence relation and its properties are used as the basis for the implementation of views with viewpoints in the Möbius evaluator.

4.3.3 Views on the vViewpoint Hierarchy

Views are used to select a subgraph of the inheritance graph. In section 4.3.1, we defined a view on an entity by means of a class. Now, we dispose of a more precise notion: viewpoints. We have seen that viewpoints are a generalization of the class hierarchy to a lattice structure (plus some more information like $\uparrow C$). This section gives the necessary definition for defining views on entities according to the viewpoint hierarchy $(\mathcal{V}, sc)$. sc is the name we use now for the $<_{C\nu}$ order.

Definition 4.3.16 (super classes set of a viewpoint) *The subclasses set of a viewpoint V is the set of all classes of which V is a super class (with the* sc *relation):*

$$\forall V \in \mathcal{V}, Sub(V) = \{C \in \mathcal{C} / sc(C, V)\}$$

◇

Definition 4.3.17 (super classes set of a viewpoint) *The super classes set of a viewpoint is the set of all classes of which V is a subclass (with the* sc *relation):*

$$\forall V \in \mathcal{V}, Sup(V) = \{C \in \mathcal{C} / sc(V, C)\}$$

◇

Definition 4.3.18 (dependence set of a viewpoint) *The dependence set of a viewpoint V is the set of all classes of which C is a subclass or a super class (with the* sc *relation) ; V belongs to its dependence set if it is a class:*

$$\forall V \in \mathcal{V}, Dep(V) =$$
$$\{C \in \mathcal{C} / sc(V, C) \text{ or } sc(C, V)\} \cup \{V \in \mathcal{C}\} =$$
$$Sub(C) \cup Sup(C) \cup \{C \in \mathcal{C}\}$$

◇

A view, for a given entity `E`, according to a viewpoint `V`, is the dynamical restriction of the class hierarchy, for the inheritance computation for `E`, to the dependence set of `V`. A query `att_name(E @ view(V), X)` indicates that the computation must be restricted to `Rep(V)` for inheritance for `E`. Concretely:

$isa(E@view(V), C)$ is true if and only if $isa(E, C)$ and $C \in Dep(V)$.

In other words, the representation set of the entity under a class is restricted to its intersection with the dependence set of the class. This set `Rep(E @ view(V))` is never empty as it always contains `entity`.

Definition 4.3.19 (representation set of an entity under a viewpoint view) *The representation set of an entity E under a viewpoint V is:*

$$\forall V \in \mathcal{V}, \forall E \in \mathcal{E}, Rep(E@view(V)) = Dep(V) \cap Rep(E)$$

◇

The rules interpreting a view are:

```
isa(E @ view(V), C) <- isa(E, C) and sc(C, V).

isa(E @ view(V), C) <- isa(E, C) and sc(V, C).

isa(E @ view(C), C).
```

4.4 Full-name

The principle of the association of attribute definition to classes allows, in particular, to ignore, thanks to inheritance, where this information comes from. It allows a user to manipulate the knowledge without a global view of its structure. For instance, when querying an entity `jean` about his age: `?- eval([age(jean, X)])`, the user does not need to know if the age is an attribute defined for all entities, for animals or just for persons. He is only concerned with the fact that `jean` inherits this definition.

When programming an application, the designer can use overriding to control inheritance. When querying the knowledge base, the user can use views and viewpoints to control inheritance. There remain cases where a strict and exact control of the origin of the attribute definitions is needed.

We must recall here, that attributes definitions, in Möbius, are identified by the triple: (name, source domain, target domain). Therefore, in any expression, designating an attribute by its name may select several definitions, while designating an attribute by its full-name (source, target domain and name) will select a single definition. The Möbius syntax for full-name is the following:

```
sd(d1)!td(d2)!attribute_name(X, Y).
```

where d1 and d2 are respectively a source and target domain of the attribute.

We also allow a partial full-name (!) using only source or target domains:

```
sd(d1)!attribute_name(X, Y).

td(d2)!attribute_name(X, Y).
```

Taking advantage of unification, one can also ask queries where source or target domains are variables that are bound when the expression is evaluated.

Chapitre 5

Instance and Identity

Instance et identité

Les principales difficultés pour la définition d'un modèle de représentation des connaissances dans un cadre relationnel et déductif sont, d'une part, la gestion de la notion d'identité et, d'autre part, la gestion de la notion d'instance. Ces difficultés sont liées à l'opposition entre le modèle relationnel orienté valeurs et les modèles dits objets orientés références.

Dans un premier temps, nous discutons quelle quantité d'information sémantique peut être associée à l'identificateur d'une entité. Les deux possibilités que nous envisageons sont les identificateurs structurés et les attributs identifiants.

Ensuite, nous présentons les facilités offertes par le modèle Möbius pour la définition du lien d'instance `isa`. *En particulier, nous montrons comment ce lien peut être défini intentionnellement.*

The purpose of knowledge base models is to represent objects and concepts of the real world. An object in the real world exists in the model via an entity it is represented by. An object in the real world exists in the model because meaning is attached to that entity; basically, the entity is recognized by the model. These considerations lead us to several issues:

- First of all we examine the opposition between the value-oriented nature of the relational context on top of which the Möbius system is built and the reference-oriented nature of conceptual and object-oriented approaches;
- We briefly discuss several points related to the notion of identity:
  - how much semantic information should an identifier contain, if any?
  - how to maintain and use semantic information for identification?
- finally, we discuss how the `isa` instance relation is defined and represented in the Möbius model.

5.1 Reference vs Value

The deductive database technology has been developed on top of the standard relational model. The relational model is value-oriented, therefore, a deductive database model is also value-oriented. Entities of the real world are represented as tuples or terms and assimilated to their value. E.g., a person with a name, a forename and an age would be represented by a tuple of the relation `person`: `(porter, graham, 25)`. Deciding, if the age changes, whether the person is the same or not is of the user interpretation responsibility.

On the other hand, by merging the notion of concept used in artificial intelligence models -nodes of semantic networks, components of conceptual graphs, etc - and the identity notion present in most programming languages - variables in Pascal -, the object-oriented models are reference-oriented. An object consists, at least, of an object identifier, a type and a value. The value is again composed of identifiers. In the following example we compare the reference and value-oriented representations.

Example 5.1.1 Let us consider a database of suppliers and parts they supply to a given department. A supplier has a name and an address. A part has a type and a price.

quantity	type	price	supplier	address	department
1000	bolt	0.50	pfund gmbh	munich	chair dep
2500	nail	0.60	dupont fils	paris	table dep
2000	bolt	0.40	robinson co	london	chair dep
2500	nail	0.40	robinson co	london	cupboard dep
1500	bolt	0.50	pfund gmbh	munich	chair dep
1000	nail	0.60	dupont fils	paris	cupboard dep
2000	nail	0.40	pfund gmbh	munich	cupboard dep
2500	nail	0.40	robinson co	london	cupboard dep
2000	screw	0.50	dupont fils	paris	table dep
1500	bolt	0.40	robinson co	london	table dep
1500	screw	0.50	pfund gmbh	munich	table dep
2000	bolt	0.50	pfund gmbh	munich	cupboard dep
1000	bolt	0.30	wang tsu	singapour	cupboard dep

A possible object-oriented representation, in the syntax of the model proposed in [Abi90][1], would be:

```
class address: string
class name: string
class type: string
class supplier:  [name: name, address: address]
class part: [type:type, cost: real]
class order: [part: part, quantity: integer, supplier: supplier]
class department: [name: name, orders: {order}]
```

To this schema correspond, for instance, the objects:

```
oid: #22
    type: supplier
    value: [name: "wang tsu", address: "singapour"]
oid: #44
    type: part
    value: [type: "bolt", price: 0.30]
oid: #26
    type: order
    value: [part: #44, quantity: 1000, supplier:#22]
oid: #35
    type: department
    value: [name: "cupboard dep",
            orders: {#26, #27, #28, #29, #30, #31}]
```

Oids are similar to keys added to relations. However they are integrated in the model and do not necessary reflect the implementation in tables. They are systematically manipulated by the model. ◇

[1]We do not explain the syntax. It is natural enough. If any understanding problems remain one can refer to the cited document

The word 'object identity' only appeared in the database vocabulary when studies were led to define object-oriented database models. However, the necessary link between the data in the database and entities of the real world forced data modeling research to concentrate on, even when not named, referential integrity and identity. Relational modeling is based on normalization. The several normal forms intend to protect the data from the defaults mainly due to the lack of identity: update anomaly, insertion anomaly, deletion anomaly. In the example 5.1.2 we illustrate these possible defaults.

Example 5.1.2 In the table given in example 5.1.1, there exist obvious dependencies among the data. The problem is to protect the integrity of the data under these dependencies when constrained by the structure of the relations chosen to store the data. In particular update operations may lead to inconsistent states.

- Update: if the address of a supplier changes, it is necessary, in the table of the example, to replace it everywhere it appears, because of the natural functional dependency between a supplier and its address.
- Insertion: a new supplier can only be added if it supplies a part.
- Deletion: the deletion of all bolts delete the supplier `wang tsu`.

Of course a solution to the second and third anomalies would be to use null values. However their use and semantics complicate the semantics of the model.

One can verify that none of these anomalies appears in the object-oriented representation. ◇

Normalization is based on the definition, in the relational model, of dependencies, i.e., integrity constraints. Codd, in the early seventies, when describing the basis of the relational data model, proposed the use of so called identifying keys. They represent the first attempt to integrate identity in the relational database model. Of course the hierarchical and network models had, already, a strong notion of identity because of the lack of independence between the data and the storage structure.

However the wished notion of identity should appear at the conceptual level. Indeed, the absence of a management of identity creates a mismatch between the application and its representation in the model. Providing solutions for this problem, at the conceptual level, was one of the purposes of models like the Entity-Relationship model (ER) or the extended Entity-Relationship model (EER). They provide modeling capabilities to express some of the natural constraints. In particular defining structured entities (with attributes) associated together brings clearly the problem of identity at the modeling level. However, the problem remain to find a mapping into a relational model that respect the implied constraints.

The same situation stands for object-oriented database models. Although the notion of identity exists, the question remains how to represent it in the storage model. For that purpose so called non first normal form (N1FN) models have been investigated. What we are interested in is the integration of object identity at the level of the model.

An entity is identified by an identifier. The identifier is an entry point to the value computation. The identifier, as a simple pointer, or as a block of information, must be

permanent in time and space. In time, because, within its life time, it represents the same real world entity. In space, because there does not exist two real world entities with the same identifier. Names are the first natural means to create identifiers. It is however easy to see that meaningful names are not always sufficient. A well known technique to integrate identifiers in a relational model is the use of surrogates. It corresponds to an implicit version of keys in a relational model. In Möbius, we choose to use surrogates to identify entities. The identifier generation and management must be guaranteed by the system. In the Möbius prototypes, the user has the choice, when creating a new entity to give the identifier explicitly - with the risk that it already exists -, or to let the system generate a new one [2]:

```
?- employee :: new(employee1).

yes
?- employee :: new(_).

yes
?- employee :: new(X).
X = o240

yes
```

5.2 Identifiers

5.2.1 Structured Identifiers

A minimal assumption, in a model with object identifiers, is that there exists, at least, some objects that contain their own meaning. These objects are necessary for the model to communicate with the user or with other systems. A classical solution consists in building the objects from sets of basic values. Usually, these basic values sets or types are integers, strings, real numbers etc. More sophisticated systems provide the user with constructors for structured values: Cartesian products, set constructors, record or list constructors.

Therefore, in the triple `(Oid, Type, Value)`, the value can refer to other objects identifiers or to basic or complex values. In the example 5.1.1, we used the value sets `integer` and `string`.

This point of view, separating values from identified objects, can be relaxed. Indeed, values, in that case, can be seen as self contained objects. The symbol `1` is both the identifier and the value for the integer one. In the real distinction is not between identifiers and values but between *printable* and *non printable* identifiers. This distinction

[2]The syntax `entity :: procedure` has been used in the Möbius experiments. It indicates that the procedure is selected according to the class of the entity. Here `new/1` is a creation procedure. It is called from the Prolog top-level

depend on the application and on the user. For instance, in a railway application, the identifier `ic230` may be meaningful for users from the railway company where only the information that it is a train between `muenchen` and `koeln` that travel on `sunday` means something for other users. There, syntactic sugar added to the query and answer language, on the basis of the distinction between printable and non printable identifier, could free the user from the burden of manipulating meaningless identifiers. We have ,for instance, experimented a functional syntax where only the variables participating at the answer are needed (by expressing cross references with an equality symbol).

Even when not directly printable, identifiers may be more than just an internal code for the model. As soon as a way to structure identifiers and operators and operations are provided in the model, identifiers can be used to store permanent information. We could, for instance, choose to identify persons by a structure storing their name and surname. This structure could be any record structure, string or algebraic term in a logic programming context. The data language must then contains operators or operations on, respectively, record structures, strings or algebraic terms to retrieve information from them. In the case of terms, the operation could be matching or unification.

The risk, now, is that the power of the data language is such that we reach undecidable situations. In particular, this is the case if the system can invent identifiers. A first order logic language, with terms as identifiers and unification, is semi-decidable, it may create infinite structures and queries evaluation may not terminate. Therefore the use of structured identifiers must be limited and controlled. We can notice, here, that interesting studies about object identity in a deductive context use terms to disambiguate quantification in rules for identifier invention (cf. [CW89]).

We use terms for the implementation of certain complex information. For instance, classes and attribute classes intentions (formulae) are stored as structured terms. But from the model point of view they are seen as atoms.

However, there is a situation where identifier invention is integrated in the model. Namely this is the case when we claim that attributes are entities. Indeed, attributes are terms of the form `att_name(arg1, arg2)` that may appears in the body of queries with variables (`eval([v1(age(X, 25), Z)])`. The risks of creation of infinite structures or of non termination are eliminated by the fact that such expression may only appear in the body of rules and that the terms must correspond to literals evaluable by the Möbius evaluator.

5.2.2 Identifying Attributes

We have seen that structured identifiers, in certain cases, can be used to record a part of the information describing the entity. This can only be the case if the persistency of the information stored is guaranteed. There exists another situation, where some features of the entity are known to be identifying, but may evolve in the application life time. We could know for instance that persons are uniquely identified in the application by their name and surname, nevertheless being able to marry and change their family name. The name and surname are identifying attributes but not their values. Namely

the information we have is that there is a one to one correspondence between a group of attribute of the person and its identifier. The identifier, in that case, must be independent from the actual values of the attributes. This clearly corresponds to a special kind of integrity constraint (a functional dependency). We propose in the following a special syntax for it and a special mechanism to recover stable state.

The feature we propose consists in defining, at the class level, lists of identifying attributes. Such a list is stored in the class attribute `features`. This information correspond to the declaration of a functional dependency between a list of attributes and the entity identifier:

```
?- employee :: set(features([username(X,Y)])).

yes
?- employee :: set(features([name(X,Y), forename(X,Z)])).

yes
```

The system is now responsible to guaranty the uniqueness of the identifier for employees sharing the same features for every kind of update. The above statements are transformed into integrity constraints:

```
forall [X1,X2,Z1]:
        isa(X1,employee) and username(X1,Z1) and
        isa(X2,employee) and username(X2,Z1) ->
        X1 = X2

forall [X1,X2,Z1,Z2]:
        isa(X1,employee) and name(X1,Z1) and forename(X1,Z2) and
        isa(X2,employee) and name(X2,Z1) and forename(X2,Z2) ->
        X1 = X2
```

These constraints will be violated if an update does not respect the functional dependencies. The update will fail. The problem is to handle the recovery of a stable state where the constraints are no more violated. The simplest, but also the heaviest, solution consists in committing the update, despite the violation, finding the conflicting entities and merging them. The algorithm below is the first approximation of such a procedure:

```
recover(Update)
begin
 if only functional dependencies were violated
 then
   inhibit constraints
   perform(Update)
   reset constraints
   for each ordered pair (X1,X2) violating the dependencies
```

```
   do merge(X1,X2)
   enddo
 endif
end

merge(X1,X2)
begin
 for each tuple T(X2) containing X2
 do replace(T(X2),T(X1))
 enddo
end
```

The ordered pairs violating the dependencies constraints are obtained by rules created for each dependency constraint. For the example above, the rules are:

```
ic(X1,X2) <- isa(X1,employee) and username(X1,Z1) and
        isa(X2,employee) and username(X2,Z1) and not
        X1 = X2

ic(X1,X2) <-
        isa(X1,employee) and name(X1,Z1) and forename(X1,Z2) and
        isa(X2,employee) and name(X2,Z1) and forename(X2,Z2) and not
        X1 = X2
```

The pairs must be ordered since if o24 and o342 are violating the constraints then o342 and o24 also violates them. A lexicographic order could do the job. A refinement would consider that system generated identifier are always greater (or lower) than user defined identifiers.

The merge procedure replaces everywhere in the database the occurrences of the second identifier by the first. Notice here that the mapping information integrated explicitly in the schema should be used to reduce the number of tuple and relations to search.

However it might happen that a simple update generates a cascade of merging. Thus the replace procedure can call, in its turn, the recover procedure. The example 5.2.1 illustrates such a situation.

Example 5.2.1 Let us consider a class woman, a class man and a class person.Men and Women are identified by their children.

```
?- class :: new(person).

yes
?- person :: add_att(child,[td(person)]).

yes
?- woman :: features([child]).
```

```
yes
?- man :: features([child]).

yes
```

the database contains the following facts:

identifier	isa	children
annie	woman	{bernard}
dagmar	woman	{x1}
x2	woman	{annie}
francois	man	{dagmar}
bernard	man	{ }
x3	man	{x2}

adding the following fact:

```
?- woman :: new(x1, [children(bernard)]).

yes
```

will violates the dependencies and cause the merging of `x1`, `x2` and `x3` with, respectively `annie`, `dagmar` and `francois`. ◇

Another practical application of features consists in using them in the manipulation language to reference existing objects. Just as in queries, object identifiers can be replaced by a functional expression denoting an object. If the object already exists it is not created. If it does not a new identifier is generated. This is illustrated by the following creation in the context of the initial example:

```
?- employee :: new([username(tom)],
                   [name(barrow), forename(thomas)]).

yes
?-  employee :: new([name(barrow), forename(thomas)],
                    [age(28)]).

yes
```

It will create a single employee since both username and the pair `name forename` are identifying attributes for employees.

The problem of relating feature to inheritance remains. Attributes are recognized through their names. However a single entity can be linked by several different attributes of the same name. The question is, when defining features in a class, what

attribute definitions do we want to refer to: all the possible attribute definitions inherited by the object with the given name, only those inherited from the class and its super classes, etc.

In fact, we have chosen in Möbius to rely on the instance hierarchy and the `isa` link, instead of the class-subclass hierarchy, for inheritance. This leads us to the explicit removal of ambiguities. We suggest that attribute definitions used in features correspond to those inherited by the entity. In the several conflictual situations above, the same mechanisms used for inheritance in queries must be used: overriding, views and full-name. For example regular feature declarations could be:

```
man :: set(features([child(X @ view(^person),Y)]))

man :: set(features([sd(person)!child(X,Y)]))
```

5.3 Entities and Classes: the ISA Link

5.3.1 Being an Entity

As stated in the principles defining our model, everything has to be an entity. This is achieved by insuring that objects in the model are instances, at least, of the class `entity`. An entity is instance of a class if there exists a link named `isa` between the entity and the class.

The inheritance mechanisms, in Möbius, are founded on the isa link. This is given by the attribute definitions semantics by translation into rules. An interesting point to underline here is that inheritance, although exploiting the class hierarchy through the `sc` link, is not limited to the class hierarchy. Indeed, in some object-oriented models and systems, object are triple `(Oid,T,V)` where `Oid` is the object identifier, `T` is a type and `V` is the value. The type of objects is defined at the level of classes by the computation of the greatest lower bound of all the super classes of a class. The difficulties of this approach arise when conflicts appear, as an order with good properties is needed to compute the greatest lower bound. In our point of view, the class hierarchy is flatten to form the instance hierarchy (where each instance is related to the classes it represents).

One of the consequences of that choice is that the implementation instance link, we call the direct instance link (`isa_d`), is not necessary. The reason for such a particularization of the direct instance link is often related to implementation problems. This is called the unique and fixed representation constraint (c.f. [Car89]) and may lead to difficulties when one wants to see objects evolving or being representative of several unrelated classes (multiple representation).

Therefore we only need a `isa` link. For simplification purposes, we have and may keep on using the `isa_d` link in the text.

Now, we can see how the instance link is stored (classes intention) or computed (classes extension).

5.3.2 Class Extension

If we follow the principles we used for attribute definitions, then the `isa` class is associated to a mapping expression: the extension. It could be for instance, a binary relation `rel_isa` relating entities and classes. It would be, however, natural to consider that different classes have different mapping. For instance, one may want to store the persons in a relation `person/1`. This could be possible, using the extension of attributes if we defined several `isa` classes, one for each class with a new mapping. However, these attribute classes are inherited when the information about mapping must be local to a class. We preferred an alternative solution. Classes are associated with a class extension, an attribute named `class_ext` which is the parameter for the unique `isa` attribute class intention. As an example, the class `person` would be associated with the class extension: `class_ext(person,ext(S,[person(S)])`, where `person(S)` is a relational expression (a base relation). The `isa` definition becomes:

```
isa(X,Y) <- class_ext(Y,ext(X,[Ext])),Ext.
```

Thanks to meta-classes and intentional attributes, the `class_ext` can be set to a common relation for several classes. A meta-class could declare an `class_ext` in intention:

```
class_ext(X,Y) <- Y = ext(S,[r_isa(S,X)]).
```

All the classes, instances of that meta-classes will have an attribute `class_ext` set to the relation `r_isa/2`.

Thus, there is a unique `isa` link, but depending on the class, its second argument, it has a distinct extension. Just like normal attributes the `isa` will be compiled into rules. In fact, classes will be compiled and create the `isa` compiled rules. These rules are added to the recursive rule defining the `isa` link as its own transitive closure under the class hierarchy (`sc` link).

5.3.3 Class Intention

There are several situations where being an instance of a class is an information that can be derived from other knowledge in the model. This is the case at least for all the super classes of the class an entity is instance. But there are more applicative examples. For instance, saying that teenagers are persons whose age is between 13 and 19.

Therefore, as we had, for attribute classes, extensions and intentions, we can define, after class extensions, class intentions. They are formulae defining conditions to be instance of a class. They are stored in the attribute `class_int` associated to a given class.

Example 5.3.1 From a class of persons `person` with an attribute age, we define a class of teenagers thanks to the `class_int` attribute:

```
class :: new(teenager,
         [class_int(int(S,
                [isa(S,person),age(S,X), 12 < X < 20]))]).
```

it corresponds to the rule:

```
isa(S,teenager) <- isa(S,person) and age(S,X) and 12 < X < 20.
```

◇

Other model where such a feature is available, consider the problem of relating automatically the class defined in intension to other classes in the class hierarchy. We argue that it is not necessary. Indeed, we have shown that the purpose of the class hierarchy is to have a conceptual support for modeling that defines the isa link. Here, the isa link is already defined. The example 5.3.2 below shows the general difficulty to position the class in intention in the hierarchy.

Example 5.3.2 This example is illustrated by the class hierarchy of figure 5.1. A person is a member of the EEC if he is French, English, Dutch or citizen of any EEC country. European electors are the persons who have the right to vote in the country they are citizen of. They vote for the European elections in the country they are citizen of. The class in intension european_elector is defined by the following rules:

```
isa(X,european_elector) <- isa(X, french) and elector_in(X,france).
...
isa(X,european_elector) <- isa(X, portuguese) and elector_in(X,portugal).
```

Intuitively, european_elector is a subclass of european_citizen. It has no existing subclass.

Now we consider the class of person having the double nationality, French and German. This class french_german is defined by the rule:

```
isa(X,french_german) <- isa(X, french) and isa(X, german).
```

It is, intuitively, a subclass of french and german.

The class european_citizen could have been defined intentionally by the rules:

```
isa(X,european_citizen) <- isa(X, french).
...
isa(X,european_citizen) <- isa(X, portuguese).
```

It is, intuitively a superclass of all the nationalities classes.

In general, a class in intention, is intuitively a sub or a super class of a class depending on the classes referred in the rules. Since it has no consequences on inheritance, we do not compute this sc link ◇

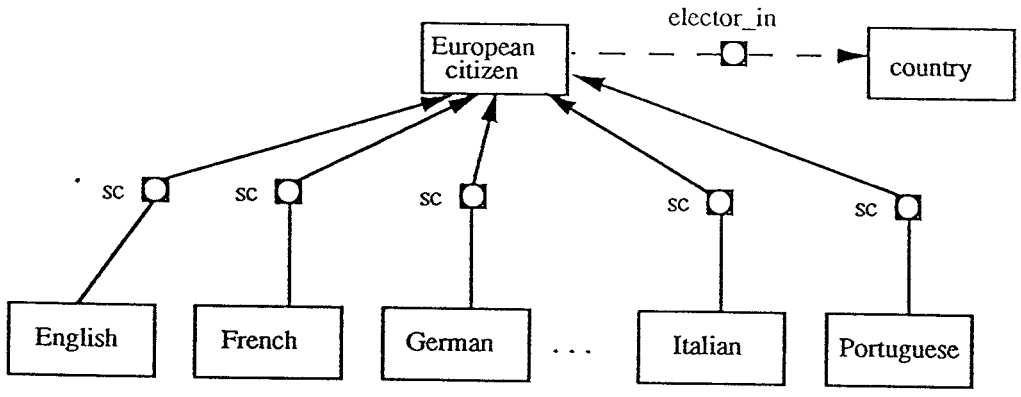

Figure 5.1: Classes in Intention

Chapitre 6

Implementation

Implantation

Les expériences d'implantation de prototypes pour le système Möbius nous ont amenés à envisager plusieurs catégories de problèmes.

Parmi ces problèmes, nous discutons, dans ce chapitre, de l'implantation d'évaluateurs pour le langage de requêtes de Möbius. Nous décrivons succintement une approche méta-interprétée qui est à la base d'un premier évaluateur implanté en Prolog (Megalog), et une approche compilée qui nous a permis d'utiliser le système cible EKS-V1 pour l'évaluation de requêtes et la gestion des contraintes d'intégrité.

L'implantation de systèmes de gestion de connaissances basés sur un modèle sémantique de données, et en particulier sur le modèle Möbius, au dessus d'un système de gestion de données relationnel, pose le problème de la mise en correspondance des deux modèles. Nous montrons comment, dans le modèle même de Möbius, sont intégrées les informations de mise en correspondance. Cette paramétrisation est un support pour l'étude ultèrieure des différentes formes de correspondances et de leurs propriétés : efficacité versus intégrité.

This chapter aims at giving the reader an idea of the problems and solutions we found when implementing prototypes for the Möbius system. In general, each aspect and feature of the Möbius system presented in this thesis has been evaluated in an implemented prototype. It has not been done in the same unique implementation.

We present here two versions of the query evaluator:

- The first one is implemented in Prolog (Megalog). It is based on a meta-interpreter and focuses on the reflective aspects of the model;
- The second is based on a translation (compilation) of the Model into Datalog rules evaluable by an existing deductive database system. The rules are evaluated by the EKS-V1 query evaluator. The experiences exploiting the EKS-V1 capabilities allowed us to test other aspect of the Möbius system such as integrity constraints and updates.

then, we discuss the interaction between the model and the query evaluator in term of possible semantic optimizations.

Finally, we discuss the problem of the Mapping of the Möbius model to a relational model.

This last issue is just a report of the problem we have faced while prototyping. There is no new solution offered by the Möbius environment.

6.1 Query Evaluator

6.1.1 Meta-interpretation in Prolog

In the Möbius model, there is no a priori distinction between data and programs, between schema, meta-schema and data: everything is an entity. However, this distinction appears in the evaluation since some data will be used by the evaluator for the computation. Namely, intentions and extensions of the attribute classes and classes will be used by the evaluator to compute the sets they denote.

The Möbius kernel is very easy to implement. It basically corresponds to a set of attributes stored in the database. The semantic network of kernel classes, attribute classes, intentions, extensions and other entities takes sense and can be extended on the basis of the query evaluator. The sets denoted by queries, intentions and extensions must be correctly computed. Therefore, the query evaluator must be sound and complete with respect to the relational semantic of Möbius.

The first implementation of a query evaluator is a simple extension of the I1 Prolog vanilla interpreter:

I1:

```
eval(L and Exp) :- eval(L), eval(Exp).
eval(L) :- ((retrieve(L) ) ;
            (rule(L <- B), eval(B) ) ).
```

Such a meta-interpreter is not sound and complete. In particular, it may not always terminate in the case of recursive rules. This interpreter is extended to take the Möbius data into account (interpreter I2):

```
I2:
eval(L and Exp) :- eval(L), eval(Exp).
eval(L) :- L =.. [N,V1,V2],
           eval(     name(Att_Id,Name) and
                     sd(Att_Id, SD) and
                     td(Att_Id, TD) ),

          ((eval( ext(Att_ID,ext(Arg1, Arg2, Ext)) ),
            retrieve(Ext) ) ;
          (eval(int(Att_Id,int(Arg1, Arg2, Int)) ),
            eval(Int) ) ),

           eval(isa(V1,SD) and isa(V2,TD)).
```

Each literal in a query is decomposed. From its name, the evaluator select several candidate attribute definitions (`Att_id`). The argument of the attribute definition will parameterize the evaluation.

The intentions and extensions (`Int`, `Ext`) will respectively generate subqueries or call a retrieval in the database.

The source and target domains (`SD`, `TD`) are used to control the compatibility of the arguments of the literal in the query. This is the way inheritance is implemented.

The attribute classes, being themselves entities, and their intentions, extensions, source and target domains, being attributes, are queried with the same evaluator.

As such, the evaluation procedure never ends. First of all, the `isa` attribute can not be evaluated with such a procedure: the evaluation of a query like `isa(X,entity)` would generate subqueries of the same form without alternative.

The first decision consists in treating the `isa` link differently. In Möbius, `isa` is not a normal attribute. Its intention is parametrized by the classes:

```
isa(X,Y) <- class_extension(X,Y).
isa(X,Y) <- class_intention(X,Y).
isa(X,Y) <- isa(X,Z) and sc(Z,Y).
```

Here, `class_extension(X,Y)` and `class_intention(X,Y)` stand for the class `Y` respective extension and extension. We do not check the source and target domains as their correctness is ensured by integrity constraints

Its evaluation uses a special rule:

```
eval([isa(Arg1,Arg2)|L]):-eval([ext(Y,ext(Arg1, Class_Ext)),
                                int(Y,int(Arg1, Class_Int))]),
            (retrieve(Class_Ext) ; eval([Class_Int]) ),
            eval(L).
```

The strong consequence is that we have to forbid the user to define attribute named `isa`. This is acceptable since the semantic of the whole system relies on the definition of `isa`. It is not a restriction as we allow classes to have extensional or intentional definitions of their instances. This definitions: `Class_Ext` and `Class_Int` are used in the evaluation of `isa`.

Still, the evaluation procedures is looping. Indeed, for the kernel attributes: `ext`, `int`. `name`, `sd`, `td`, the evaluator will infinitely try and find respectively their extension, intention, name etc. This infinite regression is stopped thanks to special purpose rules for each kernel attribute:

```
eval([int(o9,Y)|L]):- retrieve(stored_int(o9,Y)),eval(L).
eval([ext(o8,Y)|L]):- retrieve(stored_ext(o8,Y)),eval(L).
eval([name(o13,Y)|L]):- retrieve(stored_name(o13,Y)),eval(L).
eval([sd(o7,Y)|L]):- retrieve(stored_sd(o7,Y)),eval(L).
eval([td(o6,Y)|L]):- retrieve(stored_td(o6,Y)),eval(L).
```

Respectively, `o9`, `o8`, `o13`, `o7` and `o6` are the attribute classes for `ext`, `int`, `name`, `sd`, `td`. The consequence, again, is that these kernel attributes cannot be redefined or extended by the user.

We have also experimented a version of the Möbius evaluator based on a QSQ query evaluator described in [Lef91a] as a meta-interpreter. This version is sound and complete thanks to subqueries and answers memoization.

The meta-interpreted approach was sufficient to validate the metacircular approach. However it leads to an inefficient implementation. The other aspects of the Möbius Model (in particular integrity constraints) needing other development, we now present at the second prototype using the existing deductive database system EKS-V1.

6.1.2 Compilation in EKS-V1

The main drawback of the interpreted approach is the redundant computation. This leads to unacceptable response time for the query evaluation. The first kind of redundancy we have to eliminate exists when the definition of a class or an attribute is completed. In that case one needs to recompute the source, target domains, extension and intentions and redefinitions for each evaluation involving the attribute.

The idea is to materialized this information. The attribute definitions are translated into rules (closed to the rules defining their semantics). It is then necessary to update

these rules in case of a modification of the data they depend on. Since the rules can be defined by means of deductive rules themselves, the update propagation mechanism should be similar to the one used for materialized predicates ([VBK91a]).

The generated rules can now be compiled and used by the EKS-V1 system. In the following we examine on examples the form of the produced rules.

Let us consider a class of employees, `employee`, and two attribute classes, `sup_direct`, linking an employee to its direct leader, and `sup` the transitive closure of the leaders hierarchy. The attribute class `sup` has the class `employee` as a source and target domain. It can be stored extensionally in the base relation `r_oo7` and is intentionally defined by the two rules of the transitive closure:

```
sup(X, Y) <- sup_direct(X, Y).
sup(X, Y) <- sup_direct(X, Z) and sup(Z, Y).
```

The rules produced by the compilation are:

```
sup(employee, X, Y, employee) <- r_oo7(X, Y) and
                                 isa(X, employee) and
                                 isa(Y, employee)).

sup(employee, X, Y, employee) <- sup_direct(X, Y) and
                                  isa(X, employee) and
                                  isa(X, employee)).

sup(employee, X, Y, employee) <- sup_direct(X, Z) and
                                  sup(Z, Y) and
                                  isa(X, employee) and
                                  isa(Y, employee)).
```

However, if we want to take into account other aspects of the use of the attribute definition, we must create rules a little more complex. In particular if we want to benefit by features such as call by full-name, or call to super.

```
sup(X, Y) <- full_sup(_, X, Y, _).

super_sup(X, Y) <- full_super(_,X, Y).

full_super_sup(employee, X, Y) <- sc(employee, C) and
                                  full_sup(C, X, Y, _)).

full_sup(employee, X, Y, employee) <- r_oo7(X, Y) and
                                      isa(X, employee) and
                                      isa(Y, employee)).
```

```
full_sup(employee, X, Y, employee) <- sup_direct(X, Y) and
                                      isa(X, employee) and
                                      isa(X, employee)).

full_sup(employee, X, Y, employee) <- sup_direct(X, Z) and
                                      sup(Z, Y) and
                                      isa(X, employee) and
                                      isa(Y, employee)).
```

the `full_sup/4` rules corresponds to the call by full-name. The `super_sup` and `full_super_sup` correspond respectively to the classical `super` call of object-oriented languages in both normal and full-name case.

It is easy to see how to extend this compilation stage to handle other situations. For instance, if a symmetric attribute name is associated to each attribute class then the rules can easily be compiled as well:

```
sym_sup(X, Y) <- sup( Y, X).
sym_sup(X, Y) <- sup( Y, X).
```

The same process is applied to classes intentions and extensions. Classes intentions and extensions are compiled into rules for the `isa` attribute. Let us consider the class `employee` and a class `leader`. The instances of `leader` are extensionally stored in the relation `r_leader(X)` and are intentionally defined by the rule:

```
isa(X, leader) <- isa(X, employee) and sup(_, X).
```

Two rules are added to the rules of the `isa` attribute:

```
isa(X, leader) <- r_leader(X).
isa(X, leader) <- isa(X, employee) and sup(_, X).
```

Notice that, as we said when presenting this aspect of the model, the `leader` class is not a subclass of `employee` although it will behave as such.

This compiler generating incrementally the rules has been tested for a subset of Möbius. Mainly, we have excluded all the features compeling the use of complex terms. Therefore, the use of meta-classes is very limited since they mainly manipulate complex terms such as intentions or extensions. The target system is the EKS-V1 system. The attribute classes are compiled into rules in the EKS-V1 language. The EKS-V1 evaluator directly evaluates the Möbius queries. We also took advantage of the constraint checker, the EKS-V1 manipulation language and the Megalog environment for developing the Möbius environment.

6.1.3 Semantic Optimization

Such an automatic translation of the Möbius model into Datalog rules could be suspected to be inefficient. In particular, it does not make any optimization. We consciously avoided the optimization approach since it is very dangerous. Indeed, optimizations can only be made with regards to the evaluation procedure. Therefore, we consider that a translation must maintain a neat logical form. The information allowing optimizations can be injected in the query evaluator. The different actual query evaluation procedure can be radically different in terms of efficiency. An optimization for one can be a catastrophe for the other. Let us for instance consider the case of a transitive closure. The predicate `tct` is the transitive closure of the base relation `t`:

```
tct(X, Y) <- t(X, Y).
tct(X, Y) <- t(X, Z)  and tct(Z, Y).
```

Since we know the extension of the `t` base relation we could envisage to maintain an unfolded version of the above program. For instance if `t` contains the tuples `[(a,b), (b,c) , (c,d)]`, the unfolded program would be:

```
tct(a, b).
tct(b, c).
tct(c, d).
tct(a, Y) <- tct(b, Y).
tct(b, Y) <- tct(c, Y).
tct(c, Y) <- tct(d, Y).
```

In a tuple at a time evaluation, with possibly an indexing on the predicate arguments, this could be more efficient. However, with a set-oriented evaluation the modification, the number of logical inferences is increased and replaces accesses to the relational data. Since no indexing method is available, the efficiency depends on the respective cost of both procedure (inference vs retrieval). Moreover, the unfolding breaks the set at a time evaluation into a singleton at a time evaluation. We checked this example in EKSV1, and verified that the number of rules was a critical overload for the evaluator. Therefore, optimizations can not be made without considering the query evaluator. The direct translation has the advantage to keep the cleanest semantic to the rules. The specification can be used as an implementation. Now we can shortly examine how the semantic information induced by the data model can influence the evaluation and optimize it. Let us consider for instance the case of the redundant evaluation of the `isa` link induced by the automatic generation of rules from the Möbius model. Considering the three rules for two attributes classes from the class `c1` to the class `c2`:

```
a(X,Y) <- isa(X, c1) and r(X,Y) and isa(Y, c2).
b(X,Y) <- isa(X, c1) and a(X,Y) and isa(Y, c2).
b(X,Y) <- isa(X, c1) and a(X,Z) and b(Z, Y) and isa(Y, c2).
```

For `b`, if these are the only rules for the definition of `a` and `b`, it is clear that the `isa` literals will redundantly be evaluated since the appurtenance of `X` and `Y` to `c1` and `c2`

is guaranteed by the first rule. Again for a if we can ensure that the base relation r verifies the constraints on the two classes than we can drop the isa literals. In general, this can only be done if we have access to more information about the schema and the mapping. For that purpose, Möbius is an interesting tool. We do not have such optimization procedure included in an intelligent query optimizer but the needed information is accessible since it belongs to the Möbius model: attribute definitions, mapping information, are components of the model.

We need also to underline the fact that the cost of the redundancy above is limited by the use of memoization in the evaluation procedure. Indeed recursive query evaluators rely on memoization to ensure termination. But memoization is also very useful to avoid redundant computation.

6.2 Mapping

6.2.1 The Problem

The relational model is now the basis of several existing and commercial Database Management Systems. It has proven its reliability and a sufficient efficiency for a wide range of applications (such as financial applications). As stated all along this document, it appears that the standard relational technology is not a totally satisfying framework for several kind of reasonable applications. In particular, the modeling tools may not be sufficient.

Of course, by adding the power of logic and deduction to the relational model, deductive models and systems offer an already interesting framework to modeled complex applications. But we consider that the modelization tools are too raw. In other words, the definition of more sophisticated conceptual models is not necessarily based on the necessity of a higher expressive power. At least, what happens during a session between a user and a knowledge management system does not only happen in the computer. The model is also the support for a natural interaction between him and the machine. Otherwise, we would all program our computers in binary code, or using the Turing or Gödel languages. If providing supplementary constructs is not necessary the evidence that new knowledge is expressible, we still believe that it could be the case. At least, it shows how the users understands the application.

As far as the storage and retrieval of knowledge is concerned, we think that the relational model, because it corresponds to an effective technology, is a good starting point. The mapping of conceptual models into the relational model, or an extension of it, is a problem that anyone trying and implementing a system (or a prototype) based on a conceptual model has faced. Nevertheless, the problem has been little formally studied.

Here, several approaches are possible:

- One may wish to stick to the standard relational model in first normal form,
- or to try and extend it in some ways according to the new requirements due to

the complexity of the model (e.g. N1NF, NF2 models).

Again, in both the above cited options, one has the choice between a fixed canonical mapping (proving or not it is the best one with regards to efficiency and all the possible properties required) or a parameterized one (which again as to be proven correct).

It was not possible for us to study every aspects of the development of a knowledge management system. However, while trying and implementing prototypes, we had to make choices and to take decisions. As far as the mapping is concerned, on the basis that there exists a minimum valid solution (a one to one binary mapping), we have nevertheless decided to provide a parameterized form of mapping. We took the risk that an inconsistent mapping specification destroys the model integrity. The mapping parameterization is, at the moment, under the user control and responsibility. We gain the freedom to experiment several kind of mapping in the future. Indeed, Möbius is not a definitive proposal but a platform for experimentation and reflexion.

Our point of view is that the tools for an automatic or semi-automatic mapping specification are present in Möbius: views, meta-knowledge and integrity constraints, but we are still missing the formal framework for the study.

The problem can be stated as follow. On the one hand, we have a model based on the notion of entity with several variations on the notion of identity, and incorporating notions like attributes in intention and extension, classification and a language for expressing constraints and views. On the other hand, we have a relational model and the possibility to express integrity constraints. We now want to draw a correspondence between the two models and use the latter as a basis for the implementation of a system whose model is the former. We have several reasons not to adopt a fixed mapping:

- In terms of efficiency, it is not clear whether there exist a general ideal mapping;
- in terms of integrity, the mapping depends on too many semantic information to be fixed;
- A parameterizable mapping may allow to set a correspondence between a new schema and an existing relational database

Therefore, in Möbius, we do not propose an automated mapping but tools to study different kinds of mapping. Thanks to the meta-level information, the mapping does not necessarily need to be set for each attribute class or each class, it can be specified at the level of meta-classes.

6.2.2 Mapping in Möbius

The main concept of the model is the notion of attribute which associates two entities. Attributes are defined thanks to attributes classes where their own attributes are specified. The meta-data available include the name, the source domain, the target domain.

(see chapter 3). The intention (int) of an attribute class stores MDL expressions denoting sets of attributes instances of that class: deduction rules. In the same manner, classes sets of instances may be intentionally defined by means of an expression stored in the intention of the class: class_int. The base information, the mapping information, is recorded in an attribute of classes or attribute classes named respectively class_ext and ext. The contents of this attribute is a relational expression specifying where the instance or attribute is stored in the relational system.

Example 6.2.1 We want to store some information about persons in a relation rel_person with the following schema : rel_person(person_identifier, address, date_of_birth) At the conceptual level, we define an attribute date_of_birth associated to the class person. The implementation choice will be specified as follow:

```
?- eval([name(X,date_of_birth)]).  /* To get the attribute identifier */
X = oo36
        more? -- ;
yes
?- oo36 :: set(ext(ext(Self,Value,[ext(rel_person(Self,_,Value))]))).
Self = _g234
Value = _g235

yes
```

The attribute is retrieved thanks to a simple evaluation on the relational database.

```
?- eval([date_of_birth(yvonne,Y)]).
```

will process a retr_tup(rel_person(yvonne,_,Y)). ◇

Thanks to the uniformity we adopted for the mapping specification, the attribute extension is an attribute (ext). Then, a set of attribute classes sharing the same behaviour with respects to their extensional definition can be grouped together as instances of a common class. This class can define their ext attribute intentionally. For instance, a possible intention for the ext attribute could be a formula denoting a mapping specification which uses a binary relation. The name of the binary relation could be inferred from the attribute name. All the attribute classes, instances of this class (let us call it binary_attribute_class) would inherit of a mapping of binary relations.

Example 6.2.2 The intentional definition of the attribute ext (e.g. o26) can be set to a mapping into a binary relation. The name of the relation is the attribute name. We store the fact:

```
int(o26, int(Self, ext(Att_id,Value,[ext(Mapping)]),
          [name(Self, Att_name),
           sys(Mapping =.. [Att_name,Att_id,Value])]))
```

Then, the extension of the attribute `date_of_birth` (oo36) will be inherited and computed as: `ext(oo36, ext(Self,Value,[ext(date_of_birth(Self,Value))]))`. ◇

In the prototypical experimentations, we choose temporarily the solution of a binary mapping. It is clear that such a direct translation of the model to a relational structure has several drawbacks. Mainly it augments the number of joins at evaluation time and decreases the efficiency of the evaluation. However, any other form of mapping may cause problems at update time. For instance a class mapping, defining a relation per class, with a column per attribute definition associated to the class, may lead to complicated update procedures. On the other hand, of course, grouping as much information as possible in the same table may optimize the evaluation reducing the number of joins.

At the current stage of the project, we have a prototypical implementation which will allow us to do some practical experiments on the use of other forms of mapping and on the definition of semantically correct and efficient methods to process the updates performed at the conceptual level. Therefore, we still have to study how the updates procedures can be integrated in the system, how the constraints on the relational schema can have an influence on the conceptual model and, symmetrically, how we can ensure that the model semantics can not be modified or altered by the schema of the underlying storage structure.

Chapitre 7

Manipulation Language

Le langage de manipulation

Pendant la réalisation de prototypes pour le système Möbius, nous avons expérimenté diverses formes de langage de manipulation. Pour faciliter le développement des premiers prototypes nous avons d'abord décidé d'utiliser Prolog, étendu avec quelques primitives de mise à jour, comme langage de manipulation. Il semblait cependant regrétable de ne pas pouvoir exploiter l'organisation des connaissances dans la hiérarchie de classe comme support de conception et de programmation pour le langage procédural de manipulation.

Dans ce chapitre, nous donnons notre point de vue sur cette alternative en la mettant en perspective des différentes propositions pour l'intégration au langage Prolog de concepts orientés objet.

Dans le même ordre d'idée, nous analysons les différents paradigmes possibles pour un langage de mise à jour et de manipulation.

7.1 Prolog as a Manipulation Language

We have seen, up to this point, how useful is a data model for understanding and querying the structure of a database. It is a semantics support for the structure of knowledge. In particular, associating inheritance to a class hierarchy is a natural way of representing knowledge and reasoning about it. Until now, we have concentrated our efforts on this point. However, a data model needs a manipulation language, at least to update the schema and data. An intelligent database system needs both declarative and procedural facilities. For instance, complex algorithms are easier to encode in a procedural language where the control is explicit. The procedural language or facilities are used to print out results, to compute complex algorithms or to update the model and data. The declarative language or facilities are used to denote sets, answers of a query. We choose a declarative language based on first order logic.

Offering procedural and declarative aspects can be achieved in several ways. First order logic itself can be a framework for procedural computation. Deductive databases emphasize the difference between declarative and procedural logic [GMN84], [CGT89].

However one can think of a single language where procedural and declarative aspects are differentiated by the syntax. For instance, the procedural counterpart of the connectives `true`, `false`, `and`, and `not` could be called `succeed`, `fail`, `sequence (,)`, `alternative (;)` and `if succeeds then fail else succeed`. This is possible in a context where the procedural part has a `success/failure` mode. The ambiguity is that procedure literals (programs) are not syntactically different from the declarative literals (data). In particular the formulae denoting the data are assimilated to the procedures that evaluate them. Of course, this confusion is part of the success of languages like Prolog: it is sometimes possible to have both a declarative and a procedural understanding of Prolog programs (when Prolog computes the exact set of solutions denoted by the straightforward translation of the program in first order logic). In that context, declarativity would be a kind of "don't care about control" statement.

A less polemic and cleaner solution consists in separating the two languages, providing interfaces from the one to the other. The declarative language can call procedural computations, when algorithms are known to be better (or easier to describe) than the standard evaluation of their declarative equivalent or when they involve side effects. The procedural language calls the evaluation of declarative expressions and collects the answers. Here, we can consider the following alternative: choosing a language close in its syntax and semantics to the classical procedural languages (C, Pascal) or choosing a procedural logical language à la Prolog. In the first case, the interface will pose, at least, the problem of impedance mismatch: each information moving from one language to the other must be adapted, data structures must be translated. In the second case, this problem can be avoided by choosing a common subset of data structures: logical terms. Choosing logic for the two languages does not imply such a choice. For instance the declarative and procedural languages could communicate in exchanging sets, while sets are a data structure of the procedural language. A drawback of the logical solution is again the possible confusion between two languages, similar but fundamentally different in spirit, due the sometimes answer less question of choosing between the two when one

can get the same results through the two different computation.

In fine, we choose to start from Prolog as the manipulation of Möbius, being aware of the alternative and understanding the advantages and drawbacks of this choice. It is an experimental choice after which we let a question mark. Prolog needs to be augmented with a few primitive: an evaluation procedure and update procedures. The minimal update procedures perform insertion and deletion to and from the storage system. This is not satisfactory since it does not reflect the model.Therefore we propose to start with two update procedures allowing insertion and deletion of links for which an intention is available: `isa` link for classes in extension and other attributes in extension. We choose to impose that the arguments of this procedure are ground when they are called.

7.2 Augmenting Prolog with Methods?

The purpose of Möbius is to integrate deductive capabilities and object-oriented features into the same data modeling context. We clearly separate the manipulation language, a procedural language, from the declarative language. However data are organized under a class-subclass hierarchy, and it is natural to think of an organization of the manipulation procedures under this classification. Namely, should we provide a means to structure the manipulation programs according to the class hierarchy in adding, to the manipulation language, methods and inheritance mechanisms for methods?

Apart from object-oriented approaches, data modeling was little concerned with modeling manipulations or behaviours. It is nevertheless interesting, when designing a knowledge base, to think in terms of structured and active entities, like abstract data types entities or objects having a structure and a set of operations that can be performed on them. Together with encapsulation, inheritance of manipulation procedures along a class hierarchy induces an interesting programming methodology and suggests interesting features.

Relying on the relational semantics of Prolog as a manipulation language, we could think of applying directly to the procedural part of Möbius the ideas we used for the declarative part. However, the analysis we made and the solutions we proposed may not be valid in a procedural context. Inheritance and deduction, in a declarative language, are two ways of expressing knowledge and reasoning about it. Even if the one can be encoded in the other, the need of sophisticated and user adapted modeling tools justifies a model including both facilities. Two modeling paradigms integrated in one model are also a means to know more about the user intension and, therefore, opens the door to semantics optimization of the knowledge management.

In the context of a procedural language (based on logic in our case, but the analysis can be adapted to other cases) inheritance is a supplementary control structure. Let us consider a method as a set of clauses somehow associated to a class. When one of the identified conflicts arise (multiple inheritance, definition and representation), several solutions are available: in the presence of two programs P1 and P2 for the same method predicate P,(1) one can choose one, or, (2) concatenate or melt the two .

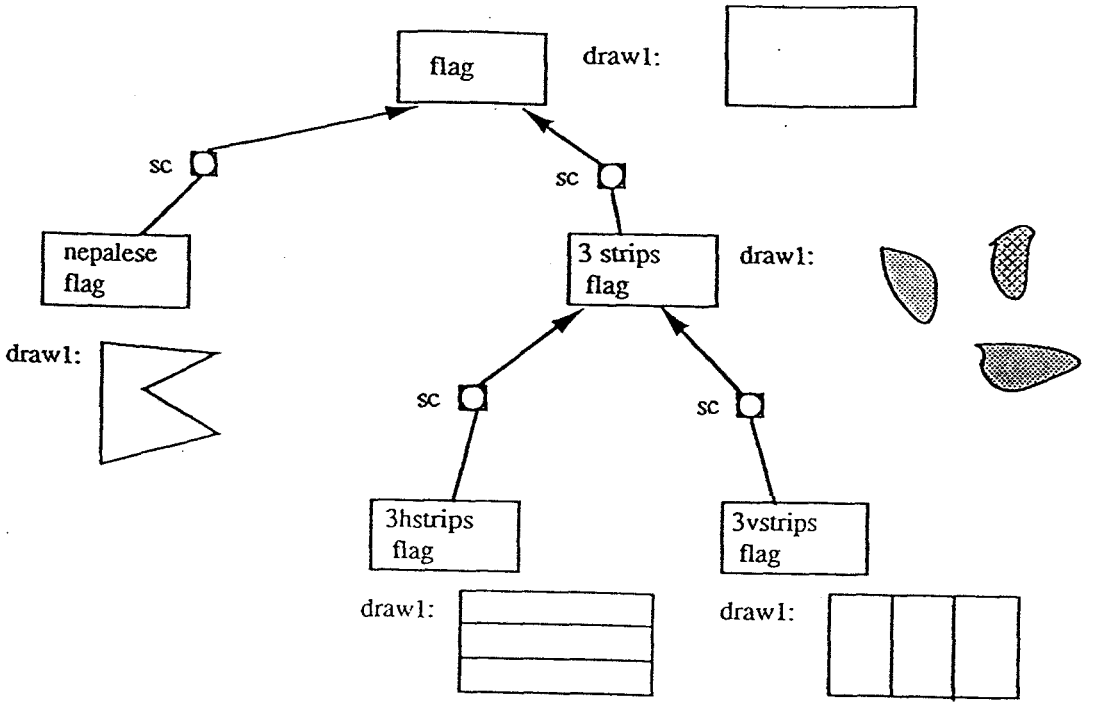

Figure 7.1: Flag Hierarchy and Methods

(1) Indeed, systematic overriding is the first solution to inheritance problems. According to the methodology associated to the object-oriented program conception, it is natural to consider, in the case of multiple definition, that the program associated to the lowest class in the hierarchy should overrides the code above (cf. example 7.2.1).

Example 7.2.1 We classify national flags. A generic class `flag` describing the structure and procedures available for every flag has a method `draw`. A flag can be drawn on the screen by calling the method procedure `draw/1` with the flag as its argument: `draw(a_flag)`. The method associated to the class `flag` draw a rectangle (cf figure 7.1). One subclass of `flag` is the class of Nepalese flags `nepalese-flag`. Unfortunately, Nepalese flags are not rectangular. The method `draw` associated to the class `nepalese-flag` overrides the method defined in its superclass. ◇

When considering multiple inheritance, as we have explained in chapter 4, the absence of a total order among the conflictual classes calls for another mechanism. Four solution are available:

- the choice of an implicit total order among the candidate classes or methods;
- the definition of an explicit total order;
- the interdiction of conflicts by explicit redefinition of the conflictual methods;
- the necessity of explicit desambiguization at run time (call by full-name, views).

If the model supports unlimited multiple representation, the solutions above can be also adopted, except for the third case where such a choice consists in avoiding two methods with the same name associated to classes unrelated by the sc order.

Of course, in order to be able to reuse the overridden code in the body of the new method, one must be able to benefit by tools like call by full-name or call to super.

(2) Let us now consider, as allowed by the structure of logical programs, that we inherit of all the conflictual code. The new problem, as opposed to the declarative case, is that the code is not a set of rules and facts but an ordered sequence of clauses. We must decide of a strategy to rearrange the two programs, P1 = (C11,...,C1n) and P2 = (C21,...,C2m), into a new sequence of clauses, the program for the method P. We can choose the sequences: (C11,...,C1n,C21,...,C2m) or (C21,...,C2m,C11,...C1n) or any sequence of Cij. In general, in any programming style, this consists in producing a new program by organizing the two inherited code in a new structured one. If we consider more sophisticated merging techniques we can simulate overriding or other intermediary solutions. Indeed, when merging the sequences of code, one can add new control items like cuts. Overriding can be simulated by producing the following code from the programs P1 and P2 for the procedure P:

```
P :-!, P1.
P :- P2.
```

Forgetting about the side effects and considering the success/failure semantics of Prolog one could think of a global program of the form:

```
P :- P1,!.
P :- P2.
```

The early proposal for the integration of object-oriented programming and logic programming paradigms [Gal86] [Zan84] and more recent paper [Dal89] have studied such ideas. In general there exists, in the procedural case (with side effects) a huge number of different and realistic solutions. Several situations are illustrated in the example below 7.2.2.

Example 7.2.2 We now add three classes to the hierarchy of example 7.2.1: the class **3strip-flag**, **3vstripflag** and **3hstripflag**, classes of the flags with three, three vertical and three horizontal strips. The class hierarchy is illustrated by the figure 7.1. The **draw** methods associated to **3vstripflag** and **3hstripflag** draw the three strips. Since the drawn items are white, the drawing method must first draw the rectangle and then draw the strips. The drawing method associated to the **3vstripflag** class computes and paints the color of the strips. It must be executed after the flag frame is drawn although it is defined before the strips drawing. ◇

When considering methods inheritance for a procedural programming language, even though it is still a useful support for conception, we end up with a new and possibly redundant control mechanism, whatever control strategy we adopt. If the strategy

is sophisticated and complex, it allows to handle all possible situations, but places a heavy burden on the programmer task. He must deal with two different of control: the programming language control, local to a method body, and the inheritance strategy. We have sympathy for the new programmer discovering, at the same time, cuts, backtracking and the influence of the order of rules and literals together with a complex inheritance principle. If the strategy is simple, for instance systematic overriding, explicit disambiguation of multiple inheritance and representation, we can wonder what the advantage of a supplementary control tool is. If, like in Möbius, the user can query the class and instance hierarchy from the manipulation language, he can program his own tests.Notice that in the case of a declarative language, inheritance is more a form of knowledge than means of control, therefore having other advantages.

In the Möbius prototypes, we have experimented both a manipulation based on methods associated to classes and a simple extension of Prolog with update primitive. The first language has the advantage of offering a natural clustering of the manipulation code together with the data (for storing the code in the database, for instance) and provides an interesting framework for the conception of applications. The second language is easier to use with respects to the understanding of control.

We know some advantages and drawbacks of the two solutions. We came to the conclusion that inheritance in a procedural context is a far more difficult problem than its counterpart in a declarative language. Several other aspects, like persistence, organization of code in modules, availability of procedures etc, have to be taken into account. Finally, in a knowledge base manipulation language, an important part of the manipulations consists in updates. Prolog with insertion and deletion primitives is also not necessarily the best solution.

7.3 Updates and Transactions

In this section we analyse the different solutions to the problem of the definition of an update language for a knowledge base management system. We try and give a definition of the notion of transaction and overview several perspectives, limitations and constraints for an update language. The actual solution we choose for Möbius is a compromise. This chapter is opened on future development.

Basic update operations are insertions and removal of elements in the storage structure: tuples of a base relation. The manipulation language needs more sophisticated constructs, in particular the classical notion of transaction. Basically, a transaction is a group of operations modifying the state of the knowledge base. A transaction can be defined as a composition of elementary transactions, computing intermediary states, and a commitment of the final new computed state. An elementary transaction is composed of three parts:

- a pre-condition: a query on the current state of the base;
- an update: a modification to the current state;

- a post-condition: a query on the new state.

In a Prolog syntax an elementary transaction procedure `e_trans(S1,P1,U,P2,S2)`, where `S1` and `S2` are the current and new states, `P1` and `P2` are the pre- and post-conditions and `U` is the update, can be defined as follow:

```
e_trans(S1,P1,U,P2,S2):- evaluate_in(P1,S1),
                         update(S1,U,S2),
                         evaluate_in(P2,S2).
```

Notice that neither the update nor the elementary transactions themselves modify the knowledge base. Only the commitment the transaction can do so.

In that context, integrity constraints can be seen as implicit post-conditions. As they have to be verified in any committed state of the knowledge base, one could think it is sufficient to check them before the commitment. After our experiences with EKS-V1, we argue that a programmer may want to use them (or a part of them) in any condition part. Therefore a mechanism is necessary to free the programmer from rewriting the integrity constraints he wants to check in extenso. Such a mechanism would allow the use of the update propagation and integrity checking techniques in the post-condition evaluation. An application of such a feature is the programming of recovery procedures when constraints are violated.

A procedural language is not the only way to specify transactions. Indeed, there exists the solution of a production rule language. A production rule language (cf. e.g.[KdMS90]) is a structured list of production rules of the form:

```
Condition -> Actions
```

A rule is selected whenever the condition part, `Condition`, is true. It can consist of a query on the current state of the base or the occurrence of an event. Several rules being candidates for being fired, an implicit or explicit control determines the rule(s) to be executed. Then the action part, `Actions` is performed. It consists of updates and possibly evaluation of queries on the new state. If we consider a production rule language where the condition part is a query on the base, an elementary transaction takes the form:

```
P1 -> update(U), evaluate(P2).
```

where `P1` and `P2` are the pre- and post-conditions and `U` is the update. Production rules are attractive since the control is distributed among the rules and partly encoded in the inference engine. It gives a flavour of declarativity and reactivity to the system. But the absence of explicit procedural control, although tempting, may make the programming task more difficult.

The update itself can be either procedural or declarative, i.e., it can be either a set or a program of basic updates or a specification of the new state to be reached. For instance

the new state can be specified by a query that has to be true. In such a case, the update on base relations must be automatically inferred. This situation is known as intensional updates or view update.

The table below summarize the functionalities of some of the existing proposals for update languages and languages involving state transitions in a relational context.

proposal	bd	production rules	procedure	backtracking	set tuple	declarative update
EKS-V1 [VBKL90]	yes	for ic	Prolog	yes	tuple	no
RDL1 [KdMS90]	yes	yes	no	no	set	no
[War84]	theory	no	Prolog	modal logic	tuple	no
[Man89]	yes	no	Prolog	modal logic	set	no
Logical objects [Con88]	no	no	Prolog	yes	tuple	yes
Linear objects [And90]	no	yes	no	no	tuple	yes

Chapitre 8

Conclusion

"Tout sera oublié et rien ne sera reparé ; le rôle de la réparation sera tenu par l'oubli." M. Kundera, *la plaisanterie.*

Dans cette thèse, nous avons présenté et discuté un modèle de représentation des connaissances : le modèle Möbius. Ce modèle se propose d'intégrer les aspects génériquement qualifiés d'orientés objet aux aspects déductifs, dans un contexte de base de données.

In this thesis we have presented and discussed a knowledge representation model: the Möbius model. It aims to integrate aspects labeled as object-oriented together with deductive ones, in a database context.

Il s'agit de fournir un ensemble d'outils permettant la description et la modélisation des connaissances. Pour atteindre cet objectif, le modèle comprend, d'une part, un support d'organisation des connaissances basé sur les notions de classe, d'instance et de hiérarchie et, d'autre part, un support de description de la connaissance en extension ou en intension basé sur les notions de définition d'attributs, de faits et de règles de déduction.

A set of tools for knowledge description and modeling is proposed. The model contains, on the one hand, a support for knowledge organization based on the notions of class, instance and hierarchy and, on the other, a support of knowledge intensional and extensional description based on the notions of attribute definition, facts and deduction rules.

Le langage de requêtes associé au modèle, son formalisme et surtout sa sémantique logique, permettent la description de vues complexes (en bénéficiant des diverses extension de Datalog - agrégation, négation -) et la définition de contraintes d'intégrité.

The query language associated with the model, its formalism and mainly its logical semantics, allow the description of complex views (taking advantage of the several extensions to Datalog (aggregation functions, negation) and the definition of integrity constraints

Nous avons particulièrement étudié les problèmes liés à la définitions, la sémantique déclarative et l'utilisation des mécanismes d'héritage. Dans cette perspective, nous avons presenté quelques outils dynamiques et statiques qui permettent une utilisation plus souple du modèle en presence de conflits d'héritage potentiels.

We studied in detail the problems related to the definition, the declarative semantics and the use of inheritance mechanisms. In this perspective, we have presented several tools, static and dynamic, allowing a more flexible use of the model when potential inheritance conflicts arise.

Enfin, nous avons présenté et discuté nos expériences de prototypage d'un système de gestion de connaissance basé sur notre modèle : l'évaluation de requêtes, le langage de manipulation et la mise en correspondance du modèle avec les structures de stockage.

Finally, we have presented and discussed our prototyping experiments of a knowledge management system based on our model: query evaluation, manipulation language and mapping of the model to relational storage structures.

Plusieurs points restent à étudier, sur la base du modèle que nous avons décrit ici.

Several points remain to be studied on the basis of the model we have presented here.

La première direction de travail consiste à améliorer l'interface linguistique. Il s'agit là d'offrir un contexte de modélisation plus riche. La démarche que nous avons suivie, en considérant que l'addition d'un support d'organisation des connaissances basé sur la classification - malgré la possibilité de coder une telle information dans un environement déductif pur - était un atout, doit s'appliquer à d'autres formes d'organisation et d'expression des connaissances. Nous pensons, en particulier, que, malgré sa grande généralité, Datalog est un langage trop brut pour l'expression naturelle des contraintes d'intégrité (par exemple, les dépendances fonctionnelles ou les contraintes entre classes).

The first research direction lies in improving the linguistic interface. It is about proposing a richer modeling context. The approach we took, considering that it was an advantage to add a new support for knowledge organization based on classification (despite the capability to code such an information in a pure deductive environment), can be applied to other form of knowledge organization and expression. We think, in particular, that, despite its generality, Datalog is too raw a language for natural expression of integrity constraints (for instance, functional dependencies or constraints among classes).

Bénéficiant de cette méta-information, comme nous bénéficions déjà de l'information sur le rôle particulier du lien `isa`*, il est possible d'envisager d'optimiser l'évaluation de requêtes (optimisation sémantique) ou d'enrichir le langage de réponses (réponses intentionnelles).*

Taking advantage of this meta-information, as we took advantage of the information about the particular role played by the `isa` link, it is possible to think of optimizing the query evaluation (semantic optimization) or enriching the answer language (intentional answers).

S'agissant d'un système de gestion des connaissances basé sur notre modèle, comme nous l'avons souligné à plusieurs reprises, plusieurs directions de reflexions restent à envisager. Il faudrait définir un paradigme et un langage pour la manipulation et la mise à jour de la base de connaissances (procédures, règles, méthodes, etc). Un aspect intéressant de ce problème est notament le contrôle et la gestion de l'évolution du schéma et des objets.

As far as a knowledge management system based on our model is concerned, as we underlined on several occasions, several research directions remain open. A paradigm and a language for knowledge manipulation and update has to be defined (procedures, rules, methods, etc). An interesting aspect of that problem is, in particular, the control and management of the schema and objects evolution.

Enfin, le problème de la mise en correspondance du modèle avec les structures de stockage reste à formaliser et à étudier.

Finally, the mapping problem remains to be formalized and studied.

Nous pensons qu'un support d'étude idéal de tous ces problèmes serait un système avec une architecture modulaire.

We believe that an ideal support for this study would be a system with a modular architecture.

Annexe A

Graphical Conventions

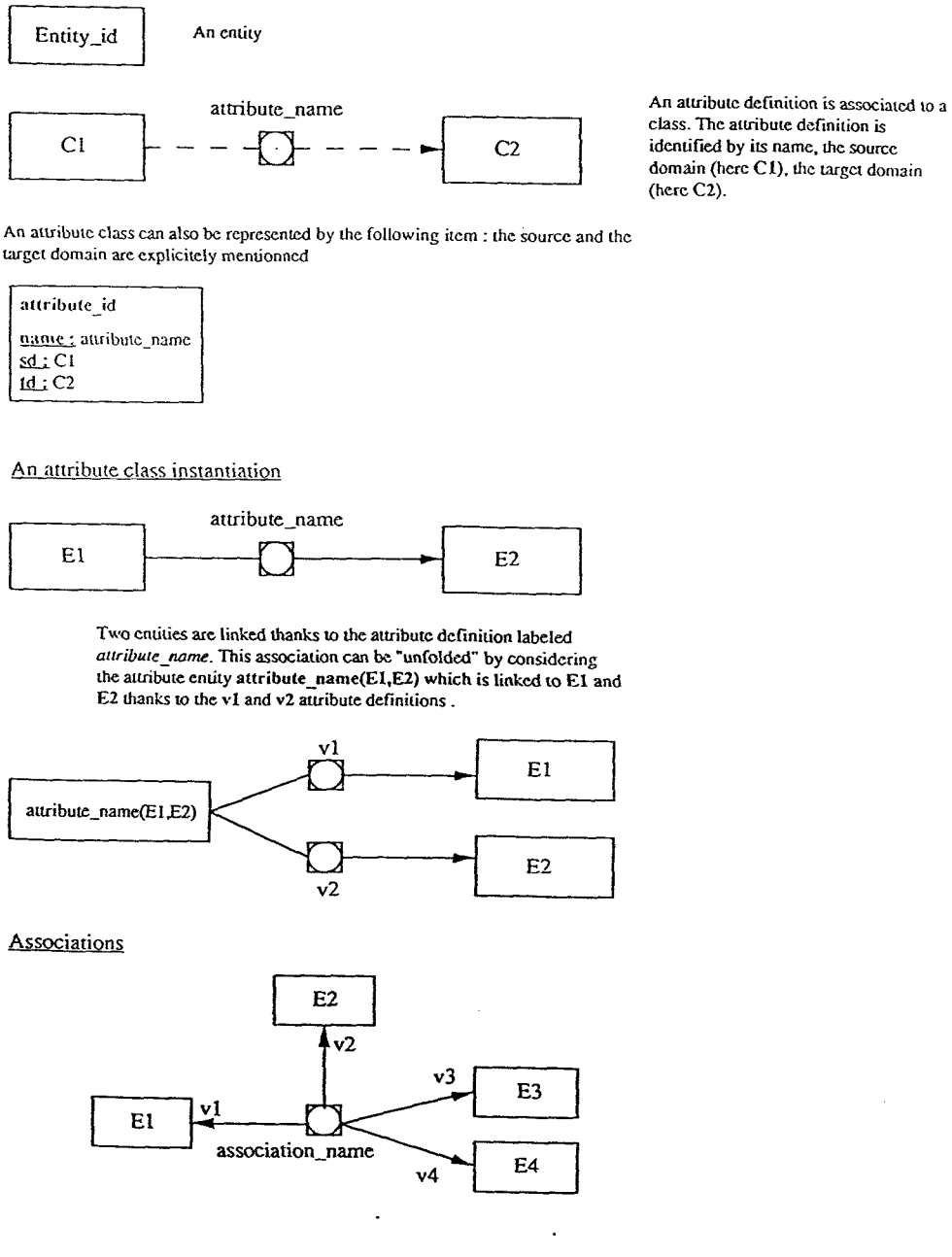

Figure A.1: Graphical Conventions

Références bibliographiques

[Aa90] M. Atkinson and al. The Object-Oriented Database System Manifesto. Technical Report TR 30-89, GIP ALTAIR, 1990.

[AB91] S. Abiteboul and A. Bonner. Objects and Views. In J. Clifford and R. King, editors, *Proc. of the ACM Conference on the Management of Data (SIGMOD)*, pages 238–247, Denver, May 1991.

[Abi90] S. Abiteboul. Towards a Deductive Object-oriented Database Language. *Data and knowledge engineering*, 5(4):263–287, 1990.

[Abr74] J.R. Abrial. *Data Semantics.* North Holland, 1974.

[AK89] S. Abiteboul and P. Kanellakis. Object Identity as a Query Language Primitive. In J. Clifford, B. Lindsay, and D. Maier, editors, *Proc. of the ACM Conference on the Management of Data (SIGMOD)*, pages 159–173, Portland, June 1989.

[AKN86] H. Ait-Kaci and R. Nasr. LOGIN: a Logic Programming Language with Built-in Inheritance. *J. Logic Programming*, (3):185–215, 1986.

[Alb85] P. Albert. Prolog et les Objets. Technical report, BULL France, 1985.

[And90] J.M. Andreoli. *Proposition pour une synthèse de la programmation logique et de la programmation par objets.* PhD thesis, 1990.

[AV88] S. Abiteboul and V. Vianu. Datalog Extensions for Database Queries and Updates. rapport de recherche 900, INRIA, septembre 1988.

[Ba88] F. Bancilhon and al. The Design and Implementation of O2, an Object-oriented Database System. In *Proc. of the ooDBS II Workshop*, Bad Munster, September 1988.

[BCP86] C. Benoit, Y. Caseau, and C. Pherivong. LORE : Un langage objet relationnel et ensembliste. In *Actes des 3èmes JLOO, Bigre+Globule n. 48*, pages 103–113, Paris, 1986.

[BDM88] F. Bry, H. Decker, and R. Manthey. A Uniform Approach to Constraint Satisfaction and Constraint Satisfiability in Deductive Databases. In *Proc. of the 1st Int. Conference on Extending Database Technology*, Venice, Italy, February 1988.

[BMSU86] F. Bancilhon, D. Maier, Y. Sagiv, and J. D. Ullman. Magic Sets and Other Strange Ways to Implement Logic Programs. In *Proc. of the 5th ACM Symposium on Principles of Database Systems (PODS)*, Cambridge, Massachussets, March 1986.

[Bob86] D.G. Bobrow. Commonloops. In *Proc. of OOPSLA*, pages 17–29, 1986.

[BP88] J. Bocca and P. Pearson. On Prolog-DBMS Connections: a Step Forward from Educe. In Gray and Lucas, editors, *Prolog and Databases: implementations and new directions*, pages 55–66, Chichester, 1988. Ellis Horwood. Chapter 4.

[Bra77] R.J. Brachman. What's in a Concept: Structural Foundations for Semantic Networks. *Int. J. Man Machine Studies*, (9):127–152, 1977.

[Bra83] R.J. Brachman. What IS-A is and isn't: Analysis of Taxonomic Links in Semantics Networks. *IEEE Computer*, 16(10):67–73, 1983.

[Bry90] F. Bry. Query Evaluation in Recursive Databases: Bottom-up and Top-down Reconciled. In W. Kim, J.-M. Nicolas and S. Nishio, editor, *Deductive and Object-Oriented Databases*. Elsevier Science Publishers B.V. (North-Holland), 1990. Proc. of the 1st International Conference on Deductive and Object-Oriented Databases (DOOD), 1989, Kyoto, Japan.

[BS85] R.J. Brachman and J.G. Schmolze. An Overview of the KL-ONE Knowledge Representation System. *Cognitive Sciences*, 9(2):171–216, 1985.

[Car89] B. Carre. *Méthodologie orientée objet pour la représentation des connaissances*. PhD thesis, Université Lille I, 1989.

[Cas86] Y. Caseau. LORE An Object Oriented Programming Environment. Technical report, CGE marcoussis, 1986.

[CCT90] F. Cacace, S. Ceri, and L. Tanca. Object-orientation and Logic Programming for Databases : a Season's Flirt or Long-term Marriage ? In *6èmes Journées Bases de Données avancées*, pages 295–317, Montpellier, September 1990.

[CGM87] U.S. Chakravarthy, J. Grant, and J. Minker. Foundations of Semantic Query Optimization for Deductive Databases. In *Foundations of Deductive Database and Logic Programming*, pages 243–273. Minker, J., 1987.

[CGM90] U.S. Chakravarthy, J. Grant, and J. Minker. Logic-Based Approach to Semantic Query Optimization. *ACM Transactions on Database Systems*, 15(2):162–206, 1990.

[CGT89] S. Ceri, G. Gottlob, and L. Tanca. What You Always Wanted to Know About Datalog (And Never Dared To Ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(1):146–166, 1989.

[CGT90] S. Ceri, G. Gottlob, and L. Tanca. *Logic Programming and Databases*. Springer-Verlag, 1990.

[Che76] P. Chen. The Entity-relationship Model, Toward a Unified View of Data. *ACM TDbS*, 1(1):9–36, 1976.

[CM81] W.F. Clocksin and C.S. Mellish. *Programming in Prolog.* Springer-Verlag, 1981.

[Coi87] P. Cointe. *The ObjVlisp Kernel : A Reflexive Lisp Architecture to Define a Uniform Object-Oriented System*, pages 155–176. North Holland, 1987.

[Con88] J.S. Conery. Logical Objets. In K.A. Bowen R.A. Kowalski, editor, *Proceedings of the Fith International Conference on Logic Programming*, pages 420–435, 1988.

[CW86] C. L. Chang and A. Walker. PROSQL: A Prolog Programming Interface with SQL/DS. In L. Kerschberg, editor, *Proc. of the First International Worshop on Expert Database Systems*, pages 233–246, Menlo Park, CA, April 1986.

[CW89] W. Chen and D.S. Warren. C-Logic of Complex Objects. In ACM-Press, editor, *Proc. of the ACM Conference on Principles of Database Systems*, pages 369–378, Philadelphia, 1989.

[Da77] R.O. Duda and al. Semantic Network Representations in Rule-based Inference Systems. Technical report, SRI international, 1977.

[Dal89] D Dalal, Mand Gangopadhyay. OOLP: A Translation Approach to Object-Oriented Programming. In *Prooceedings of the first international conference on Deductive and Object Oriented Database*, pages 555–568, Kyoto, 1989.

[Duc88] R. Ducournau. YAFOOL Version 3.22. Manuel de référence. Technical report, SEMA.METRA, Montrouge, France, 1988.

[Dug89] P. Dugerdil. Inheritance Mechanisms in the OBJLOG Language : Multiple Selective and Multiple Vertical with Points of View. In *Proc. of the Workshop on Inheritance Hierarchies in Knowledge Representation and Programming Languages*, pages 233–242, Viareggio, Italy, 1989.

[EJ90] J. Escamilla and P. Jean. Relations dans un modèle de représentation de connaissances. In *6èmes Journées Bases de Données avancées*, pages 195–212, Montpellier, September 1990.

[fADF90] Committee for Advanced DBMS Function. Third-Generation Database System Manifesto. In *Proc. of the IFIP TC2 conference on Object-Oriented Databases*, Windemere, July 1990.

[Fer83] J. Ferber. *MERING : un langage d'acteurs pour la représentation et la manipulation des connaissances.* PhD thesis, Université de Paris 6, 1983.

[Fer84] J. Ferber. Quelques aspects du caractère self réflexif du langage MERING. In *Actes des 2èmes JLOO, Bigre+Globule n. 41*, pages 277–290, Brest, Novembre 1984.

[Fin79] N. Findler. *Associative Networks.* Academic Press, 1979.

[Fre87] M. Freeston. The BANG File: a new Kind of Grid File. In Y. Dayal and U. Traiger, editors, *Proc. of the ACM Conference on the Management of Data (SIGMOD)*, pages 260–269, San Francisco, May 27-29 1987.

[Gal86] H. Gallaire. Merging Objects and Logic Programming: Relational semantics. In *Proc. of AAAI*, pages 754–758, 1986.

[GKP85] F. Giannessini, H. Kanoui, and R. Pasero. *Prolog.* Inter Editions, 1985.

[Glo89] P.Y. Gloess. Prolog et Objets et Objets et Prolog. In *AFCET.* AFCET, 1989.

[GMN84] H. Gallaire, J. Minker, and J.M. Nicolas. Logic and Databases: a Deductive Approach. *ACM Computing Surveys*, 16(2), June 1984.

[Gol85] A. Goldberg. *Smalltalk-80 : the Language and its Implementation.* Addison Wesley, 1985.

[HBD89] T. Horsfield, J. Bocca, and M. Dahmen. MegaLog User Guide. Technical report, ECRC, Munich, Germany, 1989.

[Hen78] G.G. Hendrix. Encoding Knowledge in Partitioned Networks. Technical report, SRI international, 1978.

[KC86] S. Khoshafian and G. Copeland. Object Identity. In *Proc. of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, Portland, September 1986.

[KdMS90] G. Kierman, C. de Maindreville, and E. Simon. Making Deductive Database a Practical Technology : a Step Forward. In *Proceedings of the international SIGMOD conference on the Management Of Data*, pages 237–246, 1990.

[KL89] W. Kim and F.H. Lochovsky. *Objetc-Oriented Concepts, Databases, and Applications.* acm Press, 1989.

[KLW90] M. Kifer, G. Lausen, and J. Wu. Logical Foundations of Objetc-Oriented and Frame-Based Languages. Technical Report TR 90/14, SUNY-Stony Brooks, 1990.

[KW89] M. Kifer and J. Wu. A Logic for Object-Oriented Logic Programming. In ACM-Press, editor, *Proc. of the ACM Conference on Principles of Database Systems*, pages 379–393, Philadelphia, 1989.

[Lef91a] A. Lefebvre. *Evaluation de Requêtes dans les Bases de Données Déductives : Aspects Théoriques et Pratiques.* PhD thesis, Université Paris V, June 1991.

[Lef91b] A. Lefebvre. Recursive Aggregates in the EKS-V1 System. TR-KB-34, ECRC, February 1991.

[LV90] A. Lefebvre and L. Vieille. On Deductive Query Evaluation in the DedGin* System. In W. Kim, J.-M. Nicolas and S. Nishio, editor, *Deductive and Object-Oriented Databases*, pages 123–144. Elsevier Science Publishers B.V. (North-Holland), 1990. Proc. of the 1st International Conference on Deductive and Object-Oriented Databases (DOOD), 1989, Kyoto, Japan.

[Mae87a] P. Maes. *Computational Reflection.* PhD thesis, Vreije universiteit Brussel, 1987.

[Mae87b] P. Maes. Concepts and Experiments in Computational Reflection. In *Proc. of the 2nd OOPSLA*, pages 147–155, Orlando, October 1987.

[Mai86] D. Maier. A Logic for Objects. In *Proc. of the workshop on Foundations of Deductive Databases and Logic Programming*, pages 6–26, Washington, 1986.

[Man89] S. Manchanda. Declarative Expression of Deductive Updates. In *Proceedings of the symposium on Principles Of Database systems*, pages 93–100, 1989.

[MB88] J. Mylopoulos and M. Brodie. *Readings in Artificial Intelligence and Databases.* Morgan Kaufmann, 1988.

[MBJK90] J. Mylopoulos, A. Borgida, M. Jarke, and M. Koubarakis. Telos : A Language for Representing Knowledge About Information Systems. Technical Report MIP-9011, University of Passau, 1990.

[Min87] M. Minsky. *A Framework for Representing Knowledge*, pages 211–281. McGraw-Hill, 1987.

[MKW89] R. Manthey, V. Küchenhoff, and M. Wallace. KBL: Design Proposal for a Conceptual Language of EKS. technical report TR-KB-29, ECRC, January 1989.

[MN88] P. Maes and D. Nardi. *Meta-Level Architectures and Reflection.* North Holland, 1988.

[MNS+87] K. Morris, J. F. Naughton, Y. Saraiya, J. D. Ullman, and A. Van Gelder. YAWN! (Yet Another Window on Nail!). *IEEE Data Engineering*, 10(4):28–43, December 1987.

[NCL+87] B. Nixon, L. Chung, D. Lauzon, A. Borgida, J. Mylopoulos, and M. Stanley. Implementation of a Compiler for a Semantic Data Model : Experiences with Taxis. In U. Dayal and I. Traiger, editors, *Proc. of the ACM Conference on the Management of Data (SIGMOD)*, pages 118–131, San Francisco, May 1987.

[Nix87] B. Nixon. Taxis '84: Selected Papers. Technical Report CSRG-160, University of Toronto, 1987.

[PDR91] G. Phipps, M. A. Derr, and K. A. Ross. Glue-Nail: A Deductive Database System. In *Proc. of the ACM SIGMOD Conference on Management of Data*, pages 308–317, Denver, Colorado, May 1991.

[PR89] A. Pirotte and D. Roelants. Constraints for Improving the Generation of Intensional Answers in a Deductive Database. In *Proc. of the 5th IEEE Conf. on Data Engineering*, pages 308–317, Los Angeles, 1989.

[Rec88] F. Rechenmann. SHIRKA, manuel de référence. Technical report, INRIA/ARTEMIS, 1988.

[Sek89] H. Seki. On the Power of Alexander Templates. In *Proc. of the 8th ACM Symposium on Principles of Database Systems (PODS)*, pages 150–159, Philadelphia, Pennsylvania, March 1989.

[Sow84] J.F. Sowa. *Conceptual Structures: Information Processing in Mind and Machine.* Addison Wesley, 1984.

[TS86] H. Tamaki and T. Sato. OLD Resolution with Tabulation. In *Proc. of the 3rd Int. Conference on Logic Programming*, pages 84–98, London, UK, June 1986.

[Tsu91] S. Tsur. Deductive Databases in Action. In ACM Press, editor, *Proceedings of the tenth ACM Symposium on Principles of Database Systems*, pages 142–153, Denver, May 1991.

[TZ86] S. Tsur and C. Zaniolo. LDL: A Logic-Based Data-Language. In *Proc. of the 12th VLDB Conference*, pages 33–41, Kyoto, Japan, August 1986.

[Ull89] J. D. Ullman. Bottom-Up Beats Top-Down for Datalog. In *Proc. of the 8th ACM Symposium on Principles of Database Systems (PODS)*, pages 140–149, Philadelphia, Pennsylvania, March 1989.

[VBK91a] L. Vieille, P. Bayer, and V. Küchenhoff. Integrity Checking and Materialized Views Handling by Update Propagation in the EKS-V1 System. TR-KB-35, ECRC, June 1991.

[VBK+91b] L. Vieille, P. Bayer, V. Küchenhoff, A. Lefebvre, and R. Manthey. An Overview of the EKS-V1 System. TR-KB-38, ECRC, 1991.

[VBK+91c] L. Vieille, P. Bayer, V. Küchenhoff, A. Lefebvre, and R. Manthey. Documentation for EKS-V1. TR-KB-36, ECRC, July 1991.

[VBKL90] L. Vieille, P. Bayer, V. Küchenhoff, and A. Lefebvre. EKS-V1, a User Guide. Technical Report KB-33, ECRC, July 1990. Restricted Distribution.

[Ven84] R. Venken. A Prolog Meta-interpreter for Partial Evaluation and its Application to Source to Source Transformation. In *Proc. of the European Conference on Artificial Intelligence (ECAI)*, Pisa, Italy, 1984.

[Vie86] L. Vieille. Recursive Axioms in Deductive Databases: the Query/SubQuery Approach. In L. Kerschberg, editor, *Proc. 1st Int. Conference on Expert Database Systems*, pages 179–193, Charleston, SC, USA, April 1986.

[Vie87] L. Vieille. Recursive Query Processing: Theoretical and Practical Aspects. In L. Kott, editor, *Proc. of the 2nd French-japan Symposium on Artificial Intelligence and Computer Science.*, INRIA, Sophia-Antipolis, France, November 1987.

[Wal85] M. Wallace. Reconciling Flexibility and Efficiency in a Knowledge Base Implementation. Technical report, ECRC, 1985.

[Wal86] M. Wallace. KB2: A Knowledge Base System Embedded in Prolog. Technical Report TR-KB-12, ECRC, Munich, Germany, August 1986.

[War84] D. Warren. Database Updates in Pure Prolog. In ICOT, editor, *Proceedings of the international conference on fifth generation computer systems*, pages 244–253, 1984.

[WKT88] M. Wallace, V. Küchenhoff, and A. Tomasic. KB2 Performance Review. Technical Report TR-KB-24, ECRC, Munich, Germany, January 1988.

[Zan84] C. Zaniolo. Object-Oriented Programming in Prolog. In *Proc. of IEEE int. Symp. on Logic Programming*, pages 265–270, 1984.

[Zan89] C. Zaniolo. Object Identity and Inheritance in Deductive Databases-an Evolutionary Approach . In W. Kim, J.M. Nicolas, and Nishio S., editors, *Proc. of the 1st DOOD*, pages 2–16, Kyoto, December 1989.

Résumé

Nous présentons une solution pragmatique, fédérant plusieurs approches, au problème de la représentation et de la modélisation des connaissances pour les systèmes de traitement automatique de l'information et d'aide à la décision.

Nous avons d'abord étudié les travaux antérieurs menés dans les domaines de l'intelligence artificielle, de la modélisation de données et des bases données déductives. Nous avons dégagé les concepts fondamentaux qui ont servi de base à notre modèle.

Ce modèle, que nous avons appelé Möbius, repose sur des aspects déductifs (vues et contraintes d'intégrité) et sur une organisation des connaissances selon les axes de classification, d'agrégation et de généralisation.

Nous étudions les problèmes spécifiques introduits par nos choix. Ces problèmes sont principalement liés à la nature réflexive de la définition et à l'exploitation de la taxonomie des connaissances (héritage).

Nous proposons un ensemble de solutions permettant d'organiser et de gérer le modèle : définition de niveaux, gestion statique des conflits d'héritage par masquage ou redéfinition, gestion dynamique des conflits d'héritage par la définition de points de vue.

Cette expérience a donné lieu à des implantations de prototypes. Nous présentons nos choix, nos idées et les perspectives de recherche liées à la définition de systèmes de gestion de connaissances.