# Representation de connaissances dynamiques dans SHERPA

J. Escamilla, V. Favier, P. Jean, G.T. Nguyen, D. Rieu

INRIA

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

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

Rapports de Recherche

N° 1208

*Programme 4*
*Bases de Données*

# REPRESENTATION DE CONNAISSANCES DYNAMIQUES DANS SHERPA

**José ESCAMILLA**
**Valérie FAVIER**
**Patrice JEAN**
**Gia Toan NGUYEN**
**Dominique RIEU**

GROUPE DE RECHERCHE
**GRENOBLE**

**Avril 1990**

*RR_1208*

# REPRESENTATION DE CONNAISSANCES DYNAMIQUES DANS SHERPA

José ESCAMILLA[1], Valérie FAVIER[1], Patrice JEAN[1],
NGUYEN Gia Toan[2], Dominique RIEU[1]

[1] IMAG & [2] INRIA
Laboratoire de Génie Informatique
BP 53 X
38041 GRENOBLE Cedex
Tel : 76 51 45 75
e-mail : nguyen@imag.fr

**Résumé**

On décrit un modèle de représentation des connaissances développé dans le cadre du projet SHERPA. L'objectif du projet est de définir, mettre en oeuvre et expérimenter un système de représentation de connaissances dynamiques basé sur un modèle à objets. SHERPA est un projet commun IMAG et INRIA. Les racines du modèle se trouvent en intelligence artificielle, dans le domaine des bases de données orientées-objet et en CAO. Le modèle présenté ici intègre en effet des concepts issus des langages de frames, des langages orientés-objets et répond à certains besoins d'applications qui doivent gérer des objets évolutifs. On présente les concepts de base du modèle, puis la sémantique et la mise en ouvre de l'héritage. On décrit ensuite la notion de relation sémantique entre objets. On évoque enfin la gestion d'objets évolutifs.
**Mots-clés** : Bases de connaissances, objets dynamiques, modèles.

# REPRESENTATION OF DYNAMIC KNOWLEDGE IN SHERPA

**Abstract**

A knowledge representation model developed for the Sherpa project is described. The aim of the project is to define, implement and experiment a system supporting dynamic knowledge, based on the object paradigm. Sherpa is a joint project IMAG and INRIA. The model draws on artificial intelligence techniques, on object-oriented databases and engineering design applications. It includes concepts found in frame-based knowledge representations, object-oriented programming and applications supporting evolving objects.
The basic concepts in the data model are presented first. The semantics and implementation of inheritance is then detailed. Semantic relationships and dynamically evolving objects are also described.
**Key-words** : knowledge bases, dynamic objects, data models.

## 1. INTRODUCTION

L'engouement actuel pour les modèles de données et les langages de programmation à objets (LOO) touche des domaines très variés allant du génie logiciel aux bases de données. Il est intéressant de constater que l'intelligence artificielle utilise depuis de nombreuses années pour la représentation des connaissances certains concepts comme les frames [MIN75], qui se démarquent et complètent les modèles à base d'objets [GOL84, MAS89].
Partant de ce constat, une première motivation du projet SHERPA est de fusionner des recherches en matière de systèmes de gestion de bases de données (SGBD) et de représentation des connaissances (RC). Une tendance actuelle est en effet de développer des SGBD basés sur des modèles de données issus de la programmation orientée-objet, comme ORION, O2 ou GemStone en leur adjoignant les fonctions de persistence, partage, transaction, sécurité et reprise après pannes [BAN87, BAN89, COP84]. La raison essentielle qui justifie cette approche est qu'elle permet de résoudre le décalage entre langages de programmation et SGBD, le fameux "impedance mismatch" sur lequel ont toujours buté les études antérieures. Une voie de recherche qui a été comparativement peu explorée consiste à partir de concepts utilisés depuis fort longtemps en RC [NEB85]. Comme on le montre dans ce qui suit, cette approche permet de combler plusieurs insuffisances des modèles d'objets "à la Smalltalk", par exemple pour la définition de relations sémantiques qui y sont difficiles à modéliser [CAR89]. Elle permet également de s'affranchir du principe d'encapsulation cher aux LOO qui reste aujourd'hui le principal obstacle à la mise en oeuvre des SGBD orientés objet.
Une deuxième motivation du projet SHERPA est de fournir un support à des applications dont les exigences opérationnelles ne sont pas entièrement satisfaites par les SGBD ou les systèmes de RC actuels, par exemple pour :
- la manipulation systématique d'objets incomplets et incohérents,
- l'évolution dynamique des structures de données,
- le support de relations sémantiques.

Parmi ces applications, la CAO occupe depuis longtemps une place de choix [RIE86]. Mais d'autres domaines sont également concernés. Les travaux actuels sur le génôme humain montrent par exemple que les besoins en matière de SGBD et de modèles de données extensibles sont encore loin d'être satisfaits [GAU89]. Une des

ambitions du projet SHERPA est justement de répondre aux exigences des biogénéticiens pour la modélisation et le décodage des bases de séquences en biologie moléculaire.
Plus généralement, pour répondre à la rigidité des méthodes classiques de conception d'application, il paraît intéressant d'éviter la dichotomie habituelle existant entre la phase de conception d'un système d'information et son implantation ultérieure à l'aide d'un SGBD ou d'un système de RC : il y a là ce que l'on pourrait appeler un "design mismatch" ou décalage dans la conception. Les efforts actuels sur les méthodes de conception orientées-objet ("object-oriented design") vont dans ce sens [BOO86, MEY89].
Une solution partielle consiste à intégrer dans un modèle de données ou de RC des mécanismes d'extensibilité qui permettent une évolution progressive des applications et une conception incrémentalle. L'état actuel des recherches concernant le décodage du génôme humain nécessite par exemple ce type de modèle : à tout instant la connaissance des structures et de la sémantique des informations n'est que partielle. Elle évolue constamment en fonction des connaissances acquises quotidiennement. Cette approche a également pour avantage d'éviter une conception exhaustive a priori.
Cette extensibilité est le support de la dynamique dans le projet SHERPA. L'hypothèse de base est que les modèles de RC et les modèles issus des LOO sont suffisamment complémentaires pour qu'une fusion judicieuse permette de répondre aux applications évoquées.
On présente tout d'abord les concepts de base du modèle (§ 2). On insiste sur l'héritage et les mécanismes qui le mettent en oeuvre (§ 3). On décrit ensuite la notion de relation sémantique entre objets (§ 4). On évoque enfin la gestion d'objets évolutifs (§ 5). On donne en conclusion quelques perspectives de développement de SHERPA à court et moyen terme (§ 6).

## 2. LE MODELE OBJET DE SHERPA

### 2.1 Position

Le modèle de représentation des connaissances de SHERPA s'inspire des modèles orientés-objets issus des LOO comme CLOS, Eiffel et ObjVlisp [KEE88, MEY89, COI87] : il est réflexif et présente un niveau "méta" qui permet de décrire et d'étendre les concepts de base qui

sont les métaclasses, les classes, la spécialisation de classes, les attributs, les méthodes, et l'héritage d'attributs et de méthodes entre classes. Il s'inspire également des modèles et langages de RC tels que LOOPS, ROME, OBJLOG et YAFOOL [BOB83, CAR89, DUG88, DUC88] en ce qui concerne la notion de dépendances entre objets, d'objets composites et de propagation de valeurs d'attributs entre objets. Il s'inspire enfin des langages à base de frames comme FRL ou SHIRKA pour ce qui est des descripteurs d'attributs et de l'attachement de méthodes et de réflexes aux attributs [ROB77, REC88]. Certains aspects particuliers comme la notion de relations sémantiques entre objets et celle d'attributs considérés comme des relations entre classes ont leurs racines dans des systèmes comme SRL et MERING [WRI84, FER84]. La gestion des appels de méthodes présente quant à elle des similarités avec celle mise en oeuvre dans CLOS, à savoir une appel sélectif des méthodes qui sont représentées dans un graphe particulier, indépendant du graphe d'héritage. La gestion des méthodes est donc en partie découplée de la gestion du graphe d'héritage entre classes. Le système est conçu en trois couches qui sont :

- la couche de représentation des connaissances, sujet de cette présentation et décrite dans les paragraphes 2 à 5 qui suivent,
- la couche de manipulation, où les méthodes sont gérées selon leur graphe spécifique. Cette couche propose également un langage de manipulation ad-hoc, inspiré des bases de données orientées-objets,
- la couche de stockage.

L'originalité de SHERPA réside dans des fonctionnalités qui sont rarement réunies simultanément et s'avèrent fondamentales pour le traitement de la dynamique, en particulier :

- la méta-circularité. Tout est objet. Tout objet est instance d'un autre objet. L'application de ces principes fait appel à un modèle réflexif et extensible, ce qui paraît indispensable dans un contexte d'objets évolutifs (§ 2.2).
- la gestion d'objets évolutifs, c-à-d incomplets, incohérents et de structure évolutive, à l'aide d'une notion originale de boucle de dégradation et de versions gérées automatiquement (§ 5).
- la possibilité de définir des relations sémantiques entre objets, par exemple des liens de dépendance ou de diffusion de valeurs (§ 4). La valeur d'un attribut représente soit une simple référence vers un objet, soit une relation entre des objets. Le concept d'attribut vu comme une relation permet de modéliser les liens sémantiques qui sont explicitement énoncés dans la définition de l'attribut.

- la multi-instanciation explicite. Elle consiste à rattacher une instance à plusieurs classes (par exemple "dupond" est instance des classes Personne et Mammifère). Cette fonctionnalité permet de limiter la génération de classes "creuses" c.a.d contenant très peu d'instances. De plus, la multi-instanciation explicite est nécessaire pour le traitement d'objets évolutifs et donc parfois incomplets et incohérents. Elle autorise, par exemple, une instance à appartenir à une classe dont elle viole momentanément les contraintes.
- l'héritage multiple : une classe peut hériter de plusieurs classes. Le mécanisme d'héritage permet non seulement l'héritage des attributs mais aussi celui des contraintes, des inférences et des clés (§ 3).

## 2.2 Métacircularité

L'extensibilité est une des exigences fondamentales du modèle SHERPA. Elle permet de gérer la dynamique du modèle définie ici par la possibilité d'ajouter, détruire ou modifier à tout moment les définitions des classes, c-à-d leur nom, les descripteurs de leurs attributs et les relations qu'elles entretiennent entre elles. Afin d'uniformiser le traitement et la gestion de cette dynamique, le modèle SHERPA suit donc l'approche "tout objet" comme CLOS ou ObjVlisp. On sait que l'efficacité d'un tel système peut être discutée [MAS89], mais l'objectif du projet est avant tout de développer un outil de recherche et non un produit industriel.
Le modèle est donc réflexif : il possède la propriété de se décrire lui-même. Tous les concepts du modèle sont définis à l'aide d'eux-mêmes. Il existe donc un niveau "méta" qui permet de définir et générer les concepts de méta-classes, de classes, sous-classes et d'instances. La métaclasse META est racine du graphe d'instanciation. La classe OBJET est racine du graphe d'héritage. Le modèle admet l'héritage multiple. La sémantique de l'héritage est celle d'une inclusion ensembliste stricte (§ 3). Comme ObjVlisp, la métacircularité, permet d'étendre les concepts de base disponibles par adjonction de classes et de méta-classes ad-hoc. Une description complète du niveau méta sort du cadre de cette présentation. Il possède beaucoup de similarités avec [COI87, KEE88].

## 2.3 Concepts de base

On ne donne ici qu'une approche intuitive et partielle des concepts de base. On insiste sur les notions générales et les fonctionnalités de base afin de fournir une vue globale du modèle. On évite donc autant que possible une formalisation et une technicité trop grandes.

### 2.3.0 Types, classes, méthodes

Il n'y a pas de similarité directe entre un type et une classe. Par définition, un type est caractéristique d'un attribut. Un type définit un domaine de valeurs (ou d'instances) possibles pour un attribut. Schématiquement, c'est un domaine défini par une classe auquel sont applicables des restrictions pour un attribut donné. Cette distinction est rendue nécessaire par l'accent mis ici sur la notion d'attribut, et par l'intérêt de disposer de restrictions qui soient sémantiquement gérables. On entend par restrictions gérables des contraintes de type intervalle, d'exclusion ("sauf") ou d'énumération par exemple. On exclut ainsi de la notion de type toutes les contraintes procédurales et toutes les méthodes d'inférence parce que leur résultat ne peut pas être connu du système. Une classe est un ensemble d'instances dotées d'une structure et d'un comportement donnés.
Les types ne sont pas des concepts explicites et manipulables du modèle de représentation des connaissances comme dans certains SGBD à objets [BAN89]. Cette dernière approche est en effet nécessaire dans les SGBD orientés objet à cause du principe d'encapsulation des LOO. Celui-ci est en effet en contradiction avec la notion habituelle de n-uplet ou d'enregistrement des SGBD où les attributs sont manipulés individuellement. Il est donc nécessaire d'assurer une interface minimum entre les objets vus du LOO et leur représentation persistente. Ce problème n'existe pas dans les modèles de représentation de connaissances car la structure des objets y est toujours visible. Elle permet de plus d'attacher des méthodes et réflexes aux attributs, et donc d'éviter l'invocation de méthodes créées artificiellement pour y accéder.
Le second avantage d'une approche RC est que le contrôle des méthodes peut être détaché de la structure des classes en graphe d'héritage. Cette possibilité est déjà exploitée dans certains systèmes à objets comme CLOS. Appliquée systématiquement, elle permet d'élaborer des mécanismes de contrôle très puissants sur les méthodes : structures de graphes propres aux méthodes, existence d'algorithmes multiples et spécialisation des méthodes aux valeurs

des arguments, sélection des méthodes les plus appropriées, etc. En détachant leur contrôle de la structuration des classes, elle permet donc d'implanter des mécanismes adaptés à chaque type d'application ou d'étendre le système avec des fonctions d'explication, ce qui est par nature très difficile à faire dans un LOO. Seule une approche faisant appel à ce type de caracatéristiques peut à notre avis revendiquer l'appellation de modèle de représentation de *connaissances*.

### 2.3.1 Attributs

Les classes sont définies à l'aide d'attributs. Comme tout objet, un attribut est une instance d'une classe particulière. Un ou plusieurs descripteurs sont associés à chaque attribut. Ces descripteurs permettent de définir la classe de l'attribut, la manière de calculer sa ou ses valeurs et des contraintes sur ces valeurs. Les attributs d'un objet représentent une partie son état. La définition d'un attribut dans une classe A est interprétée comme une caractéristique des instances de la classe et de ses sous-classes.
La classe où un attribut a été défini pour la première fois est appelée la classe d'origine de l'attribut. Pour faire la différence entre deux attributs de même nom et de classes d'origine différentes plusieurs approches sont possibles [MAS89]. Dans ORION, ce conflit d'héritage causé par l'héritage de deux attributs qui ont le même nom mais une classe d'origine différente peut être résolu de deux façons: soit l'utilisateur indique explicitement l'attribut dont il veut hériter, soit l'utilisateur change le nom d'un des attributs de telle sorte que le conflit disparaisse.
Le modèle SHERPA part de l'hypothèse que deux attributs de même nom qui proviennent de classes d'origine différentes n'ont pas le même sens. Par conséquent si un concepteur d'une base de connaissances définit deux attributs dans deux classes différentes, c'est parce qu'il veut exprimer que ces attributs sont des attributs différents. Ce qui entraîne que ces attributs doivent être toujours hérités, parce que chacun d'eux exprime une caractéristique différente des objets. Nous pensons qu'une bonne approche est d'utiliser le nom de la classe d'origine pour faire cette différence puisque la classe d'origine détermine le point de vue pour lequel l'attribut a été défini. Pour résoudre ce type de conflits on définit le concept de nom complet de l'attribut.

### 2.3.2 Nom complet d'un attribut

Le nom complet d'un attribut est le nom formé du nom de la classe où il est défini pour la première fois plus le nom donné à l'attribut dans cette classe. En utilisant le nom complet de l'attribut les conflits causés par l'héritage des attributs de même nom mais provenant de classe d'origine différentes peuvent être résolus. Ceci présente des analogies avec la notion d'étiquette utilisée dans OBJLOG pour la définition de points de vue [DUG88].

### 2.3.3 Constructeurs

La valeur d'un attribut peut être mono-valuée (entier, chaîne) ou multi-valuée : tas, liste, ensemble, séquence. Le constructeur n-uplet n'a pas été retenu dans un premier temps car il correspond par défaut à la structure de base d'un objet telle qu'elle est définie par sa classe. Le constructeur tableau peut quant à lui être aisément implanté par le type séquence.

### 2.3.4 Classes

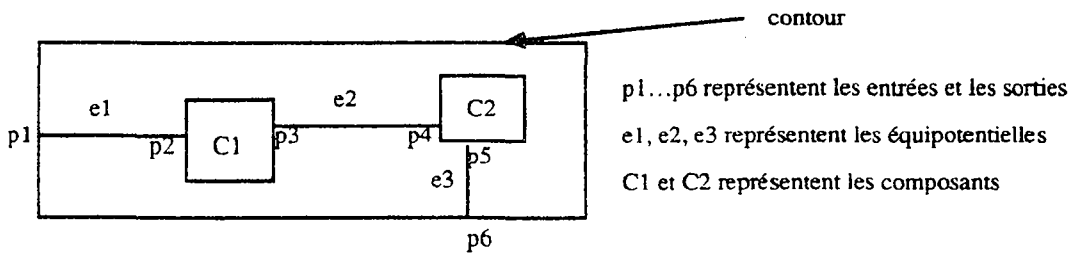
Les classes sont organisées en un graphe de spécialisation dont les arcs correspondent au lien de super-classe/sous-classe. Ce lien modélise la spécialisation stricte des ensembles : une sous-classe représente un sous-ensemble des instances de sa super-classe. Cette relation d'inclusion est mise en oeuvre par un mécanisme d'héritage décrit au paragraphe suivant. Ce mécanisme permet à toute sous-classe d'une classe C :
- d'hériter des attributs de C,
- d'affiner les attributs hérités,
- d'ajouter de nouveaux attributs.

La spécialisation est multiple c'est-à-dire qu'une classe peut hériter de plusieurs super-classes. De plus toute classe admet au moins une super-classe : la classe Objet qui est racine du graphe de spécialisation. On peut par exemple modéliser un circuit électrique de la façon suivante. L'exemple est volontairement trivial, pour n'introduire que les éléments de base.
La classe Circuit-électrique (exemple pris dans [RIE86]), d'identifiant Circuit-électrique, possède quatre attributs :
- contour : qui détermine le contour du circuit,
- entsor : liste des entrées et des sorties du circuit,
- composant : liste des composants du circuit,

- equipo : représente les équipotentielles.

```
Circuit-électrique
    nomclasse = circuit-électrique
  super = {Objet}                          /* super-classe   */
   attribut   =
     contour      :    un              rectangle
                       inst-de         attribut-obligatoire
     entsor       :    liste-de        entrée-sortie
     composant    :    liste-de        Composant
                       contrainte
                                       (not-intersect (composant))
     equipo       :    liste-de        équipotentielle
                       inférence
                                       (dessine (entsor composant))
```

Les mots en italique sont des descripteurs définis ci-dessous (§ 2.3.5). Une instance de la classe peut être le circuit suivant :




### 2.3.5 Descripteurs

Les descripteurs "*un*" et "*liste-de*" permettent de déterminer le type et la structure de la valeur de l'attribut. Par exemple, le contour est un rectangle.
Le descripteur "*inst-de*" (instance de) permet de donner la ou les classes d'appartenance d'un attribut. Si le descripteur "*inst-de*" n'est pas spécifié, alors l'attribut est par défaut instance de la classe Méta-Attribut. Contour est un attribut appartenant à la classe attribut-obligatoire.
Le descripteur "*contrainte*" permet de définir des contraintes sur un attribut (il existe également des contraintes portant sur plusieurs attributs) : (not-intersect (composant)) est une contrainte portant sur l'attribut composant. Elle spécifie qu'aucune intersection entre les composants n'est acceptée.
Le descripteur "*inférence*" associe à un attribut une ou plusieurs inférences qui permettent de calculer sa valeur. Ces inférences

utilisent des méthodes. Ces méthodes appellent des programmes définis par l'utilisateur : (dessine (entsor composant)) est une inférence qui permet de dessiner les équipotentielles (valeur de l'attribut equipo) et qui utilise la méthode "dessine". Cette méthode a pour arguments les attributs entsor et composant.

#### 2.3.5.1 Descripteur de type

Le descripteur de type est formé d'une partie obligatoire et d'une partie optionnelle. La partie obligatoire consiste à donner la classe où l'attribut prend les valeurs. La partie optionnelle est composée de restrictions sur ces valeurs. Par exemple pour l'attribut âge dans la classe de personnes "âge: entier [1 .. 100]" modélise un attribut âge qui prend ses valeurs dans les entiers restreint à l'intervalle [1..100]. Les types ne sont pas explicitement manipulables dans le modèle. Une relation type/ sous-type est développée au paragraphe 3.

#### 2.3.5.2 Descripteur d'inférence

L'expression d'une inférence est utilisée pour obtenir la valeur d'un attribut. A chaque inférence correspond une méthode réalisant effectivement l'inférence. A un attribut peuvent être associés plusieurs inférences.

exemple : age <- diff(annee-cour, annee(numéro_sécurité_sociale))
ou user

Sur cet exemple, à l'attribut age est associé deux inférences, diff et user. Diff calcule la différence entre l'année courante et l'année donnée dans le numéro de sécurité sociale, user demandant l'age à l'utilisateur. Si la méthode diff échoue, si par exemple le numéro de sécurité sociale est inconnu, alors l'inférence user s'exécute demandant l'âge de la personne à l'utilisateur.

#### 2.3.5.3 Descripteur de contrainte

L'expression d'une contrainte permet une restriction sur les valeurs possibles de l'attribut. A toute contrainte est associée une méthode réalisant effectivement la contrainte. Plusieurs contraintes peuvent être associées à un attribut, l'ensemble des contraintes pouvant ne pas être satisfait par les valeurs instanciant l'attribut. Ceci concerne la gestion d'objets incohérents (§ 5).

#### 2.3.5.4 Descripteur de diffusion

Le descripteur de diffusion permet de propager des valeurs d'attributs entre d'une part une instance possédant un attribut et d'autre part les instances valuant cet attribut. Ceci est utile pour la manipulation d'objets composites par exemple. Une caractéristique d'un composant peut ainsi être remontée au niveau d'un objet composé : la couleur d'une carrosserie est ainsi la couleur de la voiture qui en est composée. Ce point sera développé plus en détail au paragraphe 4.

### 2.3.6 Attributs pré-déclarés

Pré-déclarer un attribut dans une classe consiste à donner son nom complet (nom de l'attribut + nom de sa classe d'origine) sans l'instancier. Cette notion est analogue à celle de caractéristique différée ("deferred feature") de Eiffel [MEY89]. La pré-déclaration d'un attribut définit la même sémantique pour un nom complet d'attribut dans tout le sous-graphe qui a pour racine la classe où il est pré-déclaré. Les attributs pré-déclarés permettent aussi de maintenir la cohérence de la base comme par exemple quand on élimine la définition d'un attribut de sa classe d'origine, celui-ci devient un attribut pré-déclaré. Cette notion est complémentaire à celle de classes disjointes (§ 3.2).

### 2.3.7 Identification des objets

Les instances sont identifiées par un identificateur unique (oid) et éventuellement une ou plusieurs clés. Une clé peut être composée de plusieurs attributs. Ces attributs doivent être obligatoirement instanciés. Par extension, nous considérons qu'il peut exister d'autres attributs obligatoires n'appartenant pas à la clé. Ces attributs représentent la connaissance minimale que doit posséder l'utilisateur pour créer, accéder ou modifier une instance. Dans l'exemple précédent, "contour" est un attribut obligatoire.

### 2.3.8 Clés

Traditionnellement les systèmes orientés objets offrent une notion d'identité d'objet qui est plus forte que la notion de clés utilisée dans les SGBD [KHO86]. Tous les objets ont un identificateur unique (oid). Toutes les relations entre les objets sont de ce fait représentées en

stockant les oid des objets en question. Cette action garantit l'intégrité de référence aux objets puisque les changements de valeur des attributs n'ont aucun effet sur leur oid.
Cette notion d'identité d'objet est retenue dans le modèle mais elle co-existe avec une notion de clés définies par l'usager. L'usager peut choisir de définir une ou plusieurs clés dans une classe. La définition d'une clé consiste en une liste d'attributs qui identifient de façon unique toutes les instances d'une classe. Par exemple pour une application donnée la clé formée du nom et du prénom est une clé de la classe Personne.
On peut définir une clé dans une classe comme une fonction bijective des valeurs des attributs de la clé sur les oid des instances de la classe. Cette définition nous assure de l'unicité des valeurs de la clé, c'est à dire qu'une valeur de clé dans une classe correspond à un et un seul oid de cette classe. Inversement, tous les objets d'une classe à clé ont une valeur différente pour cette clé.
Du point de vue de l'usager il est évident que l'utilisation des clés, par rapport à l'utilisation de variables nécessaires à la gestion des oid, facilite les manipulations dans un système orienté objet. L'utilisation de variables pour stocker l'oid d'un objet présente des inconvénients évidents sauf dans les environnements de programmation. Mais l'objectif du projet SHERPA n'est pas de définir un tel environnement. Ces inconvénients sont plus évidents quand il s'agit d'ajouter des grandes quantités d'information sous forme de tables [RUM87]. Nous croyons que la notion de clé co-existant avec la notion d'identificateur d'objet offre comparativement des avantages pratiques indéniables.
Un autre problème qui se présente est de définir une clé quand un des attributs dans la clé n'a pas une valeur élémentaire comme entier ou chaîne. Supposons que la classe des Chiens puisse être munie d'une clé qui identifie uniquement ces objets. Cette clé est formée de l'attribut nom_chien, avec comme type de base une chaîne de caractères, et de l'attribut propriétaire. Ainsi on peut parler du chien appelé "fido" appartenant à "Dupont". A la différence de l'attribut nom_chien, l'attribut propriétaire prend ses valeurs dans la classe Personne. C'est donc un attribut non élémentaire.
Un problème qui se pose alors est comment assurer l'unicité d'une telle clé, ou plus précisément que va-t-on stocker dans la clé à la place de l'attribut propriétaire ? Une solution évidente est d'utiliser l'oid de l'objet.

### 2.3.9 Contraintes

Nous laissons l'utilisateur choisir si une contrainte doit toujours être respectée ou non. En effet, celui-ci peut déterminer des contraintes fortes qui doivent toujours être respectées, ou des contraintes faibles, lors de la conception d'un objet, qui peuvent être violées. Pour spécifier une contrainte faible, nous utilisons le descripteur : "*contraintv*". Par défaut une contrainte est forte, nous utilisons alors le descripteur : "*contrainte*". Les contraintes précédentes sont toutes des contraintes intra-attribut qui ne font intervenir qu'un seul attribut de la classe.

### 2.3.10 Relations

On dit que deux objets sont en relation si l'un est valeur d'un attribut de l'autre. Intuitivement relation est utilisé ici lorsqu'il y a un rapport entre deux objets. On s'intéresse plus particulièrement aux relations entre classes en tant que concepts (relations dites verticales) et aux relations entre objets quelconques (relations dites horizontales) [ESC90].

#### 2.3.10.1 Relations verticales

Les relations entre classes sont typiquement du style spécialisation. Elles sont décrites dans le paragraphe 3. Dans un graphe de spécialisation une classe plus générale est toujours représenté dans un niveau plus haut. Ces relations sont représentés graphiquement par des traits verticaux.
Les relations verticales relient des classes représentant des concepts génériques ce qui permet de les structurer. Pour ce faire on utilise la relation de spécialisation et de disjonction. La relation de spécialisation permet de lier deux concepts dont l'un est un cas plus particulier que l'autre tels que les personnes et les étudiants (§ 3.1). La relation de disjonction permet d'exprimer l'indépendance entre deux concepts comme celle entre les classes des objets animés et des objets inanimés (§ 3.2).

#### 2.3.10.2 Relations horizontales

Les relations horizontales relient des objets dans le sens le plus général du terme, comme la relation de composition entre les objets composites et leurs parties constituantes. Elles sont décrites au

paragraphe 4. On ne s'intéresse alors pas aux concepts représentés (les personnes) mais aux propriétés des concepts (les quatre membres d'une personne) et à leurs inter-relations (liens de dépendances par exemple des membres vis-à-vis d'une personne). Cette notion est à notre sens orthogonale à la précédente ce qui explique que nous la qualifiions d'horizontale. Elle est parfois appelée "héritage sélectif" [CAR89, DUG88].
Pour bien montrer la différence entre ce que nous désignons par relations verticales, prenons le cas de la relation de composition : *est_composant_de*. Si l'on considére la hiérarchie structurelle de l'homme (un homme est composé d'une tête, de deux bras ...), on ne peut pas remplacer la relation *est_composant_de* par la relation de spécialisation *sorte_de* ou d'instanciation *instance_de* : la tête n'est pas une spécialisation d'homme, ni un homme particulier, mais une partie d'un homme.
Il y a donc une opposition entre les relations *sorte_de* et *instance_de* d'une part et *est_composant_de* d'autre part. De plus il n'y a pas une seule relation générique *est_composant_de* , comme *sorte_de* qui a en général le même sens quelque soient les classes, mais une relation *est_composant_de* différente selon les classes d'objets considérés. Elle peut être plus ou moins forte par exemple selon les objets considérés, ou interférer avec la notion d'objet partagé (§ 4.4). Ainsi, *est_composant_de* entre une voiture et une porte de voiture n'a pas forcément le même sens qu'entre une personne et son cerveau : une voiture sans porte existe, alors qu'une personne sans cerveau est beaucoup plus rare.

## 3. HERITAGE

### 3.1 Spécialisation

Les relations verticales (c-à-d entre classes) permettent d'affiner des connaissances sur les concepts génériques précédemment décrits. Afin de modéliser la relation de spécialisation entre deux classes, le modèle permet la définition d'une relation classe/sous-classe. La relation de classe/ sous-classe modélise la spécialisation stricte au sens de l'inclusion des ensembles: une sous-classe d'une classe C représente un sous-ensemble des instances de la classe C. En d'autres termes la relation classe/ sous-classe représente la relation ensemble/ sous-ensemble des instances.

La sémantique donnée ici à la relation de spécialisation ne permet pas l'occurrence d'exceptions. Une exception est le changement dans une classe des définitions faites dans une super-classe qui ne sont pas applicables aux éléments de cette super-classe [TOU84]. Exemple bien connu, supposons l'existence d'une classe d'éléphants gris, ayant quatre pattes et une longue trompe. L'apparition d'un éléphant rose engendre une exception pour la classe des éléphants. Les systèmes à exceptions permettent la création d'une sous-classe de la classe des éléphants qui change leur couleur de grise en rose, mais qui hérite des autres caractéristiques telles que le nombre de pattes et la longueur de la trompe.
Il est évident qu'un système qui permet les exceptions est un système basé sur une spécialisation non-stricte. Notre mécanisme est basé sur les ensembles et l'utilisation des exceptions peut nous faire arriver à des situations qui n'ont aucun sens. De plus cette modélisation ne permet pas un fonctionnement optimal d'un mécanisme de classification.
Nous pensons que le concepteur de la base de connaissances doit modéliser des relations vraiment universelles et non pas de relations vraies par défaut. Dans le mécanisme d'héritage défini ici, l'occurrence d'une exception est interprétée comme une mauvaise définition par l'usager. L'une des justifications de ce mécanisme est de lui donner toutes les aides possibles pour redéfinir les schémas (§ 5.3) et trouver des définitions sans exception.
La spécialisation multiple autorise une classe à avoir plusieurs super-classes. Ceci permet d'augmenter le partage de propriétés grâce à la combinaison des définitions de plusieurs classes. Dans le modèle, une classe C définit donc un sous-ensemble de l'intersection des ensembles définis par ses super-classes.

### 3.2 Disjonction

Parfois il est possible de déterminer à priori que les ensembles définis par deux classes sont disjoints, c'est à dire que ces deux classes n'ont aucune instance en commun. Par exemple les classes Circuit-électrique et Circuit-hydraulique n'ont à priori aucune instance commune. Cette connaissance peut permettre à des algorithmes de classification d'améliorer leurs performances. En effet s'il est connu qu'une instance appartient à une classe donnée, l'algorithme n'essaiera pas de la classifier dans une classe déclarée disjointe.

Cette connaissance est modélisée par la relation de classes disjointes. Deux classes qui maintiennent entre elles une relation de classes disjointes n'admettent pas d'instance en commun et, par conséquent, n'admettent pas de spécialisation commune.
La notion de classes disjointes alliée à celle d'attribut prédéclaré (§ 2.3.6) permet de définir des sous-graphes sémantiquement indépendants dans le graphe de spécialisation des classes. Il est ainsi possible de définir des attributs de même nom mais de signification différente dans des sous-graphes ayant pour racines des classes disjointes. Leurs classes peuvent néanmoins avoir des super-classes communes : une classe "Mammifères" peut avoir deux sous-classes "Chevaux" et "Femme" définissant chacune un attribut "robe". La robe d'un cheval n'a en principe aucun rapport avec la robe d'une femme. L'attribut "robe" est alors prédéclaré dans "mammifères" où il ne peut pas être instancié. Les classes "cheval" et "femme" sont déclarées disjointes : leurs attributs et en particulier "robe" ont alors une signification différente. Ceci présente des analogies avec la notion de contexte et de point de vue [CAR89, DUG88, MAR90].

## 3.3 Classification et multi-instanciation

Un mécanisme de spécialisation est fondamental dans un environnement de conception mais il n'est pas suffisant. En effet, il est indispensable que l'utilisateur sache pour une instance donnée à quelles classes elle peut être rattachée. S'il a, par exemple, créé un circuit électrique dans la classe Circuit-électrique, le système devra prendre en charge la classification automatique du circuit dans la classe des Petit-circuit-électrique s'il vérifie une contrainte de taille définie sur l'attribut contour. Inversement si son circuit ne vérifie pas cette contrainte mais que l'utilisateur sait qu'il construit un circuit de petite taille (la contrainte sera respectée "plus tard"), il doit pouvoir forcer l'insertion de l'instance dans la classe des Petit-circuit-électrique. Le modèle offre pour cela plusieurs mécanismes:
• La classification automatique d'une instance. Lorsqu'une instance est créée dans une classe C, elle est rétrocédée dans les super-classes de C. Pour chaque super-classe SC, le système de classification tente de classifier l'instance dans les autres sous-classes de SC et ainsi de suite. L'instance appartiendra à toutes les classes parcourues si elle vérifie leurs contraintes de spécialisation.
• La multi-instanciation explicite. Si l'utilisateur désire forcer la classification automatique, il peut explicitement demander qu'une instance soit rattachée à une ou plusieurs classes même si toutes les

contraintes de la classe ne sont pas respectées. Nous verrons au paragraphe 5 comment gérer des instances incohérentes.

## 3.4 Mise en oeuvre de l'héritage

### 3.4.1 Héritage des attributs

La relation classe/ sous-classe est mise en œuvre par l'intermédiaire d'un mécanisme d'héritage [DUC89]. Grâce à celui-ci toutes les sous-classes d'une classe C héritent de toutes les définitions des attributs de C. Il permet aussi d'affiner les sous-classes en ajoutant des attributs ou en les contraignant .
L'occurrence de conflits de noms est possible quand le nom d'un attribut hérité est identique au nom d'un attribut défini dans la classe. L'héritage multiple (réalisation de la spécialisation multiple) augmente les possibilités de conflits de noms des attributs que l'on peut classer en deux catégories. Le conflit entre une classe et sa super-classe, un problème qui existe aussi dans l'héritage simple, et le conflit entre les super-classes d'une classe, problème propre à l'héritage multiple. Plusieurs critères arbitraires sont utilisées par les systèmes actuels pour résoudre ces conflits. LOOPS [BOB83] utilise des listes de précédence sur les classes pour choisir un attribut en cas de conflit. Smalltalk-80 [GOL84] utilise l'héritage explicite des attributs, l'usager doit décider de quelle classe il veut hériter en cas de conflit. ORION [BAN87] permet d'hériter des différents attributs en conflit en les renommant.
Dans le modèle, le conflit de nom entre une classe et sa super-classe est interprété comme une affinement de l'attribut en question, c'est à dire une modification des descripteurs de l'attribut hérité. Cet affinement consiste soit à ajouter des contraintes ou des inférences, soit à restreindre les types des attributs hérités. Le fait de permettre seulement la restriction du type (par opposition à l'élargissement) permet de conserver une relation de type/ sous-type entre un attribut défini dans une classe et son affinement dans une sous-classe (§ 3.4.2). Afin de maintenir la relation classe/ sous-classe entre les attributs d'une classe et ses sous-classes, le type défini pour un attribut "a" dans une classe C doit être un sous-type du type de "a" dans la super-classe.
Voyons maintenant la solution de conflits de nom entre les super-classes d'une classe. Le mécanisme d'héritage fait hériter de tous les attributs définis dans toutes les super-classes d'une classe. Si un

attribut est défini dans plus d'une super-classe, le type hérité est le type qui représente l'intersection des types définis dans les super-classes (s'il existe). Dans le cas où cette intersection est vide on interdit la spécialisation.

### 3.4.2 Types et sous-types

Les types dans les systèmes orientés objets ont été approchés de façon formelle [DAN88]. Ils sont analysés dans [CAR85] en faisant une analogie entre objets et enregistrements. Dans cette section on utilise cette approche pour définir une relation entre types. Elle est analogue à celle de classe/ sous-classe.
Tous les attributs sont typés. Les types sont des restrictions sur les domaines de valeurs des attributs. Nous distinguons les types de base ou scalaires (définis dans le modèle tels que entier, chaîne, etc.) des types structurés (tels que personnes, voitures, etc.) composés de plusieurs attributs. Un type structuré peut être le résultat d'une définition donnée par l'usager ou d'une inférence de type faite par le mécanisme d'héritage lors d'un intersection de types.
Les types de base sont pré-définis dans le modèle. Etant donné que tout type est représenté par une classe, on peut définir une relation de sous-type dénotée par $\leq$, équivalente à celle des sous-classes dans l'héritage multiple.
La relation d'ordre sur les types doit respecter notre sémantique de l'héritage. Dans les principes généraux de l'héritage on trouve que toute classe hérite de tous les attributs définis dans ses super-classes, mais elle a le droit d'ajouter de nouveaux attributs. Par conséquent on peut dire que tout sous-type d'un type T doit avoir au moins les mêmes attributs que le type T.
Maintenant examinons nos interprétations de sous-classe et d'attribut. La définition d'une sous-classe est interprétée comme la définition d'un sous-ensemble des éléments définis dans la classe dont elle est sous-classe. Les attributs d'une classe représentent les caractéristiques de l'ensemble des objets de la classe. Par conséquent les domaines des attributs hérités dans une classe doivent, soit conserver les types définis dans la classe dont ils sont hérités, soit redéfinir des sous-types de ceux-ci. On peut alors énoncer le premier axiome :

**A1.** Un type structuré T1 est un sous-type d'un type structuré T2 si et seulement si T1 a tous les attributs définis dans T2 et si les types

des attributs de T1 sont des sous-types des attributs correspondants de T2. Cette relation est dénotée :

$$T1 \leq T2$$

Cet axiome peut être écrit aussi de la forme suivante:

$$(ai : Ti, aj : Tj) \leq (ai : T'i) \Longleftrightarrow Ti \leq T'i$$

où i est un entier de 1 .. n, $n \geq 0$ et
j est un entier de 1 .. m, $m \geq 0$

où les ai représentent les noms complets des attributs et les Ti leurs types respectifs. Par exemple le type de étudiants est un sous-type des personnes :
(nom : chaîne, prénom : chaîne, âge : entier, école : chaîne)
$\leq$ (nom : chaîne, prénom : chaîne, âge : entier)

Notre premier axiome établit aussi que tout type structuré est un sous-type de lui-même. L'axiome suivant permet le démarrage de cette relation d'ordre en établissant que tout type de base est un sous-type de lui-même :

**A2.** Tout type de base est un sous-type de lui-même : $T \leq T$ où T est un type de base.

### 3.4.3 Les intervalles

Les intervalles ne sont pas considérés comme des contraintes mais comme des restrictions de types dont le mécanisme d'héritage connaît la sémantique. Ainsi on peut maintenir la relation de (ensemble, sous-ensemble) pour les intervalles ou effectuer l'intersection des intervalles tel qu'on le fait pour les types lors d'un cas d'héritage multiple.

### 3.4.4 Héritage des attributs pré-déclarés

La pré-déclaration d'un attribut permet de définir le nom complet d'un attribut sans l'instancier. Ce nom est transmis à toutes les sous-classes de la classe d'origine ce qui permet aux sous-classes de connaître l'existence de ce nom complet d'attribut. Un attribut est considéré pré-déclaré dans tout le graphe dénoté par la classe d'origine sauf dans le sous-graphe où une des sous-classes de la classe d'origine le défini.

## 4. RELATIONS SEMANTIQUES

Dans tout système orienté-objet se trouve la notion d'attribut qui permet de définir la structure d'un objet. En général, tous les attributs ont la même sémantique. Il est parfois souhaitable de permettre l'expression de différentes natures d'attributs. En particulier, il faut distinguer les attributs qui ne sont qu'une simple référence (le numéro de série d'un circuit) de ceux exprimant une relation ou un lien sémantique entre des objets (les différents composants d'un circuit).
Des systèmes comme LOOPS ou ORION font la distinction entre les attributs normaux et les attributs composites qui permettent d'exprimer un lien de composition entre des objets. Le reproche que l'on peut faire à ces systèmes est que la sémantique du lien exprimé est figée. On a alors recours à des solutions artificielles qui font intervenir des méthodes ou des règles.
SRL, langage à base de frames, répond en partie à ce besoin : tout est définissable par l'utilisateur. Le système est complet mais si complexe que de nombreuses fonctionnalités du langage ne peuvent pas être utilisées dans des applications réelles. Le modèle défini ici essaie de combler les lacunes des langages de classes comme LOOPS et ORION et d'y intégrer la puissance d'un langage de frames comme SRL.
Par la suite, nous désignons par *receveur* l'instance dont l'attribut est valué et par *donneurs* les instances (ou l'instance) valuant l'attribut. Par exemple, si l'attribut contour d'une instance c1 de la classe circuit est valué par une instance r1 de la classe rectangle, alors c1 est le receveur et r1 le donneur.

### 4.1 L'attribut comme référence

L'attribut utilisé comme une référence permet uniquement de désigner un autre objet. Par exemple "c1 a pour dimension 10" traduit que l'attribut dimension de c1 fait référence à l'entier 10. L'attribut référence n'a aucune sémantique propre. Sa raison d'être est l'expression de "propriétés" (au sens large) qui n'impliquent pas de définition de liens entre des objets. Il faut simplement le voir comme un cas particulier d'attribut relation que nous allons définir.

## 4.2 L'attribut comme relation

L'attribut vu comme une relation est utilisé si un lien sémantique particulier est à définir entre le donneur (l'instance dont l'attribut est valué) et les receveurs (les instances valuant l'attribut).
Un attribut-relation se caractérise par un ensemble de propriétés et d'opérateurs :
- réflexivité, non réflexivité, symétrie, transitivité. Par exemple, on peut rattacher les propriétés de non réflexivité et de transitivité à l'attribut-relation 'composant' de la classe Circuit-électrique.
- opérations d'union, d'intersection, de réciprocité et de composition applicables sur une relation.

Supposons que la classe Circuit-électrique admette les attributs-relation entrée et sortie. L'attribut entsort peut alors être considéré comme l'union des attributs-relation entrée et sortie. Chaque modification des attributs entrée ou sortie est alors automatiquement répercutée sur l'attribut entsort pour que la propriété d'union soit toujours vérifiée.

## 4.3 Dépendances

Une dépendance est associée à chaque attribut défini dans la base et permet de lier des instances : le receveur et les donneurs. L'expression d'une dépendance sur un attribut permet de spécifier qui peut manipuler, modifier ou supprimer la ou les instances valeurs de cet attribut. La dépendance est spécifiée au cours de la saisie d'un attribut. Elle est identique pour toutes les instances de la classe. Si l'attribut âge a une dépendance exclusive, toutes les instances de la classe dans laquelle âge a été défini ont un attribut âge avec une dépendance exclusive. Ceci est en accord avec la notion de classe qui veut que toutes les instances d'une classe aient le même comportement .
On définit les dépendance exclusive, partagée, nulle, existentielle et spécifique.

### 4.3.1 Dépendance exclusive

Cette dépendance permet de définir le manipulateur unique d'une instance : son propriétaire au sens commun du terme. Un parallèle peut être fait avec les langages algorithmiques classiques. En PASCAL par exemple une procédure possède des variables locales, celles-ci ne sont accessibles que par la procédure ou ses sous-procédures. Une

instance dépendante exclusivement d'une autre peut être assimilée à une variable locale qui n'a d'existence que dans le "contexte" de son instance propriétaire.

### 4.3.2 Dépendance partagée

L'objectif est de permettre plusieurs manipulateurs sur une instance, c-à-d l'accès à une valeur d'attribut depuis plusieurs autres instances. Ceci est utile quand les modifications d'une instance dépendent de plusieurs objets.

### 4.3.3 Dépendance nulle

Valeur par défaut du descripteur de dépendance, le receveur ne peut pas manipuler les donneurs. Cette dépendance est bien souvent la seule fournie dans de nombreux systèmes : une instance est simplement la valeur d'un attribut (dans notre modèle "instancie un attribut"),. La dépendance de l'attribut doit être totalement réalisée et maîtrisée par le concepteur de la base.

### 4.3.4 Dépendance existentielle

On peut avoir une dépendance existentielle sur un attribut. La sémantique est la suivante : le receveur est supprimé si les donneurs sont supprimés, sauf pour les receveurs ayant une dépendance existentielle vis-à-vis d'autres objets. On peut montrer que la dépendance existentielle interfère avec la notion de partage d'objet (§ 4.4). Il y a donc deux facteurs à prendre en compte :
- l'idée d'existence conditionnée par celle d'un autre objet,
- l'idée de dépendance transitive ou concomitante, dûe au besoin d'exister pour plusieurs objets simultanément ou pour une chaîne d'objets dépendants les uns des autres.

### 4.3.5 Dépendance spécifique

Tous les dépendances précédentes sont génériques car identiques pour toutes les instances d'une classe. Dans bien des cas, on ne peut pas savoir à la conception de la base si toutes les instances devront avoir la même. Un attribut-relation peut exprimer une dépendance existentielle entre donneurs et receveurs. L'existence des receveurs dépend alors de celle des donneurs. Un même receveur peut être soumis à plusieurs dépendances existentielles. Son existence dépend

alors de l'ensemble de ses donneurs. Il n'est donc détruit que si tous les objets donneurs dont il dépend sont supprimés.
Exemple : si un circuit est détruit et que son attribut "equipo" a une dépendance existentielle, toutes ses equipotentielles sont alors détruites. L'existence des équipotentielles d'un circuit est ainsi conditionnée par l'existence du circuit. On peut donc éviter l'accumulation d'équipotentielles (donc de connaissances) inutiles.

### 4.4 Partage d'objets

Dans de nombreux cas un partage d'objets doit pouvoir être exprimé. Par exemple, un composant peut être partagé par plusieurs circuits. Il faut alors définir les droits de chaque circuit vis à vis de ce composant et maintenir la cohérence de l'ensemble lors de la manipulation de ce composant. Le modèle permet d'exprimer si un objet est ou non partageable. Nous distinguons le partage exclusif, commun, nul et spécifique. Des circuits complexes peuvent ainsi être construits, en contrôlant parfaitement le partage des divers composants.

### 4.5 Diffusion de valeurs

La diffusion de valeurs consiste à propager des valeurs d'attributs entre des objets. Le but est double :
- permettre le partage de valeurs, c-à-d réduire la duplication d'information et donc réduire les problèmes de cohérence. Si les composants d'un circuit fonctionnent tous à la même fréquence, il paraît inutile de dupliquer l'information "fréquence de fonctionnement" dans tous les composants : mieux vaut la stocker à un endroit et partager cet "endroit" entre les divers composants.
- considérer des objets, liés par des relations, comme une seule et même entité s'échangeant des informations.
La diffusion est bidirectionnelle : elle exprime les attributs du donneur utilisables dans les receveurs et les attributs des receveurs utilisables dans le donneur [DUC89].
Ainsi la taille maximum d'un composant peut être déduite de la taille maximum du circuit auquel il appartient et des tailles des divers autres composants du circuit. Un ensemble de composants est par ce biais mieux assimilé à une même entité : le circuit intégré qu'il forme dans la réalité.

Les différentes propriétés des attributs vus comme des relations sont exprimées par des descripteurs attachés à ces attributs [ESC90].

## 5. GESTION D'OBJETS EVOLUTIFS

Peu de systèmes s'intéressent à la dynamique des instances, c'est-à-dire à la gestion et au contrôle de leur évolution. A un instant t, un objet peut par exemple ne pas avoir tous ses attributs instanciés, l'objet est incomplet, on parle alors d'incomplétude; une ou plusieurs contraintes définies sur cet objet peuvent ne pas être respectées, on parle alors de l'incohérence de l'objet.
Dans le cadre d'applications d'IA ou de CAO par exemple, la gestion de l'incomplétude et de l'incohérence des objets est nécessaire [RIE90]. En effet, lors de la conception d'un objet celui-ci n'est complet et cohérent que dans sa phase finale. Sa structure peut évoluer, ainsi que ses relations avec les autres objets : c'est le problème de l'évolution de schémas (§ 5.3). "*En IA, on accepte des graphes d'héritage très dynamiques, dans lesquels on peut ajouter - ou enlever - aussi bien des feuilles du graphe que des arcs entre objets préexistants : la connaissance évolue. Pour les langages de programmation, seules les feuilles sont ajoutables et on considère que redondances et contradictions - qui relèvent à la fois des confusions et des lacunes - sont dangereuses. Nous nous placons par la suite dans un point de vue IA.*" [DUC89].

### 5.1 Structures significatives

L'incomplétude et l'incohérence des instances sont gérées à l'aide de structures dites significatives. Pour gérer l'évolution des instances, une relation d'ordre est définie entre les structures. Si, lors d'une mise à jour, l'évolution d'un objet satisfait la relation d'ordre, nous considérons que celui-ci s'améliore. La notion d'amélioration d'un objet peut être affinée selon les applications, l'utilisateur peut en effet spécifier ses propres règles opératoires. L'utilisateur peut, par exemple, décider qu'un objet s'améliore si la valeur d'un attribut X augmente ou diminue. On précise ces notions plus loin.
Une structure significative est construite à partir d'une classe. Elle représente un état possible d'un objet de cette classe. Par exemple, on crée une structure pour les circuit-électriques sans composant, etc ...

A partir d'une classe initiale, on ne génère pas toutes les structures potentielles, car certaines d'entre elles ne représentent aucune réalité (par exemple une structure contenant une contrainte dont les arguments ne sont pas connus). On utilise des règles d'optimisation qui permettent de restreindre la fermeture à un ensemble de structures dites significatives correspondant à une réalité possible [NGU90]. Seules sont considérées comme légales les structures ainsi obtenues. Ces règles heuristiques sont indépendantes des applications, mais dépendent du modèle de données. Le nombre de structures créées étant réduit, la détermination automatique de la structure d'un objet lors de sa création puis de sa modification est accélérée.

Une instance particulière est rattachée à une structure et une seule. C'est la structure maximale qui contient toutes les propriétés satisfaites par l'objet. Par exemple, un Circuit-électrique dont on ne connaît que le contour et la liste des composants, appartient à la structure dont seuls les attributs "contour" et "composant" sont mentionnés.

Dans les systèmes traditionnels, l'instance est rattachée à sa classe; en cas d'incomplétude les attributs non instanciés prennent la valeur "nil". Dans notre modèle, l'instance value toutes les propriétés de la structure. Même dans les cas d'incohérence, l'instance vérifie les contraintes de la structure.

Le rattachement d'une instance, lors de sa création, à une structure permet de déterminer son état de complétude et de cohérence. Si l'instance est modifiée, son état évolue et sa structure de rattachement peut changer. On établit une relation d'ordre entre les structures pour permettre de définir le concept d'amélioration d'une instance. La relation d'ordre entre les structures est définie comme suit. Soit:

- deux structures S1 et S2,
- p1 le degré de complétude des objets de S1,
- p2 le degré de complétude des objets de S2,
- l1 le degré de cohérence des objets de S1,
- l2 le degré de cohérence des objets de S2.

On a:

$S1 > S2 \iff p1 > p2 \text{ et } l1 \geq l2 \text{ ou } p1 \geq p2 \text{ et } l1 > l2$

$S1 = S2 \iff p1 = p2 \text{ et } l1 = l2$

Avec cette relation d'ordre, des structures peuvent être incomparables. Les structures S1 et S2 ne sont pas comparables dans les cas suivants :

$l1 > l2$ et $p1 < p2$ ou $l1 < l2$ et $p1 > p2$.

En effet, les degrés de cohérence et de complétude varient en sens inverse. Une instance *s'améliore* si après mise à jour, sa structure finale est supérieure ou égale à sa structure initiale. Dans le cas où les deux structures ne sont pas comparables, nous considérons qu'il n'y a pas d'amélioration puisque l'on assiste à une diminution du degré de complétude ou du degré de cohérence. On constate que l'amélioration d'une instance dépend des degrés de cohérence et de complétude des structures.

Nous décomposons les opérations de mise à jour en deux étapes : d'une part la certification, d'autre part la validation. Une opération est certifiée si l'instance s'améliore. En effet, dans ce cas son degré de cohérence et/ou de complétude augmente. Une mise à jour est donc non certifiée dans les cas où l'objet ne respecte plus une ou plusieurs contraintes de la classe, et/ou si un ou plusieurs attributs de la classe ne sont plus instanciés. La validation représente la mise à jour physique des modifications. Si l'opération est certifiée, elle est automatiquement validée; sinon l'utilisateur peut ou non choisir de la valider.

Les données ne sont donc modifiées qu'après validation. Ceci permet une grande souplesse de mise à jour, car l'utilisateur peut faire des essais de modifications d'attributs, sans entraîner une mise à jour physique de l'instance dans le cas où ces modifications ne sont pas pertinentes.

## 5.2 Versions et boucles de dégradation

Nous autorisons l'utilisateur à valider une opération non certifiée, car nous supposons que cet état de l'instance n'est pas définitif. Nous générons, dans ce cas, une *boucle de dégradation*. La boucle de dégradation d'une instance est constituée d'un ensemble de versions. Chacune d'elles modélise l'instance après une modification.

Dès la première dégradation d'une instance, une version de celle-ci est créée[FAV90a]. Tant que la version courante ne possède pas des degrés de cohérence et de complétude supérieurs ou égaux à la première version, le système génère à chaque modification validée une nouvelle version. Lorsque l'instance après une série de modifications est dans une structure supérieure ou égale à celle de la première version, toute la boucle de dégradation est supprimée et l'instance n'est plus versionnalisée.

L'utilisateur peut à tout moment décider de repartir d'une version particulière. La boucle de dégradation se transforme alors en un arbre de versions. En effet, à partir d'une même version plusieurs autres versions peuvent être dérivées. Ceci permet à l'utilisateur de simuler des opérations de mise à jour.
Nous avons vu que la notion d'amélioration d'une instance dépendait des degrés de complétude et de cohérence des objets. Il nous a semblé pertinent d'autoriser l'utilisateur, pour une classe donnée, à affiner cette notion d'amélioration, en spécifiant des critères d'amélioration supplémentaires. Ces critères sont spécifiés par des règles opératoires. Ces critères portent sur des attributs de la classe. Par exemple, l'utilisateur spécifie qu'une instance de la classe Circuit-électrique s'améliore si le contour devient plus petit, ceci dans un but de minimiser la place occupée par les circuits.
Pour décider de la dégradation ou non d'une instance, on tient non seulement compte des degrés des structures, mais aussi de ces règles. Si l'une de ces règles n'est pas respectée, alors l'objet se dégrade [FAV90a].
L'affinage du concept d'évolution d'un objet permet de gérer l'amélioration d'un objet, même si celui-ci n'a pas changé de structure lors de sa mise à jour. Par exemple, même si après modification de l'attribut "contour" les degrés de complétude et de cohérence de l'objet ne diminuent pas, l'opération n'est certifiée que si le nouveau contour est plus petit que l'ancien.

### 5.3 Evolution de schémas

La dynamique des classes est un aspect important de la CFAO, car dans une base, les valeurs mais aussi les schémas des objets (c-à-d ici les définitions des classes), doivent pouvoir évoluer jusqu'à la validation complète de leur conception. Nous ne ferons dans ce paragraphe qu'esquisser cet aspect de la dynamique [NGU89a].
La mise à jour d'une classe comprend l'ajout d'une nouvelle propriété dans cette classe, la suppression d'une propriété, la modification d'une propriété. Cette dernière mise à jour porte sur les descripteurs, nous pouvons, par exemple, ajouter de nouvelles contraintes, modifier des inférences, etc...
La mise à jour d'une classe a plusieurs impacts sur les éléments de la base. Dans un premier temps, il faut vérifier qu'après sa mise à jour le schéma reste cohérent avec le reste du graphe de spécialisation. En effet, il faut que la spécialisation stricte soit toujours respectée.

Le système doit également vérifier que les instances du schéma correspondent encore au schéma modifié. Par exemple, si une nouvelle contrainte forte est ajoutée au schéma, il faut vérifier que toutes les instances de la classe vérifient bien cette contrainte, et dans le cas contraire en aviser l'utilisateur. Nous nous sommes particulièrement intéressés à l'optimisation de la mise à jour des instances d'une classe modifiée. En effet, le nombre d'instances dans une classe peut être grand, et cette mise à jour peut donc occasionner une grande perte de temps. La mise à jour des instances structure par structure optimise grandement cette étape [FAV90b].

## 6. CONCLUSIONS

SHERPA est un projet qui a pour objectif d'offrir un outil d'expérimentation novateur dans le domaine des bases de connaissances. La démarche initiale a consisté à fusionner certaines fonctionnalités issues des langages et modèles de données orientés-objets d'une part, des systèmes de représentation des connaissances utilisés en intelligence artificielle d'autre part. SHERPA rassemble ainsi certaines notions qui sont rarement présentes simultanément dans les systèmes de RC ou les LOO. Ceci est dû aux objectifs du projet qui concernent avant tout la gestion de connaissances dynamiques.
On a donné ici une description générale aussi exhaustive que possible des concepts de base d'un modèle répondant aux critères de réflexivité, d'extensibilité et de gestion d'objets évolutifs. On n'a de ce fait pas cherché à formaliser certains concepts ni à entrer dans des considérations trop techniques. Certains aspects ont été volontairement négligés, comme le typage fort, la confidentialité, la sécurité et les transactions. Certains points restent à affiner et sont donc susceptibles d'évoluer dans le futur. Certaines notions sont particulièrement étudiées car elles conditionnent la gestion de la dynamique qui est le principal objectif de recherche du projet. Par exemple :
- les relations sémantiques entre objets, que ce soit pour les liens de dépendances ou pour la diffusion de valeurs [ESC90],
- la dynamique des schémas [FAV90a],
- la gestion d'objets évolutifs [FAV90b].

Les perspectives de développement à court terme, c-à-d celles pour lesquelles des travaux sont actuellement en cours, concernent :

- le développement d'un langage de requête ad-hoc inspiré des bases de données,
- un système de maintien de la vérité [EUZ89],
- la gestion de points de vue multiples sur les objets [MAR90].

D'autres études à plus longue échéance font également l'objet de travaux exploratoires, par exemple pour la prise en compte d'objets actifs.

Les applications de SHERPA font quant à elles l'objet de trois études distinctes avec des partenaires extérieurs. Elles concernent :
- la modélisation de bases de séquences en biologie moléculaire avec le Laboratoire de Biométrie et de Génétique des Populations de l'Université Claude Bernard de Lyon [GAU89],
- la modélisation du processus de production dans un environnement de CFAO en mécanique, en collaboration avec l'INSA, l'Université Claude Bernard et l'Ecole Centrale de Lyon [RIE90],
- le développement d'un système d'aide à la résolution de problèmes, en collaboration avec le projet EVE du Laboratoire TIM3 de l'IMAG.

Ces applications ouvrent des champs d'expérimentations très vastes, très prometteurs et de ce fait paradoxalement très contraignants. Elles permettront de confronter les options choisies dans SHERPA avec des exigences opérationnelles.

## Remerciements

SHERPA est un projet IMAG et un projet INRIA. Il bénéficie de ce fait du soutien matériel de ces deux organismes. Il bénéficie également du soutien :
- du MRT pour des collaborations avec INFOSYS et ILOG d'une part, avec le Laboratoire de Biologie Moléculaire de l'Université Claude Bernard de Lyon d'autre part,
- de la Région Rhône-Alpes pour une collaboration avec l'INSA et l'Ecole Centrale de Lyon,
- du PRC-BD3.

Enfin, SHERPA est le fruit d'un travail collectif auquel participent également L. Buisson, J. Euzenat, P. Fontanille, O. Marino, F. Rechenmann, A. Sales-Simonet et P. Uvietta.

## BIBLIOGRAPHIE


[BAN87] BANERJEE J. et al. *Data model issues for object-oriented applications*. ACM Trans. on Office Information Syst. 5, 1. 1987.

[BAN89] BANCILHON F. et al. *The O2 book*. GIP Altaïr. 1989.

[BOB83] BOBROW D.G, STEFIK M. *The LOOPS manual : a data and object-oriented programming system for InterLisp*. Xerox PARC. Palo Alto. 1983.

[BOO86] BOOCH G. *Object-oriented development*. IEEE Trans. Software Engineering. 12, 2. Février 1986.

[CAR85] CARDELLI L., WEGNER P. *On understanding types, data abstraction and polymorphism*. ACM Computing Surveys. 17,4. Décembre 1985.

[CAR89]CARRE B. *Méthodologie orientée objet pour la représentation des connaissances*. Thèse de Doctorat. Université des Sciences et Techniques de Lille Flandres Artois. 1989.

[COI87] COINTE P. *Metaclasses are first class : the ObjVlisp model*. Proc. 2nd OOPSLA. Orlando (USA). Sept. 87.

[COP84] COPELAND G., MAIER D. *Making Smalltalk a database system*. Proc. ACM SIGMOD Conf. Boston (USA). Juin 1984.

[DAN88] DANFORTH S., TOMLINSON C. *Type theories and object-oriented programming*. ACM Computing Surveys. 20, 1. Mars 1988.

[DUC88] DUCOURNAU R. *YAFOOL : manuel de référence*. Sema-Matra. 1988.

[DUC89] DUCOURNAU R., HABIB M. *La multiplicité de l'héritage dans les langages à objets*. TSI. 8,1. Dunod Informatique. 1989.

[DUG88] DUGERDIL P. *Contribution à l'étude de la représentation des connaissances fondée sur les objets. Le langage OBJLOG*. Thèse de Doctorat. Université d'Aix-Marseille II. Décembre 1988.

[ESC90] ESCAMILLA J., JEAN P. *Relations verticales et horizontales dans un modèle de représentation des connaissances*. Actes INFORSID '90. Eyrolles Ed. Biarritz. Mai 1990.

[EUZ89] EUZENAT J. *Etendre les TMS vers les contextes*. Proc. RFIA '89. Paris. Sept. 1989

[FAV90a] FAVIER V., RIEU D. *Manipulation d'objets dynamiques dans les bases de connaissances*. Proc. MICAD '90. Paris. Hermès Ed. Février 1990.

[FAV90b] FAVIER V., RIEU D. *Dynamique dans les bases de connaissances. Projet SHERPA.* Modèles et Bases de Données. AFCET Informatique. Février 1990.

[FER84] FERBER J. *Quelques aspects du caractère self réflexif du langage MERING.* Proc. 2° JLOO. Brest. 1984.

[GAU89] GAUTHIER C., PERRIERE G. *Les bases de données de séquences : un outil de modélisation du génôme.* Rencontres Biologie Moléculaire et Informatique. Paris. Juin 1989.

[GOL84] GOLDBERG A. *The Smalltalk-80 system release process.* Smalltalk-80, Bits of History, Words of advice. Addison-Wesley. 1984.

[KEE88] KEENS S.E *Object-oriented programming in Common Lisp. A programmer's guide to CLOS.* Addison-Wesley. 1988.

[KHO86] KHOSHAFIAN S., COPELAND G. *Object identity.* Proc. 1st OOPSLA. Portland. Septembre 1986.

[MAR90] MARINO O. *Multiple perspectives and classification mechanisms in object-oriented representations.* Soumis à publication.

[MAS89] MASINI G. et al. *Les langages à objets.* InterEditions. Paris. 1989.

[MEY89] MEYER B. *Conception et programmation par objets, pour du logiciel de qualité.* InterEditions. Paris. 1989.

[MIN75] MINSKY M. *A framework for representing knowledge.* The psychology of computer vision. McGraw-Hill. 1975.

[MOO86] MOON D. *Object-oriented programming with Flavors.* Proc. 1st OOPSLA '86. Portland (USA). 1986.

[NEB85] NEBEL B. *How well does a vanilla loop fit into a frame ?* Data & Knowledge Engineering. 1,1. North-Holland. 1985.

[NGU90] NGUYEN G.T, RIEU D. *Heuristic control on dynamic database objects.* Proc. IFIP Conf. "The role of AI in Databases and Information Systems". Guangzhou (R.P de Chine). North-Holland. 1990.

[NGU89a] NGUYEN G.T, RIEU D. *Schema evolution in object-oriented database systems.* Data & Knowledge Engineering. 4, 1. North-Holland. Juillet 1989.

[NGU89b] NGUYEN G.T, RIEU D. *Schema change propagation in object-oriented databases.* Proc. XIth World Computer Congress. IFIP '89. San Francisco (USA). Août 1989.

[REC88] RECHENMANN F. *Shirka : un système de gestion de bases de connaissances. Manuel de référence.* IMAG. Laboratoire Artemis. Juin 1988.

[RIE86] RIEU D., NGUYEN G.T *Semantics of CAD objects for generalized databases*. Proc. 23rd Design Automation Conference. Las Vegas (USA). Juillet 1986.

[RIE90] RIEU D. et al. *SHERPA : un support d'intégration pour le processus CEDF*. Proc. CIM '90. Bordeaux. Juin 1990.

[ROB77] ROBERTS R.B, GOLDSTEIN I.P *The FRL manual* . MIT. Cambridge. 1977.

[RUM87] RUMBAUGH J. *Relations as semantic constructs in an object oriented language*. Proc. 2nd OOPSLA. Orlando (USA). Octobre 1987.

[TOU84] TOURETZKI D.S *The mathematics of inheritance systems*. PhD Thesis. Carnegie Mellon University. Mai 1984.

[WRI84] WRIGHT J.M, FOX M.S, ADAM D.L *SRL User's manual*. Carnegie Mellon University.1984.

A paraître dans les Actes du Congrès INFORSID '90. Biarritz. Mai 1990.

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

ISSN 0249 - 6399

ISSN 0249-6399